Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2508141 C 2009/11/03

(11)(21) 2 508 141

(12) BREVET CANADIEN
CANADIAN PATENT

(13) C

(86) Date de dépôt PCT/PCT Filing Date: 2003/12/02
(87) Date publication PCT/PCT Publication Date: 2004/06/17
(45) Date de délivrance/Issue Date: 2009/11/03
(85) Entrée phase nationale/National Entry: 2005/06/01
(86) N° demande PCT/PCT Application No.: AU 2003/001616
(87) N° publication PCT/PCT Publication No.: 2004/050369
(30) Priorités/Priorities: 2002/12/02 (AU2002953135);
2002/12/02 (AU2002953134)

(51) Cl.Int./Int.Cl. *B41J 2/01* (2006.01),
B41J 2/135 (2006.01), *B41J 29/393* (2006.01)

(72) Inventeurs/Inventors:
WALMSLEY, SIMON ROBERT, AU;
JACKSON PULVER, MARK, AU;
PLUNKETT, RICHARD THOMAS, AU;
SHIPTON, GARY, AU;
SILVERBROOK, KIA, AU;
LAPSTUN, PAUL, AU

(73) Propriétaire/Owner:
SILVERBROOK RESEARCH PTY LTD, AU

(74) Agent: OYEN WIGGS GREEN & MUTALA LLP

(54) Titre : COMPENSATION D'UNE BUSE HORS ETAT DE FONCTIONNEMENT
(54) Title: DEAD NOZZLE COMPENSATION

(57) Abrégé/Abstract:
A printer controller for supplying dot data to a printhead in a predetermined order, the printhead comprising at least first and second printhead modules, each of which comprises a plurality of printing nozzles and being disposed adjacent each other such that a printing width of the printhead is wider than a printing width of either of the printhead modules, the printer controller being configured to order and time supply of the dot data to the printhead modules in accordance with their respective widths, such that a difference in relative widths of the printhead modules is at least partially compensated for.

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

CORRECTED VERSION

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
17 June 2004 (17.06.2004)

PCT

(10) International Publication Number
WO 2004/050369 A1

(51) International Patent Classification[7]: B41J 2/01

(21) International Application Number:
PCT/AU2003/001616

(22) International Filing Date: 2 December 2003 (02.12.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

2002953134	2 December 2002 (02.12.2002)	AU
2002953135	2 December 2002 (02.12.2002)	AU

(71) Applicant *(for all designated States except US)*: **SILVERBROOK RESEARCH PTY LTD** [AU/AU]; 393 Darling Street, Balmain, New South Wales 2041 (AU).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **WALMSLEY, Simon, Robert** [AU/AU]; Silverbrook Research Pty Ltd, 393 Darling Street, Balmain, New South Wales 2041 (AU). **JACKSON PULVER, Mark** [AU/AU]; Silverbrook Reseach Pty Ltd, 393 Darling Street, Balmain, New South Wales 2041 (AU). **PLUNKETT, Richard, Thomas** [AU/AU]; Silverbrook Research Pty Ltd, 393 Darling Street, Balmain, New South Wales 2041 (AU). **SHIPTON, Gary** [GB/AU]; Silverbrook Research Pty Ltd, 393 Darling Street, Balmain, New South Wales 2041 (AU). **SILVERBROOK, Kia** [AU/AU]; Silverbrook Research Pty Ltd, 393 Darling Street, Balmain, New South Wales 2041 (AU). **LAPSTUN, Paul** [NO/AU]; Silverbrook Research Pty Ltd, 393 Darling Street, Balmain, New South Wales 2041 (AU).

(74) Agent: **SILVERBROOK, Kia**; Silverbrook Research Pty Ltd, 393 Darling Street, Balmain, New South Wales 2041 (AU).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (BW, GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, RO, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

(48) Date of publication of this corrected version:
12 May 2005

(15) Information about Correction:
see PCT Gazette No. 19/2005 of 12 May 2005, Section II

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: DEAD NOZZLE COMPENSATION

(57) Abstract: A printer controller for supplying dot data to a printhead in a predetermined order, the printhead comprising at least first and second printhead modules, each of which comprises a plurality of printing nozzles and being disposed adjacent each other such that a printing width of the printhead is wider than a printing width of either of the printhead modules, the printer controller being configured to order and time supply of the dot data to the printhead modules in accordance with their respective widths, such that a difference in relative widths of the printhead modules is at least partially compensated for.

WO 2004/050369 A1

DEMANDE OU BREVET VOLUMINEUX

LA PRÉSENTE PARTIE DE CETTE DEMANDE OU CE BREVET COMPREND PLUS D'UN TOME.

CECI EST LE TOME	1	DE	4
CONTENANT LES PAGES	1	À	284

NOTE : Pour les tomes additionels, veuillez contacter le Bureau canadien des brevets

JUMBO APPLICATIONS/PATENTS

THIS SECTION OF THE APPLICATION/PATENT CONTAINS MORE THAN ONE VOLUME

THIS IS VOLUME	1	OF	4
CONTAINING PAGES	1	TO	284

NOTE: For additional volumes, please contact the Canadian Patent Office

NOM DU FICHIER / FILE NAME :

NOTE POUR LE TOME / VOLUME NOTE:

TITLE: DEAD NOZZLE COMPENSATION

FIELD OF INVENTION

The present invention relates to techniques for compensating for one or more dead nozzles in a multi-nozzle printhead.

The invention has primarily been developed for use with a printhead comprising one or more printhead modules constructed using microelectromechanical systems (MEMS) techniques, and will be described with reference to this application. However, it will be appreciated that the invention can be applied to other types of printing technologies in which analogous problems are faced.

BACKGROUND OF INVENTION

Manufacturing a printhead that has relatively high resolution and print-speed raises a number of problems.

Difficulties in manufacturing pagewidth printheads of any substantial size arise due to the relatively small dimensions of standard silicon wafers that are used in printhead (or printhead module) manufacture. For example, if it is desired to make an 8 inch wide pagewidth printhead, only one such printhead can be laid out on a standard 8-inch wafer, since such wafers are circular in plan. Manufacturing a pagewidth printhead from two or more smaller modules can reduce this limitation to some extent, but raises other problems related to providing a joint between adjacent printhead modules that is precise enough to avoid visible artefacts (which would typically take the form of noticeable lines) when the printhead is used. The problem is exacerbated in relatively high-resolution applications because of the tight tolerances dictated by the small spacing between nozzles.

The quality of a joint region between adjacent printhead modules relies on factors including a precision with which the abutting ends of each module can be manufactured, the accuracy with which they can be aligned when assembled into a single printhead, and other more practical factors such as management of ink channels behind the nozzles. It will be appreciated that the difficulties include relative vertical displacement of the printhead modules with respect to each other.

Whilst some of these issues may be dealt with by careful design and manufacture, the level of precision required renders it relatively expensive to manufacture printheads within the required tolerances. It would be desirable to provide a solution to one or more of the problems associated with precision manufacture and assembly of multiple printhead modules to form a printhead, and especially a pagewidth printhead.

In some cases, it is desirable to produce a number of different printhead module types or lengths on a substrate to maximise usage of the substrate's surface area. However, different sizes and types of modules will have different numbers and layouts of print nozzles, potentially including different

horizontal and vertical offsets. Where two or more modules are to be joined to form a single printhead, there is also the problem of dealing with different seam shapes between abutting ends of joined modules, which again may incorporate vertical or horizontal offsets between the modules. Printhead controllers are usually dedicated application specific integrated circuits (ASICs) designed for specific use with a single type of printhead module, that is used by itself rather than with other modules. It would be desirable to provide a way in which different lengths and types of printhead modules could be accounted for using a single printer controller.

Printer controllers face other difficulties when two or more printhead modules are involved, especially if it is desired to send dot data to each of the printheads directly (rather than via a single printhead connected to the controller). One concern is that data delivered to different length controllers at the same rate will cause the shorter of the modules to be ready for printing before any longer modules. Where there is little difference involved, the issue may not be of importance, but for large length differences, the result is that the bandwidth of a shared memory from which the dot data is supplied to the modules is effectively left idle once one of the modules is full and the remaining module or modules is still being filled. It would be desirable to provide a way of improving memory bandwidth usage in a system comprising a plurality of printhead modules of uneven length.

In any printing system that includes multiple nozzles on a printhead or printhead module, there is the possibility of one or more of the nozzles failing in the field, or being inoperative due to manufacturing defect. Given the relatively large size of a typical printhead module, it would be desirable to provide some form of compensation for one or more " dead" nozzles. Where the printhead also outputs fixative on a per-nozzle basis, it is also desirable that the fixative is provided in such a way that dead nozzles are compensated for.

A printer controller can take the form of an integrated circuit, comprising a processor and one or more peripheral hardware units for implementing specific data manipulation functions. A number of these units and the processor may need access to a common resource such as memory. One way of arbitrating between multiple access requests for a common resource is timeslot arbitration, in which access to the resource is guaranteed to a particular requestor during a predetermined timeslot.

One difficulty with this arrangement lies in the fact that not all access requests make the same demands on the resource in terms of timing and latency. For example, a memory read requires that data be fetched from memory, which may take a number of cycles, whereas a memory write can commence immediately. Timeslot arbitration does not take into account these differences, which may result in accesses being performed in a less efficient manner than might otherwise be the case. It would be desirable to provide a timeslot arbitration scheme that improved this efficiency as compared with prior art timeslot arbitration schemes.

Also of concern when allocating resources in a timeslot arbitration scheme is the fact that the priority of

an access request may not be the same for all units. For example, it would be desirable to provide a timeslot arbitration scheme in which one requestor (typically the memory) is granted special priority such that its requests are dealt with earlier than would be the case in the absence of such priority.

In systems that use a memory and cache, a cache miss (in which an attempt to load data or an instruction from a cache fails) results in a memory access followed by a cache update. It is often desirable when updating the cache in this way to update data other than that which was actually missed. A typical example would be a cache miss for a byte resulting in an entire word or line of the cache associated with that byte being updated. However, this can have the effect of tying up bandwidth between the memory (or a memory manager) and the processor where the bandwidth is such that several cycles are required to transfer the entire word or line to the cache. It would be desirable to provide a mechanism for updating a cache that improved cache update speed and/or efficiency.

Most integrated circuits an externally provided signal as (or to generate) a clock, often provided from a dedicated clock generation circuit. This is often due to the difficulties of providing an onboard clock that can operate at a speed that is predictable. Manufacturing tolerances of such on-board clock generation circuitry can result in clock rates that vary by a factor of two, and operating temperatures can increase this margin by an additional factor of two. In some cases, the particular rate at which the clock operates is not of particular concern. However, where the integrated circuit will be writing to an internal circuit that is sensitive to the time over which a signal is provided, it may be undesirable to have the signal be applied for too long or short a time. For example, flash memory is sensitive to being written too for too long a period. It would be desirable to provide a mechanism for adjusting a rate of an on-chip system clock to take into account the impact of manufacturing variations on clockspeed.

One form of attacking a secure chip is to induce (usually by increasing) a clock speed that takes the logic outside its rated operating frequency. One way of doing this is to reduce the temperature of the integrated circuit, which can cause the clock to race. Above a certain frequency, some logic will start malfunctioning. In some cases, the malfunction can be such that information on the chip that would otherwise be secure may become available to an external connection. It would be desirable to protect an integrated circuit from such attacks.

In an integrated circuit comprising non-volatile memory, a power failure can result in unintentional behaviour. For example, if an address or data becomes unreliable due to falling voltage supplied to the circuit but there is still sufficient power to cause a write, incorrect data can be written. Even worse, the data (incorrect or not) could be written to the wrong memory. The problem is exacerbated with multi-word writes. It would be desirable to provide a mechanism for reducing or preventing spurious writes when power to an integrated circuit is failing.

In an integrated circuit, it is often desirable to reduce unauthorised access to the contents of memory. This is particularly the case where the memory includes a key or some other form of security information

that allows the integrated circuit to communicate with another entity (such as another integrated circuit, for example) in a secure manner. It would be particularly advantageous to prevent attacks involving direct probing of memory addresses by physically investigating the chip (as distinct from electronic or logical attacks via manipulation of signals and power supplied to the integrated circuit).

It is also desirable to provide an environment where the manufacturer of the integrated circuit (or some other authorised entity) can verify or authorize code to be run on an integrated circuit.

Another desideratum would be the ability of two or more entities, such as integrated circuits, to communicate with each other in a secure manner. It would also be desirable to provide a mechanism for secure communication between a first entity and a second entity, where the two entities, whilst capable of some form of secure communication, are not able to establish such communication between themselves.

In a system that uses resources (such as a printer, which uses inks) it may be desirable to monitor and update a record related to resource usage. Authenticating ink quality can be a major issue, since the attributes of inks used by a given printhead can be quite specific. Use of incorrect ink can result in anything from misfiring or poor performance to damage or destruction of the printhead. It would therefore be desirable to provide a system that enables authentication of the correct ink being used, as well as providing various support systems secure enabling refilling of ink cartridges.

In a system that prevents unauthorized programs from being loaded onto or run on an integrated circuit, it can be laborious to allow developers of software to access the circuits during software development. Enabling access to integrated circuits of a particular type requires authenticating software with a relatively high-level key. Distributing the key for use by developers is inherently unsafe, since a single leak of the key outside the organization could endanger security of all chips that use a related key to authorize programs. Having a small number of people with high-security clearance available to authenticate programs for testing can be inconvenient, particularly in the case where frequent incremental changes in programs during development require testing. It would be desirable to provide a mechanism for allowing access to one or more integrated circuits without risking the security of other integrated circuits in a series of such integrated circuits.

In symmetric key security, a message, denoted by M, is *plaintext*. The process of transforming M into *ciphertext* C, where the substance of M is hidden, is called *encryption*. The process of transforming C back into M is called *decryption*. Referring to the encryption function as E, and the decryption function as D, we have the following identities:

$$E[M] = C$$
$$D[C] = M$$

Therefore the following identity is true:

$$D[E[M]] = M$$

A symmetric encryption algorithm is one where:

- the encryption function E relies on key K_1,
- the decryption function D relies on key K_2,
- K_2 can be derived from K_1, and
- K_1 can be derived from K_2.

In most symmetric algorithms, K_1 equals K_2. However, even if K_1 does not equal K_2, given that one key can be derived from the other, a single key K can suffice for the mathematical definition. Thus:

$$E_K[M] = C$$
$$D_K[C] = M$$

The security of these algorithms rests very much in the key K. Knowledge of K allows *anyone* to encrypt or decrypt. Consequently K must remain a secret for the duration of the value of M. For example, M may be a wartime message "My current position is grid position 123-456". Once the war is over the value of M is greatly reduced, and if K is made public, the knowledge of the combat unit's position may be of no relevance whatsoever. The security of the particular symmetric algorithm is a function of two things: the strength of the algorithm and the length of the key.

An asymmetric encryption algorithm is one where:

- the encryption function E relies on key K_1,
- the decryption function D relies on key K_2,
- K_2 cannot be derived from K_1 in a reasonable amount of time, and
- K_1 cannot be derived from K_2 in a reasonable amount of time.

Thus:

$$E_{K1}[M] = C$$
$$D_{K2}[C] = M$$

These algorithms are also called *public-key* because one key K_1 can be made public. Thus anyone can encrypt a message (using K_1) but only the person with the corresponding decryption key (K_2) can decrypt and thus read the message.

In most cases, the following identity also holds:

$$E_{K2}[M] = C$$
$$D_{K1}[C] = M$$

This identity is very important because it implies that anyone with the public key K_1 can see M and know

that it came from the owner of K_2. No-one else could have generated C because to do so would imply knowledge of K_2. This gives rise to a different application, unrelated to encryption - digital signatures.

A number of public key cryptographic algorithms exist. Most are impractical to implement, and many generate a very large C for a given M or require enormous keys. Still others, while secure, are far too slow to be practical for several years. Because of this, many public key systems are hybrid - a public key mechanism is used to transmit a symmetric session key, and then the session key is used for the actual messages.

All of the algorithms have a problem in terms of key selection. A random number is simply not secure enough. The two large primes p and q must be chosen carefully - there are certain weak combinations that can be factored more easily (some of the weak keys can be tested for). But nonetheless, key selection is not a simple matter of randomly selecting 1024 bits for example. Consequently the key selection process must also be secure.

Symmetric and asymmetric schemes both suffer from a difficulty in allowing establishment of multiple relationships between one entity and a two or more others, without the need to provide multiple sets of keys. For example, if a main entity wants to establish secure communications with two or more additional entities, it will need to maintain a different key for each of the additional entities. For practical reasons, it is desirable to avoid generating and storing large numbers of keys. To reduce key numbers, two or more of the entities may use the same key to communicate with the main entity. However, this means that the main entity cannot be sure which of the entities it is communicating with. Similarly, messages from the main entity to one of the entities can be decrypted by any of the other entities with the same key. It would be desirable if a mechanism could be provided to allow secure communication between a main entity and one or more other entities that overcomes at least some of the shortcomings of prior art.

In a system where a first entity is capable of secure communication of some form, it may be desirable to establish a relationship with another entity without providing the other entity with any information related the first entity's security features. Typically, the security features might include a key or a cryptographic function. It would be desirable to provide a mechanism for enabling secure communications between a first and second entity when they do not share the requisite secret function, key or other relationship to enable them to establish trust.

A number of other aspects, features, preferences and embodiments are disclosed in the Detailed Description of the Preferred Embodiment below.

SUMMARY OF THE INVENTION

In accordance with the invention, there is provided a method of compensating for an inoperative nozzle in a printhead, the method comprising the step of:

(a) mapping dot data intended for the inoperative nozzle into one or more operative nozzles of the printhead.

Preferably, step (a) includes the substep of mapping the dot data intended for the inoperative nozzle into a nozzle that will print a dot on print media close to a position at which the inoperative nozzle would have printed a dot had it been operative.

Preferably also, step (a) includes the substep of mapping the dot data intended for the inoperative nozzle into a nozzle that will print a dot on print media immediately adjacent a position at which the inoperative nozzle would have printed a dot had it been operative.

In a preferred embodiment, step (a) includes the substeps of:

(i) determining one or more operative nozzles capable of printing a dot on print media close to a position at which the inoperative nozzle would have printed a dot had it been operative; and

(ii) mapping the dot data from the inoperative nozzle to an operative nozzle determined in substep (i).

More preferably, in the event more than one operative nozzle is determined in substep (i), the dot data is remapped to one of the operative nozzles that will print a dot on print media closest to that which would have been printed by the inoperative nozzle.

It is preferred that during successive firings of the printhead, the dot data is remapped alternately to operative nozzles that will print a dot on print media either side of that which would have been printed by the inoperative nozzle.

In an alternative embodiment, during successive firings of the printhead, the dot data is remapped randomly, pseudo-randomly, or arbitrarily to operative nozzles that will print a dot on print media either side of that which would have been printed by the inoperative nozzle.

Preferably, the printhead including a plurality of sets of the nozzles for printing a corresponding plurality of channels of dot data, wherein step (a) includes the substep of mapping the dot data intended for the inoperative nozzle into one or more operative nozzles from the same set.

In one form, step (a) includes the substep of mapping the dot data into one or more operative nozzles that will print a dot on print media close to a position at which the inoperative nozzle would have printed a dot had it been operative.

In an alternative form, step (a) includes the substep of mapping the dot data intended for the inoperative nozzle into one or more operative nozzles including at least one nozzle from a different one of the sets.

In yet another embodiment, step (a) includes the substeps of:

determining which combination of one or more available operative nozzles near the inoperative nozzle will minimise perceived error in an image that the dot data forms part of, the determination being performed on the basis of a color model; and

mapping the dot data intended for the inoperative nozzle to that combination of one or more operative nozzles.

Preferably, the inoperative nozzle is associated with a black print channel, and wherein step (a) includes remapping the dot data intended for the inoperative nozzle into a plurality of operative nozzles in other color channels to produce a process black output at or adjacent a location on print media where the inoperative nozzle would have deposited a droplet of a black printing substance in accordance with the dot data.

In a preferred embodiment, a plurality of dot data intended for a corresponding plurality of inoperative nozzles are mapped to operative nozzles.

In accordance with a second aspect of the invention, there is provided a printer controller configured to implement the method of the first aspect.

In accordance with a third aspect of the invention, there is provided a printer controller configured to implement the method of the first aspect to a printhead comprising a plurality of the nozzles.

In accordance with the invention, there is provided a method for outputting a portion of a dither matrix stored in a memory, comprising the step of:

(b) determining a start position and an end position in the memory;

(c) reading a plurality of dither values of the dither matrix from the memory, commencing at the start position; and

(d) outputting a portion of the plurality of dither values read in step (b)

Preferably two or more dither matrices are stored in the memory. A plurality of dither values are read from at least two of the dither matrices with a single read. The matrices can be different sizes.

It is preferred that each read from the memory reads at least one, and preferably two or more, lines from one or more dither matrices.

The method can also be embodied in hardware.

It is also preferred that the memory is configurable to store different dither matrices for different color channels. It is particularly preferred that a single read of the memory loads a full line for two or more dither matrices into a dither buffer. Typically, each dither matrix will be for a different color channel.

In accordance with another aspect of the invention, there is provided a printer controller for supplying dot data to a printhead in a predetermined order, the printhead comprising at least a first printhead module having a plurality of rows of printing nozzles, the printer controller being configured to order and time the supply of the dot data to the first printhead module such that a relative skew between adjacent rows of printing nozzles on the at least one printhead module, in a direction normal to a direction of printing, is at least partially compensated for.

Preferably, the printer controller is configured to at least partially compensate for the relative skew between adjacent rows in each of a plurality of sets of the adjacent rows.

In a preferred embodiment, wherein the relative skew between each of the plurality of the sets of the adjacent rows is the same.

Preferably, the printer controller is configured to compensate for the skew by introducing a relative delay into the dot data destined for at least one of the rows of printing nozzles. More preferably, the printhead is configured to print the dots at a predetermined spacing across its width, and the delay introduced by the printer controller equates to an integral multiple of the spacing.

It is particularly preferred, that the printhead defines a printable region between printing boundaries. Nozzles of at least one of the rows of at least one of the at least one printhead modules are positioned outside the printable region due to the skew between adjacent rows of the nozzles on the at least one printhead module. The printer controller is configured to introduce a relative delay into the dot data supplied to at least one of the rows such that the nozzles outside the printable region do not print.

Preferably, the at least one printhead module includes at least one pair of adjacent rows of the nozzles such that each row of the pair is configured to print the same ink. The printhead is configured to provide the dot data to the pair of adjacent rows such that the dot data is shifted serially through the first of the rows then through the second of the rows, until the dot data has been supplied to all the nozzles. More preferably, the printhead is configured to provide the dot data to the pair of adjacent rows such that the dot data is shifted serially through the first of the rows in a first direction then looped back through the second of the rows in a second direction opposite the first, until the dot data has been supplied to all the nozzles.

Preferably, the printhead is configured to print a series of printhead-width rows of the dots, and wherein the first and second rows are configured to print odd and even dots, respectively, of the printhead-width rows, the printhead controller being configured to supply the one or more first rows with odd dot data and the one or more second rows with even dot data.

Preferably, the printhead has a plurality of the pairs of rows. The printer controller is configured to supply the dot data such that any relative skew between the first and second rows of each pair of rows, in a direction normal to a direction of printing, is at least partially compensated for.

In one embodiment, each printhead module is configured to print a plurality of independent inks, and the nozzles in each row are configured to print in one of the inks. The printhead controller being configured to supply each of the inks to at least one row of at least one of the printhead modules.

Preferably, at least some of the printhead modules are of mutually unequal length, the printer controller being configured to order and time the supply of the dot data to the compensate for the unequal length.

It is also preferable that the printer controller is configured to at least partially compensate for any relative skew between adjacent rows of the nozzles on adjacent ones of the printhead modules.

In a preferred form of the invention, the printer controller is selectively configurable to compensate at least partially for a plurality of potential relative skews.

In one form, the controller is configured to compensate at least partly for a fixed amount of the skew.

In accordance with a further aspect, the invention comprises the printer engine comprising a printer controller according to the first aspect and a printhead, wherein the nozzles of the printhead are disposed in a printable region between printing boundaries of the printhead. The printhead includes at least one logical nozzle located outside the printable zone that can accept data but is not capable of printing. The logical nozzles are arranged to introduce a relative delay into the dot data supplied to at least one of the rows, such that dot data is supplied to the correct nozzles for printing.

In accordance with another aspect of the invention, there is provided a printer controller for supplying dot data to a printhead in a predetermined order, the printhead comprising at least first and second printhead modules, each comprising a plurality of printing nozzles and being disposed adjacent each other such that a printing width of the printhead is wider than a printing width of either of the printhead modules, the printer controller being configurable during or after manufacture to order and time supply of the dot data to the printhead modules such that any relative displacement between the printhead nozzles in a direction normal to the printhead printing width is at least partially compensated for.

Preferably, the printer controller is configurable to provide compensation for any of a plurality of different amounts of the relative displacement.

More preferably, where each of the printhead modules comprises a plurality of parallel rows of the printing nozzles, the printhead is configured such that each of the rows of each printhead module has a corresponding row in each of the other printhead modules. The printer controller is controllable to introduce a relative delay into the dot data supplied to one or more of the rows, thereby to provide the compensation.

In a particularly preferred embodiment, where the printhead is configured to print the dots at a predetermined spacing in a direction in which print media is supplied for printing, the delay introduced by the printer controller equates to an integral multiple of the spacing during printing.

In accordance with a further aspect of the invention, there is a printer controller for supplying dot data to a printhead in a predetermined order, the printhead comprising at least first and second printhead modules, each of which comprises a plurality of printing nozzles and being disposed adjacent each other such that a printing width of the printhead is wider than a printing width of either of the printhead modules, the printer controller being configured to order and time supply of the dot data to the printhead modules in accordance with their respective widths, such that a difference in relative widths of the printhead modules is at least partially compensated for.

Preferably, each of the printhead modules comprises a plurality of rows of the printing nozzles, the controller being configured to supply the dot data to the rows of nozzles in serial form. More preferably, each of the printhead modules comprises one or more parallel pairs of the rows, the controller being configured to serially supply the data to a first of each of the rows of nozzles in the or each pair of rows, the data being serially clocked through the first row of the or each pair of rows, then through a second row of the or each pair or rows, until all printhead nozzles have received their respective data.

It is preferred that the data is clocked through the second row in a direction substantially opposite to that in which it was clocked through the first row.

In another aspect, there is provided a printer controller for supplying dot data to a printhead comprising at least one printhead module, the printer controller being configurable to supply the dot data to a selectable one of a plurality of potential printhead module types, each having a different number of nozzles for receiving the dot data.

Preferably, the printer controller includes non-volatile memory for storing at least one parameter value, the at least one parameter value determining which of the potential printhead types the printer controller has been configured to supply the dot data to.

More preferably, the printer controller is configurable to supply the dot data to the printhead module on the basis of one or more printer module widths indicated by the at least one parameter.

In a preferred embodiment, the printer controller is configurable to supply the dot data to a plurality of the printhead modules, on the basis of one or more widths of the printhead modules indicated by the at least one parameter.

In accordance with a further aspect of the invention, there is provided a method of accounting for dead nozzle remapping in a multi-nozzle printhead, including remapping a fixative intended for a dot to be printed by the dead nozzle.

In one form, the remapping includes remapping the fixative to an operative nozzle to which dot data intended for the dead nozzle for printing at or adjacent a position at which the dead nozzle would have printed.

Alternatively, or in addition, the remapping includes preventing output of fixative onto the position where the dead nozzle would have printed a dot had it been operative.

In accordance with a further aspect of the invention, there is provided a method for arbitrating between a plurality of access requests issued in relation to a resource by a plurality of requestors, wherein each request can be one of at least two types, a first of the types having a higher latency associated with its performance than at least some of the other types, the method including the steps of:

(e) receiving a plurality of the access requests; (the requests are not placed anywhere, they are simply received)

(f) maintaining a current pointer that points to a current timeslot in a timeslot list, and at least one lookahead pointer that points to a future timeslot in the timeslot list; and

(g) in the event an access request as arbitrated via the lookahead pointer is of the first type, initiating performance of the access request earlier than the position in the list suggests it would be performed should it be started when the current pointer reached the timeslot.

Preferably, step (g) includes the substep of performing the access request indicated by the lookahead pointer immediately after the access request indicated by the current pointer is performed.

It is preferred that step (g) includes the substep of performing the access request indicated by the lookahead pointer immediately after the access request indicated by the current pointer is performed.

In a preferred embodiment, the number of timeslots between the timeslot indicated by the lookahead pointer and the timeslot indicated by the current pointer takes into account a latency difference between performing an access request of the first type and at least one of the other access request types.

In accordance with another aspect of the invention, there is provided an integrated circuit including:

a plurality of operative units, each of which is capable of issuing a request for access to a memory accessible by the integrated circuit; and

an timeslot arbitrator for arbitrating between requests issued by the operative units for access to the memory, wherein each request can be one of at least two types, a first of the types having a higher latency associated with its performance than at least some of the other types, the timeslot arbitrator being configured to:

(h) receive a plurality of the access requests;

(i) maintain a current pointer that points to a current timeslot in a timeslot list, and at least one lookahead pointer that points to a future timeslot in the timeslot list; and

(j) in the event the access request as arbitrated via the lookahead pointer is of the first type, performing the access request earlier than the position in the list suggests it should be performed should it be started when the current pointer reached the timeslot.

Preferably, the first type of access request is a memory write request.

Preferably, the integrated circuit includes a memory interface unit operatively connected with, and under the control of, the timeslot arbitrator, and wherein the memory interface is operatively connected to:

one or more of the operative units via one or more communications buses, and

the memory via a memory bus of greater width than the communications buses.

In a preferred form, the number of timeslots between the timeslot indicated by the lookahead pointer and the timeslot indicated by the current pointer takes into account a latency difference between performing an access request of the first type and at least one of access request types.

In accordance with a further aspect of the invention, there is provided a method of arbitrating between access requests from a plurality of requestors for access to a resource, wherein at least one of the requestors is defined as higher priority access to the resource, the method comprising the steps of:

(k) receiving a plurality of the access requests;

(l) in the event an access request from the at least one of the requestors is received, initiating performance of the access request in preference to the requestor as specified by the timeslot list and regardless of whether or not the at least one of the requestors is in the timeslot list.

Preferably, the at least one requestor requires lower latency access to the resource than at least one of the other requestors from which access requests can be received.

Preferably, the at least one requestor is a processor and/or the resource is a memory.

Preferably, step (l) includes the substep of performing the access request from the requestor immediately following completion of any current access request being reformed.

In a preferred form, step (l) is performed such that a frequency of the at least one requestor being granted preferential performance of its access requests is limited within a time period. More preferably, early performance of access requests from the at least one requestor is restricted to a maximum number of times within a predetermined number of timeslots.

Preferably, the requestors are hardware units on an integrated circuit and the method is implemented by a timeslot arbitrator unit on the integrated circuit.

In accordance with another aspect of the invention, there is provided a method according to the third aspect, wherein each request can also be one of at least two types, a first of the types having a higher latency associated with its performance than at least some of the other types, the method including the steps of:

(m) receiving a plurality of the access requests;

(n) maintaining a current pointer that points to a current timeslot in the timeslot list, and at least one lookahead pointer that points to a future timeslot in the timeslot list; and

(o) in the event an access request as arbitrated via the lookahead pointer is of the first type, initiating performance of the access request earlier than its position in the list suggests it should be performed should it be started when the current pointer reached the timeslot.

In accordance with another aspect of the invention, there is provided a method of updating a cache in an integrated circuit comprising:

the cache

a processor connected to the cache via a cache bus;

a memory interface connected to the cache via a first bus and to the processor via a second bus, the first bus being wider than the second bus or the cache bus; and

memory connected to the memory interface via a memory bus;

the method comprising the steps of:

(p) following a cache miss, using the processor to issue a request for first data via a first address, the first data being that associated with the cache miss;

(q) in response to the request, using the memory interface to fetch the first data from the memory, and sending the first data to the processor;

(r) sending, from the memory interface and via the first bus, the first data and additional data, the additional data being that stored in the memory adjacent the first data;

(s) updating the cache with the first data and the additional data via the first bus; and

(t) updating flags in the cache associated with the first data and the additional data, such that the updated first data and additional data in the cache is valid.

Preferably, the processor is configured to attempt a cache update with the first data upon receiving it from the memory interface, the method further including the step of preventing the attempted cache update by the processor from being successful, thereby preventing interference with the cache update of steps (s) and/or (t).

More preferably, steps (r), (s), and (t) are performed substantially simultaneously.

In one embodiment, steps (s) and (t) are performed by the memory interface.

Preferably, steps (s) and (t) are performed in response to the processor attempting to update the cache following step (r). More preferably, the memory interface is configured to monitor the processor to determine when it attempts to update the cache following step (r).

In accordance with a further aspect of the invention, there is provided an integrated circuit, comprising a processor, an onboard system clock for generating a clock signal, and clock trim circuitry, the integrated circuit being configured to:

(u) receive an external signal;

(v) determine either the number of cycles of the clock signal during a predetermined number of cycles of the external signal, or the number of cycles of the external signal during a predetermined number of cycles of the clock signal;

(w) store a trim value in the integrated circuit, the trim value having been determined on the basis of the determined number of cycles; and

(x) use the trim value to control the internal clock frequency.

Preferably, the integrated circuit is configured to, between steps (v) and (w):

output the result of the determination of step (v); and

receive the trim value from an external source.

Preferably, the integrated circuit includes non-volatile memory, and (w) includes storing the trim value in the memory. More preferably, the memory is flash RAM.

In a preferred form step (x) includes loading the trim value from the memory into a register and using the trim value in the register to control a frequency of the internal clock.

In a preferred form, the trim value is determined and stored permanently in the integrated circuit. More preferably, the circuit includes one or more fuses that are intentionally blown following step (w), thereby preventing the stored trim value from subsequently being changed.

In a preferred embodiment, the system clock further includes a voltage controlled oscillator (VCO), an output frequency of which is controlled by the trim value. More preferably, the integrated circuit further includes a digital to analog convertor configured to convert the trim value to a voltage and supply the voltage to an input of the VCO, thereby to control the output frequency of the VCO.

Preferably, the integrated circuit is configured to operate under conditions in which the signal for which the number of cycles is being determined is at a considerably higher frequency than the other signal.

More preferably, the integrated circuit is configured to operate when a ratio of the number of cycles determined in step (v) and the predetermined number of cycles is greater than about 2. It is particularly preferred that the ratio is greater than about 4.

Preferably, the integrated circuit is disposed in a package having an external pin for receiving the external signal. More preferably, the pin is a serial communication pin configurable for serial communication when the trim value is not being set.

Preferably, the trim value was also determined on the basis of a compensation factor that took into account a temperature of the integrated circuit when the number of cycles are being determined.

Preferably, the trim value received was determined by the external source, the external source having determined the trim value including a compensation factor based on a temperature of the integrated circuit when the number of cycles are being determined.

Preferably, the trim value is determined by performing a number of iterations of determining the number of cycles, and averaging the determined number.

In accordance with a further aspect of the invention, there is provided an integrated circuit comprising a processor, non-volatile memory, an input for receiving power from a power supply and a power detection unit, wherein the integrated circuit is configured to enable multi-word writes to the non-volatile memory, the power detection unit being configured to:

monitor a quality of power supplied to the input;

in the event the quality of the power drops below a predetermined threshold, preventing subsequent words in any multi-word write currently being performed from being written to the memory.

Preferably, the integrated circuit is configured to prevent any further writes of any type to the memory once the quality is determined to have dropped below the threshold.

Preferably, the quality is a voltage.

Preferably, the memory is flash memory.

Preferably, the power detection unit is configured to provide a reset signal to at least some other circuits on the integrated circuit once any current writes have been finished.

In accordance with another aspect of the invention, there is provided an integrated circuit comprising a processor, a memory that the processor can access, a memory access unit for controlling accesses to the memory, an input for receiving power for the integrated circuit from an external power source, and a power detection unit, the power detection unit being configured to:

monitor a quality of power supplied to the input;

in the event the quality of the power drops below a predetermined threshold, disabling a power supply to circuitry for use in writing to the memory, such that the memory access unit's ability to alter data in

the memory is disabled prior to address or data values to be written to the memory becoming unreliable due to failing power.

Preferably, the memory is flash memory and the power supply is one or more charge pump circuits. More preferably, a voltage output by the power supply falls fast enough that the voltage supplied to the flash memory becomes too low to enable a change in contents of the flash memory before the voltage levels of the address or data values become invalid.

Preferably, the integrated circuit is configured to cause a reset of at least some of the circuitry on the integrated circuit following disabling of the power supply.

More preferably, the integrated circuit is programmed or designed to have a variable delay between disabling of the power supply and causing the reset.

In accordance with a further aspect of the invention, there is provided an integrated circuit comprising a processor and memory, the memory storing a set of data representing program code and/or an operating value, wherein each bit of the data is stored as a bit/inverse-bit pair in corresponding pairs of physically adjacent bit cells in the memory.

Preferably, the integrated circuit further includes a memory management unit configured to receive a request for the set of data and to test, during processing of the request, whether the respective pairs of physically adjacent bit-cells that correspond to the set of data contain bit/inverse-bit pairs, thereby to confirm the validity of the set of data as stored in the memory. More preferably, the memory management unit is configured to store sets of data as sets of bit/inverse-bit pairs in the memory.

Preferably, the integrated circuit is selectively operable in either of first and second modes, wherein:

in the first mode, the memory management unit is configured to receive and process a request for the set of data, and to test, during processing of the request, whether the respective pairs of physically adjacent bit-cells corresponding to the set of data contain bit/inverse-bit pairs, thereby to confirm the validity of the set of data as stored in the memory; and

in the second mode, the memory management unit is configured to receive and process a request for data stored in the memory, without testing whether pairs of physically adjacent bit-cells contain bit/inverse-bit pairs.

More preferably:

in the first mode, the memory management unit is configured to store a set of data associated with a memory write request as a corresponding set of bit/inverse-bit pairs, each of the bit/inverse-bit pairs being physically adjacent each other; and

in the second mode, the memory management unit is configured to store a set of data associated with a memory write request as the set of data without corresponding inverse-bits.

Preferably, the integrated circuit is configured to boot into the first mode by default.

Preferably, the integrated circuit is configured to implement a defensive action in the event the test fails.

More preferably, the defensive action includes resetting the integrated circuit.

In an alternative embodiment, the defensive reaction includes returning second data other than that the subject of the test.

Preferably, the second data is a string of identical digits.

Preferably, the defensive reaction is different depending upon whether the set of data represents program code or an operating value.

More preferably, in the event the test fails and the set of data is an operating value, the integrated circuit is configured to replace the failed value with a substitute value.

More preferably, the substitute value is selected to disrupt a program running on the integrated circuit.

Preferably, the substitute causes at least some circuitry on the integrated circuit to reset.

In a preferred embodiment, in the event the test fails, the integrated circuit is permanently prevented from running software.

Preferably, in the event the test fails, the integrated circuit is configured to delete from the memory some or all of the bit values associated with the set of data.

More preferably, in the event the test fails, the integrated circuit is configured to delete some or all of the contents of the memory.

In accordance with another aspect of the invention, there is provided an integrated circuit comprising a processor and memory storing:

secret information accessible via a first address, the secret information comprising a string of bit values;

an inverse-string accessible via a second address, the inverse-string comprising a string of bit values, wherein each of the bit values in the inverse-string is the logical inverse of a bit value at a corresponding bit position in the secret information, the integrated circuit being programmed with code configured to:

(i) receive a request for the secret information; and

(ii) test whether the bit-values of the inverse string are the inverse of the bit-values at respective corresponding bit positions of the secret information.

In accordance with a further aspect of the invention, there is provided a method of ensuring validity of secret information stored in a memory in the form of a string of bit values accessible via a first address, the memory also storing an inverse-string accessible via a second address, the inverse-string comprising a string of bit values that are the logical inverses of the bit values at corresponding respective bit positions of the secret information, the method including the steps of:

receiving a request for the secret information; and

testing whether the bit-values of the inverse string are the inverse of the bit-values at respective corresponding bit positions of the secret information.

Preferably, the integrated circuit is configured and programmed to perform a defensive action in the event the test fails.

More preferably, the defensive action incudes deleting or destroying some or all of the contents of the memory in the event the test fails. Preferably, the defensive action includes deleting or destroying at least the secret information and/or the inverse string.

Preferably, the defensive action includes preventing the processor from executing software.

Preferably, the defensive action includes resetting some or all of logic on the integrated circuit.

Preferably, the first and second addresses are at the same address in the memory.

Preferably, the string and inverse string are stored at different sub-addresses within the same address.

In accordance with a further aspect of the invention, there is provided method of manufacturing a plurality of the integrated circuits, comprising the steps, for each of the plurality of integrated circuits, of:

storing the secret information and the inverse string at the first and second addresses in the memory of the integrated circuit; and

storing the code on the integrated circuit;

wherein the first and second addresses are randomly, pseudo-randomly or arbitrarily selected for each of the integrated circuits and the code for each integrated circuit is customised to know the first and second addresses of its secret information and inverse string.

Preferably, the first and second addresses are restricted to one of two potential locations in the memory of each integrated circuit, the secret information and the inverse string for each integrated circuit being allocated to the first and second addresses randomly, pseudo-randomly or arbitrarily.

More preferably, the secret information differs between at least two of the integrated circuits.

In accordance with another aspect of the invention, there is provided a plurality of integrated circuits, each of the integrated circuits comprising a processor and non-volatile memory, and including code for running identical software processes, wherein each of the integrated circuits also includes secret information used by the software process, the secret information in each chip being located in a different location in the memory relative to a plurality of the other chips.

Preferably, the code on each integrated circuit is such that the software process of each chip knows the location in memory via which the secret information is accessible.

In accordance with a further aspect of the invention, there is provided a method of manufacturing a plurality of the integrated circuits, including the steps of:

manufacturing a plurality of physical integrated circuits; and

injecting, into the non-volatile memory of each of the integrated circuits:

code for running a software process; and

secret information;

wherein the secret information is positioned in relatively different locations of the non-volatile memories and the code on each integrated circuit is such that the software process of each integrated circuit knows the location in memory via which the secret information is accessible on that integrated circuit.

In accordance with another aspect of the invention, there is provided an integrated circuit comprising a processor and non-volatile memory, the non-volatile memory storing a first number and a second number, wherein the second number is the result of an encryption function taking a third number and secret information as operands, the integrated circuit comprising software configured to decrypt the second number using the first number, thereby to determine the secret information as required.

Preferably, the first and third numbers are the same.

Preferably, the first and second numbers are of the same length.

Preferably, the first number is a random number that was generated using a stochastic process.

Preferably, the encryption function is an XOR logical function.

Preferably, the software is configured to decrypt the second number by performing an XOR logical function using the first and second numbers as operands.

In accordance with a further aspect of the invention, there is provided a method of manufacturing a plurality of integrated circuits in accordance with claim 1, including the steps, for each integrated circuit, of:

determining the first number, the third number and the secret information;

generating the second number by way of an encryption function that uses the third number and the secret information as operands;

storing the first and second numbers on the integrated circuit.

Preferably, the first number is different amongst at least a plurality of the integrated circuits.

Preferably, the first numbers are determined randomly, pseudo-randomly, or arbitrarily.

Preferably, the first number is stored on the integrated circuit first, then extracted therefrom for use in generating the third and thence the second number.

In accordance with another aspect of the invention, there is provided a method of enabling authenticated communication of information between at least a primary entity and each of one or more secondary entities, each of the one or more secondary entities having an identifier associated with it, the method including the steps of:

allocating first secret information to the primary entity;

for each of the one or more secondary entities, determining second secret information, the second secret information being the result of a one way function applied to that second secret entity's identifier and the first secret information;

allocating the second secret information to the or each secondary entity.

Preferably, the identifiers allocated to the secondary entities are generated stochastically, pseudo-randomly or arbitrarily.

Preferably, the one way function is a hash function.

Preferably, the first secret information is a key. More preferably, the one way function is a SHA function.

Preferably, each of the entities is implemented in an integrated circuit.

Preferably, each of the entities is implemented in an integrated circuit separate from the integrated circuits in which the other entities are implemented.

Preferably, one or more of the secondary entities are implemented in a corresponding plurality of integrated circuits.

Preferably, the primary entity is implemented in an integrated circuit.

Preferably, both the primary and secondary entities are implemented in integrated circuits.

Preferably, the first entity wishes to communicate with one of the second entities, the method including the steps, in the first entity, of:

receiving data from the second entity;

using the data and the first secret information to generate the second secret information associated with the second entity.

Preferably, the data contains an identifier for the second entity

Preferably, the first entity wishes to send an authenticated message to the second entity, the method including the steps, in the first entity, of:

using the generated second secret information to sign a message, thereby generating a digital signature;

outputting the message and the digital signature for use by the second entity, which can validate the message by using the digital signature and its own copy of the second secret information.

Preferably, the generated signature includes a nonce from the first entity, and the output from the first entity includes the nonce, thereby enabling the second entity to validate the message using the digital signature, the nonce, and its own copy of the second secret information.

Preferably, the data contains a first nonce.

Preferably, the first entity wishes to send an authenticated message to the second entity, the method including the steps, in the first entity, of:

using the generated second secret information and the first nonce to sign a message, thereby generating a digital signature;

outputting the message and the digital signature for use by the second entity, which can validate the message by using the digital signature and its own copy of the second secret information.

Preferably, the generated signature includes a second nonce from the first entity, and the output from the first entity includes the second nonce, thereby enabling the second entity to validate the message using the digital signature, the first and second nonces, and its own copy of the second secret information.

Preferably, the first entity wishes to send an encrypted message to the second entity, the method including the steps, in the first entity, of:

using the generated second secret information to encrypt a message, thereby generating an encrypted message;

outputting the encrypted message for use by the second entity, which can decrypt the message by using its own copy of the second secret information.

Preferably, the encrypted message includes a nonce from the first entity, and the output from the first entity includes the nonce, thereby enabling the second entity to decrypt the message using the nonce, and its own copy of the second secret information.

Preferably, the first entity wishes to send an encrypted message that incorporates the first nonce to the second entity, the method including the steps, in the first entity, of:

using the generated second secret information to encrypt a message and the first nonce, thereby generating an encrypted message;

outputting the encrypted message for use by the second entity, which can decrypt the encrypted message by using its own copy of the second secret information.

Preferably, the encrypted message includes a second nonce from the first entity, and the output from the first entity includes the second nonce.

In accordance with another aspect of the invention, there is provided a method of generating and sending a message from a first entity, the method including the steps of:

determining a message including an action;

generating an authentication code on the basis of the action and a parameter, the parameter being indicative of an attribute of the action; and

sending the message and authentication code from the first entity.

In accordance with a further aspect of the invention, there is provided a method of generating and sending a message from a first entity, the first entity including an identifier that distinguishes it from a plurality of other entities of a similar type, the method comprising the steps of:

determining a message including an action;

generating an authentication code on the basis of the action and a parameter, the parameter being based on the identifier; and

sending the message and authentication code from the first entity.

Preferably, the action is a function, and the parameter is indicative of the function.

More preferably, the entity is capable of generating messages for each of a plurality of types of function, and the parameter is indicative of the type of function comprised by the message that is sent.

Preferably, the message includes one or more operands of the function.

Preferably, the function is a read function and the one or more operands include an address to be read.

Preferably, the function is a write function and the one or more operands include data to be written.

Preferably, the types of function include at least a read and a write, wherein the authentication step produces a different authentication code depending upon whether the action is a read or a write.

Preferably, the authentication step produces includes authentication codes

In accordance with another aspect of the invention, there is provided a method of generating a first authentication code for a first message for a first function, wherein operands for the first authentication function used to generate the first authentication code include at least part of the first message and at least one identifier associated with the first function.

Preferably, the method further including the steps of verifying the authentication code in accordance with the at least one identifier associated with the first function.

Preferably, the identifier is indicative of a type of the function.

Preferably, the at least one identifier is indicative of the entity generating the authentication code.

Preferably, the at least one identifier is indicative of an entity for which the authentication code is generated.

Preferably, the method includes the step, prior to generating the authentication code, of receiving a request from the entity for the first message, the request including information indicative of an identity of the entity.

In accordance with another aspect of the invention, there is provided a method of preventing a first action associated with a first message from being performed in a target entity, the method including the steps of:

sending the first message to the target entity, the first message being configured to cause the entity to perform the first action and a second action; and

sending a second message to the target entity, the second message being configured to cause the entity to perform a third action;

wherein the entity is configured such that performance of the second action and the third action is mutually incompatible.

In accordance with a further aspect of the invention, there is provided a method of attempting first write and a second write to first and second security fields in a target entity, the method including the steps of:

sending a first message to the target entity, the first message being configured to cause the entity to perform an action and to update the first and second security fields; and

sending a second message to the target entity, the second message being configured to cause the entity to update the first and second security fields;

wherein the security fields have write restrictions associated with them such that updating the security fields in accordance with the first message prevents subsequent updating of the security fields in accordance with the second message, and wherein updating the security fields in accordance with the second

message prevents subsequent updating of the security fields with the first message, and wherein the first action is only performed when updating of the security fields by the first message is successful.

In accordance with a further aspect of the invention, there is provided a method of performing a second attempted write to two security fields in a target entity to prevent subsequent application of an earlier attempted write to a data field, wherein:

each of the first and second security fields has a monotonically changeable write restriction associated with it; and

the first attempted write included a first data value for the data field and first and second security values for the first and second security fields respectively;

the method including the step of sending a second write to the target entity, the second write including third and fourth security values for the first and second security fields respectively, wherein the write restrictions are such that application of the third and fourth security values to the first and second fields are mutually incompatible with application of the first and second security values to the first and second security fields, such that if any of the fields cannot be written to, none of them are written to.

Preferably, the write restrictions:

prevent the second write from being performed in the event that the first write was previously performed; and

prevent the first write from subsequently being performed in the event that the second write is performed.

Preferably, the second write is sent in response to a notification that the first write was not successfully performed.

Preferably, the second write is sent in response to a notification that the first write was not received.

Preferably, the second write is sent after a predetermined time has elapsed after sending the first write without receiving confirmation of the first write being received or successfully performed.

Preferably, the method further includes the step of sending the second write a plurality of times until the second write is successfully received and/or performed.

Preferably, the method further includes the step of verifying the successful second write by performing an authenticated read of the first and second security fields.

Preferably, the method further includes updating a value related to the data field.

Preferably, the write permissions are such that only decrementing or incrementing of the security fields are permitted.

More preferably, the write permissions are such that only decrementing of the security fields is permitted, and wherein:

the value in the first security field prior to the first attempted write was x;

the value in the second security field prior to the first attempted write was y;

the value of the first security data was x-a;

the value of the second security data was y-b;

the value of the third security data being x-c; and

the value of the fourth security data being y-d;

wherein a<b, d<c and a, b, c and d are >0.

It is particularly preferable that a=d and b=c, and even more preferable that a=d=1 and b=c=2.

Preferably, the target entity is a first integrated circuit and the messages are sent by a second integrated circuit.

In accordance with another aspect of the invention, there is provided a method of enabling selection of one or more pieces of secret information stored in a first entity, the first entity also storing at least one value indicative of at least one attribute for each of the one or more pieces of secret information, the method comprising the steps of:

(y) receiving at the first entity a request from a second entity for one or more of the values for one or more of the pieces of secret information stored in the first entity; and

(z) in response to the request, outputting the values to the second entity.

Preferably, each of the pieces of secret information has an associated index and the request in step (y) includes one or more of the indexes to identify those pieces of secret information for which the values are requested.

Preferably, the request in step (y) is a request for the values all of the pieces of secret information and the response in step (z) orders the values such that the second entity can determine which values are associated with which piece of secret information, and can use the order to generate an index for the secret information.

Preferably, the method further includes the steps, in the first entity and following step (z), of:

(iii) receiving a request from the second entity identifying a function and identifying the index of a piece of secret information to be used in performing the function; and

(iv) performing the function using the identified piece of secret information.

Preferably, the method further includes the steps, in the first entity and following step (z), of:

(v) receiving a request from the second entity identifying a function and a piece of secret information to be used in performing the function; and

(vi) performing the function using at least the identified piece of secret information, the identified piece of secret information being identified in the request of step (v) on the basis of at least one of the values output in step (z).

Preferably, the secret information is stored in one or more physical locations of the first entity, and wherein the values are not indicative of those physical locations.

Preferably, the first entity is implemented in a first integrated circuit and the second entity is implemented in a second integrated circuit.

Preferably, the first integrated circuit includes a memory for storing the pieces of secret information and the values.

Preferably, there is a plurality of the first integrated circuits, wherein the physical location of a piece of the secret information having particular attributes is mutually different for at least some of the first integrated circuits.

Preferably, each of the pieces of secret information is a key for use with a corresponding authentication, encryption or decryption function.

Preferably, the integrated circuit is programmed and configured to apply at least one of the authentication, encryption or decryption functions to data using the corresponding key as an operand.

Preferably, the attribute stored for at least one of the pieces of secret information is the length of that at least one of the pieces of secret information.

Preferably, the attribute stored for at least one of the pieces of secret information is the authentication, encryption or decryption type associated with that at least one of the pieces of secret information.

Preferably, the attribute value stored for at least one of the pieces of secret information is indicative of a permission associated with that at least one of the pieces of secret information.

In accordance with a further aspect of the invention, there is provided a system including first and second integrated circuits, the first integrated circuit implementing the first entity of claim 1, the second integrated circuit being programmed and configured to issue a request to the first integrated circuit for attribute values of any secret information stored by the first integrated circuit, and the first integrated circuit being programmed and configured to respond to the request by supplying the attribute values of the pieces of secret information to the external source.

Preferably, the second integrated circuit is a printer controller chip and the first integrated circuit is a peripheral chip in communication with the printer controller.

Preferably, the printer controller chip is installed in a printer and the peripheral chip is in a package that is releasably attachable to the printer via a connector, the connector enabling communication between the printer controller chip and the peripheral chip.

Preferably, the printer controller chip and the peripheral chip are installed in a printer.

Preferably, the package is an ink refill cartridge.

Preferably, the package is a performance setting cartridge configured to set a performance level of the printer.

Preferably, in the event at least one of the pieces of secret information can be altered or updated, the first integrated circuit is configured to alter the attribute values associated with that at least one piece of secret information as required by the alteration or update such that the update or alteration of the at least one piece of secret information and its associated attributes is atomic.

In accordance with another aspect of the invention, there is provided an integrated circuit including an on-board system clock, the integrated circuit including a clock filter configured to determine a temperature of the integrated circuit and to alter an output of the system clock based on the temperature.

Preferably, the clock filter is configured to alter the output of the system clock in the event the temperature is outside a predetermined temperature range.

More preferably, altering the output includes preventing the clock signal from reaching one or more logical circuits on the integrated circuit to which it would otherwise be applied.

It is particularly preferred that the predetermined temperature range is selected such that a temperature-related speed of the system clock output that is not due to the clock filter is within a predetermined frequency range. It is desirable that the frequency range be within an operating frequency of some or all of the logic circuitry to which the system clock is supplied.

In the preferred form of the invention, the clock filter is configured to prevent the system clock from reaching some or all of the logic circuitry in the event the temperature falls below a predetermined level. This level is chosen to be high enough that race conditions, in which the clock speeds up to the point where logic circuitry behaviour becomes unpredictable, are avoided.

In accordance with another aspect of the present invention, there is provided a method of manufacturing a series of integrated circuits having related functionality, the method including the steps of:

(vii) determining an identifier;

(viii) permanently storing the identifier on one of the integrated circuits;

(ix) repeating steps (vii) and (viii) for each integrated circuit in the series;

wherein the identifiers for the series are determined in such a way that knowing the identifier of one of the integrated circuits does not improve the ability of an attacker to determine the identifier of any of the other integrated circuits.

Preferably, the identifier for each integrated circuit is determined using a stochastic mechanism, thereby rendering highly improbable the replication of some or all of the series of identifiers stored on the series of the integrated circuits.

In accordance with another aspect of the invention, there is provided a series of integrated circuits having related functionality, wherein each of the integrated circuits incorporates an identifier determined and stored in accordance with the first aspect.

Preferably, each of the integrated circuits is a printer controller.

In accordance with another aspect of the invention, there is provided a first integrated circuit of a series of integrated circuits according to the second aspect, operable in first and second mode, wherein in the first mode, supervisor code can access the identifier and in the second mode, user code cannot access the identifier.

Preferably, the supervisor mode is available to a program upon verification of that program by a boot program of the integrated circuit.

Preferably, the identifier is mapped into a key K.

Preferably, K is the identifier.

Preferably, K is created by applying a hash function or one-way function to the identifier.

Preferably, the integrated circuit is configured to produce and output a message, the message including a result of encrypting K.

In accordance with another aspect of the invention, there is provided a method of injecting a key into a target integrated circuit, comprising the step of receiving the message generated by the first integrated circuit of claim 10, and transferring a second key into the target integrated circuit, the second key being based on K.

Preferably, the method includes generating the second key by manipulating K with a function.

More preferably, the function uses K and a code associated with the target integrated circuit as operands.

Preferably, the code is a code that is relatively unique to the target integrated circuit.

Preferably, K and the second key enable secure communication between the first integrated circuit and the target integrated circuit.

Preferably, the second integrated circuit is configured to communicate securely with a third integrated circuit, thereby enabling it to act as an intermediary between the first integrated circuit and the third integrated circuit, allowing secure communication therebetween.

Preferably, the first integrated circuit and the third integrated circuit do not share a key for use in the secure communication.

Preferably, the first integrated circuit is a printer controller configured to perform an authenticated read of the third integrated circuit by securely communicating via the second integrated circuit.

Preferably, the authenticated read relates to monitoring or updating usage of a resource.

In accordance with another aspect of the invention,, there is provided a method of enabling software development for an integrated circuit, the integrated circuit being configured to run a boot program that prevents unverified software from subsequently being loaded onto, or run by, the integrated circuit, the method including the step of loading an intermediate program onto the integrated circuit, the intermediate program being customised for a particular one or more of a plurality of potential integrated circuits that, when run on the processor, enables loading or running of code on only the particular one or more integrated circuits.

Preferably, the intermediate program enables the loading or running of unverified code on only the particular one or more integrated circuits.

Preferably, the intermediate program enables the loading or running of the code only when the code includes data indicative of the particular one or more integrated circuits.

Preferably, the intermediate program includes an intermediate boot key, such that the intermediate program enables loading or running of the code only when the code is verified in accordance with the intermediate boot key.

In accordance with another aspect of the invention, there is provided an integrated circuit configured to run a boot program that prevents unverified software from subsequently being loaded onto, or run by, the integrated circuit.

Preferably, the integrated circuit is programmed with program code configured to:

receive software data and a digital signature of the software data

generate a first digest from the software data; and

compare the first digest against a second digest obtained via the digital signature that accompanied the received software data;

wherein the program is considered valid when the first and second digests match.

Preferably, one or both of the digests were generated using a SHA1 function.

Preferably, the boot program contains a plurality of keys, and one of the keys is selected for use in generating the first digest, the key being selected in accordance with a selection criterion.

Preferably, the selection criterion is time-based, a particular one of the keys being selected depending on the time the selection is made.

Preferably, the selection criteria relates to a physical arrangement or configuration of the integrated circuit.

Preferably, the physical arrangement or configuration includes one or more of the following:

one or more pads wired to a reference voltage or to ground;

one or more fuses, one or more of which has been blown; or

the contents of non-volatile memory.

Preferably, the integrated circuit is programmed with program code configured to:

receive encrypted software data,

decrypt the software data; and

validate the software data;

wherein the decrypted software is executed only when the validation is successful.

Preferably, the encryption function is RSA.

Preferably, the boot program contains a plurality of keys, and one of the keys is selected for use in decrypting the software data, the key being selected in accordance with a selection criterion.

Preferably, the selection criterion is time-based, a particular one of the keys being selected depending on the time the selection is made.

Preferably, the selection criteria relates to a physical arrangement or configuration of the integrated circuit.

Preferably, the physical arrangement or configuration includes one or more of the following:

one or more pads wired to a reference voltage or to ground;

one or more fuses, one or more of which has been blown; or

the contents of non-volatile memory.

In accordance with a further aspect of the invention, there is provided a method of passing validated information along a series of entities, the series of entities including a source entity, a series of at least one intermediate entity, and a target entity, wherein each of the entities shares a validation parameter with its immediately neighbouring entity or entities in the series, the method comprising the steps, commencing in the source entity, of:

(x) in the current entity, generating a validation code for the information, the validation code being based on the validation parameter shared between the current entity and the next entity in the series;

(xi) outputting the validation code;

(xii) receiving the validation code in the next entity in the series and making that entity the current entity;

(xiii) verifying the information via the validation code in the current entity using the validation parameter required to verify it;

(xiv) repeating steps (x) to (xi) until the last intermediate entity in the series has output the validation code it generated;

(xv) receiving the validation code in the target entity and verifying the information via the validation code and the validation parameter required to verify it.

Preferably, step (xi) includes the substep of outputting the information.

Preferably, step (xv) includes receiving the information and using it during the verification.

Preferably, step (xii) includes receiving the information and using it during the verification.

Preferably, a controller is in contact with at least some of the entities, the controller being configured to pass the information and/or the validation codes between adjacent entities in the series.

Preferably, step (x) is performed in response to an instruction issued by the controller.

Preferably, the instruction includes a request for the information upon which the validation is to be performed.

Preferably, the validation code is a digital signature produced by a digital signature function using the information and the validation parameter as operands.

Preferably, the validation parameter is a key

Preferably, the key is a symmetric key

Preferably, the validation parameter is an asymmetric key-pair, and the public and private components of the key-pair are in respective neighbouring entities in the series.

Preferably, the validation code is a digital signature generated with a digital signature function using the key or key-pair component, the information and at least one nonce as inputs.

Preferably, the at least one nonce is generated in the current entity in response to an instruction issued by the neighbouring entity of the current entity closer to the target entity.

Preferably, the at least one nonce is randomly, pseudo-randomly or arbitrarily generated number.

Preferably, the at least one nonce is supplied to the current entity in an instruction issued by the neighbouring entity of the current entity closer to the target entity.

Preferably, the nonce is randomly, pseudo-randomly or arbitrarily generated number.

Preferably, a different validation parameter is used for the validation step performed at any two adjacent entities.

Preferably, at least one of the entities is an integrated circuit.

Preferably, the target entity is a printer controller integrated circuit.

Preferably, the source entity is a printer controller integrated circuit.

Preferably, either the source entity or the target entity is a printer controller integrated circuit and the at least one intermediate entity is a verification chip associated with the printer controller.

Preferably, the controller is a printer controller integrated circuit.

Preferably, one of the entities is the controller.

Preferably, the printer controller has a relatively unique identity and the verification chip includes a key based on the unique identity.

Preferably, the source or target entity is an integrated circuit associated with a package that contains ink.

In accordance with a further aspect of the invention, there is provided an integrated circuit incorporating microelectromechanical systems (MEMS), having a total area greater than an area of at least one of the reticles used to manufacture it.

Preferably, the integrated circuit includes a stitch region where the multiple reticle fields overlapped during manufacturing of the integrated circuit.

Preferably, the surface area of the integrated circuit is larger than a single stepping field of a reticle used to manufacture the integrated circuit.

In accordance with another aspect of the invention, there is provided an integrated circuit according to the first aspect, manufactured using at least two different types of reticles.

Preferably, the integrated circuit is manufactured using multiple applications of the same reticle.

Preferably, the integrated circuit is a printhead.

In accordance with a further aspect of the invention, there is provided a method of manufacturing an integrated circuit incorporating MEMS, comprising laying out the integrated circuit using a plurality of overlapping reticles.

Preferably, the overlapping reticles are the same as each other.

Preferably, the reticles are different to each other.

Preferably, the reticles are different lengths.

Preferably, a plurality of the integrated circuits are manufactured on a single substrate wafer, wherein each of the integrated circuits is manufactured incorporating MEMS, comprising laying out of the integrated circuit using a plurality of overlapping reticles.

Preferably, at least some of the integrated circuits are different to each other.

Preferably, the integrated circuits are of different lengths.

In accordance with another aspect of the invention, there is provided a method of laying out an integrated circuit, the method including the steps of:

defing a layout of an integrated circuit;

defining a joint path;

modifying at least one element within an overlap area adjacent the joint path to take into account reticle field overlap during a subsequent manufacturing step.

In accordance with another embodiment of the invention, there is provided, in a system comprising a plurality of consumers of one or more common resources, a method of tracking usage of the one or more common resources, comprising the steps of:

from each consumer, broadcasting to each of the other consumers a value indicative of an amount of the one or more resources consumed;

at each of the consumers, receiving the broadcasted values from the other consumers; and

in each consumer, storing a record of the total of the values that the consumer broadcasted and the values received from the other consumers.

Preferably, a memory stores a total indicative of the sum of all values broadcast by the consumers, the method further comprising the steps of, for each of at least a plurality of the consumers:

performing an authenticated read of the total in the memory;

comparing the total in the consumer's record with the total read from the memory; and

in the event the totals do not match, performing an action.

Preferably, the memory is in one of the consumers and comprises that consumer's record.

Preferably, the action includes halting printing and or outputting an error message;

Preferably, the values are broadcast in a non-secure manner.

Preferably, the value is signless, thereby preventing recrediting of the total in the memory.

Preferably, the consumers are printer controllers.

Preferably, each of the printer controllers controls printing to a different part of print media to be printed.

Preferably, the resource is ink and the one or more values represent one or more corresponding inks consumed by one or more printheads associated with the printer controllers.

BRIEF DESCRIPTION OF THE DRAWINGS

Preferred and other embodiments of the invention will now be described, by way of example only, with reference to the accompanying drawings, in which:

Figure 1 is an example of state machine notation

Figure 2 shows document data flow in a printer

Figure 3 is an example of a single printer controller (hereinafter "SoPEC") A4 simplex printer system

Figure 4 is an example of a dual SoPEC A4 duplex printer system

Figure 5 is an example of a dual SoPEC A3 simplex printer system

Figure 6 is an example of a quad SoPEC A3 duplex printer system

Figure 7 is an example of a SoPEC A4 simplex printing system with an extra SoPEC used as DRAM storage

Figure 8 is an example of an A3 duplex printing system featuring four printing SoPECs

Figure 9 shows pages containing different numbers of bands

Figure 10 shows the contents of a page band

Figure 11 illustrates a page data path from host to SoPEC

Figure 12 shows a page structure

Figure 13 shows a SoPEC system top level partition

Figure 14 shows a SoPEC CPU memory map (not to scale)

Figure 15 is a block diagram of CPU

Figure 16 shows CPU bus transactions

Figure 17 shows a state machine for a CPU subsystem slave

Figure 18 shows a SoPEC CPU memory map (not to scale)

Figure 19 shows an external signal view of a memory management unit (hereinafter "MMU") sub-block partition

Figure 20 shows an internal signal view of an MMU sub-block partition

Figure 21 shows a DRAM write buffer

Figure 22 shows DIU waveforms for multiple transactions

Figure 23 shows a SoPEC LEON CPU core

Figure 24 shows a cache data RAM wrapper

Figure 25 shows a realtime debug unit block diagram

Figure 26 shows interrupt acknowledge cycles for single and pending interrupts

Figure 27 shows an A3 duplex system featuring four printing SoPECs with a single SoPEC DRAM device

Figure 28 is an SCB block diagram

Figure 29 is a logical view of the SCB of figure 28

Figure 30 shows an ISI configuration with four SoPEC devices

Figure 31 shows half-duplex interleaved transmission from ISIMaster to ISISlave

Figure 32 shows ISI transactions

Figure 33 shows an ISI long packet

Figure 34 shows an ISI ping packet

Figure 35 shows a short ISI packet

Figure 36 shows successful transmission of two long packets with sequence bit toggling

Figure 37 shows sequence bit operation with errored long packet

Figure 38 shows sequence bit operation with ACK error

Figure 39 shows an ISI sub-block partition

Figure 40 shows an ISI serial interface engine functional block diagram

Figure 41 is an SIE edge detection and data IO diagram

Figure 42 is an SIE Rx/Tx state machine Tx cycle state diagram

Figure 43 shows an SIE Rx/Tx state machine Tx bit stuff '0' cycle state diagram

Figure 44 shows an SIE Rx/Tx state machine Tx bit stuff '1' cycle state diagram

Figure 45 shows an SIE Rx/Tx state machine Rx cycle state diagram

Figure 46 shows an SIE Tx functional timing example

Figure 47 shows an SIE Rx functional timing example

Figure 48 shows an SIE Rx/Tx FIFO block diagram

Figure 49 shows SIE Rx/Tx FIFO control signal gating

Figure 50 shows an SIE bit stuffing state machine Tx cycle state diagram

Figure 51 shows an SIE bit stripping state machine Rx cycle state diagram

Figure 52 shows a CRC16 generation/checking shift register

Figure 53 shows circular buffer operation

Figure 54 shows duty cycle select

Figure 55 shows a GPIO partition

Figure 56 shows a motor control RTL diagram

Figure 57 is an input de-glitch RTL diagram

Figure 58 is a frequency analyser RTL diagram

Figure 59 shows a brushless DC controller

Figure 60 shows a period measure unit

Figure 61 shows line synch generation logic

Figure 62 shows an ICU partition

Figure 63 is an interrupt clear state diagram

Figure 63A Timers sub-block partition diagram

Figure 64 is a watchdog timer RTL diagram

Figure 65 is a generic timer RTL diagram

Figure 66 is a schematic of a timing pulse generator

Figure 67 is a Pulse generator RTL diagram

Figure 68 shows a SoPEC clock relationship

Figure 69 shows a CPR block partition

Figure 70 shows reset deglitch logic

Figure 71 shows reset synchronizer logic

Figure 72 is a clock gate logic diagram

Figure 73 shows a PLL and Clock divider logic

Figure 74 shows a PLL control state machine diagram

Figure 75 shows a LSS master system-level interface

Figure 76 shows START and STOP conditions

Figure 77 shows an LSS transfer of 2 data bytes

Figure 78 is an example of an LSS write to a QA Chip

Figure 79 is an example of an LSS read from QA Chip

Figure 80 shows an LSS block diagram

Figure 81 shows an LSS multi-command transaction

Figure 82 shows start and stop generation based on previous bus state

Figure 83 shows an LSS master state machine

Figure 84 shows LSS master timing

Figure 85 shows a SoPEC system top level partition

Figure 86 shows an ead bus with 3 cycle random DRAM read accesses

Figure 87 shows interleaving of CPU and non-CPU read accesses

Figure 88 shows interleaving of read and write accesses with 3 cycle random DRAM accesses

Figure 89 shows interleaving of write accesses with 3 cycle random DRAM accesses

Figure 90 shows a read protocol for a SoPEC Unit making a single 256-bit access

Figure 91 shows a read protocol for a SoPEC Unit making a single 256-bit access

Figure 92 shows a write protocol for a SoPEC Unit making a single 256-bit access

Figure 93 shows a protocol for a posted, masked, 128-bit write by the CPU

Figure 94 shows a write protocol shown for CDU making four contiguous 64-bit accesses

Figure 95 shows timeslot-based arbitration

Figure 96 shows timeslot-based arbitration with separate pointers

Figure 97 shows a first example (a) of separate read and write arbitration

Figure 98 shows a second example (b) of separate read and write arbitration

Figure 99 shows a third example (c) ofseparate read and write arbitration

Figure 100 shows a DIU partition

Figure 101 shows a DIU partition

Figure 102 shows multiplexing and address translation logic for two memory instances

Figure 103 shows a timing of dau_dcu_valid, dcu_dau_adv and dcu_dau_wadv

Figure 104 shows a DCU state machine

Figure 105 shows random read timing

Figure 106 shows random write timing

Figure 107 shows refresh timing

Figure 108 shows page mode write timing

Figure 109 shows timing of non-CPU DIU read access

Figure 110 shows timing of CPU DIU read access

Figure 111 shows a CPU DIU read access

Figure 112 shows timing of CPU DIU write access

Figure 113 shows timing of a non-CDU / non-CPU DIU write access

Figure 114 shows timing of CDU DIU write access

Figure 115 shows command multiplexor sub-block partition

Figure 116 shows command multiplexor timing at DIU requestors interface

Figure 117 shows generation of re_arbitrate and re_arbitrate_wadv

Figure 118 shows CPU interface and arbitration logic

Figure 119 shows arbitration timing

Figure 120 shows setting *RotationSync* to enable a new rotation.

Figure 121 shows a timeslot based arbitration

Figure 122 shows a timeslot based arbitration with separate pointers

Figure 123 shows a CPU pre-access write lookahead pointer

Figure 124 shows arbitration hierarchy

Figure 125 shows hierarchical round-robin priority comparison

Figure 126 shows a read multiplexor partition

Figure 127 shows a read command queue (4 deep buffer)

Figure 128 shows state-machines for shared read bus accesses

Figure 129 shows a write multiplexor partition

Figure 130 shows a read multiplexer timing for back-to-back shared read bus transfer

Figure 131 shows a write multiplexer partition

Figure 132 shows a block diagram of a PCU

Figure 133 shows PCU accesses to PEP registers

Figure 134 shows command arbitration and execution

Figure 135 shows DRAM command access state machine

Figure 136 shows an outline of contone data flow with respect to CDU

Figure 137 shows a DRAM storage arrangement for a single line of JPEG 8x8 blocks in 4 colors

Figure 138 shows a read control unit state machine

Figure 139 shows a memory arrangement of JPEG blocks

Figure 140 shows a contone data write state machine

Figure 141 shows lead-in and lead-out clipping of contone data in multi-SoPEC environment

Figure 142 shows a block diagram of CFU

Figure 143 shows a DRAM storage arrangement for a single line of JPEG blocks in 4 colors

Figure 144 shows a block diagram of color space converter

Figure 145 shows a converter/invertor

Figure 146 shows a high-level block diagram of LBD in context

Figure 147 shows a schematic outline of the LBD and the SFU

Figure 148 shows a block diagram of lossless bi-level decoder

Figure 149 shows a stream decoder block diagram

Figure 150 shows a command controller block diagram

Figure 151 shows a state diagram for command controller (CC) state machine

Figure 152 shows a next edge unit block diagram

Figure 153 shows a next edge unit buffer diagram

Figure 193 shows generation of Lastdotintag/1

Figure 194 shows generation of Dot Position Valid

Figure 195 shows generation of write enable to the TFU

Figure 196 shows generation of Tag Dot Number

Figure 197 shows TDI Architecture

Figure 198 shows data flow through the TDI

Figure 199 shows raw tag data interface block diagram

Figure 200 shows an RTDI State Flow Diagram

Figure 201 shows a relationship between TE_endoftagdata, cdu_startofbandstore and cdu_endofbandstore

Figure 202 shows a TDi State Flow Diagram

Figure 203 shows mapping of the tag data to codewords 0-7

Figure 204 shows coding and mapping of uncoded fixed tag data for (15,5) RS encoder

Figure 205 shows mapping of pre-coded fixed tag data

Figure 206 shows coding and mapping of variable tag data for (15,7) RS encoder

Figure 207 shows coding and mapping of uncoded fixed tag data for (15,7) RS encoder

Figure 208 shows mapping of 2D decoded variable tag data

Figure 209 shows a simple block diagram for an m=4 Reed Solomon encoder

Figure 210 shows an RS encoder I/O diagram

Figure 211 shows a (15,5) & (15,7) RS encoder block diagram

Figure 212 shows a (15,5) RS encoder timing diagram

Figure 213 shows a (15,7) RS encoder timing diagram

Figure 214 shows a circuit for multiplying by alpha^3

Figure 215 shows adding two field elements

Figure 216 shows an RS encoder implementation

Figure 217 shows an encoded tag data interface

Figure 218 shows an encoded fixed tag data interface

Figure 219 shows an encoded variable tag data interface

.Figure 220 shows an encoded variable tag data sub-buffer

Figure 221 shows a breakdown of the tag format structure

Figure 222 shows a TFSI FSM state flow diagram

Figure 223 shows a TFS block diagram

Figure 224 shows a table A interface block diagram

Figure 225 shows a table A address generator

Figure 226 shows a table C interface block diagram

Figure 227 shows a table B interface block diagram

Figure 228 shows interfaces between TE, TFU and HCU

Figure 229 shows a 16-byte FIFO in TFU

Figure 230 shows a high level block diagram showing the HCU and its external interfaces

Figure 231 shows a block diagram of the HCU

Figure 232 shows a block diagram of the control unit

Figure 233 shows a block diagram of determine advdot unit

Figure 234 shows a page structure

Figure 235 shows a block diagram of a margin unit

Figure 236 shows a block diagram of a dither matrix table interface

Figure 237 shows an example of reading lines of dither matrix from DRAM

Figure 238 shows a state machine to read dither matrix table

Figure 239 shows a contone dotgen unit

Figure 240 shows a block diagram of dot reorg unit

Figure 241 shows an HCU to DNC interface (also used in DNC to DWU, LLU to PHI)

Figure 242 shows SFU to HCU interface (all feeders to HCU)

Figure 243 shows representative logic of the SFU to HCU interface

Figure 244 shows a high-level block diagram of DNC

Figure 245 shows a dead nozzle table format

Figure 246 shows set of dots operated on for error diffusion

Figure 247 shows a block diagram of DNC

Figure 248 shows a sub-block diagram of ink replacement unit

Figure 249 shows a dead nozzle table state machine

Figure 250 shows logic for dead nozzle removal and ink replacement

Figure 251 shows a sub-block diagram of error diffusion unit

Figure 252 shows a maximum length 32-bit LFSR used for random bit generation

Figure 253 shows a high-level data flow diagram of DWU in context

Figure 254 shows a printhead nozzle layout for 36-nozzle bi-lithic printhead

Figure 255 shows a printhead nozzle layout for a 36-nozzle bi-lithic printhead

Figure 256 shows a dot line store logical representation

Figure 257 shows a conceptual view of printhead row alignment

Figure 258 shows a conceptual view of printhead rows (as seen by the LLU and PHI)

Figure 259 shows a comparison of 1.5x v 2x buffering

Figure 260 shows an even dot order in DRAM (increasing sense, 13320 dot wide line)

Figure 261 shows an even dot order in DRAM (decreasing sense, 13320 dot wide line)

Figure 262 shows a dotline FIFO data structure in DRAM

Figure 263 shows a DWU partition

Figure 264 shows a buffer address generator sub-block

Figure 265 shows a DIU Interface sub-block

Figure 266 shows an interface controller state diagram

Figure 267 shows a high level data flow diagram of LLU in context

Figure 268 shows paper and printhead nozzles relationship (example with D_1=D_2=5)

Figure 269 shows printhead structure and dot generate order

Figure 270 shows an order of dot data generation and transmission

Figure 271 shows a conceptual view of printhead rows

Figure 272 shows a dotline FIFO data structure in DRAM (LLU specification)

Figure 273 shows an LLU partition

Figure 274 shows a dot generator RTL diagram

Figure 275 shows a DIU interface

Figure 276 shows an interface controller state diagram

Figure 277 shows high-level data flow diagram of PHI in context

Figure 278 shows power on reset

Figure 279 shows printhead data rate equalization

Figure 280 shows a printhead structure and dot generate order

Figure 281 shows an order of dot data generation and transmission

Figure 282 shows an order of dot data generation and transmission (single printhead case)

Figure 283 shows printhead interface timing parameters

Figure 284 shows printhead timing with margining

Figure 285 shows a PHI block partition

Figure 286 shows a sync generator state diagram

Figure 287 shows a line sync de-glitch RTL diagram

Figure 288 shows a fire generator state diagram

Figure 289 shows a PHI controller state machine

Figure 290 shows a datapath unit partition

Figure 291 shows a dot order controller state diagram

Figure 292 shows a data generator state diagram

Figure 293 shows data serializer timing

Figure 294 shows a data serializer RTL Diagram

Figure 295 shows printhead types 0 to 7

Figure 296 shows an ideal join between two dilithic printhead segments

Figure 297 shows an example of a join between two bilithic printhead segments

Figure 298 shows printable vs non-printable area under new definition

(looking at colors as if 1 row only)

Figure 299 shows identification of printhead nozzles and shift-register sequences for printheads in arrangement 1

Figure 300 shows demultiplexing of data within the printheads in arrangement 1

Figure 301 shows double data rate signalling for a type 0 printhead in arrangement 1

Figure 302 shows double data rate signalling for a type 1 printhead in arrangement 1

Figure 303 shows identification of printheads nozzles and shift-register sequences for printheads in arrangement 2

Figure 304 shows demultiplexing of data within the printheads in arrangement 2

Figure 305 shows double data rate signalling for a type 0 printhead in arrangement 2

Figure 306 shows double data rate signalling for a type 1 printhead in arrangement 2

Figure 307 shows all 8 printhead arrangements

Figure 308 shows a printhead structure

Figure 309 shows a column Structure

Figure 310 shows a printhead dot shift register dot mapping to page

Figure 311 shows data timing during printing

Figure 312 shows print quality

Figure 313 shows fire and select shift register setup for printing

Figure 314 shows a fire pattern across butt end of printhead chips

Figure 315 shows fire pattern generation

Figure 316 shows determination of select shift register value

Figure 317 shows timing for printing signals

figure 318 shows initialisation of printheads

figure 319 shows a nozzle test latching circuit

figure 320 shows nozzle testing

figure 321 shows a temperature reading

figure 322 shows CMOS testing

figure 323 shows a reticle layout

figure 324 shows a stepper pattern on Wafer

Figure 325 shows relationship between datasets

Figure 326 shows a validation hierarchy

Figure 327 shows development of operating system code

Figure 328 shows protocol for directly verifying reads from ChipR

Figure 329 shows a protocol for signature translation protocol

Figure 330 shows a protocol for a direct authenticated write

Figure 331 shows an alternative protocol for a direct authenticated write

Figure 332 shows a protocol for basic update of permissions

Figure 333 shows a protocol for a multiple-key update

Figure 334 shows a protocol for a single-key authenticated read

Figure 335 shows a protocol for a single-key authenticated write

Figure 336 shows a protocol for a single-key update of permissions

Figure 337 shows a protocol for a single-key update

Figure 338 shows a protocol for a multiple-key single-M authenticated read

Figure 339 shows a protocol for a multiple-key authenticated write

Figure 340 shows a protocol for a multiple-key update of permissions

Figure 341 shows a protocol for a multiple-key update

Figure 342 shows a protocol for a multiple-key multiple-M authenticated read

Figure 343 shows a protocol for a multiple-key authenticated write

Figure 344 shows a protocol for a multiple-key update of permissions

Figure 345 shows a protocol for a multiple-key update

Figure 346 shows relationship of permissions bits to M[n] access bits

Figure 347 shows 160-bit maximal period LFSR

Figure 348 shows clock filter

Figure 349 shows tamper detection line

Figure 350 shows an oversize nMOS transistor layout of Tamper Detection Line

Figure 351 shows a Tamper Detection Line

Figure 352 shows how Tamper Detection Lines cover the Noise Generator

Figure 353 shows a prior art FET Implementation of CMOS inverter

Figure 354 shows non-flashing CMOS

Figure 355 shows components of a printer-based refill device

Figure 356 shows refilling of printers by printer-based refill device

Figure 357 shows components of a home refill station

Figure 358 shows a three-ink reservoir unit

Figure 359 shows refill of ink cartridges in a home refill station

Figure 360 shows components of a commercial refill station

Figure 361 shows an ink reservoir unit

Figure 362 shows refill of ink cartridges in a commercial refill station (showing a single refill unit)

Figure 363 shows equivalent signature generation

Figure 364 shows a basic field definition

Figure 365 shows an example of defining field sizes and positions

Figure 366 shows permissions

Figure 367 shows a first example of permissions for a field

Figure 368 shows a second example of permissions for a field

Figure 369 shows field attributes

Figure 370 shows an output signature generation data format for Read

Figure 371 shows an input signature verification data format for Test

Figure 372 shows an output signature generation data format for Translate

Figure 373 shows an input signature verification data format for WriteAuth

Figure 374 shows input signature data format for ReplaceKey

Figure 375 shows a key replacement map

Figure 376 shows a key replacement map after K_1 is replaced

Figure 377 shows a key replacement process

Figure 378 shows an output signature data format for GetProgramKey

Figure 379 shows transfer and rollback process

Figure 380 shows an upgrade flow

Figure 381 shows authorised ink refill paths in the printing system

Figure 382 shows an input signature verification data format for XferAmount

Figure 383 shows a transfer and rollback process

Figure 384 shows an upgrade flow

Figure 385 shows authorised upgrade paths in the printing system

Figure 386 shows a direct signature validation sequence

Figure 387 shows signature validation using translation

Figure 388 shows setup of preauth field attributes

Figure 389 shows a high level block diagram of QA Chip

Figure 390 shows an analogue unit

Figure 391 shows a serial bus protocol for trimming

Figure 392 shows a block diagram of a trim unit

Figure 393 shows a block diagram of a CPU of the QA chip

Figure 394 shows block diagram of an MIU

Figure 395 shows a block diagram of memory components

Figure 396 shows a first byte sent to an IOU

Figure 397 shows a block diagram of the IOU

Figure 398 shows a relationship between external SDa and SClk and generation of internal signals

Figure 399 shows block diagram of ALU

Figure 400 shows a block diagram of DataSel

Figure 401 shows a block diagram of ROR

Figure 402 shows a block diagram of the ALU's IO block

Figure 403 shows a block diagram of PCU

Figure 404 shows a block diagram of an Address Generator Unit

Figure 405 shows a block diagram for a Counter Unit

Figure 406 shows a block diagram of PMU

Figure 407 shows a state machine for PMU

Figure 408 shows a block diagram of MRU

Figure 409 shows simplified MAU state machine

Figure 410 shows power-on reset behaviour

Figure 411 shows a ring oscillator block diagram

Figure 412 shows a system clock duty cycle

DETAILED DESCRIPTION OF PREFERRED AND OTHER EMBODIMENTS

It will be appreciated that the detailed description that follows takes the form of a highly detailed design of the invention, including supporting hardware and software. A high level of detailed disclosure is provided to ensure that one skilled in the art will have ample guidance for implementing the invention.

Imperative phrases such as "must", "requires", "necessary" and "important" (and similar language) should be read as being indicative of being necessary only for the preferred embodiment actually being described. As such, unless the opposite is clear from the context, imperative wording should not be interpreted as such. Nothing in the detailed description is to be understood as limiting the scope of the invention, which is intended to be defined as widely as is defined in the accompanying claims.

Indications of expected rates, frequencies, costs, and other quantitative values are exemplary and estimated only, and are made in good faith. Nothing in this specification should be read as implying that a particular commercial embodiment is or will be capable of a particular performance level in any measurable area.

It will be appreciated that the principles, methods and hardware described throughout this document can be applied to other fields. Much of the security-related disclosure, for example, can be applied to many other fields that require secure communications between entities, and certainly has application far beyond the field of printers.

SYSTEM OVERVIEW

The preferred of the present invention is implemented in a printer using microelectromechanical systems (MEMS) printheads. The printer can receive data from, for example, a personal computer such as an IBM compatible PC or Apple computer. In other embodiments, the printer can receive data directly from, for example, a digital still or video camera. The particular choice of communication link is not important, and can be based, for example, on USB, Firewire, Bluetooth or any other wireless or hardwired communications protocol.

PRINT SYSTEM OVERVIEW

3 Introduction

This document describes the SoPEC (Small office home office Print Engine Controller) ASIC (Application Specific Integrated Circuit) suitable for use in, for example, SoHo printer products. The SoPEC ASIC is intended to be a low cost solution for bi-lithic printhead control, replacing the multichip solutions in larger more professional systems with a single chip. The increased cost competitiveness is achieved by integrating several systems such as a modified PEC1 printing pipeline, CPU control system, peripherals and memory sub-system onto one SoC ASIC, reducing component count and simplifying board design.

This section will give a general introduction to Memjet printing systems, introduce the components that make a bi-lithic printhead system, describe possible system architectures and show how several SoPECs can be used to achieve A3 and A4 duplex printing. The section "SoPEC ASIC" describes the SoC SoPEC ASIC, with subsections describing the CPU, DRAM and Print Engine Pipeline subsystems. Each section gives a detailed description of the blocks used and their operation within the overall print system. The final section describes the bi-lithic printhead construction and associated implications to the system due to its makeup.

4 Nomenclature

4.1 Bi-lithic Printhead Notation

A bi-lithic based printhead is constructed from 2 printhead ICs of varying sizes. The notation M:N is used to express the size relationship of each IC, where M specifies one printhead IC in inches and N specifies the remaining printhead IC in inches.

The 'SoPEC/MoPEC Bilithic Printhead Reference' document [10] contains a description of the bi-lithic printhead and related terminology.

4.2 Definitions

The following terms are used throughout this specification:

Bi-lithic printhead	Refers to printhead constructed from 2 printhead ICs
CPU	Refers to CPU core, caching system and MMU.
ISI-Bridge chip	A device with a high speed interface (such as USB2.0, Ethernet or IEEE1394) and one or more ISI interfaces. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.
ISIMaster	The ISIMaster is the only device allowed to initiate communication on the Inter Sopec Interface (ISI) bus. The ISIMaster interfaces with the host.
ISISlave	Multi-SoPEC systems will contain one or more ISISlave SoPECs connected to the ISI bus. ISISlaves can only respond to communication initiated by the ISIMaster.
LEON	Refers to the LEON CPU core.
LineSyncMaster	The LineSyncMaster device generates the line synchronisation pulse that all SoPECs in the system must synchronise their line outputs to.
Multi-SoPEC	Refers to SoPEC based print system with multiple SoPEC devices
Netpage	Refers to page printed with tags (normally in infrared ink).
PEC1	Refers to Print Engine Controller version 1, precursor to SoPEC used to control printheads constructed from multiple angled printhead segments.
Printhead IC	Single MEMS IC used to construct bi-lithic printhead
PrintMaster	The PrintMaster device is responsible for coordinating all aspects of the print operation. There may only be one PrintMaster in a system.

QA Chip	Quality Assurance Chip
Storage SoPEC	An ISISlave SoPEC used as a DRAM store and which does not print.
Tag	Refers to pattern which encodes information about its position and orientation which allow it to be optically located and its data contents read.

4.3 ACRONYM AND ABBREVIATIONS

The following acronyms and abbreviations are used in this specification

CFU	Contone FIFO Unit
CPU	Central Processing Unit
DIU	DRAM Interface Unit
DNC	Dead Nozzle Compensator
DRAM	Dynamic Random Access Memory
DWU	DotLine Writer Unit
GPIO	General Purpose Input Output
HCU	Halftoner Compositor Unit
ICU	Interrupt Controller Unit
ISI	Inter SoPEC Interface
LDB	Lossless Bi-level Decoder
LLU	Line Loader Unit
LSS	Low Speed Serial interface
MEMS	Micro Electro Mechanical System
MMU	Memory Management Unit
PCU	SoPEC Controller Unit
PHI	PrintHead Interface
PSS	Power Save Storage Unit
RDU	Real-time Debug Unit
ROM	Read Only Memory
SCB	Serial Communication Block
SFU	Spot FIFO Unit
SMG4	Silverbrook Modified Group 4.
SoPEC	Small office home office Print Engine Controller
SRAM	Static Random Access Memory
TE	Tag Encoder
TFU	Tag FIFO Unit
TIM	Timers Unit
USB	Universal Serial Bus

4.4 PSEUDOCODE NOTATION

In general the pseudocode examples use C like statements with some exceptions.

Symbol and naming convections used for pseudocode.

//	Comment
=	Assignment

==,!=,<,>	Operator equal, not equal, less than, greater than
+,-,*,/,%	Operator addition, subtraction, multiply, divide, modulus
&,|,^,<<,>>,~	Bitwise AND, bitwise OR, bitwise exclusive OR, left shift, right shift, complement
AND,OR,NOT	Logical AND, Logical OR, Logical inversion
[XX:YY]	Array/vector specifier
{a, b, c}	Concatenation operation
++, --	Increment and decrement

4.4.1 Register and signal naming conventions

In general register naming uses the C style conventions with capitalization to denote word delimiters. Signals use RTL style notation where underscore denote word delimiters. There is a direct translation between both convention. For example the *CmdSourceFifo* register is equivalent to *cmd_source_fifo* signal.

4.5 STATE MACHINE NOTATION

State machines should be described using the pseudocode notation outlined above. State machine descriptions use the convention of underline to indicate the cause of a transition from one state to another and plain text (no underline) to indicate the effect of the transition i.e. signal transitions which occur when the new state is entered.

A sample state machine is shown in Figure 1.

5 Printing Considerations

A bi-lithic printhead produces 1600 dpi bi-level dots. On low-diffusion paper, each ejected drop forms a 22.5µm diameter dot. Dots are easily produced in isolation, allowing dispersed-dot dithering to be exploited to its fullest. Since the bi-lithic printhead is the width of the page and operates with a constant paper velocity, color planes are printed in perfect registration, allowing ideal dot-on-dot printing. Dot-on-dot printing minimizes 'muddying' of midtones caused by inter-color bleed.

A page layout may contain a mixture of images, graphics and text. Continuous-tone (contone) images and graphics are reproduced using a stochastic dispersed-dot dither. Unlike a clustered-dot (or amplitude-modulated) dither, a *dispersed-dot* (or frequency-modulated) dither reproduces high spatial frequencies (i.e. image detail) almost to the limits of the dot resolution, while simultaneously reproducing lower spatial frequencies to their full color depth, when spatially integrated by the eye.

A *stochastic* dither matrix is carefully designed to be free of objectionable low-frequency patterns when tiled across the image. As such its size typically exceeds the minimum size required to support a particular number of intensity levels (e.g. 16×16× 8 bits for 257 intensity levels).

Human contrast sensitivity peaks at a spatial frequency of about 3 cycles per degree of visual field and then falls off logarithmically, decreasing by a factor of 100 beyond about 40 cycles per degree and becoming immeasurable beyond 60 cycles per degree [25][25]. At a normal viewing distance of 12 inches (about 300mm), this translates roughly to 200-300 cycles per inch (cpi) on the printed page, or 400-600 samples per inch according to Nyquist's theorem.

In practice, contone resolution above about 300 ppi is of limited utility outside special applications such as medical imaging. Offset printing of magazines, for example, uses contone resolutions in the range 150 to 300 ppi. Higher resolutions contribute slightly to color error through the dither.

Black text and graphics are reproduced directly using bi-level black dots, and are therefore not anti-aliased (i.e. low-pass filtered) before being printed. Text should therefore be *supersampled* beyond the perceptual limits discussed above, to produce smoother edges when spatially integrated by the eye. Text resolution up to about 1200 dpi continues to contribute to perceived text sharpness (assuming low-diffusion paper, of course).

A Netpage printer, for example, may use a contone resolution of 267 ppi (i.e. 1600 dpi / 6), and a black text and graphics resolution of 800 dpi. A high end office or departmental printer may use a contone resolution of 320 ppi (1600 dpi / 5) and a black text and graphics resolution of 1600 dpi. Both formats are capable of exceeding the quality of commercial (offset) printing and photographic reproduction.

6 Document Data Flow

6.1 CONSIDERATIONS

Because of the page-width nature of the bi-lithic printhead, each page must be printed at a constant speed to avoid creating visible artifacts. This means that the printing speed can't be varied to match the input data rate. Document rasterization and document printing are therefore decoupled to ensure the printhead has a constant supply of data. A page is never printed until it is fully rasterized. This can be achieved by storing a compressed version of each rasterized page image in memory. This decoupling also allows the RIP(s) to run ahead of the printer when rasterizing simple pages, buying time to rasterize more complex pages.

Because contone color images are reproduced by stochastic dithering, but black text and line graphics are reproduced directly using dots, the compressed page image format contains a separate foreground bi-level black layer and background contone color layer. The black layer is composited over the contone layer after the contone layer is dithered (although the contone layer has an optional black component). A final layer of Netpage tags (in infrared or black ink) is optionally added to the page for printout.

Figure 2 shows the flow of a document from computer system to printed page.

At 267 ppi for example, a A4 page (8.26 inches × 11.7 inches) of contone CMYK data has a size of 26.3MB. At 320 ppi, an A4 page of contone data has a size of 37.8MB. Using lossy contone compression algorithms such as JPEG [27], contone images compress with a ratio up to 10:1 without noticeable loss of quality, giving compressed page sizes of 2.63MB at 267 ppi and 3.78 MB at 320 ppi.

At 800 dpi, a A4 page of bi-level data has a size of 7.4MB. At 1600 dpi, a Letter page of bi-level data has a size of 29.5 MB. Coherent data such as text compresses very well. Using lossless bi-level compression algorithms such as SMG4 fax as discussed in Section 8.1.2.3.1, ten-point plain text compresses with a ratio of about 50:1. Lossless bi-level compression across an average page is about 20:1 with 10:1 possible for pages which compress poorly. The requirement for SoPEC is to be able to print text at 10:1 compression. Assuming 10:1 compression gives compressed page sizes of 0.74 MB at 800 dpi, and 2.95 MB at 1600 dpi.

Once dithered, a page of CMYK contone image data consists of 116MB of bi-level data. Using lossless bi-level compression algorithms on this data is pointless precisely because the optimal dither is stochastic - i.e. since it introduces hard-to-compress disorder.

Netpage tag data is optionally supplied with the page image. Rather than storing a compressed bi-level data layer for the Netpage tags, the tag data is stored in its raw form. Each tag is supplied up to 120 bits of raw variable data (combined with up to 56 bits of raw fixed data) and covers up to a 6mm × 6mm area (at 1600 dpi). The absolute maximum number of tags on a A4 page is 15,540 when the tag is only 2mm × 2mm (each tag is 126 dots × 126 dots, for a total coverage of 148 tags × 105 tags). 15,540 tags of 128 bits per tag gives a compressed tag page size of 0.24 MB.

The multi-layer compressed page image format therefore exploits the relative strengths of lossy JPEG contone image compression, lossless bi-level text compression, and tag encoding. The format is compact enough to be storage-efficient, and simple enough to allow straightforward real-time expansion during printing.

Since text and images normally don't overlap, the normal worst-case page image size is image only, while the normal best-case page image size is text only. The addition of worst case Netpage tags adds 0.24MB to the page image size. The worst-case page image size is text over image plus tags. The average page size assumes a quarter of an average page contains images. Table 1 shows data sizes for compressed Letter page for these different options.

Table 1. Data sizes for A4 page (8.26 inches × 11.7 inches)

	267 ppi contone 800 dpi bi-level	320 ppi contone 1600 dpi bi-level
Image only (contone), 10:1 compression	2.63 MB	3.78 MB
Text only (bi-level), 10:1 compression	0.74 MB	2.95 MB
Netpage tags, 1600 dpi	0.24 MB	0.24 MB
Worst case (text + image + tags)	3.61 MB	6.67 MB
Average (text + 25% image + tags)	1.64 MB	4.25 MB

6.2 Document Data Flow

The Host PC rasterizes and compresses the incoming document on a page by page basis. The page is restructured into bands with one or more bands used to construct a page. The compressed data is then transferred to the SoPEC device via the USB link. A complete band is stored in SoPEC embedded memory. Once the band transfer is complete the SoPEC device reads the compressed data, expands the band, normalizes contone, bi-level and tag data to 1600 dpi and transfers the resultant calculated dots to the bi-lithic printhead.

The document data flow is

- The RIP software rasterizes each page description and compress the rasterized page image.
- The infrared layer of the printed page optionally contains encoded Netpage [5] tags at a programmable density.

- The compressed page image is transferred to the SoPEC device via the USB normally on a band by band basis.
- The print engine takes the compressed page image and starts the page expansion.
- The first stage page expansion consists of 3 operations performed in parallel
- expansion of the JPEG-compressed contone layer
- expansion of the SMG4 fax compressed bi-level layer
- encoding and rendering of the bi-level tag data.
- The second stage dithers the contone layer using a programmable dither matrix, producing up to four bi-level layers at full-resolution.
- The second stage then composites the bi-level tag data layer, the bi-level SMG4 fax de-compressed layer and up to four bi-level JPEG de-compressed layers into the full-resolution page image.
- A fixative layer is also generated as required.
- The last stage formats and prints the bi-level data through the bi-lithic printhead via the printhead interface.

The SoPEC device can print a full resolution page with 6 color planes. Each of the color planes can be generated from compressed data through any channel (either JPEG compressed, bi-level SMG4 fax compressed, tag data generated, or fixative channel created) with a maximum number of 6 data channels from page RIP to bi-lithic printhead color planes.

The mapping of data channels to color planes is programmable, this allows for multiple color planes in the printhead to map to the same data channel to provide for redundancy in the printhead to assist dead nozzle compensation.

Also a data channel could be used to gate data from another data channel. For example in stencil mode, data from the bilevel data channel at 1600 dpi can be used to filter the contone data channel at 320 dpi, giving the effect of 1600 dpi contone image.

6.3 PAGE CONSIDERATIONS DUE TO SOPEC

The SoPEC device typically stores a complete page of document data on chip. The amount of storage available for compressed pages is limited to 2Mbytes, imposing a fixed maximum on compressed page size. A comparison of the compressed image sizes in Table 2 indicates that SoPEC would not be capable of printing worst case pages unless they are split into bands and printing commences before all the bands for the page have been downloaded. The page sizes in the table are shown for comparison purposes and would be considered reasonable for a professional level printing system. The SoPEC device is aimed at the consumer level and would not be required to print pages of that complexity. Target document types for the SoPEC device are shown Table 2.

Table 2. Page content targets for SoPEC

Page Content Description	Calculation	Size

		(MByte)
Best Case picture Image, 267ppi with 3 colors, A4 size	8.26x11.7x267x267x3 @10:1	1.97
Full page text, 800dpi A4 size	8.26x11.7x800x800 @ 10:1	0.74
Mixed Graphics and Text - Image of 6 inches x 4 inches @ 267 ppi and 3 colors - Remaining area text ~73 inches^2, 800 dpi	6x4x267x267x3 @ 5:1 800x800x73 @ 10:1	1.55
Best Case Photo, 3 Colors, 6.6 MegaPixel Image	6.6 Mpixel @ 10:1	2.00

If a document with more complex pages is required, the page RIP software in the host PC can determine that there is insufficient memory storage in the SoPEC for that document. In such cases the RIP software can take two courses of action. It can increase the compression ratio until the compressed page size will fit in the SoPEC device, at the expense of document quality, or divide the page into bands and allow SoPEC to begin printing a page band before all bands for that page are downloaded. Once SoPEC starts printing a page it cannot stop, if SoPEC consumes compressed data faster than the bands can be downloaded a buffer underrun error could occur causing the print to fail. A buffer underrun occurs if a line synchronisation pulse is received before a line of data has been transferred to the printhead.

Other options which can be considered if the page does not fit completely into the compressed page store are to slow the printing or to use multiple SoPECs to print parts of the page. A Storage SoPEC (Section 7.2.5) could be added to the system to provide guaranteed bandwidth data delivery. The print system could also be constructed using an ISI-Bridge chip (Section 7.2.6) to provide guaranteed data delivery.

7 Memjet Printer Architecture

The SoPEC device can be used in several printer configurations and architectures.

In the general sense every SoPEC based printer architecture will contain:

- One or more SoPEC devices.
- One or more bi-lithic printheads.
- Two or more LSS busses.
- Two or more QA chips.
- USB 1.1 connection to host or ISI connection to Bridge Chip.
- ISI bus connection between SoPECs (when multiple SoPECs are used).

Some example printer configurations as outlined in Section 7.2. The various system components are outlined briefly in Section 7.1.

7.1 System Components

7.1.1 SoPEC Print Engine Controller

The SoPEC device contains several system on a chip (SoC) components, as well as the print engine pipeline control application specific logic.

7.1.1.1 Print Engine Pipeline (PEP) Logic

The PEP reads compressed page store data from the embedded memory, optionally decompresses the data and formats it for sending to the printhead. The print engine pipeline functionality includes expanding the page image, dithering the contone layer, compositing the black layer over the contone layer, rendering of Netpage tags, compensation for dead nozzles in the printhead, and sending the resultant image to the bi-lithic printhead.

7.1.1.2 Embedded CPU

SoPEC contains an embedded CPU for general purpose system configuration and management. The CPU performs page and band header processing, motor control and sensor monitoring (via the GPIO) and other system control functions. The CPU can perform buffer management or report buffer status to the host. The CPU can optionally run vendor application specific code for general print control such as paper ready monitoring and LED status update.

7.1.1.3 Embedded Memory Buffer

A 2.5Mbyte embedded memory buffer is integrated onto the SoPEC device, of which approximately 2Mbytes are available for compressed page store data. A compressed page is divided into one or more bands, with a number of bands stored in memory. As a band of the page is consumed by the PEP for printing a new band can be downloaded. The new band may be for the current page or the next page.

Using banding it is possible to begin printing a page before the complete compressed page is downloaded, but care must be taken to ensure that data is always available for printing or a buffer underrun may occur.

An Storage SoPEC acting as a memory buffer (Section 7.2.5) or an ISI-Bridge chip with attached DRAM (Section 7.2.6) could be used to provide guaranteed data delivery.

7.1.1.4 Embedded USB 1.1 Device

The embedded USB 1.1 device accepts compressed page data and control commands from the host PC, and facilitates the data transfer to either embedded memory or to another SoPEC device in multi-SoPEC systems.

7.1.2 Bi-lithic Printhead

The printhead is constructed by abutting 2 printhead ICs together. The printhead ICs can vary in size from 2 inches to 8 inches, so to produce an A4 printhead several combinations are possible. For example two printhead ICs of 7 inches and 3 inches could be used to create a A4 printhead (the notation is 7:3). Similarly 6 and 4 combination (6:4), or 5:5 combination. For an A3 printhead it can be constructed from 8:6 or an 7:7 printhead IC combination. For photographic printing smaller printheads can be constructed.

7.1.3 LSS interface bus

Each SoPEC device has 2 LSS system buses for communication with QA devices for system authentication and ink usage accounting. The number of QA devices per bus and their position in the system is unrestricted with the exception that *PRINTER_QA* and *INK_QA* devices should be on separate LSS busses.

7.1.4 QA devices

Each SoPEC system can have several QA devices. Normally each printing SoPEC will have an associated *PRINTER_QA*. Ink cartridges will contain an *INK_QA* chip. *PRINTER_QA* and *INK_QA* devices should be on separate LSS busses. All QA chips in the system are physically identical with flash memory contents defining *PRINTER_QA* from *INK_QA* chip.

7.1.5 ISI interface

The Inter-SoPEC Interface (ISI) provides a communication channel between SoPECs in a multi-SoPEC system. The ISIMaster can be SoPEC device or an ISI-Bridge chip depending on the printer configuration. Both compressed data and control commands are transferred via the interface.

7.1.6 ISI-Bridge Chip

A device, other than a SoPEC with a USB connection, which provides print data to a number of slave SoPECs. A bridge chip will typically have a high bandwidth connection, such as USB2.0, Ethernet or IEEE1394, to a host and may have an attached external DRAM for compressed page storage. A bridge chip would have one or more ISI interfaces. The use of multiple ISI buses would allow the construction of independent print systems within the one printer. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.

7.2 POSSIBLE SOPEC SYSTEMS

Several possible SoPEC based system architectures exist. The following sections outline some possible architectures. It is possible to have extra SoPEC devices in the system used for DRAM storage. The QA chip configurations shown are indicative of the flexibility of LSS bus architecture, but not limited to those configurations.

7.2.1 A4 Simplex with 1 SoPEC device

In Figure 3, a single SoPEC device can be used to control two printhead ICs. The SoPEC receives compressed data through the USB device from the host. The compressed data is processed and transferred to the printhead.

7.2.2 A4 Duplex with 2 SoPEC devices

In Figure 4, two SoPEC devices are used to control two bi-lithic printheads, each with two printhead ICs. Each bi-lithic printhead prints to opposite sides of the same page to achieve duplex printing. The SoPEC connected to the USB is the ISIMaster SoPEC, the remaining SoPEC is an ISISlave. The ISIMaster receives all the compressed page data for both SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A4 page every 2 seconds in this configuration since the USB 1.1 connection to the host may not have enough bandwidth. An alternative would be for each SoPEC to have its own USB 1.1 connection. This would allow a faster average print speed.

7.2.3 A3 Simplex with 2 SoPEC devices

In Figure 5, two SoPEC devices are used to control one A3 bi-lithic printhead. Each SoPEC controls only one printhead IC (the remaining PHI port typically remains idle). This system uses the SoPEC with the USB connection as the ISIMaster. In this dual SoPEC configuration the compressed page store data is split across 2 SoPECs giving a total of 4Mbyte page store, this allows the system to

use compression rates as in an A4 architecture, but with the increased page size of A3. The ISIMaster receives all the compressed page data for all SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A3 page every 2 seconds in this configuration since the USB 1.1 connection to the host will only have enough bandwidth to supply 2Mbytes every 2 seconds. Pages which require more than 2MBytes every 2 seconds will therefore print more slowly. An alternative would be for each SoPEC to have its own USB 1.1 connection. This would allow a faster average print speed.

7.2.4 A3 Duplex with 4 SoPEC devices

In Figure 6 a 4 SoPEC system is shown. It contains 2 A3 bi-lithic printheads, one for each side of an A3 page. Each printhead contain 2 printhead ICs, each printhead IC is controlled by an independent SoPEC device, with the remaining PHI port typically unused. Again the SoPEC with USB 1.1 connection is the ISIMaster with the other SoPECs as ISISlaves. In total, the system contains 8Mbytes of compressed page store (2Mbytes per SoPEC), so the increased page size does not degrade the system print quality, from that of an A4 simplex printer. The ISIMaster receives all the compressed page data for all SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A3 page every 2 seconds in this configuration since the USB 1.1 connection to the host will only have enough bandwidth to supply 2Mbytes every 2 seconds. Pages which require more than 2MBytes every 2 seconds will therefore print more slowly. An alternative would be for each SoPEC or set of SoPECs on the same side of the page to have their own USB 1.1 connection (as ISISlaves may also have direct USB connections to the host). This would allow a faster average print speed.

7.2.5 SoPEC DRAM storage solution: A4 Simplex with 1 printing SoPEC and 1 memory SoPEC

Extra SoPECs can be used for DRAM storage e.g. in Figure 7 an A4 simplex printer can be built with a single extra SoPEC used for DRAM storage. The DRAM SoPEC can provide guaranteed bandwidth delivery of data to the printing SoPEC. SoPEC configurations can have multiple extra SoPECs used for DRAM storage.

7.2.6 ISI-Bridge chip solution: A3 Duplex system with 4 SoPEC devices

In Figure 8, an ISI-Bridge chip provides slave-only ISI connections to SoPEC devices. Figure 8 shows a ISI-Bridge chip with 2 separate ISI ports. The ISI-Bridge chip is the ISIMaster on each of the ISI busses it is connected to. All connected SoPECs are ISISlaves. The ISI-Bridge chip will typically have a high bandwidth connection to a host and may have an attached external DRAM for compressed page storage.

An alternative to having a ISI-Bridge chip would be for each SoPEC or each set of SoPECs on the same side of a page to have their own USB 1.1 connection. This would allow a faster average print speed.

8 Page Format and Printflow

When rendering a page, the RIP produces a page header and a number of bands (a non-blank page requires at least one band) for a page. The page header contains high level rendering

parameters, and each band contains compressed page data. The size of the band will depend on the memory available to the RIP, the speed of the RIP, and the amount of memory remaining in SoPEC while printing the previous band(s). Figure 9 shows the high level data structure of a number of pages with different numbers of bands in the page.

Each compressed band contains a mandatory band header, an optional bi-level plane, optional sets of interleaved contone planes, and an optional tag data plane (for Netpage enabled applications). Since each of these planes is optional[1], the band header specifies which planes are included with the band. Figure 10 gives a high-level breakdown of the contents of a page band.

A single SoPEC has maximum rendering restrictions as follows:

- 1 bi-level plane
- 1 contone interleaved plane set containing a maximum of 4 contone planes
- 1 tag data plane
- a bi-lithic printhead with a maximum of 2 printhead ICs

The requirement for single-sided A4 single SoPEC printing is

- average contone JPEG compression ratio of 10:1, with a local minimum compression ratio of 5:1 for a single line of interleaved JPEG blocks.
- average bi-level compression ratio of 10:1, with a local minimum compression ratio of 1:1 for a single line.

If the page contains rendering parameters that exceed these specifications, then the RIP or the Host PC must split the page into a format that can be handled by a single SoPEC.

In the general case, the SoPEC CPU must analyze the page and band headers and generate an appropriate set of register write commands to configure the units in SoPEC for that page. The various bands are passed to the destination SoPEC(s) to locations in DRAM determined by the host.

The host keeps a memory map for the DRAM, and ensures that as a band is passed to a SoPEC, it is stored in a suitable free area in DRAM. Each SoPEC is connected to the ISI bus or USB bus via its Serial communication Block (SCB). The SoPEC CPU configures the SCB to allow compressed data bands to pass from the USB or ISI through the SCB to SoPEC DRAM. Figure 11 shows an example data flow for a page destined to be printed by a single SoPEC. Band usage information is generated by the individual SoPECs and passed back to the host.

SoPEC has an addressing mechanism that permits circular band memory allocation, thus facilitating easy memory management. However it is not strictly necessary that all bands be stored together. As long as the appropriate registers in SoPEC are set up for each band, and a given band is contiguous[2], the memory can be allocated in any way.

---

[1]Although a band must contain at least one plane

[2]Contiguous allocation also includes wrapping around in SoPEC's band store memory.

8.1 PRINT ENGINE EXAMPLE PAGE FORMAT

This section describes a possible format of compressed pages expected by the embedded CPU in SoPEC. The format is generated by software in the host PC and interpreted by embedded software in SoPEC. This section indicates the type of information in a page format structure, but implementations need not be limited to this format. The host PC can optionally perform the majority of the header processing.

The compressed format and the print engines are designed to allow real-time page expansion during printing, to ensure that printing is never interrupted in the middle of a page due to data underrun.

The page format described here is for a single black bi-level layer, a contone layer, and a Netpage tag layer. The black bi-level layer is defined to composite *over* the contone layer.

The black bi-level layer consists of a bitmap containing a 1-bit *opacity* for each pixel. This black layer *matte* has a resolution which is an integer or non-integer factor of the printer's dot resolution. The highest supported resolution is 1600 dpi, i.e. the printer's full dot resolution.

The contone layer, optionally passed in as YCrCb, consists of a 24-bit CMY or 32-bit CMYK *color* for each pixel. This contone image has a resolution which is an integer or non-integer factor of the printer's dot resolution. The requirement for a single SoPEC is to support 1 side per 2 seconds A4/Letter printing at a resolution of 267 ppi, i.e. one-sixth the printer's dot resolution.

Non-integer scaling can be performed on both the contone and bi-level images. Only integer scaling can be performed on the tag data.

The black bi-level layer and the contone layer are both in compressed form for efficient storage in the printer's internal memory.

8.1.1 Page structure

A single SoPEC is able to print with full edge bleed for Letter and A3 via different stitch part combinations of the bi-lithic printhead. It imposes no margins and so has a printable page area which corresponds to the size of its paper. The target page size is constrained by the printable page area, less the explicit (target) left and top margins specified in the page description. These relationships are illustrated below.

8.1.2 Compressed page format

Apart from being implicitly defined in relation to the printable page area, each page description is complete and self-contained. There is no data stored separately from the page description to which the page description refers.[3] The page description consists of a page header which describes the size and resolution of the page, followed by one or more page bands which describe the actual page content.

8.1.2.1 Page header

Table 3 shows an example format of a page header.

---

[3]SoPEC relies on dither matrices and tag structures to have already been set up, but these are not considered to be part of a general page format. It is trivial to extend the page format to allow exact specification of dither matrices and tag structures.

Table 3. Page header format

field	format	description
signature	16-bit integer	Page header format signature.
version	16-bit integer	Page header format version number.
structure size	16-bit integer	Size of page header.
band count	16-bit integer	Number of bands specified for this page.
target resolution (dpi)	16-bit integer	Resolution of target page. This is always 1600 for the Memjet printer.
target page width	16-bit integer	Width of target page, in dots.
target page height	32-bit integer	Height of target page, in dots.
target left margin for black and contone	16-bit integer	Width of target left margin, in dots, for black and contone.
target top margin for black and contone	16-bit integer	Height of target top margin, in dots, for black and contone.
target right margin for black and contone	16-bit integer	Width of target right margin, in dots, for black and contone.
target bottom margin for black and contone	16-bit integer	Height of target bottom margin, in dots, for black and contone.
target left margin for tags	16-bit integer	Width of target left margin, in dots, for tags.
target top margin for tags	16-bit integer	Height of target top margin, in dots, for tags.
target right margin for tags	16-bit integer	Width of target right margin, in dots, for tags.
target bottom margin for tags	16-bit integer	Height of target bottom margin, in dots, for tags.
generate tags	16-bit integer	Specifies whether to generate tags for this page (0 - no, 1 - yes).
fixed tag data	128-bit integer	This is only valid if generate tags is set.
tag vertical scale factor	16-bit integer	Scale factor in vertical direction from tag data resolution to target resolution. Valid range = 1-511. Integer scaling only
tag horizontal scale factor	16-bit integer	Scale factor in horizontal direction from tag data resolution to target resolution. Valid range = 1-511. Integer scaling only.
bi-level layer vertical scale factor	16-bit integer	Scale factor in vertical direction from bi-level resolution to target resolution (must be 1 or greater). May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.

bi-level layer horizontal scale factor	16-bit integer	Scale factor in horizontal direction from bi-level resolution to target resolution (must be 1 or greater). May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
bi-level layer page width	16-bit integer	Width of bi-level layer page, in pixels.
bi-level layer page height	32-bit integer	Height of bi-level layer page, in pixels.
contone flags	16 bit integer	Defines the color conversion that is required for the JPEG data. Bits 2-0 specify how many contone planes there are (e.g. 3 for CMY and 4 for CMYK). Bit 3 specifies whether the first 3 color planes need to be converted back from YCrCb to CMY. Only valid if b2-0 = 3 or 4. 0 - no conversion, leave JPEG colors alone 1 - color convert. Bits 7-4 specifies whether the YCrCb was generated directly from CMY, or whether it was converted to RGB first via the step: R = 255-C, G = 255-M, B = 255-Y. Each of the color planes can be individually inverted. Bit 4: 0 - do not invert color plane 0 1 - invert color plane 0 Bit 5: 0 - do not invert color plane 1 1 - invert color plane 1 Bit 6: 0 - do not invert color plane 2 1 - invert color plane 2 Bit 7: 0 - do not invert color plane 3 1 - invert color plane 3 Bit 8 specifies whether the contone data is JPEG compressed or non-compressed: 0 - JPEG compressed 1 - non-compressed The remaining bits are reserved (0).
contone vertical scale factor	16-bit integer	Scale factor in vertical direction from contone channel resolution to target resolution. Valid

		range = 1-255. May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
contone horizontal scale factor	16-bit integer	Scale factor in horizontal direction from contone channel resolution to target resolution. Valid range = 1-255. May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
contone page width	16-bit integer	Width of contone page, in contone pixels.
contone page height	32-bit integer	Height of contone page, in contone pixels.
reserved	up to 128 bytes	Reserved and 0 pads out page header to multiple of 128 bytes.

The page header contains a signature and version which allow the CPU to identify the page header format. If the signature and/or version are missing or incompatible with the CPU, then the CPU can reject the page.

The contone flags define how many contone layers are present, which typically is used for defining whether the contone layer is CMY or CMYK. Additionally, if the color planes are CMY, they can be optionally stored as YCrCb, and further optionally color space converted from CMY directly or via RGB. Finally the contone data is specified as being either JPEG compressed or non-compressed.

The page header defines the resolution and size of the target page. The bi-level and contone layers are clipped to the target page if necessary. This happens whenever the bi-level or contone scale factors are not factors of the target page width or height.

The target left, top, right and bottom margins define the positioning of the target page within the printable page area.

The tag parameters specify whether or not Netpage tags should be produced for this page and what orientation the tags should be produced at (landscape or portrait mode). The fixed tag data is also provided.

The contone, bi-level and tag layer parameters define the page size and the scale factors.

8.1.2.2 Band format

Table 4 shows the format of the page band header.

Table 4. Band header format

field	format	description
signature	16-bit integer	Page band header format signature.

version	16-bit integer	Page band header format version number.
structure size	16-bit integer	Size of page band header.
bi-level layer band height	16-bit integer	Height of bi-level layer band, in black pixels.
bi-level layer band data size	32-bit integer	Size of bi-level layer band data, in bytes.
contone band height	16-bit integer	Height of contone band, in contone pixels.
contone band data size	32-bit integer	Size of contone plane band data, in bytes.
tag band height	16-bit integer	Height of tag band, in dots.
tag band data size	32-bit integer	Size of unencoded tag data band, in bytes. Can be 0 which indicates that no tag data is provided.
reserved	up to 128 bytes	Reserved and 0 pads out band header to multiple of 128 bytes.

The bi-level layer parameters define the height of the black band, and the size of its compressed band data. The variable-size black data follows the page band header.

The contone layer parameters define the height of the contone band, and the size of its compressed page data. The variable-size contone data follows the black data.

The tag band data is the set of variable tag data half-lines as required by the tag encoder. The format of the tag data is found in Section 26.5.2. The tag band data follows the contone data.

Table 5 shows the format of the variable-size compressed band data which follows the page band header.

Table 5. Page band data format

field	format	Description
black data	Modified G4 facsimile bitstream[4]	Compressed bi-level layer.
contone data	JPEG bytestream	Compressed contone datalayer.
tag data map	Tag data array	Tag data format. See Section 26.5.2.

The start of each variable-size segment of band data should be aligned to a 256-bit DRAM word boundary.

The following sections describe the format of the compressed bi-level layers and the compressed contone layer. section 26.5.1 on page 442 describes the format of the tag data structures.

8.1.2.3 Bi-level data compression

The (typically 1600 dpi) black bi-level layer is losslessly compressed using Silverbrook Modified Group 4 (SMG4) compression which is a version of Group 4 Facsimile compression [22] without Huffman and with simplified run length encodings. Typically compression ratios exceed 10:1. The encoding are listed in Table 6 and Table 7.

Table 6. Bi-Level group 4 facsimile style compression encodings

	Encoding	Description
same as Group 4	1000	Pass Command: a0 ← b2, skip next two edges

[4] See section 8.1.2.3 on page 63 for note regarding the use of this standard

Facsimile		
	1	Vertical(0): a0 ← b1, color = !color
	110	Vertical(1): a0 ← b1 + 1, color = !color
	010	Vertical(-1): a0 ← b1 - 1, color = !color
	110000	Vertical(2): a0 ← b1 + 2, color = !color
	010000	Vertical(-2): a0 ← b1 - 2, color = !color
Unique to this implementation	100000	Vertical(3): a0 ← b1 + 3, color = !color
	000000	Vertical(-3): a0 ← b1 - 3, color = !color
	<RL><RL>100	Horizontal: a0 ← a0 + <RL> + <RL>

SMG4 has a pass through mode to cope with local negative compression. Pass through mode is activated by a special run-length code. Pass through mode continues to either end of line or for a pre-programmed number of bits, whichever is shorter. The special run-length code is always executed as a run-length code, followed by pass through. The pass through escape code is a medium length run-length with a run of less than or equal to 31.

Table 7. Run length (RL) encodings

	Encoding	Description
Unique to this implementation	RRRRR1	Short Black Runlength (5 bits)
	RRRRR1	Short White Runlength (5 bits)
	RRRRRRRRRR10	Medium Black Runlength (10 bits)
	RRRRRRRR10	Medium White Runlength (8 bits)
	RRRRRRRRRR10	Medium Black Runlength with RRRRRRRRRR <= 31, Enter pass through
	RRRRRRRR10	Medium White Runlength with RRRRRRRR <= 31, Enter pass through
	RRRRRRRRRRRRRR00	Long Black Runlength (15 bits)
	RRRRRRRRRRRRRR00	Long White Runlength (15 bits)

Since the compression is a bitstream, the encodings are read right (least significant bit) to left (most significant bit). The run lengths given as RRRR in Table are read in the same way (least significant bit at the right to most significant bit at the left).

Each band of bi-level data is optionally self contained. The first line of each band therefore is based on a 'previous' blank line or the last line of the previous band.

8.1.2.3.1 Group 3 and 4 facsimile compression

The Group 3 Facsimile compression algorithm [22] losslessly compresses bi-level data for transmission over slow and noisy telephone lines. The bi-level data represents scanned black text and graphics on a white background, and the algorithm is tuned for this class of images (it is explicitly not tuned, for example, for *halftoned* bi-level images). The *1D* Group 3 algorithm

runlength-encodes each scanline and then Huffman-encodes the resulting runlengths. Runlengths in the range 0 to 63 are coded with *terminating* codes. Runlengths in the range 64 to 2623 are coded with *make-up* codes, each representing a multiple of 64, followed by a terminating code. Runlengths exceeding 2623 are coded with multiple make-up codes followed by a terminating code. The Huffman tables are fixed, but are separately tuned for black and white runs (except for make-up codes above 1728, which are common). When possible, the *2D* Group 3 algorithm encodes a scanline as a set of short edge deltas (0, ±1, ±2, ±3) with reference to the previous scanline. The delta symbols are entropy-encoded (so that the zero delta symbol is only one bit long etc.) Edges within a 2D-encoded line which can't be delta-encoded are runlength-encoded, and are identified by a prefix. 1D- and 2D-encoded lines are marked differently. 1D-encoded lines are generated at regular intervals, whether actually required or not, to ensure that the decoder can recover from line noise with minimal image degradation. 2D Group 3 achieves compression ratios of up to 6:1 [32].

The Group 4 Facsimile algorithm [22] losslessly compresses bi-level data for transmission over *error-free* communications lines (i.e. the lines are truly error-free, or error-correction is done at a lower protocol level). The Group 4 algorithm is based on the 2D Group 3 algorithm, with the essential modification that since transmission is assumed to be error-free, 1D-encoded lines are no longer generated at regular intervals as an aid to error-recovery. Group 4 achieves compression ratios ranging from 20:1 to 60:1 for the CCITT set of test images [32].

The design goals and performance of the Group 4 compression algorithm qualify it as a compression algorithm for the bi-level layers. However, its Huffman tables are tuned to a lower scanning resolution (100-400 dpi), and it encodes runlengths exceeding 2623 awkwardly.

8.1.2.4 Contone data compression

The contone layer (CMYK) is either a non-compressed bytestream or is compressed to an interleaved JPEG bytestream. The JPEG bytestream is complete and self-contained. It contains all data required for decompression, including quantization and Huffman tables.

The contone data is optionally converted to YCrCb before being compressed (there is no specific advantage in color-space converting if not compressing). Additionally, the CMY contone pixels are optionally converted (on an individual basis) to RGB before color conversion using R=255-C, G=255-M, B=255-Y. Optional bitwise inversion of the K plane may also be performed. Note that this CMY to RGB conversion is not intended to be accurate for display purposes, but rather for the purposes of later converting to YCrCb. The inverse transform will be applied before printing.

8.1.2.4.1 JPEG compression

The JPEG compression algorithm [27] lossily compresses a contone image at a specified quality level. It introduces imperceptible image degradation at compression ratios below 5:1, and negligible image degradation at compression ratios below 10:1 [33].

JPEG typically first transforms the image into a color space which separates luminance and chrominance into separate color channels. This allows the chrominance channels to be subsampled without appreciable loss because of the human visual system's relatively greater sensitivity to luminance than chrominance. After this first step, each color channel is compressed separately.

The image is divided into 8×8 pixel blocks. Each block is then transformed into the frequency domain via a discrete cosine transform (DCT). This transformation has the effect of concentrating image energy in relatively lower-frequency coefficients, which allows higher-frequency coefficients to be more crudely quantized. This quantization is the principal source of compression in JPEG. Further compression is achieved by ordering coefficients by frequency to maximize the likelihood of adjacent zero coefficients, and then runlength-encoding runs of zeroes. Finally, the runlengths and non-zero frequency coefficients are entropy coded. Decompression is the inverse process of compression.

8.1.2.4.2 Non-compressed format

If the contone data is non-compressed, it must be in a block-based format bytestream with the same pixel order as would be produced by a JPEG decoder. The bytestream therefore consists of a series of 8×8 block of the original image, starting with the top left 8×8 block, and working horizontally across the page (as it will be printed) until the top rightmost 8×8 block, then the next row of 8×8 blocks (left to right) and so on until the lower row of 8×8 blocks (left to right). Each 8×8 block consists of 64 8-bit pixels for color plane 0 (representing 8 rows of 8 pixels in the order top left to bottom right) followed by 64 8-bit pixels for color plane 1 and so on for up to a maximum of 4 color planes.

If the original image is not a multiple of 8 pixels in X or Y, padding must be present (the extra pixel data will be ignored by the setting of margins).

8.1.2.4.3 Compressed format

If the contone data is compressed the first memory band contains JPEG headers (including tables) plus MCUs (minimum coded units). The ratio of space between the various color planes in the JPEG stream is 1:1:1:1. No subsampling is permitted. Banding can be completely arbitrary i.e there can be multiple JPEG images per band or 1 JPEG image divided over multiple bands. The break between bands is only memory alignment based.

8.1.2.4.4 Conversion of RGB to YCrCb (in RIP)

YCrCb is defined as per CCIR 601-1 [24] except that Y, Cr and Cb are normalized to occupy all 256 levels of an 8-bit binary encoding and take account of the actual hardware implementation of the inverse transform within SoPEC.

The exact color conversion computation is as follows:

- Y* = (9805/32768)R + (19235/32768)G + (3728/32768)B
- Cr* = (16375/32768)R - (13716/32768)G - (2659/32768)B + 128
- Cb* = -(5529/32768)R - (10846/32768)G + (16375/32768)B + 128

Y, Cr and Cb are obtained by rounding to the nearest integer. There is no need for saturation since ranges of Y*, Cr* and Cb* after rounding are [0-255], [1-255] and [1-255] respectively. *Note that full accuracy is possible with 24 bits*. See [14] for more information.

SoPEC ASIC

9 Overview

The Small Office Home Office Print Engine Controller (SoPEC) is a page rendering engine ASIC that takes compressed page images as input, and produces decompressed page images at up to 6

channels of bi-level dot data as output. The bi-level dot data is generated for the Memjet bi-lithic printhead. The dot generation process takes account of printhead construction, dead nozzles, and allows for fixative generation.

A single SoPEC can control 2 bi-lithic printheads and up to 6 color channels at 10,000 lines/sec[5], equating to 30 pages per minute. A single SoPEC can perform full-bleed printing of A3, A4 and Letter pages. The 6 channels of colored ink are the expected maximum in a consumer SOHO, or office Bi-lithic printing environment:

- CMY, for regular color printing.
- K, for black text, line graphics and gray-scale printing.
- IR (infrared), for Netpage-enabled [5] applications.
- F (fixative), to enable printing at high speed. Because the bi-lithic printer is capable of printing so fast, a fixative may be required to enable the ink to dry before the page touches the page already printed. Otherwise the pages may bleed on each other. In low speed printing environments the fixative may not be required.

SoPEC is *color space agnostic*. Although it can accept contone data as CMYX or RGBX, where X is an optional 4th channel, it also can accept contone data in any print color space. Additionally, SoPEC provides a mechanism for arbitrary mapping of input channels to output channels, including combining dots for ink optimization, generation of channels based on any number of other channels etc. However, inputs are typically CMYK for contone input, K for the bi-level input, and the optional Netpage tag dots are typically rendered to an infra-red layer. A fixative channel is typically generated for fast printing applications.

SoPEC is *resolution agnostic*. It merely provides a mapping between input resolutions and output resolutions by means of scale factors. The expected output resolution is 1600 dpi, but SoPEC actually has no knowledge of the physical resolution of the Bi-lithic printhead.

SoPEC is *page-length agnostic*. Successive pages are typically split into bands and downloaded into the page store as each band of information is consumed and becomes free.

SoPEC provides an interface for synchronization with other SoPECs. This allows simple multi-SoPEC solutions for simultaneous A3/A4/Letter duplex printing. However, SoPEC is also capable of printing only a portion of a page image. Combining synchronization functionality with partial page rendering allows multiple SoPECs to be readily combined for alternative printing requirements including simultaneous duplex printing and wide format printing.

Table 8 lists some of the features and corresponding benefits of SoPEC.

Table 8. Features and Benefits of SoPEC

Feature	Benefits
Optimised print architecture in hardware	30ppm full page photographic quality color printing from a desktop PC

[5]10,000 lines per second equates to 30 A4/Letter pages per minute at 1600 dpi

0.13micron CMOS (>3 million transistors)	High speed Low cost High functionality
900 Million dots per second	Extremely fast page generation
10,000 lines per second at 1600 dpi	0.5 A4/Letter pages per SoPEC chip per second
1 chip drives up to 133,920 nozzles	Low cost page-width printers
1 chip drives up to 6 color planes	99% of SoHo printers can use 1 SoPEC device
Integrated DRAM	No external memory required, leading to low cost systems
Power saving sleep mode	SoPEC can enter a power saving sleep mode to reduce power dissipation between print jobs
JPEG expansion	Low bandwidth from PC Low memory requirements in printer
Lossless bitplane expansion	High resolution text and line art with low bandwidth from PC (e.g. over USB)
Netpage tag expansion	Generates interactive paper
Stochastic dispersed dot dither	Optically smooth image quality No moire effects
Hardware compositor for 6 image planes	Pages composited in real-time
Dead nozzle compensation	Extends printhead life and yield Reduces printhead cost
Color space agnostic	Compatible with all inksets and image sources including RGB, CMYK, spot, CIE L*a*b*, hexachrome, YCrCbK, sRGB and other
Color space conversion	Higher quality / lower bandwidth
Computer interface	USB1.1 interface to host and ISI interface to ISI-Bridge chip thereby allowing connection to IEEE 1394, Bluetooth etc.
Cascadable in resolution	Printers of any resolution
Cascadable in color depth	Special color sets e.g. hexachrome can be used
Cascadable in image size	Printers of any width up to 16 inches
Cascadable in pages	Printers can print both sides simultaneously
Cascadable in speed	Higher speeds are possible by having each SoPEC print one vertical strip of the page.
Fixative channel data generation	Extremely fast ink drying without wastage
Built-in security	Revenue models are protected
Undercolor removal on dot-by-dot basis	Reduced ink usage

Does not require fonts for high speed operation	No font substitution or missing fonts
Flexible printhead configuration	Many configurations of printheads are supported by one chip type
Drives Bi-lithic printheads directly	No print driver chips required, results in lower cost
Determines dot accurate ink usage	Removes need for physical ink monitoring system in ink cartridges

9.1 PRINTING RATES

The required printing rate for SoPEC is 30 sheets per minute with an inter-sheet spacing of 4 cm. To achieve a 30 sheets per minute print rate, this requires:

300mm x 63 (dot/mm) / 2 sec = 105.8 μseconds per line, with no inter-sheet gap.

340mm x 63 (dot/mm) / 2 sec = 93.3 μseconds per line, with a 4 cm inter-sheet gap.

A printline for an A4 page consists of 13824 nozzles across the page [2]. At a system clock rate of 160 MHz 13824 dots of data can be generated in 86.4 μseconds. Therefore data can be generated fast enough to meet the printing speed requirement. It is necessary to deliver this print data to the print-heads.

Printheads can be made up of 5:5, 6:4, 7:3 and 8:2 inch printhead combinations [2]. Print data is transferred to both print heads in a pair simultaneously. This means the longest time to print a line is determined by the time to transfer print data to the longest print segment. There are 9744 nozzles across a 7 inch printhead. The print data is transferred to the printhead at a rate of 106 MHz (2/3 of the system clock rate) per color plane. This means that it will take 91.9 μs to transfer a single line for a 7:3 printhead configuration. So we can meet the requirement of 30 sheets per minute printing with a 4 cm gap with a 7:3 printhead combination. There are 11160 across an 8 inch printhead. To transfer the data to the printhead at 106 MHz will take 105.3 μs. So an 8:2 printhead combination printing with an inter-sheet gap will print slower than 30 sheets per minute.

9.2 SoPEC BASIC ARCHITECTURE

From the highest point of view the SoPEC device consists of 3 distinct subsystems

- CPU Subsystem
- DRAM Subsystem
- Print Engine Pipeline (PEP) Subsystem

See Figure 13 for a block level diagram of SoPEC.

9.2.1 CPU Subsystem

The CPU subsystem controls and configures all aspects of the other subsystems. It provides general support for interfacing and synchronising the external printer with the internal print engine. It also controls the low speed communication to the QA chips. The CPU subsystem contains various peripherals to aid the CPU, such as GPIO (includes motor control), interrupt controller, LSS Master and general timers. The Serial Communications Block (SCB) on the CPU subsystem provides a full speed USB1.1 interface to the host as well as an Inter SoPEC Interface (ISI) to other SoPEC devices.

9.2.2 DRAM Subsystem

The DRAM subsystem accepts requests from the CPU, Serial Communications Block (SCB) and blocks within the PEP subsystem. The DRAM subsystem (in particular the DIU) arbitrates the various requests and determines which request should win access to the DRAM. The DIU arbitrates based on configured parameters, to allow sufficient access to DRAM for all requestors. The DIU also hides the implementation specifics of the DRAM such as page size, number of banks, refresh rates etc.

9.2.3 Print Engine Pipeline (PEP) subsystem

The Print Engine Pipeline (PEP) subsystem accepts compressed pages from DRAM and renders them to bi-level dots for a given print line destined for a printhead interface that communicates directly with up to 2 segments of a bi-lithic printhead.

The first stage of the page expansion pipeline is the CDU, LBD and TE. The CDU expands the JPEG-compressed contone (typically CMYK) layer, the LBD expands the compressed bi-level layer (typically K), and the TE encodes Netpage tags for later rendering (typically in IR or K ink). The output from the first stage is a set of buffers: the CFU, SFU, and TFU. The CFU and SFU buffers are implemented in DRAM.

The second stage is the HCU, which dithers the contone layer, and composites position tags and the bi-level spot0 layer over the resulting bi-level dithered layer. A number of options exist for the way in which compositing occurs. Up to 6 channels of bi-level data are produced from this stage. Note that not all 6 channels may be present on the printhead. For example, the printhead may be CMY only, with K pushed into the CMY channels and IR ignored. Alternatively, the position tags may be printed in K if IR ink is not available (or for testing purposes).

The third stage (DNC) compensates for dead nozzles in the printhead by color redundancy and error diffusing dead nozzle data into surrounding dots.

The resultant bi-level 6 channel dot-data (typically CMYK-IRF) is buffered and written out to a set of line buffers stored in DRAM via the DWU.

Finally, the dot-data is loaded back from DRAM, and passed to the printhead interface via a dot FIFO. The dot FIFO accepts data from the LLU at the system clock rate (*pclk*), while the PHI removes data from the FIFO and sends it to the printhead at a rate of 2/3 times the system clock rate (see Section 9.1).

9.3 SoPEC Block Description

Looking at Figure 13, the various units are described here in summary form:

Table 9. Units within SoPEC

Subsystem	Unit Acronym	Unit Name	Description
DRAM	DIU	DRAM interface unit	Provides the interface for DRAM read and write access for the various SoPEC units, CPU and the SCB block. The DIU provides arbitration between competing units controls DRAM

			access.
	DRAM	Embedded DRAM	20Mbits of embedded DRAM,
CPU	CPU	Central Processing Unit	CPU for system configuration and control
	MMU	Memory Management Unit	Limits access to certain memory address areas in CPU user mode
	RDU	Real-time Debug Unit	Facilitates the observation of the contents of most of the CPU addressable registers in SoPEC in addition to some pseudo-registers in realtime.
	TIM	General Timer	Contains watchdog and general system timers
	LSS	Low Speed Serial Interfaces	Low level controller for interfacing with the QA chips
	GPIO	General Purpose IOs	General IO controller, with built-in Motor control unit, LED pulse units and de-glitch circuitry
	ROM	Boot ROM	16 KBytes of System Boot ROM code
	ICU	Interrupt Controller Unit	General Purpose interrupt controller with configurable priority, and masking.
	CPR	Clock, Power and Reset block	Central Unit for controlling and generating the system clocks and resets and powerdown mechanisms
	PSS	Power Save Storage	Storage retained while system is powered down
	USB	Universal Serial Bus Device	USB device controller for interfacing with the host USB.
	ISI	Inter-SoPEC Interface	ISI controller for data and control communication with other SoPEC's in a multi-SoPEC system
	SCB	Serial Communication Block	Contains both the USB and ISI blocks.
Print Engine Pipeline (PEP)	PCU	PEP controller	Provides external CPU with the means to read and write PEP Unit registers, and read and write DRAM in single 32-bit chunks.
	CDU	Contone decoder unit	Expands JPEG compressed contone layer and writes decompressed contone to DRAM
	CFU	Contone FIFO Unit	Provides line buffering between CDU and HCU
	LBD	Lossless Bi-level Decoder	Expands compressed bi-level layer.
	SFU	Spot FIFO Unit	Provides line buffering between LBD and HCU
	TE	Tag encoder	Encodes tag data into line of tag dots.

TFU	Tag FIFO Unit	Provides tag data storage between TE and HCU
HCU	Halftoner compositor unit	Dithers contone layer and composites the bi-level spot 0 and position tag dots.
DNC	Dead Nozzle Compensator	Compensates for dead nozzles by color redundancy and error diffusing dead nozzle data into surrounding dots.
DWU	Dotline Writer Unit	Writes out the 6 channels of dot data for a given printline to the line store DRAM
LLU	Line Loader Unit	Reads the expanded page image from line store, formatting the data appropriately for the bi-lithic printhead.
PHI	PrintHead Interface	Is responsible for sending dot data to the bi-lithic printheads and for providing line synchronization between multiple SoPECs. Also provides test interface to printhead such as temperature monitoring and Dead Nozzle Identification.

9.4 ADDRESSING SCHEME IN SOPEC

SoPEC must address

- 20 Mbit DRAM.
- PCU addressed registers in PEP.
- CPU-subsystem addressed registers.

SoPEC has a unified address space with the CPU capable of addressing all CPU-subsystem and PCU-bus accessible registers (in PEP) and all locations in DRAM. The CPU generates byte-aligned addresses for the whole of SoPEC.

22 bits are sufficient to byte address the whole SoPEC address space.

9.4.1 DRAM addressing scheme

The embedded DRAM is composed of 256-bit words. However the CPU-subsystem may need to write individual bytes of DRAM. Therefore it was decided to make the DIU byte addressable. 22 bits are required to byte address 20 Mbits of DRAM.

Most blocks read or write 256-bit words of DRAM. Therefore only the top 17 bits i.e. bits 21 to 5 are required to address 256-bit word aligned locations.

The exceptions are

- CDU which can write 64-bits so only the top 19 address bits i.e. bits 21-3 are required.
- The CPU-subsystem always generates a 22-bit byte-aligned DIU address but it will send flags to the DIU indicating whether it is an 8, 16 or 32-bit write.

All DIU accesses must be within the same 256-bit aligned DRAM word.

9.4.2 PEP Unit DRAM addressing

PEP Unit configuration registers which specify DRAM locations should specify 256-bit aligned DRAM addresses i.e. using address bits 21:5. Legacy blocks from PEC1 e.g. the LBD and TE may need to specify 64-bit aligned DRAM addresses if these reused blocks DRAM addressing is difficult to modify. These 64-bit aligned addresses require address bits 21:3. However, these 64-bit aligned addresses should be programmed to start at a 256-bit DRAM word boundary.

Unlike PEC1, there are no constraints in SoPEC on data organization in DRAM except that all data structures must start on a 256-bit DRAM boundary. If data stored is not a multiple of 256-bits then the last word should be padded.

9.4.3 CPU subsystem bus addressed registers

The CPU subsystem bus supports 32-bit word aligned read and write accesses with variable access timings. See section 11.4 for more details of the access protocol used on this bus. The CPU subsystem bus does not currently support byte reads and writes but this can be added at a later date if required by imported IP.

9.4.4 PCU addressed registers in PEP

The PCU only supports 32-bit register reads and writes for the PEP blocks. As the PEP blocks only occupy a subsection of the overall address map and the PCU is explicitly selected by the MMU when a PEP block is being accessed the PCU does not need to perform a decode of the higher-order address bits. See Table 11 for the PEP subsystem address map.

9.5 SoPEC Memory Map

9.5.1 Main memory map

The system wide memory map is shown in Figure 14 below. The memory map is discussed in detail in Section 11 11Central Processing Unit (CPU).

9.5.2 CPU-bus peripherals address map

The address mapping for the peripherals attached to the CPU-bus is shown in Table 10 below. The MMU performs the decode of *cpu_adr[21:12]* to generate the relevant *cpu_block_select* signal for each block. The addressed blocks decode however many of the lower order bits of *cpu_adr[11:2]* are required to address all the registers within the block.

Table 10. CPU-bus peripherals address map

Block_base	Address
ROM_base	0x0000_0000
MMU_base	0x0001_0000
TIM_base	0x0001_1000
LSS_base	0x0001_2000
GPIO_base	0x0001_3000
SCB_base	0x0001_4000
ICU_base	0x0001_5000
CPR_base	0x0001_6000
DIU_base	0x0001_7000

PSS_base	0x0001_8000
Reserved	0x0001_9000 to 0x0001_FFFF
PCU_base	0x0002_0000 to 0x0002_BFFF

9.5.3 PCU Mapped Registers (PEP blocks) address map

The PEP blocks are addressed via the PCU. From Figure 14, the PCU mapped registers are in the range 0x0002_0000 to 0x0002_BFFF. From Table 11 it can be seen that there are 12 sub-blocks within the PCU address space. Therefore, only four bits are necessary to address each of the sub-blocks within the PEP part of SoPEC. A further 12 bits may be used to address any configurable register within a PEP block. This gives scope for 1024 configurable registers per sub-block (the PCU mapped registers are all 32-bit addressed registers so the upper 10 bits are required to individually address them). This address will come either from the CPU or from a command stored in DRAM. The bus is assembled as follows:

- address[15:12] = sub-block address,
  - address[n:2] = register address within sub-block, only the number of bits required to decode the registers within each sub-block are used,
  - address[1:0] = byte address, unused as PCU mapped registers are all 32-bit addressed registers.

So for the case of the HCU, its addresses range from 0x7000 to 0x7FFF within the PEP subsystem or from 0x0002_7000 to 0x0002_7FFF in the overall system.

Table 11. PEP blocks address map

Block_base	Address
PCU_base	0x0002_0000
CDU_base	0x0002_1000
CFU_base	0x0002_2000
LBD_base	0x0002_3000
SFU_base	0x0002_4000
TE_base	0x0002_5000
TFU_base	0x0002_6000
HCU_base	0x0002_7000
DNC_base	0x0002_8000
DWU_base	0x0002_9000
LLU_base	0x0002_A000
PHI_base	0x0002_B000 to 0x0002_BFFF

9.6 BUFFER MANAGEMENT IN SOPEC

As outlined in Section 9.1, SoPEC has a requirement to print 1 side every 2 seconds i.e. 30 sides per minute.

9.6.1 Page buffering

Approximately 2 Mbytes of DRAM are reserved for compressed page buffering in SoPEC. If a page is compressed to fit within 2 Mbyte then a complete page can be transferred to DRAM before printing. However, the time to transfer 2 Mbyte using USB 1.1 is approximately 2 seconds. The worst case cycle time to print a page then approaches 4 seconds. This reduces the worst-case print speed to 15 pages per minute.

9.6.2 Band buffering

The SoPEC page-expansion blocks support the notion of page banding. The page can be divided into bands and another band can be sent down to SoPEC while we are printing the current band. Therefore we can start printing once at least one band has been downloaded.

The band size granularity should be carefully chosen to allow efficient use of the USB bandwidth and DRAM buffer space. It should be small enough to allow seamless 30 sides per minute printing but not so small as to introduce excessive CPU overhead in orchestrating the data transfer and parsing the band headers. Band-finish interrupts have been provided to notify the CPU of free buffer space. It is likely that the host PC will supervise the band transfer and buffer management instead of the SoPEC CPU.

If SoPEC starts printing before the complete page has been transferred to memory there is a risk of a buffer underrun occurring if subsequent bands are not transferred to SoPEC in time e.g. due to insufficient USB bandwidth caused by another USB peripheral consuming USB bandwidth. A buffer underrun occurs if a line synchronisation pulse is received before a line of data has been transferred to the printhead and causes the print job to fail at that line. If there is no risk of buffer underrun then printing can safely start once at least one band has been downloaded.

If there is a risk of a buffer underrun occurring due to an interruption of compressed page data transfer, then the safest approach is to only start printing once we have loaded up the data for a complete page. This means that a worst case latency in the region of 2 seconds (with USB1.1) will be incurred before printing the first page. Subsequent pages will take 2 seconds to print giving us the required sustained printing rate of 30 sides per minute.

A Storage SoPEC (Section 7.2.5) could be added to the system to provide guaranteed bandwidth data delivery. The print system could also be constructed using an ISI-Bridge chip (Section 7.2.6) to provide guaranteed data delivery.

The most efficient page banding strategy is likely to be determined on a per page/ print job basis and so SoPEC will support the use of bands of any size.

10 SoPEC Use Cases

10.1 INTRODUCTION

This chapter is intended to give an overview of a representative set of scenarios or *use cases* which SoPEC can perform. SoPEC is by no means restricted to the particular use cases described and not every SoPEC system is considered here.

In this chapter we discuss SoPEC use cases under four headings:

1) Normal operation use cases.
2) Security use cases.
3) Miscellaneous use cases.

4) Failure mode use cases.

Use cases for both single and multi-SoPEC systems are outlined.

Some tasks may be composed of a number of sub-tasks.

The realtime requirements for SoPEC software tasks are discussed in " 11 Central Processing Unit (CPU)" under Section 11.3 Realtime requirements.

10.2 NORMAL OPERATION IN A SINGLE SOPEC SYSTEM WITH USB HOST CONNECTION

SoPEC operation is broken up into a number of sections which are outlined below. Buffer management in a SoPEC system is normally performed by the host.

10.2.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

A typical powerup sequence is:

1) Execute reset sequence for complete SoPEC.
2) CPU boot from ROM.
3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation. USB Wakeup.
4) Download and authentication of program (see Section 10.5.2).
5) Execution of program from DRAM.
6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
7) Download and authenticate any further *datasets*.

10.2.2 USB wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block (chapter 16). Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. In a single SoPEC system, wakeup can be initiated following a USB reset from the SCB.

A typical USB wakeup sequence is:

1) Execute reset sequence for sections of SoPEC in sleep mode.
2) CPU boot from ROM, if CPU-subsystem was in sleep mode.
3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.
4) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.2).
5) Execution of program from DRAM.
6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
7) Download and authenticate using results in PSS of any further *datasets* (programs).

10.2.3 Print initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:

1) Check amount of ink remaining via QA chips.
2) Download static data e.g. dither matrices, dead nozzle tables from host to DRAM.

3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly.

4) Initiate printhead pre-heat sequence, if required.

10.2.4 First page download

Buffer management in a SoPEC system is normally performed by the host.

First page, first band download and processing:

1) The host communicates to the SoPEC CPU over the USB to check that DRAM space remaining is sufficient to download the first band.

2) The host downloads the first band (with the page header) to DRAM.

3) When the complete page header has been downloaded the SoPEC CPU processes the page header, calculates PEP register commands and writes directly to PEP registers or to DRAM.

4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Remaining bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.

2) Download the next band with the band header to DRAM.

3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

10.2.5 Start printing

1) Wait until at least one band of the first page has been downloaded.

One approach is to only start printing once we have loaded up the data for a complete page. If we start printing before the complete page has been transferred to memory we run the risk of a buffer underrun occurring because compressed page data was not transferred to SoPEC in time e.g. due to insufficient USB bandwidth caused by another USB peripheral consuming USB bandwidth.

2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes. A rapid startup order for the PEP units is outlined in Table 12.

Table 12. Typical PEP Unit startup order for printing a page.

Step#	Unit
1	DNC
2	DWU
3	HCU
4	PHI
5	LLU
6	CFU, SFU, TFU
7	CDU
8	TE, LBD

3) Print ready interrupt occurs (from PHI).

4) Start motor control, if first page, otherwise feed the next page. This step could occur before the print ready interrupt.
5) Drive LEDs, monitor paper status.
6) Wait for page alignment via page sensor(s) GPIO interrupt.
7) CPU instructs PHI to start producing line syncs and hence commence printing, or wait for an external device to produce line syncs.
8) Continue to download bands and process page and band headers for next page.

10.2.6 Next page(s) download

As for first page download, performed during printing of current page.

10.2.7 Between bands

When the finished band flags are asserted band related registers in the CDU, LBD, TE need to be re-programmed before the subsequent band can be printed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the CPU or most likely by updating from shadow registers.

The finished band flag interrupts the CPU to tell the CPU that the area of memory associated with the band is now free.

10.2.8 During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).
2) Decrement ink remaining (via QA chips).
3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.2.9 Page finish

These operations are typically performed when the page is finished:

1) Page finished interrupt occurs from PHI.
2) Shutdown the PEP blocks by de-asserting their Go registers. A typical shutdown order is defined in Table 13. This will set the PEP Unit state-machines to their idle states without resetting their configuration registers.
3) Communicate ink usage to QA chips, if required.

Table 13. End of page shutdown order for PEP Units.

Step#	Unit
1	PHI (will shutdown by itself in the normal case at the end of a page)
2	DWU (shutting this down stalls the DNC and therefore the HCU and above)
3	LLU (should already be halted due to PHI at end of last line of page)
4	TE (this is the only dot supplier likely to be running, halted by the HCU)
5	CDU (this is likely to already be halted due to end of contone band)

6	CFU, SFU, TFU, LBD (order unimportant, and should already be halted due to end of band)
7	HCU, DNC (order unimportant, should already have halted)

10.2.10 Start of next page

These operations are typically performed before printing the next page:

1) Re-program the PEP Units via PCU command processing from DRAM based on page header.
2) Go to Start printing.

10.2.11 End of document

1) Stop motor control.

10.2.12 Sleep mode

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block described in Section 16.

1) Instruct host PC via USB that SoPEC is about to sleep.
2) Store reusable authentication results in Power-Safe Storage (PSS).
3) Put SoPEC into defined sleep mode.

10.3 NORMAL OPERATION IN A MULTI-SOPEC SYSTEM - ISIMASTER SOPEC

In a multi-SoPEC system the host generally manages program and compressed page download to all the SoPECs. Inter-SoPEC communication is over the ISI link which will add a latency.

In the case of a multi-SoPEC system with just one USB 1.1 connection, the SoPEC with the USB connection is the ISIMaster. The ISI-bridge chip is the ISIMaster in the case of an ISI-Bridge SoPEC configuration. While it is perfectly possible for an ISISlave to have a direct USB connection to the host we do not treat this scenario explicitly here to avoid possible confusion.

In a multi-SoPEC system one of the SoPECs will be the PrintMaster. This SoPEC must manage and control sensors and actuators e.g. motor control. These sensors and actuators could be distributed over all the SoPECs in the system. An ISIMaster SoPEC may also be the PrintMaster SoPEC.

In a multi-SoPEC system each printing SoPEC will generally have its own PRINTER_QA chip (or at least access to a PRINTER_QA chip that contains the SoPEC's SOPEC_id_key) to validate operating parameters and ink usage. The results of these operations may be communicated to the PrintMaster SoPEC.

In general the ISIMaster may need to be able to:

- Send messages to the ISISlaves which will cause the ISISlaves to send their status to the ISIMaster.
- Instruct the ISISlaves to perform certain operations.

As the ISI is an insecure interface commands issued over the ISI are regarded as *user mode* commands. *Supervisor mode* code running on the SoPEC CPUs will allow or disallow these commands. The software protocol needs to be constructed with this in mind.

The ISIMaster will initiate all communication with the ISISlaves.

SoPEC operation is broken up into a number of sections which are outlined below.

10.3.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

1) Execute reset sequence for complete SoPEC.
2) CPU boot from ROM.
3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation USB Wakeup
4) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster (unless the SoPEC CPU has explicitly disabled this function).
5) Download and authentication of program (see Section 10.5.3).
6) Execution of program from DRAM.
7) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
8) Download and authenticate any further *datasets* (programs).
9) The initial *dataset* may be broadcast to all the ISISlaves.
10) ISIMaster master SoPEC then waits for a short time to allow the authentication to take place on the ISISlave SoPECs.
11) Each ISISlave SoPEC is polled for the result of its program code authentication process.
12) If all ISISlaves report successful authentication the OEM code module can be distributed and authenticated. OEM code will most likely reside on one SoPEC.

10.3.2 USB wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. For an ISIMaster SoPEC connected to the host via USB, wakeup can be initiated following a USB reset from the SCB.

A typical USB wakeup sequence is:

1) Execute reset sequence for sections of SoPEC in sleep mode.
2) CPU boot from ROM, if CPU-subsystem was in sleep mode.
3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.
4) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster (unless the SoPEC CPU has explicitly disabled this function).
5) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.3).
6) Execution of program from DRAM.
7) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
8) Download and authenticate any further *datasets* (programs) using results in Power-Safe Storage (PSS) (see Section 10.5.3).
9) Following steps as per Powerup.

10.3.3 Print initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:

1) Check amount of ink remaining via QA chips which may be present on a ISISlave SoPEC.

2) Download static data e.g. dither matrices, dead nozzle tables from host to DRAM.
3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly. Instruct ISISlaves to also perform this operation.
4) Initiate printhead pre-heat sequence, if required. Instruct ISISlaves to also perform this operation

10.3.4 First page download

Buffer management in a SoPEC system is normally performed by the host.

1) The host communicates to the SoPEC CPU over the USB to check that DRAM space remaining is sufficient to download the first band.
2) The host downloads the first band (with the page header) to DRAM.
3) When the complete page header has been downloaded the SoPEC CPU processes the page header, calculates PEP register commands and write directly to PEP registers or to DRAM.
4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Poll ISISlaves for DRAM status and download compressed data to ISISlaves.

Remaining first page bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.
2) Download the next band with the band header to DRAM.
3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

Poll ISISlaves for DRAM status and download compressed data to ISISlaves.

10.3.5 Start printing

1) Wait until at least one band of the first page has been downloaded.
2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes, in the suggested order defined in Table .
3) Print ready interrupt occurs (from PHI). Poll ISISlaves until print ready interrupt.
4) Start motor control (which may be on an ISISlave SoPEC), if first page, otherwise feed the next page. This step could occur before the print ready interrupt.
5) Drive LEDS, monitor paper status (which may be on an ISISlave SoPEC).
6) Wait for page alignment via page sensor(s) GPIO interrupt (which may be on an ISISlave SoPEC).
7) If the LineSyncMaster is a SoPEC its CPU instructs PHI to start producing master line syncs. Otherwise wait for an external device to produce line syncs.
8) Continue to download bands and process page and band headers for next page.

10.3.6 Next page(s) download

As for first page download, performed during printing of current page.

10.3.7 Between bands

When the finished band flags are asserted band related registers in the CDU, LBD and TE need to be re-programmed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the

CPU or by updating from shadow registers. The finished band flag interrupts to the CPU, tell the CPU that the area of memory associated with the band is now free.

10.3.8 During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).
2) Decrement ink remaining (via QA chips).
3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.3.9 Page finish

These operations are typically performed when the page is finished:

1) Page finished interrupt occurs from PHI. Poll ISISlaves for page finished interrupts.
2) Shutdown the PEP blocks by de-asserting their Go registers in the suggested order in Table . This will set the PEP Unit state-machines to their startup states.
3) Communicate ink usage to QA chips, if required.

10.3.10 Start of next page

These operations are typically performed before printing the next page:

1) Re-program the PEP Units via PCU command processing from DRAM based on page header.
2) Go to Start printing.

10.3.11 End of document

1) Stop motor control. This may be on an ISISlave SoPEC.

10.3.12 Sleep mode

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. This may be as a result of a command from the host or as a result of a timeout.

1) Inform host PC of which parts of SoPEC system are about to sleep.
2) Instruct ISISlaves to enter sleep mode.
3) Store reusable cryptographic results in Power-Safe Storage (PSS).
4) Put ISIMaster SoPEC into defined sleep mode.

10.4 NORMAL OPERATION IN A MULTI-SOPEC SYSTEM - ISISLAVE SOPEC

This section the outline typical operation of an ISISlave SoPEC in a multi-SoPEC system. The ISIMaster can be another SoPEC or an ISI-Bridge chip. The ISISlave communicates with the host either via the ISIMaster or using a direct connection such as USB. For this use case we consider only an ISISlave that does not have a direct host connection. Buffer management in a SoPEC system is normally performed by the host.

10.4.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

A typical powerup sequence is:

1) Execute reset sequence for complete SoPEC.
2) CPU boot from ROM.

3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation.
4) Download and authentication of program (see Section 10.5.3).
5) Execution of program from DRAM.
6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
7) SoPEC identification by sampling GPIO pins to determine ISIId. Communicate ISIId to ISIMaster.
8) Download and authenticate any further *datasets*.

10.4.2 ISI wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. In an ISISlave SoPEC, wakeup can be initiated following an ISI reset from the SCB.

A typical ISI wakeup sequence is:

1) Execute reset sequence for sections of SoPEC in sleep mode.
2) CPU boot from ROM, if CPU-subsystem was in sleep mode.
3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.
4) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.3).
5) Execution of program from DRAM.
6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
7) SoPEC identification by sampling GPIO pins to determine ISIId. Communicate ISIId to ISIMaster.
8) Download and authenticate any further *datasets*.

10.4.3 Print initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:

1) Check amount of ink remaining via QA chips.
2) Download static data e.g. dither matrices, dead nozzle tables from ISI to DRAM.
3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly.
4) Initiate printhead pre-heat sequence, if required.

10.4.4 First page download

Buffer management in a SoPEC system is normally performed by the host via the ISI.

1) Check DRAM space remaining is sufficient to download the first band.
2) The host downloads the first band (with the page header) to DRAM via the ISI.
3) When the complete page header has been downloaded, process the page header, calculate PEP register commands and write directly to PEP registers or to DRAM.
4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Remaining first page bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.
2) The host downloads the first band (with the page header) to DRAM via the ISI.
3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

10.4.5 Start printing

1) Wait until at least one band of the first page has been downloaded.
2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes, in the order defined in Table .
3) Print ready interrupt occurs (from PHI). Communicate to PrintMaster via ISI.
4) Start motor control, if attached to this ISISlave, when requested by PrintMaster, if first page, otherwise feed next page. This step could occur before the print ready interrupt
5) Drive LEDS, monitor paper status, if on this ISISlave SoPEC, when requested by PrintMaster
6) Wait for page alignment via page sensor(s) GPIO interrupt, if on this ISISlave SoPEC, and send to PrintMaster.
7) Wait for line sync and commence printing.
8) Continue to download bands and process page and band headers for next page.

10.4.6 Next page(s) download

As for first band download, performed during printing of current page.

10.4.7 Between bands

When the finished band flags are asserted band related registers in the CDU, LBD and TE need to be re-programmed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the CPU or by updating from shadow registers. The finished band flag interrupts to the CPU tell the CPU that the area of memory associated with the band is now free.

10.4.8 During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).
2) Decrement ink remaining (via QA chips).
3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.4.9 Page finish

These operations are typically performed when the page is finished:

1) Page finished interrupt occurs from PHI. Communicate page finished interrupt to PrintMaster.
2) Shutdown the PEP blocks by de-asserting their Go registers in the suggested order in Table . This will set the PEP Unit state-machines to their startup states.
3) Communicate ink usage to QA chips, if required.

10.4.10 Start of next page

These operations are typically performed before printing the next page:

1) Re-program the PEP Units via PCU command processing from DRAM based on page header.

2) Go to Start printing.

10.4.11 End of document

Stop motor control, if attached to this ISISlave, when requested by PrintMaster.

10.4.12 Powerdown

In this mode SoPEC is no longer powered.

1) Powerdown ISISlave SoPEC when instructed by ISIMaster.

10.4.13 Sleep

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. This may be as a result of a command from the host or ISIMaster or as a result of a timeout.

1) Store reusable cryptographic results in Power-Safe Storage (PSS).
2) Put SoPEC into defined sleep mode.

10.5 SECURITY USE CASES

Please see the 'SoPEC Security Overview' [9] document for a more complete description of SoPEC security issues. The SoPEC boot operation is described in the ROM chapter of the SoPEC hardware design specification, Section 17.2.

10.5.1 Communication with the QA chips

Communication between SoPEC and the QA chips (i.e. INK_QA and PRINTER_QA) will take place on at least a per power cycle and per page basis. Communication with the QA chips has three principal purposes: validating the presence of genuine QA chips (i.e the printer is using approved consumables), validation of the amount of ink remaining in the cartridge and authenticating the operating parameters for the printer. After each page has been printed, SoPEC is expected to communicate the number of dots fired per ink plane to the QA chipset. SoPEC may also initiate decoy communications with the QA chips from time to time.

Process:

- When validating ink consumption SoPEC is expected to principally act as a conduit between the PRINTER_QA and INK_QA chips and to take certain actions (basically enable or disable printing and report status to host PC) based on the result. The communication channels are insecure but all traffic is signed to guarantee authenticity.

Known Weaknesses

- All communication to the QA chips is over the LSS interfaces using a serial communication protocol. This is open to observation and so the communication protocol could be reverse engineered. In this case both the PRINTER_QA and INK_QA chips could be replaced by impostor devices (e.g. a single FPGA) that successfully emulated the communication protocol. As this would require physical modification of each printer this is considered to be an acceptably low risk. Any messages that are not signed by one of the symmetric keys (such as the SoPEC_id_key) could be reverse engineered. The imposter device must also have access to the appropriate keys to crack the system.
- If the secret keys in the QA chips are exposed or cracked then the system, or parts of it, is compromised.

Assumptions:

[1] The QA chips are not involved in the authentication of downloaded SoPEC code

[2] The QA chip in the ink cartridge (INK_QA) does not directly affect the operation of the cartridge in any way i.e. it does not inhibit the flow of ink etc.

[3] The INK_QA and PRINTER_QA chips are identical in their virgin state. They only become a INK_QA or PRINTER_QA after their FlashROM has been programmed.

10.5.2 Authentication of downloaded code in a single SoPEC system

Process:

1) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster (unless the SoPEC CPU has explicitly disabled this function).
2) The program is downloaded to the embedded DRAM.
3) The CPU calculates a SHA-1 hash digest of the downloaded program.
4) The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.
5) If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.
6) The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.
7) If the hash values do not match then the host PC is notified of the failure and the SoPEC will await a new program download.
8) If the hash values match then the CPU starts executing the downloaded program.
9) If, as is very likely, the downloaded program wishes to download subsequent programs (such as OEM code) it is responsible for ensuring the authenticity of everything it downloads. The downloaded program may contain public keys that are used to authenticate subsequent downloads, thus forming a hierarchy of authentication. The SoPEC ROM does not control these authentications - it is solely concerned with verifying that the first program downloaded has come from a trusted source.
10) At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.
11) The OEM code is expected to perform some simple 'turn on the lights' tasks after which the host PC is informed that the printer is ready to print and the *Start Printing* use case comes into play.

Known Weaknesses:

- If the Silverbrook private boot0key is exposed or cracked then the system is seriously compromised. A ROM mask change would be required to reprogram the boot0key.

10.5.3 Authentication of downloaded code in a multi-SoPEC system

10.5.3.1 ISIMaster SoPEC Process:

1) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster.
2) The SCB is configured to broadcast the data received from the host PC.
3) The program is downloaded to the embedded DRAM and broadcasted to all ISISlave SoPECs over the ISI.
4) The CPU calculates a SHA-1 hash digest of the downloaded program.
5) The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.
6) If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.
7) The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.
8) If the hash values do not match then the host PC is notified of the failure and the SoPEC will await a new program download.
9) If the hash values match then the CPU starts executing the downloaded program.
10) It is likely that the downloaded program will poll each ISISlave SoPEC for the result of its authentication process and to determine the number of slaves present and their ISIIds.
11) If any ISISlave SoPEC reports a failed authentication then the ISIMaster communicates this to the host PC and the SoPEC will await a new program download.
12) If all ISISlaves report successful authentication then the downloaded program is responsible for the downloading, authentication and distribution of subsequent programs within the multi-SoPEC system.
13) At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.
14) The OEM code is expected to perform some simple 'turn on the lights' tasks after which the master SoPEC determines that all SoPECs are ready to print. The host PC is informed that the printer is ready to print and the *Start Printing* use case comes into play.

10.5.3.2 ISISlave SoPEC Process:

1) When the CPU comes out of reset the SCB will be in slave mode, and the SCB is already configured to receive data from both the ISI and USB.
2) The program is downloaded (via ISI or USB) to embedded DRAM.
3) The CPU calculates a SHA-1 hash digest of the downloaded program.

4) The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.

5) If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.

6) The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.

7) If the hash values do not match, then the ISISlave device will await a new program again

8) If the hash values match then the CPU starts executing the downloaded program.

9) It is likely that the downloaded program will communicate the result of its authentication process to the ISIMaster. The downloaded program is responsible for determining the SoPECs ISIId, receiving and authenticating any subsequent programs.

10) At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.

11) The OEM code is expected to perform some simple 'turn on the lights' tasks after which the master SoPEC is informed that this slave is ready to print. The *Start Printing* use case then comes into play.

Known Weaknesses

- If the Silverbrook private boot0key is exposed or cracked then the system is seriously compromised.
- ISI is an open interface i.e. messages sent over the ISI are in the clear. The communication channels are insecure but all traffic is signed to guarantee authenticity. As all communication over the ISI is controlled by Supervisor code on both the ISIMaster and ISISlave then this also provides some protection against software attacks.

10.5.4 Authentication and upgrade of operating parameters for a printer

The SoPEC IC will be used in a range of printers with different capabilities (e.g. A3/A4 printing, printing speed, resolution etc.). It is expected that some printers will also have a software upgrade capability which would allow a user to purchase a license that enables an upgrade in their printer's capabilities (such as print speed). To facilitate this it must be possible to securely store the operating parameters in the PRINTER_QA chip, to securely communicate these parameters to the SoPEC and to securely reprogram the parameters in the event of an upgrade. Note that each printing SoPEC (as opposed to a SoPEC that is only used for the storage of data) will have its own PRINTER_QA chip (or at least access to a PRINTER_QA that contains the SoPEC's SoPEC_id_key). Therefore both ISIMaster and ISISlave SoPECs will need to authenticate operating parameters.

Process:

1) Program code is downloaded and authenticated as described in sections 10.5.2 and 10.5.3 above.

2) The program code has a function to create the SoPEC_id_key from the unique SoPEC_id that was programmed when the SoPEC was manufactured.

3) The SoPEC retrieves the signed operating parameters from its PRINTER_QA chip. The PRINTER_QA chip uses the SoPEC_id_key (which is stored as part of the pairing process executed during printhead assembly manufacture & test) to sign the operating parameters which are appended with a random number to thwart replay attacks.

4) The SoPEC checks the signature of the operating parameters using its SoPEC_id_key. If this signature authentication process is successful then the operating parameters are considered valid and the overall boot process continues. If not the error is reported to the host PC.

5) Operating parameters may also be set or upgraded using a second key, the *PrintEngineLicense_key*, which is stored on the PRINTER_QA and used to authenticate the change in operating parameters.

Known Weaknesses:

It may be possible to retrieve the unique SoPEC_id by placing the SoPEC in test mode and scanning it out. It is certainly possible to obtain it by reverse engineering the device. Either way the SoPEC_id (and by extension the SoPEC_id_key) so obtained is valid only for that specific SoPEC and so printers may only be compromised one at a time by parties with the appropriate specialised equipment. Furthermore even if the SoPEC_id is compromised, the other keys in the system, which protect the authentication of consumables and of program code, are unaffected.

10.6 Miscellaneous Use Cases

There are many miscellaneous use cases such as the following examples. Software running on the SoPEC CPU or host will decide on what actions to take in these scenarios.

10.6.1 Disconnect / Re-connect of QA chips.

1) Disconnect of a QA chip between documents or if ink runs out mid-document.

2) Re-connect of a QA chip once authenticated e.g. ink cartridge replacement should allow the system to resume and print the next document

10.6.2 Page arrives before print ready interrupt.

1) Engage clutch to stop paper until print ready interrupt occurs.

10.6.3 Dead-nozzle table upgrade

This sequence is typically performed when dead nozzle information needs to be updated by performing a printhead dead nozzle test.

1) Run printhead nozzle test sequence

2) Either host or SoPEC CPU converts dead nozzle information into dead nozzle table.

3) Store dead nozzle table on host.

4) Write dead nozzle table to SoPEC DRAM.

10.7 Failure Mode Use Cases

10.7.1 System errors and security violations

System errors and security violations are reported to the SoPEC CPU and host. Software running on the SoPEC CPU or host will then decide what actions to take.

Silverbrook code authentication failure.

1) Notify host PC of authentication failure.
2) Abort print run.

OEM code authentication failure.

1) Notify host PC of authentication failure.
2) Abort print run.

Invalid QA chip(s).

1) Report to host PC.
2) Abort print run.

MMU security violation interrupt.

1) This is handled by exception handler.
2) Report to host PC
3) Abort print run.

Invalid address interrupt from PCU.

1) This is handled by exception handler.
2) Report to host PC.
3) Abort print run.

Watchdog timer interrupt.

1) This is handled by exception handler.
2) Report to host PC.
3) Abort print run.

Host PC does not acknowledge message that SoPEC is about to power down.

1) Power down anyway.

10.7.2 Printing errors

Printing errors are reported to the SoPEC CPU and host. Software running on the host or SoPEC CPU will then decide what actions to take.

Insufficient space available in SoPEC compressed band-store to download a band.

1) Report to the host PC.

Insufficient ink to print.

1) Report to host PC.

Page not downloaded in time while printing.

1) Buffer underrun interrupt will occur.
2) Report to host PC and abort print run.

JPEG decoder error interrupt.

1) Report to host PC.

CPU SUBSYSTEM

11 Central Processing Unit (CPU)

11.1 OVERVIEW

The CPU block consists of the CPU core, MMU, cache and associated logic. The principal tasks for the program running on the CPU to fulfill in the system are:

Communications:

- Control the flow of data from the USB interface to the DRAM and ISI
- Communication with the host via USB or ISI
- Running the USB device driver

PEP Subsystem Control:

- Page and band header processing (may possibly be performed on host PC)
- Configure printing options on a per band, per page, per job or per power cycle basis
- Initiate page printing operation in the PEP subsystem
- Retrieve dead nozzle information from the printhead interface (PHI) and forward to the host PC
- Select the appropriate firing pulse profile from a set of predefined profiles based on the printhead characteristics
- Retrieve printhead temperature via the PHI

Security:

- Authenticate downloaded program code
- Authenticate printer operating parameters
- Authenticate consumables via the PRINTER_QA and INK_QA chips
- Monitor ink usage
- Isolation of OEM code from direct access to the system resources

Other:

- Drive the printer motors using the GPIO pins
- Monitoring the status of the printer (paper jam, tray empty etc.)
- Driving front panel LEDs
- Perform post-boot initialisation of the SoPEC device
- Memory management (likely to be in conjunction with the host PC)
- Miscellaneous housekeeping tasks

To control the Print Engine Pipeline the CPU is required to provide a level of performance at least equivalent to a 16-bit Hitachi H8-3664 microcontroller running at 16 MHz. An as yet undetermined amount of additional CPU performance is needed to perform the other tasks, as well as to provide the potential for such activity as Netpage page assembly and processing, RIPing etc. The extra performance required is dominated by the signature verification task and the SCB (including the USB) management task. An operating system is not required at present. A number of CPU cores have been evaluated and the LEON P1754 is considered to be the most appropriate solution. A diagram of the CPU block is shown in Figure 15 below.

11.2 DEFINITIONS OF I/OS

Table 14. CPU Subsystem I/Os

Port name	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
Pclk	1	In	Global clock
CPU to DIU DRAM interface			
cpu_adr[21:2]	20	Out	Address bus for both DRAM and peripheral access
cpu_dataout[31:0]	32	Out	Data out to both DRAM and peripheral devices. This should be driven at the same time as the *cpu_adr* and request signals.
dram_cpu_data[255:0]	256	In	Read data from the DRAM
cpu_diu_rreq	1	Out	Read request to the DIU DRAM
diu_cpu_rack	1	In	Acknowledge from DIU that read request has been accepted.
diu_cpu_rvalid	1	In	Signal from DIU telling SoPEC Unit that valid read data is on the *dram_cpu_data* bus
cpu_diu_wdatavalid	1	Out	Signal from the CPU to the DIU indicating that the data currently on the *cpu_diu_wdata* bus is valid and should be committed to the DIU posted write buffer
diu_cpu_write_rdy	1	In	Signal from the DIU indicating that the posted write buffer is empty
cpu_diu_wdadr[21:4]	18	Out	Write address bus to the DIU
cpu_diu_wdata[127:0]	128	Out	Write data bus to the DIU
cpu_diu_wmask[15:0]	16	Out	Write mask for the *cpu_diu_wdata* bus. Each bit corresponds to a byte of the 128-bit *cpu_diu_wdata* bus.
CPU to peripheral blocks			
cpu_rwn	1	Out	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	Out	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access
cpu_cpr_sel	1	Out	CPR block select.
cpr_cpu_rdy	1	In	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the CPR block and for a read cycle this means the data on *cpr_cpu_data* is valid.
cpr_cpu_berr	1	In	CPR bus error signal to the CPU.

cpr_cpu_data[31:0]	32	In	Read data bus from the CPR block
cpu_gpio_sel	1	Out	GPIO block select.
gpio_cpu_rdy	1	In	GPIO ready signal to the CPU.
gpio_cpu_berr	1	In	GPIO bus error signal to the CPU.
gpio_cpu_data[31:0]	32	In	Read data bus from the GPIO block
cpu_icu_sel	1	Out	ICU block select.
icu_cpu_rdy	1	In	ICU ready signal to the CPU.
icu_cpu_berr	1	In	ICU bus error signal to the CPU.
icu_cpu_data[31:0]	32	In	Read data bus from the ICU block
cpu_lss_sel	1	Out	LSS block select.
lss_cpu_rdy	1	In	LSS ready signal to the CPU.
lss_cpu_berr	1	In	LSS bus error signal to the CPU.
lss_cpu_data[31:0]	32	In	Read data bus from the LSS block
cpu_pcu_sel	1	Out	PCU block select.
pcu_cpu_rdy	1	In	PCU ready signal to the CPU.
pcu_cpu_berr	1	In	PCU bus error signal to the CPU.
pcu_cpu_data[31:0]	32	In	Read data bus from the PCU block
cpu_scb_sel	1	Out	SCB block select.
scb_cpu_rdy	1	In	SCB ready signal to the CPU.
scb_cpu_berr	1	In	SCB bus error signal to the CPU.
scb_cpu_data[31:0]	32	In	Read data bus from the SCB block
cpu_tim_sel	1	Out	Timers block select.
tim_cpu_rdy	1	In	Timers block ready signal to the CPU.
tim_cpu_berr	1	In	Timers bus error signal to the CPU.
tim_cpu_data[31:0]	32	In	Read data bus from the Timers block
cpu_rom_sel	1	Out	ROM block select.
rom_cpu_rdy	1	In	ROM block ready signal to the CPU.
rom_cpu_berr	1	In	ROM bus error signal to the CPU.
rom_cpu_data[31:0]	32	In	Read data bus from the ROM block
cpu_pss_sel	1	Out	PSS block select.
pss_cpu_rdy	1	In	PSS block ready signal to the CPU.
pss_cpu_berr	1	In	PSS bus error signal to the CPU.
pss_cpu_data[31:0]	32	In	Read data bus from the PSS block
cpu_diu_sel	1	Out	DIU register block select.
diu_cpu_rdy	1	In	DIU register block ready signal to the CPU.
diu_cpu_berr	1	In	DIU bus error signal to the CPU.
diu_cpu_data[31:0]	32	In	Read data bus from the DIU block
Interrupt signals			

icu_cpu_ilevel[3:0]	3	In	An interrupt is asserted by driving the appropriate priority level on *icu_cpu_ilevel*. These signals must remain asserted until the CPU executes an interrupt acknowledge cycle.
	3	Out	Indicates the level of the interrupt the CPU is acknowledging when *cpu_iack* is high
cpu_iack	1	Out	Interrupt acknowledge signal. The exact timing depends on the CPU core implementation
Debug signals			
diu_cpu_debug_valid	1	In	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.
tim_cpu_debug_valid	1	In	Signal indicating the data on the *tim_cpu_data* bus is valid debug data.
scb_cpu_debug_valid	1	In	Signal indicating the data on the *scb_cpu_data* bus is valid debug data.
pcu_cpu_debug_valid	1	In	Signal indicating the data on the *pcu_cpu_data* bus is valid debug data.
lss_cpu_debug_valid	1	In	Signal indicating the data on the *lss_cpu_data* bus is valid debug data.
icu_cpu_debug_valid	1	In	Signal indicating the data on the *icu_cpu_data* bus is valid debug data.
gpio_cpu_debug_valid	1	In	Signal indicating the data on the *gpio_cpu_data* bus is valid debug data.
cpr_cpu_debug_valid	1	In	Signal indicating the data on the *cpr_cpu_data* bus is valid debug data.
debug_data_out	32	Out	Output debug data to be muxed on to the GPIO & PHI pins
debug_data_valid	1	Out	Debug valid signal indicating the validity of the data on *debug_data_out*. This signal is used in all debug configurations
debug_cntrl	33	Out	Control signal for each PHI bound debug data line indicating whether or not the debug data should be selected by the pin mux

11.3 REALTIME REQUIREMENTS

The SoPEC realtime requirements have yet to be fully determined but they may be split into three categories: hard, firm and soft

11.3.1 Hard realtime requirements

Hard requirements are tasks that must be completed before a certain deadline or failure to do so will result in an error perceptible to the user (printing stops or functions incorrectly). There are three hard realtime tasks:

- Motor control: The motors which feed the paper through the printer at a constant speed during printing are driven directly by the SoPEC device. Four periodic signals with different phase relationships need to be generated to ensure the paper travels smoothly through the printer. The generation of these signals is handled by the GPIO hardware (see section 13.2 for more details) but the CPU is responsible for enabling these signals (i.e. to start or stop the motors) and coordinating the movement of the paper with the printing operation of the printhead.
- Buffer management: Data enters the SoPEC via the SCB at an uneven rate and is consumed by the PEP subsystem at a different rate. The CPU is responsible for managing the DRAM buffers to ensure that neither overrun nor underrun occur. This buffer management is likely to be performed under the direction of the host.
- Band processing: In certain cases PEP registers may need to be updated between bands. As the timing requirements are most likely too stringent to be met by direct CPU writes to the PCU a more likely scenario is that a set of shadow registers will programmed in the compressed page units before the current band is finished, copied to band related registers by the finished band signals and the processing of the next band will continue immediately. An alternative solution is that the CPU will construct a DRAM based set of commands (see section 21.8.5 for more details) that can be executed by the PCU. The task for the CPU here is to parse the band headers stored in DRAM and generate a DRAM based set of commands for the next number of bands. The location of the DRAM based set of commands must then be written to the PCU before the current band has been processed by the PEP subsystem. It is also conceivable (but currently considered unlikely) that the host PC could create the DRAM based commands. In this case the CPU will only be required to point the PCU to the correct location in DRAM to execute commands from.

11.3.2 Firm requirements

Firm requirements are tasks that should be completed by a certain time or failure to do so will result in a degradation of performance but not an error. The majority of the CPU tasks for SoPEC fall into this category including all interactions with the QA chips, program authentication, page feeding, configuring PEP registers for a page or job, determining the firing pulse profile, communication of printer status to the host over the USB and the monitoring of ink usage. The authentication of downloaded programs and messages will be the most compute intensive operation the CPU will be required to perform. Initial investigations indicate that the LEON processor, running at 160 MHz, will easily perform three authentications in under a second.

Table 15. Expected firm requirements

Requirement	Duration
Power-on to start of printing first page [USB and slave SoPEC	~ 8 secs ??

enumeration, 3 or more RSA signature verifications, code and compressed page data download and chip initialisation]	
Wake-up from sleep mode to start printing [3 or more SHA-1 / RSA operations, code and compressed page data download and chip re-initialisation	~ 2 secs
Authenticate ink usage in the printer	~ 0.5 secs
Determining firing pulse profile	~ 0.1 secs
Page feeding, gap between pages	OEM dependent
Communication of printer status to host PC	~ 10 ms
Configuring PEP registers	??

11.3.3 Soft requirements

Soft requirements are tasks that need to be done but there are only light time constraints on when they need to be done. These tasks are performed by the CPU when there are no pending higher priority tasks. As the SoPEC CPU is expected to be lightly loaded these tasks will mostly be executed soon after they are scheduled.

11.4 BUS PROTOCOLS

As can be seen from Figure 15 above there are different buses in the CPU block and different protocols are used for each bus. There are three buses in operation:

11.4.1 AHB bus

The LEON CPU core uses an AMBA2.0 AHB bus to communicate with memory and peripherals (usually via an APB bridge). See the AMBA specification [38], section 5 of the LEON users manual [37] and section 11.6.6.1 of this document for more details.

11.4.2 CPU to DIU bus

This bus conforms to the DIU bus protocol described in Section 20.14.8. Note that the address bus used for DIU reads (i.e. *cpu_adr(21:2)*) is also that used for CPU subsystem with bus accesses while the write address bus (*cpu_diu_wadr*) and the read and write data buses (*dram_cpu_data* and *cpu_diu_wdata*) are private buses between the CPU and the DIU. The effective bus width differs between a read (256 bits) and a write (128 bits). As certain CPU instructions may require byte write access this will need to be supported by both the DRAM write buffer (in the AHB bridge) and the DIU. See section 11.6.6.1 for more details.

11.4.3 CPU Subsystem Bus

For access to the on-chip peripherals a simple bus protocol is used. The MMU must first determine which particular block is being addressed (and that the access is a valid one) so that the appropriate block select signal can be generated. During a write access CPU write data is driven out with the address and block select signals in the first cycle of an access. The addressed slave peripheral responds by asserting its ready signal indicating that it has registered the write data and the access can complete. The write data bus is common to all peripherals and is also used for CPU writes to the embedded DRAM. A read access is initiated by driving the address and select signals during the first cycle of an access. The addressed slave responds by placing the read data on its

bus and asserting its ready signal to indicate to the CPU that the read data is valid. Each block has a separate point-to-point data bus for read accesses to avoid the need for a tri-stateable bus.

All peripheral accesses are 32-bit (Programming note: *char* or *short* C types should not be used to access peripheral registers). The use of the ready signal allows the accesses to be of variable length. In most cases accesses will complete in two cycles but three or four (or more) cycles accesses are likely for PEP blocks or IP blocks with a different native bus interface. All PEP blocks are accessed via the PCU which acts as a bridge. The PCU bus uses a similar protocol to the CPU subsystem bus but with the PCU as the bus master.

The duration of accesses to the PEP blocks is influenced by whether or not the PCU is executing commands from DRAM. As these commands are essentially register writes the CPU access will need to wait until the PCU bus becomes available when a register access has been completed. This could lead to the CPU being stalled for up to 4 cycles if it attempts to access PEP blocks while the PCU is executing a command. The size and probability of this penalty is sufficiently small to have any significant impact on performance.

In order to support user mode (i.e. OEM code) access to certain peripherals the CPU subsystem bus propagates the CPU function code signals (*cpu_acode[1:0]*). These signals indicate the type of address space (i.e. User/Supervisor and Program/Data) being accessed by the CPU for each access. Each peripheral must determine whether or not the CPU is in the correct mode to be granted access to its registers and in some cases (e.g. Timers and GPIO blocks) different access permissions can apply to different registers within the block. If the CPU is not in the correct mode then the violation is flagged by asserting the block's bus error signal (*block_cpu_berr*) with the same timing as its ready signal (*block_cpu_rdy*) which remains deasserted. When this occurs invalid read accesses should return 0 and write accesses should have no effect.

Figure 16 shows two examples of the peripheral bus protocol in action. A write to the LSS block from code running in supervisor mode is successfully completed. This is immediately followed by a read from a PEP block via the PCU from code running in user mode. As this type of access is not permitted the access is terminated with a bus error. The bus error exception processing then starts directly after this - no further accesses to the peripheral should be required as the exception handler should be located in the DRAM.

Each peripheral acts as a slave on the CPU subsystem bus and its behavior is described by the state machine in section 11.4.3.1

11.4.3.1 CPU subsystem bus slave state machine

CPU subsystem bus slave operation is described by the state machine in Figure 17.This state machine will be implemented in each CPU subsystem bus slave. The only new signals mentioned here are the *valid_access* and *reg_available* signals. The *valid_access* is determined by comparing the *cpu_acode* value with the block or register (in the case of a block that allow user access on a per register basis such as the GPIO block) access permissions and asserting *valid_access* if the permissions agree with the CPU mode. The *reg_available* signal is only required in the PCU or in blocks that are not capable of two-cycle access (e.g. blocks containing imported IP with different bus protocols). In these blocks the *reg_available* signal is an internal signal used to insert wait

states (by delaying the assertion of *block_cpu_rdy*) until the CPU bus slave interface can gain access to the register.

When reading from a register that is less than 32 bits wide the CPU subsystems bus slave should return zeroes on the unused upper bits of the *block_cpu_data* bus.

To support debug mode the contents of the register selected for debug observation, *debug_reg*, are always output on the *block_cpu_data* bus whenever a read access is not taking place. See section 11.8 for more details of debug operation.

11.5 LEON CPU

The LEON processor is an open-source implementation of the IEEE-1754 standard (SPARC V8) instruction set. LEON is available from and actively supported by Gaisler Research (www.gaisler.com).

The following features of the LEON-2 processor will be utilised on SoPEC:

- IEEE-1754 (SPARC V8) compatible integer unit with 5-stage pipeline
- Separate instruction and data cache (Harvard architecture). 1 kbyte direct mapped caches will be used for both.
- Full implementation of AMBA-2.0 AHB on-chip bus

The standard release of LEON incorporates a number of peripherals and support blocks which will not be included on SoPEC. The LEON core as used on SoPEC will consist of: 1) the LEON integer unit, 2) the instruction and data caches (currently 1kB each), 3) the cache control logic, 4) the AHB interface and 5) possibly the AHB controller (although this functionality may be implemented in the LEON AHB bridge).

The version of the LEON database that the SoPEC LEON components will be sourced from is LEON2-1.0.7 although later versions may be used if they offer worthwhile functionality or bug fixes that affect the SoPEC design.

The LEON core will be clocked using the system clock, *pclk*, and reset using the *prst_n_section[1]* signal. The ICU will assert all the hardware interrupts using the protocol described in section 11.9.

The LEON hardware multipliers and floating-point unit are not required. SoPEC will use the recommended 8 register window configuration.

Further details of the SPARC V8 instruction set and the LEON processor can be found in [36] and [37] respectively.

11.5.1 LEON Registers

Only two of the registers described in the LEON manual are implemented on SoPEC - the LEON configuration register and the Cache Control Register (CCR). The addresses of these registers are shown in Table 16. The configuration register bit fields are described below and the CCR is described in section 11.7.1.1.

11.5.1.1 LEON configuration register

The LEON configuration register allows runtime software to determine the settings of LEONs various configuration options. This is a read-only register whose value for the SoPEC ASIC will be

0x1071_8C00. Further descriptions of many of the bitfields can be found in the LEON manual. The values used for SoPEC are highlighted in bold for clarity.

Table 16. LEON Configuration Register

Field Name	bit(s)	Description
WriteProtection	1:0	Write protection type. 00 - none 01 - standard
PCICore	3:2	PCI core type 00 - none 01 - InSilicon 10 - ESA 11 - Other
FPUType	5:4	FPU type. 00 - none 01 - Meiko
MemStatus	6	0 - No memory status and failing address register present 1 - Memory status and failing address register present
Watchdog	7	0 - Watchdog timer not present (Note this refers to the LEON watchdog timer in the LEON timer block). 1 - Watchdog timer present
UMUL/SMUL	8	0 - UMUL/SMUL instructions are not implemented 1 - UMUL/SMUL instructions are implemented
UDIV/SDIV	9	0 - UMUL/SMUL instructions are not implemented 1 - UMUL/SMUL instructions are implemented
DLSZ	11:10	Data cache line size in 32-bit words: 00 - 1 word 01 - 2 words 10 - 4 words 11 - 8 words
DCSZ	14:12	Data cache size in kBbytes = 2^{DCSZ}. SoPEC DCSZ = 0.
ILSZ	16:15	Instruction cache line size in 32-bit words: 00 - 1 word 01 - 2 words 10 - 4 words 11 - 8 words
ICSZ	19:17	Instruction cache size in kBbytes = 2^{ICSZ}. SoPEC ICSZ = 0.
RegWin	24:20	The implemented number of SPARC register windows - 1. SoPEC value = 7.

UMAC/SMAC	25	0 - UMAC/SMAC instructions are not implemented 1 - UMAC/SMAC instructions are implemented
Watchpoints	28:26	The implemented number of hardware watchpoints. SoPEC value = 4.
SDRAM	29	0 - SDRAM controller not present 1 - SDRAM controller present
DSU	30	0 - Debug Support Unit not present 1 - Debug Support Unit present
Reserved	31	Reserved. SoPEC value = 0.

11.6 MEMORY MANAGEMENT UNIT (MMU)

Memory Management Units are typically used to protect certain regions of memory from invalid accesses, to perform address translation for a virtual memory system and to maintain memory page status (swapped-in, swapped-out or unmapped)

The SoPEC MMU is a much simpler affair whose function is to ensure that all regions of the SoPEC memory map are adequately protected. The MMU does not support virtual memory and physical addresses are used at all times. The SoPEC MMU supports a full 32-bit address space. The SoPEC memory map is depicted in Figure 18 below.

The MMU selects the relevant bus protocol and generates the appropriate control signals depending on the area of memory being accessed. The MMU is responsible for performing the address decode and generation of the appropriate block select signal as well as the selection of the correct block read bus during a read access. The MMU will need to support all of the bus transactions the CPU can produce including interrupt acknowledge cycles, aborted transactions etc.

When an MMU error occurs (such as an attempt to access a supervisor mode only region when in user mode) a bus error is generated. While the LEON can recognise different types of bus error (e.g. data store error, instruction access error) it handles them in the same manner as it handles all traps i.e it will transfer control to a trap handler. No extra state information is be stored because of the nature of the trap. The location of the trap handler is contained in the TBR (Trap Base Register). This is the same mechanism as is used to handle interrupts.

11.6.1 CPU-bus peripherals address map

The address mapping for the peripherals attached to the CPU-bus is shown in Table 17 below. The MMU performs the decode of the high order bits to generate the relevant *cpu_block_select* signal. Apart from the PCU, which decodes the address space for the PEP blocks, each block only needs to decode as many bits of *cpu_adr[11:2]* as required to address all the registers within the block.

Table 17. CPU-bus peripherals address map

Block_base	Address
ROM_base	0x0000_0000
MMU_base	0x0001_0000

TIM_base	0x0001_1000
LSS_base	0x0001_2000
GPIO_base	0x0001_3000
SCB_base	0x0001_4000
ICU_base	0x0001_5000
CPR_base	0x0001_6000
DIU_base	0x0001_7000
PSS_base	0x0001_8000
Reserved	0x0001_9000 to 0x0001_FFFF
PCU_base	0x0002_0000

11.6.2 DRAM Region Mapping

The embedded DRAM is broken into 8 regions, with each region defined by a lower and upper bound address and with its own access permissions.

The association of an area in the DRAM address space with a MMU region is completely under software control. Table 18 below gives one possible region mapping. Regions should be defined according to their access requirements and position in memory. Regions that share the same access requirements and that are contiguous in memory may be combined into a single region. The example below is purely for indicative purposes - real mappings are likely to differ significantly from this. Note that the RegionBottom and RegionTop fields in this example include the DRAM base address offset (0x4000_0000) which is not required when programming the *RegionNTop* and *RegionNBottom* registers. For more details, see 11.6.5.1 and 11.6.5.2.

Table 18. Example region mapping

Region	RegionBottom	RegionTop	Description
0	0x4000_0000	0x4000_0FFF	Silverbrook OS (supervisor) data
1	0x4000_1000	0x4000_BFFF	Silverbrook OS (supervisor) code
2	0x4000_C000	0x4000_C3FF	Silverbrook (supervisor/user) data
3	0x4000_C400	0x4000_CFFF	Silverbrook (supervisor/user) code
4	0x4026_D000	0x4026_D3FF	OEM (user) data
5	0x4026_D400	0x4026_DFFF	OEM (user) code
6	0x4027_E000	0x4027_FFFF	Shared Silverbrook/OEM space
7	0x4000_D000	0x4026_CFFF	Compressed page store (supervisor data)

11.6.3 Non-DRAM regions

As shown in Figure 18 the DRAM occupies only 2.5 MBytes of the total 4 GB SoPEC address space. The non-DRAM regions of SoPEC are handled by the MMU as follows:

ROM (0x0000_0000 to 0x0000_FFFF): The ROM block will control the access types allowed. The *cpu_acode[1:0]* signals will indicate the CPU mode and access type and the ROM block will assert *rom_cpu_berr* if an attempted access is forbidden. The protocol is described in more detail in

section 11.4.3. The ROM block access permissions are hard wired to allow all read accesses except to the *FuseChipID* registers which may only be read in supervisor mode.

MMU Internal Registers (0x0001_0000 to 0x0001_0FFF): The MMU is responsible for controlling the accesses to its own internal registers and will only allow data reads and writes (no instruction fetches) from supervisor data space. All other accesses will result in the *mmu_cpu_berr* signal being asserted in accordance with the CPU native bus protocol.

CPU Subsystem Peripheral Registers (0x0001_1000 to 0x0001_FFFF): Each peripheral block will control the access types allowed. Every peripheral will allow supervisor data accesses (both read and write) and some blocks (e.g. Timers and GPIO) will also allow user data space accesses as outlined in the relevant chapters of this specification. Neither supervisor nor user instruction fetch accesses are allowed to any block as it is not possible to execute code from peripheral registers. The bus protocol is described in section 11.4.3.

PCU Mapped Registers (0x0002_0000 to 0x0002_BFFF): All of the PEP blocks registers which are accessed by the CPU via the PCU will inherit the access permissions of the PCU. These access permissions are hard wired to allow supervisor data accesses only and the protocol used is the same as for the CPU peripherals.

Unused address space (0x0002_C000 to 0x3FFF_FFFF and 0x4028_0000 to 0xFFFF_FFFF): All accesses to the unused portion of the address space will result in the *mmu_cpu_berr* signal being asserted in accordance with the CPU native bus protocol. These accesses will not propagate outside of the MMU i.e. no external access will be initiated.

11.6.4 Reset exception vector and reference zero traps

When a reset occurs the LEON processor starts executing code from address 0x0000_0000. A common software bug is zero-referencing or null pointer de-referencing (where the program attempts to access the contents of address 0x0000_0000). To assist software debug the MMU will assert a bus error every time the locations 0x0000_0000 to 0x0000_000F (i.e. the first 4 words of the reset trap) are accessed after the reset trap handler has legitimately been retrieved immediately after reset.

11.6.5 MMU Configuration Registers

The MMU configuration registers include the RDU configuration registers and two LEON registers. Note that all the MMU configuration registers may only be accessed when the CPU is running in supervisor mode.

Table 19. MMU Configuration Registers

Address offset from MMU_base	Register	#bits	Reset	Description
0x00	Region0Bottom[21:5]	17	0x0_0000	This register contains the physical address that marks the bottom of region 0
0x04	Region0Top[21:5]	17	0xF_FFFF	This register contains the physical address that marks the top of region 0. Region 0 covers the

				entire address space after reset whereas all other regions are zero-sized initially.
0x08	Region1Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 1
0x0C	Region1Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 1
0x10	Region2Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 2
0x14	Region3Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 2
0x18	Region3Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 3
0x1C	Region3Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 3
0x20	Region4Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 4
0x24	Region4Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 4
0x28	Region5Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 5
0x2C	Region5Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 5
0x30	Region6Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 6
0x34	Region6Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 6
0x38	Region7Bottom[21:5]	17	0xF_FFFF	This register contains the physical address that marks the bottom of region 7
0x3C	Region7Top[21:5]	17	0x0_0000	This register contains the physical address that marks the top of region 7
0x40	Region0Control	6	0x07	Control register for region 0
0x44	Region1Control	6	0x07	Control register for region 1
0x48	Region2Control	6	0x07	Control register for region 2
0x4C	Region3Control	6	0x07	Control register for region 3
0x50	Region4Control	6	0x07	Control register for region 4
0x54	Region5Control	6	0x07	Control register for region 5
0x58	Region6Control	6	0x07	Control register for region 6
0x5C	Region7Control	6	0x07	Control register for region 7
0x60	RegionLock	8	0x00	Writing a 1 to a bit in the RegionLock register

				locks the value of the corresponding Region-Top, RegionBottom and RegionControl registers. The lock can only be cleared by a reset and any attempt to write to a locked register will result in a bus error.
0x64	BusTimeout	8	0xFF	This register should be set to the number of *pclk* cycles to wait after an access has started before aborting the access with a bus error. Writing 0 to this register disables the bus time-out feature.
0x68	ExceptionSource	6	0x00	This register identifies the source of the last exception. See Section 11.6.5.3 for details.
0x6C	DebugSelect	7	0x00	Contains address of the register selected for debug observation. It is expected that a number of pseudo-registers will be made available for debug observation and these will be outlined during the implementation phase.
0x80 to 0x108	RDU Registers			See Table for details.
0x140	LEON Configuration Register	32	0x1071_8 C00	The LEON configuration register is used by software to determine the configuration of this LEON implementation. See section 11.5.1.1 for details. This register is ReadOnly.
0x144	LEON Cache Control Register	32	0x0000_0 000	The LEON Cache Control Register is used to control the operation of the caches. See section 11.6 for details.

11.6.5.1 RegionTop and RegionBottom registers

The 20 Mbit of embedded DRAM on SoPEC is arranged as 81920 words of 256 bits each. All region boundaries need to align with a 256-bit word. Thus only 17 bits are required for the *RegionNTop* and *RegionNBottom* registers. Note that the bottom 5 bits of the *RegionNTop* and *RegionNBottom* registers cannot be written to and read as '0' i.e. the *RegionNTop* and *RegionNBottom* registers represent byte-aligned DRAM addresses

Both the *RegionNTop* and *RegionNBottom* registers are inclusive i.e. the addresses in the registers are included in the region. Thus the size of a region is (*RegionNTop* - *RegionNBottom*) +1 DRAM words.

If DRAM regions overlap (there is no reason for this to be the case but there is nothing to prohibit it either) then only accesses allowed by all overlapping regions are permitted. That is if a DRAM address appears in both Region1 and Region3 (for example) the *cpu_acode* of an access is

checked against the access permissions of both regions. If both regions permit the access then it will proceed but if either or both regions do not permit the access then it will not be allowed.

The MMU does not support negatively sized regions i.e. the value of the *RegionNTop* register should always be greater than or equal to the value of the *RegionNBottom* register. If *RegionNTop* is lower in the address map than *RegionNTop* then the region is considered to be zero-sized and is ignored.

When both the *RegionNTop* and *RegionNBottom* registers for a region contain the same value the region is then simply one 256-bit word in length and this corresponds to the smallest possible active region.

11.6.5.2 Region Control registers

Each memory region has a control register associated with it. The *RegionNControl* register is used to set the access conditions for the memory region bounded by the *RegionNTop* and *RegionNBottom* registers. Table 20 describes the function of each bit field in the *RegionNControl* registers. All bits in a *RegionNControl* register are both readable and writable by design. However, like all registers in the MMU, the *RegionNControl* registers can only be accessed by code running in supervisor mode.

Table 20. Region Control Register

Field Name	bit(s)	Description
SupervisorAccess	2:0	Denotes the type of access allowed when the CPU is running in Supervisor mode. For each access type a 1 indicates the access is permitted and a 0 indicates the access is not permitted. bit0 - Data read access permission bit1 - Data write access permission bit2 - Instruction fetch access permission
UserAccess	5:3	Denotes the type of access allowed when the CPU is running in User mode. For each access type a 1 indicates the access is permitted and a 0 indicates the access is not permitted. bit3 - Data read access permission bit4 - Data write access permission bit5 - Instruction fetch access permission

11.6.5.3 ExceptionSource Register

The SPARC V8 architecture allows for a number of types of memory access error to be trapped. These trap types and trap handling in general are described in chapter 7 of the SPARC architecture manual [36]. However on the LEON processor only *data_store_error* and *data_access_exception* trap types will result from an external (to LEON) bus error. According to the SPARC architecture manual the processor will automatically move to the next register window (i.e. it decrements the current window pointer) and copies the program counters (PC and nPC) to two local registers in the

new window. The supervisor bit in the PSR is also set and the PSR can be saved to another local register by the trap handler (this does not happen automatically in hardware). The *ExceptionSource* register aids the trap handler by identifying the source of an exception. Each bit in the *ExceptionSource* register is set when the relevant trap condition and should be cleared by the trap handler by writing a '1' to that bit position.

Table 21. ExceptionSource Register

Field Name	bit(s)	Description
DramAccessExcptn	0	The permissions of an access did not match those of the DRAM region it was attempting to access. This bit will also be set if an attempt is made to access an undefined DRAM region (i.e. a location that is not within the bounds of any RegionTop/RegionBottom pair)
PeriAccessExcptn	1	An access violation occurred when accessing a CPU subsystem block. This occurs when the access permissions disagree with those set by the block.
UnusedAreaExcptn	2	An attempt was made to access an unused part of the memory map
LockedWriteExcptn	3	An attempt was made to write to a regions registers (RegionTop/Bottom/Control) after they had been locked.
ResetHandlerExcptn	4	An attempt was made to access a ROM location between 0x0000_0000 and 0x0000_000F after the reset handler was executed. The most likely cause of such an access is the use of an uninitialised pointer or structure.
TimeoutExcptn	5	A bus timeout condition occurred.

11.6.6 MMU Sub-block partition

As can be seen from Figure 19 and Figure 20 the MMU consists of three principal sub-blocks. For clarity the connections between these sub-blocks and other SoPEC blocks and between each of the sub-blocks are shown in two separate diagrams.

11.6.6.1 LEON AHB Bridge

The LEON AHB bridge consists of an AHB bridge to DIU and an AHB to CPU subsystem bus bridge. The AHB bridge will convert between the AHB and the DIU and CPU subsystem bus protocols but the address decoding and enabling of an access happens elsewhere in the MMU. The AHB bridge will always be a slave on the AHB. Note that the AMBA signals from the LEON core are contained within the ahbso and ahbsi records. The LEON records are described in more detail in section 11.7. Glue logic may be required to assist with enabling memory accesses, endianness coherency, interrupts and other miscellaneous signalling.

Table 22. LEON AHB bridge I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
LEON core to LEON AHB signals (ahbsi and ahbso records)			
ahbsi.haddr[31:0]	32	In	AHB address bus
ahbsi.hwdata[31:0]	32	In	AHB write data bus
ahbso.hrdata[31:0]	32	Out	AHB read data bus
ahbsi.hsel	1	In	AHB slave select signal
ahbsi.hwrite	1	In	AHB write signal: 1 - Write access 0 - Read access
ahbsi.htrans	2	In	Indicates the type of the current transfer: 00 - IDLE 01 - BUSY 10 - NONSEQ 11 - SEQ
ahbsi.hsize	3	In	Indicates the size of the current transfer: 000 - Byte transfer 001 - Halfword transfer 010 - Word transfer 011 - 64-bit transfer (unsupported?) 1xx - Unsupported larger wordsizes
ahbsi.hburst	3	In	Indicates if the current transfer forms part of a burst and the type of burst: 000 - SINGLE 001 - INCR 010 - WRAP4 011 - INCR4 100 - WRAP8 101 - INCR8 110 - WRAP16 111 - INCR16
ahbsi.hprot	4	In	Protection control signals pertaining to the current access: hprot[0] - Opcode(0) / Data(1) access hprot[1] - User(0) / Supervisor access hprot[2] - Non-bufferable(0) / Bufferable(1) access (unsupported)

			hprot[3] - Non-cacheable(0) / Cacheable access
ahbsi.hmaster	4	In	Indicates the identity of the current bus master. This will always be the LEON core.
ahbsi.hmastlock	1	In	Indicates that the current master is performing a locked sequence of transfers.
ahbso.hready	1	Out	Active high ready signal indicating the access has completed
ahbso.hresp	2	Out	Indicates the status of the transfer: 00 - OKAY 01 - ERROR 10 - RETRY 11 - SPLIT
ahbso.hsplit[15:0]	16	Out	This 16-bit split bus is used by a slave to indicate to the arbiter which bus masters should be allowed attempt a split transaction. This feature will be unsupported on the AHB bridge
Toplevel/ Common LEON AHB bridge signals			
cpu_dataout[31:0]	32	Out	Data out bus to both DRAM and peripheral devices.
cpu_rwn	1	Out	Read/NotWrite signal. 1 = Current access is a read access, 0 = Current access is a write access
icu_cpu_ilevel[3:0]	4	In	An interrupt is asserted by driving the appropriate priority level on *icu_cpu_ilevel*. These signals must remain asserted until the CPU executes an interrupt acknowledge cycle.
cpu_icu_ilevel[3:0]	4	In	Indicates the level of the interrupt the CPU is acknowledging when *cpu_iack* is high
cpu_iack	1	Out	Interrupt acknowledge signal. The exact timing depends on the CPU core implementation
cpu_start_access	1	Out	Start Access signal indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.
cpu_ben[1:0]	2	Out	Byte enable signals.
dram_cpu_data[255:0]	256	In	Read data from the DRAM.
diu_cpu_rreq	1	Out	Read request to the DIU.

diu_cpu_rack	1	In	Acknowledge from DIU that read request has been accepted.
diu_cpu_rvalid	1	In	Signal from DIU indicating that valid read data is on the *dram_cpu_data* bus
cpu_diu_wdatavalid	1	Out	Signal from the CPU to the DIU indicating that the data currently on the *cpu_diu_wdata* bus is valid and should be committed to the DIU posted write buffer
diu_cpu_write_rdy	1	In	Signal from the DIU indicating that the posted write buffer is empty
cpu_diu_wdadr[21:4]	18	Out	Write address bus to the DIU
cpu_diu_wdata[127:0]	128	Out	Write data bus to the DIU
cpu_diu_wmask[15:0]	16	Out	Write mask for the *cpu_diu_wdata* bus. Each bit corresponds to a byte of the 128-bit *cpu_diu_wdata* bus.
LEON AHB bridge to MMU Control Block signals			
cpu_mmu_adr	32	Out	CPU Address Bus.
mmu_cpu_data	32	In	Data bus from the MMU
mmu_cpu_rdy	1	In	Ready signal from the MMU
cpu_mmu_acode	2	Out	Access code signals to the MMU
mmu_cpu_berr	1	In	Bus error signal from the MMU
dram_access_en	1	In	DRAM access enable signal. A DRAM access cannot be initiated unless it has been enabled by the MMU control unit.

Description:

The LEON AHB bridge must ensure that all CPU bus transactions are functionally correct and that the timing requirements are met. The AHB bridge also implements a 128-bit DRAM write buffer to improve the efficiency of DRAM writes, particularly for multiple successive writes to DRAM. The AHB bridge is also responsible for ensuring endianness coherency i.e. guaranteeing that the correct data appears in the correct position on the data buses (*hrdata, cpu_dataout* and *cpu_mmu_wdata)* for every type of access. This is a requirement because the LEON uses big-endian addressing while the rest of SoPEC is little-endian.

The LEON AHB bridge will assert request signals to the DIU if the MMU control block deems the access to be a legal access. The validity (i.e. is the CPU running in the correct mode for the address space being accessed) of an access is determined by the contents of the relevant *RegionNControl* register. As the SPARC standard requires that all accesses are aligned to their word size (i.e. byte, half-word, word or double-word) and so it is not possible for an access to traverse a 256-bit boundary (as required by the DIU). Invalid DRAM accesses are not propagated to the DIU and will result in an error response (ahbso.hresp = '01') on the AHB. The DIU bus protocol

is described in more detail in section 20.9. The DIU will return a 256-bit dataword on *dram_cpu_data[255:0]* for every read access.

The CPU subsystem bus protocol is described in section 11.4.3. While the LEON AHB bridge performs the protocol translation between AHB and the CPU subsystem bus the select signals for each block are generated by address decoding in the CPU subsystem bus interface. The CPU subsystem bus interface also selects the correct read data bus, ready and error signals for the block being addressed and passes these to the LEON AHB bridge which puts them on the AHB bus. It is expected that some signals (especially those external to the CPU block) will need to be registered here to meet the timing requirements. Careful thought will be required to ensure that overall CPU access times are not excessively degraded by the use of too many register stages.

11.6.6.1.1 DRAM write buffer

The DRAM write buffer improves the efficiency of DRAM writes by aggregating a number of CPU write accesses into a single DIU write access. This is achieved by checking to see if a CPU write is to an address already in the write buffer and if so the write is immediately acknowledged (i.e. the *ahbsi.hready* signal is asserted without any wait states) and the DRAM write buffer updated accordingly. When the CPU write is to a DRAM address other than that in the write buffer then the current contents of the write buffer are sent to the DIU (where they are placed in the posted write buffer) and the DRAM write buffer is updated with the address and data of the CPU write. The DRAM write buffer consists of a 128-bit data buffer, an 18-bit write address tag and a 16-bit write mask. Each bit of the write mask indicates the validity of the corresponding byte of the write buffer as shown in Figure 21 below.

The operation of the DRAM write buffer is summarised by the following set of rules:

1) The DRAM write buffer only contains DRAM write data i.e. peripheral writes go directly to the addressed peripheral.

2) CPU writes to locations within the DRAM write buffer or to an empty write buffer (i.e. the write mask bits are all 0) complete with zero wait states regardless of the size of the write (byte/half-word/word/ double-word).

3) The contents of the DRAM write buffer are flushed to DRAM whenever a CPU write to a location outside the write buffer occurs, whenever a CPU read from a location within the write buffer occurs or whenever a write to a peripheral register occurs.

4) A flush resulting from a peripheral write will not cause any extra wait states to be inserted in the peripheral write access.

5) Flushes resulting from a DRAM accesses will cause wait states to be inserted until the DIU posted write buffer is empty. If the DIU posted write buffer is empty at the time the flush is required then no wait states will be inserted for a flush resulting from a CPU write or one wait state will be inserted for a flush resulting from a CPU read (this is to ensure that the DIU sees the write request ahead of the read request). Note that in this case further wait states will also be inserted as a result of the delay in servicing the read request by the DIU.

11.6.6.1.2 DIU interface waveforms

Figure 22 below depicts the operation of the AHB bridge over a sample sequence of DRAM transactions consisting of a read into the DCache, a double-word store to an address other than that currently in the DRAM write buffer followed by an ICache line refill. To avoid clutter a number of AHB control signals that are inputs to the MMU have been grouped together as ahbsi.CONTROL and only the ahbso.HREADY is shown of the output AHB control signals.

The first transaction is a single word load ('LD'). The MMU (specifically the MMU control block) uses the first cycle of every access (i.e. the address phase of an AHB transaction) to determine whether or not the access is a legal access. The read request to the DIU is then asserted in the following cycle (assuming the access is a valid one) and is acknowledged by the DIU a cycle later. Note that the time from *cpu_diu_rreq* being asserted and *diu_cpu_rack* being asserted is variable as it depends on the DIU configuration and access patterns of DIU requestors. The AHB bridge will insert wait states until it sees the *diu_cpu_rvalid* signal is high, indicating the data ('LD1') on the *dram_cpu_data* bus is valid. The AHB bridge terminates the read access in the same cycle by asserting the ahbso.HREADY signal (together with an 'OKAY' HRESP code). The AHB bridge also selects the appropriate 32 bits ('RD1') from the 256-bit DRAM line data ('LD1') returned by the DIU corresponding to the word address given by A1.

The second transaction is an AHB two-beat incrementing burst issued by the LEON acache block in response to the execution of a double-word store instruction. As LEON is a big endian processor the address issued ('A2') during the address phase of the first beat of this transaction is the address of the most significant word of the double-word while the address for the second beat ('A3') is that of the least significant word i.e. A3 = A2 +4. The presence of the DRAM write buffer allows these writes to complete without the insertion of any wait states. This is true even when, as shown here, the DRAM write buffer needs to be flushed into the DIU posted write buffer, provided the DIU posted write buffer is empty. If the DIU posted write buffer is not empty (as would be signified by *diu_cpu_write_rdy* being low) then wait states would be inserted until it became empty. The *cpu_diu_wdata* buffer builds up the data to be written to the DIU over a number of transactions ('BD1' and 'BD2' here) while the *cpu_diu_wmask* records every byte that has been written to since the last flush - in this case the lowest word and then the second lowest word are written to as a result of the double-word store operation.

The final transaction shown here is a DRAM read caused by an ICache miss. Note that the pipelined nature of the AHB bus allows the address phase of this transaction to overlap with the final data phase of the previous transaction. All ICache misses appear as single word loads ('LD') on the AHB bus. In this case we can see that the DIU is slower to respond to this read request than to the first read request because it is processing the write access caused by the DRAM write buffer flush. The ICache refill will complete just after the window shown in Figure 22.

11.6.6.2 CPU Subsystem Bus Interface

The CPU Subsystem Interface block handles all valid accesses to the peripheral blocks that comprise the CPU Subsystem.

Table 23. CPU Subsystem Bus Interface I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
Toplevel/Common CPU Subsystem Bus Interface signals			
cpu_cpr_sel	1	Out	CPR block select.
cpu_gpio_sel	1	Out	GPIO block select.
cpu_icu_sel	1	Out	ICU block select.
cpu_lss_sel	1	Out	LSS block select.
cpu_pcu_sel	1	Out	PCU block select.
cpu_scb_sel	1	Out	SCB block select.
cpu_tim_sel	1	Out	Timers block select.
cpu_rom_sel	1	Out	ROM block select.
cpu_pss_sel	1	Out	PSS block select.
cpu_diu_sel	1	Out	DIU block select.
cpr_cpu_data[31:0]	32	In	Read data bus from the CPR block
gpio_cpu_data[31:0]	32	In	Read data bus from the GPIO block
icu_cpu_data[31:0]	32	In	Read data bus from the ICU block
lss_cpu_data[31:0]	32	In	Read data bus from the LSS block
pcu_cpu_data[31:0]	32	In	Read data bus from the PCU block
scb_cpu_data[31:0]	32	In	Read data bus from the SCB block
tim_cpu_data[31:0]	32	In	Read data bus from the Timers block
rom_cpu_data[31:0]	32	In	Read data bus from the ROM block
pss_cpu_data[31:0]	32	In	Read data bus from the PSS block
diu_cpu_data[31:0]	32	In	Read data bus from the DIU block
cpr_cpu_rdy	1	In	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the CPR block and for a read cycle this means the data on *cpr_cpu_data* is valid.
gpio_cpu_rdy	1	In	GPIO ready signal to the CPU.
icu_cpu_rdy	1	In	ICU ready signal to the CPU.
lss_cpu_rdy	1	In	LSS ready signal to the CPU.
pcu_cpu_rdy	1	In	PCU ready signal to the CPU.
scb_cpu_rdy	1	In	SCB ready signal to the CPU.
tim_cpu_rdy	1	In	Timers block ready signal to the CPU.
rom_cpu_rdy	1	In	ROM block ready signal to the CPU.
pss_cpu_rdy	1	In	PSS block ready signal to the CPU.

diu_cpu_rdy	1	In	DIU register block ready signal to the CPU.
cpr_cpu_berr	1	In	Bus Error signal from the CPR block
gpio_cpu_berr	1	In	Bus Error signal from the GPIO block
icu_cpu_berr	1	In	Bus Error signal from the ICU block
lss_cpu_berr	1	In	Bus Error signal from the LSS block
pcu_cpu_berr	1	In	Bus Error signal from the PCU block
scb_cpu_berr	1	In	Bus Error signal from the SCB block
tim_cpu_berr	1	In	Bus Error signal from the Timers block
rom_cpu_berr	1	In	Bus Error signal from the ROM block
pss_cpu_berr	1	In	Bus Error signal from the PSS block
diu_cpu_berr	1	In	Bus Error signal from the DIU block
CPU Subsystem Bus Interface to MMU Control Block signals			
cpu_adr[19:12]	8	In	Toplevel CPU Address bus. Only bits 19-12 are required to decode the peripherals address space
peri_access_en	1	In	Enable Access signal. A peripheral access cannot be initiated unless it has been enabled by the MMU Control Unit
peri_mmu_data[31:0]	32	Out	Data bus from the selected peripheral
peri_mmu_rdy	1	Out	Data Ready signal. Indicates the data on the *peri_mmu_data* bus is valid for a read cycle or that the data was successfully written to the peripheral for a write cycle.
peri_mmu_berr	1	Out	Bus Error signal. Indicates a bus error has occurred in accessing the selected peripheral
CPU Subsystem Bus Interface to LEON AHB bridge signals			
cpu_start_access	1	In	Start Access signal from the LEON AHB bridge indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.

Description:

The CPU Subsystem Bus Interface block performs simple address decoding to select a peripheral and multiplexing of the returned signals from the various peripheral blocks. The base addresses used for the decode operation are defined in Table . Note that access to the MMU configuration registers are handled by the MMU Control Block rather than the CPU Subsystem Bus Interface block. The CPU Subsystem Bus Interface block operation is described by the following pseudocode:

```
masked_cpu_adr = cpu_adr[17:12]
```

```
case (masked_cpu_adr)
  when TIM_base[17:12]
    cpu_tim_sel = peri_access_en    // The peri_access_en
signal will have the
    peri_mmu_data = tim_cpu_data    // timing required for
block selects
    peri_mmu_rdy = tim_cpu_rdy
    peri_mmu_berr = tim_cpu_berr
    all_other_selects = 0    //  Shorthand to ensure other
cpu_block_sel signals
                                    //  remain deasserted
  when LSS_base[17:12]
    cpu_lss_sel = peri_access_en
    peri_mmu_data = lss_cpu_data
    peri_mmu_rdy = lss_cpu_rdy
    peri_mmu_berr = lss_cpu_berr
    all_other_selects = 0
  when GPIO_base[17:12]
    cpu_gpio_sel = peri_access_en
    peri_mmu_data = gpio_cpu_data
    peri_mmu_rdy = gpio_cpu_rdy
    peri_mmu_berr = gpio_cpu_berr
    all_other_selects = 0
  when SCB_base[17:12]
    cpu_scb_sel = peri_access_en
    peri_mmu_data = scb_cpu_data
    peri_mmu_rdy = scb_cpu_rdy
    peri_mmu_berr = scb_cpu_berr
    all_other_selects = 0
  when ICU_base[17:12]
    cpu_icu_sel = peri_access_en
    peri_mmu_data = icu_cpu_data
    peri_mmu_rdy = icu_cpu_rdy
    peri_mmu_berr = icu_cpu_berr
    all_other_selects = 0
  when CPR_base[17:12]
    cpu_cpr_sel = peri_access_en
    peri_mmu_data = cpr_cpu_data
    peri_mmu_rdy = cpr_cpu_rdy
    peri_mmu_berr = cpr_cpu_berr
    all_other_selects = 0
```

```
when ROM_base[17:12]
   cpu_rom_sel = peri_access_en
   peri_mmu_data = rom_cpu_data
   peri_mmu_rdy = rom_cpu_rdy
   peri_mmu_berr = rom_cpu_berr
   all_other_selects = 0
when PSS_base[17:12]
   cpu_pss_sel = peri_access_en
   peri_mmu_data = pss_cpu_data
   peri_mmu_rdy = pss_cpu_rdy
   peri_mmu_berr = pss_cpu_berr
   all_other_selects = 0
when DIU_base[17:12]
   cpu_diu_sel = peri_access_en
   peri_mmu_data = diu_cpu_data
   peri_mmu_rdy = diu_cpu_rdy
   peri_mmu_berr = diu_cpu_berr
   all_other_selects = 0
when PCU_base[17:12]
   cpu_pcu_sel = peri_access_en
   peri_mmu_data = pcu_cpu_data
   peri_mmu_rdy = pcu_cpu_rdy
   peri_mmu_berr = pcu_cpu_berr
   all_other_selects = 0
when others
   all_block_selects = 0
   peri_mmu_data = 0x00000000
   peri_mmu_rdy = 0
   peri_mmu_berr = 1
end case
```

11.6.6.3 MMU Control Block

The MMU Control Block determines whether every CPU access is a valid access. No more than one cycle is to be consumed in determining the validity of an access and all accesses must terminate with the assertion of either *mmu_cpu_rdy* or *mmu_cpu_berr*. To safeguard against stalling the CPU a simple bus timeout mechanism will be supported.

Table 24. MMU Control Block I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.

pclk	1	In	Global clock
Toplevel/Common MMU Control Block signals			
cpu_adr[21:2]	22	Out	Address bus for both DRAM and peripheral access.
cpu_acode[1:0]	2	Out	CPU access code signals (*cpu_mmu_acode*) retimed to meet the CPU Subsystem Bus timing requirements
dram_access_en	1	Out	DRAM Access Enable signal. Indicates that the current CPU access is a valid DRAM access.
MMU Control Block to LEON AHB bridge signals			
cpu_mmu_adr[31:0]	32	In	CPU core address bus.
cpu_dataout[31:0]	32	In	Toplevel CPU data bus
mmu_cpu_data[31:0]	32	Out	Data bus to the CPU core. Carries the data for all CPU read operations
cpu_rwn	1	In	Toplevel CPU Read/notWrite signal.
cpu_mmu_acode[1:0]	2	In	CPU access code signals
mmu_cpu_rdy	1	Out	Ready signal to the CPU core. Indicates the completion of all valid CPU accesses.
mmu_cpu_berr	1	Out	Bus Error signal to the CPU core. This signal is asserted to terminate an invalid access.
cpu_start_access	1	In	Start Access signal from the LEON AHB bridge indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.
cpu_iack	1	In	Interrupt Acknowledge signal from the CPU. This signal is only asserted during an interrupt acknowledge cycle.
cpu_ben[1:0]	2	In	Byte enable signals indicating which bytes of the 32-bit bus are being accessed.
MMU Control Block to CPU Subsystem Bus Interface signals			
cpu_adr[17:12]	8	Out	Toplevel CPU Address bus. Only bits 17-12 are required to decode the peripherals address space
peri_access_en	1	Out	Enable Access signal. A peripheral access cannot be initiated unless it has been enabled by the MMU Control Unit
peri_mmu_data[31:0]	32	In	Data bus from the selected peripheral
peri_mmu_rdy	1	In	Data Ready signal. Indicates the data on the *peri_mmu_data* bus is valid for a read cycle or that the data was successfully written to the peripheral for a write cycle.

peri_mmu_berr	1	In	Bus Error signal. Indicates a bus error has occurred in accessing the selected peripheral

Description:

The MMU Control Block is responsible for the MMU's core functionality, namely determining whether or not an access to any part of the address map is valid. An access is considered valid if it is to a mapped area of the address space and if the CPU is running in the appropriate mode for that address space. Furthermore the MMU control block must correctly handle the special cases that are: an interrupt acknowledge cycle, a reset exception vector fetch, an access that crosses a 256-bit DRAM word boundary and a bus timeout condition. The following pseudocode shows the logic required to implement the MMU Control Block functionality. It does not deal with the timing relationships of the various signals - it is the designer's responsibility to ensure that these relationships are correct and comply with the different bus protocols. For simplicity the pseudocode is split up into numbered sections so that the functionality may be seen more easily.

It is important to note that the style used for the pseudocode will differ from the actual coding style used in the RTL implementation. The pseudocode is only intended to capture the required functionality, to clearly show the criteria that need to be tested rather than to describe how the implementation should be performed. In particular the different comparisons of the address used to determine which part of the memory map, which DRAM region (if applicable) and the permission checking should all be performed in parallel (with results ORed together where appropriate) rather than sequentially as the pseudocode implies.

PS0 Description: This first segment of code defines a number of constants and variables that are used elsewhere in this description. Most signals have been defined in the I/O descriptions of the MMU sub-blocks that precede this section of the document. The *post_reset_state* variable is used later (in section PS4) to determine if we should trap a null pointer access.

PS0:

```
const UnusedBottom = 0x002AC000
const DRAMTop = 0x4027FFFF
const UserDataSpace = b01
const UserProgramSpace = b00
const SupervisorDataSpace = b11
const SupervisorProgramSpace = b10
const ResetExceptionCycles = 0x2

cpu_adr_peri_masked[5:0] = cpu_mmu_adr[17:12]
cpu_adr_dram_masked[16:0] = cpu_mmu_adr & 0x003FFFE0

if (prst_n == 0) then      // Initialise everything
   cpu_adr = cpu_mmu_adr[21:2]
   peri_access_en = 0
```

```
dram_access_en = 0
mmu_cpu_data = peri_mmu_data
mmu_cpu_rdy = 0
mmu_cpu_berr = 0
post_reset_state = TRUE
access_initiated = FALSE
cpu_access_cnt = 0

  // The following is used to determine if we are coming out
of reset for the purposes of
  // reset exception vector redirection. There may be a
convenient signal in the CPU core
  // that we could use instead of this.
  if ((cpu_start_access == 1) AND (cpu_access_cnt <
ResetExceptionCycles) AND
       (clock_tick == TRUE)) then
    cpu_access_cnt = cpu_access_cnt +1
  else
    post_reset_state = FALSE
```

PS1 Description: This section is at the top of the hierarchy that determines the validity of an access. The address is tested to see which macro-region (i.e. Unused, CPU Subsystem or DRAM) it falls into or whether the reset exception vector is being accessed.

```
PS1:
  if (cpu_mmu_adr >= UnusedBottom) then
      // The access is to an invalid area of the address
space. See section PS2

  elsif ((cpu_mmu_adr > DRAMTop) AND (cpu_mmu_adr <
UnusedBottom)) then
    // We are in the CPU Subsystem/PEP Subsystem address
space. See section PS3

  // Only remaining possibility is an access to DRAM address
space
  // First we need to intercept the special case for the
reset exception vector

  elsif (cpu_mmu_adr < 0x00000010) then
    // The reset exception is being accessed. See section PS4
```

```
  elsif    ((cpu_adr_dram_masked    >=    Region0Bottom)    AND
(cpu_adr_dram_masked <=
        Region0Top) ) then
      // We are in Region0. See section PS5

  elsif    ((cpu_adr_dram_masked    >=    RegionNBottom)    AND
(cpu_adr_dram_masked <=
        RegionNTop) ) then  // we are in RegionN
       // Repeat the Region0 (i.e. section PS5) logic for
each of Region1 to Region7

  else  // We could end up here if there were gaps in the
DRAM regions
    peri_access_en = 0
    dram_access_en = 0
    mmu_cpu_berr = 1     // we have an unknown access error,
most likely due to hitting
    mmu_cpu_rdy = 0      // a gap in the DRAM regions

  // Only thing remaining is to implement a bus timeout
function. This is done in PS6

  end
```

PS2 Description: Accesses to the large unused area of the address space are trapped by this section. No bus transactions are initiated and the *mmu_cpu_berr* signal is asserted.

PS2:

```
  elsif (cpu_mmu_adr >= UnusedBottom) then
    peri_access_en = 0  // The access is to an invalid area
of the address space
    dram_access_en = 0
    mmu_cpu_berr = 1
    mmu_cpu_rdy = 0
```

PS3 Description: This section deals with accesses to CPU Subsystem peripherals, including the MMU itself. If the MMU registers are being accessed then no external bus transactions are required. Access to the MMU registers is only permitted if the CPU is making a data access from supervisor mode, otherwise a bus error is asserted and the access terminated. For non-MMU accesses then transactions occur over the CPU Subsystem Bus and each peripheral is responsible for determining whether or not the CPU is in the correct mode (based on the *cpu_acode* signals) to be permitted

access to its registers. Note that all of the PEP registers are accessed via the PCU which is on the CPU Subsystem Bus.

```
PS3:
  elsif ((cpu_mmu_adr > DRAMTop) AND (cpu_mmu_adr < UnusedBottom)) then
      // We are in the CPU Subsystem/PEP Subsystem address space

      cpu_adr = cpu_mmu_adr[21:2]
      if (cpu_adr_peri_masked == MMU_base) then  // access is to local registers
        peri_access_en = 0
        dram_access_en = 0
        if (cpu_acode == SupervisorDataSpace) then
          for (i=0; i<26; i++) {
            if ((i == cpu_mmu_adr[6:2]) then   // selects the addressed register
              if (cpu_rwn == 1) then
                mmu_cpu_data[16:0] = MMUReg[i]    // MMUReg[i] is one of the
                mmu_cpu_rdy = 1                   // registers in Table
                mmu_cpu_berr = 0
              else  // write cycle
                MMUReg[i] = cpu_dataout[16:0]
                mmu_cpu_rdy = 1
                mmu_cpu_berr = 0
            else   // there is no register mapped to this address
              mmu_cpu_berr = 1   // do we really want a bus_error here as registers
                mmu_cpu_rdy = 0  // are just mirrored in other blocks

        else // we have an access violation
          mmu_cpu_berr = 1
          mmu_cpu_rdy = 0

      else // access is to something else on the CPU Subsystem Bus
```

```
peri_access_en = 1
dram_access_en = 0
mmu_cpu_data = peri_mmu_data
mmu_cpu_rdy = peri_mmu_rdy
mmu_cpu_berr = peri_mmu_berr
```

PS4 Description: The only correct accesses to the locations beneath 0x00000010 are fetches of the reset trap handling routine and these should be the first accesses after reset. Here we trap all other accesses to these locations regardless of the CPU mode. The most likely cause of such an access will be the use of a null pointer in the program executing on the CPU.

PS4:

```
elsif (cpu_mmu_adr < 0x00000010) then
  if (post_reset_state == TRUE)) then
    cpu_adr = cpu_mmu_adr[21:2]
    peri_access_en = 1
    dram_access_en = 0
    mmu_cpu_data = peri_mmu_data
    mmu_cpu_rdy = peri_mmu_rdy
    mmu_cpu_berr = peri_mmu_berr
  else   // we have a problem (almost certainly a null
pointer)
    peri_access_en = 0
    dram_access_en = 0
    mmu_cpu_berr = 1
    mmu_cpu_rdy = 0
```

PS5 Description: This large section of pseudocode simply checks whether the access is within the bounds of DRAM Region0 and if so whether or not the access is of a type permitted by the *Region0Control* register. If the access is permitted then a DRAM access is initiated. If the access is not of a type permitted by the *Region0Control* register then the access is terminated with a bus error.

PS5:

```
elsif    ((cpu_adr_dram_masked    >=    Region0Bottom)    AND
(cpu_adr_dram_masked <=
        Region0Top) ) then  // we are in Region0

    cpu_adr = cpu_mmu_adr[21:2]
    if (cpu_rwn == 1) then
```

```
        if    ((cpu_acode    ==    SupervisorProgramSpace    AND
Region0Control[2] == 1))
              OR     (cpu_acode     ==     UserProgramSpace     AND
Region0Control[5] == 1)) then
                                    //  this  is  a  valid  instruction
fetch from Region0
                                    //  The  dram_cpu_data  bus  goes
directly to the LEON
                                    //  AHB  bridge  which  also  handles
the hready generation
                peri_access_en = 0
                dram_access_en = 1
                mmu_cpu_berr = 0

          elsif    ((cpu_acode    ==    SupervisorDataSpace    AND
Region0Control[0] == 1)
                OR     (cpu_acode     ==     UserDataSpace     AND
Region0Control[3] == 1)) then
                                                //  this  is  a  valid
read access from Region0
                peri_access_en = 0
                dram_access_en = 1
                mmu_cpu_berr = 0

           else                                 // we have an access
violation
                peri_access_en = 0
                dram_access_en = 0
                mmu_cpu_berr = 1
                mmu_cpu_rdy = 0

       else                                  // it is a write access
         if     ((cpu_acode     ==     SupervisorDataSpace     AND
Region0Control[1] == 1)
                OR     (cpu_acode     ==     UserDataSpace     AND
Region0Control[4] == 1)) then
                                              //  this  is  a  valid
write access to Region0
                peri_access_en = 0
                dram_access_en = 1
                mmu_cpu_berr = 0
```

```
        else                                    // we have an access
violation
            peri_access_en = 0
            dram_access_en = 0
            mmu_cpu_berr = 1
            mmu_cpu_rdy = 0
```

PS6 Description: This final section of pseudocode deals with the special case of a bus timeout. This occurs when an access has been initiated but has not completed before the *BusTimeout* number of *pclk* cycles. While access to both DRAM and CPU/PEP Subsystem registers will take a variable number of cycles (due to DRAM traffic, PCU command execution or the different timing required to access registers in imported IP) each access should complete before a timeout occurs. Therefore it should not be possible to stall the CPU by locking either the CPU Subsystem or DIU buses. However given the fatal effect such a stall would have it is considered prudent to implement bus timeout detection.

PS6:

```
  // Only thing remaining is to implement a bus timeout
function.

  if ((cpu_start_access == 1) then
    access_initiated = TRUE
    timeout_countdown = BusTimeout

  if ((mmu_cpu_rdy == 1 ) OR (mmu_cpu_berr ==1 )) then
    access_initiated = FALSE
    peri_access_en = 0
    dram_access_en = 0

  if ((clock_tick == TRUE) AND (access_initiated == TRUE) AND
(BusTimeout != 0))
    if (timeout_countdown > 0) then
      timeout_countdown--
    else  // timeout has occurred
      peri_access_en = 0          // abort the access
      dram_access_en = 0
      mmu_cpu_berr = 1
      mmu_cpu_rdy = 0
```

11.7 LEON CACHES

The version of LEON implemented on SoPEC features 1 kB of ICache and 1 kB of DCache. Both caches are direct mapped and feature 8 word lines so their data RAMs are arranged as 32 x 256-bit

and their tag RAMs as 32 x 30-bit (itag) or 32 x 32-bit (dtag). Like most of the rest of the LEON code used on SoPEC the cache controllers are taken from the leon2-1.0.7 release. The LEON cache controllers and cache RAMs have been modified to ensure that an entire 256-bit line is refilled at a time to make maximum use out of the memory bandwidth offered by the embedded DRAM organization (DRAM lines are also 256-bit). The data cache controller has also been modified to ensure that user mode code cannot access the DCache contents unless it is authorised to do so. A block diagram of the LEON CPU core as implemented on SoPEC is shown in Figure 23 below. In this diagram dotted lines are used to indicate hierarchy and red items represent signals or wrappers added as part of the SoPEC modifications. LEON makes heavy use of VHDL records and the records used in the CPU core are described in Table 25. Unless otherwise stated the records are defined in the iface.vhd file (part of the LEON release) and this should be consulted for a complete breakdown of the record elements.

Table 25. Relevant LEON records

Record Name	Description
rfi	Register File Input record. Contains address, datain and control signals for the register file.
rfo	Register File Output record. Contains the data out of the dual read port register file.
ici	Instruction Cache In record. Contains program counters from different stages of the pipeline and various control signals
ico	Instruction Cache Out record. Contains the fetched instruction data and various control signals. This record is also sent to the DCache (i.e. icol) so that diagnostic accesses (e.g. *lda/sta*) can be serviced.
dci	Data Cache In record. Contains address and data buses from different stages of the pipeline (execute & memory) and various control signals
dco	Data Cache Out record. Contains the data retrieved from either memory or the caches and various control signals. This record is also sent to the ICache (i.e. dcol) so that diagnostic accesses (e.g. *lda/sta*) can be serviced.
iui	Integer Unit In record. This record contains the interrupt request level and a record for use with LEONs Debug Support Unit (DSU)
iuo	Integer Unit Out record. This record contains the acknowledged interrupt request level with control signals and a record for use with LEONs Debug Support Unit (DSU)
mcii	Memory to Cache Icache In record. Contains the address of an Icache miss and various control signals
mcio	Memory to Cache Icache Out record. Contains the returned data from memory and various control signals
mcdi	Memory to Cache Dcache In record. Contains the address and data of a

	Dcache miss or write and various control signals
mcdo	Memory to Cache Dcache Out record. Contains the returned data from memory and various control signals
ahbi	AHB In record. This is the input record for an AHB master and contains the data bus and AHB control signals. The destination for the signals in this record is the AHB controller. This record is defined in the amba.vhd file
ahbo	AHB Out record. This is the output record for an AHB master and contains the address and data buses and AHB control signals. The AHB controller drives the signals in this record. This record is defined in the amba.vhd file
ahbsi	AHB Slave In record. This is the input record for an AHB slave and contains the address and data buses and AHB control signals. It is used by the DCache to facilitate cache snooping (this feature is not enabled in SoPEC). This record is defined in the amba.vhd file
crami	Cache RAM In record. This record is composed of records of records which contain the address, data and tag entries with associated control signals for both the ICache RAM and DCache RAM
cramo	Cache RAM Out record. This record is composed of records of records which contain the data and tag entries with associated control signals for both the ICache RAM and DCache RAM
iline_rdy	Control signal from the ICache controller to the instruction cache memory. This signal is active (high) when a full 256-bit line (on *dram_cpu_data*) is to be written to cache memory.
dline_rdy	Control signal from the DCache controller to the data cache memory. This signal is active (high) when a full 256-bit line (on dram_cpu_data) is to be written to cache memory.
dram_cpu_data	256-bit data bus from the embedded DRAM

11.7.1 Cache controllers

The LEON cache module consists of three components: the ICache controller (icache.vhd), the DCache controller (dcache.vhd) and the AHB bridge (acache.vhd) which translates all cache misses into memory requests on the AHB bus.

In order to enable full line refill operation a few changes had to be made to the cache controllers. The ICache controller was modified to ensure that whenever a location in the cache was updated (i.e. the cache was enabled and was being refilled from DRAM) all locations on that cache line had their valid bits set to reflect the fact that the full line was updated. The *iline_rdy* signal is asserted by the ICache controller when this happens and this informs the cache wrappers to update all locations in the idata RAM for that line.

A similar change was made to the DCache controller except that the entire line was only updated following a read miss and that existing write through operation was preserved. The DCache controller uses the *dline_rdy* signal to instruct the cache wrapper to update all locations in the ddata RAM for a line. An additional modification was also made to ensure that a double-word load

instruction from a non-cached location would only result in one read access to the DIU i.e. the second read would be serviced by the data cache. Note that if the DCache is turned off then a double-word load instruction will cause two DIU read accesses to occur even though they will both be to the same 256-bit DRAM line.

The DCache controller was further modified to ensure that user mode code cannot access cached data to which it does not have permission (as determined by the relevant *RegionNControl* register settings at the time the cache line was loaded). This required an extra 2 bits of tag information to record the user read and write permissions for each cache line. These user access permissions can be updated in the same manner as the other tag fields (i.e. address and valid bits) namely by line refill, STA instruction or cache flush. The user access permission bits are checked every time user code attempts to access the data cache and if the permissions of the access do not agree with the permissions returned from the tag RAM then a cache miss occurs. As the MMU evaluates the access permissions for every cache miss it will generate the appropriate exception for the forced cache miss caused by the errant user code. In the case of a prohibited read access the trap will be immediate while a prohibited write access will result in a deferred trap. The deferred trap results from the fact that the prohibited write is committed to a write buffer in the DCache controller and program execution continues until the prohibited write is detected by the MMU which may be several cycles later. Because the errant write was treated as a write miss by the DCache controller (as it did not match the stored user access permissions) the cache contents were not updated and so remain coherent with the DRAM contents (which do not get updated because the MMU intercepted the prohibited write). Supervisor mode code is not subject to such checks and so has free access to the contents of the data cache.

In addition to AHB bridging, the ACache component also performs arbitration between ICache and DCache misses when simultaneous misses occur (the DCache always wins) and implements the Cache Control Register (CCR). The leon2-1.0.7 release is inconsistent in how it handles cacheability: For instruction fetches the cacheability (i.e. is the access to an area of memory that is cacheable) is determined by the ICache controller while the ACache determines whether or not a data access is cacheable. To further complicate matters the DCache controller does determine if an access resulting from a cache snoop by another AHB master is cacheable (Note that the SoPEC ASIC does not implement cache snooping as it has no need to do so). This inconsistency has been cleaned up in more recent LEON releases but is preserved here to minimise the number of changes to the LEON RTL. The cache controllers were modified to ensure that only DRAM accesses (as defined by the SoPEC memory map) are cached.

The only functionality removed as a result of the modifications was support for burst fills of the ICache. When enabled burst fills would refill an ICache line from the location where a miss occurred up to the end of the line. As the entire line is now refilled at once (when executing from DRAM) this functionality is no longer required. Furthermore more substantial modifications to the ICache controller would be needed if we wished to preserve this function without adversely affecting full line refills. The CCR was therefore modified to ensure that the instruction burst fetch bit (bit16) was tied low and could not be written to.

11.7.1.1 LEON Cache Control Register

The CCR controls the operation of both the I and D caches. Note that the bitfields used on the SoPEC implementation of this register are based on the LEON v1.0.7 implementation and some bits have their values tied off. See section 4 of the LEON manual for a description of the LEON cache controllers.

Table 26. LEON Cache Control Register

Field Name	bit(s)	Description
ICS	1:0	Instruction cache state: 00 - disabled 01 - frozen 10 - disabled 11 - enabled
Reserved	13:6	Reserved. Reads as 0.
DCS	3:2	Data cache state: 00 - disabled 01 - frozen 10 - disabled 11 - enabled
IF	4	ICache freeze on interrupt 0 - Do not freeze the ICache contents on taking an interrupt 1 - Freeze the ICache contents on taking an interrupt
DF	5	DCache freeze on interrupt 0 - Do not freeze the DCache contents on taking an interrupt 1 - Freeze the DCache contents on taking an interrupt
Reserved	13:6	Reserved. Reads as 0.
DP	14	Data cache flush pending. 0 - No DCache flush in progress 1 - DCache flush in progress This bit is ReadOnly.
IP	15	Instruction cache flush pending. 0 - No ICache flush in progress 1 - ICache flush in progress This bit is ReadOnly.
IB	16	Instruction burst fetch enable. This bit is tied low on SoPEC because it would interfere with the operation of the cache wrappers. Burst refill functionality is automatically provided in SoPEC by the cache wrappers.
Reserved	20:17	Reserved. Reads as 0.

FI	21	Flush instruction cache. Writing a 1 this bit will flush the ICache. Reads as 0.
FD	22	Flush data cache. Writing a 1 this bit will flush the DCache. Reads as 0.
DS	23	Data cache snoop enable. This bit is tied low in SoPEC as there is no requirement to snoop the data cache.
Reserved	31:24	Reserved. Reads as 0.

11.7.2 Cache wrappers

The cache RAMs used in the leon2-1.0.7 release needed to be modified to support full line refills and the correct IBM macros also needed to be instantiated. Although they are described as RAMs throughout this document (for consistency), register arrays are actually used to implement the cache RAMs. This is because IBM SRAMs were not available in suitable configurations (offered configurations were too big) to implement either the tag or data cache RAMs. Both instruction and data tag RAMs are implemented using dual port (1 Read & 1 Write) register arrays and the clocked write-through versions of the register arrays were used as they most closely approximate the single port SRAM LEON expects to see.

11.7.2.1 Cache Tag RAM wrappers

The itag and dtag RAMs differ only in their width - the itag is a 32x30 array while the dtag is a 32x32 array with the extra 2 bits being used to record the user access permissions for each line. When read using a LDA instruction both tags return 32-bit words. The tag fields are described in Table 27 and Table 28 below. Using the IBM naming conventions the register arrays used for the tag RAMs are called RA032X30D2P2W1R1M3 for the itag and RA032X32D2P2W1R1M3 for the dtag. The *ibm_syncram* wrapper used for the tag RAMs is a simple affair that just maps the wrapper ports on to the appropriate ports of the IBM register array and ensures the output data has the correct timing by registering it. The tag RAMs do not require any special modifications to handle full line refills.

Table 27. LEON Instruction Cache Tag

Field Name	bit(s)	Description
Valid	7:0	Each valid bit indicates whether or not the corresponding word of the cache line contains valid data
Reserved	9:8	Reserved - these bits do not exist in the itag RAM. Reads as 0.
Address	31:10	The tag address of the cache line

Table 28. LEON Data Cache Tag

Field Name	bit(s)	Description
Valid	7:0	Each valid bit indicates whether or not the corresponding word of the cache line contains valid data

URP	8	User read permission. 0 - User mode reads will force a refill of this line 1 - User mode code can read from this cache line.
UWP	9	User write permission. 0 - User mode writes will not be written to the cache 1 - User mode code can write to this cache line.
Address	31:10	The tag address of the cache line

11.7.2.2 Cache Data RAM wrappers

The cache data RAM contains the actual cached data and nothing else. Both the instruction and data cache data RAMs are implemented using 8 32x32-bit register arrays and some additional logic to support full line refills. Using the IBM naming conventions the register arrays used for the tag RAMs are called RA032X32D2P2W1R1M3. The *ibm_cdram_wrap* wrapper used for the tag RAMs is shown in Figure 24 below.

To the cache controllers the cache data RAM wrapper looks like a 256x32 single port SRAM (which is what they expect to see) with an input to indicate when a full line refill is taking place (the *line_rdy* signal). Internally the 8-bit address bus is split into a 5-bit lineaddress, which selects one of the 32 256-bit cache lines, and a 3-bit wordaddress which selects one of the 8 32-bit words on the cache line. Thus each of the 8 32x32 register arrays contains one 32-bit word of each cache line. When a full line is being refilled (indicated by both the *line_rdy* and *write* signals being high) every register array is written to with the appropriate 32 bits from the *linedatain* bus which contains the 256-bit line returned by the DIU after a cache miss. When just one word of the cache line is to be written (indicated by the *write* signal being high while the *line_rdy* is low) then the wordaddress is used to enable the write signal to the selected register array only - all other write enable signals are kept low. The data cache controller handles byte and half-word write by means of a read-modify-write operation so writes to the cache data RAM are always 32-bit.

The wordaddress is also used to select the correct 32-bit word from the cache line to return to the LEON integer unit.

11.8 REALTIME DEBUG UNIT (RDU)

The RDU facilitates the observation of the contents of most of the CPU addressable registers in the SoPEC device in addition to some pseudo-registers in realtime. The contents of pseudo-registers, i.e. registers that are collections of otherwise unobservable signals and that do not affect the functionality of a circuit, are defined in each block as required. Many blocks do not have pseudo-registers and some blocks (e.g. ROM, PSS) do not make debug information available to the RDU as it would be of little value in realtime debug.

Each block that supports realtime debug observation features a *DebugSelect* register that controls a local mux to determine which register is output on the block's data bus (i.e. *block_cpu_data*). One small drawback with reusing the blocks data bus is that the debug data cannot be present on the same bus during a CPU read from the block. An accompanying active high *block_cpu_debug_valid* signal is used to indicate when the data bus contains valid debug data and when the bus is being

used by the CPU. There is no arbitration for the bus as the CPU will always have access when required. A block diagram of the RDU is shown in Figure 25.

Table 29. RDU I/Os

Port name	Pins	I/O	Description
diu_cpu_data	32	In	Read data bus from the DIU block
cpr_cpu_data	32	In	Read data bus from the CPR block
gpio_cpu_data	32	In	Read data bus from the GPIO block
icu_cpu_data	32	In	Read data bus from the ICU block
lss_cpu_data	32	In	Read data bus from the LSS block
pcu_cpu_debug_data	32	In	Read data bus from the PCU block
scb_cpu_data	32	In	Read data bus from the SCB block
tim_cpu_data	32	In	Read data bus from the TIM block
diu_cpu_debug_valid	1	In	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.
tim_cpu_debug_valid	1	In	Signal indicating the data on the *tim_cpu_data* bus is valid debug data.
scb_cpu_debug_valid	1	In	Signal indicating the data on the *scb_cpu_data* bus is valid debug data.
pcu_cpu_debug_valid	1	In	Signal indicating the data on the *pcu_cpu_data* bus is valid debug data.
lss_cpu_debug_valid	1	In	Signal indicating the data on the *lss_cpu_data* bus is valid debug data.
icu_cpu_debug_valid	1	In	Signal indicating the data on the *icu_cpu_data* bus is valid debug data.
gpio_cpu_debug_valid	1	In	Signal indicating the data on the *gpio_cpu_data* bus is valid debug data.
cpr_cpu_debug_valid	1	In	Signal indicating the data on the *cpr_cpu_data* bus is valid debug data.
debug_data_out	32	Out	Output debug data to be muxed on to the PHI/GPIO/other pins
debug_data_valid	1	Out	Debug valid signal indicating the validity of the data on *debug_data_out*. This signal is used in all debug configurations
debug_cntrl	33	Out	Control signal for each debug data line indicating whether or not the debug data should be selected by the pin mux

5

As there are no spare pins that can be used to output the debug data to an external capture device some of the existing I/Os will have a debug multiplexer placed in front of them to allow them be used as debug pins. Furthermore not every pin that has a debug mux will always be available to carry the debug data as they may be engaged in their primary purpose e.g. as a GPIO pin. The RDU therefore outputs a *debug_cntrl* signal with each debug data bit to indicate whether the mux associated with each debug pin should select the debug data or the normal data for the pin. The *DebugPinSel1* and *DebugPinSel2* registers are used to determine which of the 33 potential debug pins are enabled for debug at any particular time.

As it may not always be possible to output a full 32-bit debug word every cycle the RDU supports the outputting of an n-bit sub-word every cycle to the enabled debug pins. Each debug test would then need to be re-run a number of times with a different portion of the debug word being output on the n-bit sub-word each time. The data from each run should then be correlated to create a full 32-bit (or whatever size is needed) debug word for every cycle. The *debug_data_valid* and *pclk_out* signals will accompany every sub-word to allow the data to be sampled correctly. The *pclk_out* signal is sourced close to its output pad rather than in the RDU to minimise the skew between the rising edge of the debug data signals (which should be registered close to their output pads) and the rising edge of *pclk_out*.

As multiple debug runs will be needed to obtain a complete set of debug data the n-bit sub-word will need to contain a different bit pattern for each run. For maximum flexibility each debug pin has an associated *DebugDataSrc* register that allows any of the 32 bits of the debug data word to be output on that particular debug data pin. The debug data pin must be enabled for debug operation by having its corresponding bit in the *DebugPinSel* registers set for the selected debug data bit to appear on the pin.

The size of the sub-word is determined by the number of enabled debug pins which is controlled by the *DebugPinSel* registers. Note that the *debug_data_valid* signal is always output. Furthermore *debug_cntrl[0]* (which is configured by *DebugPinSel1*) controls the mux for both the *debug_data_valid* and *pclk_out* signals as both of these must be enabled for any debug operation.

The mapping of *debug_data_out[n]* signals onto individual pins will take place outside the RDU. This mapping is described in Table 30 below.

Table 30. DebugPinSel mapping

bit #	Pin
DebugPinSel1	phi_frclk. The *debug_data_valid* signal will appear on this pin when enabled. Enabling this pin also automatically enables the phi_readl pin which will output the *pclk_out* signal
DebugPinSel2(0-31)	gpio[0...31]

Table 31. RDU Configuration Registers

Address offset from	Register	#bits	Reset	Description

MMU_base				
0x80	DebugSrc	4	0x00	Denotes which block is supplying the debug data. The encoding of this block is given below. 0 - MMU 1 - TIM 2 - LSS 3- GPIO 4 - SCB 5 - ICU 6 - CPR 7 - DIU 8 – PCU
0x84	DebugPinSel 1	1	0x0	Determines whether the phi_frclk and phi_readl pins are used for debug output. 1 - Pin outputs debug data 0 - Normal pin function
0x88	DebugPinSel 2	32	0x000 0_000 0	Determines whether a pin is used for debug data output. 1 - Pin outputs debug data 0 - Normal pin function
0x8C to 0x108	DebugDataSr c[31:0]	32 x 5	0x00	Selects which bit of the 32-bit debug data word will be output on debug_data_out[N]

11.9 INTERRUPT OPERATION

The interrupt controller unit (see chapter 14) generates an interrupt request by driving interrupt request lines with the appropriate interrupt level. LEON supports 15 levels of interrupt with level 15 as the highest level (the SPARC architecture manual [36] states that level 15 is non-maskable but we have the freedom to mask this if desired). The CPU will begin processing an interrupt exception when execution of the current instruction has completed and it will only do so if the interrupt level is higher than the current processor priority. If a second interrupt request arrives with the same level as an executing interrupt service routine then the exception will not be processed until the executing routine has completed.

When an interrupt trap occurs the LEON hardware will place the program counters (PC and nPC) into two local registers. The interrupt handler routine is expected, as a minimum, to place the PSR register in another local register to ensure that the LEON can correctly return to its pre-interrupt state. The 4-bit interrupt level (*irl*) is also written to the trap type (*tt*) field of the TBR (Trap Base Register) by hardware. The TBR then contains the vector of the trap handler routine the processor will then jump. The TBA (Trap Base Address) field of the TBR must have a valid value before any interrupt processing can occur so it should be configured at an early stage.

Interrupt pre-emption is supported while ET (Enable Traps) bit of the PSR is set. This bit is cleared during the initial trap processing. In initial simulations the ET bit was observed to be cleared for up to 30 cycles. This causes significant additional interrupt latency in the worst case where a higher priority interrupt arrives just as a lower priority one is taken.

The interrupt acknowledge cycles shown in Figure 26 below are derived from simulations of the LEON processor. The SoPEC toplevel interrupt signals used in this diagram map directly to the LEON interrupt signals in the *iui* and *iuo* records. An interrupt is asserted by driving its (encoded) level on the *icu_cpu_ilevel[3:0]* signals (which map to *iui.irl[3:0]*). The LEON core responds to this, with variable timing, by reflecting the level of the taken interrupt on the *cpu_icu_ilevel[3:0]* signals (mapped to *iuo.irl[3:0])* and asserting the acknowledge signal *cpu_iack (iuo.intack)*.The interrupt controller then removes the interrupt level one cycle after it has seen the level been acknowledged by the core. If there is another pending interrupt (of lower priority) then this should be driven on *icu_cpu_ilevel[3:0]* and the CPU will take that interrupt (the level 9 interrupt in the example below) once it has finished processing the higher priority interrupt. The *cpu_icu_ilevel[3:0]* signals always reflect the level of the last taken interrupt, even when the CPU has finished processing all interrupts.

11.10 BOOT OPERATION

See section 17.2 for a description of the SoPEC boot operation.

11.11 SOFTWARE DEBUG

Software debug mechanisms are discussed in the "SoPEC Software Debug" document [15].

12 Serial Communications Block (SCB)

12.1 OVERVIEW

The Serial Communications Block (SCB) handles the movement of all data between the SoPEC and the host device (e.g. PC) and between master and slave SoPEC devices. The main components of the SCB are a Full-Speed (FS) USB Device Core, a FS USB Host Core, a Inter-SoPEC Interface (ISI), a DMA manager, the SCB Map and associated control logic. The need for these components and the various types of communication they provide is evident in a multi-SoPEC printer configuration.

12.1.1 Multi-SoPEC systems

While single SoPEC systems are expected to form the majority of SoPEC systems the SoPEC device must also support its use in multi-SoPEC systems such as that shown in Figure 27. A SoPEC may be assigned any one of a number of identities in a multi-SoPEC system. A SoPEC may be one or more of a PrintMaster, a LineSyncMaster, an ISIMaster, a StorageSoPEC or an ISISlave SoPEC.

12.1.1.1 ISIMaster device

The ISIMaster is the only device that controls the common ISI lines (see Figure 30) and typically interfaces directly with the host. In most systems the ISIMaster will simply be the SoPEC connected to the USB bus. Future systems, however, may employ an ISI-Bridge chip to interface between the host and the ISI bus and in such systems the ISI-Bridge chip will be the ISIMaster. There can only be one ISIMaster on an ISI bus.

Systems with multiple SoPECs may have more than one host connection, for example there could be two SoPECs communicating with the external host over their FS USB links (this would of course require two USB cables to be connected), but still only one ISIMaster.

While it is not expected to be required, it is possible for a device to hand over its role as the ISIMaster to another device on the ISI i.e. the ISIMaster is not necessarily fixed.

12.1.1.2 PrintMaster device

The PrintMaster device is responsible for co-ordinating all aspects of the print operation. This includes starting the print operation in all printing SoPECs and communicating status back to the external host. When the ISIMaster is a SoPEC device it is also likely to be the PrintMaster as well. There may only be one PrintMaster in a system and it is most likely to be a SoPEC device.

12.1.1.3 LineSyncMaster device

The LineSyncMaster device generates the *lsync* pulse that all SoPECs in the system must synchronize their line outputs with. Any SoPEC in the system could act as a LineSyncMaster although the PrintMaster is probably the most likely candidate. It is possible that the LineSyncMaster may not be a SoPEC device at all - it could, for example, come from some OEM motor control circuitry. There may only be one LineSyncMaster in a system.

12.1.1.4 Storage device

For certain printer types it may be realistic to use one SoPEC as a storage device without using its print engine capability - that is to effectively use it as an ISI-attached DRAM. A storage SoPEC would receive data from the ISIMaster (most likely to be an ISI-Bridge chip) and then distribute it to the other SoPECs as required. No other type of data flow (e.g. ISISlave -> storage SoPEC -> ISISlave) would need to be supported in such a scenario. The SCB supports this functionality at no additional cost because the CPU handles the task of transferring outbound data from the embedded DRAM to the ISI transmit buffer. The CPU in a storage SoPEC will have almost nothing else to do.

12.1.1.5 ISISlave device

Multi-SoPEC systems will contain one or more ISISlave SoPECs. An ISISlave SoPEC is primarily used to generate dot data for the printhead IC it is driving. An ISISlave will not transmit messages on the ISI without first receiving permission to do so, via a ping packet (see section 12.4.4.6), from the ISIMaster

12.1.1.6 ISI-Bridge device

SoPEC is targeted at the low-cost small office / home office (SoHo) market. It may also be used in future systems that target different market segments which are likely to have a high speed interface capability. A future device, known as an ISI-Bridge chip, is envisaged which will feature both a high speed interface (such as High-Speed (HS) USB, Ethernet or IEEE1394) and one or more ISI interfaces. The use of multiple ISI buses would allow the construction of independent print systems within the one printer. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.

12.1.1.7 External host

The external host is most likely (but is not required) to be, a PC. Any system that can act as a USB host or that can interface to an ISI-Bridge chip could be the external host. In particular, with the

development of USB On-The-Go (USB OTG), it is possible that a number of USB OTG enabled products such as PDAs or digital cameras will be able to directly interface with a SoPEC printer.

12.1.1.8 External USB device

The external USB device is most likely (but is not required) to be, a digital camera. Any system that can act as a USB device could be connected as an external USB device. This is to facilitate printing in the absence of a PC.

12.1.2 Types of communication

12.1.2.1 Communications with external host

The external host communicates directly with the ISIMaster in order to print pages. When the ISIMaster is a SoPEC, the communications channel is FS USB.

12.1.2.1.1 External host to ISIMaster communication

The external host will need to communicate the following information to the ISIMaster device:

- Communications channel configuration and maintenance information
- Most data destined for PrintMaster, ISISlave or storage SoPEC devices. This data is simply relayed by the ISIMaster
- Mapping of virtual communications channels, such as USB endpoints, to ISI destination

12.1.2.1.2 ISIMaster to external host communication

The ISIMaster will need to communicate the following information to the external host:

- Communications channel configuration and maintenance information
- All data originating from the PrintMaster, ISISlave or storage SoPEC devices and destined for the external host. This data is simply relayed by the ISIMaster

12.1.2.1.3 External host to PrintMaster communication

The external host will need to communicate the following information to the PrintMaster device:

- Program code for the PrintMaster
- Compressed page data for the PrintMaster
- Control messages to the PrintMaster
- Tables and static data required for printing e.g. dead nozzle tables, dither matrices etc.
- Authenticatable messages to upgrade the printer's capabilities

12.1.2.1.4 PrintMaster to external host communication

The PrintMaster will need to communicate the following information to the external host:

- Printer status information (i.e. authentication results, paper empty/jammed etc.)
- Dead nozzle information
- Memory buffer status information
- Power management status
- Encrypted SoPEC_id for use in the generation of PRINTER_QA keys during factory programming

12.1.2.1.5 External host to ISISlave communication

All communication between the external host and ISISlave SoPEC devices must be direct (via a dedicated connection between the external host and the ISISlave) or must take place via the

ISIMaster. In the case of a SoPEC ISIMaster it is possible to configure each individual USB endpoint to act as a control channel to an ISISlave SoPEC if desired, although the endpoints will be more usually used to transport data. The external host will need to communicate the following information to ISISlave devices over the comms/ISI:

- Program code for ISISlave SoPEC devices
- Compressed page data for ISISlave SoPEC devices
- Control messages to the ISISlave SoPEC (where a control channel is supported)
- Tables and static data required for printing e.g. dead nozzle tables, dither matrices etc.
- Authenticatable messages to upgrade the printer's capabilities

12.1.2.1.6 ISISlave to external host communication

All communication between the ISISlave SoPEC devices and the external host must take place via the ISIMaster. The ISISlave will need to communicate the following information to the external host over the comms/ISI:

- Responses to the external host's control messages (where a control channel is supported)
- Dead nozzle information from the ISISlave SoPEC.
- Encrypted SoPEC_id for use in the generation of PRINTER_QA keys during factory programming

12.1.2.2 Communication with external USB device

12.1.2.2.1 ISIMaster to External USB device communication

- Communications channel configuration and maintenance information.

12.1.2.2.2 External USB device to ISIMaster communication

- Print data from a function on the external USB device.

12.1.2.3 Communication over ISI

12.1.2.3.1 ISIMaster to PrintMaster communication

The ISIMaster and PrintMaster will often be the same physical device. When they are different devices then the following information needs to be exchanged over the ISI:

- All data from the external host destined for the PrintMaster (see section 12.1.2.1.4).

This data is simply relayed by the ISIMaster

12.1.2.3.2 PrintMaster to ISIMaster communication

The ISIMaster and PrintMaster will often be the same physical device. When they are different devices then the following information needs to be exchanged over the ISI:

- All data from the PrintMaster destined for the external host (see section 12.1.2.1.4).

This data is simply relayed by the ISIMaster

12.1.2.3.3 ISIMaster to ISISlave communication

The ISIMaster may wish to communicate the following information to the ISISlaves:

- All data (including program code such as ISIId enumeration) originating from the external host and destined for the ISISlave (see section 12.1.2.1.5). This data is simply relayed by the ISIMaster
- wake up from sleep mode

12.1.2.3.4 ISISlave to ISIMaster communication

The ISISlave may wish to communicate the following information to the ISIMaster:

- All data originating from the ISISlave and destined for the external host (see section 12.1.2.1.6). This data is simply relayed by the ISIMaster

12.1.2.3.5 PrintMaster to ISISlave communication

When the PrintMaster is not the ISIMaster all ISI communication is done in response to ISI ping packets (see 12.4.4.6). When the PrintMaster is the ISIMaster then it will of course communicate directly with the ISISlaves. The PrintMaster SoPEC may wish to communicate the following information to the ISISlaves:

- Ink status e.g. requests for *dotCount* data i.e. the number of dots in each color fired by the printheads connected to the ISISlaves
- configuration of GPIO ports e.g. for clutch control and lid open detect
- power down command telling the ISISlave to enter sleep mode
- ink cartridge fail information

This list is not complete and the time constraints associated with these requirements have yet to be determined.

In general the PrintMaster may need to be able to:

- send messages to an ISISlave which will cause the ISISlave to return the contents of ISISlave registers to the PrintMaster or
- to program ISISlave registers with values sent by the PrintMaster

This should be under the control of software running on the CPU which writes messages to the ISI/SCB interface.

12.1.2.3.6 ISISlave to PrintMaster communication

ISISlaves may need to communicate the following information to the PrintMaster:

- ink status e.g. *dotCount* data i.e. the number of dots in each color fired by the printheads connected to the ISISlaves
- band related information e.g. finished band interrupts
- page related information i.e. buffer underrun, page finished interrupts
- MMU security violation interrupts
- GPIO interrupts and status e.g. clutch control and lid open detect
- printhead temperature
- printhead dead nozzle information from SoPEC printhead nozzle tests
- power management status

This list is not complete and the time constraints associated with these requirements have yet to be determined.

As the ISI is an insecure interface commands issued over the ISI should be of limited capability e.g. only limited register writes allowed. The software protocol needs to be constructed with this in mind. In general ISISlaves may need to return register or status messages to the PrintMaster or ISIMaster. They may also need to indicate to the PrintMaster or ISIMaster that a particular interrupt

has occurred on the ISISlave. This should be under the control of software running on the CPU which writes messages to the ISI block.

12.1.2.3.7 ISISlave to ISISlave communication

The amount of information that will need to be communicated between ISISlaves will vary considerably depending on the printer configuration. In some systems ISISlave devices will only need to exchange small amounts of control information with each other while in other systems (such as those employing a storage SoPEC or extra USB connection) large amounts of compressed page data may be moved between ISISlaves. Scenarios where ISISlave to ISISlave communication is required include: (a) when the PrintMaster is not the ISIMaster, (b) QA Chip ink usage protocols, (c) data transmission from data storage SoPECs, (d) when there are multiple external host connections supplying data to the printer.

12.1.3 SCB Block Diagram

The SCB consists of four main sub-blocks, as shown in the basic block diagram of Figure 28.

12.1.4 Definitions of I/Os

The toplevel I/Os of the SCB are listed in Table 32. A more detailed description of their functionality will be given in the relevant sub-block sections.

Table 32. SCB I/O

Port name	s	I/O	Description
Clocks and Resets			
prst_n	1	In	System reset signal. Active low.
Pclk	1	In	System clock.
usbclk	1	In	48MHz clock for the USB device and host cores. The cores also require a 12MHz clock, which will be generated locally by dividing the 48MHz clock by 4.
isi_cpr_reset_n	1	Out	Signal from the ISI indicating that ISI activity has been detected while in sleep mode and so the chip should be reset. Active low.
usbd_cpr_reset_n	1	Out	Signal from the USB device that a USB reset has occurred. Active low.
USB device IO transceiver signals			
usbd_ts	1	Out	USB device IO transceiver (BUSB2_PM) driver three-state control. Active high enable.
usbd_a	1	Out	USB device IO transceiver (BUSB2_PM) driver data input.
usbd_se0	1	Out	USB device IO transceiver (BUSB2_PM) single-ended zero input. Active high.

usbd_zp	1	In	USB device IO transceiver (BUSB2_PM) D+ receiver output.
usbd_zm	1	In	USB device IO transceiver (BUSB2_PM) D- receiver output.
usbd_z	1	In	USB device IO transceiver (BUSB2_PM) differential receiver output.
usbd_pull_up_en	1	Out	USB device pull-up resistor enable. Switches power to the external pull-up resistor, connected to the D+ line that is required for device identification to the USB. Active high.
usbd_vbus_sense	1	In	USB device VBUS power sense. Used to detect power on VBUS. NOTE: The IBM Cu11 PADS are 3.3V, VBUS is 5V. An external voltage conversion will be necessary, e.g. resistor divider network. Active high.
USB host IO transceiver signals			
usbh_ts	1	Out	USB host IO transceiver (BUSB2_PM) driver three-state control. Active high enable
usbh_a	1	Out	USB host IO transceiver (BUSB2_PM) driver data input.
usbh_se0	1	Out	USB host IO transceiver (BUSB2_PM) single-ended zero input. Active high.
usbh_zp	1	In	USB host IO transceiver (BUSB2_PM) D+ receiver output.
usbh_zm	1	In	USB host IO transceiver (BUSB2_PM) D- receiver output.
usbh_z	1	In	USB host IO transceiver (BUSB2_PM) differential receiver output.
usbh_over_current	1	In	USB host port power over current indicator. Active high.
usbh_power_en	1	Out	USB host VBUS power enable. Used for port power switching. Active high.
CPU Interface			
cpu_adr[n:2]	n-1	In	CPU address bus.
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
scb_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as

			follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_scb_sel	1	In	Block select from the CPU. When *cpu_scb_sel* is high both *cpu_adr* and *cpu_dataout* are valid
scb_cpu_rdy	1	Out	Ready signal to the CPU. When *scb_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the SCB and for a read cycle this means the data on *scb_cpu_data* is valid.
scb_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
scb_cpu_debug_valid	1	Out	Signal indicating that the data currently on *scb_cpu_data* is valid debug data
Interrupt signals			
dma_icu_irq	1	Out	DMA interrupt signal to the interrupt controller block.
isi_icu_irq	1	Out	ISI interrupt signal to the interrupt controller block.
usb_icu_irq[1:0]	2	Out	USB host and device interrupt signals to the ICU. Bit 0 - USB Host interrupt Bit 1 - USB Device interrupt
DIU interface			
scb_diu_wadr[21:5]	17	Out	Write address bus to the DIU
scb_diu_data[63:0]	64	Out	Data bus to the DIU.
scb_diu_wreq	1	Out	Write request to the DIU
diu_scb_wack	1	In	Acknowledge from the DIU that the write request was accepted.
scb_diu_wvalid	1	Out	Signal from the SCB to the DIU indicating that the data currently on the *scb_diu_data[63:0]* bus is valid
scb_diu_wmask[7:0]	7	Out	Byte aligned write mask. A "1" in a bit field of "*scb_diu_wmask[7:0]*" means that the corresponding byte will be written to DRAM.

scb_diu_rreq	1	Out	Read request to the DIU.
scb_diu_radr[21:5]	17	Out	Read address bus to the DIU
diu_scb_rack	1	In	Acknowledge from the DIU that the read request was accepted.
diu_scb_rvalid	1	In	Signal from the DIU to the SCB indicating that the data currently on the *diu_data[63:0]* bus is valid
diu_data[63:0]	64	In	Common DIU data bus.
GPIO interface			
isi_gpio_dout[3:0]	4	Out	ISI output data to GPIO pins
isi_gpio_e[3:0]	4	Out	ISI output enable to GPIO pins
gpio_isi_din[3:0]	4	In	Input data from GPIO pins to ISI

12.1.5 SCB Data Flow

A logical view of the SCB is shown in Figure 29, depicting the transfer of data within the SCB.

12.2 USBD (USB DEVICE SUB-BLOCK)

12.2.1 Overview

The FS USB device controller core and associated SCB logic are referred to as the USB Device (USBD).

A SoPEC printer has FS USB device capability to facilitate communication between an external USB host and a SoPEC printer. The USBD is self-powered. It connects to an external USB host via a dedicated USB interface on the SoPEC printer, comprising a USB connector, the necessary discretes for USB signalling and the associated SoPEC ASIC I/Os.

The FS USB device core will be third party IP from Synopsys: TymeWare™ USB1.1 Device Controller (UDCVCI). Refer to the UDCVCI User Manual [20] for a description of the core.

The device core does not support LS USB operation. Control and bulk transfers are supported by the device. Interrupt transfers are not considered necessary because the required interrupt-type functionality can be achieved by sending query messages over the control channel on a scheduled basis. There is no requirement to support isochronous transfers.

The device core is configured to support 6 USB endpoints (EPs): the default control EP (EP0), 4 bulk OUT EPs (EP1, EP2, EP3, EP4) and 1 bulk IN EP (EP5). It should be noted that the direction of each EP is with respect to the USB host, i.e. *IN* refers to data transferred to the external host and *OUT* refers to data transferred from the external host. The 4 bulk OUT EPs will be used for the transfer of data from the external host to SoPEC, e.g. compressed page data, program data or control messages. Each bulk OUT EP can be mapped on to any target destination in a multi-SoPEC system, via the SCB Map configuration registers. The bulk IN EP is used for the transfer of data from SoPEC to the external host, e.g. a print image downloaded from a digital camera that requires processing on the external host system. Any feedback data will be returned to the external host on EP0, e.g. status information.

The device core does not provide internal buffering for any of its EPs (with the exception of the 8 byte setup data payload for control transfers). All EP buffers are provided in the SCB. Buffers will be

grouped according to EP direction and associated packet destination. The SCB Map configuration registers contain a *DestISIId* and *DestISISubId* for each OUT EP, defining their EP mapping and therefore their packet destination. Refer to section Section 12.4 ISI (Inter SoPEC Interface Sub-block) for further details on *ISIId* and *ISISubId*. Refer to section Section 12.5 CTRL (Control Sub-block) for further details on the mapping of OUT EPs.

12.2.2 USBD effective bandwidth

The effective bandwidth between an external USB host and the printer will be influenced by:

- Amount of activity from other devices that share the USB with the printer.
- Throughput of the device controller core.
- EP buffering implementation.
- Responsiveness of the external host system CPU in handling USB interrupts.

To maximize bandwidth to the printer it is recommended that no other devices are active on the USB between the printer and the external host. If the printer is connected to a HS USB external host or hub it may limit the bandwidth available to other devices connected to the same hub but it would not significantly affect the bandwidth available to other devices upstream of the hub. The EP buffering should not limit the USB device core throughput, under normal operating conditions. Used in the recommended configuration, under ideal operating conditions, it is expected that an effective bandwidth of 8-9 Mbit/s will be achieved with bulk transfers between the external host and the printer.

12.2.3 IN EP packet buffer

The IN EP packet buffer stores packets originating from the LEON CPU that are destined for transmission over the USB to the external USB host. CPU writes to the buffer are 32 bits wide. USB device core reads from the buffer 32 bits wide.

128 bytes of local memory are required in total for EP0-IN and EP5-IN buffering. The IN EP buffer is a single, 2-port local memory instance, with a dedicated read port and a dedicated write port. Both ports are 32 bits wide. Each IN EP has a dedicated 64 byte packet location available in the memory array to buffer a single USB packet (maximum USB packet size is 64 bytes). Each individual 64 byte packet location is structured as 16 x 32 bit words and is read/written in a FIFO manner.

When the device core reads a packet entry from the IN EP packet buffer, the buffer must retain the packet until the device core performs a status write, informing the SCB that the packet has been accepted by the external USB host and can be flushed. The CPU can therefore only write a single packet at a time to each IN EP. Any subsequent CPU write request to a buffer location containing a valid packet will be refused, until that packet has been successfully transmitted.

12.2.4 OUT EP packet buffer

The OUT EP packet buffer stores packets originating from the external USB host that are destined for transmission over DMAChannel0, DMAChannel1 or the ISI. The SCB control logic is responsible for routing the OUT EP packets from the OUT EP packet buffer to DMA or to the ISITx Buffer, based on the SCB Map configuration register settings. USB core writes to the buffer are 32 bits wide. DMA and ISI associated reads from the buffer are both 64 bits wide.

512 bytes of local memory are required in total for EP0-OUT, EP1-OUT, EP2-OUT, EP3-OUT and EP4-OUT buffering. The OUT EP packet buffer is a single, 2-port local memory instance, with a dedicated read port and a dedicated write port. Both ports are 64 bits wide. Byte enables are used for the 32 bit wide USB device core writes to the buffer. Each OUT EP can be mapped to DMAChannel0, DMAChannel1 or the ISI.

The OUT EP packet buffer is partitioned accordingly, resulting in three distinct packet FIFOs:

- USBDDMA0FIFO, for USB packets destined for DMAChannel0 on the local SoPEC.
- USBDDMA1FIFO, for USB packets destined for DMAChannel1 on the local SoPEC.
- USBDISIFIFO, for USB packets destined for transmission over the ISI.

12.2.4.1 USBDDMAnFIFO

This description applies to USBDDMA0FIFO and USBDDMA1FIFO, where '*n*' represents the respective DMA channel, i.e. *n*=0 for USBDDMA0FIFO, *n*=1 for USBDDMA1FIFO.

USBDDMAnFIFO services any EPs mapped to DMAChannel*n* on the local SoPEC device. This implies that a packet originating from an EP with an associated *ISIId* that matches the local SoPEC *ISIId* and an *ISISubId=n* will be written to USBDDMAnFIFO, if there is space available for that packet.

USBDDMAnFIFO has a capacity of 2 x 64 byte packet entries, and can therefore buffer up to 2 USB packets. It can be considered as a 2 packet entry FIFO. Packets will be read from it in the same order in which they were written, i.e. the first packet written will be the first packet read and the second packet written will be the second packet read. Each individual 64 byte packet location is structured as 8 x 64 bit words and is read/written in a FIFO manner.

The USBDDMAnFIFO has a write granularity of 64 bytes, to allow for the maximum USB packet size. The USBDDMAnFIFO will have a read granularity of 32 bytes to allow for the DMA write access bursts of 4 x 64 bit words, i.e. the DMA Manager will read 32 byte chunks at a time from the USBDDMAnFIFO 64byte packet entries, for transfer to the DIU.

It is conceivable that a packet which is not a multiple 32 bytes in size may be written to the USBDDMAnFIFO. When this event occurs, the DMA Manager will read the contents of the remaining address locations associated with the 32 byte chunk in the USBDDMAnFIFO, transferring the packet plus whatever data is present in those locations, resulting in a 32 byte packet (a burst of 4 x 64 bit words) transfer to the *DIU*.

The DMA channels should achieve an effective bandwidth of 160 Mbits/sec (1 bit/cycle) and should never become blocked, under normal operating conditions. As the USB bandwidth is considerably less, a 2 entry packet FIFO for each DMA channel should be sufficient.

12.2.4.2 USBDISIFIFO

USBDISIFIFO services any EPs mapped to ISI. This implies that a packet originating from an EP with an associated *ISIId* that does not match the local SoPEC *ISIId* will be written to USBDISIFIFO if there is space available for that packet.

USBDISIFIFO has a capacity of 4 x 64 byte packet entries, and can therefore buffer up to 4 USB packets. It can be considered as a 4 packet entry FIFO. Packets will be read from it in the same order in which they were written, i.e. the first packet written will be the first packet read and the

second packet written will be the second packet read, etc. Each individual 64 byte packet location is structured as 8 x 64 bit words and is read/written in a FIFO manner.

The ISI long packet format will be used to transfer data across the ISI. Each ISI long packet data payload is 32 bytes. The USBDISIFIFO has a write granularity of 64 bytes, to allow for the maximum USB packet size. The USBDISIFIFO will have a read granularity of 32 bytes to allow for the ISI packet size, i.e. the SCB will read 32 byte chunks at a time from the USBDISIFIFO 64byte packet entries, for transfer to the ISI.

It is conceivable that a packet which is not a multiple 32 bytes in size may be written to the USBDISIFIFO, either intentionally or due to a software error. A maskable interrupt per EP is provided to flag this event. There will be 2 options for dealing with this scenario on a per EP basis:

- Discard the packet.
- Read the contents of the remaining address locations associated with the 32 byte chunk in the USBDISIFIFO, transferring the irregular size packet plus whatever data is present in those locations, resulting in a 32 byte packet transfer to the *ISITxBuffer*.

The ISI should achieve an effective bandwidth of 100 Mbits/sec (4 wire configuration). It is possible to encounter a number of retries when transmitting an ISI packet and the LEON CPU will require access to the ISI transmit buffer. However, considering the relatively low bandwidth of the USB, a 4 packet entry FIFO should be sufficient.

12.2.5 Wake-up from sleep mode

The SoPEC will be placed in sleep mode after a suspend command is received by the USB device core. The USB device core will continue to be powered and clocked in sleep mode. A USB reset, as opposed to a device resume, will be required to bring SoPEC out of its sleep state as the sleep state is hoped to be logically equivalent to the power down state.

The USB reset signal originating from the USB controller will be propagated to the CPR (as *usb_cpr_reset_n*) if the *USBWakeupEnable* bit of the *WakeupEnable* register (see Table) has been set. The *USBWakeupEnable* bit should therefore be set just prior to entering sleep mode.

There is a scenario that would require SoPEC to initiate a USB remote wake-up (i.e. where SoPEC signals resume to the external USB host after being suspended by the external USB host). A digital camera (or other supported external USB device) could be connected to SoPEC via the internal SoPEC USB host controller core interface. There may be a need to transfer data from this external USB device, via SoPEC, to the external USB host system for processing. If the USB connecting the external host system and SoPEC was suspended, then SoPEC would need to initiate a USB remote wake-up.

12.2.6 Implementation

12.2.6.1 USBD Sub-block Partition

* Block diagram
* Definition of I/Os

12.2.6.2 USB Device IP Core

12.2.6.3 PVCI Target

12.2.6.4 IN EP Buffer

12.2.6.5 OUT EP Buffer

12.3 USBH (USB HOST SUB-BLOCK)

12.3.1 Overview

The SoPEC USB Host Controller (HC) core, associated SCB logic and associated SoPEC ASIC I/Os are referred to as the USB Host (USBH).

A SoPEC printer has FS USB host capability, to facilitate communication between an external USB device and a SoPEC printer. The USBH connects to an external USB device via a dedicated USB interface on the SoPEC printer, comprising a USB connector, the necessary discretes for USB signalling and the associated SoPEC ASIC I/Os.

The FS USB HC core are third party IP from Synopsys: DesignWare^{R} USB1.1 OHCI Host Controller with PVCI (UHOSTC_PVCI). Refer to the UHOSTC_PVCI User Manual [18] for details of the core. Refer to the Open Host Controller Interface (OHCI) Specification Release [19] for details of OHCI operation.

The HC core supports Low-Speed (LS) USB devices, although compatible external USB devices are most likely to be FS devices. It is expected that communication between an external USB device and a SoPEC printer will be achieved with control and bulk transfers. However, isochronous and interrupt transfers are also supported by the HC core.

There will be 2 communication channels between the Host Controller Driver (HCD) software running on the LEON CPU and the HC core:

- OHCI operational registers in the HC core. These registers are control, status, list pointers and a pointer to the Host Controller Communications Area (HCCA) in shared memory. A target Peripheral Virtual Component Interface (PCVI) on the HC core will provide LEON with direct read/write access to the operational registers. Refer to the OHCI Specification for details of these registers.
- HCCA in SoPEC eDRAM. An initiator Peripheral Virtual Component Interface (PCVI) on the HC core will provide the HC with DMA read/write access to an address space in eDRAM. The HCD running on LEON will have read/write access to the same address space. Refer to the OHCI Specification for details of the HCCA.

The target PVCI interface is a 32 bit word aligned interface, with byte enables for write access. All read/ write access to the target PVCI interface by the LEON CPU will be 32 bit word aligned. The byte enables will not be used, as all registers will be read and written as 32 bit words.

The initiator PVCI interface is a 32 bit word aligned interface with byte enables for write access. All DMA read/write accesses are 256 bit word aligned, in bursts of 4 x 64 bit words. As there is no guarantee that the read/write requests from the HC core will start at a 256 bit boundary or be 256 bits long, it is necessary to provide 8 byte enables for each of the 64 bit words in a write burst form the HC core to DMA. The signal *scb_diu_wmask* serves this purpose.

Configuration of the HC core will be performed by the HCD.

12.3.2 Read/Write Buffering

The HC core maximum burst size for a read/write access is 4 x 32 bit words. This implies that the minimum buffering requirements for the HC core will be a 1 entry deep address register and a 4

entry deep data register. It will be necessary to provide data and address mapping functionality to convert the 4 x 32 bit word HC core read/write bursts into 4 x 64 bit word DMA read/write bursts. This will meet the minimum buffering requirements.

12.3.3 USBH effective bandwidth

The effective bandwidth between an external USB device and a SoPEC printer will be influenced by:

- Amount of activity from other devices that share the USB with the external USB device.
- Throughput of the HC core.
- HC read/write buffering implementation.
- Responsiveness of the LEON CPU in handling USB interrupts.

Effective bandwidth between an external USB device and a SoPEC printer is not an issue. The primary application of this connectivity is the download of a print image from a digital camera. Printing speed is not important for this type of print operation. However, to maximize bandwidth to the printer it is recommended that no other devices are active on the USB between the printer and the external USB device. The HC read/write buffering in the SCB should not limit the USB HC core throughput, under normal operating conditions.

Used in the recommended configuration, under ideal operating conditions, it is expected that an effective bandwidth of 8-9 Mbit/s will be achieved with bulk transfers between the external USB device and the SoPEC printer.

12.3.4 Implementation

12.3.5 USBH Sub-block Partition

* USBH Block Diagram

* Definition of I/Os.

12.3.5.1 USB Host IP Core

12.3.5.2 PVCI Target

12.3.5.3 PVCI Initiator

12.3.5.4 Read/Write Buffer

12.4 ISI (INTER SOPEC INTERFACE SUB-BLOCK)

12.4.1 Overview

The ISI is utilised in all system configurations requiring more than one SoPEC. An example of such a system which requires four SoPECs for duplex A3 printing and an additional SoPEC used as a storage device is shown in Figure 27.

The ISI performs much the same function between an ISISlave SoPEC and the ISIMaster as the USB connection performs between the ISIMaster and the external host. This includes the transfer of all program data, compressed page data and message (i.e. commands or status information) passing between the ISIMaster and the ISISlave SoPECs. The ISIMaster initiates all communication with the ISISlaves.

12.4.2 ISI Effective Bandwidth

The ISI will need to run at a speed that will allow error free transmission on the PCB while minimising the buffering and hardware requirements on SoPEC. While an ISI speed of 10 Mbit/s is

adequate to match the effective FS USB bandwidth it would limit the system performance when a high-speed connection (e.g. USB2.0, IEEE1394) is used to attach the printer to the PC. Although they would require the use of an extra ISI-Bridge chip such systems are envisaged for more expensive printers (compared to the low-cost basic SoPEC powered printers that are initially being targeted) in the future.

An ISI line speed (i.e. the speed of each individual ISI wire) of 32 Mbit/s is therefore proposed as it will allow ISI data to be over-sampled 5 times (at a *pclk* frequency of 160MHz). The total bandwidth of the ISI will depend on the number of pins used to implement the interface. The ISI protocol will work equally well if 2 or 4 pins are used for transmission/reception. The *ISINumPins* register is used to select between a 2 or 4 wire ISI, giving peak raw bandwidths of 64 Mbit/s and 128 Mbit/s respectively. Using either a 2 or 4 wire ISI solution would allow the movement of data in to and out of a storage SoPEC (as described in 12.1.1.4 above), which is the most bandwidth hungry ISI use, in a timely fashion.

The *ISINumPins* register is used to select between a 2 or 4 wire ISI. A 2 wire ISI is the default setting for *ISINumPins* and this may be changed to a 4 wire ISI after initial communication has been established between the ISIMaster and all ISISlaves. Software needs to ensure that the switch from 2 to 4 wires is handled in a controlled and coordinated fashion so that nothing is transmitted on the ISI during the switch over period.

The maximum effective bandwidth of a two wire ISI, after allowing for protocol overheads and bus turnaround times, is expected to be approx. 50 Mbit/s.

12.4.3 ISI Device Identification and Enumeration

The *ISIMasterSel* bit of the *ISICntrl* register (see section Table) determines whether a SoPEC is an ISIMaster (*ISIMasterSel* = 1), or an ISISlave (*ISIMasterSel* = 0).

SoPEC defaults to being an ISISlave (*ISIMasterSel* = 0) after a power-on reset - i.e. it will not transmit data on the ISI without first receiving a ping. If a SoPEC's *ISIMasterSel* bit is changed to 1, then that SoPEC will become the ISIMaster, transmitting data without requiring a ping, and generating pings as appropriately programmed.

ISIMasterSel can be set to 1 explicitly by the CPU writing directly to the *ISICntrl* register.

ISIMasterSel can also be automatically set to 1 when activity occurs on any of USB endpoints 2-4 and the *AutoMasterEnable* bit of the *ISICntrl* register is also 1 (the default reset condition). Note that if *AutoMasterEnable* is 0, then activity on USB endpoints 2-4 will not result in *ISIMasterSel* being set to 1. USB endpoints 2-4 are chosen for the automatic detection since the power-on-reset condition has USB endpoints 0 and 1 pointing to ISIId 0 (which matches the local SoPEC's ISIId after power-on reset). Thus any transmission on USB endpoints 2-4 indicate a desire to transmit on the ISI which would usually indicate ISIMaster status. The automatic setting of *ISIMasterSel* can be disabled by clearing *AutoMasterEnable*, thereby allowing the SoPEC to remain an ISISlave while still making use of the USB endpoints 2-4 as external destinations.

Thus the setting of a SoPEC being ISIMaster or ISISlave can be completely under software control, or can be completely automatic.

The ISIId is established by software downloaded over the ISI (in broadcast mode) which looks at the input levels on a number of GPIO pins to determine the ISIId. For any given printer that uses a multi-SoPEC configuration it is expected that there will always be enough free GPIO pins on the ISISlaves to support this enumeration mechanism.

12.4.4 ISI protocol

The ISI is a serial interface utilizing a 2/4 wire half-duplex configuration such as the 2-wire system shown in Figure 30 below. An ISIMaster must always be present and a variable number of ISISlaves may also be on the ISI bus. The ISI protocol supports up to 14 addressable slaves, however to simplify electrical issues the ISI drivers need only allow for 5-6 ISI devices on a particular ISI bus. The ISI bus enables broadcasting of data, ISIMaster to ISISlave communication, ISISlave to ISIMaster communication and ISISlave to ISISlave communication. Flow control, error detection and retransmission of errored packets is also supported. ISI transmission is asynchronous and a *Start* field is present in every transmitted packet to ensure synchronization for the duration of the packet.

To maximize the effective ISI bandwidth while minimising pin requirements a half-duplex interleaved transmission scheme is used. Figure 31 below shows how a 16-bit word is transmitted from an ISIMaster to an ISISlave over a 2-wire ISI bus. Since data will be interleaved over the wires and a 4-wire ISI is also supported, all ISI packets should be a multiple of 4 bits.

All ISI transactions are initiated by the ISIMaster and every non-broadcast data packet needs to be acknowledged by the addressed recipient. An ISISlave may only transmit when it receives a ping packet (see section 12.4.4.6) addressed to it. To avoid bus contention all ISI devices must wait *ISITurnAround* bit-times (5 *pclk* cycles per bit) after detecting the end of a packet before transmitting a packet (assuming they are required to transmit). All non-transmitting ISI devices must tristate their Tx drivers to avoid line contention. The ISI protocol is defined to avoid devices driving out of order (e.g. when an ISISlave is no longer being addressed). As the ISI uses standard I/O pads there is no physical collision detection mechanism.

There are three types of ISI packet: a long packet (used for data transmission), a ping packet (used by the ISIMaster to prompt ISISlaves for packets) and a short packet (used to acknowledge receipt of a packet). All ISI packets are delineated by a *Start* and *Stop* fields and transmission is atomic i.e. an ISI packet may not be split or halted once transmission has started.

12.4.4.1 ISI transactions

The different types of ISI transactions are outlined in Figure 32 below. As described later all NAKs are inferred and ACKs are not addressed to any particular ISI device.

12.4.4.2 Start Field Description

The *Start* field serves two purposes: To allow the start of a packet be unambiguously identified and to allow the receiving device synchronise to the data stream. The symbol, or data value, used to identify a *Start* field must not legitimately occur in the ensuing packet. Bit stuffing is used to guarantee that the *Start* symbol will be unique in any valid (i.e. error free) packet. The ISI needs to see a valid Start symbol before packet reception can commence i.e. the receive logic constantly looks for a *Start* symbol in the incoming data and will reject all data until it sees a *Start* symbol.

Furthermore if a *Start* symbol occurs (incorrectly) during a data packet it will be treated as the start of a new packet. In this case the partially received packet will be discarded.

The data value of the *Start* symbol should guarantee that an adequate number of transitions occur on the physical ISI lines to allow the receiving ISI device to determine the best sampling window for the transmitted data. The *Start* symbol should also be sufficiently long to ensure that the bit stuffing overhead is low but should still be short enough to reduce its own contribution to the packet overhead. A *Start* symbol of b01010101 is therefore used as it is an effective compromise between these constraints.

Each SoPEC in a multi-SoPEC system will derive its system clock from a unique (i.e. one per SoPEC) crystal. The system clocks of each device will drift relative to each other over any period of time. The system clocks are used for generation and sampling of the ISI data. Therefore the sampling window can drift and could result in incorrect data values being sampled at a later point in time. To overcome this problem the ISI receive circuitry tracks the sampling window against the incoming data to ensure that the data is sampled in the centre of the bit period.

12.4.4.3 Stop Field Description

A 1 bit-time *Stop* field of b1 per ISI line ensures that all ISI lines return to the high state before the next packet is transmitted. The stop field is driven on to each ISI line simultaneously, i.e. b11 for a 2-wire ISI and b1111 for a 4-wire ISI would be interleaved over the respective ISI lines. Each ISI line is driven high for 1 bit-time. This is necessary because the first bit of the *Start* field is b0.

12.4.4.4 Bit Stuffing

This involves the insertion of bits into the bitstream at the transmitting SoPEC to avoid certain data patterns. The receiving SoPEC will strip these inserted bits from the bitstream.

Bit-stuffing will be performed when the *Start* symbol appears at a location other than the start field of any packet, i.e. when the bit pattern b0101010 occurs at the transmitter, a 0 will be inserted to escape the *Start* symbol, resulting in the bit pattern b01010100. Conversely, when the bit pattern b0101010 occurs at the receiver, if the next bit is a '0' it will be stripped, if it is a '1' then a *Start* symbol is detected.

If the frequency variations in the quartz crystal were large enough, it is conceivable that the resultant frequency drift over a large number of consecutive 1s or 0s could cause the receiving SoPEC to loose synchronisation.[6] The quartz crystal that will be used in SoPEC systems is rated for 32MHz @ 100ppm. In a multi-SoPEC system with a 32MHz+100ppm crystal and a 32MHz-100ppm crystal, it would take approximately 5000 *pclk* cycles to cause a drift of 1 pclk cycle. This means that we would only need to bit-stuff somewhere before 1000 ISI bits of consecutive 1s or consecutive 0s, to ensure adequate synchronization. As the maximum number of bits transmitted per ISI line in a packet is 145, it should not be

---

[6]Current max packet size ~= 290 bits = 145 bits per ISI line (on a 2 wire ISI) = 725 160MHz cycles. Thus the *pclk*s in the two communicating ISI devices should not drift by more than one cycle in 725 i.e. 1379 ppm. Careful analysis of the crystal, PLL and oscillator specs and the sync detection circuit is needed here to ensure our solution is robust.

necessary to perform bit-stuffing for consecutive 1s or 0s. We may wish to constrain the spec of *xtalin* and also *xtalin* for the ISI-Bridge chip to ensure the ISI cannot drift out of sync during packet reception.

Note that any violation of bit stuffing will result in the *RxFrameErrorSticky* status bit being set and the incoming packet will be treated as an errored packet.

12.4.4.5 ISI Long Packet

The format of a long ISI packet is shown in Figure 33 below. Data may only be transferred between ISI devices using a long packet as both the short and ping packets have no payload field. Except in the case of a broadcast packet, the receiving ISI device will always reply to a long packet with an explicit ACK (if no error is detected in the received packet) or will not reply at all (e.g. an error is detected in the received packet), leaving the transmitter to infer a NAK. As with all ISI packets the bitstream of a long packet is transmitted with its lsb (the leftmost bit in Figure 33) first. Note that the total length (in bits) of an ISI long packet differs slightly between a 2 and 4-wire ISI system due to the different number of bits required for the *Start* and *Stop* fields.

All long packets begin with the *Start* field as described earlier. The *PktDesc* field is described in Table 33.

Table 33. PktDesc field description

Bit	Description
0:1	00 - Long packet 01 - Reserved 10 - Ping packet 11 – Reserved
2	Sequence bit value. Only valid for long packets. See section 12.4.4.9 for a description of sequence bit operation

Any ISI device in the system may transmit a long packet but only the ISIMaster may initiate an ISI transaction using a long packet. An ISISlave may only send a long packet in reply to a ping message from the ISIMaster. A long packet from an ISISlave may be addressed to any ISI device in the system.

The *Address* field is straightforward and complies with the ISI naming convention described in section 12.5.

The payload field is exactly what is in the transmit buffer of the transmitting ISI device and gets copied into the receive buffer of the addressed ISI device(s). When present the payload field is always 256 bits.

To ensure strong error detection a 16-bit CRC is appended.

12.4.4.6 ISI Ping Packet

The ISI ping packet is used to allow ISISlaves to transmit on the ISI bus. As can be seen from Figure 34 below the ping packet can be viewed as a special case of the long packet. In other words it is a long packet without any payload. Therefore the *PktDesc* field is the same as a long packet *PktDesc*, with the exception of the sequence bit, which is not valid for a ping packet. Both the *ISISubId* and the sequence bit are fixed at 1 for all ping packets. These values were chosen to maximize the hamming distance from an ACK symbol and to minimize the likelihood of bit stuffing.

The *ISISubId* is unused in ping packets because the ISIMaster is addressing the ISI device rather than one of the DMA channels in the device. The ISISlave may address any *ISIId.ISISubId* in response if it wishes. The ISISlave will respond to a ping packet with either an explicit ACK (if it has nothing to send), an inferred NAK (if it detected an error in the ping packet) or a long packet (containing the data it wishes to send). Note that inferred NAKs do not result in the retransmission of a ping packet. This is because the ping packet will be retransmitted on a predetermined schedule (see 12.4.4.11 for more details).

An ISISlave should never respond to a ping message to the broadcast *ISIId* as this must have been sent in error. An ISI ping packet will never be sent in response to any packet and may only originate from an ISIMaster.

12.4.4.7 ISI Short Packet

The ISI short packet is only 17 bits long, including the *Start* and *Stop* fields. A value of b11101011 is proposed for the ACK symbol. As a 16-bit CRC is inappropriate for such a short packet it is not used. In fact there is only one valid value for a short ACK packet as the *Start*, ACK and *Stop* symbols all have fixed values. Short packets are only used for acknowledgements (i.e. explicit ACKs). The format of a short ISI packet is shown in Figure 35 below. The ACK value is chosen to ensure that no bit stuffing is required in the packet and to minimize its hamming distance from ping and long ISI packets.

12.4.4.8 Error Detection and Retransmission

The 16-bit CRC will provide a high degree of error detection and the probability of transmission errors occurring is very low as the transmission channel (i.e. PCB traces) will have a low inherent bit error rate. The number of undetected errors should therefore be minute.

The HDLC standard CRC-16 (i.e. $G(x) = x^{16} + x^{12} + x^5 + 1$) is to be used for this calculation, which is to be performed serially. It is calculated over the entire packet (excluding the *Start* and *Stop* fields).

A simple retransmission mechanism frees the CPU from getting involved in error recovery for most errors because the probability of a transmission error occurring more than once in succession is very, very low in normal circumstances.

After each non-short ISI packet is transmitted the transmitting device will open a reply window. The size of the reply window will be *ISIShortReplyWin* bit times when a short packet is expected in reply, i.e. the size of a short packet, allowing for worst case bit stuffing, bus turnarounds and timing differences. The size of the reply window will be *ISILongReplyWin* bit times when a long packet is expected in reply, i.e. this will be the max size of a long packet, allowing for worst case bit stuffing, bus turnarounds and timing differences. In both cases if an ACK is received the window will close and another packet can be transmitted but if an ACK is not received then the full length of the window must be waited out.

As no reply should be sent to a broadcast packet, no reply window should be required however all other long packets open a reply window in anticipation of an ACK. While the desire is to minimize the time between broadcast transmissions the simplest solution should be employed. This would imply the same size reply window as other long packets.

When a packet has been received without any errors the receiving ISI device must transmit its acknowledge packet (which may be either a long or short packet) before the reply window closes. When detected errors do occur the receiving ISI device will not send any response. The transmitting ISI device interprets this lack of response as a NAK indicating that errors were detected in the transmitted packet or that the receiving device was unable to receive the packet for some reason (e.g. its buffers are full). If a long packet was transmitted the transmitting ISI device will keep the transmitted packet in its transmit buffer for retransmission. If the transmitting device is the ISIMaster it will retransmit the packet immediately while if the transmitting device is an ISISlave it will retransmit the packet in response to the next ping it receives from the ISIMaster.

The transmitting ISI device will continue retransmitting the packet when it receives a NAK until it either receives an ACK or the number of retransmission attempts equals the value of the *NumRetries* register. If the transmission was unsuccessful then the transmitting device sets the *TxErrorSticky* bit in its *ISIIntStatus* register. The receiving device also sets the *RxErrorSticky* bit in its *ISIIntStatus* register whenever it detects a CRC error in an incoming packet and is not required to take any further action, as it is up to the transmitting device to detect and rectify the problem. The *NumRetries* registers in all ISI devices should be set to the same value for consistent operation.

Note that successful transmission or reception of ping packets do not affect retransmission operation.

Note that a transmit error will cause the ISI to stop transmitting. CPU intervention will be required to resolve the source of the problem and to restart the ISI transmit operation. Receive errors however do not affect receive operation and they are collected to facilitate problem debug and to monitor the quality of the ISI physical channel. Transmit or receive errors should be extremely rare and their occurrence will most likely indicate a serious problem.

Note that broadcast packets are never acknowledged to avoid contention on the common ISI lines. If an ISISlave detects an error in a broadcast packet it should use the message passing mechanism described earlier to alert the ISIMaster to the error if it so wishes.

12.4.4.9 Sequence Bit Operation

To ensure that communication between transmitting and receiving ISI devices is correctly ordered a sequence bit is included in every long packet to keep both devices in step with each other. The sequence bit field is a constant for short or ping packets as they are not used for data transmission. In addition to the transmitted sequence bit all ISI devices keep two local sequence bits, one for each *ISISubId*. Furthermore each ISI device maintains a transmit sequence bit for each *ISIId* and *ISISubId* it is in communication with. For packets sourced from the external host (via USB) the transmit sequence bit is contained in the relevant *USBEPnDest* register while for packets sourced from the CPU the transmit sequence bit is contained in the *CPUISITxBuffCntrl* register. The sequence bits for received packets are stored in *ISISubId0Seq* and *ISISubId1Seq* registers. All ISI devices will initialize their sequence bits to 0 after reset. It is the responsibility of software to ensure that the sequence bits of the transmitting and receiving ISI devices are correctly initialized each time a new source is selected for any *ISIId.ISISubId* channel.

Sequence bits are ignored by the receiving ISI device for broadcast packets. However the broadcasting ISI device is free to toggle the sequence in the broadcast packets since they will not affect operation. The SCB will do this for all USB source data so that there is no special treatment for the sequence bit of a broadcast packet in the transmitting device. CPU sourced broadcasts will have sequence bits toggled at the discretion of the program code.

Each SoPEC may also ignore the sequence bit on either of its ISISubId channels by setting the appropriate bit in the *ISISubIdSeqMask* register. The sequence bit should be ignored for ISISubId channels that will carry data that can originate from more than one source and is self ordering e.g. control messages.

A receiving ISI device will toggle its sequence bit addressed by the ISISubId only when the receiver is able to accept data and receives an error-free data packet addressed to it. The transmitting ISI device will toggle its sequence bit for that ISIId.ISISubId channel only when it receives a valid ACK handshake from the addressed ISI device.

Figure 36 shows the transmission of two long packets with the sequence bit in both the transmitting and receiving devices toggling from 0 to 1 and back to 0 again. The toggling operation will continue in this manner in every subsequent transmission until an error condition is encountered.

When the receiving ISI device detects an error in the transmitted long packet or is unable to accept the packet (because of full buffers for example) it will not return any packet and it will not toggle its local sequence bit. An example of this is depicted in Figure 37. The absence of any response prompts the transmitting device to retransmit the original (seq=0) packet. This time the packet is received without any errors (or buffer space may have been freed) so the receiving ISI device toggles its local sequence bit and responds with an ACK. The transmitting device then toggles its local sequence bit to a 1 upon correct receipt of the ACK.

However it is also possible for the ACK packet from the receiving ISI device to be corrupted and this scenario is shown in Figure 38. In this case the receiving device toggles its local sequence bit to 1 when the long packet is received without error and replies with an ACK to the transmitting device. The transmitting device does not receive the ACK correctly and so does not change its local sequence bit. It then retransmits the seq=0 long packet. When the receiving device finds that there is a mismatch between the transmitted sequence bit and the expected (local) sequence bit is discards the long packet and replies with an ACK. When the transmitting ISI device correctly receives the ACK it updates its local sequence bit to a 1, thus restoring synchronization. Note that when the *ISISubIdSeqMask* bit for the addressed ISISubId is set then the retransmitted packet is not discarded and so a duplicate packet will be received. The data contained in the packet should be self-ordering and so the software handling these packets (most likely control messages) is expected to deal with this eventuality.

12.4.4.10 Flow Control

The ISI also supports flow control by treating it in exactly the same manner as an error in the received packet. Because the SCB enjoys greater guaranteed bandwidth to DRAM than both the ISI

and USB can supply flow control should not be required during normal operation. Any blockage on a DMA channel will soon result in the *NumRetries* value being exceeded and transmission from that SoPEC being halted. If a SoPEC NAKs a packet because its RxBuffer is full it will flag an overflow condition. This condition can potentially cause a CPU interrupt, if the corresponding interrupt is enabled. The *RxOverflowSticky* bit of its *ISIIntStatus* register reflects this condition. Because flow control is treated in the same manner as an error the transmitting ISI device will not be able to differentiate a flow control condition from an error in the transmitted packet.

12.4.4.11 Auto-ping Operation

While the CPU of the ISIMaster could send a ping packet by writing the appropriate header to the *CPUISITxBuffCntrl* register it is expected that all ping packets will be generated in the ISI itself. The use of automatically generated ping packets ensures that ISISlaves will be given access to the ISI bus with a programmable minimum guaranteed frequency in addition to whenever it would otherwise be idle. Five registers facilitate the automatic generation of ping messages within the ISI: *PingSchedule0, PingSchedule1, PingSchedule2, ISITotalPeriod* and *ISILocalPeriod*. Auto-pinging will be enabled if any bit of any of the *PingScheduleN* registers is set and disabled if all *PingScheduleN* registers are 0x0000.

Each bit of the 15-bit *PingScheduleN* register corresponds to an ISIId that is used in the *Address* field of the ping packet and a 1 in the bit position indicates that a ping packet is to be generated for that ISIId. A 0 in any bit position will ensure that no ping packet is generated for that ISIId. As ISISlaves may differ in their bandwidth requirement (particularly if a storage SoPEC is present) three different PingSchedule registers are used to allow an ISISlave receive up to three times the number of pings as another active ISISlave. When the ISIMaster is not sending long packets (sourced from either the CPU or USB in the case of a SoPEC ISIMaster) ISI ping packets will be transmitted according to the pattern given by the three *PingScheduleN* registers. The ISI will start with the lsb of *PingSchedule0* register and work its way from lsb through msb of each of the *PingScheduleN* registers. When the msb of *PingSchedule2* is reached the ISI returns to the lsb of *PingSchedule0* and continues to cycle through each bit position of each *PingScheduleN* register. The ISI has more than enough time to work out the destination of the next ping packet while a ping or long packet is being transmitted.

With the addition of auto-ping operation we now have three potential sources of packets in an ISIMaster SoPEC: USB, CPU and auto-ping. Arbitration between the CPU and USB for access to the ISI is handled outside the ISI. To ensure that local packets get priority whenever possible and that ping packets can have some guaranteed access to the ISI we use two 4-bit counters whose reload value is contained in the *ISITotalPeriod* and *ISILocalPeriod* registers. As we saw in section 12.4.4.1 every ISI transaction is initiated by the ISIMaster transmitting either a long packet or a ping packet. The *ISITotalPeriod* counter is decremented for every ISI transaction (i.e. either long or ping) when its value is non-zero. The *ISILocalPeriod* counter is decremented for every local packet that is transmitted. Neither counter is decremented by a retransmitted packet. If the *ISITotalPeriod* counter is zero then ping packets will not change its value from zero. Both the *ISITotalPeriod* and

ISILocalPeriod counters are reloaded by the next local packet transmit request after the *ISITotalPeriod* counter has reached zero and this local packet has priority over pings.

The amount of guaranteed ISI bandwidth allocated to both local and ping packets is determined by the values of the *ISITotalPeriod* and *ISILocalPeriod* registers. Local packets will always be given priority when the *ISILocalPeriod* counter is non-zero. Ping packets will be given priority when the *ISILocalPeriod* counter is zero and the *ISITotalPeriod* counter is still non-zero.

Note that ping packets are very likely to get more than their guaranteed bandwidth as they will be transmitted whenever the ISI bus would otherwise be idle (i.e. no pending local packets). In particular when the *ISITotalPeriod* counter is zero it will not be reloaded until another local packet is pending and so ping packets transmitted when the *ISITotalPeriod* counter is zero will be in addition to the guaranteed bandwidth. Local packets on the other hand will never get more than their guaranteed bandwidth because each local packet transmitted decrements both counters and will cause the counters to be reloaded when the *ISITotalPeriod* counter is zero. The difference between the values of the *ISITotalPeriod* and *ISILocalPeriod* registers determines the number of automatically generated ping packets that are guaranteed to be transmitted every *ISITotalPeriod* number of ISI transactions. If the *ISITotalPeriod* and *ISILocalPeriod* values are the same then the local packets will always get priority and could totally exclude ping packets if the CPU always has packets to send.

For example if *ISITotalPeriod* = 0xC; *ISILocalPeriod* = 0x8; *PingSchedule0* = 0x0E; *PingSchedule1* = 0x0C and *PingSchedule2* = 0x08 then four ping messages are guaranteed to be sent in every 12 ISI transactions. Furthermore ISIId3 will receive 3 times the number of ping packets as ISId1 and ISIId2 will receive twice as many as ISId1. Thus over a period of 36 contended ISI transactions (allowing for two full rotations through the three *PingScheduleN* registers) when local packets are always pending 24 local packets will be sent, ISId1 will receive 2 ping packets, ISId2 will receive 4 pings and ISId3 will receive 6 ping packets. If local traffic is less frequent then the ping frequency will automatically adjust upwards to consume all remaining ISI bandwidth.

12.4.5 Wake-up from Sleep Mode

Either the PrintMaster SoPEC or the external host may place any of the ISISlave SoPECs in sleep mode prior to going into sleep mode itself. The ISISlave device should then ensure that its *ISIWakeupEnable* bit of the *WakeupEnable* register (see Table 34) is set prior to entering sleep mode. In an ISISlave device the ISI block will continue to receive power and clock during sleep mode so that it may monitor the *gpio_isi_din* lines for activity. When ISI activity is detected during sleep mode and the *ISIWakeupEnable* bit is set the ISI asserts the *isi_cpr_reset_n* signal. This will bring the rest of the chip out of sleep mode by means of a wakeup reset. See chapter 16 for more details of reset propagation.

12.4.6 Implementation

Although the ISI consists of either 2 or 4 ISI data lines over which a serial data stream is demultiplexed, each ISI line is treated as a separate serial link at the physical layer. This permits a certain amount of skew between the ISI lines that could not be tolerated if the lines were treated as

a parallel bus. A lower Bit Error Rate (BER) can be achieved if the serial data recovery is performed separately on each serial link. Figure 39 illustrates the ISI sub block partitioning.

12.4.6.1 ISI Sub-block Partition

* Definition of I/Os.

Table 34. ISI I/O

Port name	Pins	I/O	Description
Clock and Reset			
isi_pclk	1	In	ISI primary clock.
isi_reset_n	1	In	ISI reset. Active low. Asserting *isi_reset_n* will reset all ISI logic. Synchronous to isi_pclk.
Configuration			
isi_go	1	In	ISI GO. Active high. When GO is de-asserted, all ISI statemachines are reset to their idle states, all ISI output signals are de-asserted, but all ISI counters retain their values. When GO is asserted, all ISI counters are reset and all ISI statemachines and output signals will return to their normal mode of operation.
isi_master_select	1	In	ISI master select. Determines whether the SoPEC is an ISIMaster or not 1 = ISIMaster 0 = ISISlave
isi_id[3:0]	4	In	ISI ID for this device.
isi_retries[3:0]	4	In	ISI number of retries. Number of times a transmitting ISI device will attempt retransmission of a NAK'd packet before aborting the transmission and flagging an error. The value of this configuration signal should not be changed while there are valid packets in the Tx buffer.
isi_ping_schedule0[14 :0]	15	In	ISI auto ping schedule #0. Denotes which ISIIds will be receive ping packets. Note that bit0 refers to ISIId0, bit1 to ISIId1...bit14 to ISIId14. Setting a bit in this schedule will enable auto ping generation for the corresponding ISI ID. The ISI will start from the bit 0 of *isi_ping_schedule0* and cycle through to bit 14, generating pings for each bit that is set. This operation will be performed in sequence from

			isi_ping_schedule0 through *isi_ping_schedule2.*
isi_ping_schedule1[14 :0]	15	In	As per *isi_ping_schedule0.*
isi_ping_schedule2[14 :0]	15	In	As per *isi_ping_schedule0.*
isi_total_period[3:0]	4	In	Reload value of the ISI Total Period Counter.
isi_local_period[3:0]	4	In	Reload value of the ISI Local Period Counter.
isi_number_pins	1	In	Number of active ISI data pins. Used to select how many serial data pins will be used to transmit and receive data. Should reflect the number of ISI device data pins that are in use. 1 = isi_data[3:0] active 0 = isi_data[1:0] active
isi_turn_around[3:0]	4	In	ISI bus turn around time in ISI clock cycles (32MHz).
isi_short_reply_win[4:0]	5	In	ISI long packet reply window in ISI clock cycles (32MHz).
isi_long_reply_win[8:0]	9	In	ISI long packet reply window in ISI clock cycles (32MHz).
isi_tx_enable	1	In	ISI transmit enable. Active high. Enables ISI transmission of long or ping packets. ACKs may still be transmitted when this bit is 0. The value of this configuration signal should not be changed while there are valid packets in the Tx buffer.
isi_rx_enable	1	In	ISI receive enable. Active high. Enables ISI packet reception. Any activity on the ISI bus will be ignored when this signal is de-asserted. This signal should only be de-asserted if the ISI block is not required for use in the design.
isi_bit_stuff_rate[3:0]	1	In	ISI bit stuffing limit. Allows the bit stuffing counter value to be programmed. Is loaded into the 4 upper bits of the 7bit wide bit stuffing counter. The lower bits are always loaded with b111, to prevent bit stuffing for less than 7 consecutive ones or zeroes. E.g. b000 : stuff_count = b0000111 : bit stuff after 7 consecutive 0/1 b111 : stuff_count = b1111111 : bit stuff after127 consecutive 0/1
Serial Link Signals			

isi_ser_data_in[3:0]	4	In	ISI Serial data inputs. Each bit corresponds to a separate serial link.
isi_ser_data_out[3:0]	4	Out	ISI Serial data outputs. Each bit corresponds to a separate serial link.
isi_ser_data_en[3:0]	4	Out	ISI Serial data driver enables. Active high. Each bit corresponds to a separate serial link.
Tx Packet Buffer			
isi_tx_wr_en	1	In	ISI Tx FIFO write enable. Active high. Asserting *isi_tx_wr_en* will write the 64 bit data on *isi_tx_wr_data* to the FIFO, providing that space is available in the FIFO. If *isi_tx_wr_en* remains asserted after the last entry in the current packet is written, the write operation will wrap around to the start of the next packet, providing that space is available for a second packet in the FIFO.
isi_tx_wr_data[63:0]	64	In	ISI Tx FIFO write data.
isi_tx_ping	1	In	ISI Tx FIFO ping packet select. Active high. Asserting *isi_tx_ping* will queue a ping packet for transmission, as opposed to a long packet. Although there is no data payload for a ping packet, a packet location in the FIFO is used as a 'place holder' for the ping packet. Any data written to the associated packet location in the FIFO will be discarded when the ping packet is transmitted.
isi_tx_id[3:0]	5	In	ISI Tx FIFO packet ID. ISI ID for each packet written to the FIFO. Registered when the last entry of the packet is written.
isi_tx_sub_id	1	In	ISI Tx FIFO packet sub ID. ISI sub ID for each packet written to the FIFO. Registered when the last entry of the packet is written.
isi_tx_pkt_count[1:0]	2	Out	ISI Tx FIFO packet count. Indicates the number of packets contained in the FIFO. The FIFO has a capacity of 2 x 256 bit packets. Range is b00->b10.
isi_tx_word_count[2:0]	3	Out	ISI Tx FIFO current packet word count. Indicates the number of words contained in the current Tx packet location of the Tx FIFO. Each packet location has a capacity of 4 x 64 bit words. Range is b000->b100.

isi_tx_empty	1	Out	ISI Tx FIFO empty. Active high. Indicates that no packets are present in the FIFO.
isi_tx_full	1	Out	ISI Tx FIFO full. Active high. Indicates that 2 packets are present in the FIFO, therefore no more packets can be transmitted.
isi_tx_over_flow	1	Out	ISI Tx FIFO over flow. Active high. Indicates that a write operation was performed on a full FIFO. The write operation will have no effect on the contents of the FIFO or the write pointer.
isi_tx_error	1	Out	ISI Tx FIFO error. Active high. Indicates that an error occurred while transmitting the packet currently at the head of the FIFO. This will happen if the number of transmission attempts exceeds *isi_tx_retries*.
isi_tx_desc[2:0]	3	Out	ISI Tx packet descriptor field. ISI packet descriptor field for the packet currently at the head of the FIFO. See Table for details. Only valid when *isi_tx_empty*=0, i.e. when there is a valid packet in the FIFO.
isi_tx_addr[4:0]	5	Out	ISI Tx packet address field. ISI address field for the packet currently at the head of the FIFO. See Table for details. Only valid when *isi_tx_empty*=0, i.e. when there is a valid packet in the FIFO.
Rx Packet FIFO			
isi_rx_rd_en	1	In	ISI Rx FIFO read enable. Active high. Asserting *isi_rx_rd_en* will drive *isi_rx_rd_data* with valid data, from the Rx packet at the head of the FIFO, providing that data is available in the FIFO. If *isi_rx_rd_en* remains asserted after the last entry is read from the current packet, the read operation will wrap around to the start of the next packet, providing that a second packet is available in the FIFO.
isi_rx_rd_data[63:0]	64	Out	ISI Rx FIFO read data.
isi_rx_sub_id	1	Out	ISI Rx packet sub ID. Indicates the ISI sub ID associated with the packet at the head of the Rx FIFO.
isi_rx_pkt_count[1:0]	2	Out	ISI Rx FIFO packet count. Indicates the number of packets contained in the FIFO.

			The FIFO has a capacity of 2 x 256 bit packets. Range is b00->b10.
isi_rx_word_count[2:0]	3	Out	ISI Rx FIFO current packet word count. Indicates the number of words contained in the Rx packet location at the head of the FIFO. Each packet location has a capacity of 4 x 64 bit words. Range is b000->b100.
isi_rx_empty	1	Out	ISI Rx FIFO empty. Active high. Indicates that no packets are present in the FIFO.
isi_rx_full	1	Out	ISI Rx FIFO full. Active high. Indicates that 2 packets are present in the FIFO, therefore no more packets can be received.
isi_rx_over_flow	1	Out	ISI Rx FIFO over flow. Active high. Indicates that a packet was addressed to the local ISI device, but the Rx FIFO was full, resulting in a NAK.
isi_rx_under_run	1	Out	ISI Rx FIFO under run. Active high. Indicates that a read operation was performed on an empty FIFO. The invalid read will return the contents of the memory location currently addressed by the FIFO read pointer and will have no effect on the read pointer.
isi_rx_frame_error	1	Out	ISI Rx framing error. Active high. Asserted by the ISI when a framing error is detected in the received packet, which can be caused by an incorrect *Start* or *Stop* field or by bit stuffing errors. The associated packet will be dropped.
isi_rx_crc_error	1	Out	ISI Rx CRC error. Active high. Asserted by the ISI when a CRC error is detected in an incoming packet. Other than dropping the errored packet ISI reception is unaffected by a CRC Error.

12.4.6.2 ISI Serial Interface Engine (isi_sie)

There are 4 instantiations of the *isi_sie* sub block in the ISI, 1 per ISI serial link. The *isi_sie* is responsible for Rx serial data sampling, Tx serial data output and bit stuffing.

Data is sampled based on a phase detection mechanism. The incoming ISI serial data stream is over sampled 5 times per ISI bit period. The phase of the incoming data is determined by detecting transitions in the ISI serial data stream, which indicates the ISI bit boundaries. An ISI bit boundary is defined as the sample phase at which a transition was detected.

The basic functional components of the isi_sie are detailed in Figure 40. These components are simply a grouping of logical functionality and do not necessarily represent hierarchy in the design.

12.4.6.2.1 SIE Edge Detection and Data I/O

The basic structure of the data I/O and edge detection mechanism is detailed in Figure 41.

NOTE: Serial data from the receiver in the pad MUST be synchronized to the *isi_pclk* domain with a 2 stage shift register external to the ISI, to reduce the risk of metastability. *ser_data_out* and *ser_data_en* should be registered externally to the ISI.

The Rx/Tx statemachine drives *ser_data_en*, *stuff_1_en* and *stuff_0_en.* The signals *stuff_1_en* and *stuff_0_en* cause a one or a zero to be driven on *ser_data_out* when they are asserted, otherwise *fifo_rd_data* is selected.

12.4.6.2.2 SIE Rx/Tx Statemachine

The Rx/Tx statemachine is responsible for the transmission of ISI Tx data and the sampling of ISI Rx data. Each ISI bit period is 5 *isi_pclk* cycles in duration.

The Tx cycle of the Rx/Tx statemachine is illustrated in Figure 42. It generates each ISI bit that is transmitted. States tx0->tx4 represent each of the 5 *isi_pclk* phases that constitute a Tx ISI bit period. *ser_data_en* controls the tristate enable for the ISI line driver in the bidirectional pad, as shown in Figure 41. *rx_tx_cycle* is asserted during both Rx and Tx states to indicate an active Rx or Tx cycle. It is primarily used to enable bit stuffing.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

The Tx cycle for Tx bit stuffing when the Rx/Tx statemachine inserts a '0' into the bitstream can be seen in Figure 43.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated

The Tx cycle for Tx bit stuffing when the RxTx statemachine inserts a '1' into the bitstream can be seen in Figure 44.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated

The *tx** and *stuff** states are detailed separately for clarity. They could be easily combined when coding the statemachine, however it would be better for verification and debugging if they were kept separate.

The Rx cycle of the ISI Rx/Tx statemachine is detailed in Figure 45. The Rx cycle of the Rx/Tx Statemachine, samples each ISI bit that is received. States rx0->rx4 represent each of the 5 *isi_pclk* phases that constitute a Rx ISI bit period.

The optimum sample position for an ideal ISI bit period is 2 *isi_pclk* cycles after the ISI bit boundary sample, which should result in a data sample close to the centre of the ISI bit period.

rx_sample is asserted during the *rx2* state to indicate a valid ISI data sample on *rx_bit,* unless the bit should be stripped when flagged by the bit stuffing statemachine, in which case *rx_sample* is not asserted during *rx2* and the bit is not written to the FIFO. When *edge* is asserted, it resets the Rx cycle to the rx0 state, from any rx state. This is how the *isi_sie* tracks the phase of the incoming data. The Rx cycle will cycle through states rx0->rx4 until *edge* is asserted to reset the sample phase, or a *tx_req* is asserted indicating that the ISI needs to transmit.

Due to the 5 times oversampling a maximum phase error of 0.4 of an ISI bit period (2 *isi_pclk* cycles out of 5) can be tolerated.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

An example of the Tx data generation mechanism is detailed in Figure 46. *tx_req* and *fifo_wr_tx* are driven by the framer block.

An example of the Rx data sampling functional timing is detailed in Figure 47. The dashed lines on the *ser_data_in_ff* signal indicate where the Rx/Tx statemachine perceived the bit boundary to be, based on the phase of the last ISI bit boundary. It can be seen that data is sampled during the same phase as the previous bit was, in the absence of a transition.

12.4.6.2.3 SIE Rx/Tx FIFO

The Rx/Tx FIFO is a 7 x 1 bit synchronous look-ahead FIFO that is shared for Tx and Rx operations. It is required to absorb any Rx/Tx latency caused by bit stripping/stuffing on a per ISI line basis, i.e. some ISI lines may require bit stripping/stuffing during an ISI bit period while the others may not, which would lead to a loss of synchronization between the data of the different ISI lines, if a FIFO were not present in each *isi_sie*.

The basic functional components of the FIFO are detailed in Figure 48. *tx_ready* is driven by the Rx/Tx statemachine and selects which signals control the read and write operations. *tx_ready*=1 during ISI transmission and selects the *fifo_*tx* control and data signals. *tx_ready*=0 during ISI reception and selects the *fifo_*rx* control and data signals. *fifo_reset* is driven by the Rx/Tx statemachine. It is active high and resets the FIFO and associated logic before/after transmitting a packet to discard any residual data.

The size of the FIFO is based on the maximum bit stuffing frequency and the size of the shift register used to segment/re-assemble the multiple serial streams in the ISI framing logic. The maximum bit stuffing frequency is every 7 consecutive ones or zeroes. The shift register used is 32 bits wide. This implies that the maximum number of stuffed bits encountered in the time it takes to fill/empty the shift register if 4. This would suggest that 4 x 1 bit would be the minimum *ideal* size of the FIFO. However it is necessary to allow for different skew and phase error between the ISI lines, hence a 7 x 1 bit FIFO.

The FIFO is controlled by the *isi_sie* during packet reception and is controlled by the *isi_frame* block during packet transmission. This is illustrated in Figure 49. The signal *tx_ready* selects which mode the FIFO control signals operate in. When *tx_ready*=0, i.e. Rx mode, the *isi_sie* control signals *rx_sample*, *fifo_rd_rx* and *ser_data_in_ff* are selected. When *tx_ready*=1, i.e. Tx mode, the *sie_frame* control signals *fifo_wr_tx*, *fifo_rd_tx* and *fifo_wr_data_tx* are selected.

12.4.6.3 Bit Stuffing

Programmable bit stuffing is implemented in the *isi_sie*. This is to allow the system to determine the amount of bit stuffing necessary for a specific ISI system devices. It is unlikely that bit stuffing would be required in a system using a 100ppm rated crystal. However, a programmable bit stuffing implementation is much more versatile and robust.

The bit stuffing logic consists of a counter and a statemachine that track the number of consecutive ones or zeroes that are transmitted or received and flags the Rx/Tx statemachine when the bit stuffing limit has been reached. The counter, *stuff_count*, is a 7 bit counter, which decrements when *rx_sample* is asserted on a Rx cycle or when *fifo_rd_tx* is asserted on a Tx cycle. The upper 4 bits of *stuff_count* are loaded with *isi_bit_stuff_rate*. The lower 3 bits of *stuff_count* are always loaded with b111, i.e. for *isi_bit_stuff_rate* = b000, the counter would be loaded with b0000111. This is to

prevent bit stuffing for less than 7 consecutive ones or zeroes. This allows the bit stuffing limit to be set in the range 7->127 consecutive ones or zeroes.

NOTE: It is extremely important that a change in the bit stuffing rate, *isi_bit_stuff_rate*, is carefully co-ordinated between ISI devices in a system. It is obvious that ISI devices will not be able to communicate reliably with each other with different bit stuffing settings. It is recommended that all ISI devices in a system default to the safest bit stuffing rate (*isi_bit_stuff_rate* = b000) at reset. The system can then co-ordinate the change to an optimum bit stuffing rate.

The ISI bit stuffing statemachine Tx cycle is shown in Figure 50. The counter is loaded when *stuff_count_load* is asserted.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

The ISI bit stuffing statemachine Rx cycle is shown in Figure 51. It should be noted that the statemachine enters the strip state when *stuff_count=0x2*. This is because the statemachine can only transition to *rx0* or *rx1* when *rx_sample* is asserted as it needs to be synchronized to changes in sampling phase introduced by the Rx/Tx statemachine. Therefore a one or a zero has already been sampled by the time it enters *rx0* or *rx1*. This is not the case for the Tx cycle, as it will always have a stable 5 isi_pclk cycles per bit period and relies purely on the data value when entering *tx0* or *tx1*. The Tx cycle therefore enters *stuff1* or *stuff0* when *stuff_count=0x1*.

NOTE: All statemachine signals are assumed to be '0' unless otherwise stated.

12.4.6.4 ISI Framing and CRC sub-block (isi_frame)

12.4.6.4.1 CRC Generation/Checking

A Cyclic Redundancy Checksum (CRC) is calculated over all fields except the start and stop fields for each long or ping packet transmitted. The receiving ISI device will perform the same calculation on the received packet to verify the integrity of the packet. The procedure used in the CRC generation/checking is the same as the Frame Checking Sequence (FCS) procedure used in HDLC, detailed in ITU-T Recommendation T30[39].

For generation/checking of the CRC field, the shift register illustrated in Figure 52 is used to perform the modulo 2 division on the packet contents by the polynomial $G(x) = x^{16} + x^{12} + x^5 + 1$.

To generate the CRC for a transmitted packet, where T(x) = [Packet Descriptor field, Address field, Data Payload field] (a ping packet will not contain a data payload field).

- Set the shift register to 0xFFFF.
- Shift *T(x)* through the shift register, LSB first. This can occur in parallel with the packet transmission.
- Once the each bit of *T(x)* has been shifted through the register, it will contain the remainder of the modulo 2 division *T(x)*/G(x).
- Perform a ones complement of the register contents, giving the CRC field which is transmitted MSB first, immediately following the last bit of *M(x*

To check the CRC for a received packet, where R(x) = [Packet Descriptor field, Address field, Data Payload field, CRC field] (a ping packet will not contain a data payload field).

- Set the shift register to 0xFFFF.

- Shift R(x) through the shift register, LSB first. This can occur in parallel with the packet reception.
- Once each bit of the packet has been shifted through the register, it will contain the remainder of the modulo 2 division *R(x)*/G(x).
- The remainder should equal b00011101000011111, for a packet without errors.

12.5 CTRL (CONTROL SUB-BLOCK)

12.5.1 Overview

The CTRL is responsible for high level control of the SCB sub-blocks and coordinating access between them. All control and status registers for the SCB are contained within the CTRL and are accessed via the CPU interface. The other major components of the CTRL are the SCB Map logic and the DMA Manager logic.

12.5.2 SCB Mapping

In order to support maximum flexibility when moving data through a multi-SoPEC system it is possible to map any USB endpoint onto either DMAChannel within any SoPEC in the system. The SCB map, and indeed the SCB itself is based around the concept of an ISIId and an ISISubId. Each SoPEC in the system has a unique ISIId and two ISISubIds, namely ISISubId0 and ISISubId1. We use the convention that ISISubId0 corresponds to DMAChannel0 in each SoPEC and ISISubId1 corresponds to DMAChannel1. The naming convention for the ISIId is shown in Table 35 below and this would correspond to a multi-SoPEC system such as that shown in Figure 27. We use the term ISIId instead of SoPECId to avoid confusion with the unique ChipID used to create the SoPEC_id and SoPEC_id_key (see chapter 17 and [9] for more details).

Table 35. ISIId naming convention

ISIId	SoPEC to which it refers
0 - 14	Standard device ISIIds (0 is the power-on reset value)
15	Broadcast ISIId

The combined ISIId and ISISubId therefore allows the ISI to address DMAChannel0 or DMAChannel1 on any SoPEC device in the system. The ISI, DMA manager and SCB map hardware use the ISIId and ISISubId to handle the different data streams that are active in a multi-SoPEC system as does the software running on the CPU of each SoPEC. In this document we will identify DMAChannels as *ISIx.y* where *x* is the ISIId and *y* is the ISISubId. Thus ISI2.1 refers to DMAChannel1 of ISISlave2. Any data sent to a broadcast channel, i.e. ISI15.0 or ISI15.1, are received by every ISI device in the system including the ISIMaster (which may be an ISI-Bridge). The USB device controller and software stacks however have no understanding of the ISIId and ISISubId but the Silverbrook printer driver software running on the external host does make use of the ISIId and ISISubId. USB is simply used as a data transport - the mapping of USB device endpoints onto ISIId and SubId is communicated from the external host Silverbrook code to the SoPEC Silverbrook code through USB control (or possibly bulk data) messages i.e. the mapping

information is simply data payload as far as USB is concerned. The code running on SoPEC is responsible for parsing these messages and configuring the SCB accordingly.

The use of just two DMAChannels places some limitations on what can be achieved without software intervention. For every SoPEC in the system there are more potential sources of data than there are sinks. For example an ISISlave could receive both control and data messages from the ISIMaster SoPEC in addition to control and data from the external host, either specifically addressed to that particular ISISlave or over the broadcast ISI channel. However all ISISlaves only have two possible data sinks, i.e. DMAChannel0 and DMAChannel1. Another example is the ISIMaster in a multi-SoPEC system which may receive control messages from each SoPEC in addition to control and data information from the external host (e.g. over USB). In this case all of the control messages are in contention for access to DMAChannel0. We resolve these potential conflicts by adopting the following conventions:

1) Control messages may be interleaved in a memory buffer: The memory buffer that the DMAChannel0 points to should be regarded as a central pool of control messages. Every control message must contain fields that identify the size of the message, the source and the destination of the control message. Control messages may therefore be multiplexed over a DMAChannel which allows several control message sources to address the same DMAChannel. Furthermore, if SoPEC-type control messages contain source and destination fields it is possible for the external host to send control messages to individual SoPECs over the ISI15.0 broadcast channel.

2) Data messages should not be interleaved in a memory buffer: As data messages are typically part of a much larger block of data that is being transferred it is not possible to control their contents in the same manner as is possible with the control messages. Furthermore we do not want the CPU to have to perform reassembly of data blocks. Data messages from different sources cannot be interleaved over the same DMAChannel - the SCB map must be reconfigured each time a different data source is given access to the DMAChannel.

3) Every reconfiguration of the SCB map requires the exchange of control messages: SoPEC's SCB map reset state is shown in Table and any subsequent modifications to this map require the exchange of control messages between the SoPEC and the external host. As the external host is expected to control the movement of data in any SoPEC system it is anticipated that all changes to the SCB map will be performed in response to a request from the external host. While the SoPEC could autonomously reconfigure the SCB map (this is entirely up to the software running on the SoPEC) it should not do so without informing the external host in order to avoid data being mis-routed.

An example of the above conventions in operation is worked through in section 12.5.2.3.

12.5.2.1 SCB map rules

The operation of the SCB map is described by these 2 rules:

Rule 1: A packet is routed to the DMA manager if it originates from the USB device core and has an ISIId that matches the local SoPEC ISIId.

Rule 2: A packet is routed to the ISI if it originates from the CPU or has an ISIId that does not match the local SoPEC ISIId.

If the CPU erroneously addresses a packet to the ISIId contained in the *ISIId* register (i.e. the ISIId of the local SoPEC) then that packet will be transmitted on the ISI rather than be sent to the DMA manager. While this will usually cause an error on the ISI there is one situation where it could be beneficial, namely for initial dialog in a 2 SoPEC system as both devices come out of reset with an ISIId of 0.

12.5.2.2 External host to ISIMaster SoPEC communication

Although the SCB map configuration is independent of ISIMaster status, the following discussion on SCB map configurations assumes the ISIMaster is a SoPEC device rather than an ISI bridge chip, and that only a single USB connection to the external host is present. The information should apply broadly to an ISI-Bridge but we focus here on an ISIMaster SoPEC for clarity.

As the ISIMaster SoPEC represents the printer device on the PC USB bus it is required by the USB specification to have a dedicated control endpoint, EP0. At boot time the ISIMaster SoPEC will also require a bulk data endpoint to facilitate the transfer of program code from the external host. The simplest SCB map configuration, i.e. for a single stand-alone SoPEC, is sufficient for external host to ISIMaster SoPEC communication and is shown in Table 36.

Table 36. Single SoPEC SCB map configuration

Source	Sink
EP0	ISI0.0
EP1	ISI0.1
EP2	nc
EP3	nc
EP4	nc

In this configuration all USB control information exchanged between the external host and SoPEC over EP0 (which is the only bidirectional USB endpoint). SoPEC specific control information (printer status, DNC info etc.) is also exchanged over EP0.

All packets sent to the external host from SoPEC over EP0 must be written into the DMA mapped EP buffer by the CPU (LEON-PC dataflow in Figure 29). All packets sent from the external host to SoPEC are placed in DRAM by the DMA Manager, where they can be read by the CPU (PC-DIU dataflow in Figure 29). This asymmetry is because in a multi-SoPEC environment the CPU will need to examine all incoming control messages (i.e. messages that have arrived over DMAChannel0) to ascertain their source and destination (i.e. they could be from an ISISlave and destined for the external host) and so the additional overhead in having the CPU move the short control messages to the EP0 FIFO is relatively small. Furthermore we wish to avoid making the SCB more complicated than necessary, particularly when there is no significant performance gain to be had as the control traffic will be relatively low bandwidth.

The above mechanisms are appropriate for the types of communication outlined in sections 12.1.2.1.1 through 12.1.2.1.4

12.5.2.3 Broadcast communication

The SCB configuration for broadcast communication is also the default, post power-on reset, configuration for SoPEC and is shown in Table 37.

Table 37. Default SoPEC SCB map configuration

Source	Sink
EP0	ISI0.0
EP1	ISI0.1
EP2	ISI15.0
EP3	ISI15.1
EP4	ISI1.1

USB endpoints EP2 and EP3 are mapped onto ISISubID0 and ISISubId1 of ISIId15 (the broadcast ISIId channel). EP0 is used for control messages as before and EP1 is a bulk data endpoint for the ISIMaster SoPEC. Depending on what is convenient for the boot loader software, EP1 may or may not be used during the initial program download, but EP1 is highly likely to be used for compressed page or other program downloads later. For this reason it is part of the default configuration. In this setup the USB device configuration will take place, as it always must, by exchanging messages over the control channel (EP0).

One possible boot mechanism is where the external host sends the bootloader1 program code to all SoPECs by broadcasting it over EP3. Each SoPEC in the system then authenticates and executes the bootloader1 program. The ISIMaster SoPEC then polls each ISISlave (over the ISIx.0 channel). Each ISISlave ascertains its ISIId by sampling the particular GPIO pins required by the bootloader1 and reporting its presence and status back to the ISIMaster. The ISIMaster then passes this information back to the external host over EP0. Thus both the external host and the ISIMaster have knowledge of the number of SoPECs, and their ISIIds, in the system. The external host may then reconfigure the SCB map to better optimise the SCB resources for the particular multi-SoPEC system. This could involve simplifying the default configuration to a single SoPEC system or remapping the broadcast channels onto DMAChannels in individual ISISlaves.

The following steps are required to reconfigure the SCB map from the configuration depicted in Table to one where EP3 is mapped onto ISI1.0:

1) The external host sends a control message(s) to the ISIMaster SoPEC requesting that USB EP3 be remapped to ISI1.0

2) The ISIMaster SoPEC sends a control message to the external host informing it that EP3 has now been mapped to ISI1.0 (and therefore the external host knows that the previous mapping of ISI15.1 is no longer available through EP3).

3) The external host may now send control messages directly to ISISlave1 without requiring any CPU intervention on the ISIMaster SoPEC

12.5.2.4 External host to ISISlave SoPEC communication

If the ISIMaster is configured correctly (e.g. when the ISIMaster is a SoPEC, and that SoPEC's SCB map is configured correctly) then data sent from the external host destined for an ISISlave will be transmitted on the ISI with the correct address. The ISI automatically forwards any data addressed to it (including broadcast data) to the DMA channel with the appropriate ISISubId. If the ISISlave has data to send to the external host it must do so by sending a control message to the ISIMaster identifying the external host as the intended recipient. It is then the ISIMaster's responsibility to forward this message to the external host.

With this configuration the external host can communicate with the ISISlave via broadcast messages only and this is the mechanism by which the bootloader1 program is downloaded. The ISISlave is unable to communicate with the external host (or the ISIMaster) until the bootlloader1 program has successfully executed and the ISISlave has determined what its ISIId is. After the bootloader1 program (and possibly other programs) has executed the SCB map of the ISIMaster may be reconfigured to reflect the most appropriate topology for the particular multi-SoPEC system it is part of.

All communication from an ISISlave to external host is either achieved directly (if there is a direct USB connection present for example) or by sending messages via the ISIMaster. The ISISlave can never initiate communication to the external host. If an ISISlave wishes to send a message to the external host via the ISIMaster it must wait until it is pinged by the ISIMaster and then send a the message in a long packet addressed to the ISIMaster. When the ISIMaster receives the message from the ISISlave it first examines it to determine the intended destination and will then copy it into the EP0 FIFO for transmission to the external host. The software running on the ISIMaster is responsible for any arbitration between messages from different sources (including itself) that are all destined for the external host.

The above mechanisms are appropriate for the types of communication outlined in sections 12.1.2.1.5 and 12.1.2.1.6.

12.5.2.5 ISIMaster to ISISlave communication

All ISIMaster to ISISlave communication takes place over the ISI. Immediately after reset this can only be by means of broadcast messages. Once the bootloader1 program has successfully executed on all SoPECs in a multi-SoPEC system the ISIMaster can communicate with each SoPEC on an individual basis.

If an ISISlave wishes to send a message to the ISIMaster it may do so in response to a ping packet from the ISIMaster. When the ISIMaster receives the message from the ISISlave it must interpret the message to determine if the message contains information required to be sent to the external host. In the case of the ISIMaster being a SoPEC, software will transfer the appropriate information into the EP0 FIFO for transmission to the external host.

The above mechanisms are appropriate for the types of communication outlined in sections 12.1.2.3.3 and 12.1.2.3.4.

12.5.2.6 ISISlave to ISISlave communication

ISISlave to ISISlave communication is expected to be limited to two special cases: (a) when the PrintMaster is not the ISIMaster and (b) when a storage SoPEC is used. When the PrintMaster is

not the ISIMaster then it will need to send control messages (and receive responses to these messages) to other ISISlaves. When a storage SoPEC is present it may need to send data to each SoPEC in the system. All ISISlave to ISISlave communication will take place in response to ping messages from the ISIMaster.

12.5.2.7 Use of the SCB map in an ISISlave with a external host connection

After reset any SoPEC (regardless of ISIMaster/Slave status) with an active USB connection will route packets from EP0,1 to DMA channels 0,1 because the default SCB map is to map EP0 to ISIId0.0 and EP1 to ISIId0.1 and the default ISIId is 0. At some later time the SoPEC learns its true ISIId for the system it is in and re-configures its ISIId and SCB map registers accordingly. Thus if the true ISIId is 3 the external host could reconfigure the SCB map so that EP0 and EP1 (or any other endpoints for that matter) map to ISIId3.0 and 3.1 respectively. The co-ordination of the updating of the ISIId registers and the SCB map is a matter for software to take care of. While the *AutoMasterEnable* bit of the *ISICntrl* register is set the external host must not send packets down EP2-4 of the USB connection to the device intended to be an ISISlave. When *AutoMasterEnable* has been cleared the external host may send data down any endpoint of the USB connection to the ISISlave.

The SCB map of an ISISlave can be configured to route packets from any EP to any ISIId.ISISubId (just as an ISIMaster can). As with an ISIMaster these packets will end up in the SCBTxBuffer but while an ISIMaster would just transmit them when it got a local access slot (from ping arbitration) the ISISlave can only transmit them in response to a ping. All this would happen without CPU intervention on the ISISlave (or ISIMaster) and as long as the ping frequency is sufficiently high it would enable maximum use of the bandwidth on both USB buses.

12.5.3 DMA Manager

The DMA manager manages the flow of data between the SCB and the embedded DRAM. Whilst the CPU could be used for the movement of data in SoPEC, a DMA manager is a more efficient solution as it will handle data in a more predictable fashion with less latency and requiring less buffering. Furthermore a DMA manager is required to support the ISI transfer speed and to ensure that the SoPEC could be used with a high speed ISI-Bridge chip in the future.

The DMA manager utilizes 2 write channels (DMAChannel0, DMAChannel1) and 1 read/write channel (DMAChannel2) to provide 2 independent modes of access to DRAM via the DIU interface:

- USBD/ISI type access.
- USBH type access.

DIU read and write access is in bursts of 4x64 bit words. Byte aligned write enables are provided for write access. Data for DIU write accesses will be read directly from the buffers contained in the respective SCB sub-blocks. There is no internal SCB DMA buffer. The DMA manager handles all issues relating to byte/ word/longword address alignment, data endianness and transaction scheduling. If a DMA channel is disabled during a DMA access, the access will be completed.

Arbitration will be performed between the following DIU access requests:

- USBD write request.
- ISI write request.

- USBH write request.
- USBH read request.

DMAChannel0 will have absolute priority over any DMA requestors. In the absence of DMAChannel0 DMA requests, arbitration will be performed in a round robin manner, on a per cycle basis over the other channels.

12.5.3.1 DMA Effective Bandwidth

The DIU bandwidth available to the DMA manager must be set to ensure adequate bandwidth for all data sources, to avoid back pressure on the USB and the ISI. This is achieved by setting the output (i.e. DIU) bandwidth to be greater than the combined input bandwidths (i.e. USBD + USBH + ISI). The required bandwidth is expected to be 160 Mbits/s (1 bit/cycle @ 160MHz). The guaranteed DIU bandwidth for the SCB is programmable and may need further analysis once there is better knowledge of the data throughput from the USB IP cores.

12.5.3.2 USBD/ISI DMA access

The DMA manager uses the two independent unidirectional write channels for this type of DMA access, one for each ISISubId, to control the movement of data. Both DMAChannel0 and DMAChannel1 only support write operation and can transfer data from any USB device DMA mapped EP buffer and from the ISI receive buffer to separate circular buffers in DRAM, corresponding to each DMA channel.

While the DMA manager performs the work of moving data the CPU controls the destination and relative timing of data flows to and from the DRAM. The management of the DRAM data buffers requires the CPU to have accurate and timely visibility of both the DMA and PEP memory usage. In other words when the PEP has completed processing of a page band the CPU needs to be aware of the fact that an area of memory has been freed up to receive incoming data. The management of these buffers may also be performed by the external host.

12.5.3.2.1 Circular buffer operation

The DMA manager supports the use of circular buffers for both DMAChannels. Each circular buffer is controlled by 5 registers: *DMAnBottomAdr, DMAnTopAdr, DMAnMaxAdr, DMAnCurrWPtr* and *DMAnIntAdr*. The operation of the circular buffers is shown in Figure 53 below.

Here we see two snapshots of the status of a circular buffer with (b) occurring sometime after (a) and some CPU writes to the registers occurring in between (a) and (b). These CPU writes are most likely to be as a result of a finished band interrupt (which frees up buffer space) but could also have occurred in a DMA interrupt service routine resulting from *DMAnIntAdr* being hit. The DMA manager will continue filling the free buffer space depicted in (a), advancing the *DMAnCurrWPtr* after each write to the DIU. Note that the *DMACurrWPtr* register always points to the next address the DMA manager will write to. When the DMA manager reaches the address in *DMAnIntAdr* (i.e. *DMACurrWPtr = DMAnIntAdr*) it will generate an interrupt if the *DMAnIntAdrMask* bit in the *DMAMask* register is set. The purpose of the *DMAnIntAdr* register is to alert the CPU that data (such as a control message or a page or band header) has arrived that it needs to process. The interrupt routine servicing the DMA interrupt will change the *DMAnIntAdr* value to the next location that data of interest to the CPU will have arrived by.

In the scenario shown in Figure 53 the CPU has determined (most likely as a result of a finished band interrupt) that the filled buffer space in (a) has been freed up and is therefore available to receive more data. The CPU therefore moves the *DMAnMaxAdr* to the end of the section that has been freed up and moves the *DMAnIntAdr* address to an appropriate offset from the *DMAnMaxAdr* address. The DMA manager continues to fill the free buffer space and when it reaches the address in *DMAnTopAdr* it wraps around to the address in *DMAnBottomAdr* and continues from there. DMA transfers will continue indefinitely in this fashion until the DMA manager reaches the address in the *DMAnMaxAdr* register.

The circular buffer is initialized by writing the top and bottom addresses to the *DMAnTopAdr* and *DMAnBottomAdr* registers, writing the start address (which does not have to be the same as the *DMAnBottomAdr* even though it usually will be) to the *DMAnCurrWPtr* register and appropriate addresses to the *DMAnIntAdr* and *DMAnMaxAdr* registers. The DMA operation will not commence until a 1 has been written to the relevant bit of the *DMAChanEn* register.

While it is possible to modify the *DMAnTopAdr* and *DMAnBottomAdr* registers after the DMA has started it should be done with caution. The *DMAnCurrWPtr* register should not be written to while the DMAChannel is in operation. DMA operation may be stalled at any time by clearing the appropriate bit of the *DMAChanEn* register or by disabling an SCB mapping or ISI receive operation.

12.5.3.2.2 Non-standard buffer operation

The DMA manager was designed primarily for use with a circular buffer. However because the DMA pointers are tested for equality (i.e. interrupts generated when *DMAnCurrWPtr* = *DMAIntAdr* or *DMAnCurrWPtr* = *DMAMaxAdr*) and no bounds checking is performed on their values (i.e. neither *DMAnIntAdr* nor *DMAnMaxAdr* are checked to see if they lie between *DMAnBottomAdr* and *DMAnTopAdr*) a number of non-standard buffer arrangements are possible. These include:

- Dustbin buffer: If *DMAnBottomAdr, DMAnTopAdr* and *DMAnCurrWPtr* all point to the same location and both *DMAnIntAdr* and *DMAnMaxAdr* point to anywhere else then all data for that DMA channel will be dumped into the same location without ever generating an interrupt. This is the equivalent to writing to /dev/null on Unix systems.
- Linear buffer: If *DMAnMaxAdr* and *DMAnTopAdr* have the same value then the DMA manager will simply fill from *DMAnBottomAdr* to *DMAnTopAdr* and then stop. *DMAnIntAdr* should be outside this buffer or have its interrupt disabled.

12.5.3.3 USBH DMA access

The USBH requires DMA access to DRAM in to provide a communication channel between the USB HC and the USB HCD via a shared memory resource. The DMA manager uses two independent channels for this type of DMA access, one for reads and one for writes. The DRAM addresses provided to the DIU interface are generated based on addresses defined in the USB HC core operational registers, in USBH section 12.3.

12.5.3.4 Cache coherency

As the CPU will be processing some of the data transferred (particularly control messages and page/band headers) into DRAM by the DMA manager, care needs to be taken to ensure that the

data it uses is the most recently transferred data. Because the DMA manager will be updating the circular buffers in DRAM without the knowledge of the cache controller logic in the LEON CPU core the contents of the cache can become outdated. This situation can be easily handled by software, for example by flushing the relevant cache lines, and so there is no hardware support to enforce cache coherency.

12.5.4 ISI transmit buffer arbitration

The SCB control logic will arbitrate access to the ISI transmit buffer (ISITxBuffer) interface on the ISI block. There are two sources of ISI Tx packets:

- CPUISITxBuffer, contained in the SCB control block.
- ISI mapped USB EP OUT buffers, contained in the USB device block.

This arbitration is controlled by the *ISITxBuffArb* register which contains a high priority bit for both the CPU and the USB. If only one of these bits is set then the corresponding source always has priority. Note that if the CPU is given absolute priority over the USB, then the software filling the ISI transmit buffer needs to ensure that sufficient USB traffic is allowed through. If both bits of the *ISITxBufferArb* have the same value then arbitration will take place on a round robin basis.

The control logic will use the *USBEPnDest registers*, as it will use the *CPUISITxBuffCntrl* register, to determine the destination of the packets in these buffers. When the ISITxBuffer has space for a packet, the SCB control logic will immediately seek to refill it. Data will be transferred directly from the CPUISITxBuffer and the ISI mapped USB EP OUT buffers to the ISITxBuffer without any intermediate buffering.

As the speed at which the ISITxBuffer can be emptied is at least 5 times greater than it can be filled by USB traffic, the ISI mapped USB EP OUT buffers should not overflow using the above scheme in normal operation. There are a number of scenarios which could lead to the USB EPs being temporarily blocked such as the CPU having priority, retransmissions on the ISI bus, channels being enabled (*ChannelEn* bit of the *USBEPnDest* register) with data already in their associated endpoint buffers or short packets being sent on the USB. Care should be taken to ensure that the USB bandwidth is efficiently utilised at all times.

12.5.5 Implementation

12.5.5.1 CTRL Sub-block Partition

* Block Diagram

* Definition of I/Os

12.5.5.2 SCB Configuration Registers

The SCB register map is listed in Table 38. Registers are grouped according to which SCB sub-block their functionality is associated. All configuration registers reside in the CTRL sub-block. The Reset values in the table indicates the 32 bit hex value that will be returned when the CPU reads the associated address location after reset. All Registers pre-fixed with *Hc* refer to Host Controller Operational Registers, as defined in the OHCI Spec[19].

The SCB will only allow supervisor mode accesses to data space (i.e. *cpu_acode*[1:0] = b11). All other accesses will result in *scb_cpu_berr* being asserted.

TDB: Is read access necessary for ISI Rx/Tx buffers? Could implement the ISI interface as simple FIFOs as opposed to a memory interface.

Table 38. SCB control block configuration registers

Addre ss Offset from SCB_ base	Register	#Bits	Reset	Description
CTRL				
0x000	SCBResetN	4	0x0000000F	SCB software reset. Allows individual sub-blocks to be reset separately or together. Once a reset for a block has been initiated, by writing a 0 to the relevant register field, it can not be suppressed. Each field will be set after reset. Writing 0x0 to the SCBReset register will have the same effect as CPR generated hardware reset.
0x004	SCBGo	2	0x00000000	SCB Go. Allows the ISI and CTRL sub-blocks to be selected separately or together. When go is de-asserted for a particular sub-block, its statemachines are reset to their idle states and its interface signals are de-asserted. The sub-block counters and configuration registers retain their values. When go is asserted for a particular sub-block, its counters are reset. The sub-block configuration registers retain their values, i.e. they don't get reset. The sub-block statemachines and interface signals will return to their normal mode of operation. The CTRL field should be de-asserted before disabling the clock from any part of the SCB to avoid erroneous SCB DMA requests when the clock is enabled again. NOTE: This functionality has not been provided for the USBH and USBD sub-

				blocks because of the USB IP cores that they contain. We do not have direct control over the IP core statemachines and counters, and it would cause unpredictable behaviour if the cores were disabled in this way during operation.
0x008	SCBWakeupEn	2	0x00000000	USB/ISI WakeUpEnable register
0x00C	SCBISITxBufferArb	2	0x00000000	ISI transmit buffer access priority register.
0x010	SCBDebugSel[11:2]	10	0x00000000	SCB Debug select register.
0x014	USBEP0Dest	7	0x00000020	This register determines which of the data sinks the data arriving in EP0 should be routed to.
0x018	USBEP1Dest	7	0x00000021	Data sink mapping for USB EP1
0x01C	USBEP2Dest	7	0x0000003E	Data sink mapping for USB EP2
0x020	USBEP3Dest	7	0x0000003F	Data sink mapping for USB EP3
0x024	USBEP4Dest	7	0x00000023	Data sink mapping for USB EP4
0x028	DMA0BottomAdr[21:5]	17		DMAChannel0 bottom address register.
0x02C	DMA0TopAdr[21:5]	17		DMAChannel0 top address register.
0x030	DMA0CurrWPtr[21:5]	17		DMAChannel0 current write pointer.
0x034	DMA0IntAdr[21:5]	17		DMAChannel0 interrupt address register.
0x038	DMA0MaxAdr[21:5]	17		DMAChannel0 max address register.
0x03C	DMA1BottomAdr[21:5]	17		As per *DMA0BottomAdr*.
0x040	DMA1TopAdr[21:5]	17		As per *DMA0TopAdr*.
0x044	DMA1CurrWPtr[21:5]	17		As per *DMA0CurrWPtr*.
0x048	DMA1IntAdr[21:5]	17		As per *DMA0IntAdr*.
0x04C	DMA1MaxAdr[21:5]	17		As per *DMA0MaxAdr*.
0x050	DMAAccessEn	3	0x00000003	DMA access enable.

0x054	DMAStatus	4	0x00000000	DMA status register.
0x058	DMAMask	4	0x00000000	DMA mask register.
0x05C - 0x098	CPUISITxBuff[7:0]	32x8	n/a	CPU ISI transmit buffer. 32-byte packet buffer, containing the payload of a CPU sourced packet destined for transmission over the ISI. The CPU has full write access to the *CPUISITxBuff*. NOTE: The CPU does not have read access to *CPUISITxBuff*. This is because the CPU is the source of the data and to avoid arbitrating read access between the CPU and the CTRL sub-block. Any CPU reads from this address space will return 0x00000000.
0x09C	CPUISITxBuffCtrl	9	0x00000000	CPU ISI transmit buffer control register.
USBD				
0x100	USBDIntStatus	19	0x00000000	USBD Interrupt event status register.
0x104	USBDISIFIFOStatus	16	0x00000000	USBD ISI mapped OUT EP packet FIFO status register.
0x108	USBDDMA0FIFOStatus	8	0x00000000	USBD DMAChannel0 mapped OUT EP packet FIFO status register.
0x10C	USBDDMA1FIFOStatus	8	0x00000000	USBD DMAChannel1 mapped OUT EP packet FIFO status register.
0x110	USBDResume	1	0x00000000	USBD core resume register.
0x114	USBDSetup	4	0x00000000	USBD setup/configuration register.
0x118 - 0x154	USBDEp0InBuff[15:0]	32x16	n/a	USBD EP0-IN buffer. 64-byte packet buffer in the, containing the payload of a USB packet destined for EP0-IN. The CPU has full write access to the *USBDEp0InBuff*. NOTE: The CPU does not have read access to *USBDEp0InBuff*. This is because the CPU is the source of the data and to avoid arbitrating read access between the CPU and the USB device core. Any CPU reads from this

				address space will return 0x00000000.
0x158	USBDEp0InBuffCtrl	1	0x00000000	USBD EP0-IN buffer control register.
0x15C - 0x198	USBDEp5InBuff[15:0]	32x16	n/a	USBD EP5-IN buffer. As per *USBDEp0InBuff.*
0x19C	USBDEp5InBuffCtrl	1	0x00000000	USBD EP5-IN buffer control register.
0x1A0	USBDMask	19	0x00000000	USBD interrupt mask register.
0x1A4	USBDDebug	30	0x00000000	USBD debug register.
USBH				
0x200	HcRevision			Refer to [19] for #Bits, Reset, Description.
0x204	HcControl			Refer to [19] for #Bits, Reset, Description.
0x208	HcCommandStatus			Refer to [19] for #Bits, Reset, Description.
0x20C	HcInterruptStatus			Refer to [19] for #Bits, Reset, Description.
0x210	HcInterruptEnable			Refer to [19] for #Bits, Reset, Description.
0x214	HcInterruptDisable			Refer to [19] for #Bits, Reset, Description.
0x218	HcHCCA			Refer to [19] for #Bits, Reset, Description.
0x21C	HcPeriodCurrentED			Refer to [19] for #Bits, Reset, Description.
0x220	HcControlHeadED			Refer to [19] for #Bits, Reset, Description.
0x224	HcControlCurrentED			Refer to [19] for #Bits, Reset, Description.
0x228	HcBulkHeadED			Refer to [19] for #Bits, Reset, Description.
0x22C	HcBulkCurrentED			Refer to [19] for #Bits, Reset, Description.
0x230	HcDoneHead			Refer to [19] for #Bits, Reset, Description.
0x234	HcFmInterval			Refer to [19] for #Bits, Reset, Description.
0x238	HcFmRemaining			Refer to [19] for #Bits, Reset,

				Description.
0x23C	HcFmNumber			Refer to [19] for #Bits, Reset, Description.
0x240	HcPeriodicStart			Refer to [19] for #Bits, Reset, Description.
0x244	HcLSTheshold			Refer to [19] for #Bits, Reset, Description.
0x248	HcRhDescriptorA			Refer to [19] for #Bits, Reset, Description.
0x24C	HcRhDescriptorB			Refer to [19] for #Bits, Reset, Description.
0x250	HcRhStatus			Refer to [19] for #Bits, Reset, Description.
0x254	HcRhPortStatus[1]			Refer to [19] for #Bits, Reset, Description.
0x258	USBHStatus	3	0x00000000	USBH status register.
0x25C	USBHMask	2	0x00000000	USBH interrupt mask register.
0x260	USBHDebug	2	0x00000000	USBH debug register.
ISI				
0x300	ISICntrl	4	0x0000000B	ISI Control register
0x304	ISIId	4	0x00000000	ISIId for this SoPEC.
0x308	ISINumRetries	4	0x00000002	Number of ISI retransmissions register.
0x30C	ISIPingSchedule0	15	0x00000000	ISI Ping schedule 0 register.
0x310	ISIPingSchedule1	15	0x00000000	ISI Ping schedule 1 register.
0x314	ISIPingSchedule2	15	0x00000000	ISI Ping schedule 2 register.
0x318	ISITotalPeriod	4	0x0000000F	Reload value of the *ISITotalPeriod* counter.
0x31C	ISILocalPeriod	4	0x0000000F	Reload value of the *ISILocalPeriod* counter.
0x320	ISIIntStatus	4	0x00000000	ISI interrupt status register.
0x324	ISITxBuffStatus	27	0x00000000	ISI Tx buffer status register.
0x328	ISIRxBuffStatus	27	0x00000000	ISI Rx buffer status register.
0x32C	ISIMask	4	0x00000000	ISI Interrupt mask register.
0x330 - 0x34C	ISITxBuffEntry0[7:0]	32x8	n/a	ISI transmit Buff, packet entry #0. 32-byte packet entry in the *ISITxBuff,* containing the payload of an ISI Tx packet. CPU read access to *ISITxBuffEntry0* is provided for observability only i.e. CPU

				reads of the *ISITxBuffEntry0* do not alter the state of the buffer. The CPU does not have write access to the *ISITxBuffEntry0.*
0x350 - 0x36C	ISITxBuffEntry1[7:0]	32x8	n/a	ISI transmit Buff, packet entry #1. As per *ISITxBuffEntry0*.
0x370 - 0x38C	ISIRxBuffEntry0[7:0]	32x8	n/a	ISI receive Buff, packet entry #0. 32-byte packet entry in the *ISIRxBuff*, containing the payload of an ISI Rx packet. Note that the only error-free long packets are placed in the *ISIRxBuffEntry0*. Both ping and ACKs are consumed in the ISI. CPU access to *ISIRxBuffEntry0* is provided for observability only i.e. CPU reads of the *ISIRxBuffEntry0* do not alter the state of the buffer.
0x390 - 0x3AC	ISIRxBuffEntry1[7:0]	32x8	n/a	ISI receive Buff, packet entry #1. As per *ISIRxBuffEntry0*.
0x3B0	ISISubId0Seq	1	0x00000000	ISI sub ID 0 sequence bit register.
0x3B4	ISISubId1Seq	1	0x00000000	ISI sub ID 1 sequence bit register.
0x3B8	ISISubIdSeqMask	2	0x00000000	ISI sub ID sequence bit mask register.
0x3BC	ISINumPins	1	0x00000000	ISI number of pins register.
0x3C0	ISITurnAround	4	0x0000000F	ISI bus turn around register.
0x3C4	ISITShortReplyWin	5	0x0000001F	ISI short packet reply window.
0x3C8	ISITLongReplyWin	9	0x000001FF	ISI long packet reply window.
0x3CC	ISIDebug	4	0x00000000	ISI debug register.

A detailed description of each register format follows. The CPU has full read access to all registers. Write access to the fields of each register is defined as:

- Full: The CPU has full write access to the field, i.e. the CPU can write a 1 or a 0 to each bit.
- Clear: The CPU can clear the field by writing a 1 to each bit. Writing a 0 to this type of field will have no effect.
- None: The CPU has no write access to the field, i.e. a CPU write will have no effect on the field.

12.5.5.2.1 SCBResetN

Table 39. SCBResetN register format

Field Name	Bit(s)	write access	Description
CTRL	0	Full	*scb_ctrl* sub-block reset. Setting this field will reset the SCB control sub-block logic, including all configuration registers. 0 = reset 1 = default state
ISI	1	Full	*scb_isi* sub-block reset. Setting this field will reset the ISI sub-block logic. 0 = reset 1 = default state
USBH	2	Full	*scb_usbh* sub-block reset. Setting this field will reset the USB host controller core and associated logic. 0 = reset 1 = default state
USBD	3	Full	*scb_usbd* sub-block reset. Setting this field will reset the USB device controller core and associated logic. 0 = reset 1 = default state

12.5.5.2.2 SCBGo

Table 40. SCBGo register format

Field Name	Bit(s)	write access	Description
CTRL	0	Full	*scb_ctrl* sub-block go. 0 = halted 1 = running
ISI	1	Full	*scb_isi* sub-block go. 0 = halted 1 = running

12.5.5.2.3 SCBWakeUpEn

This register is used to gate the propagation of the USB and ISI reset signals to the CPR block.

Table 41. SCBWakeUpEn register format

Field Name	Bit(s)	write access	Description
USBWakeUpEn	0	Full	*usb_cpr_reset_n* propagation enable. 1 = enable 0 = disable

ISIWakeUpEn	1	Full	*isi_cpr_reset_n* propagation enable. 1 = enable 0 = disable

12.5.5.2.4 SCBISITxBufferArb

This register determines which source has priority at the ISITxBuffer interface on the ISI block. When a bit is set priority is given to the relevant source. When both bits have the same value, arbitration will be performed in a round-robin manner.

Table 42. SCBISITxBufferArb register format

Field Name	Bit(s)	write access	Description
CPUPriority	0	Full	CPU priority 1 = high priority 0 = low priority
USBPriority	1	Full	USB priority 1 = high priority 0 = low priority

12.5.5.2.5 SCBDebugSel

Contains address of the register selected for debug observation as it would appear on *cpu_adr*. The contents of the selected register are output in the *scb_cpu_data* bus while *cpu_scb_sel* is low and *scb_cpu_debug_valid* is asserted to indicate the debug data is valid. It is expected that a number of pseudo-registers will be made available for debug observation and these will be outlined with the implementation details.

Table 43. SCBDebugSel register format

Field Name	Bit(s)	write access	Description
CPUAdr	11:2	Full	*cpu_adr* register address.

12.5.5.2.6 USBEPnDest

This register description applies to *USBEP0Dest*, *USBEP1Dest*, *USBEP2Dest*, *USBEP3Dest*, *USBEP4Dest*. The SCB has two routing options for each packet received, based on the *DestISIId* associated with the packets source EP:

- To the DMA Manager
- To the ISI

The SCB map therefore does not need special fields to identify the DMAChannels on the ISIMaster SoPEC as this is taken care of by the SCB hardware. Thus the *USBEP0Dest* and *USBEP1Dest* registers should be programmed with 0x20 and 0x21 (for ISI0.0 and ISI0.1) respectively to ensure data arriving on these endpoints is moved directly to DRAM.

Table 44. USBEPnDest register format

Field Name	Bit(s)	Write access	Description
SequenceBit	0	Full	Sequence bit for packets going from USBEPn to DestISIId.DestISISubId. Every CPU write to this register initialises the value of the sequence bit and this is subsequently updated by the ISI after every successful long packet transmission.
DestISIId	4:1	Full	Destination ISI ID. Denotes the ISIId of the target SoPEC as per Table
DestISISubId	5	Full	Destination ISI sub ID. Indicates which DMAChannel of the target SoPEC the endpoint is mapped onto: 0 = DMAChannel0 1 = DMAChannel1
ChannelEn	6	Full	Communication channel enable bit for EPn. This enables/disables the communication channel for EPn. When disabled, the SCB will not accept USB packets addressed to EPn. 0 = Channel disabled 1 = Channel enabled

If the local SoPEC is connected to an external USB host, it is recommended that the EP0 communication channel should always remain enabled and mapped to DMAChannel0 on the local SoPEC, as this is intended as the primary control communication channel between the external USB host and the local SoPEC.

A SoPEC ISIMaster should map as many USB endpoints, under the control of the external host, as are required for the multi-SoPEC system it is part of. As already mentioned this mapping may be dynamically reconfigured.

12.5.5.2.7 DMAnBottomAdr

This register description applies to *DMA0BottomAdr* and *DMA1BottomAdr*.

Table 45. DMAnBottomAdr register format

Field Name	Bit(s)	Write access	Description
DMAnBottomAdr	21:5	Full	The 256-bit aligned DRAM address of the bottom of the circular buffer (inclusive) serviced by DMAChanneln

12.5.5.2.8 DMAnTopAdr

This register description applies to *DMA0TopAdr* and *DMA1TopAdr*.

Table 46. DMAnTopAdr register format

Field Name	Bit(s)	Write access	Description
DMAnTopAdr	21:5	Full	The 256-bit aligned DRAM address of the top of the circular buffer (inclusive) serviced by DMAChanneln

12.5.5.2.9 DMAnCurrWPtr

This register description applies to DMA0CurrWPtr and DMA1CurrWPtr.

Table 47. DMAnCurrWptr register format

Field Name	Bit(s)	Write access	Description
DMAnCurrWPtr	21:5	Full	The 256-bit aligned DRAM address of the next location DMAChannel0 will write to. This register is set by the CPU at the start of a DMA operation and dynamically updated by the DMA manager during the operation.

12.5.5.2.10 DMAnIntAdr

This register description applies to *DMA0IntAdr* and *DMA1IntAdr*.

Table 48. DMAnIntAdr register format

	Bit(s)	Write access	Description
DMAnIntAdr	21:5	Full	The 256-bit aligned DRAM address of the location that will trigger an interrupt when reached by DMAChanneln buffer.

12.5.5.2.11 DMAnMaxAdr

This register description applies to *DMA0MaxAdr* and *DMA1MaxAdr*.

Table 49. DMAnMaxAdr register format

Field Name	Bit(s)	Write access	Description
DMAnMaxAdr	21:5	Full	The 256-bit aligned DRAM address of the last free location that in the DMAChanneln circular buffer. DMAChannel0 transfers will stop when it reaches this address.

12.5.5.2.12 DMAAccessEn

This register enables DMA access for the various requestors, on a per channel basis.

Table 50. DMAAccessEn register format

Field Name	Bit(s)	Write access	Description
DMAChannel0En	0	Full	DMA Channel #0 access enable. This uni-directional write channel is used by the USBD and the ISI. 1 = enable 0 = disable
DMAChannel1En	1	Full	As per *USBDISI0En*.
DMAChannel2En	2	Full	DMA Channel #2 access enable. This bi-directional read/write channel is used by the USBH. 1 = enable 0 = disable

12.5.5.2.13 DMAStatus

The status bits are not sticky bits i.e. they reflect the 'live' status of the channel. *DMAChannelNIntAdrHit* and *DMAChannelNMaxAdrHit* status bits may only be cleared by writing to the relevant *DMAnIntAdr* or *DMAnMaxAdr* register.

Table 51. DMAStatus register format

Field Name	Bit(s)	Write access	Description
DMAChannel0IntAdrHit	0	None	DMA channel #0 interrupt address hit. 1= DMAChannel0 has reached the address contained in the *DMA0IntAdr* register. 0 = default state
DMAChannel0MaxAdrHit	1	None	DMA channel #0 max address hit. 1 = DMAChannel0 has reached the

			address contained in the *DMA0MaxAdr* register. 0 = default state
DMAChannel1IntAdrHit	3	None	As per *DMAChannel0IntAdrHit.*
DMAChannel1MaxAdrHit	4	None	As per *DMAChannel0MaxAdrHit.*

12.5.5.2.14 DMAMask register

All bits of the *DMAMask* are both readable and writable by the CPU. The DMA manager cannot alter the value of this register. All interrupts are generated in an edge sensitive manner i.e. the DMA manager will generate a *dma_icu_irq* pulse each time a status bit goes high and its corresponding mask bit is enabled.

Table 52. DMAMask register format

Field Name	Bit(s)	Write access	Description
DMAChannel0IntAdrHitIntEn	0	Full	*DMAChannel0IntAdrHit* status interrupt enable. 1 = enable 0 = disable
DMAChannel0MaxAdrHitIntEn	1	Full	*DMAChannel0MaxAdrHit* status interrupt enable. 1 = enable 0 = disable
DMAChannel1IntAdrHitIntEn	2	Full	As per *DMAChannel0IntAdrHitIntEn*
DMAChannel1MaxAdrHitIntEn	3	Full	As per *DMAChannel0MaxAdrHitIntEn*

12.5.5.2.15 CPUISITxBuffCtrl register

Table 53. CPUISITxBuffCtrl register format

Field Name	Bit(s)	Write access	Description
PktValid	0	full	This field should be set by the CPU to indicate the validity of the *CPUISITxBuff* contents. This field will be cleared by the SCB once the contents of the *CPUISITxBuff* has been copied to the *ISITxBuff.* NOTE: The CPU should not clear this field under normal operation. If the CPU clears this field during a packet transfer to the *ISITxBuff*, the transfer

			will be aborted - this is not recommended. 1 = valid packet. 0 = default state.
PktDesc	3:1	full	*PktDesc* field, as per Table , of the packet contained in the *CPUISITxBuff.* The CPU is responsible for maintaining the correct sequence bit value for each *ISIId.ISISubId* channel it communicates with. Only valid when *CPU-ISITxBuffCtrl.PktValid* = 1.
DestISIId	7:4	full	Denotes the *ISIId* of the target SoPEC as per Table .
DestISISubId	8	full	Indicates which DMAChannel of the target SoPEC the packet in the *CPUISITxBuff is destined for.* 1 = DMAChannel1 0 = DMAChannel0

12.5.5.2.16 USBDIntStatus

The *USBDIntStatus* register contains status bits that are related to conditions that can cause an interrupt to the CPU, if the corresponding interrupt enable bits are set in the *USBDMask* register. The field name extension *Sticky* implies that the status condition will remain registered until cleared by a CPU write of 1 to each bit of the field.

NOTE: There is no *Ep0IrregPktSticky* field because the default control EP will frequently receive packets that are not multiples of 32 bytes during normal operation.

Table 54. USBDIntStatus register format

Field Name	Bit(s)	Write access	Description
CoreSuspendSticky	0	Clear	Device core USB suspend flag. Sticky. 1 = USB suspend state. Set when device core *udcvci_suspend* signal transitions from 1 -> 0. 0 = default value.
CoreUSBResetSticky	1	Clear	Device core USB reset flag. Sticky. 1 = USB reset. Set when device core *udcvci_reset* signal transitions from 1 -> 0. 0 = default value.
CoreUSBSOFSticky	2	Clear	Device core USB Start Of Frame (SOF) flag. Sticky.

			1 = USB SOF. Set when device core *udcvci_sof* signal transitions from 1 -> 0 0 = default value.
CPUISITxBuffEmptySticky	3	Clear	CPU ISI transmit buffer empty flag. Sticky. 1 = empty. 0 = default value.
CPUEp0InBuffEmptySticky	4	Clear	CPU EP0 IN buffer empty flag. Sticky. 1 = empty. 0 = default value.
CPUEp5InBuffEmptySticky	5	Clear	CPU EP5 IN buffer empty flag. Sticky. 1 = empty. 0 = default value.
Ep0InNAKSticky	6	clear	EP0-IN NAK flag. Sticky This flag is set if the USB device core issues a read request for EP0-IN and there is not a valid packet present in the EP0-IN buffer. The core will therefore send a NAK response to the IN token that was received from external USB host. This is an indicator of any back-pressure on the USB caused by EP0-IN. 1 = NAK sent. 0 = default value
Ep5InNAKSticky	7	Clear	As per *Ep0InNAK.*
Ep0OutNAKSticky	8	Clear	EP0-OUT NAK flag. Sticky This flag is set if the USB device core issues a write request for EP0-OUT and there is no space in the OUT EP buffer for a the packet. The core will therefore send a NAK response to the OUT token that was received from external USB host. This is an indicator of any back-pressure on the USB caused by EP0-OUT. 1 = NAK sent. 0 = default value
Ep1OutNAKSticky	9	Clear	As per *Ep0OutNAK.*
Ep2OutNAKSticky	10	Clear	As per *Ep0OutNAK.*
Ep3OutNAKSticky	11	Clear	As per *Ep0OutNAK.*
Ep4OutNAKSticky	12	Clear	As per *Ep0OutNAK.*
Ep1IrregPktSticky	13	Clear	EP1-OUT irregular sized packet flag. Sticky.

			Indicates a packet that is not a multiple of 32 bytes in size was received by EP1-OUT. 1 = irregular sized packet received. 0 = default value.
Ep2IrregPktSticky	14	Clear	As per *Ep1IrregPktSticky.*
Ep3IrregPktSticky	15	Clear	As per *Ep1IrregPktSticky.*
Ep4IrregPktSticky	16	Clear	As per *Ep1IrregPktSticky.*
OutBuffOverFlowSticky	17	Clear	OUT EP buffer overflow flag. Sticky. This flag is set if the USB device core attempted to write a packet of more than 64 bytes to the OUT EP buffer. This is a fatal error, suggesting a problem in the USB device IP core. The SCB will take no further action. 1 = overflow condition detected. 0 = default value.
InBuffUnderRunSticky	18	clear	IN EP buffer underrun flag. Sticky. This flag is set if the USB device core attempted to read more data than was present from the IN EP buffer. This is a fatal error, suggesting a problem in the USB device IP core. The SCB will take no further action. 1 = underrun condition detected. 0 = default value.

12.5.5.2.17 USBDISIFIFOStatus

This register contains the status of the ISI mapped OUT EP packet FIFO. This is a secondary status register and will not cause any interrupts to the CPU.

Table 55. USBDISIFIFOStatus register format

5

Field Name	Bit(s)	Write access	Description
Entry0Valid	0	none	FIFO entry #0 valid field. This flag will be set by the USBD when the USB device core indicates the validity of packet entry #0 in the FIFO. 1 = valid USB packet in ISI OUT EP buffer 0. 0 = default value.
Entry0Source	3:1	none	FIFO entry #0 source field. Contains the EP associated with packet entry #0 in the FIFO. Binary Coded Decimal. Only valid when *ISIBuff0PktValid* = 1.

Entry1Valid	4	none	As per *Entry0Valid*.
Entry1Source	7:5	none	As per *Entry0Source*.
Entry2Valid	8	none	As per *Entry0Valid*.
Entry2Source	11:9	none	As per *Entry0Source*.
Entry3Valid	12	none	As per *Entry0Valid*.
Entry3Source	15:13	none	As per *Entry0Source*.

12.5.5.2.18 USBDDMA0FIFOStatus

This register description applies to *USBDDMA0FIFOStatus* and *USBDDMA1FIFOStatus.*

This register contains the status of the DMAChannelN mapped OUT EP packet FIFO. This is a secondary status register and will not cause any interrupts to the CPU.

Table 56. USBDDMANFIFOStatus register format

Field Name	Bit(s)	Write access	Description
Entry0Valid	0	none	FIFO entry #0 valid field. This flag will be set by the USBD when the USB device core indicates the validity of packet entry #0 in the FIFO. 1 = valid USB packet in ISI OUT EP buffer 0. 0 = default value.
Entry0Source	3:1	none	FIFO entry #0 source field. Contains the EP associated with packet entry #0 in the FIFO. Binary Coded Decimal. Only valid when *Entry0Valid* = 1.
Entry1Valid	4	none	As per *Entry0Valid*.
Entry1Source	7:5	none	As per *Entry0Source*.

12.5.5.2.19 USBDResume

This register causes the USB device core to initiate *resume* signalling to the external USB host.

Only applicable when the device core is in the *suspend* state.

Table 57. USBDResume register format

Field Name	Bit(s)	Write access	Description
USBDResume	0	full	USBD core resume register. The USBD will clear this register upon resume notification from the device core. 1 = generate resume signalling. 0 = default value.

12.5.5.2.20 USBDSetup

This register controls the general setup/configuration of the USBD.

Table 58. USBDSetup register format

Field Name	Bit(s)	write access	Description
Ep1IrregPktCntrl	0	full	EP 1 OUT irregular sized packet control. An irregular sized packet is defined as a packet that is not a multiple of 32 bytes. 1 = discard irregular sized packets. 0 = read 32 bytes from buffer, regardless of packet size.
Ep2IrregPktCntrl	1	full	As per Ep1IrregPktDiscard
Ep3IrregPktCntrl	2	full	As per Ep1IrregPktDiscard
Ep4IrregPktCntrl	3	full	As per Ep1IrregPktDiscard

12.5.5.2.21 USBDEpNInBuffCtrl register

This register description applies to *USBDEp0InBuffCtrl* and *USBDEp5InBuffCtrl*.

Table 59. USBDEpNInBuffCtrl register format

Field Name	Bit(s)	Write access	Description
PktValid	0	full	Setting this register validates the contents of *USBDEpNInBuff*. This field will be cleared by the SCB once the packet has been successfully transmitted to the external USB host. NOTE: The CPU should not clear this field under normal operation. If the CPU clears this field during a packet transfer to the *USB*, the transfer will be aborted - this is not recommended. 1 = valid packet. 0 = default state.

12.5.5.2.22 USBDMask

This register serves as an interrupt mask for all USBD status conditions that can cause a CPU interrupt. Setting a field enables interrupt generation for the associated status event. Clearing a field disables interrupt generation for the associated status event. All interrupts will be generated in an edge sensitive manner, i.e. when the associated status register transitions from 0 -> 1.

Table 60. USBDMask register format

Field Name	Bit(s)	Write access	Description
CoreSuspendStickyEn	0	full	*CoreSuspendSticky* status interrupt enable.
CoreUSBResetStickyEn	1	full	*CoreUSBResetSticky* status interrupt enable.
CoreUSBSOFStickyEn	2	full	*CoreUSBSOFSticky* status interrupt enable.
CPUISITxBuffEmptyStickyEn	3	full	*CPUISITxBuffEmptySticky* status interrupt enable.
CPUEp0InBuffEmptyStickyEn	4	full	*CPUEp0InBuffEmptySticky* status interrupt enable.

CPUEp5InBuffEmptyStickyEn	5	full	*CPUEp5InBuffEmptySticky* status interrupt enable.
Ep0InNAKStickyEn	6	full	*Ep0InNAKSticky* status interrupt enable.
Ep5InNAKStickyEn	7	full	*Ep5InNAKSticky* status interrupt enable.
Ep0OutNAKStickyEn	8	full	*Ep0OutNAKSticky* status interrupt enable.
Ep1OutNAKStickyEn	9	full	*Ep1OutNAKSticky* status interrupt enable.
Ep2OutNAKStickyEn	10	full	*Ep2OutNAKSticky* status interrupt enable.
Ep3OutNAKStickyEn	11	full	*Ep3OutNAKSticky* status interrupt enable.
Ep4OutNAKStickyEn	12	full	*Ep4OutNAKSticky* status interrupt enable.
Ep1IrregPktStickyEn	13	full	*Ep1IrregPktSticky* status interrupt enable.
Ep2IrregPktStickyEn	14	full	*Ep2IrregPktSticky* status interrupt enable.
Ep3IrregPktStickyEn	15	full	*Ep3IrregPktSticky* status interrupt enable.
Ep4IrregPktStickyEn	16	full	*Ep4IrregPktSticky* status interrupt enable.
OutBuffOverFlowStickyEn	17	full	*OutBuffOverFlowSticky* status interrupt enable.
InBuffUnderRunStickyEn	18	full	*InBuffUnderRunSticky* status interrupt enable.

12.5.5.2.23 USBDDebug

This register is intended for debug purposes only. Contains non-sticky versions of all interrupt capable status bits, which are referred to as *dynamic* in the table.

Table 61. USBDDebug register format

5

Field Name	Bit(s)	write access	Description
CoreTimeStamp	10:0	none	USB device core frame number.
CoreSuspend	11	none	Dynamic version of *CoreSuspendSticky*.
CoreUSBReset	12	none	Dynamic version of *CoreUSBResetSticky*.
CoreUSBSOF	13	none	Dynamic version of *CoreUSBSOFSticky*.
CPUISITxBuffEmpty	14	none	Dynamic version of *CPUISITxBuffEmptySticky*.
CPUEp0InBuffEmpty	15	none	Dynamic version of *CPUEp0InBuffEmptySticky*.
CPUEp5InBuffEmpty	16	none	Dynamic version of *CPUEp5InBuffEmptySticky*.
Ep0InNAK	17	none	Dynamic version of *Ep0InNAKSticky*.
Ep5InNAK	18	none	Dynamic version of *Ep5InNAKSticky*.
Ep0OutNAK	19	none	Dynamic version of *Ep0OutNAKSticky*.
Ep1OutNAK	20	none	Dynamic version of *Ep1OutNAKSticky*.
Ep2OutNAK	21	none	Dynamic version of *Ep2OutNAKSticky*.
Ep3OutNAK	22	none	Dynamic version of *Ep3OutNAKSticky*.
Ep4OutNAK	23	none	Dynamic version of *Ep4OutNAKSticky*.
Ep1IrregPkt	24	none	Dynamic version of *Ep1IrregPktSticky*.
Ep2IrregPkt	25	none	Dynamic version of *Ep2IrregPktSticky*.
Ep3IrregPkt	26	none	Dynamic version of *Ep3IrregPktSticky*.

Ep4IrregPkt	27	none	Dynamic version of *Ep4IrregPktSticky*.
OutBuffOverFlow	28	none	Dynamic version of *OutBuffOverFlowSticky*.
InBuffUnderRun	29	none	Dynamic version of *InBuffUnderRunSticky*.

12.5.5.2.24 USBHStatus

This register contains all status bits associated with the USBH. The field name extension *Sticky* implies that the status condition will remain registered until cleared by a CPU write.

Table 62. USBHStatus register format

Field Name	Bit(s)	Write access	Description
CoreIRQSticky	0	clear	HC core IRQ interrupt flag. Sticky Set when HC core *UHOSTC_IrqN* output signal transitions from 0 -> 1. Refer to OHCI spec for details on HC interrupt processing. 1 = IRQ interrupt from core. 0 = default value.
CoreSMISticky	1	clear	HC core SMI interrupt flag. Sticky Set when HC core *UHOSTC_SmiN* output signal transitions from 0 -> 1. Refer to OHCI spec for details on HC interrupt processing. 1 = SMI interrupt from HC. 0 = default value.
CoreBuffAcc	2	none	HC core buffer access flag. HC core *UHOSTC_BufAcc* output signal. Indicates whether the HC is accessing a descriptor or a buffer in shared system memory. 1 = buffer access 0 = descriptor access.

12.5.5.2.25 USBHMask

This register serves as an interrupt mask for all USBH status conditions that can cause a CPU interrupt. All interrupts will be generated in an edge sensitive manner, i.e. when the associated status register transitions from 0 -> 1.

Table 63. USBHMask register format

Field Name	Bit(s)	Write access	Description
CoreIRQIntEn	0	full	*CoreIRQSticky* status interrupt enable. 1 = enable. 0 = disable.
CoreSMIIntEn	1	full	*CoreSMISticky* status interrupt enable.

			1 = enable. 0 = disable.

12.5.5.2.26 USBHDebug

This register is intended for debug purposes only. Contains non-sticky versions of all interrupt capable status bits, which are referred to as *dynamic* in the table.

Table 64. USBHDebug register format

Field Name	Bit(s)	write access	Description
CoreIRQ	0	none	Dynamic version of *CoreIRQSticky*.
CoreSMI	1	None	Dynamic version of *CoreSMISticky*.

12.5.5.2.27 ISICntrl

This register controls the general setup/configuration of the ISI.

Note that the reset value of this register allows the SoPEC to automatically become an ISIMaster (*AutoMasterEnable* = 1) if any USB packets are received on endpoints 2-4. On becoming an ISIMaster the *ISIMasterSel* bit is set and any USB or CPU packets destined for other ISI devices are transmitted. The CPU can override this capability at any time by clearing the *AutoMasterEnable* bit.

Table 65. ISICntrl register format

Field Name	Bit(s)	Write access	Description
TxEnable	0	Full	ISI transmit enable. Enables ISI transmission of long or ping packets. ACKs may still be transmitted when this bit is 0. This is cleared by transmit errors and needs to be restarted by the CPU. 1 = Transmission enabled 0 = Transmission disabled
RxEnable	1	Full	ISI receive enable. Enables ISI reception. This is can only be cleared by the CPU and it is only anticipated that reception will be disabled when the ISI in not in use and the ISI pins are being used by the GPIO for another purpose. 1 = Reception enabled 0 = Reception disabled
ISIMasterSel	2	Full	ISI master select. Determines whether the SoPEC is an ISIMaster or not 1 = ISIMaster 0 = ISISlave

AutoMasterEnable	3	Full	ISI auto master enable. Enables the device to automatically become the ISIMaster if activity is detected on USB endpoints2-4. 1 = auto-master operation enabled 0 = auto-master operation disabled

12.5.5.2.28 ISIId

Table 66. ISIId register format

Field Name	Bit(s)	Write access	Description
ISIId	3:0	Full	ISIId for this SoPEC. SoPEC resets to being an ISISlave with ISIId0. 0xF (the broadcast ISIId) is an illegal value and should not be written to this register.

12.5.5.2.29 ISINumRetries

Table 67. ISINumRetries register format

Field Name	Bit(s)	Write access	Description
ISINumRetries	3:0	Full	Number of ISI retransmissions to attempt in response to an inferred NAK before aborting a long packet transmission

12.5.5.2.30 ISIPingScheduleN

This register description applies to *ISIPingSchedule0*, *ISIPingSchedule1* and *ISIPingSchedule2.*

Table 68. ISIPingScheduleN register format

Field Name	Bit(s)	Write access	Description
ISIPingSchedule	14:0	Full	Denotes which ISIIds will be receive ping packets. Note that bit0 refers to ISIId0, bit1 to ISIId1...bit14 to ISIId14.

12.5.5.2.31 ISITotalPeriod

Table 69. ISITotalPeriod register format

Field Name	Bit(s)	Write access	Description
ISITotalPeriod	3:0	Full	Reload value of the *ISITotalPeriod* counter

12.5.5.2.32 ISILocalPeriod

Table 70. ISILocalPeriod register format

Field Name	Bit(s)	Write access	Description
ISILocalPeriod	3:0	Full	Reload value of the *ISILocalPeriod* counter

12.5.5.2.33 ISIIntStatus

The *ISIIntStatus* register contains status bits that are related to conditions that can cause an interrupt to the CPU, if the corresponding interrupt enable bits are set in the *ISIMask* register.

Table 71. ISIIntStatus register

5

Field Name	Bit(s)	Write access	Description
TxErrorSticky	0	None	ISI transmit error flag. Sticky. Receiving ISI device would not accept the transmitted packet. Only set after *NumRetries* unsuccessful retransmissions. (excluding ping packets). This bit is cleared by the ISI after transmission has been re-enabled by the CPU setting the *TxEnable* bit of the *ISICntrl* register. 1= transmit error. 0 = default state.
RxFrameErrorSticky	1	Clear	ISI receive framing error flag. Sticky. This bit is set by the ISI when a framing error detected in the received packet, which can be caused by an incorrect *Start* or *Stop* field or by bit stuffing errors. 1 = framing error detected. 0 = default state.
RxCRCErrorSticky	2	Clear	ISI receive CRC error flag. This bit is set by the ISI when a CRC error is detected in an incoming packet. Other than dropping the errored packet ISI reception is unaffected by a CRC Error. 1 = CRC error 0 = default state.
RxBuffOverFlowSticky	3	Clear	ISI receive buffer over flow flag. Sticky. An overflow has occurred in the ISI receive buffer and a packet had to be dropped. 1 = over flow condition detected. 0 = default state.

12.5.5.2.34 ISITxBuffStatus

The *ISITxBuffStatus* register contains status bits that are related to the ISI Tx buffer. This is a secondary status register and will not cause any interrupts to the CPU.

Table 72. ISITxBuffStatus register format

Field Name	Bit(s)	Write access	Description
Entry0PktValid	0	None	ISI Tx buffer entry #0 packet valid flag. This flag will be set by the ISI when a valid ISI packet is written to entry #0 in the *ISITxBuff* for transmission over the ISI bus. A Tx packet is considered valid when it is 32 bytes in size and the ISI has written the packet header information to *Entry0PktDesc*, *Entry0DestISIId* and *Entry0DestISISubId*. 1 = packet valid. 0 = default value.
Entry0PktDesc	3:1	None	ISI Tx buffer entry #0 packet descriptor. PktDesc field as per Table for the packet entry #0 in the *ISITxBuff*. Only valid when *Entry0PktValid* = 1.
Entry0DestISIId	7:4	None	ISI Tx buffer entry #0 destination ISI ID. Denotes the ISIId of the target SoPEC as per Table . Only valid when *Entry0PktValid* = 1.
Entry0DestISISubId	8	None	ISI Tx buffer entry #0 destination ISI sub ID. Indicates which DMAChannel on the target SoPEC that packet entry #0 in the *ISITxBuff* is destined for. Only valid when *Entry0PktValid* = 1. 1 = DMAChannel1 0 = DMAChannel0
Entry1PktValid	9	None	As per *Entry0PktValid*.
Entry1PktDesc	12:10	None	As per *Entry0PktDesc*.
Entry1DestISIId	16:13	None	As per *Entry0DestISIId*.
Entry1DestISISubId	17	None	As per *Entry0DestISISubId*.

12.5.5.2.35 ISIRxBuffStatus

The *ISIRxBuffStatus* register contains status bits that are related to the ISI Rx buffer. This is a secondary status register and will not cause any interrupts to the CPU.

Table 73. ISIRxBuffStatus register format

Field Name	Bit(s)	Write access	Description
Entry0PktValid	0	None	ISI Rx buffer entry #0 packet valid flag. This flag will be set by the ISI when a valid ISI packet is received and written to entry #0 of the *ISIRxBuff*. A Rx packet is considered valid when it is 32 bytes in size and no framing or CRC errors were detected.

			1 = valid packet 0 = default value
Entry0PktDesc	3:1	None	ISI Rx buffer entry #0 packet descriptor. PktDesc field as per Table for packet entry #0 of the *ISIRxBuff*. Only valid when *Entry0PktValid* = 1.
Entry0DestISIId	7:4	None	ISI Rx buffer 0 destination ISI ID. Denotes the ISIId of the target SoPEC as per Table . This should always correspond to the local SoPEC ISIId. Only valid when *Entry0PktValid* = 1.
Entry0DestISISubId	8	None	ISI Rx buffer 0 destination ISI sub ID. Indicates which DMAChannel on the target SoPEC that entry #0 of the *ISIRxBuff* is destined for. Only valid when *Entry0PktValid* = 1. 1 = DMAChannel1 0 = DMAChannel0
Entry1PktValid	9	None	As per *Entry0PktValid*.
Entry1PktDesc	12:10	None	As per *Entry0PktDesc*.
Entry1DestISIId	16:13	None	As per *Entry0DestISIId*.
Entry1DestISISubId	17	None	As per *Entry0DestISISubId*.

12.5.5.2.36 ISIMask register

An interrupt will be generated in an edge sensitive manner i.e. the ISI will generate an *isi_icu_irq* pulse each time a status bit goes high and the corresponding bit of the *ISIMask* register is enabled.

Table 74. ISIMask register

5

Field Name	Bit(s)	Write access	Description
TxErrorIntEn	0	Full	*TxErrorSticky* status interrupt enable. 1 = enable. 0 = disable.
RxFrameErrorIntEn	1	Full	*RxFrameErrorSticky* status interrupt enable. 1 = enable. 0 = disable.
RxCRCErrorIntEn	2	Full	*RxCRCErrorSticky* status interrupt enable. 1 = enable. 0 = disable.
RxBuffOverFlowIntEn	3	Full	*RxBuffOverFlowSticky* status interrupt enable. 1 = enable. 0 = disable.

12.5.5.2.37 ISISubIdNSeq

This register description applies to *ISISubId0Seq* and *ISISubId0Seq*.

Table 75. ISISubIdNSeq register format

Field Name	Bit(s)	Write access	Description
ISISubIdNSeq	0	Full	ISI sub ID channel N sequence bit. This bit may be initialised by the CPU but is updated by the ISI each time an error-free long packet is received.

12.5.5.2.38 ISISubIdSeqMask

Table 76. ISISubIdSeqMask register format

Field Name	Bit(s)	Write access	Description
ISISubIdSeq0Mask	0	Full	ISI sub ID channel 0 sequence bit mask. Setting this bit ensures that the sequence bit will be ignored for incoming packets for the ISISubId. 1 = ignore sequence bit. 0 = default state.
ISISubIdSeq1Mask	1	Full	As per *ISISubIdSeq0Mask*.

12.5.5.2.39 ISINumPins

Table 77. ISINumPins register format

Field Name	Bit(s)	Write access	Description
ISINumPins	0	Full	Select number of active ISI pins. 1 = 4 pins 0 = 2 pins

12.5.5.2.40 ISITurnAround

The ISI bus turnaround time will reset to its maximum value of 0xF to provide a safer starting mode for the ISI bus. This value should be set to a value that is suitable for the physical implementation of the ISI bus, i.e. the lowest turn around time that the physical implementation will allow without significant degradation of signal integrity.

Table 78. ISITurnAround register format

Field Name	Bit(s)	Write access	Description
ISITurnAround	3:0	Full	ISI bus turn around time in ISI clock cycles (32MHz).

12.5.5.2.41 ISIShortReplyWin

The ISI short packet reply window time will reset to its maximum value of 0x1F to provide a safer starting mode for the ISI bus. This value should be set to a value that will allow for expected frequency of bit stuffing and receiver response timing.

Table 79. ISIShortReplyWin register format

Field Name	Bit(s)	Write access	Description
ISIShortReplyWin	4:0	Full	ISI long packet reply window in ISI clock cycles (32MHz).

12.5.5.2.42 ISILongReplyWin

The ISI long packet reply window time will reset to its maximum value of 0x1FF to provide a safer starting mode for the ISI bus. This value should be set to a value that will allow for expected frequency of bit stuffing and receiver response timing.

Table 80. ISILongReplyWin register format

Field Name	Bit(s)	Write access	Description
ISILongReplyWin	8:0	Full	ISI long packet reply window in ISI clock cycles (32MHz).

12.5.5.2.43 ISIDebug

This register is intended for debug purposes only. Contains non-sticky versions of all interrupt capable status bits, which are referred to as *dynamic* in the table.

Table 81. ISIDebug register format

Field Name	Bit(s)	Write access	Description
TxError	0	None	Dynamic version of *TxErrorSticky*.
RxFrameError	1	None	Dynamic version of *RxFrameErrorSticky*.
RxCRCError	2	None	Dynamic version of *RxCRCErrorSticky*.
RxBuffOverFlow	3	None	Dynamic version of *RxBuffOverFlowSticky*.

12.5.5.3 CPU Bus Interface

12.5.5.4 Control Core Logic

12.5.5.5 DIU Bus Interface

12.6 DMA REGS

All of the circular buffer registers are 256-bit word aligned as required by the DIU. The *DMAnBottomAdr* and *DMAnTopAdr* registers are inclusive i.e. the addresses contained in those registers form part of the circular buffer. The *DMAnCurrWPtr* always points to the next location the DMA manager will write to so interrupts are generated whenever the DMA manager reaches the address in either the *DMAnIntAdr* or *DMAnMaxAdr* registers rather than when it actually writes to these locations. It therefore can not write to the location in the *DMAnMaxAdr* register.

SCB Map regs

The SCB map is configured by mapping a USB endpoint on to a data sink. This is performed on a endpoint basis i.e. each endpoint has a configuration register to allow its data sink be selected. Mapping an endpoint on to a data sink does not initiate any data flow - each endpoint/data sink needs to be enabled by writing to the appropriate configuration registers for the USBD, ISI and DMA manager.

13. General Purpose IO (GPIO)

13.1 OVERVIEW

The General Purpose IO block (GPIO) is responsible for control and interfacing of GPIO pins to the rest of the SoPEC system. It provides easily programmable control logic to simplify control of GPIO functions. In all there are 32 GPIO pins of which any pin can assume any output or input function. Possible output functions are

- 4 Stepper Motor control Outputs
- 12 Brushless DC Motor Control Output (total of 2 different controllers each with 6 outputs)
- 4 General purpose high drive pulsed outputs capable of driving LEDs.
- 4 Open drain IOs used for LSS interfaces
- 4 Normal drive low impedance IOs used for the ISI interface in Multi-SoPEC mode

Each of the pins can be configured in either input or output mode, each pin is independently controlled. A programmable de-glitching circuit exists for a fixed number of input pins. Each input is a schmidt trigger to increase noise immunity should the input be used without the de-glitch circuit.

The mapping of the above functions and their alternate use in a slave SoPEC to GPIO pins is shown in Table 82 below.

Table 82. GPIO pin type

GPIO pin(s)	Pin IO Type	Default Function
gpio[3:0]	Normal drive, low impedance IO (35 Ohm), Integrated pull-up resistor	Pins 1 and 0 in ISI Mode, pins 2 and 3 in input mode
gpio[7:4]	High drive, normal impedance IO (65 Ohm), intended for LED drivers	Input Mode
gpio[31:8]	Normal drive, normal impedance IO (65 Ohm), no pull-up	Input Mode

13.2 Stepper Motor control

The motor control pins can be directly controlled by the CPU or the motor control logic can be used to generate the phase pulses for the stepper motors. The controller consists of two central counters from which the control pins are derived. The central counters have several registers (see Table) used to configure the cycle period, the phase, the duty cycle, and counter granularity.

There are two motor master counters (0 and 1) with identical features. The period of the master counters are defined by the *MotorMasterClkPeriod[1:0]* and *MotorMasterClkSrc* registers i.e. both master counters are derived from the same *MotorMasterClkSrc*. The *MotorMasterClkSrc* defines

the timing pulses used by the master counters to determine the timing period. The *MotorMasterClkSrc* can select clock sources of 1µs,100µs,10ms and *pclk* timing pulses.

The *MotorMasterClkPeriod[1:0]* registers are set to the number of timing pulses required before the timing period re-starts. Each master counter is set to the relevant *MotorMasterClkPeriod* value and counts down a unit each time a timing pulse is received.

The master counters reset to *MotorMasterClkPeriod* value and count down. Once the value hits zero a new value is reloaded from the *MotorMasterClkPeriod[1:0]* registers. This ensures that no master clock glitch is generated when changing the clock period.

Each of the IO pins for the motor controller are derived from the master counters. Each pin has independent configuration registers. The *MotorMasterClkSelect[3:0]* registers define which of the two master counters to use as the source for each motor control pin. The master counter value is compared with the configured *MotorCtrlLow* and *MotorCtrlHigh* registers (bit fields of the *MotorCtrlConfig* register). If the count is equal to *MotorCtrlHigh* value the motor control is set to 1, if the count is equal to *MotorCtrlLow* value the motor control pin is set to 0.

This allows the phase and duty cycle of the motor control pins to be varied at *pclk* granularity.

The motor control generators keep a working copy of the *MotorCtrlLow, MotorCtrlHigh* values and update the configured value to the working copy when it is safe to do so. This allows the phase or duty cycle of a motor control pin to be safely adjusted by the CPU without causing a glitch on the output pin.

Note that when reprogramming the *MotorCtrlLow, MotorCtrlHigh* registers to reorder the sequence of the transition points (e.g changing from low point less than high point to low point greater than high point and vice versa) care must still taken to avoid introducing glitching on the output pin.

13.3 LED CONTROL

LED lifetime and brightness can be improved and power consumption reduced by driving the LEDs with a pulsed rather than a DC signal. The source clock for each of the LED pins is a 7.8kHz (128µs period) clock generated from the 1µs clock pulse from the Timers block. The *LEDDutySelect* registers are used to create a signal with the desired waveform. Unpulsed operation of the LED pins can be achieved by using CPU IO direct control, or setting *LEDDutySelect* to 0. By default the LED pins are controlled by the LED control logic.

13.4 LSS INTERFACE VIA GPIO

In some SoPEC system configurations one or more of the LSS interfaces may not be used. Unused LSS interface pins can be reused as general IO pins by configuring the *IOModeSelect* registers. When a mode select register for a particular GPIO pin is set to 23,22,21,20 the GPIO pin is connected to LSS control IOs 3 to 0 respectively.

13.5 ISI INTERFACE VIA GPIO

In Multi-SoPEC mode the SCB block (in particular the ISI sub-block) requires direct access to and from the GPIO pins. Control of the ISI interface pins is determined by the *IOModeSelect* registers. When a mode select register for a particular GPIO pin is set to 27,26,25,24 the GPIO pin connected to the ISI control bits 3 to 0 respectively. By default the GPIO pins 1 to 0 are directly controlled by the ISI block.

In single SoPEC systems the pins can be re-used by the GPIO.

13.6 CPU GPIO CONTROL

The CPU can assume direct control of any (or all) of the IO pins individually. On a per pin basis the CPU can turn on direct access to the pin by configuring the *IOModeSelect* register to CPU direct mode. Once set the IO pin assumes the direction specified by the *CpuIODirection* register. When in output mode the value in register *CpuIOOut* will be directly reflected to the output driver. When in input mode the status of the input pin can be read by reading *CpuIOIn* register. When writing to the *CpuIOOut* register the value being written is XORed with the current value in *CpuIOOut.* The CPU can also read the status of the 10 selected de-glitched inputs by reading the *CpuIOInDeGlitch* register.

13.7 PROGRAMMABLE DE-GLITCHING LOGIC

Each IO pin can be filtered through a de-glitching logic circuit, the pin that the de-glitching logic is connected to is configured by the *InputPinSelect* registers. There are 10 de-glitching circuits, so a maximum of 10 input pin can be de-glitched at any time.

The de-glitch circuit can be configured to sample the IO pin for a predetermined time before concluding that a pin is in a particular state. The exact sampling length is configurable, but each de-glitch circuit must use one of two possible configured values (selected by *DeGlitchSelect*). The sampling length is the same for both high and low states. The *DeGlitchCount* is programmed to the number of system time units that a state must be valid for before the state is passed on. The time units are selected by *DeGlitchClkSel* and can be one of 1μs,100μs,10ms and *pclk* pulses.

For example if *DeGlitchCount* is set to 10 and *DeGlitchClkSel* set to 3, then the selected input pin must consistently retain its value for 10 system clock cycles (*pclk*) before the input state will be propagated from *CpuIOIn* to *CpuIOInDeglitch.*

13.8 INTERRUPT GENERATION

Any of the selected input pins (selected by *InputPinSelect*) can generate an interrupt from the raw or deglitched version of the input pin. There are 10 possible interrupt sources from the GPIO to the interrupt controller, one interrupt per input pin. The *InterruptSrcSelect* register determines whether the raw input or the deglitched version is used as the interrupt source.

The interrupt type, masking and priority can be programmed in the interrupt controller.

13.9 FREQUENCY ANALYSER

The frequency analyser measures the duration between successive positive edges on a selected input pin (selected by *InputPinSelect*) and reports the last period measured (*FreqAnaLastPeriod*) and a running average period (*FreqAnaAverage*).

The running average is updated each time a new positive edge is detected and is calculated by *FreqAnaAverage* = (*FreqAnaAverage* / 8) * 7 + *FreqAnaLastPeriod* / 8.

The analyser can be used with any selected input pin (or its deglitched form), but only one input at a time can be selected. The input is selected by the *FreqAnaPinSelect* (range of 0 to 9) and its deglitched form can be selected by *FreqAnaPinFormSelect.*

13.10 BRUSHLESS DC (BLDC) MOTOR CONTROLLERS

The GPIO contains 2 brushless DC (BLDC) motor controllers. Each controller consists of 3 hall inputs, a direction input, and six possible outputs. The outputs are derived from the input state and a pulse width modulated (PWM) input from the Stepper Motor controller, and is given by the truth table in Table 83.

Table 83. Truth Table for BLDC Motor Controllers

direction	hc	hb	ha	q6	q5	q4	q3	q2	q1
0	0	0	1	0	0	0	1	PWM	0
0	0	1	1	PWM	0	0	1	0	0
0	0	1	0	PWM	0	0	0	0	1
0	1	1	0	0	0	PWM	0	0	1
0	1	0	0	0	1	PWM	0	0	0
0	1	0	1	0	1	0	0	PWM	0
0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0
1	0	0	1	0	0	PWM	0	0	1
1	0	1	1	PWM	0	0	0	0	1
1	0	1	0	PWM	0	0	1	0	0
1	1	1	0	0	0	0	1	PWM	0
1	1	0	0	0	1	0	0	PWM	0
1	1	0	1	0	1	PWM	0	0	0
1	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

All inputs to a BLDC controller must be de-glitched. Each controller has its inputs hardwired to de-glitch circuits. Controller 1 hall inputs are de-glitched by circuits 2 to 0, and its direction input is de-glitched by circuit 3. Controller 2 inputs are de-glitched by circuits 6 to 4 for hall inputs and 7 for direction input.

Each controller also requires a PWM input. The stepper motor controller outputs are reused, output 0 is connected to BLDC controller 1, and output 1 to BLDC controller 2.

The controllers have two modes of operation, internal and external direction control (configured by *BLDCMode*). If a controller is in external direction mode the direction input is taken from a de-glitched circuit, if it is in internal direction mode the direction input is configured by the *BLDCDirection* register.

The BLDC controller outputs are connected to the GPIO output pins by configuring the *IOModeSelect* register for each pin. e.g Setting the mode register to 8 will connect q1 Controller 1 to drive the pin.

13.11 IMPLEMENTATION

13.11.1 Definitions of I/O

Table 84. I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
tim_pulse[2:0]	3	In	Timers block generated timing pulses. 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse
CPU Interface			
cpu_adr[8:2]	8	In	CPU address bus. Only 7 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
gpio_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_gpio_sel	1	In	Block select from the CPU. When *cpu_gpio_sel* is high both *cpu_adr* and *cpu_dataout* are valid
gpio_cpu_rdy	1	Out	Ready signal to the CPU. When *gpio_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the GPIO block and for a read cycle this means the data on *gpio_cpu_data* is valid.
gpio_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
gpio_cpu_debug_valid	1	Out	Debug Data valid on *gpio_cpu_data* bus. Active high
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
IO Pins			
gpio_o[31:0]	32	Out	General purpose IO output to IO driver
gpio_i[31:0]	32	In	General purpose IO input from IO receiver
gpio_e[31:0]	32	Out	General purpose IO output control. Active high driving
GPIO to LSS			

lss_gpio_dout[1:0]	2	In	LSS bus data output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
gpio_lss_din[1:0]	2	Out	LSS bus data input Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_e[1:0]	2	In	LSS bus data output enable, active high Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_clk[1:0]	2	In	LSS bus clock output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
GPIO to ISI			
gpio_isi_din[1:0]	2	Out	Input data from IO receivers to ISI.
isi_gpio_dout[1:0]	2	In	Data output from ISI to IO drivers
isi_gpio_e[1:0]	2	In	GPIO ISI pins output enable (active high) from ISI interface
usbh_gpio_power_en	1	In	Port Power enable from the USB host core, active high
gpio_usbh_over_current	1	Out	Over current detect to the USB host core, active high
Miscellaneous			
gpio_icu_irq[9:0]	10	Out	GPIO pin interrupts
gpio_cpr_wakeup	1	Out	SoPEC wakeup to the CPR block active high.
Debug			
debug_data_out[31:0]	32	In	Output debug data to be muxed on to the GPIO pins
debug_cntrl[31:0]	32	In	Control signal for each GPIO bound debug data line indicating whether or not the debug data should be selected by the pin mux

13.11.2 Configuration registers

The configuration registers in the GPIO are programmed via the CPU interface. Refer to section 11.4.3 on page 96 for a description of the protocol and timing diagrams for reading and writing registers in the GPIO. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the GPIO. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *gpio_cpu_data*. Table 85 lists the configuration registers in the GPIO block

Table 85. GPIO Register Definition

Address GPIO_base +	Register	#bits	Reset	Description
0x000-0x07C	IOModeSelect[31:0]	32x5	See Table for default values	Specifies the mode of operation for each GPIO pin. One 5 bit bus per pin. Possible assignment values and correspond controller outputs are as follows Value - Controlled by 3 to 0 - Output, LED controller 4 to 1 7 to 4 - Output Stepper Motor control 4-1 13 to 8 - Output BLDC 1 Motor control 6-1 19 to 14 - Output BLDC 2 Motor control 6-1 23 to 20 - LSS control 4-1 27 to 24 - ISI control 4-1 28 - CPU Direct Control 29 - USB power enable output 30 - Input Mode
0x080-0xA4	InputPinSelect[9:0]	10x5	0x00	Specifies which pins should be selected as inputs. Used to select the pin source to the DeGlitch Circuits.
CPU IO Control				
0x0B0	CpuIOUserModeMask	32	0x0000_0000	User Mode Access Mask to CPU GPIO control register. When 1 user access is enabled. One bit per gpio pin. Enables access to *CpuIODirection*, *CpuIOOut* and *CpuIOIn* in user mode.
0x0B4	CpuIOSuperModeMask	32	0xFFFF_FFFF	Supervisor Mode Access Mask to CPU GPIO control register. When 1 supervisor access is enabled. One bit per gpio pin. Enables access to *CpuIODirection*, *CpuIOOut* and *CpuIOIn* in supervisor mode.
0x0B8	CpuIODirection	32	0x0000_0000	Indicates the direction of each IO pin, when controlled by the CPU 0 - Indicates Input Mode 1 - Indicates Output Mode
0x0BC	CpuIOOut	32	0x0000_0000	Value used to drive output pin in CPU direct mode. bits31:0 - Value to drive on output GPIO pins When written to the register assumes the

				new value XORed with the current value.
0x0C0	CpuIOIn	32	External pin value	Value received on each input pin regardless of mode. Read Only register.
0x0C4	CpuDeGlitchUserModeMask	10	0x000	User Mode Access Mask to *CpuIOInDeglitch* control register. When 1 user access is enabled, otherwise bit reads as zero.
0x0C8	CpuIOInDeglitch	10	0x000	Deglitched version of selected input pins. The input pins are selected by the *InputPinSelect* register. Note that after reset this register will reflect the external pin values 256 *pclk* cycles after they have stabilized. Read Only register.
Deglitch control				
0x0D0-0x0D4	DeGlitchCount[1:0]	2x8	0xFF	Deglitch circuit sample count in *DeGlitchClkSrc* selected units.
0x0D8-0x0DC	DeGlitchClkSrc[1:0]	2x2	0x3	Specifies the unit use of the GPIO deglitch circuits: 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse 3 - *pclk*
0x0E0	DeGlitchSelect	10	0x000	Specifies which deglitch count (*DeGlitchCount*) and unit select (*DeGlitchClkSrc*) should be used with each de-glitch circuit 0 - Specifies *DeGlitchCount[0]* and *DeGlitchClkSrc[0]* 1 - Specifies *DeGlitchCount[1]* and *DeGlitchClkSrc[1]*
Motor Control				
0x0E4	MotorCtrlUserModeEnable	1	0x0	User Mode Access enable to Motor control configuration registers. When 1 user access is enabled. Enables user access to *MotorMasterClkPeriod*, *MotorMasterClkSrc*, *MotorDutySelect*, *MotorPhaseSelect*, *MotorMasterClockEnable*, *Motor-MasterClkSelect, BLDCMode* and

				BLDCDirection registers
0x0E8-0x0EC	MotorMasterClkPeriod[1:0]	2x16	0x0000	Specifies the motor controller master clock periods in *MotorMasterClkSrc* selected units
0x0F0	MotorMasterClkSrc	2	0x0	Specifies the unit use by the motor controller master clock generator: 0 - 1 μs pulse 1 - 100 μs pulse 2 - 10 ms pulse 3 - *pclk*
0x0F4-0x100	MotorCtrlConfig[3:0]	4x32	0x0000_0000	Specifies the transition points in the clock period for each motor control pin. One register per pin bits 15:0 - *MotorCtrlLow*, high to low transition point bits 31:16 - *MotorCtrlHigh*, low to high transition point
0x104	MotorMasterClkSelect	4	0x0	Specifies which motor master clock should be used as a pin generator source 0 - Clock derived from *MotorMasterClockPeriod[0]* 1 -Clock derived from *MotorMasterClockPeriod[1]*
0x108	MotorMasterClockEnable	2	0x0	Enable the motor master clock counter. When 1 count is enabled Bit 0 - Enable motor master clock 0 Bit 1 - Enable motor master clock 1
BLDC Motor Controllers				
0x10C	BLDCMode	2	0x0	Specifies the Mode of operation of the BLDC Controller. One bit per Controller. 0- External direction control 1- Internal direction control
0x110	BLDCDirection	2	0x0	Specifies the direction input of the BLDC controller. Only used when BLDC controller is an internal direction control mode. One bit per controller.
LED control				
0x114	LEDCtrlUserModeEnable	4	0x0	User Mode Access enable to LED control configuration registers. When 1 user access is enabled.

				One bit per *LEDDutySelect* select register.
0x118-0x124	LEDDutySelect [3:0]	4x3	0x0	Specifies the duty cycle for each LED control output. See Figure 54 for encoding details. The *LEDDutySelect[3:0]* registers determine the duty cycle of the LED controller outputs
Frequency Analyser				
0x130	FreqAnaUserModeEnable	1	0x0	User Mode Access enable to Frequency analyser configuration registers. When 1 user access is enabled. Controls access to *FreqAnaPinFormSelect*, *FreqAnaLastPeriod*, *FreqAnaAverage* and *FreqAnaCountInc.*
0x134	FreqAnaPinSelect	4	0x00	Selects which selected input should be used for the frequency analyses.
0x138	FreqAnaPinFormSelect	1	0x0	Selects if the frequency analyser should use the raw input or the deglitched form. 0 - Deglitched form of input pin 1 - Raw form of input pin
0x13C	FreqAnaLastPeriod	16	0x0000	Frequency Analyser last period of selected input pin.
0x140	FreqAnaAverage	16	0x0000	Frequency Analyser average period of selected input pin.
0x144	FreqAnaCountInc	20	0x0000 0	Frequency Analyser counter increment amount. For each clock cycle no edge is detected on the selected input pin the accumulator is incremented by this amount.
0x148	FreqAnaCount	32	0x0000 _0000	Frequency Analyser running counter (Working register)
Miscellaneous				
0x150	InterruptSrcSelect	10	0x3FF	Interrupt source select.1 bit per selected input. Determines whether the interrupt source is direct form the selected input pin or the deglitched version. Input pins are selected by the *DeGlitchPinSelect* register. 0 - Selected input direct 1 - Deglitched selected input
0x154	DebugSelect[8:2]	7	0x00	Debug address select. Indicates the address of the register to report on the

				gpio_cpu_data bus when it is not otherwise being used.
0x158-0x15C	MotorMasterCount[1:0]	2x16	0x0000	Motor master clock counter values. Bus 0 - Master clock count 0 Bus 1 - Master clock count 1 Read Only registers
0x160	WakeUpInputMask	10	0x000	Indicates which deglitched inputs should be considered to generate the CPR wakeup. Active high
0x164	WakeUpLevel	1	0	Defines the level to detect on the masked GPIO inputs to generate a wakeup to the CPR 0 - Level 0 1 - Level 1
0x168	USBOverCurrentPinSelect	4	0x00	Selects which deglitched input should be used for the USB over current detect.

13.11.2.1 Supervisor and user mode access

The configuration registers block examines the CPU access type (*cpu_acode* signal) and determines if the access is allowed to that particular register, based on configured user access registers. If an access is not allowed the GPIO will issue a bus error by asserting the *gpio_cpu_berr* signal.

All supervisor and user program mode accesses will result in a bus error.

Access to the *CpuIODirection*, *CpuIOOut* and *CpuIOIn* is filtered by the *CpuIOUserModeMask* and *CpuIOSuperModeMask* registers. Each bit masks access to the corresponding bits in the *CpuIO** registers for each mode, with *CpuIOUserModeMask* filtering user data mode access and *CpuIOSuperModeMask* filtering supervisor data mode access.

The addition of the *CpuIOSuperModeMask* register helps prevent potential conflicts between user and supervisor code read modify write operations. For example a conflict could exist if the user code is interrupted during a read modify write operation by a supervisor ISR which also modifies the *CpuIO** registers.

An attempt to write to a disabled bit in user or supervisor mode will be ignored, and an attempt to read a disabled bit returns zero. If there are no user mode enabled bits then access is not allowed in user mode and a bus error will result. Similarly for supervisor mode.

When writing to the *CpuIOOut* register, the value being written is XORed with the current value in the *CpuIOOut* register, and the result is reflected on the GPIO pins.

The pseudocode for determining access to the *CpuIOOut* register is shown below. Similar code could be shown for the *CpuIODirection* and *CpuIOIn* registers. Note that when writing to *CpuIODirection* data is deposited directly and not XORed with the existing data (as in the *CpuIOOut* case).

```
if (cpu_acode == SUPERVISOR_DATA_MODE) then
  // supervisor mode
  if (CpuIOSuperModeMask[31:0] == 0 ) then
    // access is denied, and bus error
    gpio_cpu_berr = 1
  elsif (cpu_rwn == 1) then
    // read mode (no filtering needed)
    gpio_cpu_data[31:0] = CpuIOOut[31:0]
  else
    // write mode,filtered by mask
    mask[31:0]                    =    (cpu_dataout[31:0]    &
CpuIOSuperModeMask[31:0])
    CpuIOOut[31:0]  =  (cpu_dataout[31:0]  ^     mask[31:0]  )
//bitwise XOR operator
elsif (cpu_acode == USER_DATA_MODE) then
  // user datamode
  if (CpuIOUserModeMask[31:0] == 0 ) then
    // access is denied, and bus error
    gpio_cpu_berr = 1
  elsif (cpu_rwn == 1) then
    // read mode, filtered by mask
    gpio_cpu_data          =     (     CpuIOOut[31:0]      &
CpuIOUserModeMask[31:0])
  else
    // write mode,filtered by mask
    mask[31:0]                    =    (cpu_dataout[31:0]    &
CpuIOUserModeMask[31:0])
    CpuIOOut[31:0]  =  (cpu_dataout[31:0]  ^     mask[31:0]  )
//bitwise XOR operator
else
  // access is denied, bus error
  gpio_cpu_berr = 1
```

Table 86 details the access modes allowed for registers in the GPIO block. In supervisor mode all registers are accessible. In user mode forbidden accesses will result in a bus error (*gpio_cpu_berr* asserted).

Table 86. GPIO supervisor and user access modes

Register Address	Registers	Access Permitted
0x000-0x07C	IOModeSelect[31:0]	Supervisor data mode only

0x080-0x94	InputPinSelect[9:0]	Supervisor data mode only
CPU IO Control		
0x0B0	CpuIOUserModeMask	Supervisor data mode only
0x0B4	CpuIOSuperModeMask	Supervisor data mode only
0x0B8	CpuIODirection	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x0BC	CpuIOOut	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x0C0	CpuIOIn	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x0C4	CpuDeGlitchUserModeMask	Supervisor data mode only
0x0C8	CpuIOInDeglitch	CpuDeGlitchUserModeMask filtered. Unrestricted Supervisor data mode access
Deglitch control		
0x0D0-0x0D4	DeGlitchCount[1:0]	Supervisor data mode only
0x0D8-0x0DC	DeGlitchClkSrc[1:0]	Supervisor data mode only
0x0E0	DeGlitchSelect	Supervisor data mode only
Motor Control		
0x0E4	MotorCtrlUserModeEnable	Supervisor data mode only
0x0E8-0x0EC	MotorMasterClkPeriod[1:0]	MotorCtrlUserModeEnable enabled.
0x0F0	MotorMasterClkSrc	MotorCtrlUserModeEnable enabled.
0x0F4-0x100	MotorCtrlConfig[3:0]	MotorCtrlUserModeEnable enabled
0x104	MotorMasterClkSelect	MotorCtrlUserModeEnable enabled
0x108	MotorMasterClockEnable	MotorCtrlUserModeEnable enabled
BLDC Motor Controllers		
0x10C	BLDCMode	MotorCtrlUserModeEnable Enabled
0x110	BLDCDirection	MotorCtrlUserModeEnable Enabled
LED control		
0x114	LEDCtrlUserModeEnable	Supervisor data mode only
0x118-0x124	LEDDutySelect[3:0]	LEDCtrlUserModeEnable[3:0] enabled
Frequency Analyser		
0x130	FreqAnaUserModeEnable	Supervisor data mode only
0x134	FreqAnaPinSelect	FreqAnaUserModeEnable enabled
0x138	FreqAnaPinFormSelect	FreqAnaUserModeEnable enabled
0x13C	FreqAnaLastPeriod	FreqAnaUserModeEnable enabled
0x140	FreqAnaAverage	FreqAnaUserModeEnable enabled

0x144	FreqAnaCountInc	FreqAnaUserModeEnable enabled
0x148	FreqAnaCount	FreqAnaUserModeEnable enabled
Miscellaneous		
0x150	InterruptSrcSelect	Supervisor data mode only
0x154	DebugSelect[8:2]	Supervisor data mode only
0x158-0x15C	MotorMasterCount[1:0]	Supervisor data mode only
0x160	WakeUpInputMask	Supervisor data mode only
0x164	WakeUpLevel	Supervisor data mode only
0x168	USBOverCurrentPinSelect	Supervisor data mode only

13.11.3 GPIO partition

13.11.4 IO control

The IO control block connects the IO pin drivers to internal signalling based on configured setup registers and debug control signals.

```
// Output Control
for (i=0; i<32 ; i++) {
if (debug_cntrl[i] == 1) then   // debug mode
  gpio_e[i] = 1;gpio_o[i] =debug_data_out[i]
else // normal mode
  case io_mode_select[i] is
    0 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[0]          // LED
output 1
    1 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[1]          // LED
output 2
    2 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[2]          // LED
output 3
    3 : gpio_e[i] =1 ;gpio_o[i] =led_ctrl[3]          // LED
output 4
    4 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[0]      // Stepper
Motor Control 1
    5 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[1]      // Stepper
Motor Control 2
    6 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[2]      // Stepper
Motor Control 3
    7 : gpio_e[i] =1 ;gpio_o[i] =motor_ctrl[3]      // Stepper
Motor Control 4
    8 : gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][0]    // BLDC
Motor Control 1,output 1
    9 : gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][1]    // BLDC
Motor Control 1,output 2
```

```
        10: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][2]     // BLDC
Motor Control 1,output 3
        11: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][3]     // BLDC
Motor Control 1,output 4
        12: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][4]     // BLDC
Motor Control 1,output 5
        13: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[0][5]     // BLDC
Motor Control 1,output 6
        14: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][0]     // BLDC
Motor Control 2,output 1
        15: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][1]     // BLDC
Motor Control 2,output 2
        16: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][2]     // BLDC
Motor Control 2,output 3
        17: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][3]     // BLDC
Motor Control 2,output 4
        18: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][4]     // BLDC
Motor Control 2,output 5
        19: gpio_e[i] =1 ;gpio_o[i] =bldc_ctrl[1][5]     // BLDC
Motor Control 2,output 6
        20: gpio_e[i] =1 ;gpio_o[i] =lss_gpio_clk[0]    // LSS Clk
0
        21: gpio_e[i] =1 ;gpio_o[i] =lss_gpio_clk[1]    // LSS Clk
1
        22:       gpio_e[i]       =lss_gpio_e[0]       ;gpio_o[i]
=lss_gpio_dout[0];  // LSS Data 0
            gpio_lss_din[0] = gpio_i[i]
        23:       gpio_e[i]       =lss_gpio_e[1]       ;gpio_o[i]
=lss_gpio_dout[1];  // LSS Data 1
            gpio_lss_din[1] = gpio_i[i]
        24:       gpio_e[i]       =isi_gpio_e[0]       ;gpio_o[i]
=isi_gpio_dout[0];  // ISI Control 1
            gpio_isi_din[0] = gpio_i[i]
        25:       gpio_e[i]       =isi_gpio_e[1]       ;gpio_o[i]
=isi_gpio_dout[1];  // ISI Control 2
            gpio_isi_din[1] = gpio_i[i]
        26:       gpio_e[i]       =isi_gpio_e[2]       ;gpio_o[i]
=isi_gpio_dout[2];  // ISI Control 3
            gpio_isi_din[2] = gpio_i[i]
        27:       gpio_e[i]       =isi_gpio_e[3]       ;gpio_o[i]
=isi_gpio_dout[3];  // ISI Control 4
```

```
        gpio_isi_din[3] = gpio_i[i]
    28: gpio_e[i] =cpu_io_dir[i] ;gpio_o[i] =cpu_io_out[i];
// CPU Direct
    29:   gpio_e[i]   =1   ;gpio_o[i]   =usbh_gpio_power_en
// USB host power enable
    30:       gpio_e[i]       =0       ;gpio_o[i]       =0
// Input only mode
  end case
  // all gpio are always readable by the CPU
  cpu_io_in[i] = gpio_i[i];
  }
```

The input selection pseudocode, for determining which pin connects to which de-glitch circuit.

```
for( i=0 ;i < 10 ; i++) {
  pin_num           = input_pin_select[i]
  deglitch_input[i] = gpio_i[pin_num]
  }
```

The *gpio_usbh_over_current* output to the USB core is driven by a selected deglitched input (configured by the *USBOverCurrentPinSelect* register).

```
index = USBOverCurrentPinSelect
gpio_usbh_over_current = cpu_io_in_deglitch[index]
```

13.11.5 Wakeup generator

The wakeup generator compares the deglitched inputs with the configured mask (WakeUpInputMask) and level (WakeUpLevel), and determines whether to generate a wakeup to the CPR block.

```
for (i =0;i<10; i++) {
  if (wakeup_level = 0) then  // level 0 active
    wakeup   =   wakeup   OR   wakeup_input_mask[i]   AND   NOT
cpu_io_in_deglitch[i]
  else                          // level 1 active
    wakeup    =    wakeup    OR    wakeup_input_mask[i]    AND
cpu_io_in_deglitch[i]
  }
// assign the output
gpio_cpr_wakeup = wakeup
```

13.11.6 LED pulse generator

The pulse generator logic consists of a 7-bit counter that is incremented on a 1μs pulse from the timers block (*tim_pulse[0]*). The LED control signal is generated by comparing the count value with the configured duty cycle for the LED (*led_duty_sel*).

The logic is given by:

```
for (i=0 i<4 ;i++) { // for each LED pin
  // period divided into 8 segments
  period_div8 = cnt[6:4];
  if (period_div8 < led_duty_sel[i]) then
    led_ctrl[i] = 1
  else
    led_ctrl[i] = 0
  }
// update the counter every 1us pulse
if (tim_pulse[0] == 1) then
  cnt ++
```

13.11.7 Stepper Motor control

The motor controller consists of 2 counters, and 4 phase generator logic blocks, one per motor control pin. The counters decrement each time a timing pulse (*cnt_en*) is received. The counters start at the configured clock period value (*motor_mas_clk_period*) and decrement to zero. If the counters are enabled (via *motor_mas_clk_enable*), the counters will automatically restart at the configured clock period value, otherwise they will wait until the counters are re-enabled.

The timing pulse period is one of *pclk*, 1μs, 100μs, 1ms depending on the *motor_mas_clk_sel* signal. The counters are used to derive the phase and duty cycle of each motor control pin.

```
// decrement logic
if (cnt_en == 1) then
  if ((mas_cnt == 0) AND (motor_mas_clk_enable == 1)) then
    mas_cnt = motor_mas_clk_period[15:0]
  elsif ((mas_cnt == 0) AND (motor_mas_clk_enable == 0)) then
    mas_cnt = 0
  else
    mas_cnt --
else   // hold the value
  mas_cnt = mas_cnt
```

The phase generator block generates the motor control logic based on the selected clock generator (*motor_mas_clk_sel*) the motor control high transition point (*curr_motor_ctrl_high*) and the motor control low transition point (*curr_motor_ctrl_low*).

The phase generator maintains current copies of the *motor_ctrl_config* configuration value (*motor_ctrl_config[31:16]* becomes *curr_motor_ctrl_high* and *motor_ctrl_config[15:0]* becomes

curr_motor_ctrl_low). It updates these values to the current register values when it is safe to do so without causing a glitch on the output motor pin.

Note that when reprogramming the *motor_ctrl_config* register to reorder the sequence of the transition points (e.g changing from low point less than high point to low point greater than high point and vice versa) care must taken to avoid introducing glitching on the output pin.

There are 4 instances one per motor control pin.

The logic is given by:

```
// select the input counter to use
if (motor_mas_clk_sel == 1) then
  count  = mas_cnt[1]
else
  count  = mas_cnt[0]
// Generate the phase and duty cycle
if (count == curr_motor_ctrl_low) then
  motor_ctrl = 0
elsif (count == curr_motor_ctrl_high) then
  motor_ctrl = 1
else
  motor_ctrl = motor_ctrl // remain the same
// update the current registers at period boundary
if (count == 0) then
  curr_motor_ctrl_high  =  motor_ctrl_config[31:16]          //
update to new high value
  curr_motor_ctrl_low   =  motor_ctrl_config[15:0]           //
update to new high value
```

13.11.8 Input deglitch

The input deglitch logic rejects input states of duration less than the configured number of time units (*deglitch_cnt*), input states of greater duration are reflected on the output *cpu_io_in_deglitch*. The time units used (either *pclk*, 1µs, 100µs, 1ms) by the deglitch circuit is selected by the *deglitch_clk_src* bus.

There are 2 possible sets of *deglitch_cnt* and *deglitch_clk_src* that can be used to deglitch the input pins. The values used are selected by the *deglitch_sel* signal.

There are 10 deglitch circuits in the GPIO. Any GPIO pin can be connected to a deglitch circuit. Pins are selected for deglitching by the *InputPinSelect* registers.

Each selected input can be used to generate an interrupt. The interrupt can be generated from the raw input signal (*deglitch_input*) or a deglitched version of the input (*cpu_io_in_deglitch*). The interrupt source is selected by the *interrupt_src_select* signal.

The counter logic is given by

```
if (deglitch_input != deglitch_input_delay) then
```

```
    cnt        = deglitch_cnt
    output_en = 0
  elsif (cnt == 0 ) then
    cnt        = cnt
    output_en = 1
  elsif (cnt_en == 1) then
    cnt --
    output_en = 0
```

13.11.9 Frequency Analyser

The frequency analyser block monitors a selected deglitched input (*cpu_io_in_deglitch*) or a direct selected input (*deglitch_input*) and detects positive edges. The selected input is configured by *FreqAnaPinSelect* and *FreqAnaPinFormSel* registers. Between successive positive edges detected on the input it increments a counter (*FreqAnaCount*) by a programmed amount (*FreqAnaCountInc*) on each clock cycle. When a positive edge is detected the *FreqAnaLastPeriod* register is updated with the top 16 bits of the counter and the counter is reset. The frequency analyser also maintains a running average of the *FreqAnaLastPeriod* register. Each time a positive edge is detected on the input the *FreqAnaAverage* register is updated with the new calculated *FreqAnaLastPeriod.* The average is calculated as 7/8 the current value plus 1/8 of the new value. The *FreqAnaLastPeriod, FreqAnaCount* and *FreqAnaAverage* registers can be written to by the CPU.

The pseudocode is given by

```
if ((pin == 1) AND pin_delay ==0 ))then     // positive edge
detected
  freq_ana_lastperiod[15:0] = freq_ana_count[31:16]
  freq_ana_average[15:0]          =   freq_ana_average[15:0]   -
freq_ana_average[15:3]
                                                              +
freq_ana_lastperiod[15:3]
  freq_ana_count[15:0]       = 0
else
  freq_ana_count[31:0]              =   freq_ana_count[31:0]   +
freq_ana_count_inc[19:0]
// implement the configuration register write
if (wr_last_en == 1) then
  freq_ana_lastperiod = wr_data
elsif (wr_average_en == 1 ) then
  freq_ana_average = wr_data
elsif (wr_freq_count_en == 1) then
  freq_ana_count = wr_data
```

13.11.10 BLDC Motor Controller

The BLDC controller logic is identical for both instances, only the input connections are different. The logic implements the truth table shown in Table . The six *q* outputs are combinationally based on the *direction*, *ha*, *hb*, *hc* and *pwm* inputs. The direction input has 2 possible sources selected by the mode, the pseudocode is as follows

```
// determine if in internal or external direction mode
if (mode == 1) then          // internal mode
  direction = int_direction
else                         // external mode
  direction = ext_direction
```

14 Interrupt Controller Unit (ICU)

The interrupt controller accepts up to N input interrupt sources, determines their priority, arbitrates based on the highest priority and generates an interrupt request to the CPU. The ICU complies with the interrupt acknowledge protocol of the CPU. Once the CPU accepts an interrupt (i.e. processing of its service routine begins) the interrupt controller will assert the next arbitrated interrupt if one is pending.

Each interrupt source has a fixed vector number N, and an associated configuration register, *IntReg[N]*. The format of the *IntReg[N]* register is shown in Table 87 below.

Table 87. IntReg[N] register format

Field	bit(s)	Description
Priority	3:0	Interrupt priority
Type	5:4	Determines the triggering conditions for the interrupt 00 - Positive edge 10 - Negative edge 01 - Positive level 11 - Negative level
Mask	6	Mask bit. 1 - Interrupts from this source are enabled, 0 - Interrupts from this source are disabled. Note that there may be additional masks in operation at the source of the interrupt.
Reserved	31:7	Reserved. Write as 0.

Once an interrupt is received the interrupt controller determines the priority and maps the programmed priority to the appropriate CPU priority levels, and then issues an interrupt to the CPU. The programmed interrupt priority maps directly to the LEON CPU interrupt levels. Level 0 is no interrupt. Level 15 is the highest interrupt level.

14.1 INTERRUPT PREEMPTION

With standard LEON pre-emption an interrupt can only be pre-empted by an interrupt with a higher priority level. If an interrupt with the same priority level (1 to 14) as the interrupt being serviced becomes pending then it is not acknowledged until the current service routine has completed. Note that the level 15 interrupt is a special case, in that the LEON processor will continue to take level 15 interrupts (i.e re-enter the ISR) as long as level 15 is asserted on the *icu_cpu_ilevel*. Level 0 is also a special case, in that LEON consider level 0 interrupts as no interrupt, and will not issue an acknowledge when level 0 is presented on the *icu_cpu_ilevel* bus.

Thus when pre-emption is required, interrupts should be programmed to different levels as interrupt priorities of the same level have no guaranteed servicing order. Should several interrupt sources be programmed with the same priority level, the lowest value interrupt source will be serviced first and so on in increasing order.

The interrupt is directly acknowledged by the CPU and the ICU automatically clears the pending bit of the lowest value pending interrupt source mapped to the acknowledged interrupt level.

All interrupt controller registers are only accessible in supervisor data mode. If the user code wishes to mask an interrupt it must request this from the supervisor and the supervisor software will resolve user access levels.

14.2 INTERRUPT SOURCES

The mapping of interrupt sources to interrupt vectors (and therefore *IntReg[N]* registers) is shown in Table 88 below. Please refer to the appropriate section of this specification for more details of the interrupt sources.

Table 88. Interrupt sources vector table

Vector	Source	Description
0	Timers	WatchDog Timer Update request
1	Timers	Generic Timer 1 interrupt
2	Timers	Generic Timer 2 interrupt
3	PCU	PEP Sub-system Interrupt- TE finished band
4	PCU	PEP Sub-system Interrupt- LBD finished band
5	PCU	PEP Sub-system Interrupt- CDU finished band
6	PCU	PEP Sub-system Interrupt- CDU error
7	PCU	PEP Sub-system Interrupt- PCU finished band
8	PCU	PEP Sub-system Interrupt- PCU Invalid address interrupt
9	PHI	PEP Sub-system Interrupt- PHI Line Sync Interrupt
10	PHI	PEP Sub-system Interrupt- PHI Buffer underrun
11	PHI	PEP Sub-system Interrupt- PHI Page finished
12	PHI	PEP Sub-system Interrupt- PHI Print ready
13	SCB	USB Host interrupt
14	SCB	USB Device interrupt
15	SCB	ISI interrupt

16	SCB	DMA interrupt
17	LSS	LSS interrupt, LSS interface 0 interrupt request
18	LSS	LSS interrupt, LSS interface 1 interrupt request
19-28	GPIO	GPIO general purpose interrupts
29	Timers	Generic Timer 3 interrupt

14.3 IMPLEMENTATION

14.3.1 Definitions of I/O

Table 89. Interrupt Controller Unit I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU interface			
cpu_adr[7:2]	6	In	CPU address bus. Only 6 bits are required to decode the address space for the ICU block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
icu_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_icu_sel	1	In	Block select from the CPU. When *cpu_icu_sel* is high both *cpu_adr* and *cpu_dataout* are valid
icu_cpu_rdy	1	Out	Ready signal to the CPU. When *icu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the ICU block and for a read cycle this means the data on *icu_cpu_data* is valid.
icu_cpu_ilevel[3:0]	4	Out	Indicates the priority level of the current active interrupt.
cpu_iack	1	In	Interrupt request acknowledge from the LEON core.
cpu_icu_ilevel[3:0]	4	In	Interrupt acknowledged level from the LEON core
icu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access

			11 - Supervisor data access
icu_cpu_debug_valid	1	Out	Debug Data valid on *icu_cpu_data* bus. Active high
Interrupts			
tim_icu_wd_irq	1	In	Watchdog timer interrupt signal from the Timers block
tim_icu_irq[2:0]	3	In	Generic timer interrupt signals from the Timers block
gpio_icu_irq[9:0]	10	In	GPIO pin interrupts
usb_icu_irq[1:0]	2	In	USB host and device interrupts from the SCB Bit 0 - USB Host interrupt Bit 1 - USB Device interrupt
isi_icu_irq	1	In	ISI interrupt from the SCB
dma_icu_irq	1	In	DMA interrupt from the SCB
lss_icu_irq[1:0]	2	In	LSS interface interrupt request
cdu_finishedband	1	In	Finished band interrupt request from the CDU
cdu_icu_jpegerror	1	In	JPEG error interrupt from the CDU
lbd_finishedband	1	In	Finished band interrupt request from the LBD
te_finishedband	1	In	Finished band interrupt request from the TE
pcu_finishedband	1	In	Finished band interrupt request from the PCU
pcu_icu_address_invalid	1	In	Invalid address interrupt request from the PCU
phi_icu_underrun	1	In	Buffer underrun interrupt request from the PHI
phi_icu_page_finish	1	In	Page finished interrupt request from the PHI
phi_icu_print_rdy	1	In	Print ready interrupt request from the PHI
phi_icu_linesync_int	1	In	Line sync interrupt request from the PHI

14.3.2 Configuration registers

The configuration registers in the ICU are programmed via the CPU interface. Refer to section 11.4 on page 96 for a description of the protocol and timing diagrams for reading and writing registers in the ICU. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the ICU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *icu_pcu_data*. Table 90 lists the configuration registers in the ICU block.

The ICU block will only allow supervisor data mode accesses (i.e. *cpu_acode[1:0]* = *SUPERVISOR_DATA*). All other accesses will result in *icu_cpu_berr* being asserted.

Table 90. ICU Register Map

Address	Register	#bits	Reset	Description

ICU_base +				
0x00 – 0x74	IntReg[29:0]	30x7	0x00	Interrupt vector configuration register
0x88	IntClear	30	0x0000 _0000	Interrupt pending clear register. If written with a one it clears corresponding interrupt Bits[30:0] - Interrupts sources 30 to 0 (Reads as zero)
0x90	IntPending	30	0x0000 _0000	Interrupt pending register. (Read Only) Bits[30:0]- Interrupts sources 30 to 0
0xA0	IntSource	5	0x1F	Indicates the interrupt source of the last acknowledged interrupt. The *NoInterrupt* value is defined as all bits set to one. (Read Only)
0xC0	DebugSelect[7:2]	6	0x00	Debug address select. Indicates the address of the register to report on the *icu_cpu_data* bus when it is not otherwise being used.

14.3.3 ICU partition

14.3.4 Interrupt detect

The ICU contains multiple instances of the interrupt detect block, one per interrupt source. The interrupt detect block examines the interrupt source signal, and determines whether it should generate request pending (*int_pend*) based on the configured interrupt type and the interrupt source conditions. If the interrupt is not masked the interrupt will be reflected to the interrupt arbiter via the *int_active* signal. Once an interrupt is pending it remains pending until the interrupt is accepted by the CPU or it is level sensitive and gets removed. Masking a pending interrupt has the effect of removing the interrupt from arbitration but the interrupt will still remain pending.

When the CPU accepts the interrupt (using the normal ISR mechanism), the interrupt controller automatically generates an interrupt clear for that interrupt source (*cpu_int_clear*). Alternatively if the interrupt is masked, the CPU can determine pending interrupts by polling the *IntPending* registers. Any active pending interrupts can be cleared by the CPU without using an ISR via the *IntClear* registers.

Should an interrupt clear signal (either from the interrupt clear unit or the CPU) and a new interrupt condition happen at the same time, the interrupt will remain pending. In the particular case of a level sensitive interrupt, if the level remains the interrupt will stay active regardless of the clear signal.

The logic is shown below:

```
mask          = int_config[6]
type          = int_config[5:4]
int_pend       = last_int_pend                    // the last pending
interrupt
// update the pending FF
// test for interrupt condition
```

```
if (type == NEG_LEVEL) then
  int_pend = NOT(int_src)
elsif (type == POS_LEVEL)
  int_pend = int_src
elsif ((type == POS_EDGE ) AND (int_src == 1) AND
(last_int_src == 0))
  int_pend = 1
elsif ((type == NEG_EDGE ) AND (int_src == 0) AND
(last_int_src == 1))
  int_pend = 1
elsif ((int_clear == 1 )OR (cpu_int_clear==1)) then
  int_pend = 0
else
  int_pend = last_int_pend // stay the same as before
// mask the pending bit
if (mask == 1) then
  int_active = int_pend
else
  int_active = 0
// assign the registers
last_int_src  = int_src
last_int_pend = int_pend
```

14.3.5 Interrupt arbiter

The interrupt arbiter logic arbitrates a winning interrupt request from multiple pending requests based on configured priority. It generates the interrupt to the CPU by setting *icu_cpu_ilevel* to a non-zero value. The priority of the interrupt is reflected in the value assigned to *icu_cpu_ilevel,* the higher the value the higher the priority, 15 being the highest, and 0 considered no interrupt.

```
// arbitrate with the current winner
int_ilevel       = 0
for (i=0;i<30;i++) {
  if ( int_active[i] == 1) then {
    if (int_config[i][3:0] > win_int_ilevel[3:0] ) then
      win_int_ilevel[3:0] = int_config[i][3:0]
      }
    }
  }
// assign the CPU interrupt level
int_ilevel = win_int_ilevel[3:0]
```

14.3.6 Interrupt clear unit

The interrupt clear unit is responsible for accepting an interrupt acknowledge from the CPU, determining which interrupt source generated the interrupt, clearing the pending bit for that source and updating the *IntSource* register.

When an interrupt acknowledge is received from the CPU, the interrupt clear unit searches through each interrupt source looking for interrupt sources that match the acknowledged interrupt level (*cpu_icu_ilevel*) and determines the winning interrupt (lower interrupt source numbers have higher priority). When found the interrupt source pending bit is cleared and the *IntSource* register is updated with the interrupt source number.

The LEON interrupt acknowledge mechanism automatically disables all other interrupts temporarily until it has correctly saved state and jumped to the ISR routine. It is the responsibility of the ISR to re-enable the interrupts. To prevent the *IntSource* register indicating the incorrect source for an interrupt level, the ISR must read and store the *IntSource* value before re-enabling the interrupts via the Enable Traps (ET) field in the Processor State Register (PSR) of the LEON.

See section 11.9 on page 132 for a complete description of the interrupt handling procedure.

After reset the state machine remains in *Idle* state until an interrupt acknowledge is received from the CPU (indicated by *cpu_iack*). When the acknowledge is received the state machine transitions to the *Compare* state, resetting the source counter (*cnt*) to the number of interrupt sources.

While in the *Compare* state the state machine cycles through each possible interrupt source in decrementing order. For each active interrupt source the programmed priority (*int_priority[cnt][3:0]*) is compared with the acknowledged interrupt level from the CPU (*cpu_icu_ilevel*), if they match then the interrupt is considered the new winner. This implies the last interrupt source checked has the highest priority, e.g interrupt source zero has the highest priority and the first source checked has the lowest priority. After all interrupt sources are checked the state machine transitions to the *IntClear* state, and updates the *int_source* register on the transition.

Should there be no active interrupts for the acknowledged level (e.g. a level sensitive interrupt was removed), the *IntSource* register will be set to *NoInterrupt* . *NoInterrupt* is defined as the highest possible value that *IntSource* can be set to (in this case 0x1F), and the state machine will return to *Idle*.

The exact number of compares performed per clock cycle is dependent the number of interrupts, and logic area to logic speed trade-off, and is left to the implementer to determine. A comparison of all interrupt sources must complete within 8 clock cycles (determined by the CPU acknowledge hardware).

When in the *IntClear* state the state machine has determined the interrupt source to clear (indicated by the *int_source* register). It resets the pending bit for that interrupt source, transitions back to the *Idle* state and waits for the next acknowledge from the CPU.

The minimum time between successive interrupt acknowledges from the CPU is 8 cycles.

15 Timers Block (TIM)

The Timers block contains general purpose timers, a watchdog timer and timing pulse generator for use in other sections of SoPEC.

15.1 WATCHDOG TIMER

The watchdog timer is a 32 bit counter value which counts down each time a timing pulse is received. The period of the timing pulse is selected by the *WatchDogUnitSel* register. The value at any time can be read from the *WatchDogTimer* register and the counter can be reset by writing a non-zero value to the register. When the counter transitions from 1 to 0, a system wide reset will be triggered as if the reset came from a hardware pin.

The watchdog timer can be polled by the CPU and reset each time it gets close to 1, or alternatively a threshold (*WatchDogIntThres*) can be set to trigger an interrupt for the watchdog timer to be serviced by the CPU. If the *WatchDogIntThres* is set to N, then the interrupt will be triggered on the N to N-1 transition of the *WatchDogTimer*. This interrupt can be effectively masked by setting the threshold to zero. The watchdog timer can be disabled, without causing a reset, by writing zero to the *WatchDogTimer* register.

15.2 TIMING PULSE GENERATOR

The timing block contains a timing pulse generator clocked by the system clock, used to generate timing pulses of programmable periods. The period is programmed by accessing the *TimerStartValue* registers. Each pulse is of one system clock duration and is active high, with the pulse period accurate to the system clock frequency. The periods after reset are set to 1us, 100us and 100 ms.

The timing pulse generator also contains a 64-bit free running counter that can be read or reset by accessing the *FreeRunCount* registers. The free running counter can be used to determine elapsed time between events at system clock accuracy or could be used as an input source in low-security random number generator.

15.3 GENERIC TIMERS

SoPEC contains 3 programmable generic timing counters, for use by the CPU to time the system. The timers are programmed to a particular value and count down each time a timing pulse is received. When a particular timer decrements from 1 to 0, an interrupt is generated. The counter can be programmed to automatically restart the count, or wait until re-programmed by the CPU. At any time the status of the counter can be read from *GenCntValue*, or can be reset by writing to *GenCntValue* register. The auto-restart is activated by setting the *GenCntAuto* register, when activated the counter restarts at *GenCntStartValue*. A counter can be stopped or started at any time, without affecting the contents of the *GenCntValue* register, by writing a 1 or 0 to the relevent *GenCntEnable* register.

15.4 IMPLEMENTATION

15.4.1 Definitions of I/O

Table 91. Timers block I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
tim_pulse[2:0]	3	Out	Timers block generated timing pulses, each one *pclk*

			wide 0 - Nominal 1μs pulse 1 - Nominal 100 μs pulse 2 - Nominal 10ms pulse
CPU interface			
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for the ICU block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
tim_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_tim_sel	1	In	Block select from the CPU. When *cpu_tim_sel* is high both *cpu_adr* and *cpu_dataout* are valid
tim_cpu_rdy	1	Out	Ready signal to the CPU. When *tim_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the TIM block and for a read cycle this means the data on *tim_cpu_data* is valid.
tim_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
tim_cpu_debug_valid	1	Out	Debug Data valid on *tim_cpu_data* bus. Active high
Miscellaneous			
tim_icu_wd_irq	1	Out	Watchdog timer interrupt signal to the ICU block
tim_icu_irq[2:0]	3	Out	Generic timer interrupt signals to the ICU block
tim_cpr_reset_n	1	Out	Watch dog timer system reset.

15.4.2 Timers sub-block partition

15.4.3 Watchdog timer

The watchdog timer counts down from pre-programmed value, and generates a system wide reset when equal to one. When the counter passes a pre-programmed threshold (*wdog_tim_thres*) value an interrupt is generated (*tim_icu_wd_irq*) requesting the CPU to update the counter. Setting the counter to zero disables the watchdog reset. In supervisor mode the watchdog counter can be written to or read from at any time, in user mode access is denied. Any accesses in user mode will generate a bus error.

The counter logic is given by

```
if (wdog_wen == 1) then
  wdog_tim_cnt = write_data       // load new data
elsif ( wdog_tim_cnt == 0) then
  wdog_tim_cnt = wdog_tim_cnt     // count disabled
elsif ( cnt_en == 1 ) then
  wdog_tim_cnt--
else
  wdog_tim_cnt = wdog_tim_cnt
```

The timer decode logic is

```
if (( wdog_tim_cnt == wdog_tim_thres) AND (wdog_tim_cnt != 0
)AND (cnt_en == 1)) then
  tim_icu_wd_irq = 1
else
  tim_icu_wd_irq = 0
// reset generator logic
if (wdog_tim_cnt == 1) AND (cnt_en == 1) then
  tim_cpr_reset_n = 0
else
  tim_cpr_reset_n = 1
```

15.4.4 Generic timers

The generic timers block consists of 3 identical counters. A timer is set to a pre-configured value (*GenCntStartValue*) and counts down once per selected timing pulse (*gen_unit_sel*). The timer can be enabled or disabled at any time (*gen_tim_en*), when disabled the counter is stopped but not cleared. The timer can be set to automatically restart (*gen_tim_auto*) after it generates an interrupt. In supervisor mode a timer can be written to or read from at any time, in user mode access is determined by the *GenCntUserModeEnable* register settings.

The counter logic is given by

```
if (gen_wen == 1) then
  gen_tim_cnt = write_data
elsif (( cnt_en == 1 )AND (gen_tim_en == 1 )) then
  if ( gen_tim_cnt == 1) OR ( gen_tim_cnt == 0)   then   //
counter may need re-starting
    if (gen_tim_auto == 1) then
      gen_tim_cnt = gen_tim_cnt_st_value
    else
      gen_tim_cnt = 0                                    // hold
count at zero
  else
    gen_tim_cnt--
```

```
else
  gen_tim_cnt = gen_tim_cnt
```

The decode logic is

```
if (gen_tim_cnt == 1)AND ( cnt_en == 1 )AND (gen_tim_en == 1
) then
  tim_icu_irq = 1
else
  tim_icu_irq = 0
```

15.4.5 Timing pulse generator

The timing pulse generator contains a general free running 64-bit timer and 3 timing pulse generators producing timing pulses of one cycle duration with a programmable period. The period is programmed by changed the *TimerStartValue* registers, but have a nominal starting period of 1μs, 100μs and 1ms. In supervisor mode the free running timer register can be written to or read from at any time, in user mode access is denied. The status of each of the timers can be read by accessing the *PulseTimerStatus* registers in supervisor mode. Any accesses in user mode will result in a bus error.

15.4.5.1 Free Run Timer

The increment logic block increments the timer count on each clock cycle. The counter wraps around to zero and continues incrementing if overflow occurs. When the timing register (*FreeRunCount*) is written to, the configuration registers block will set the *free_run_wen* high for a clock cycle and the value on *write_data* will become the new count value. If *free_run_wen[1]* is 1 the higher 32 bits of the counter will be written to, otherwise if *free_run_wen[0]* the lower 32 bits are written to. It is the responsibility of software to handle these writes in a sensible manner.

The increment logic is given by

```
if (free_run_wen[1] == 1) then
  free_run_cnt[63:32] = write_data
elsif (free_run_wen[0] == 1) then
  free_run_cnt[31:0]  = write_data
else
  free_run_cnt ++
```

15.4.5.2 Pulse Timers

The pulse timer logic generates timing pulses of 1 clock cycle length and programmable period. Nominally they generate pulse periods of 1μs, 100μs and 1ms. The logic for timer 0 is given by:

```
// Nominal 1us generator
if (pulse_0_cnt == 0 ) then
  pulse_0_cnt = timer_start_value[0]
  tim_pulse[0]= 1
else
  pulse_0_cnt --
  tim_pulse[0]= 0
```

The logic for timer 1 is given by:

```
// 100us generator
if ((pulse_1_cnt == 0) AND (tim_pulse[0] == 1)) then
  pulse_1_cnt = timer_start_value[1]
  tim_pulse[1]= 1
elsif (tim_pulse[0] == 1) then
  pulse_1_cnt --
  tim_pulse[1]= 0
else
  pulse_1_cnt = pulse_1_cnt
  tim_pulse[1]= 0
```

The logic for the timer 2 is given by:

```
// 10ms generator
if ((pulse_2_cnt == 0 ) AND (tim_pulse[1] == 1)) then
  pulse_2_cnt = timer_start_value[2]
  tim_pulse[2]= 1
elsif (tim_pulse[1] == 1) then
  pulse_2_cnt --
  tim_pulse[2]= 0
else
  pulse_2_cnt = pulse_2_cnt
  tim_pulse[2]= 0
```

15.4.6 Configuration registers

The configuration registers in the TIM are programmed via the CPU interface. Refer to section 11.4.3 on page 96 for a description of the protocol and timing diagrams for reading and writing registers in the TIM. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the TIM. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *tim_pcu_data*. Table 92 lists the configuration registers in the TIM block .

Table 92. Timers Register Map

Address TIM_base +	Register	#bits	Reset	Description
0x00	WatchDogUnitSel	2	0x0	Specifies the units used for the watchdog timer: 0 - Nominal 1 μs pulse 1 - Nominal 100 μs pulse

				2 - Nominal 10 ms pulse 3 - *pclk*
0x04	WatchDogTimer	32	0xFFFF_FFFF	Specifies the number of units to count before watchdog timer triggers.
0x08	WatchDogIntThres	32	0x0000_0000	Specifies the threshold value below which the watchdog timer issues an interrupt
0x0C-0x10	FreeRunCount[1:0]	2x32	0x0000_0000	Direct access to the free running counter register. Bus 0 - Access to bits 31-0 Bus 1 - Access to bits 63-32
0x14 to 0x1C	GenCntStartValue[2:0]	3x32	0x0000_0000	Generic timer counter start value, number of units to count before event
0x20 to 0x28	GenCntValue[2:0]	3x32	0x0000_0000	Direct access to generic timer counter registers
0x2C to 0x34	GenCntUnitSel[2:0]	3x2	0x0	Generic counter unit select. Selects the timing units used with corresponding counter: 0 - Nominal1 μs pulse 1 - Nominal100 μs pulse 2 - Nominal 10 ms pulse 3 - *pclk*
0x38 to 0x40	GenCntAuto[2:0]	3x1	0x0	Generic counter auto re-start select. When high timer automatically restarts, otherwise timer stops.
0x44 to 0x4C	GenCntEnable[2:0]	3x1	0x0	Generic counter enable. 0 - Counter disabled 1 - Counter enabled
0x50	GenCntUserMode Enable	3	0x0	User Mode Access enable to generic timer configuration register. When 1 user access is enabled. Bit 0 - Generic timer 0 Bit 1 - Generic timer 1 Bit 2 - Generic timer 2
0x54 to 0x5C	TimerStartValue[2:0]	3x8	0x7F, 0x63, 0x63	Timing pulse generator start value. Indicates the start value for each timing pulse timers. For timer 0 the start value specifies the timer period in *pclk* cycles - 1.

				For timer 1 the start value specifies the timer period in timer 0 intervals -1 . For timer 2 the start value specifies the timer period in timer 1 intervals -1. Nominally the timers generate pulses at 1us,100us and 10ms intervals respecitively.
0x60	DebugSelect[6:2]	5	0x00	Debug address select. Indicates the address of the register to report on the *tim_cpu_data* bus when it is not otherwise being used.
Read Only Registers				
0x64	PulseTimerStatus	24	0x00	Current pulse timer values, and pulses 7:0 - Timer 0 count 15:8 - Timer 1 count 23:16 - Timer 2 count 24 - Timer 0 pulse 25 - Timer 1 pulse 26 - Timer 2 pulse

15.4.6.1 Supervisor and user mode access

The configuration registers block examines the CPU access type (*cpu_acode* signal) and determines if the access is allowed to that particular register, based on configured user access registers. If an access is not allowed the block will issue a bus error by asserting the *tim_cpu_berr* signal.

The timers block is fully accessible in supervisor data mode, all registers can written to and read from. In user mode access is denied to all registers in the block except for the generic timer configuration registers that are granted user data access. User data access for a generic timer is granted by setting corresponding bit in the *GenCntUserModeEnable* register. This can only be changed in supervisor data mode. If a particular timer is granted user data access then all registers for configuring that timer will be accessible. For example if timer 0 is granted user data access the *GenCntStartValue[0]*, *GenCntUnitSel[0]*, *GenCntAuto[0]*, *GenCntEnable[0]* and *GenCntValue[0]* registers can all be written to and read from without any restriction.

Attempts to access a user data mode disabled timer configuration register will result in a bus error.

Table 93 details the access modes allowed for registers in the TIM block. In supervisor data mode all registers are accessable. All forbidden accesses will result in a bus error (*tim_cpu_berr* asserted).

Table 93. TIM supervisor and user access modes

Register Address	Registers	Access Permission
0x00	WatchDogUnitSel	Supervisor data mode only
0x04	WatchDogTimer	Supervisor data mode only
0x08	WatchDogIntThres	Supervisor data mode only
0x0C-0x10	FreeRunCount	Supervisor data mode only
0x14	GenCntStartValue[0]	GenCntUserModeEnable[0]
0x18	GenCntStartValue[1]	GenCntUserModeEnable[1]
0x1C	GenCntStartValue[2]	GenCntUserModeEnable[2]
0x20	GenCntValue[0]	GenCntUserModeEnable[0]
0x24	GenCntValue[1]	GenCntUserModeEnable[1]
0x28	GenCntValue[2]	GenCntUserModeEnable[2]
0x2C	GenCntUnitSel[0]	GenCntUserModeEnable[0]
0x30	GenCntUnitSel[1]	GenCntUserModeEnable[1]
0x34	GenCntUnitSel[2]	GenCntUserModeEnable[2]
0x38	GenCntAuto[0]	GenCntUserModeEnable[0]
0x3C	GenCntAuto[1]	GenCntUserModeEnable[1]
0x40	GenCntAuto[2]	GenCntUserModeEnable[2]
0x44	GenCntEnable[0]	GenCntUserModeEnable[0]
0x48	GenCntEnable[1]	GenCntUserModeEnable[1]
0x4C	GenCntEnable[2]	GenCntUserModeEnable[2]
0x50	GenCntUserModeEnable	Supervisor data mode only
0x54-0x5C	TimerStartValue[2:0]	Supervisor data mode only
0x60	DebugSelect	Supervisor data mode only
0x64	PulseTimerStatus	Supervisor data mode only

16 Clocking, Power and Reset (CPR)

The CPR block provides all of the clock, power enable and reset signals to the SoPEC device.

16.1 POWERDOWN MODES

The CPR block is capable of powering down certain sections of the SoPEC device. When a section is powered down (i.e. put in sleep mode) no state is retained(except the PSS storage), the CPU must re-initialize the section before it can be used again.

For the purpose of powerdown the SoPEC device is divided into sections:

Table 94. Powerdown sectioning

Section	Block
Print Engine Pipeline SubSystem (Section 0)	PCU

	CDU
	CFU
	LBD
	SFU
	TE
	TFU
	HCU
	DNC
	DWU
	LLU
	PHI
CPU-DRAM (Section 1)	DRAM
	CPU/MMU
	DIU
	TIM
	ROM
	LSS
	PSS
	ICU
ISI Subsystem (Section 2)	ISI (SCB)
	DMA Ctrl (SCB)
	GPIO
USB Subsystem (Section 3)	USB (SCB)

Note that the CPR block is not located in any section. All configuration registers in the CPR block are clocked by an ungateable clock and have special reset conditions.

16.1.1 Sleep mode

Each section can be put into sleep mode by setting the corresponding bit in the *SleepModeEnable* register. To re-enable the section the sleep mode bit needs to be cleared and then the section should be reset by writing to the relevant bit in the *ResetSection* register. Each block within the section should then be re-configured by the CPU.

If the CPU system (section 1) is put into sleep mode, the SoPEC device will remain in sleep mode until a system level reset is initiated from the reset pin, or a wakeup reset by the SCB block as a result of activity on either the USB or ISI bus. The watchdog timer cannot reset the device as it is in section 1 also, and will be in sleep mode.

If the CPU and ISI subsystem are in sleep mode only a reset from the USB or a hardware reset will re-activate the SoPEC device.

If all sections are put into sleep mode, then only a system level reset initiated by the reset pin will re-activate the SoPEC device.

Like all software resets in SoPEC the *ResetSection* register is active-low i.e. a 0 should be written to each bit position requiring a reset. The *ResetSection* register is self-reseting.

16.1.2 Sleep Mode powerdown procedure

When powering down a section, the section may retain it's current state (although not gauranteed to). It is possible when powering back up a section that inconsistancies between interface state machines could cause incorrect operation. In order to prevent such condition from happening, all blocks in a section must be disabled before powering down. This will ensure that blocks are restored in a benign state when powered back up.

In the case of PEP section units setting the *Go* bit to zero will disable the block. The DRAM subsystem can be effectively disabled by setting the *RotationSync* bit to zero, and the SCB system disabled by setting the *DMAAccessEn* bits to zero turning off the DMA access to DRAM. Other CPU subsystem blocks without any DRAM access do not need to be disabled.

16.2 RESET SOURCE

The SoPEC device can be reset by a number of sources. When a reset from an internal source is initiated the reset source register (*ResetSrc*) stores the reset source value. This register can then be used by the CPU to determine the type of boot sequence required.

16.3 CLOCK RELATIONSHIP

The crystal oscillator excites a 32MHz crystal through the xtalin and xtalout pins. The 32MHz output is used by the PLL to derive the master VCO frequency of 960MHz. The master clock is then divided to produce 320MHz clock (*clk320*), 160MHz clock (*clk160*) and 48MHz (*clk48*) clock sources.

The phase relationship of each clock from the PLL will be defined. The relationship of internal clocks *clk320, clk48* and *clk160* to *xtalin* will be undefined.

At the output of the clock block, the skew between each *pclk* domain (*pclk_section[2:0]* and *jclk*) should be within skew tolerances of their respective domains (defined as less than the hold time of a D-type flip flop).

The skew between *doclk* and *pclk* should also be less than the skew tolerances of their respective domains.

The *usbclk* is derived from the PLL output and has no relationship with the other clocks in the system and is considered asynchronous.

16.4 PLL CONTROL

The PLL in SoPEC can be adjusted by programming the *PLLRangeA*, *PLLRangeB*, *PLLTunebits* and *PLLMult* registers. If these registers are changed by the CPU the values are not updated until the *PLLUpdate* register is written to. Writing to the *PLLUpdate* register triggers the PLL control state machine to update the PLL configuration in a safe way. When an update is active (as indicated by *PLLUpdate* register) the CPU must not change any of the configuration registers, doing so could cause the PLL to lose lock indefintely, requiring a hardware reset to recover. Configuring the PLL registers in an inconsistent way can also cause the PLL to lose lock, care must taken to keep the PLL configuration within specified parameters.

The VCO frequency of the PLL is calculated by the number of divider in the feedback path. PLL output A is used as the feedback source.

VCOfreq = *REFCLK* x *PLLMult* x *PLLRangeA* x External divider

VCOfreq = 32 x 3 x 10 x 1 = 960 Mhz.

In the default PLL setup, *PLLMult* is set to 3, *PLLRangeA* is set to 3 which corresponds to a divide by 10, *PLLRangeB* is set to 5 which corresponds to a divide by 3.

PLLouta = VCOfreq / *PLLRangeA* = 960Mhz / 10 = 96 Mhz

PLLoutb = VCOfreq / *PLLRangeB* = 960Mhz / 3 = 320 Mhz

See [16] for complete PLL setup parameters.

16.5 IMPLEMENTATION

16.5.1 Definitions of I/O

Table 95. CPR I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
Xtalin	1	In	Crystal input, direct from IO pin.
Xtalout	1	Inout	Crystal output, direct to IO pin.
pclk_section[3:0]	4	Out	System clocks for each section
Doclk	1	Out	Data out clock (2x *pclk*) for the PHI block
Jclk	1	Out	Gated version of system clock used to clock the JPEG decoder core in the CDU
Usbclk	1	Out	USB clock, nominally at 48 Mhz
jclk_enable	1	In	Gating signal for *jclk*. When 1 *jclk* is enabled
reset_n	1	In	Reset signal from the *reset_n* pin
usb_cpr_reset_n	1	In	Reset signal from the USB block
isi_cpr_reset_n	1	In	Reset signal from the ISI block
tim_cpr_reset_n	1	In	Reset signal from watch dog timer.
gpio_cpr_wakeup	1	In	SoPEC wake up from the GPIO, active high.
prst_n_section[3:0]	4	Out	System resets for each section, synchronous active low
dorst_n	1	Out	Reset for PHI block, synchronous to *doclk*
jrst_n	1	Out	Reset for JPEG decoder core in CDU block, synchronous to *jclk*
usbrst_n	1	Out	Reset for the USB block, synchronous to *usbclk*
CPU interface			
cpu_adr[5:2]	3	In	CPU address bus. Only 4 bits are required to decode the address space for the CPR block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
cpr_cpu_data[31:0]	32	Out	Read data bus to the CPU

cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_cpr_sel	1	In	Block select from the CPU. When *cpu_cpr_sel* is high both *cpu_adr* and *cpu_dataout* are valid
cpr_cpu_rdy	1	Out	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *cpr_cpu_data* is valid.
cpr_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpr_cpu_debug_valid	1	Out	Debug Data valid on *cpr_cpu_data* bus. Active high

16.5.2 Configuration registers

The configuration registers in the CPR are programmed via the CPU interface. Refer to section 11.4 on page 96 for a description of the protocol and timing diagrams for reading and writing registers in the CPR. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the CPR. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *cpr_pcu_data*. Table 96 lists the configuration registers in the CPR block.

The CPR block will only allow supervisor data mode accesses (i.e. *cpu_acode[1:0]* = *SUPERVISOR_DATA*). All other accesses will result in *cpr_cpu_berr* being asserted .

Table 96. CPR Register Map

Address CPR_base +	Register	#bits	Reset	Description
0x00	SleepModeEnable	4	0x0[a]	Sleep Mode enable, when high a section of logic is put into powerdown. Bit 0 - Controls section 0 Bit 1 - Controls section 1 Bit 2 - Controls section 2 Bit 3 - Controls section 3 Note that the SleepModeEnable register

				has special reset conditions. See Section 16.5.6 for details
0x04	ResetSrc	5	0x1[a]	Reset Source register, indicating the source of the last reset (or wake-up) Bit 0 - External Reset Bit 1 - USB wakeup reset Bit 2 - ISI wakeup reset Bit 3 - Watchdog timer reset Bit 4 - GPIO wake-up (Read Only Register)
0x08	ResetSection	4	0xF	Active-low synchronous reset for each section, self-resetting. Bit 0 - Controls section 0 Bit 1 - Controls section 1 Bit 2 - Controls section 2 Bit 3 - Controls section 3
0x0C	DebugSelect[5:2]	4	0x0	Debug address select. Indicates the address of the register to report on the *cpr_cpu_data* bus when it is not otherwise being used.
PLL Control				
0x10	PLLTuneBits	10	0x3BC	PLL tuning bits
0x14	PLLRangeA	4	0x3	PLLOUT A frequency selector (defaults to 60Mhz to 125Mhz)
0x18	PLLRangeB	3	0x5	PLLOUT B frequency selector (defaults to 200Mhz to 400Mhz)
0x1C	PLLMultiplier	5	0x03	PLL multiplier selector, defaults to *refclk* x 3
0x20	PLLUpdate	1	0x0	PLL update control. A write (of any value) to this register will cause the PLL to lose lock for ~100us. Reading the register indicates the status of the update. 0 - PLL update complete 1 - PLL update active No writes to *PLLTuneBits*,*PLLRangeA*,*PLL-RangeB*,*PLLMultiplier* or *PLLUpdate* are allowed while the PLL update is

				active.

a. Reset value depends on reset source. External reset shown.

16.5.3 CPR Sub-block partition

16.5.4 reset_n deglitch

The external reset_n signal is deglitched for about 1µs. reset_n must maintain a state for 1us second before the state is passed into the rest of the device. All deglitch logic is clocked on *bufrefclk*.

16.5.5 Sync reset

The reset synchronizer retimes an asynchronous reset signal to the clock domain that it resets. The circuit prevents the inactive edge of reset occurring when the clock is rising

16.5.6 Reset generator logic

The reset generator logic is used to determine which clock domains should be reset, based on configured reset values (*reset_section_n*), the external reset (*reset_n*), watchdog timer reset (*tim_cpr_reset_n*), the USB reset (*usb_cpr_reset_n*), the GPIO wakeup control (*gpio_cpr_wakeup*) and the ISI reset (*isi_cpr_reset_n*). The reset direct from the IO pin (*reset_n*) is synchronized and de-glitched before feeding the reset logic.

All resets are lengthened to at least 16 *pclk* cycles, regardless of the duration of the input reset. The clock for a particular section must be running for the reset to have an effect. The clocks to each section can be enabled/disabled using the *SleepModeEnable* register.

Resets from the ISI or USB block reset everything except its own section (section 2 or 3).

Table 97. Reset domains

Reset signal	Domain
reset_dom[0]	Section 0 pclk domain (PEP)
reset_dom[1]	Section 1 pclk domain (CPU)
reset_dom[2]	Section 2 pclk domain (ISI)
reset_dom[3]	Section 3 usbclk/pclk domain (USB)
reset_dom[4]	doclk domain
reset_dom[5]	jclk domain

The logic is given by

```
if (reset_dg_n == 0) then
  reset_dom[5:0]       = 0x00      // reset everything
  reset_src[4:0]       = 0x01
  cfg_reset_n          = 0
  sleep_mode_en[3:0]   = 0x0       // re-awaken all sections
elsif (tim_cpr_reset_n == 0) then
  reset_dom[5:0]        = 0x00      // reset everything except
CPR config
```

```
    reset_src[4:0]       = 0x08
    cfg_reset_n          = 1          // CPR config stays the same
    sleep_mode_en[1]     = 0          // re-awaken section 1 only
(awake already)
elsif (usb_cpr_reset_n == 0) then
    reset_dom[5:0]       = 0x08       // all except USB domain +
CPR config
    reset_src[4:0]       = 0x02
    cfg_reset_n          = 1          // CPR config stays the same
    sleep_mode_en[1]     = 0          // re-awaken section 1 only,
section 3 is awake
elsif (isi_cpr_reset_n == 0) then
    reset_dom[5:0]       = 0x04       // all except ISI domain +
CPR config
    reset_src[4:0]       = 0x04
    cfg_reset_n          = 1          // CPR config stays the same
    sleep_mode_en[1]     = 0          // re-awaken section 1 only,
section 2 is awake
elsif (gpio_cpr_wakeup = 1) then
    reset_dom[5:0]       = 0x3C       // PEP and CPU sections only
    reset_src[4:0]       = 0x10
    cfg_reset_n          = 1          // CPR config stays the same
    sleep_mode_en[1]     = 0          // re-awaken section 1 only,
section 2 is awake
else
    // propagate resets from reset section register
    reset_dom[5:0]       = 0x3F                  // default to on
    cfg_reset_n            = 1                          // CPR cfg
registers are not in any section
    sleep_mode_en[3:0]   = sleep_mode_en[3:0] // stay the same
by default
    if (reset_section_n[0] == 0) then
      reset_dom[5] = 0                  // jclk domain
      reset_dom[4] = 0                  // doclk domain
      reset_dom[0] = 0                  // pclk section 0 domain
    if (reset_section_n[1] == 0) then
      reset_dom[1] = 0                  // pclk section 1 domain
    if (reset_section_n[2] == 0) then
      reset_dom[2] = 0                    // pclk section 2 domain
(ISI)
    if (reset_section_n[3] == 0) then
```

```
reset_dom[3] = 0                    // USB domain
```

16.5.7 Sleep logic

The sleep logic is used to generate gating signals for each of SoPECs clock domains. The gate enable (*gate_dom*) is generated based on the configured *sleep_mode_en* and the internally generated *jclk_enable* signal.

The logic is given by

```
// clock gating for sleep modes
gate_dom[5:0] = 0x0                    // default to all clocks
on
  if (sleep_mode_en[0] == 1) then    // section 0 sleep
    gate_dom[0] = 1                 // pclk section 0
    gate_dom[4] = 1                 // doclk domain
    gate_dom[5] = 1                 // jclk domain
  if (sleep_mode_en[1] == 1) then    // section 1 sleep
    gate_dom[1] = 1                 // pclk section 1
  if (sleep_mode_en[2] == 1) then    // section 2 sleep
    gate_dom[2] = 1                 // pclk section 2
  if (sleep_mode_en[3] == 1) then    // section 3 sleep
    gate_dom[3] = 1                 // usb section 3
// the jclk can be turned off by CDU signal
if (jclk_enable == 0) then
  gate_dom[5] = 1
```

The clock gating and sleep logic is clocked with the *master_pclk* clock which is not gated by this logic, but is synchronous to other *pclk_section* and *jclk* domains.

Once a section is in sleep mode it cannot generate a reset to restart the device. For example if section 1 is in sleep mode then the watchdog timer is effectively disabled and cannot trigger a reset.

16.5.8 Clock gate logic

The clock gate logic is used to safely gate clocks without generating any glitches on the gated clock. When the enable is high the clock is active otherwise the clock is gated.

16.5.9 Clock generator Logic

The clock generator block contains the PLL, crystal oscillator, clock dividers and associated control logic. The PLL VCO frequency is at 960MHz locked to a 32 MHz *refclk* generated by the crystal oscillator. In test mode the *xtalin* signal can be driven directly by the test clock generator, the test clock will be reflected on the *refclk* signal to the PLL.

16.5.9.1 Clock divider A

The clock divider A block generates the 48MHz clock from the input 96MHz clock (*pllouta*) generated by the PLL. The divider is enabled only when the PLL has acquired lock.

16.5.9.2 Clock divider B

The clock divider B block generates the 160MHz clocks from the input 320MHz clock (*plloutb*) generated by the PLL. The divider is enabled only when the PLL has acquired lock.

16.5.9.3 PLL control state machine

The PLL will go out of lock whenever *pll_reset* goes high (the PLL reset is the only active high reset in the device) or if the configuration bits *pll_rangea*, *pll_rangeb*, *pll_mult*, *pll_tune* are changed. The PLL control state machine ensures that the rest of the device is protected from glitching clocks while the PLL is being reset or it's configuration is being changed.

In the case of a hardware reset (the reset is deglitched), the state machine first disables the output clocks (via the *clk_gate* signal), it then holds the PLL in reset while its configuration bits are reset to default values. The state machine then releases the PLL reset and waits approx. 100us to allow the PLL to regain lock. Once the lock time has elapsed the state machine re-enables the output clocks and resets the remainder of the device via the *reset_dg_n* signal.

When the CPU changes any of the configuration registers it must write to the PLLupdate register to allow the state machine to update the PLL to the new configuration setup. If a PLLUpdate is detected the state machine first gates the output clocks. It then holds the PLL in reset while the PLL configuration registers are updated. Once updated the PLL reset is released and the state machine waits approx 100us for the PLL to regain lock before re-enabling the output clocks. Any write to the PLLUpdate register will cause the state machine to perform the update operation regardless of whether the configuration values changed or not.

All logic in the clock generator is clocked on *bufrefclk* which is always an active clock regardless of the state of the PLL.

17 ROM Block

17.1 OVERVIEW

The ROM block interfaces to the CPU bus and contains the SoPEC boot code. The ROM block consists of the CPU bus interface, the ROM macro and the ChipID macro. The current ROM size is 16 KBytes implemented as a 4096 x32 macro. Access to the ROM is not cached because the CPU enjoys fast (no more than one cycle slower than a cache access), unarbitrated access to the ROM.

Each SoPEC device is required to have a unique ChipID which is set by blowing fuses at manufacture. IBM's 300mm ECID macro and a custom 112-bit ECID macro are used to implement the ChipID offering 224-bits of laser fuses. The exact number of fuse bits to be used for the ChipID will be determined later but all bits are made available to the CPU. The ECID macros allows all 224 bits to be read out in parallel and the ROM block will make all 224 bits available in the *FuseChipID[N]* registers which are readable by the CPU in supervisor mode only.

17.2 BOOT OPERATION

The are two boot scenarios for the SoPEC device namely after power-on and after being awoken from sleep mode. When the device is in sleep mode it is hoped that power will actually be removed from the DRAM, CPU and most other peripherals and so the program code will need to be freshly downloaded each time the device wakes up from sleep mode. In order to reduce the wakeup boot time (and hence the perceived print latency) certain data items are stored in the PSS block (see section 18). These data items include the SHA-1 hash digest expected for the program(s) to be

downloaded, the master/slave SoPEC id and some configuration parameters. All of these data items are stored in the PSS by the CPU prior to entering sleep mode. The SHA-1 value stored in the PSS is calculated by the CPU by decrypting the signature of the downloaded program using the appropriate public key stored in ROM. This compute intensive decryption only needs to take place once as part of the power-on boot sequence - subsequent wakeup boot sequences will simply use the resulting SHA-1 digest stored in the PSS. Note that the digest only needs to be stored in the PSS before entering sleep mode and the PSS can be used for temporary storage of any data at all other times.

The CPU is expected to be in supervisor mode for the entire boot sequence described by the pseudocode below. Note that the boot sequence has not been finalised but is expected to be close to the following:

```
if (ResetSrc == 1)  then   // Reset was a power-on reset
  configure_sopec   // need to configure peris (USB, ISI,
DMA, ICU etc.)
  // Otherwise reset was a wakeup reset so peris etc. were
already configured
  PAUSE: wait until IrqSemaphore != 0   // i.e. wait until an
interrupt has been serviced
  if (IrqSemaphore == DMAChan0Msg) then
    parse_msg(DMAChan0MsgPtr)  // this routine will parse the
message and take any
                               // necessary action e.g. programming
the DMAChannel1 registers
  elsif (IrqSemaphore == DMAChan1Msg) then   // program has
been downloaded
    CalculatedHash = gen_sha1(ProgramLocn, ProgramSize)
    if (ResetSrc == 1)  then
      ExpectedHash = sig_decrypt(ProgramSig,public_key)
    else
      ExpectedHash = PSSHash
    if (ExpectedHash == CalculatedHash) then
      jmp(PrgramLocn)  // transfer control to the downloaded
program
    else
      send_host_msg("Program Authentication Failed")
      goto PAUSE:
    elsif (IrqSemaphore == timeout) then   // nothing has
happened
      if (ResetSrc == 1)  then
```

```
            sleep_mode()  // put SoPEC into sleep mode to be woken
up by USB/ISI activity
          else  // we were woken up but nothing happened
            reset_sopec(PowerOnReset)
        else
          goto PAUSE
```

The boot code places no restrictions on the activity of any programs downloaded and authenticated by it other than those imposed by the configuration of the MMU i.e. the principal function of the boot code is to authenticate that any programs downloaded by it are from a trusted source. It is the responsibility of the downloaded program to ensure that any code it downloads is also authenticated and that the system remains secure. The downloaded program code is also responsible for setting the SoPEC ISIId (see section 12.5 for a description of the ISIId) in a multi -SoPEC system. See the "SoPEC Security Overview" document [9] for more details of the SoPEC security features.

17.3 IMPLEMENTATION

17.3.1 Definitions of I/O

Table 98. ROM Block I/O

Port name	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
Pclk	1	In	Global clock
CPU Interface			
cpu_adr[14:2]	13	In	CPU address bus. Only 13 bits are required to decode the address space for this block.
rom_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_rom_sel	1	In	Block select from the CPU. When *cpu_rom_sel* is high *cpu_adr* is valid
rom_cpu_rdy	1	Out	Ready signal to the CPU. When *rom_cpu_rdy* is high it indicates the last cycle of the access. For a read cycle this means the data on

			rom_cpu_data is valid.
rom_cpu_berr	1	Out	ROM bus error signal to the CPU indicating an invalid access.

17.3.2 Configuration registers

The ROM block will only allow read accesses to the *FuseChipID* registers and the ROM with supervisor data space permissions (i.e. *cpu_acode[1:0]* = 11). Write accesses with supervisor data space permissions will have no effect. All other accesses with will result in *rom_cpu_berr* being asserted. The CPU subsystem bus slave interface is described in more detail in section 9.4.3.

Table 99. ROM Block Register Map

Address ROM_base +	Register	#bits	Reset	Description
0x4000	FuseChipID0	32	n/a	Value of corresponding fuse bits 31 to 0 of the IBM 112-bit ECID macro. (Read only)
0x4004	FuseChipID1	32	n/a	Value of corresponding fuse bits 63 to 32 of the IBM 112-bit ECID macro. (Read only)
0x4008	FuseChipID2	32	n/a	Value of corresponding fuse bits 95 to 64 of the IBM 112-bit ECID macro. (Read only)
0x400C	FuseChipID3	16	n/a	Value of corresponding fuse bits 111 to 96 of the IBM 112-bit ECID macro. (Read only)
0x4010	FuseChipID4	32	n/a	Value of corresponding fuse bits 31 to 0 of the Custom 112-bit ECID macro. (Read only)
0x4014	FuseChipID5	32	n/a	Value of corresponding fuse bits 63 to 32 of the Custom 112-bit ECID macro. (Read only)
0x4018	FuseChipID6	32	n/a	Value of corresponding fuse bits 95 to 64 of the Custom 112-bit ECID macro. (Read only)
0x401C	FuseChipID7	16	n/a	Value of corresponding fuse bits 111 to 96 of the Custom 112-bit ECID macro. (Read only)

17.3.3 Sub-Block Partition

IBM offer two variants of their ROM macros; A high performance version (ROMHD) and a low power version (ROMLD). It is likely that the low power version will be used unless some implementation issue requires the high performance version. Both versions offer the same bit

density. The sub-block partition diagram below does not include the clocking and test signals for the ROM or ECID macros. The CPU subsystem bus interface is described in more detail in section 11.4.3.

17.3.4 Table 100. ROM Block internal signals

Port name	Width	Description
Clocks and Resets		
prst_n	1	Global reset. Synchronous to pclk, active low.
Pclk	1	Global clock
Internal Signals		
rom_adr[11:0]	12	ROM address bus
rom_sel	1	Select signal to the ROM macro instructing it to access the location at *rom_adr*
rom_oe	1	Output enable signal to the ROM block
rom_data[31:0]	32	Data bus from the ROM macro to the CPU bus interface
rom_dvalid	1	Signal from the ROM macro indicating that the data on *rom_data* is valid for the address on *rom_adr*
fuse_data[31:0]	32	Data from the *FuseChipID[N]* register addressed by *fuse_reg_adr*
fuse_reg_adr[2:0]	3	Indicates which of the *FuseChipID* registers is being addressed

Sub-block signal definition

18 Power Safe Storage (PSS) Block

18.1 OVERVIEW

The PSS block provides 128 bytes of storage space that will maintain its state when the rest of the SoPEC device is in sleep mode. The PSS is expected to be used primarily for the storage of decrypted signatures associated with downloaded programmed code but it can also be used to store any information that needs to survive sleep mode (e.g. configuration details). Note that the signature digest only needs to be stored in the PSS before entering sleep mode and the PSS can be used for temporary storage of any data at all other times.

Prior to entering sleep mode the CPU should store all of the information it will need on exiting sleep mode in the PSS. On emerging from sleep mode the boot code in ROM will read the *ResetSrc* register in the CPR block to determine which reset source caused the wakeup. The reset source information indicates whether or not the PSS contains valid stored data, and the PSS data determines the type of boot sequence to execute. If for any reason a full power-on boot sequence should be performed (e.g. the printer driver has been updated) then this is simply achieved by initiating a full software reset.

Note that a reset or a powerdown (powerdown is implemented by clock gating) of the PSS block will not clear the contents of the 128 bytes of storage. If clearing of the PSS storage is required, then the CPU must write to each location individually.

18.2 IMPLEMENTATION

The storage area of the PSS block will be implemented as a 128-byte register array. The array is located from PSS_base through to PSS_base+0x7F in the address map. The PSS block will only allow read or write accesses with supervisor data space permissions (i.e. *cpu_acode[1:0]* = 11). All other accesses will result in *pss_cpu_berr* being asserted. The CPU subsystem bus slave interface is described in more detail in section 11.4.3.

18.2.1 Definitions of I/O

Table 101. PSS Block I/O

Port name	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
Pclk	1	In	Global clock
CPU Interface			
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for this block.
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
pss_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpus_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_pss_sel	1	In	Block select from the CPU. When *cpu_pss_sel* is high both *cpu_adr* and *cpu_dataout* are valid
pss_cpu_rdy	1	Out	Ready signal to the CPU. When *pss_cpu_rdy* is high it indicates the last cycle of the access. For a read cycle this means the data on *pss_cpu_data* is valid.
pss_cpu_berr	1	Out	PSS bus error signal to the CPU indicating an invalid access.

19 Low Speed Serial Interface (LSS)

19.1 OVERVIEW

The Low Speed Serial Interface (LSS) provides a mechanism for the internal SoPEC CPU to communicate with external QA chips via two independent LSS buses. The LSS communicates through the GPIO block to the QA chips. This allows the QA chip pins to be reused in multi-

SoPEC environments. The LSS Master system-level interface is illustrated in Figure 75. Note that multiple QA chips are allowed on each LSS bus.

19.2 QA COMMUNICATION

The SoPEC data interface to the QA Chips is a low speed, 2 pin, synchronous serial bus. Data is transferred to the QA chips via the *lss_data* pin synchronously with the *lss_clk* pin. When the *lss_clk* is high the data on *lss_data* is deemed to be valid. Only the LSS master in SoPEC can drive the *lss_clk* pin, this pin is an input only to the QA chips. The LSS block must be able to interface with an open-collector pull-up bus. This means that when the LSS block should transmit a logical zero it will drive 0 on the bus, but when it should transmit a logical 1 it will leave high-impedance on the bus (i.e. it doesn't drive the bus). If all the agents on the LSS bus adhere to this protocol then there will be no issues with bus contention.

The LSS block controls all communication to and from the QA chips. The LSS block is the bus master in all cases. The LSS block interprets a command register set by the SoPEC CPU, initiates transactions to the QA chip in question and optionally accepts return data. Any return information is presented through the configuration registers to the SoPEC CPU. The LSS block indicates to the CPU the completion of a command or the occurrence of an error via an interrupt. The LSS protocol can be used to communicate with other LSS slave devices (other than QA chips). However should a LSS slave device hold the clock low (for whatever reason), it will be in violation of the LSS protocol and is not supported. The LSS clock is only ever driven by the LSS master.

19.2.1 Start and stop conditions

All transmissions on the LSS bus are initiated by the LSS master issuing a START condition and terminated by the LSS master issuing a STOP condition. START and STOP conditions are always generated by the LSS master. As illustrated in Figure 76, a START condition corresponds to a high to low transition on *lss_data* while *lss_clk* is high. A STOP condition corresponds to a low to high transition on *lss_data* while *lss_clk* is high.

19.2.2 Data transfer

Data is transferred on the LSS bus via a byte orientated protocol. Bytes are transmitted serially. Each byte is sent most significant bit (MSB) first through to least significant bit (LSB) last. One clock pulse is generated for each data bit transferred. Each byte must be followed by an acknowledge bit.

The data on the *lss_data* must be stable during the HIGH period of the *lss_clk* clock. Data may only change when *lss_clk* is low. A transmitter outputs data after the falling edge of *lss_clk* and a receiver inputs the data at the rising edge of *lss_clk*. This data is only considered as a valid data bit at the next *lss_clk* falling edge provided a START or STOP is not detected in the period before the next *lss_clk* falling edge. All clock pulses are generated by the LSS block. The transmitter releases the *lss_data* line (high) during the acknowledge clock pulse (ninth clock pulse). The receiver must pull down the *lss_data* line during the acknowledge clock pulse so that it remains stable low during the HIGH period of this clock pulse.

Data transfers follow the format shown in Figure 77. The first byte sent by the LSS master after a START condition is a primary id byte, where bits 7-2 form a 6-bit primary id (0 is a global id and will address all QA Chips on a particular LSS bus), bit 1 is an even parity bit for the primary id, and bit 0 forms the read/ write sense. Bit 0 is high if the following command is a read to the primary id given or low for a write command to that id. An acknowledge is generated by the QA chip(s) corresponding to the given id (if such a chip exists) by driving the *lss_data* line low synchronous with the LSS master generated ninth *lss_clk*.

19.2.3 Write procedure

The protocol for a write access to a QA Chip over the LSS bus is illustrated in Figure 79 below. The LSS master in SoPEC initiates the transaction by generating a START condition on the LSS bus. It then transmits the primary id byte with a 0 in bit 0 to indicate that the following command is a write to the primary id. An acknowledge is generated by the QA chip corresponding to the given primary id. The LSS master will clock out M data bytes with the slave QA Chip acknowledging each successful byte written. Once the slave QA chip has acknowledged the M^{th} data byte the LSS master issues a STOP condition to complete the transfer. The QA chip gathers the M data bytes together and interprets them as a command. See QA Chip Interface Specification for more details on the format of the commands used to communicate with the QA chip[8]. Note that the QA chip is free to not acknowledge any byte transmitted. The LSS master should respond by issuing an interrupt to the CPU to indicate this error. The CPU should then generate a STOP condition on the LSS bus to gracefully complete the transaction on the LSS bus.

19.2.4 Read procedure

The LSS master in SoPEC initiates the transaction by generating a START condition on the LSS bus. It then transmits the primary id byte with a 1 in bit 0 to indicate that the following command is a read to the primary id. An acknowledge is generated by the QA chip corresponding to the given primary id. The LSS master releases the *lss_data* bus and proceeds to clock the expected number of bytes from the QA chip with the LSS master acknowledging each successful byte read. The last expected byte is not acknowledged by the LSS master. It then completes the transaction by generating a STOP condition on the LSS bus. See QA Chip Interface Specification for more details on the format of the commands used to communicate with the QA chip[8].

19.3 IMPLEMENTATION

A block diagram of the LSS master is given in Figure 80. It consists of a block of configuration registers that are programmed by the CPU and two identical LSS master units that generate the signalling protocols on the two LSS buses as well as interrupts to the CPU. The CPU initiates and terminates transactions on the LSS buses by writing an appropriate command to the command register, writes bytes to be transmitted to a buffer and reads bytes received from a buffer, and checks the sources of interrupts by reading status registers.

19.3.1 Definitions of IO

Table 102. LSS IO pins definitions

Port name	Pins	I/O	Description

Clocks and Resets			
Pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU Interface			
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
cpu_acode[1:0]	2	In	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access
cpu_lss_sel	1	In	Block select from the CPU. When *cpu_lss_sel* is high both *cpu_adr* and *cpu_dataout* are valid
lss_cpu_rdy	1	Out	Ready signal to the CPU. When *lss_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the LSS block and for a read cycle this means the data on *lss_cpu_data* is valid.
lss_cpu_berr	1	Out	LSS bus error signal to the CPU.
lss_cpu_data[31:0]	32	Out	Read data bus to the CPU
lss_cpu_debug_valid	1	Out	Active high. Indicates the presence of valid debug data on *lss_cpu_data*.
GPIO for LSS buses			
lss_gpio_dout[1:0]	2	Out	LSS bus data output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
gpio_lss_din[1:0]	2	In	LSS bus data input Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_e[1:0]	2	Out	LSS bus data output enable, active high Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_clk[1:0]	2	Out	LSS bus clock output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
ICU interface			
lss_icu_irq[1:0]	2	Out	LSS interrupt requests Bit 0 - interrupt associated with LSS bus 0 Bit 1 - interrupt associated with LSS bus 1

19.3.2 Configuration registers

The configuration registers in the LSS block are programmed via the CPU interface. Refer to section 11.4 on page 96 for the description of the protocol and timing diagrams for reading and writing registers in the LSS block. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the LSS block. Table 103 lists the configuration registers in the LSS block. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *lss_cpu_data*.

The input *cpu_acode* signal indicates whether the current CPU access is supervisor, user, program or data. The configuration registers in the LSS block can only be read or written by a supervisor data access, i.e. when *cpu_acode* equals b11. If the current access is a supervisor data access then the LSS responds by asserting *lss_cpu_rdy* for a single clock cycle.

If the current access is anything other than a supervisor data access, then the LSS generates a bus error by asserting *lss_cpu_berr* for a single clock cycle instead of *lss_cpu_rdy* as shown in section 11.4 on page 96. A write access will be ignored, and a read access will return zero.

Table 103. LSS Control Registers

Address (LSS_base +)	Register	#bits	Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the LSS.
0x04	LssClockHighLow-Duration	16	0x00C8	*Lss_clk* has a 50:50 duty cycle, this register defines the period of *lss_clk* by means of specifying the duration (in *pclk* cycles) that *lss_clk* is low (or high). The reset value specifies transmission over the LSS bus at a nominal rate of 400kHz, corresponding to a low (or high) duration of 200 *pclk* (160Mhz) cycles. Register should not be set to values less than 8.
0x08	LssClocktoDataHold	6	0x3	Specifies the number of *pclk* cycles that Data must remain valid for after the falling edge of *lss_clk.* Minimum value is 3 cycles, and must to programmed to be less than *LssClockHighLowDuration.*
LSS bus 0 registers				

0x10	Lss0IntStatus	3	0x0	LSS bus 0 interrupt status registers Bit 0 - command completed successfully Bit 1 - error during processing of command, not -acknowledge received after transmission of primary id byte on LSS bus 0 Bit 2 - error during processing of command, not -acknowledge received after transmission of data byte on LSS bus 0 All the bits in *Lss0IntStatus* are cleared when the *Lss0Cmd* register gets written to. (Read only register)
0x14	Lss0CurrentState	4	0x0	Gives the current state of the LSS bus 0 state machine. (Read only register). (Encoding will be specified upon state machine implementation)
0x18	Lss0Cmd	21	0x00_0000	Command register defining sequence of events to perform on LSS bus 0 before interrupting CPU. A write to this register causes all the bits in the *Lss0IntStatus* register to be cleared as well as generating a *lss0_new_cmd* pulse.
0x1C - 0x2C	Lss0Buffer[4:0]	5x32	0x0000_0000	LSS Data buffer. Should be filled with transmit data before transmit command, or read data bytes received after a valid read command.
LSS bus 1 registers				
0x30	Lss1IntStatus	3	0x0	LSS bus 1 interrupt status registers Bit 0 - command completed successfully Bit 1 - error during processing of command, not -acknowledge received after transmission of primary id byte on LSS bus 1 Bit 2 - error during processing of command, not -acknowledge received after transmission of data byte on LSS bus 1 All the bits in *Lss1IntStatus* are cleared when

				the *Lss1Cmd* register gets written to. (Read only register)
0x34	Lss1CurrentState	4	0x0	Gives the current state of the LSS bus 1 state machine. (Read only register) (Encoding will be specified upon state machine implementation)
0x38	Lss1Cmd	21	0x00_0000	Command register defining sequence of events to perform on LSS bus 1 before interrupting CPU. A write to this register causes all the bits in the *Lss1IntStatus* register to be cleared as well as generating a *lss1_new_cmd* pulse.
0x3C - 0x4C	Lss1Buffer[4:0]	5x32	0x0000_0000	LSS Data buffer. Should be filled with transmit data before transmit command, or read data bytes received after a valid read command.
Debug registers				
0x50	LssDebugSel[6:2]	5	0x00	Selects register for debug output. This value is used as the input to the register decode logic instead of *cpu_adr[6:2]* when the LSS block is not being accessed by the CPU, i.e. when *cpu_lss_sel* is 0. The output *lss_cpu_debug_valid* is asserted to indicate that the data on *lss_cpu_data* is valid debug data. This data can be mutliplexed onto chip pins during debug mode.

19.3.2.1 LSS command registers

The LSS command registers define a sequence of events to perform on the respective LSS bus before issuing an interrupt to the CPU. There is a separate command register and interrupt for each LSS bus. The format of the command is given in Table 104. The CPU writes to the command register to initiate a sequence of events on an LSS bus. Once the sequence of events has completed or an error has occurred, an interrupt is sent back to the CPU.

Some example commands are:

- a single START condition (*Start* = 1, *IdByteEnable* = 0, *RdWrEnable* = 0, *Stop* = 0)
- a single STOP condition (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 0, *Stop* = 1)
- a START condition followed by transmission of the id byte (*Start* = 1, *IdByteEnable* = 1, *RdWrEnable* = 0, *Stop* = 0, *IdByte* contains primary id byte)
- a write transfer of 20 bytes from the data buffer (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 1, *RdWrSense* = 0, *Stop* = 0, *TxRxByteCount* = 20)

- a read transfer of 8 bytes into the data buffer (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 1, *RdWrSense* = 1, *ReadNack = 0, Stop* = 0, *TxRxByteCount* = 8)
- a complete read transaction of 16 bytes (*Start* = 1, *IdByteEnable* = 1, *RdWrEnable* = 1, *RdWrSense* = 1, *ReadNack = 1, Stop* = 1, *IdByte* contains primary id byte, *TxRxByteCount* = 16), etc.

The CPU can thus program the number of bytes to be transmitted or received (up to a maximum of 20) on the LSS bus before it gets interrupted. This allows it to insert arbitrary delays in a transfer at a byte boundary. For example the CPU may want to transmit 30 bytes to a QA chip but insert a delay between the 20th and 21st bytes sent. It does this by first writing 20 bytes to the data buffer. It then writes a command to generate a START condition, send the primary id byte and then transmit the 20 bytes from the data buffer. When interrupted by the LSS block to indicate successful completion of the command the CPU can then write the remaining 10 bytes to the data buffer. It can then wait for a defined period of time before writing a command to transmit the 10 bytes from the data buffer and generate a STOP condition to terminate the transaction over the LSS bus.

An interrupt to the CPU is generated for one cycle when any bit in *LssNIntStatus* is set. The CPU can read *LssNIntStatus* to discover the source of the interrupt. The *LssNIntStatus* registers are cleared when the CPU writes to the *LssNCmd* register. A null command write to the *LssNCmd* register will cause the *LssNIntStatus* registers to clear and no new command to start. A null command is defined as *Start*, *IdbyteEnable*, *RdWrEnable* and *Stop* all set to zero.

Table 104. LSS command register description

bit(s)	name	Description
0	Start	When 1, issue a START condition on the LSS bus.
1	IdByteEnable	ID byte transmit enable: 1 - transmit byte in *IdByte* field 0 - ignore byte in *IdByte* field
2	RdWrEnable	Read/write transfer enable: 0 - ignore settings of *RdWrSense*, *ReadNack* and *TxRxByteCount* 1 - if RdWrSense is 0, then perform a write transfer of *TxRxByteCount* bytes from the data buffer. if RdWrSense is 1, then perform a read transfer of *TxRxByteCount* bytes into the data buffer. Each byte should be acknowledged and the last byte received is acknowledged/not-acknowledged according to the setting of *ReadNack*.

3	RdWrSense	Read/write sense indicator: 0 - write 1 – read
4	ReadNack	Indicates, for a read transfer, whether to issue an acknowledge or a not-acknowledge after the last byte received (indicated by *TxRxByteCount*). 0 - issue acknowledge after last byte received 1 - issue not-acknowledge after last byte received.
5	Stop	When 1, issue a STOP condition on the LSS bus.
7:6	reserved	Must be 0
15:8	IdByte	Byte to be transmitted if *IdByteEnable* is 1. Bit 8 corresponds to the LSB.
20:16	TxRxByteCount	Number of bytes to be transmitted from the data buffer or the number of bytes to be received into the data buffer. The maximum value that should be programmed is 20, as the size of the data buffer is 20 bytes. Valid values are 1 to 20, 0 is valid when RdWrEnable = 0, other cases are invalid andundefined.

The data buffer is implemented in the LSS master block. When the CPU writes to the *LssNBuffer* registers the data written is presented to the LSS master block via the *lssN_buffer_wrdata* bus and configuration registers block pulses the *lssN_buffer_wen* bit corresponding to the register written. For example if *LssNBuffer[2]* is written to *lssN_buffer_wen[2]* will be pulsed. When the CPU reads the *LssNBuffer* registers the configuration registers block reflect the *lssN_buffer_rdata* bus back to the CPU.

19.3.3 LSS master unit

The LSS master unit is instantiated for both LSS bus 0 and LSS bus 1. It controls transactions on the LSS bus by means of the state machine shown in Figure 83, which interprets the commands that are written by the CPU. It also contains a single 20 byte data buffer used for transmitting and receiving data.

The CPU can write data to be transmitted on the LSS bus by writing to the *LssNBuffer* registers. It can also read data that the LSS master unit receives on the LSS bus by reading the same registers. The LSS master always transmits or receives bytes to or from the data buffer in the same order.

For a transmit command, *LssNBuffer[0][7:0]* gets transmitted first, then *LssNBuffer[0][15:8], LssNBuffer[0][23:16], LssNBuffer[0][31:24], LssNBuffer[1][7:0]* and so on until *TxRxByteCount* number of bytes are transmitted. A receive command fills data to the buffer in the same order. Each new command the buffer start point is reset.

All state machine outputs, flags and counters are cleared on reset. After a reset the state machine goes to the *Reset* state and initialises the LSS pins (*lss_clk* is set to 1, *lss_data* is tristated and

allowed to be pulled up to 1). When the reset condition is removed the state machine transitions to the *Wait* state.

It remains in the *Wait* state until *lss_new_cmd* equals 1. If the *Start* bit of the command is 0 the state machine proceeds directly to the *CheckIdByteEnable* state. If the *Start* bit is 1 it proceeds to the *GenerateStart* state and issues a START condition on the LSS bus.

In the *CheckIdByteEnable* state, if the *IdByteEnable* bit of the command is 0 the state machine proceeds directly to the *CheckRdWrEnable* state. If the *IdByteEnable* bit is 1 the state machine enters the *SendIdByte* state and the byte in the *IdByte* field of the command is transmitted on the LSS. The *WaitForIdAck* state is then entered. If the byte is acknowledged, the state machine proceeds to the *CheckRdWrEnable* state. If the byte is not-acknowledged, the state machine proceeds to the *GenerateInterrupt* state and issues an interrupt to indicate a not-acknowledge was received after transmission of the primary id byte.

In the *CheckRdWrEnable* state, if the *RdWrEnable* bit of the command is 0 the state machine proceeds directly to the *CheckStop* state. If the *RdWrEnable* bit is 1, *count* is loaded with the value of the *TxRxByteCount* field of the command and the state machine enters either the *ReceiveByte* state if the *RdWrSense* bit of the command is 1 or the *TransmitByte* state if the *RdWrSense* bit is 0.

For a write transaction, the state machine keeps transmitting bytes from the data buffer, decrementing *count* after each byte transmitted, until *count* is 1. If all the bytes are successfully transmitted the state machine proceeds to the *CheckStop* state. If the slave QA chip not-acknowledges a transmitted byte, the state machine indicates this error by issuing an interrupt to the CPU and then entering the *GenerateInterrupt* state.

For a read transaction, the state machine keeps receiving bytes into the data buffer, decrementing *count* after each byte transmitted, until *count* is 1. After each byte received the LSS master must issue an acknowledge. After the last expected byte (i.e. when count is 1) the state machine checks the *ReadNack* bit of the command to see whether it must issue an acknowledge or not-acknowledge for that byte. The *CheckStop* state is then entered.

In the *CheckStop* state, if the *Stop* bit of the command is 0 the state machine proceeds directly to the *GenerateInterrupt* state. If the *Stop* bit is 1 it proceeds to the *GenerateStop* state and issues a STOP condition on the LSS bus before proceeding to the *GenerateInterrupt* state. In both cases an interrupt is issued to indicate successful completion of the command.

The state machine then enters the *Wait* state to await the next command. When the state machine reenters the *Wait* state the output pins (*lss_data* and *lss_clk*) are not changed, they retain the state of the last command. This allows the possibility of multi-command transactions. The CPU may abort the current transfer at any time by performing a write to the *Reset* register of the LSS block.

19.3.3.1 START and STOP generation

START and STOP conditions, which signal the beginning and end of data transmission, occur when the LSS master generates a falling and rising edge respectively on the data while the clock is high.

In the *GenerateStart* state, *lss_gpio_clk* is held high with *lss_gpio_e* remaining deasserted (so the data line is pulled high externally) for *LssClockHighLowDuration pclk* cycles. Then *lss_gpio_e* is asserted and *lss_gpio_dout* is pulled low (to drive a 0 on the data line, creating a falling edge) with *lss_gpio_clk* remaining high for another *LssClockHighLowDuration pclk* cycles.

In the *GenerateStop* state, both *lss_gpio_clk* and *lss_gpio_dout* are pulled low followed by the assertion of *lss_gpio_e* to drive a 0 while the clock is low. After *LssClockHighLowDuration pclk* cycles, *lss_gpio_clk* is set high. After a further *LssClockHighLowDuration pclk* cycles, *lss_gpio_e* is deasserted to release the data bus and create a rising edge on the data bus during the high period of the clock.

If the bus is not in the required state for start and stop generation (*lss_clk*=1, *lss_data*=1 for start, and *lss_clk*=1, *lss_data*=0), the state machine moves the bus to the correct state and proceeds as described above. Figure 82 shows the transition timing from any bus state to start and stop generation

19.3.3.2 Clock pulse generation

The LSS master holds *lss_gpio_clk* high while the LSS bus is inactive. A clock pulse is generated for each bit transmitted or received over the LSS bus. It is generated by first holding *lss_gpio_clk* low for *LssClockHighLowDuration pclk* cycles, and then high for *LssClockHighLowDuration pclk* cycles.

19.3.3.3 Data De-glitching

When data is received in the LSS block it is passed to a de-glitching circuit. The de-glitch circuit samples the data 3 times on pclk and compares the samples. If all 3 samples are the same then the data is passed, otherwise the data is ignored.

Note that the LSS data input on SoPEC is double registered in the GPIO block before being passed to the LSS.

19.3.3.4 Data reception

The input data, *gpio_lss_di*, is first synchronised to the *pclk* domain by means of two flip-flops clocked by *pclk* (the double register resides in the GPIO block) . The LSS master generates a clock pulse for each bit received. The output *lss_gpio_e* is deasserted *LssClockToDataHold pclk* cycles after the falling edge of *lss_gpio_clk* to release the data bus. The value on the synchronised *gpio_lss_di* is sampled *Tstrobe* number of clock cycles after the rising edge of *lss_gpio_clk* (the data is de-glitched over a further 3 stage register to avoid possible glitch detection). See Figure 84 for further timing information.

In the *ReceiveByte* state, the state machine generates 8 clock pulses. At each *Tstrobe* time after the rising edge of *lss_gpio_clk* the synchronised *gpio_lss_di* is sampled. The first bit sampled is *LssNBuffer[0][7]*, the second *LssNBuffer[0][6]*, etc to *LssNBuffer[0][0].* For each byte received the state machine either sends an NAK or an ACK depending on the command configuration and the number of bytes received.

In the *SendNack* state the state machine generates a single clock pulse. *lss_gpio_e* is deasserted and the LSS data line is pulled high externally to issue a not-acknowledge.

In the *SendAck* state the state machine generates a single clock pulse. *lss_gpio_e* is asserted and a 0 driven on *lss_gpio_dout* after *lss_gpio_clk* falling edge to issue an acknowledge.

19.3.3.5 Data transmission

The LSS master generates a clock pulse for each bit transmitted. Data is output on the LSS bus on the falling edge of *lss_gpio_clk*.

When the LSS master drives a logical zero on the bus it will assert *lss_gpio_e* and drive a 0 on *lss_gpio_dout* after *lss_gpio_clk* falling edge. *lss_gpio_e* will remain asserted and *lss_gpio_dout* will remain low until the next *lss_clk* falling edge.

When the LSS master drives a logical one *lss_gpio_e* should be deasserted at *lss_gpio_clk* falling edge and remain deasserted at least until the next *lss_gpio_clk* falling edge. This is because the LSS bus will be externally pulled up to logical one via a pull-up resistor.

In the *SendId byte* state, the state machine generates 8 clock pulses to transmit the byte in the *IdByte* field of the current valid command. On each falling edge of *lss_gpio_clk* a bit is driven on the data bus as outlined above. On the first falling edge *IdByte[7]* is driven on the data bus, on the second falling edge *IdByte[6]* is driven out, etc.

In the *TransmitByte* state, the state machine generates 8 clock pulses to transmit the byte at the output of the transmit FIFO. On each falling edge of *lss_gpio_clk* a bit is driven on the data bus as outlined above. On the first falling edge *LssNBuffer[0][7]* is driven on the data bus, on the second falling edge *LssNBuffer[0][6]* is driven out, etc on to *LssNBuffer[0][7]* bits.

In the *WaitForAck* state, the state machine generates a single clock pulse. At *Tstrobe* time after the rising edge of *lss_gpio_clk* the synchronized *gpio_lss_di* is sampled. A 0 indicates an acknowledge and *ack_detect* is pulsed, a 1 indicates a not-acknowledge and *nack_detect* is pulsed.

19.3.3.6 Data rate control

The CPU can control the data rate by setting the clock period of the LSS bus clock by programming appropriate value in *LssClockHighLowDuration*. The default setting for the register is 200 (*pclk* cycles) which corresponds to transmission rate of 400kHz on the LSS bus (the *lss_clk* is high for *LssClockHighLowDuration* cycles then low for *LssClockHighLowDuration* cycles). The *lss_clk* will always have a 50:50 duty cycle. The *LssClockHighLowDuration* register should not be set to values less than 8.

The hold time of *lss_data* after the falling edge of *lss_clk* is programmable by the *LssClocktoDataHold* register. This register should not be programmed to less than 2 or greater than the *LssClockHighLowDuration* value.

19.3.3.7 LSS master timing parameters

The LSS master timing parameters are shown in Figure 84 and the associated values are shown in Table 105 .

Table 105. LSS master timing parameters

Parameter	Description	min	nom	max	unit
LSS Master Driving					

Tp	LSS clock period divided by 2	8	200	FFFF	pclk cycles
Tstart_delay	Time to start data edge from rising clock edge	Tp + *LssClocktoDataHold*			pclk cycles
Tstop_delay	Time to stop data edge from rising clock edge	Tp + *LssClocktoDataHold*			pclk cycles
Tdata_setup	Time from data setup to rising clock edge	Tp - 2 - *LssClocktoDataHold*			pclk cycles
Tdata_hold	Time from falling clock edge to data hold	*LssClocktoDataHold*			pclk cycles
Tack_setup	Time that outgoing (N)Ack is setup before *lss_clk* rising edge	Tp - 2 - *LssClocktoDataHold*			pclk cycles
Tack_hold	Time that outgoing (N)Ack is held after *lss_clk* falling edge	*LssClocktoDataHold*			pclk cycles
LSS Master Sampling					
Tstrobe	LSS master strobe point for incoming data and (N)Ack values	Tp -2		Tp -2	pclk cycles

DRAM SUBSYSTEM

20 DRAM Interface Unit (DIU)

20.1 OVERVIEW

Figure 85 shows how the DIU provides the interface between the on-chip 20 Mbit embedded DRAM and the rest of SoPEC. In addition to outlining the functionality of the DIU, this chapter provides a top-level overview of the memory storage and access patterns of SoPEC and the buffering required in the various SoPEC blocks to support those access requirements.

The main functionality of the DIU is to arbitrate between requests for access to the embedded DRAM and provide read or write accesses to the requesters. The DIU must also implement the initialisation sequence and refresh logic for the embedded DRAM.

The arbitration scheme uses a fully programmable timeslot mechanism for non-CPU requesters to meet the bandwidth and latency requirements for each unit, with unused slots re-allocated to provide best effort accesses. The CPU is allowed high priority access, giving it minimum latency, but allowing bounds to be placed on its bandwidth consumption.

The interface between the DIU and the SoPEC requesters is similar to the interface on PEC1 i.e. separate control, read data and write data busses.

The embedded DRAM is used principally to store:

- CPU program code and data.
- PEP (re)programming commands.
- Compressed pages containing contone, bi-level and raw tag data and header information.
- Decompressed contone and bi-level data.
- Dotline store during a print.

- Print setup information such as tag format structures, dither matrices and dead nozzle information.

20.2 IBM Cu-11 Embedded DRAM

20.2.1 Single bank

SoPEC will use the 1.5 V core voltage option in IBM's 0.13 µm class Cu-11 process.

The random read/write cycle time and the refresh cycle time is 3 cycles at 160 MHz [16]. An open page access will complete in 1 cycle if the page mode select signal is clocked at 320 MHz or 2 cycles if the page mode select signal is clocked every 160 MHz cycle. The page mode select signal will be clocked at 160 MHz in SoPEC in order to simplify timing closure. The DRAM word size is 256 bits.

Most SoPEC requesters will make single 256 bit DRAM accesses (see Section 20.4). These accesses will take 3 cycles as they are random accesses i.e. they will most likely be to a different memory row than the previous access.

The entire 20 Mbit DRAM will be implemented as a single memory bank. In Cu-11, the maximum single instance size is 16 Mbit. The first 1 Mbit tile of each instance contains an area overhead so the cheapest solution in terms of area is to have only 2 instances. 16 Mbit and 4Mbit instances would together consume an area of 14.63 mm^2 as would 2 times 10 Mbit instances. 4 times 5 Mbit instances would require 17.2 mm^2.

The instance size will determine the frequency of refresh. Each refresh requires 3 clock cycles. In Cu-11 each row consists of 8 columns of 256-bit words. This means that 10 Mbit requires 5120 rows. A complete DRAM refresh is required every 3.2 ms. Two times 10 Mbit instances would require a refresh every 100 clock cycles, if the instances are refreshed in parallel.

The SoPEC DRAM will be constructed as two 10 Mbit instances implemented as a single memory bank.

20.3 SoPEC Memory Usage Requirements

The memory usage requirements for the embedded DRAM are shown in Table 106 .

Table 106. Memory Usage Requirements

Block	Size	Description
Compressed page store	2048 Kbytes	Compressed data page store for Bi-level and contone data
Decompressed Contone Store	108 Kbyte	13824 lines with scale factor 6 = 2304 pixels, store 12 lines, 4 colors = 108 kB 13824 lines with scale factor 5 = 2765 pixels, store 12 lines, 4 colors = 130 kB
Spot line store	5.1 Kbyte	13824 dots/line so 3 lines is 5.1 kB

Tag Format Structure	Typically 12 Kbyte (2.5 mm tags @ 800 dpi)	55 kB in for 384 dot line tags 2.5 mm tags (1/10th inch) @ 1600 dpi require 160 dot lines = 160/384 x55 or 23 kB 2.5 mm tags (1/10th inch) @ 800 dpi require 80/384 x55 = 12 kB
Dither Matrix store	4 Kbytes	64x64 dither matrix is 4 kB 128x128 dither matrix is 16 kB 256x256 dither matrix is 64 kB
DNC Dead Nozzle Table	1.4 Kbytes	Delta encoded, (10 bit delta position + 6 dead nozzle mask) x% Dnozzle 5% dead nozzles requires (10+6)x 692 Dnoz zles = 1.4 Kbytes
Dot-line store	369.6 Kbytes	Assume each color row is separated by 5 dot lines on the print head The dot line store will be 0+5+10…50+55 = 330 half dot lines + 48 extra half dot lines (4 per dot row) + 60 extra half dot lines estimated to account for printhead misalignment = 438 half dot lines. 438 half dot lines of 6912 dots = 369.6Kbytes
PCU Program code	8 Kbytes	1024 commands of 64 bits = 8 kB
CPU	64 Kbytes	Program code and data
TOTAL	2620 Kbytes (12 Kbyte TFS storage)	

Note:

- Total storage is fixed to 2560 Kbytes to align to 20 Mbit DRAM. This will mean that less space than noted in Table may be available for the compressed band store.

20.4 SoPEC Memory Access Patterns

Table 107 shows a summary of the blocks on SoPEC requiring access to the embedded DRAM and their individual memory access patterns. Most blocks will access the DRAM in single 256-bit accesses. All accesses must be padded to 256-bits except for 64-bit CDU write accesses and CPU write accesses. Bits which should not be written are masked using the individual DRAM bit write inputs or byte write inputs, depending on the foundry. Using single 256-bit accesses means that the buffering required in the SoPEC DRAM requesters will be minimized.

Table 107. Memory access patterns of SoPEC DRAM Requesters

DRAM requester	Direction	Memory access pattern
CPU	R	Single 256-bit reads.
	W	Single 32-bit, 16-bit or 8-bit writes.
SCB	R	Single 256-bit reads.
	W	Single 256-bit writes, with byte enables.
CDU	R	Single 256-bit reads of the compressed contone data.
	W	Each CDU access is a write to 4 consecutive DRAM words in the same row but only 64 bits of each word are written with the remaining bits write masked. The access time for this 4 word page mode burst is 3 + 2 + 2 +2 = 9 cycles if the page mode select signal is clocked at 160 MHz.
CFU	R	Single 256 bit reads.
LBD	R	Single 256 bit reads.
SFU	R	Separate single 256 bit reads for previous and current line but sharing the same DIU interface
	W	Single 256 bit writes.
TE(TD)	R	Single 256 bit reads. Each read returns 2 times 128 bit tags.
TE(TFS)	R	Single 256 bit reads. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.
HCU	R	Single 256 bit reads. 128 x 128 dither matrix requires 4 reads per line with double buffering. 256 x 256 dither matrix requires 8 reads at the end of the line with single buffering.
DNC	R	Single 256 bit dead nozzle table reads. Each dead nozzle table read con tains 16 dead-nozzle tables entries each of 10 delta bits plus 6 dead nozzle mask bits.
DWU	W	Single 256 bit writes since enable/disable DRAM access per color plane.
LLU	R	Single 256 bit reads since enable/disable DRAM access per color plane.
PCU	R	Single 256 bit reads. Each PCU command is 64 bits so each 256 bit word can contain 4 PCU commands. PCU reads from DRAM used for reprogramming PEP should be executed with minimum latency. If this occurs between pages then there will be free bandwidth as most of the other SoPEC Units will not be requesting from DRAM. If this occurs between bands then the LDB, CDU and TE bandwidth will be free. So the PCU should have a high priority to access to any spare bandwidth.

Refresh		Single refresh.

20.5 BUFFERING REQUIRED IN SOPEC DRAM REQUESTERS

If each DIU access is a single 256-bit access then we need to provide a 256-bit double buffer in the DRAM requester. If the DRAM requester has a 64-bit interface then this can be implemented as an 8 x 64-bit FIFO.

Table 108. Buffer sizes in SoPEC DRAM requesters

DRAM Requester	Direction	Access patterns	Buffering required in block
CPU	R	Single 256-bit reads.	Cache.
	W	Single 32-bit writes but allowing 16-bit or byte addressable writes.	None.
SCB	R	Single 256-bit reads.	Double 256-bit buffer.
	W	Single 256-bit writes, with byte enables.	Double 256-bit buffer.
CDU	R	Single 256-bit reads of the compressed contone data.	Double 256-bit buffer.
	W	Each CDU access is a write to 4 consecutive DRAM words in the same row but only 64 bits of each word are written with the remaining bits write masked.	Double half JPEG block buffer.
CFU	R	Single 256 bit reads.	Triple 256-bit buffer.
LBD	R	Single 256 bit reads.	Double 256-bit buffer.
SFU	R	Separate single 256 bit reads for previous and cur rent line but sharing the same DIU interface	Double 256-bit buffer for each read channel.
	W	Single 256 bit writes.	Double 256-bit buffer.
TE(TD)	R	Single 256 bit reads.	Double 256-bit buffer.
TE(TFS)	R	Single 256 bit reads. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.	Double line-buffer for 136 bytes implemented in TE.
HCU	R	Single 256 bit reads. 128 x 128 dither matrix requires 4 reads per line with double buffering. 256 x 256 dither matrix requires 8 reads at the end of the line with single buffering.	Configurable between dou ble 128 byte buffer and single 256 byte buffer.
DNC	R	Single 256 bit reads	Double 256-bit buffer. Deeper buffering could

			be specified to cope with local clusters of dead nozzles.
DWU	W	Single 256 bit writes per enabled odd/even color plane.	Double 256-bit buffer per color plane.
LLU	R	Single 256 bit reads per enabled odd/even color plane.	Double 256-bit buffer per color plane.
PCU	R	Single 256 bit reads. Each PCU command is 64 bits so each 256 bit DRAM read can contain 4 PCU com mands. Requested command is read from DRAM together with the next 3 contiguous 64-bits which are cached to avoid unnecessary DRAM reads.	Single 256-bit buffer.
Refresh		Single refresh.	None.

20.6 SoPEC DIU BANDWIDTH REQUIREMENTS

Table 109. SoPEC DIU Bandwidth Requirements

Block Name	Direction	Number of cycles between each 256-bit DRAM access to meet peak bandwidth	Peak Bandwidth which must be supplied (bits/cycle)	Average Bandwidth (bits/cycle)	Example number of allocated timeslots[1]
CPU	R				
	W				
SCB	R				
	W	3482	0.734	0.3933	1
CDU	R	128 (SF = 4), 288 (SF = 6), 1:1 compression4	64/n2 (SF=n), 1.8 (SF = 6), 4 (SF = 4) (1:1 compression)	32/10*n2 (SF=n), 0.09 (SF = 6), 0.2 (SF = 4) (10:1 compression)5	1 (SF=6) 2 (SF=4)
	W	For individual accesses: 16 cycles (SF = 4), 36 cycles (SF = 6), n2 cycles (SF=n). Will be	64/n2 (SF=n), 1.8 (SF = 6), 4 (SF = 4)	32/n2 (SF=n)7, 0.9 (SF = 6), 2 (SF = 4)	2 (SF=6)8 4 (SF=4)

		implemented as a page mode burst of 4 accesses every 64 cycles (SF = 4), 144 (SF =6), 4*n2 (SF =n) cycles6			
CFU	R	32 (SF = 4), 48 (SF = 6)9	32/n (SF=n), 5.4 (SF = 6), 8 (SF = 4)	32/n (SF=n), 5.4 (SF = 6), 8 (SF = 4)	6 (SF=6) 8 (SF=4)
LBD	R	256 (1:1 compression)10	1 (1:1 compression)	0.1 (10:1 compression)11	1
SFU	R	12812	2	2	2
	W	25613	1	1	1
TE(TD)	R	25214	1.02	1.02	1
TE(TFS)	R	5 reads per line15	0.093	0.093	0
HCU	R	4 reads per line for 128 x 128 dither matrix16	0.074	0.074	0
DNC	R	106 (5% dead-nozzles 10-bit delta encoded)17	2.4 (clump of dead nozzles)	0.8 (equally spaced dead nozzles)	3
DWU	W	6 writes every 25618	6	6	6
LLU	R	8 reads every 25619	8	6	8
PCU	R	25620	1	1	1
Refresh		10021	2.56	2.56	3 (effective)
TOTAL			SF = 6: 34.9 SF = 4: 41.9 excluding CPU	SF = 6: 27.5 SF = 4: 31.2 excluding CPU	SF = 6: 36 excluding CPU. SF= 4: 41 excluding CPU

Notes:

1: The number of allocated timeslots is based on 64 timeslots each of 1 bit/cycle but broken down to a granularity of 0.25 bit/cycle. Bandwidth is allocated based on peak bandwidth.

2: Wire-speed bandwidth for a 4 wire SCB configuration is 32 Mbits/s for each wire plus 12 Mbit/s for USB. This is a maximum of 138 Mbit/s. The maximum effective data rate is 26 Mbits/s for each wire plus 8 Mbit/s for USB. This is 112 Mbit/s. 112 Mbit/s is 0.734 bits/cycle or 256 bits every 348 cycles.

3: Wire-speed bandwidth for a 2 wire SCB configuration is 32 Mbits/s for each wire plus 12 Mbit/s for USB. This is a maximum of 74 Mbit/s. The maximum effective data rate is 26 Mbits/s for each wire plus 8 Mbit/s for USB. This is 60 Mbit/s. 60 Mbit/s is 0.393 bits/cycle or 256 bits every 650 cycles.

4: At 1:1 compression CDU must read a 4 color pixel (32 bits) every SF^2 cycles.

5: At 10:1 average compression CDU must read a 4 color pixel (32 bits) every $10*SF^2$ cycles.

6: 4 color pixel (32 bits) is required, on average, by the CFU every SF^2 (scale factor) cycles.

The time available to write the data is a function of the size of the buffer in DRAM. 1.5 buffering means 4 color pixel (32 bits) must be written every $SF^2 / 2$ (scale factor) cycles. Therefore, at a scale factor of SF, 64 bits are required every SF^2 cycles.

Since 64 valid bits are written per 256-bit write (Figure n page379 on page **Error! Bookmark not defined.**) then the DRAM is accessed every SF^2 cycles i.e. at SF4 an access every 16 cycles, at SF6 an access every 36 cycles.

If a page mode burst of 4 accesses is used then each access takes (3 + 2 + 2 +2) equals 9 cycles. This means at SF, a set of 4 back-to-back accesses must occur every $4*SF^2$ cycles. This assumes the page mode select signal is clocked at 160 MHz. CDU timeslots therefore take 9 cycles.

For scale factors lower than 4 double buffering will be used.

7: The peak bandwidth is twice the average bandwidth in the case of 1.5 buffering.

8: Each CDU(W) burst takes 9 cycles instead of 4 cycles for other accesses so CDU timeslots are longer.

9: 4 color pixel (32 bits) read by CFU every SF cycles. At SF4, 32 bits is required every 4 cycles or 256 bits every 32 cycles. At SF6, 32bits every 6 cycles or 256 bits every 48 cycles.

10: At 1:1 compression require 1 bit/cycle or 256 bits every 256 cycles.

11: The average bandwidth required at 10:1 compression is 0.1 bits/cycle.

12: Two separate reads of 1 bit/cycle.

13: Write at 1 bit/cycle.

14: Each tag can be consumed in at most 126 dot cycles and requires 128 bits. This is a maximum rate of 256 bits every 252 cycles.

15: 17 x 64 bit reads per line in PEC1 is 5 x 256 bit reads per line in SoPEC. Double-line buffered storage.

16: 128 bytes read per line is 4 x 256 bit reads per line. Double-line buffered storage.

17: 5% dead nozzles 10-bit delta encoded stored with 6-bit dead nozzle mask requires 0.8 bits/cycle read access or a 256-bit access every 320 cycles. This assumes the dead nozzles are evenly spaced out. In practice dead nozzles are likely to be clumped. Peak bandwidth is estimated as 3 times average bandwidth.

18: 6 bits/cycle requires 6 x 256 bit writes every 256 cycles.

19: 6 bits/160 MHz SoPEC cycle average but will peak at 2 x 6 bits per 106 MHz print head cycle or 8 bits/ SoPEC cycle. The PHI can equalise the DRAM access rate over the line so that the

peak rate equals the average rate of 6 bits/ cycle. The print head is clocked at an effective speed of 106 MHz.

20: Assume one 256 read per 256 cycles is sufficient i.e. maximum latency of 256 cycles per access is allowable.

21: Refresh must occur every 3.2 ms. Refresh occurs row at a time over 5120 rows of 2 parallel 10 Mbit instances. Refresh must occur every 100 cycles. Each refresh takes 3 cycles.

20.7 DIU BUS TOPOLOGY

20.7.1 Basic topology

Table 110. SoPEC DIU Requesters

Read	Write	Other
CPU	CPU	Refresh
SCB	SCB	
CDU	CDU	
CFU	SFU	
LBD	DWU	
SFU		
TE(TD)		
TE(TFS)		
HCU		
DNC		
LLU		
PCU		

Table 110 shows the DIU requesters in SoPEC. There are 12 read requesters and 5 write requesters in SoPEC as compared with 8 read requesters and 4 write requesters in PEC1. Refresh is an additional requester.

In PEC1, the interface between the DIU and the DIU requesters had the following main features:

- separate control and address signals per DIU requester multiplexed in the DIU according to the arbitration scheme,
- separate 64-bit write data bus for each DRAM write requester multiplexed in the DIU,
- common 64-bit read bus from the DIU with separate enables to each DIU read requester.

Timing closure for this bussing scheme was straight-forward in PEC1. This suggests that a similar scheme will also achieve timing closure in SoPEC. SoPEC has 5 more DRAM requesters but it will be in a 0.13 um process with more metal layers and SoPEC will run at approximately the same speed as PEC1.

Using 256-bit busses would match the data width of the embedded DRAM but such large busses may result in an increase in size of the DIU and the entire SoPEC chip. The SoPEC requestors would require double 256-bit wide buffers to match the 256-bit busses. These buffers, which must

be implemented in flip-flops, are less area efficient than 8-deep 64-bit wide register arrays which can be used with 64-bit busses. SoPEC will therefore use 64-bit data busses. Use of 256-bit busses would however simplify the DIU implementation as local buffering of 256-bit DRAM data would not be required within the DIU.

20.7.1.1 CPU DRAM access

The CPU is the only DIU requestor for which access latency is critical. All DIU write requesters transfer write data to the DIU using separate point-to-point busses. The CPU will use the *cpu_dataout[31:0]* bus. CPU reads will not be over the shared 64-bit read bus. Instead, CPU reads will use a separate 256-bit read bus.

20.7.2 Making more efficient use of DRAM bandwidth

The embedded DRAM is 256-bits wide. The 4 cycles it takes to transfer the 256-bits over the 64-bit data busses of SoPEC means that effectively each access will be at least 4 cycles long. It takes only 3 cycles to actually do a 256-bit random DRAM access in the case of IBM DRAM.

20.7.2.1 Common read bus

If we have a common read data bus, as in PEC1, then if we are doing back to back read accesses the next DRAM read cannot start until the read data bus is free. So each DRAM read access can occur only every 4 cycles. This is shown in Figure 86 with the actual DRAM access taking 3 cycles leaving 1 unused cycle per access.

20.7.2.2 Interleaving CPU and non-CPU read accesses

The CPU has a separate 256-bit read bus. All other read accesses are 256-bit accesses are over a shared 64-bit read bus. Interleaving CPU and non-CPU read accesses means the effective duration of an interleaved access timeslot is the DRAM access time (3 cycles) rather than 4 cycles.

Figure 87 shows interleaved CPU and non-CPU read accesses.

20.7.2.3 Interleaving read and write accesses

Having separate write data busses means write accesses can be interleaved with each other and with read accesses. So now the effective duration of an interleaved access timeslot is the DRAM access time (3 cycles) rather than 4 cycles. Interleaving is achieved by ordering the DIU arbitration slot allocation appropriately.

Figure 88 shows interleaved read and write accesses. Figure 89 shows interleaved write accesses.

256-bit write data takes 4 cycles to transmit over 64-bit busses so a 256-bit buffer is required in the DIU to gather the write data from the write requester. The exception is CPU write data which is transferred in a single cycle.

Figure 89 shows multiple write accesses being interleaved to obtain 3 cycle DRAM access. Since two write accesses can overlap two sets of 256-bit write buffers and multiplexors to connect two write requestors simultaneously to the DIU are required.

Write requestors only require approximately one third of the total non-CPU bandwidth. This means that a rule can be introduced such that non-CPU write requestors are not allocated

adjacent timeslots. This means that a single 256-bit write buffer and multiplexor to connect the one write requestor at a time to the DIU is all that is required.

Note that if the rule prohibiting back-to-back non-CPU writes is not adhered to, then the second write slot of any attempted such pair will be disregarded and re-allocated under the unused read round-robin scheme.

20.7.3 Bus widths summary

Table 111. SoPEC DIU Requesters Data Bus Width

Read	Bus access width	Write	Bus access width
CPU	256 (separate)	CPU	32
SCB	64 (shared)	SCB	64
CDU	64 (shared)	CDU	64
CFU	64 (shared)	SFU	64
LBD	64 (shared)	DWU	64
SFU	64 (shared)		
TE(TD)	64 (shared)		
TE(TFS)	64 (shared)		
HCU	64 (shared)		
DNC	64 (shared)		
LLU	64 (shared)		
PCU	64 (shared)		

20.7.4 Conclusions

Timeslots should be programmed to maximise interleaving of shared read bus accesses with other accesses for 3 cycle DRAM access. The interleaving is achieved by ordering the DIU arbitration slot allocation appropriately. CPU arbitration has been designed to maximise interleaving with non-CPU requesters

20.8 SoPEC DRAM ADDRESSING SCHEME

The embedded DRAM is composed of 256-bit words. However the CPU-subsystem may need to write individual bytes of DRAM. Therefore it was decided to make the DIU byte addressable. 22 bits are required to byte address 20 Mbit of DRAM.

Most blocks read or write 256 bit words of DRAM. Therefore only the top 17 bits i.e. bits 21 to 5 are required to address 256-bit word aligned locations.

The exceptions are

- CDU which can write 64-bits so only the top 19 address bits i.e. bits 21-3 are required.
- CPU writes can be 8, 16 or 32-bits. The *cpu_diu_wmask[1:0]* pins indicate whether to write 8, 16 or 32 bits.

All DIU accesses must be within the same 256-bit aligned DRAM word. The exception is the CDU write access which is a write of 64-bits to each of 4 contiguous 256-bit DRAM words.

20.8.1 Write Address Constaints Specific to the CDU

Note the following conditions which apply to the CDU write address, due to the four masked page-mode writes which occur whenever a CDU write slot is arbitrated.

- The CDU address presented to the DIU is cdu_diu_wadr[21:3].
- Bits [4:3] indicate which 64-bit segment out of 256 bits should be written in 4 successive masked page-mode writes.
- Each 10-Mbit DRAM macro has an input address port of width [15:0]. Of these bits, [2:0] are the "page address". Page-mode writes, where you just vary these LSBs (i.e. the "page" or column address), but keep the rest of the address constant, are faster than random writes. This is taken advantage of for CDU writes.
- To guarantee against trying to span a page boundary, the DIU treats "cdu_diu_wadr[6:5]" as being fixed at "00".
- From cdu_diu_wadr[21:3], a initial address of cdu_diu_wadr[21:7] , concatenated with "00", is used as the starting location for the first CDU write. This address is then auto-incremented a further three times.

20.9 DIU PROTOCOLS

The DIU protocols are

- Pipelined i.e. the following transaction is initiated while the previous transfer is in progress.
- Split transaction i.e. the transaction is split into independent address and data transfers.

20.9.1 Read Protocol except CPU

The SoPEC read requestors, except for the CPU, perform single 256-bit read accesses with the read data being transferred from the DIU in 4 consecutive cycles over a shared 64-bit read bus, *diu_data[63:0]*. The read address *<unit>_diu_radr[21:5]* is 256-bit aligned.

The read protocol is:

- *<unit>_diu_rreq* is asserted along with a valid *<unit>_diu_radr[21:5]*.
- The DIU acknowledges the request with *diu_<unit>_rack*. The request should be deasserted. The minimum number of cycles between *<unit>_diu_rreq* being asserted and the DIU generating an *diu_<unit>_rack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.14.10).
- The read data is returned on *diu_data[63:0]* and its validity is indicated by *diu_<unit>_rvalid*. The overall 256 bits of data are transferred over four cycles in the order : [63:0] -> [127:64] -> [191:128] -> [255:192].
- When four *diu_<unit>_rvalid* pulses have been received then if there is a further request *<unit>_diu_rreq* should be asserted again. *diu_<unit>_rvalid* will be always be asserted by the DIU for four consecutive cycles. There is a *fixed* gap of 2 cycles between *diu_<unit>_rack* and the first *diu_<unit>_rvalid* pulse. For more detail on the timing of such reads and the implications for back-to-back sequences, see Section 20.14.10.

20.9.2 Read Protocol for CPU

The CPU performs single 256-bit read accesses with the read data being transferred from the DIU over a dedicated 256-bit read bus for DRAM data, *dram_cpu_data[255:0]*. The read address *cpu_adr[21:5]* is 256-bit aligned.

The CPU DIU read protocol is:

- *cpu_diu_rreq* is asserted along with a valid *cpu_adr[21:5]*.
- The DIU acknowledges the request with *diu_cpu_rack*. The request should be deasserted. The minimum number of cycles between *cpu_diu_rreq* being asserted and the DIU generating a *cpu_diu_rack* strobe is 1 cycle (1 cycle to perform the arbitration - see Section 20.14.10).
- The read data is returned on *dram_cpu_data[255:0]* and its validity is indicated by *diu_cpu_rvalid*.
- When the *diu_cpu_rvalid* pulse has been received then if there is a further request *cpu_diu_rreq* should be asserted again. The *diu_cpu_rvalid* pulse with a gap of 1 cycle after rack (1 cycle for the read data to be returned from the DRAM - see Section 20.14.10).

20.9.3 Write Protocol except CPU and CDU

The SoPEC write requestors, except for the CPU and CDU, perform single 256-bit write accesses with the write data being transferred to the DIU in 4 consecutive cycles over dedicated point-to-point 64-bit write data busses. The write address *<unit>_diu_wadr[21:5]* is 256-bit aligned.

The write protocol is:

- *<unit>_diu_wreq* is asserted along with a valid *<unit>_diu_wadr[21:5]*.
- The DIU acknowledges the request with *diu_<unit>_wack*. The request should be deasserted. The minimum number of cycles between *<unit>_diu_wreq* being asserted and the DIU generating an *diu_<unit>_wack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.14.10).
- In the clock cycles following *diu_<unit>_wack* the SoPEC Unit outputs the *<unit>_diu_data[63:0]*, asserting *<unit>_diu_wvalid*. The first *<unit>_diu_wvalid* pulse can occur the clock cycle after *diu_<unit>_wack*. *<unit>_diu_wvalid* remains asserted for the following 3 clock cycles. This allows for reading from an SRAM where new data is available in the clock cycle after the address has changed e.g. the address for the second 64-bits of write data is available the cycle after *diu_<unit>_wack* meaning the second 64-bits of write data is a further cycle later. The overall 256 bits of data is transferred over four cycles in the order : [63:0] -> [127:64] -> [191:128] -> [255:192].
- Note that for SCB writes, each 64-bit quarter-word has an 8-bit byte enable mask associated with it. A different mask is used with each quarter-word. The 4 mask values are transferred along with their associated data, as shown in Figure 92.
- If four consecutive *<unit>_diu_wvalid* pulses are not provided by the requester, then the arbitration logic will disregard the write and re-allocate the slot under the unused read round-robin scheme.

Once all the write data has been output then if there is a further request *<unit>_diu_wreq* should be asserted again.

20.9.4 CPU Write Protocol

The CPU performs single 128-bit writes to the DIU on a dedicated write bus, *cpu_diu_wdata[127:0]*. There is an accompanying write mask, *cpu_diu_wmask[15:0]*, consisting of 16 byte enables and the CPU also supplies a 128-bit aligned write address on *cpu_diu_wadr[21:4]*. Note that writes are *posted* by the CPU to the DIU and stored in a 1-deep buffer. When the DAU subsequently arbitrates in favour of the CPU, the contents of the buffer are written to DRAM.

The CPU write protocol, illustrated in Figure 93., is as follows :-

- The DIU signals to the CPU via *diu_cpu_write_rdy* that its write buffer is empty and that the CPU may post a write whenever it wishes.
- The CPU asserts *cpu_diu_wdatavalid* to enable a write into the buffer and to confirm the validity of the write address, data and mask.
- The DIU de-asserts *diu_cpu_write_rdy* in the following cycle to indicate that its buffer is full and that the posted write is pending execution.
- When the CPU is next awarded a DRAM access by the DAU, the buffer's contents are written to memory. The DIU re-asserts *diu_cpu_write_rdy* once the write data has been captured by DRAM, namely in the "MSN1" DCU state.
- The CPU can then, if it wishes, asynchronously use the new value of *.diu_cpu_write_rdy* to enable a new posted write in the same "MSN1" cycle.

20.9.5 CDU Write Protocol

The CDU performs four 64-bit word writes to 4 contiguous 256-bit DRAM addresses with the first address specified by *cdu_diu_wadr[21:3]*. The write address *cdu_diu_wadr[21:5]* is 256-bit aligned with bits *cdu_diu_wadr[4:3]* allowing the 64-bit word to be selected.

The write protocol is:

- *cdu_diu_wdata* is asserted along with a valid *cdu_diu_wadr[21:3]*.
- The DIU acknowledges the request with *diu_cdu_wack*. The request should be deasserted. The minimum number of cycles between *cdu_diu_wreq* being asserted and the DIU generating an *diu_cdu_wack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.14.10).
- In the clock cycles following *diu_cdu_wack* the CDU outputs the *cdu_diu_data[63:0]*, together with asserted *cdu_diu_wvalid*. The first *cdu_diu_wvalid* pulse can occur the clock cycle after *diu_cdu_wack*. *cdu_diu_wvalid* remains asserted for the following 3 clock cycles. This allows for reading from an SRAM where new data is available in the clock cycle after the address has changed e.g. the address for the second 64-bits of write data is available the cycle after *diu_cdu_wack* meaning the second 64-bits of write data is a further cycle later. Data is transferred over the 4-cycle window in an order, such that each successive 64 bits will be written to a monotonically increasing (by 1 location) 256-bit DRAM word.
- If four consecutive *cdu_diu_wvalid* pulses are not provided with the data, then the arbitration logic will disregard the write and re-allocate the slot under the unused read round-robin scheme.

- Once all the write data has been output then if there is a further request *cdu_diu_wreq* should be asserted again.

20.10 DIU ARBITRATION MECHANISM

The DIU will arbitrate access to the embedded DRAM. The arbitration scheme is outlined in the next sections.

20.10.1 Timeslot based arbitration scheme

Table summarised the bandwidth requirements of the SoPEC requestors to DRAM. If we allocate the DIU requestors in terms of peak bandwidth then we require 35.25 bits/cycle (at SF =6) and 40.75 bits/ cycle (at SF = 4) for all the requestors except the CPU.

A timeslot scheme is defined with 64 main timeslots. The number of used main timeslots is programmable between 1 and 64.

Since DRAM read requestors, except for the CPU, are connected to the DIU via a 64-bit data bus each 256-bit DRAM access requires 4 *pclk* cycles to transfer the read data over the shared read bus. The timeslot rotation period for 64 timeslots each of 4 *pclk* cycles is 256 *pclk* cycles or 1.6 μs, assuming *pclk* is 160 MHz. Each timeslot represents a 256-bit access every 256 *pclk* cycles or 1 bit/cycle. This is the granularity of the majority of DIU requestors bandwidth requirements in Table .

The SoPEC DIU requesters can be represented using 4 bits (Table n page288 on page 298). Using 64 timeslots means that to allocate each timeslot to a requester, a total of 64 x 5-bit configuration registers are required for the 64 main timeslots.

Timeslot based arbitration works by having a pointer point to the current timeslot. When re-arbitration is signaled the arbitration winner is the current timeslot and the pointer advances to the next timeslot. Each timeslot denotes a single access. The duration of the timeslot depends on the access.

Note that advancement through the timeslot rotation is dependent on an enable bit, *RotationSync*, being set. The consequences of clearing and setting this bit are described in section 20.14.12.2.1 on page 325.

If the SoPEC Unit assigned to the current timeslot is not requesting then the unused timeslot arbitration mechanism outlined in Section 20.10.6 is used to select the arbitration winner.

Note that there is *always* an arbitration winner for every slot. This is because the unused read re-allocation scheme includes refresh in its round-robin protocol. If all other blocks are not requesting, an early refresh will act as fall-back for the slot.

20.10.2 Separate read and write arbitration windows

For write accesses, except the CPU, 256-bits of write data are transferred from the SoPEC DIU write requestors over 64-bit write busses in 4 clock cycles. This write data transfer latency means that writes accesses, except for CPU writes and also the CDU, must be arbitrated 4 cycles in advance. (The CDU is an exception because CDU writes can start once the first 64-bits of write data have been transferred since each 64-bits is associated with a write to a different 256-bit word).

Since write arbitration must occur 4 cycles in advance, and the minimum duration of a timeslot duration is 3 cycles, the arbitration rules must be modified to initiate write accesses in advance. Accordingly, there is a write timeslot lookahead pointer shown in Figure 96 two timeslots in advance of the current timeslot pointer.

The following examples illustrate separate read and write timeslot arbitration with no adjacent write timeslots. (Recall rule on adjacent write timeslots introduced in Section 20.7.2.3 on page 267.)

In Figure 97 writes are arbitrated two timeslots in advance. Reads are arbitrated in the same timeslot as they are issued. Writes can be arbitrated in the same timeslot as a read. During arbitration the command address of the arbitrated SoPEC Unit is captured.

Other examples are shown in Figure 98 and Figure 99. The actual timeslot order is always the same as the programmed timeslot order i.e. out of order accesses do not occur and data coherency is never an issue.

Each write must always incur a latency of two timeslots.

Startup latency may vary depending on the position of the first write timeslot. This startup latency is not important.

Table 112 shows the 4 scenarios depending on whether the current timeslot and write timeslot lookahead pointers point to read or write accesses.

Table 112. Arbitration with separate windows for read and write accesses

current timeslot pointer	write timeslot lookahead pointer	actions
Read	write	Initiate DRAM read, Initiate write arbitration
Read1	read2	Initiate DRAM read1.
Write1	write2	Initiate write2 arbitration. Execute DRAM write1.
Write	read	Execute DRAM write.

If the current timeslot pointer points to a read access then this will be initiated immediately.

If the write timeslot lookahead pointer points to a write access then this access is arbitrated immediately, or immediately after the read access associated with the current timeslot pointer is initiated.

When a write access is arbitrated the DIU will capture the write address. When the current timeslot pointer advances to the write timeslot then the actual DRAM access will be initiated. Writes will therefore be arbitrated 2 timeslots in advance of the DRAM write occurring.

At initialisation, the write lookahead pointer points to the first timeslot. The current timeslot pointer is invalid until the write lookahead pointer advances to the third timeslot when the current timeslot pointer will point to the first timeslot. Then both pointers advance in tandem.

CPU write accesses are excepted from the lookahead mechanism.

If the selected SoPEC Unit is not requesting then there will be separate read and write selection for unused timeslots. This is described in Section 20.10.6.

20.10.3 Arbitration of CPU accesses

What distinguishes the CPU from other SoPEC requestors, is that the CPU requires minimum latency DRAM access i.e. preferably the CPU should get the next available timeslot whenever it requests.

The minimum CPU read access latency is estimated in Table 113. This is the time between the CPU making a request to the DIU and receiving the read data back from the DIU.

Table 113. Estimated CPU read access latency ignoring caching

CPU read access latency	Duration
CPU cache miss	1 cycle
CPU MMU logic issues request and DIU arbitration completes	1 cycle
Transfer the read address to the DRAM	1 cycle
DRAM read latency	1 cycle
Register the read data in CPU bridge	1 cycle
Register the read data in CPU	1 cycle
CPU cache miss	1 cycle
CPU MMU logic issues request and DIU arbitration completes	1 cycle
TOTAL gap between requests	6 cycles

If the CPU, as is likely, requests DRAM access again immediately after receiving data from the DIU then the CPU could access every second timeslot if the access latency is 6 cycles. This assumes that interleaving is employed so that timeslots last 3 cycles. If the CPU access latency were 7 cycles, then the CPU would only be able to access every third timeslot.

If a cache hit occurs the CPU does not require DRAM access. For its next DIU access it will have to wait for its next assigned DIU slot. Cache hits therefore will reduce the number of DRAM accesses but not speed up any of those accesses.

To avoid the CPU having to wait for its next timeslot it is desirable to have a mechanism for ensuring that the CPU always gets the next available timeslot without incurring any latency on the non-CPU timeslots.

This can be done by defining each timeslot as consisting of a CPU access preceding a non-CPU access. Each timeslot will last 6 cycles i.e. a CPU access of 3 cycles and a non-CPU access of 3 cycles. This is exactly the interleaving behaviour outlined in Section 20.7.2.2. If the CPU does not

require an access, the timeslot will take 3 or 4 and the timeslot rotation will go faster. A summary is given in Table 114.

Table 114. Timeslot access times.

Access	Duration	Explanation
CPU access + non-CPU access	3 + 3 = 6 cycles	Interleaved access
non-CPU access	4 cycles	Access and preceding access both to shared read bus
non-CPU access	3 cycles	Access and preceding access not both to shared read bus
CDU write access	3+2+2+2 = 9 cycles	Page mode select signal is clocked at 160 MHz

CDU write accesses require 9 cycles. CDU write accesses preceded by a CPU access require 12 cycles. CDU timeslots therefore take longer than all other DIU requestors timeslots.

With a 256 cycle rotation there can be 42 accesses of 6 cycles.

For low scale factor applications, it is desirable to have more timeslots available in the same 256 cycle rotation. So two counters of 4-bits each are defined allowing the CPU to get a maximum of (*CPUPreAccessTimeslots* + 1) pre-accesses for every (*CPUTotalTimeslots* + 1) main slots. A timeslot counter starts at *CPUTotalTimeslots* and decrements every timeslot, while another counter starts at *CPUPreAccessTimeslots* and decrements every timeslot in which the CPU uses its access. When the CPU pre-access counter goes to zero before *CPUTotalTimeslots,* no further CPU accesses are allowed. When the *CPUTotalTimeslots* counter reaches zero both counters are reset to their respective initial values.

The CPU is not included in the list of SoPEC DIU requesters, Table , for the main timeslot allocations. The CPU cannot therefore be allocated main timeslots. It relies on pre-accesses in advance of such slots as the sole method for DRAM transfers.

CPU access to DRAM can never be fully disabled, since to do so would render SoPEC inoperable. Therefore the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* register values are interpreted as follows : In each succeeding window of (*CPUTotalTimeslots*+ 1) slots, the maximum quota of CPU pre-accesses allowed is (*CPUPreAccessTimeslots* + 1). The "+ 1" implementations mean that the CPU quota cannot be made zero.

The various modes of operation are summarised in Table 115 with a nominal rotation period of 256 cycles.

Table 115. CPU timeslot allocation modes with nominal rotation period of 256 cycles

Access Type	Nominal Timeslot duration	Number of timeslots	Notes
CPU Pre-access i.e.	6 cycles	42 timeslots	Each access is CPU + non-CPU. If CPU does not use a timeslot then

CPUPreAccessTimeslots = CPUTotalTimeslots			rotation is faster.
Fractional CPU Pre-access i.e. *CPUPreAccessTimeslots < CPUTotalTimeslots*	4 or 6 cycles	42-64 timeslots	Each CPU + non-CPU access requires a 6 cycle timeslot.
			Individual non-CPU timeslots take 4 cycles if current access and preceding access are both to shared read bus.
			Individual non-CPU timeslots take 3 cycles if current access and preceding access are not both to shared read bus.

20.10.4 CDU accesses

As indicated in Section 20.10.3, CDU write accesses require 9 cycles. CDU write accesses preceded by a CPU access require 12 cycles. CDU timeslots therefore take longer than all other DIU requestors timeslots. This means that when a write timeslot is unused it cannot be re-allocated to a CDU write as CDU accesses take 9 cycles. The write accesses which the CDU write could otherwise replace require only 3 or 4 cycles.

Unused CDU write accesses can be replaced by any other write access according to 20.10.6.1 Unused write timeslots allocation on page 277.

20.10.5 Refresh controller

Refresh is not included in the list of SoPEC DIU requesters, Table , for the main timeslot allocations. Timeslots cannot therefore be allocated to refresh.

The DRAM must be refreshed every 3.2 ms. Refresh occurs row at a time over 5120 rows of 2 parallel 10 Mbit instances. A refresh operation must therefore occur every 100 cycles. The *refresh_period* register has a default value of 99. Each refresh takes 3 cycles.

A refresh counter will count down the number of cycles between each refresh. When the down-counter reaches 0, the refresh controller will issue a refresh request and the down-counter is reloaded with the value in *refresh_period* and the count-down resumes immediately. Allocation of main slots must take into account that a refresh is required at *least* once every 100 cycles.

Refresh is included in the unused read and write timeslot allocation. If unused timeslot allocation results in refresh occurring early by *N* cycles, then the refresh counter will have counted down to *N*. In this case, the refresh counter is reset to *refresh_period* and the count-down recommences.

Refresh can be preceded by a CPU access in the same way as any other access. This is controlled by the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* configuration registers. Refresh will therefore not affect CPU performance. A sequence of accesses including refresh might therefore be CPU, refresh, CPU, actual timeslot.

20.10.6 Allocating unused timeslots

Unused slots are re-allocated separately depending on whether the unused access was a read access or a write access. This is best-effort traffic. Only unused non-CPU accesses are re-allocated.

20.10.6.1 Unused write timeslots allocation

Unused write timeslots are re-allocated according to a *fixed* priority order shown in Table 116 .

Table 116. Unused write timeslot priority order

Name	Priority Order
SCB(W)	1
SFU(W)	2
DWU	3
Unused read timeslot allocation	4

CDU write accesses cannot be included in the unused timeslot allocation for write as CDU accesses take 9 cycles. The write accesses which the CDU write could otherwise replace require only 3 or 4 cycles.

Unused write timeslot allocation occurs two timeslots in advance as noted in Section 20.10.2. If the units at priorities 1-3 are not requesting then the timeslot is re-allocated according to the unused read timeslot allocation scheme described in Section 20.10.6.2. However, the unused read timeslot allocation will occur when the current timeslot pointer of Figure 96 reaches the timeslot i.e. it will not occur in advance.

20.10.6.2 Unused read timeslots allocation

Unused read timeslots are re-allocated according to a two level round-robin scheme. The SoPEC Units included in read timeslot re-allocation is shown in Table 117.

Table 117. Unused read timeslot allocation

Name
SCB(R)
CDU(R)
CFU
LBD
SFU(R)

TE(TD)
TE(TFS)
HCU
DNC
LLU
PCU
CPU / Refresh

Each SoPEC requestor has an associated bit, *ReadRoundRobinLevel*, which indicates whether it is in level 1 or level 2 round-robin.

Table 118. Read round-robin level selection

Level	Action
ReadRoundRobinLevel = 0	Level 1
ReadRoundRobinLevel = 1	Level 2

A pointer points to the most recent winner on each of the round-robin levels. Re-allocation is carried out by traversing level 1 requesters, starting with the one immediately succeeding the last level 1 winner. If a requesting unit is found, then it wins arbitration and the level 1 pointer is shifted to its position. If no level 1 unit wants the slot, then level 2 is similarly examined and its pointer adjusted.

Since refresh occupies a (shared) position on one of the two levels and continually requests access, there will always be some round-robin winner for any unused slot.

20.10.6.2.1 Shared CPU / Refresh Round-Robin Position

Note that the CPU can conditionally be allowed to take part in the unused read round-robin scheme. Its participation is controlled via the configuration bit *EnableCPURoundRobin*. When this bit is set, the CPU and refresh *share* a joint position in the round-robin order, shown in Table . When cleared, the position is occupied by refresh alone.

If the shared position is next in line to be awarded an unused non-CPU read/write slot, then the CPU will have first option on the slot. Only if the CPU doesn't want the access, will it be granted to refresh. If the CPU is excluded from the round robin, then any awards to the position benefit refresh.

20.11 GUIDELINES FOR PROGRAMMING THE DIU

Some guidelines for programming the DIU arbitration scheme are given in this section together with an example.

20.11.1 Circuit Latency

Circuit latency is a fixed service delay which is incurred, as and from the acceptance by the DIU arbitration logic of a block's pending read/write request. It is due to the processing time of the request, readying the data, plus the DRAM access time. Latencies differ for read and write requests. See Tables 79 and 80 for respective breakdowns.

If a requesting block is currently stalled, then the *longest* time it will have to wait between issuing a new request for data and actually receiving it would be its timeslot period, plus the circuit latency overhead, along with any intervening non-standard slot durations, such as refresh and CDU(W). In any case, a stalled block will always incur this latency as an additional overhead, when coming out of a stall.

In the case where a block starts up or unstalls, it will start processing newly-received data at a time beyond its serviced timeslot equivalent to the circuit latency. If the block's timeslots are evenly spaced apart in time to match its processing rate, (in the hope of minimising stalls,) then the earliest that the block could restall, if not re-serviced by the DIU, would be the same latency delay beyond its next timeslot occurrence. Put another way, the latency incurred at start-up pushes the potential DIU-induced stall point out by the same fixed delta beyond each successive timeslot allocated to the block. This assumes that a block re-requests access well in advance of its upcoming timeslots. Thus, for a given stall-free run of operation, the circuit latency overhead is only incurred inititially when unstalling.

While a block can be stalled as a result of how quickly the DIU services its DRAM requests, it is also prone to stalls caused by its upstream or downstream neighbours being able to supply or consume data which is transferred between the blocks directly, (as opposed to via the DIU). Such neighbour-induced stalls, often occurring at events like end of line, will have the effect that a block's DIU read buffer will tend to fill, as the block stops processing read data. Its DIU write buffer will also tend to fill, unable to despatch to DRAM until the downstream block frees up shared-access DRAM locations. This scenario is beneficial, in that when a block unstalls as a result of its neighbour releasing it, then that block's read/write DIU buffers will have a fill state less likely to stall it a second time, as a result of DIU service delays.

A block's slots should be scheduled with a *service guarantee* in mind. This is dictated by the block's processing rate and hence, required access to the DRAM. The rate is expressed in terms of bits per cycle across a processing window, which is typically (though not always) 256 cycles. Slots should be evenly interspersed in this window (or "rotation") so that the DIU can fulfill the block's service needs.

The following ground rules apply in calculating the distribution of slots for a given non-CPU block:-

- The block can, at maximum, suffer a stall *once* in the rotation, (i.e. unstall and restall) and hence incur the circuit latency described above.

This rule is, by definition, always fulfilled by those blocks which have a service requirement of only 1 bit/cycle (equivalent to 1 slot/rotation) or fewer. It can be shown that the rule is also satisfied by those blocks requiring more than 1 bit/cycle. See Section 20.12.1 Slot Distributions and Stall Calculations for Individual Blocks, on page 285.

- Within the rotation, certain slots will be unavailable, due to their being used for refresh. (See Section 20.11.2 **Refresh latencies**)
- In programming the rotation, account must be taken of the fact that any CDU(W) accesses will consume an extra 6 cycles/access, over and above the norm, in CPU pre-access mode, or 5 cycles/access without pre-access.
- The total delay overhead due to latency, refreshes and CDU(W) can be factored into the service guarantee for *all* blocks in the rotation by deleting *once*, (i.e. reducing the rotation window,) that number of slots which equates to the cumulative duration of these various anomalies.
- The use of lower scale factors will imply a more frequent demand for slots by non-CPU blocks. The percentage of slots in the overall rotation which can therefore be designated as CPU pre-access ones should be calculated last, based on what can be accommodated in the light of the non-CPU slot need.

Read latency is summarised below in Table 119 .

Table 119. Read latency

Non-CPU read access latency	Duration
non-CPU read requestor internally generates DIU request	1 cycle
register the non- CPU read request	1 cycle
complete the arbitration of the request	1 cycle
transfer the read address to the DRAM	1 cycle
DRAM read latency	1 cycle
register the DRAM read data in DIU	1 cycle
register the 1st 64-bits of read data in requester	1 cycle
register the 2nd 64-bits of read data in requester	1 cycle
register the 3rd 64-bits of read data in requester	1 cycle
register the 4th 64-bits of read data in requester	1 cycle
TOTAL	10 cycles

Write latency is summarised in Table 120.

Table 120. Write latency

Non-CPU write access latency	Duration
non-CPU write requestor internally generates DIU request	1 cycle

register the non-CPU write request	1 cycle
complete the arbitration of the request	1 cycle
transfer the acknowledge to the write requester	1 cycle
transfer the 1st 64 bits of write data to the DIU	1 cycle
transfer the 2nd 64 bits of write data to the DIU	1 cycle
transfer the 3rd 64 bits of write data to the DIU	1 cycle
transfer the 4th 64 bits of write data to the DIU	1 cycle
Write to DRAM with locally registered write data	1 cycle
TOTAL	9 cycles

Timeslots removed to allow for read latency will also cover write latency, since the former is the larger of the two.

20.11.2 Refresh latencies

The number of allocated timeslots for each requester needs to take into account that a refresh must occur every 100 cycles. This can be achieved by deleting timeslots from the rotation since the number of timeslots is made programmable.

Refresh is preceded by a CPU access in the same way as any other access. This is controlled by the *CPUPreAccessTimeslots* and *CPUTotalTimeslots* configuration registers. Refresh will therefore not affect CPU performance.

As an example, in CPU pre-access mode each timeslot will last 6 cycles. If the timeslot rotation has 50 timeslots then the rotation will last 300 cycles. The refresh controller will trigger a refresh every 100 cycles. Up to 47 timeslots can be allocated to the rotation ignoring refresh. Three timeslots deleted from the 50 timeslot rotation will allow for the latency of a refresh every 100 cycles.

20.11.3 Ensuring sufficient DNC and PCU access

PCU command reads from DRAM are exceptional events and should complete in as short a time as possible. Similarly, we must ensure there is sufficient free bandwidth for DNC accesses e.g. when clusters of dead nozzles occur. In Table DNC is allocated 3 times average bandwidth. PCU and DNC can also be allocated to the level 1 round-robin allocation for unused timeslots so that unused timeslot bandwidth is preferentially available to them.

20.11.4 Basing timeslot allocation on peak bandwidths

Since the embedded DRAM provides sufficient bandwidth to use 1:1 compression rates for the CDU and LBD, it is possible to simplify the main timeslot allocation by basing the allocation on peak bandwidths. As combined bi-level and tag bandwidth at 1:1 scaling is only 5 bits/cycle, we will usually only consider the contone scale factor as the variable in determining timeslot allocations.

If slot allocation is based on peak bandwidth requirements then DRAM access will be *guaranteed* to all SoPEC requesters. If we do not allocate slots for peak bandwidth requirements then we can also allow for the peaks *deterministically* by adding some cycles to the print line time.

20.11.5 Adjacent timeslot restrictions

20.11.5.1 Non-CPU write adjacent timeslot restrictions

Non-CPU write requestors should not be assigned adjacent timeslots as described in Section 20.7.2.3. This is because adjacent timeslots assigned to non-CPU requestors would require two sets of 256-bit write buffers and multiplexors to connect two write requestors simultaneously to the DIU. Only one 256-bit write buffer and multiplexor is implemented. Recall from section 20.7.2.3 on page 267 that if adjacent non-CPU writes are attempted, that the second write of any such pair will be disregarded and re-allocated under the unused read scheme. .

20.11.5.2 Same DIU requestor adjacent timeslot restrictions

All DIU requesters have state-machines which request and transfer the read or write data before requesting again. From Figure 90 read requests have a minimum separation of 9 cycles. From Figure 92 write requests have a minimum separation of 7 cycles. Therefore adjacent timeslots should not be assigned to a particular DIU requester because the requester will not be able to make use of all these slots.

In the case that a CPU access precedes a non-CPU access timeslots last 6 cycles so write and read requesters can only make use of every second timeslot. In the case that timeslots are not preceded by CPU accesses timeslots last 4 cycles so the same write requester can use every second timeslot but the same read requestor can use only every third timeslot. Some DIU requestors may introduce additional pipeline delays before they can request again. Therefore timeslots should be separated by more than the minimum to allow a margin.

20.11.6 Line margin

The SFU must output 1 bit/cycle to the HCU. Since *HCUNumDots* may not be a multiple of 256 bits the last 256-bit DRAM word on the line can contain extra zeros. In this case, the SFU may not be able to provide 1 bit/cycle to the HCU. This could lead to a stall by the SFU. This stall could then propagate if the margins being used by the HCU are not sufficient to hide it. The maximum stall can be estimated by the calculation: DRAM service period - X scale factor * dots used from last DRAM read for HCU line.

Similarly, if the line length is not a multiple of 256-bits then e.g. the LLU could read data from DRAM which contains padded zeros. This could lead to a stall. This stall could then propagate if the page margins cannot hide it.

A single addition of 256 cycles to the line time will suffice for all DIU requesters to mask these stalls.

20.12 EXAMPLE OUTLINE DIU PROGRAMMING

Table 121. Timeslot allocation based on peak bandwidth

Block Name	Direction	Peak Bandwidth which must be supplied (bits/cycle)	MainTimeslots allocated
SCB	R		

	W	0.734[7]	1
CDU	R	0.9 (SF = 6), 2 (SF = 4)	1 (SF = 6) 2 (SF = 4)
	W	1.8 (SF = 6), [8] 4 (SF = 4)	2 (SF = 6) 4 (SF = 4)
CFU	R	5.4 (SF = 6), 8 (SF = 4)	6 (SF = 6) 8 (SF = 4)
LBD	R	1	1
SFU	R	2	2
	W	1	1
TE(TD)	R	1.02	1
TE(TFS)	R	0.093	0
HCU	R	0.074	0
DNC	R	2.4	3
DWU	W	6	6
LLU	R	8	8
PCU	R	1	1
TOTAL			33 (SF=6) 38 (SF=4)

Table 121 shows an allocation of main timeslots based on the peak bandwidths of Table . The bandwidth required for each unit is calculated allowing extra cycles for read and write circuit latency for each access requiring a bandwidth of more than 1 bit/cycle. Fractional bandwidth is supplied via unused read slots.

The timeslot rotation is 256 cycles. Timeslots are deleted from the rotation to allow for circuit latencies for accesses of up to 1 bit per cycle i.e. 1 timeslot per rotation.

Example 1: Scale-factor = 6

Program the *MainTimeslot* configuration register (Table) for peak required bandwidths of SoPEC Units according to the scale factor.

Program the read round-robin allocation to share unused read slots. Allocate PCU, DNC, HCU and TFS to level 1 read round-robin.

- Assume scale-factor of 6 and peak bandwidths from Table .
  - Assign all DIU requestors except TE(TFS) and HCU to multiples of 1 timeslot, as indicated in Table , where each timeslot is 1 bit/cycle. This requires 33 timeslots.
  - No timeslots are explicitly allocated for the fractional bandwidth requirements of TE(TFS) and HCU accesses. Instead, these units are serviced via unused read slots.

---

[7] The SCB figure of 0.734 bits/cycle applies to *multi*-SoPEC systems. For *single*-SoPEC systems, the figure is 0.050 bits/cycle.

[8] Bandwidth for CDU(W) is *peak* value. Because of 1.5 buffering in DRAM, peak CDU(W) b/w equals 2 x average CDU(W) b/w. For CDU(R), peak b/w = average CDU(R) b/w.

- Allow 3 timeslots to allow for 3 refreshes in the rotation.
- Therefore, 36 scheduled slots are used in the rotation for main timeslots and refreshes, some or all of which may be able to have a CPU pre-access, provided they fit in the rotation window.
- Each of the 2 CDU(W) accesses requires 9 cycles. Per access, this implies an overhead of 1 slot (12 cycles instead of 6) in pre-access mode, or 1.25 slots (9 cycles instead of 4) for no pre-access. The cumulative overhead of the two accesses is either 2 slots (pre-access) or 3 slots (no pre-access).
- Assuming all blocks require a service guarantee of no more than a single stall across 256 bits, allow 10 cycles for read latency, which also takes care of 9-cycle write latency. This can be accounted for by reserving 2 six-cycle slots (CPU pre-access) or 3 four-cycle slots (no pre-access).
- Assume a 256 cycle timeslot rotation.
- CDU(W) and read latency reduce the number of available cycles in a rotation to: 256 - 2x6 - 2x6 = 232 cycles (CPU pre-access) or 256 - 3x4 - 3x4 = 232 cycles (no pre-access).
- As a result, 232 cycles available for 36 accesses implies each access can take 232 / 36 = 6.44 cycles maximum. So, all accesses can have a pre-access.
- Therefore the CPU achieves a pre-access ratio of 36 / 36 = 100% of slots in the rotation.

Example 2: Scale-factor = 4

Program the *MainTimeslot* configuration register (Table) for peak required bandwidths of SoPEC Units according to the scale factor. Program the read round-robin allocation to share unused read slots. Allocate PCU, DNC, HCU and TFS to level 1 read round-robin.

- Assume scale-factor of 4 and peak bandwidths from Table .
- Assign all DIU requestors except TE(TFS) and HCU multiples of 1 timeslot, as indicated in Table , where each timeslot is 1 bit/cycle. This requires 38 timeslots.
- No timeslots are explicitly allocated for the fractional bandwidth requirements of TE(TFS) and HCU accesses. Instead, these units are serviced via unused read slots.
- Allow 3 timeslots to allow for 3 refreshes in the rotation.
- Therefore, 41 scheduled slots are used in the rotation for main timeslots and refreshes, some or all of which can have a CPU pre-access, provided they fit in the rotation window.
- Each of the 4 CDU(W) accesses requires 9 cycles. Per access, this implies an overhead of 1 slot (12 cycles instead of 6) for pre-access mode, or 1.25 slots (9 cycles instead of 4) for no pre-access. The cumulative overhead of the four accesses is either 4 slots (pre-access) or 5 slots (no pre-access).
- Assuming all blocks require a service guarantee of no more than a single stall across 256 bits, allow 10 cycles for read latency, which also takes care of 9-cycle write latency. This can be accounted for by reserving 2 six-cycle slots (CPU pre-access) or 3 four-cycle slots (no pre-access).

DEMANDE OU BREVET VOLUMINEUX

LA PRÉSENTE PARTIE DE CETTE DEMANDE OU CE BREVET COMPREND PLUS D'UN TOME.

CECI EST LE TOME	1	DE	4
CONTENANT LES PAGES	1	À	284

NOTE : Pour les tomes additionels, veuillez contacter le Bureau canadien des brevets

JUMBO APPLICATIONS/PATENTS

THIS SECTION OF THE APPLICATION/PATENT CONTAINS MORE THAN ONE VOLUME

THIS IS VOLUME	1	OF	4
CONTAINING PAGES	1	TO	284

NOTE: For additional volumes, please contact the Canadian Patent Office

NOM DU FICHIER / FILE NAME :

NOTE POUR LE TOME / VOLUME NOTE:

CLAIMS

1. A method of compensating for an inoperative nozzle in a bi-lithic printhead, the bi-lithic printhead including a plurality of sets of nozzles for printing a corresponding plurality of channels of dot data, the method comprising the steps of:

(a) rendering compressed pages to form a bi-level layer for a given print line intended for the bi-lithic printhead;

(b) expanding the compressed bi-level layer;

(c) compositing the bi-level layer to produce bi-level dots;

(d) determining which combination of one or more available operative nozzles near the inoperative nozzle will minimise perceived error in an image that the dot data forms part of, the determination being performed on the basis of a color model;

(e) mapping the dot data intended for the inoperative nozzle to that combination of one or more operative nozzles from the same set; and,

(f) passing resultant bi-level channel dot data to the bi-lithic printhead.

2. The method according to claim 1, including mapping the dot data intended for the inoperative nozzle into a nozzle that will print a dot on print media close to a position at which the inoperative nozzle would have printed a dot had it been operative.

3. The method according to claim 1, including mapping the dot data intended for the inoperative nozzle into a nozzle that will print a dot on print media immediately adjacent a position at which the inoperative nozzle would have printed a dot had it been operative.

4. The method according to 1, including the substeps of: (i) determining one or more operative nozzles capable of printing a dot on print media close to a position at which the inoperative nozzle would have printed a dot had it been operative; and (ii) mapping the dot data from the inoperative nozzle to an operative nozzle determined in substep (i).

5. The method according to claim 4, wherein, in the event more than one operative nozzle

is determined in substep (i), the dot data is remapped to one of the operative nozzles that will print a dot on print media closest to that which would have been printed by the inoperative nozzle.

6. The method according to claim 5, wherein, during successive firings of the printhead, the dot data is remapped alternately to operative nozzles that will print a dot on print media either side of that which would have been printed by the inoperative nozzle.

7. The method according to claim 5, wherein, during successive firings of the printhead, the dot data is remapped randomly, pseudo-randomly, or arbitrarily to operative nozzles that will print a dot on print media either side of that which would have been printed by the inoperative nozzle.

8. The method according to claim 1, including mapping the dot data into one or more operative nozzles that will print a dot on print media close to a position at which the inoperative nozzle would have printed a dot had it been operative.

9. The method according to claim 1, including mapping the dot data intended for the inoperative nozzle into one or more operative nozzles including at least one nozzle from a different one of the sets.

10. The method according to claim 1, wherein the inoperative nozzle is associated with a black print channel, and wherein the method includes remapping the dot data intended for the inoperative nozzle into a plurality of operative nozzles in other color channels to produce a process black output at or adjacent a location on print media where the inoperative nozzle would have deposited a droplet of a black printing substance in accordance with the dot data.

11. The method according to claim 1, wherein a plurality of dot data intended for a corresponding plurality of inoperative nozzles are mapped to operative nozzles.

12. A printer controller configured to implement the method of claim 1.

13. A printer controller configured to implement the method of claim 1 to a bi-lithic printhead comprising a plurality of the nozzles.

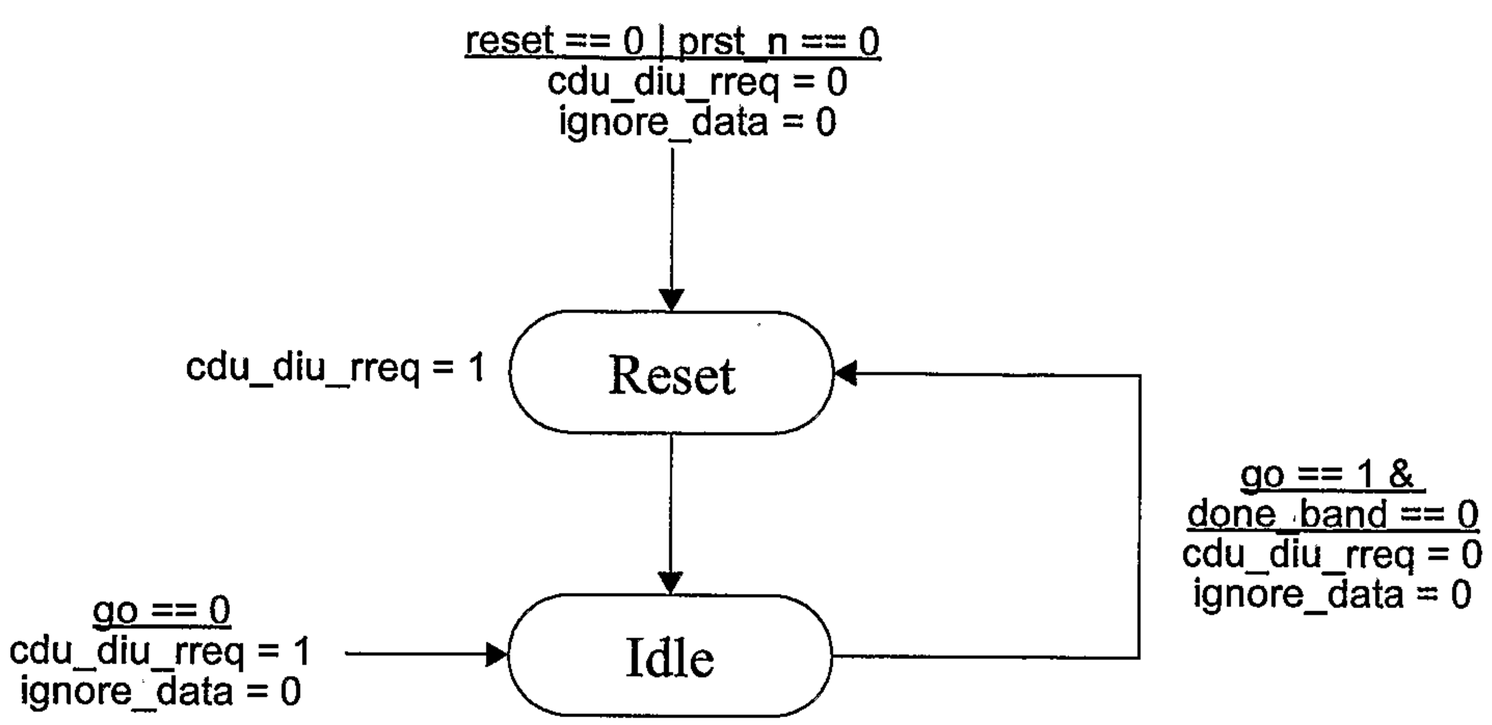

reset == 0 | prst_n == 0
cdu_diu_rreq = 0
ignore_data = 0
cdu_diu_rreq = 1
Reset
go == 1 &
done_band == 0
cdu_diu_rreq = 0
ignore_data = 0
go == 0
cdu_diu_rreq = 1
ignore_data = 0
Idle

FIG. 1

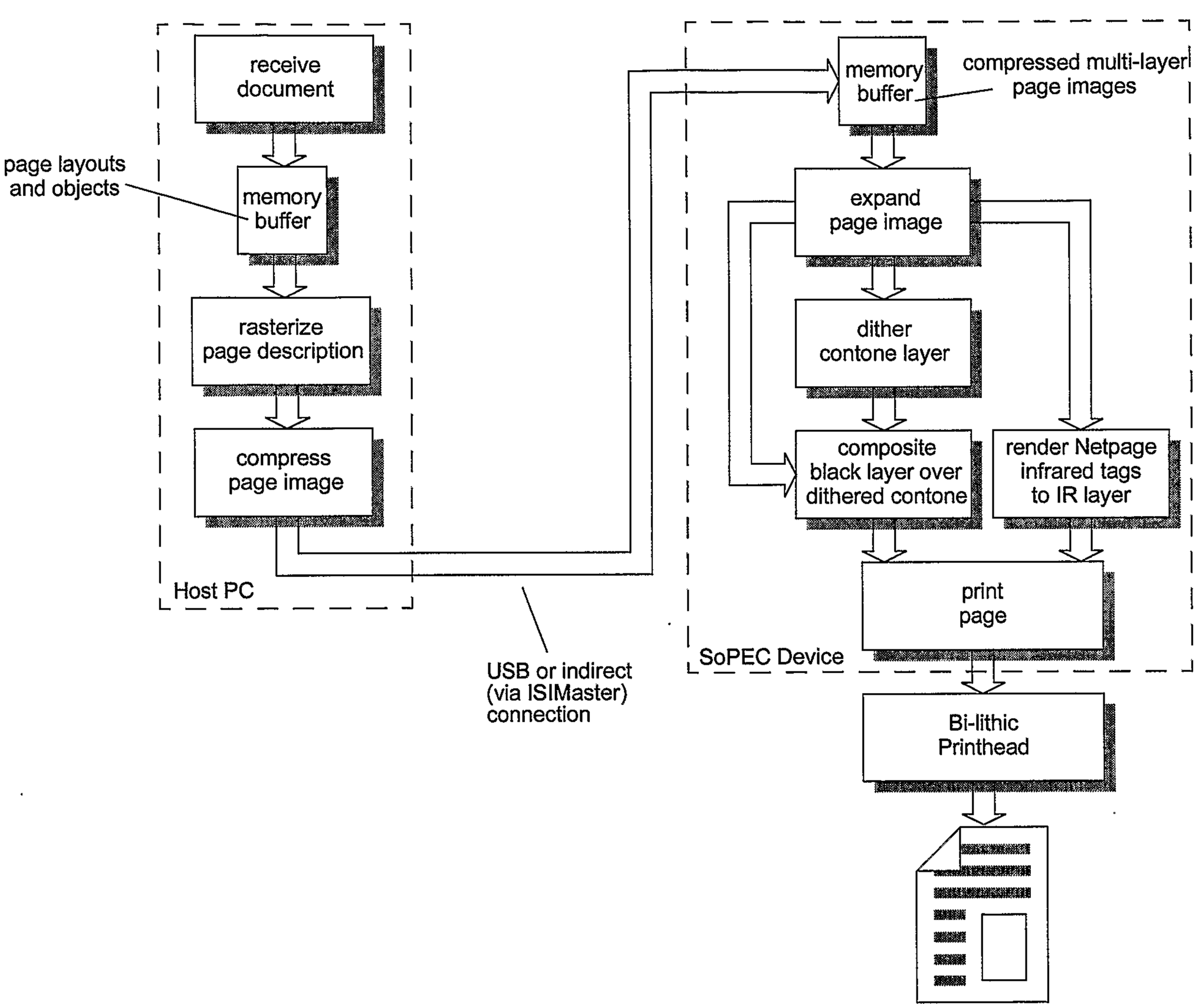

receive document
page layouts and objects
memory buffer
rasterize page description
compress page image
Host PC
USB or indirect (via ISIMaster) connection
memory buffer
compressed multi-layer page images
expand page image
dither contone layer
composite black layer over dithered contone
render Netpage infrared tags to IR layer
print page
SoPEC Device
Bi-lithic Printhead

FIG. 2

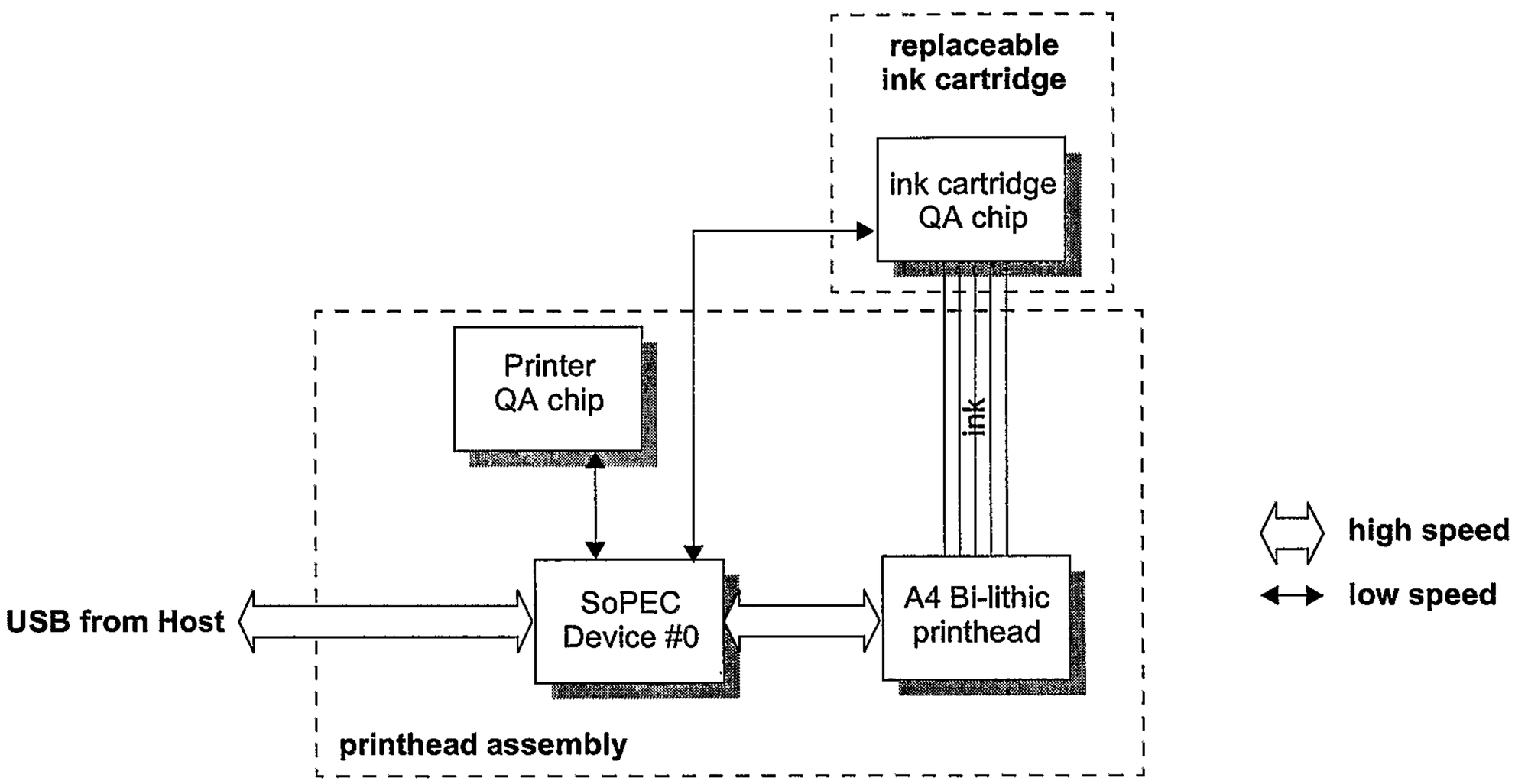

replaceable
ink cartridge
ink cartridge
QA chip
Printer
QA chip
ink
USB from Host
SoPEC
Device #0
A4 Bi-lithic
printhead
printhead assembly
high speed
low speed

FIG. 3

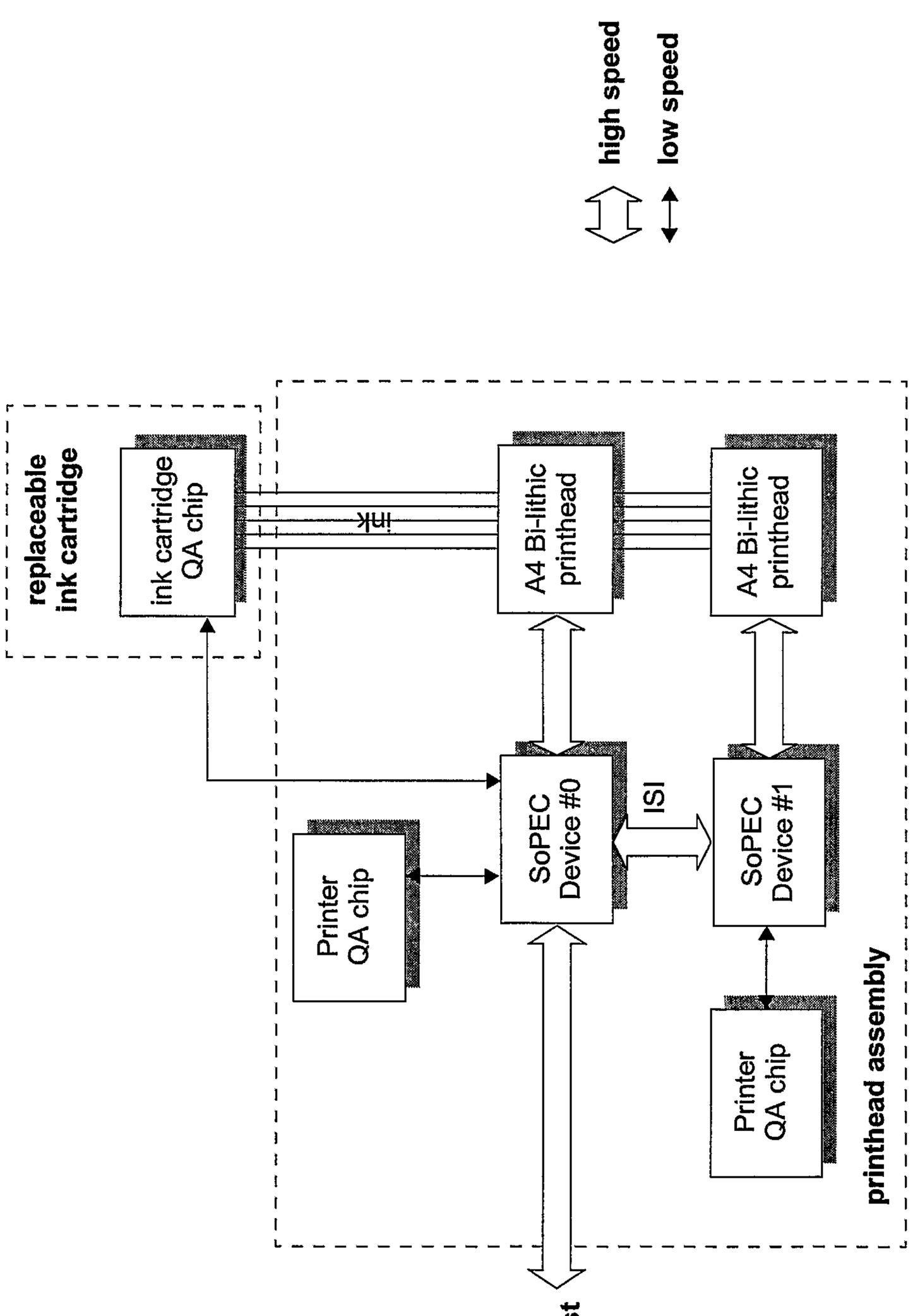

high speed
low speed
replaceable ink cartridge
ink cartridge QA chip
ink
A4 Bi-lithic printhead
A4 Bi-lithic printhead
Printer QA chip
SoPEC Device #0
ISI
SoPEC Device #1
Printer QA chip
printhead assembly
USB from Host

FIG. 4

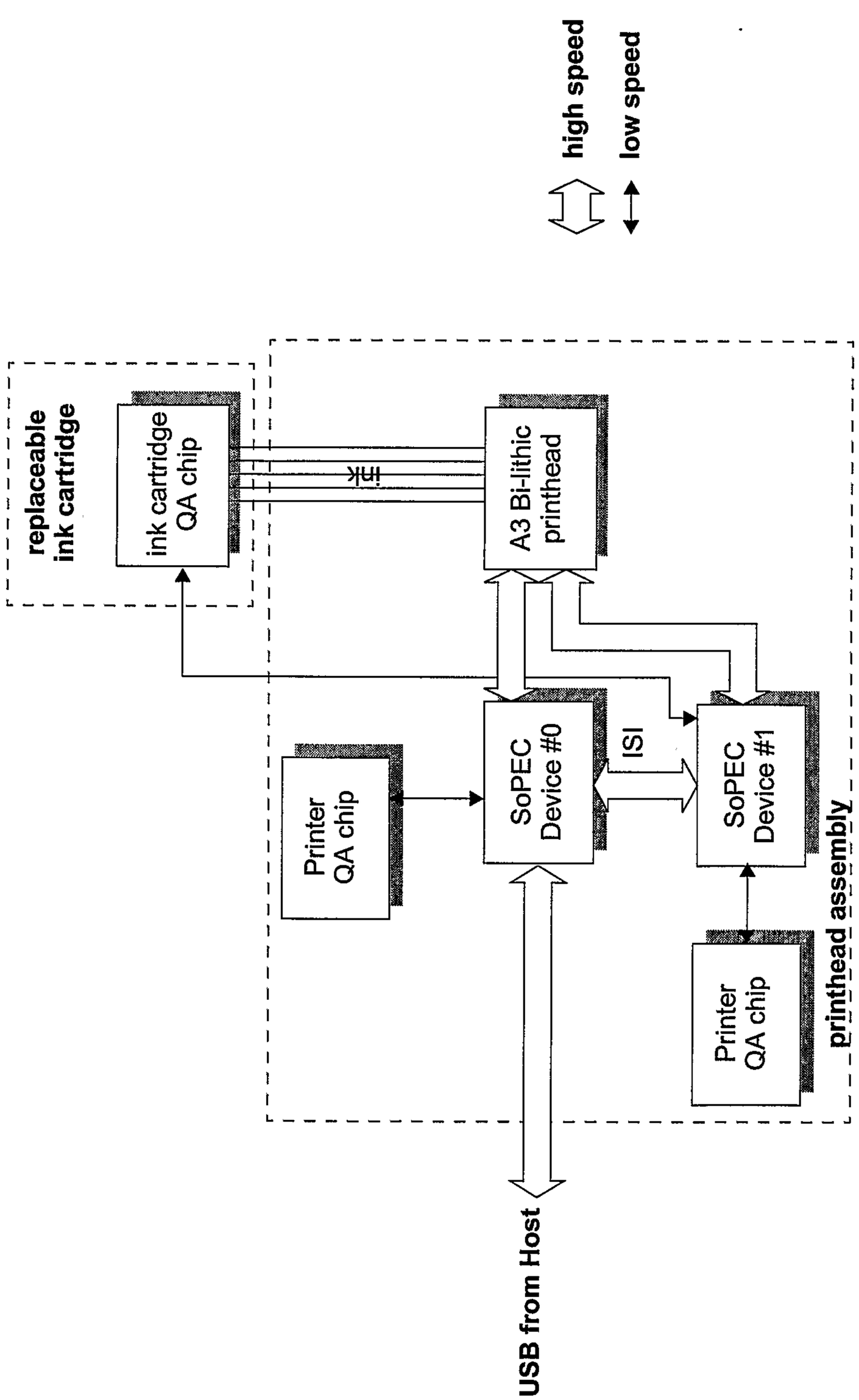

high speed
low speed
replaceable ink cartridge
ink cartridge QA chip
ink
A3 Bi-lithic printhead
Printer QA chip
SoPEC Device #0
ISI
SoPEC Device #1
Printer QA chip
printhead assembly
USB from Host

FIG. 5

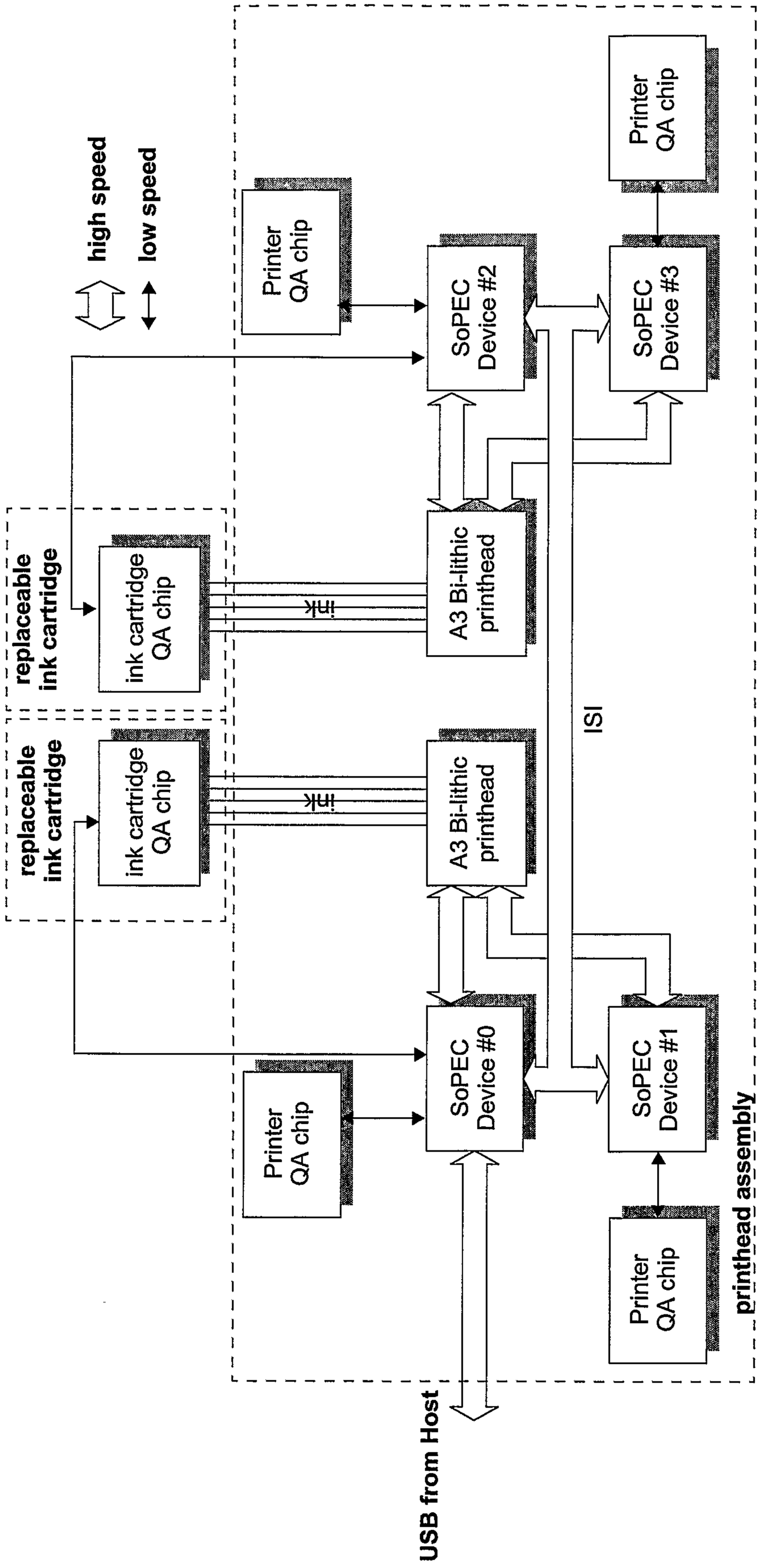

printhead assembly
ISI
USB from Host
high speed
low speed
replaceable ink cartridge
ink cartridge QA chip
A3 Bi-lithic printhead
Printer QA chip
SoPEC Device #0
SoPEC Device #1
SoPEC Device #2
SoPEC Device #3

FIG. 6

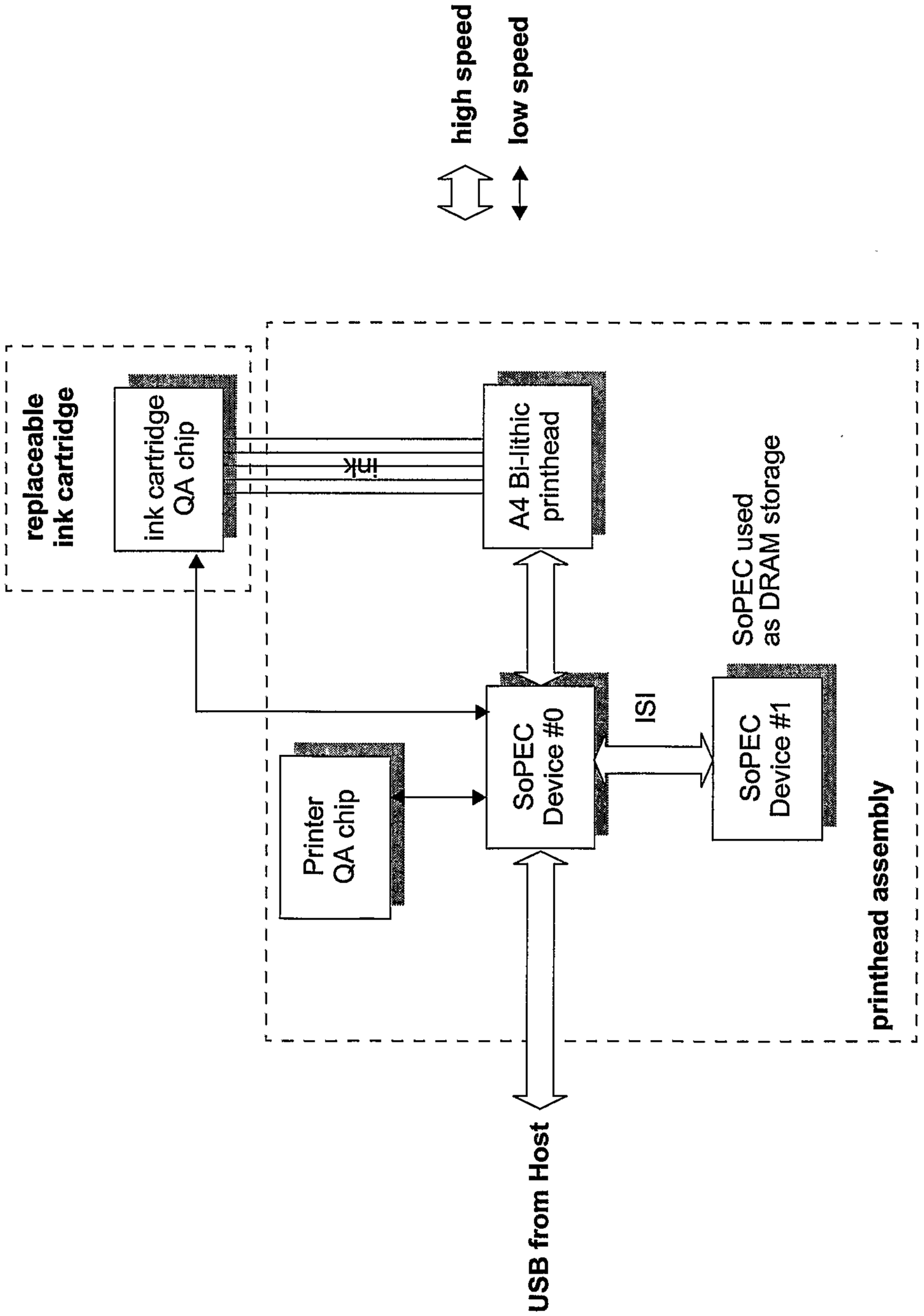

high speed
low speed
replaceable ink cartridge
ink cartridge QA chip
ink
A4 Bi-lithic printhead
SoPEC used as DRAM storage
SoPEC Device #0
ISI
SoPEC Device #1
Printer QA chip
printhead assembly
USB from Host

FIG. 7

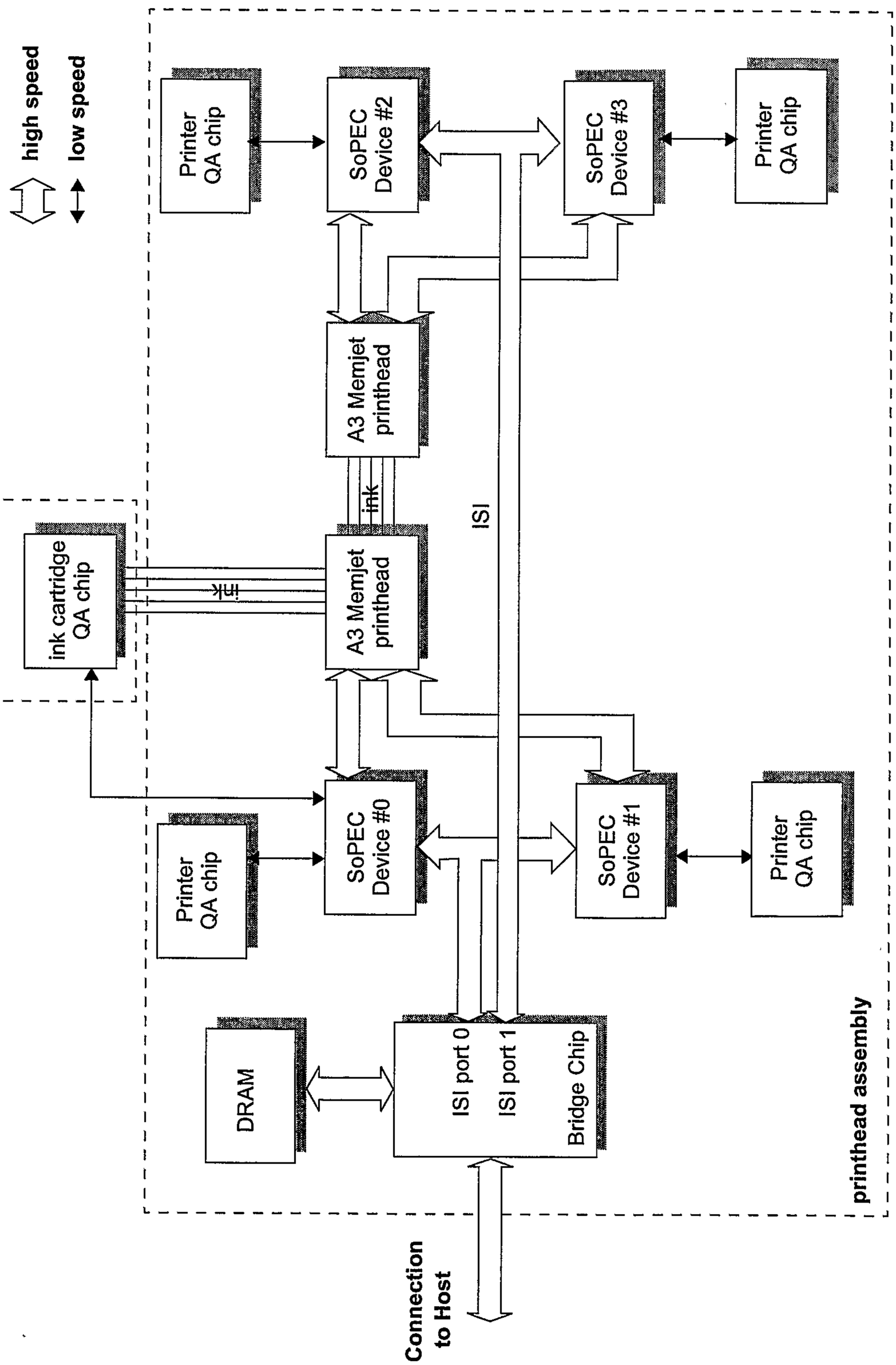

high speed
low speed
Printer QA chip
SoPEC Device #2
SoPEC Device #3
Printer QA chip
A3 Memjet printhead
ink
ISI
replaceable ink cartridge
ink cartridge QA chip
A3 Memjet printhead
Printer QA chip
SoPEC Device #0
SoPEC Device #1
Printer QA chip
DRAM
ISI port 0
ISI port 1
Bridge Chip
printhead assembly
Connection to Host

FIG. 8

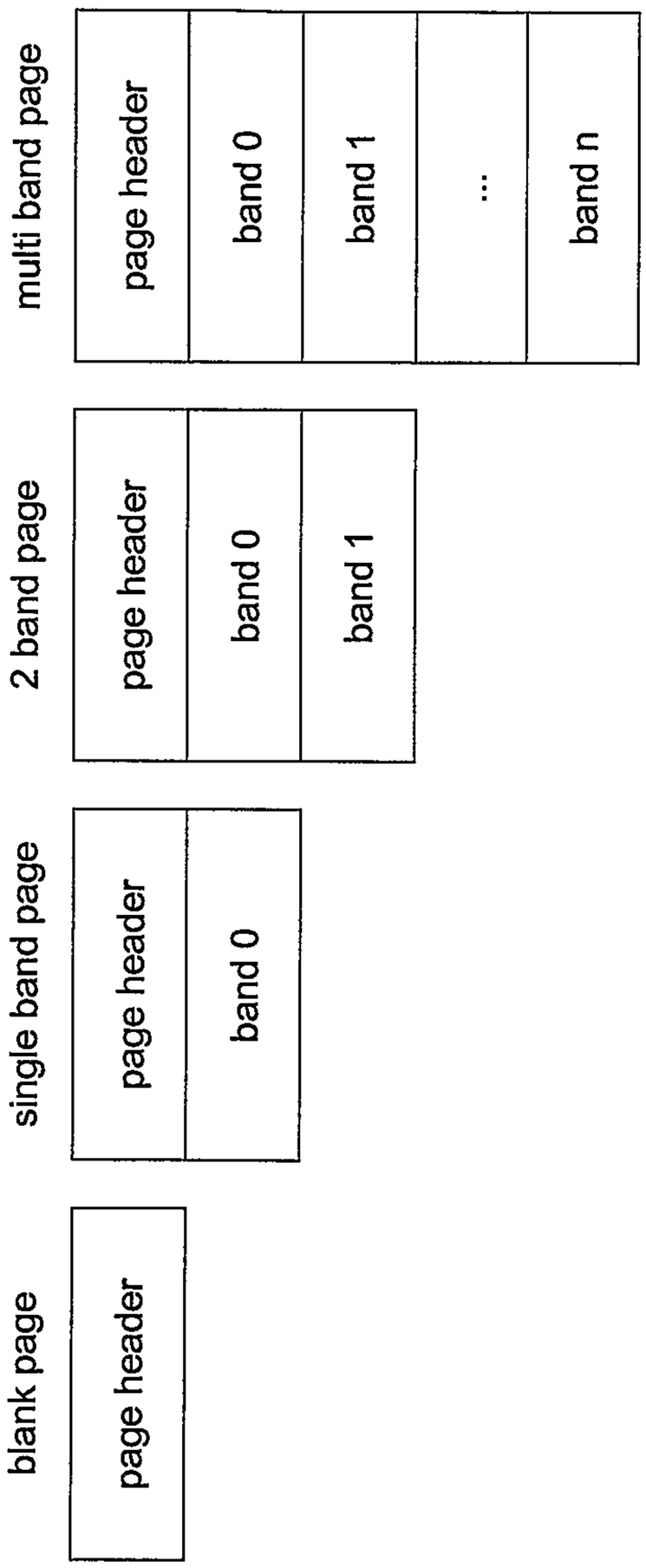

multi band page
page header
band 0
band 1
...
band n
2 band page
page header
band 0
band 1
single band page
page header
band 0
blank page
page header

FIG. 9

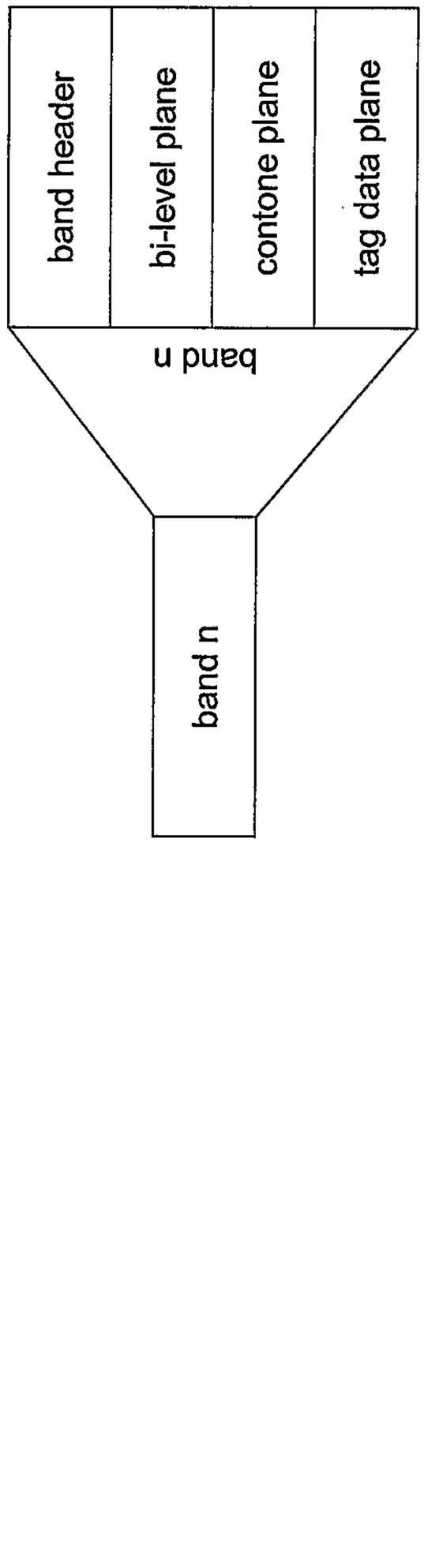

band header
bi-level plane
contone plane
tag data plane
band n
band n

FIG. 10

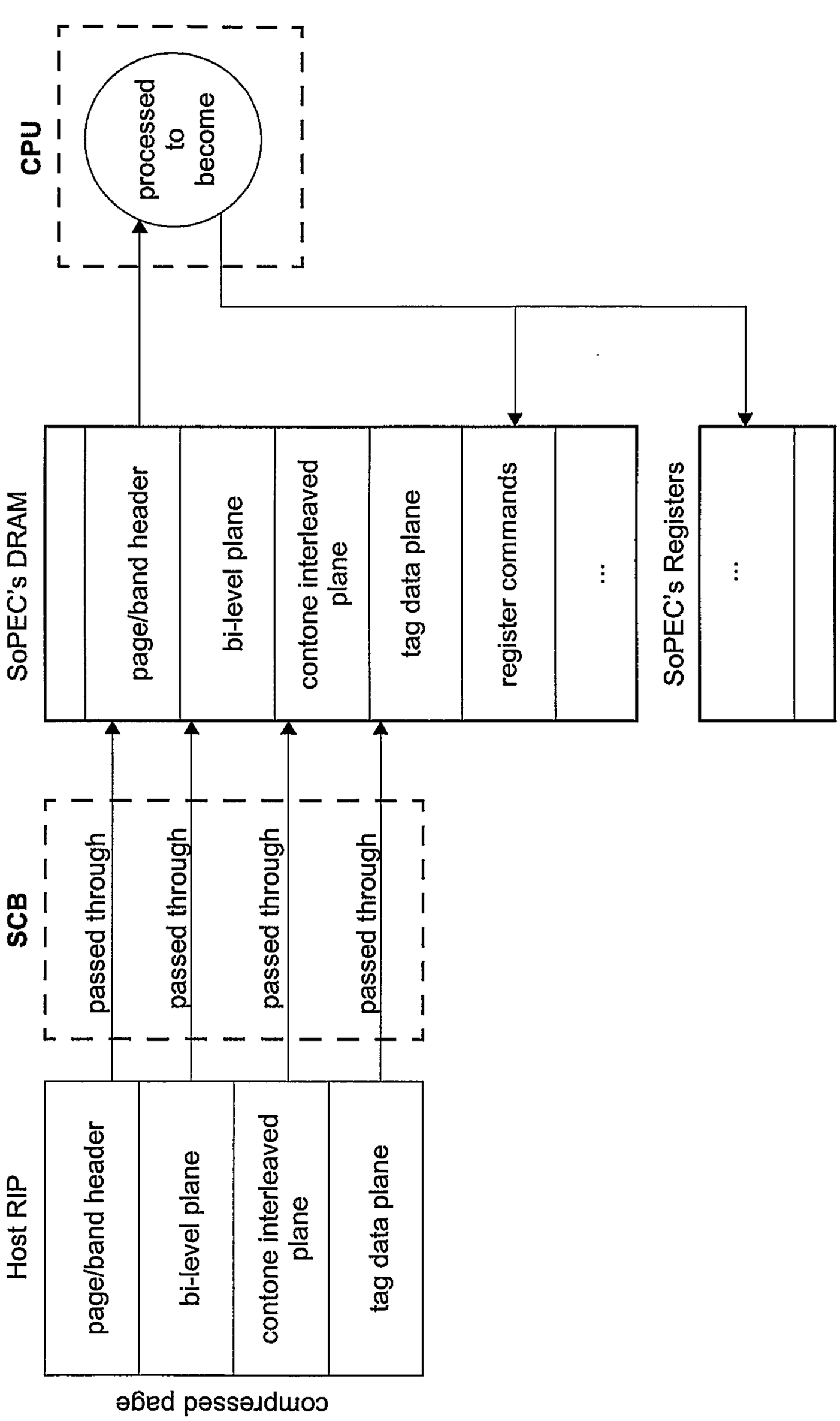

CPU
processed to become
SoPEC's DRAM
page/band header
bi-level plane
contone interleaved plane
tag data plane
register commands
...
SoPEC's Registers
...
SCB
passed through
passed through
passed through
passed through
Host RIP
page/band header
bi-level plane
contone interleaved plane
tag data plane
compressed page

FIG. 11

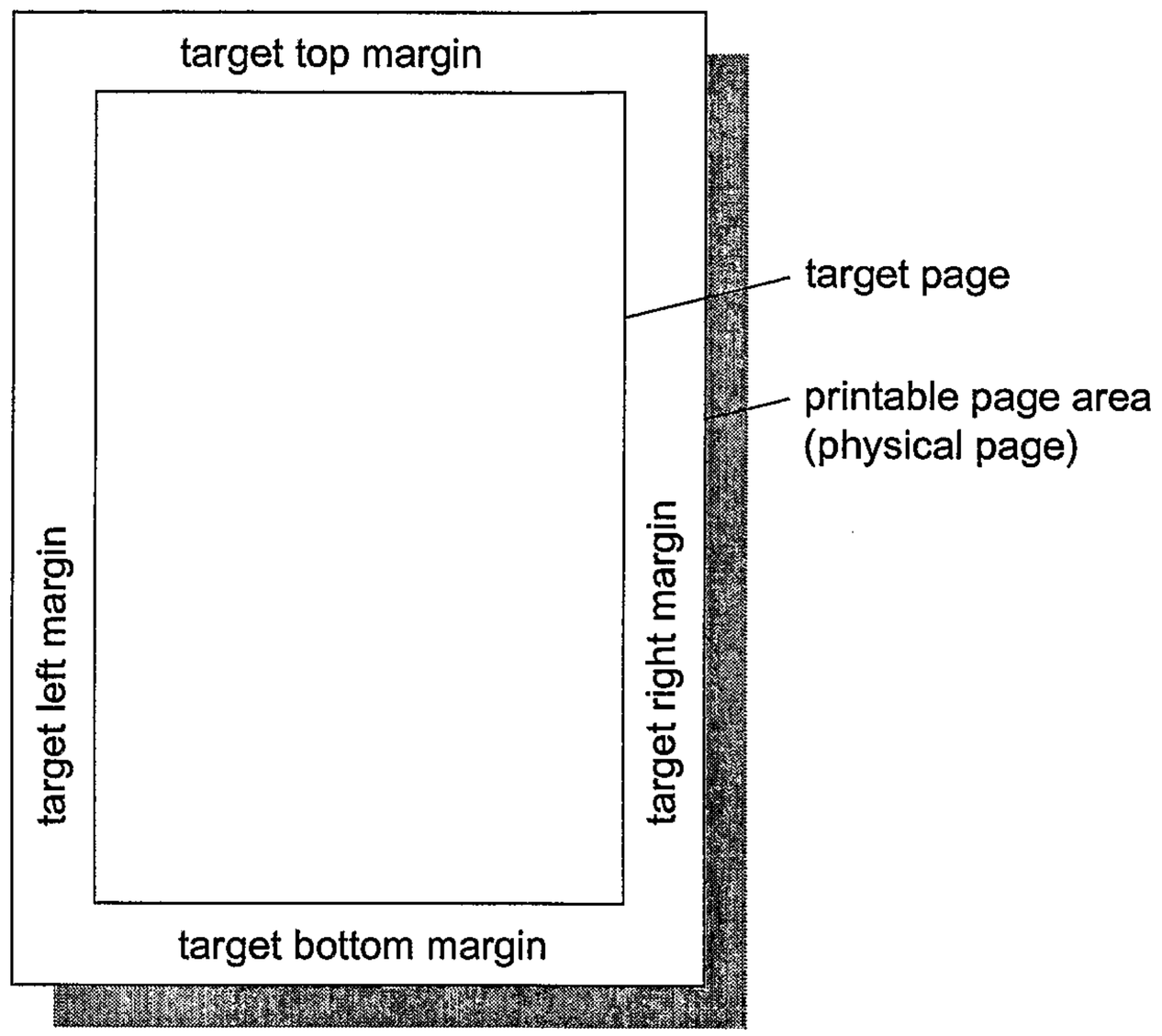

target top margin
target left margin
target right margin
target bottom margin
target page
printable page area
(physical page)

FIG. 12

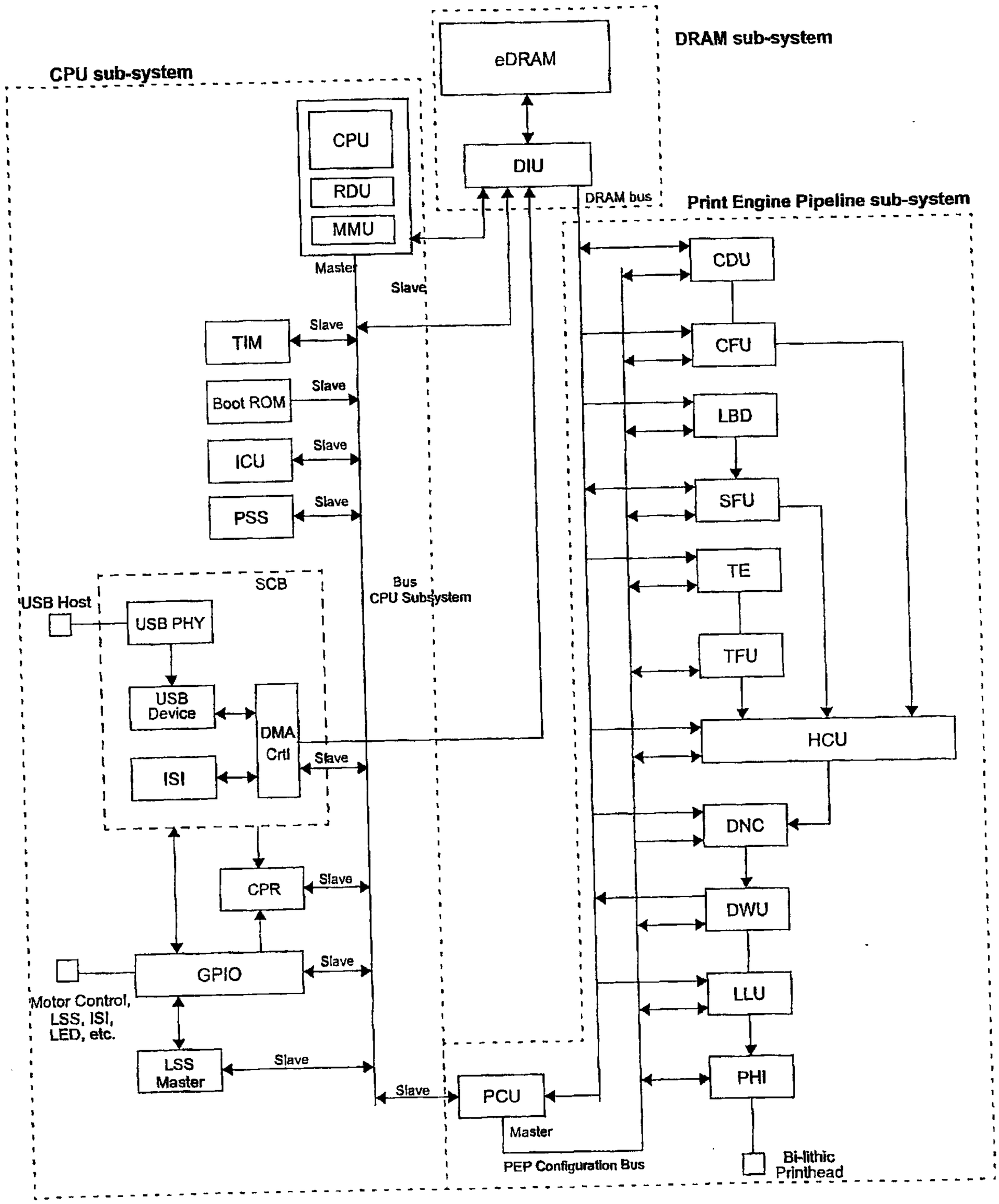

DRAM sub-system
eDRAM
CPU sub-system
CPU
RDU
MMU
DIU
DRAM bus
Print Engine Pipeline sub-system
Master
Slave
CDU
TIM
Slave
CFU
Boot ROM
Slave
LBD
ICU
Slave
SFU
PSS
Slave
TE
SCB
Bus
CPU Subsystem
USB Host
USB PHY
TFU
USB
Device
DMA
Crtl
HCU
ISI
Slave
DNC
CPR
Slave
DWU
GPIO
Slave
LLU
Motor Control,
LSS, ISI,
LED, etc.
LSS
Master
Slave
PHI
Slave
PCU
Master
PEP Configuration Bus
Bi-lithic
Printhead

FIG. 13

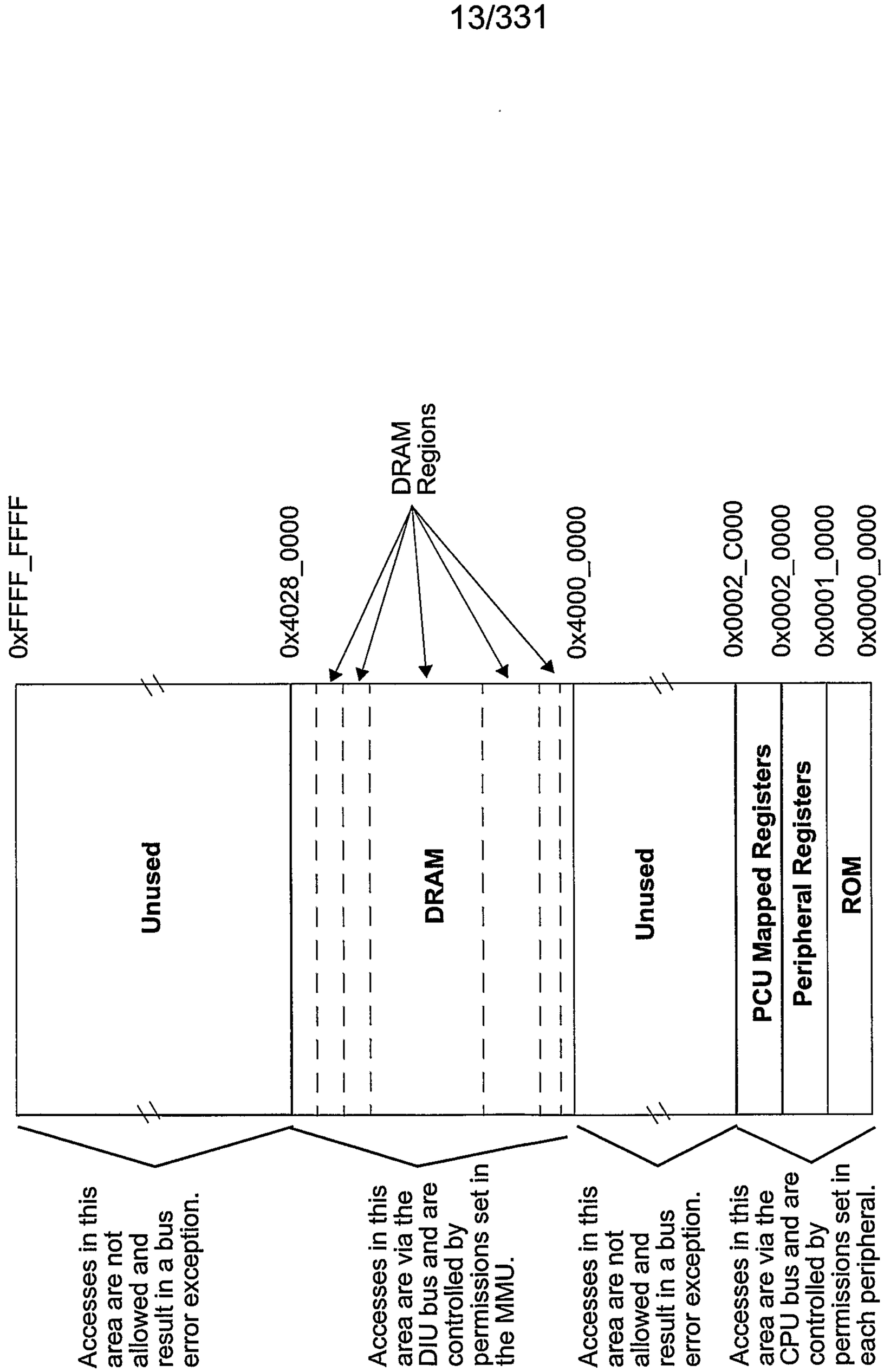

0xFFFF_FFFF
0x4028_0000
DRAM Regions
0x4000_0000
0x0002_C000
0x0002_0000
0x0001_0000
0x0000_0000
Unused
DRAM
Unused
PCU Mapped Registers
Peripheral Registers
ROM
Accesses in this area are not allowed and result in a bus error exception.
Accesses in this area are via the DIU bus and are controlled by permissions set in the MMU.
Accesses in this area are not allowed and result in a bus error exception.
Accesses in this area are via the CPU bus and are controlled by permissions set in each peripheral.

FIG. 14

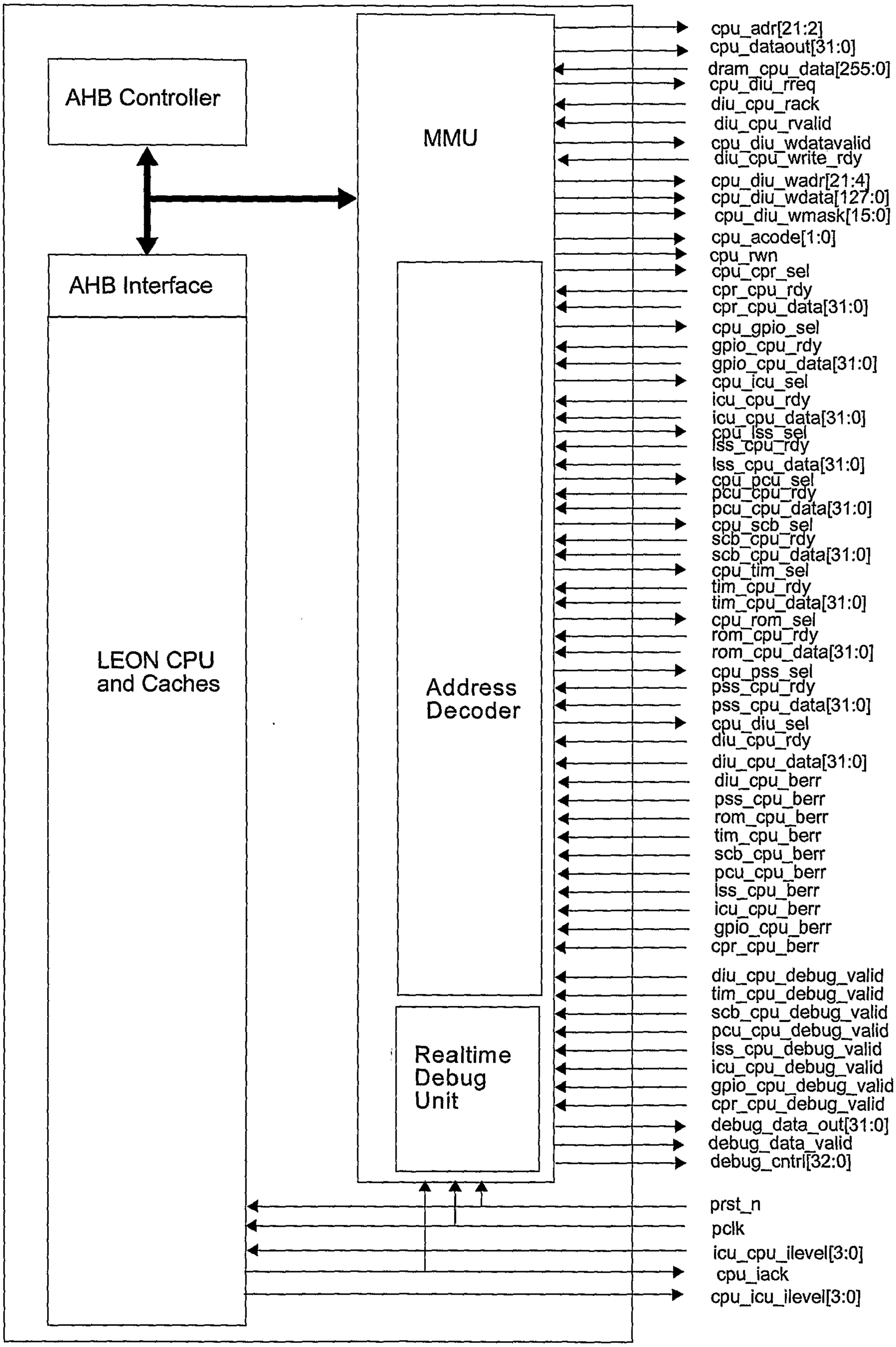

AHB Controller
AHB Interface
LEON CPU and Caches
MMU
Address Decoder
Realtime Debug Unit
cpu_adr[21:2]
cpu_dataout[31:0]
dram_cpu_data[255:0]
cpu_diu_rreq
diu_cpu_rack
diu_cpu_rvalid
cpu_diu_wdatavalid
diu_cpu_write_rdy
cpu_diu_wadr[21:4]
cpu_diu_wdata[127:0]
cpu_diu_wmask[15:0]
cpu_acode[1:0]
cpu_rwn
cpu_cpr_sel
cpr_cpu_rdy
cpr_cpu_data[31:0]
cpu_gpio_sel
gpio_cpu_rdy
gpio_cpu_data[31:0]
cpu_icu_sel
icu_cpu_rdy
icu_cpu_data[31:0]
cpu_lss_sel
lss_cpu_rdy
lss_cpu_data[31:0]
cpu_pcu_sel
pcu_cpu_rdy
pcu_cpu_data[31:0]
cpu_scb_sel
scb_cpu_rdy
scb_cpu_data[31:0]
cpu_tim_sel
tim_cpu_rdy
tim_cpu_data[31:0]
cpu_rom_sel
rom_cpu_rdy
rom_cpu_data[31:0]
cpu_pss_sel
pss_cpu_rdy
pss_cpu_data[31:0]
cpu_diu_sel
diu_cpu_rdy
diu_cpu_data[31:0]
diu_cpu_berr
pss_cpu_berr
rom_cpu_berr
tim_cpu_berr
scb_cpu_berr
pcu_cpu_berr
lss_cpu_berr
icu_cpu_berr
gpio_cpu_berr
cpr_cpu_berr
diu_cpu_debug_valid
tim_cpu_debug_valid
scb_cpu_debug_valid
pcu_cpu_debug_valid
lss_cpu_debug_valid
icu_cpu_debug_valid
gpio_cpu_debug_valid
cpr_cpu_debug_valid
debug_data_out[31:0]
debug_data_valid
debug_cntrl[32:0]
prst_n
pclk
icu_cpu_ilevel[3:0]
cpu_iack
cpu_icu_ilevel[3:0]

FIG. 15

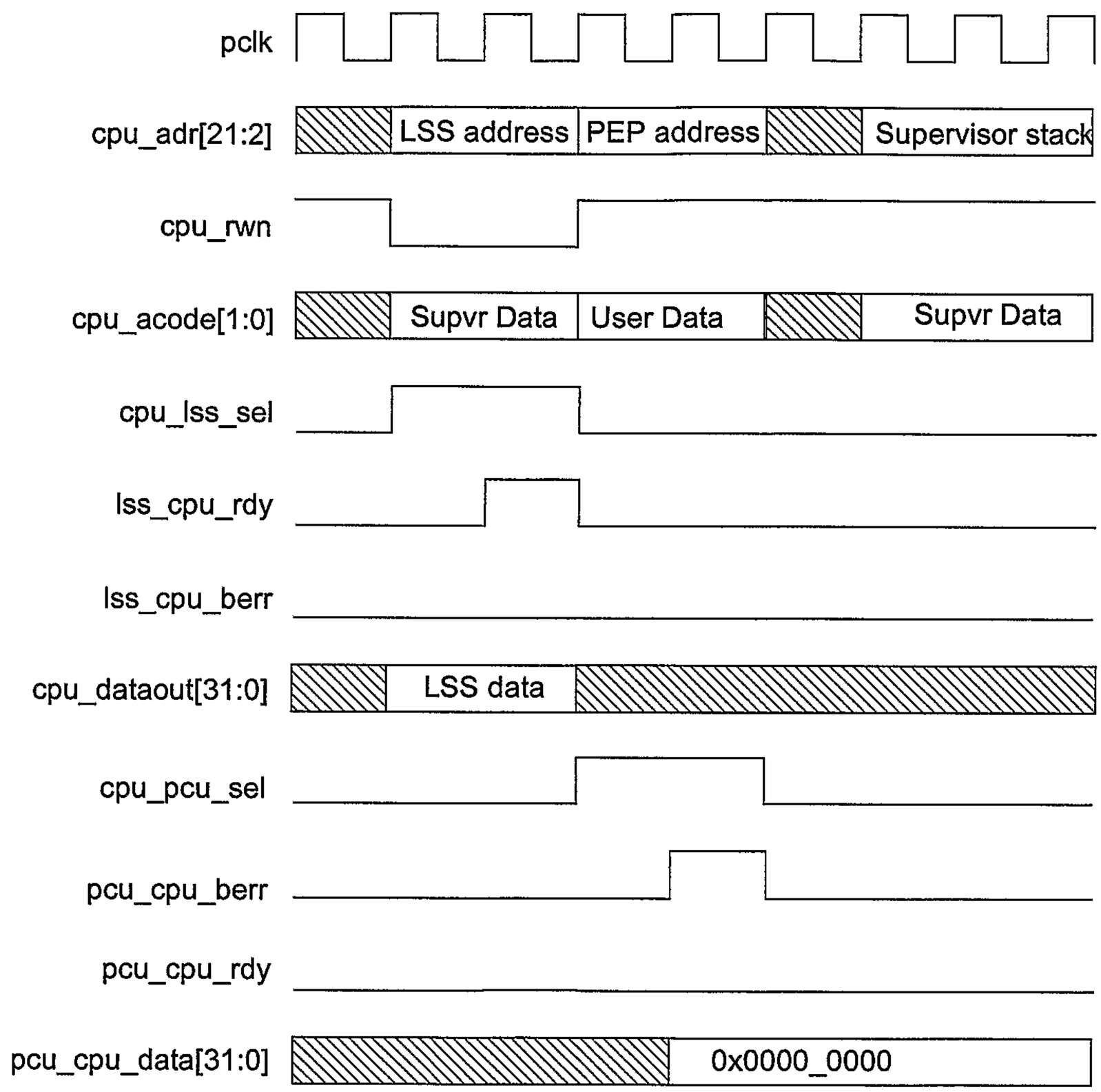

pclk
cpu_adr[21:2]
LSS address
PEP address
Supervisor stack
cpu_rwn
cpu_acode[1:0]
Supvr Data
User Data
Supvr Data
cpu_lss_sel
lss_cpu_rdy
lss_cpu_berr
cpu_dataout[31:0]
LSS data
cpu_pcu_sel
pcu_cpu_berr
pcu_cpu_rdy
pcu_cpu_data[31:0]
0x0000_0000

FIG. 16

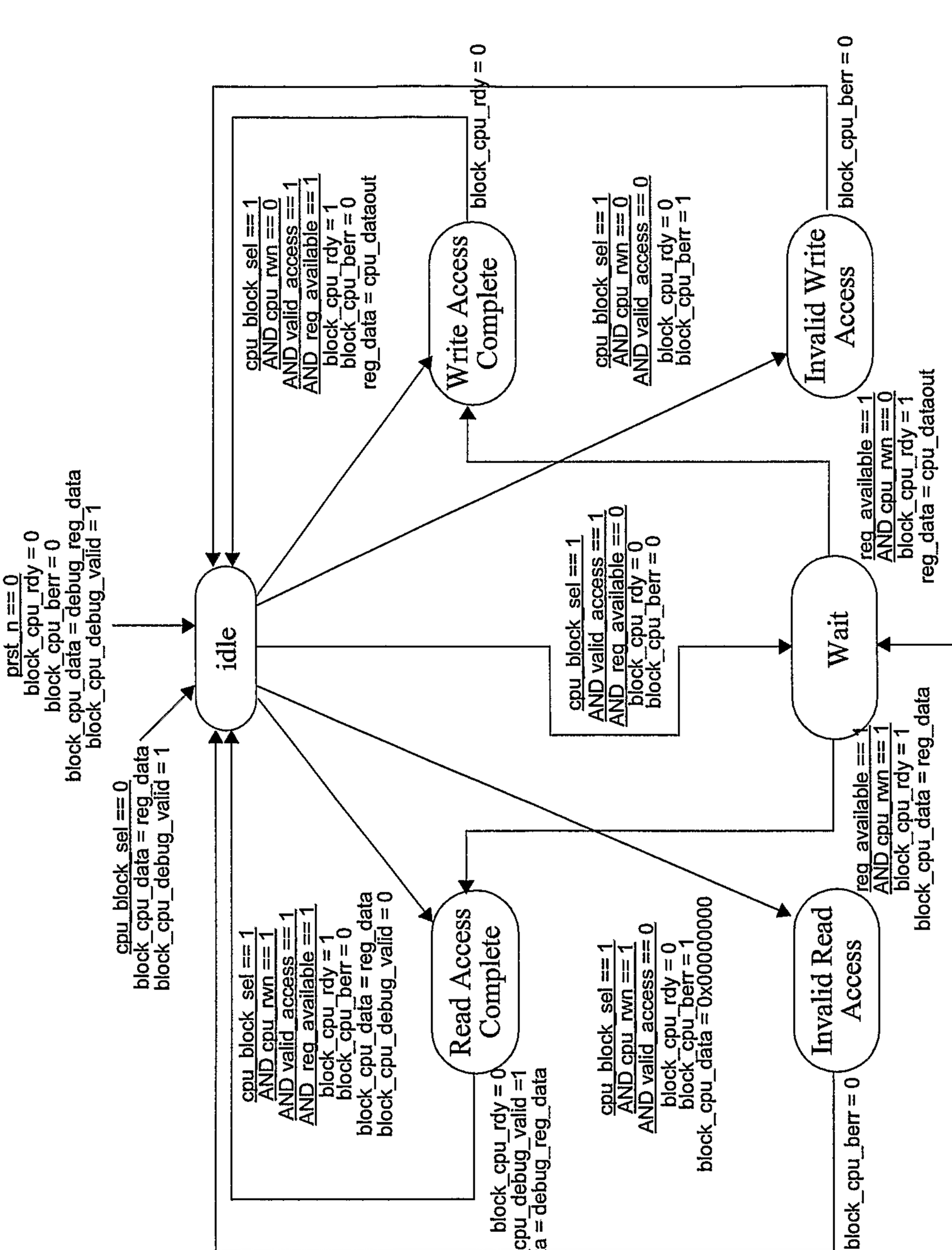

idle
Write Access Complete
Read Access Complete
Invalid Write Access
Invalid Read Access
Wait
prst_n == 0
block_cpu_rdy = 0
block_cpu_berr = 0
block_cpu_data = debug_reg_data
block_cpu_debug_valid = 1
cpu_block_sel == 0
block_cpu_data = reg_data
block_cpu_debug_valid = 1
cpu_block_sel == 1
AND cpu_rwn == 0
AND valid_access == 1
AND reg_available == 1
block_cpu_rdy = 1
block_cpu_berr = 0
reg_data = cpu_dataout
block_cpu_rdy = 0
cpu_block_sel == 1
AND cpu_rwn == 0
AND valid_access == 0
block_cpu_rdy = 0
block_cpu_berr = 1
block_cpu_berr = 0
cpu_block_sel == 1
AND cpu_rwn == 1
AND valid_access == 1
AND reg_available == 1
block_cpu_rdy = 1
block_cpu_berr = 0
block_cpu_data = reg_data
block_cpu_debug_valid = 0
block_cpu_rdy = 0
block_cpu_debug_valid =1
block_cpu_debug_reg_data
cpu_block_sel == 1
AND cpu_rwn == 1
AND valid_access == 0
block_cpu_rdy = 0
block_cpu_berr = 1
block_cpu_data = 0x00000000
reg_available == 1
AND cpu_rwn == 1
block_cpu_rdy = 1
block_cpu_data = reg_data
reg_available == 0

FIG. 17

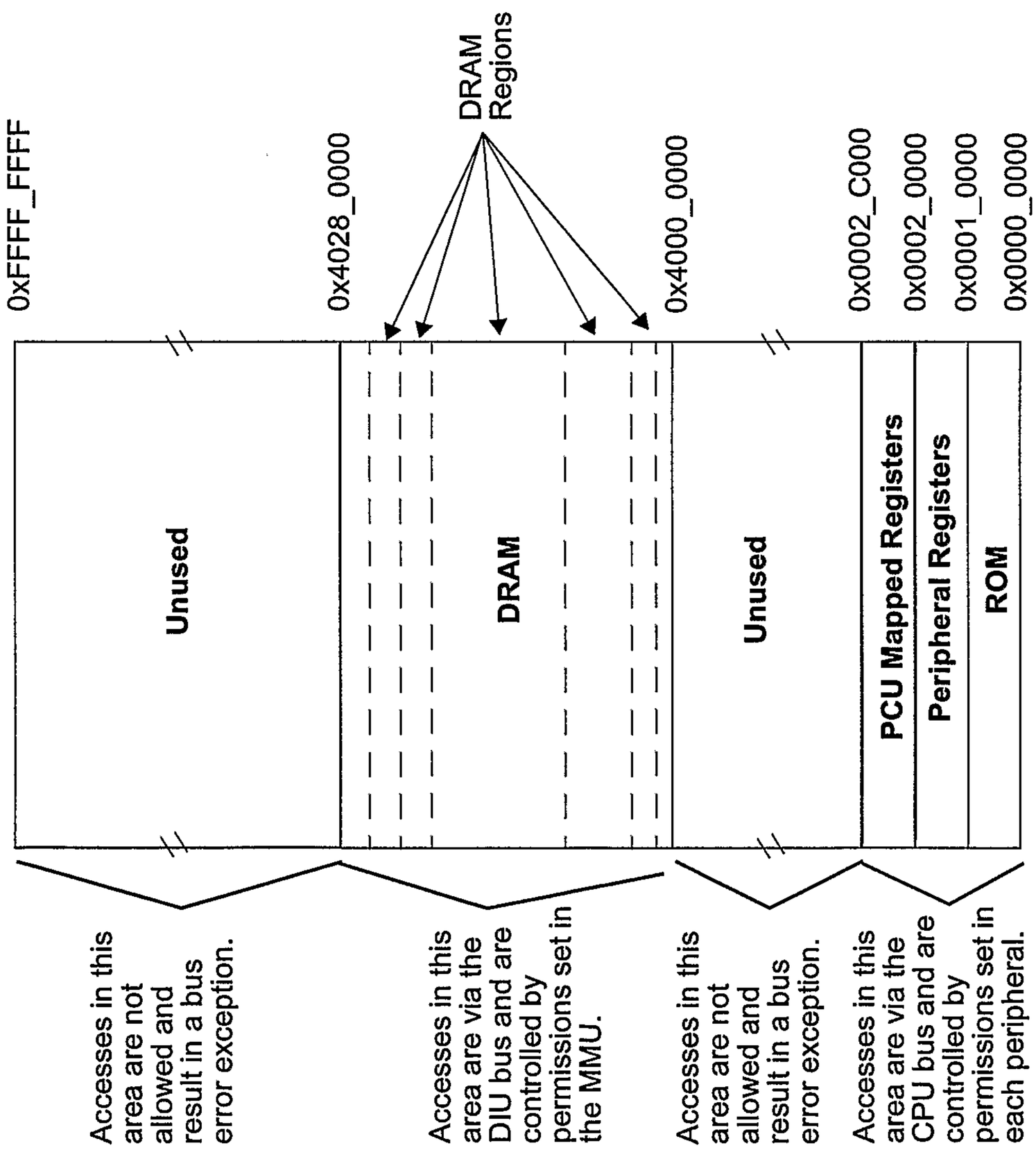

0xFFFF_FFFF
0x4028_0000
DRAM Regions
0x4000_0000
0x0002_C000
0x0002_0000
0x0001_0000
0x0000_0000
Unused
DRAM
Unused
PCU Mapped Registers
Peripheral Registers
ROM
Accesses in this area are not allowed and result in a bus error exception.
Accesses in this area are via the DIU bus and are controlled by permissions set in the MMU.
Accesses in this area are not allowed and result in a bus error exception.
Accesses in this area are via the CPU bus and are controlled by permissions set in each peripheral.

FIG. 18

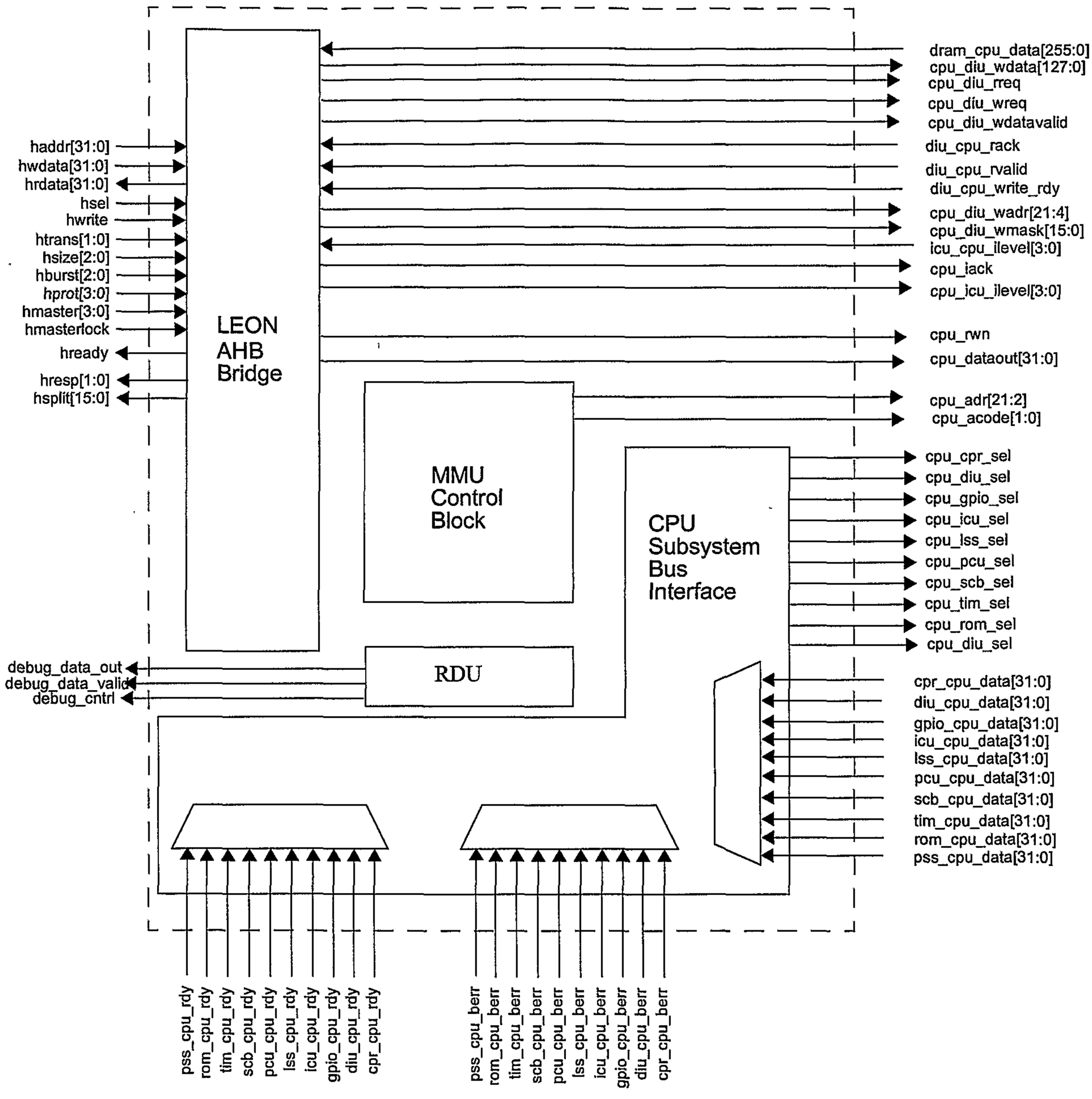

haddr[31:0]
hwdata[31:0]
hrdata[31:0]
hsel
hwrite
htrans[1:0]
hsize[2:0]
hburst[2:0]
hprot[3:0]
hmaster[3:0]
hmasterlock
hready
hresp[1:0]
hsplit[15:0]
LEON AHB Bridge
MMU Control Block
CPU Subsystem Bus Interface
RDU
debug_data_out
debug_data_valid
debug_cntrl
dram_cpu_data[255:0]
cpu_diu_wdata[127:0]
cpu_diu_rreq
cpu_diu_wreq
cpu_diu_wdatavalid
diu_cpu_rack
diu_cpu_rvalid
diu_cpu_write_rdy
cpu_diu_wadr[21:4]
cpu_diu_wmask[15:0]
icu_cpu_ilevel[3:0]
cpu_iack
cpu_icu_ilevel[3:0]
cpu_rwn
cpu_dataout[31:0]
cpu_adr[21:2]
cpu_acode[1:0]
cpu_cpr_sel
cpu_diu_sel
cpu_gpio_sel
cpu_icu_sel
cpu_lss_sel
cpu_pcu_sel
cpu_scb_sel
cpu_tim_sel
cpu_rom_sel
cpu_diu_sel
cpr_cpu_data[31:0]
diu_cpu_data[31:0]
gpio_cpu_data[31:0]
icu_cpu_data[31:0]
lss_cpu_data[31:0]
pcu_cpu_data[31:0]
scb_cpu_data[31:0]
tim_cpu_data[31:0]
rom_cpu_data[31:0]
pss_cpu_data[31:0]
pss_cpu_rdy
rom_cpu_rdy
tim_cpu_rdy
scb_cpu_rdy
pcu_cpu_rdy
lss_cpu_rdy
icu_cpu_rdy
gpio_cpu_rdy
diu_cpu_rdy
cpr_cpu_rdy
pss_cpu_berr
rom_cpu_berr
tim_cpu_berr
scb_cpu_berr
pcu_cpu_berr
lss_cpu_berr
icu_cpu_berr
gpio_cpu_berr
diu_cpu_berr
cpr_cpu_berr

FIG. 19

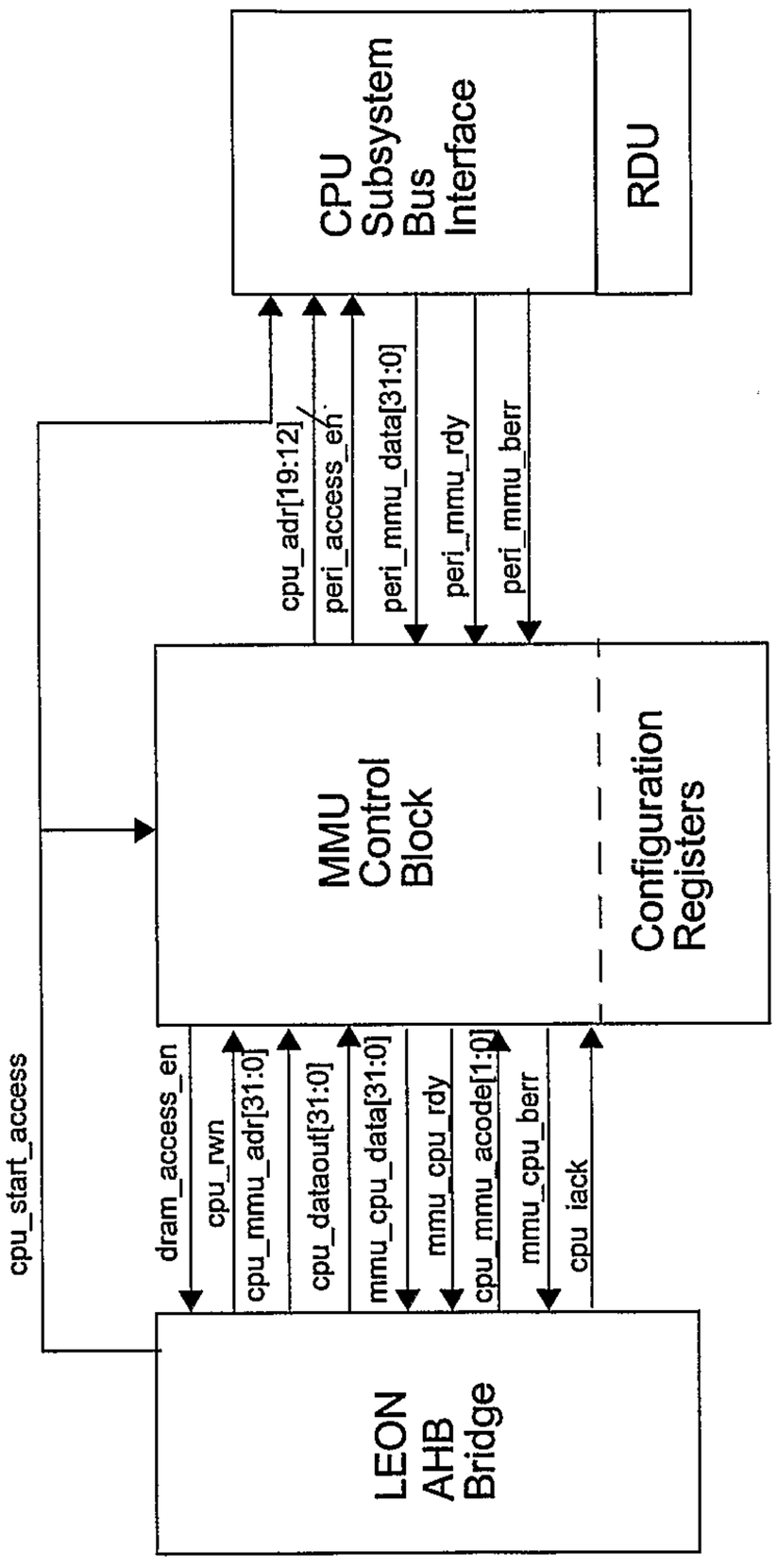

CPU Subsystem Bus Interface
RDU
cpu_adr[19:12]
peri_access_en
peri_mmu_data[31:0]
peri_mmu_rdy
peri_mmu_berr
MMU Control Block
Configuration Registers
cpu_start_access
dram_access_en
cpu_rwn
cpu_mmu_adr[31:0]
cpu_dataout[31:0]
mmu_cpu_data[31:0]
mmu_cpu_rdy
cpu_mmu_acode[1:0]
mmu_cpu_berr
cpu_iack
LEON AHB Bridge

FIG. 20

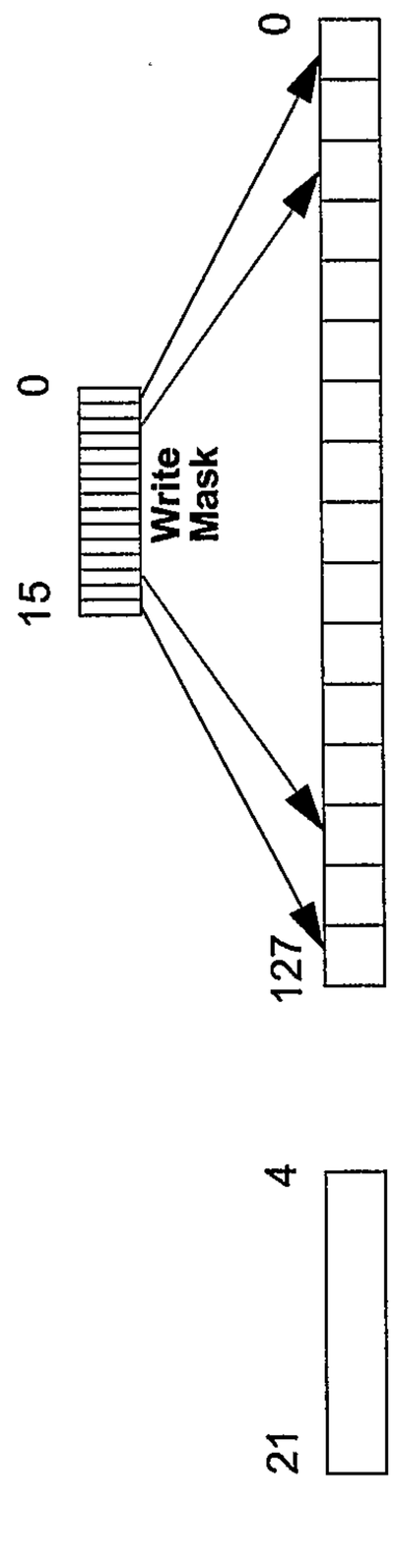

15
0
Write Mask
0
127
Write Data Buffer
21
4
Write Address Tag

FIG. 21

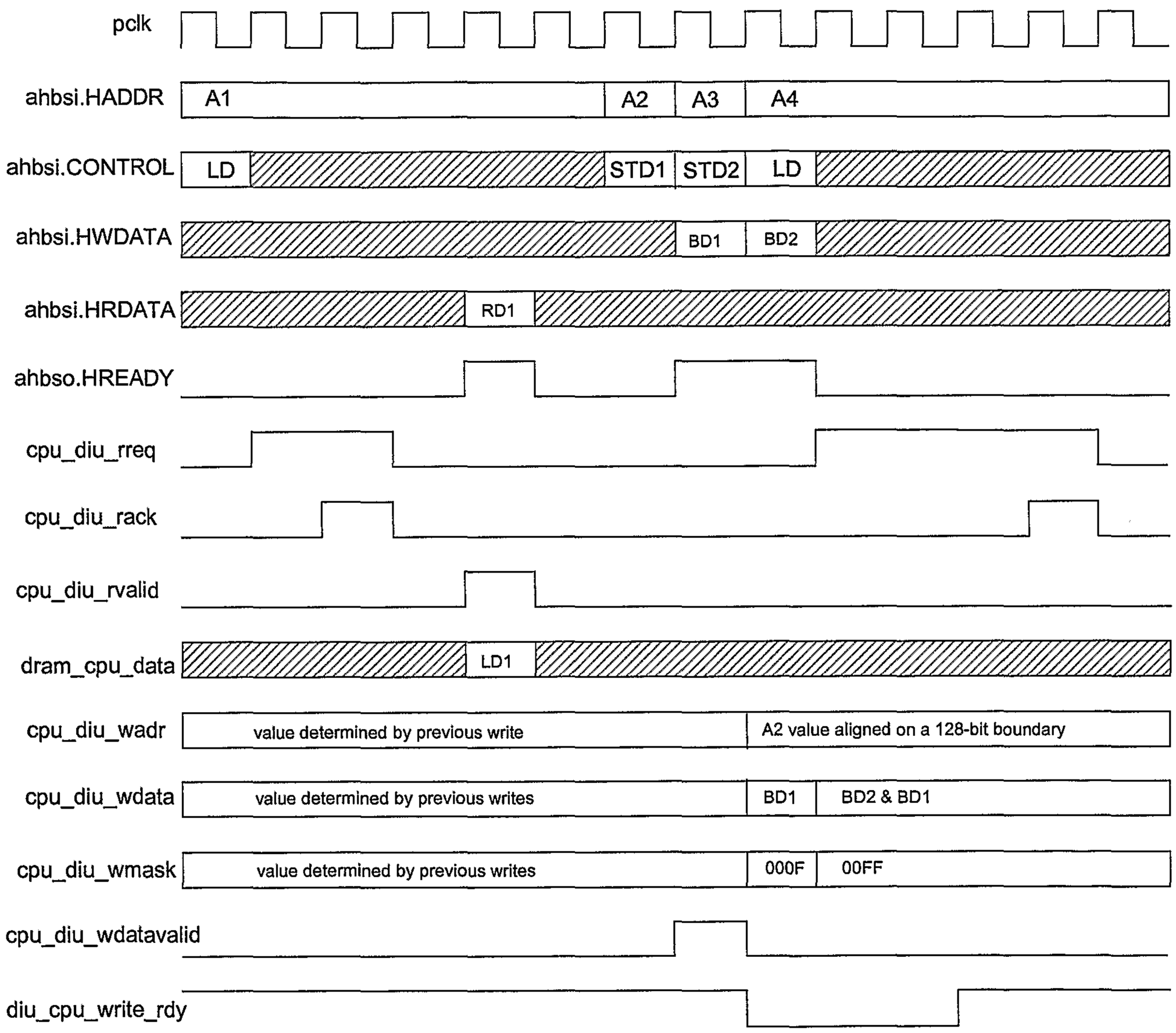

pclk
ahbsi.HADDR
A1
A2
A3
A4
ahbsi.CONTROL
LD
STD1
STD2
LD
ahbsi.HWDATA
BD1
BD2
ahbsi.HRDATA
RD1
ahbso.HREADY
cpu_diu_rreq
cpu_diu_rack
cpu_diu_rvalid
dram_cpu_data
LD1
cpu_diu_wadr
value determined by previous write
A2 value aligned on a 128-bit boundary
cpu_diu_wdata
value determined by previous writes
BD1
BD2 & BD1
cpu_diu_wmask
value determined by previous writes
000F
00FF
cpu_diu_wdatavalid
diu_cpu_write_rdy

FIG. 22

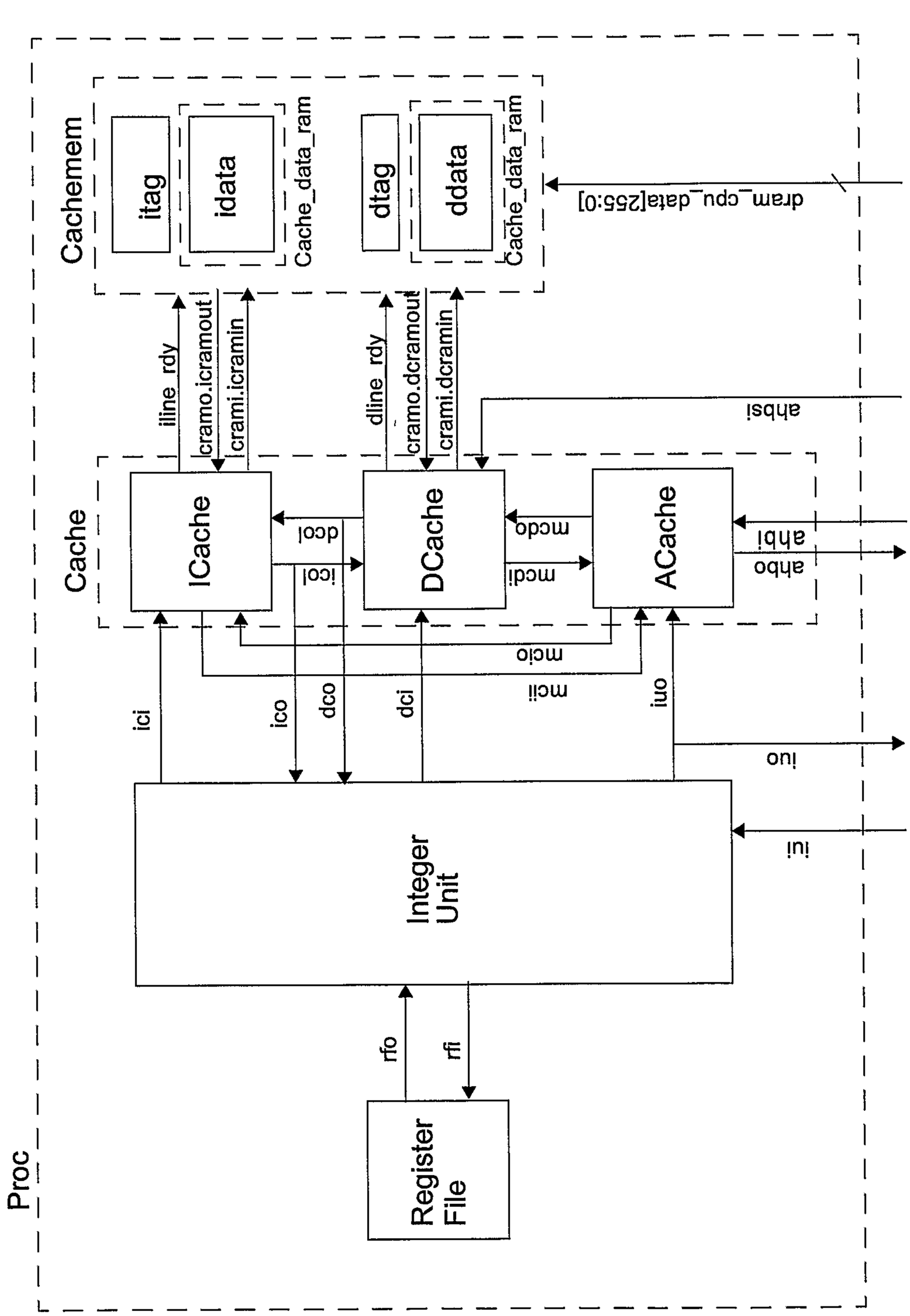

Proc
Cachemem
itag
idata
Cache_data_ram
dtag
ddata
Cache_data_ram
dram_cpu_data[255:0]
ahbsi
iline_rdy
cramo.icramout
crami.icramin
dline_rdy
cramo.dcramout
crami.dcramin
Cache
ICache
DCache
ACache
dcol
icol
mcdo
mcdi
ahbi
ahbo
mcio
mcii
ici
ico
dco
dci
oni
iuo
iui
Integer Unit
rfo
rfi
Register File

FIG. 23

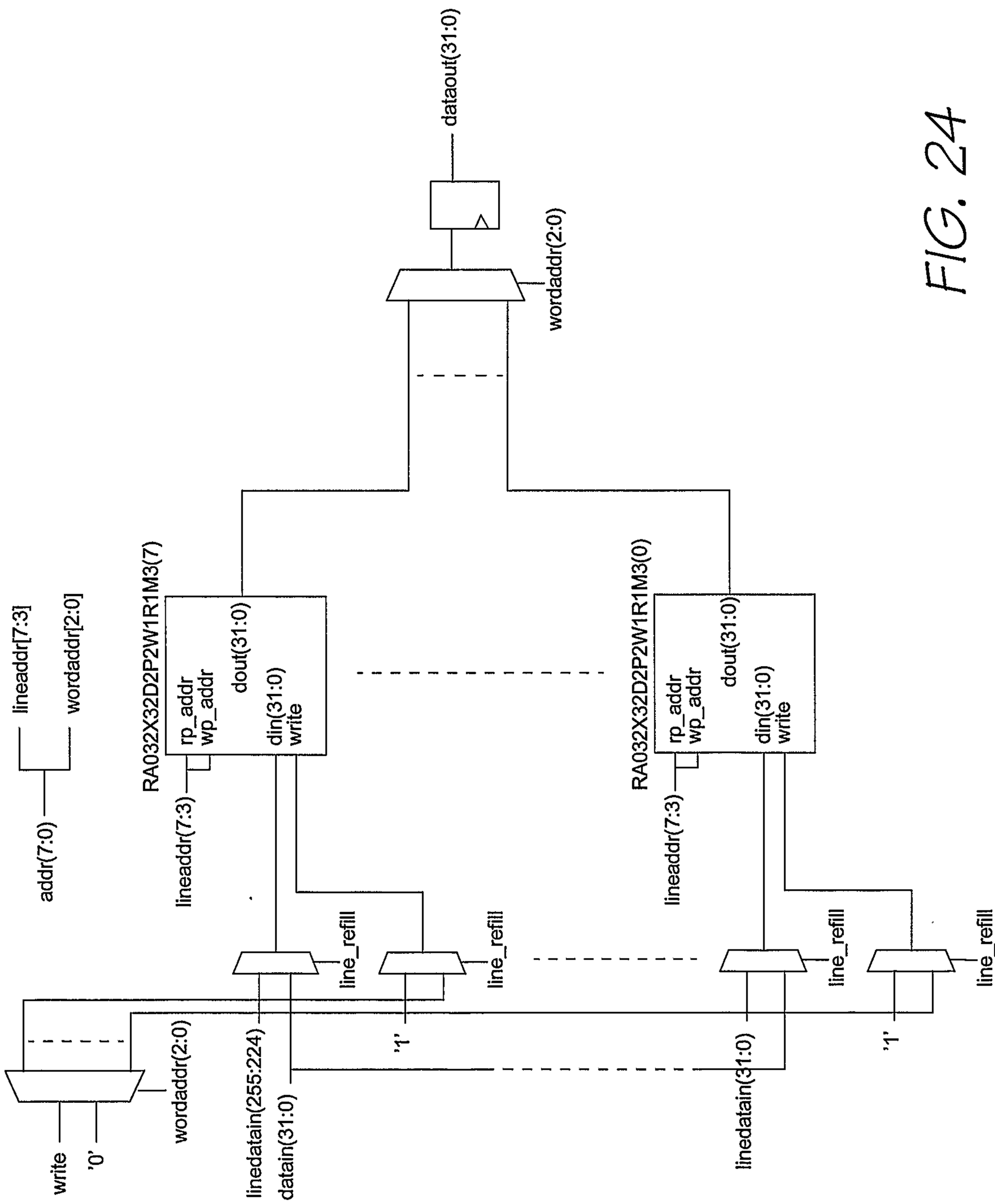

dataout(31:0)
wordaddr(2:0)
lineaddr[7:3]
wordaddr[2:0]
addr(7:0)
RA032X32D2P2W1R1M3(7)
rp_addr
wp_addr
dout(31:0)
din(31:0)
write
lineaddr(7:3)
RA032X32D2P2W1R1M3(0)
line_refill
'1'
'0'
linedatain(255:224)
datain(31:0)
linedatain(31:0)

FIG. 24

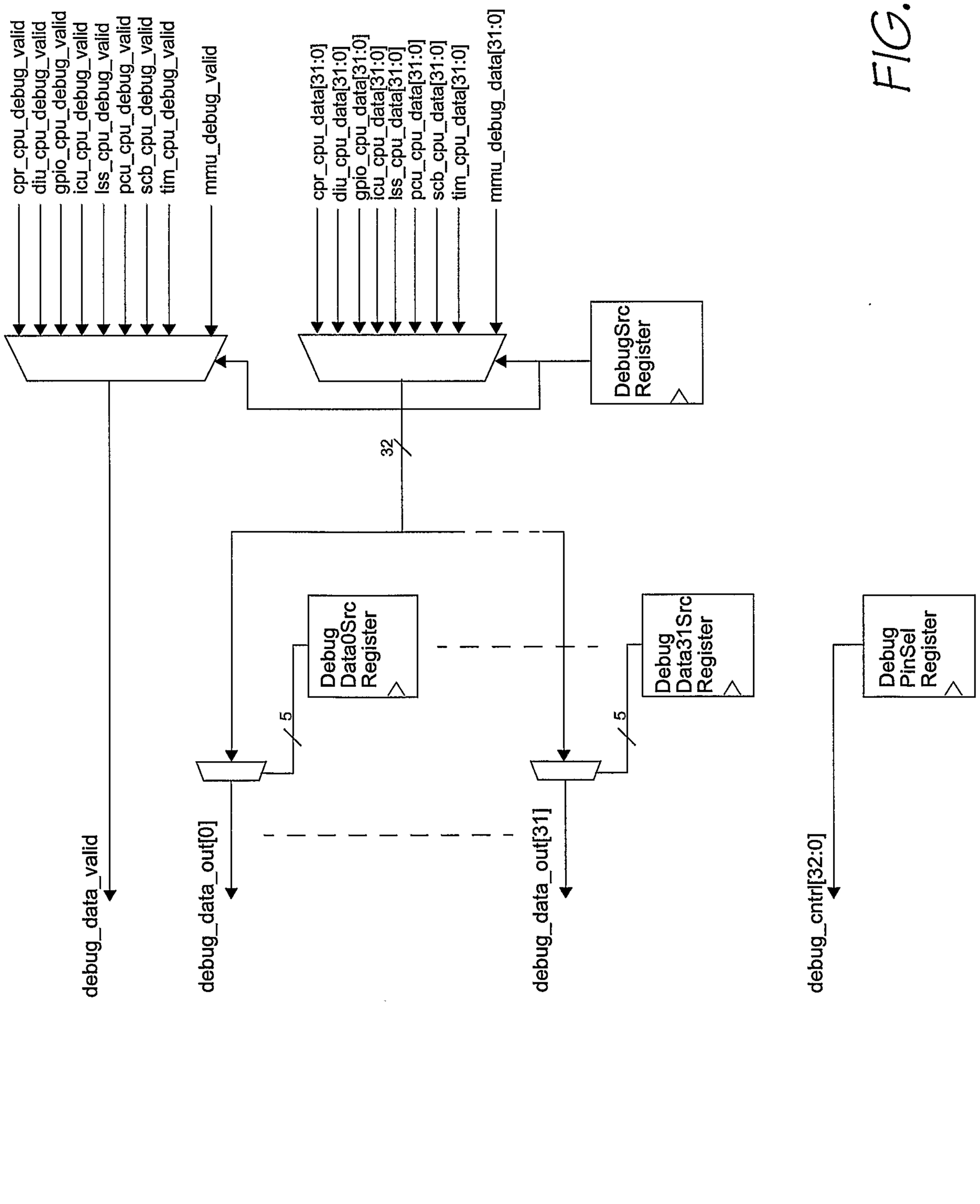

cpr_cpu_debug_valid
diu_cpu_debug_valid
gpio_cpu_debug_valid
icu_cpu_debug_valid
lss_cpu_debug_valid
pcu_cpu_debug_valid
scb_cpu_debug_valid
tim_cpu_debug_valid
mmu_debug_valid
cpr_cpu_data[31:0]
diu_cpu_data[31:0]
gpio_cpu_data[31:0]
icu_cpu_data[31:0]
lss_cpu_data[31:0]
pcu_cpu_data[31:0]
scb_cpu_data[31:0]
tim_cpu_data[31:0]
mmu_debug_data[31:0]
Debug
Regist
32
Debug
Data0Src
Register
Debug
Data31Src
Register
Debug
PinSel
Register
5
5
debug_data_valid
debug_data_out[0]
debug_data_out[31]
debug_cntrl[32:0]

FIG. 25

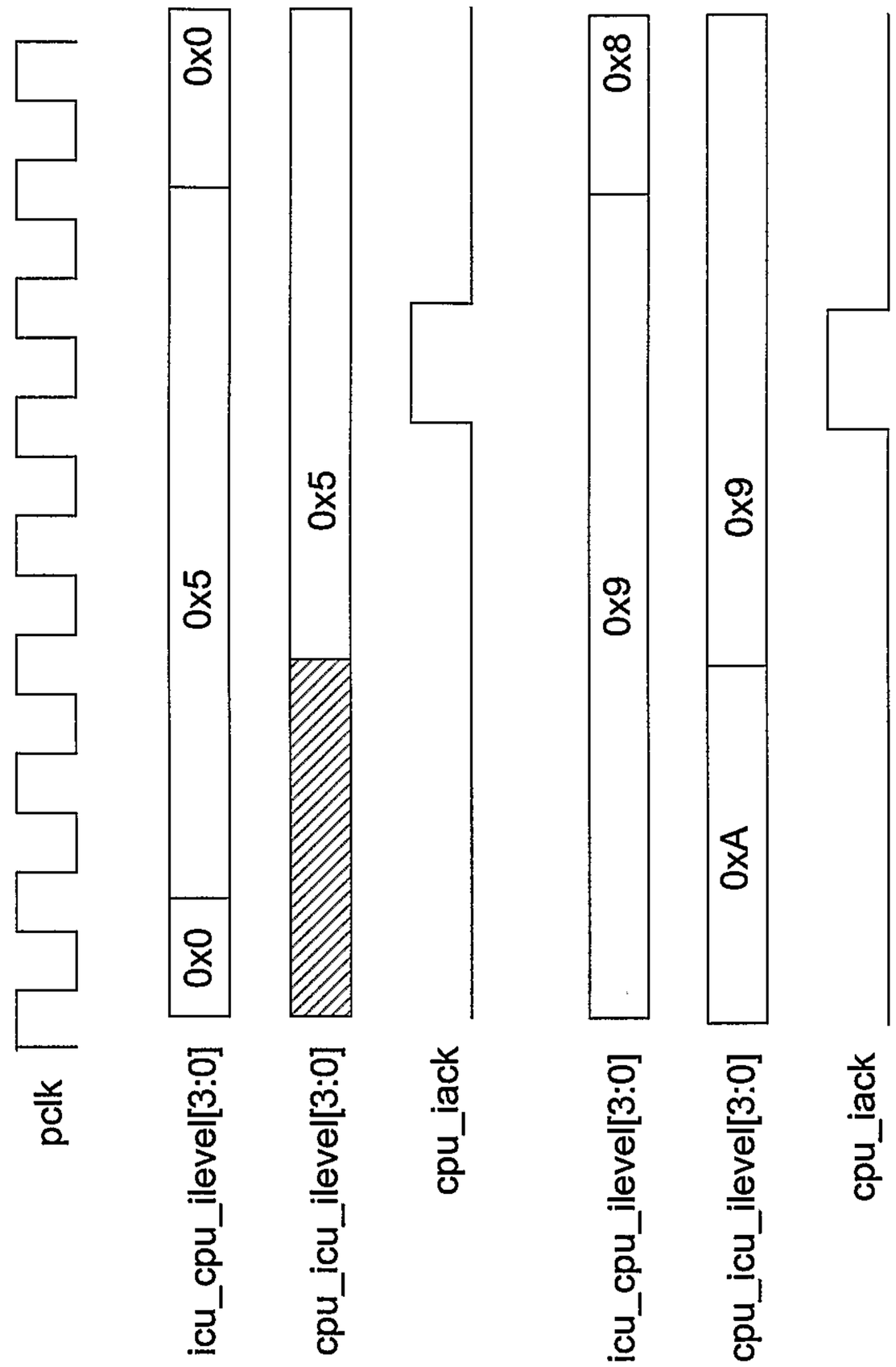

pclk
icu_cpu_ilevel[3:0]
0x0
0x5
cpu_icu_ilevel[3:0]
0x0
0x5
cpu_iack
icu_cpu_ilevel[3:0]
0x8
0x9
cpu_icu_ilevel[3:0]
0x9
0xA
cpu_iack

FIG. 26

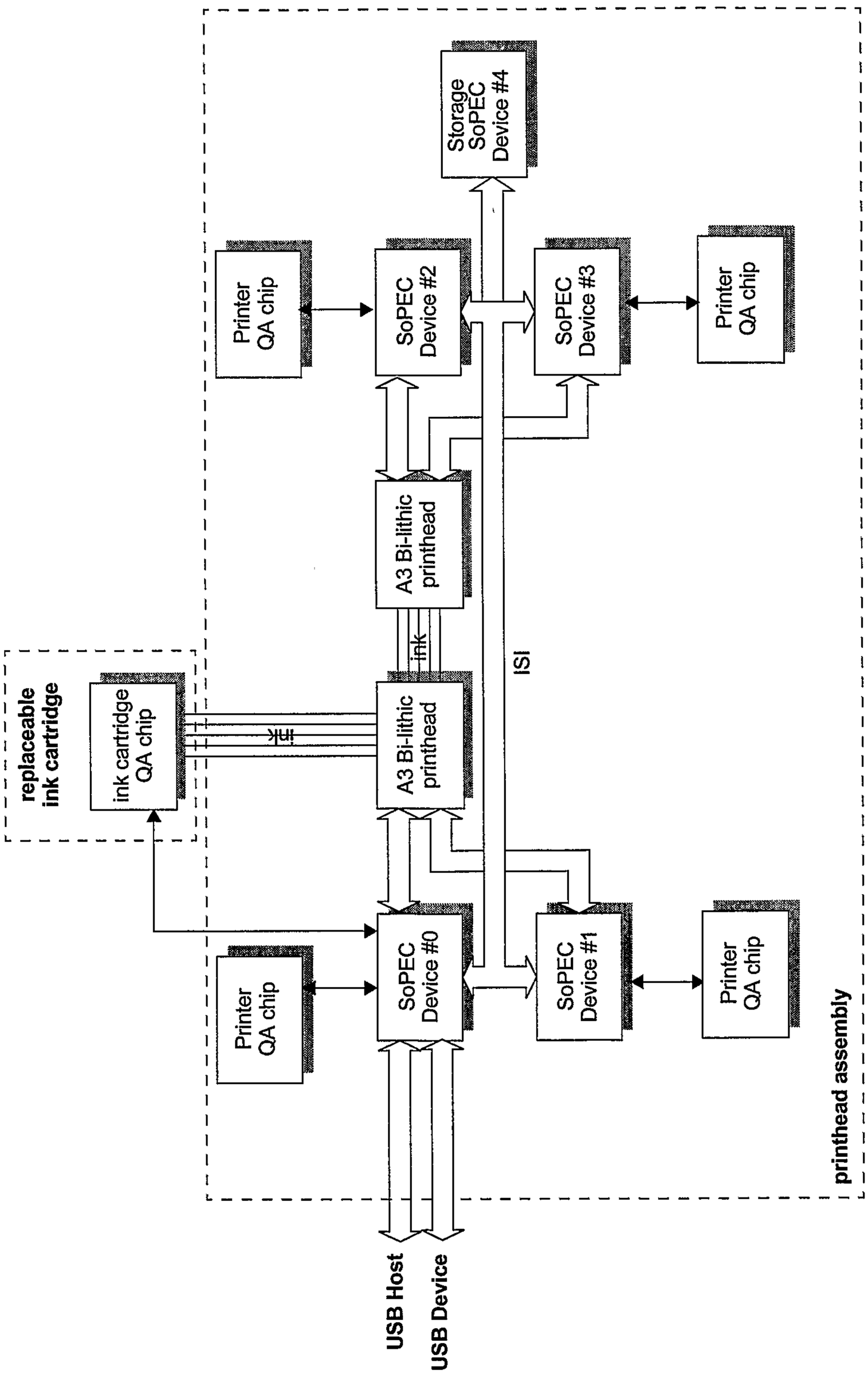

Storage SoPEC Device #4
Printer QA chip
SoPEC Device #2
SoPEC Device #3
Printer QA chip
A3 Bi-lithic printhead
ink
ISI
replaceable ink cartridge
ink cartridge QA chip
A3 Bi-lithic printhead
Printer QA chip
SoPEC Device #0
SoPEC Device #1
Printer QA chip
printhead assembly
USB Host
USB Device

FIG. 27

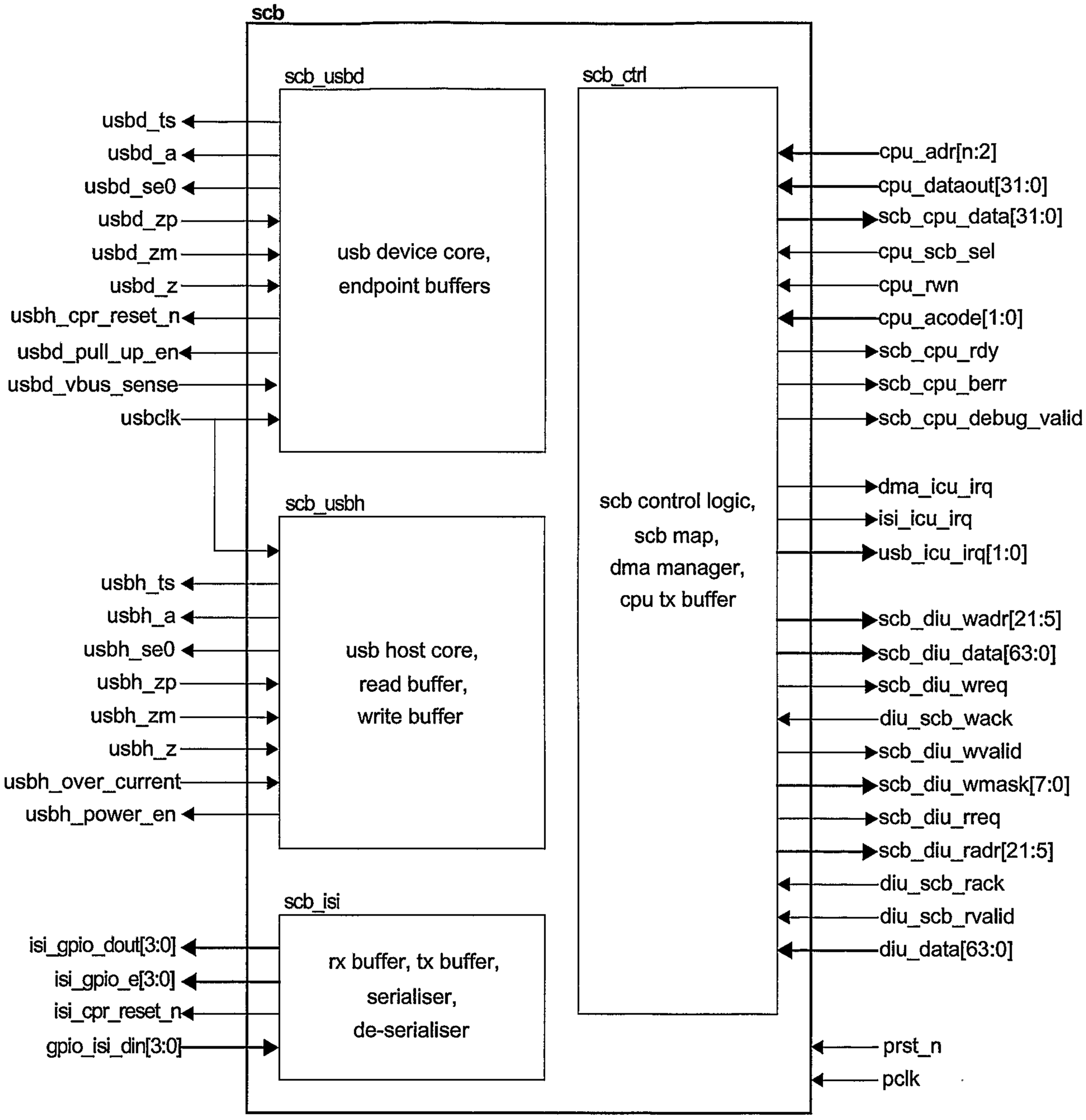

scb
scb_usbd
usbd_ts
usbd_a
usbd_se0
usbd_zp
usbd_zm
usbd_z
usbh_cpr_reset_n
usbd_pull_up_en
usbd_vbus_sense
usbclk
usb device core,
endpoint buffers
scb_usbh
usbh_ts
usbh_a
usbh_se0
usbh_zp
usbh_zm
usbh_z
usbh_over_current
usbh_power_en
usb host core,
read buffer,
write buffer
scb_isi
isi_gpio_dout[3:0]
isi_gpio_e[3:0]
isi_cpr_reset_n
gpio_isi_din[3:0]
rx buffer, tx buffer,
serialiser,
de-serialiser
scb_ctrl
scb control logic,
scb map,
dma manager,
cpu tx buffer
cpu_adr[n:2]
cpu_dataout[31:0]
scb_cpu_data[31:0]
cpu_scb_sel
cpu_rwn
cpu_acode[1:0]
scb_cpu_rdy
scb_cpu_berr
scb_cpu_debug_valid
dma_icu_irq
isi_icu_irq
usb_icu_irq[1:0]
scb_diu_wadr[21:5]
scb_diu_data[63:0]
scb_diu_wreq
diu_scb_wack
scb_diu_wvalid
scb_diu_wmask[7:0]
scb_diu_rreq
scb_diu_radr[21:5]
diu_scb_rack
diu_scb_rvalid
diu_data[63:0]
prst_n
pclk

FIG. 28

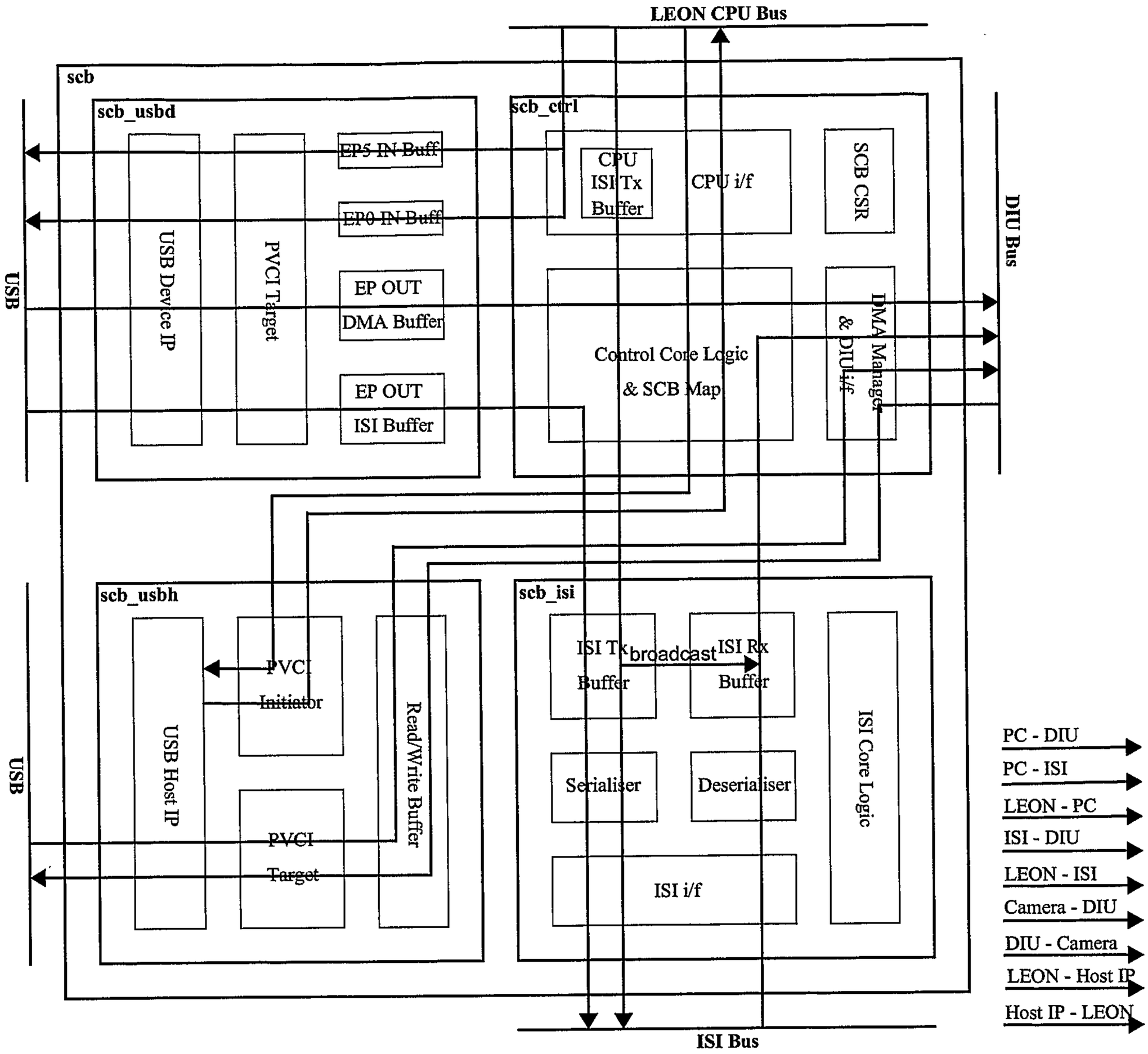

LEON CPU Bus
scb
scb_usbd
scb_ctrl
EP5 IN Buff
EP0 IN Buff
EP OUT
DMA Buffer
EP OUT
ISI Buffer
USB Device IP
PVCI Target
USB
CPU
ISI Tx
Buffer
CPU i/f
SCB CSR
DIU Bus
Control Core Logic
& SCB Map
DMA Manager
& DIU i/f
scb_usbh
scb_isi
PVCI
Initiator
PVCI
Target
USB Host IP
Read/Write Buffer
ISI Tx
Buffer
broadcast
ISI Rx
Buffer
Serialiser
Deserialiser
ISI i/f
ISI Core Logic
ISI Bus
PC - DIU
PC - ISI
LEON - PC
ISI - DIU
LEON - ISI
Camera - DIU
DIU - Camera
LEON - Host IP
Host IP - LEON

FIG. 29

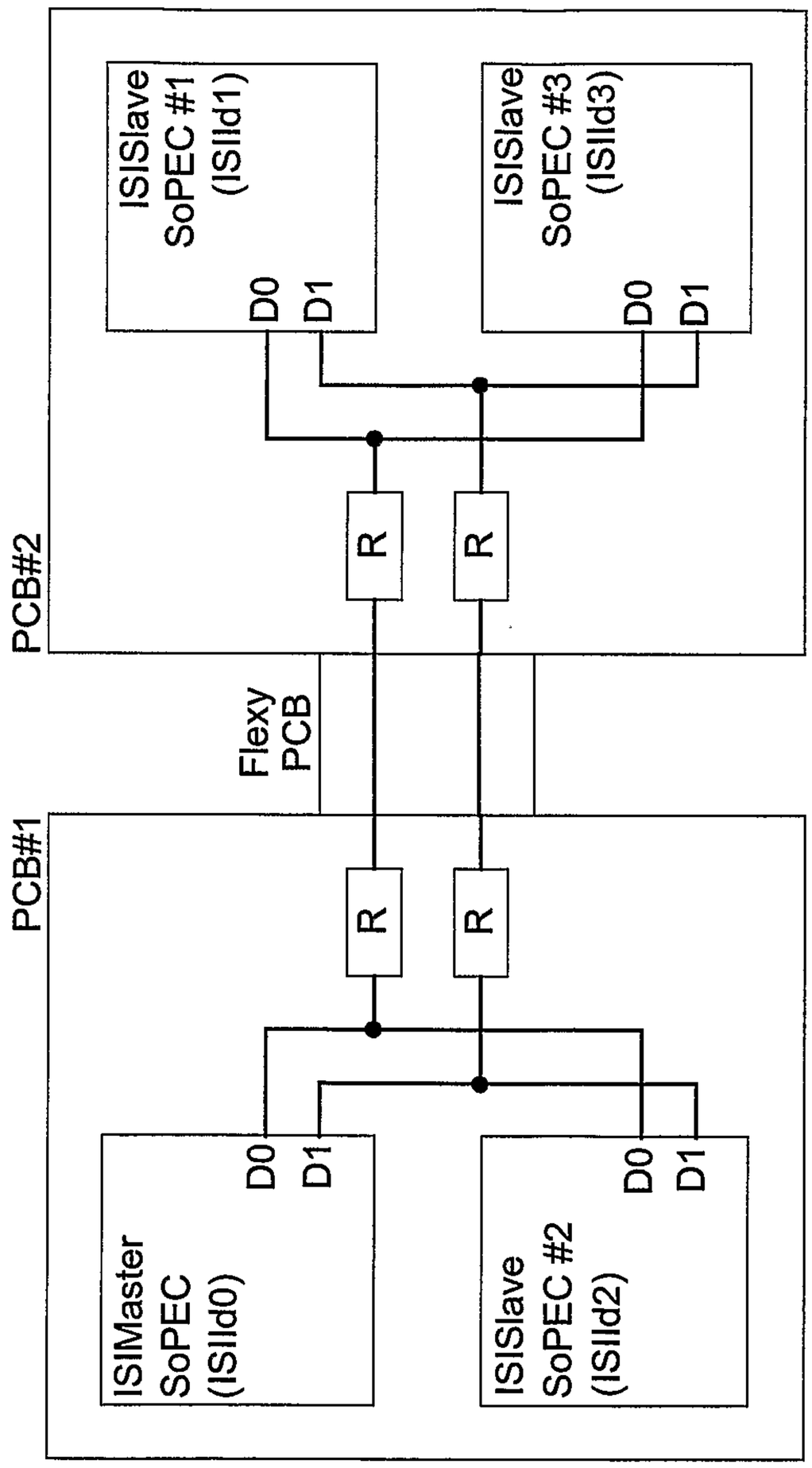

PCB#1
PCB#2
Flexy PCB
ISIMaster SoPEC (ISIId0)
ISISlave SoPEC #1 (ISIId1)
ISISlave SoPEC #2 (ISIId2)
ISISlave SoPEC #3 (ISIId3)
D0
D1
R

FIG. 30

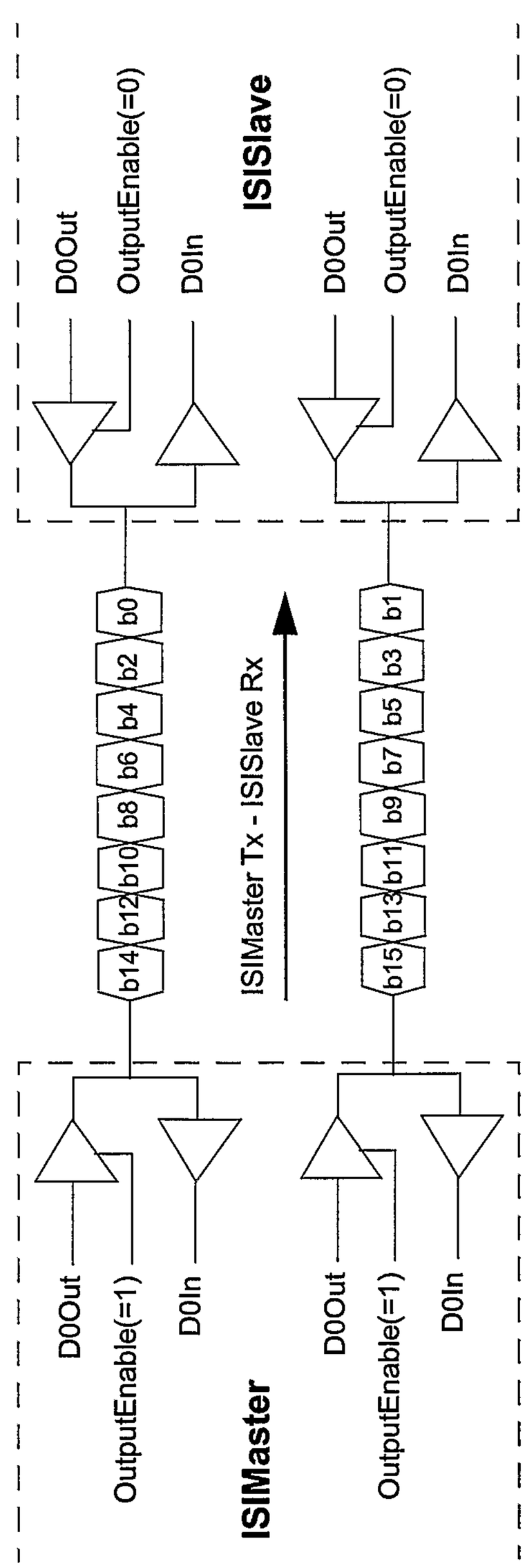
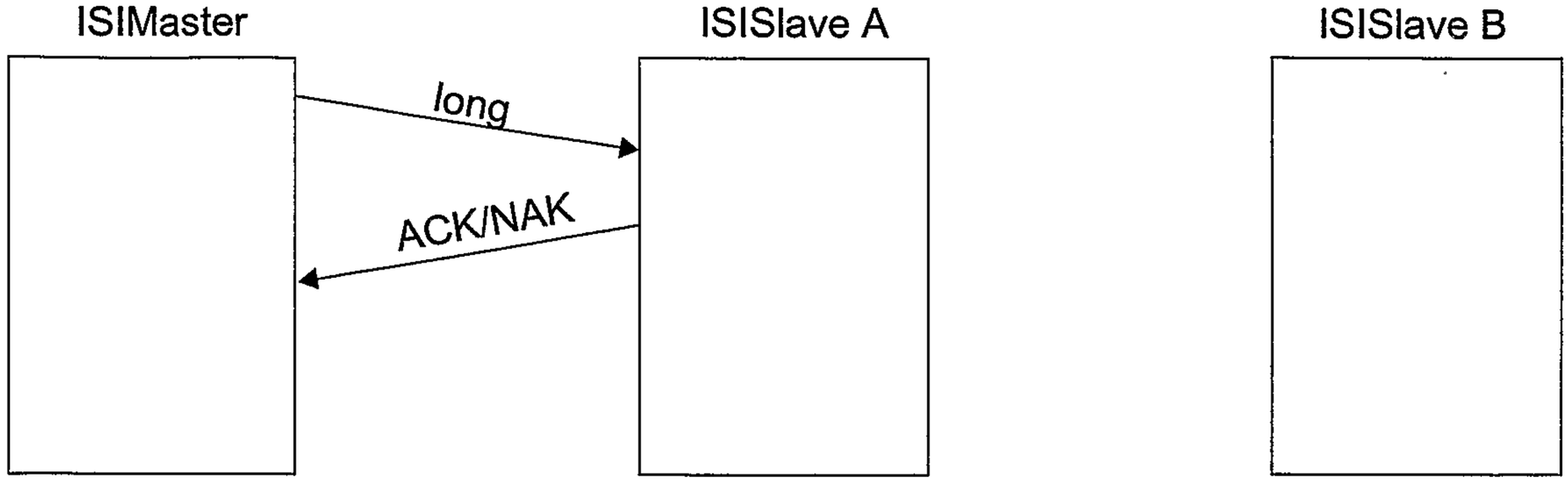
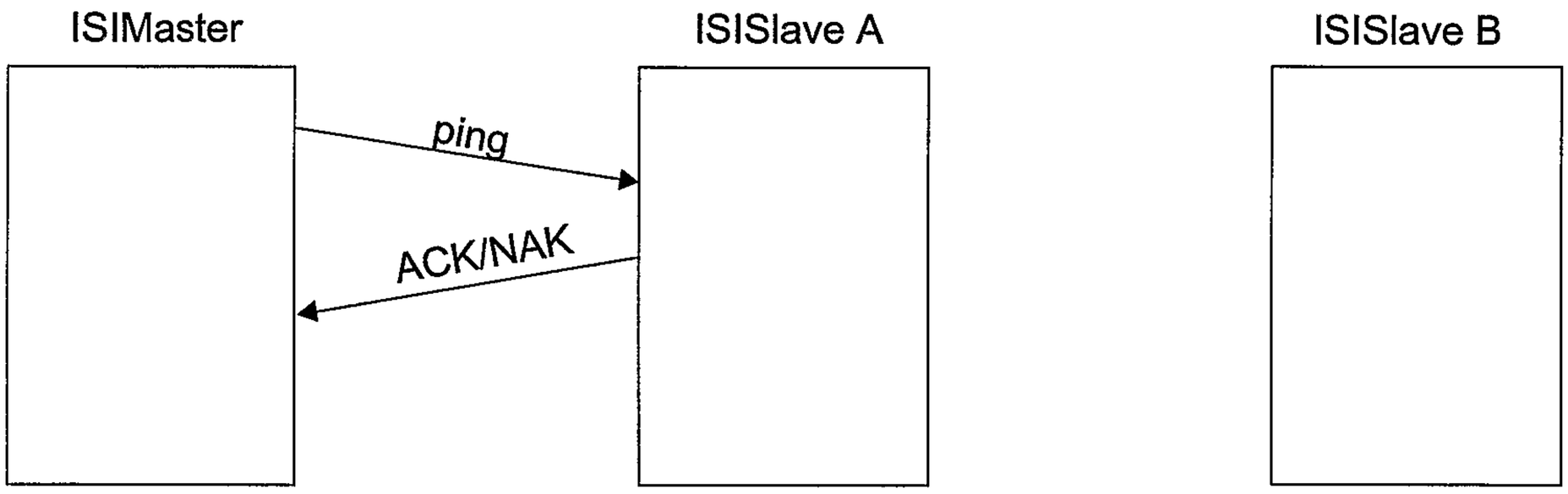

ISISlave
D0Out
OutputEnable(=0)
D0In
D0Out
OutputEnable(=0)
D0In
b0
b2
b4
b6
b8
b10
b12
b14
b1
b3
b5
b7
b9
b11
b13
b15
ISIMaster Tx - ISISlave Rx
ISIMaster
D0Out
OutputEnable(=1)
D0In
D0Out
OutputEnable(=1)
D0In

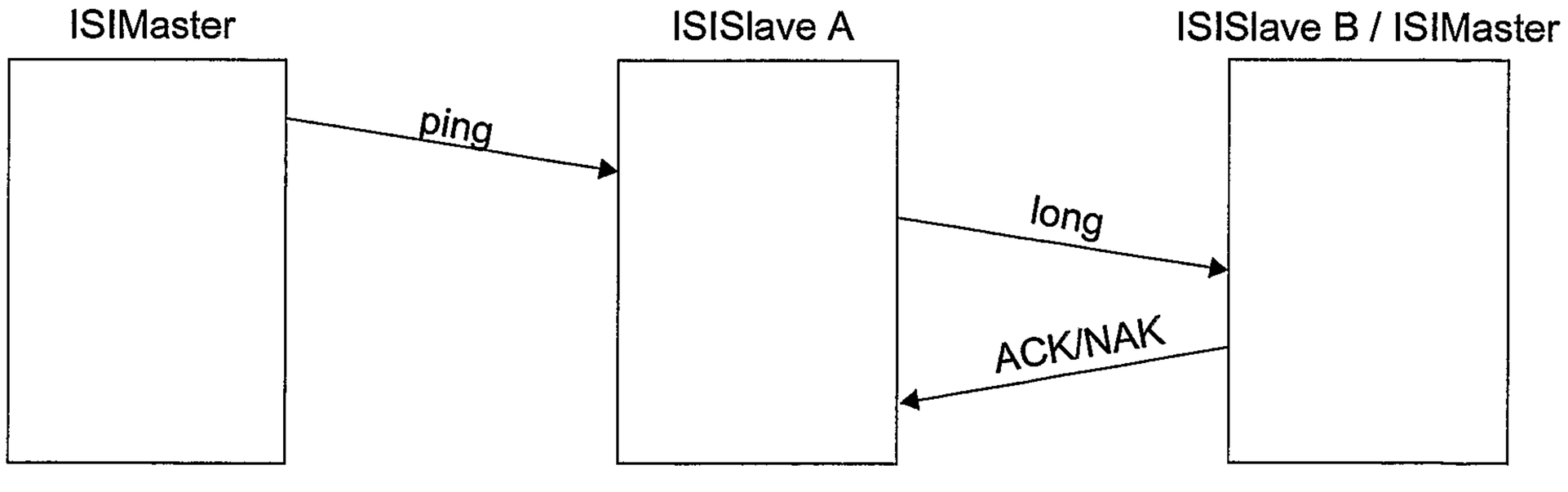
FIG. 31



Transaction 1: Long packet to an addressed ISISlave



Transaction 2: Ping packet to an addressed ISISlave. ISISlave has nothing to send



Transaction 3: Ping packet to an addressed ISISlave. ISISlaveA responds with a long packet to ISISlaveB (or the ISIMaster) and ISISlaveB (or the ISIMaster) responds with an ACK or NAK.

FIG. 32

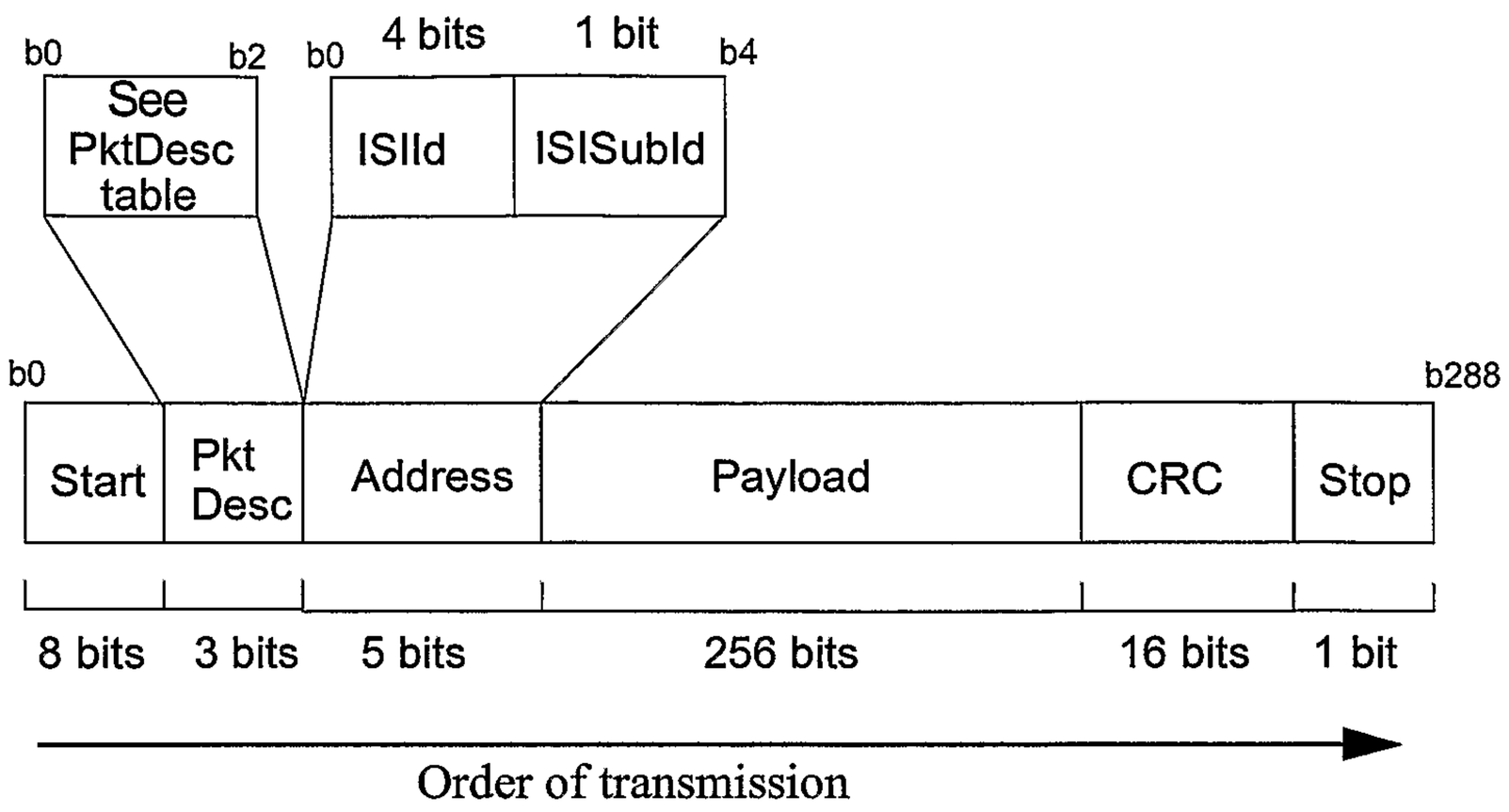

b0
b2
See PktDesc table
b0
4 bits
1 bit
b4
ISIId
ISISubId
b0
b288
Start
Pkt Desc
Address
Payload
CRC
Stop
8 bits
3 bits
5 bits
256 bits
16 bits
1 bit
Order of transmission

FIG. 33

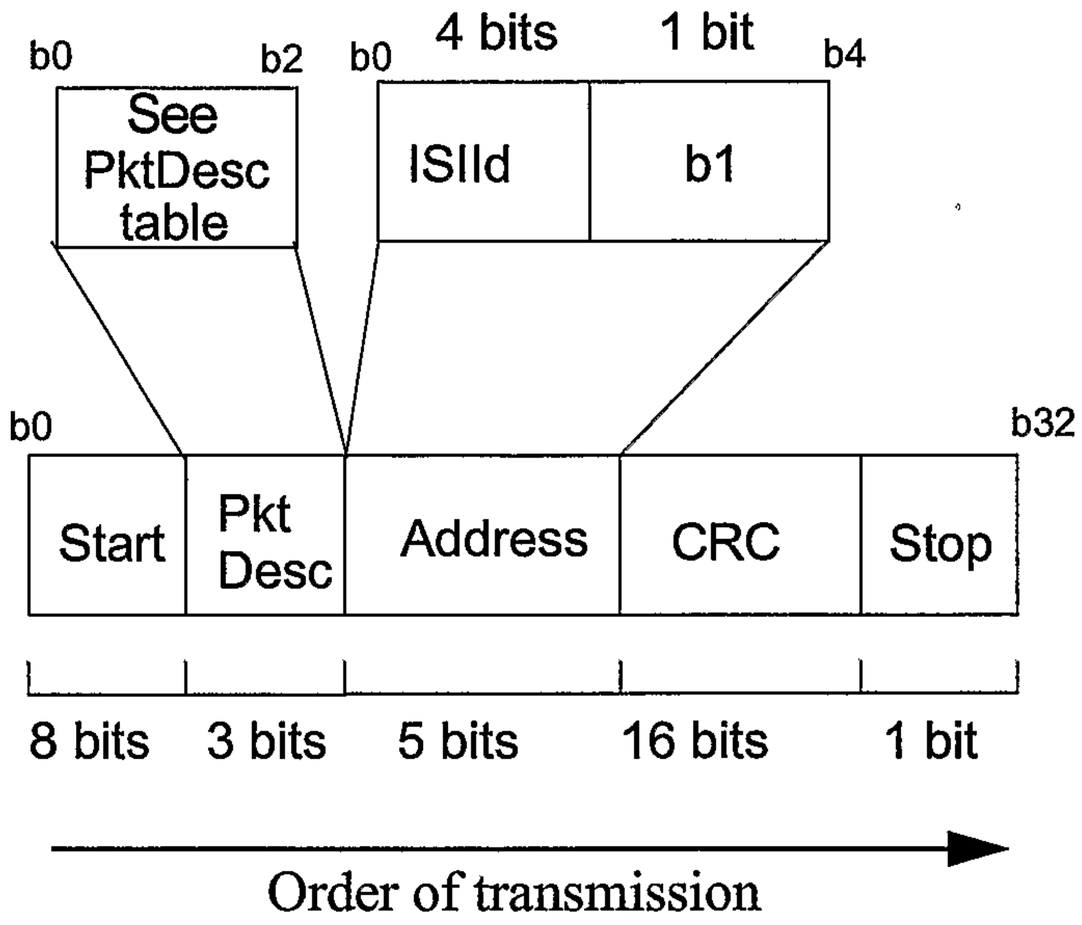

b0
b2
See PktDesc table
b0
4 bits
1 bit
b4
ISIId
b1
b0
b32
Start
Pkt Desc
Address
CRC
Stop
8 bits
3 bits
5 bits
16 bits
1 bit
Order of transmission

FIG. 34

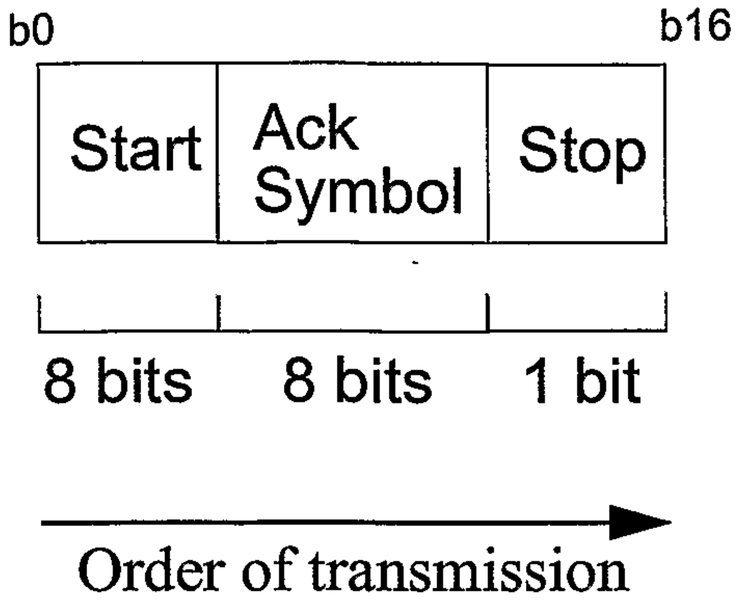

b0
b16
Start
Ack Symbol
Stop
8 bits
8 bits
1 bit
Order of transmission

FIG. 35

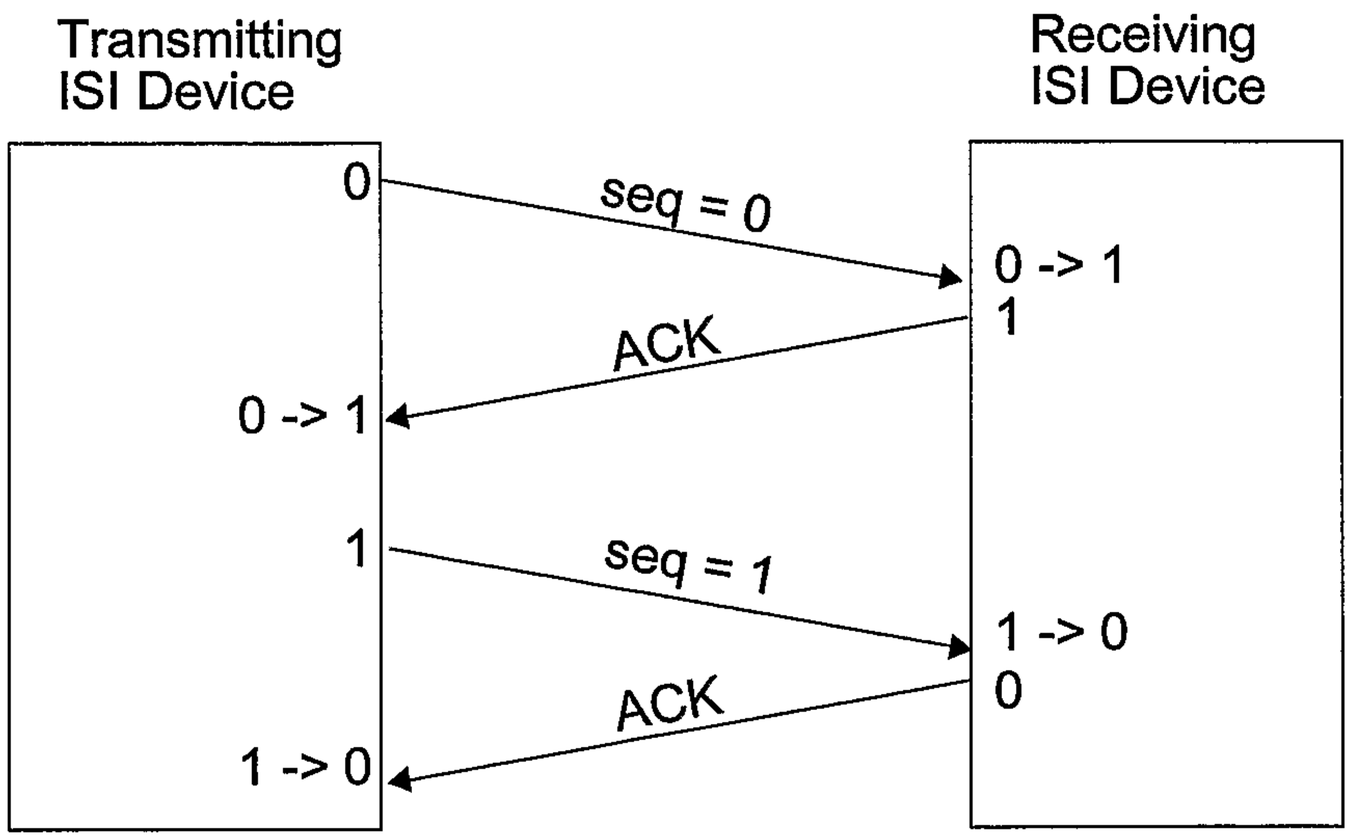

Transmitting ISI Device
Receiving ISI Device
0
seq = 0
0 -> 1
1
ACK
0 -> 1
1
seq = 1
1 -> 0
0
ACK
1 -> 0

FIG. 36

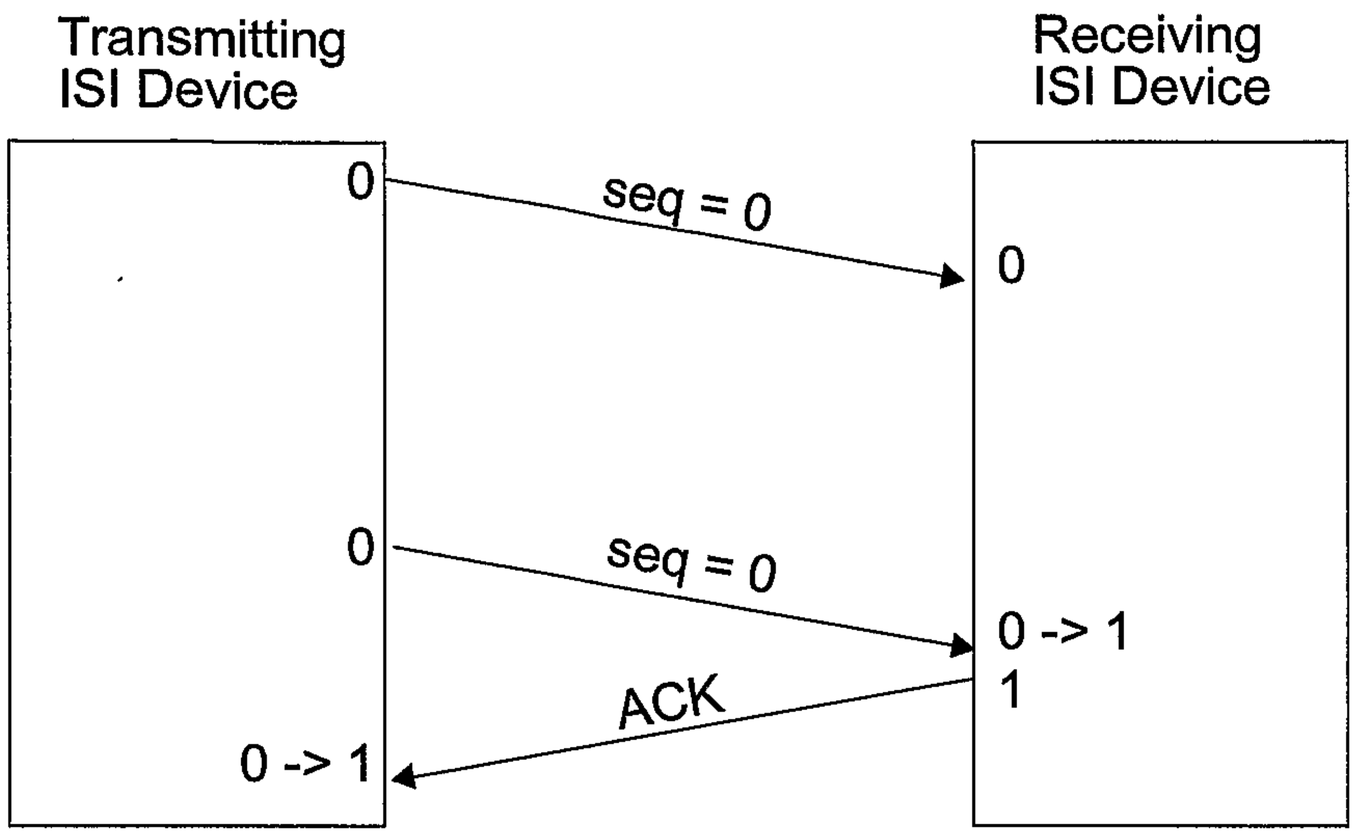

Transmitting ISI Device
Receiving ISI Device
0
seq = 0
0
0
seq = 0
0 -> 1
1
ACK
0 -> 1

FIG. 37

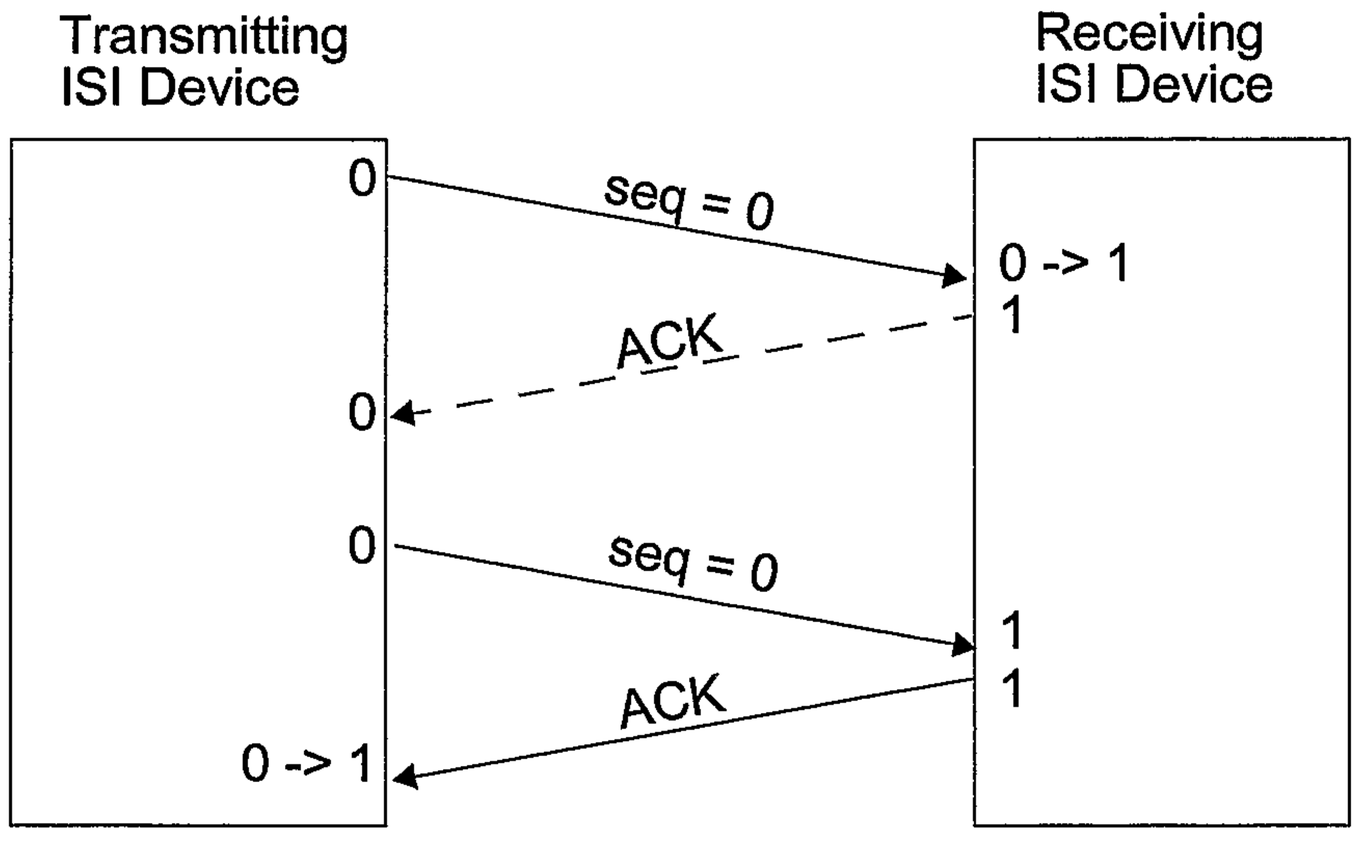

Transmitting ISI Device
Receiving ISI Device
0
seq = 0
0 -> 1
1
ACK
0
seq = 0
1
1
ACK
0 -> 1

FIG. 38

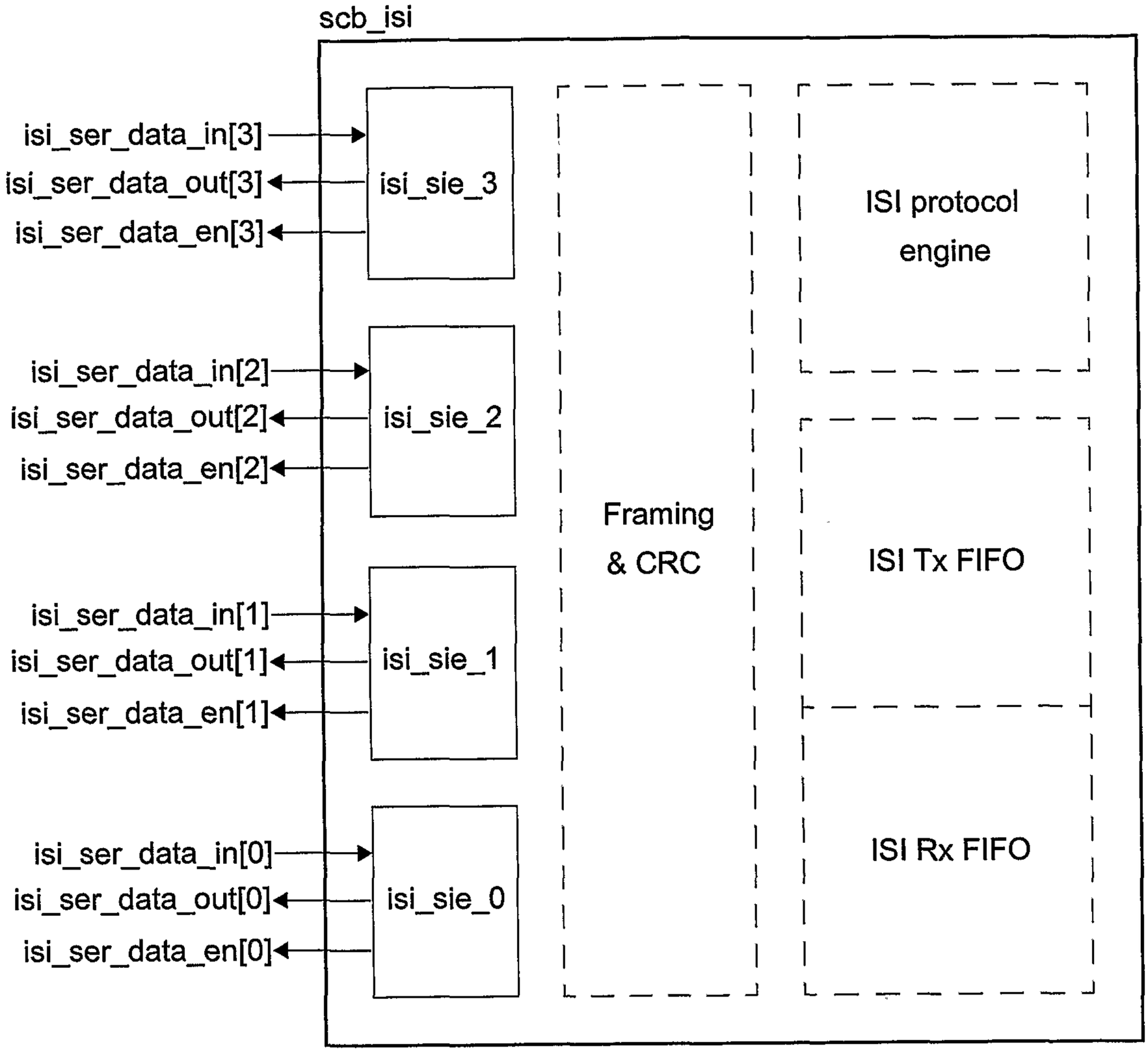

scb_isi
isi_ser_data_in[3]
isi_ser_data_out[3]
isi_ser_data_en[3]
isi_sie_3
isi_ser_data_in[2]
isi_ser_data_out[2]
isi_ser_data_en[2]
isi_sie_2
isi_ser_data_in[1]
isi_ser_data_out[1]
isi_ser_data_en[1]
isi_sie_1
isi_ser_data_in[0]
isi_ser_data_out[0]
isi_ser_data_en[0]
isi_sie_0
Framing
& CRC
ISI protocol
engine
ISI Tx FIFO
ISI Rx FIFO

FIG. 39

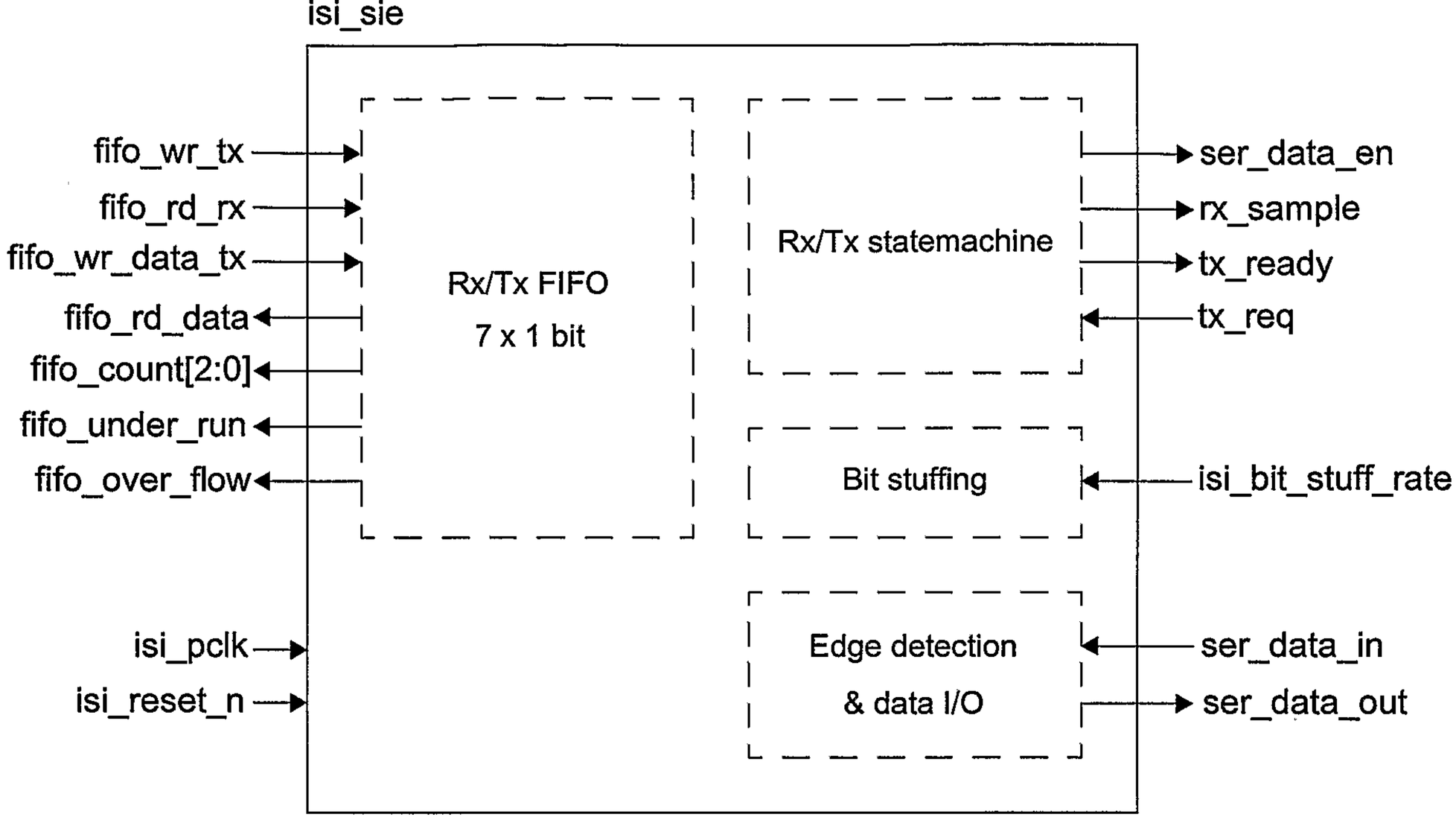

isi_sie
fifo_wr_tx
fifo_rd_rx
fifo_wr_data_tx
fifo_rd_data
fifo_count[2:0]
fifo_under_run
fifo_over_flow
Rx/Tx FIFO
7 x 1 bit
Rx/Tx statemachine
Bit stuffing
Edge detection
& data I/O
ser_data_en
rx_sample
tx_ready
tx_req
isi_bit_stuff_rate
ser_data_in
ser_data_out
isi_pclk
isi_reset_n

FIG. 40

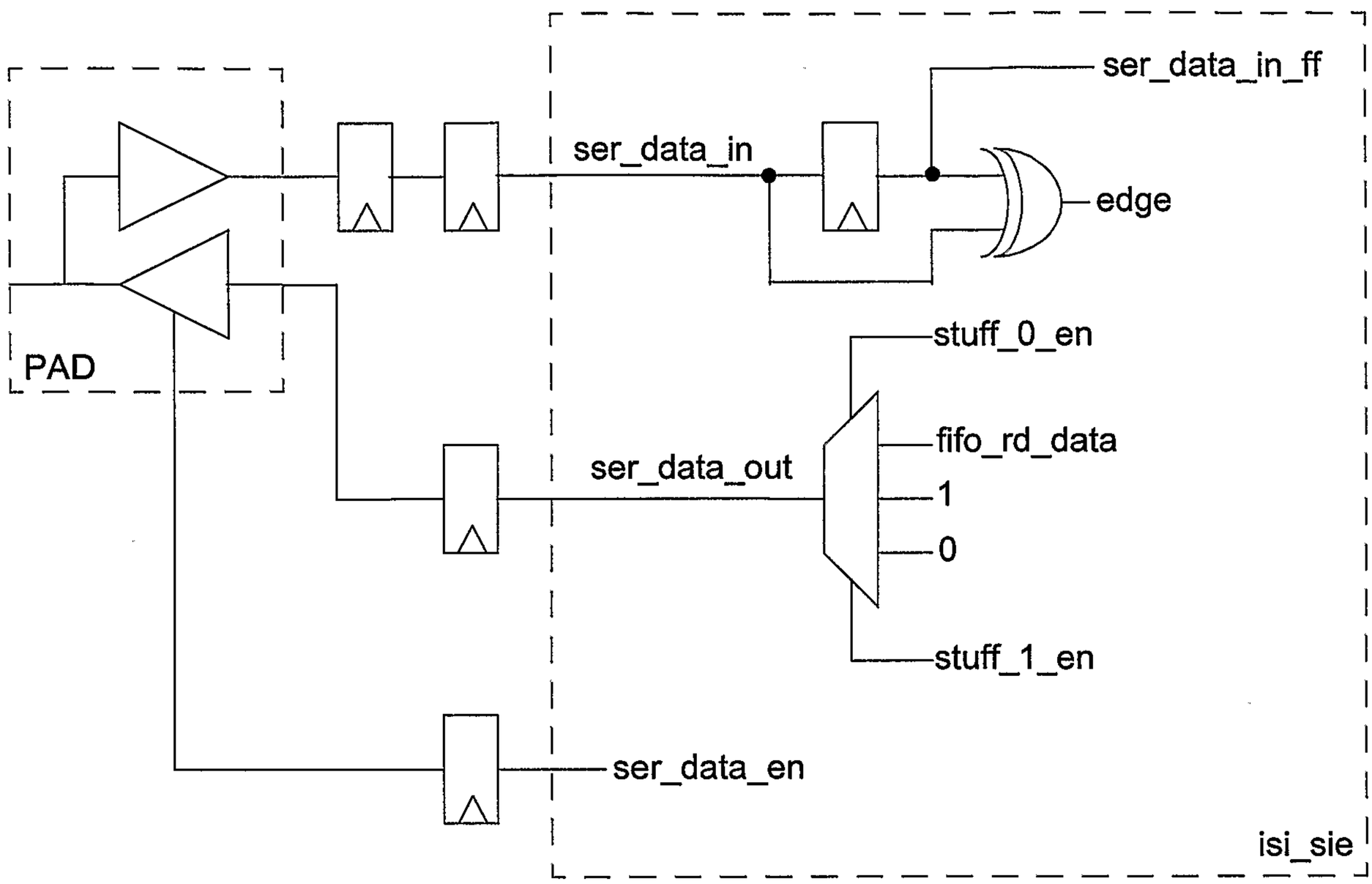

ser_data_in_ff
ser_data_in
edge
PAD
stuff_0_en
fifo_rd_data
ser_data_out
1
0
stuff_1_en
ser_data_en
isi_sie

FIG. 41

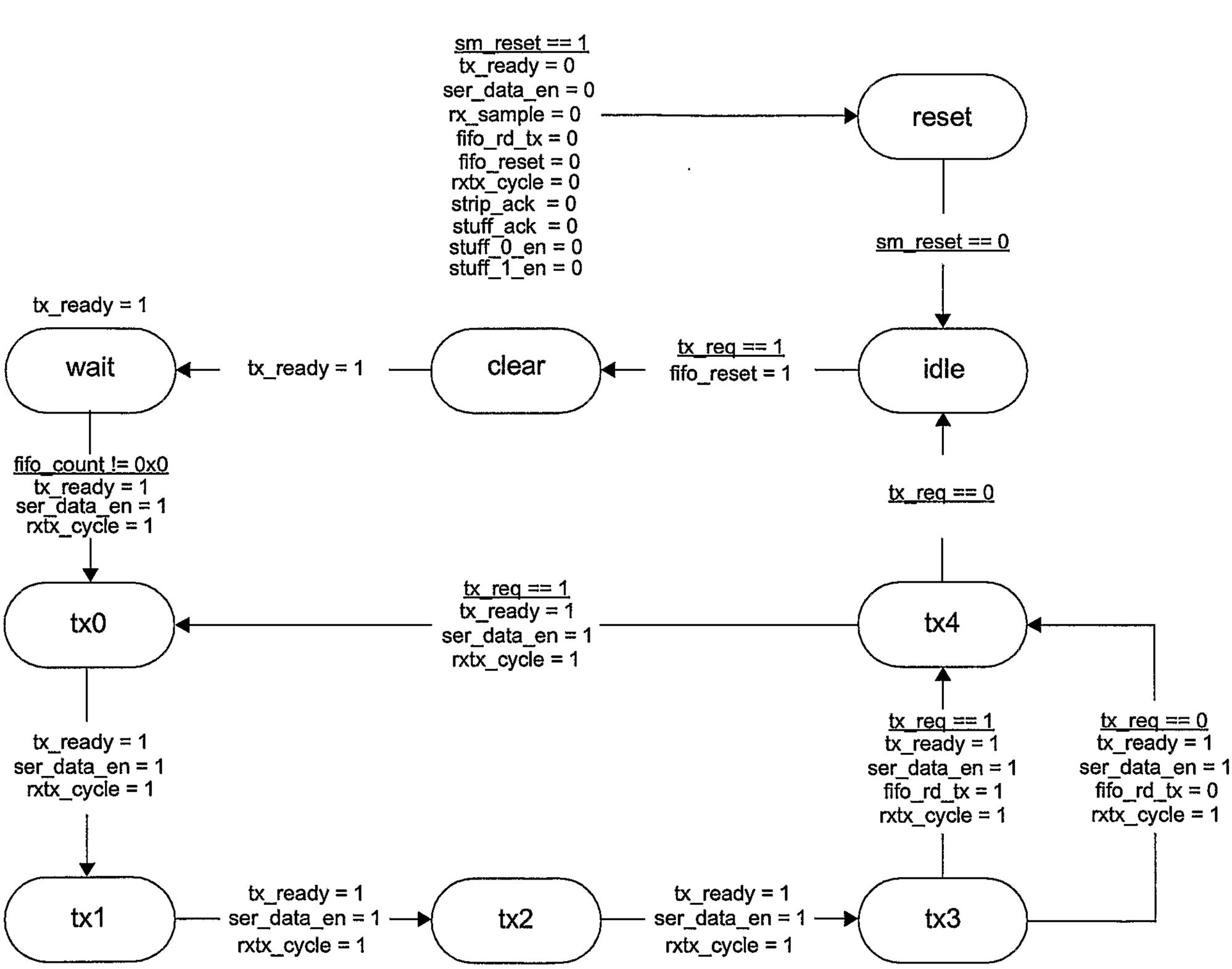

sm_reset == 1
tx_ready = 0
ser_data_en = 0
rx_sample = 0
fifo_rd_tx = 0
fifo_reset = 0
rxtx_cycle = 0
strip_ack = 0
stuff_ack = 0
stuff_0_en = 0
stuff_1_en = 0
reset
sm_reset == 0
idle
tx_req == 1
fifo_reset = 1
clear
tx_ready = 1
wait
tx_ready = 1
fifo_count != 0x0
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
tx_req == 0
tx4
tx_req == 1
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
tx0
tx_req == 1
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
tx_req == 1
tx_ready = 1
ser_data_en = 1
fifo_rd_tx = 1
rxtx_cycle = 1
tx_req == 0
tx_ready = 1
ser_data_en = 1
fifo_rd_tx = 0
rxtx_cycle = 1
tx1
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
tx2
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
tx3

FIG. 42

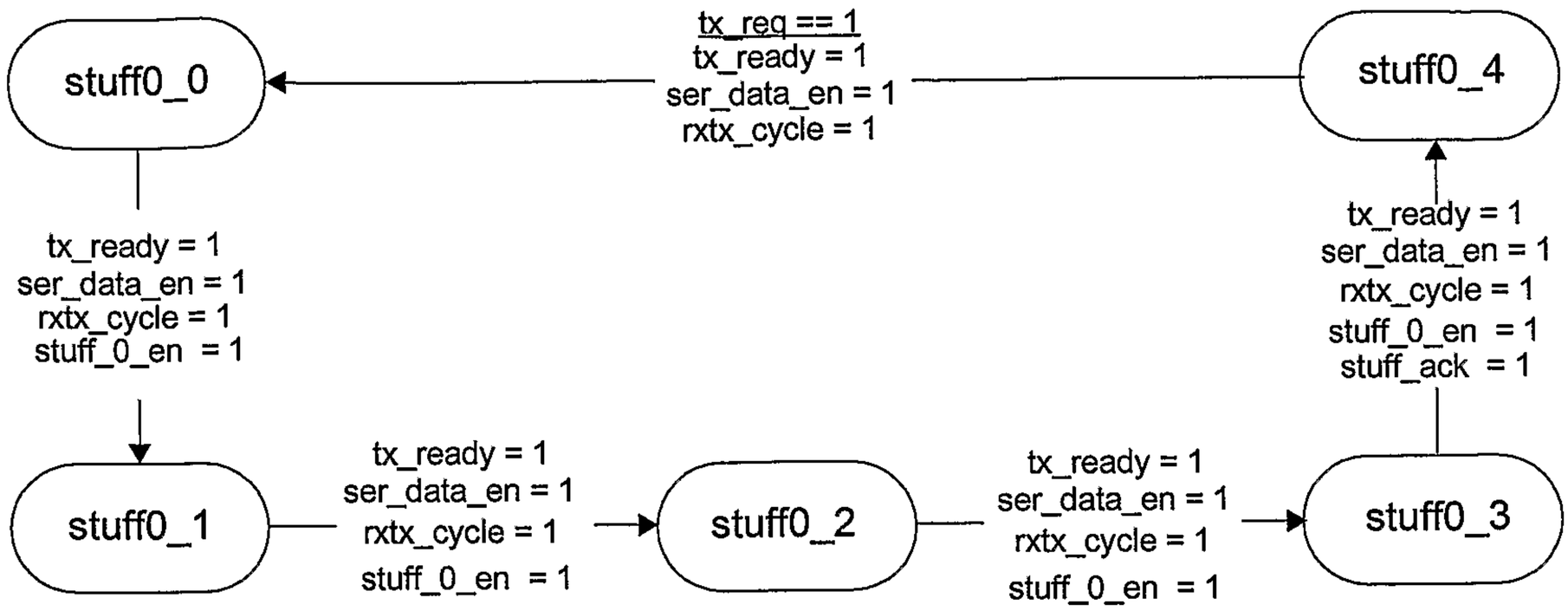

stuff0_0
tx_req == 1
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff0_4
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff_0_en = 1
stuff_ack = 1
stuff0_1
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff_0_en = 1
stuff0_2
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff_0_en = 1
stuff0_3

FIG. 44

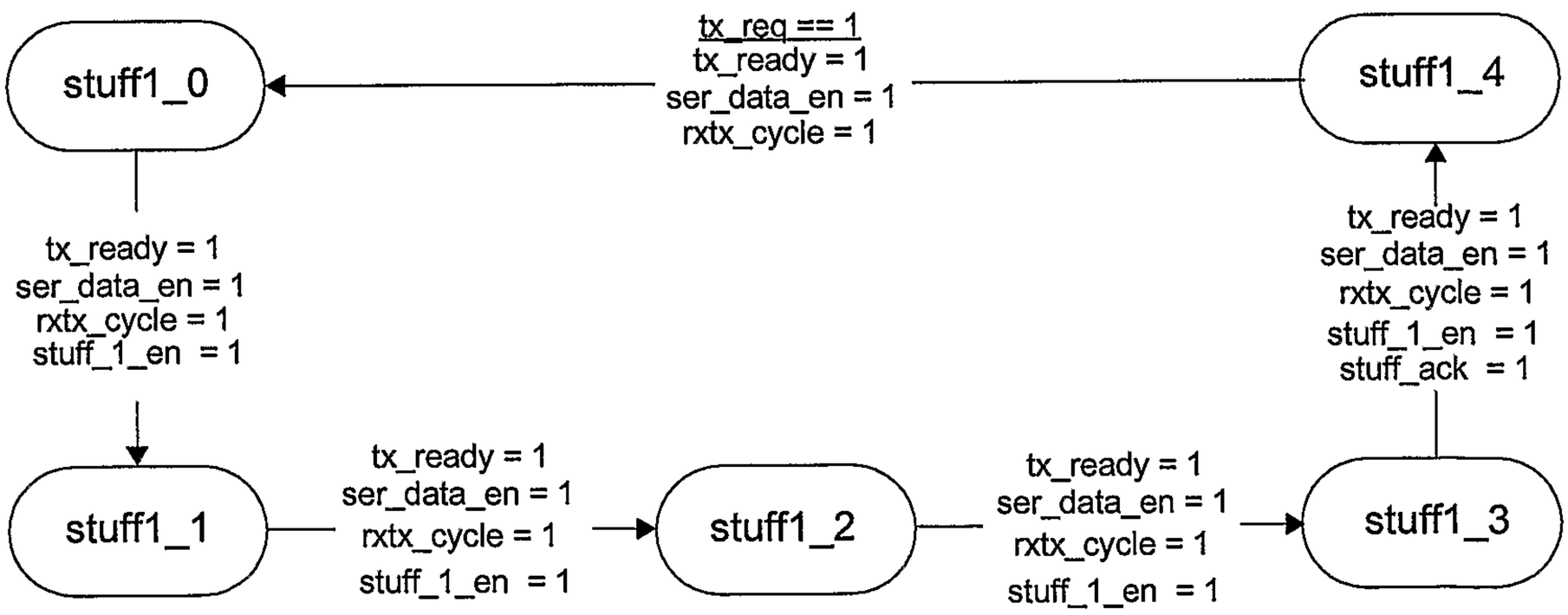

stuff1_0
tx_req == 1
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff1_4
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff_1_en = 1
stuff_ack = 1
stuff1_1
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff_1_en = 1
stuff1_2
tx_ready = 1
ser_data_en = 1
rxtx_cycle = 1
stuff_1_en = 1
stuff1_3

FIG. 44

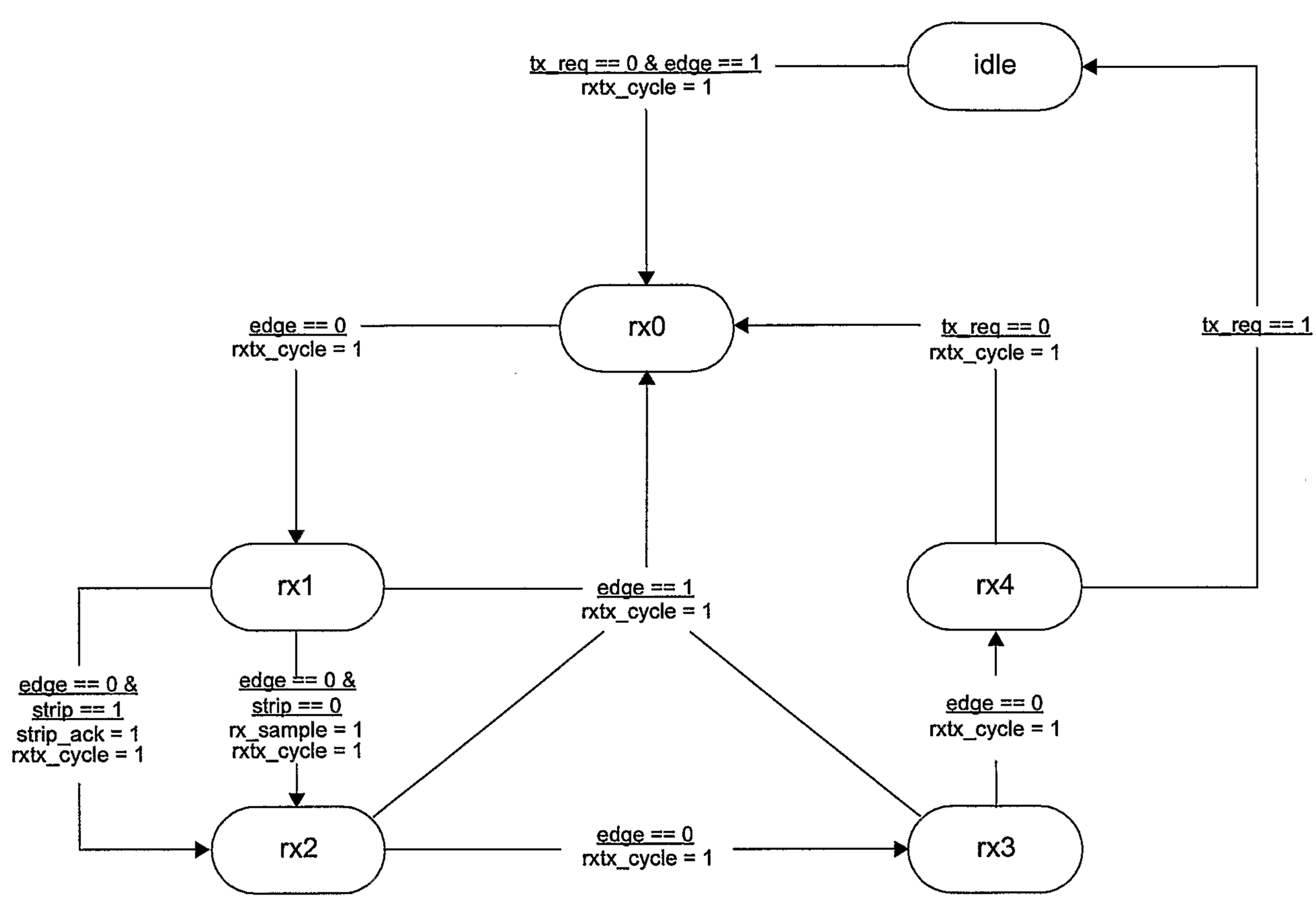

idle
tx_req == 0 & edge == 1
rxtx_cycle = 1
rx0
edge == 0
rxtx_cycle = 1
tx_req == 0
rxtx_cycle = 1
tx_req == 1
rx1
edge == 1
rxtx_cycle = 1
rx4
edge == 0 &
strip == 1
strip_ack = 1
rxtx_cycle = 1
edge == 0 &
strip == 0
rx_sample = 1
rxtx_cycle = 1
edge == 0
rxtx_cycle = 1
rx2
edge == 0
rxtx_cycle = 1
rx3

FIG. 45

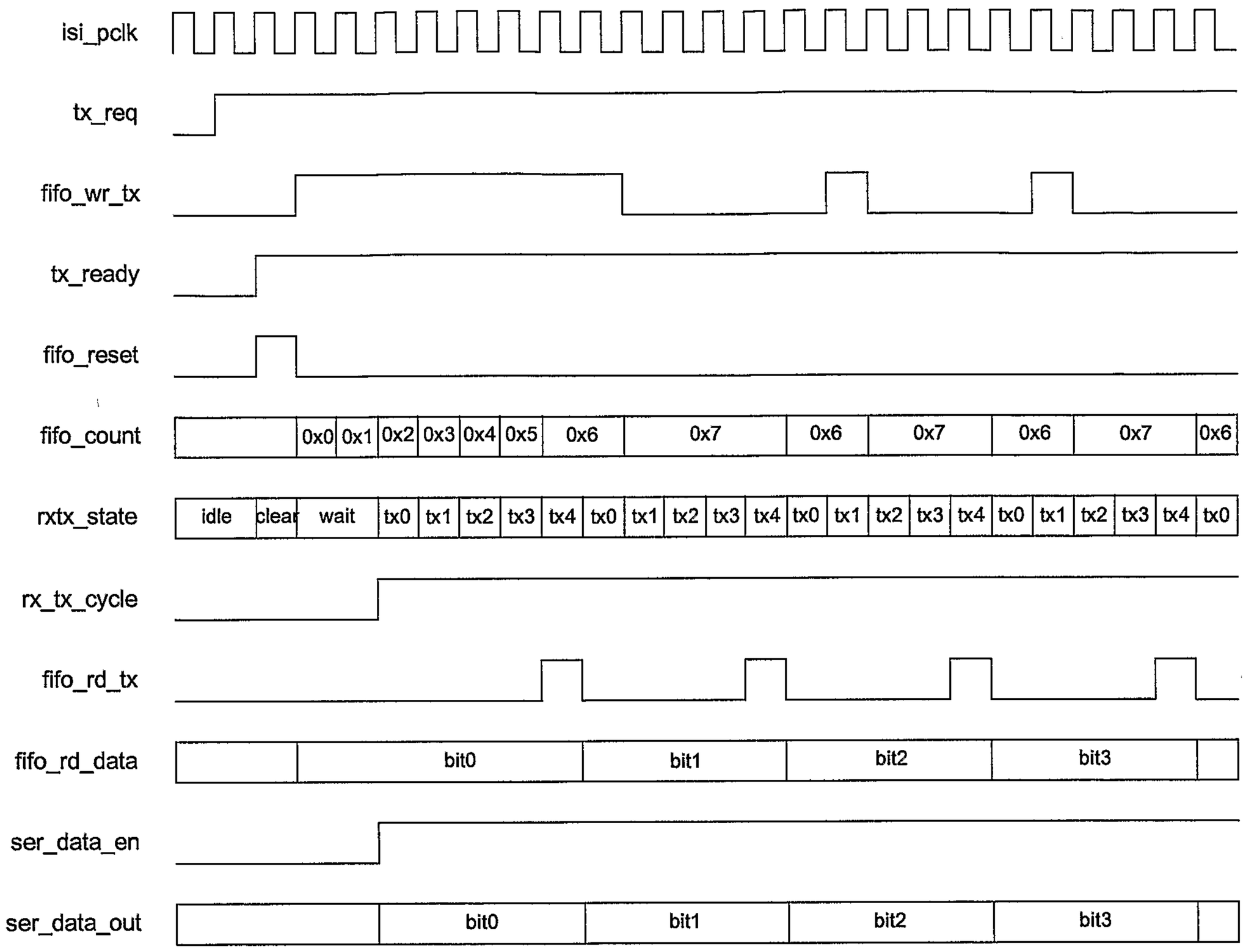

isi_pclk
tx_req
fifo_wr_tx
tx_ready
fifo_reset
fifo_count
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x6 0x7 0x6 0x7 0x6
rxtx_state
idle clear wait tx0 tx1 tx2 tx3 tx4 tx0 tx1 tx2 tx3 tx4 tx0 tx1 tx2 tx3 tx4 tx0 tx1 tx2 tx3 tx4 tx0
rx_tx_cycle
fifo_rd_tx
fifo_rd_data
bit0 bit1 bit2 bit3
ser_data_en
ser_data_out
bit0 bit1 bit2 bit3

FIG. 46

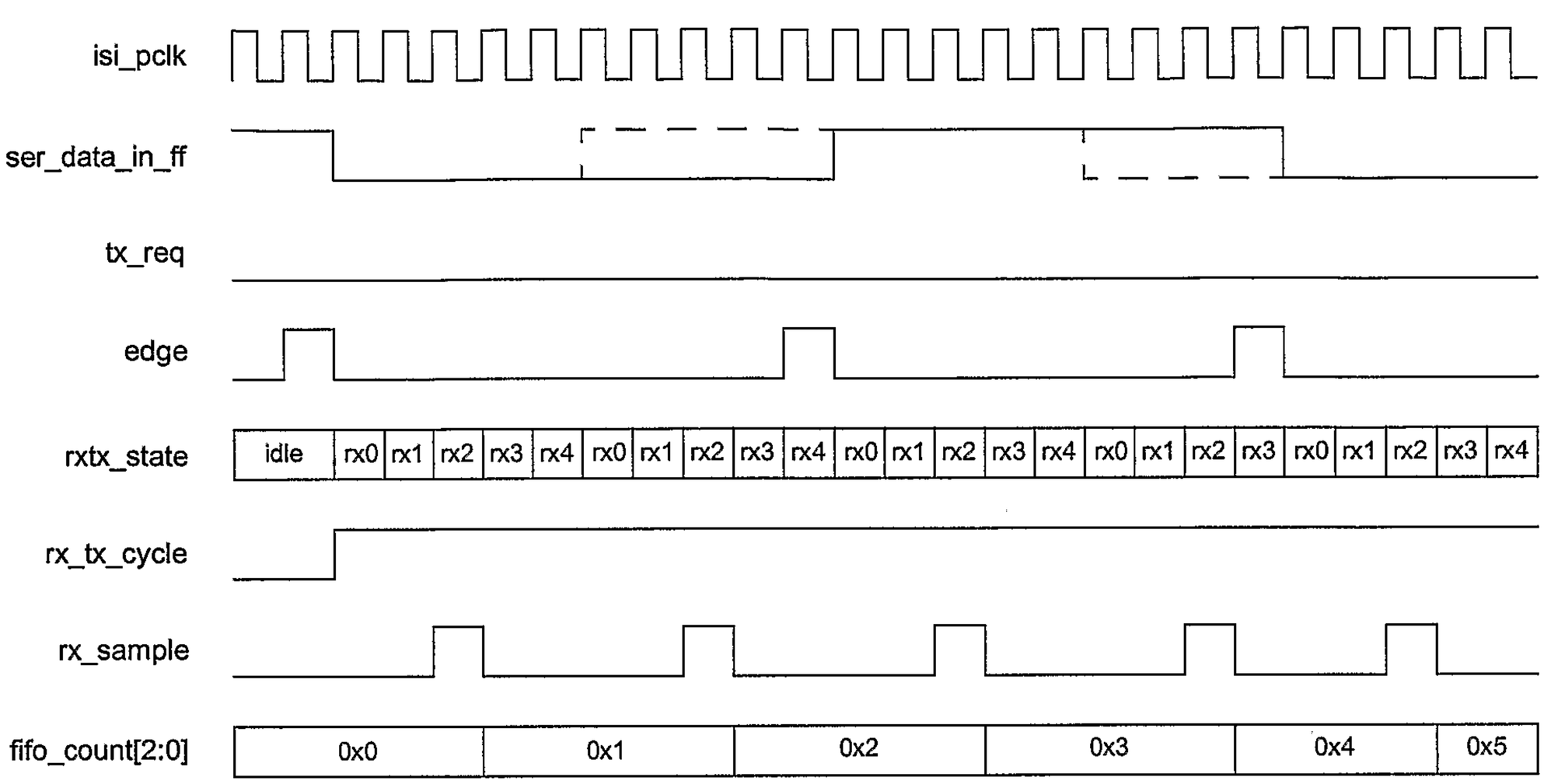

isi_pclk
ser_data_in_ff
tx_req
edge
rxtx_state
idle
rx0
rx1
rx2
rx3
rx4
rx_tx_cycle
rx_sample
fifo_count[2:0]
0x0
0x1
0x2
0x3
0x4
0x5

FIG. 47

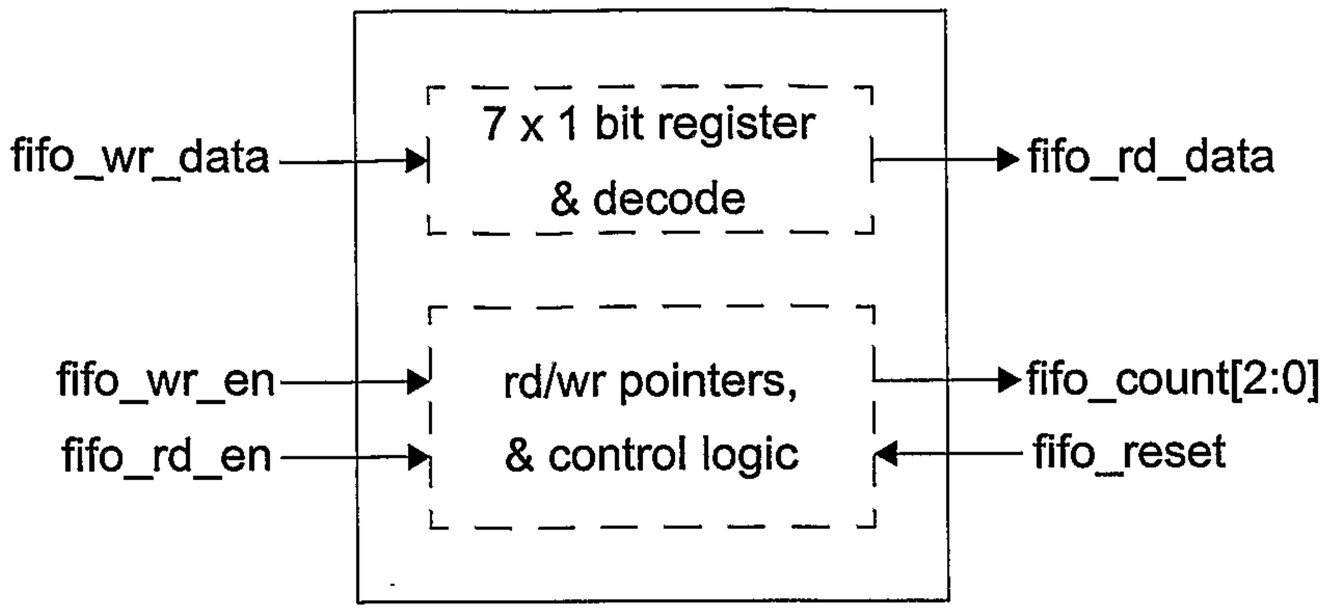

7 x 1 bit register
& decode
rd/wr pointers,
& control logic
fifo_wr_data
fifo_wr_en
fifo_rd_en
fifo_rd_data
fifo_count[2:0]
fifo_reset

FIG. 48

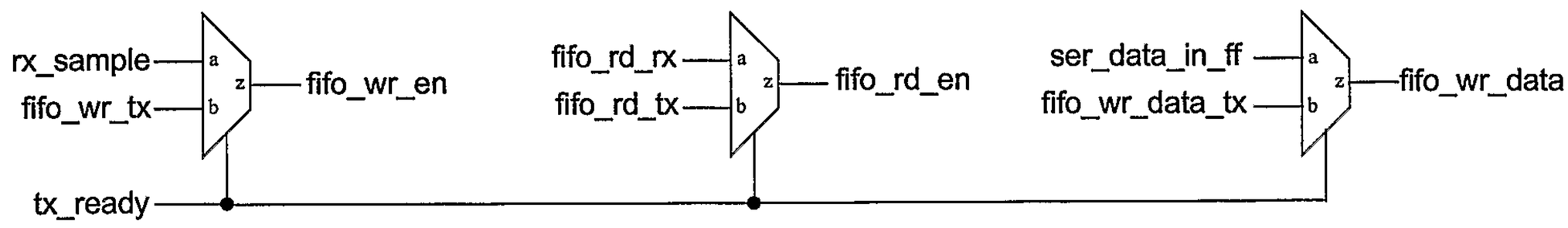

rx_sample
fifo_wr_tx
a
b
z
fifo_wr_en
tx_ready
fifo_rd_rx
fifo_rd_tx
fifo_rd_en
ser_data_in_ff
fifo_wr_data_tx
fifo_wr_data

FIG. 49

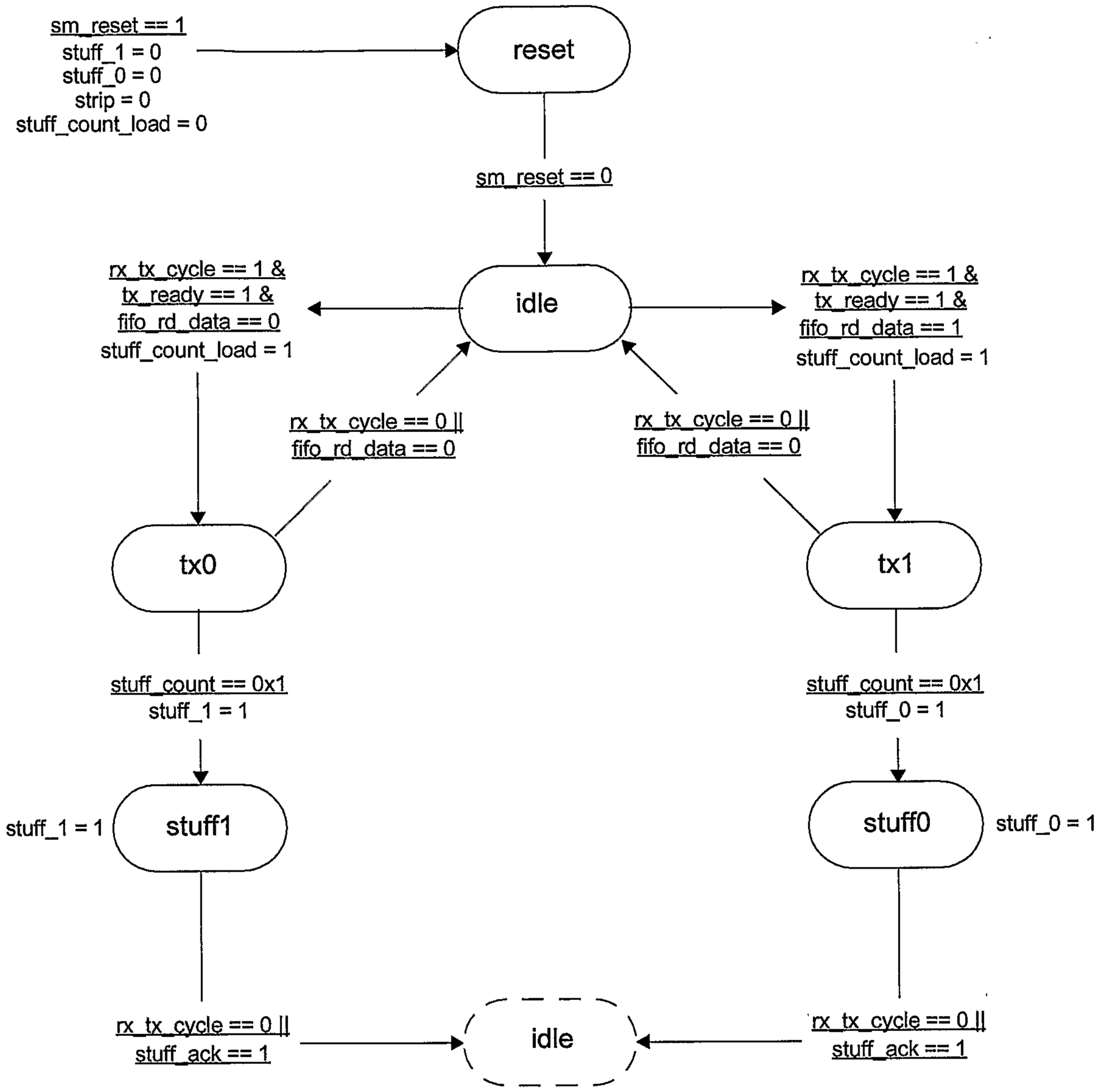

sm_reset == 1
stuff_1 = 0
stuff_0 = 0
strip = 0
stuff_count_load = 0
reset
sm_reset == 0
idle
rx_tx_cycle == 1 &
tx_ready == 1 &
fifo_rd_data == 0
stuff_count_load = 1
rx_tx_cycle == 1 &
tx_ready == 1 &
fifo_rd_data == 1
stuff_count_load = 1
rx_tx_cycle == 0 ||
fifo_rd_data == 0
rx_tx_cycle == 0 ||
fifo_rd_data == 0
tx0
tx1
stuff_count == 0x1
stuff_1 = 1
stuff_count == 0x1
stuff_0 = 1
stuff_1 = 1
stuff1
stuff0
stuff_0 = 1
rx_tx_cycle == 0 ||
stuff_ack == 1
idle
rx_tx_cycle == 0 ||
stuff_ack == 1

FIG. 50

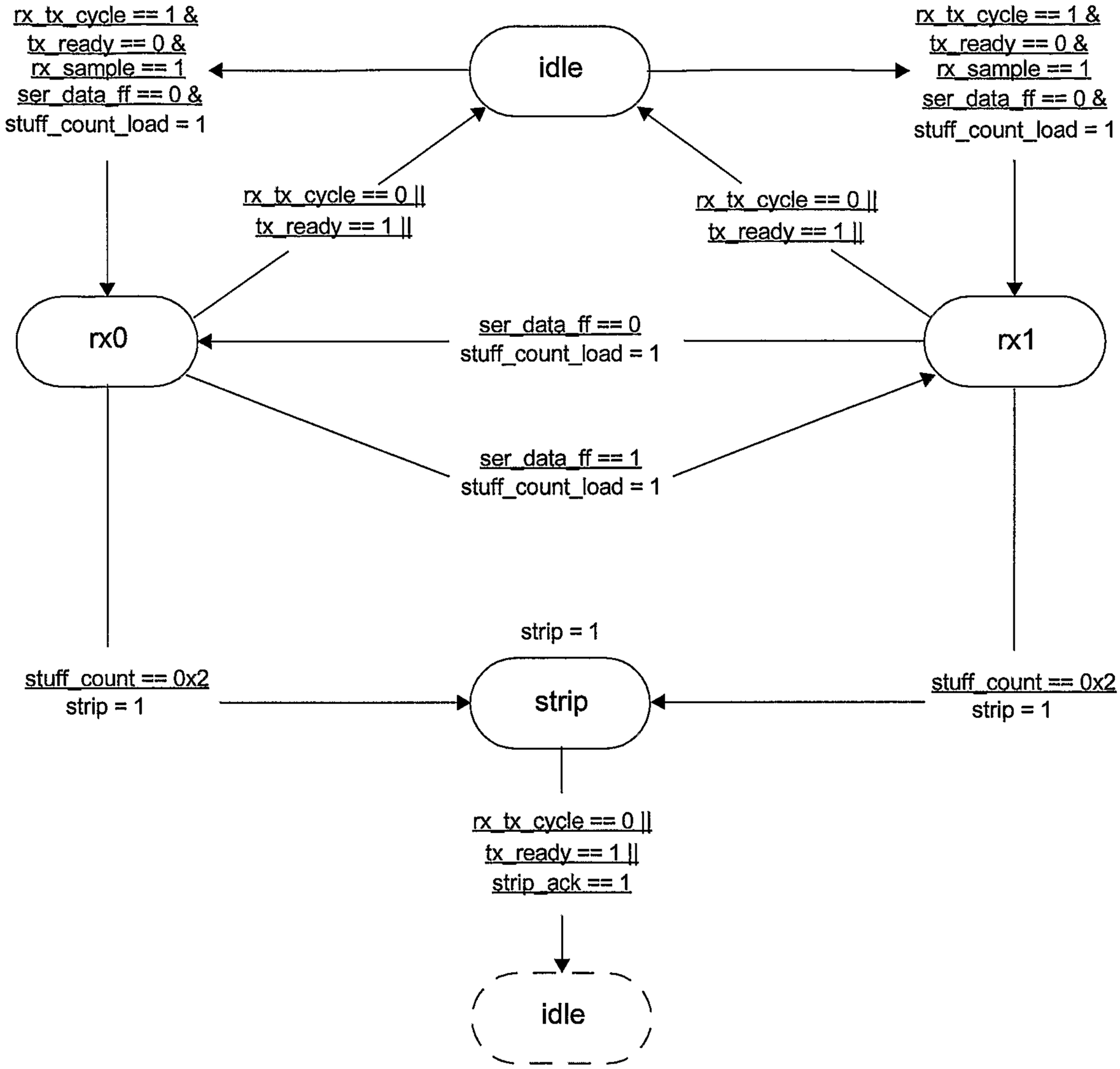

idle
rx_tx_cycle == 1 &
tx_ready == 0 &
rx_sample == 1
ser_data_ff == 0 &
stuff_count_load = 1
rx_tx_cycle == 1 &
tx_ready == 0 &
rx_sample == 1
ser_data_ff == 0 &
stuff_count_load = 1
rx_tx_cycle == 0 ||
tx_ready == 1 ||
rx_tx_cycle == 0 ||
tx_ready == 1 ||
rx0
rx1
ser_data_ff == 0
stuff_count_load = 1
ser_data_ff == 1
stuff_count_load = 1
strip = 1
stuff_count == 0x2
strip = 1
strip
stuff_count == 0x2
strip = 1
rx_tx_cycle == 0 ||
tx_ready == 1 ||
strip_ack == 1
idle

FIG. 51

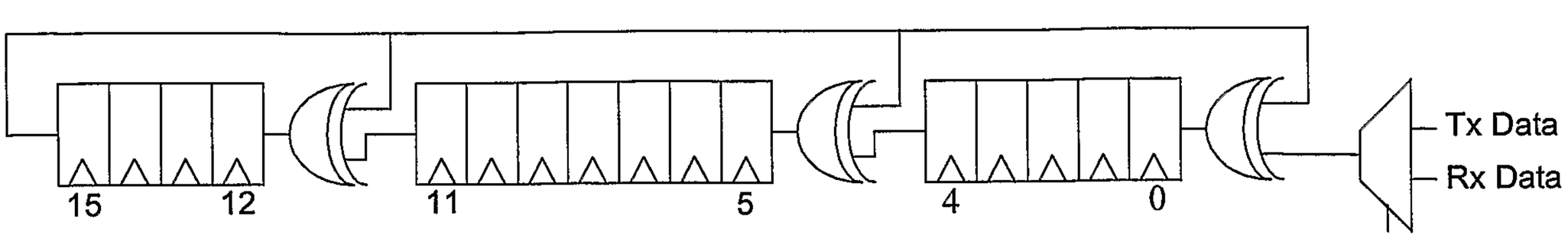

15
12
11
5
4
0
Tx Data
Rx Data

FIG. 52

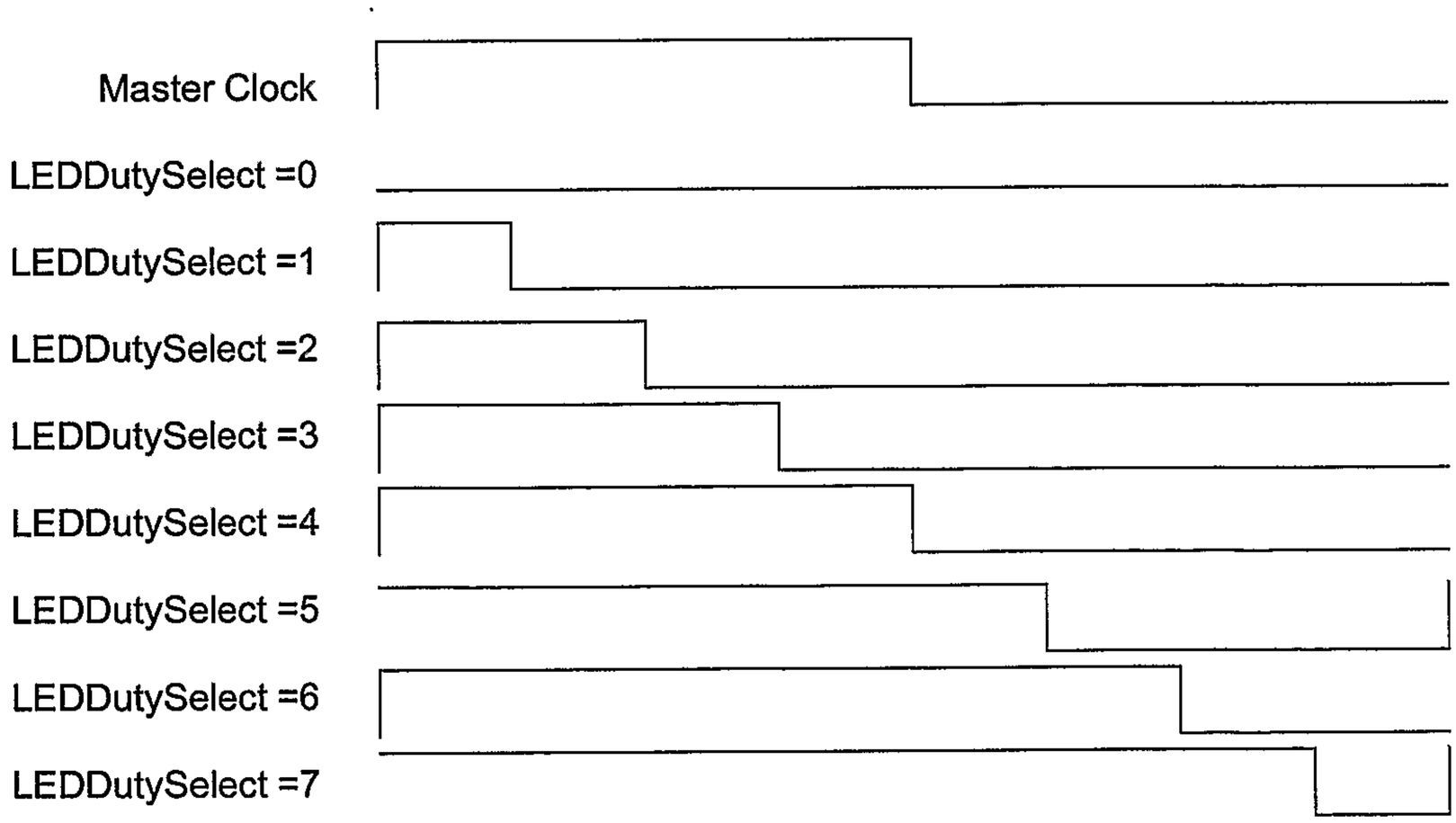

Master Clock
LEDDutySelect =0
LEDDutySelect =1
LEDDutySelect =2
LEDDutySelect =3
LEDDutySelect =4
LEDDutySelect =5
LEDDutySelect =6
LEDDutySelect =7

FIG. 54

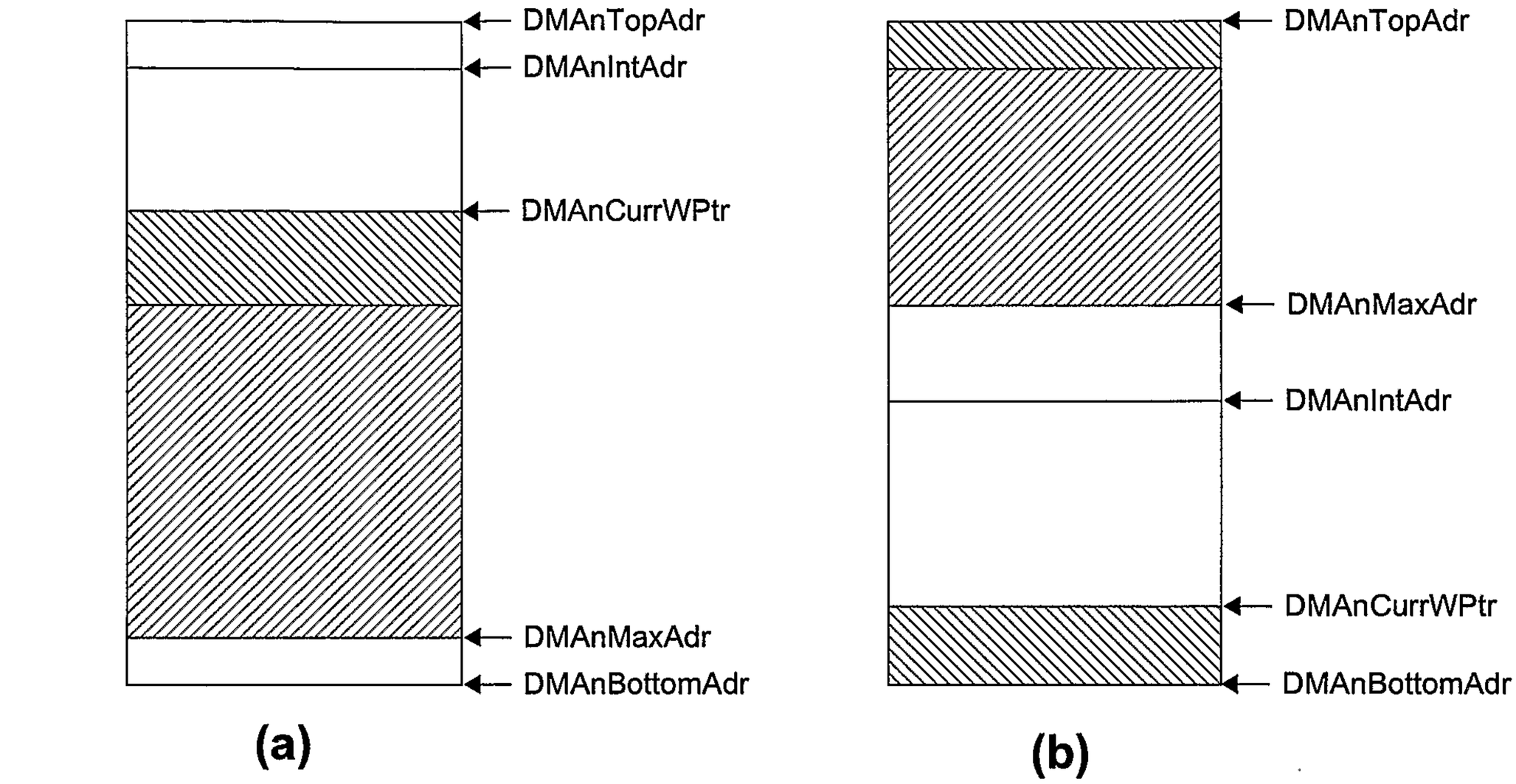

DMAnTopAdr
DMAnIntAdr
DMAnCurrWPtr
DMAnMaxAdr
DMAnBottomAdr
(a)
DMAnTopAdr
DMAnMaxAdr
DMAnIntAdr
DMAnCurrWPtr
DMAnBottomAdr
(b)
Key:
Free buffer space
Filled buffer space (unprocessed data)
Buffer space filled since last write to the DMAnIntAdr/DMAnMaxAdr registers

FIG. 53

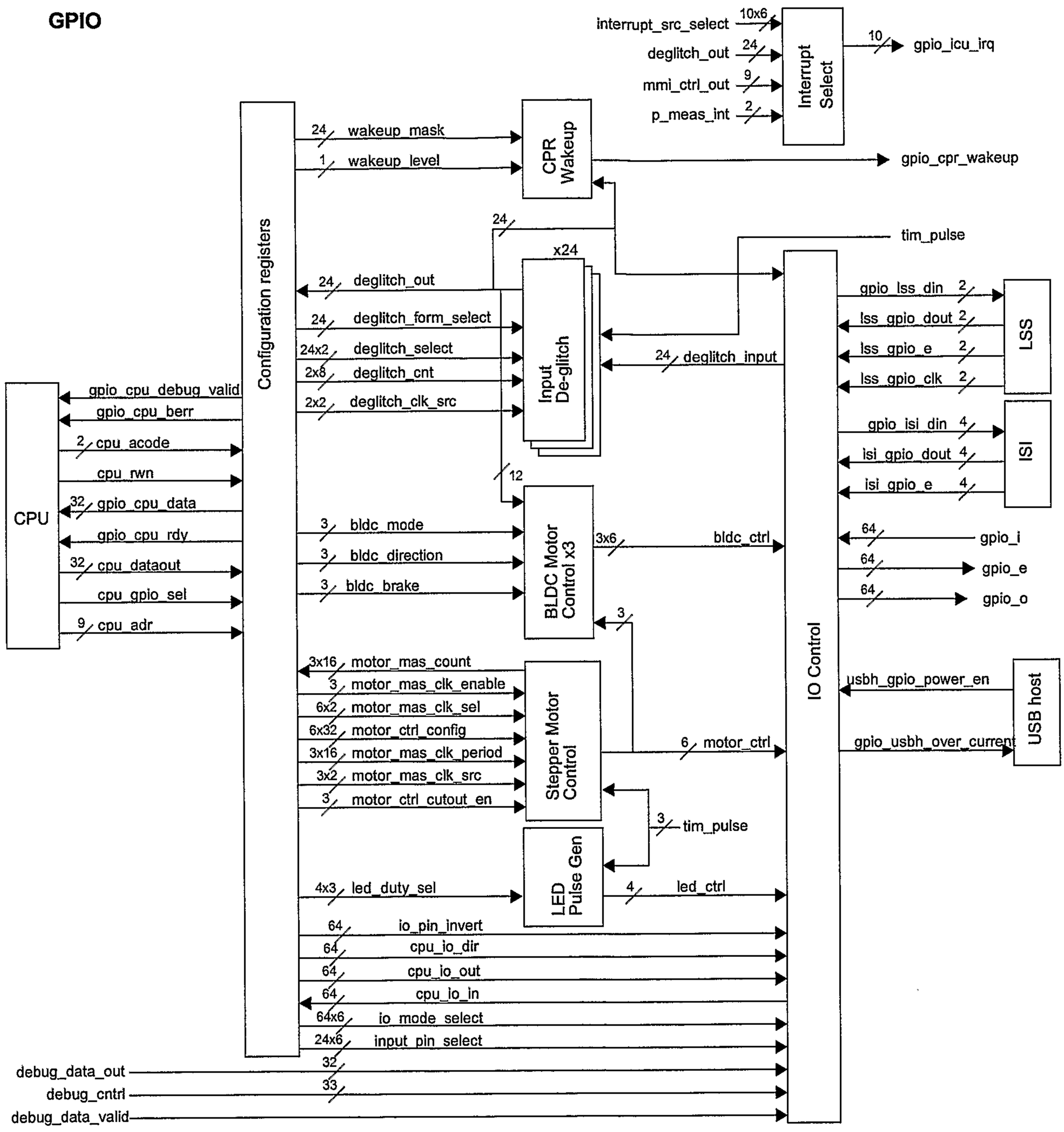

GPIO
CPU
gpio_cpu_debug_valid
gpio_cpu_berr
2 cpu_acode
cpu_rwn
32 gpio_cpu_data
gpio_cpu_rdy
32 cpu_dataout
cpu_gpio_sel
9 cpu_adr
Configuration registers
24 wakeup_mask
1 wakeup_level
CPR Wakeup
gpio_cpr_wakeup
24
x24
24 deglitch_out
24 deglitch_form_select
24x2 deglitch_select
2x8 deglitch_cnt
2x2 deglitch_clk_src
Input De-glitch
24 deglitch_input
tim_pulse
12
3 bldc_mode
3 bldc_direction
3 bldc_brake
BLDC Motor Control x3
3x6
bldc_ctrl
3
3x16 motor_mas_count
3 motor_mas_clk_enable
6x2 motor_mas_clk_sel
6x32 motor_ctrl_config
3x16 motor_mas_clk_period
3x2 motor_mas_clk_src
3 motor_ctrl_cutout_en
Stepper Motor Control
6 motor_ctrl
3 tim_pulse
LED Pulse Gen
4x3 led_duty_sel
4 led_ctrl
64 io_pin_invert
64 cpu_io_dir
64 cpu_io_out
64 cpu_io_in
64x6 io_mode_select
24x6 input_pin_select
32
33
debug_data_out
debug_cntrl
debug_data_valid
IO Control
interrupt_src_select 10x6
deglitch_out 24
mmi_ctrl_out 9
p_meas_int 2
Interrupt Select
10 gpio_icu_irq
gpio_lss_din 2
lss_gpio_dout 2
lss_gpio_e 2
lss_gpio_clk 2
LSS
gpio_isi_din 4
isi_gpio_dout 4
isi_gpio_e 4
ISI
64 gpio_i
64 gpio_e
64 gpio_o
usbh_gpio_power_en
gpio_usbh_over_current
USB host

FIG. 55

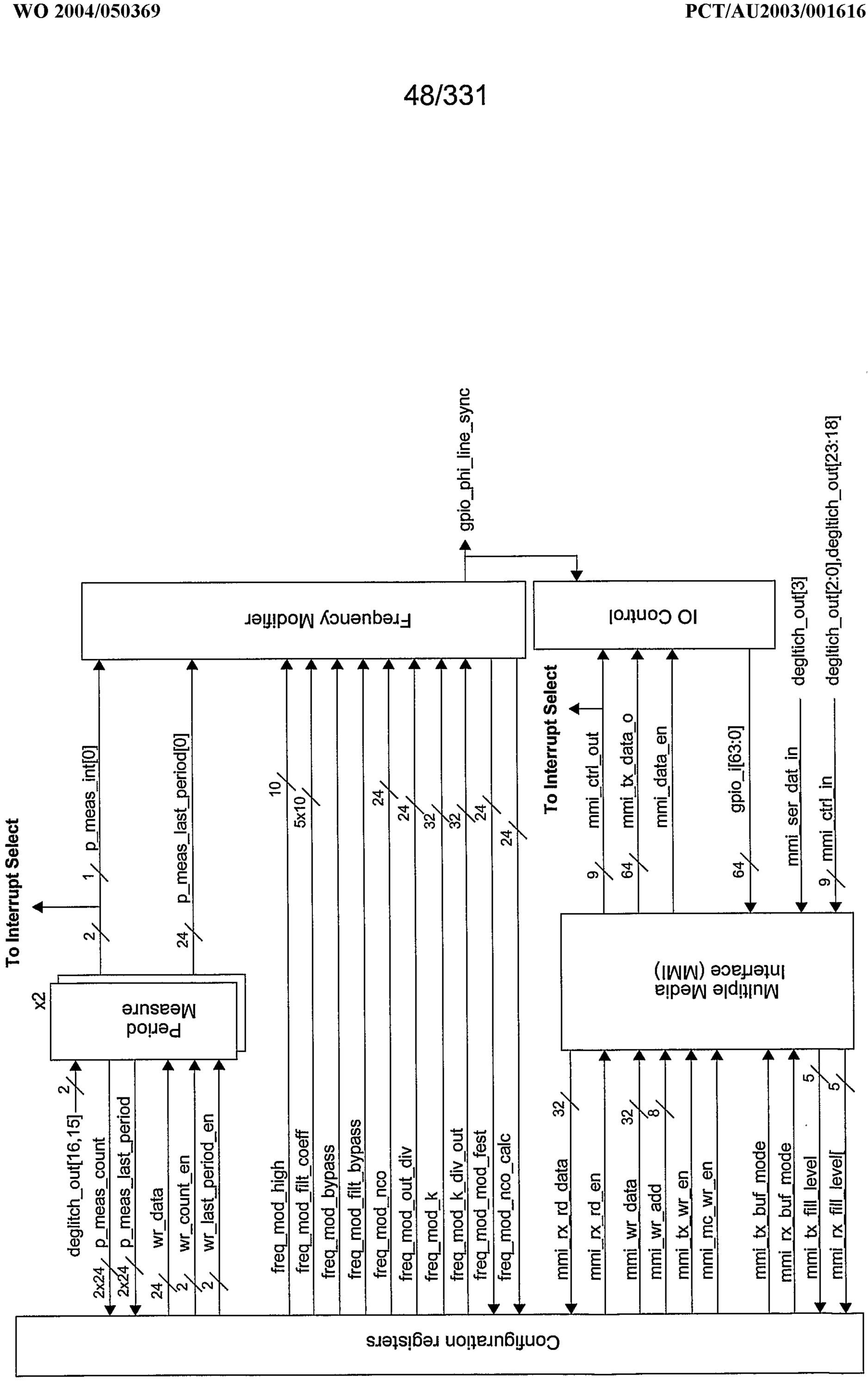

Configuration registers
deglitch_out[16,15]
p_meas_count
p_meas_last_period
wr_data
wr_count_en
wr_last_period_en
freq_mod_high
freq_mod_filt_coeff
freq_mod_bypass
freq_mod_filt_bypass
freq_mod_nco
freq_mod_out_div
freq_mod_k
freq_mod_k_div_out
freq_mod_mod_fest
freq_mod_nco_calc
mmi_rx_rd_data
mmi_rx_rd_en
mmi_wr_data
mmi_wr_add
mmi_tx_wr_en
mmi_mc_wr_en
mmi_tx_buf_mode
mmi_rx_buf_mode
mmi_tx_fill_level
mmi_rx_fill_level[
Multiple Media Interface (MMI)
mmi_ctrl_out
mmi_tx_data_o
mmi_data_en
gpio_i[63:0]
mmi_ser_dat_in
mmi_ctrl_in
To Interrupt Select
IO Control
gpio_phi_line_sync
Frequency Modifier
deglitch_out[3]
deglitch_out[2:0],deglitch_out[23:18]
Period Measure
x2
p_meas_int[0]
p_meas_last_period[0]
To Interrupt Select

FIG. 56

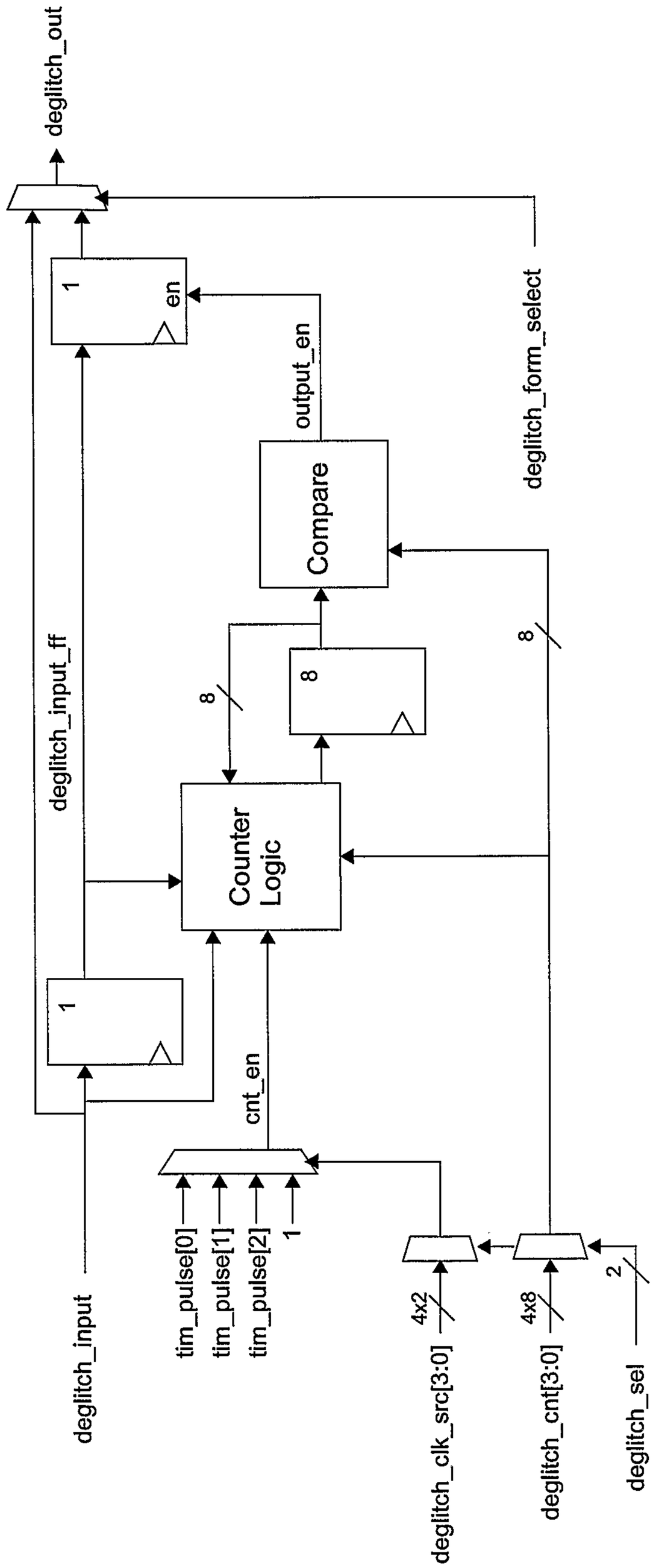

deglitch_out
1
en
output_en
deglitch_form_select
Compare
deglitch_input_ff
8
8
Counter Logic
1
cnt_en
deglitch_input
tim_pulse[0]
tim_pulse[1]
tim_pulse[2]
1
4x2
deglitch_clk_src[3:0]
4x8
deglitch_cnt[3:0]
2
deglitch_sel

FIG. 57

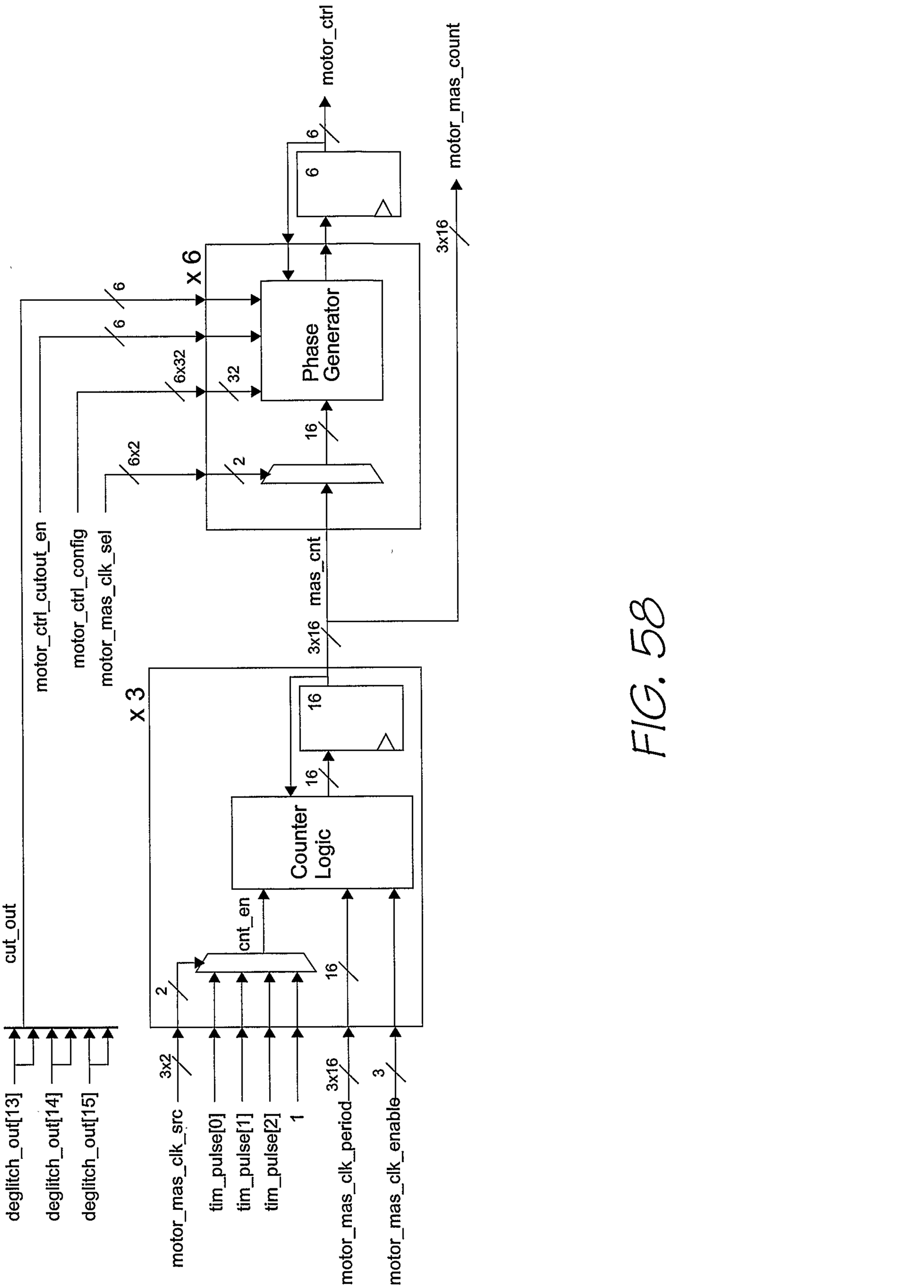

motor_ctrl
motor_mas_count
6
6
x 6
Phase Generator
6x32
32
16
6x2
2
3x16
motor_ctrl_cutout_en
motor_ctrl_config
motor_mas_clk_sel
mas_cnt
x 3
16
Counter Logic
cut_out
deglitch_out[13]
deglitch_out[14]
deglitch_out[15]
motor_mas_clk_src
tim_pulse[0]
tim_pulse[1]
tim_pulse[2]
1
motor_mas_clk_period
motor_mas_clk_enable
3x2
3x16
3

FIG. 58

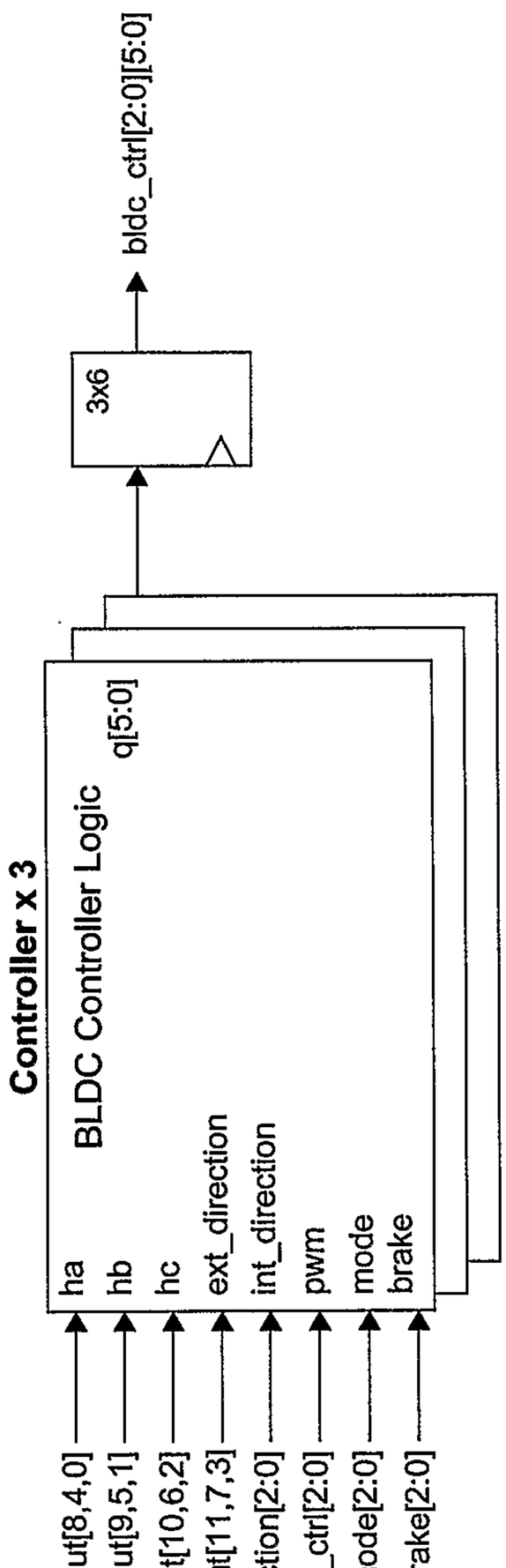

Controller x 3
BLDC Controller Logic
ha
hb
hc
ext_direction
int_direction
pwm
mode
brake
q[5:0]
3x6
bldc_ctrl[2:0][5:0]
deglitch_out[8,4,0]
deglitch_out[9,5,1]
deglitch_out[10,6,2]
deglitch_out[11,7,3]
bldc_direction[2:0]
motor_ctrl[2:0]
bldc_mode[2:0]
bldc_brake[2:0]

FIG. 59

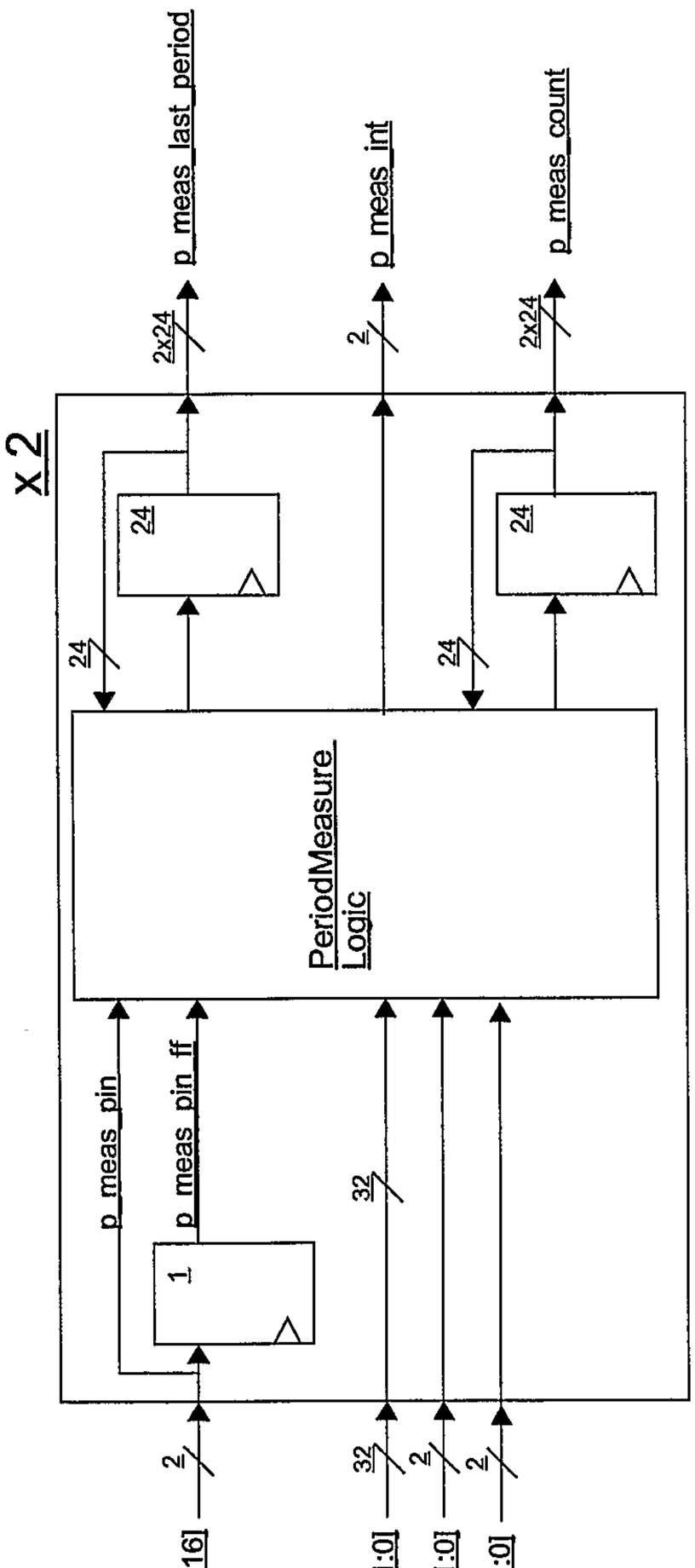

x2
PeriodMeasure Logic
p_meas_last_period
p_meas_int
p_meas_count
p_meas_pin
p_meas_pin_ff
deglitch_out[17,16]
wr_data[31:0]
wr_last_period_en[1:0]
wr_count_en[1:0]

FIG. 60

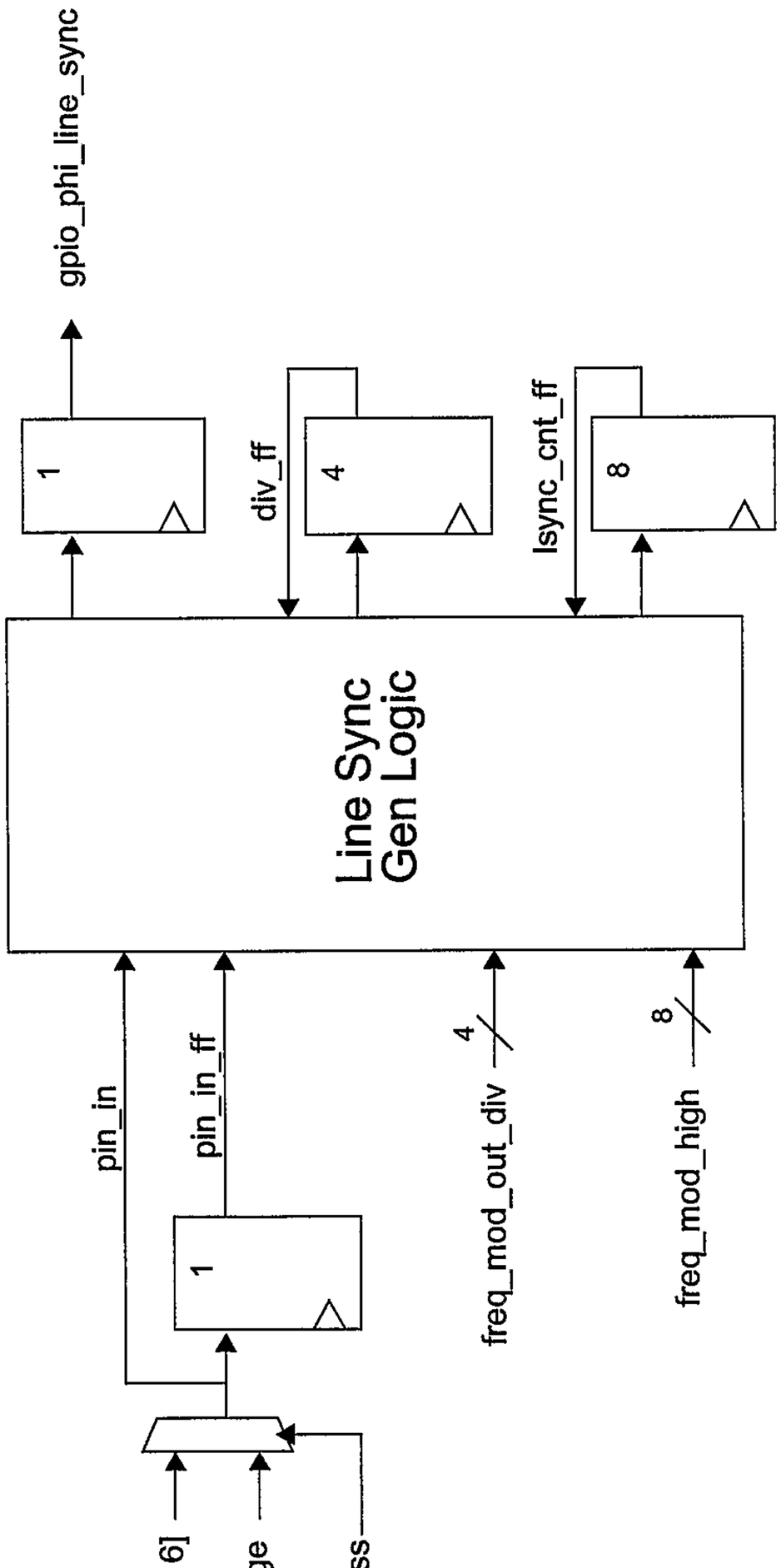

gpio_phi_line_sync
1
div_ff
4
lsync_cnt_ff
8
Line Sync
Gen Logic
pin_in
pin_in_ff
1
freq_mod_out_div
4
freq_mod_high
8
deglitch_out[16]
nco_edge
freq_mod_bypass

FIG. 61

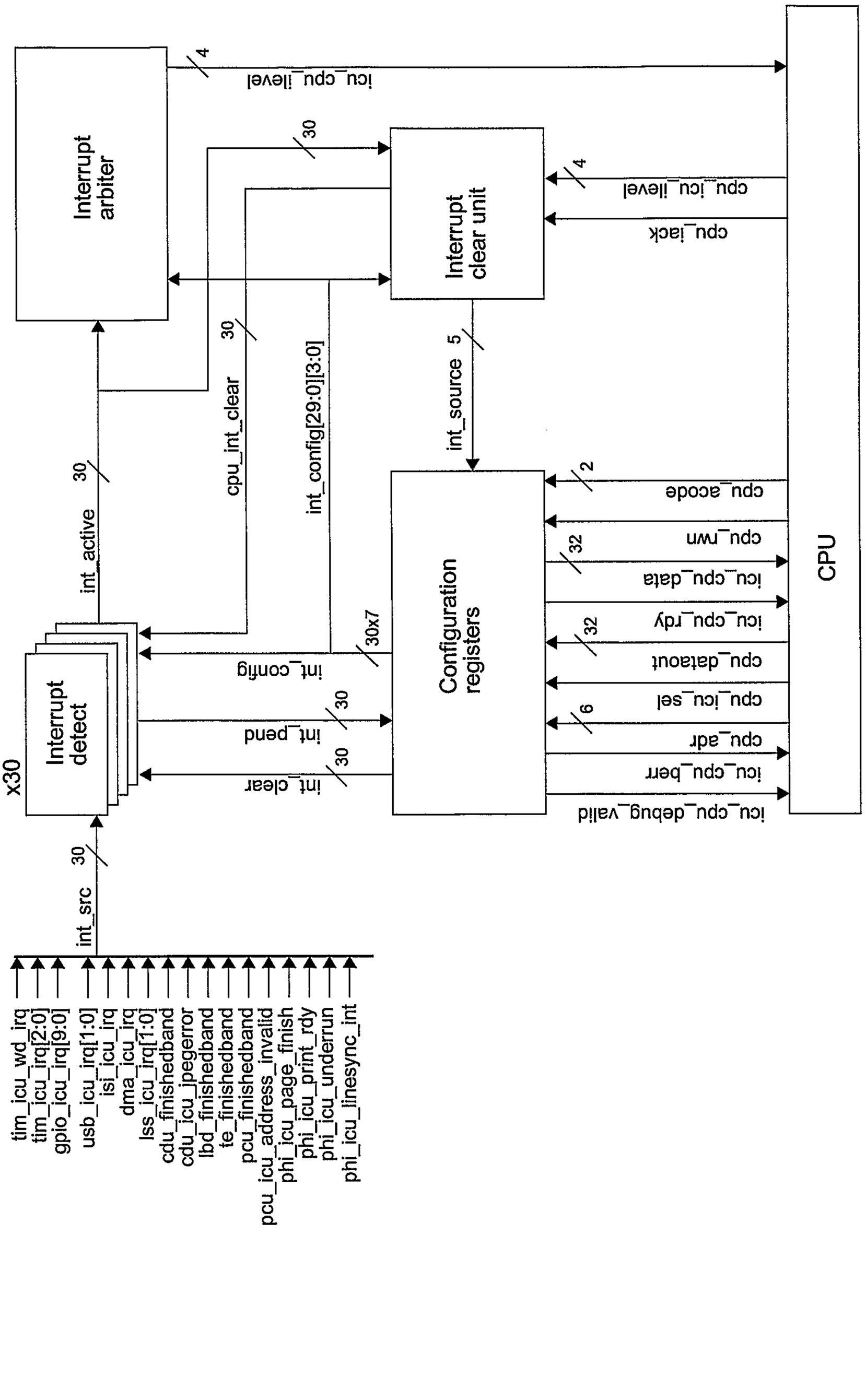

Interrupt arbiter
Interrupt clear unit
Configuration registers
Interrupt detect
x30
CPU
icu_cpu_ilevel
4
30
cpu_icu_ilevel
cpu_iack
int_source 5
int_config[29:0][3:0]
cpu_int_clear
int_active
int_src
int_config
int_pend
int_clear
30x7
cpu_acode
cpu_rwn
icu_cpu_data
icu_cpu_rdy
cpu_datаout
cpu_icu_sel
cpu_adr
icu_cpu_berr
icu_cpu_debug_valid
2
32
6
tim_icu_wd_irq
tim_icu_irq[2:0]
gpio_icu_irq[9:0]
usb_icu_irq[1:0]
isi_icu_irq
dma_icu_irq
lss_icu_irq[1:0]
cdu_finishedband
cdu_icu_jpegerror
lbd_finishedband
te_finishedband
pcu_finishedband
pcu_icu_address_invalid
phi_icu_page_finish
phi_icu_print_rdy
phi_icu_underrun
phi_icu_linesync_int

FIG. 62

Machine remains in same state by default
All outputs are zero unless otherwise stated

State Description:

Idle : Normal Idle state

Compare: Compare interrupt level. Determines the interrupt source

IntClear: Interrupt clear, clear the pending bit for the current interrupt vector

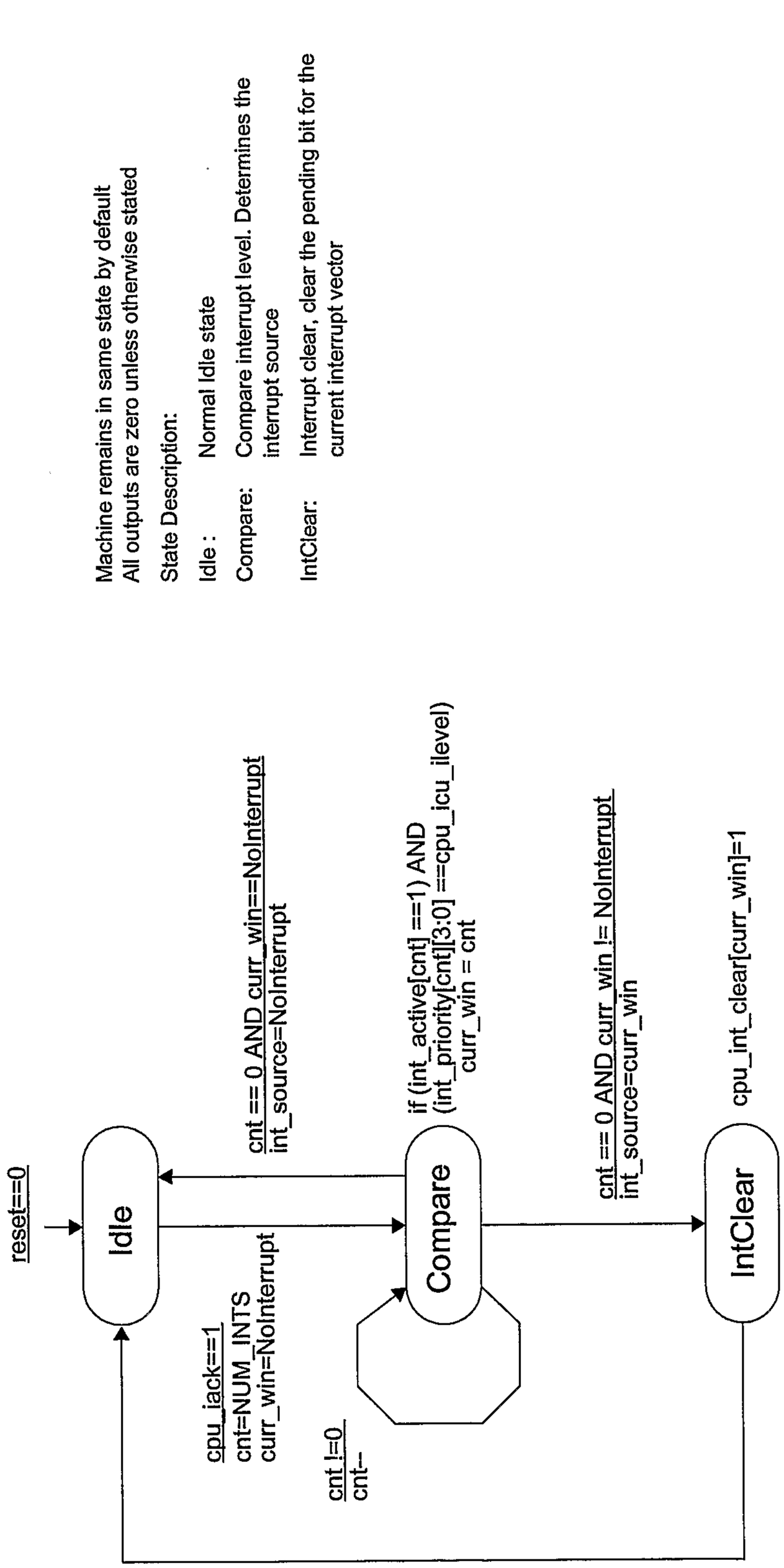


FIG. 63

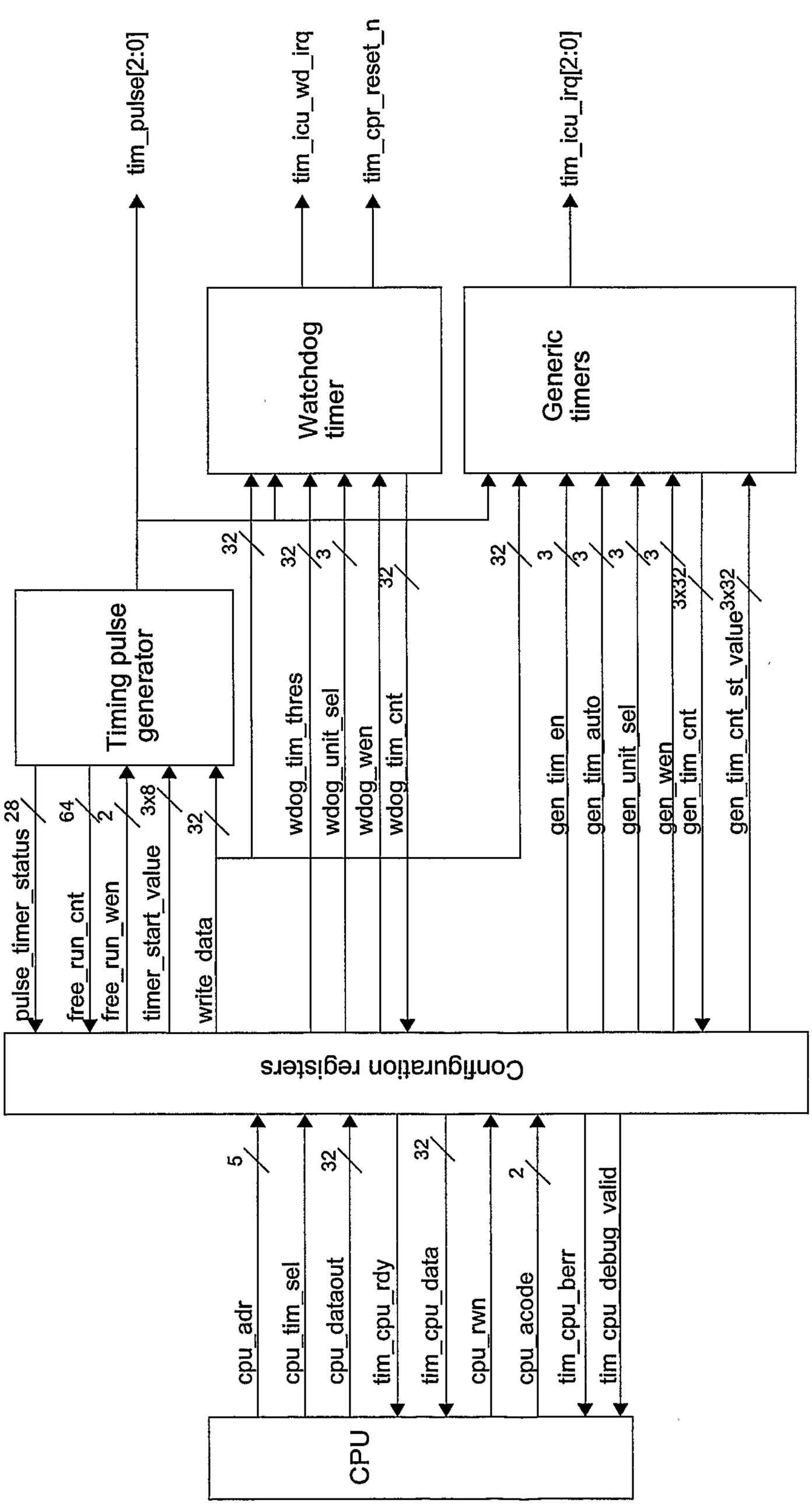

tim_pulse[2:0]
tim_icu_wd_irq
tim_cpr_reset_n
tim_icu_irq[2:0]
Watchdog timer
Generic timers
Timing pulse generator
Configuration registers
CPU
pulse_timer_status 28
free_run_cnt 64
free_run_wen 2
timer_start_value 3x8
write_data 32
wdog_tim_thres 32
wdog_unit_sel 3
wdog_wen
wdog_tim_cnt 32
gen_tim_en 32
gen_tim_auto 3
gen_unit_sel 3
gen_wen 3
gen_tim_cnt 3x32
gen_tim_cnt_st_value 3x32
cpu_adr 5
cpu_tim_sel
cpu_dataout 32
tim_cpu_rdy
tim_cpu_data 32
cpu_rwn
cpu_acode 2
tim_cpu_berr
tim_cpu_debug_valid

FIG. 63A

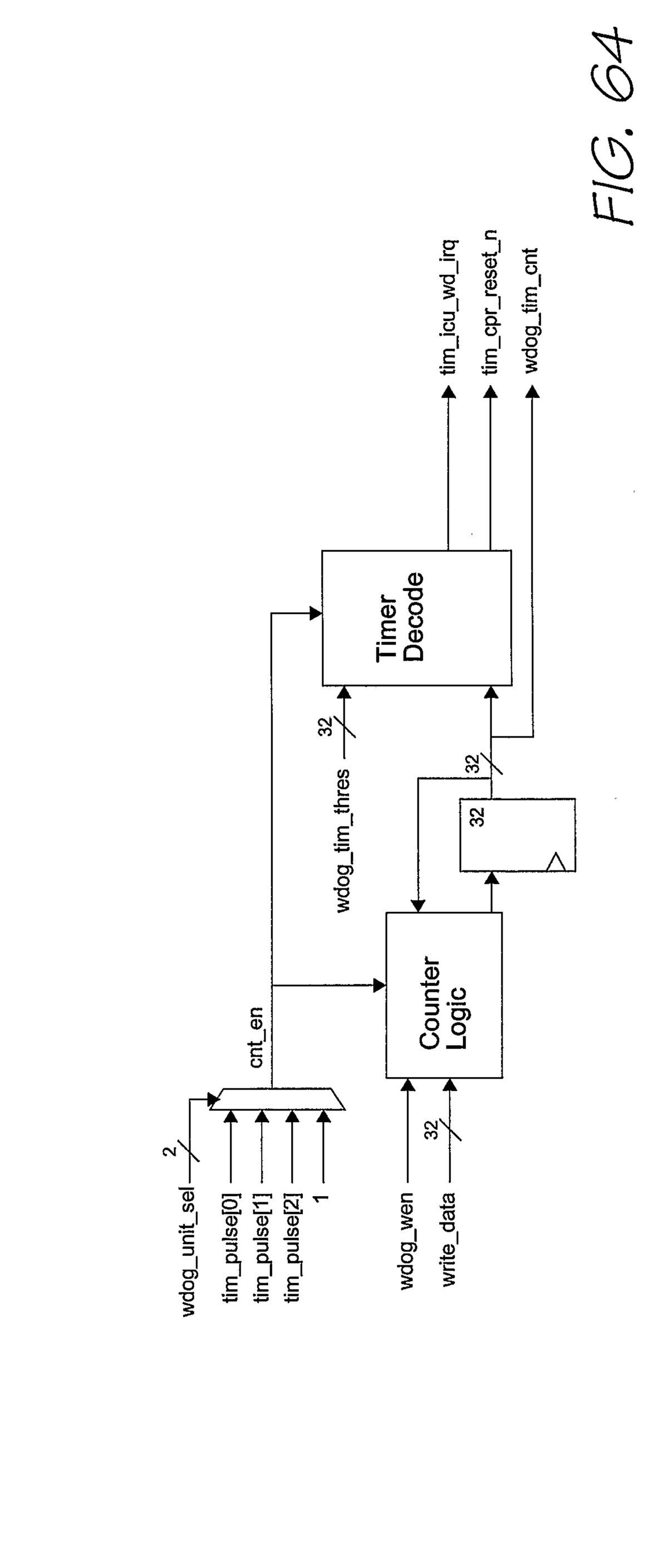

wdog_unit_sel
2
tim_pulse[0]
tim_pulse[1]
tim_pulse[2]
1
cnt_en
wdog_wen
write_data
32
Counter Logic
wdog_tim_thres
32
Timer Decode
tim_cpr_reset_n
wdog_tim_cnt
tim_icu_wd_irq

FIG. 64

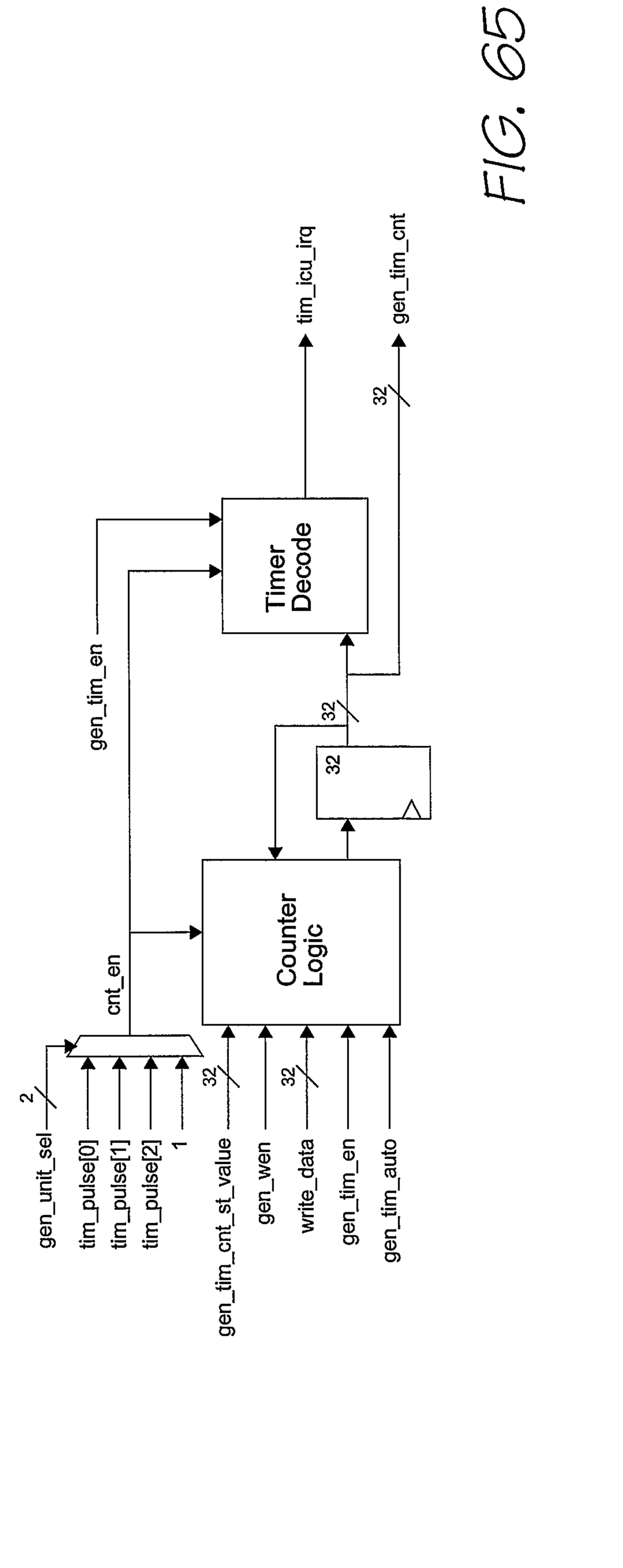

gen_unit_sel
2
tim_pulse[0]
tim_pulse[1]
tim_pulse[2]
1
gen_tim_cnt_st_value
gen_wen
write_data
gen_tim_en
gen_tim_auto
cnt_en
gen_tim_en
32
Counter Logic
32
Timer Decode
32
gen_tim_cnt
tim_icu_irq

FIG. 65

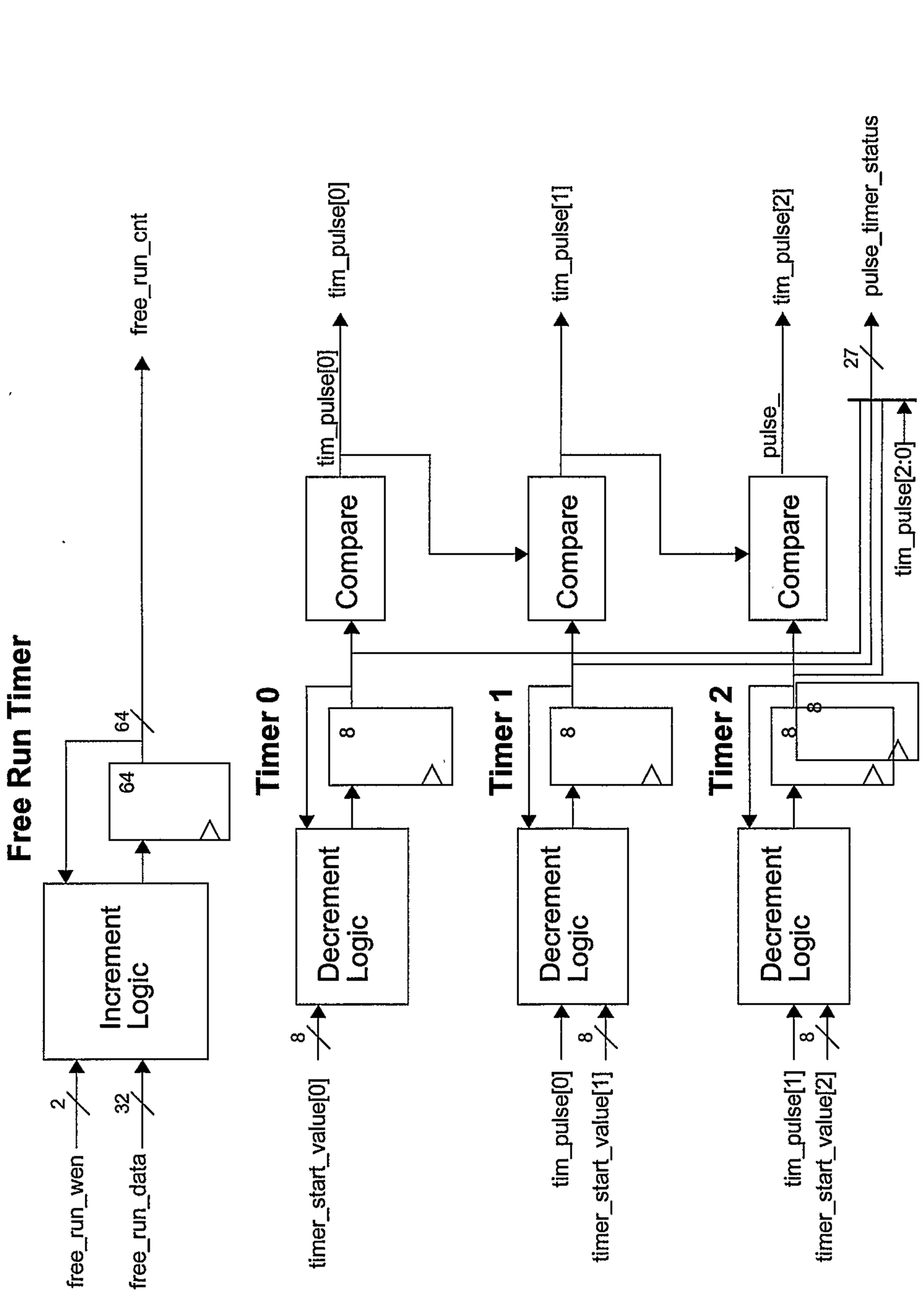

Free Run Timer
Increment Logic
64
64
free_run_cnt
2
free_run_wen
32
free_run_data
Timer 0
Timer 1
Timer 2
Decrement Logic
8
Compare
tim_pulse[0]
tim_pulse[1]
tim_pulse[2]
pulse_
27
pulse_timer_status
tim_pulse[2:0]
timer_start_value[0]
timer_start_value[1]
timer_start_value[2]

FIG. 66

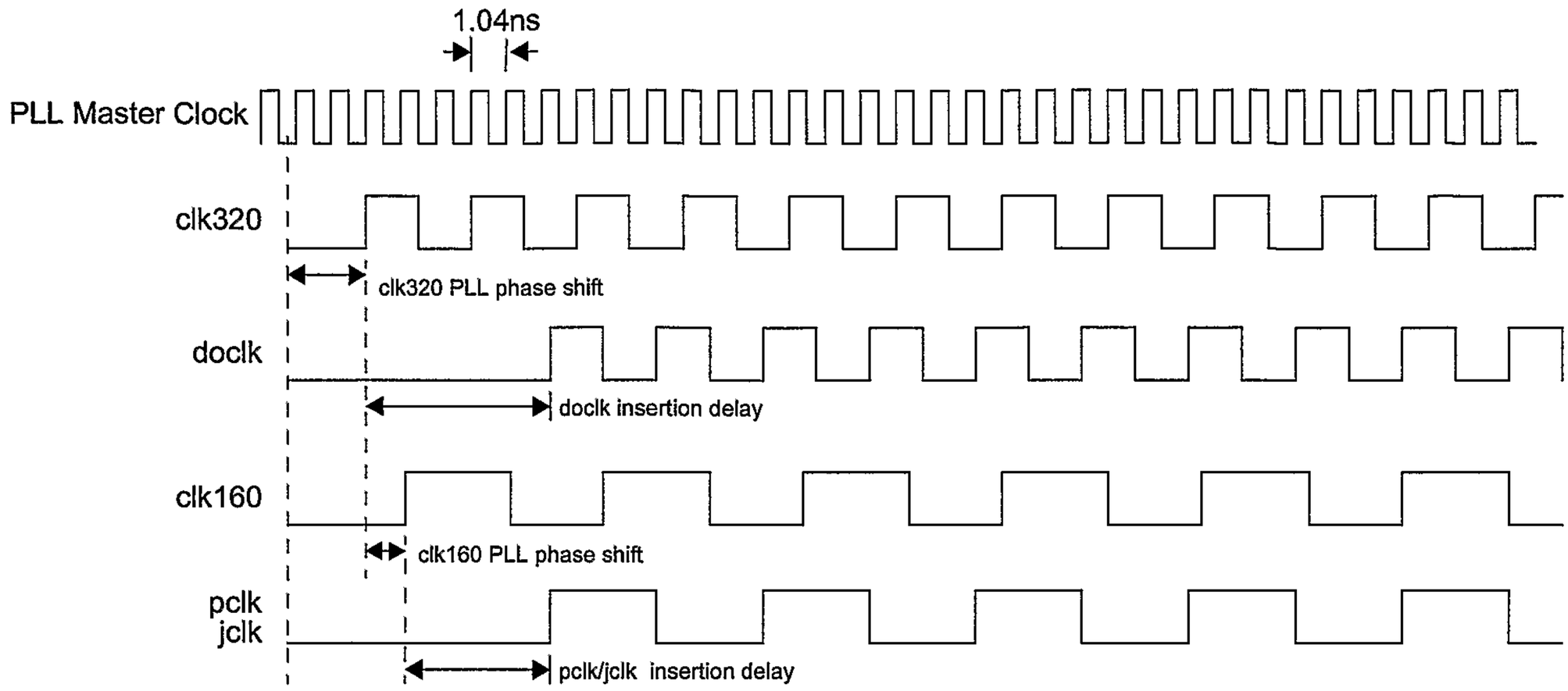

1.04ns
PLL Master Clock
clk320
clk320 PLL phase shift
doclk
doclk insertion delay
clk160
clk160 PLL phase shift
pclk
jclk
pclk/jclk insertion delay

FIG. 67

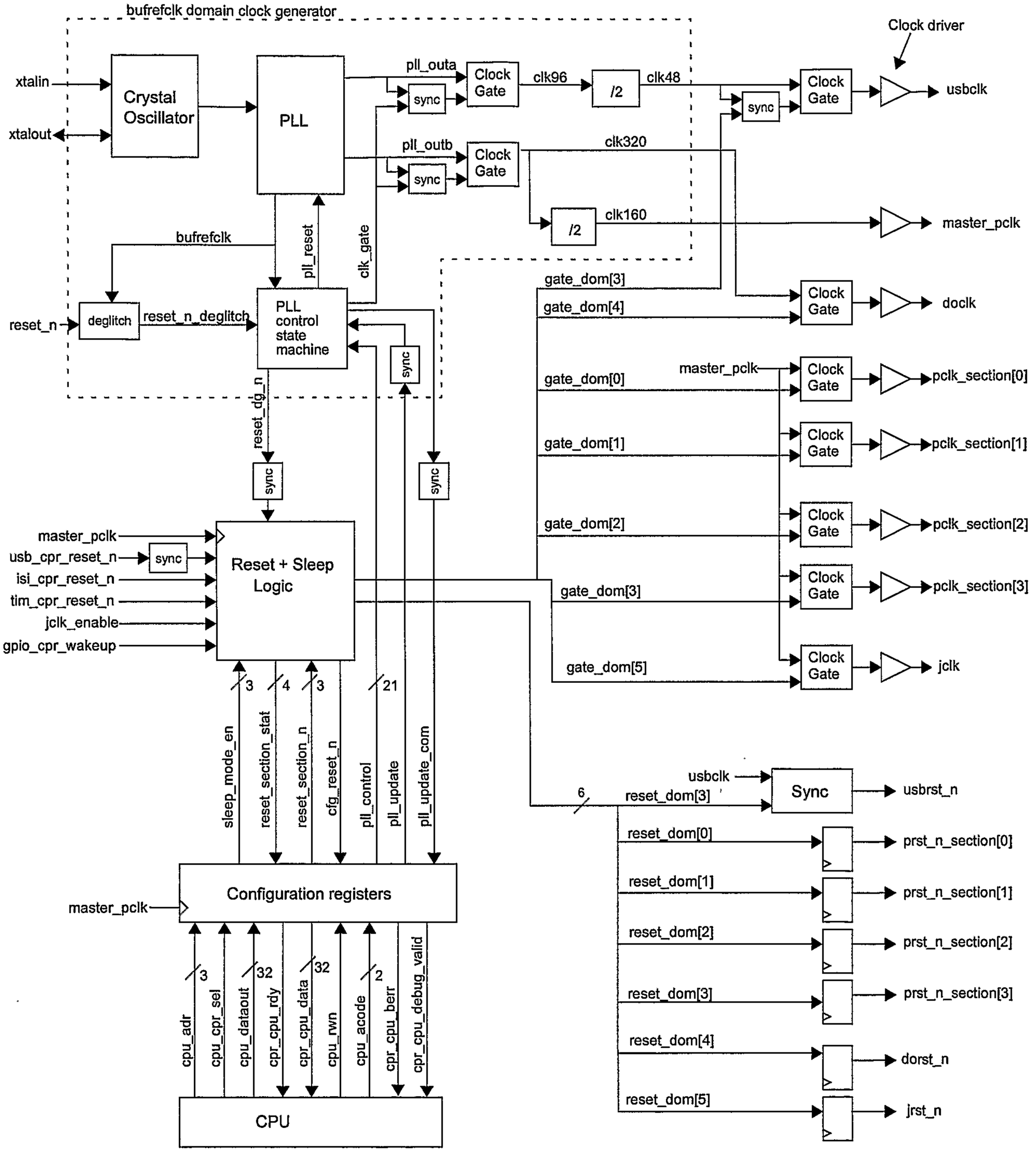

bufrefclk domain clock generator
xtalin
xtalout
Crystal Oscillator
PLL
pll_outa
pll_outb
sync
Clock Gate
clk96
/2
clk48
clk320
clk160
bufrefclk
pll_reset
clk_gate
reset_n
deglitch
reset_n_deglitch
PLL control state machine
reset_dg_n
Clock driver
usbclk
master_pclk
gate_dom[3]
gate_dom[4]
gate_dom[0]
gate_dom[1]
gate_dom[2]
gate_dom[5]
doclk
pclk_section[0]
pclk_section[1]
pclk_section[2]
pclk_section[3]
jclk
usb_cpr_reset_n
isi_cpr_reset_n
tim_cpr_reset_n
jclk_enable
gpio_cpr_wakeup
Reset + Sleep Logic
Configuration registers
sleep_mode_en
reset_section_stat
reset_section_n
cfg_reset_n
pll_control
pll_update
pll_update_com
Sync
reset_dom[0]
reset_dom[1]
reset_dom[2]
reset_dom[3]
reset_dom[4]
reset_dom[5]
usbrst_n
prst_n_section[0]
prst_n_section[1]
prst_n_section[2]
prst_n_section[3]
dorst_n
jrst_n
cpu_adr
cpu_cpr_sel
cpu_dataout
cpr_cpu_rdy
cpr_cpu_data
cpu_rwn
cpu_acode
cpr_cpu_berr
cpr_cpu_debug_valid
CPU

FIG. 68

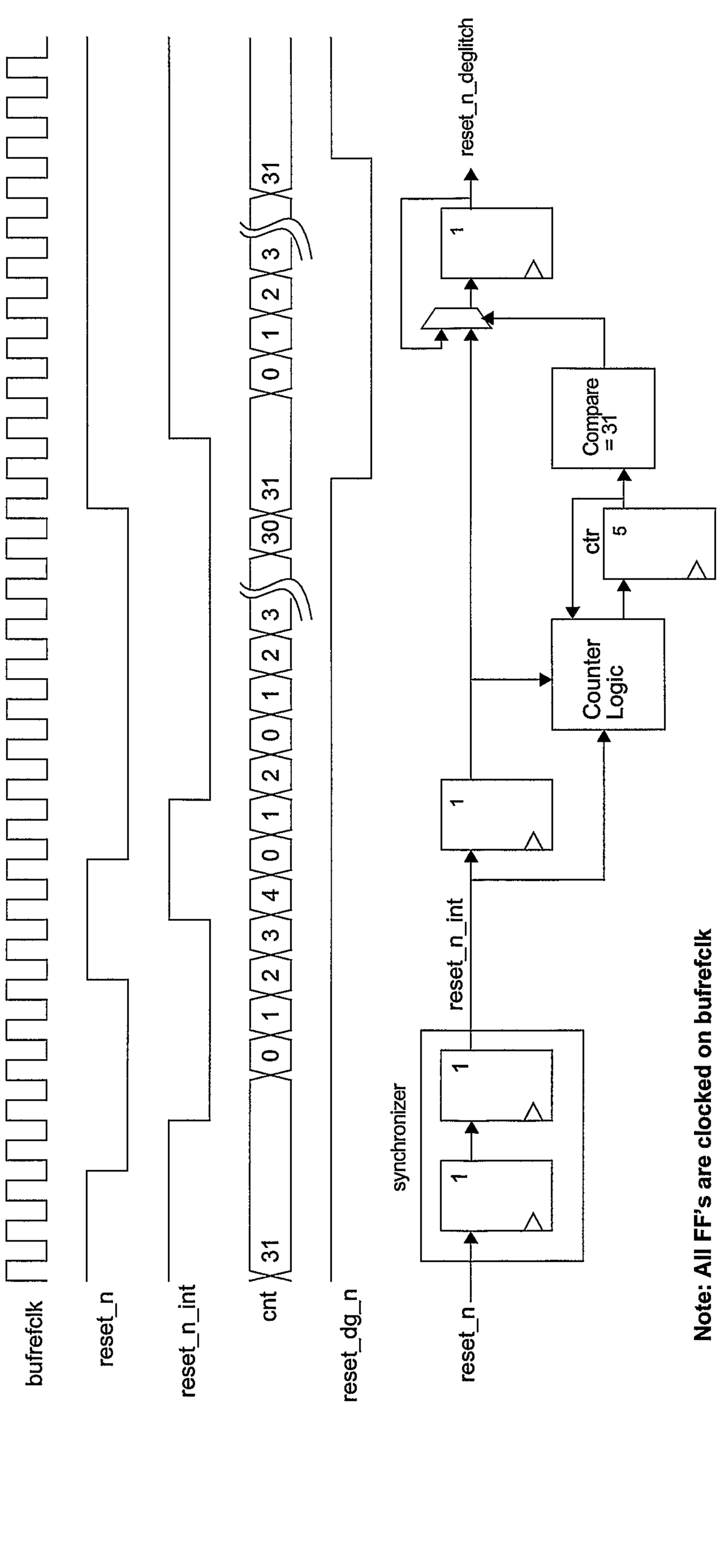

bufrefclk
reset_n
reset_n_int
cnt
reset_dg_n
reset_n
synchronizer
1
1
reset_n_int
Counter Logic
ctr
5
Compare =31
1
reset_n_deglitch
31
0
1
2
3
4
30
31
0
1
2
3
0
1
2
3
31
Note: All FF's are clocked on bufrefclk

FIG. 69

FIG. 70

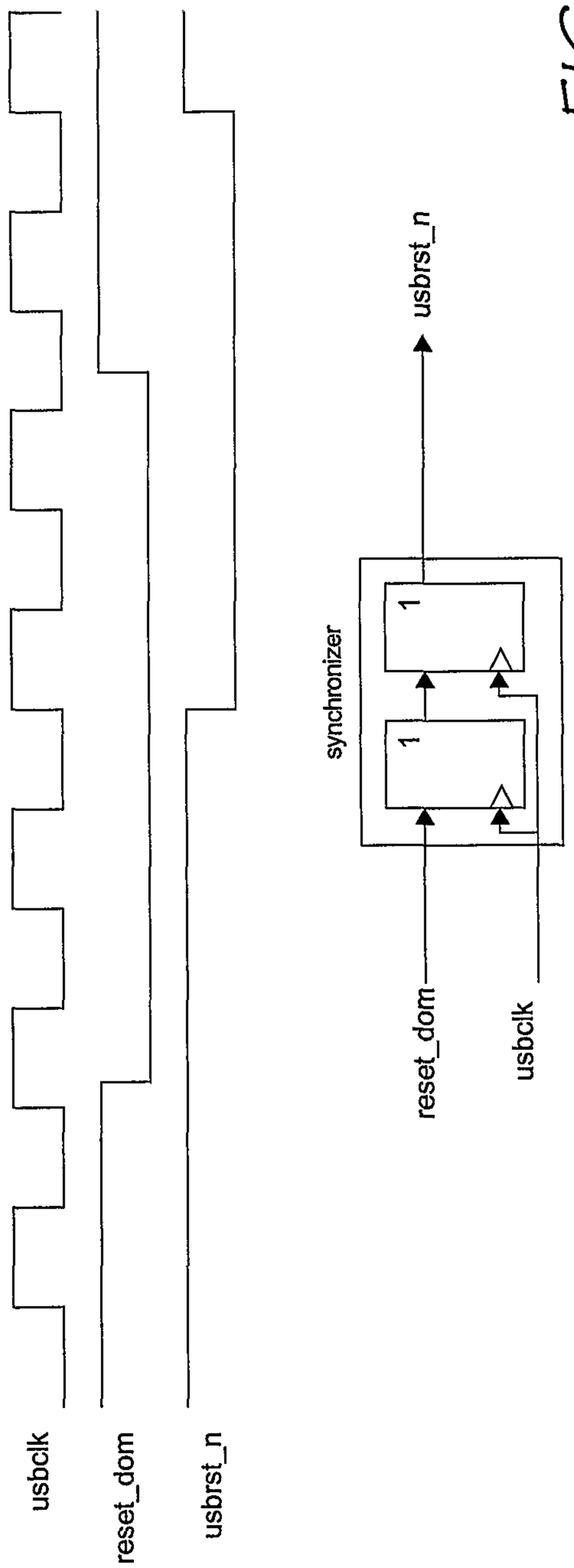
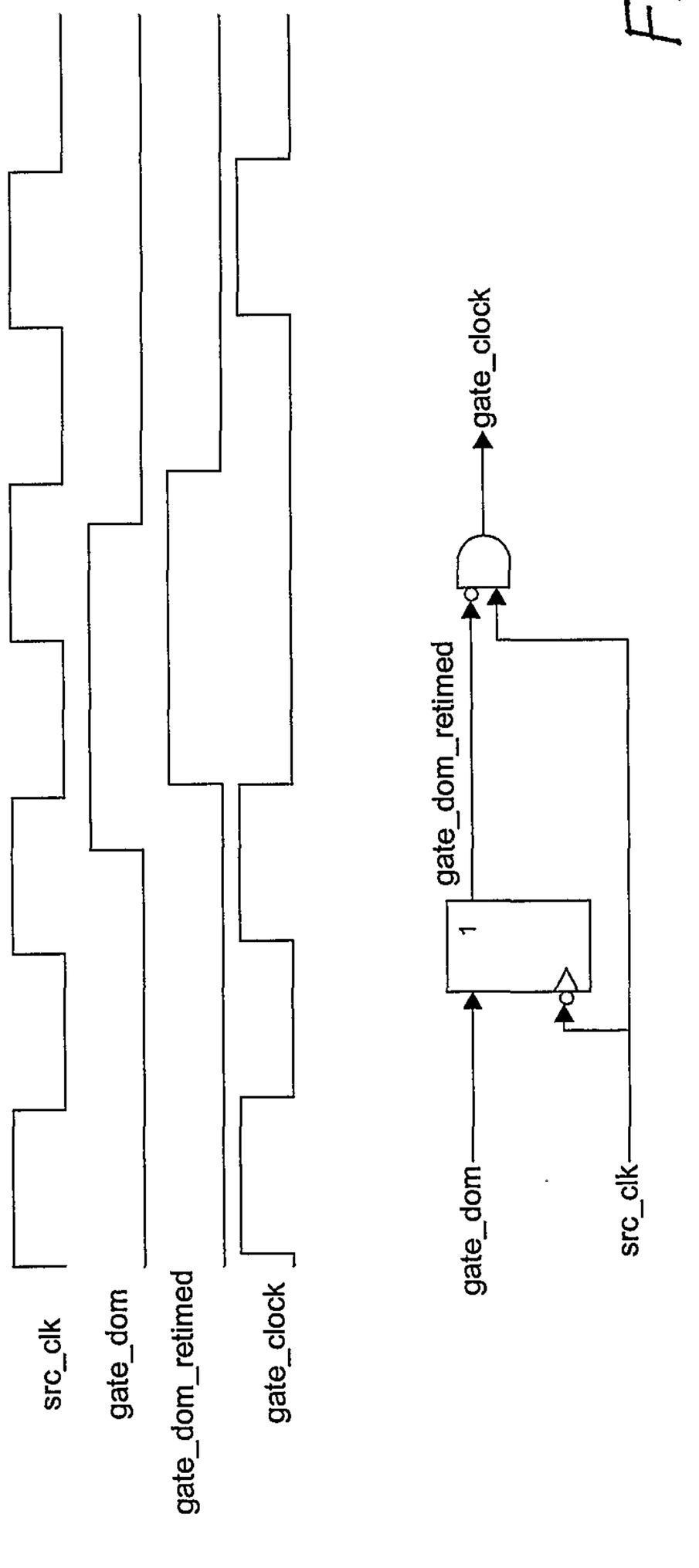

usbclk
reset_dom
usbrst_n
usbrst_n
synchronizer
1
1
reset_dom
usbclk

FIG. 71


src_clk
gate_dom
gate_dom_retimed
gate_clock
gate_clock
gate_dom_retimed
1
gate_dom
src_clk

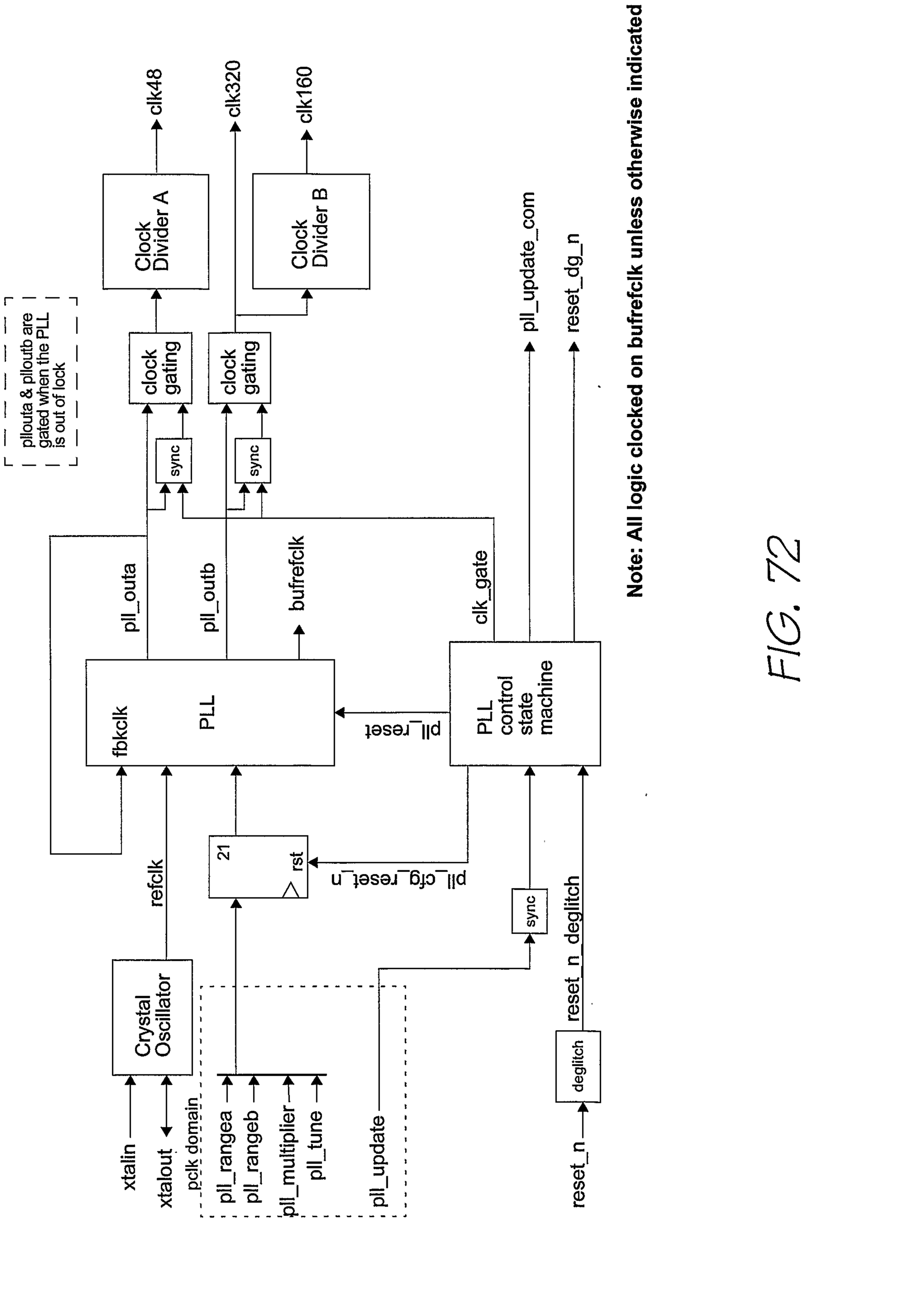

Note: All logic clocked on bufrefclk unless otherwise indicated
pll_update_com
reset_dg_n
clk48
clk320
clk160
Clock Divider A
Clock Divider B
clock gating
clock gating
sync
sync
pllouta & ploutb are gated when the PLL is out of lock
pll_outa
pll_outb
bufrefclk
clk_gate
fbkclk
PLL
pll_reset
PLL control state machine
refclk
21
rst
pll_cfg_reset_n
sync
reset_n_deglitch
Crystal Oscillator
deglitch
xtalin
xtalout
pclk domain
pll_rangea
pll_rangeb
pll_multiplier
pll_tune
pll_update
reset_n

FIG. 72

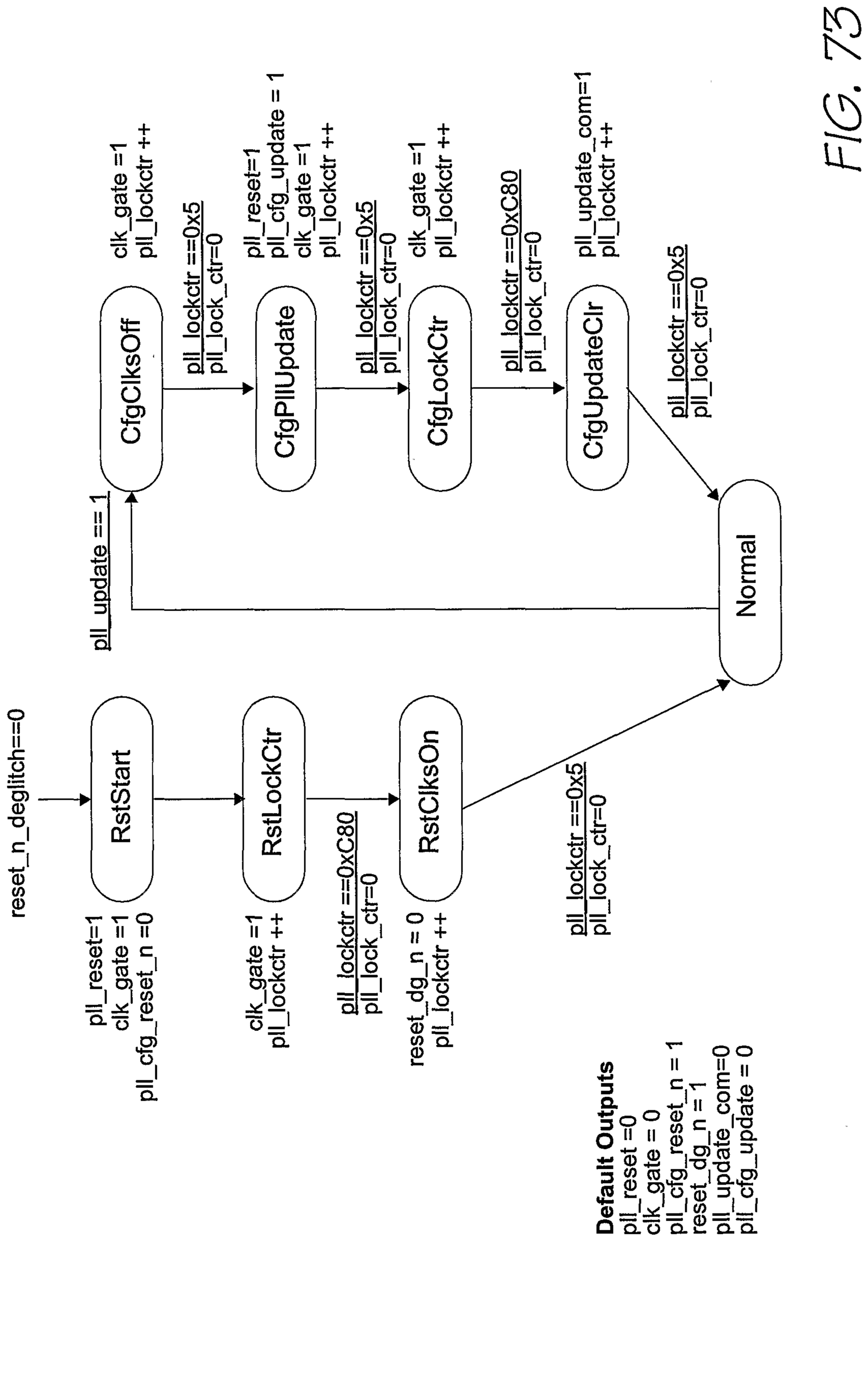

reset_n_deglitch==0
RstStart
pll_reset=1
clk_gate =1
pll_cfg_reset_n =0
RstLockCtr
clk_gate =1
pll_lockctr ++
pll_lockctr ==0x5
pll_lock_ctr=0
RstClksOn
reset_dg_n = 0
pll_lockctr ++
pll_lockctr ==0xC80
pll_lock_ctr=0
Normal
pll_update == 1
CfgClksOff
clk_gate =1
pll_lockctr ++
pll_lockctr ==0x5
pll_lock_ctr=0
CfgPllUpdate
pll_reset=1
pll_cfg_update = 1
clk_gate =1
pll_lockctr ++
pll_lockctr ==0x5
pll_lock_ctr=0
CfgLockCtr
clk_gate =1
pll_lockctr ++
pll_lockctr ==0xC80
pll_lock_ctr=0
CfgUpdateClr
pll_update_com=1
pll_lockctr ++
pll_lockctr ==0x5
pll_lock_ctr=0
Default Outputs
pll_reset =0
clk_gate = 0
pll_cfg_reset_n = 1
reset_dg_n = 1
pll_update_com=0
pll_cfg_update = 0

FIG. 7B

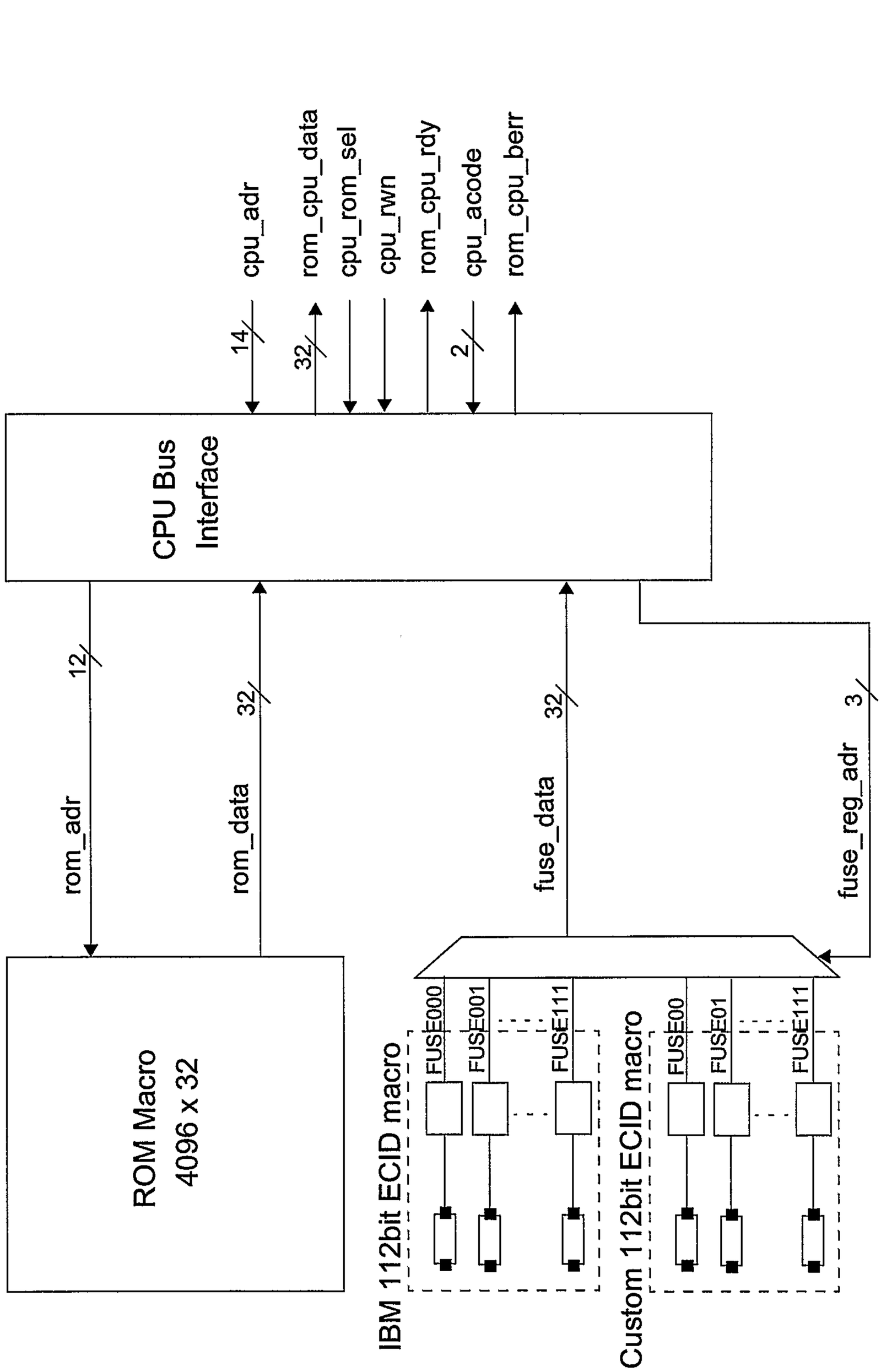

CPU Bus Interface
cpu_adr
rom_cpu_data
cpu_rom_sel
cpu_rwn
rom_cpu_rdy
cpu_acode
rom_cpu_berr
14
32
2
rom_adr
rom_data
fuse_data
fuse_reg_adr
12
32
32
3
ROM Macro
4096 x 32
IBM 112bit ECID macro
Custom 112bit ECID macro
FUSE000
FUSE001
FUSE111
FUSE00
FUSE01
FUSE111

FIG. 74

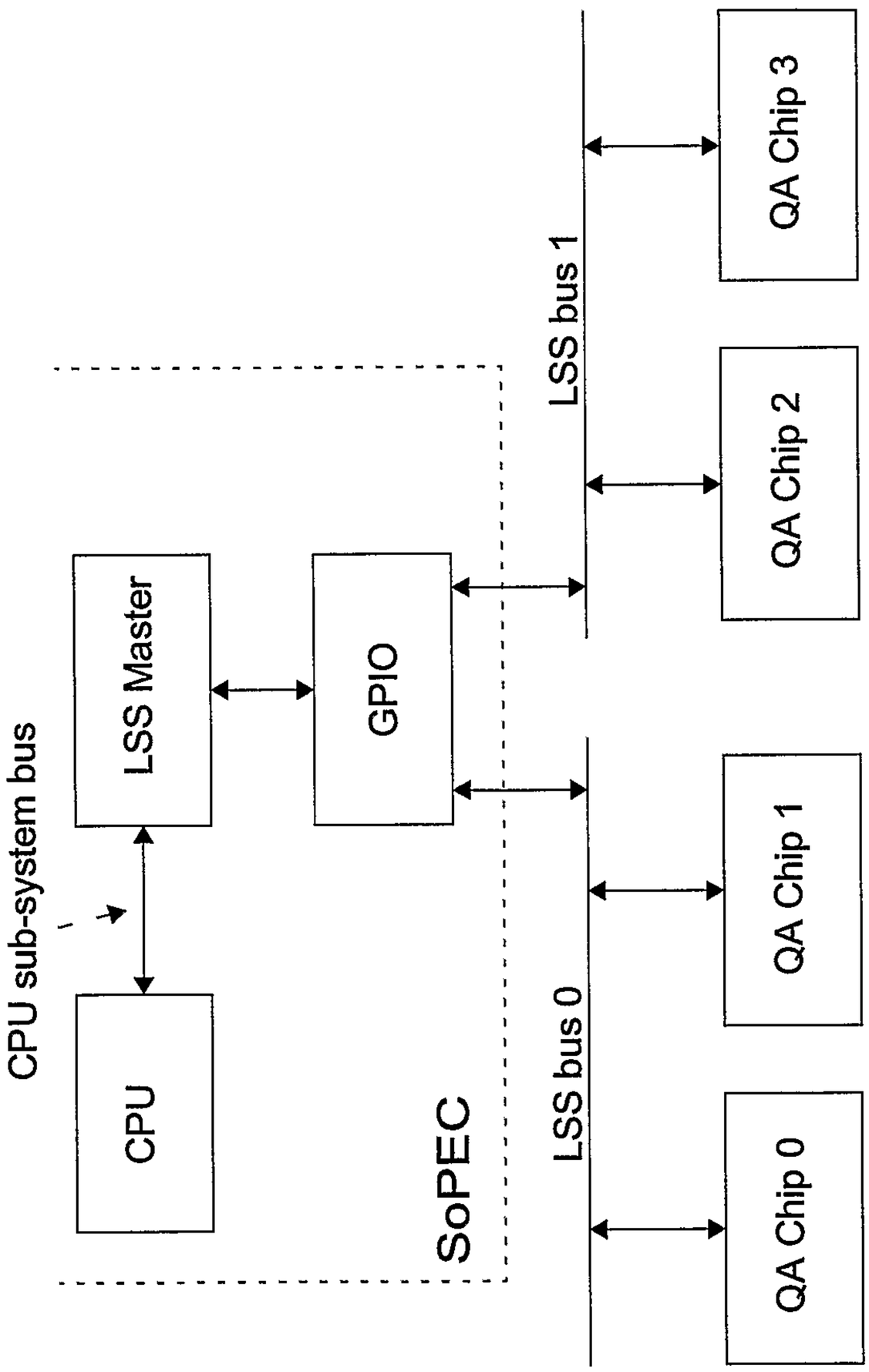

CPU sub-system bus
CPU
LSS Master
GPIO
SoPEC
LSS bus 1
LSS bus 0
QA Chip 3
QA Chip 2
QA Chip 1
QA Chip 0

FIG. 75

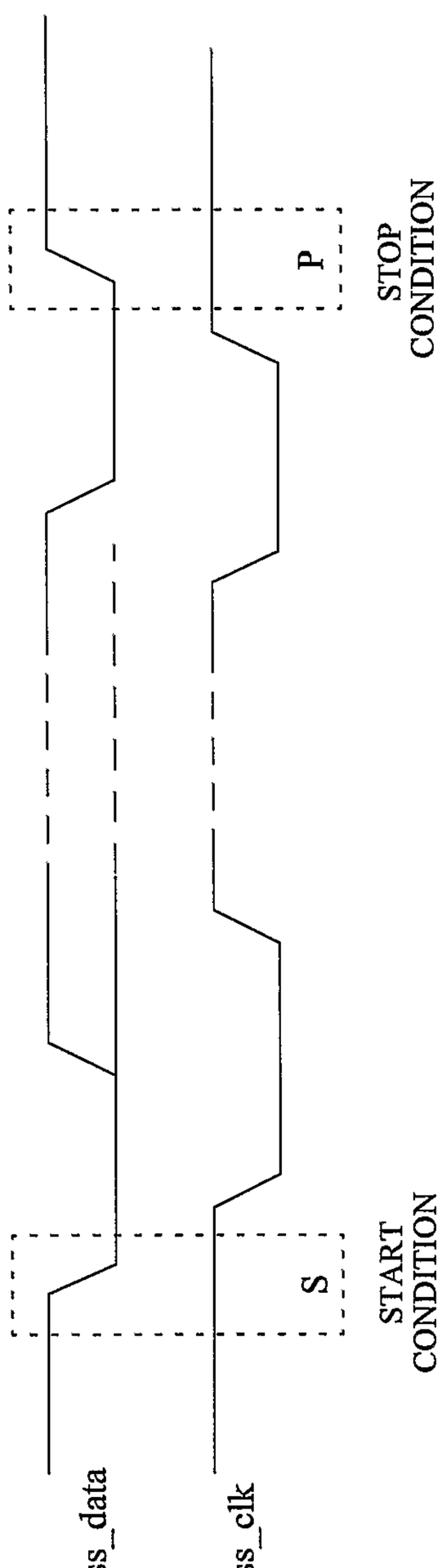

lss_data
lss_clk
S
START CONDITION
P
STOP CONDITION

FIG. 76

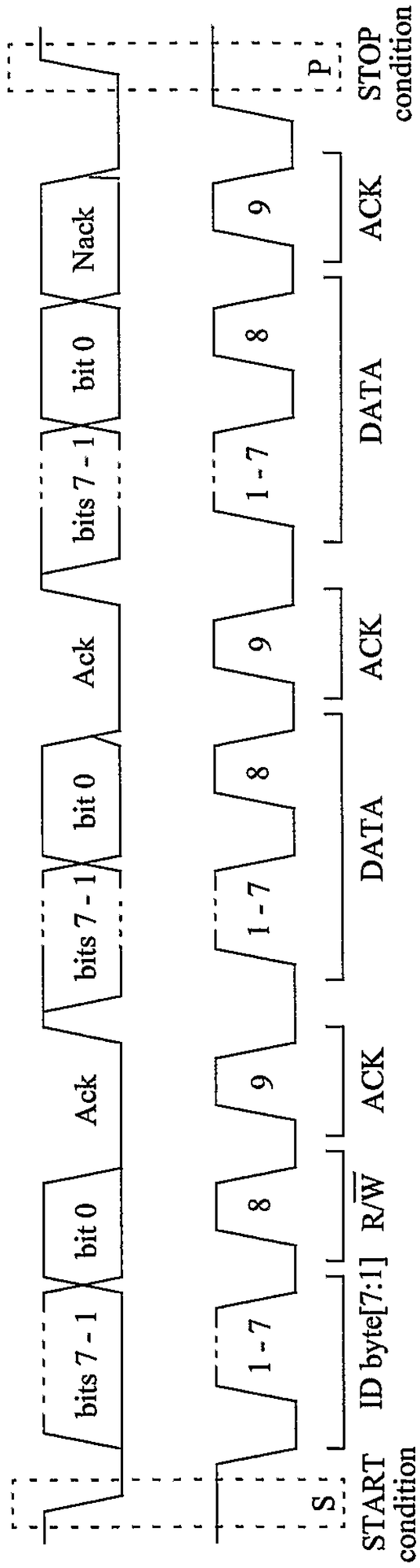

lss_data
lss_clk
S
START condition
ID byte[7:1]
R/W
ACK
DATA
bits 7 - 1
bit 0
Ack
Nack
1 - 7
8
9
P
STOP condition

FIG. 77

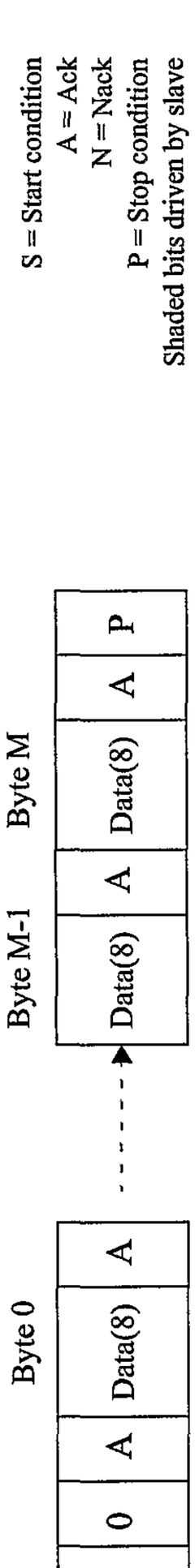

S = Start condition
A = Ack
N = Nack
P = Stop condition
Shaded bits driven by slave
Byte 0
S
ID byte[7:1]
0
A
Data(8)
A
Byte M-1
Data(8)
A
Byte M
Data(8)
P

FIG. 78

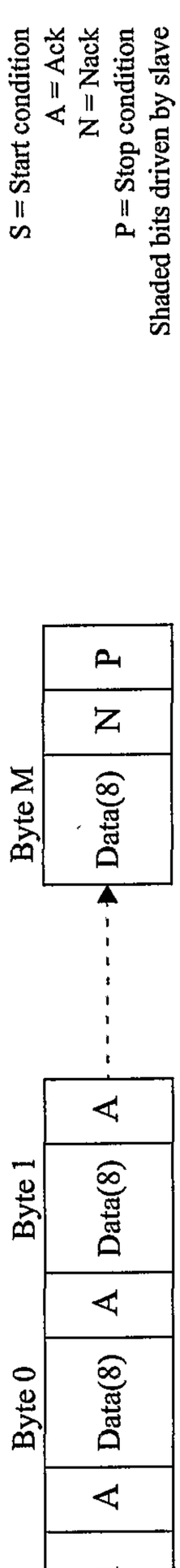

S = Start condition
A = Ack
N = Nack
P = Stop condition
Shaded bits driven by slave
Byte 0
S
ID byte[7:1]
1
A
Data(8)
A
Byte 1
Data(8)
A
Byte M
Data(8)
N
P

FIG. 79

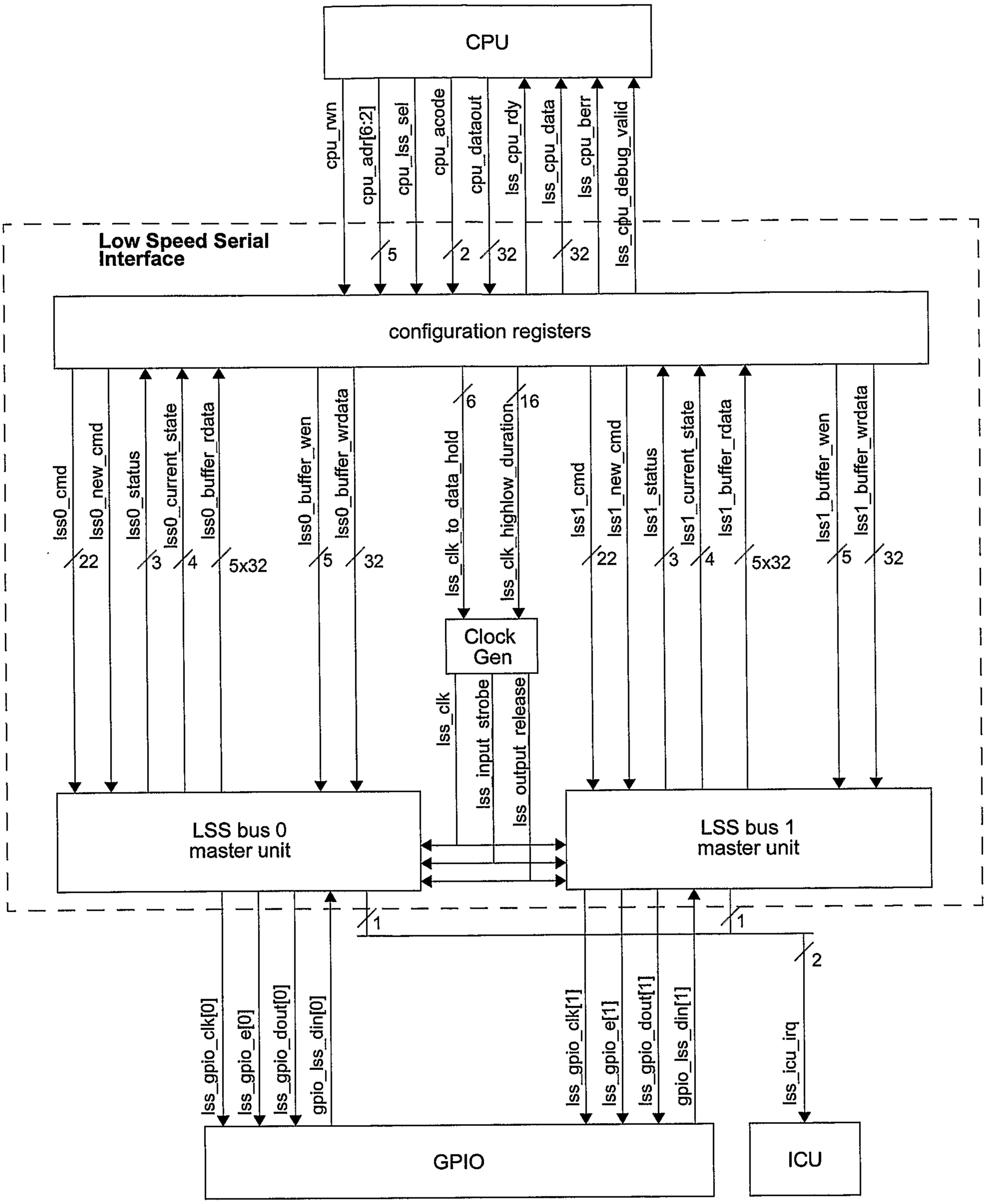

CPU
cpu_rwn
cpu_adr[6:2]
cpu_lss_sel
cpu_acode
cpu_dataout
lss_cpu_rdy
lss_cpu_data
lss_cpu_berr
cpu_debug_valid
5
2
32
32
Low Speed Serial Interface
configuration registers
lss0_cmd
lss0_new_cmd
lss0_status
lss0_current_state
lss0_buffer_rdata
lss0_buffer_wen
lss0_buffer_wrdata
lss_clk_to_data_hold
lss_clk_highlow_duration
lss1_cmd
lss1_new_cmd
lss1_status
lss1_current_state
lss1_buffer_rdata
lss1_buffer_wen
lss1_buffer_wrdata
22
3
4
5x32
5
32
6
16
22
3
4
5x32
5
32
Clock Gen
lss_clk
lss_input_strobe
lss_output_release
LSS bus 0 master unit
LSS bus 1 master unit
1
1
2
lss_gpio_clk[0]
lss_gpio_e[0]
lss_gpio_dout[0]
gpio_lss_din[0]
lss_gpio_clk[1]
lss_gpio_e[1]
lss_gpio_dout[1]
gpio_lss_din[1]
lss_icu_irq
GPIO
ICU

FIG. 80

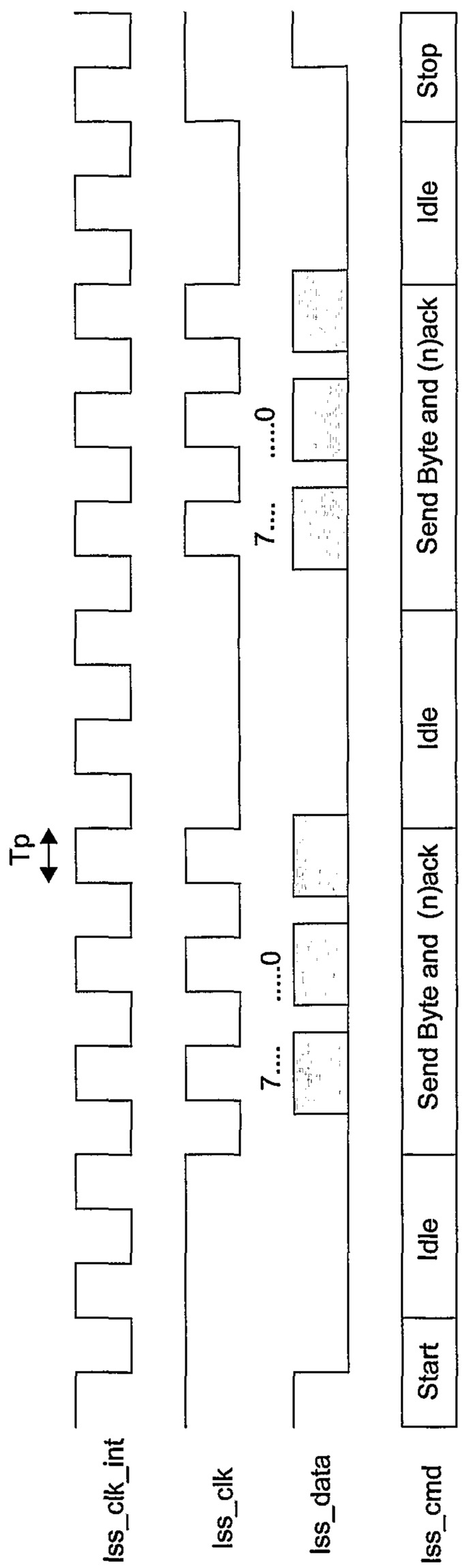

lss_clk_int
lss_clk
lss_data
lss_cmd
Start
Idle
Send Byte and (n)ack
Idle
Stop
7.... 0
Tp

FIG. 81

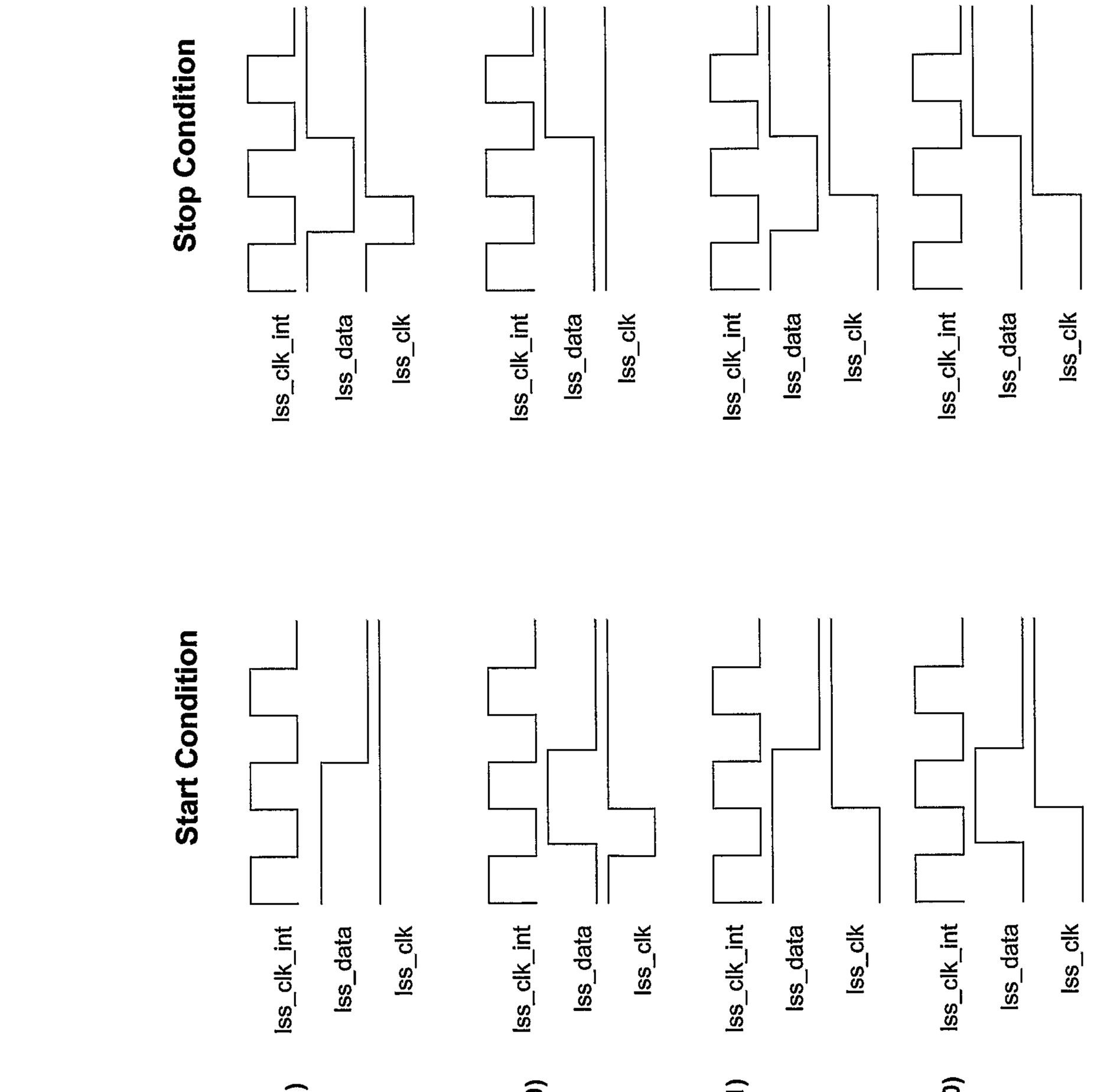

Stop Condition
lss_clk_int
lss_data
lss_clk
lss_clk_int
lss_data
lss_clk
lss_clk_int
lss_data
lss_clk
lss_clk_int
lss_data
lss_clk
Start Condition
lss_clk_int
lss_data
lss_clk
lss_clk_int
lss_data
lss_clk
lss_clk_int
lss_data
lss_clk
lss_clk_int
lss_data
lss_clk
Bus state case 1
(lss_clk=1,lss_data=1)
Bus state case 2
(lss_clk=1,lss_data=0)
Bus state case 3
(lss_clk=0,lss_data=1)
Bus state case 4
(lss_clk=0,lss_data=0)

FIG. 82

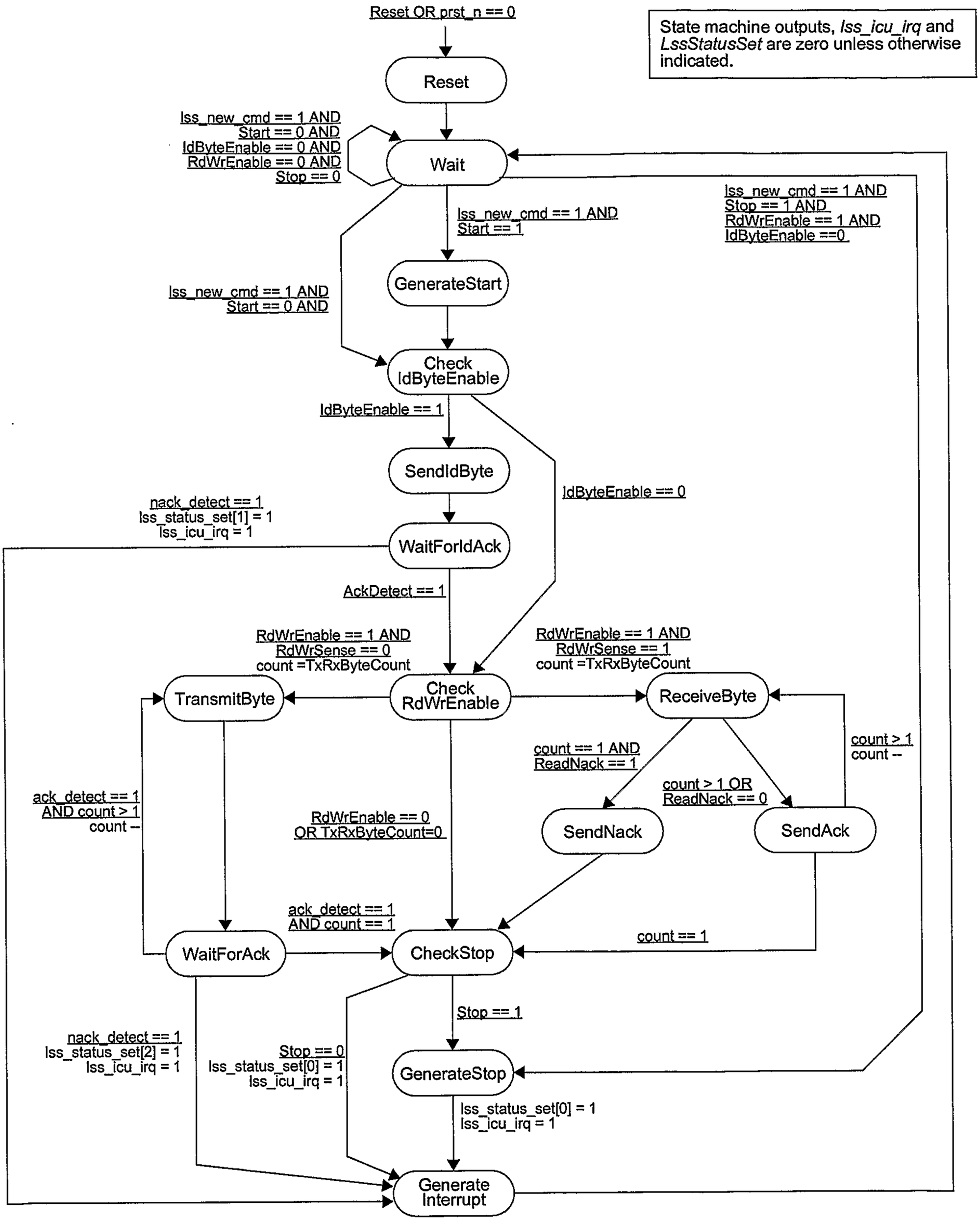

Reset OR prst_n == 0
Reset
State machine outputs, lss_icu_irq and LssStatusSet are zero unless otherwise indicated.
lss_new_cmd == 1 AND Start == 0 AND IdByteEnable == 0 AND RdWrEnable == 0 AND Stop == 0
Wait
lss_new_cmd == 1 AND Start == 1
lss_new_cmd == 1 AND Stop == 1 AND RdWrEnable == 1 AND IdByteEnable ==0
lss_new_cmd == 1 AND Start == 0 AND
GenerateStart
Check IdByteEnable
IdByteEnable == 1
SendIdByte
IdByteEnable == 0
nack_detect == 1 lss_status_set[1] = 1 lss_icu_irq = 1
WaitForIdAck
AckDetect == 1
RdWrEnable == 1 AND RdWrSense == 0 count =TxRxByteCount
RdWrEnable == 1 AND RdWrSense == 1 count =TxRxByteCount
TransmitByte
Check RdWrEnable
ReceiveByte
count == 1 AND ReadNack == 1
count > 1 OR ReadNack == 0
count > 1 count --
ack_detect == 1 AND count > 1 count --
RdWrEnable == 0 OR TxRxByteCount=0
SendNack
SendAck
ack_detect == 1 AND count == 1
count == 1
WaitForAck
CheckStop
Stop == 1
nack_detect == 1 lss_status_set[2] = 1 lss_icu_irq = 1
Stop == 0 lss_status_set[0] = 1 lss_icu_irq = 1
GenerateStop
lss_status_set[0] = 1 lss_icu_irq = 1
Generate Interrupt

FIG. 83

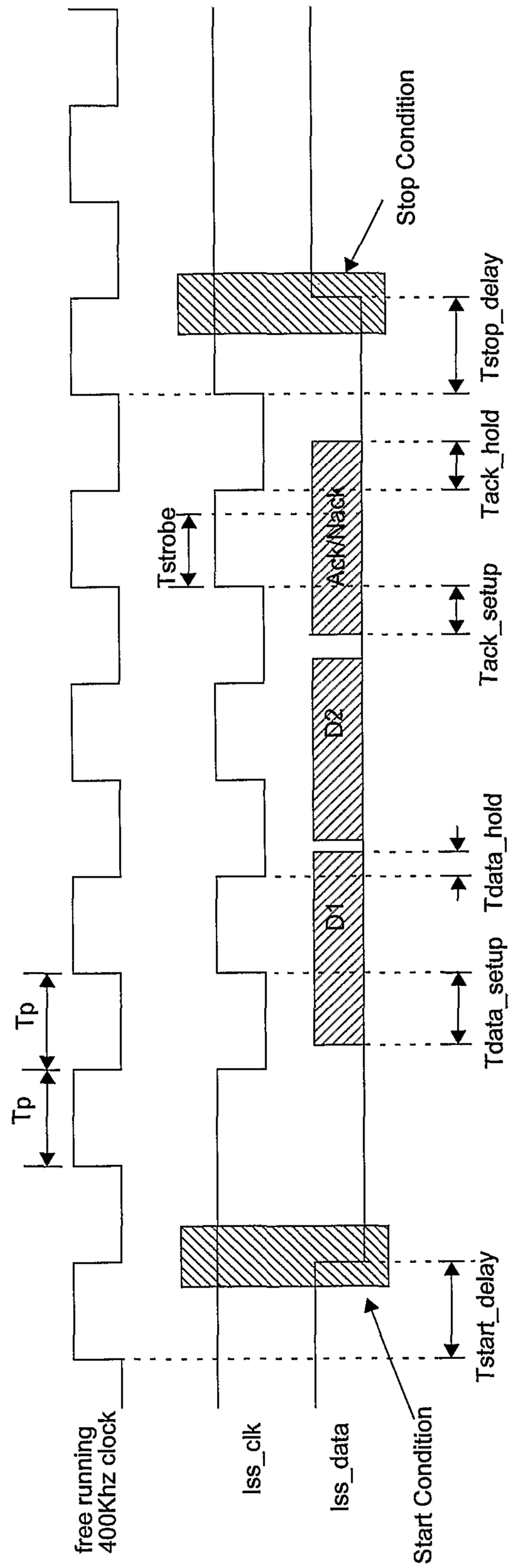

free running
400Khz clock
lss_clk
lss_data
Start Condition
Tstart_delay
Tp
Tp
Tdata_setup
D1
Tdata_hold
D2
Tack_setup
Ack/Nack
Tstrobe
Tack_hold
Stop Condition
Tstop_delay

FIG. 84

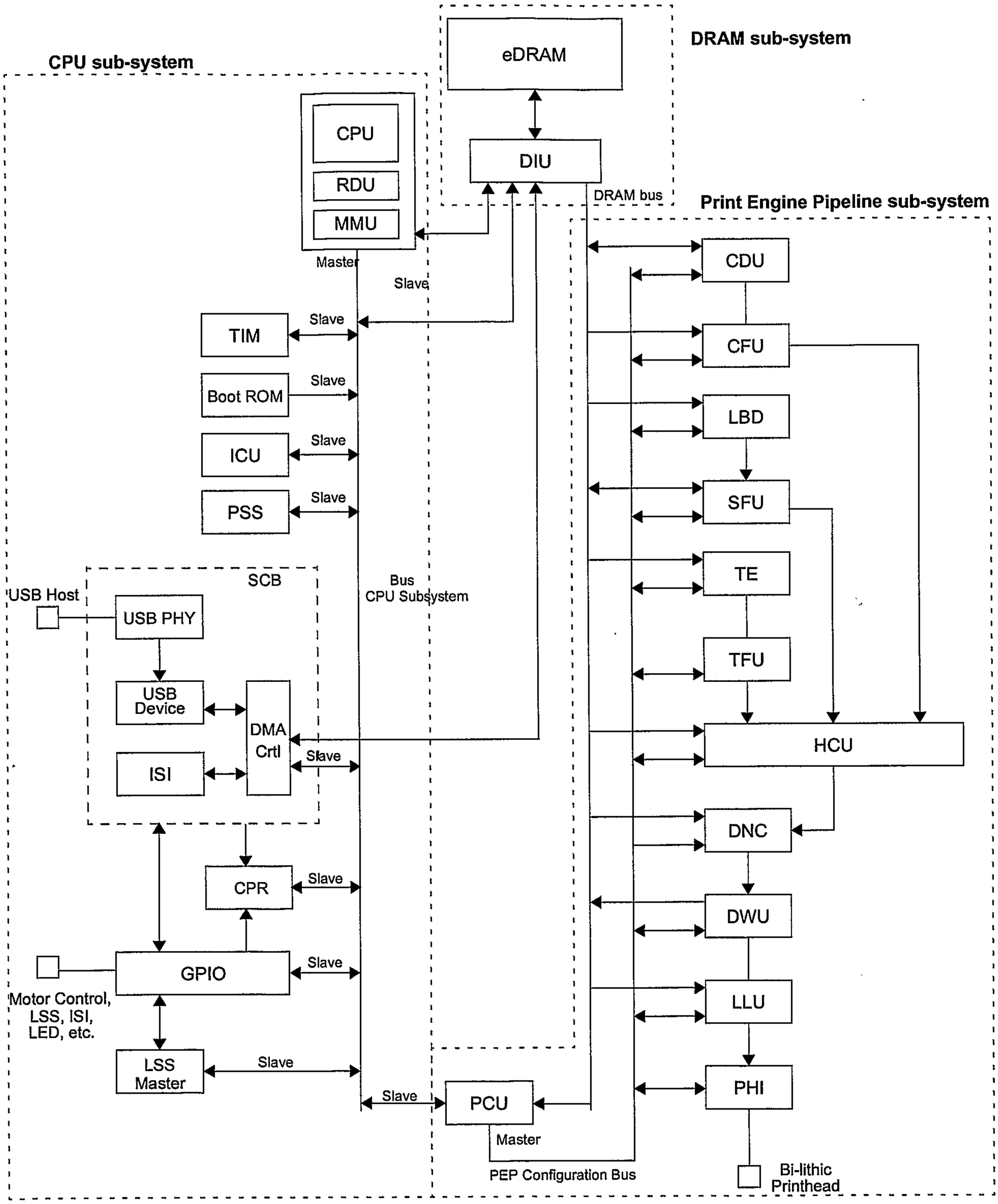

CPU sub-system
DRAM sub-system
eDRAM
CPU
RDU
MMU
DIU
DRAM bus
Print Engine Pipeline sub-system
Master
Slave
CDU
TIM
Slave
CFU
Boot ROM
Slave
LBD
ICU
Slave
SFU
PSS
Slave
SCB
TE
USB Host
USB PHY
Bus
CPU Subsystem
TFU
USB
Device
DMA
Crtl
HCU
ISI
Slave
DNC
CPR
Slave
DWU
GPIO
Slave
LLU
Motor Control,
LSS, ISI,
LED, etc.
LSS
Master
Slave
Slave
PCU
PHI
Master
PEP Configuration Bus
Bi-lithic
Printhead

FIG. 85

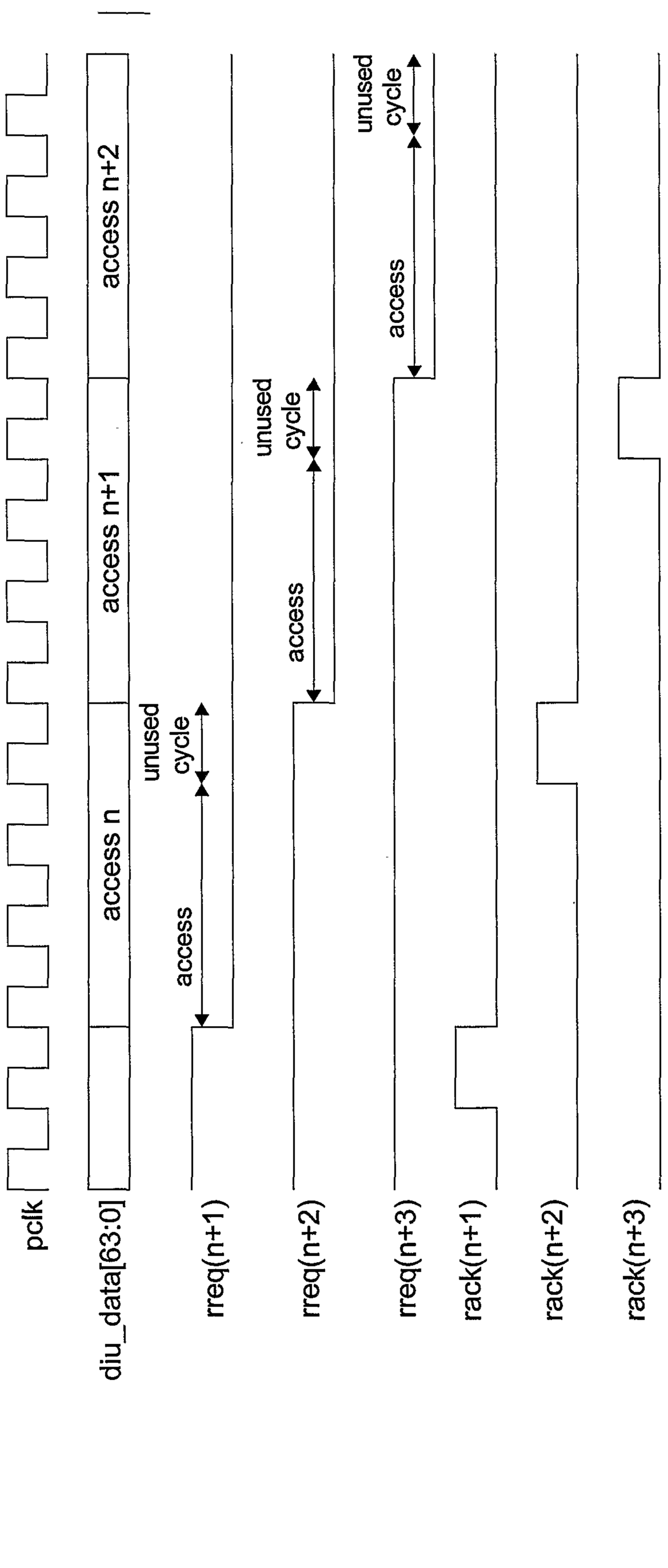

pclk
diu_data[63:0]
rreq(n+1)
rreq(n+2)
rreq(n+3)
rack(n+1)
rack(n+2)
rack(n+3)
access n
unused cycle
access
access n+1
unused cycle
access
access n+2
unused cycle
access

FIG. 86

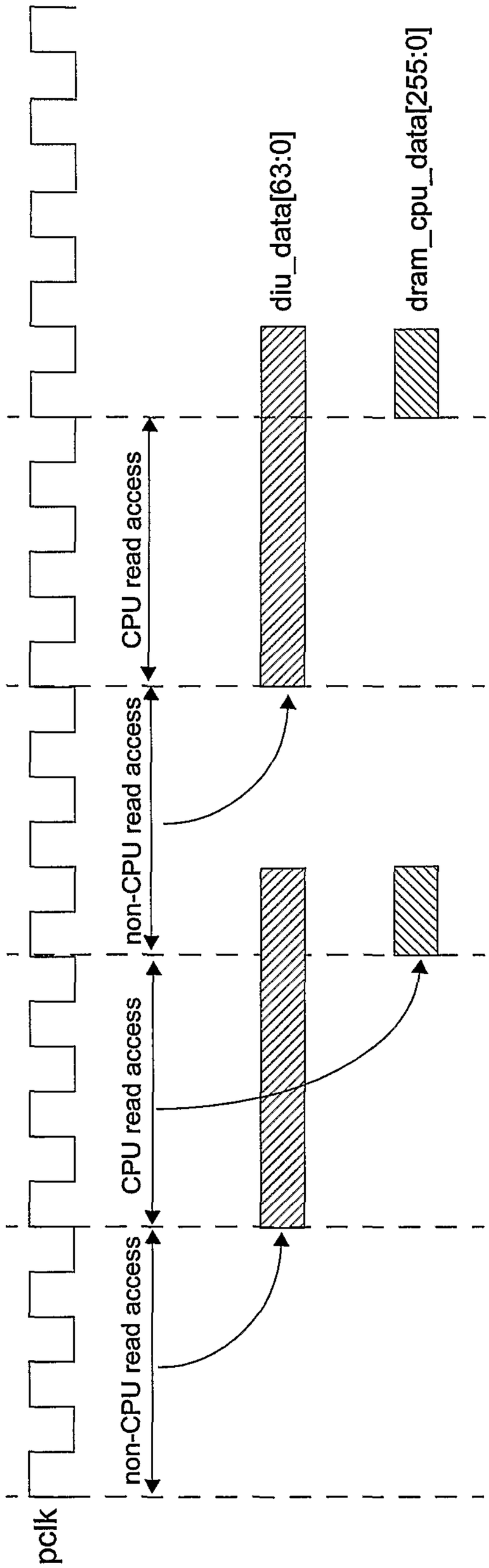

pclk
non-CPU read access
CPU read access
non-CPU read access
CPU read access
diu_data[63:0]
dram_cpu_data[255:0]

FIG. 87

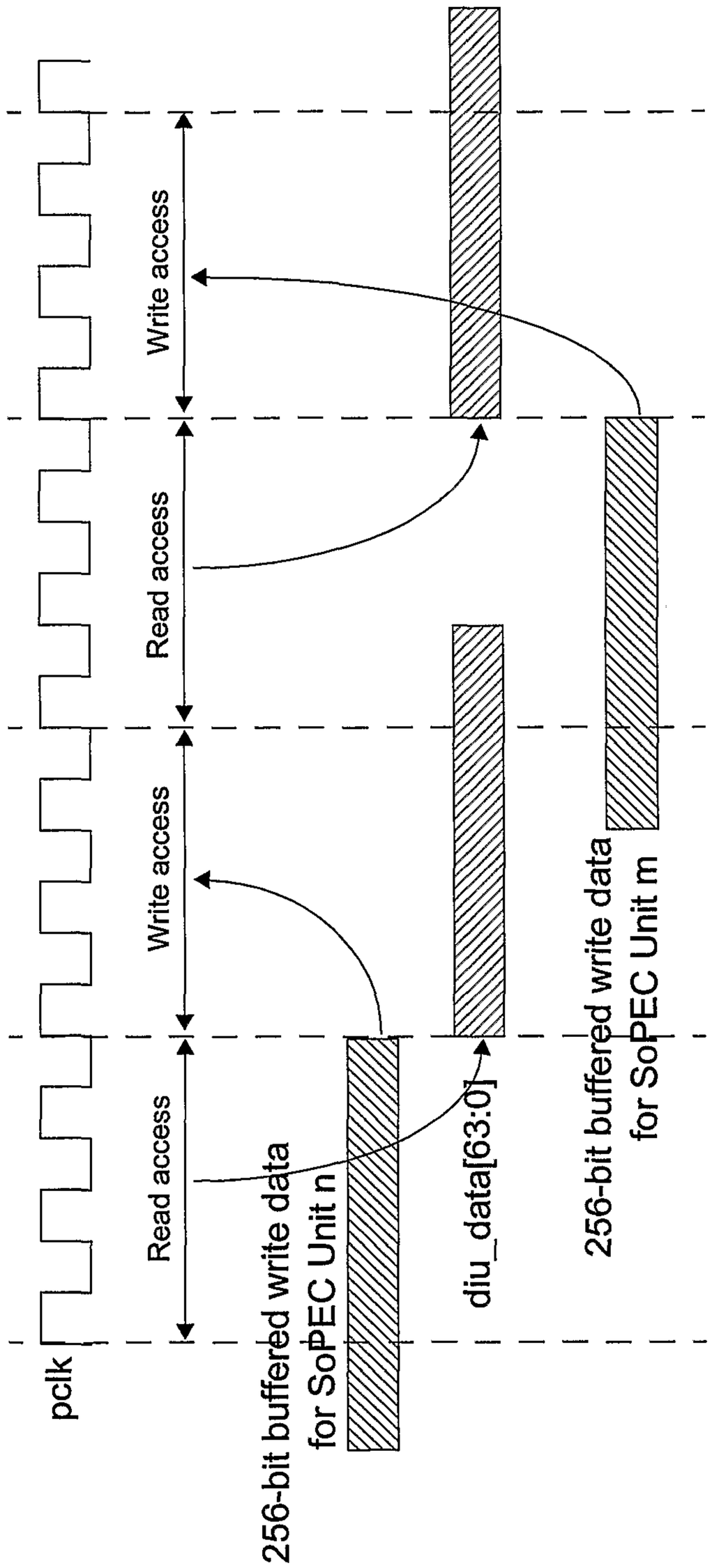

Write access
Read access
Write access
Read access
pclk
256-bit buffered write data for SoPEC Unit n
diu_data[63:0]
256-bit buffered write data for SoPEC Unit m

FIG. 88

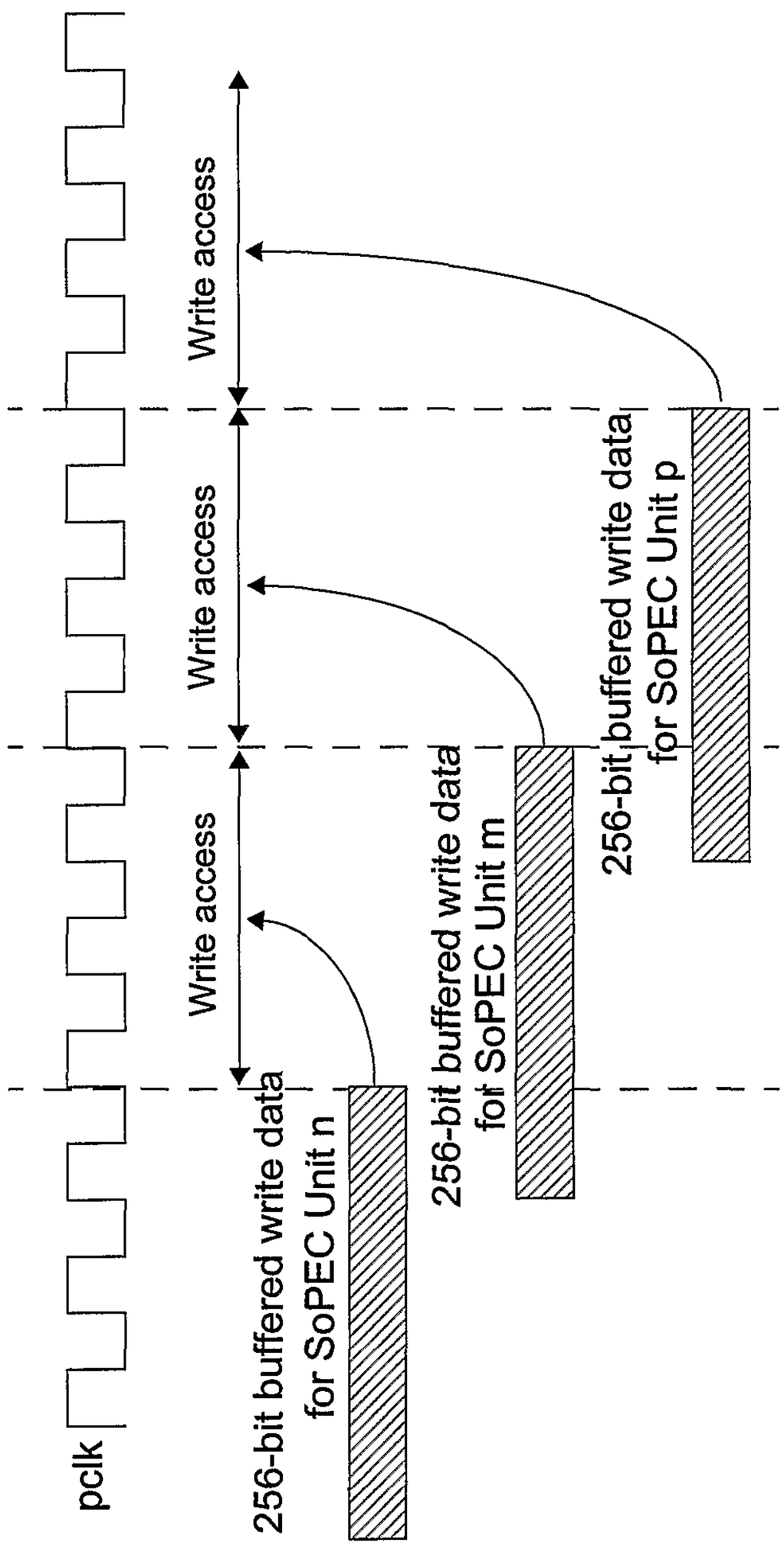

pclk
Write access
Write access
Write access
256-bit buffered write data for SoPEC Unit n
256-bit buffered write data for SoPEC Unit m
256-bit buffered write data for SoPEC Unit p

FIG. 89

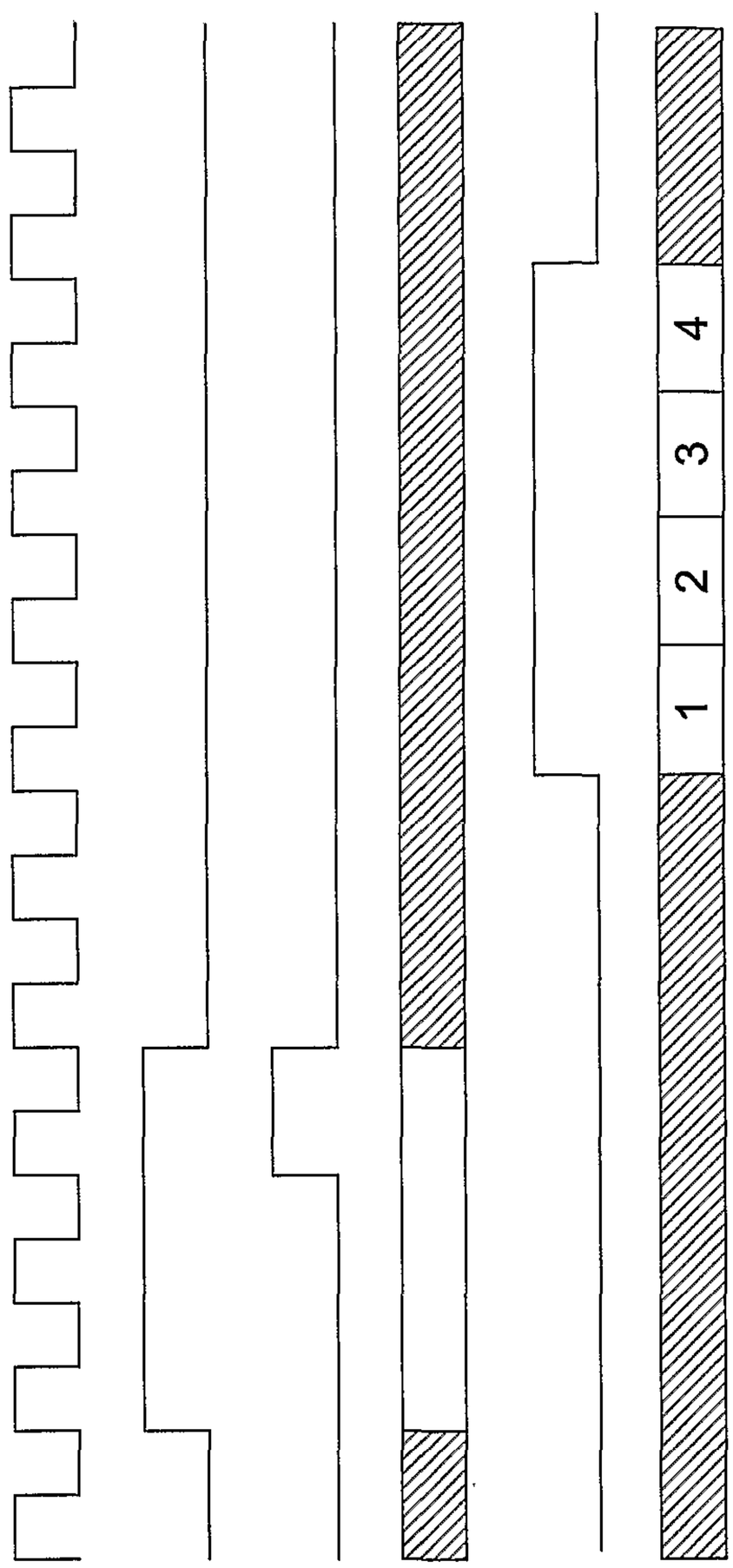

pclk
<unit>_diu_rreq
diu_<unit>_rack
<unit>_diu_radr[21:5]
diu_<unit>_rvalid
diu_data[63:0]
1
2
3
4

FIG. 90

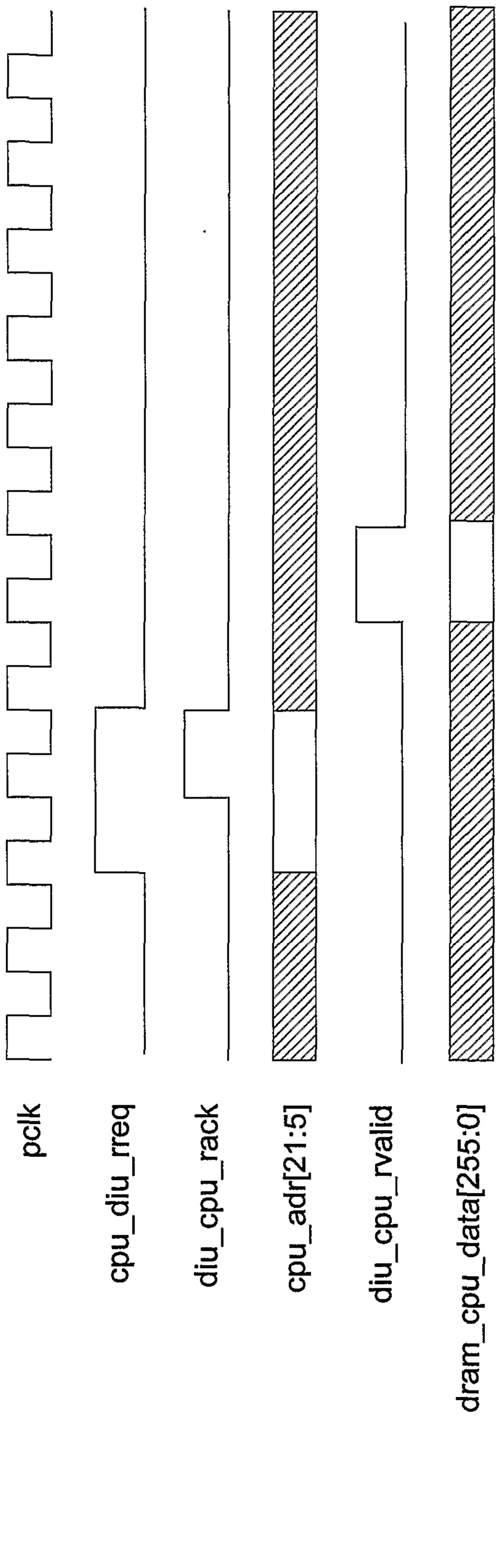

pclk
cpu_diu_rreq
diu_cpu_rack
cpu_adr[21:5]
diu_cpu_rvalid
dram_cpu_data[255:0]

FIG. 91

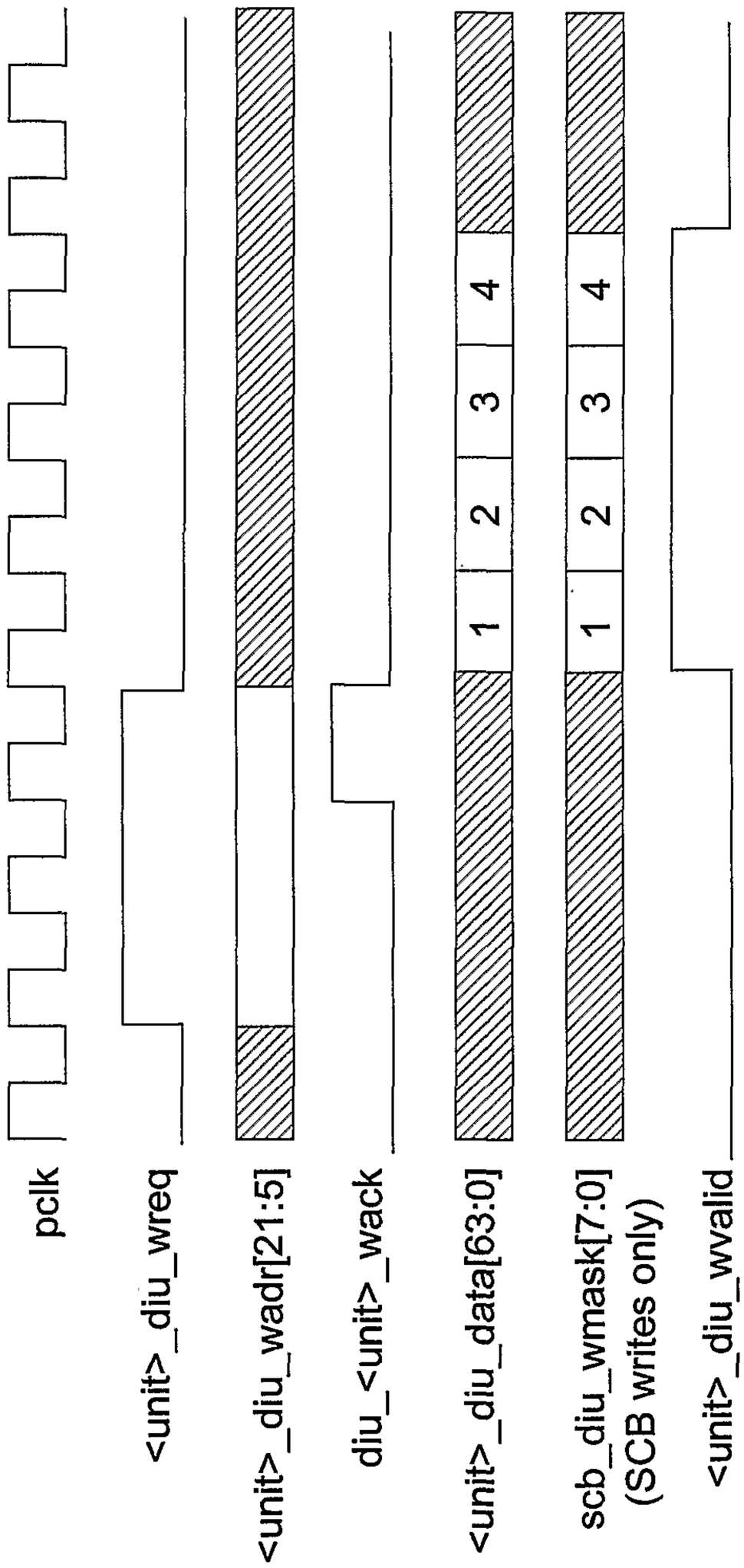

pclk
<unit>_diu_wreq
<unit>_diu_wadr[21:5]
diu_<unit>_wack
<unit>_diu_data[63:0]
scb_diu_wmask[7:0]
(SCB writes only)
<unit>_diu_wvalid
4
3
2
1

FIG. 92

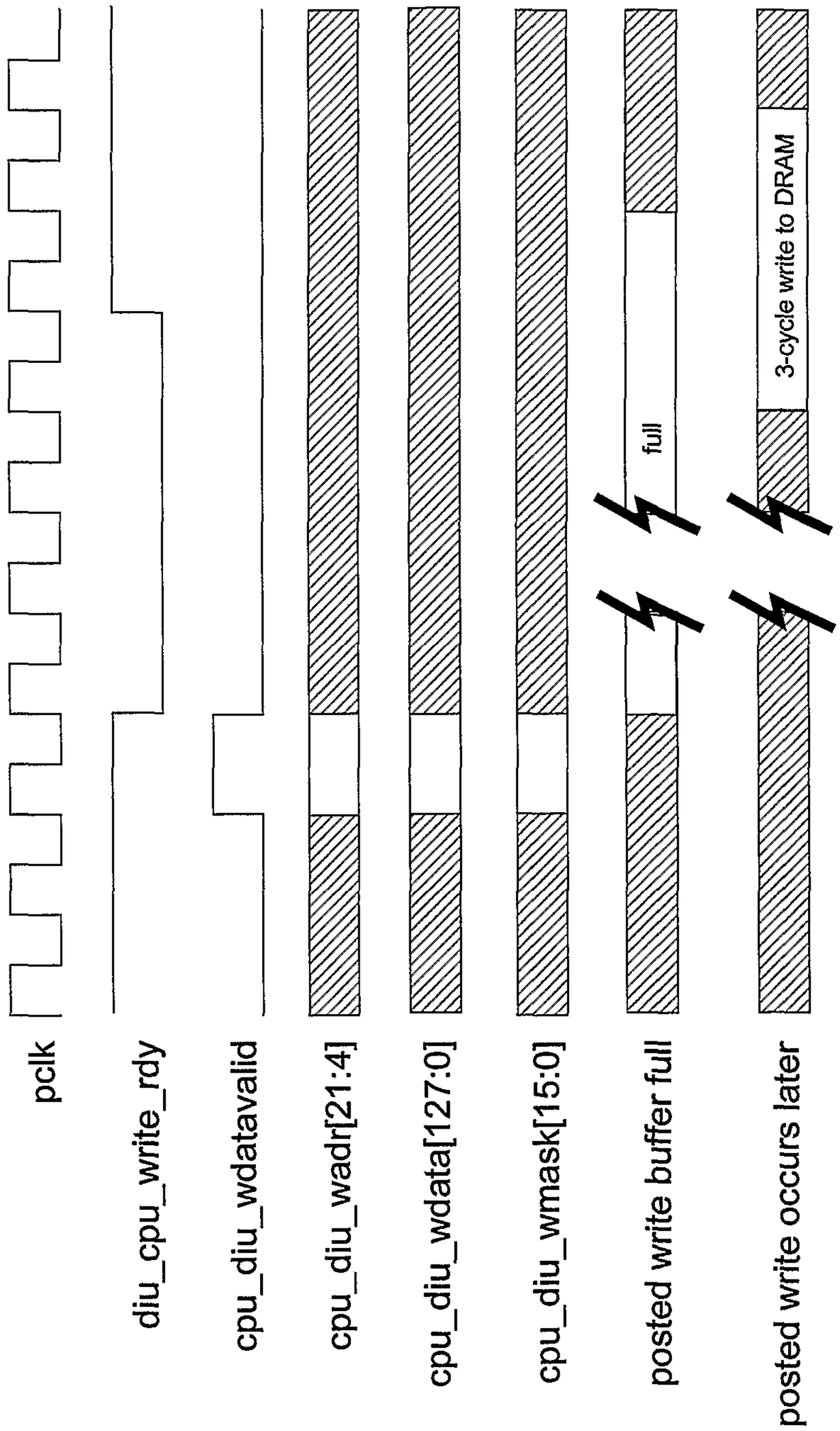

pclk
diu_cpu_write_rdy
cpu_diu_wdatavalid
cpu_diu_wadr[21:4]
cpu_diu_wdata[127:0]
cpu_diu_wmask[15:0]
posted write buffer full
posted write occurs later
full
3-cycle write to DRAM

FIG. 93

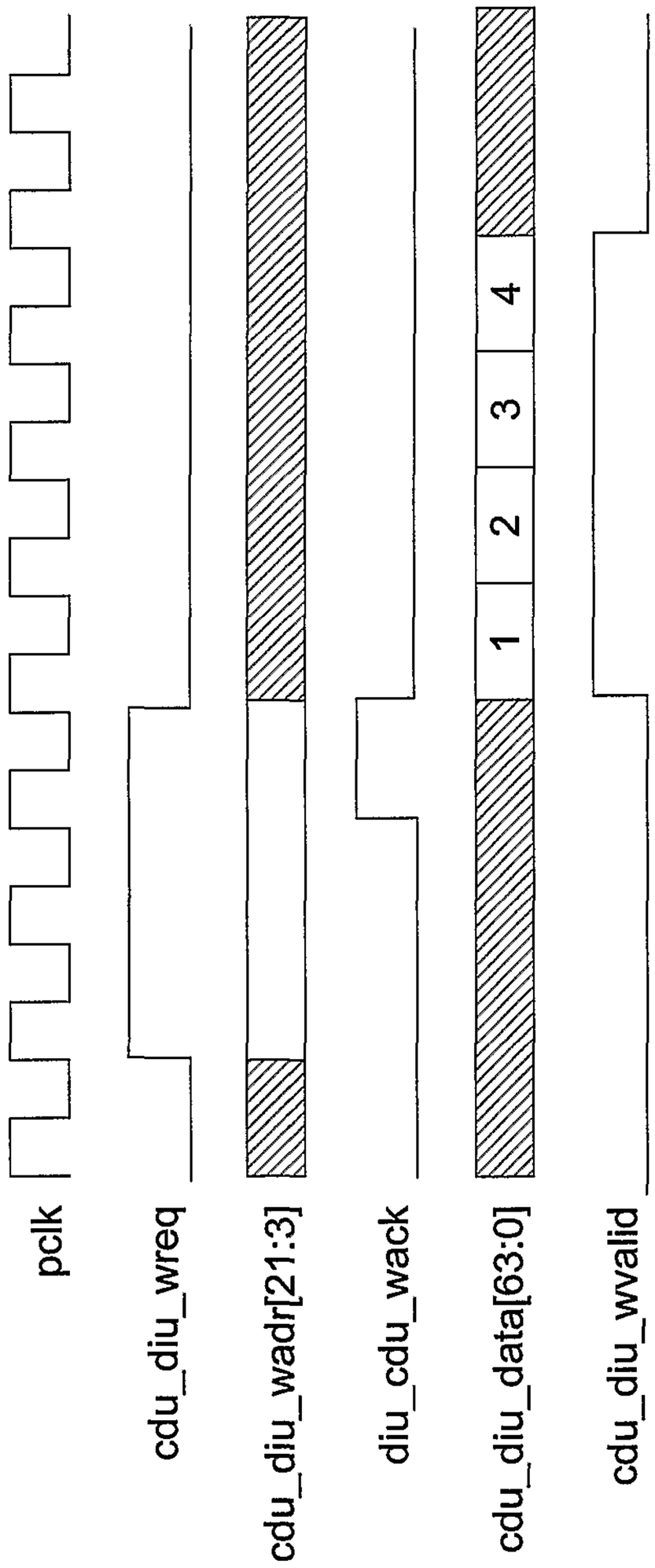

pclk
cdu_diu_wreq
diu_wadr[21:3]
diu_cdu_wack
diu_data[63:0]
du_diu_wvalid
1
2
3
4

FIG. 94

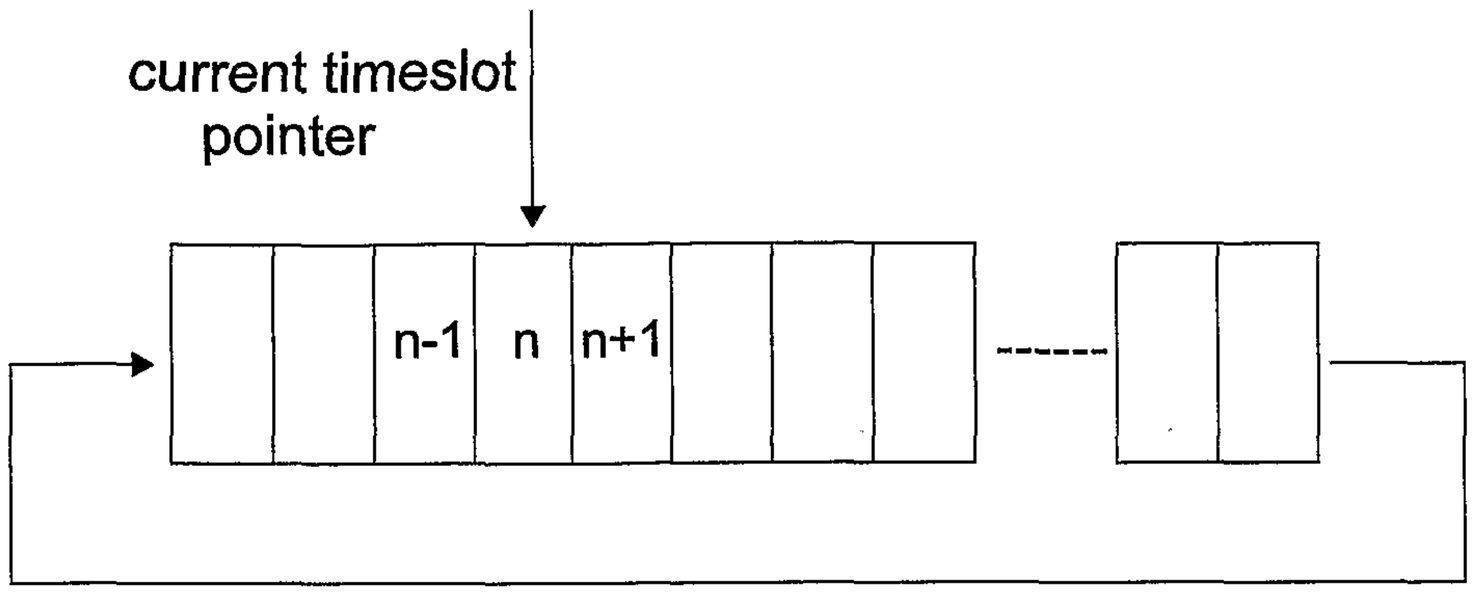


FIG. 95

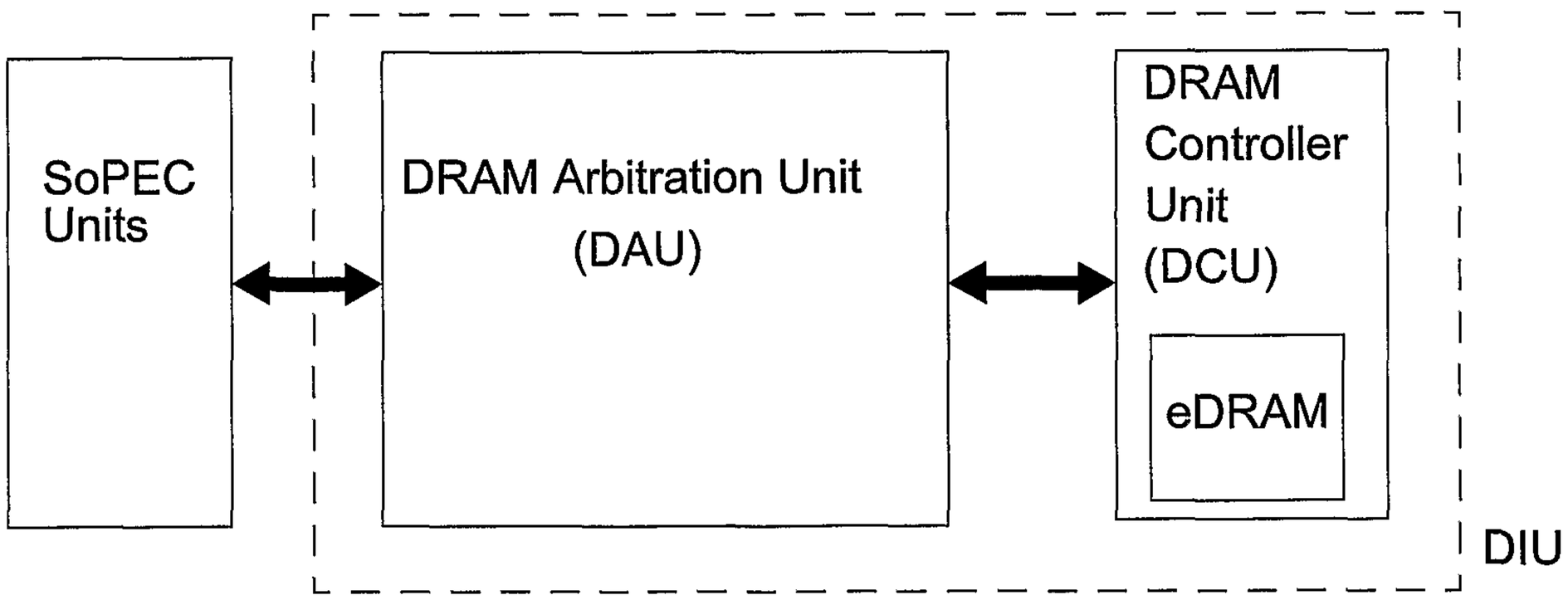


FIG. 100

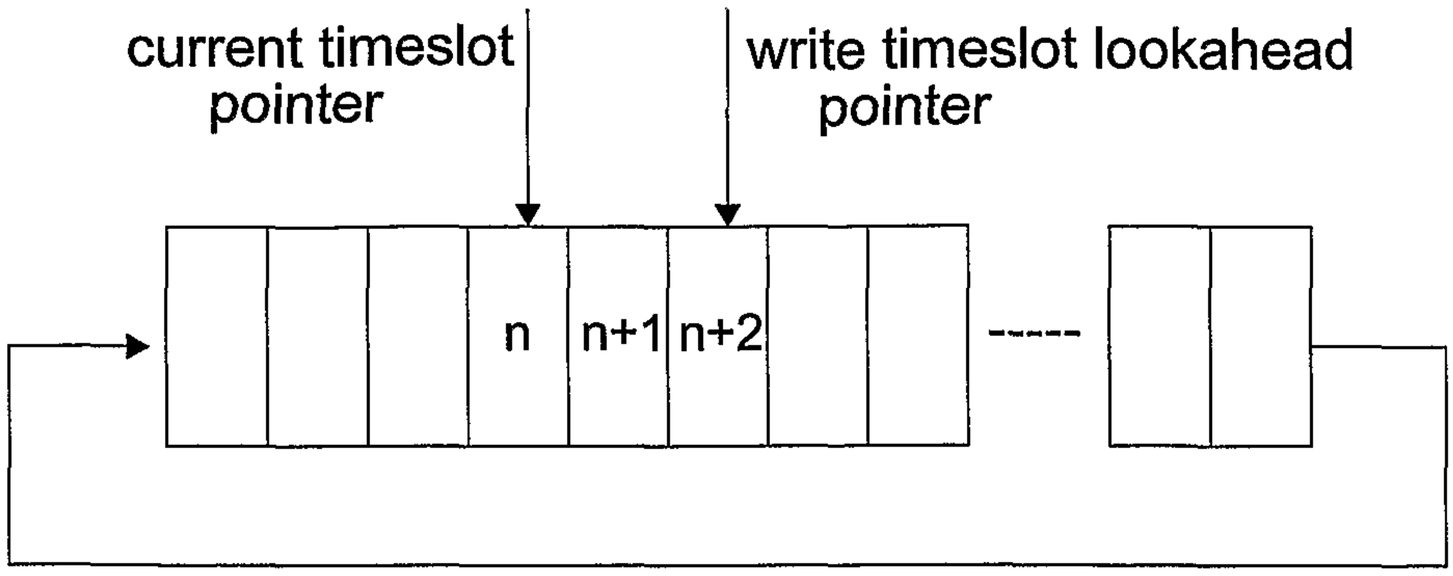

current timeslot pointer
write timeslot lookahead pointer
n
n+1
n+2

FIG. 96

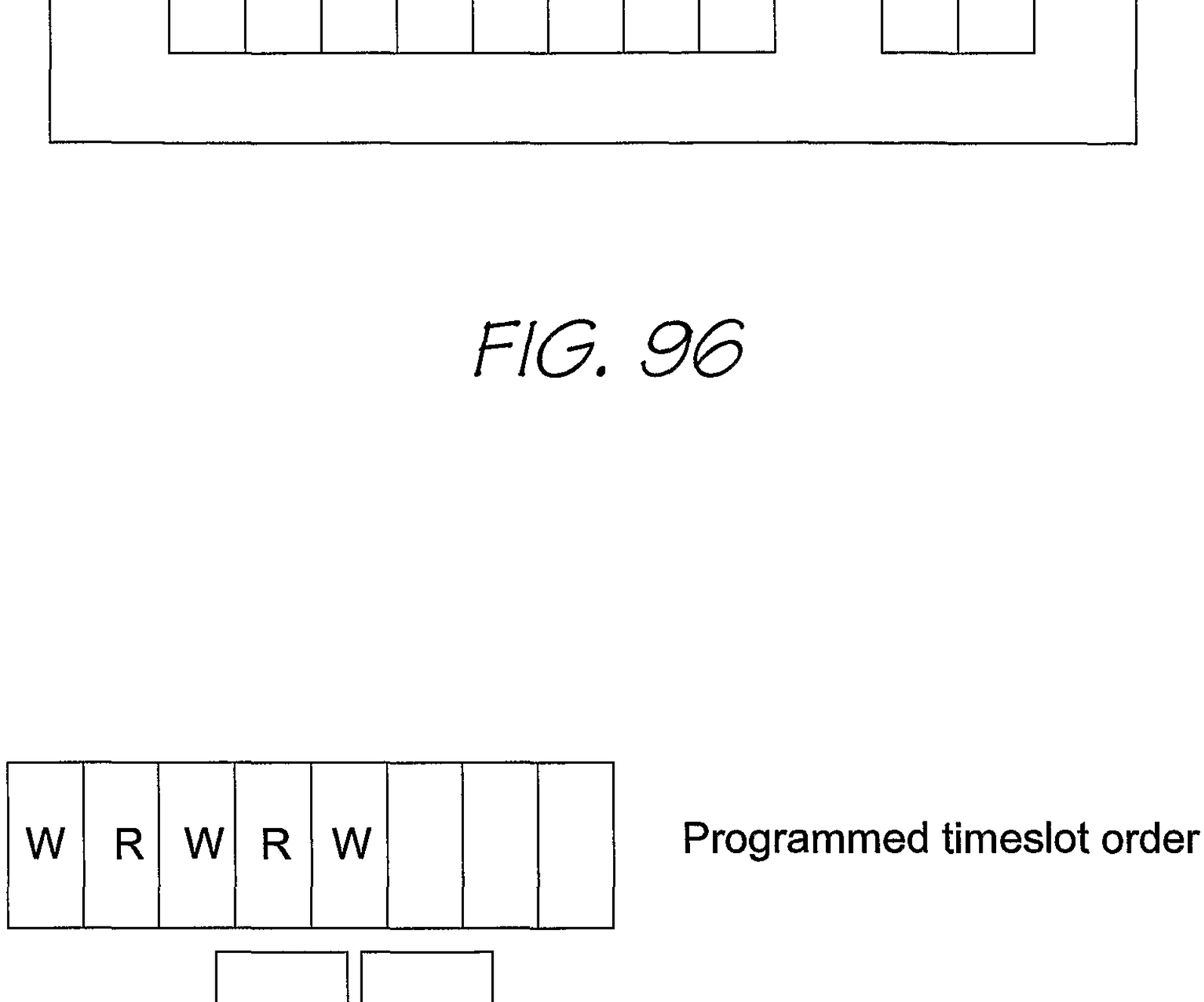

W R W R W
W W R W R
W R W R W
Programmed timeslot order
Timeslot arbitration order
Actual timeslot order
write latency

FIG. 97

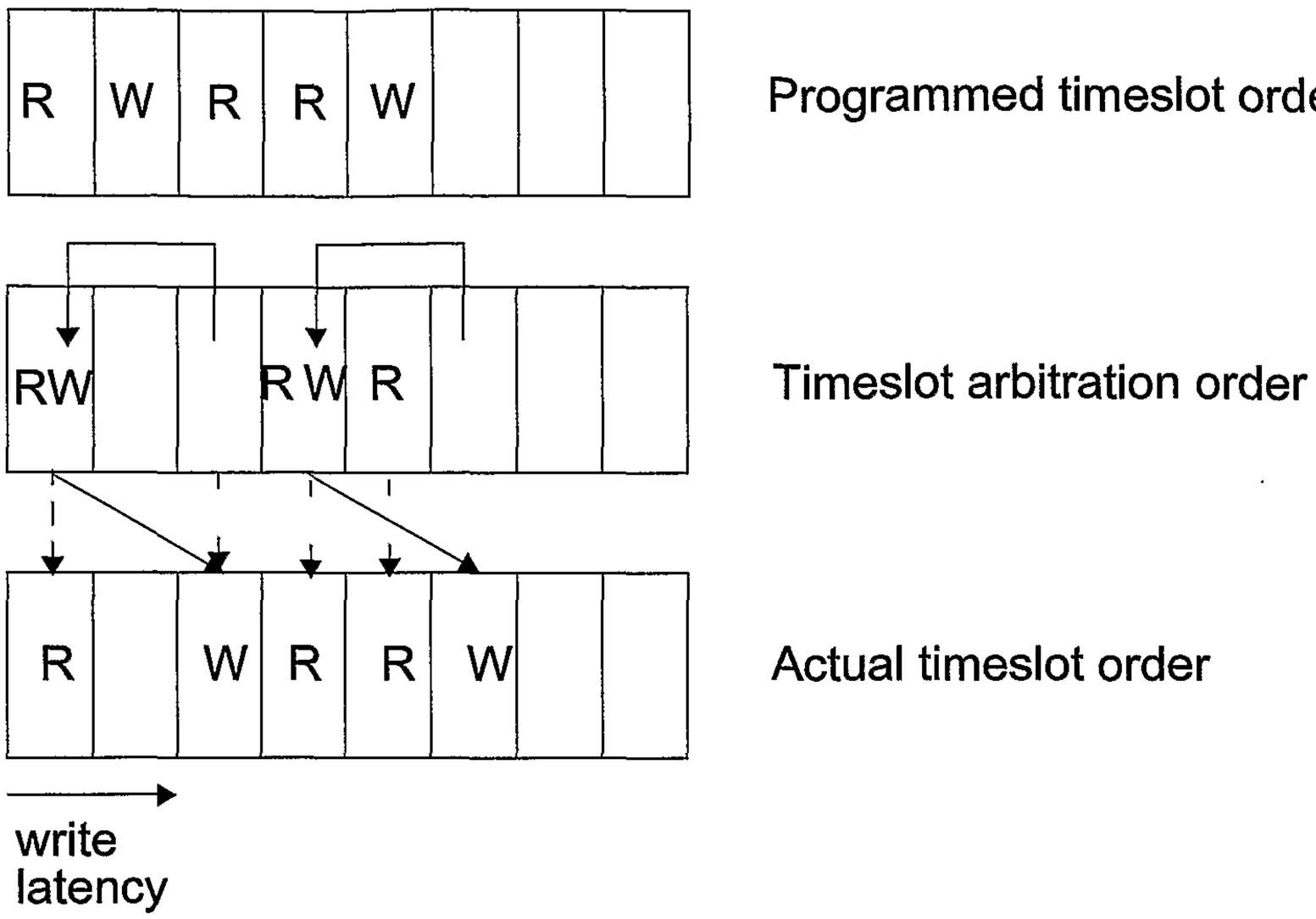

R W R R W
Programmed timeslot order
RW R W R
Timeslot arbitration order
R W R R W
Actual timeslot order
write latency

FIG. 98

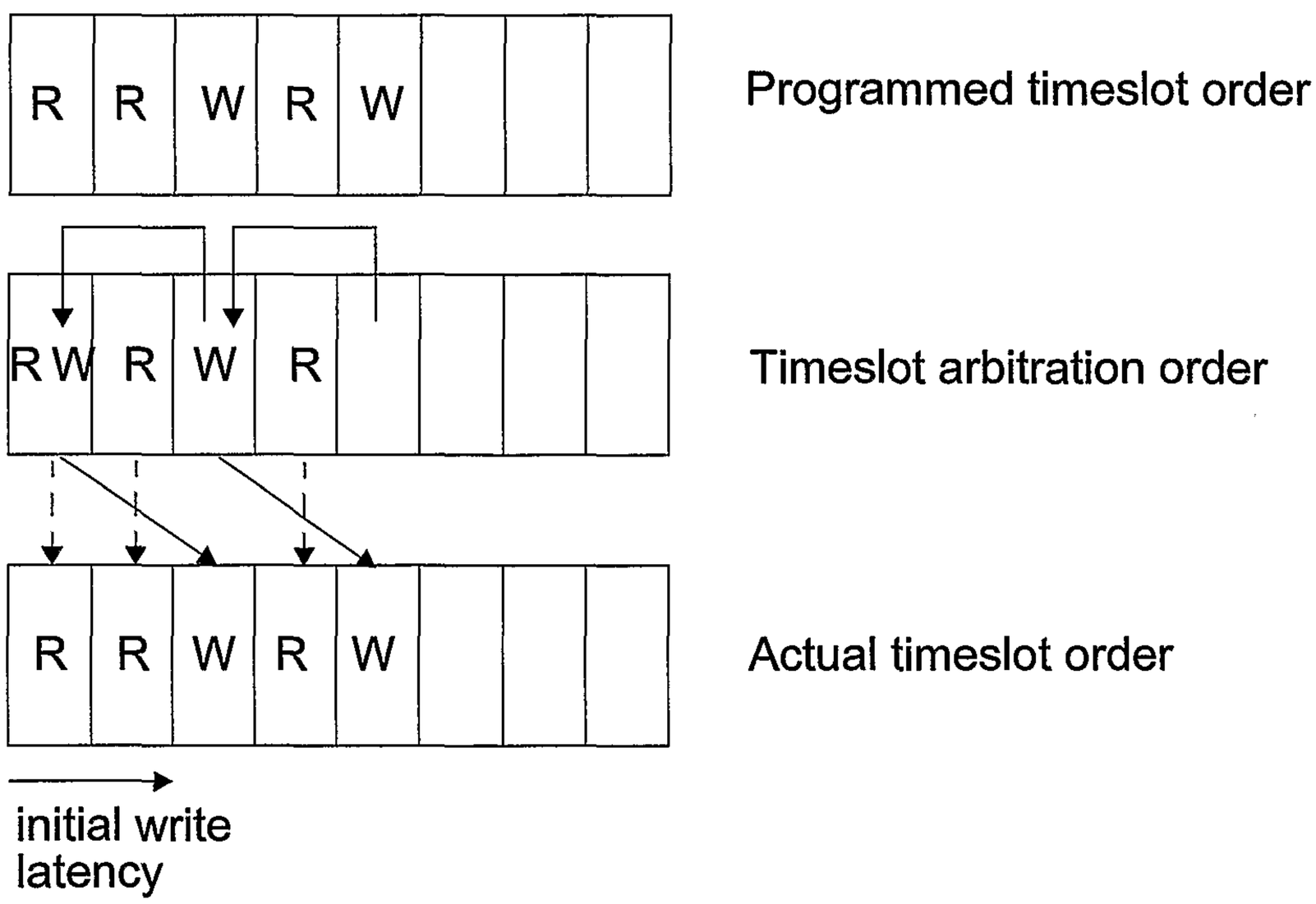

R R W R W
Programmed timeslot order
R W R W R
Timeslot arbitration order
R R W R W
Actual timeslot order
initial write latency

FIG. 99

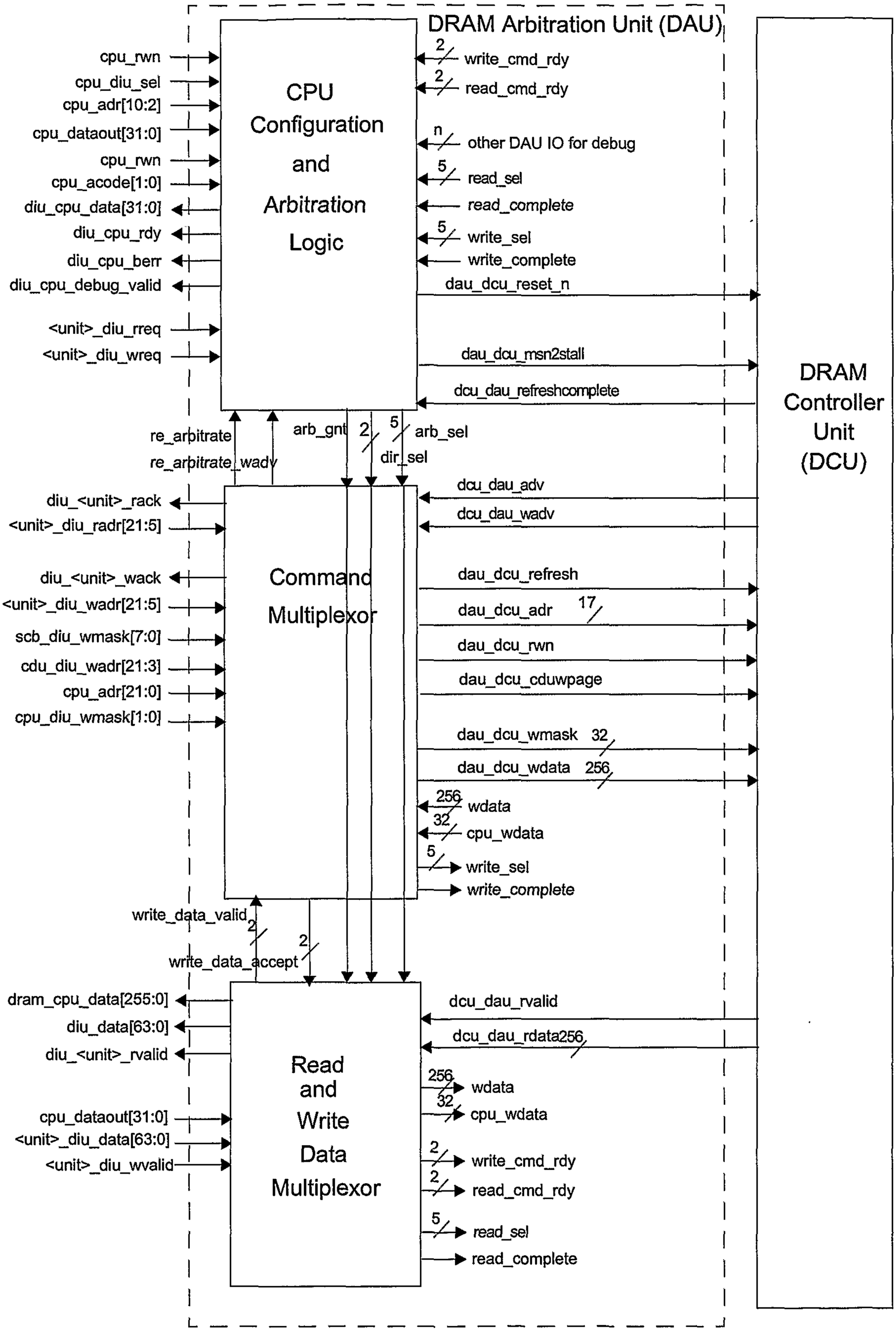

DRAM Arbitration Unit (DAU)
CPU Configuration and Arbitration Logic
Command Multiplexor
Read and Write Data Multiplexor
DRAM Controller Unit (DCU)
cpu_rwn
cpu_diu_sel
cpu_adr[10:2]
cpu_dataout[31:0]
cpu_rwn
cpu_acode[1:0]
diu_cpu_data[31:0]
diu_cpu_rdy
diu_cpu_berr
diu_cpu_debug_valid
<unit>_diu_rreq
<unit>_diu_wreq
re_arbitrate
re_arbitrate_wadv
diu_<unit>_rack
<unit>_diu_radr[21:5]
diu_<unit>_wack
<unit>_diu_wadr[21:5]
scb_diu_wmask[7:0]
cdu_diu_wadr[21:3]
cpu_adr[21:0]
cpu_diu_wmask[1:0]
write_data_valid
write_data_accept
dram_cpu_data[255:0]
diu_data[63:0]
diu_<unit>_rvalid
cpu_dataout[31:0]
<unit>_diu_data[63:0]
<unit>_diu_wvalid
2 write_cmd_rdy
2 read_cmd_rdy
n other DAU IO for debug
5 read_sel
read_complete
5 write_sel
write_complete
dau_dcu_reset_n
dau_dcu_msn2stall
dcu_dau_refreshcomplete
arb_gnt 2
5 arb_sel
dir_sel
dcu_dau_adv
dcu_dau_wadv
dau_dcu_refresh
dau_dcu_adr 17
dau_dcu_rwn
dau_dcu_cduwpage
dau_dcu_wmask 32
dau_dcu_wdata 256
256 wdata
32 cpu_wdata
5 write_sel
write_complete
dcu_dau_rvalid
dcu_dau_rdata256
256 wdata
32 cpu_wdata
2 write_cmd_rdy
2 read_cmd_rdy
5 read_sel
read_complete

FIG. 101

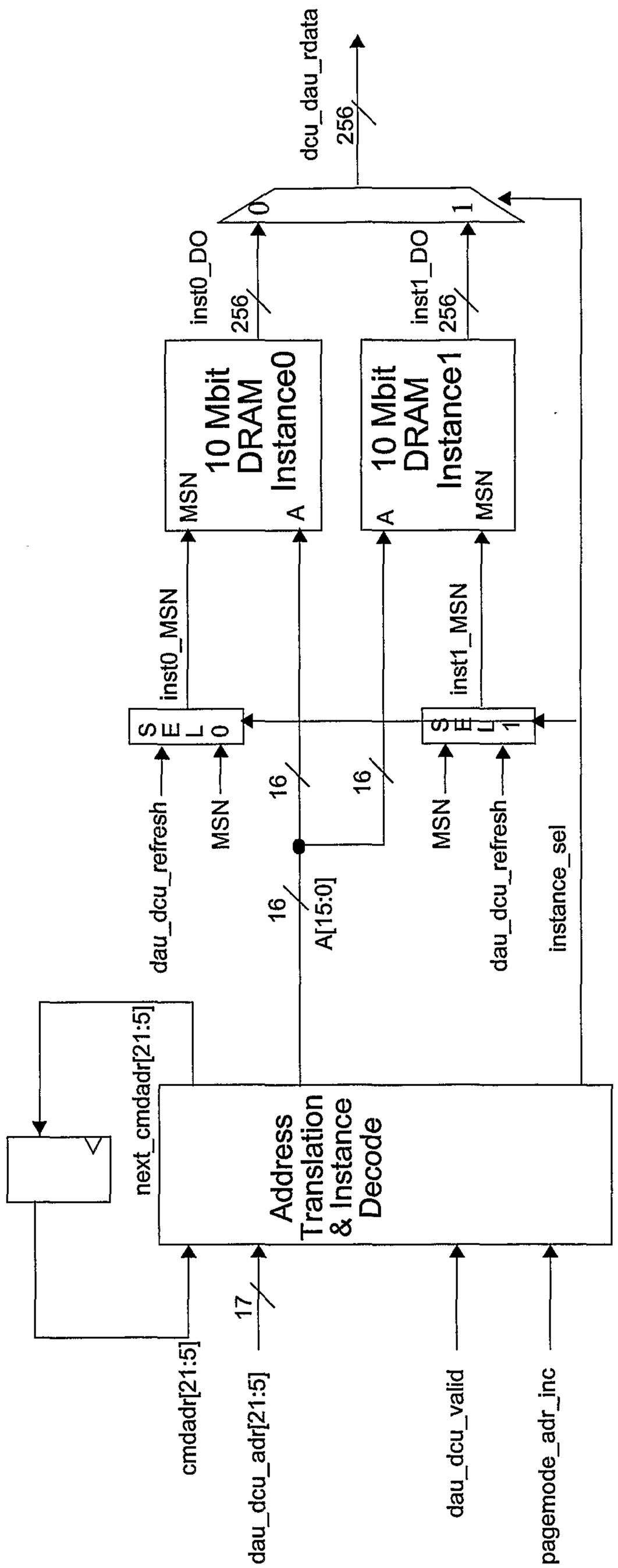

dcu_dau_rdata
256
0
1
inst0_DO
inst1_DO
256
256
10 Mbit DRAM Instance0
10 Mbit DRAM Instance1
MSN
A
A
MSN
inst0_MSN
inst1_MSN
S E L 0
S E L 1
dau_dcu_refresh
MSN
16
16
16
A[15:0]
MSN
dau_dcu_refresh
instance_sel
next_cmdadr[21:5]
Address Translation & Instance Decode
cmdadr[21:5]
17
dau_dcu_adr[21:5]
dau_dcu_valid
pagemode_adr_inc

FIG. 102

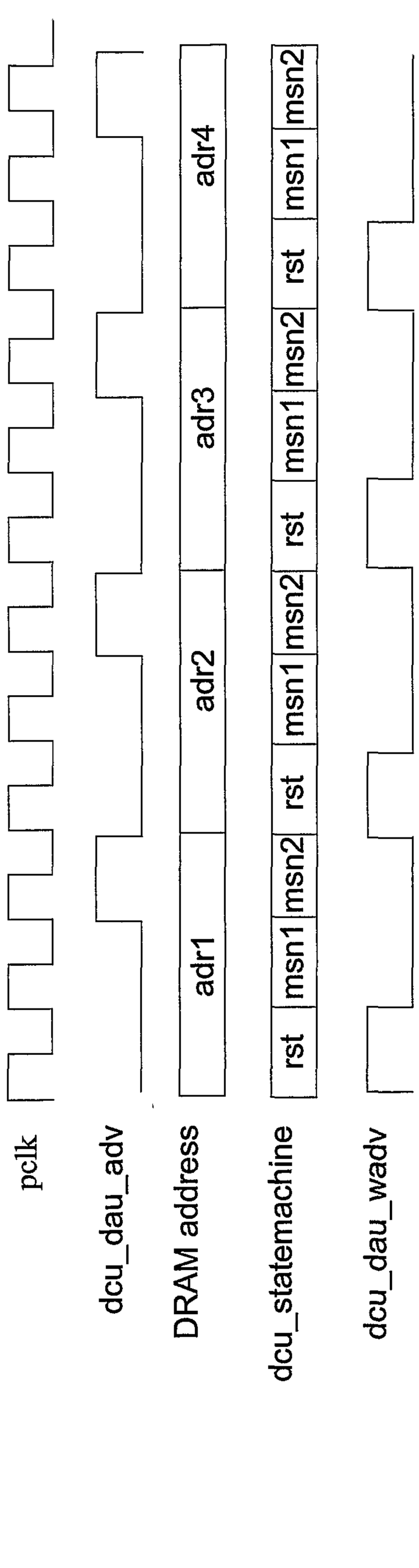

pclk
dcu_dau_adv
DRAM address
dcu_statemachine
dcu_dau_wadv
adr1
adr2
adr3
adr4
rst
msn1
msn2

FIG. 103

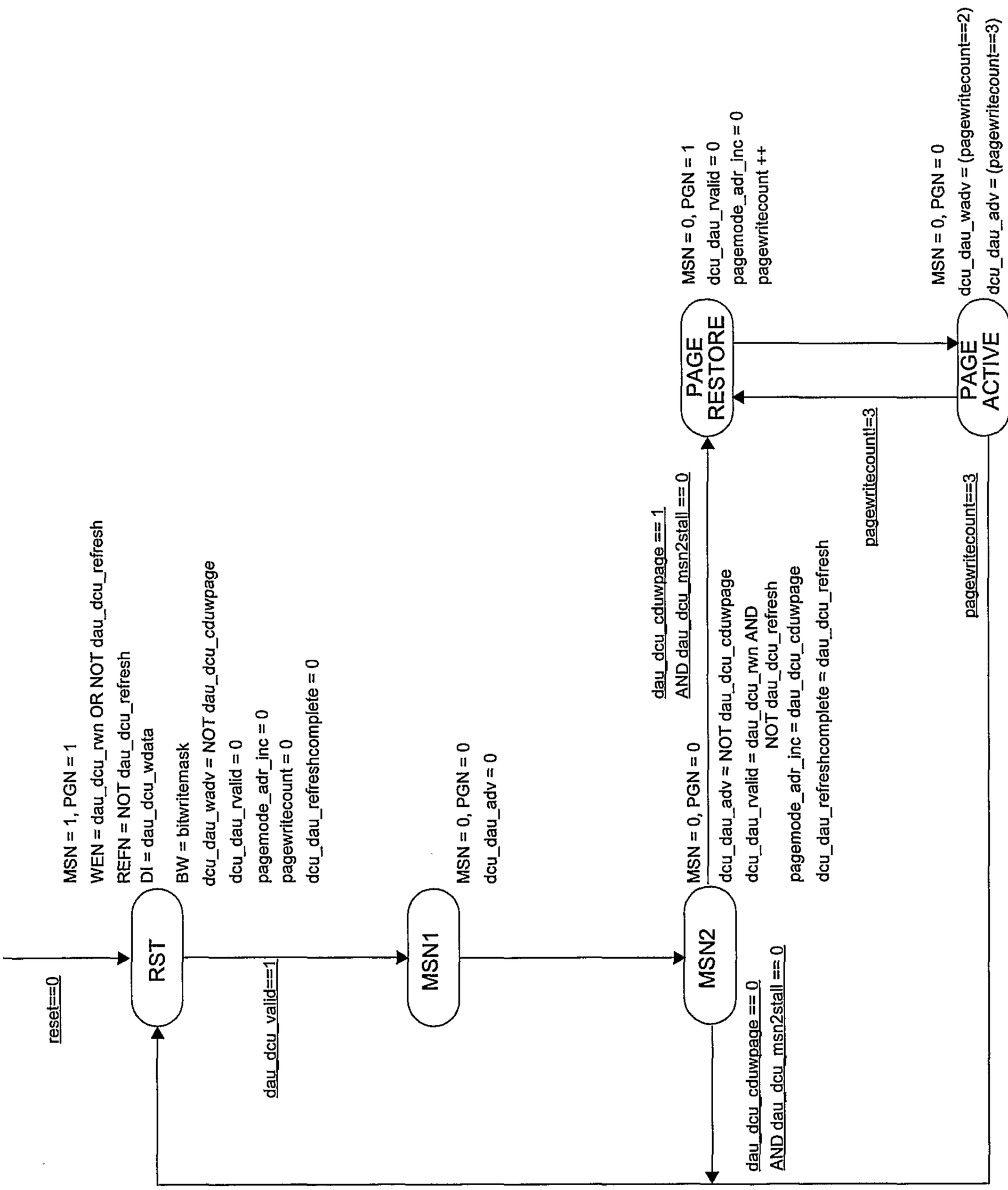

reset==0
RST
MSN = 1, PGN = 1
WEN = dau_dcu_rwn OR NOT dau_dcu_refresh
REFN = NOT dau_dcu_refresh
DI = dau_dcu_wdata
BW = bitwritemask
dcu_dau_wadv = NOT dau_dcu_cduwpage
dcu_dau_rvalid = 0
pagemode_adr_inc = 0
pagewritecount = 0
dcu_dau_refreshcomplete = 0
dau_dcu_valid==1
MSN1
MSN = 0, PGN = 0
dcu_dau_adv = 0
dau_dcu_cduwpage == 1
AND dau_dcu_msn2stall == 0
MSN2
MSN = 0, PGN = 0
dcu_dau_adv = NOT dau_dcu_cduwpage
dcu_dau_rvalid = dau_dcu_rwn AND
NOT dau_dcu_refresh
pagemode_adr_inc = dau_dcu_cduwpage
dcu_dau_refreshcomplete = dau_dcu_refresh
dau_dcu_cduwpage == 0
AND dau_dcu_msn2stall == 0
pagewritecount!=3
pagewritecount==3
PAGE RESTORE
MSN = 0, PGN = 1
dcu_dau_rvalid = 0
pagemode_adr_inc = 0
pagewritecount ++
PAGE ACTIVE
MSN = 0, PGN = 0
dcu_dau_wadv = (pagewritecount==2)
dcu_dau_adv = (pagewritecount==3)
pagemode_adr_inc = 1

FIG. 104

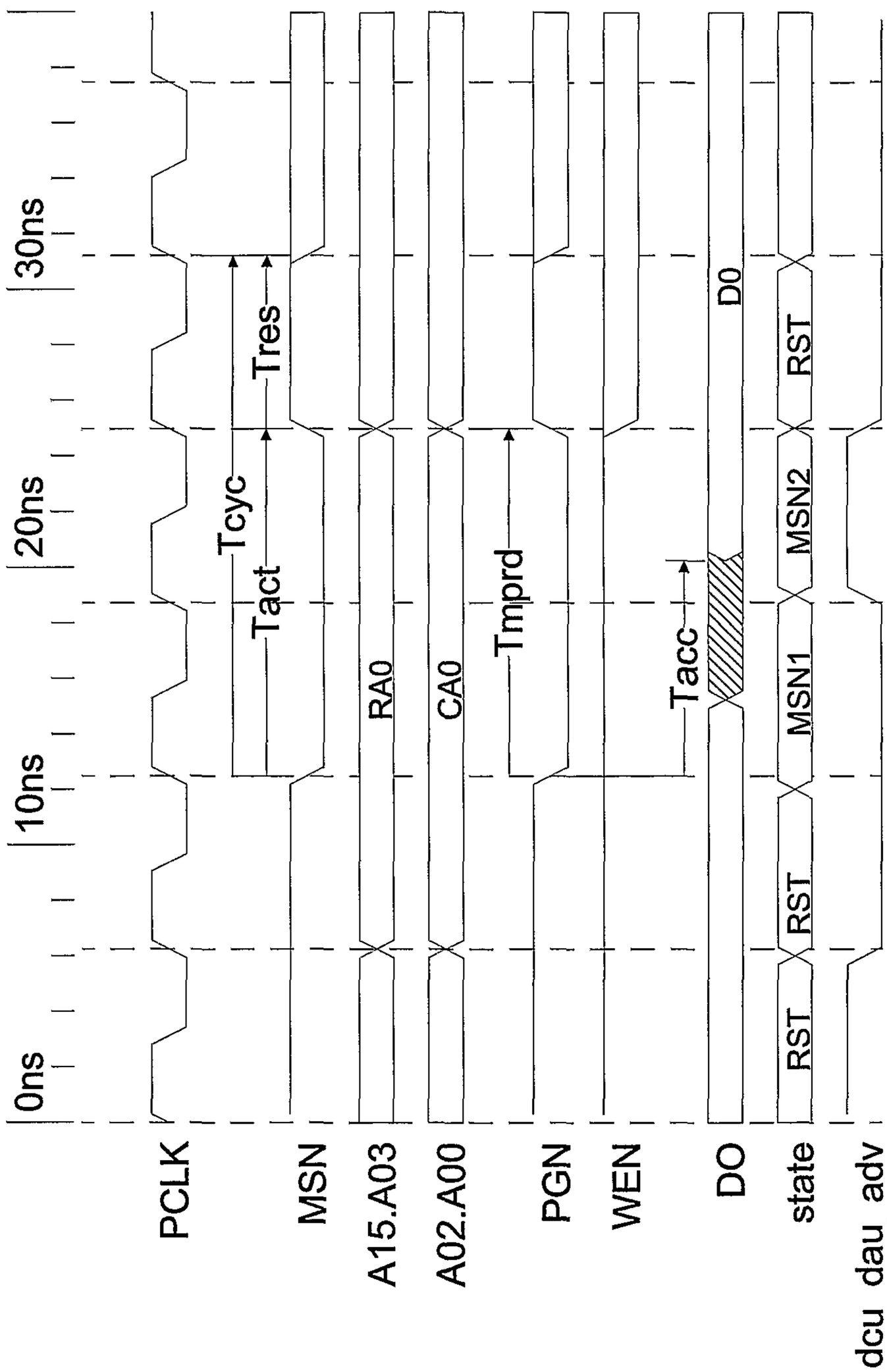

30ns
Tres
D0
RST
20ns
Tcyc
Tact
RA0
CA0
Tmprd
Tacc
MSN2
MSN1
RST
RST
10ns
0ns
PCLK
MSN
A15.A03
A02.A00
PGN
WEN
DO
state
dcu_dau_adv

FIG. 105

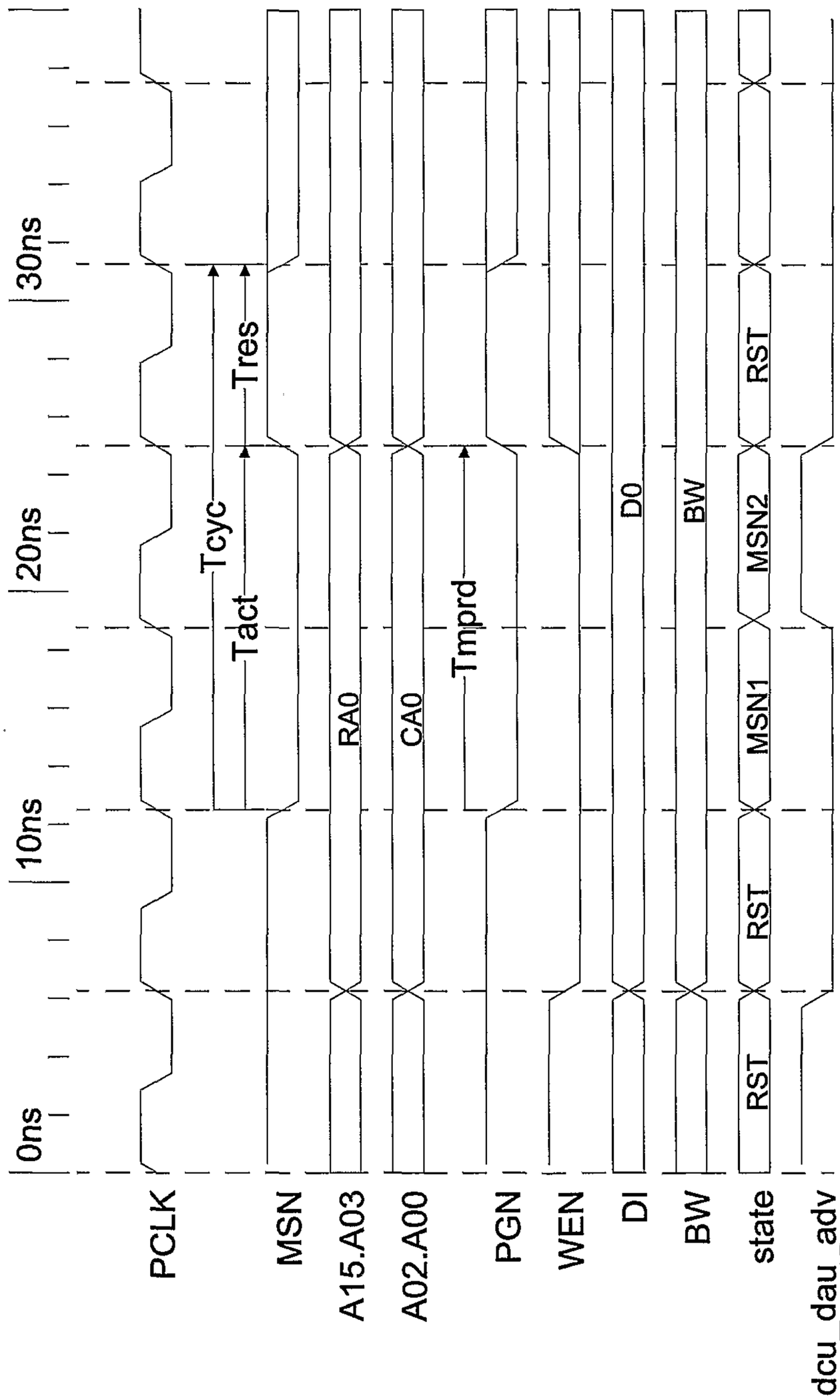

20ns
30ns
10ns
0ns
Tcyc
Tres
Tact
Tmprd
RA0
CA0
D0
BW
MSN2
RST
MSN1
RST
RST
PCLK
MSN
A15.A03
A02.A00
PGN
WEN
DI
BW
state
dcu_dau_adv

FIG. 106

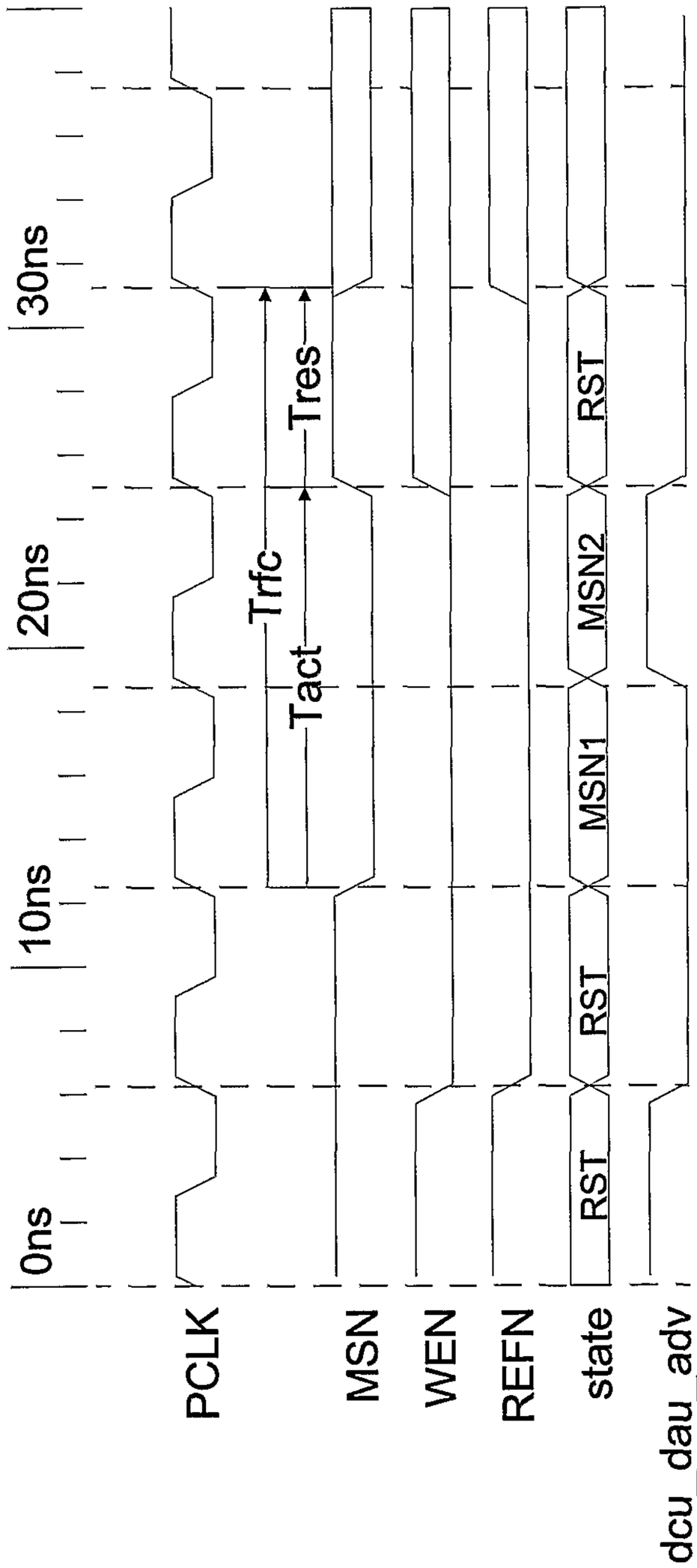

0ns
10ns
20ns
30ns
PCLK
MSN
WEN
REFN
state
dcu_dau_adv
RST
RST
RST
MSN1
MSN2
Tact
Trfc
Tres

FIG. 107

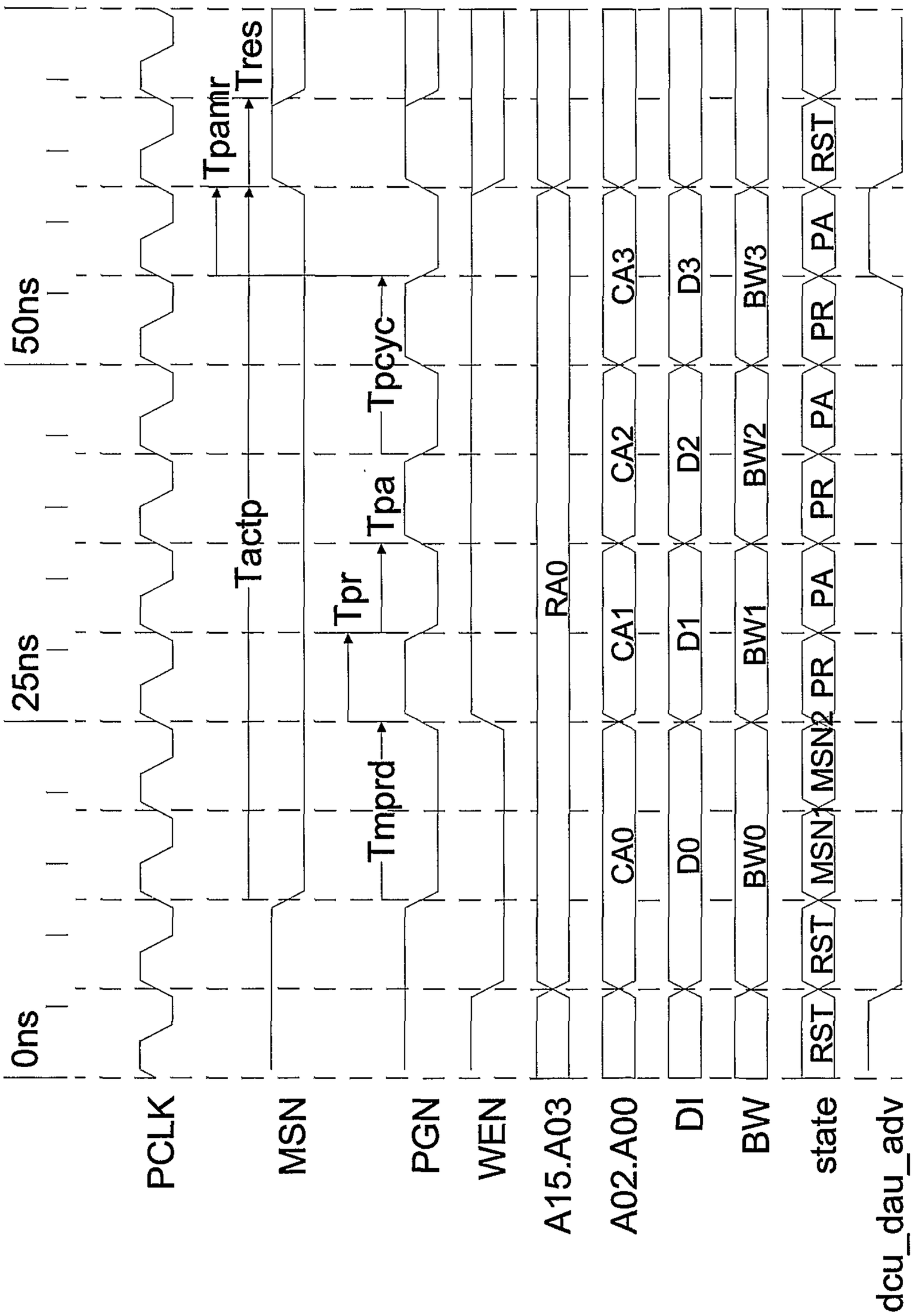

50ns
Tpamr
Tres
Tpcyc
CA3
D3
BW3
PA
PR
PA
RST
25ns
Tactp
Tpr
Tpa
RA0
CA2
D2
BW2
0ns
Tmprd
CA1
D1
BW1
CA0
D0
BW0
RST
MSN1
MSN2
PR
PA
RST
PCLK
MSN
PGN
WEN
A15.A03
A02.A00
DI
BW
state
dcu_dau_adv

FIG. 108

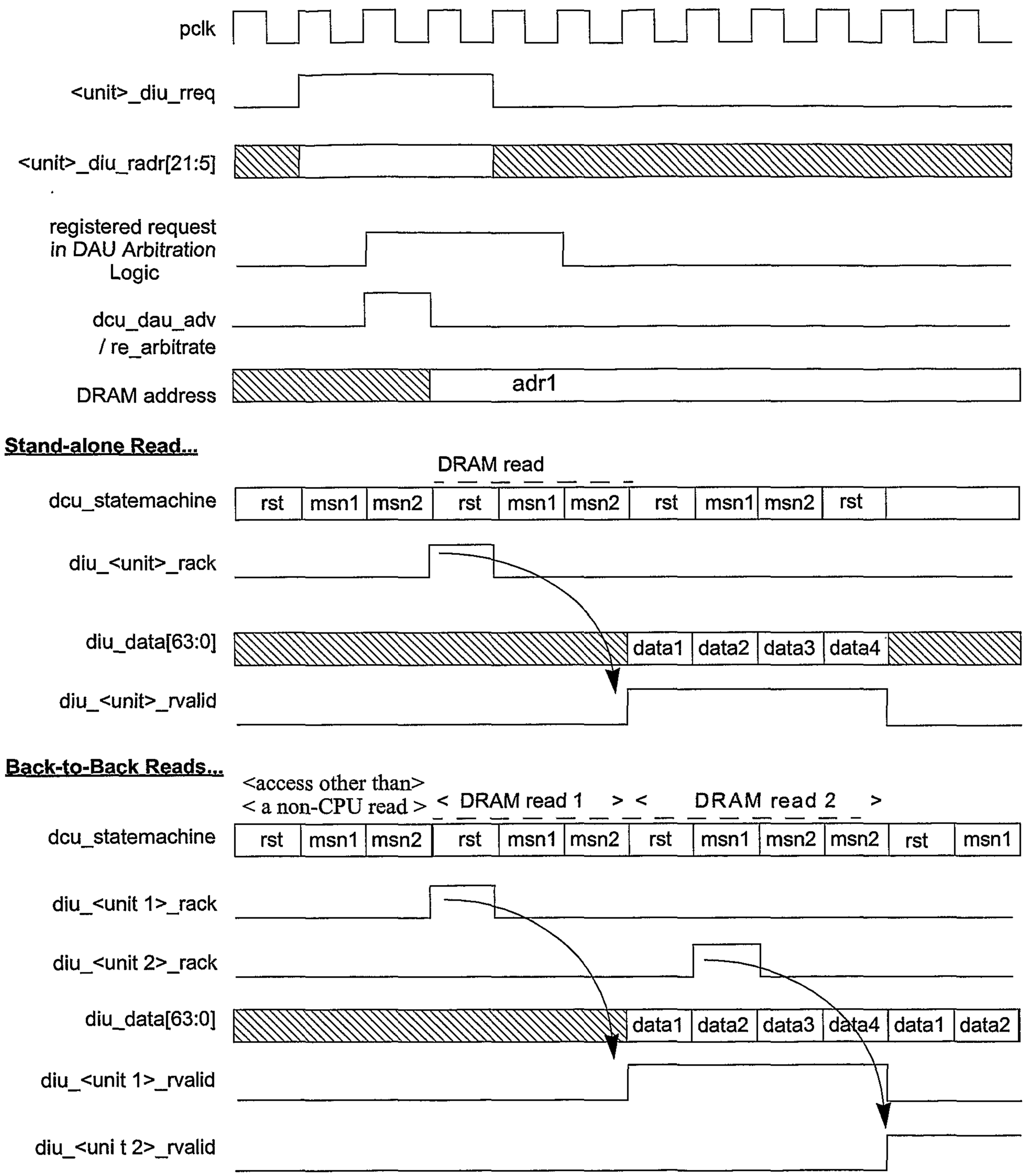

pclk
<unit>_diu_rreq
<unit>_diu_radr[21:5]
registered request in DAU Arbitration Logic
dcu_dau_adv / re_arbitrate
DRAM address
adr1
Stand-alone Read...
DRAM read
dcu_statemachine
rst
msn1
msn2
rst
msn1
msn2
rst
msn1
msn2
rst
diu_<unit>_rack
diu_data[63:0]
data1
data2
data3
data4
diu_<unit>_rvalid
Back-to-Back Reads...
<access other than>
< a non-CPU read >
< DRAM read 1 >
< DRAM read 2 >
dcu_statemachine
rst
msn1
msn2
rst
msn1
msn2
rst
msn1
msn2
msn2
rst
msn1
diu_<unit 1>_rack
diu_<unit 2>_rack
diu_data[63:0]
data1
data2
data3
data4
data1
data2
diu_<unit 1>_rvalid
diu_<uni t 2>_rvalid

FIG. 109

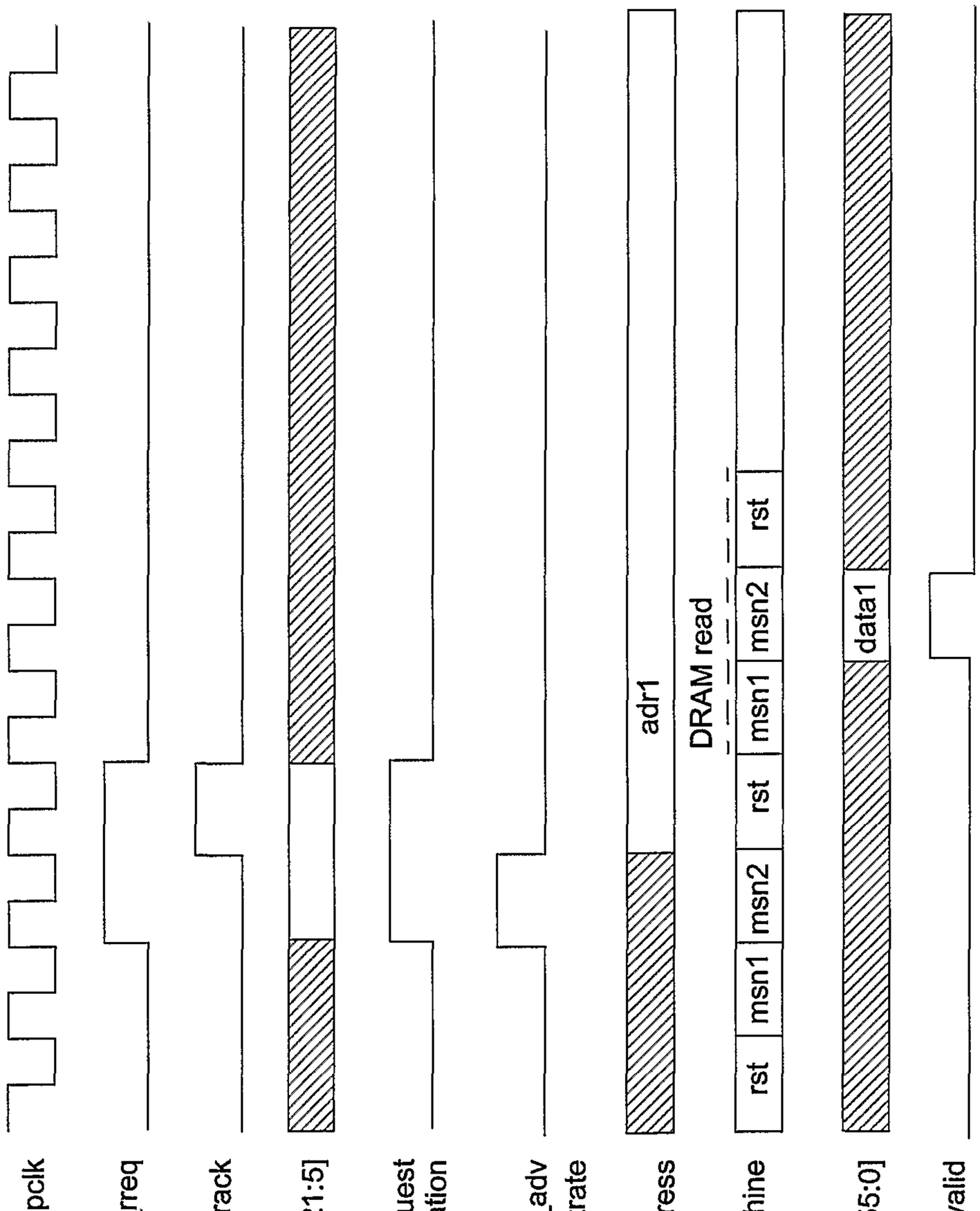

pclk
cpu_diu_rreq
diu_cpu_rack
cpu_adr[21:5]
un-registered request in DAU Arbitration Logic
dcu_dau_adv / re-arbitrate
DRAM address
dcu_statemachine
dram_diu_data[255:0]
diu_cpu_rvalid
adr1
DRAM read
rst
msn1
msn2
data1

FIG. 110

CPU MMU logic delay	CPU generates request	Arbitration occurs	cycle 1	DCU Address setup cycle (RST)	cycle 2	DRAM access (MSN1)	Tacc = 8 ns	cycle 3	DRAM access (MSN2)	cycle 4	CPU captures read data in AHB bridge	cycle 5	CPU MMU logic delay	CPU generates request	Arbitration occurs	cycle 6

FIG. 111

FIG. 112

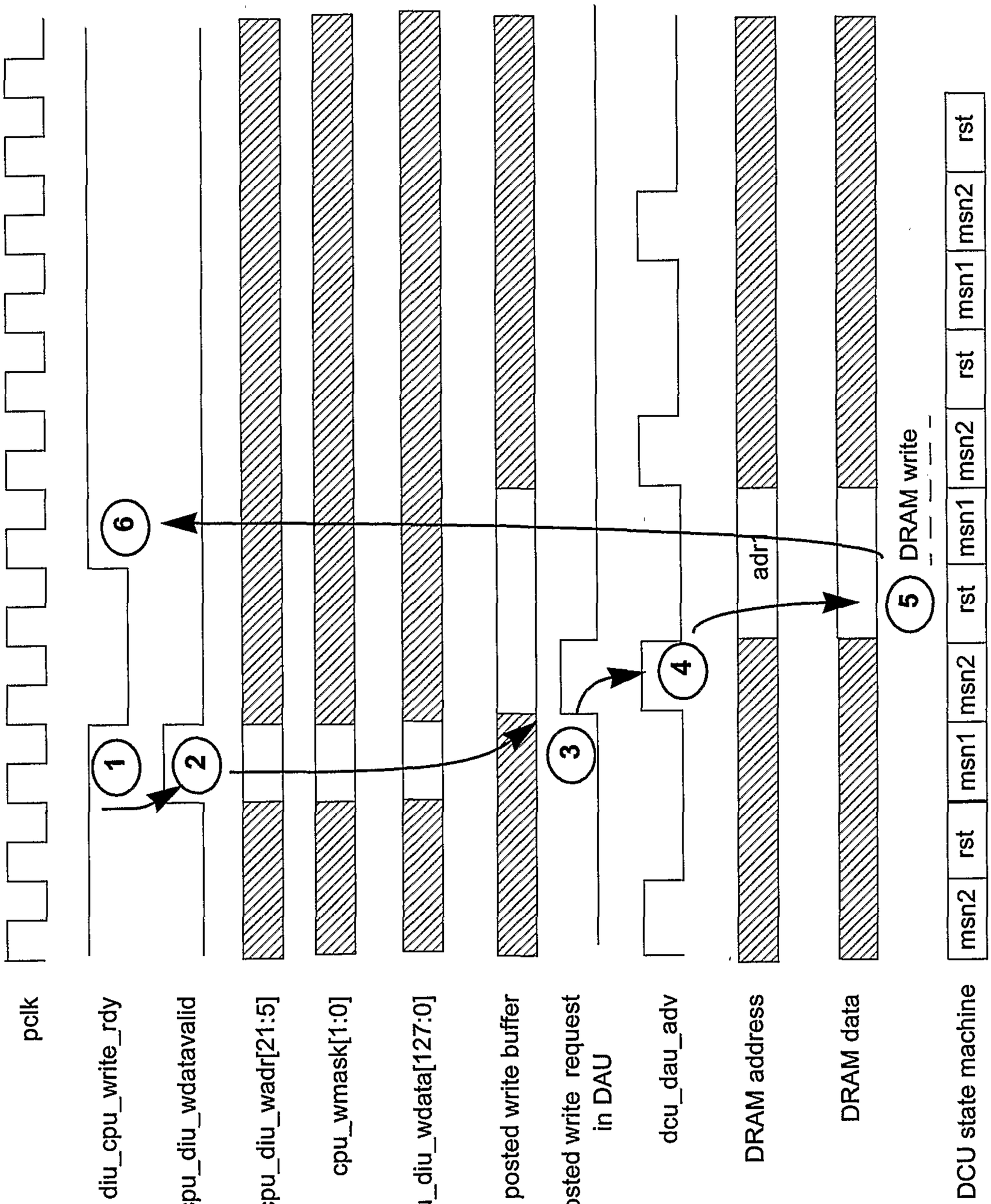
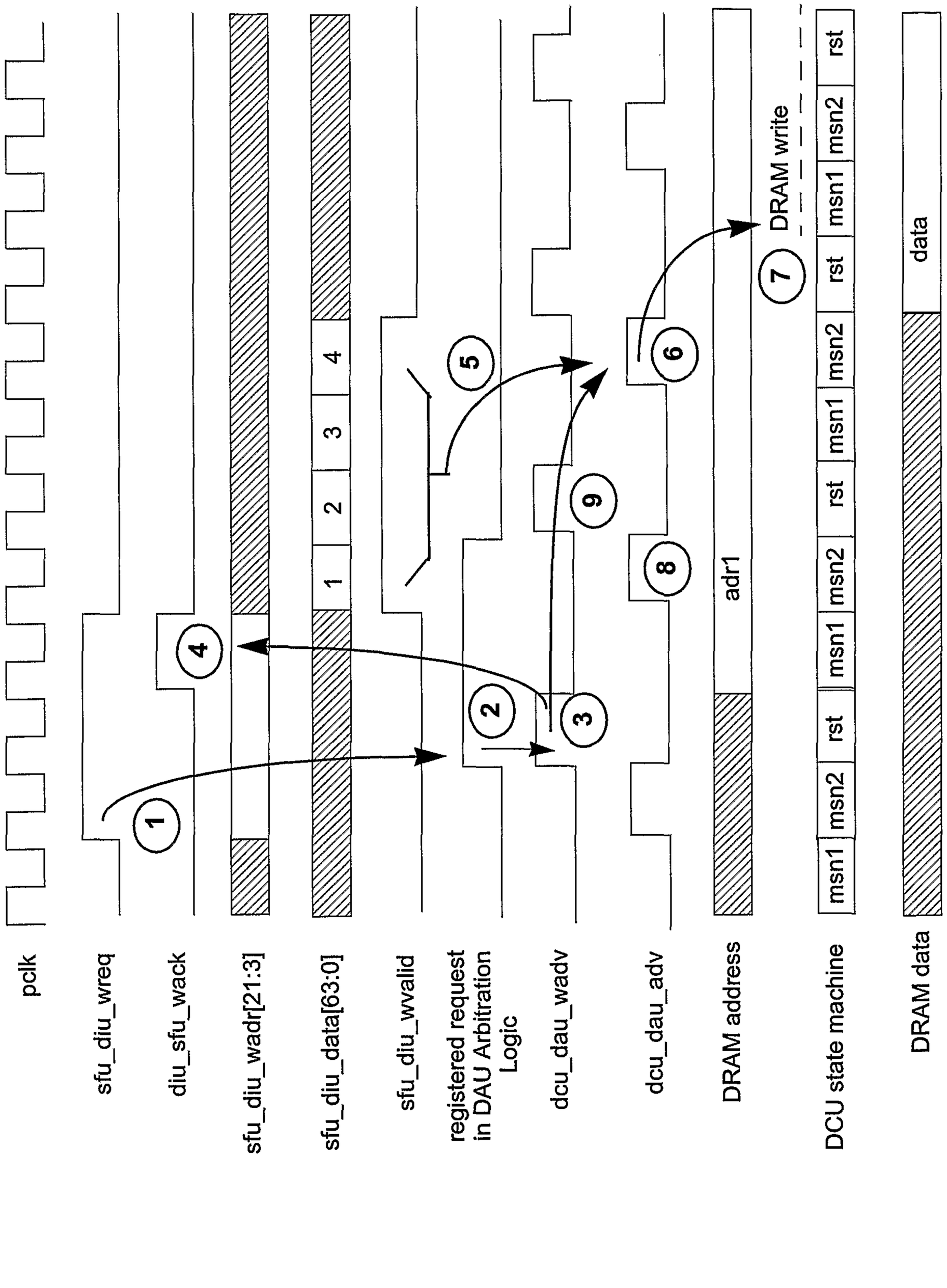

pclk
diu_cpu_write_rdy
cpu_diu_wdatavalid
cpu_diu_wadr[21:5]
cpu_wmask[1:0]
cpu_diu_wdata[127:0]
posted write buffer
posted write request in DAU
dcu_dau_adv
DRAM address
DRAM data
DCU state machine
msn2
rst
msn1
msn2
rst
msn1
msn2
rst
adr1
DRAM write
1
2
3
4
5
6

FIG. 113


pclk
sfu_diu_wreq
diu_sfu_wack
sfu_diu_wadr[21:3]
sfu_diu_data[63:0]
sfu_diu_wvalid
registered request in DAU Arbitration Logic
dcu_dau_wadv
dcu_dau_adv
DRAM address
DCU state machine
DRAM data
adr1
data
msn1
msn2
rst
DRAM write
1
2
3
4
5
6
7
8
9

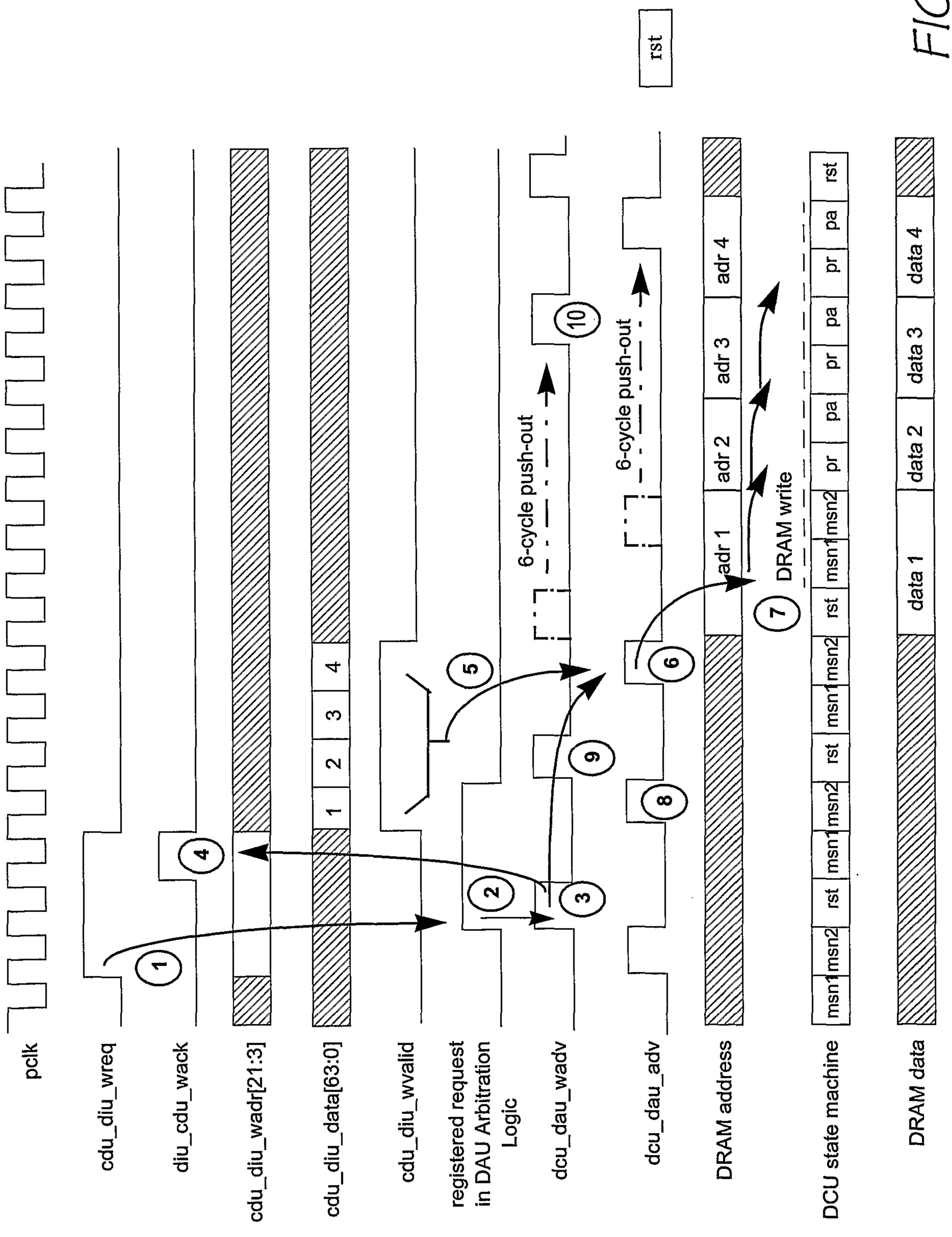

pclk
cdu_diu_wreq
diu_cdu_wack
cdu_diu_wadr[21:3]
cdu_diu_data[63:0]
cdu_diu_wvalid
registered request in DAU Arbitration Logic
dcu_dau_wadv
dcu_dau_adv
DRAM address
DCU state machine
DRAM data
rst
msn1
msn2
pr
pa
adr 1
adr 2
adr 3
adr 4
data 1
data 2
data 3
data 4
6-cycle push-out
DRAM write
1
2
3
4
5
6
7
8
9
10

FIG. 114

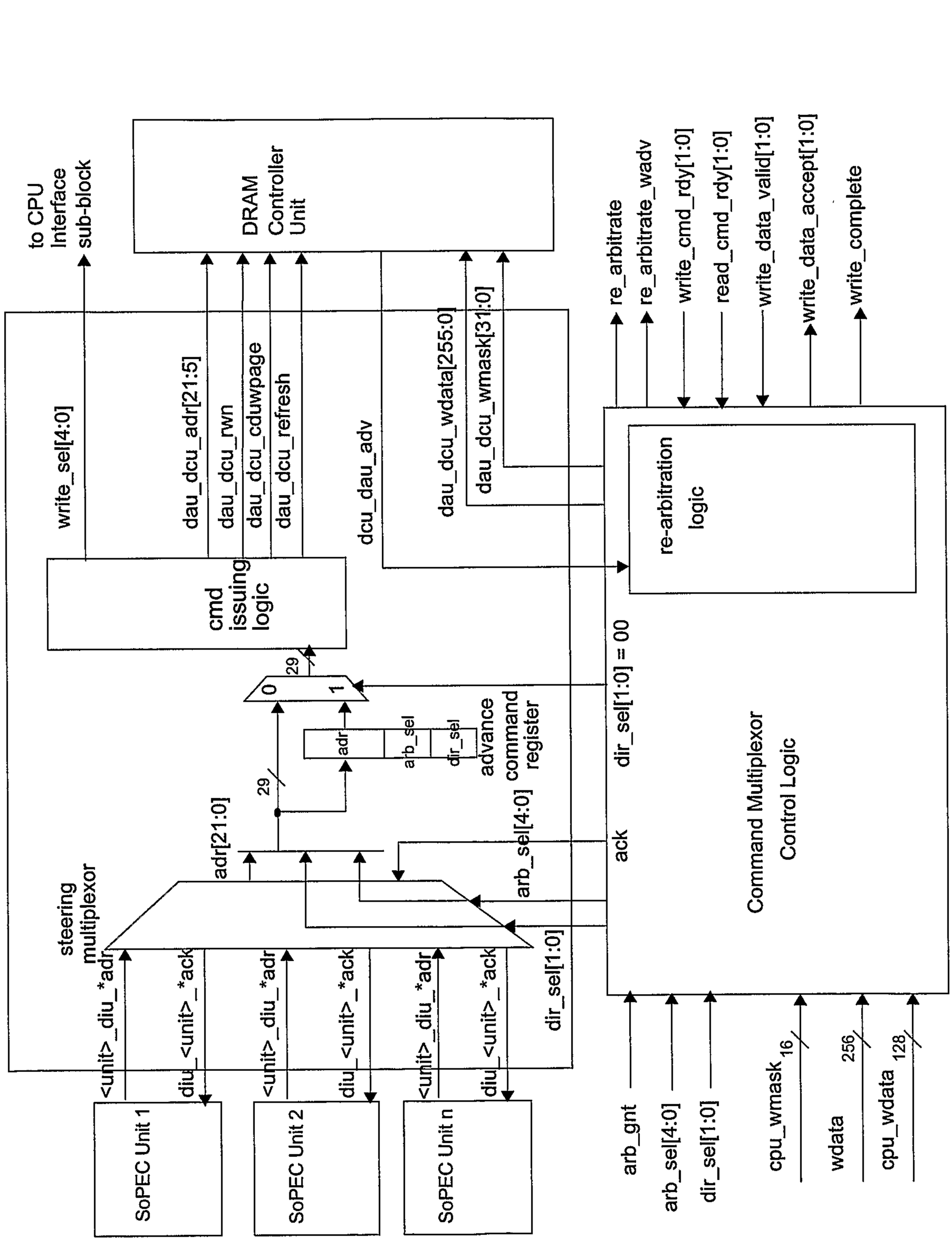

to CPU Interface sub-block
DRAM Controller Unit
write_sel[4:0]
dau_dcu_adr[21:5]
dau_dcu_rwn
dau_dcu_cduwpage
dau_dcu_refresh
dcu_dau_adv
dau_dcu_wdata[255:0]
dau_dcu_wmask[31:0]
re_arbitrate
re_arbitrate_wadv
write_cmd_rdy[1:0]
read_cmd_rdy[1:0]
write_data_valid[1:0]
write_data_accept[1:0]
write_complete
re-arbitration logic
cmd issuing logic
29
0
1
adr
arb_sel
dir_sel
advance command register
29
adr[21:0]
arb_sel[4:0]
ack
dir_sel[1:0] = 00
Command Multiplexor Control Logic
steering multiplexor
<unit>_diu_*adr
diu_<unit>_*ack
dir_sel[1:0]
SoPEC Unit 1
SoPEC Unit 2
SoPEC Unit n
arb_gnt
arb_sel[4:0]
dir_sel[1:0]
cpu_wmask 16
wdata 256
cpu_wdata 128

FIG. 115

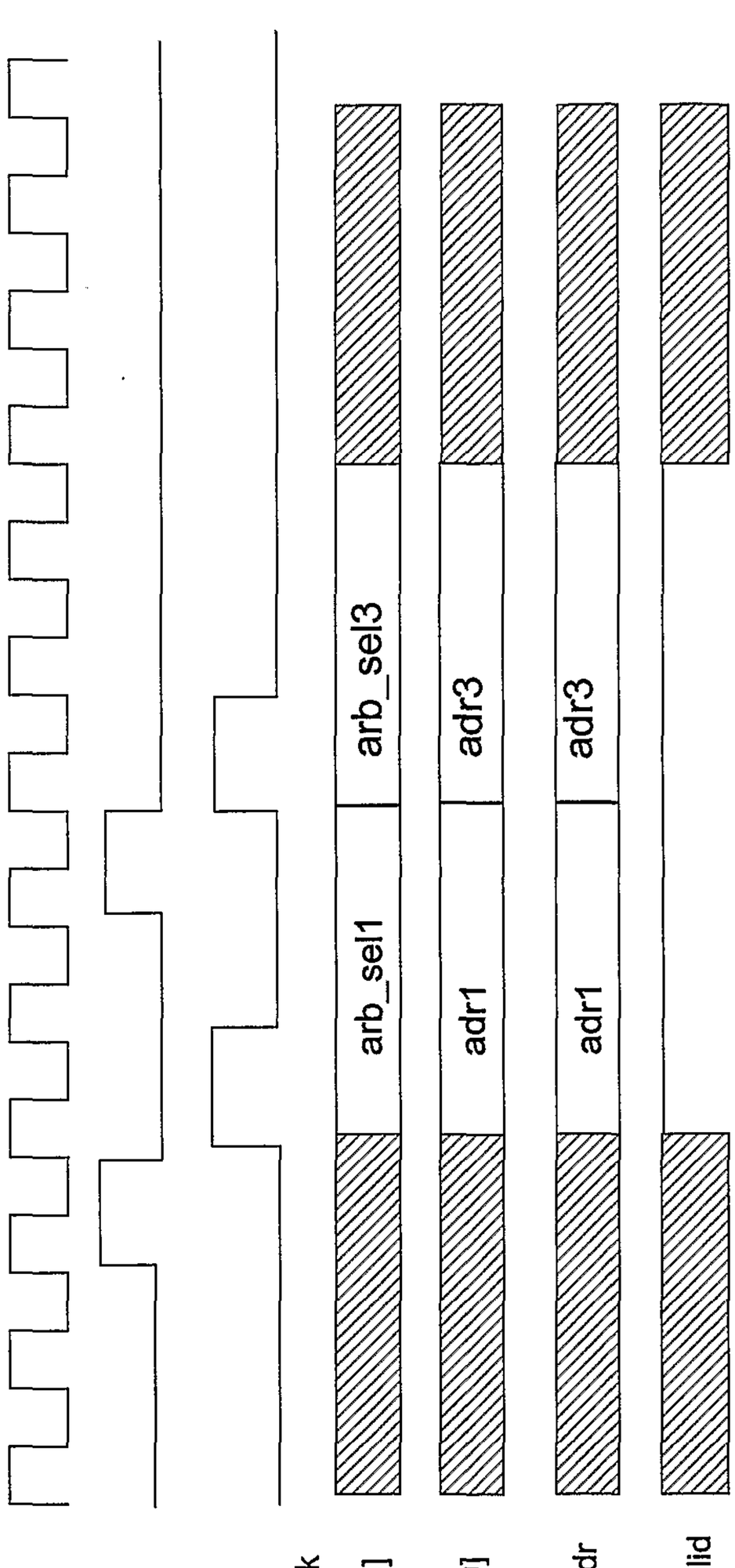

pclk
re_arbitrate
arb_gnt / diu_<unit>_*ack
arb_sel[3:0]
adr[21:0]
dau_dcu_adr
dau_dcu_valid
arb_sel3
adr3
adr3
arb_sel1
adr1
adr1

FIG. 116

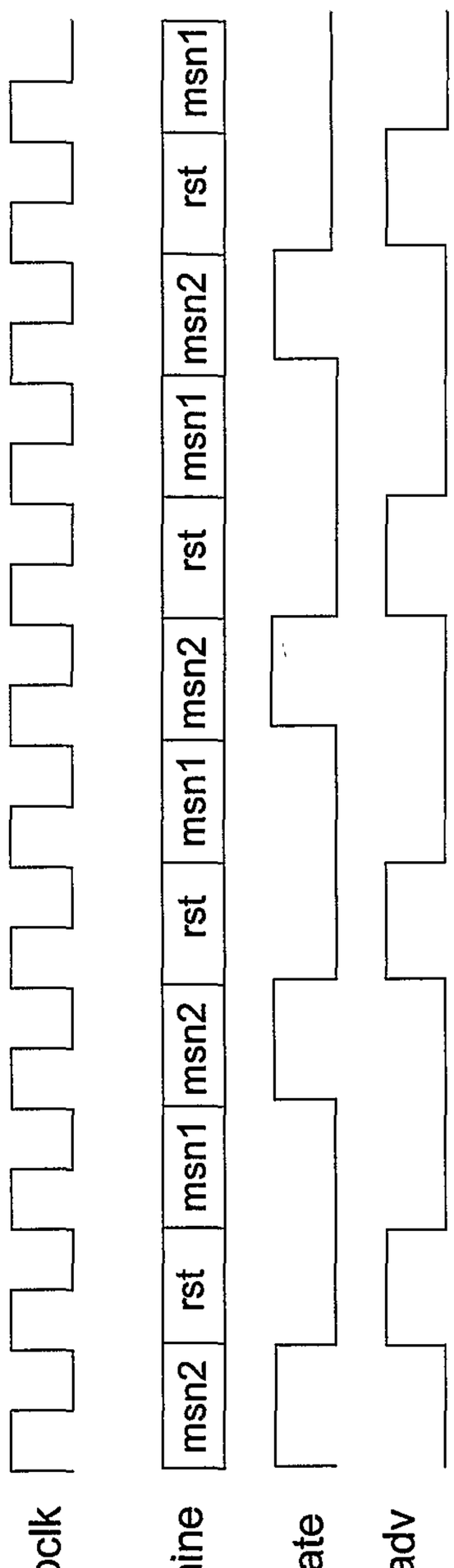

pclk
DCU state machine
re_arbitrate
re_arbitrate_wadv
msn2
rst
msn1
msn2
rst
msn1
msn2
rst
msn1
msn2
rst
msn1

FIG. 117

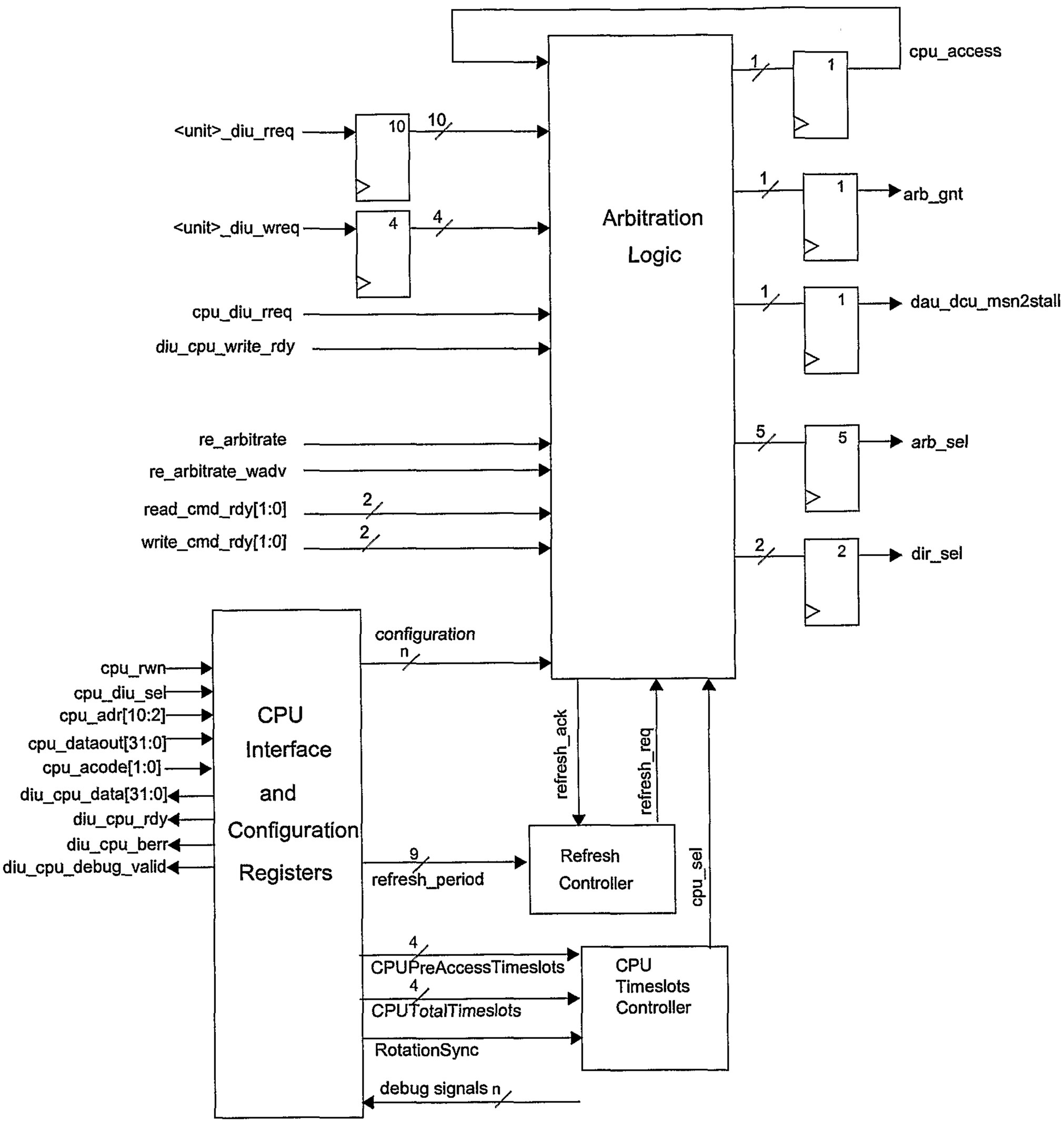

cpu_access
1
1
<unit>_diu_rreq
10
10
<unit>_diu_wreq
4
4
Arbitration
Logic
1
1
arb_gnt
1
1
dau_dcu_msn2stall
cpu_diu_rreq
diu_cpu_write_rdy
re_arbitrate
re_arbitrate_wadv
read_cmd_rdy[1:0]
2
write_cmd_rdy[1:0]
2
5
5
arb_sel
2
2
dir_sel
configuration
n
cpu_rwn
cpu_diu_sel
cpu_adr[10:2]
cpu_dataout[31:0]
cpu_acode[1:0]
diu_cpu_data[31:0]
diu_cpu_rdy
diu_cpu_berr
diu_cpu_debug_valid
CPU
Interface
and
Configuration
Registers
refresh_ack
refresh_req
cpu_sel
9
refresh_period
Refresh
Controller
4
CPUPreAccessTimeslots
4
CPUTotalTimeslots
RotationSync
CPU
Timeslots
Controller
debug signals n

FIG. 118

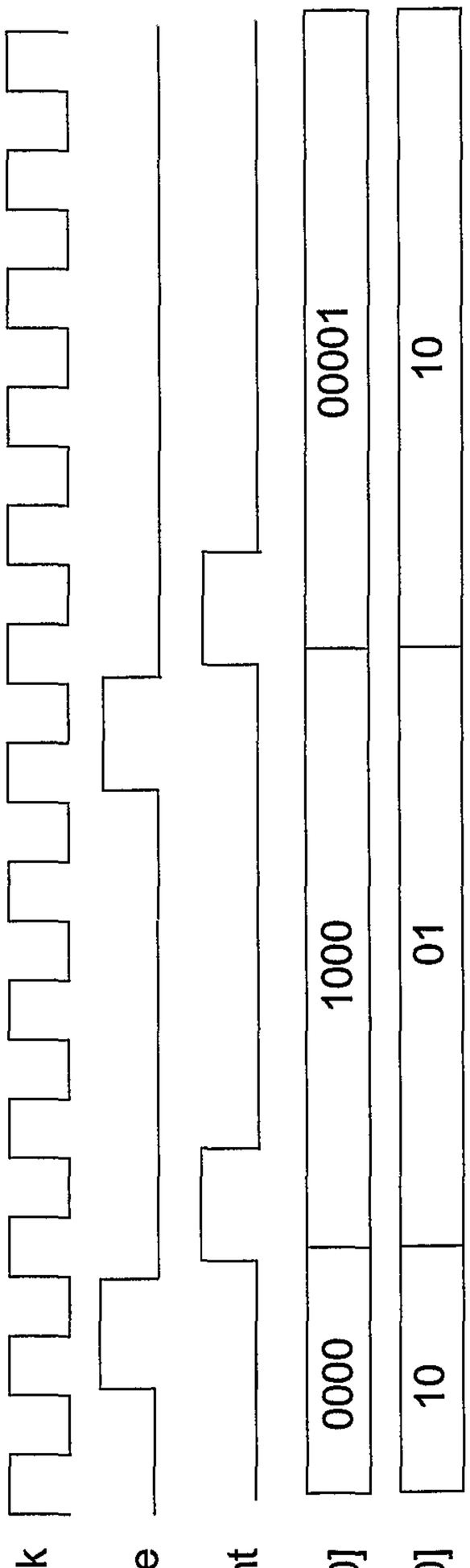

pclk
re_arbitrate
arb_gnt
arb_sel[3:0]
dir_sel[1:0]
0000
1000
00001
10
01
10

FIG. 119

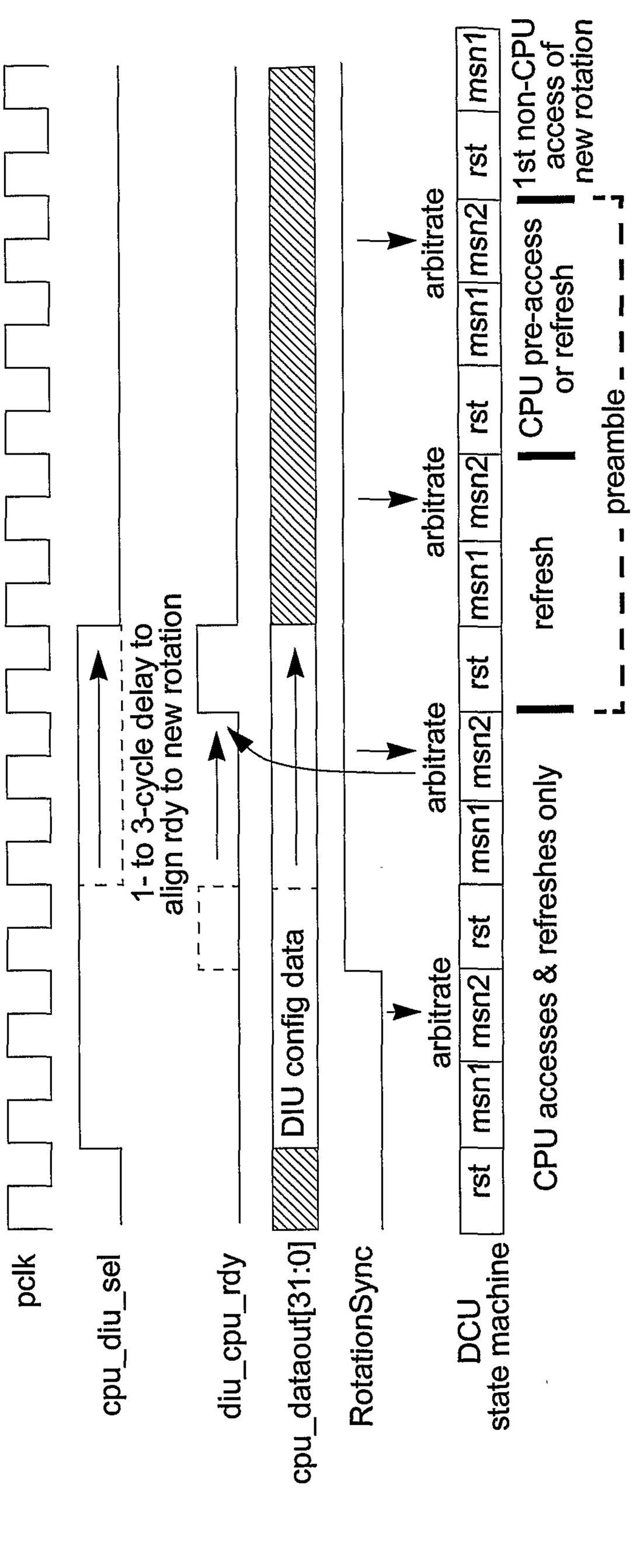

pclk
cpu_diu_sel
diu_cpu_rdy
cpu_dataout[31:0]
RotationSync
DCU state machine
CPU accesses & refreshes only
preamble
rst
msn1
msn2
arbitrate
DIU config data
1- to 3-cycle delay to align rdy to new rotation
refresh
CPU pre-access or refresh
1st non-CPU access of new rotation

FIG. 120

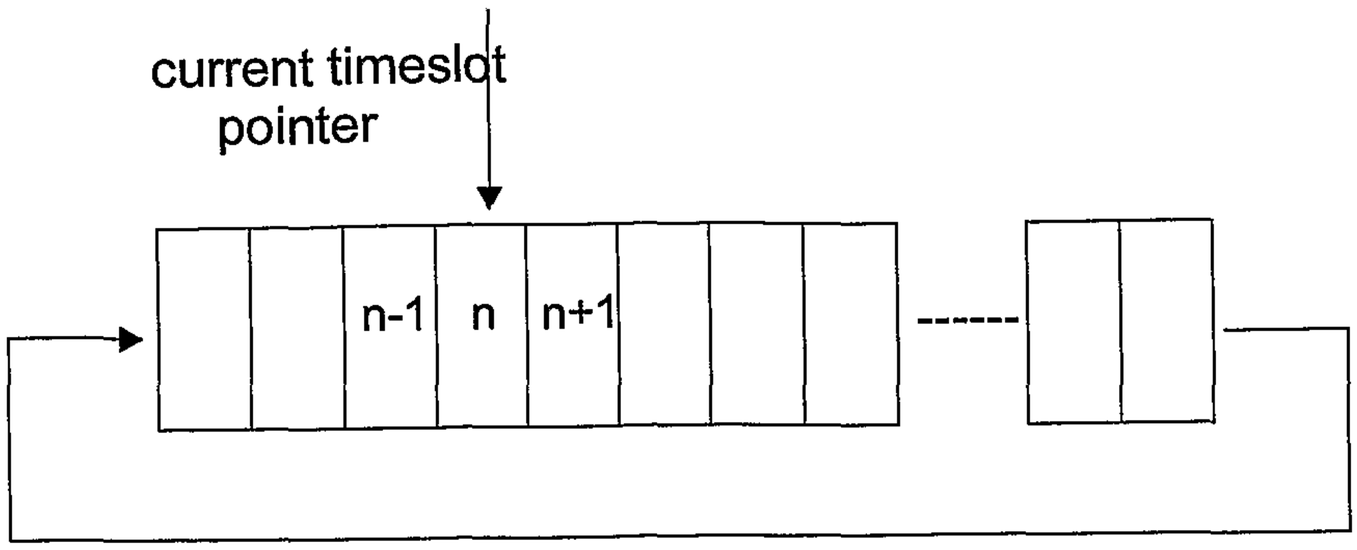


FIG. 121

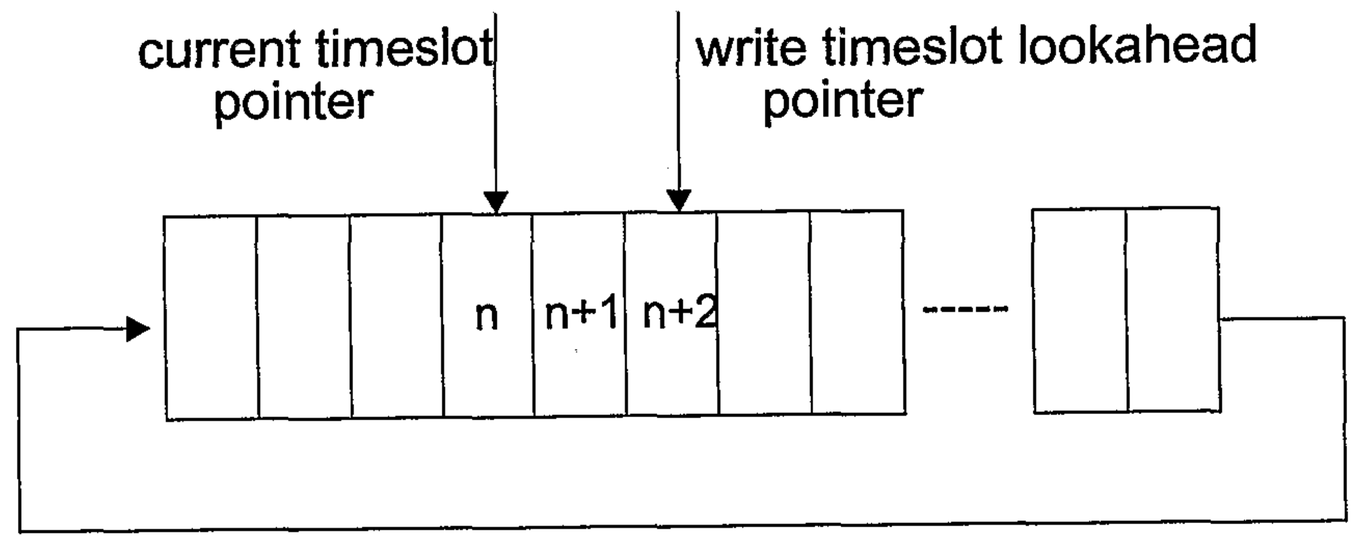


FIG. 122

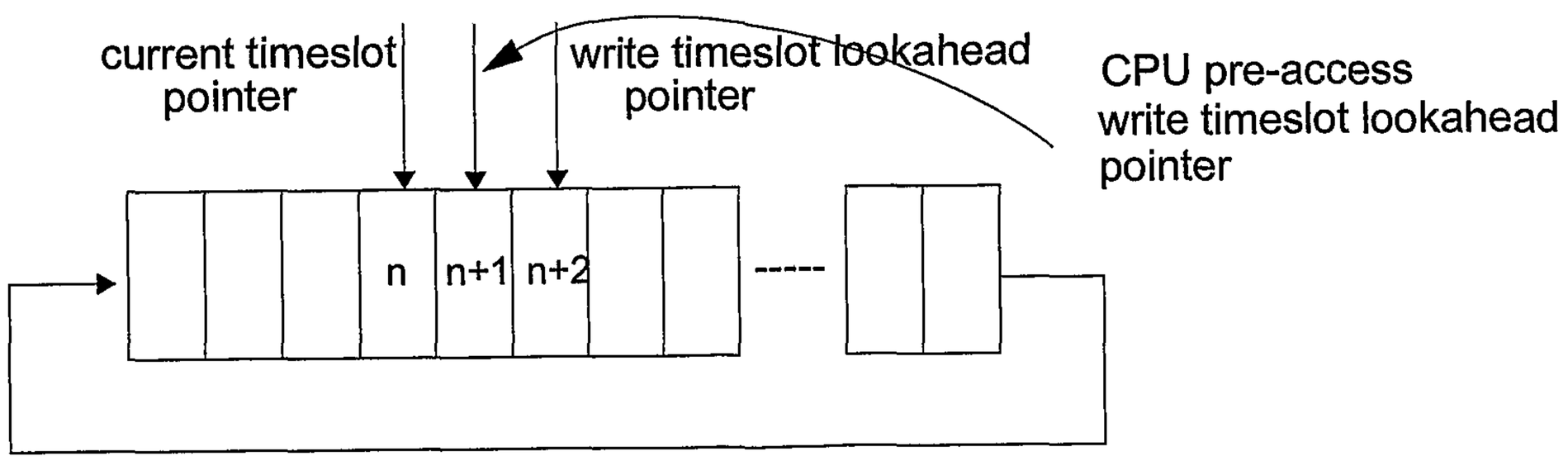


FIG. 123

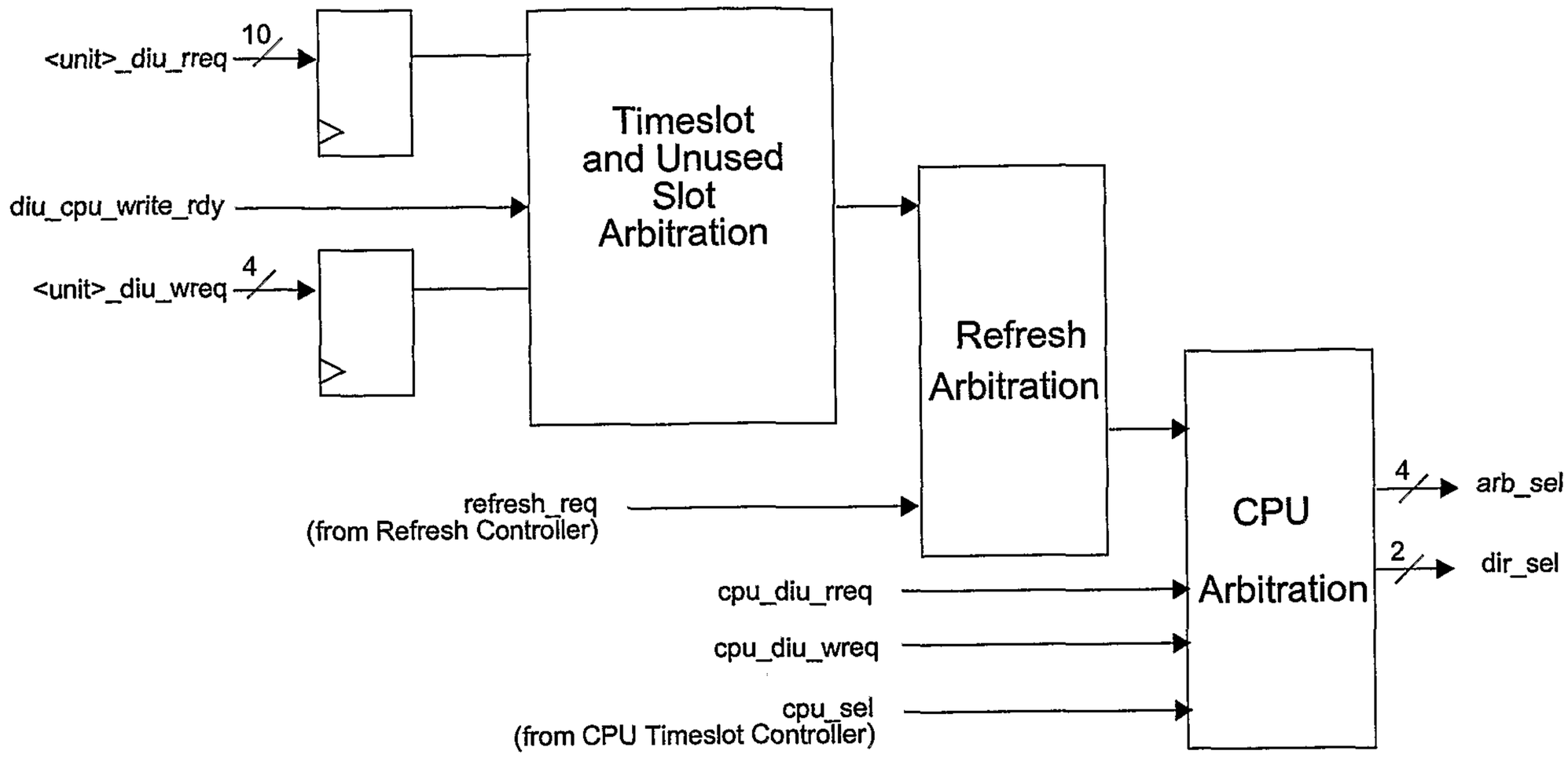


FIG. 124

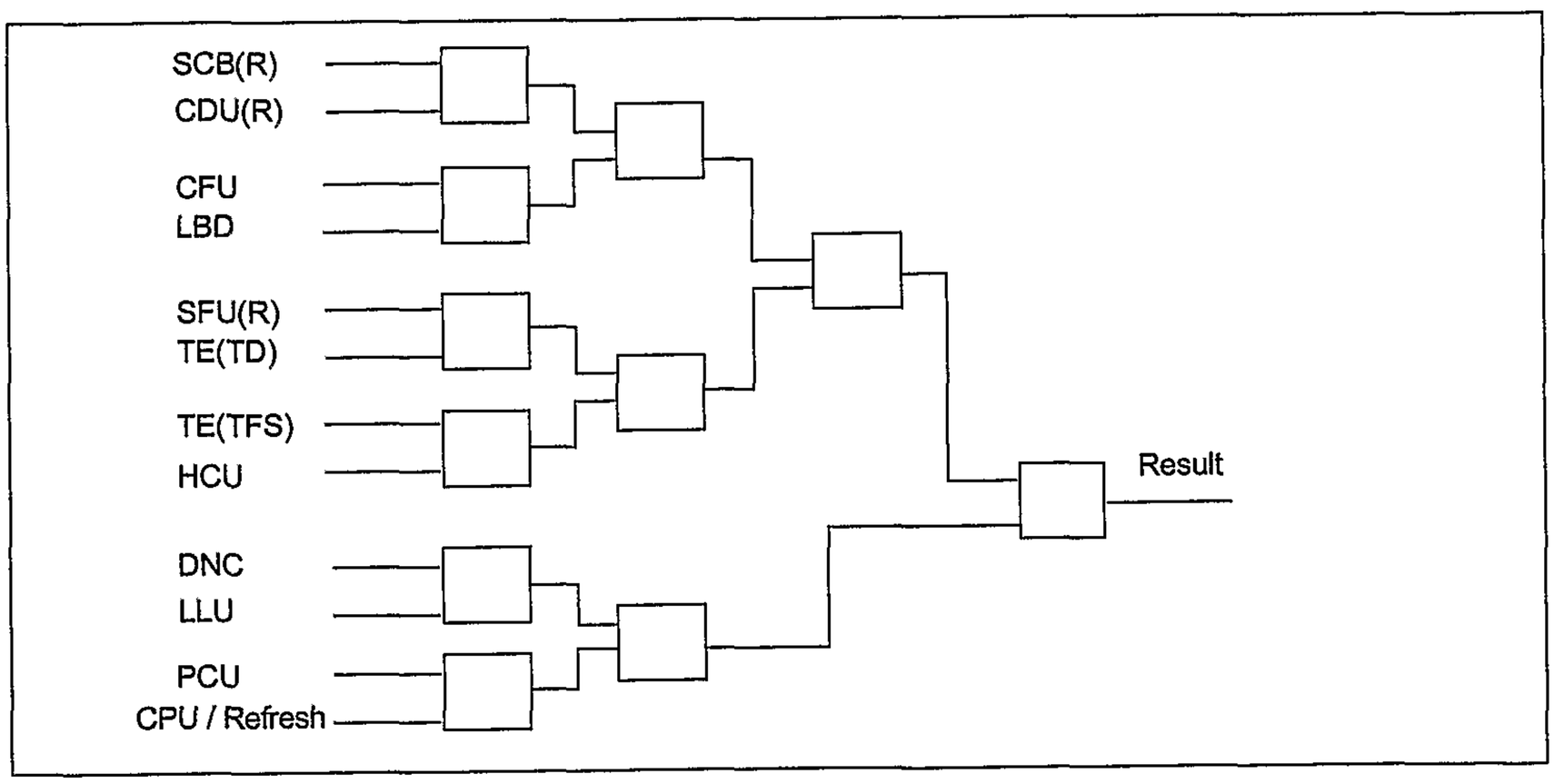


FIG. 125

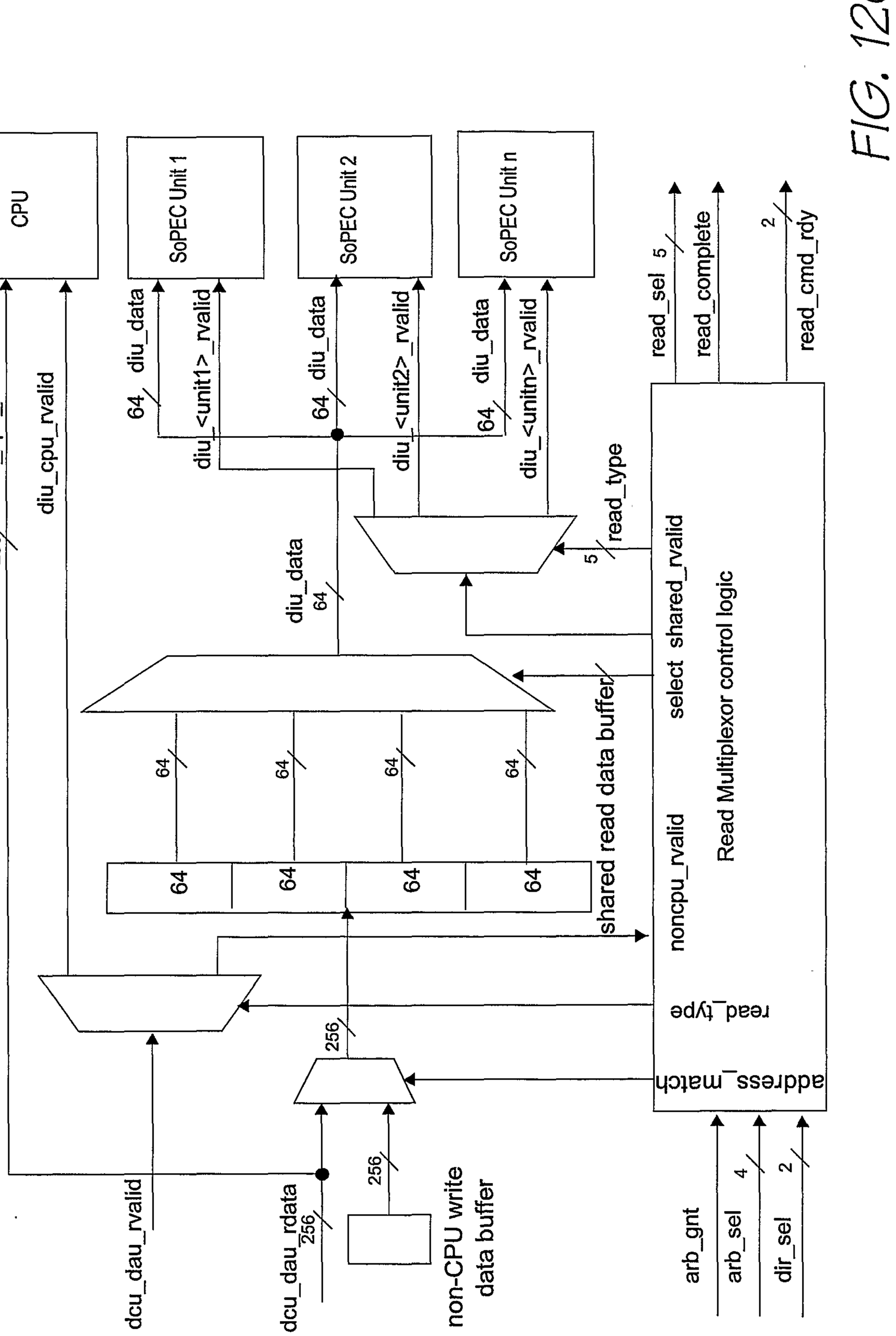

CPU
SoPEC Unit 1
SoPEC Unit 2
SoPEC Unit n
256 dram_cpu_data
diu_cpu_rvalid
64 diu_data
diu_<unit1>_rvalid
diu_<unit2>_rvalid
diu_<unitn>_rvalid
read_sel 5
read_complete
2 read_cmd_rdy
5 read_type
select shared_rvalid
Read Multiplexor control logic
shared read data buffer
64
noncpu_rvalid
read_type
address_match
arb_gnt
arb_sel 4
dir_sel 2
dcu_dau_rvalid
dcu_dau_rdata 256
256
non-CPU write data buffer

FIG. 126

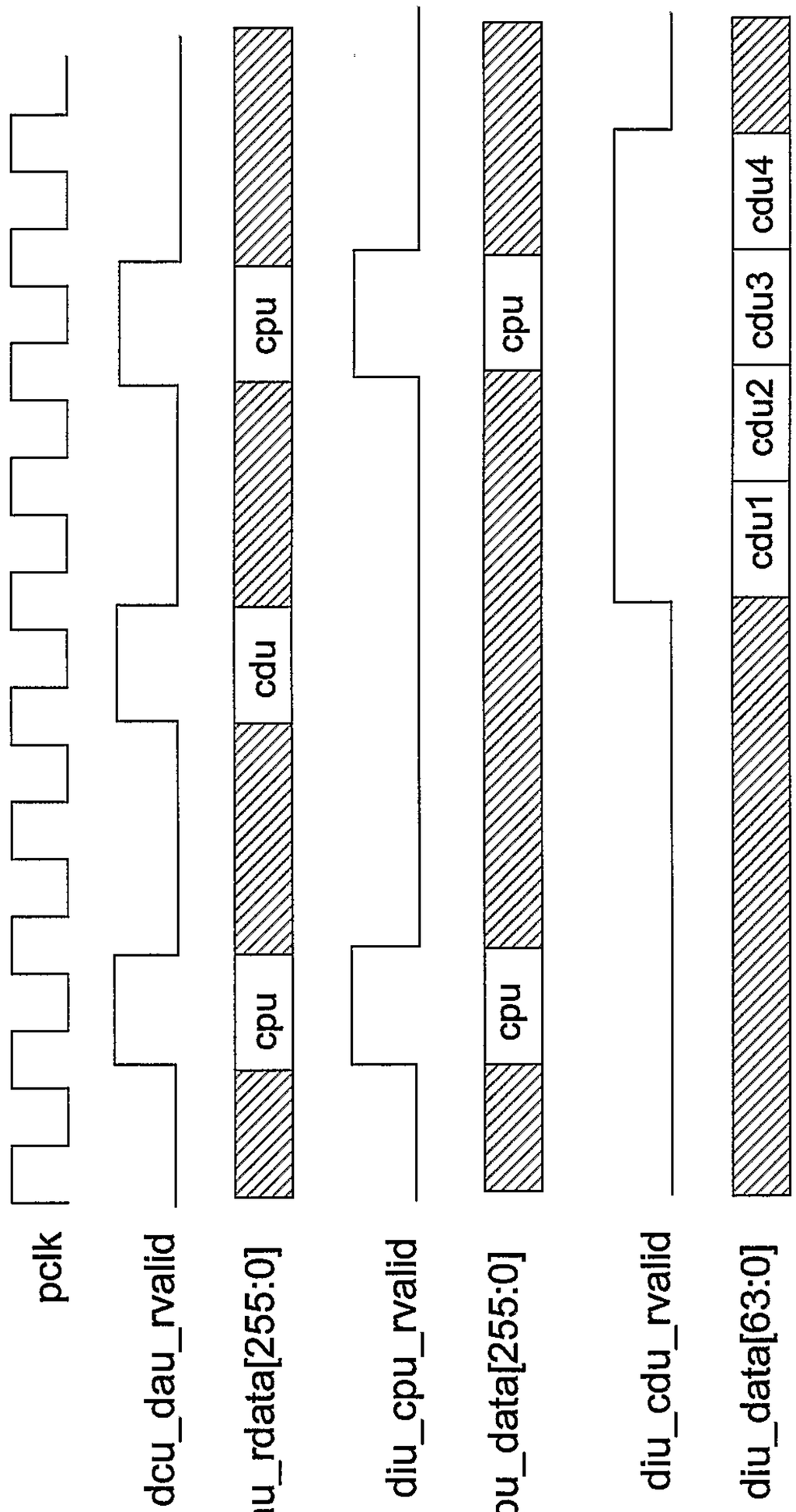

pclk
dcu_dau_rvalid
dcu_dau_rdata[255:0]
diu_cpu_rvalid
dram_cpu_data[255:0]
diu_cdu_rvalid
diu_data[63:0]
cpu
cdu
cdu1
cdu2
cdu3
cdu4

FIG. 127

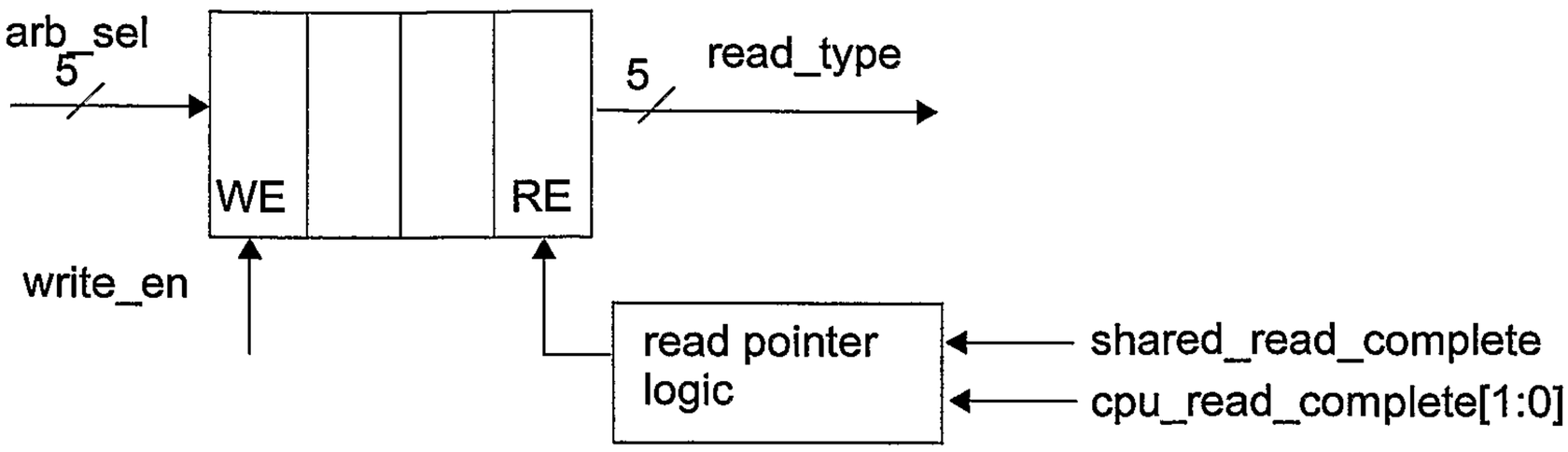


FIG. 128

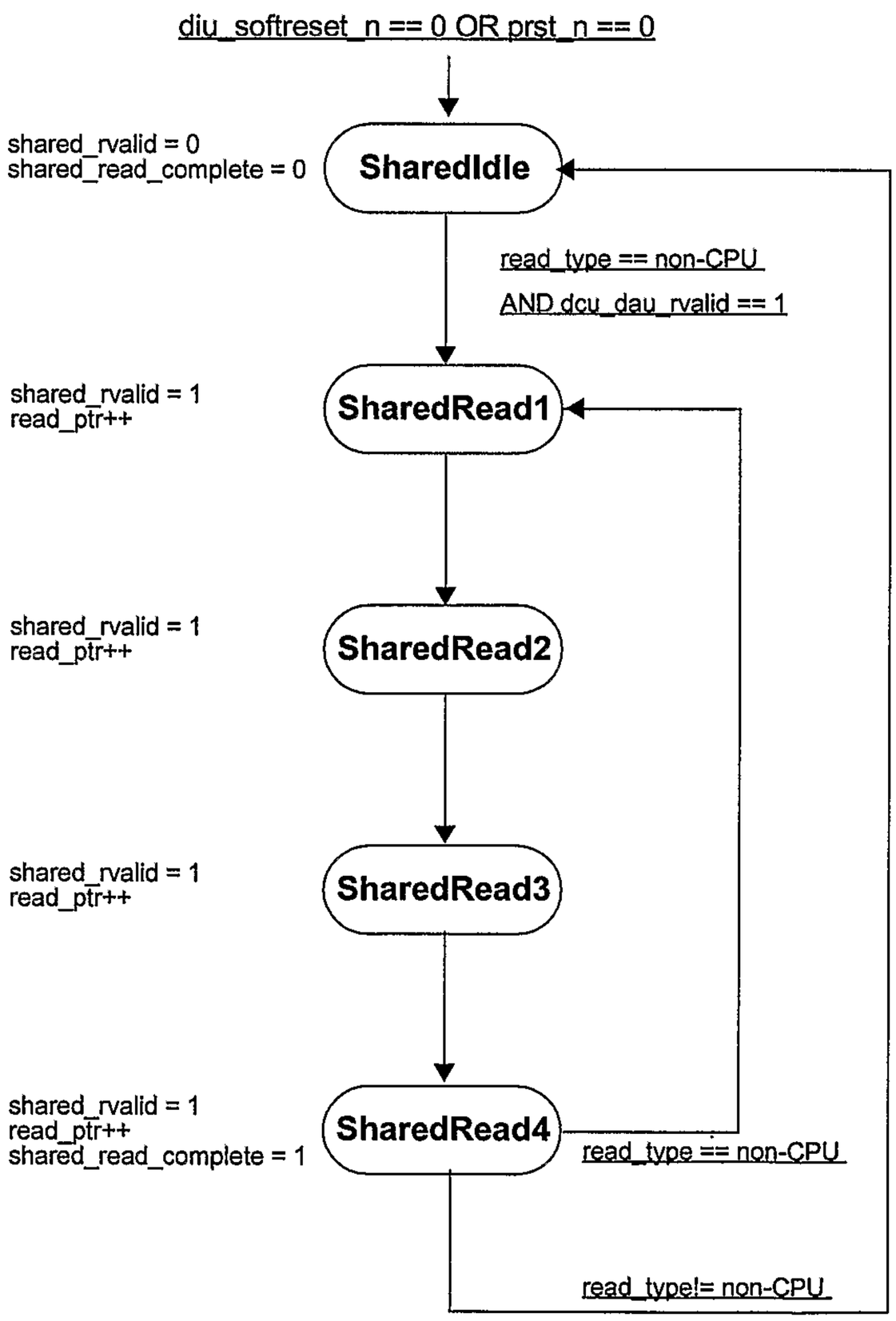


FIG. 129

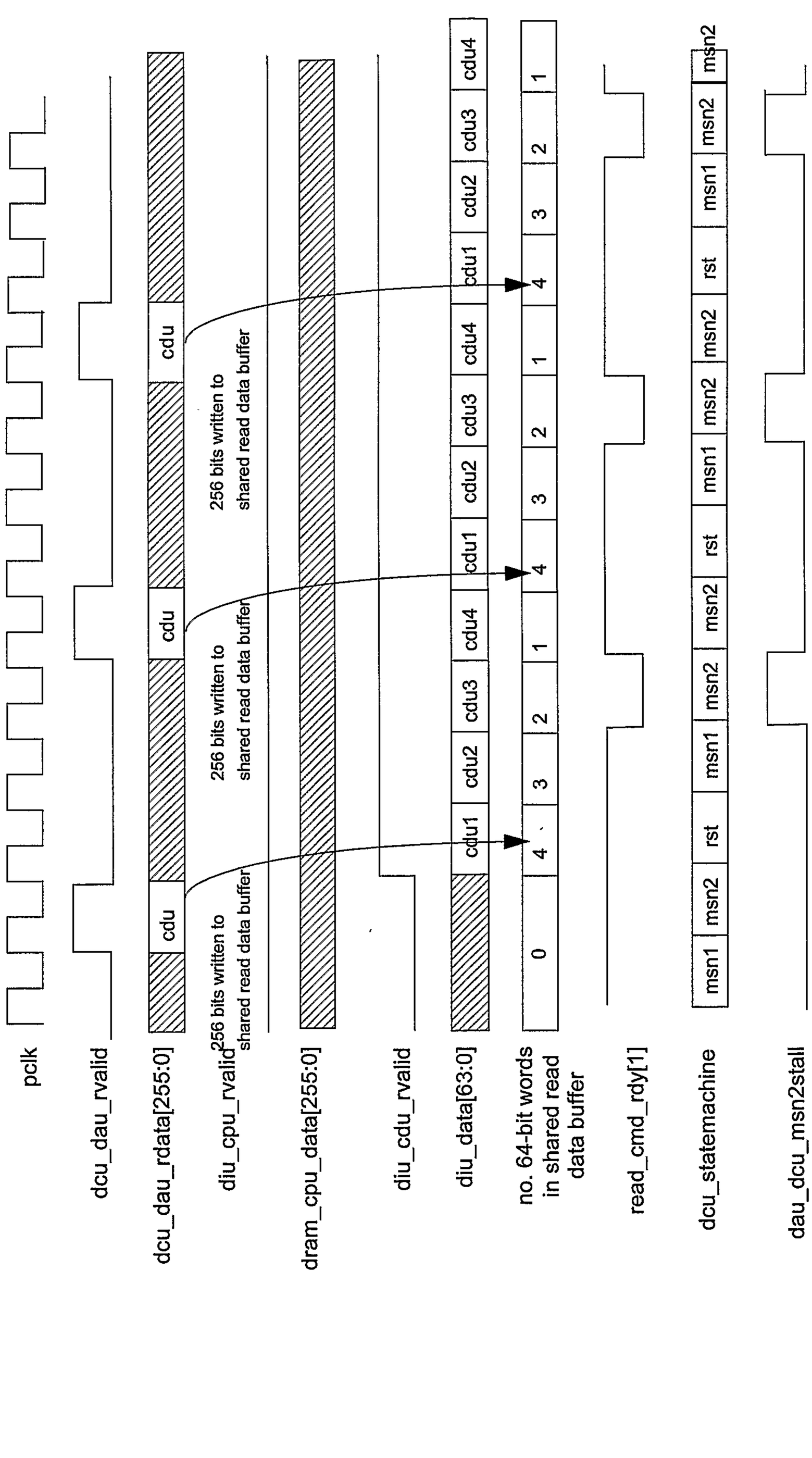

pclk
dcu_dau_rvalid
dcu_dau_rdata[255:0]
diu_cpu_rvalid
256 bits written to shared read data buffer
dram_cpu_data[255:0]
diu_cdu_rvalid
diu_data[63:0]
no. 64-bit words in shared read data buffer
read_cmd_rdy[1]
dcu_statemachine
dau_dcu_msn2stall
cdu
cdu1
cdu2
cdu3
cdu4
msn1
msn2
rst
0
1
2
3
4

FIG. 130

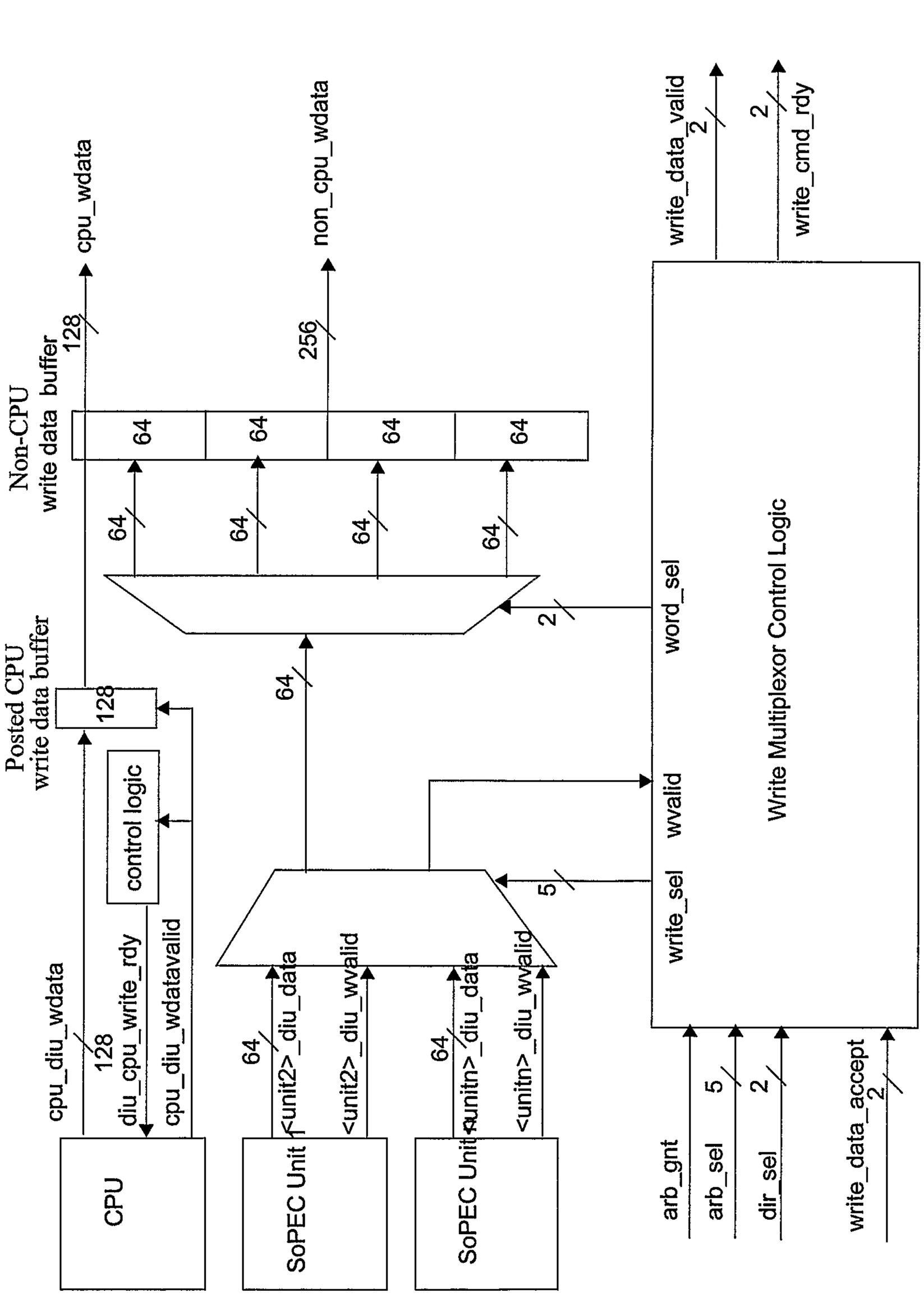

Non-CPU write data buffer
128
cpu_wdata
256
non_cpu_wdata
64
64
64
64
64
64
64
64
write_data_valid
2
2
write_cmd_rdy
Write Multiplexor Control Logic
word_sel
2
write_sel
wvalid
5
Posted CPU write data buffer
128
control logic
cpu_diu_wdata
128
diu_cpu_write_rdy
cpu_diu_wdatavalid
64
<unit2>_diu_data
<unit2>_diu_wvalid
64
<unitn>_diu_data
<unitn>_diu_wvalid
CPU
SoPEC Unit 1
SoPEC Unit n
arb_gnt
arb_sel
5
dir_sel
2
write_data_accept
2

FIG. 131

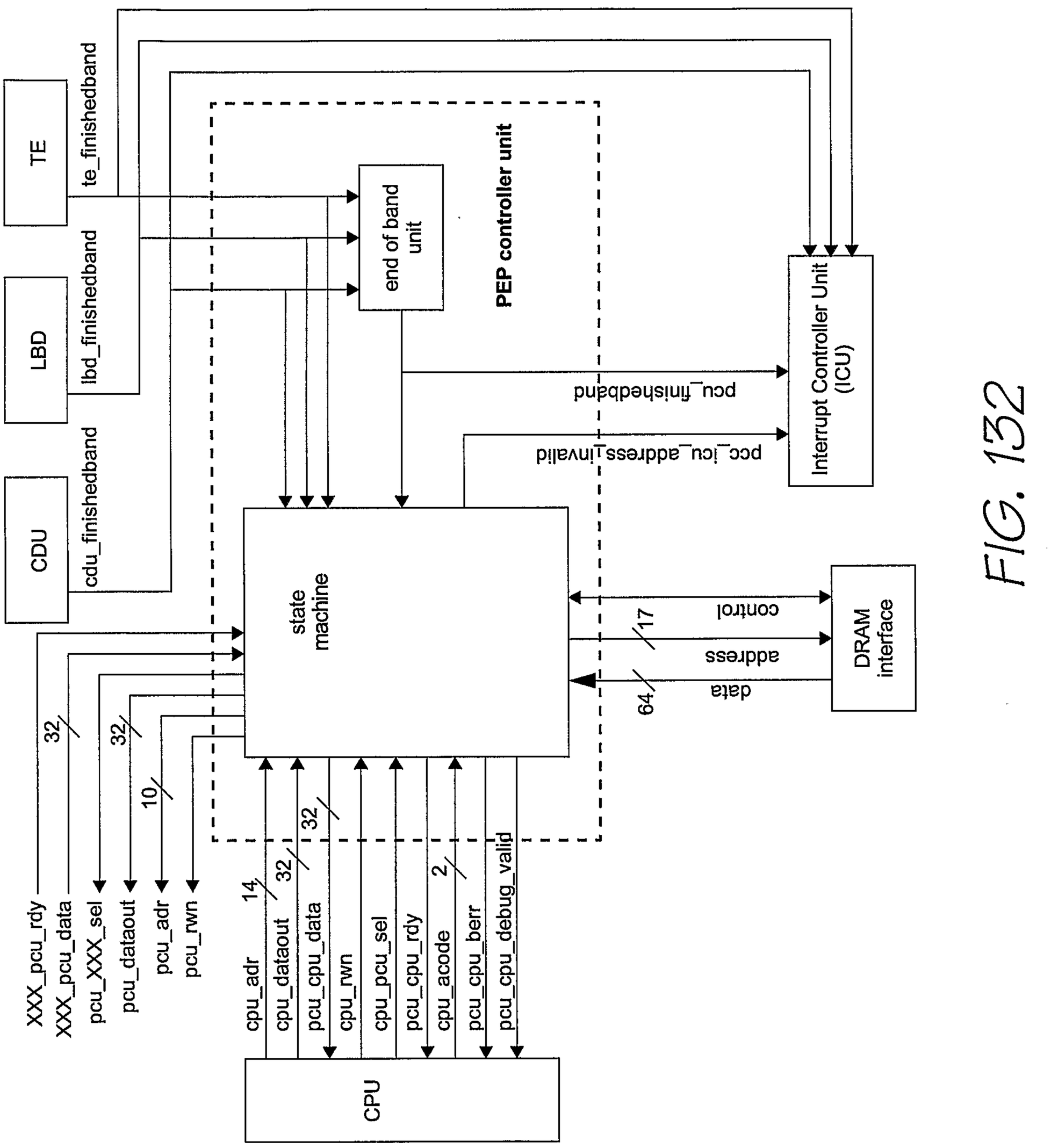

TE
LBD
CDU
te_finishedband
lbd_finishedband
cdu_finishedband
PEP controller unit
end of band unit
state machine
CPU
Interrupt Controller Unit (ICU)
DRAM interface
pcu_finishedband
pcc_icu_address_invalid
control
address
data
17
64
32
32
10
14
32
32
2
XXX_pcu_rdy
XXX_pcu_data
pcu_XXX_sel
pcu_dataout
pcu_adr
pcu_rwn
cpu_adr
cpu_dataout
pcu_cpu_data
cpu_rwn
cpu_pcu_sel
pcu_cpu_rdy
cpu_acode
pcu_cpu_berr
pcu_cpu_debug_valid

FIG. 132

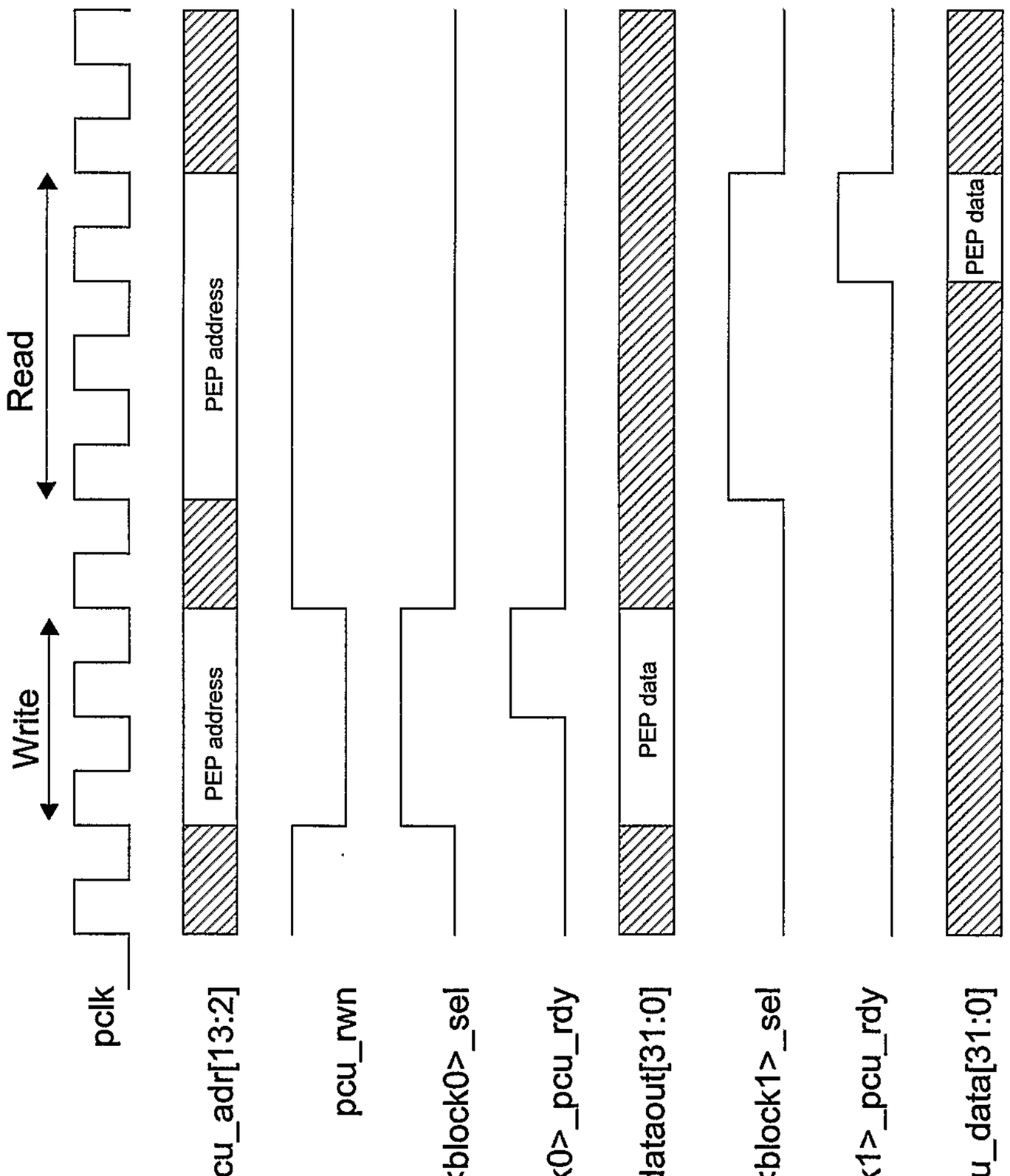

pclk
Write
Read
pcu_adr[13:2]
PEP address
PEP address
pcu_rwn
pcu_<block0>_sel
<block0>_pcu_rdy
pcu_dataout[31:0]
PEP data
pcu_<block1>_sel
<block1>_pcu_rdy
<block1>_pcu_data[31:0]
PEP data

FIG. 133

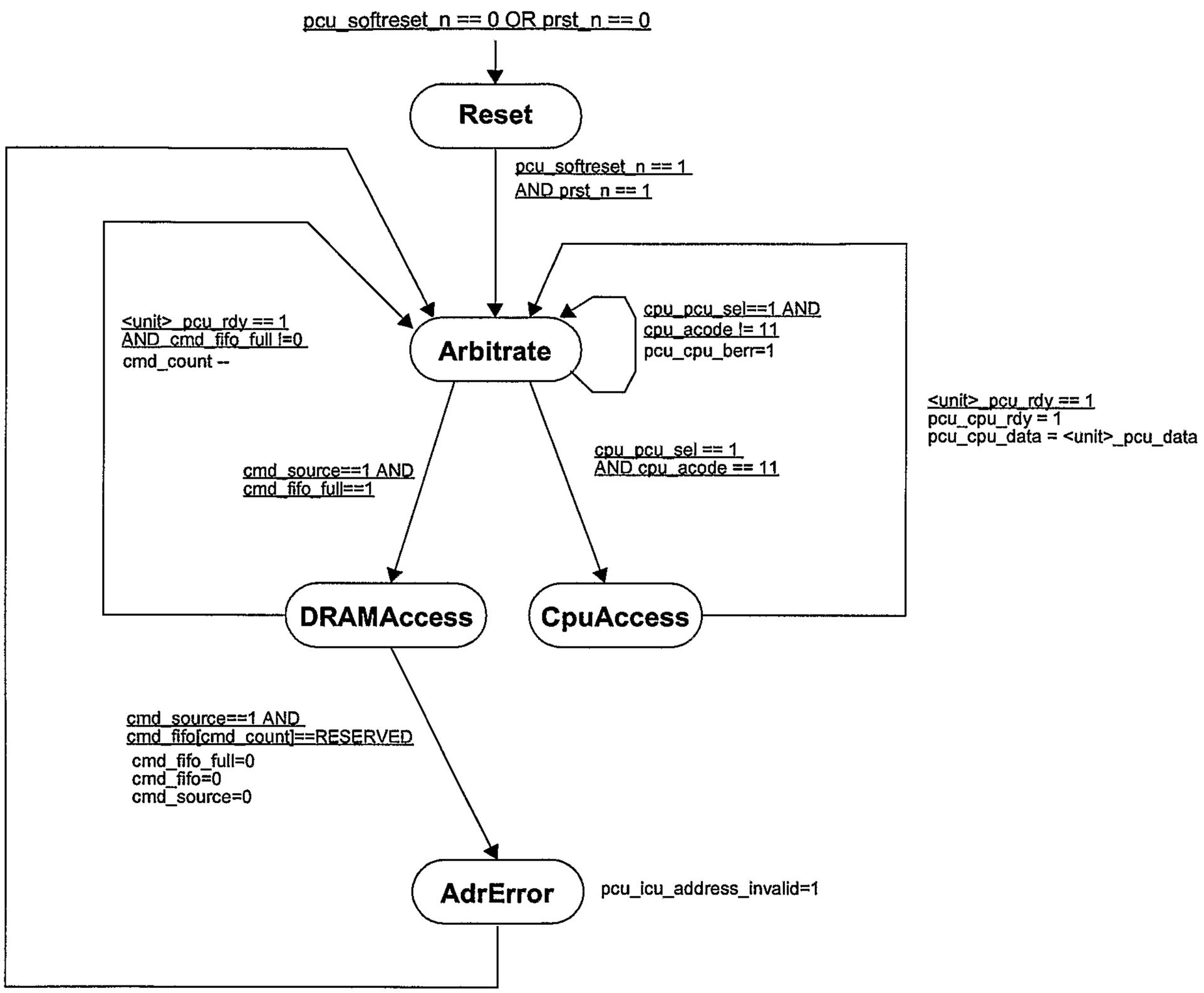

State Machine A
pcu_softreset_n == 0 OR prst_n == 0
Reset
pcu_softreset_n == 1
AND prst_n == 1
Arbitrate
<unit>_pcu_rdy == 1
AND cmd_fifo_full !=0
cmd_count --
cpu_pcu_sel==1 AND
cpu_acode != 11
pcu_cpu_berr=1
<unit>_pcu_rdy == 1
pcu_cpu_rdy = 1
pcu_cpu_data = <unit>_pcu_data
cmd_source==1 AND
cmd_fifo_full==1
cpu_pcu_sel == 1
AND cpu_acode == 11
DRAMAccess
CpuAccess
cmd_source==1 AND
cmd_fifo[cmd_count]==RESERVED
cmd_fifo_full=0
cmd_fifo=0
cmd_source=0
AdrError
pcu_icu_address_invalid=1

FIG. 134

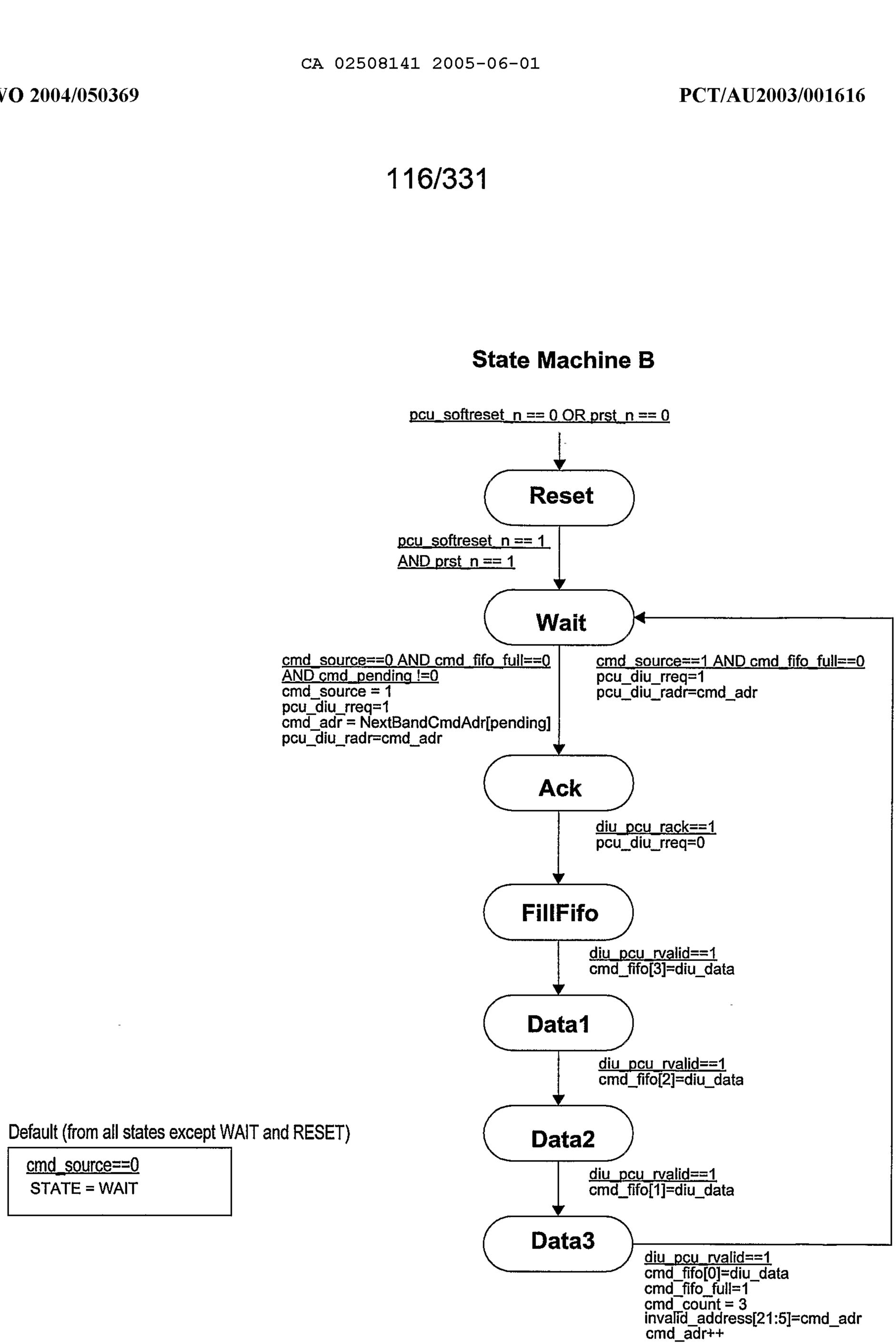

State Machine B
pcu_softreset_n == 0 OR prst_n == 0
Reset
pcu_softreset_n == 1
AND prst_n == 1
Wait
cmd_source==0 AND cmd_fifo_full==0
AND cmd_pending !=0
cmd_source = 1
pcu_diu_rreq=1
cmd_adr = NextBandCmdAdr[pending]
pcu_diu_radr=cmd_adr
cmd_source==1 AND cmd_fifo_full==0
pcu_diu_rreq=1
pcu_diu_radr=cmd_adr
Ack
diu_pcu_rack==1
pcu_diu_rreq=0
FillFifo
diu_pcu_rvalid==1
cmd_fifo[3]=diu_data
Data1
diu_pcu_rvalid==1
cmd_fifo[2]=diu_data
Data2
diu_pcu_rvalid==1
cmd_fifo[1]=diu_data
Data3
diu_pcu_rvalid==1
cmd_fifo[0]=diu_data
cmd_fifo_full=1
cmd_count = 3
invalid_address[21:5]=cmd_adr
cmd_adr++
Default (from all states except WAIT and RESET)
cmd_source==0
STATE = WAIT

FIG. 135

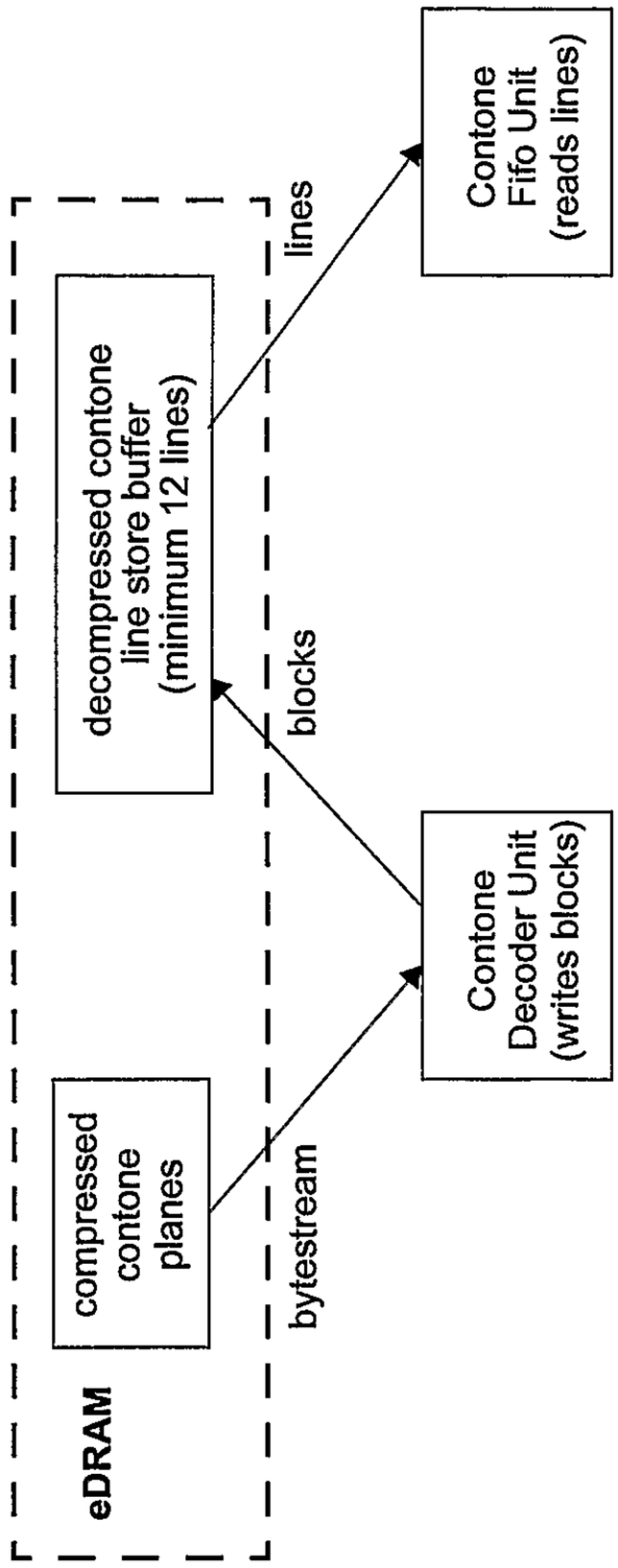

eDRAM
compressed contone planes
bytestream
decompressed contone line store buffer (minimum 12 lines)
blocks
lines
Contone Decoder Unit (writes blocks)
Contone Fifo Unit (reads lines)

FIG. 136

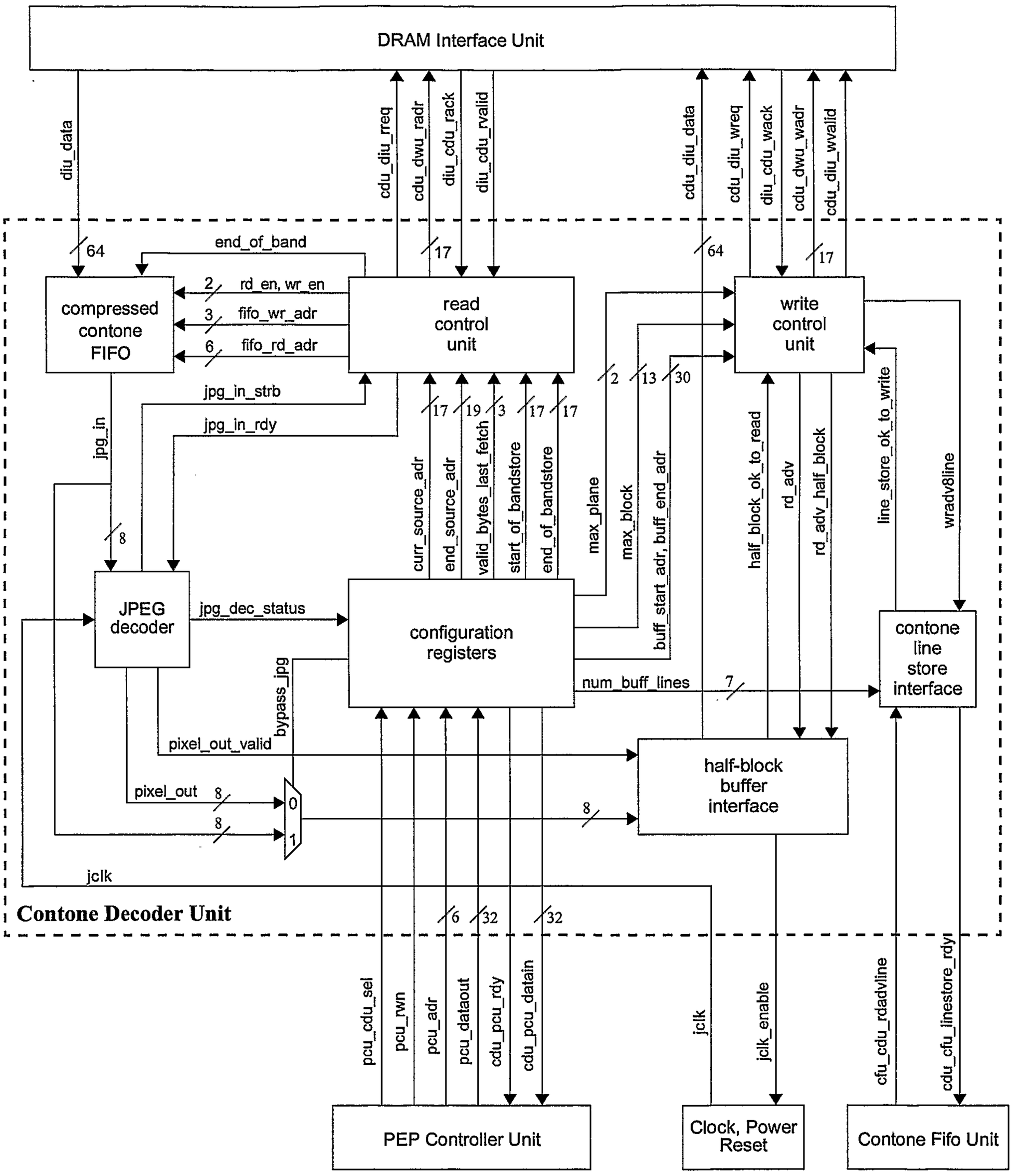

DRAM Interface Unit
diu_data
cdu_diu_rreq
cdu_dwu_radr
diu_cdu_rack
diu_cdu_rvalid
cdu_diu_data
cdu_diu_wreq
diu_cdu_wack
cdu_dwu_wadr
cdu_diu_wvalid
64
17
64
17
end_of_band
compressed contone FIFO
2 rd_en, wr_en
3 fifo_wr_adr
6 fifo_rd_adr
read control unit
write control unit
jpg_in_strb
jpg_in_rdy
jpg_in
8
17
19
3
17
17
curr_source_adr
end_source_adr
valid_bytes_last_fetch
start_of_bandstore
end_of_bandstore
max_plane
2
13
30
max_block
start_adr, buff_end_adr
buff_start
half_block_ok_to_read
rd_adv
rd_adv_half_block
line_store_ok_to_write
wradv8line
JPEG decoder
jpg_dec_status
configuration registers
num_buff_lines
7
contone line store interface
bypass_jpg
pixel_out_valid
half-block buffer interface
pixel_out
8
8
0
1
8
jclk
Contone Decoder Unit
6
32
32
pcu_cdu_sel
pcu_rwn
pcu_adr
pcu_dataout
cdu_pcu_rdy
cdu_pcu_datain
jclk
jclk_enable
cfu_cdu_rdadvline
cdu_cfu_linestore_rdy
PEP Controller Unit
Clock, Power Reset
Contone Fifo Unit

FIG. 137

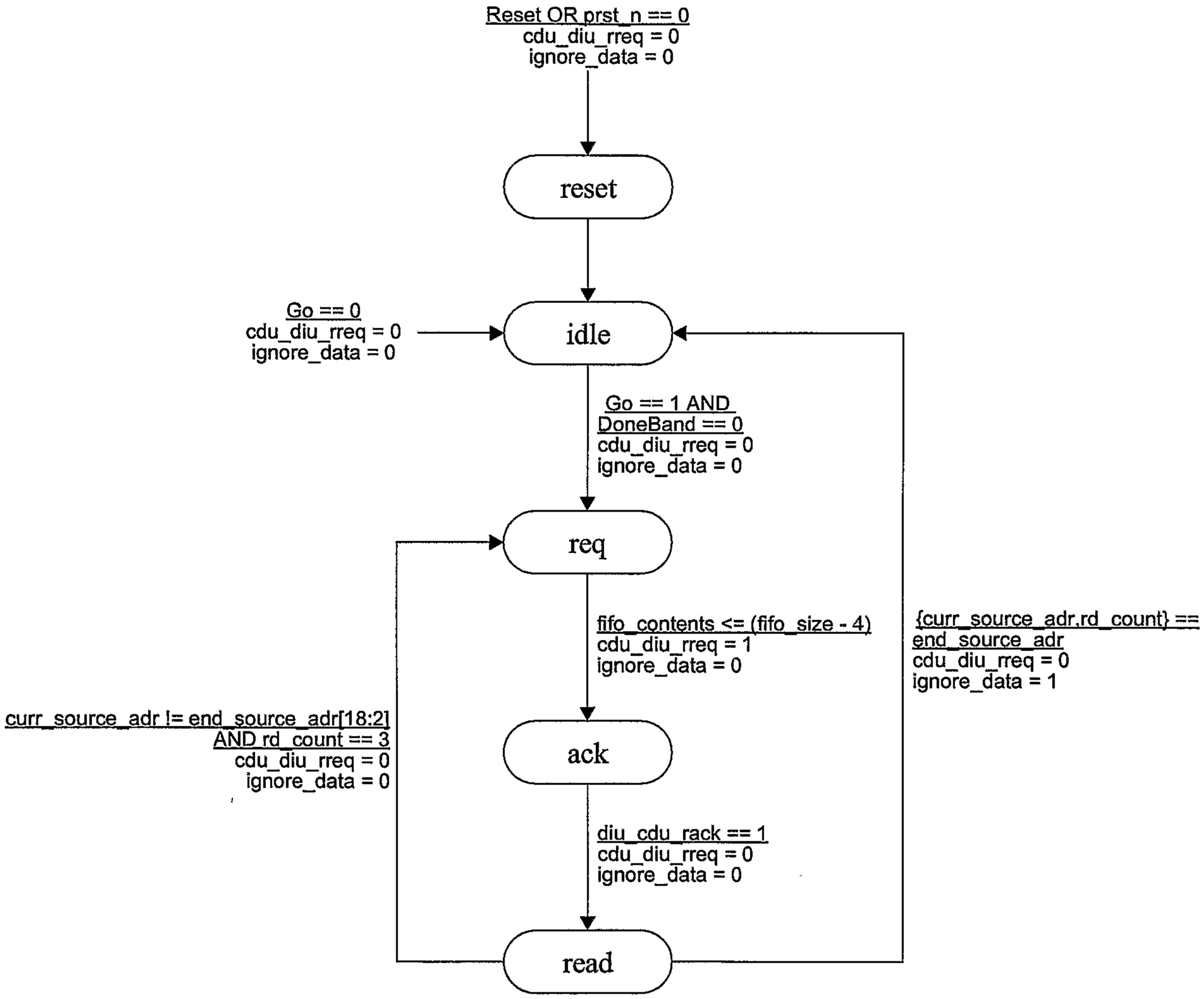

Reset OR prst_n == 0
cdu_diu_rreq = 0
ignore_data = 0
reset
Go == 0
cdu_diu_rreq = 0
ignore_data = 0
idle
Go == 1 AND
DoneBand == 0
cdu_diu_rreq = 0
ignore_data = 0
req
fifo_contents <= (fifo_size - 4)
cdu_diu_rreq = 1
ignore_data = 0
{curr_source_adr,rd_count} ==
end_source_adr
cdu_diu_rreq = 0
ignore_data = 1
curr_source_adr != end_source_adr[18:2]
AND rd_count == 3
cdu_diu_rreq = 0
ignore_data = 0
ack
diu_cdu_rack == 1
cdu_diu_rreq = 0
ignore_data = 0
read

FIG. 138

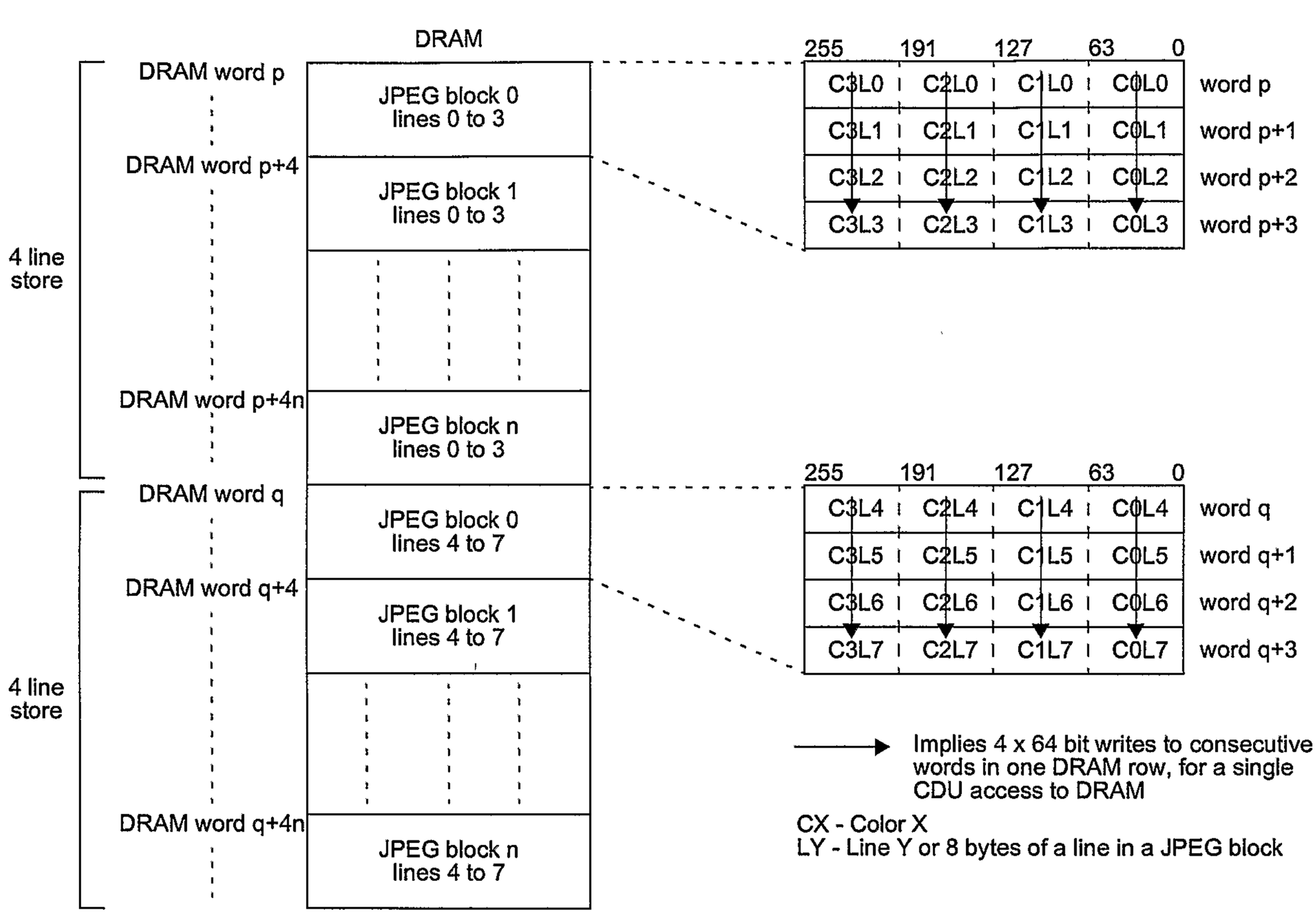

DRAM
DRAM word p
DRAM word p+4
DRAM word p+4n
JPEG block 0 lines 0 to 3
JPEG block 1 lines 0 to 3
JPEG block n lines 0 to 3
4 line store
DRAM word q
DRAM word q+4
DRAM word q+4n
JPEG block 0 lines 4 to 7
JPEG block 1 lines 4 to 7
JPEG block n lines 4 to 7
4 line store
255 191 127 63 0
C3L0 C2L0 C1L0 C0L0 word p
C3L1 C2L1 C1L1 C0L1 word p+1
C3L2 C2L2 C1L2 C0L2 word p+2
C3L3 C2L3 C1L3 C0L3 word p+3
255 191 127 63 0
C3L4 C2L4 C1L4 C0L4 word q
C3L5 C2L5 C1L5 C0L5 word q+1
C3L6 C2L6 C1L6 C0L6 word q+2
C3L7 C2L7 C1L7 C0L7 word q+3
Implies 4 x 64 bit writes to consecutive words in one DRAM row, for a single CDU access to DRAM
CX - Color X
LY - Line Y or 8 bytes of a line in a JPEG block

FIG. 139

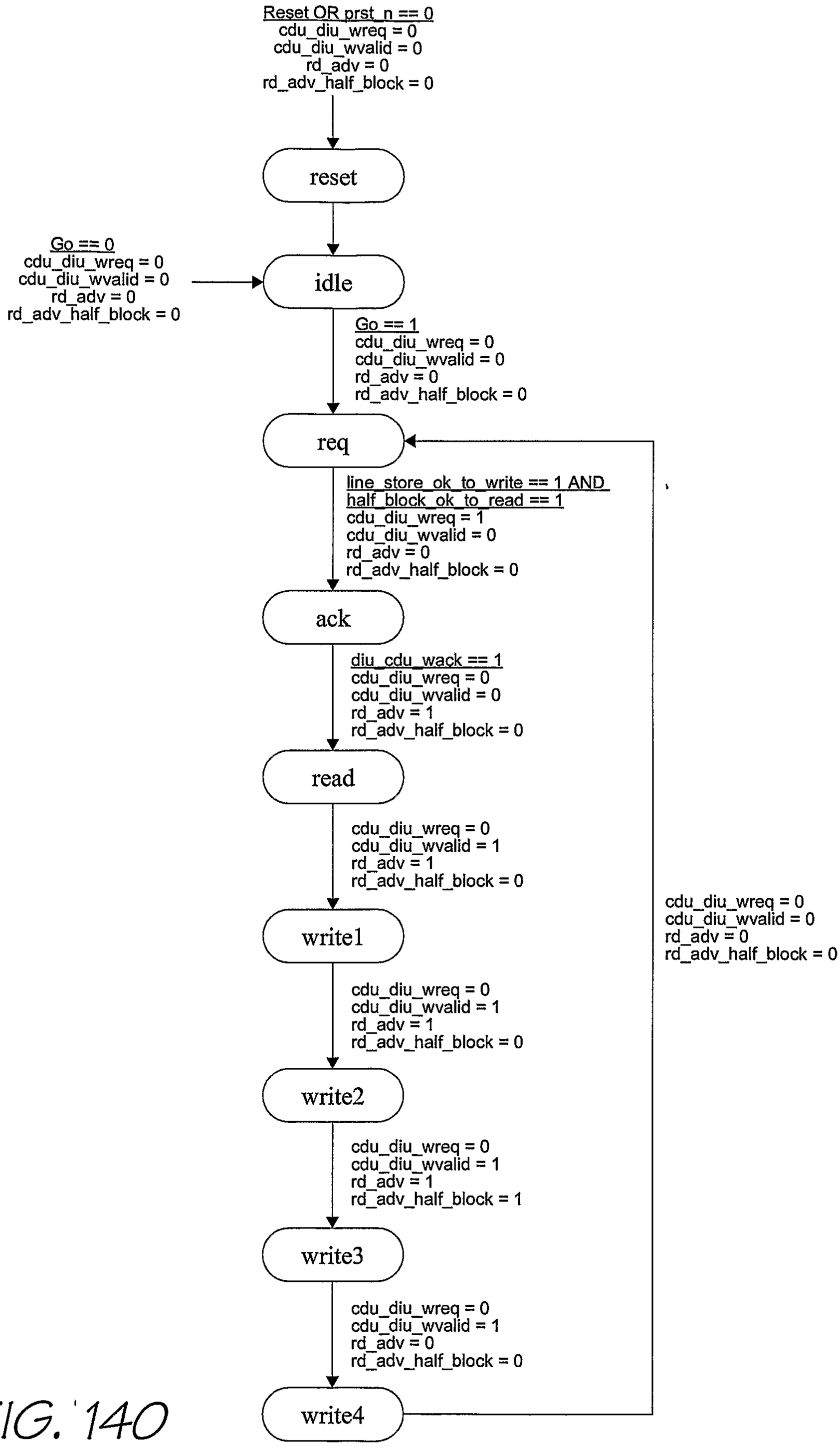

Reset OR prst_n == 0
cdu_diu_wreq = 0
cdu_diu_wvalid = 0
rd_adv = 0
rd_adv_half_block = 0
reset
Go == 0
cdu_diu_wreq = 0
cdu_diu_wvalid = 0
rd_adv = 0
rd_adv_half_block = 0
idle
Go == 1
cdu_diu_wreq = 0
cdu_diu_wvalid = 0
rd_adv = 0
rd_adv_half_block = 0
req
line_store_ok_to_write == 1 AND
half_block_ok_to_read == 1
cdu_diu_wreq = 1
cdu_diu_wvalid = 0
rd_adv = 0
rd_adv_half_block = 0
ack
diu_cdu_wack == 1
cdu_diu_wreq = 0
cdu_diu_wvalid = 0
rd_adv = 1
rd_adv_half_block = 0
read
cdu_diu_wreq = 0
cdu_diu_wvalid = 1
rd_adv = 1
rd_adv_half_block = 0
write1
cdu_diu_wreq = 0
cdu_diu_wvalid = 1
rd_adv = 1
rd_adv_half_block = 0
write2
cdu_diu_wreq = 0
cdu_diu_wvalid = 1
rd_adv = 1
rd_adv_half_block = 1
write3
cdu_diu_wreq = 0
cdu_diu_wvalid = 1
rd_adv = 0
rd_adv_half_block = 0
write4
cdu_diu_wreq = 0
cdu_diu_wvalid = 0
rd_adv = 0
rd_adv_half_block = 0

FIG. 140

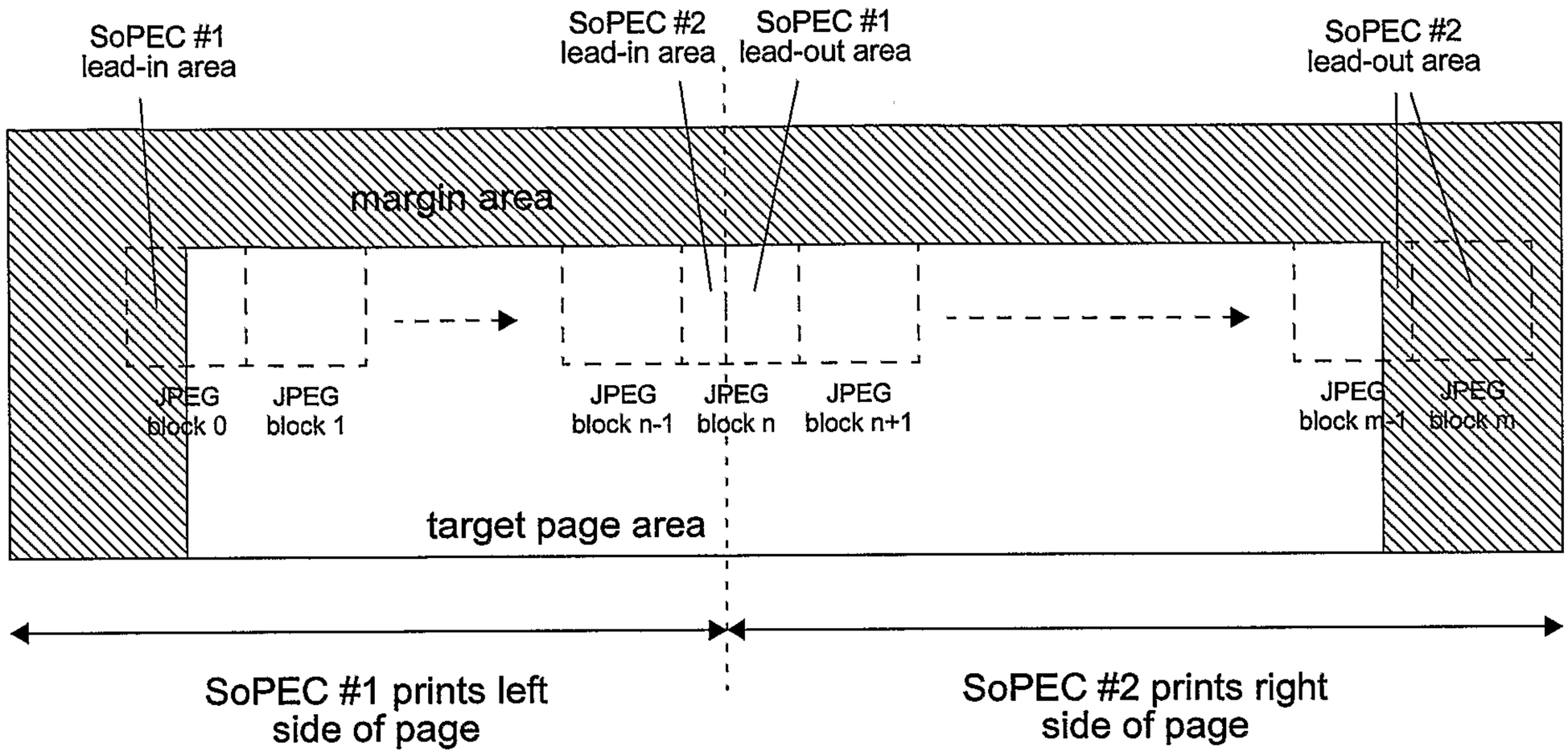

SoPEC #1
lead-in area
SoPEC #2
lead-in area
SoPEC #1
lead-out area
SoPEC #2
lead-out area
margin area
JPEG
block 0
JPEG
block 1
JPEG
block n-1
JPEG
block n
JPEG
block n+1
JPEG
block m-1
JPEG
block m
target page area
SoPEC #1 prints left
side of page
SoPEC #2 prints right
side of page

FIG. 141

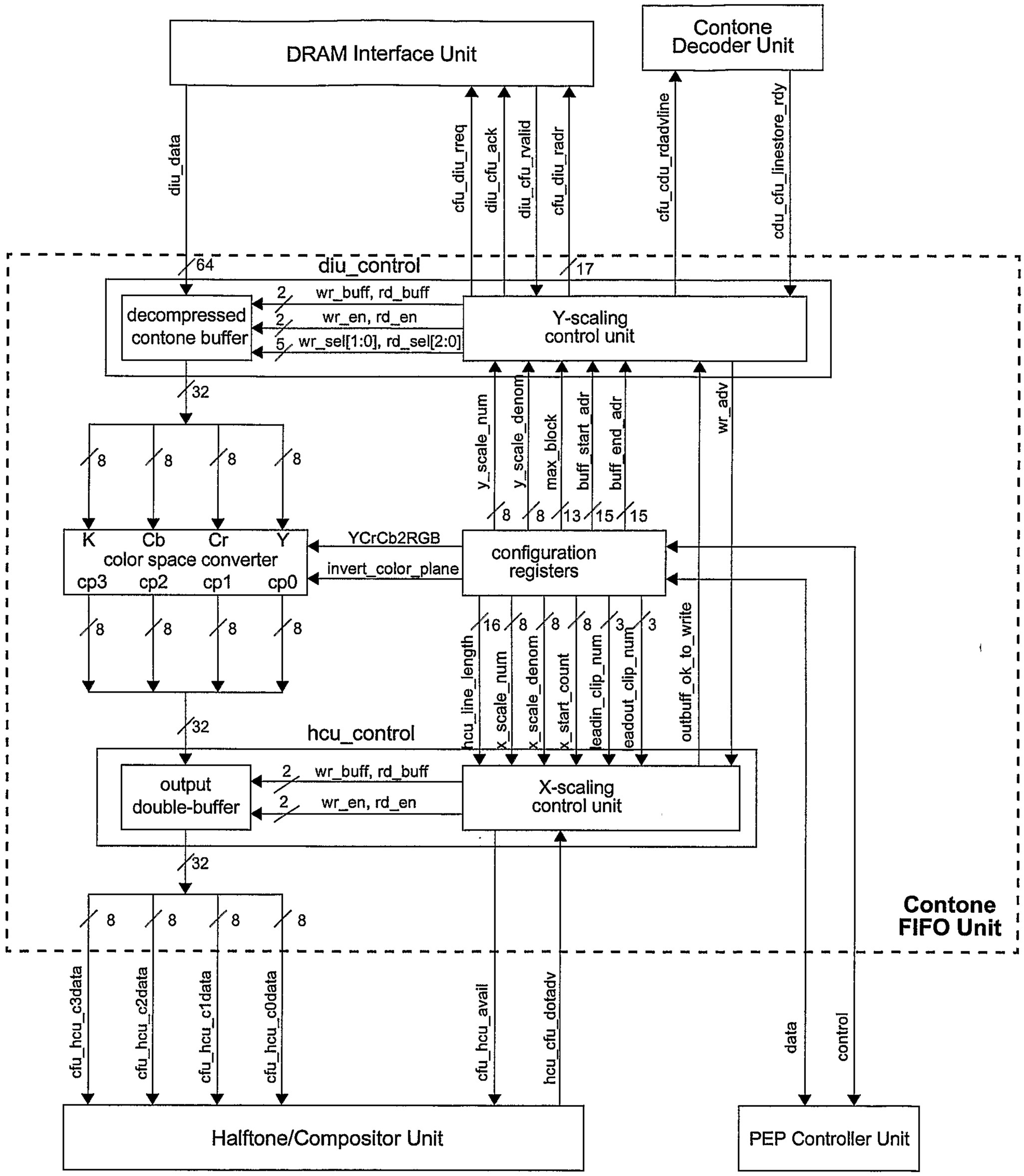

DRAM Interface Unit
Contone Decoder Unit
diu_data
cfu_diu_rreq
diu_cfu_ack
diu_cfu_rvalid
cfu_diu_radr
cfu_cdu_rdadvline
cdu_cfu_linestore_rdy
diu_control
64
17
2
wr_buff, rd_buff
2
wr_en, rd_en
5
wr_sel[1:0], rd_sel[2:0]
decompressed contone buffer
Y-scaling control unit
32
y_scale_num
y_scale_denom
max_block
buff_start_adr
buff_end_adr
wr_adv
8
8
13
15
15
K
Cb
Cr
Y
color space converter
cp3
cp2
cp1
cp0
YCrCb2RGB
invert_color_plane
configuration registers
16
8
8
8
3
3
hcu_line_length
x_scale_num
x_scale_denom
x_start_count
leadin_clip_num
leadout_clip_num
outbuff_ok_to_write
hcu_control
output double-buffer
2
wr_buff, rd_buff
2
wr_en, rd_en
X-scaling control unit
32
Contone FIFO Unit
cfu_hcu_c3data
cfu_hcu_c2data
cfu_hcu_c1data
cfu_hcu_c0data
cfu_hcu_avail
hcu_cfu_dotadv
data
control
Halftone/Compositor Unit
PEP Controller Unit

FIG. 142

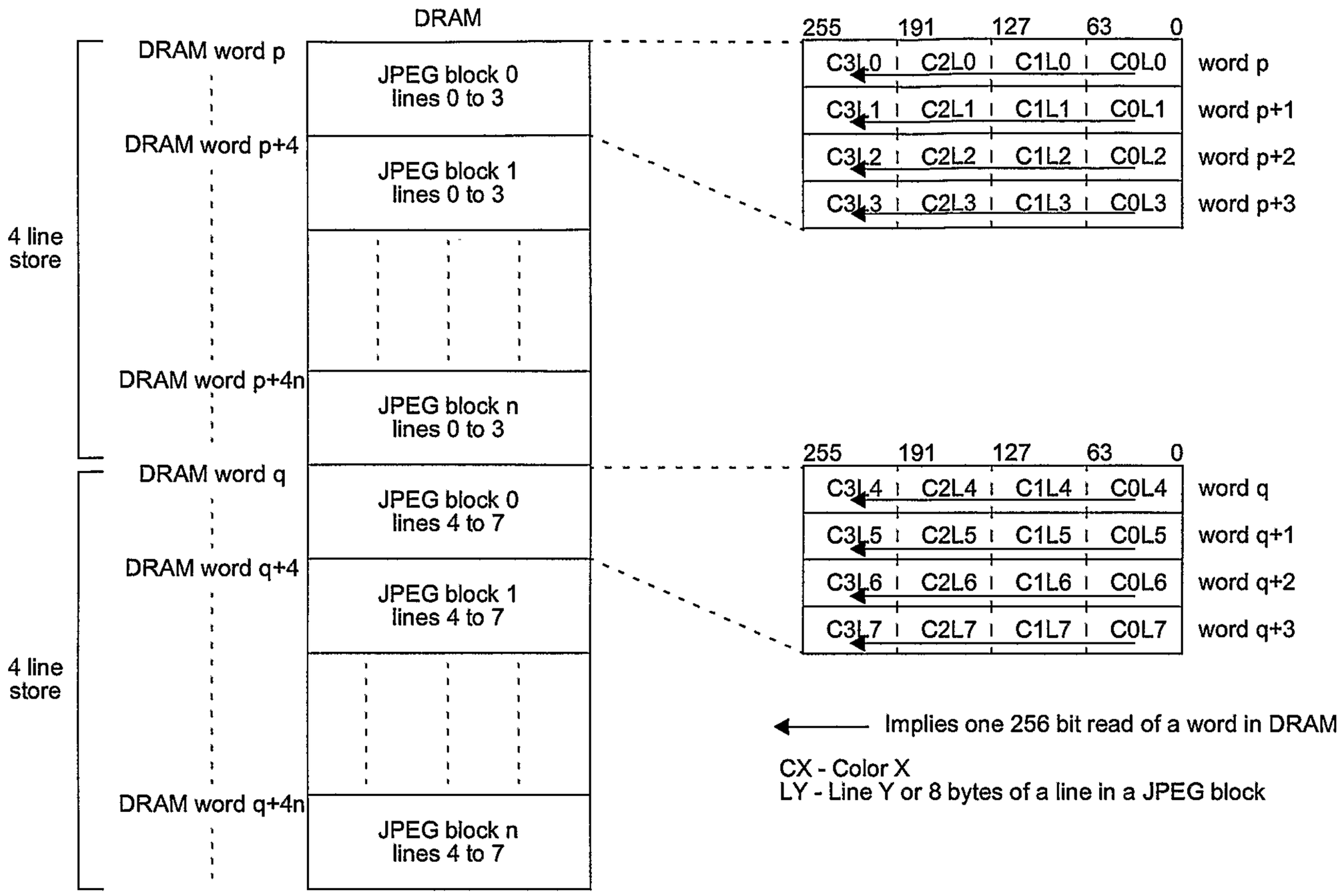

DRAM
DRAM word p
DRAM word p+4
DRAM word p+4n
JPEG block 0 lines 0 to 3
JPEG block 1 lines 0 to 3
JPEG block n lines 0 to 3
4 line store
DRAM word q
DRAM word q+4
DRAM word q+4n
JPEG block 0 lines 4 to 7
JPEG block 1 lines 4 to 7
JPEG block n lines 4 to 7
4 line store
255 191 127 63 0
C3L0 C2L0 C1L0 C0L0 word p
C3L1 C2L1 C1L1 C0L1 word p+1
C3L2 C2L2 C1L2 C0L2 word p+2
C3L3 C2L3 C1L3 C0L3 word p+3
255 191 127 63 0
C3L4 C2L4 C1L4 C0L4 word q
C3L5 C2L5 C1L5 C0L5 word q+1
C3L6 C2L6 C1L6 C0L6 word q+2
C3L7 C2L7 C1L7 C0L7 word q+3
Implies one 256 bit read of a word in DRAM
CX - Color X
LY - Line Y or 8 bytes of a line in a JPEG block

FIG. 143

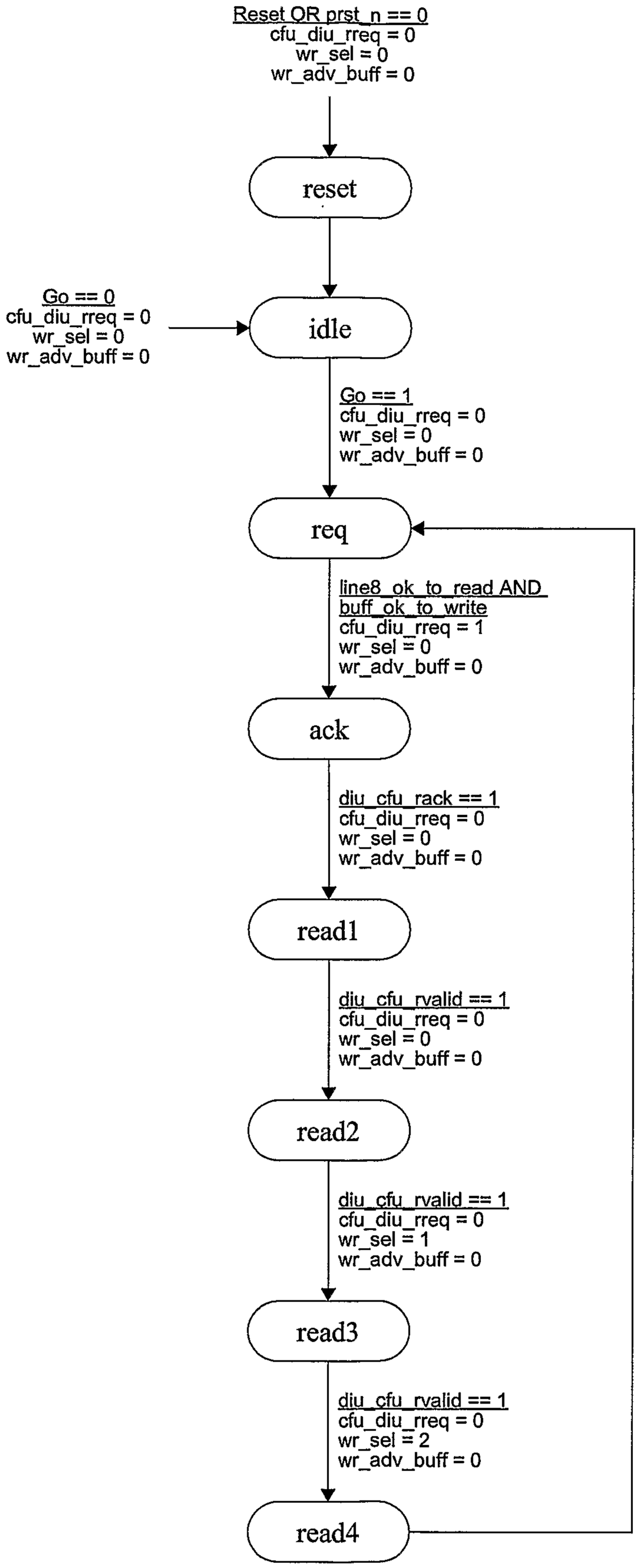

Reset OR prst_n == 0
cfu_diu_rreq = 0
wr_sel = 0
wr_adv_buff = 0
reset
Go == 0
cfu_diu_rreq = 0
wr_sel = 0
wr_adv_buff = 0
idle
Go == 1
cfu_diu_rreq = 0
wr_sel = 0
wr_adv_buff = 0
req
line8_ok_to_read AND
buff_ok_to_write
cfu_diu_rreq = 1
wr_sel = 0
wr_adv_buff = 0
ack
diu_cfu_rack == 1
cfu_diu_rreq = 0
wr_sel = 0
wr_adv_buff = 0
read1
diu_cfu_rvalid == 1
cfu_diu_rreq = 0
wr_sel = 0
wr_adv_buff = 0
diu_cfu_rvalid == 1 AND
cfu_diu_rreq = 0
wr_sel = 3
wr_adv_buff = 1
read2
diu_cfu_rvalid == 1
cfu_diu_rreq = 0
wr_sel = 1
wr_adv_buff = 0
read3
diu_cfu_rvalid == 1
cfu_diu_rreq = 0
wr_sel = 2
wr_adv_buff = 0
read4

FIG. 144

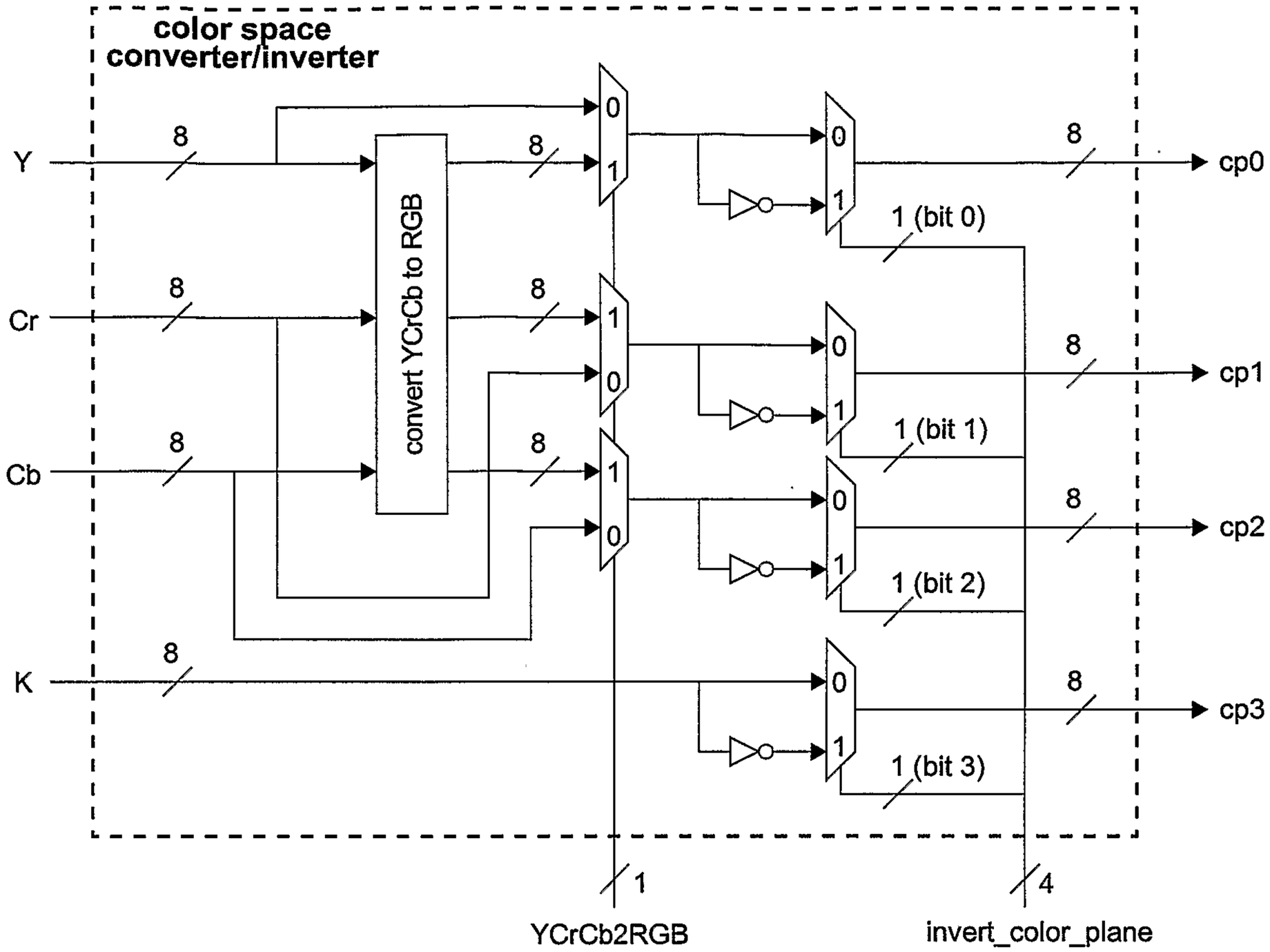


FIG. 145

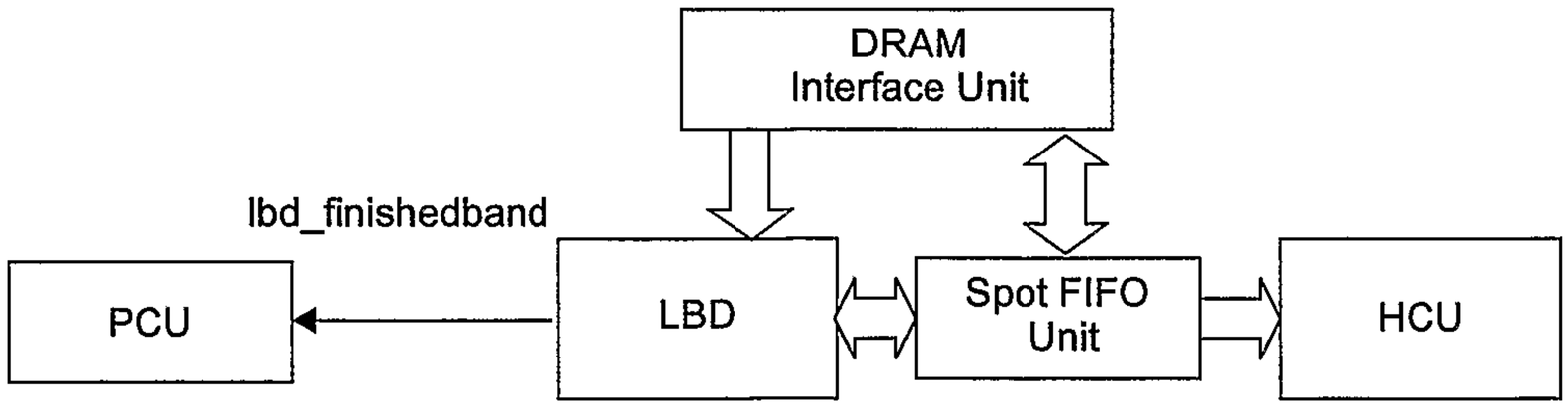


FIG. 146

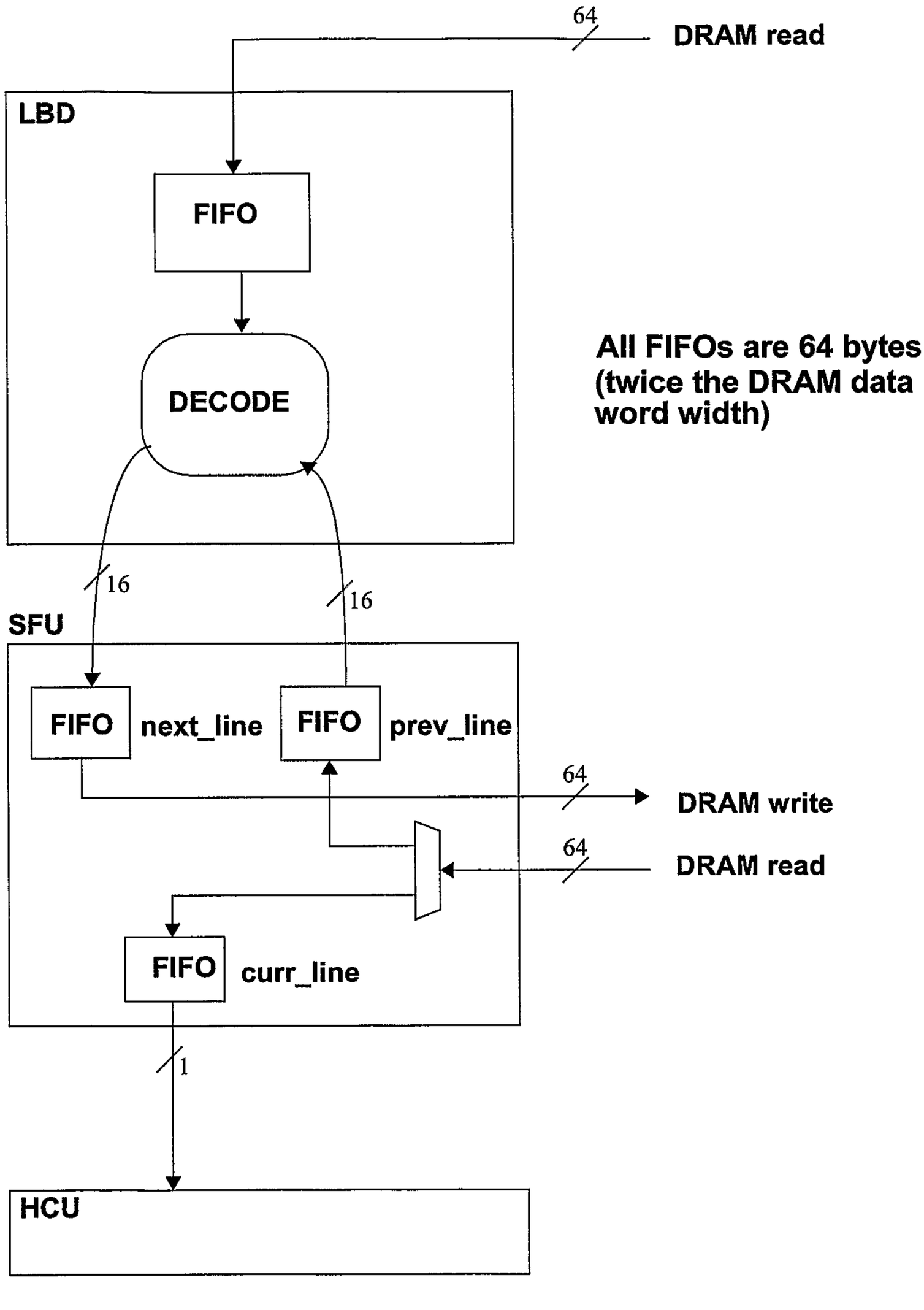

64
DRAM read
LBD
FIFO
DECODE
All FIFOs are 64 bytes (twice the DRAM data word width)
16
16
SFU
FIFO
next_line
FIFO
prev_line
64
DRAM write
64
DRAM read
FIFO
curr_line
1
HCU

FIG. 147

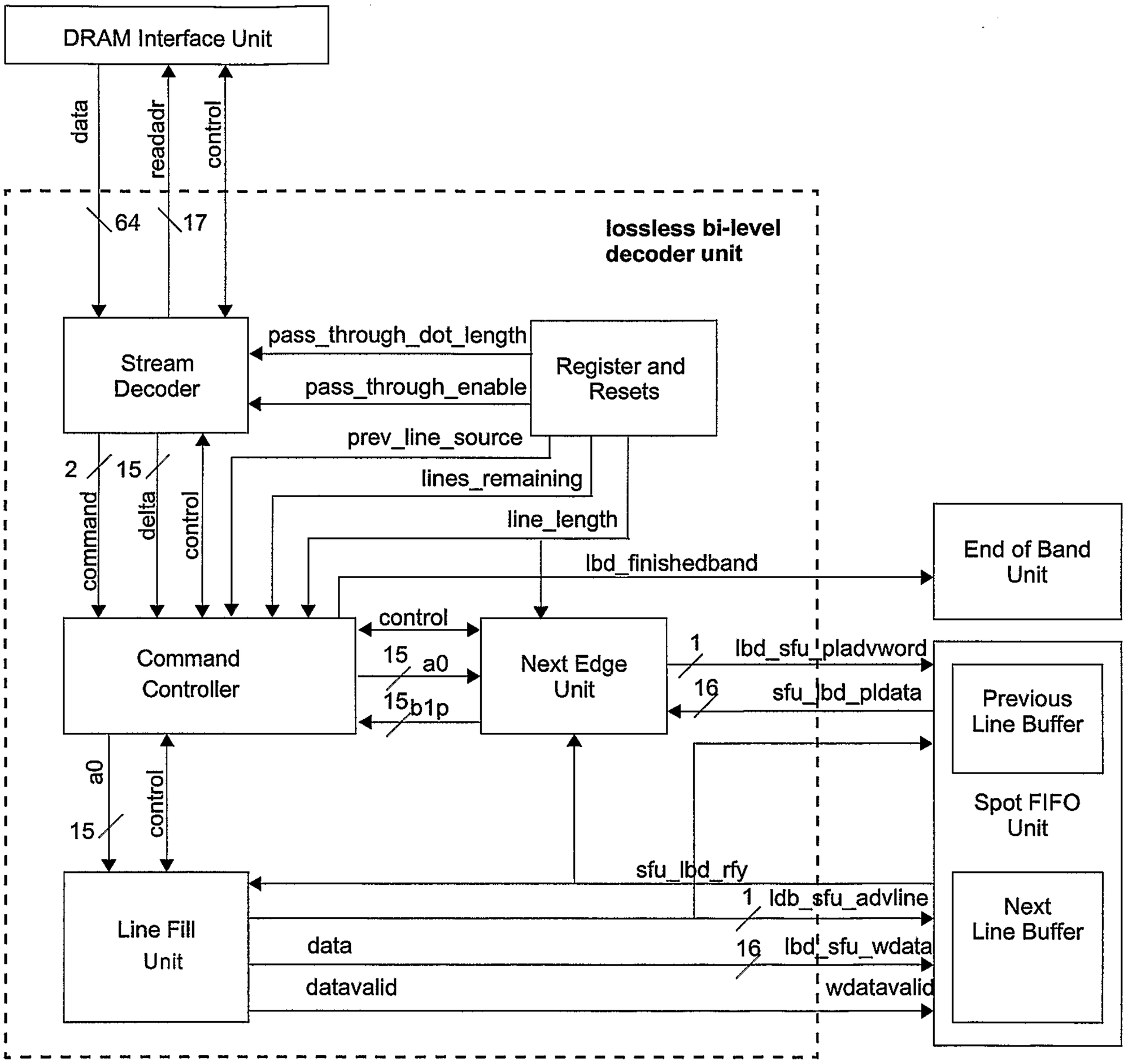

DRAM Interface Unit
data
readadr
control
64
17
lossless bi-level decoder unit
Stream Decoder
pass_through_dot_length
pass_through_enable
Register and Resets
prev_line_source
lines_remaining
line_length
2
15
command
delta
control
lbd_finishedband
End of Band Unit
control
Command Controller
15
a0
15
b1p
Next Edge Unit
1
lbd_sfu_pladvword
16
sfu_lbd_pldata
Previous Line Buffer
Spot FIFO Unit
a0
15
control
sfu_lbd_rfy
1
ldb_sfu_advline
Next Line Buffer
Line Fill Unit
data
16
lbd_sfu_wdata
datavalid
wdatavalid

FIG. 148

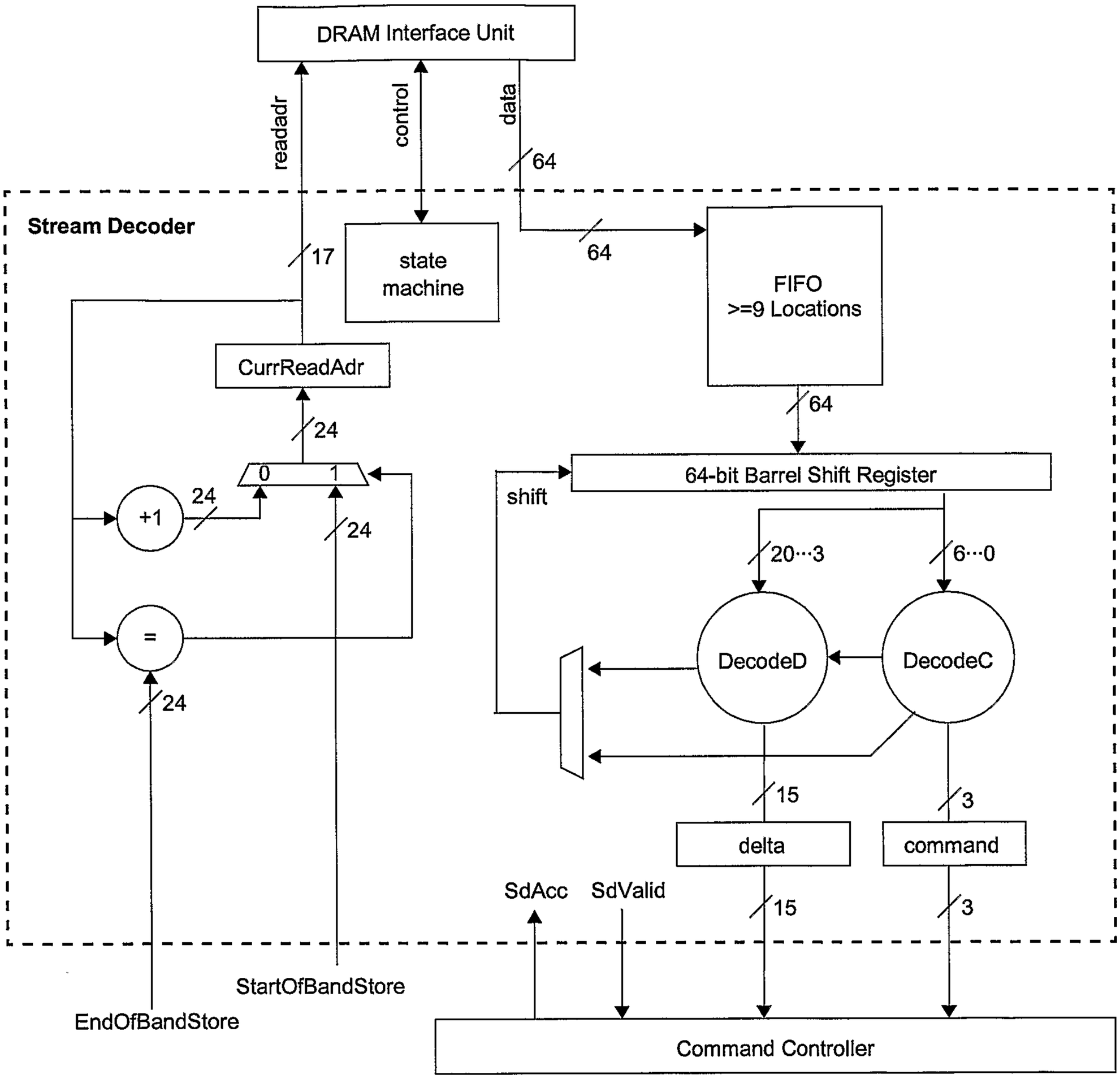

DRAM Interface Unit
readadr
control
data
64
Stream Decoder
17
state machine
64
FIFO >=9 Locations
CurrReadAdr
24
64
0
1
64-bit Barrel Shift Register
shift
+1
24
24
20···3
6···0
=
DecodeD
DecodeC
24
15
3
delta
command
SdAcc
SdValid
15
3
StartOfBandStore
EndOfBandStore
Command Controller

FIG. 149

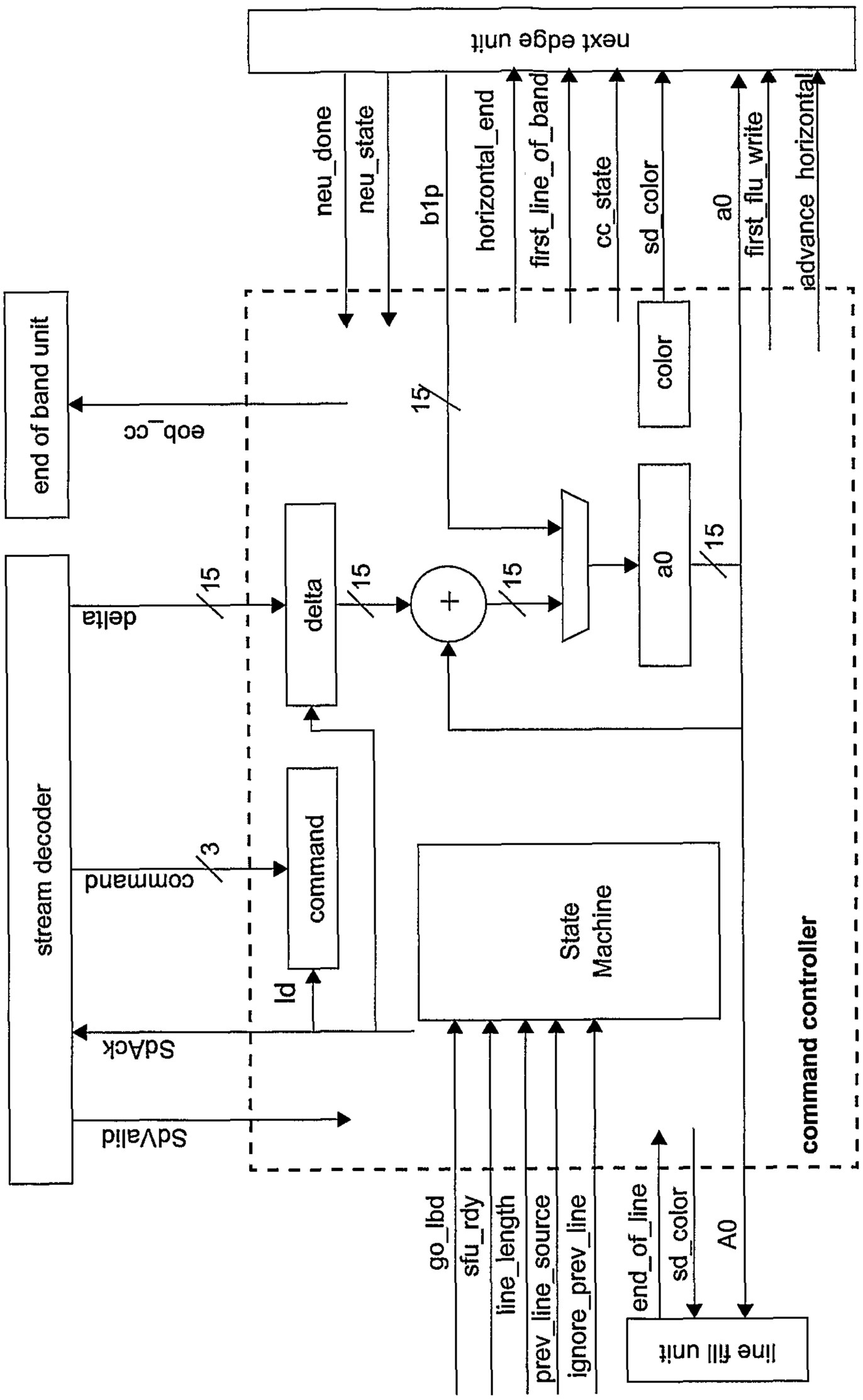

next edge unit
neu_done
neu_state
b1p
horizontal_end
first_line_of_band
cc_state
sd_color
a0
first_flu_write
advance_horizontal
end of band unit
eob_cc
15
color
delta
15
15
15
a0
15
stream decoder
command
3
command
ld
State Machine
SdAck
SdValid
command controller
go_lbd
sfu_rdy
line_length
prev_line_source
ignore_prev_line
end_of_line
sd_color
A0
line fill unit

FIG. 150

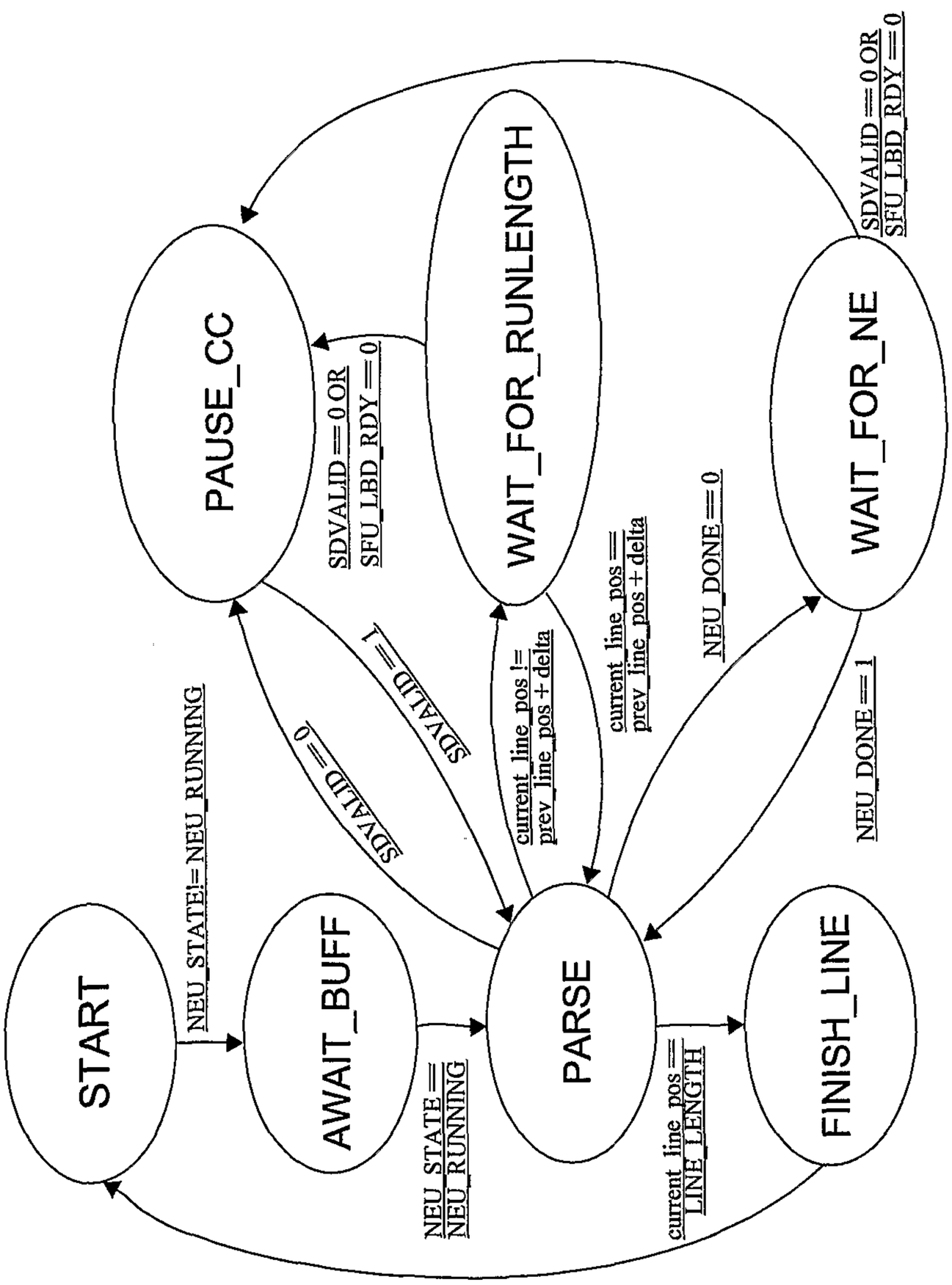

START
NEU_STATE!= NEU_RUNNING
AWAIT_BUFF
NEU_STATE == NEU_RUNNING
PARSE
current_line_pos == LINE_LENGTH
FINISH_LINE
SDVALID == 0
SDVALID == 1
current_line_pos != prev_line_pos + delta
current_line_pos == prev_line_pos + delta
NEU_DONE == 0
NEU_DONE == 1
SDVALID == 0 OR SFU_LBD_RDY == 0
SDVALID == 0 OR SFU_LBD_RDY == 0

FIG. 151

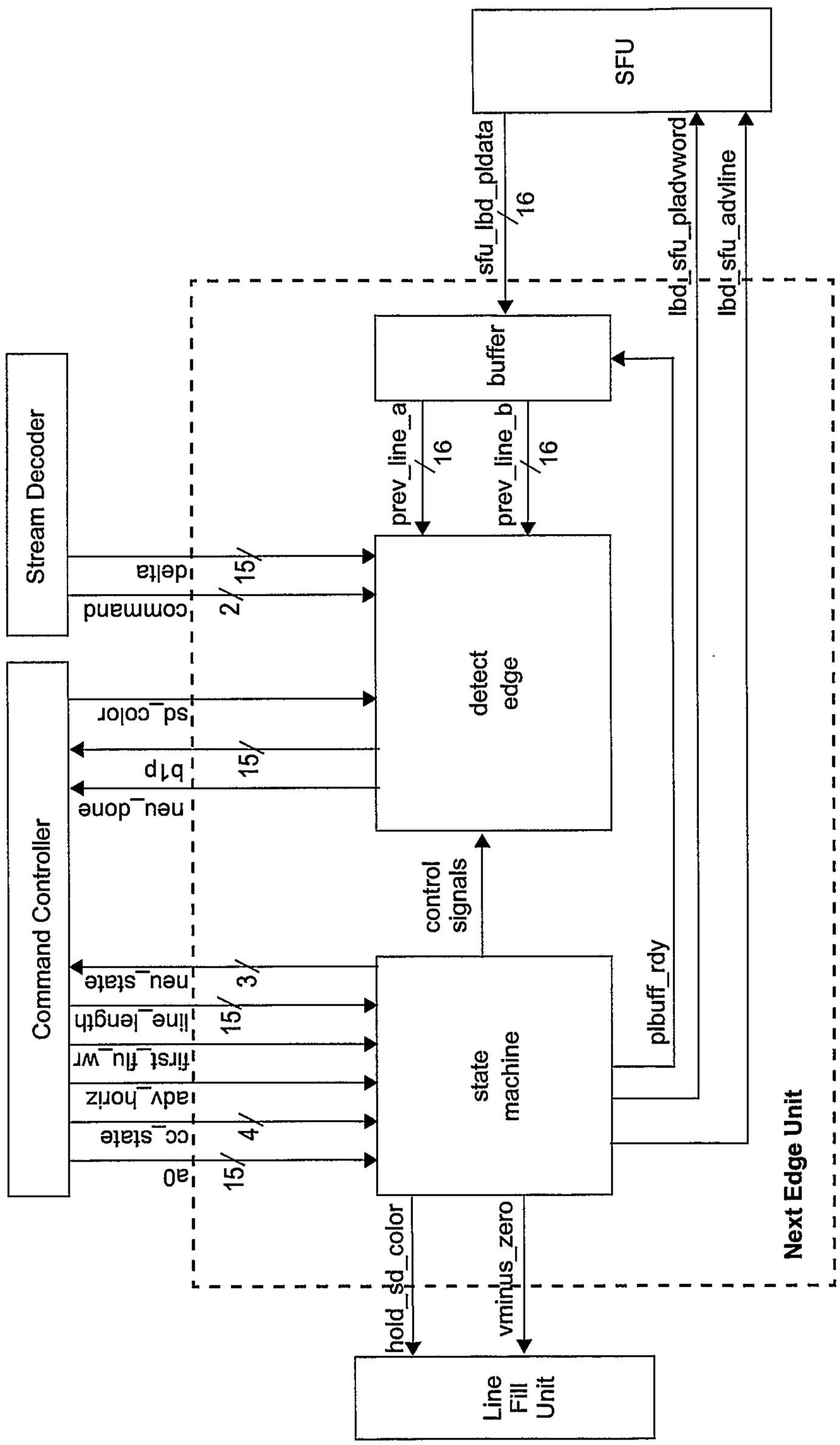

SFU
sfu_lbd_pldata
16
lbd_sfu_pladvword
lbd_sfu_advline
buffer
prev_line_a
16
prev_line_b
16
Stream Decoder
delta
15
command
2
detect edge
sd_color
b1p
15
neu_done
Command Controller
control signals
neu_state
3
line_length
15
first_flu_wr
adv_horiz
cc_state
4
a0
15
state machine
plbuff_rdy
hold_sd_color
vminus_zero
Next Edge Unit
Line Fill Unit

FIG. 152

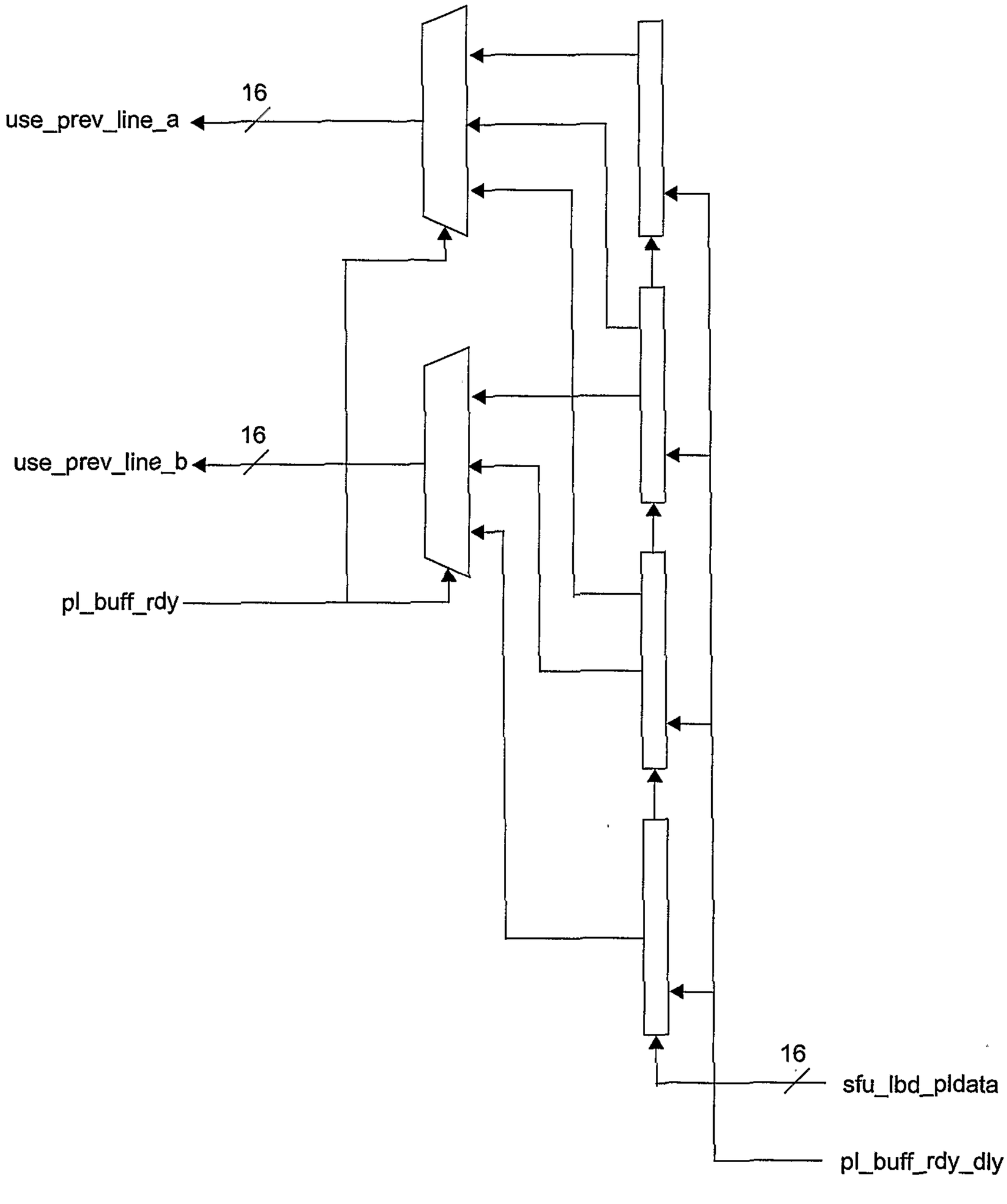

16
use_prev_line_a
16
use_prev_line_b
pl_buff_rdy
16
sfu_lbd_pldata
pl_buff_rdy_dly

FIG. 153

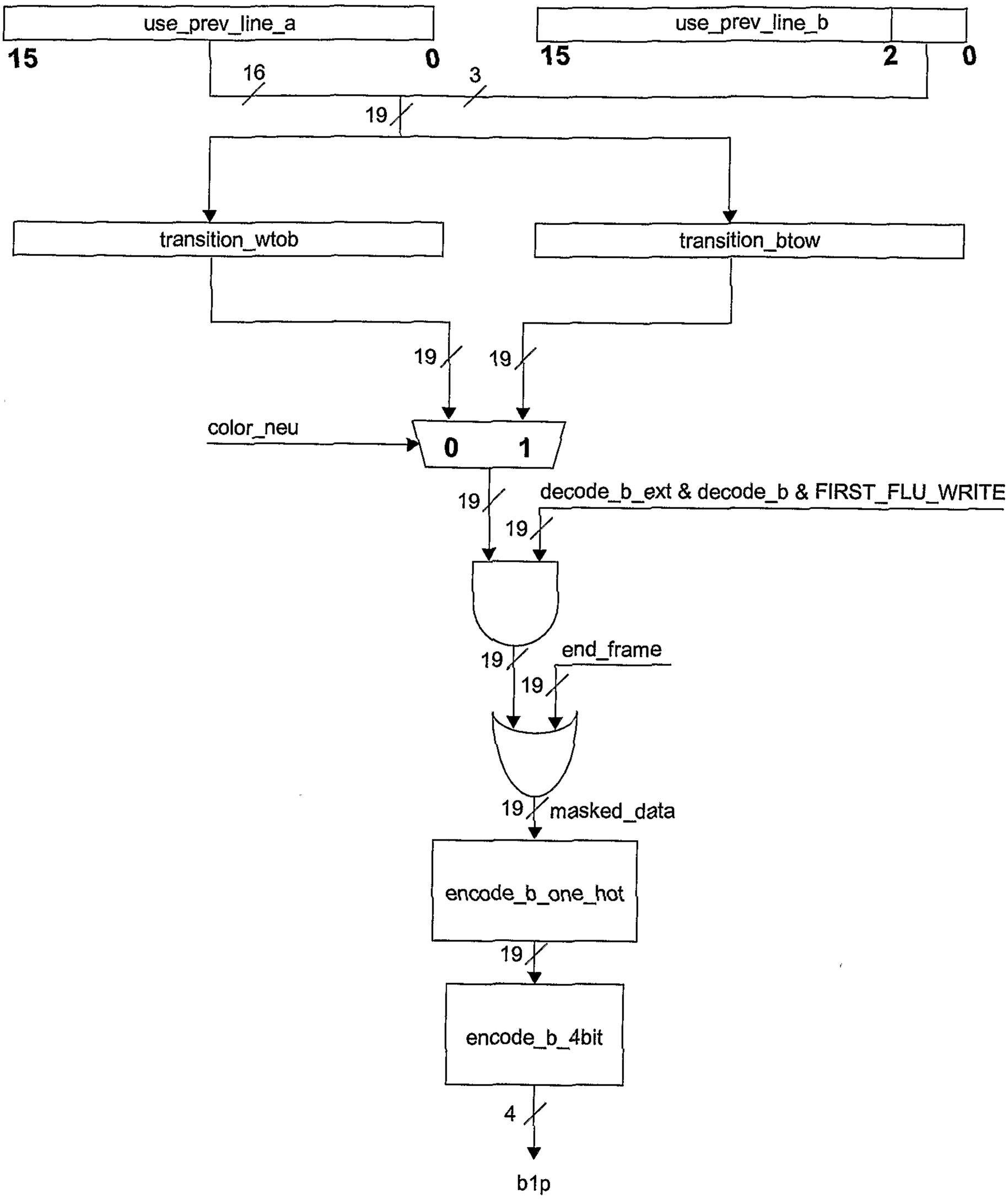

use_prev_line_a
15
0
use_prev_line_b
15
2
0
16
3
19
transition_wtob
transition_btow
19
19
color_neu
0
1
19
decode_b_ext & decode_b & FIRST_FLU_WRITE
19
19
end_frame
19
masked_data
encode_b_one_hot
19
encode_b_4bit
4
b1p

FIG. 154

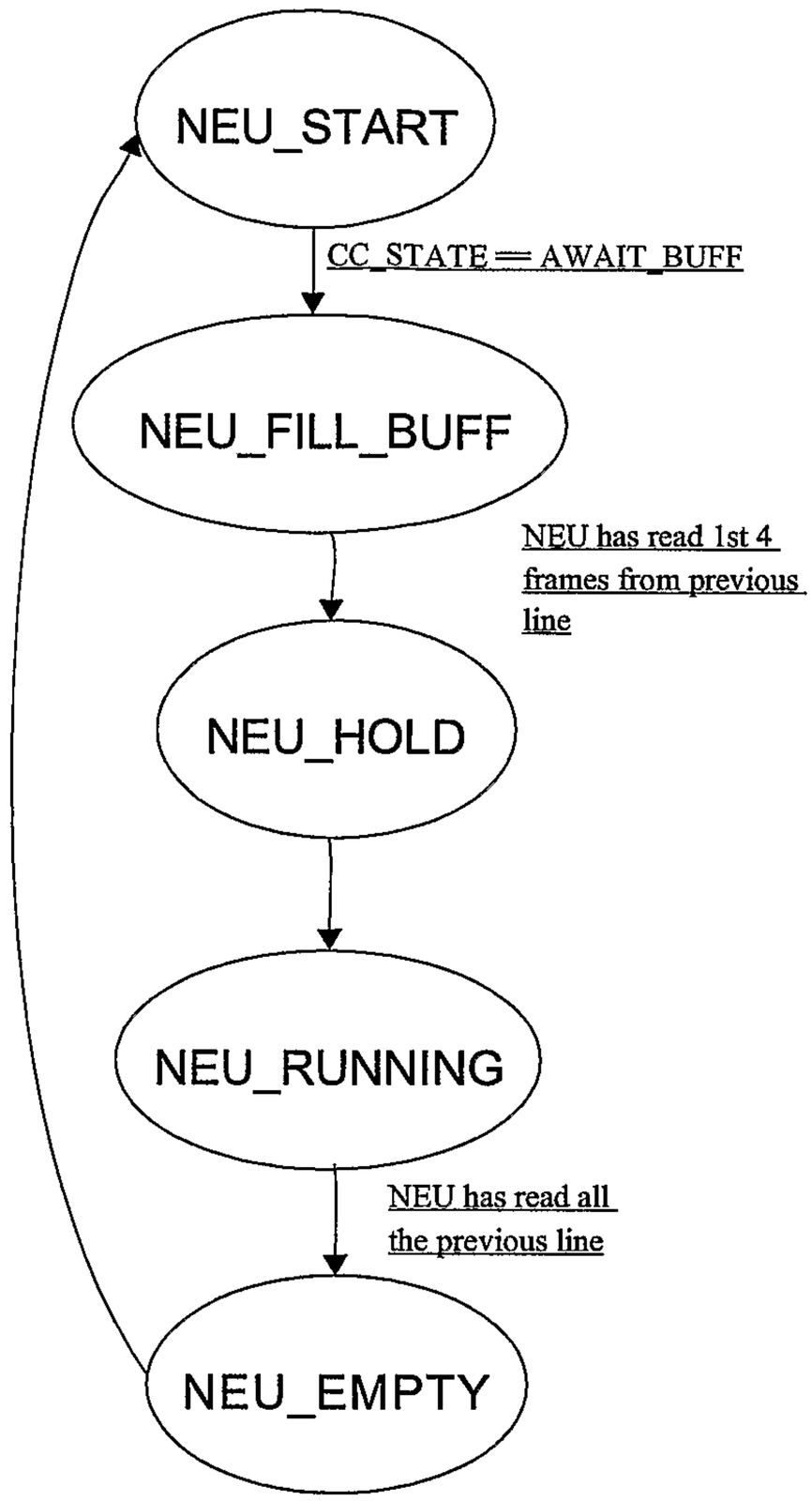

NEU_START
CC_STATE = AWAIT_BUFF
NEU_FILL_BUFF
NEU has read 1st 4 frames from previous line
NEU_HOLD
NEU_RUNNING
NEU has read all the previous line
NEU_EMPTY

FIG. 155

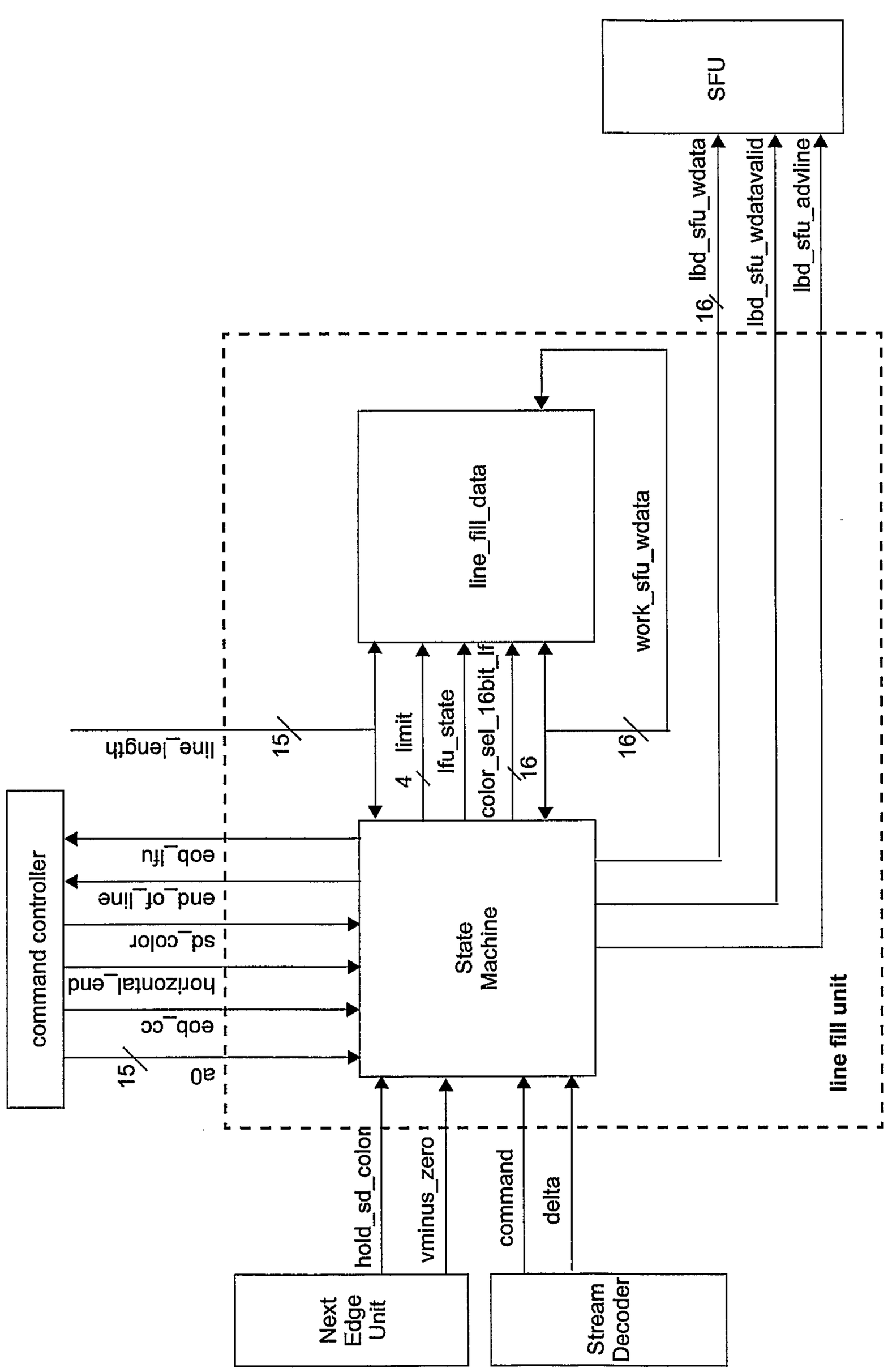

SFU
lbd_sfu_wdata
lbd_sfu_wdatavalid
lbd_sfu_advline
16
line_fill_data
work_sfu_wdata
line_length
15
4
limit
lfu_state
color_sel_16bit_lf
16
16
command controller
eob_lfu
end_of_line
sd_color
horizontal_end
eob_cc
15
a0
State Machine
line fill unit
hold_sd_colon
vminus_zero
command
delta
Next Edge Unit
Stream Decoder

FIG. 156

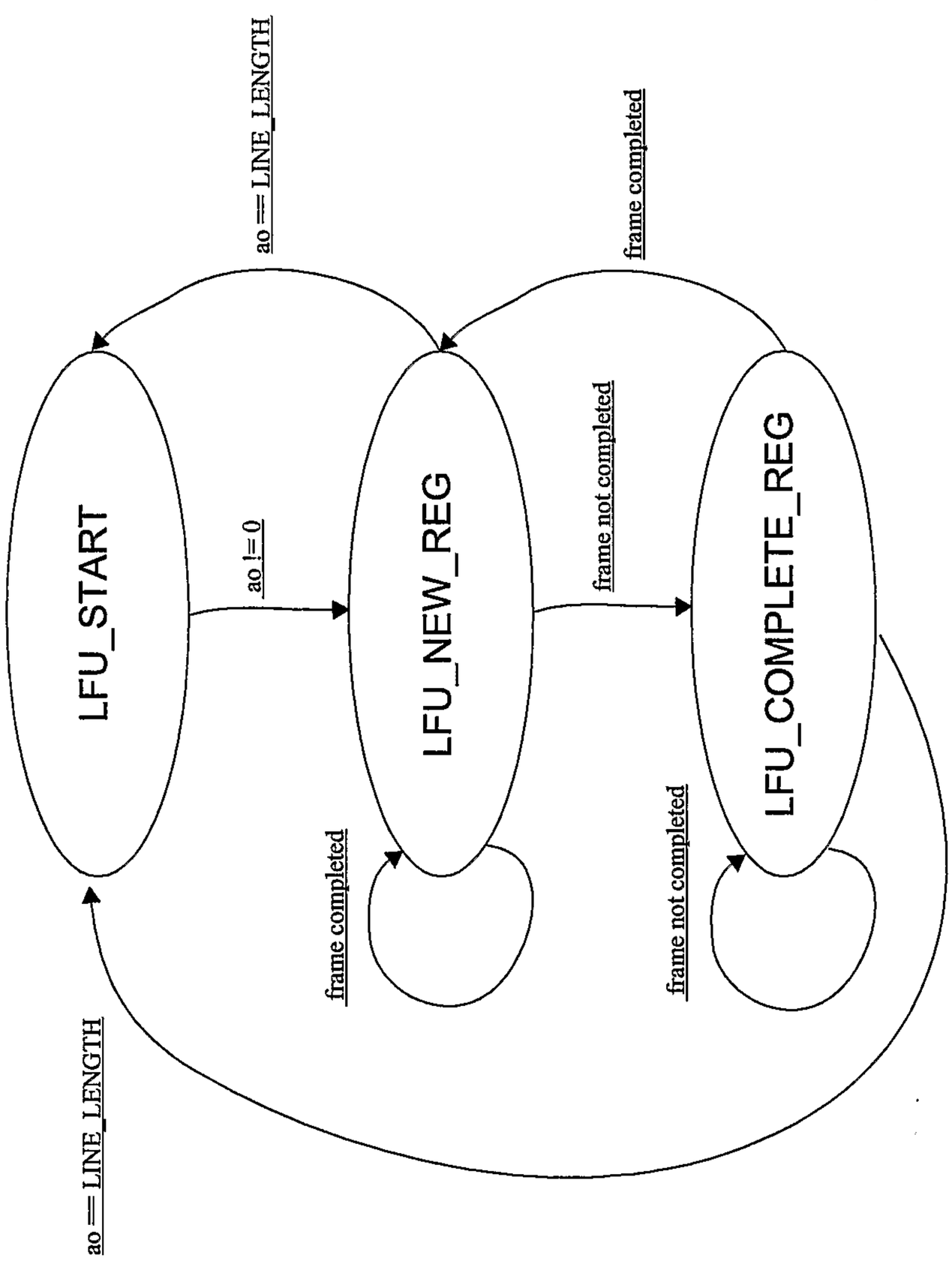

ao == LINE_LENGTH
LFU_START
ao != 0
frame completed
LFU_NEW_REG
frame not completed
frame completed
LFU_COMPLETE_REG
frame not completed
ao == LINE_LENGTH

FIG. 157

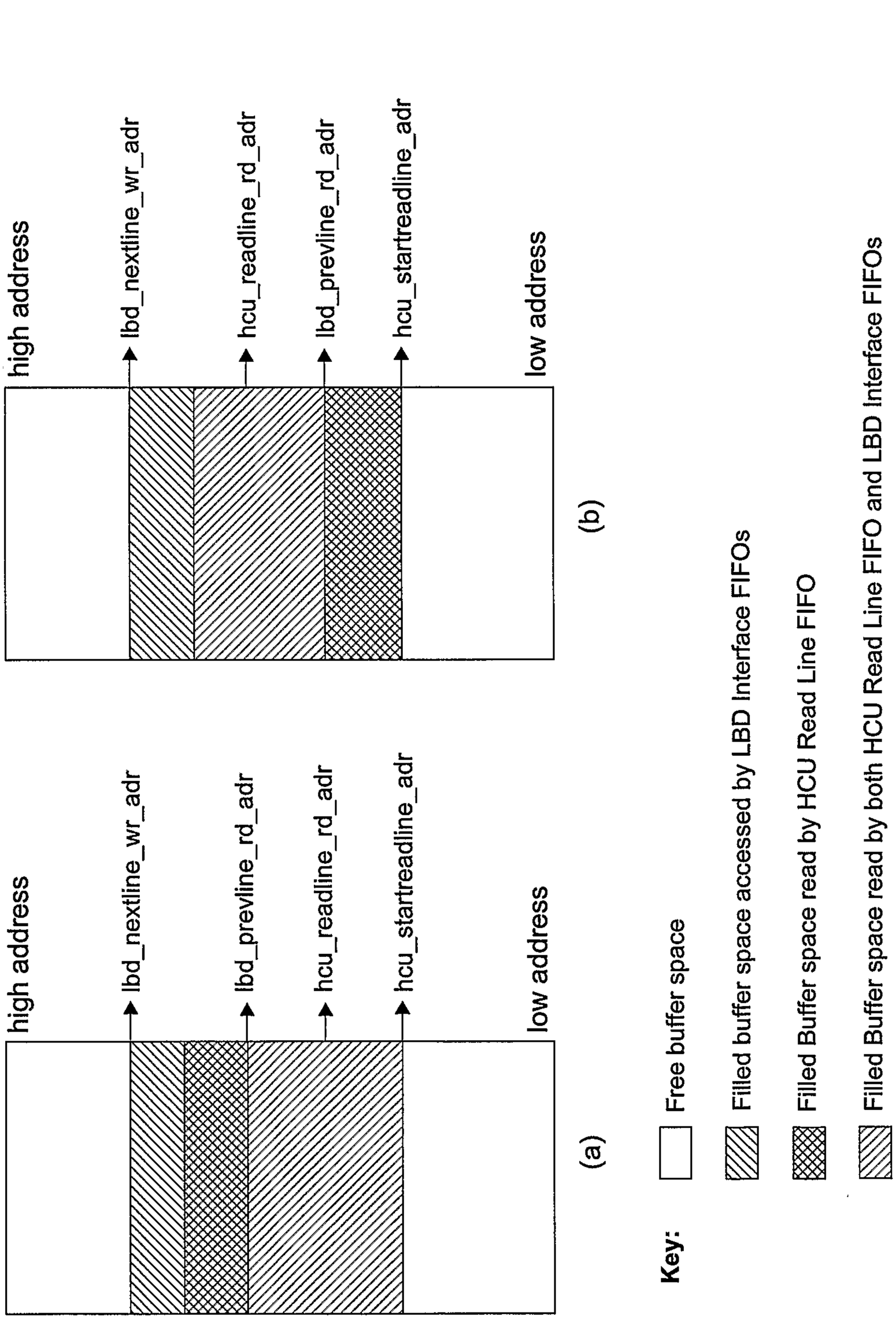

high address
lbd_nextline_wr_adr
hcu_readline_rd_adr
lbd_prevline_rd_adr
hcu_startreadline_adr
low address
(b)
high address
lbd_nextline_wr_adr
lbd_prevline_rd_adr
hcu_readline_rd_adr
hcu_startreadline_adr
low address
(a)
Free buffer space
Filled buffer space accessed by LBD Interface FIFOs
Filled buffer space read by HCU Read Line FIFO
Filled Buffer space read by both HCU Read Line FIFO and LBD Interface FIFOs
Key:

FIG. 158

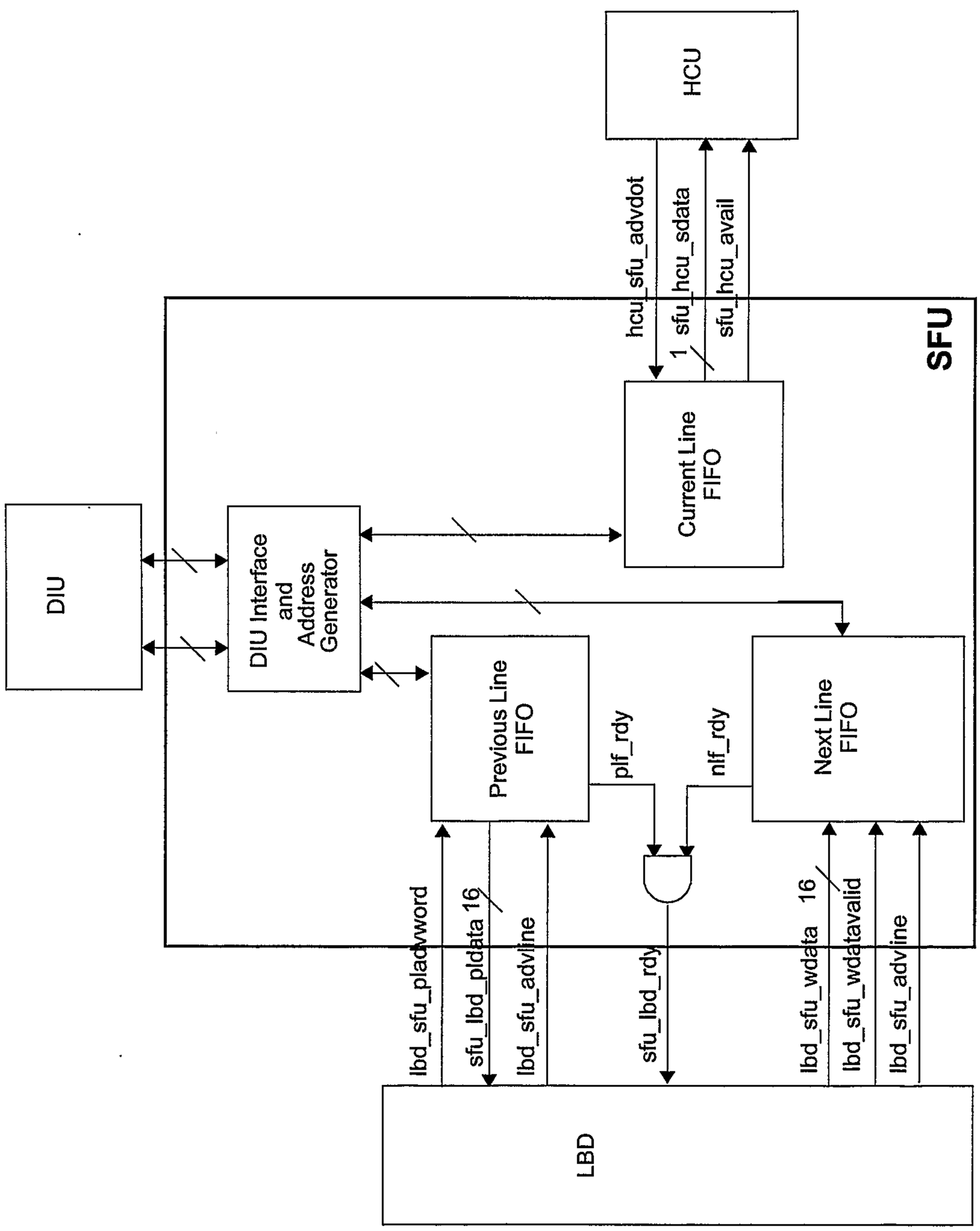

HCU
hcu_sfu_advdot
1 sfu_hcu_sdata
sfu_hcu_avail
SFU
Current Line FIFO
DIU
DIU Interface and Address Generator
Previous Line FIFO
plf_rdy
nlf_rdy
Next Line FIFO
lbd_sfu_pladvword
sfu_lbd_pldata 16
lbd_sfu_advline
sfu_lbd_rdy
lbd_sfu_wdata 16
lbd_sfu_wdatavalid
lbd_sfu_advline
LBD

FIG. 159

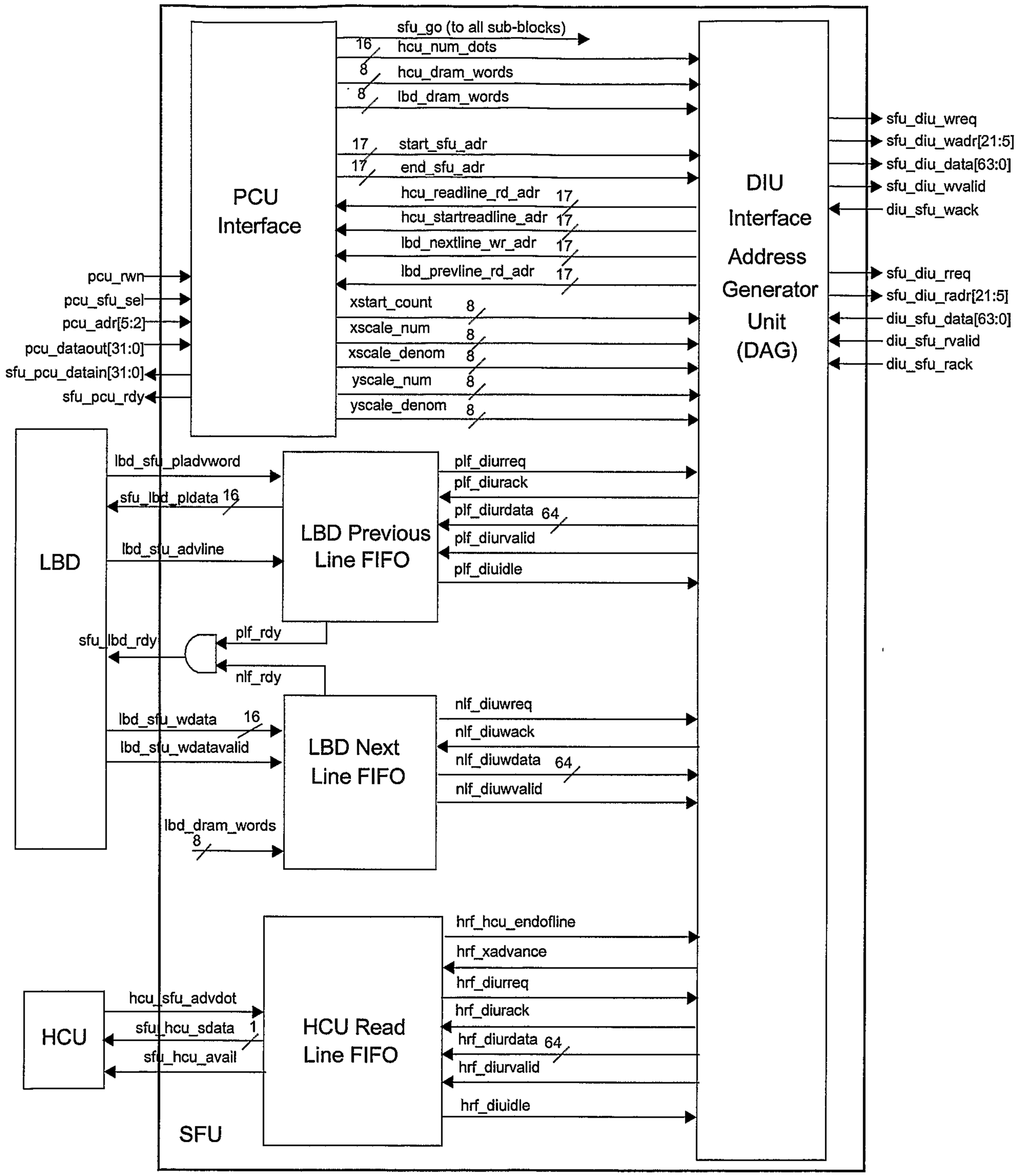

sfu_go (to all sub-blocks)
16 hcu_num_dots
8 hcu_dram_words
8 lbd_dram_words
17 start_sfu_adr
17 end_sfu_adr
hcu_readline_rd_adr 17
hcu_startreadline_adr 17
lbd_nextline_wr_adr 17
lbd_prevline_rd_adr 17
xstart_count 8
xscale_num 8
xscale_denom 8
yscale_num 8
yscale_denom 8
PCU Interface
pcu_rwn
pcu_sfu_sel
pcu_adr[5:2]
pcu_dataout[31:0]
sfu_pcu_datain[31:0]
sfu_pcu_rdy
DIU Interface
Address Generator Unit (DAG)
sfu_diu_wreq
sfu_diu_wadr[21:5]
sfu_diu_data[63:0]
sfu_diu_wvalid
diu_sfu_wack
sfu_diu_rreq
sfu_diu_radr[21:5]
diu_sfu_data[63:0]
diu_sfu_rvalid
diu_sfu_rack
LBD
lbd_sfu_pladvword
sfu_lbd_pldata 16
lbd_sfu_advline
LBD Previous Line FIFO
plf_diurreq
plf_diurack
plf_diurdata 64
plf_diurvalid
plf_diuidle
sfu_lbd_rdy
plf_rdy
nlf_rdy
lbd_sfu_wdata 16
lbd_sfu_wdatavalid
LBD Next Line FIFO
nlf_diuwreq
nlf_diuwack
nlf_diuwdata 64
nlf_diuwvalid
lbd_dram_words
8
HCU
hcu_sfu_advdot
sfu_hcu_sdata 1
sfu_hcu_avail
HCU Read Line FIFO
hrf_hcu_endofline
hrf_xadvance
hrf_diurreq
hrf_diurack
hrf_diurdata 64
hrf_diurvalid
hrf_diuidle
SFU

FIG. 160

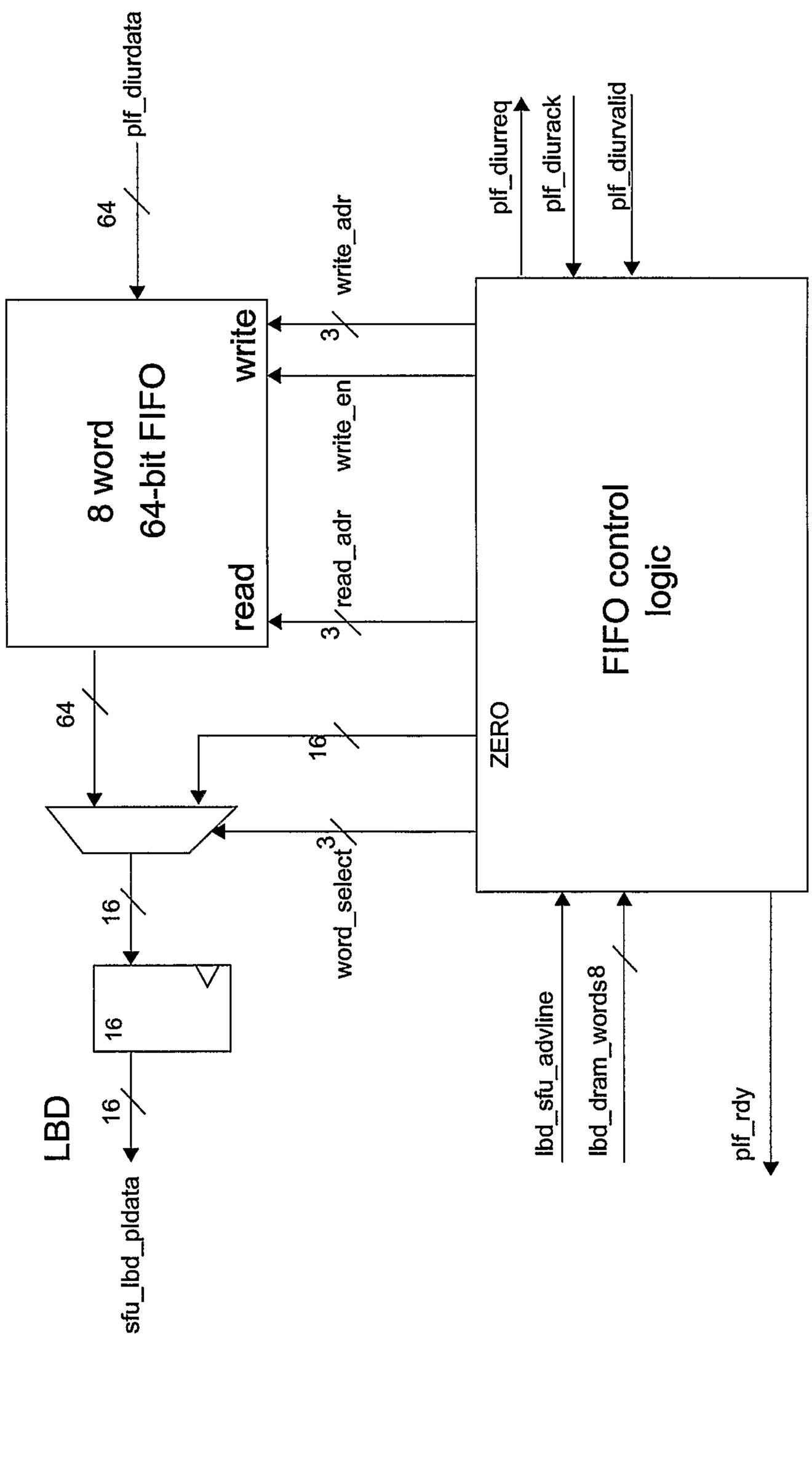

64
plf_diurdata
write_adr
plf_diurreq
plf_diurack
plf_diurvalid
8 word
64-bit FIFO
write
read
write_en
read_adr
FIFO control
logic
ZERO
64
16
word_select
16
16
LBD
16
sfu_lbd_pldata
lbd_sfu_advline
lbd_dram_words8
plf_rdy

FIG. 161

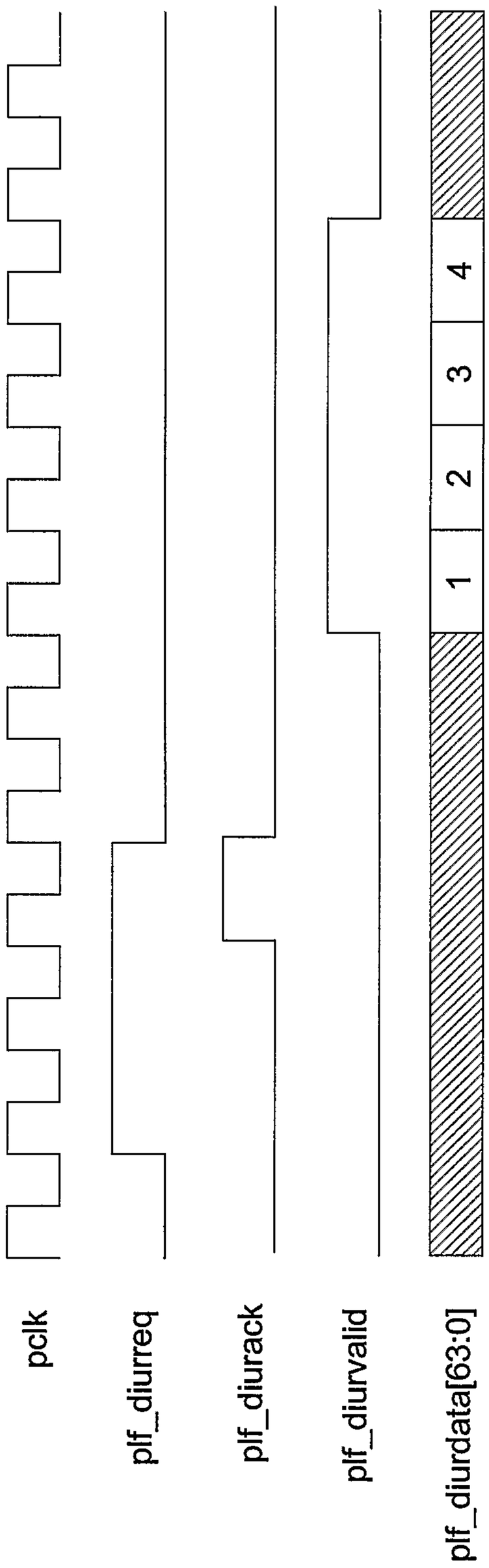

pclk
plf_diurreq
plf_diurack
plf_diurvalid
plf_diurdata[63:0]
4
3
2
1

FIG. 162

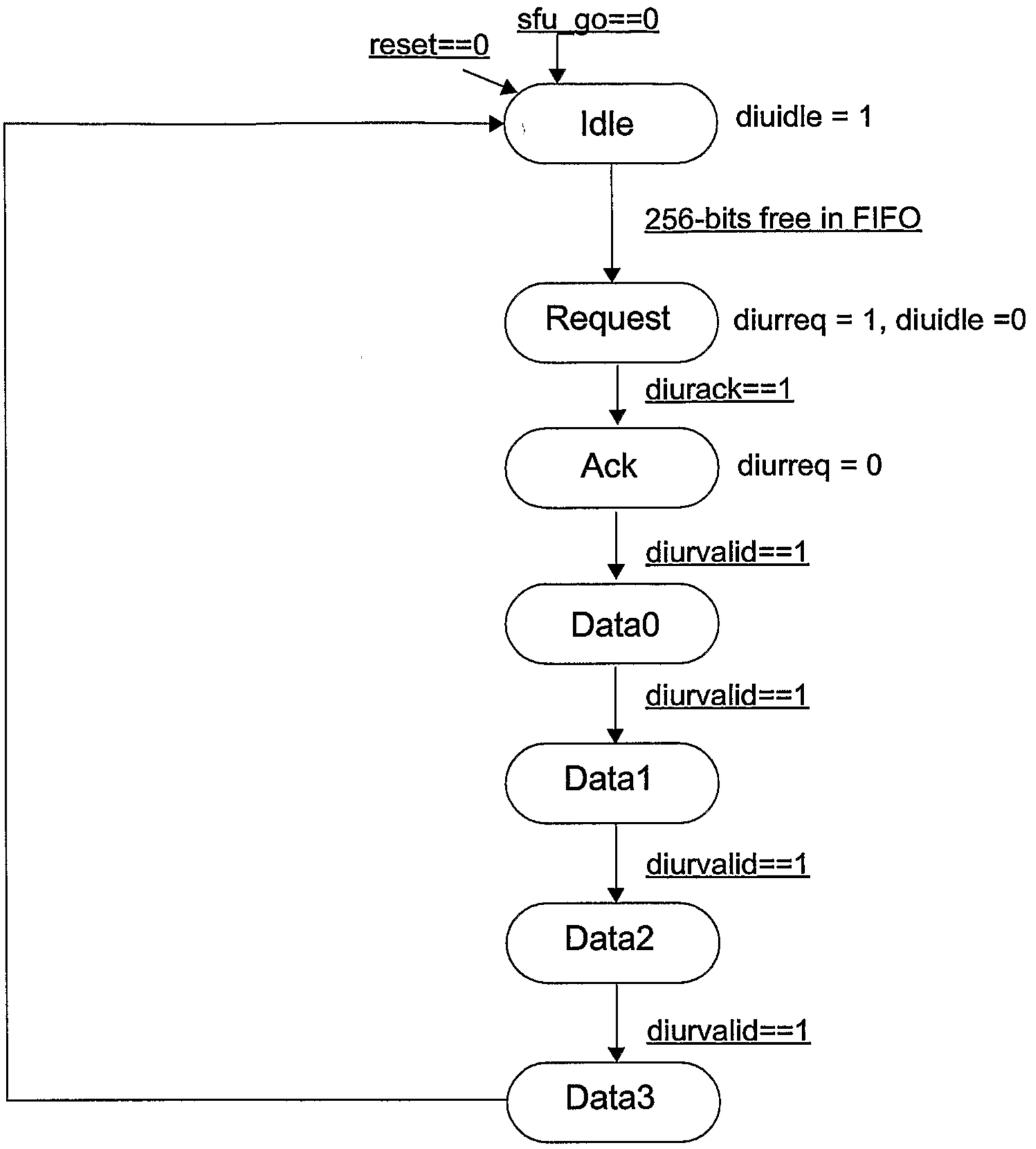

sfu_go==0
reset==0
Idle
diuidle = 1
256-bits free in FIFO
Request
diurreq = 1, diuidle =0
diurack==1
Ack
diurreq = 0
diurvalid==1
Data0
diurvalid==1
Data1
diurvalid==1
Data2
diurvalid==1
Data3

FIG. 163

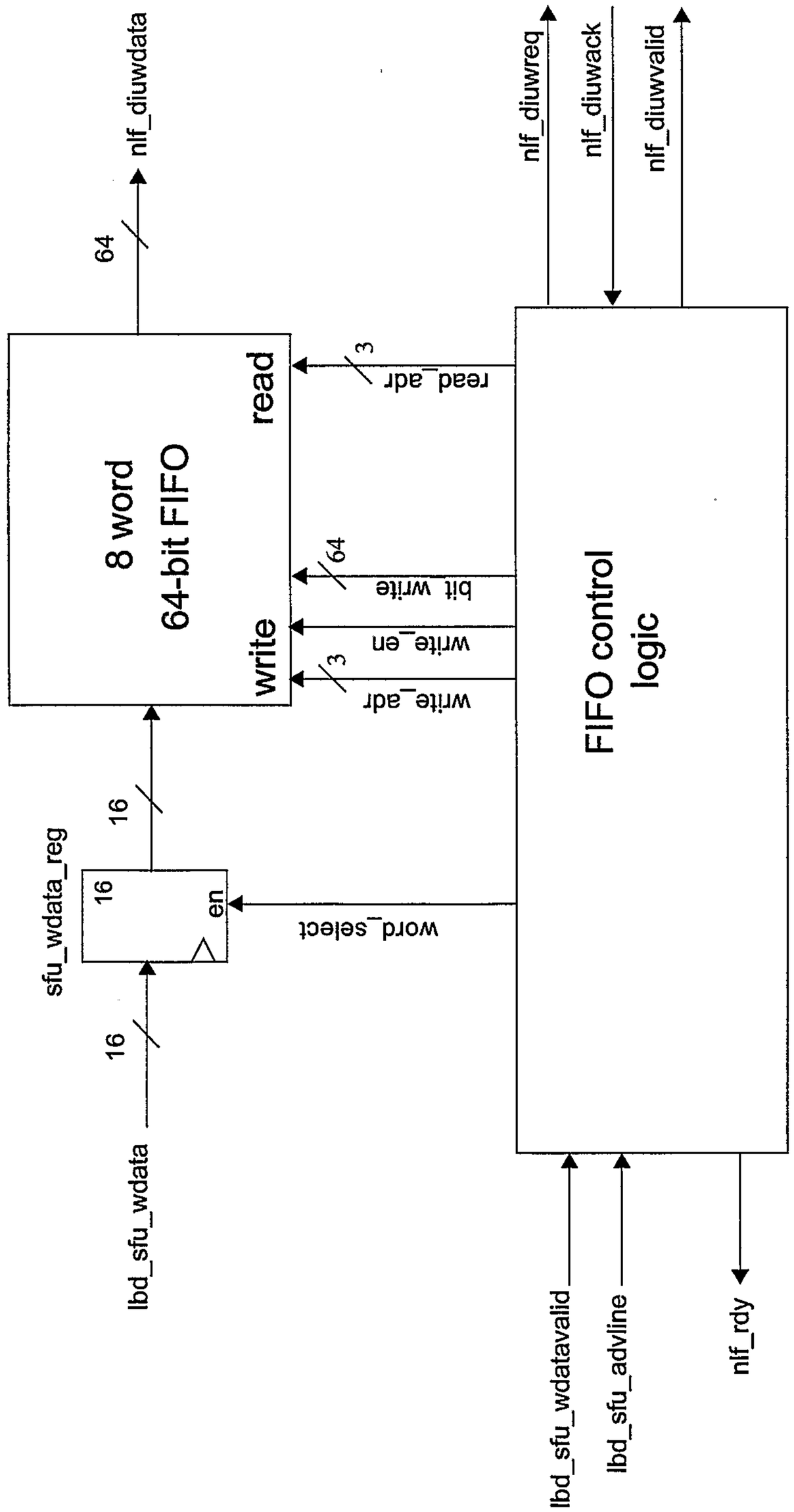

lbd_sfu_wdata
16
sfu_wdata_reg
16
en
16
8 word
64-bit FIFO
write
read
64
nlf_diuwdata
3
read_adr
nlf_diuwreq
nlf_diuwack
nlf_diuwvalid
64
bit_write
write_en
3
write_adr
word_select
FIFO control
logic
lbd_sfu_wdatavalid
lbd_sfu_advline
nlf_rdy

FIG. 164

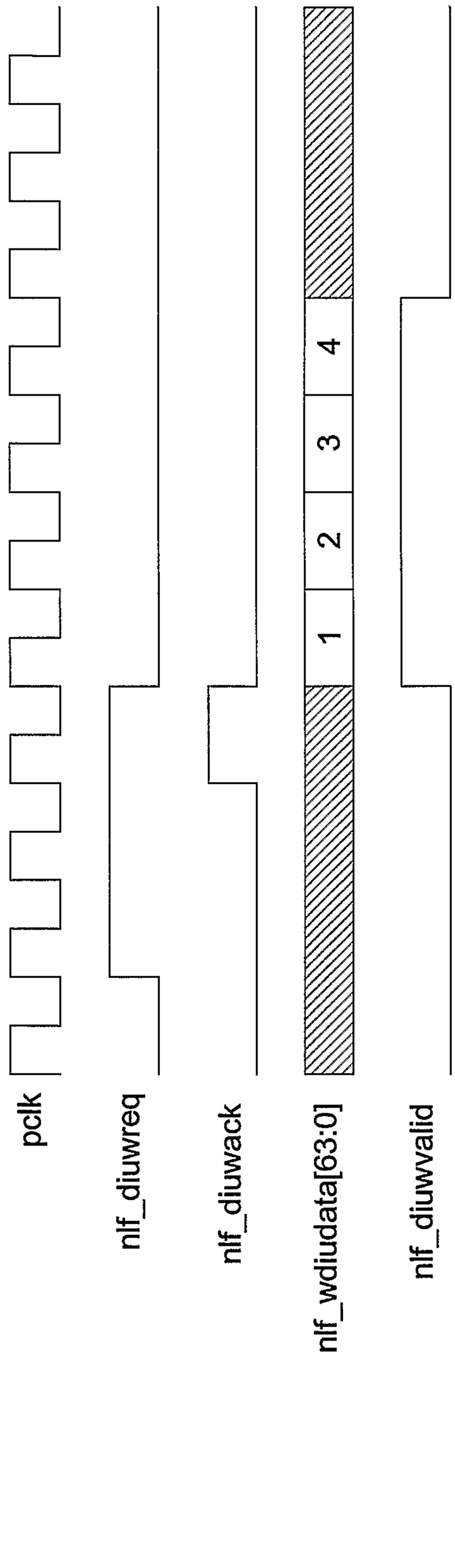

4
3
2
1
pclk
nlf_diuwreq
nlf_diuwack
nlf_wdiudata[63:0]
nlf_diuwvalid

FIG. 165

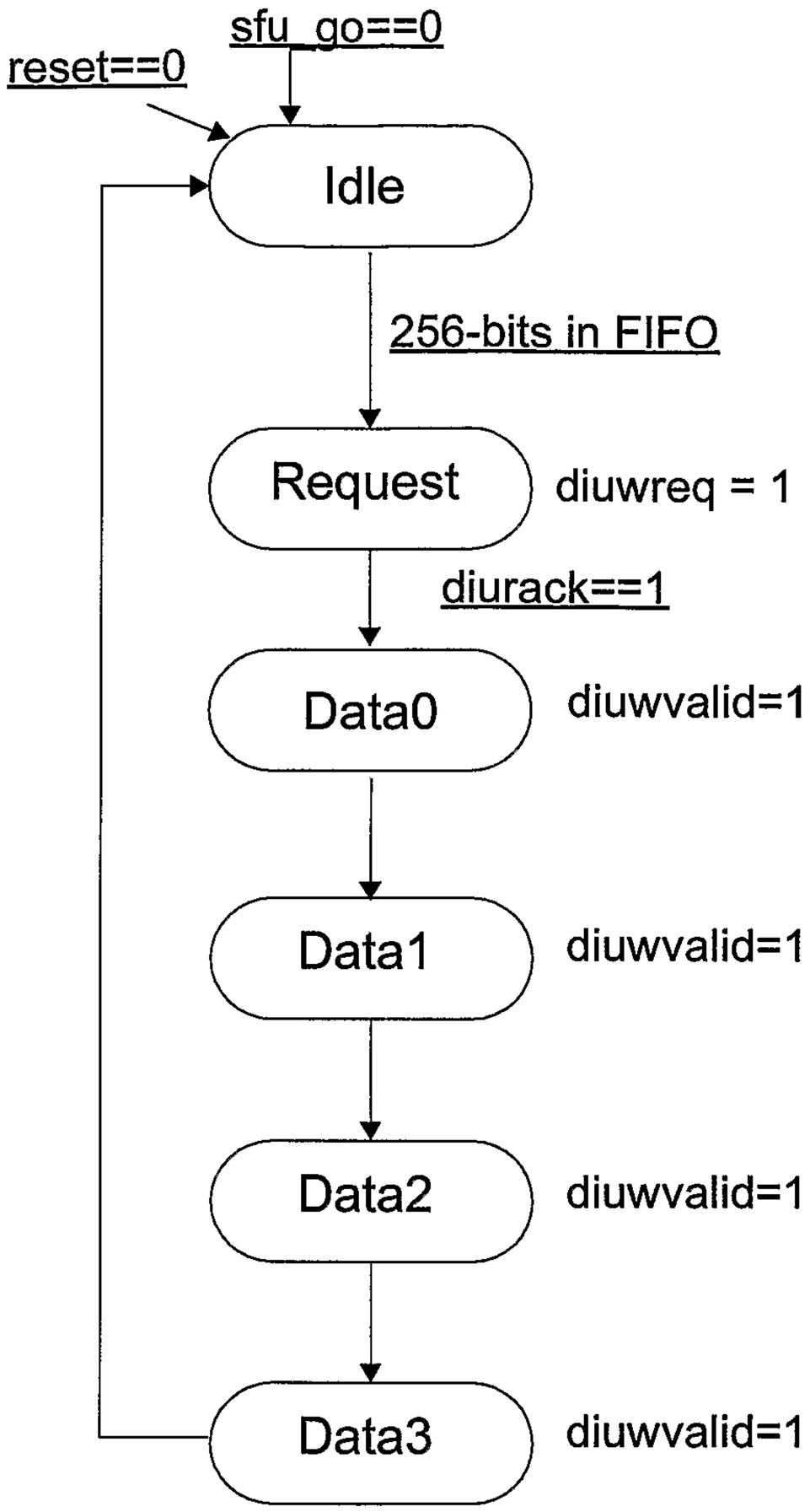

sfu_go==0
reset==0
Idle
256-bits in FIFO
Request
diuwreq = 1
diurack==1
Data0
diuwvalid=1
Data1
diuwvalid=1
Data2
diuwvalid=1
Data3
diuwvalid=1

FIG. 166

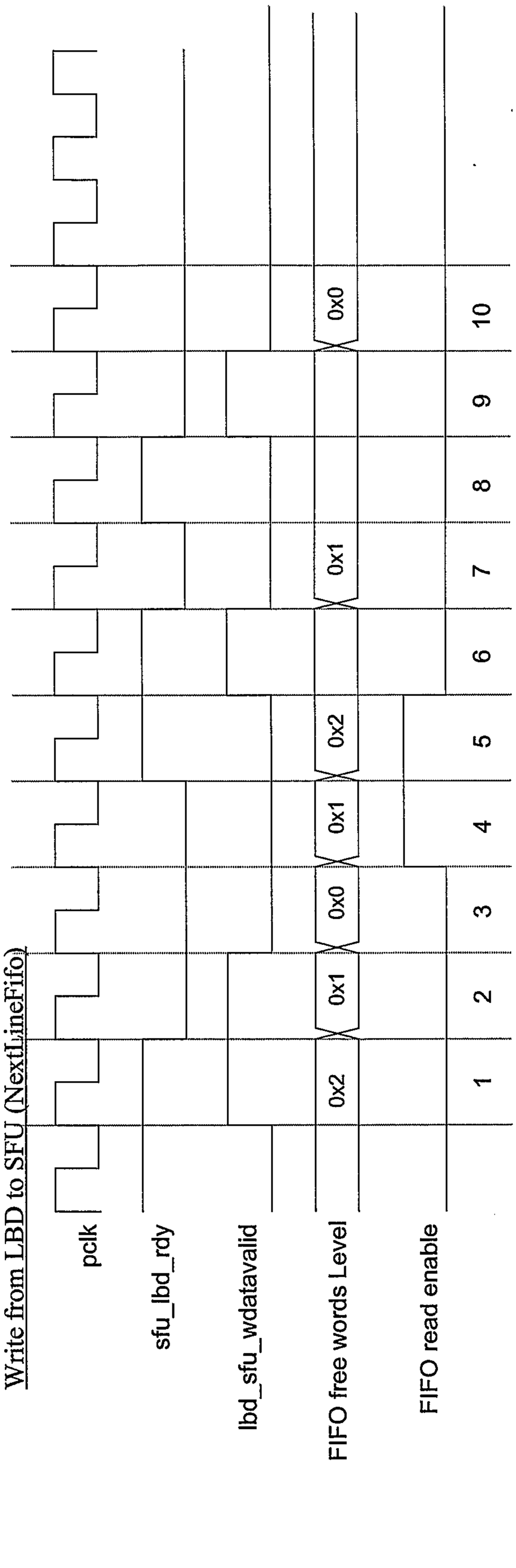

Write from LBD to SFU (NextLineFifo)
pclk
sfu_lbd_rdy
lbd_sfu_wdatavalid
FIFO free words Level
FIFO read enable
0x2
0x1
0x0
0x1
0x2
0x1
0x0
0x1
0x2
1
2
3
4
5
6
7
8
9
10

FIG. 167

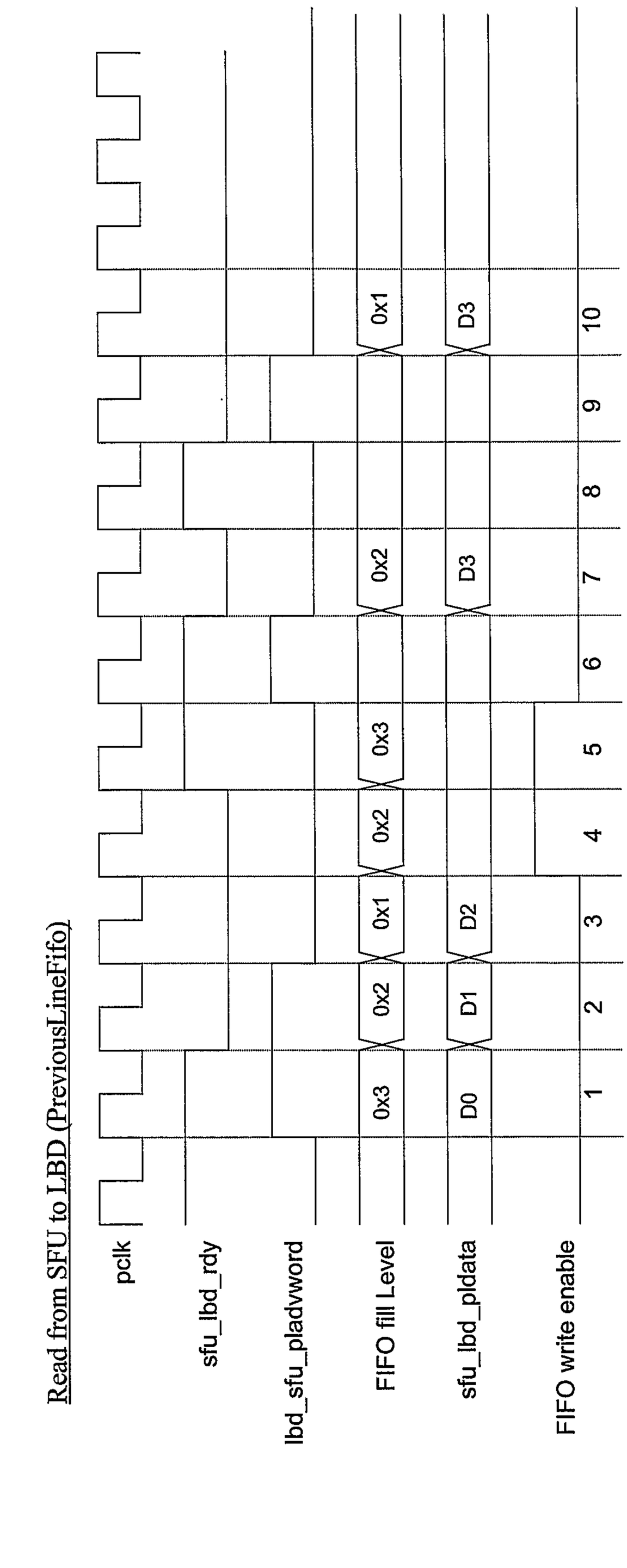

Read from SFU to LBD (PreviousLineFifo)
pclk
sfu_lbd_rdy
lbd_sfu_pladvword
FIFO fill Level
sfu_lbd_pldata
FIFO write enable
0x3
0x2
0x1
0x2
0x3
0x2
0x1
D0
D1
D2
D3
D3
1
2
3
4
5
6
7
8
9
10

FIG. 168

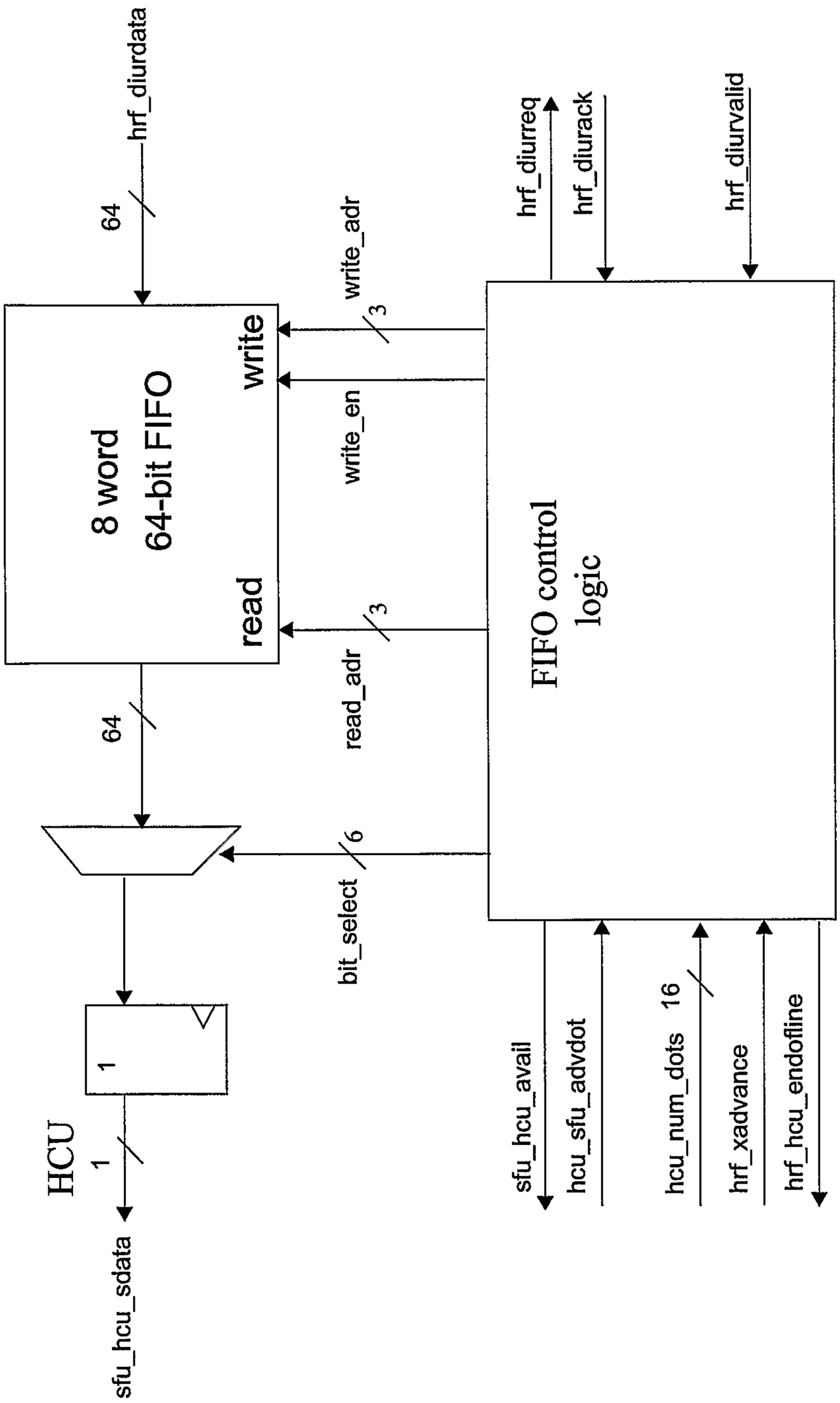

hrf_diurdata
64
write_adr
3
write_en
write
read
8 word
64-bit FIFO
hrf_diurreq
hrf_diurack
hrf_diurvalid
FIFO control
logic
64
read_adr
3
bit_select
6
HCU
1
1
sfu_hcu_sdata
sfu_hcu_avail
hcu_sfu_advdot
hcu_num_dots
16
hrf_xadvance
hrf_hcu_endofline

FIG. 169

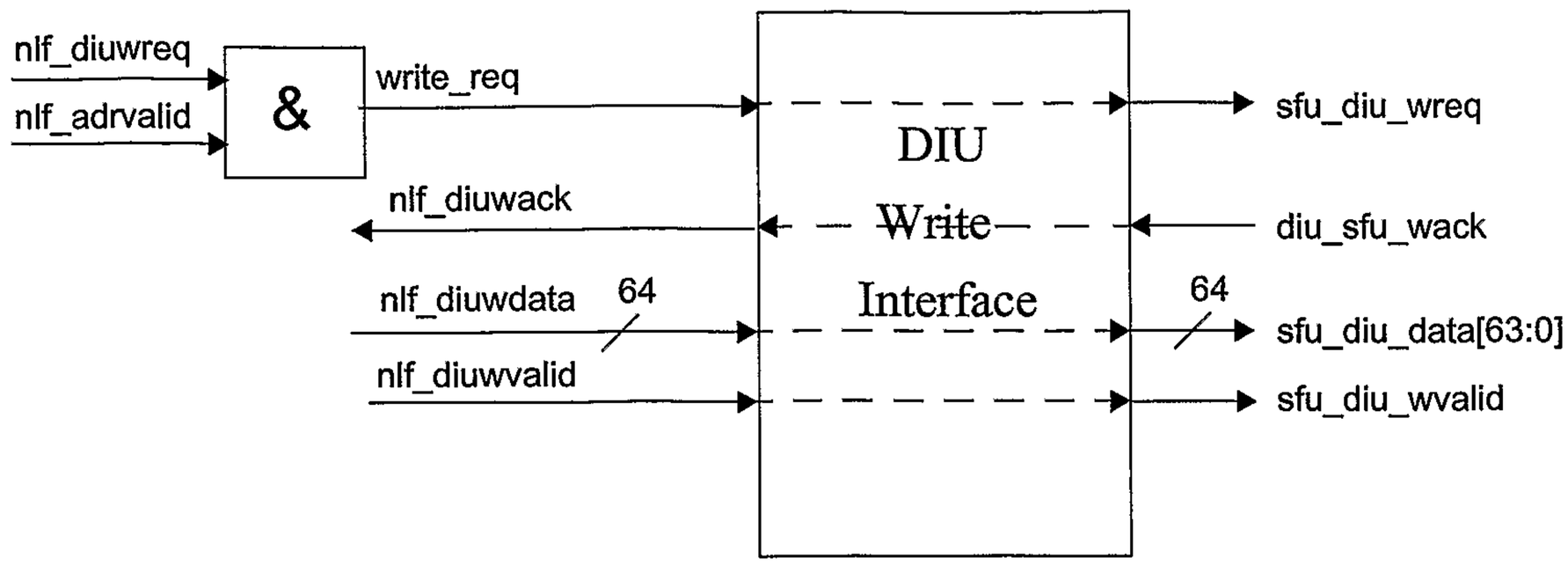


FIG. 170

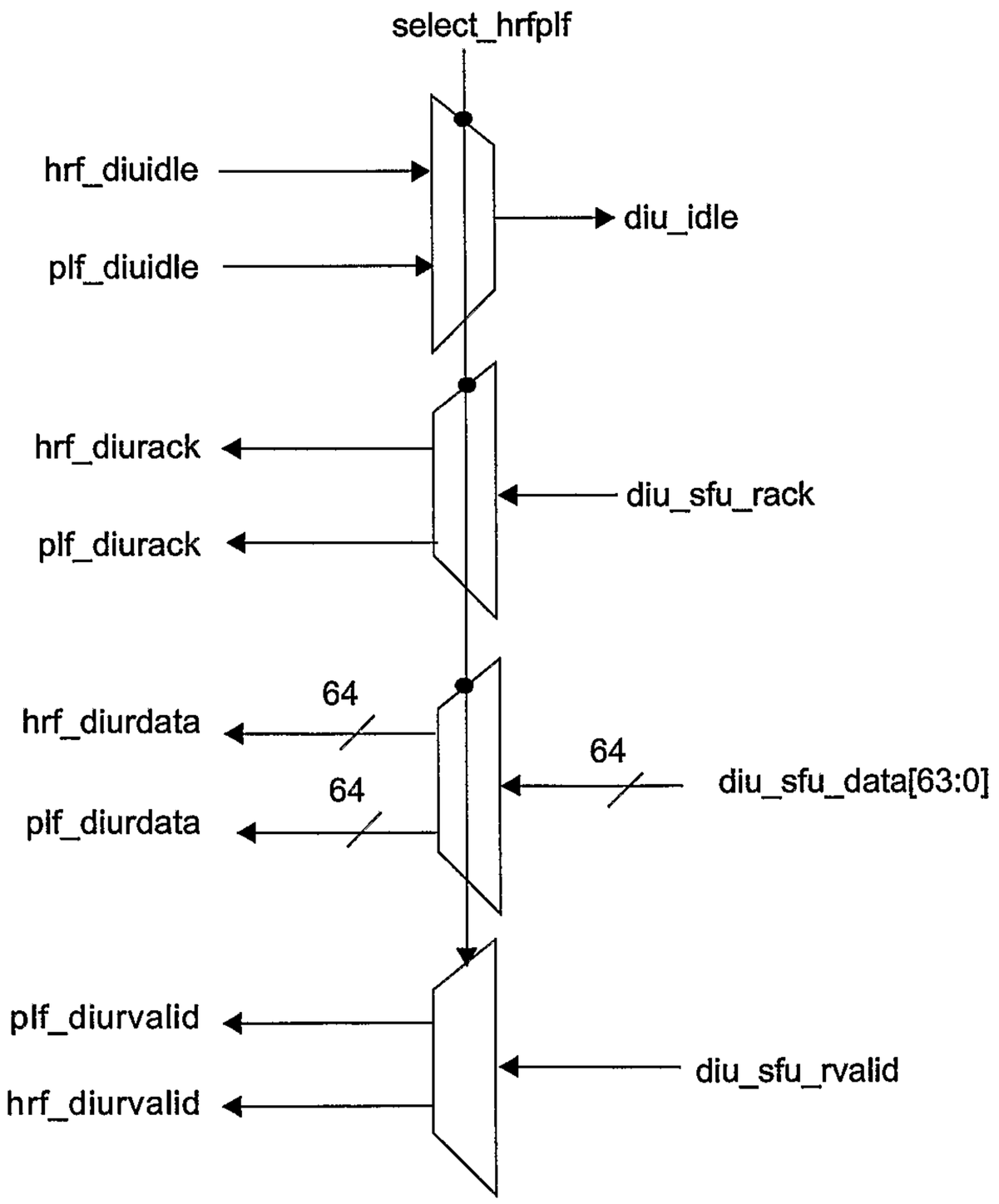


FIG. 171

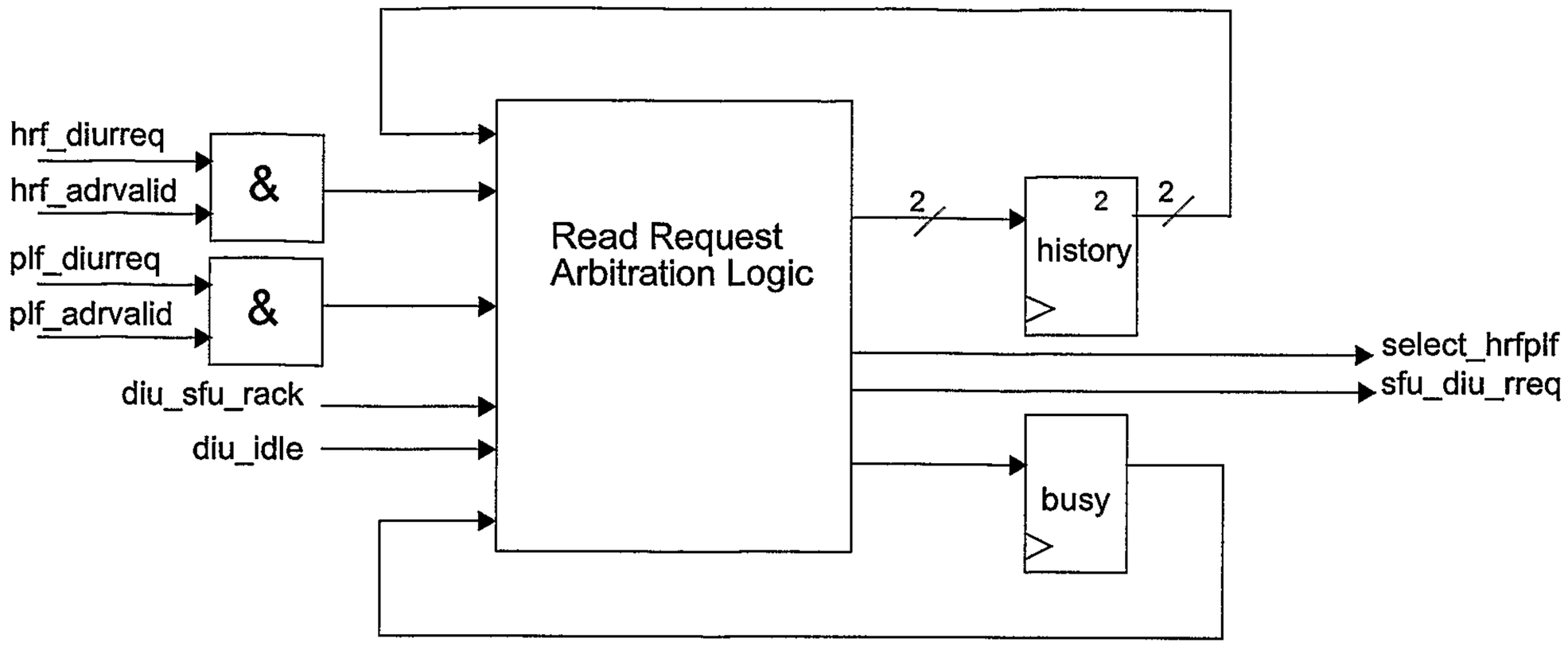


FIG. 172

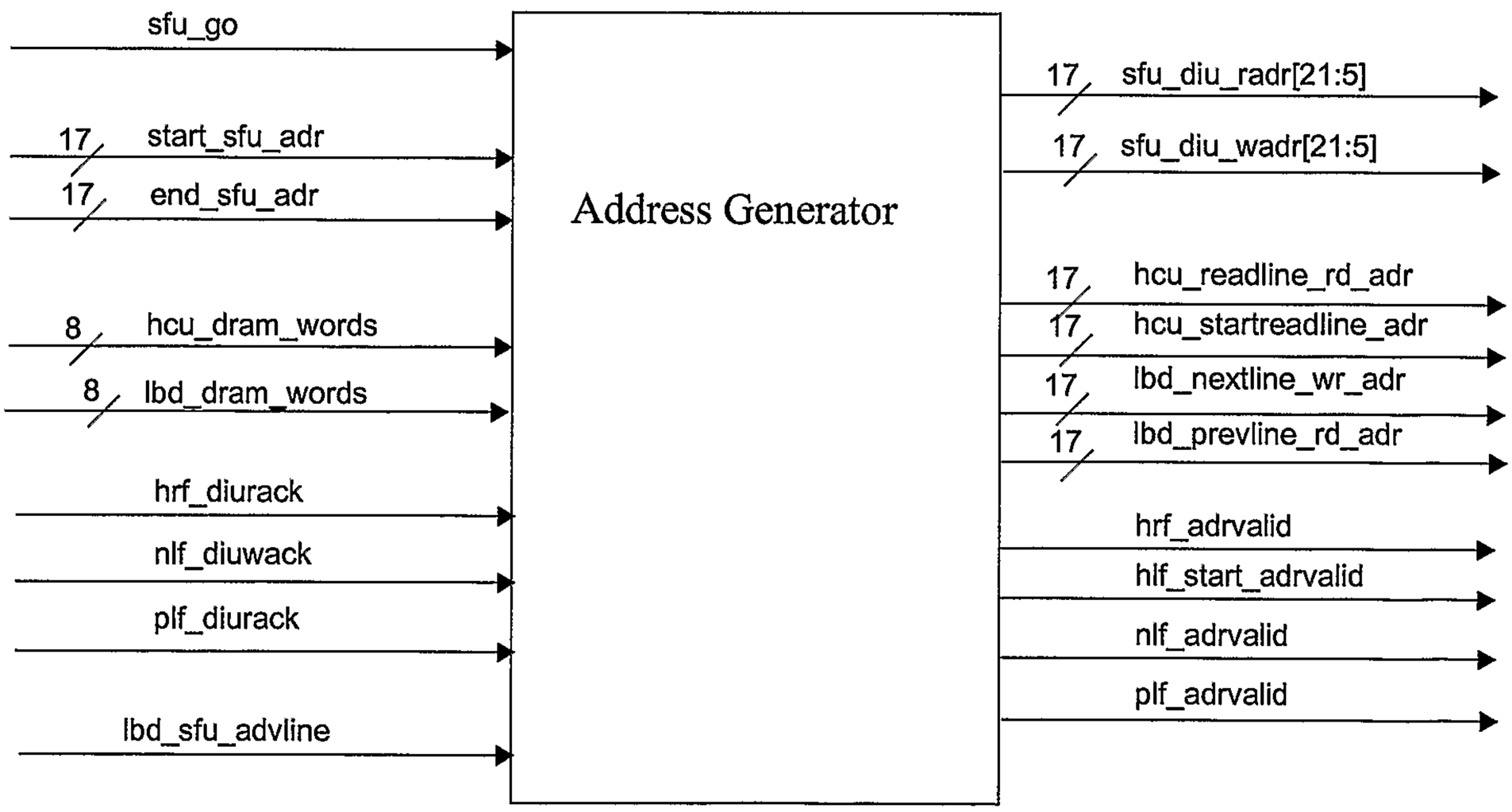


FIG. 173

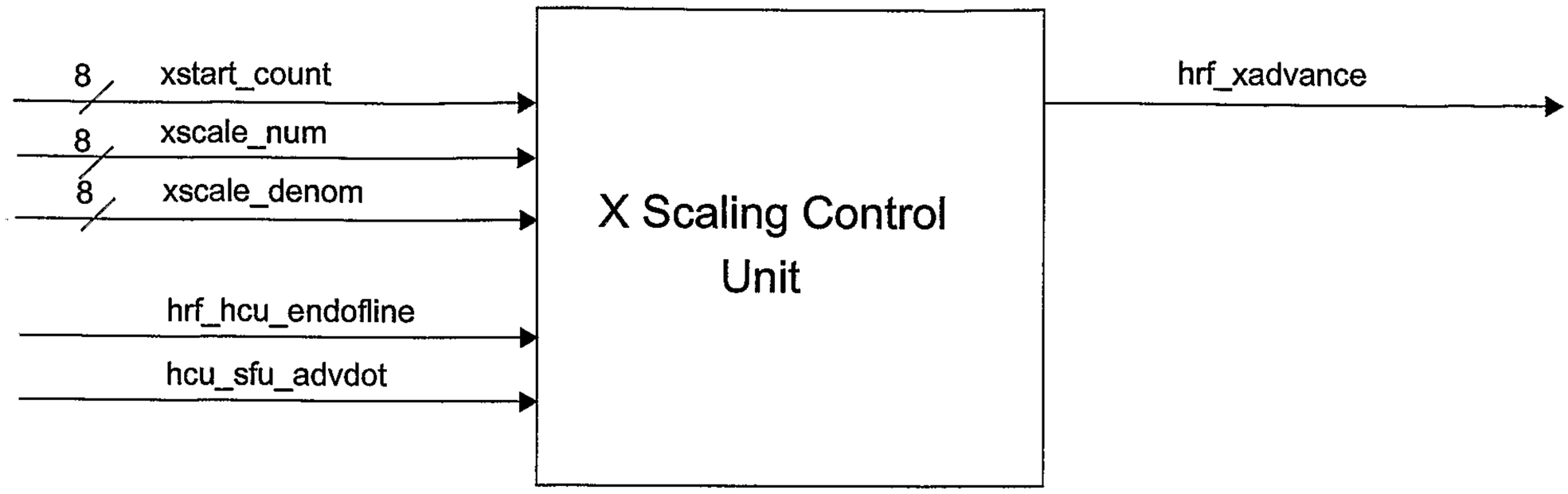


FIG. 174

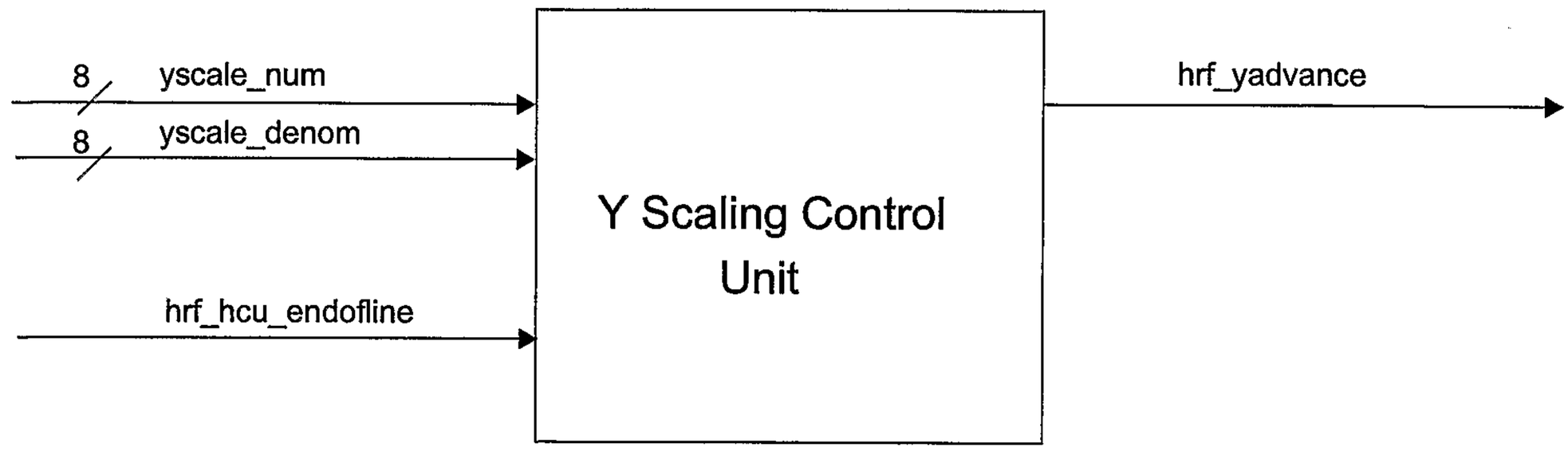


FIG. 175

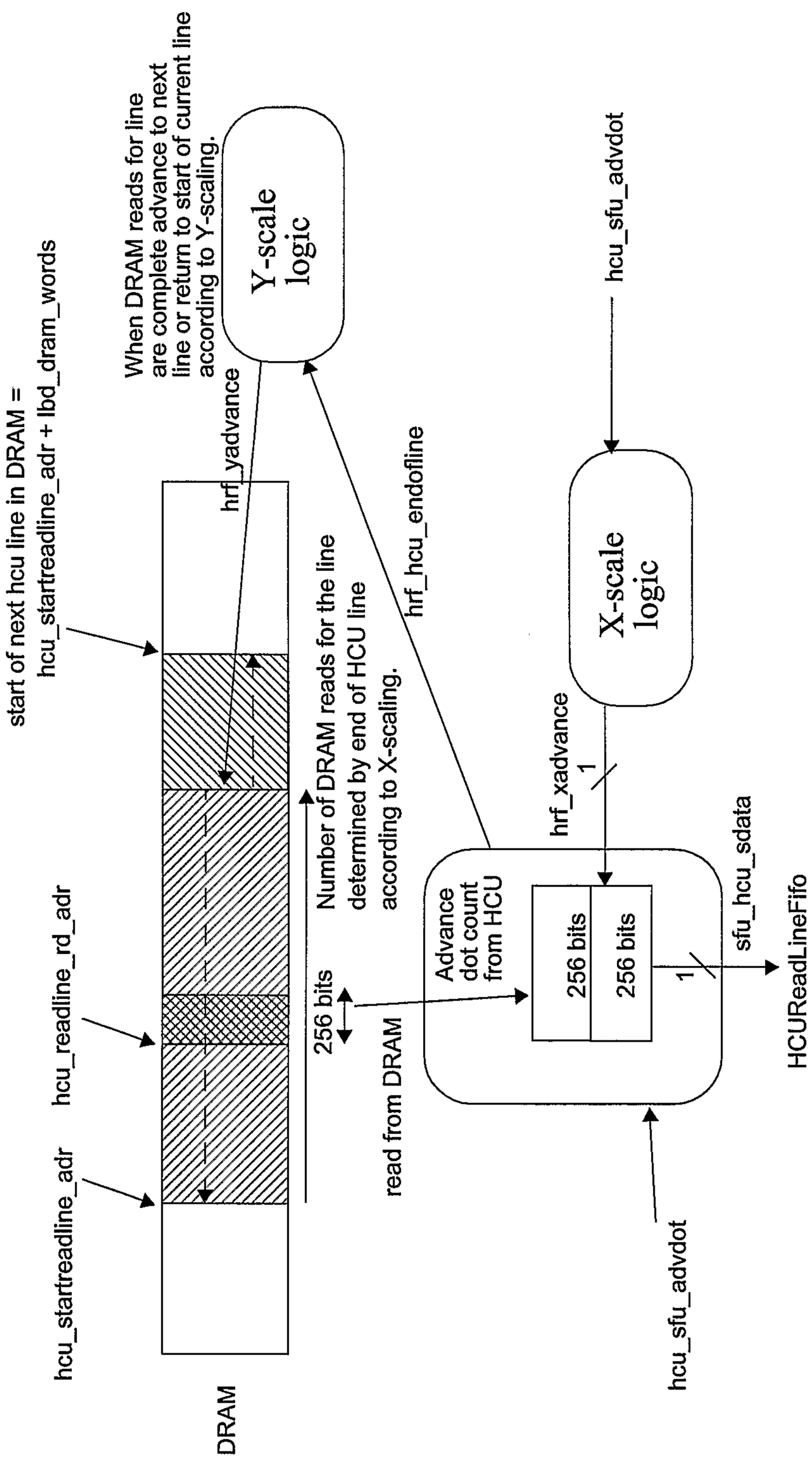

start of next hcu line in DRAM =
hcu_startreadline_adr + lbd_dram_words
hcu_startreadline_adr
hcu_readline_rd_adr
DRAM
When DRAM reads for line are complete advance to next line or return to start of current line according to Y-scaling.
Y-scale logic
hrf_yadvance
hrf_hcu_endofline
Number of DRAM reads for the line determined by end of HCU line according to X-scaling.
256 bits
read from DRAM
Advance dot count from HCU
256 bits
256 bits
hrf_xadvance
1
X-scale logic
hcu_sfu_advdot
sfu_hcu_sdata
1
HCUReadLineFifo
hcu_sfu_advdot

FIG. 176

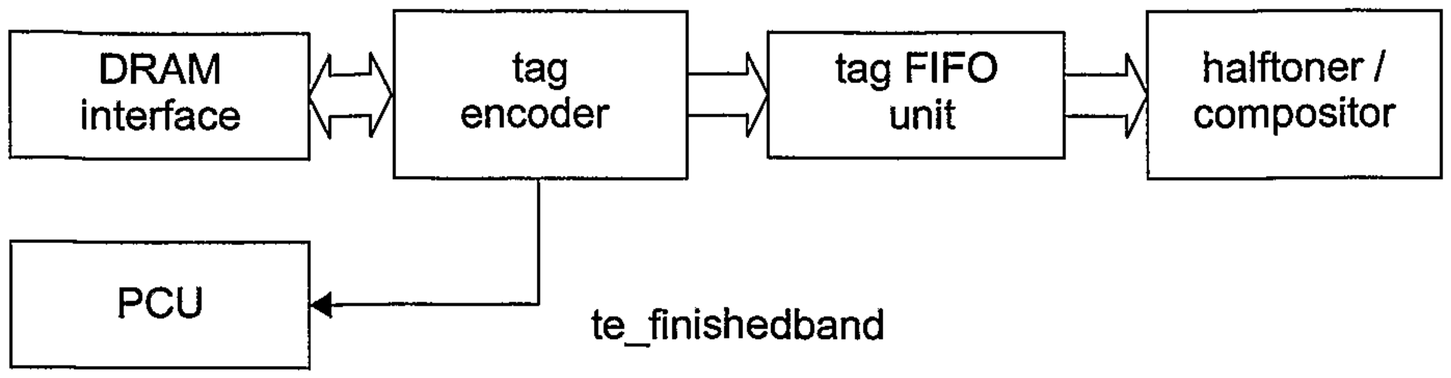


FIG. 177

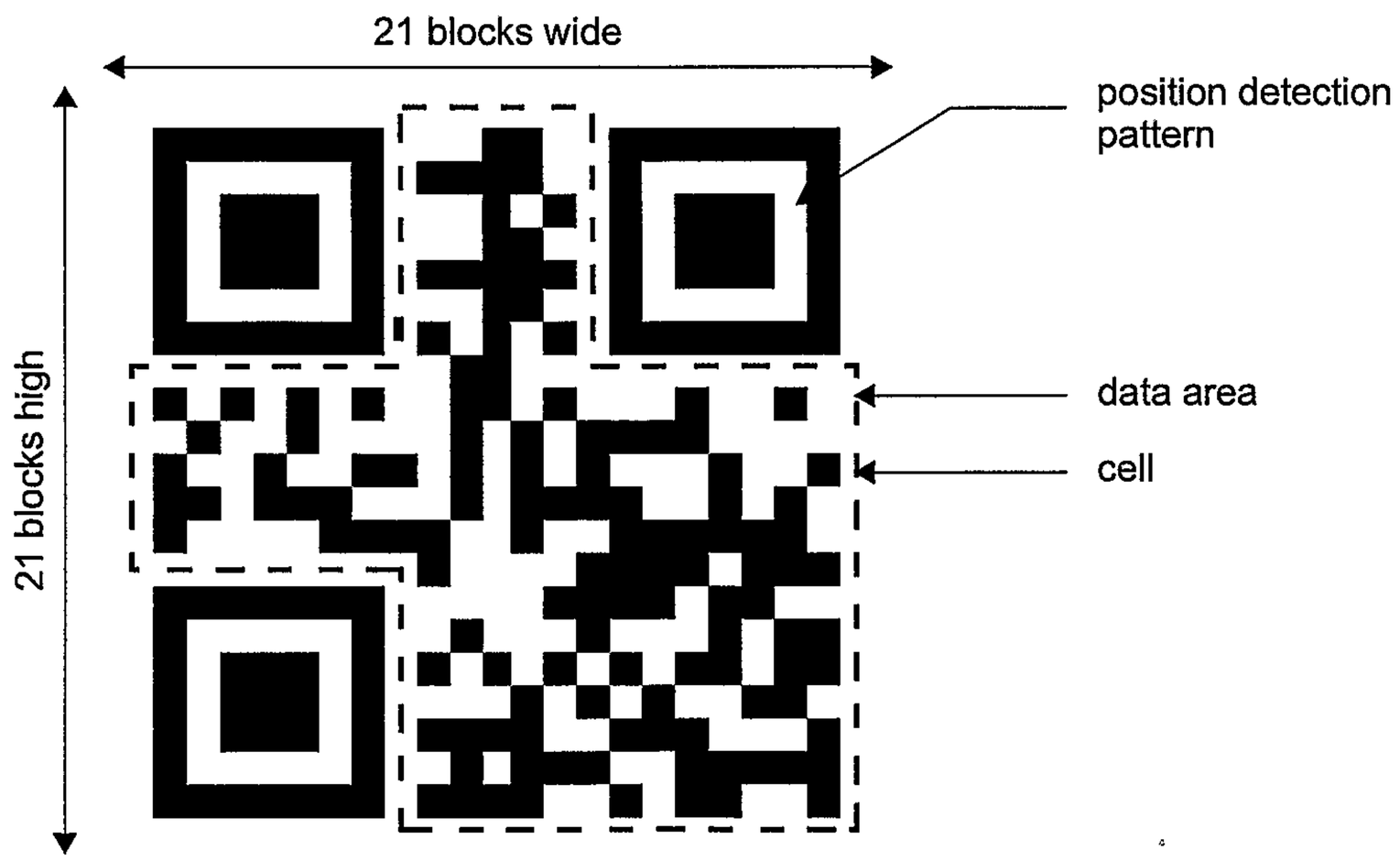
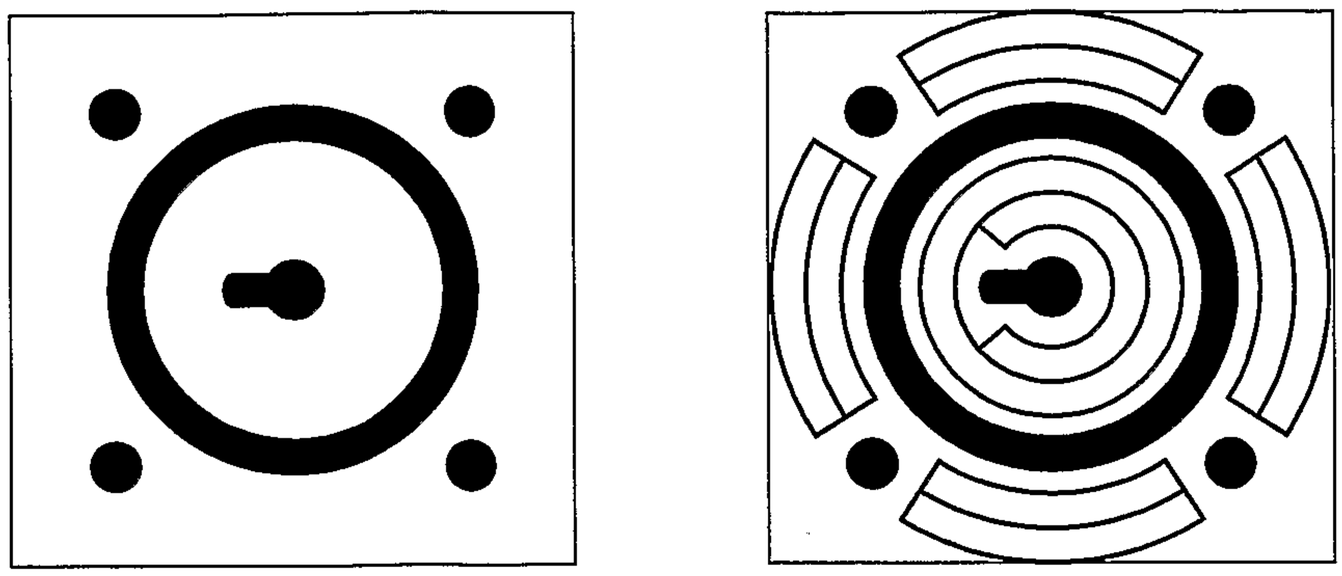


FIG. 178



(a) Netpage tag background pattern

(b) Netpage tag showing data area

FIG. 179

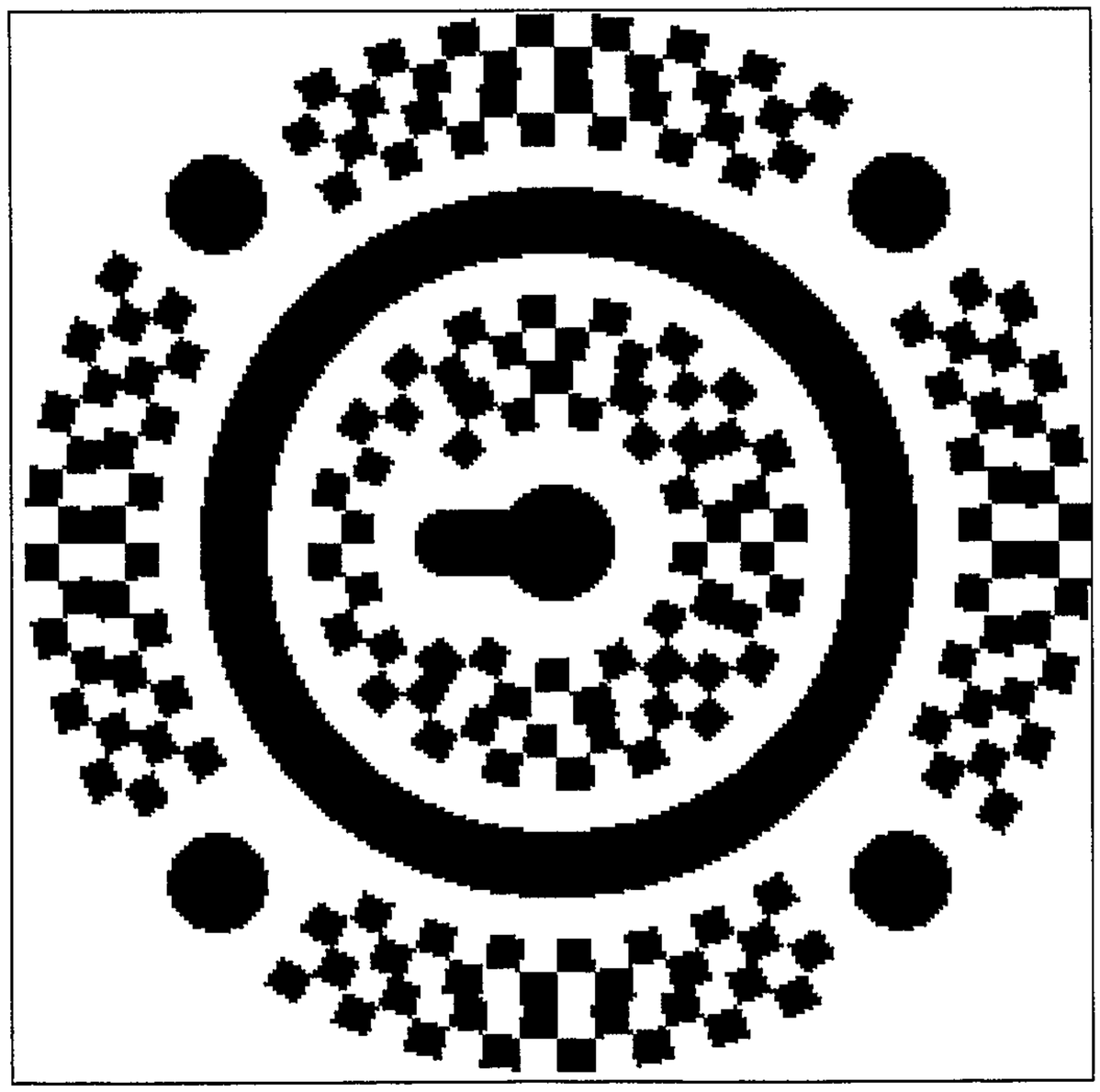


FIG. 180

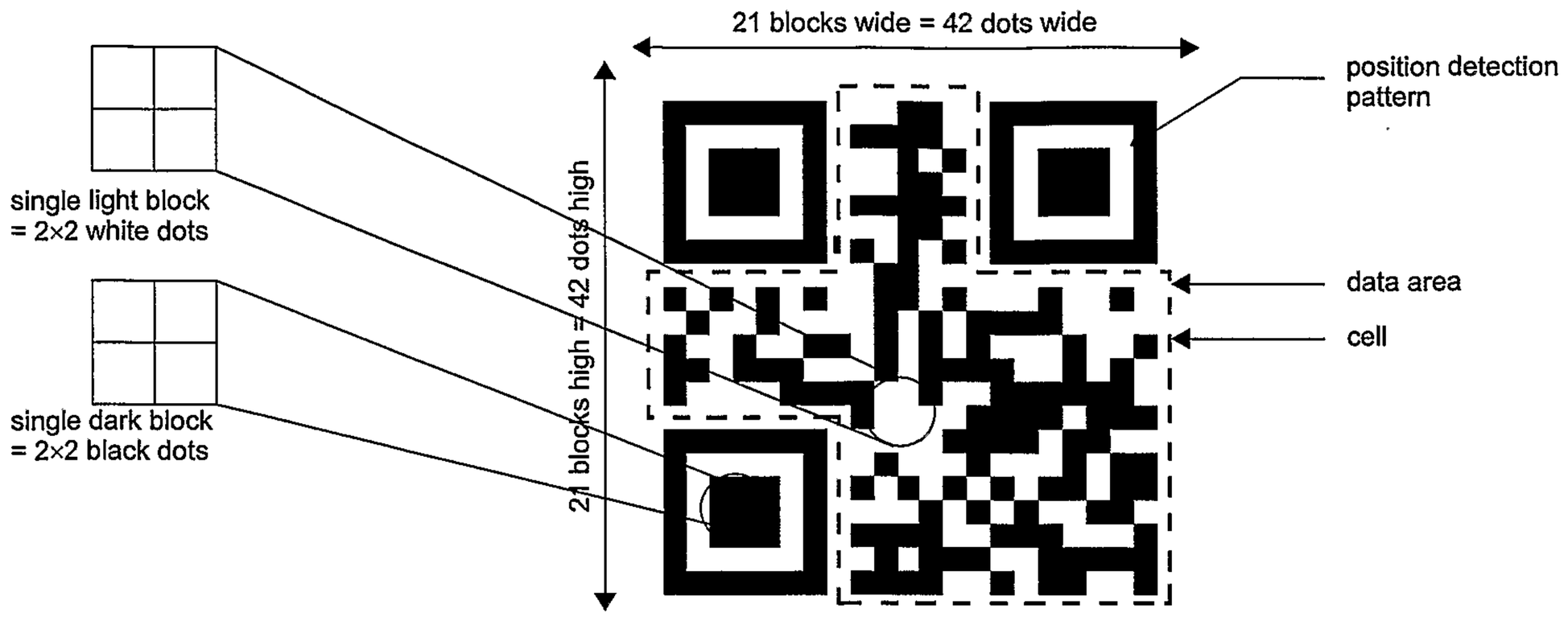


FIG. 181

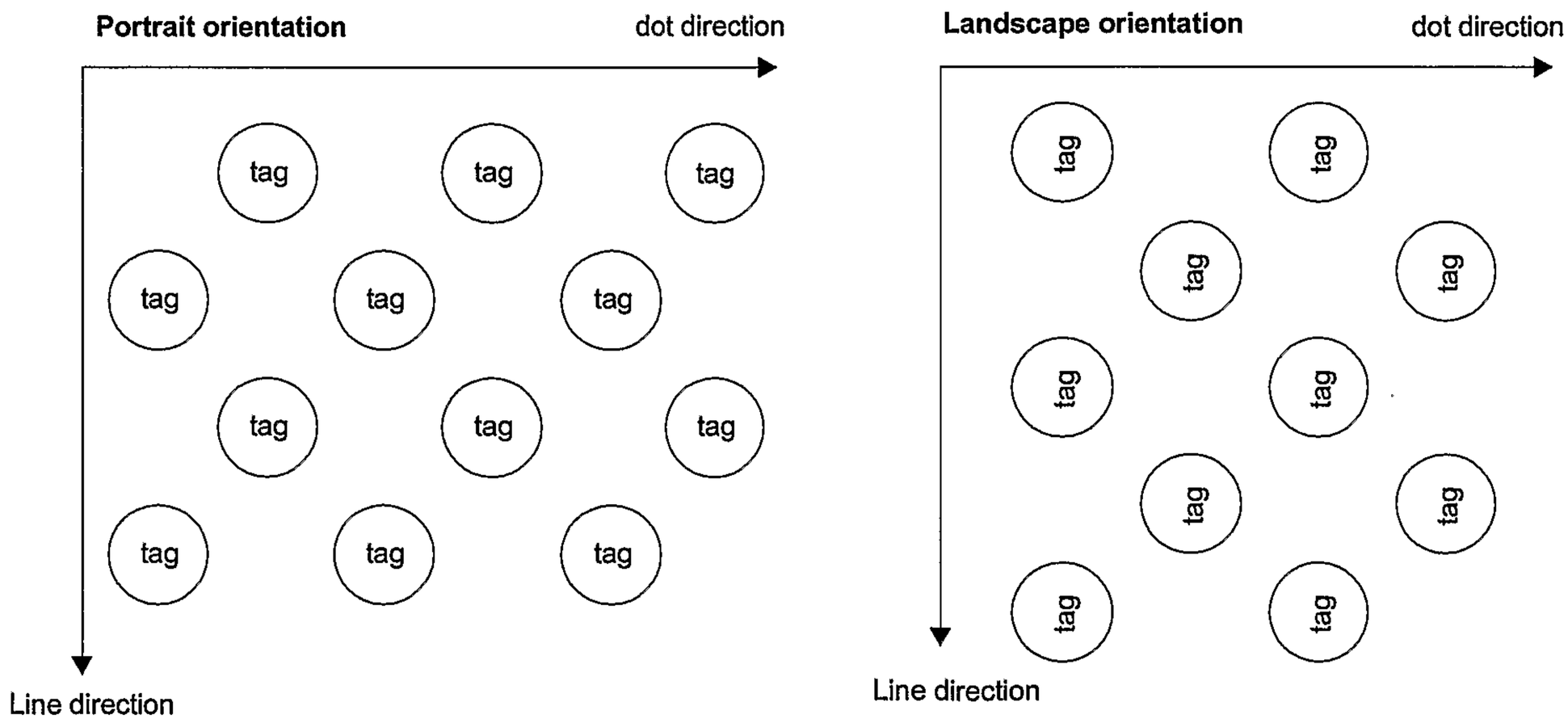


FIG. 182

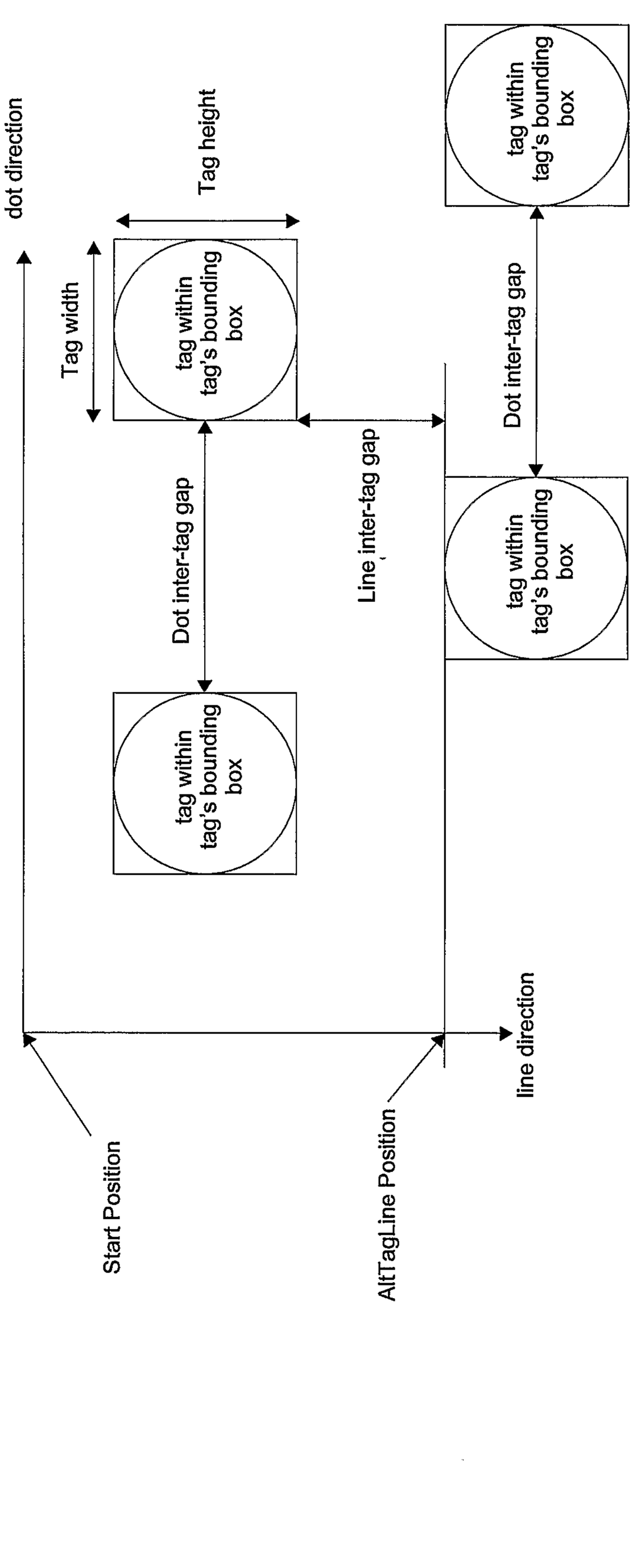

dot direction
Tag height
Tag width
tag within tag's bounding box
Dot inter-tag gap
Line inter-tag gap
Dot inter-tag gap
tag within tag's bounding box
tag within tag's bounding box
tag within tag's bounding box
line direction
Start Position
AltTagLine Position

FIG. 183

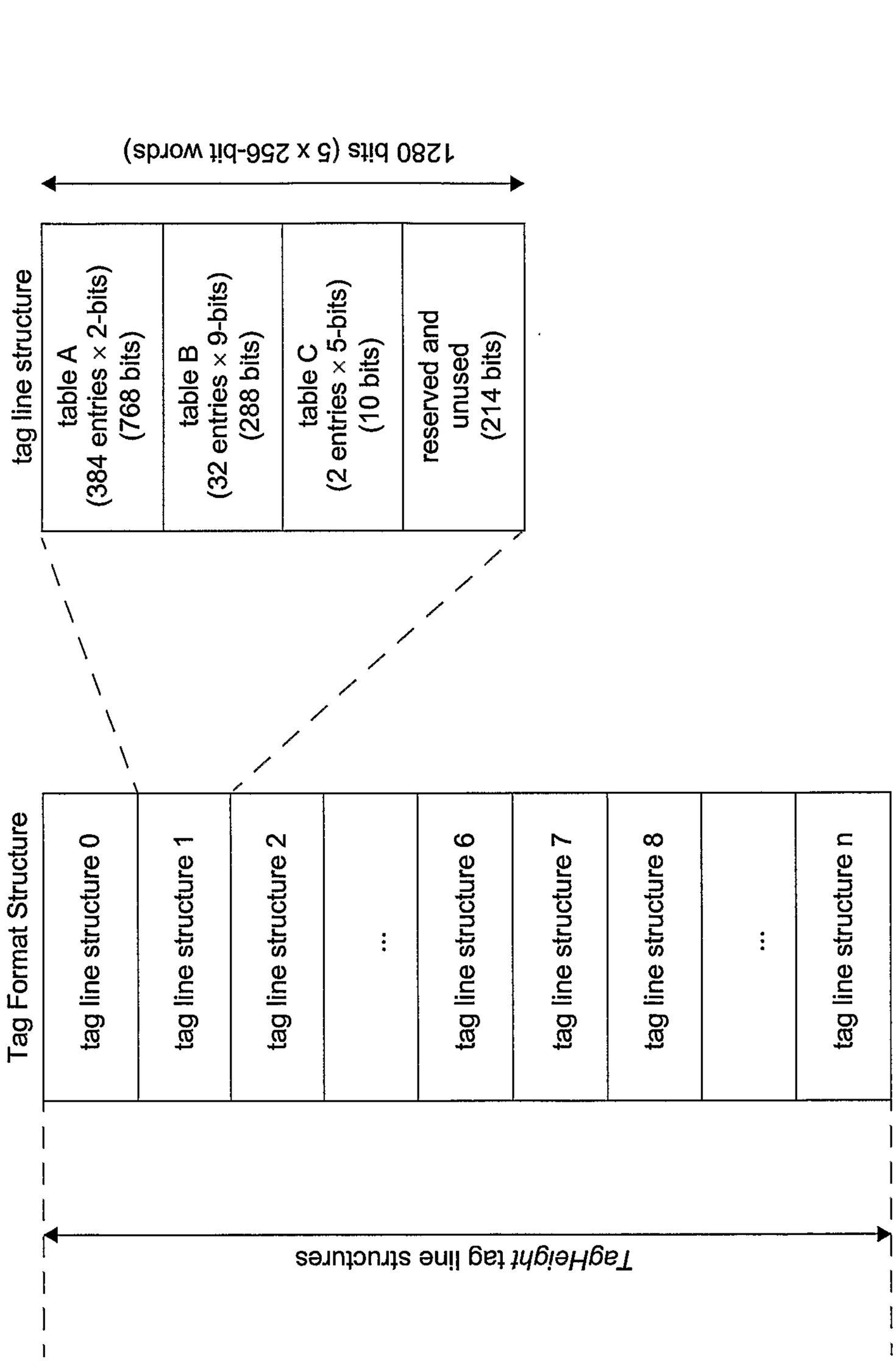

Tag Format Structure
tag line structure 0
tag line structure 1
tag line structure 2
...
tag line structure 6
tag line structure 7
tag line structure 8
...
tag line structure n
TagHeight tag line structures
tag line structure
table A
(384 entries × 2-bits)
(768 bits)
table B
(32 entries × 9-bits)
(288 bits)
table C
(2 entries × 5-bits)
(10 bits)
reserved and
unused
(214 bits)
1280 bits (5 x 256-bit words)

FIG. 184

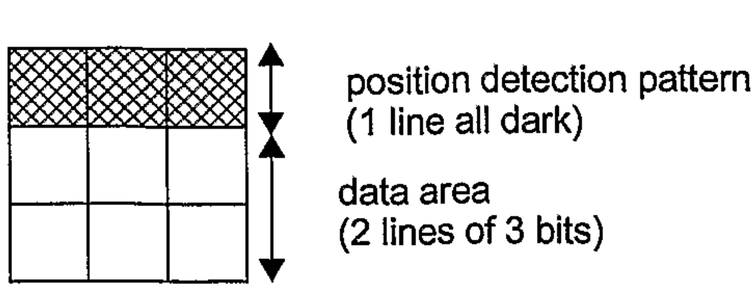


FIG. 185

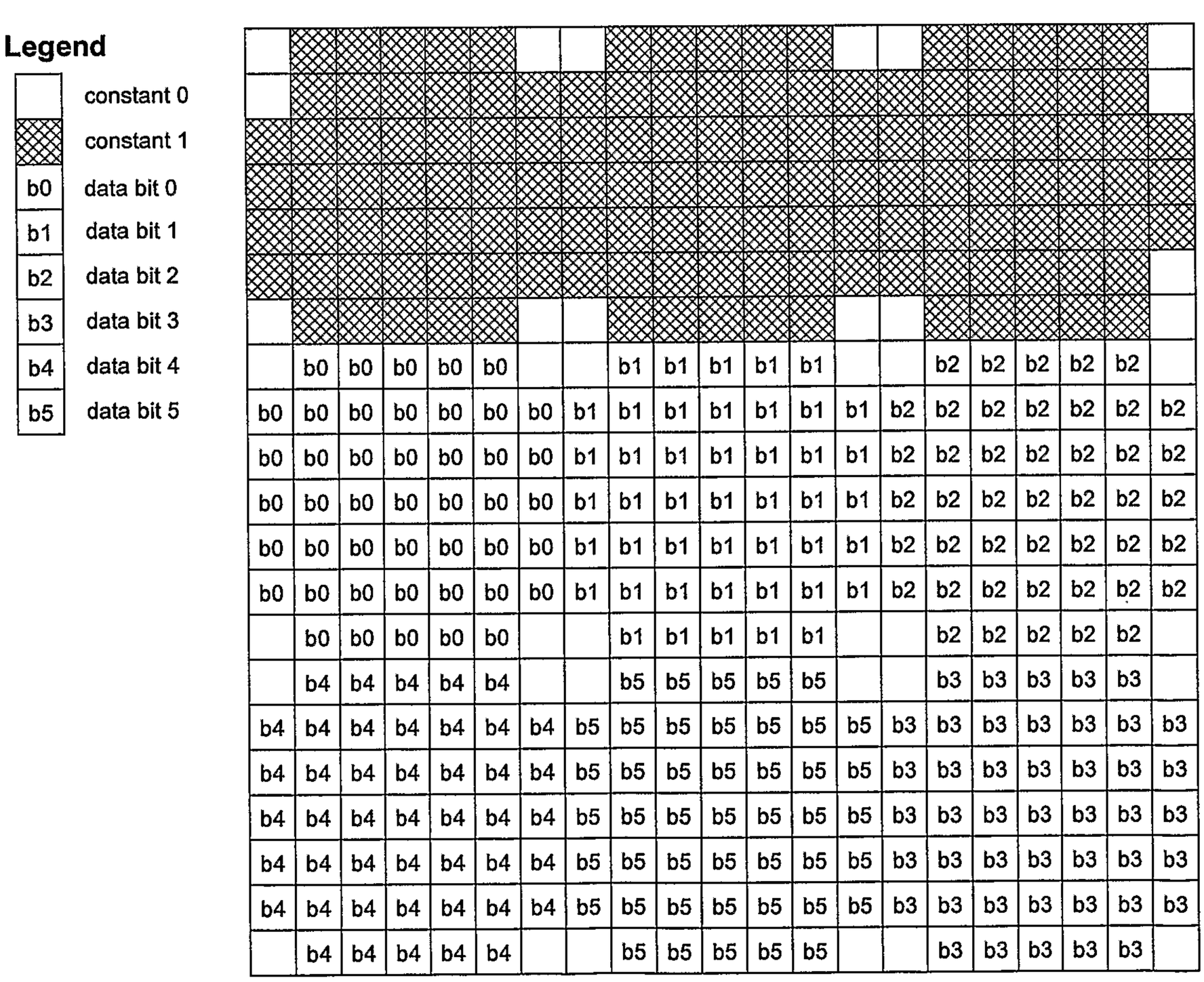


FIG. 186

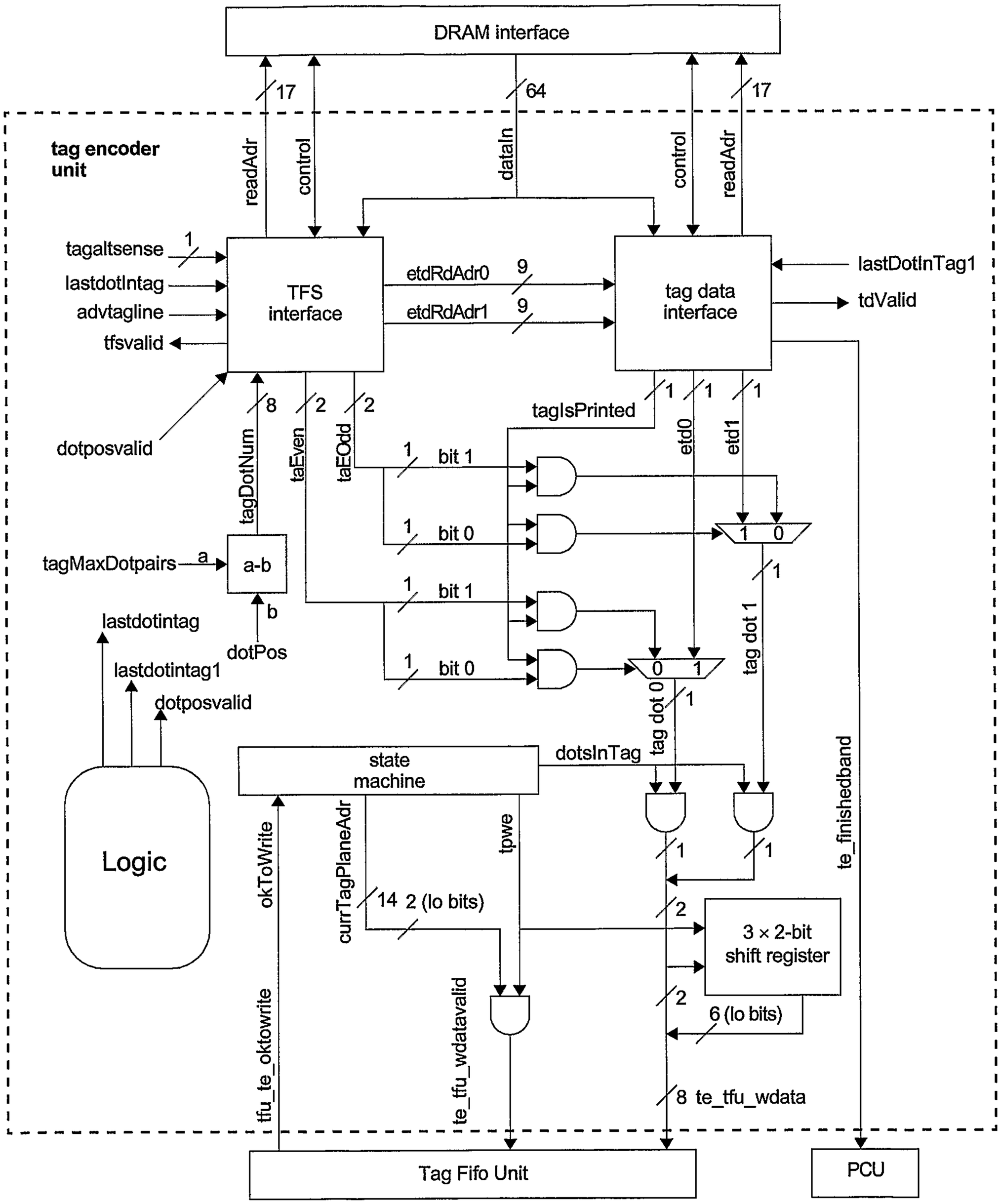

DRAM interface
17
64
17
tag encoder unit
readAdr
control
dataIn
control
readAdr
tagaltsense
lastdotIntag
advtagline
tfsvalid
TFS interface
etdRdAdr0
etdRdAdr1
9
9
tag data interface
lastDotInTag1
tdValid
dotposvalid
8
2
2
tagDotNum
taEven
taEOdd
tagIsPrinted
1
1
1
etd0
etd1
1 bit 1
1 bit 0
1 bit 1
1 bit 0
tagMaxDotpairs
a
a-b
b
dotPos
lastdotintag
lastdotintag1
dotposvalid
1 0
1
tag dot 1
0 1
1
tag dot 0
Logic
state machine
dotsInTag
okToWrite
currTagPlaneAdr
14
2 (lo bits)
tpwe
1
1
2
3 × 2-bit shift register
2
6 (lo bits)
te_finishedband
tfu_te_oktowrite
te_tfu_wdatavalid
8 te_tfu_wdata
Tag Fifo Unit
PCU

FIG. 187

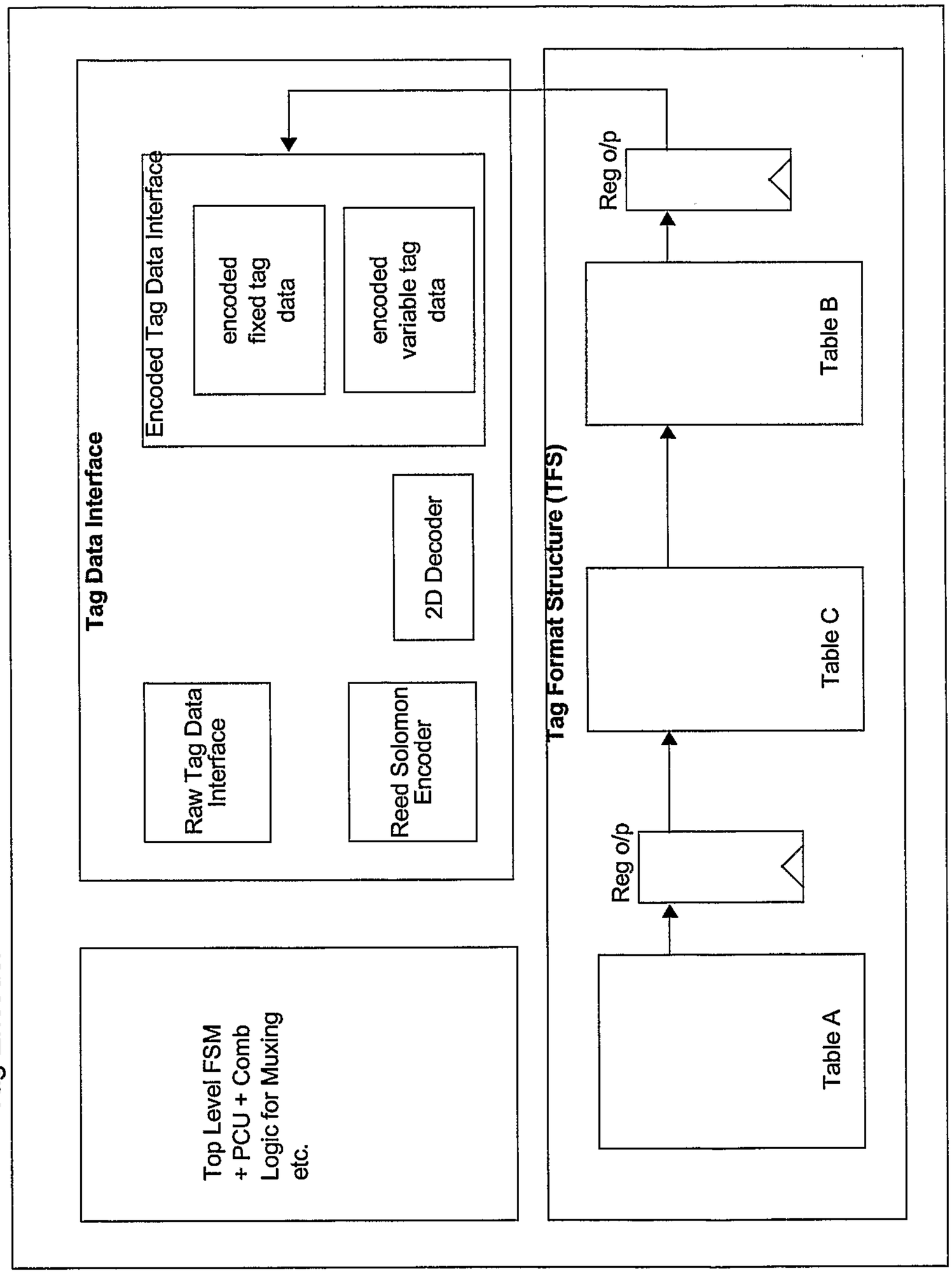

Tag Encoder
Top Level FSM
+ PCU + Comb
Logic for Muxing
etc.
Tag Data Interface
Raw Tag Data Interface
Reed Solomon Encoder
2D Decoder
Encoded Tag Data Interface
encoded fixed tag data
encoded variable tag data
Tag Format Structure (TFS)
Table A
Reg o/p
Table C
Reg o/p
Table B
Reg o/p

FIG. 188

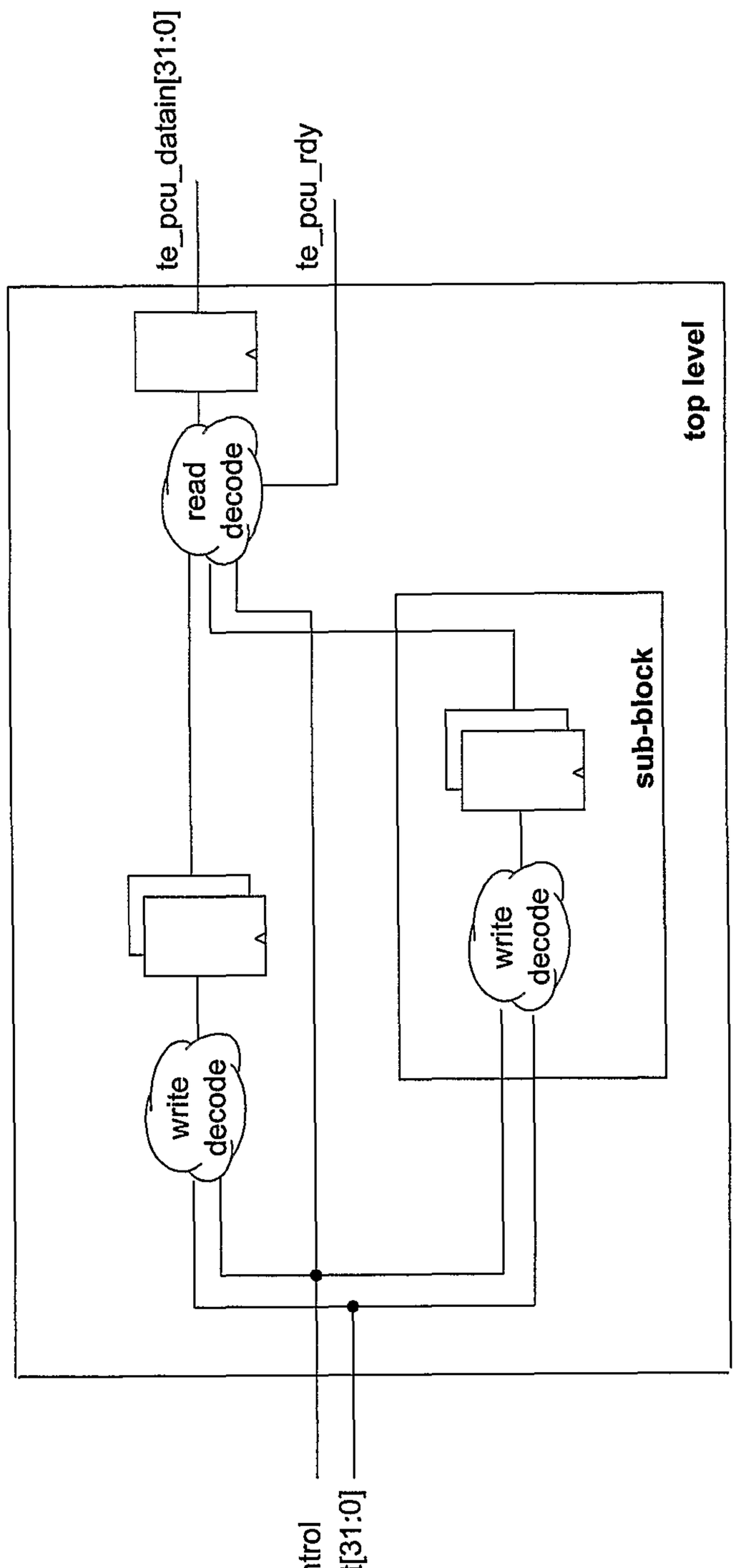

te_pcu_datain[31:0]
te_pcu_rdy
top level
read decode
sub-block
write decode
write decode
control
pcu_dataout[31:0]

FIG. 189

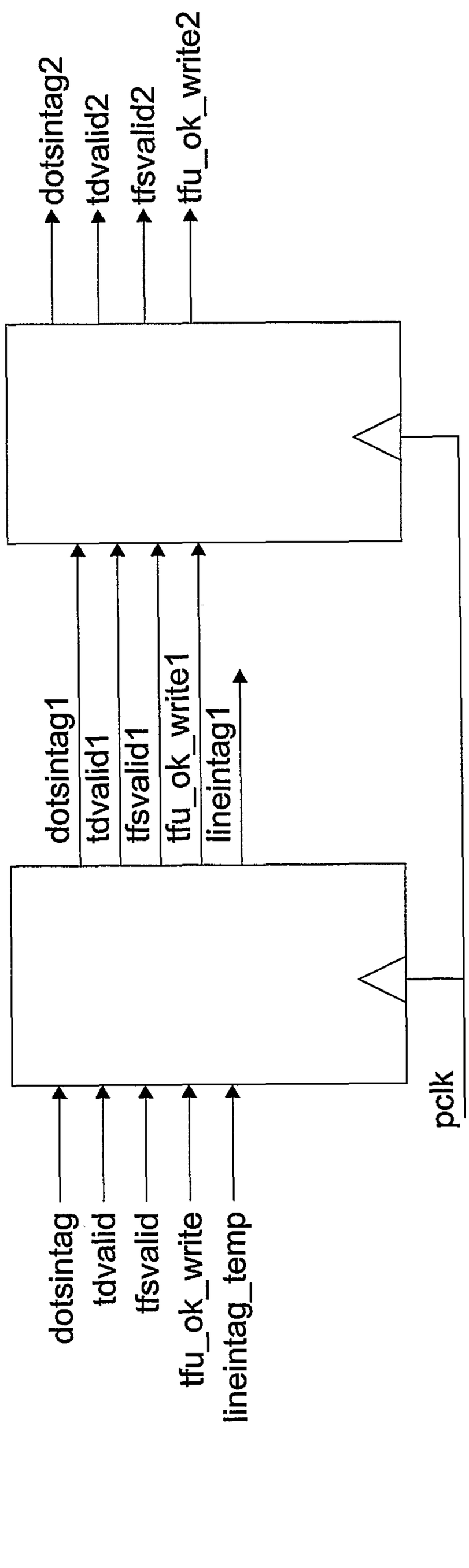

dotsintag2
tdvalid2
tfsvalid2
tfu_ok_write2
dotsintag1
tdvalid1
tfsvalid1
tfu_ok_write1
lineintag1
dotsintag
tdvalid
tfsvalid
tfu_ok_write
lineintag_temp
pclk

FIG. 191

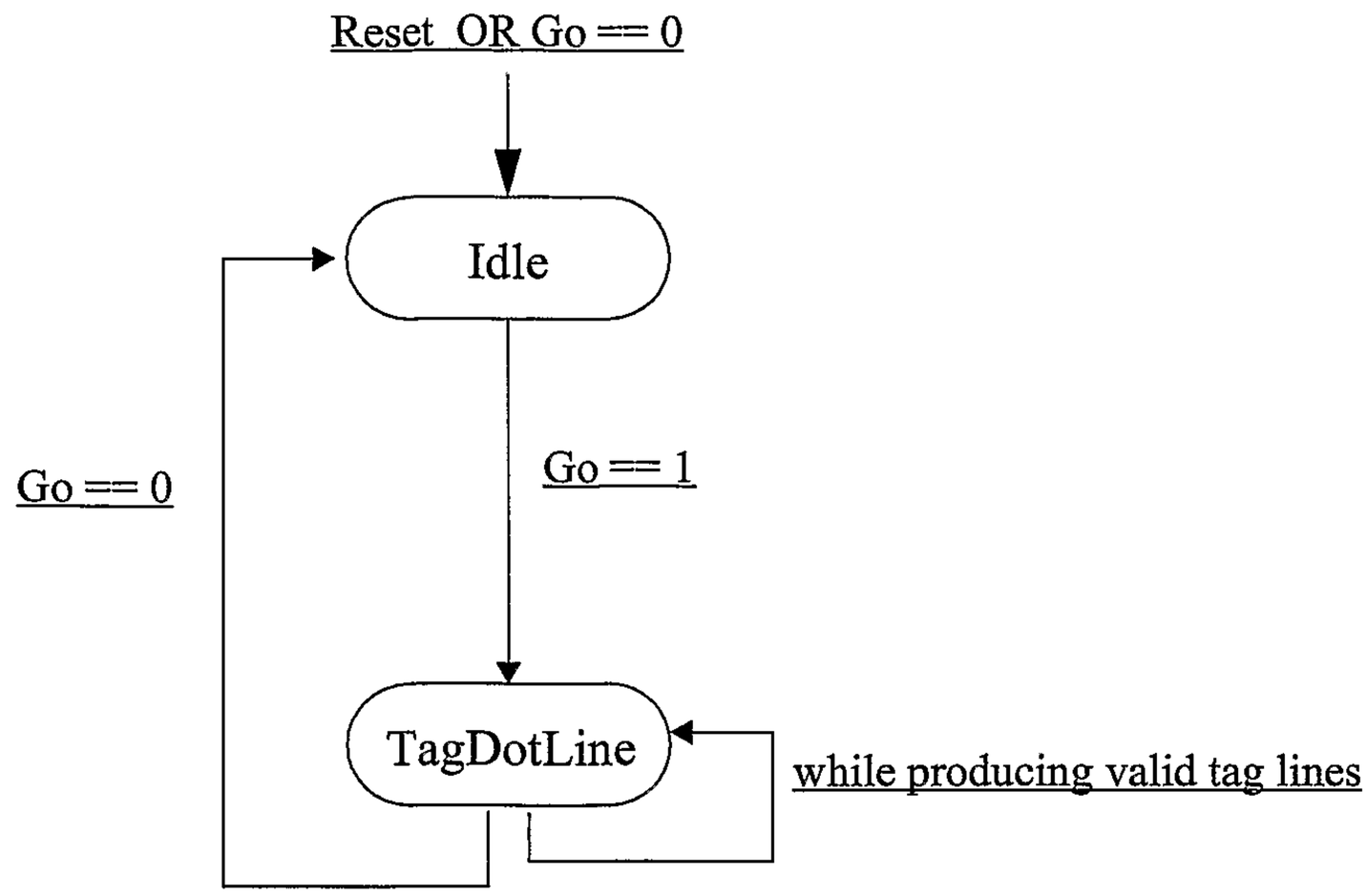


FIG. 190

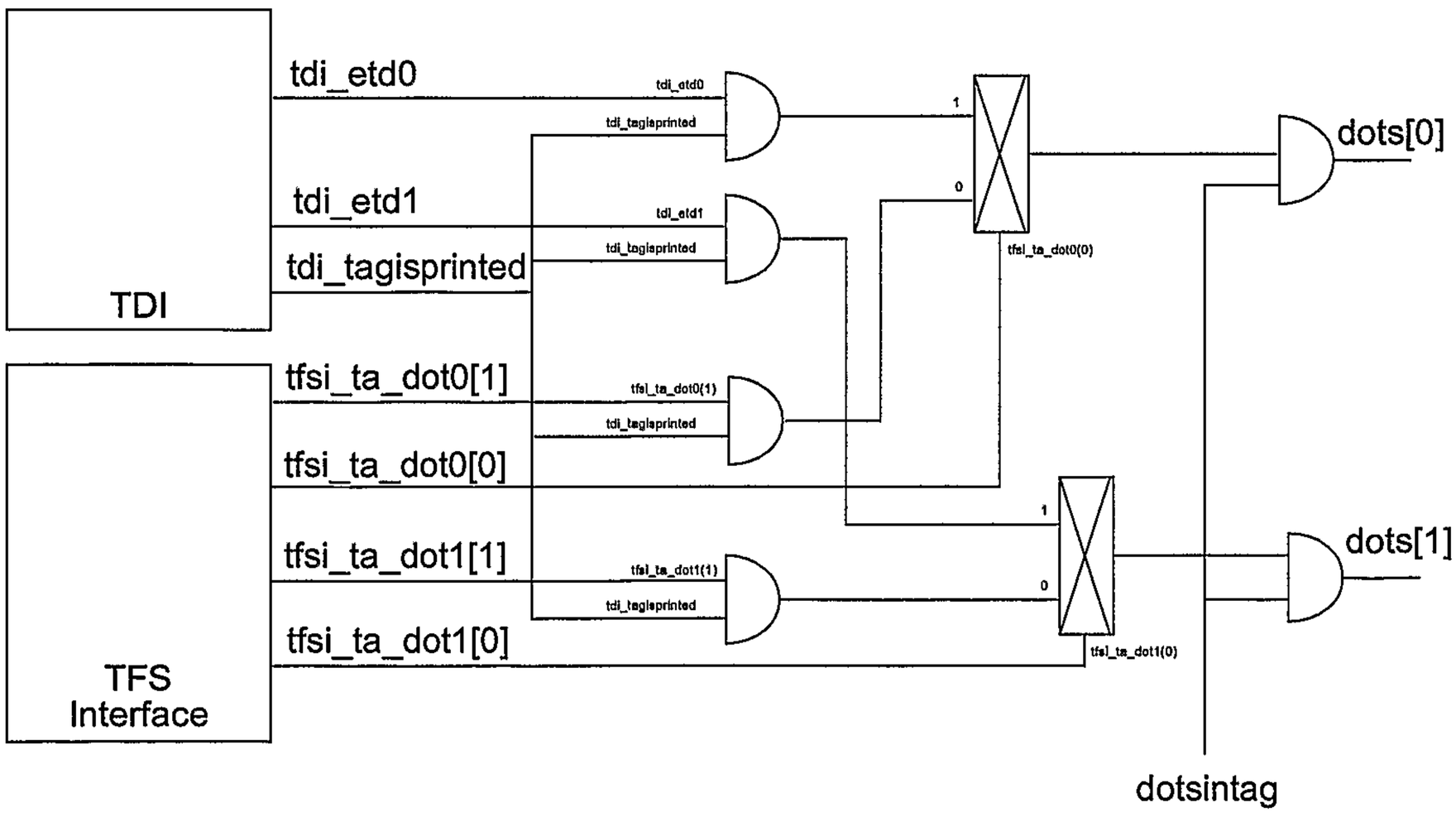


FIG. 192

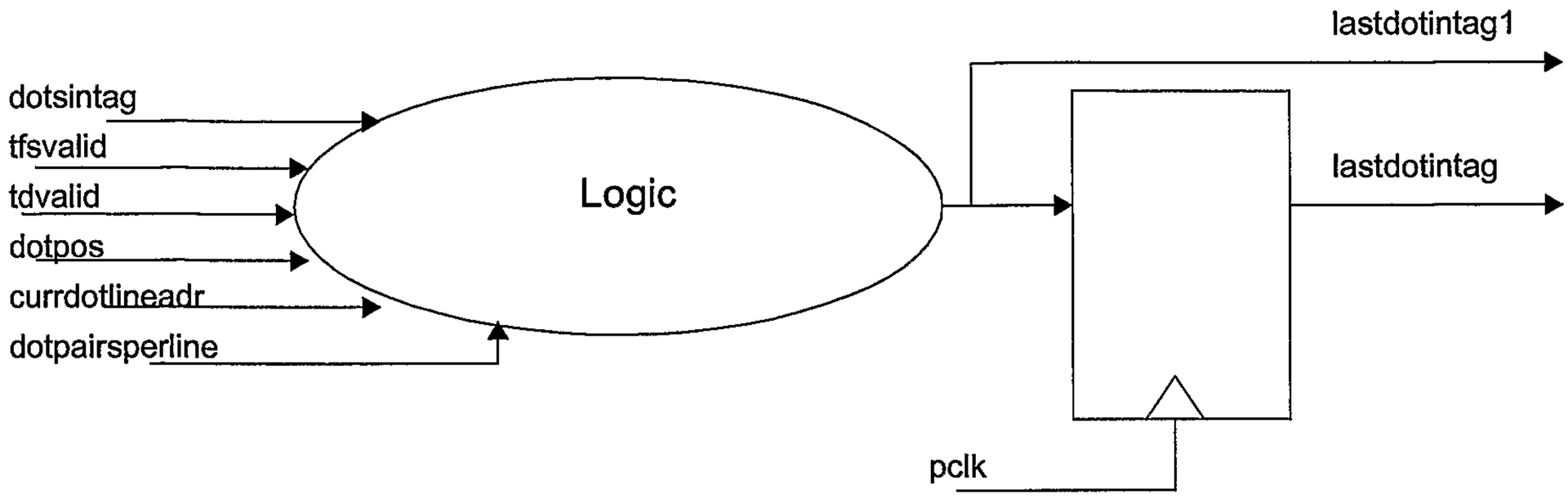


FIG. 193

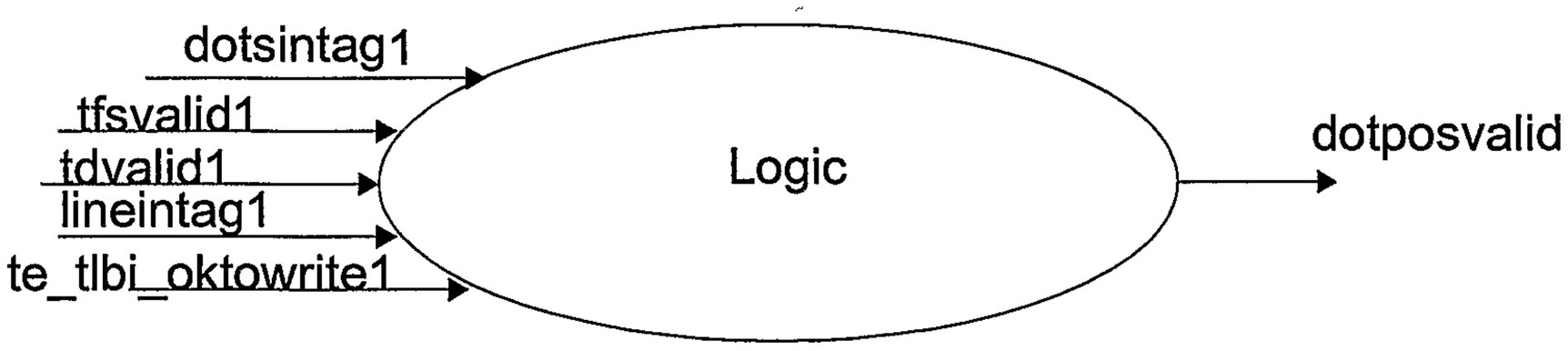


FIG. 194

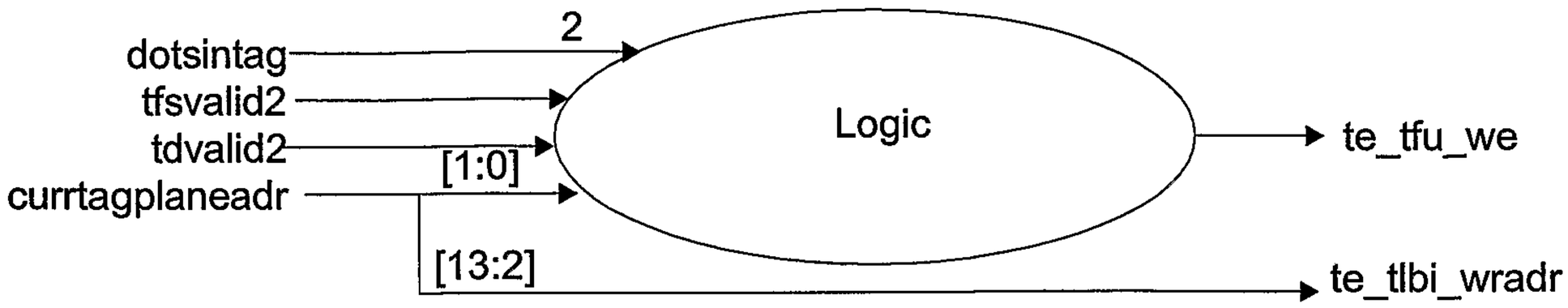


FIG. 195

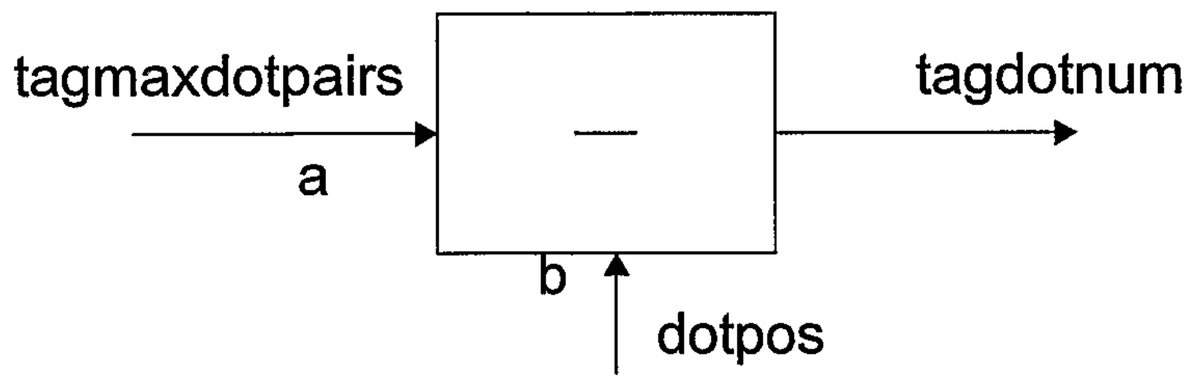


FIG. 196

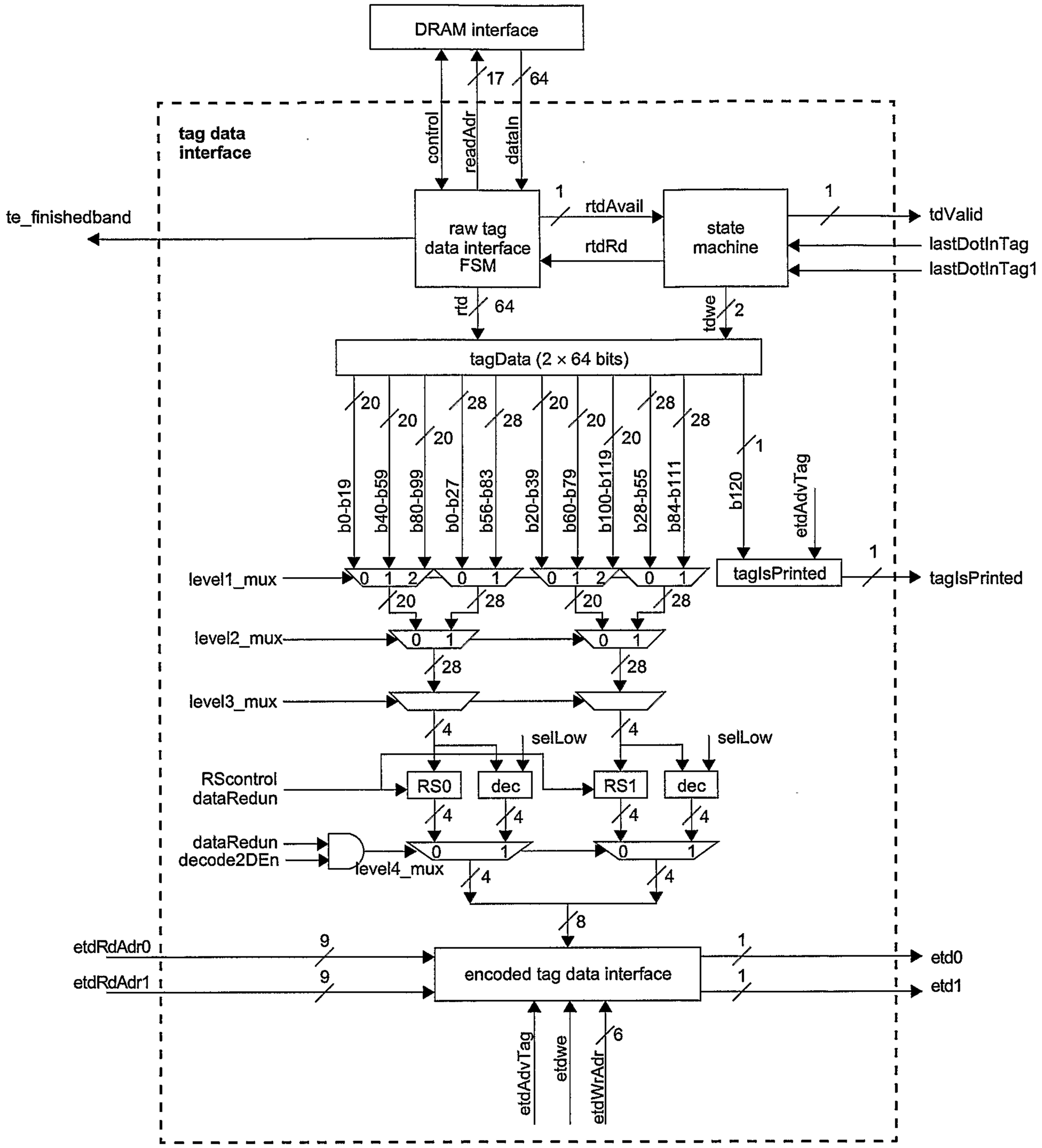

DRAM interface
17
64
tag data interface
control
readAdr
dataIn
te_finishedband
raw tag data interface FSM
1
rtdAvail
rtdRd
state machine
1
tdValid
lastDotInTag
lastDotInTag1
rtd
64
tdwe
2
tagData (2 × 64 bits)
20
28
b0-b19
b40-b59
b80-b99
b0-b27
b56-b83
b20-b39
b60-b79
b100-b119
b28-b55
b84-b111
b120
1
etdAdvTag
level1_mux
level2_mux
level3_mux
level4_mux
0 1 2
0 1
tagIsPrinted
1
tagIsPrinted
RScontrol
dataRedun
RS0
dec
RS1
dec
selLow
4
dataRedun
decode2DEn
8
etdRdAdr0
etdRdAdr1
9
encoded tag data interface
1
etd0
etd1
etdAdvTag
etdwe
etdWrAdr
6

FIG. 197

FIG. 198

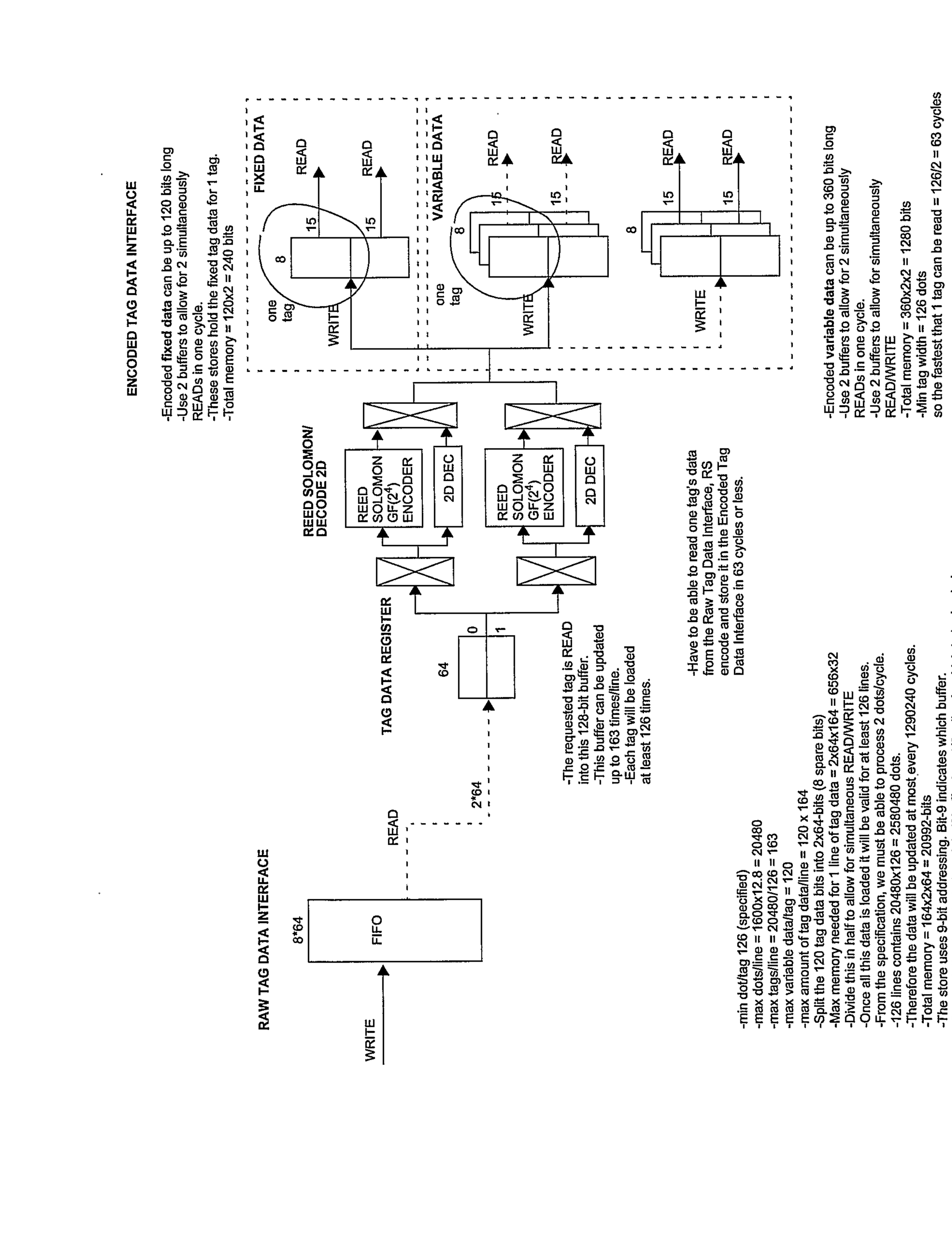
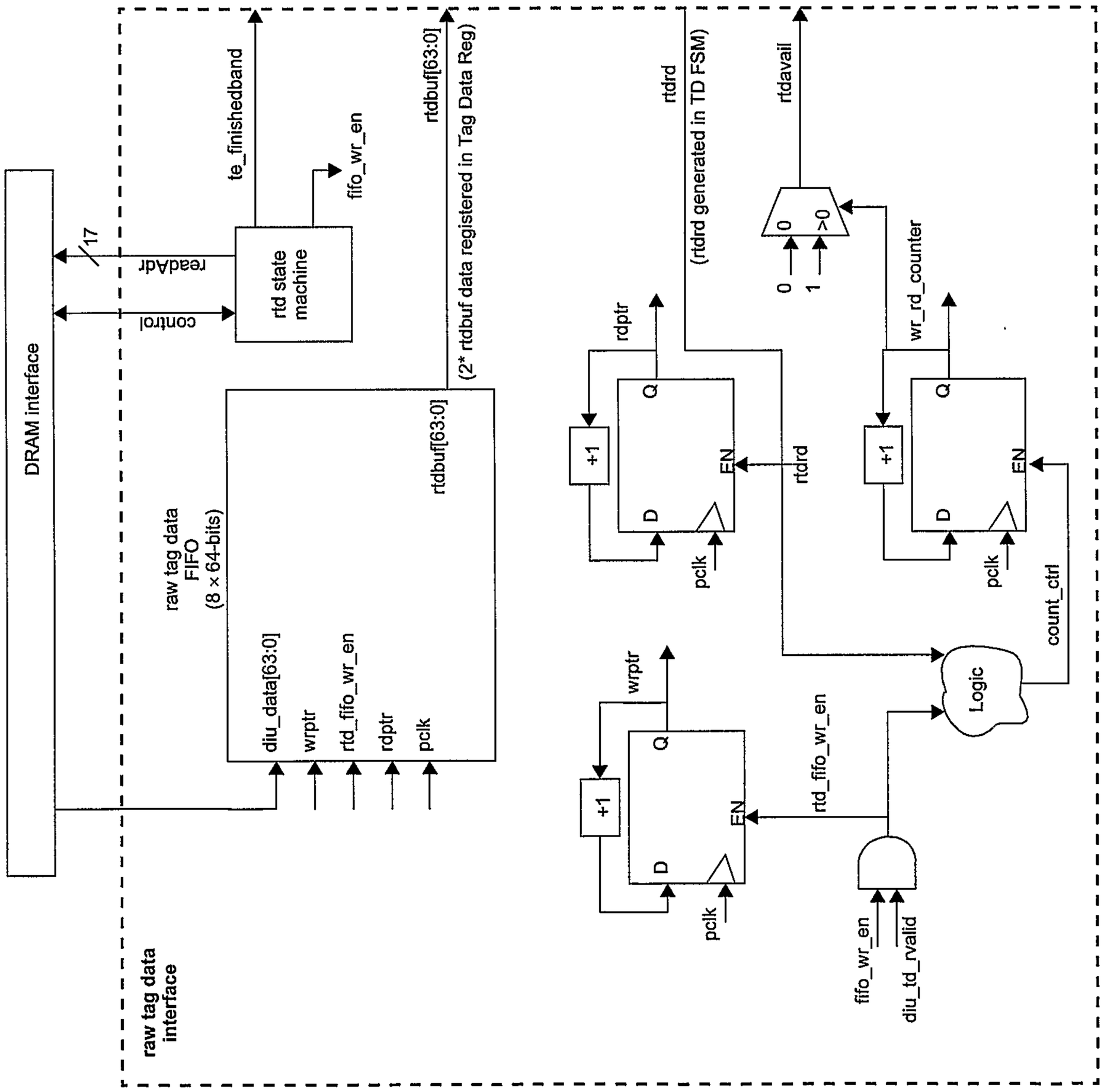

ENCODED TAG DATA INTERFACE
-Encoded fixed data can be up to 120 bits long
-Use 2 buffers to allow for 2 simultaneously READs in one cycle.
-These stores hold the fixed tag data for 1 tag.
-Total memory = 120x2 = 240 bits
FIXED DATA
VARIABLE DATA
one tag
8
15
READ
WRITE
REED SOLOMON/ DECODE 2D
REED SOLOMON GF(2^4) ENCODER
2D DEC
TAG DATA REGISTER
64
0
1
2*64
READ
-The requested tag is READ into this 128-bit buffer.
-This buffer can be updated up to 163 times/line.
-Each tag will be loaded at least 126 times.
-Encoded variable data can be up to 360 bits long
-Use 2 buffers to allow for 2 simultaneously READs in one cycle.
-Use 2 buffers to allow for simultaneously READ/WRITE
-Total memory = 360x2x2 = 1280 bits
-Min tag width = 126 dots
so the fastest that 1 tag can be read = 126/2 = 63 cycles
-Have to be able to read one tag's data from the Raw Tag Data Interface, RS encode and store it in the Encoded Tag Data Interface in 63 cycles or less.
RAW TAG DATA INTERFACE
8*64
FIFO
WRITE
READ
2*64
-min dot/tag 126 (specified)
-max dots/line = 1600x12.8 = 20480
-max tags/line = 20480/126 = 163
-max variable data/tag = 120
-max amount of tag data/line = 120 x 164
-Split the 120 tag data bits into 2x64-bits
-Max memory needed for 1 line of tag da
-Divide this in half to allow for simultaneo
-Once all this data is loaded it will be vali
-From the specification, we must be able
-126 lines contains 20480x126 = 258048
-Therefore the data will be updated at m
-Total memory = 164x2x64 = 20992-bits
-The store uses 9-bit addressing. Bit-9 in
-Once printing has started each half buff
i.e. for a 12,8 inch line it has 10240 dots
for an 8 inch line it has 6400 dots or

FIG. 199


raw tag data interface
DRAM interface
17
readAdr
control
rtd state machine
te_finishedband
fifo_wr_en
rtdbuf[63:0]
(2* rtdbuf data registered in Tag Data Reg)
raw tag data FIFO (8 × 64-bits)
diu_data[63:0]
wrptr
rtd_fifo_wr_en
rdptr
pclk
rtdbuf[63:0]
rtdrd
(rtdrd generated in TD FSM)
rtdavail
0
>0
0
1
rdptr
wr_rd_counter
+1
D
Q
EN
pclk
rtdrd
count_ctrl
wrptr
rtd_fifo_wr_en
Logic
fifo_wr_en
diu_td_rvalid

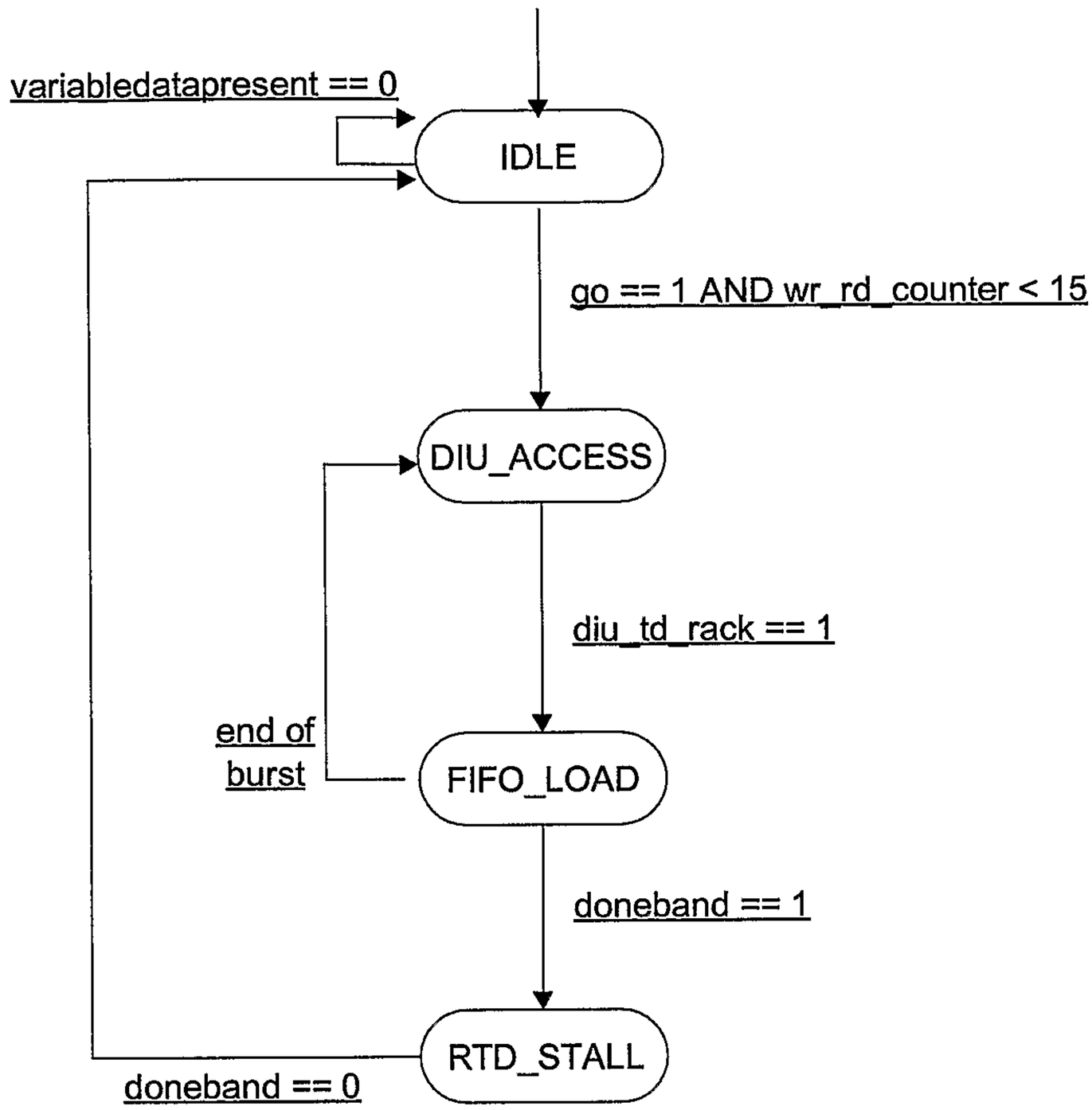


FIG. 200

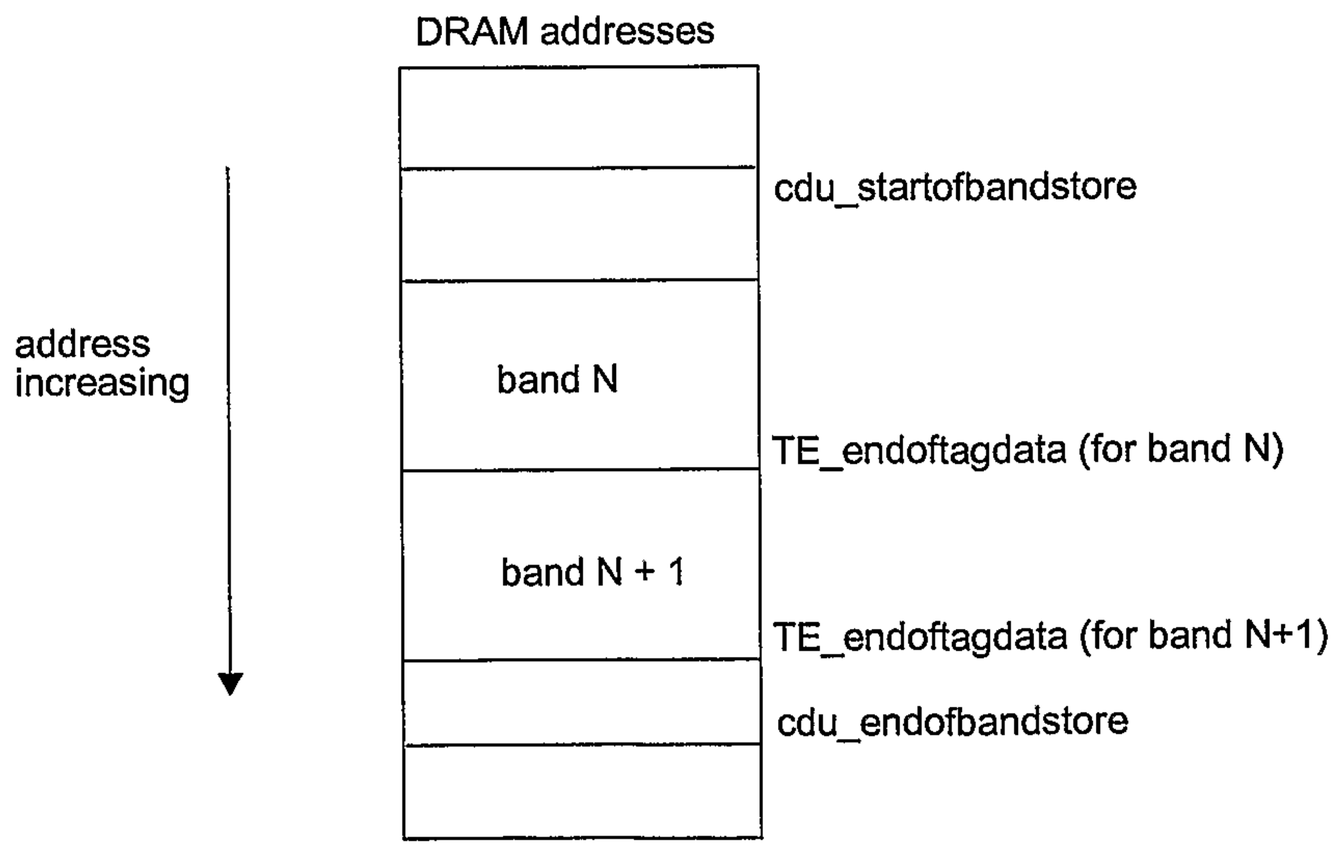


FIG. 201

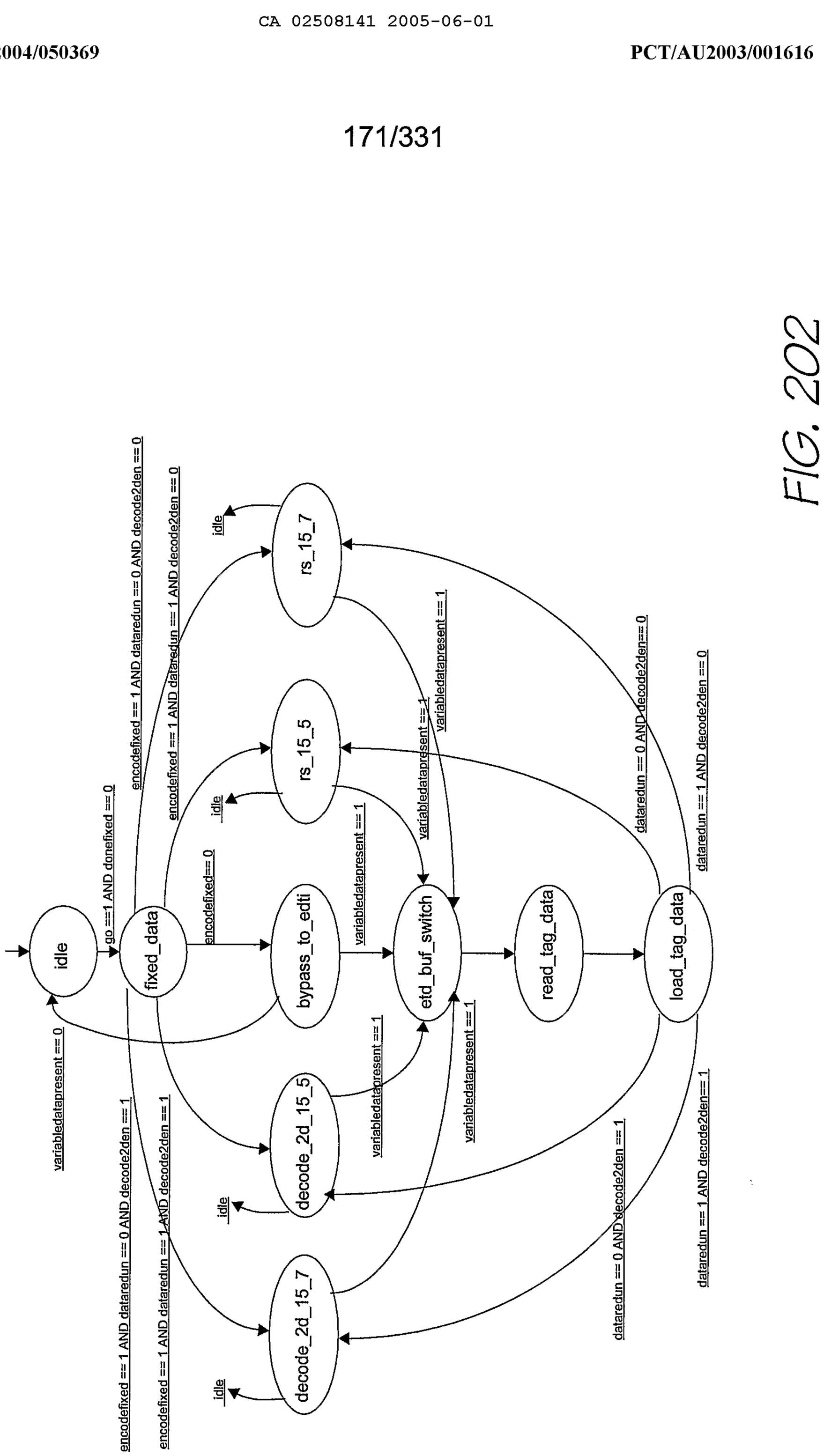

idle
go ==1 AND donefixed == 0
fixed_data
encodefixed == 1 AND dataredun == 0 AND decode2den == 0
encodefixed == 1 AND dataredun == 1 AND decode2den == 0
idle
rs_15_5
variableldatapresent == 1
variableldatapresent == 1
idle
rs_15_7
bypass_to_edfi
variableldatapresent == 1
etd_buf_switch
variableldatapresent == 1
read_tag_data
load_tag_data
dataredun == 0 AND decode2den== 0
dataredun == 1 AND decode2den == 0
variableldatapresent == 0
encodefixed == 1 AND dataredun == 0 AND decode2den == 1
encodefixed == 1 AND dataredun == 1 AND decode2den == 1
idle
decode_2d_15_5
variableldatapresent == 1
variableldatapresent == 1
idle
decode_2d_15_7
dataredun == 0 AND decode2den == 1
dataredun == 1 AND decode2den== 1

FIG. 202

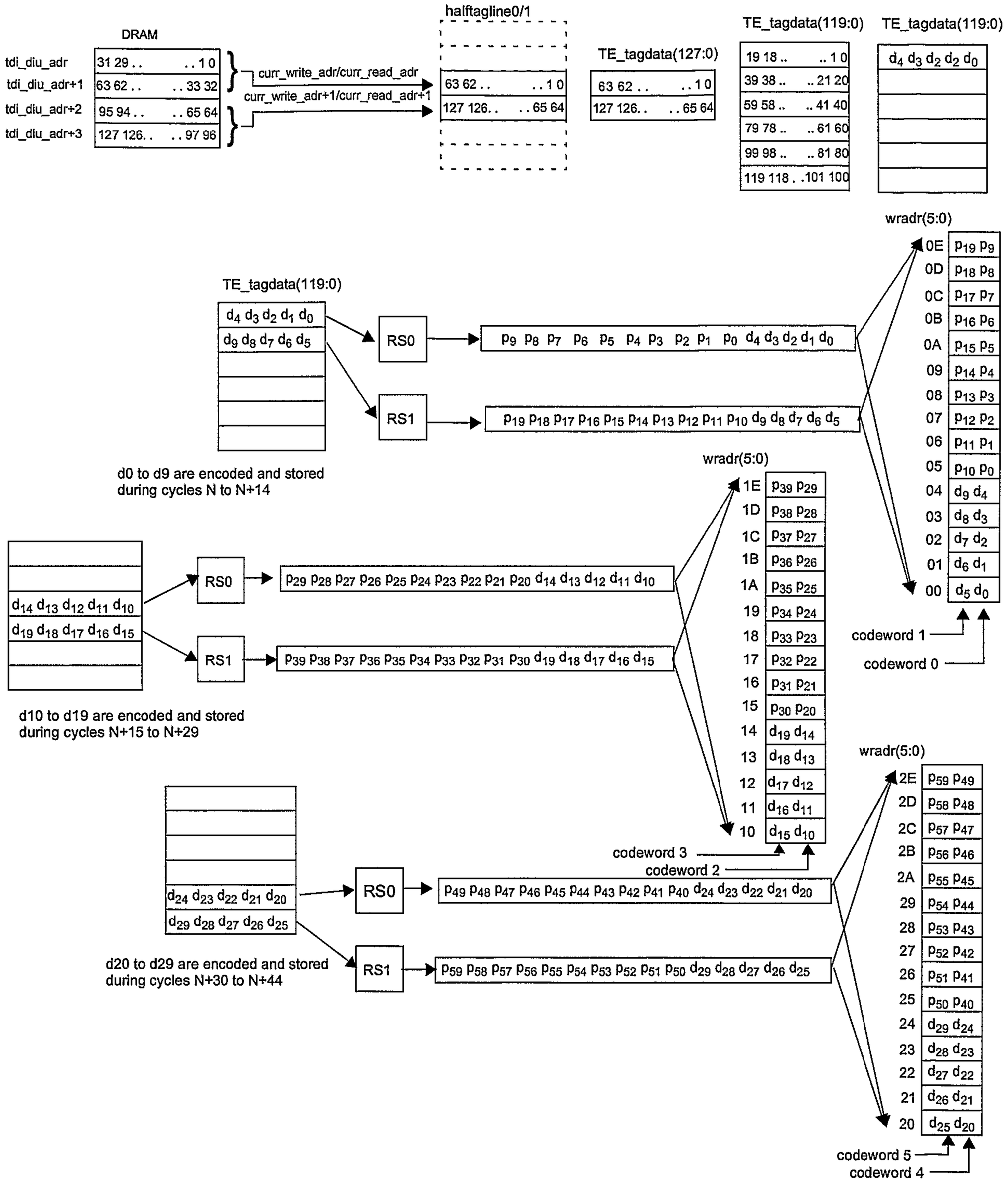

DRAM
tdi_diu_adr
tdi_diu_adr+1
tdi_diu_adr+2
tdi_diu_adr+3
31 29 . . . . 1 0
63 62 . . . . 33 32
95 94 . . . . 65 64
127 126. . . . 97 96
curr_write_adr/curr_read_adr
curr_write_adr+1/curr_read_adr+1
halftagline0/1
63 62 . . . . 1 0
127 126. . . . 65 64
TE_tagdata(127:0)
63 62 . . . . 1 0
127 126. . . . 65 64
TE_tagdata(119:0)
19 18 . . . . 1 0
39 38 . . . . 21 20
59 58 . . . . 41 40
79 78 . . . . 61 60
99 98 . . . . 81 80
119 118 . . . 101 100
TE_tagdata(119:0)
d4 d3 d2 d2 d0
wradr(5:0)
0E p19 p9
0D p18 p8
0C p17 p7
0B p16 p6
0A p15 p5
09 p14 p4
08 p13 p3
07 p12 p2
06 p11 p1
05 p10 p0
04 d9 d4
03 d8 d3
02 d7 d2
01 d6 d1
00 d5 d0
codeword 1
codeword 0
TE_tagdata(119:0)
d4 d3 d2 d1 d0
d9 d8 d7 d6 d5
RS0
RS1
p9 p8 p7 p6 p5 p4 p3 p2 p1 p0 d4 d3 d2 d1 d0
p19 p18 p17 p16 p15 p14 p13 p12 p11 p10 d9 d8 d7 d6 d5
d0 to d9 are encoded and stored during cycles N to N+14
wradr(5:0)
1E p39 p29
1D p38 p28
1C p37 p27
1B p36 p26
1A p35 p25
19 p34 p24
18 p33 p23
17 p32 p22
16 p31 p21
15 p30 p20
14 d19 d14
13 d18 d13
12 d17 d12
11 d16 d11
10 d15 d10
d14 d13 d12 d11 d10
d19 d18 d17 d16 d15
RS0
RS1
p29 p28 p27 p26 p25 p24 p23 p22 p21 p20 d14 d13 d12 d11 d10
p39 p38 p37 p36 p35 p34 p33 p32 p31 p30 d19 d18 d17 d16 d15
d10 to d19 are encoded and stored during cycles N+15 to N+29
wradr(5:0)
2E p59 p49
2D p58 p48
2C p57 p47
2B p56 p46
2A p55 p45
29 p54 p44
28 p53 p43
27 p52 p42
26 p51 p41
25 p50 p40
24 d29 d24
23 d28 d23
22 d27 d22
21 d26 d21
20 d25 d20
codeword 3
codeword 2
codeword 5
codeword 4
d24 d23 d22 d21 d20
d29 d28 d27 d26 d25
RS0
RS1
p49 p48 p47 p46 p45 p44 p43 p42 p41 p40 d24 d23 d22 d21 d20
p59 p58 p57 p56 p55 p54 p53 p52 p51 p50 d29 d28 d27 d26 d25
d20 to d29 are encoded and stored during cycles N+30 to N+44

FIG. 203

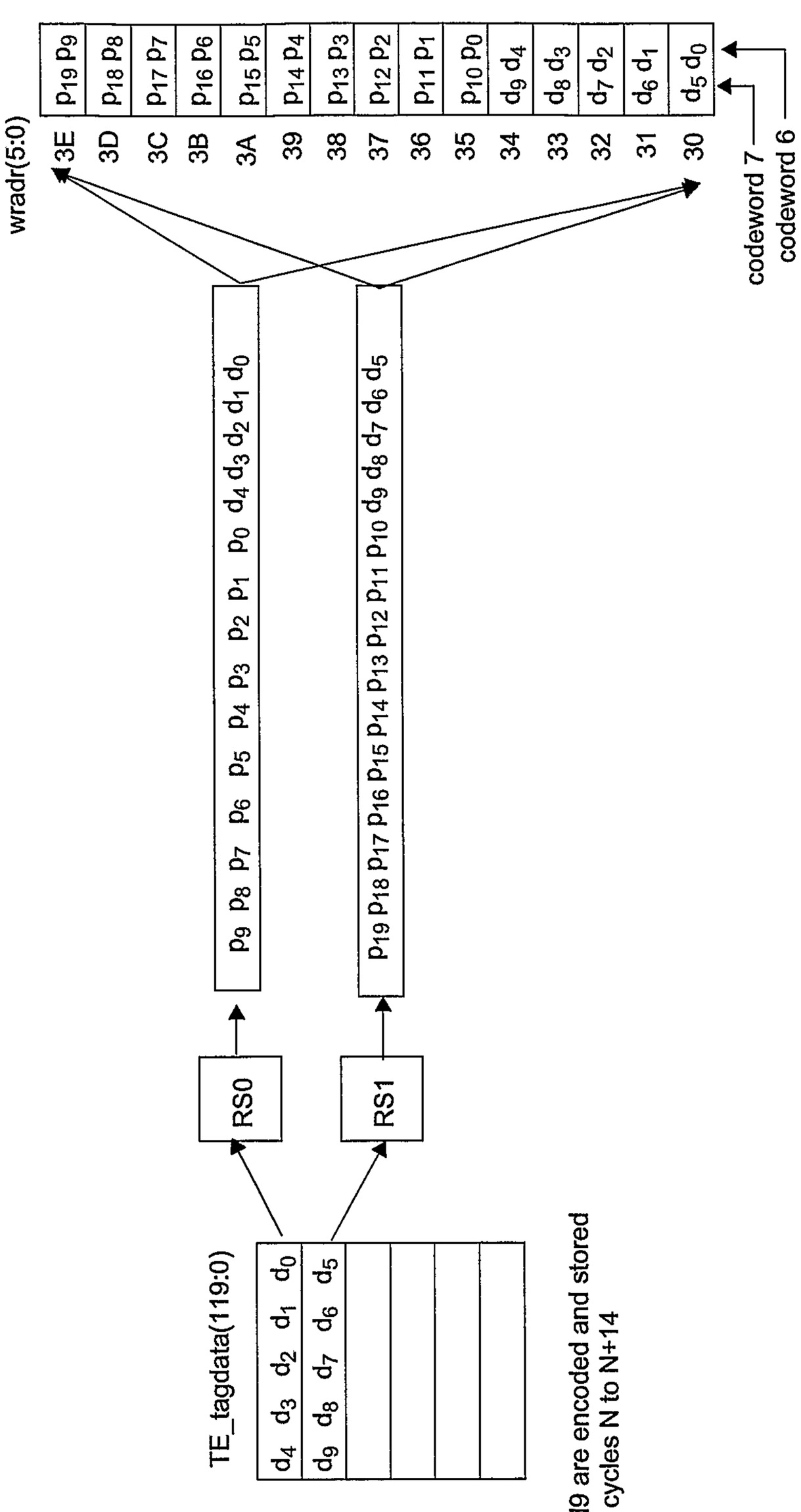

wradr(5:0)
3E 3D 3C 3B 3A 39 38 37 36 35 34 33 32 31 30
p19 p9 | p18 p8 | p17 p7 | p16 p6 | p15 p5 | p14 p4 | p13 p3 | p12 p2 | p11 p1 | p10 p0 | d9 d4 | d8 d3 | d7 d2 | d6 d1 | d5 d0
codeword 7
codeword 6
p2 p1 p0 d4 d3 d2 d1 d0
p12 p11 p10 d9 d8 d7 d6 d5
p9 p8 p7 p6 p5 p4 p3
p19 p18 p17 p16 p15 p14 p13
RS0
RS1
TE_tagdata(119:0)
d4 d3 d2 d1 d0
d9 d8 d7 d6 d5
d0 to d9 are encoded and stored
during cycles N to N+14

FIG. 204

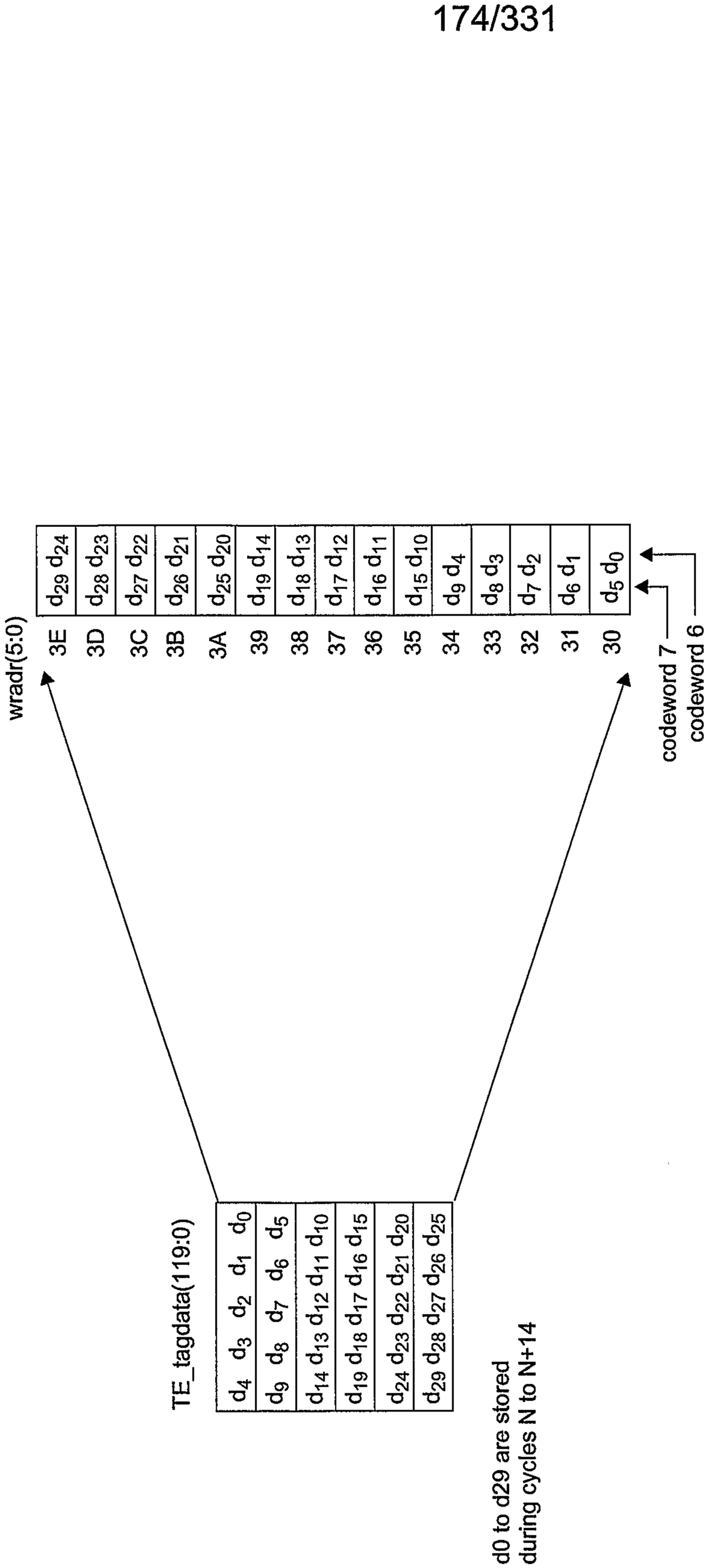

wradr(5:0)
3E d_{29} d_{24}
3D d_{28} d_{23}
3C d_{27} d_{22}
3B d_{26} d_{21}
3A d_{25} d_{20}
39 d_{19} d_{14}
38 d_{18} d_{13}
37 d_{17} d_{12}
36 d_{16} d_{11}
35 d_{15} d_{10}
34 d_9 d_4
33 d_8 d_3
32 d_7 d_2
31 d_6 d_1
30 d_5 d_0
codeword 7
codeword 6
TE_tagdata(119:0)
d_4 d_3 d_2 d_1 d_0
d_9 d_8 d_7 d_6 d_5
d_{14} d_{13} d_{12} d_{11} d_{10}
d_{19} d_{18} d_{17} d_{16} d_{15}
d_{24} d_{23} d_{22} d_{21} d_{20}
d_{29} d_{28} d_{27} d_{26} d_{25}
d0 to d29 are stored
during cycles N to N+14

FIG. 205

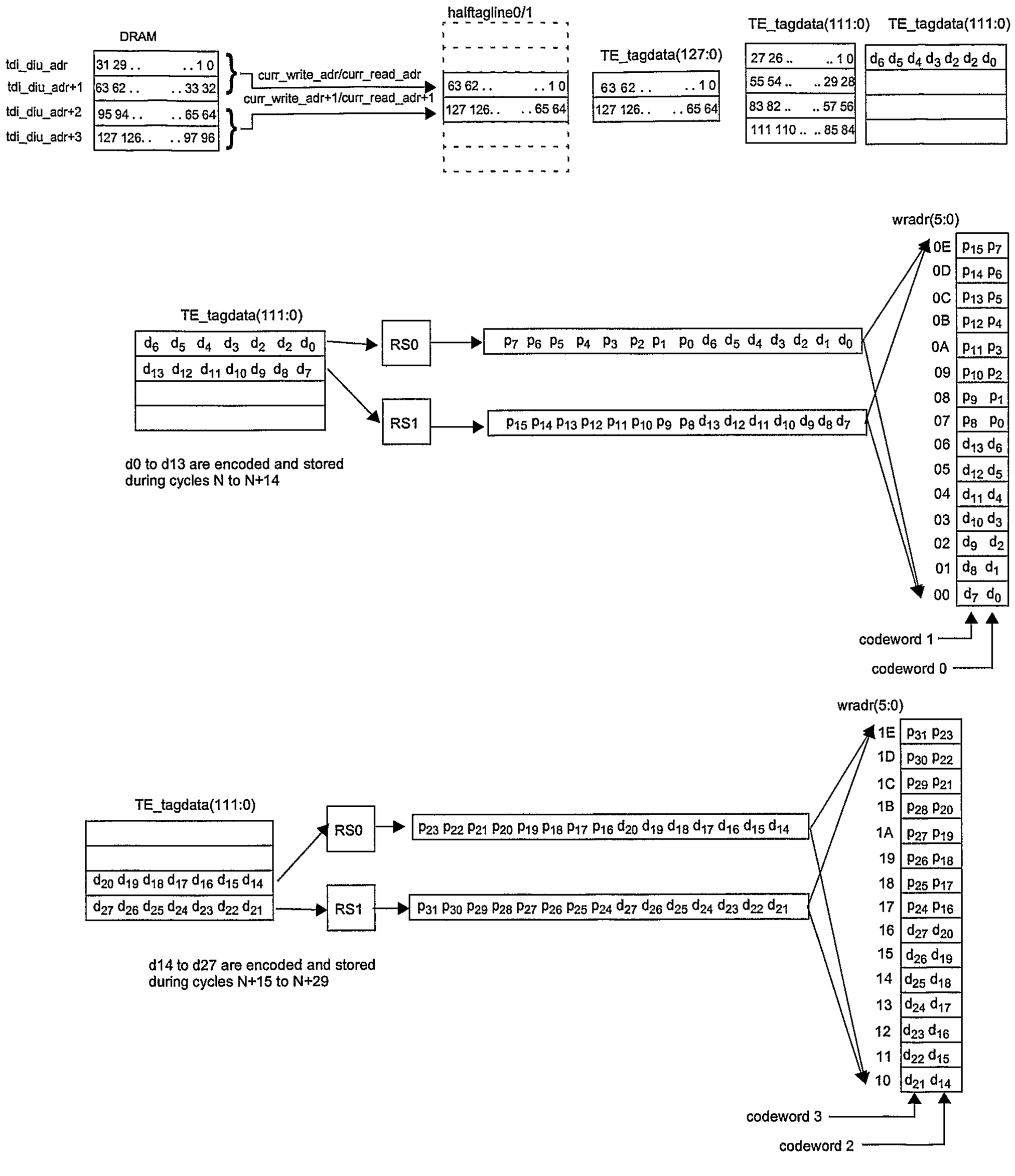

DRAM
tdi_diu_adr
tdi_diu_adr+1
tdi_diu_adr+2
tdi_diu_adr+3
31 29 . . . . 1 0
63 62 . . . . 33 32
95 94 . . . . 65 64
127 126. . . . 97 96
curr_write_adr/curr_read_adr
curr_write_adr+1/curr_read_adr+1
halftagline0/1
63 62 . . . . 1 0
127 126. . . . 65 64
TE_tagdata(127:0)
63 62 . . . . 1 0
127 126. . . . 65 64
TE_tagdata(111:0)
27 26 . . . . 1 0
55 54 . . . . 29 28
83 82 . . . . 57 56
111 110 . . . . 85 84
TE_tagdata(111:0)
d6 d5 d4 d3 d2 d2 d0
TE_tagdata(111:0)
d6 d5 d4 d3 d2 d2 d0
d13 d12 d11 d10 d9 d8 d7
RS0
RS1
p7 p6 p5 p4 p3 p2 p1 p0 d6 d5 d4 d3 d2 d1 d0
p15 p14 p13 p12 p11 p10 p9 p8 d13 d12 d11 d10 d9 d8 d7
d0 to d13 are encoded and stored
during cycles N to N+14
wradr(5:0)
0E p15 p7
0D p14 p6
0C p13 p5
0B p12 p4
0A p11 p3
09 p10 p2
08 p9 p1
07 p8 p0
06 d13 d6
05 d12 d5
04 d11 d4
03 d10 d3
02 d9 d2
01 d8 d1
00 d7 d0
codeword 1
codeword 0
TE_tagdata(111:0)
d20 d19 d18 d17 d16 d15 d14
d27 d26 d25 d24 d23 d22 d21
RS0
RS1
p23 p22 p21 p20 p19 p18 p17 p16 d20 d19 d18 d17 d16 d15 d14
p31 p30 p29 p28 p27 p26 p25 p24 d27 d26 d25 d24 d23 d22 d21
d14 to d27 are encoded and stored
during cycles N+15 to N+29
wradr(5:0)
1E p31 p23
1D p30 p22
1C p29 p21
1B p28 p20
1A p27 p19
19 p26 p18
18 p25 p17
17 p24 p16
16 d27 d20
15 d26 d19
14 d25 d18
13 d24 d17
12 d23 d16
11 d22 d15
10 d21 d14
codeword 3
codeword 2

FIG. 206

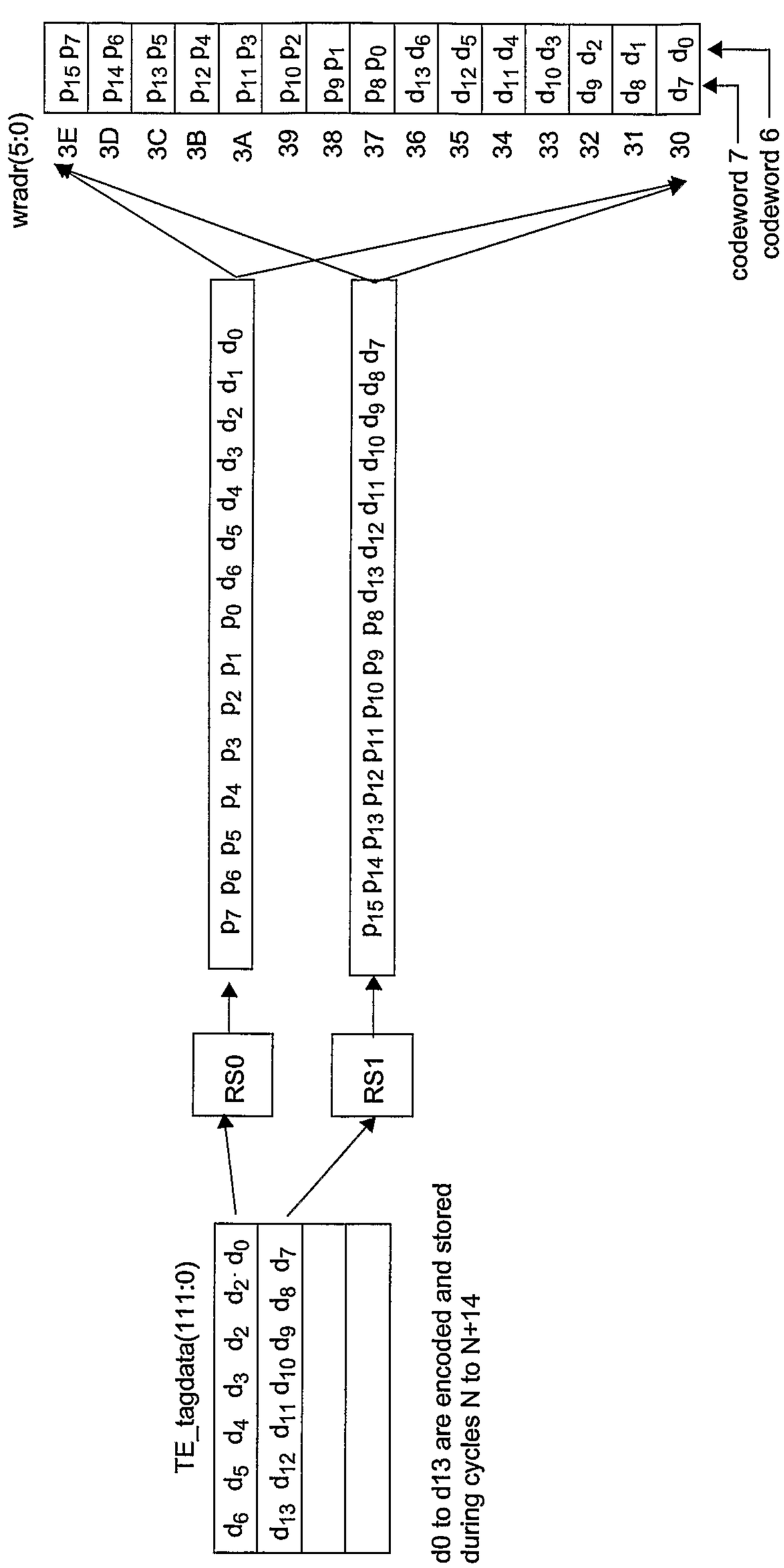

wradr(5:0)
3E
3D
3C
3B
3A
39
38
37
36
35
34
33
32
31
30
p15 p7
p14 p6
p13 p5
p12 p4
p11 p3
p10 p2
p9 p1
p8 p0
d13 d6
d12 d5
d11 d4
d10 d3
d9 d2
d8 d1
d7 d0
codeword 7
codeword 6
p7 p6 p5 p4 p3 p2 p1 p0 d6 d5 d4 d3 d2 d1 d0
p15 p14 p13 p12 p11 p10 p9 p8 d13 d12 d11 d10 d9 d8 d7
RS0
RS1
TE_tagdata(111:0)
d6 d5 d4 d3 d2 d2 d0
d13 d12 d11 d10 d9 d8 d7
d0 to d13 are encoded and stored
during cycles N to N+14

FIG. 207

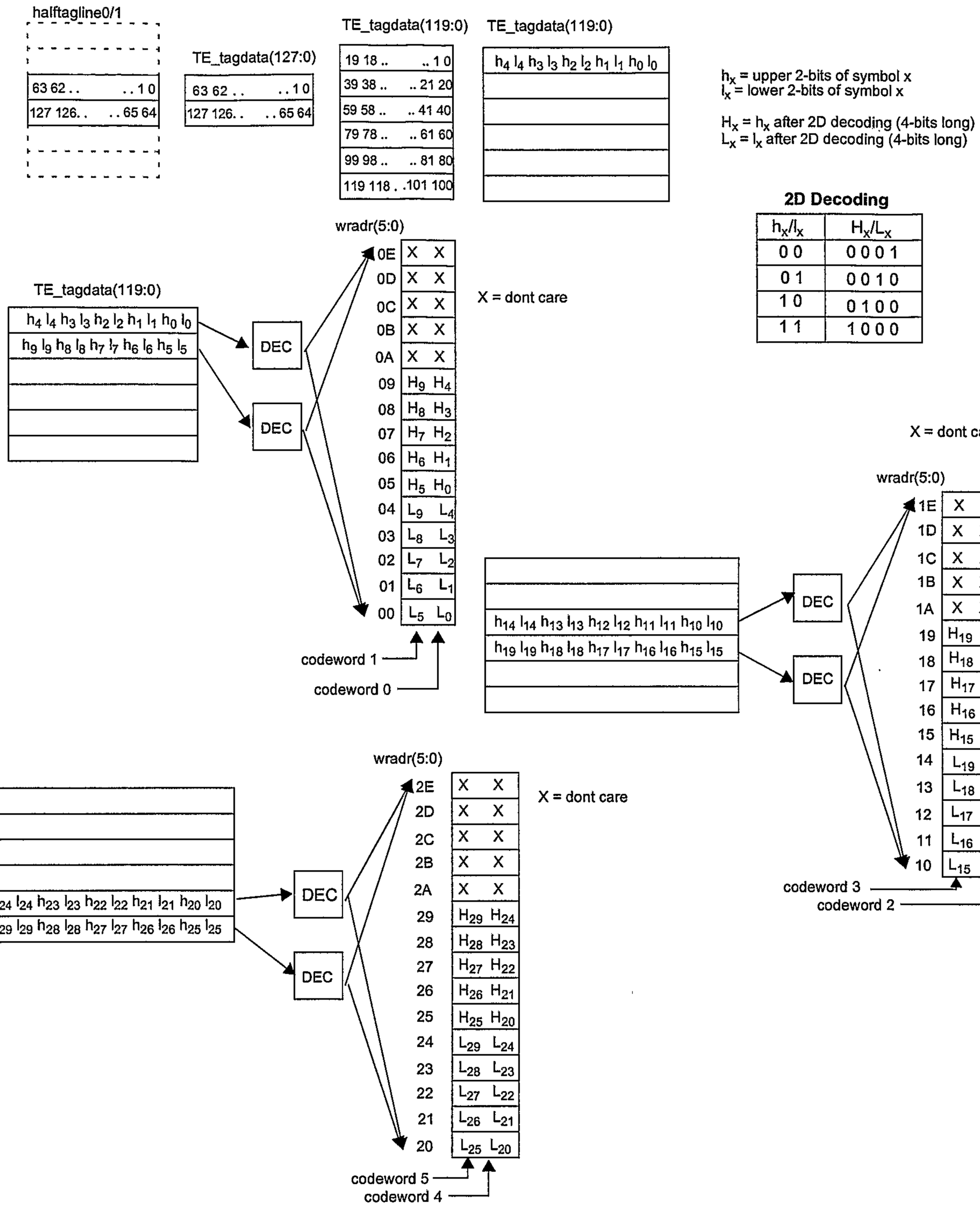

halftagline0/1
63 62 . . . . 1 0
127 126. . . . 65 64
TE_tagdata(127:0)
63 62 . . . . 1 0
127 126. . . . 65 64
TE_tagdata(119:0)
19 18 . . . . 1 0
39 38 . . . . 21 20
59 58 . . . . 41 40
79 78 . . . . 61 60
99 98 . . . . 81 80
119 118 . . . 101 100
TE_tagdata(119:0)
h4 l4 h3 l3 h2 l2 h1 l1 h0 l0
hx = upper 2-bits of symbol x
lx = lower 2-bits of symbol x
Hx = hx after 2D decoding (4-bits long)
Lx = lx after 2D decoding (4-bits long)
2D Decoding
hx/lx
Hx/Lx
0 0
0 0 0 1
0 1
0 0 1 0
1 0
0 1 0 0
1 1
1 0 0 0
wradr(5:0)
0E X X
0D X X
0C X X
0B X X
0A X X
09 H9 H4
08 H8 H3
07 H7 H2
06 H6 H1
05 H5 H0
04 L9 L4
03 L8 L3
02 L7 L2
01 L6 L1
00 L5 L0
X = dont care
TE_tagdata(119:0)
h4 l4 h3 l3 h2 l2 h1 l1 h0 l0
h9 l9 h8 l8 h7 l7 h6 l6 h5 l5
DEC
DEC
codeword 1
codeword 0
h14 l14 h13 l13 h12 l12 h11 l11 h10 l10
h19 l19 h18 l18 h17 l17 h16 l16 h15 l15
DEC
DEC
X = dont care
wradr(5:0)
1E X X
1D X X
1C X X
1B X X
1A X X
19 H19 H14
18 H18 H13
17 H17 H12
16 H16 H11
15 H15 H10
14 L19 L14
13 L18 L13
12 L17 L12
11 L16 L11
10 L15 L10
codeword 3
codeword 2
h24 l24 h23 l23 h22 l22 h21 l21 h20 l20
h29 l29 h28 l28 h27 l27 h26 l26 h25 l25
DEC
DEC
wradr(5:0)
2E X X
2D X X
2C X X
2B X X
2A X X
29 H29 H24
28 H28 H23
27 H27 H22
26 H26 H21
25 H25 H20
24 L29 L24
23 L28 L23
22 L27 L22
21 L26 L21
20 L25 L20
X = dont care
codeword 5
codeword 4

FIG. 208

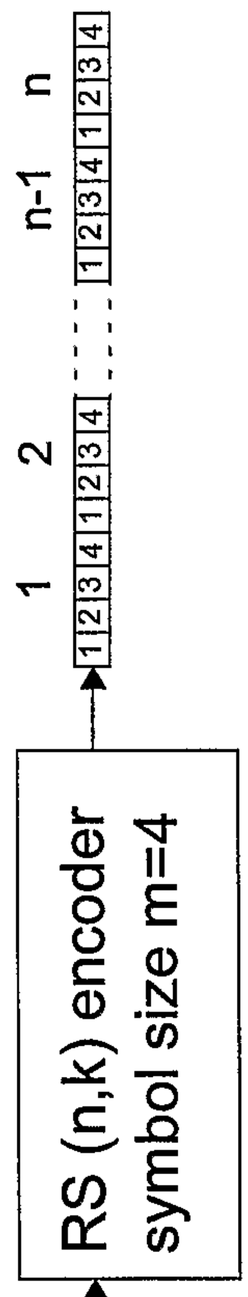


FIG. 209

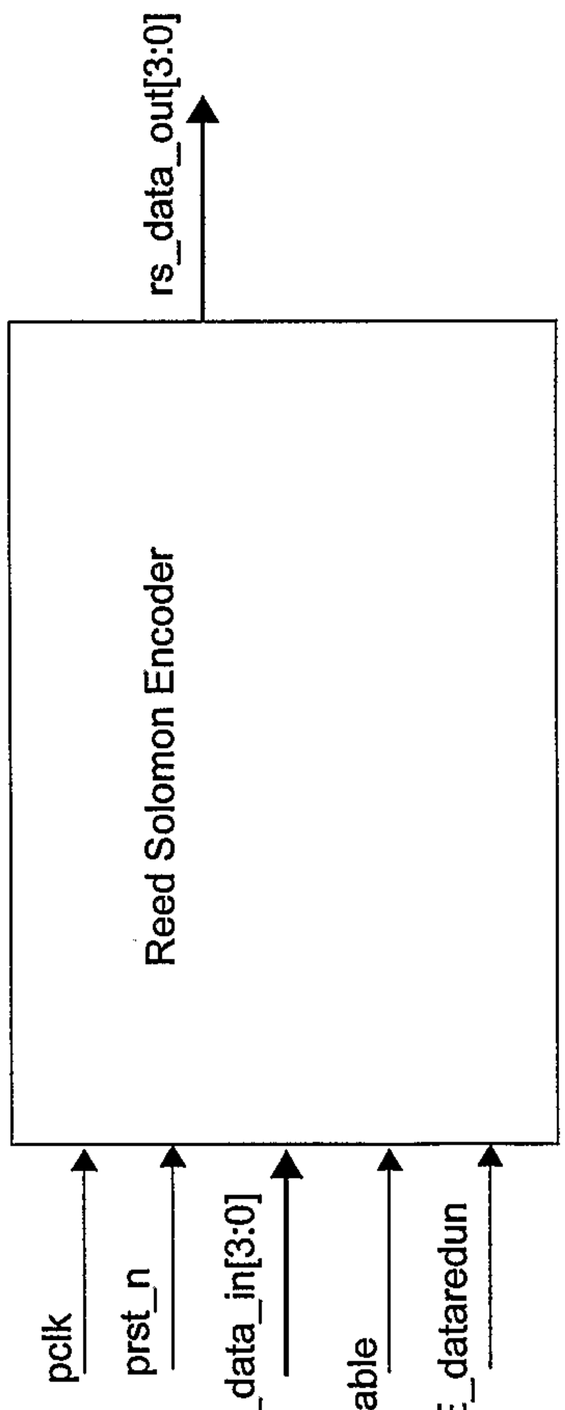


FIG. 210

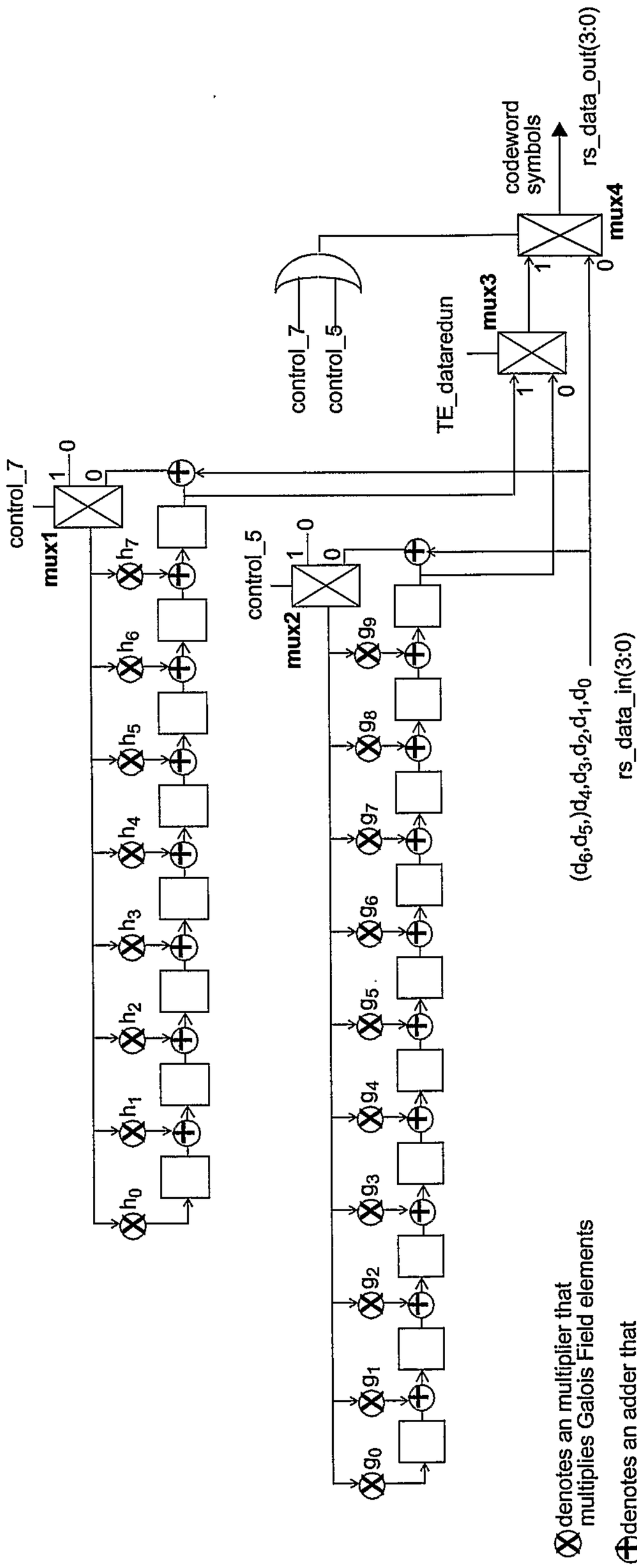

control_7
mux1
control_5
mux2
control_7
control_5
TE_dataredun
mux3
codeword symbols
rs_data_out
mux4
h_0 h_1 h_2 h_3 h_4 h_5 h_6 h_7
g_0 g_1 g_2 g_3 g_4 g_5 g_6 g_7 g_8 g_9
$(d_6,d_5,)d_4,d_3,d_2,d_1,d_0$
rs_data_in(3:0)
denotes an multiplier that multiplies Galois Field elements
denotes an adder that adds Galois Field elements

FIG. 211

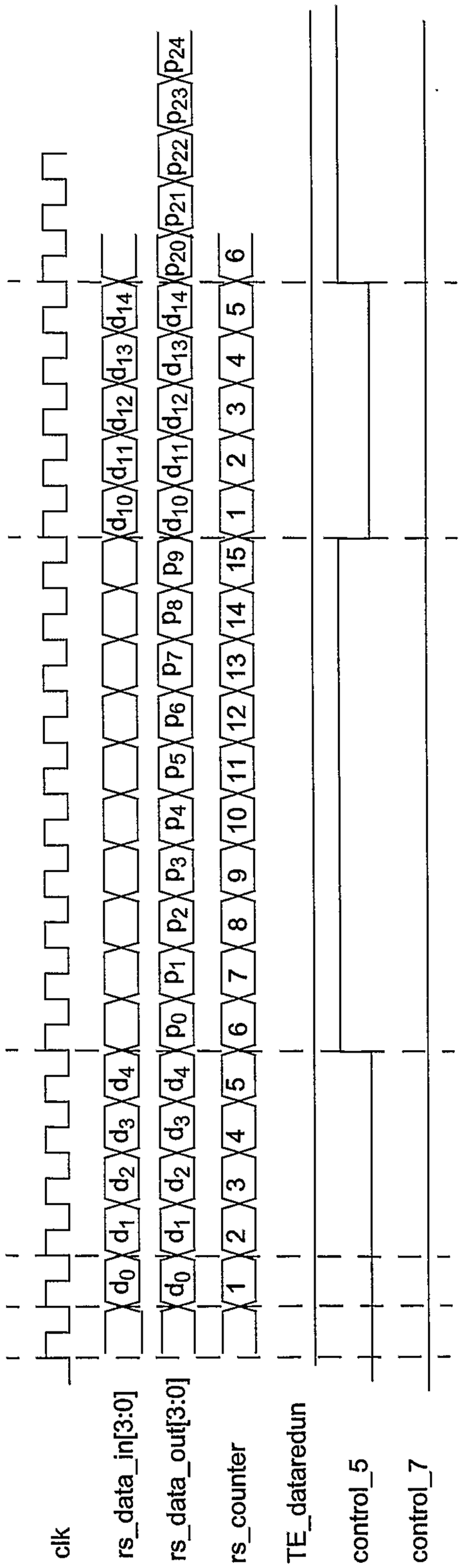

clk
rs_data_in[3:0]
rs_data_out[3:0]
rs_counter
TE_dataredun
control_5
control_7

FIG. 212

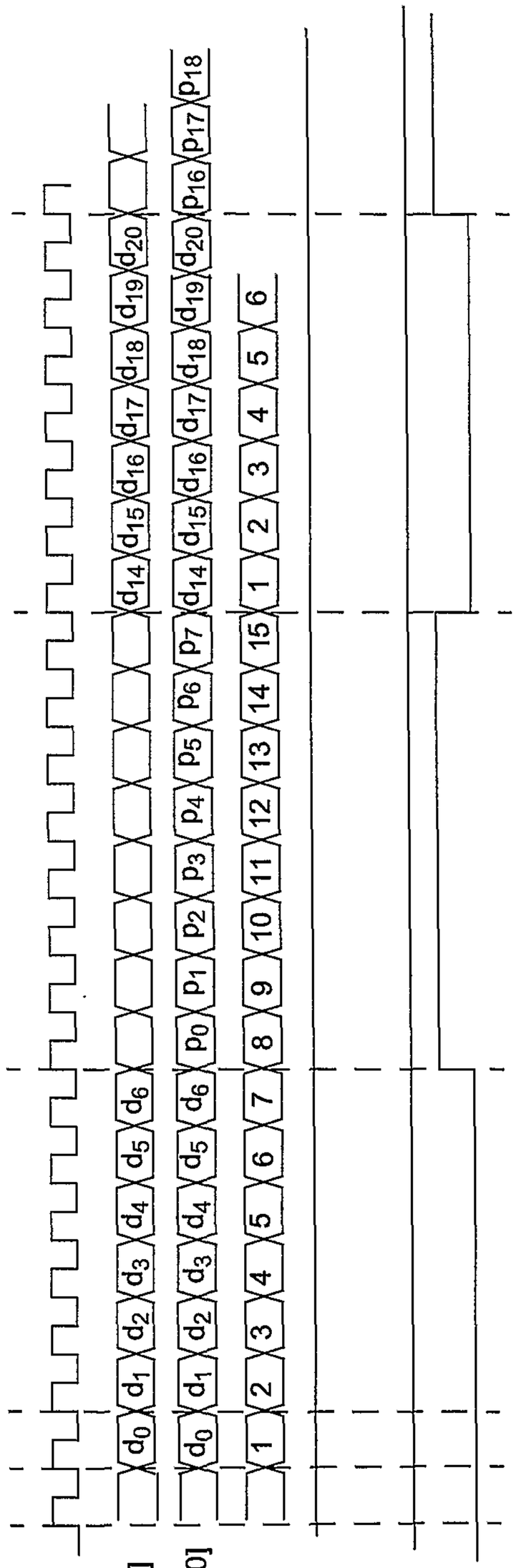

clk
rs_data_in[3:0]
rs_data_out[3:0]
rs_counter
TE_dataredun
control_5
control_7
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11 d12 d13 d14 d15 d16 d17 d18 d19 d20
p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 p16 p17 p18
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 1 2 3 4 5 6

FIG. 213

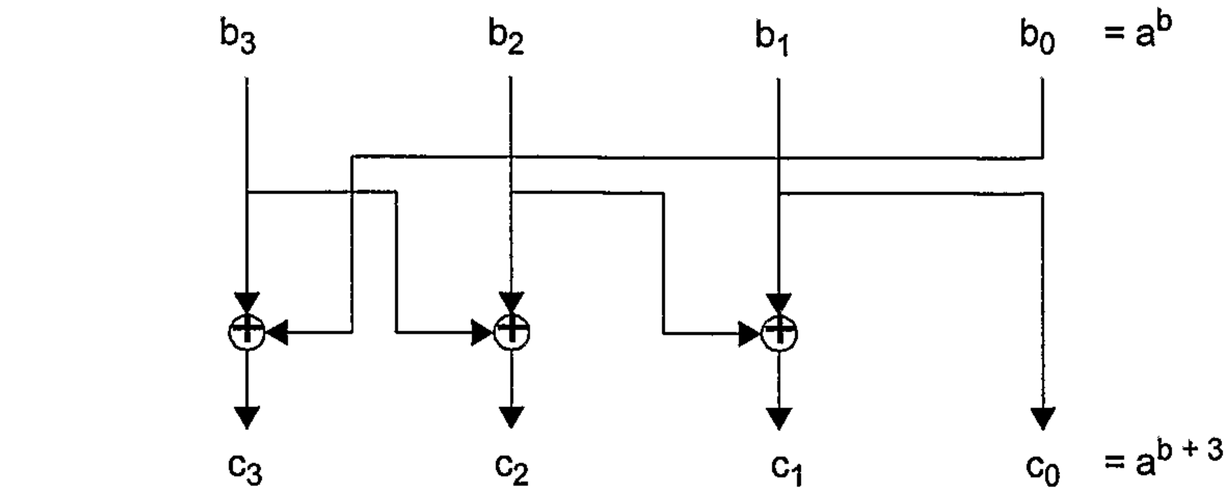


FIG. 214

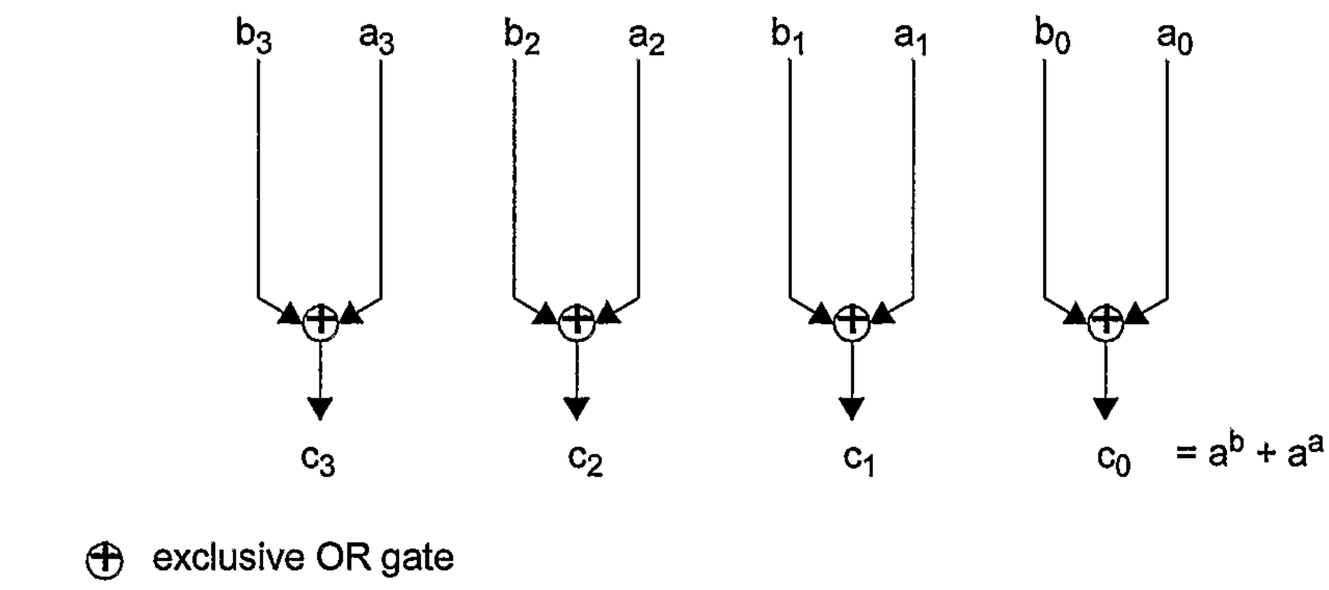


FIG. 215

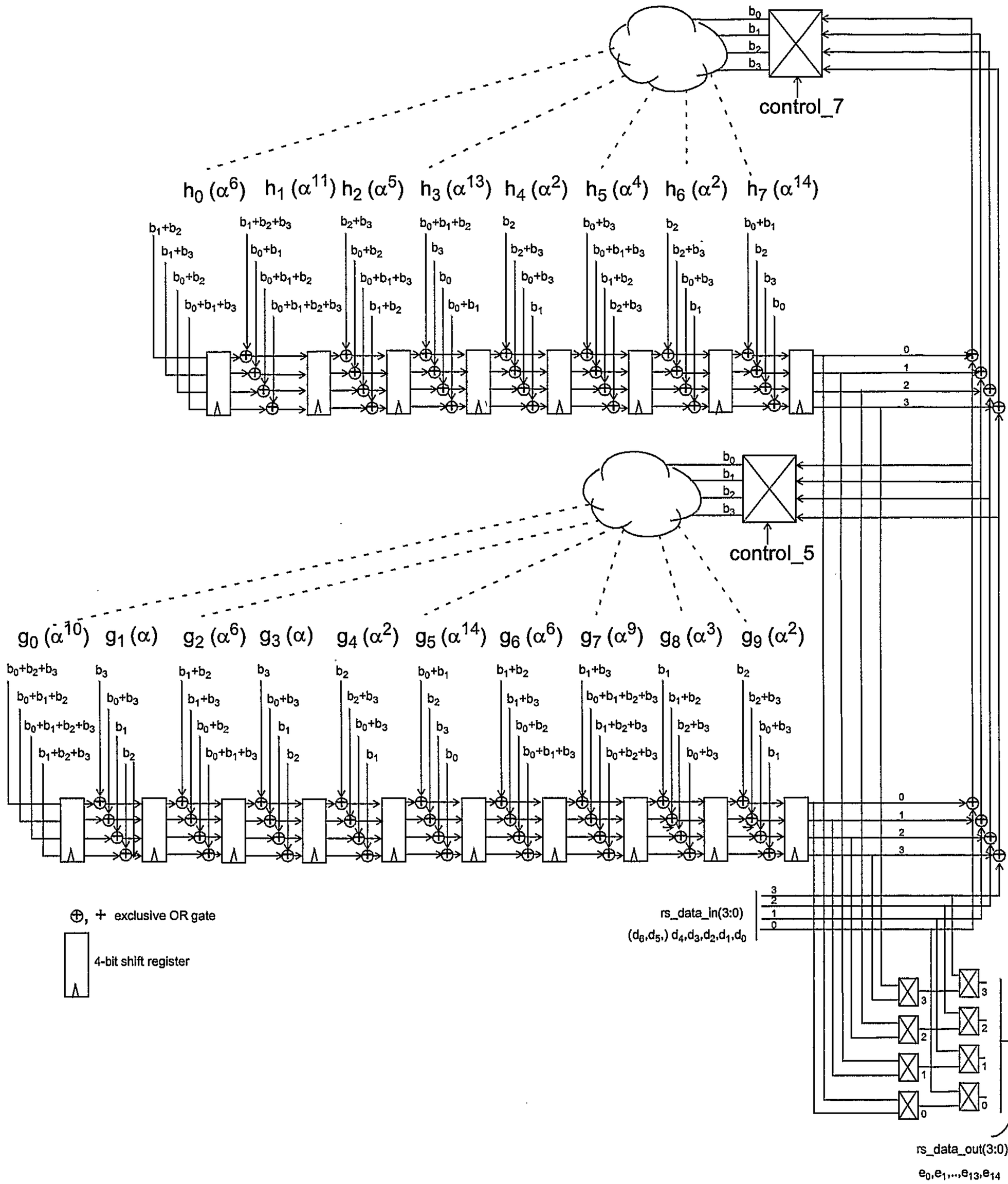

control_7
control_5
rs_data_in(3:0)
(d6,d5,) d4,d3,d2,d1,d0
rs_data_out(3:0)
e0,e1,..,e13,e14
⊕, + exclusive OR gate
4-bit shift register

FIG. 216

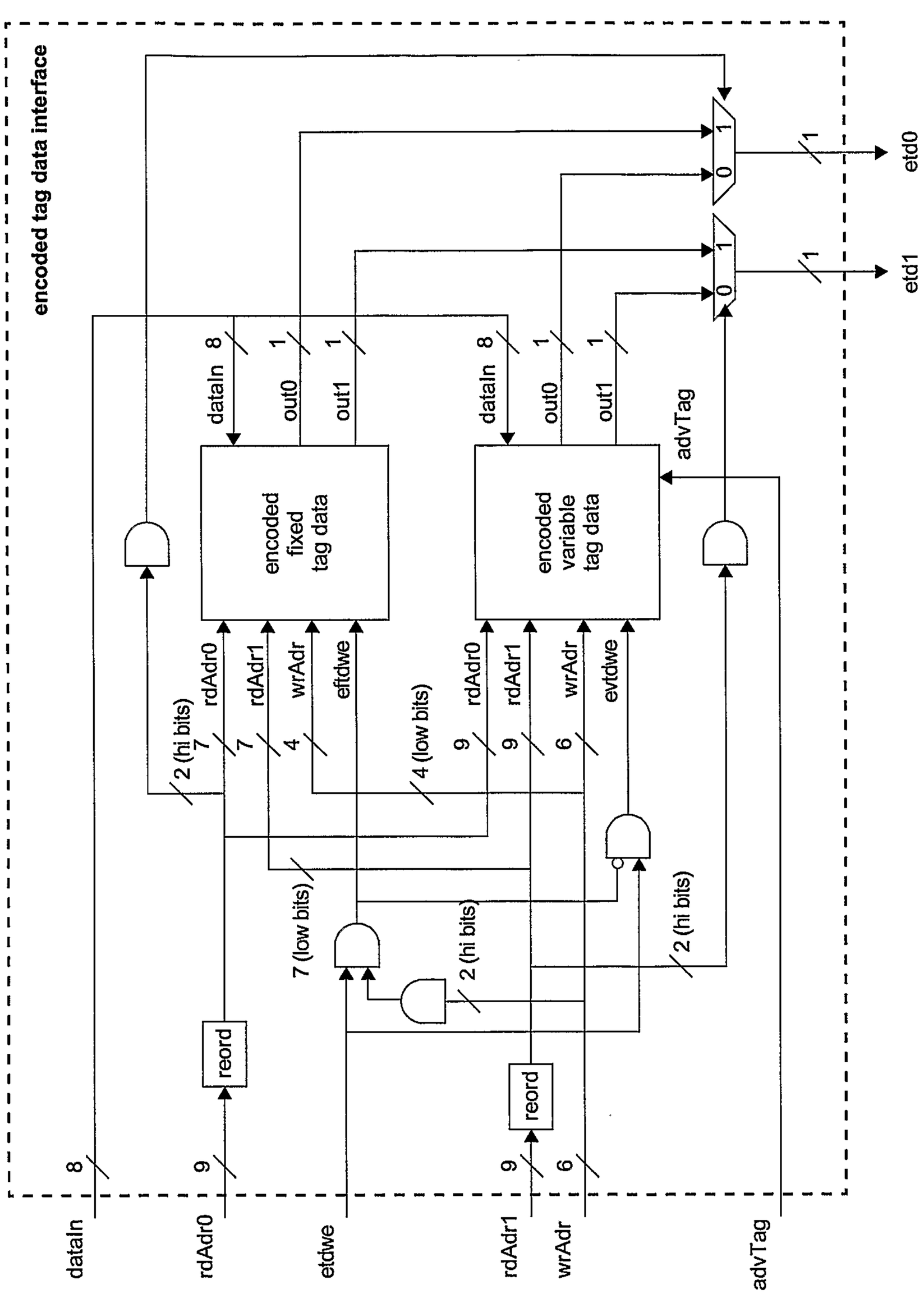

encoded tag data interface
dataIn
8
out0
1
out1
1
encoded fixed tag data
2 (hi bits)
rdAdr0
7
rdAdr1
7
wrAdr
4
eftdwe
7 (low bits)
reord
encoded variable tag data
4 (low bits)
rdAdr0
9
rdAdr1
9
wrAdr
6
evtdwe
2 (hi bits)
2 (hi bits)
reord
advTag
etd0
etd1
dataIn
rdAdr0
etdwe
rdAdr1
wrAdr
advTag

FIG. 217

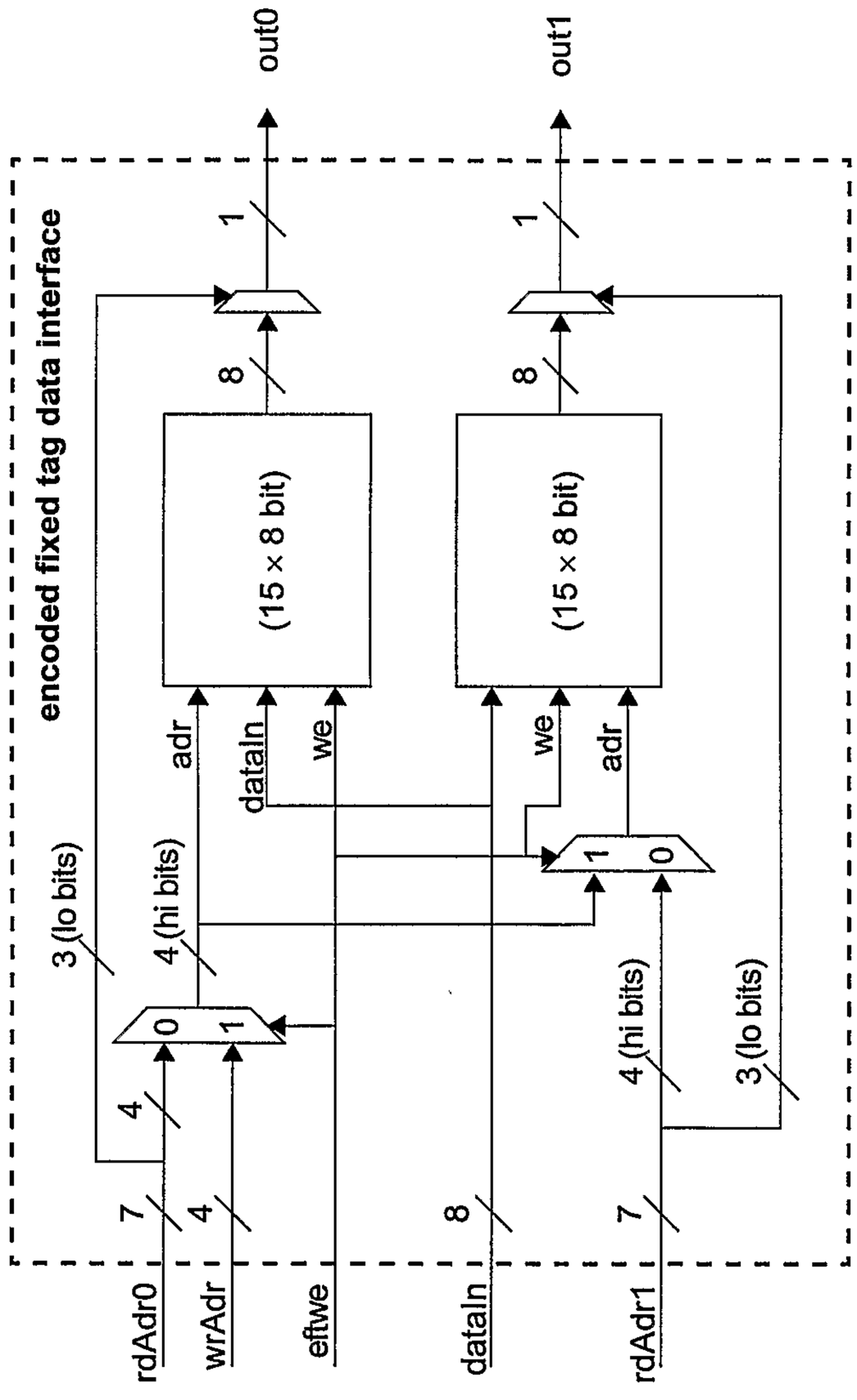

encoded fixed tag data interface
out0
out1
1
8
(15 × 8 bit)
adr
dataIn
we
3 (lo bits)
4 (hi bits)
0
1
4
7
rdAdr0
wrAdr
eftwe
8
dataIn
rdAdr1

FIG. 218

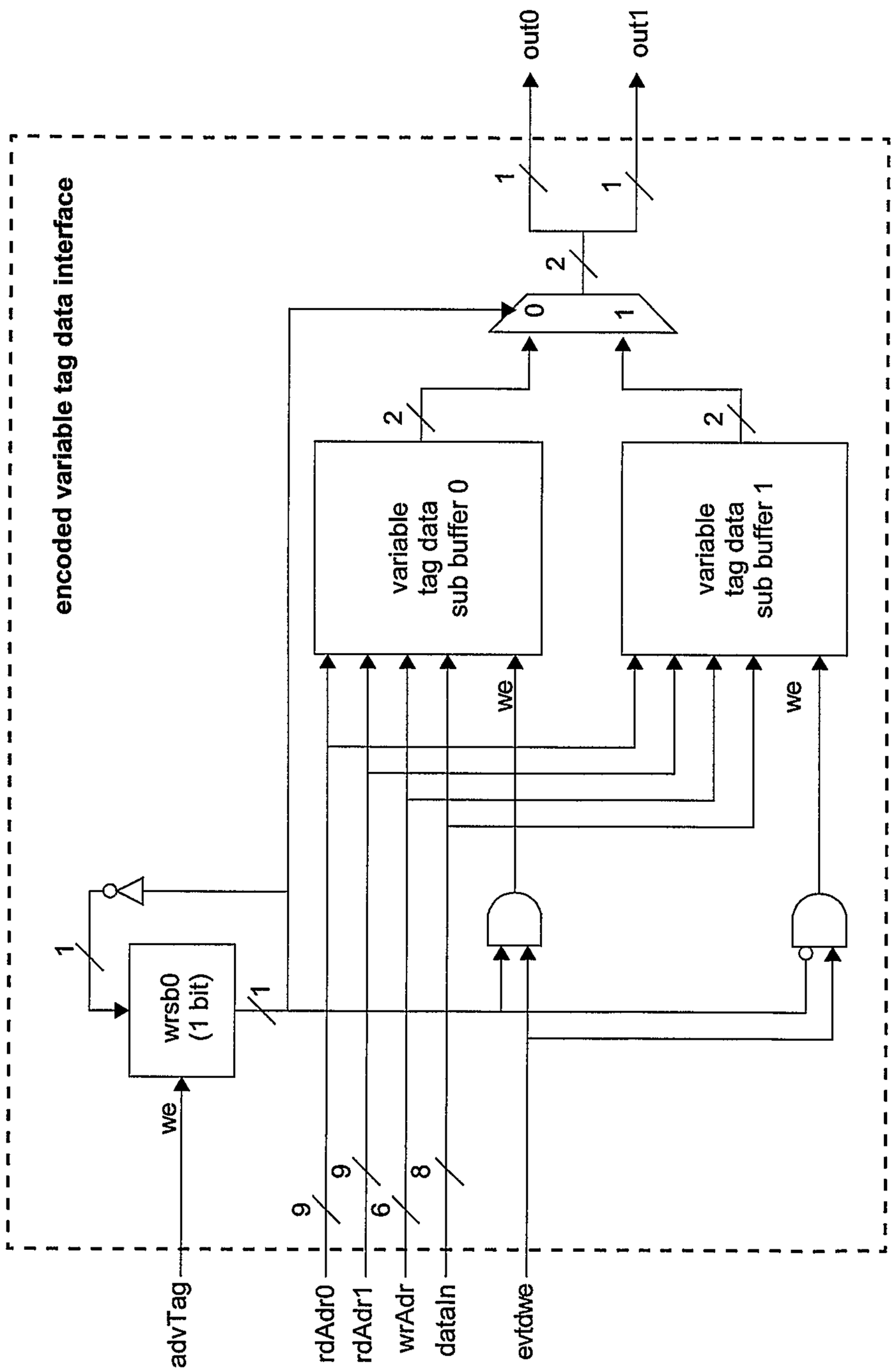

encoded variable tag data interface
out0
out1
1
1
2
0
1
2
2
variable tag data sub buffer 0
variable tag data sub buffer 1
we
we
1
wrsb0 (1 bit)
1
we
9
9
6
8
advTag
rdAdr0
rdAdr1
wrAdr
dataIn
evtdwe

FIG. 219

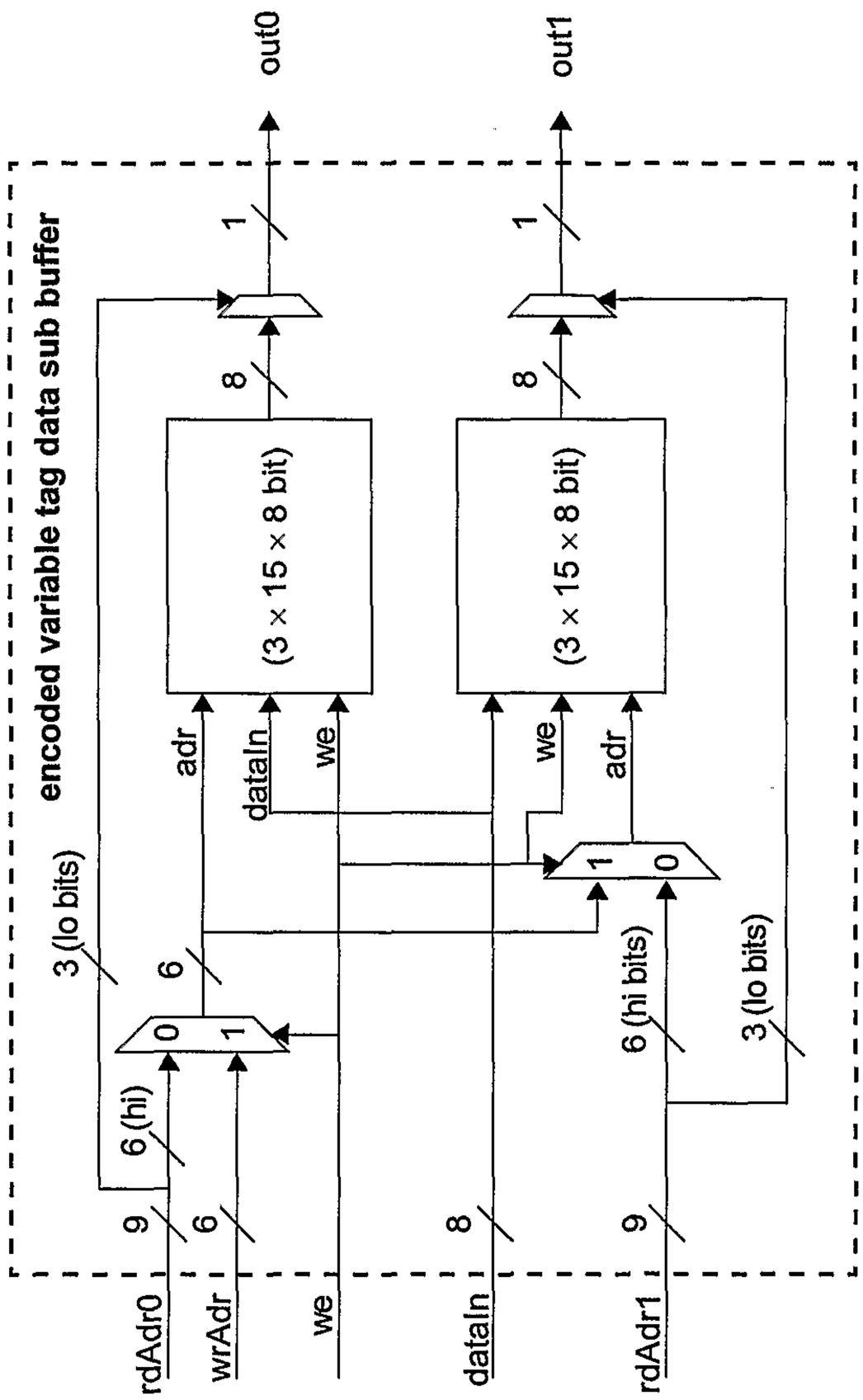

encoded variable tag data sub buffer
out0
out1
1
8
(3 × 15 × 8 bit)
adr
dataIn
we
3 (lo bits)
6
0
1
6 (hi)
6 (hi bits)
3 (lo bits)
9
6
8
rdAdr0
wrAdr
we
dataIn
rdAdr1

FIG. 220

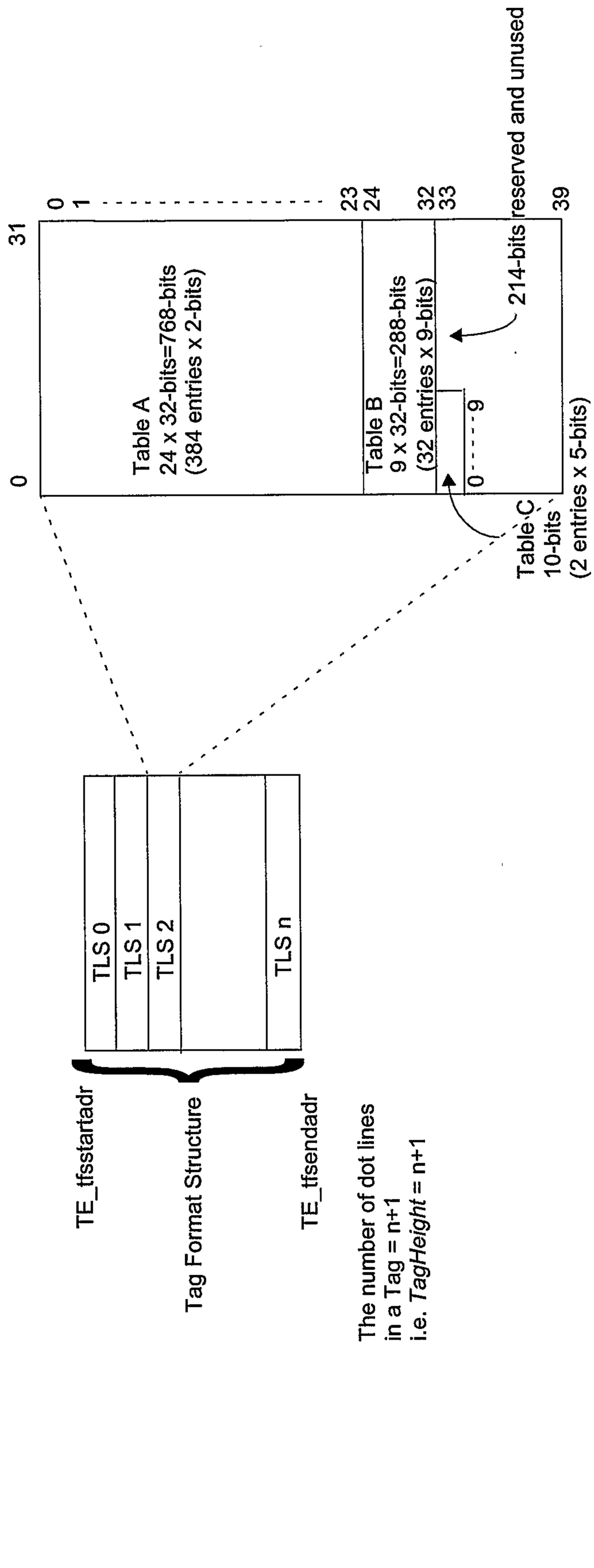

TE_tfsstartadr
TLS 0
TLS 1
TLS 2
TLS n
TE_tfsendadr
Tag Format Structure
The number of dot lines
in a Tag = n+1
i.e. TagHeight = n+1
0
31
Table A
24 x 32-bits=768-bits
(384 entries x 2-bits)
0
1
23
24
Table B
9 x 32-bits=288-bits
(32 entries x 9-bits)
32
33
39
214-bits reserved and unused
0 - - - - - 9
Table C
10-bits
(2 entries x 5-bits)

FIG. 221

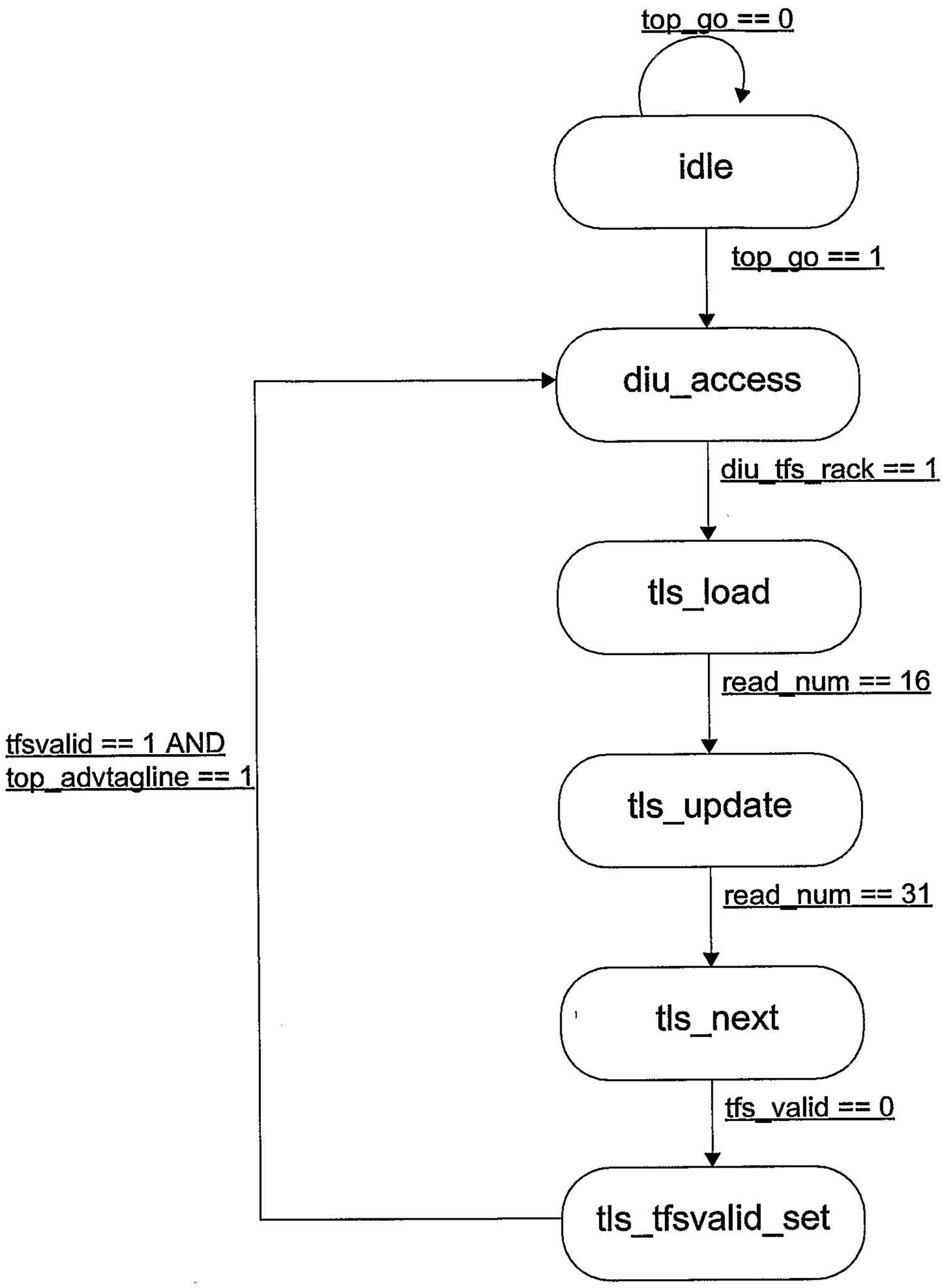

top_go == 0
idle
top_go == 1
diu_access
diu_tfs_rack == 1
tls_load
read_num == 16
tls_update
read_num == 31
tls_next
tfs_valid == 0
tls_tfsvalid_set
tfsvalid == 1 AND
top_advtagline == 1

FIG. 222

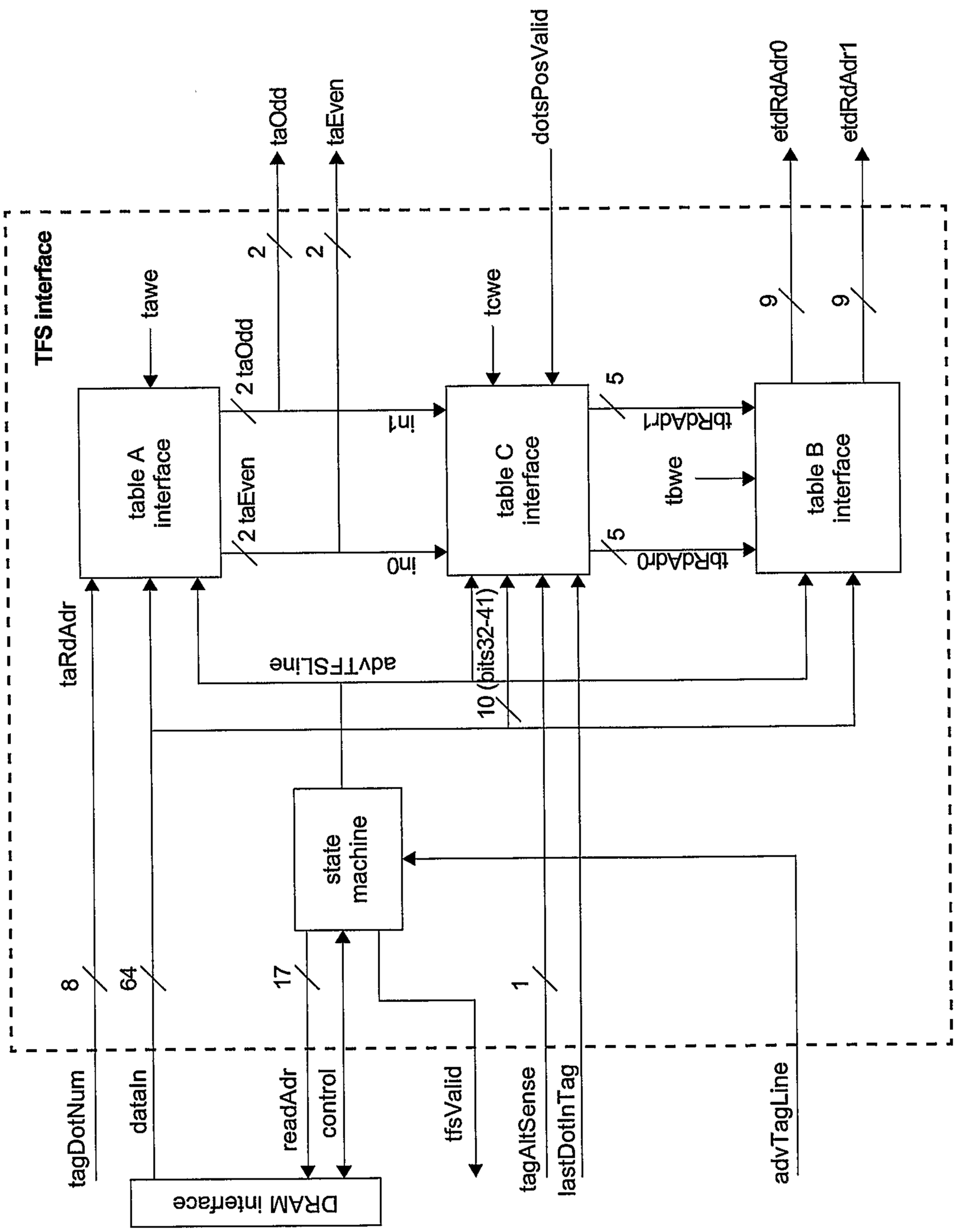

TFS interface
tawe
2 taOdd
taOdd
taEven
dotsPosValid
etdRdAdr0
etdRdAdr1
tcwe
in1
tbRdAdr1
tbwe
table A interface
table C interface
table B interface
2 taEven
in0
tbRdAdr0
taRdAdr
advTFSLine
10 (bits32-41)
state machine
DRAM interface
tagDotNum
dataIn
readAdr
control
tfsValid
tagAltSense
lastDotInTag
advTagLine

FIG. 223

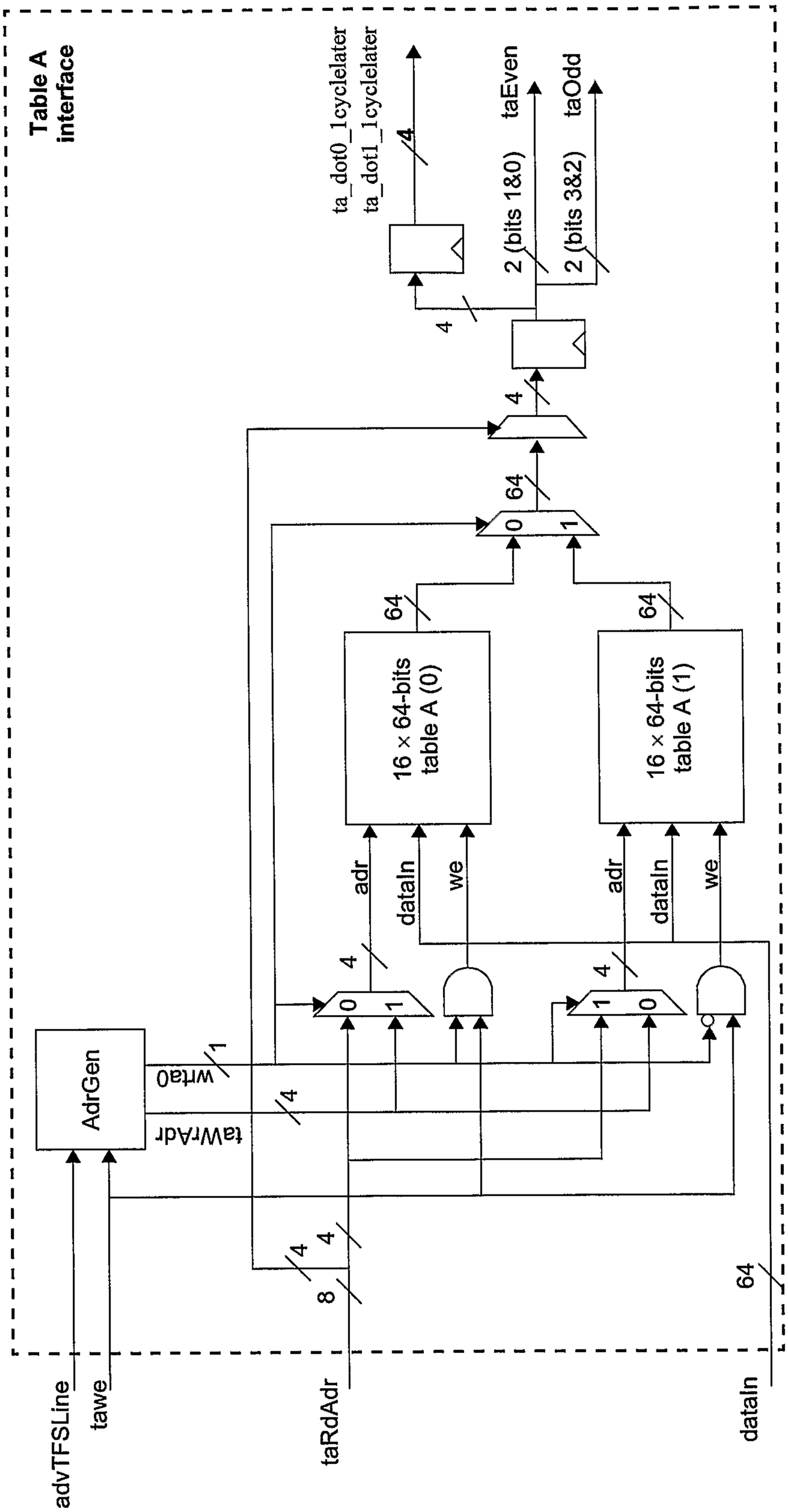


Table A interface
ta_dot0_1cyclelater
ta_dot1_1cyclelater
4
2 (bits 1&0)
taEven
2 (bits 3&2)
taOdd
4
4
64
0
1
64
64
16 × 64-bits table A (0)
16 × 64-bits table A (1)
adr
dataIn
we
adr
dataIn
we
4
0
1
4
1
0
AdrGen
wrta0
1
taWrAdr
4
4
4
8
64
advTFSLine
tawe
taRdAdr
dataIn

FIG. 224

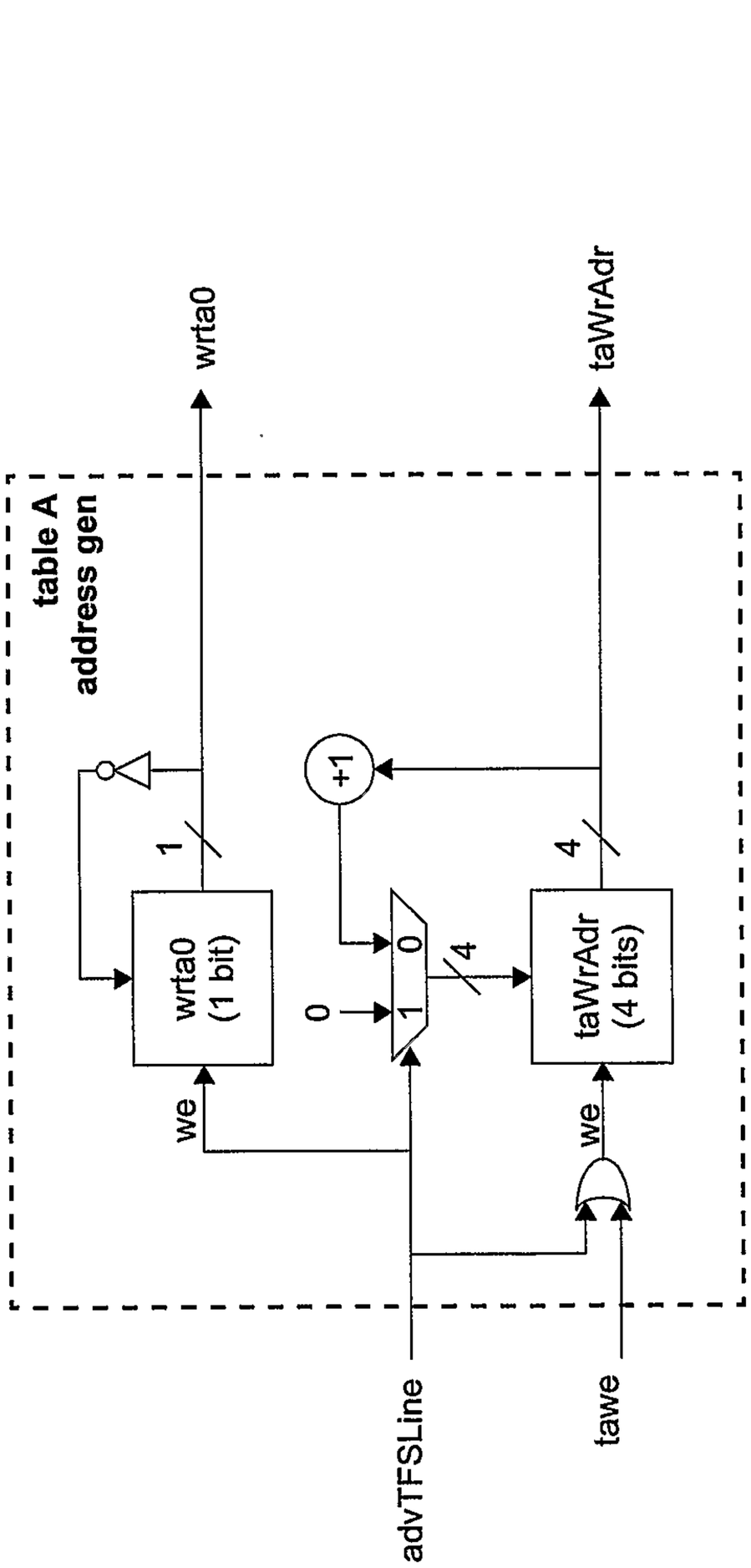


table A address gen
wrta0
taWrAdr
wrta0 (1 bit)
+1
taWrAdr (4 bits)
we
we
advTFSLine
tawe

FIG. 225

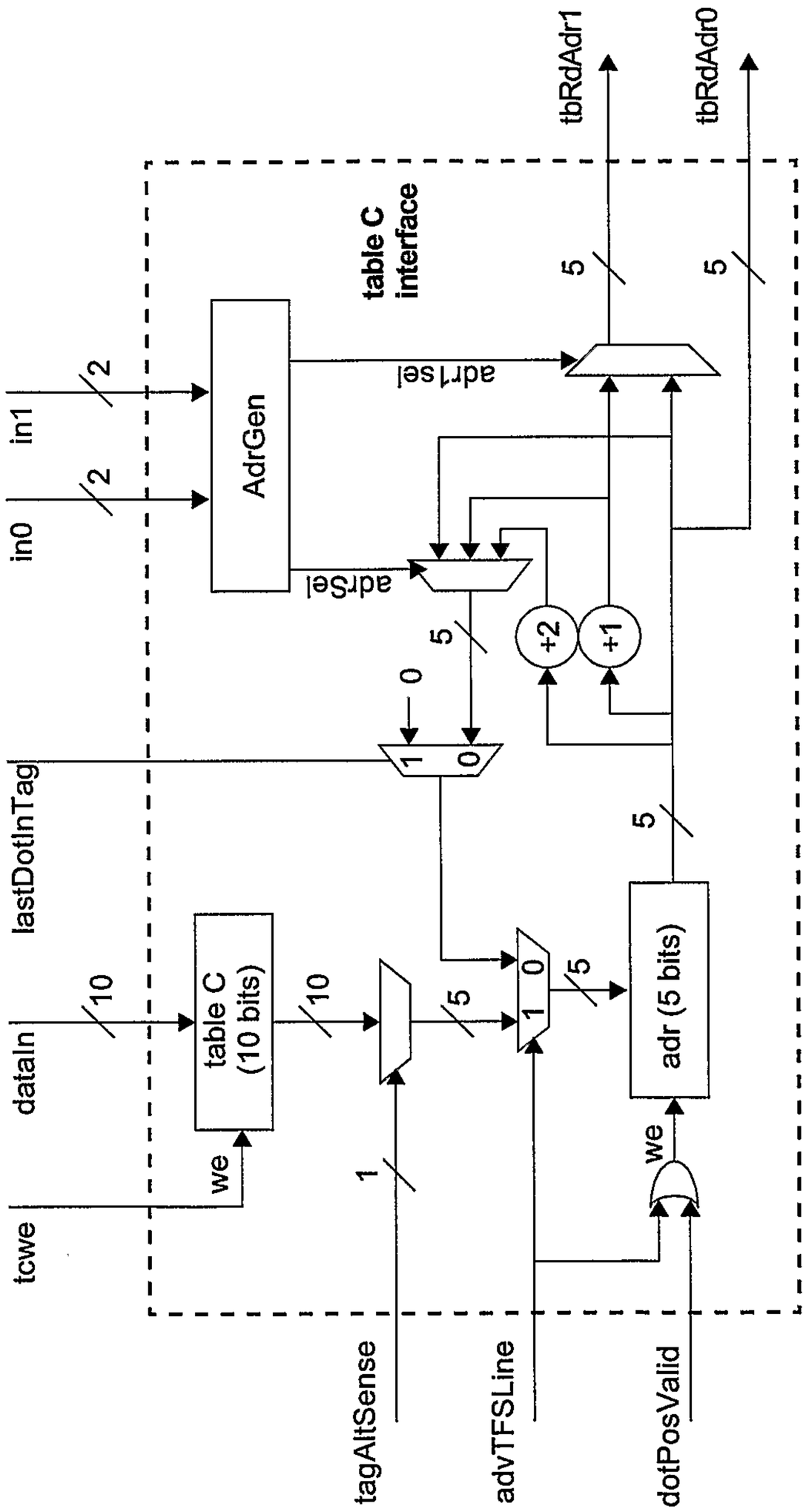

tbRdAdr1
tbRdAdr0
table C interface
5
5
in1
2
in0
2
AdrGen
adr1sel
adrSel
5
0
+2
+1
1
0
lastDotInTag
5
dataIn
10
table C (10 bits)
10
5
1
5
adr (5 bits)
tcwe
we
we
tagAltSense
advTFSLine
dotPosValid

FIG. 226

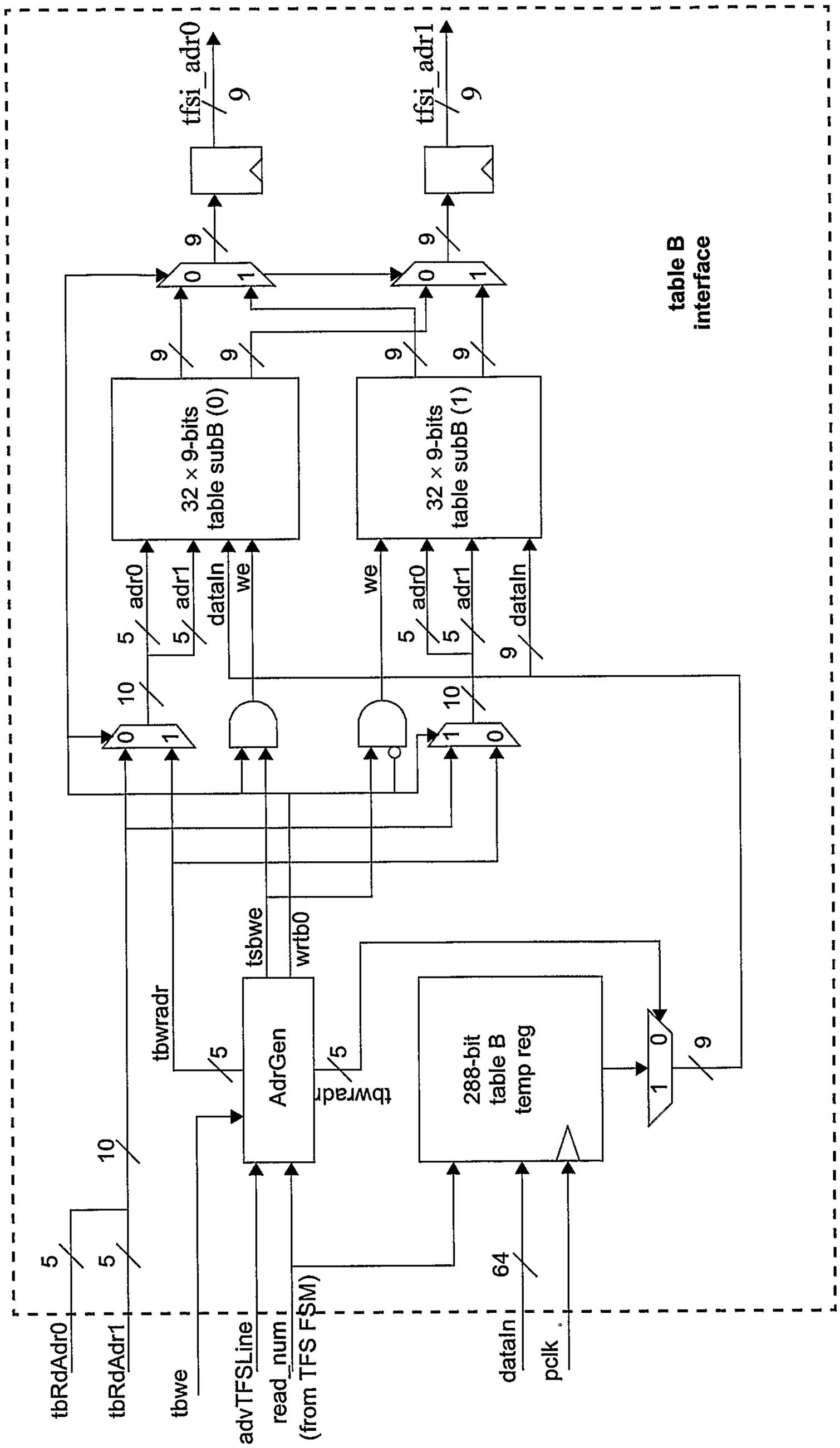


table B
interface
tfsi_adr0
tfsi_adr1
9
32 × 9-bits
table subB (0)
32 × 9-bits
table subB (1)
adr0
adr1
dataIn
we
tsbwe
wrtb0
tbwradr
AdrGen
tbwradr
288-bit
table B
temp reg
tbRdAdr0
tbRdAdr1
tbwe
advTFSLine
read_num
(from TFS FSM)
dataIn
pclk

FIG. 227

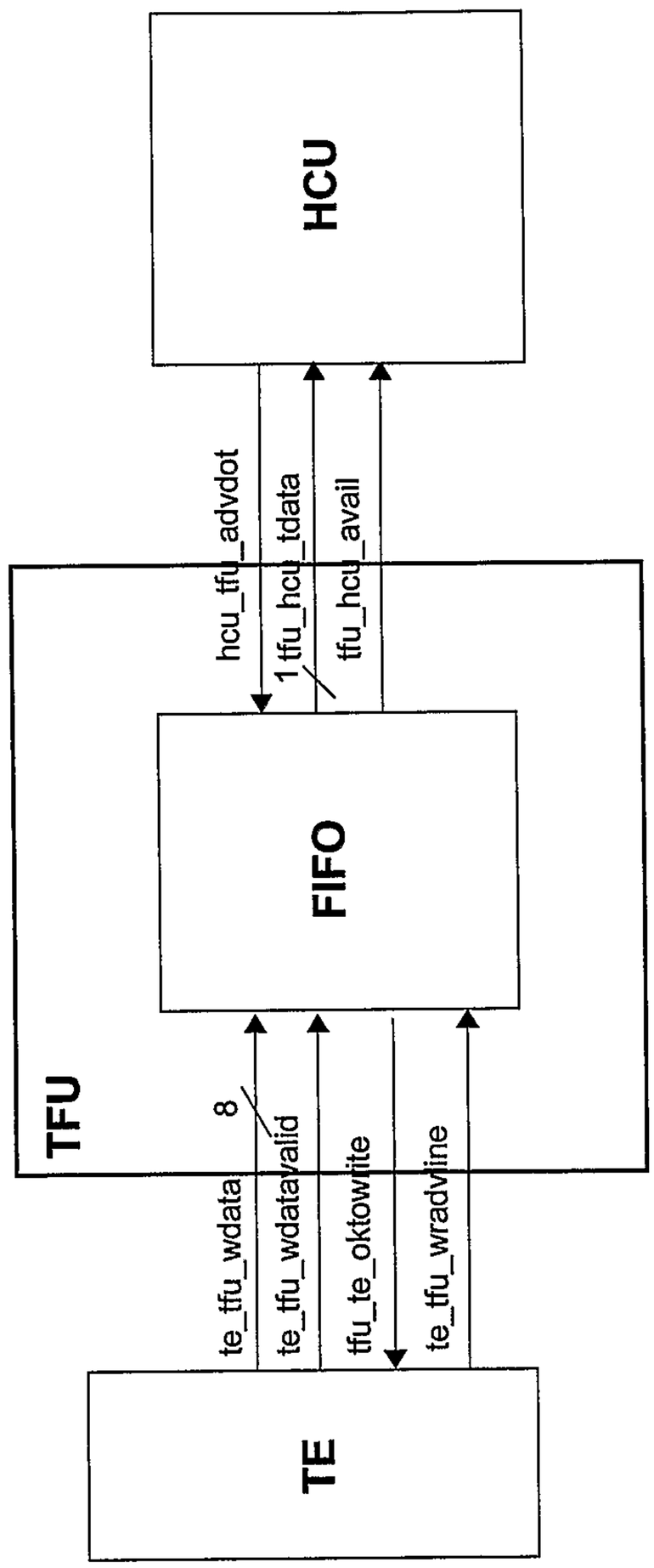


FIG. 228

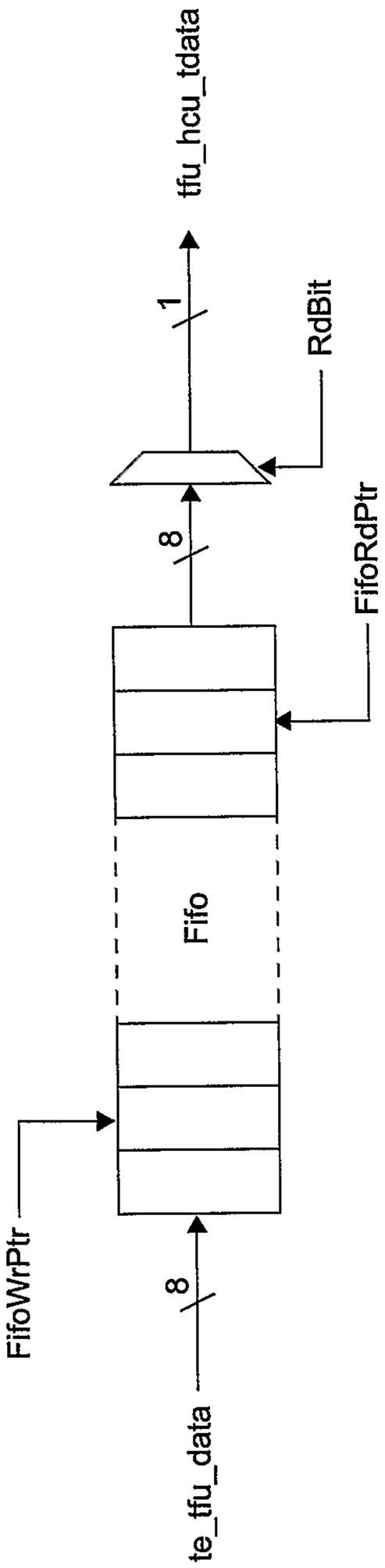


FIG. 229

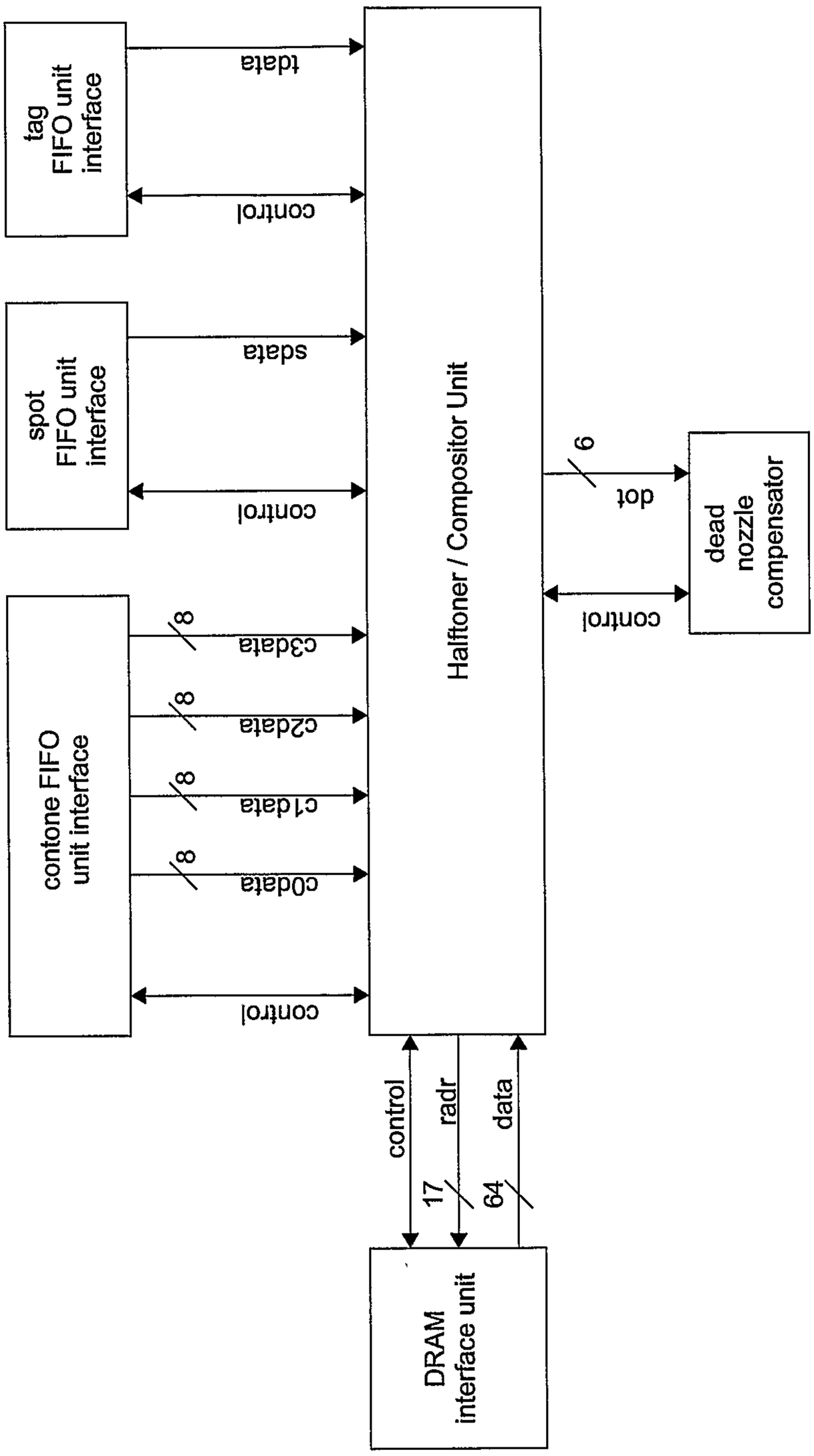

tag FIFO unit interface
tdata
control
spot FIFO unit interface
sdata
control
Halftoner / Compositor Unit
9
dot
dead nozzle compensator
control
contone FIFO unit interface
8
c3data
8
c2data
8
c1data
8
c0data
control
control
radr
data
17
64
DRAM interface unit

FIG. 230

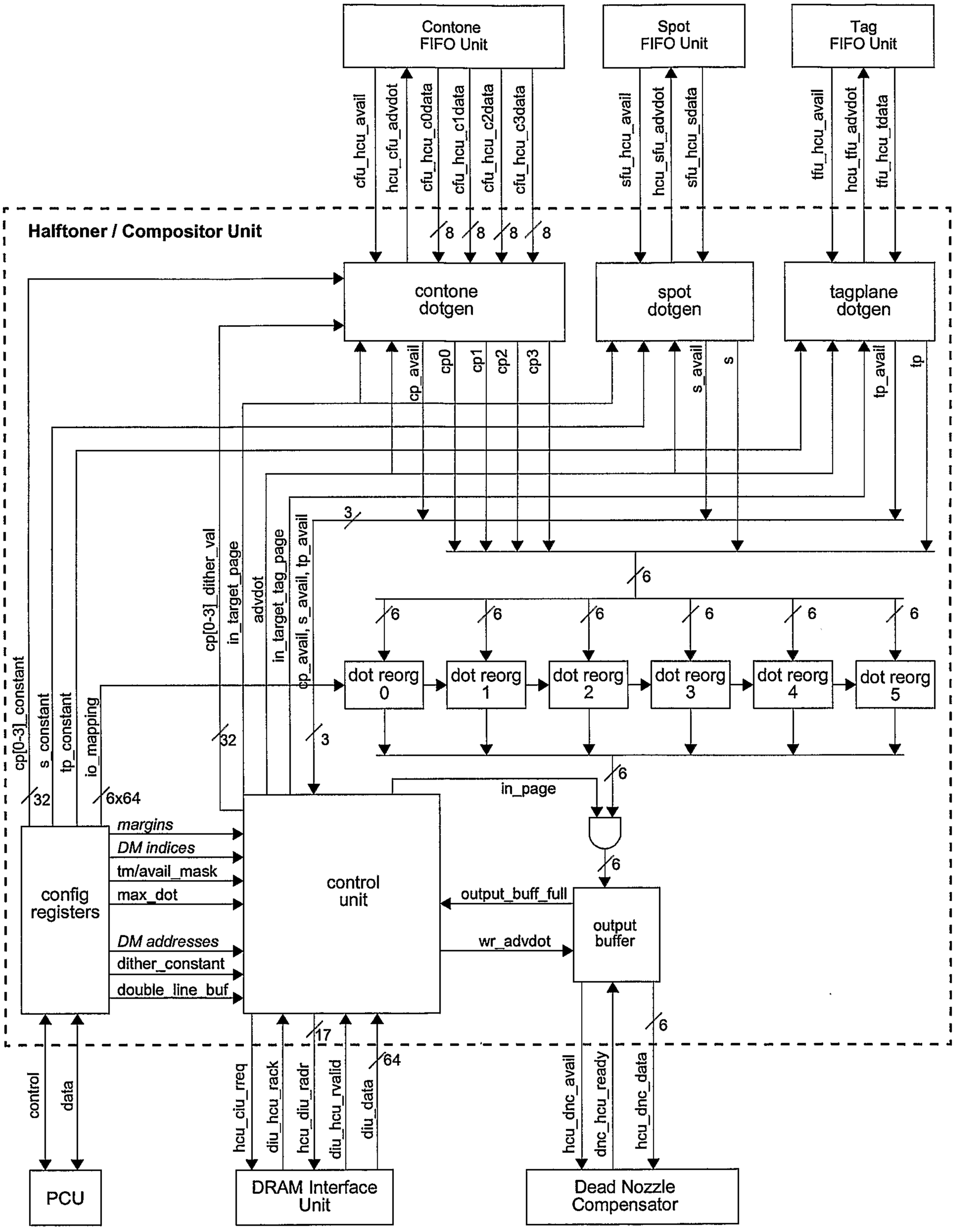
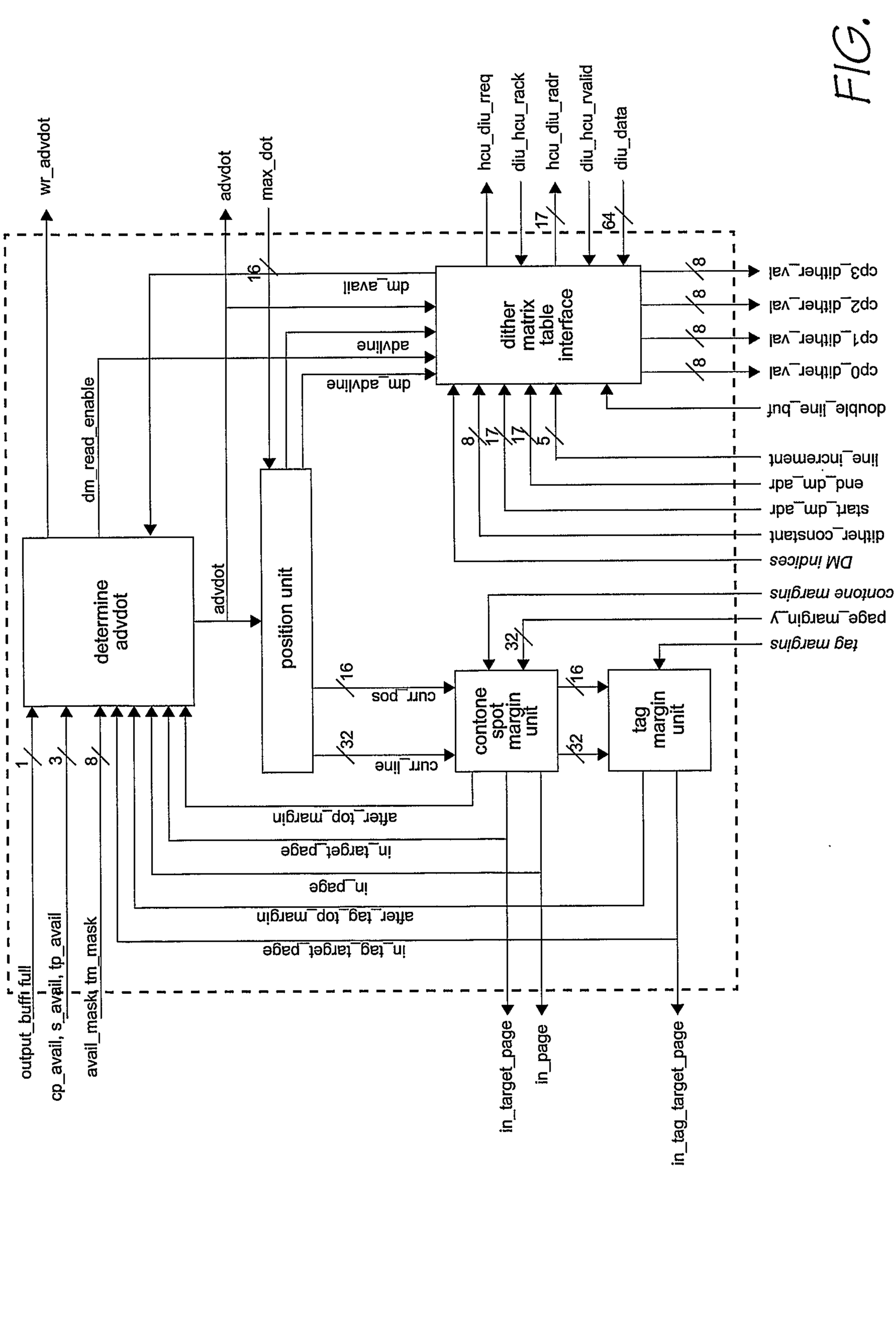

Contone FIFO Unit
Spot FIFO Unit
Tag FIFO Unit
cfu_hcu_avail
hcu_cfu_advdot
cfu_hcu_c0data
cfu_hcu_c1data
cfu_hcu_c2data
cfu_hcu_c3data
sfu_hcu_avail
hcu_sfu_advdot
sfu_hcu_sdata
tfu_hcu_avail
hcu_tfu_advdot
tfu_hcu_tdata
8
Halftoner / Compositor Unit
contone dotgen
spot dotgen
tagplane dotgen
cp_avail
cp0
cp1
cp2
cp3
s_avail
s
tp_avail
tp
3
6
dot reorg 0
dot reorg 1
dot reorg 2
dot reorg 3
dot reorg 4
dot reorg 5
cp[0-3]_constant
s_constant
tp_constant
io_mapping
cp[0-3]_dither_val
in_target_page
advdot
in_target_tag_page
cp_avail, s_avail, tp_avail
32
6x64
in_page
margins
DM indices
tm/avail_mask
max_dot
DM addresses
dither_constant
double_line_buf
config registers
control unit
output_buff_full
wr_advdot
output buffer
control
data
PCU
hcu_ciu_rreq
diu_hcu_rack
hcu_diu_radr
diu_hcu_rvalid
diu_data
17
64
DRAM Interface Unit
hcu_dnc_avail
dnc_hcu_ready
hcu_dnc_data
Dead Nozzle Compensator

FIG. 231

FIG. 232


wr_advdot
advdot
max_dot
hcu_diu_rreq
diu_hcu_rack
hcu_diu_radr
diu_hcu_rvalid
diu_data
dm_read_enable
dm_avail
advline
dm_advline
dither matrix table interface
cp3_dither_val
cp2_dither_val
cp1_dither_val
cp0_dither_val
double_line_buf
line_increment
end_dm_adr
start_dm_adr
dither_constant
DM indices
contone margins
page_margin_y
tag margins
determine advdot
position unit
curr_pos
curr_line
contone spot margin unit
tag margin unit
after_top_margin
in_target_page
in_page
after_tag_top_margin
in_tag_target_page
output_buffi full
cp_avail, s_avail, tp_avail
avail_mask, tm_mask
in_target_page
in_page
in_tag_target_page

233

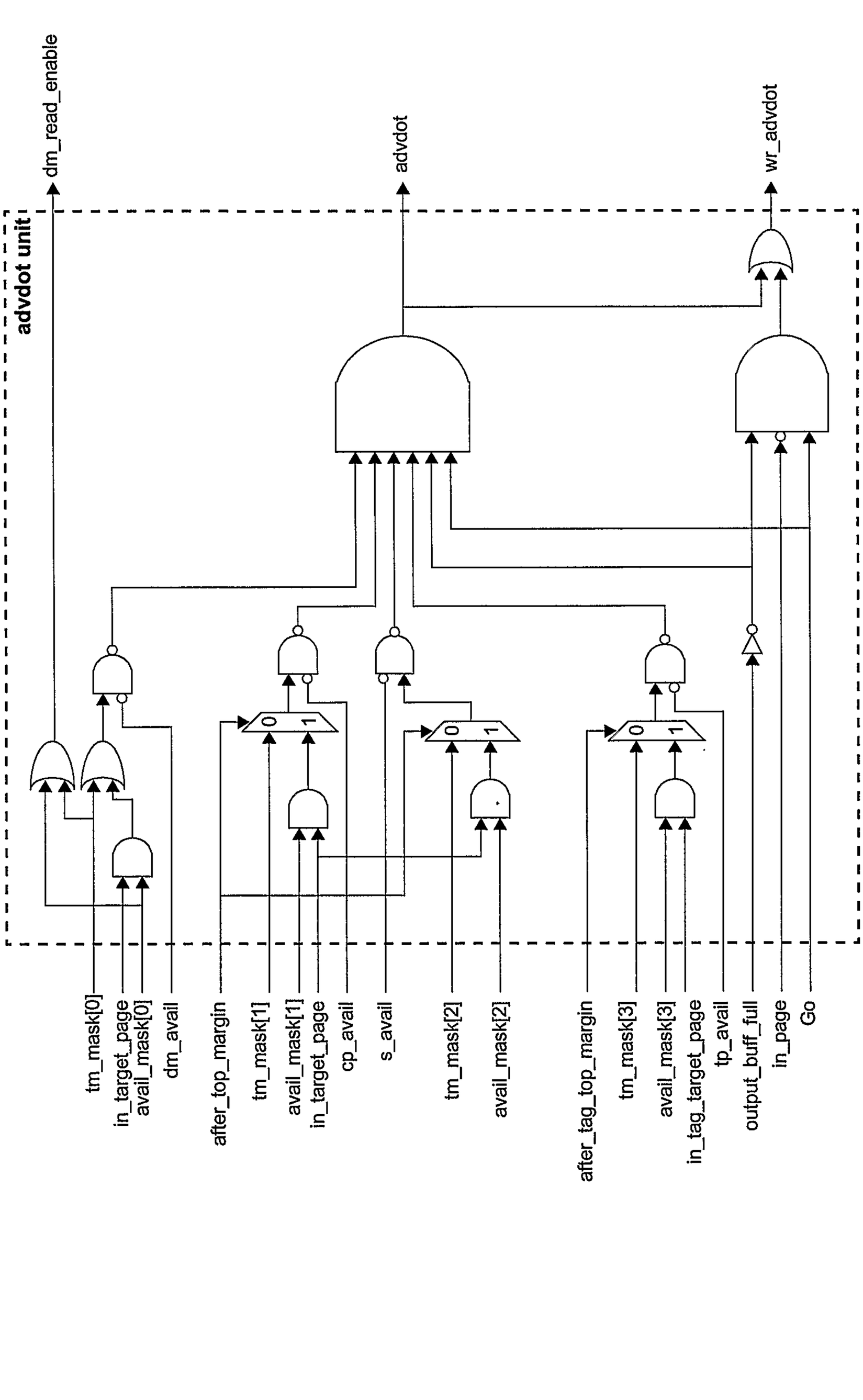

advdot unit
dm_read_enable
advdot
wr_advdot
tm_mask[0]
in_target_page
avail_mask[0]
dm_avail
after_top_margin
tm_mask[1]
avail_mask[1]
in_target_page
cp_avail
s_avail
tm_mask[2]
avail_mask[2]
after_tag_top_margin
tm_mask[3]
avail_mask[3]
in_tag_target_page
tp_avail
output_buff_full
in_page
Go
0
1

FIG. 233

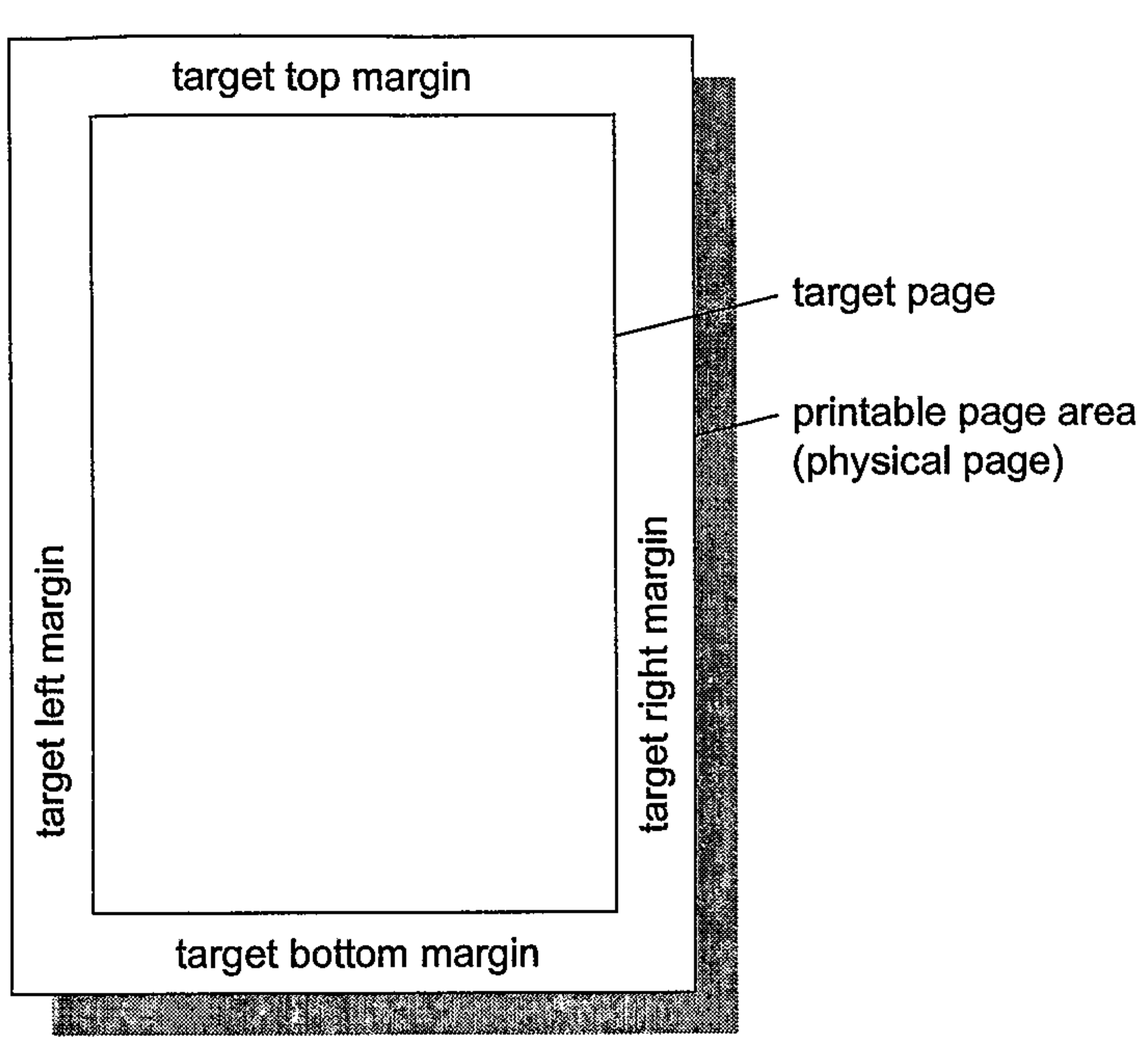

target top margin
target page
printable page area
(physical page)
target left margin
target right margin
target bottom margin

FIG. 234

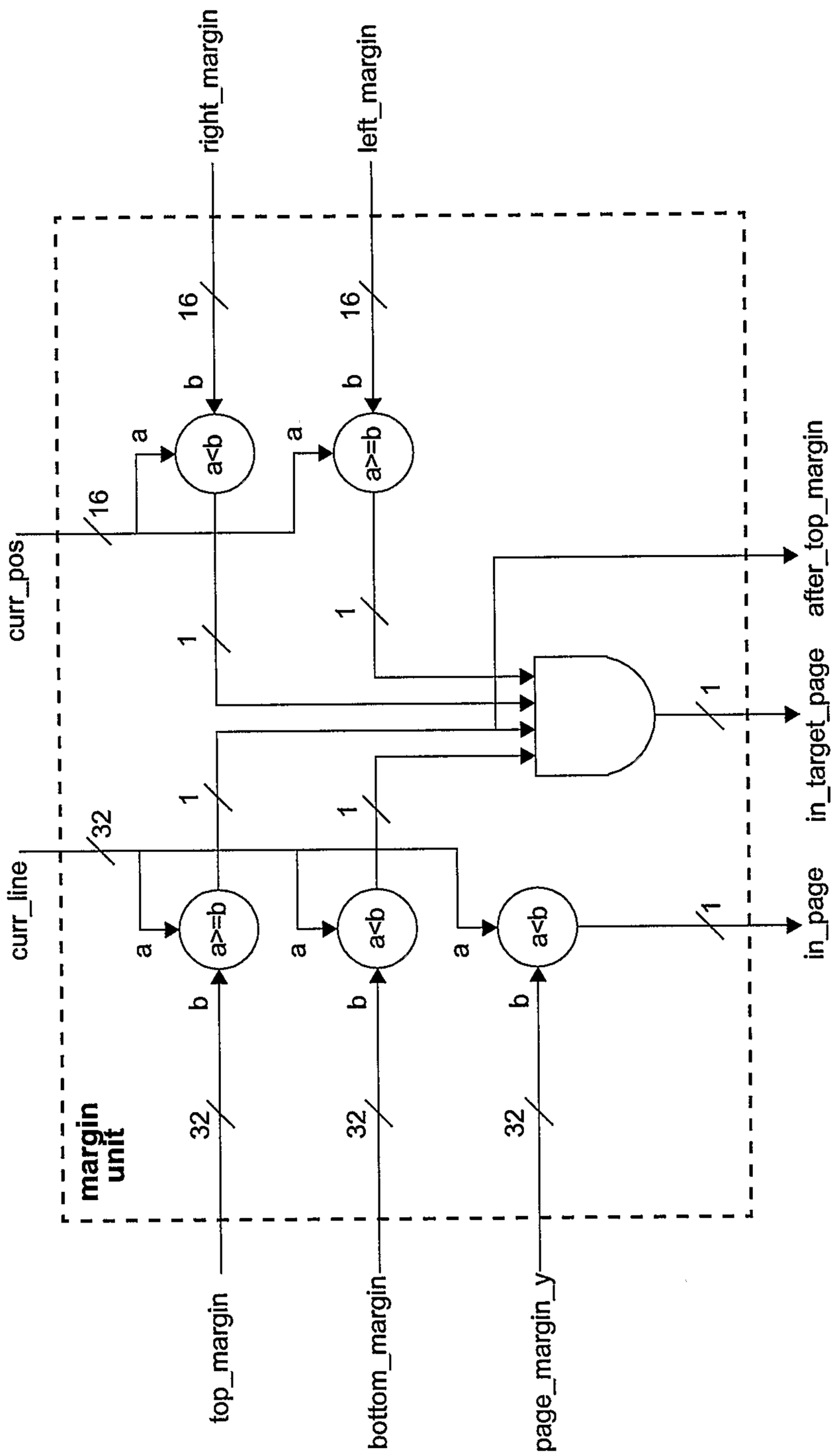

right_margin
left_margin
curr_pos
16
a
b
a<b
a>=b
1
after_top_margin
in_target_page
curr_line
32
in_page
margin unit
top_margin
bottom_margin
page_margin_y

FIG. 235

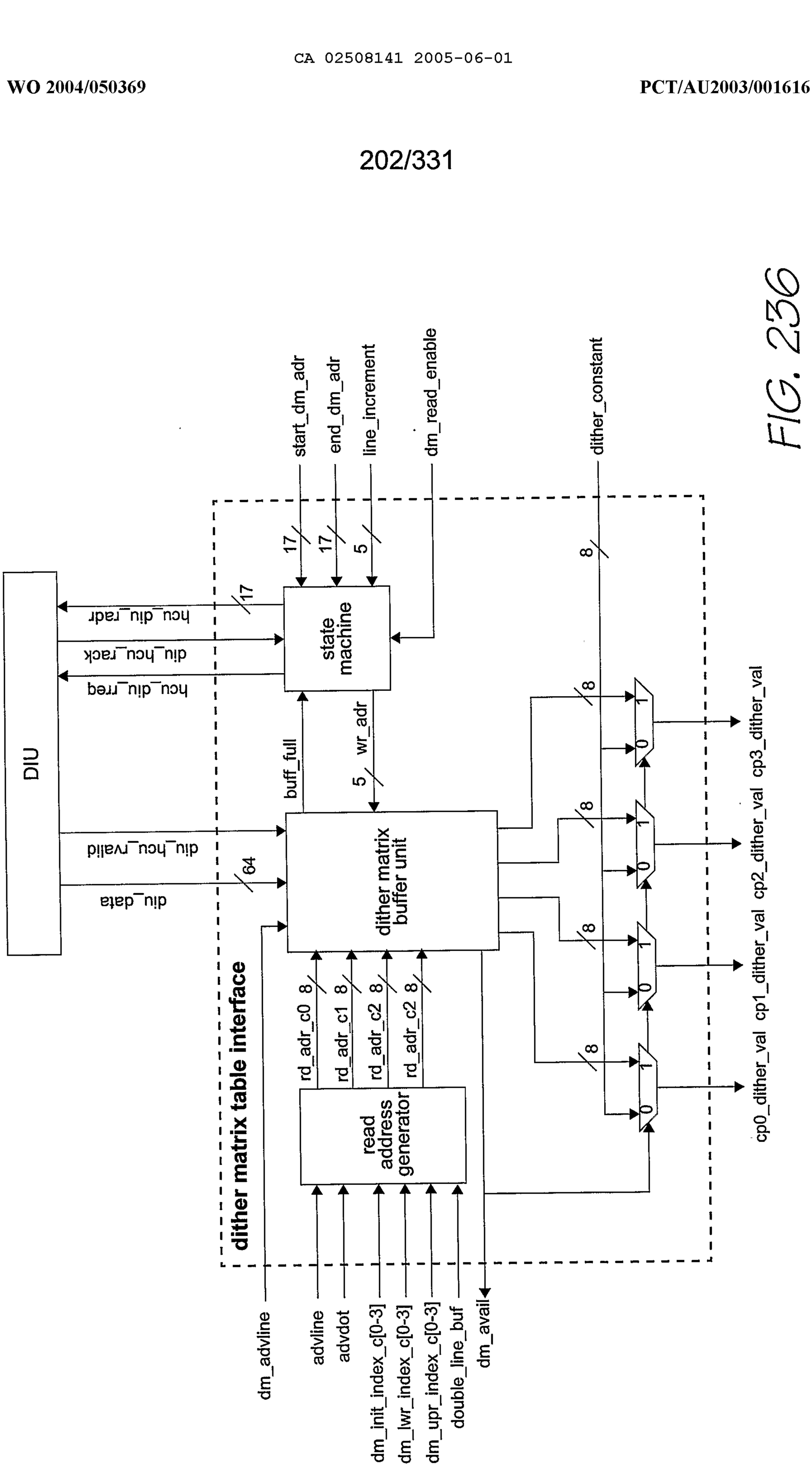

DIU
dither matrix table interface
hcu_diu_rreq
diu_hcu_rack
hcu_diu_radr
diu_hcu_rvalid
diu_data
64
17
state machine
dither matrix buffer unit
buff_full
wr_adr
5
17
17
5
8
rd_adr_c0 8
rd_adr_c1 8
rd_adr_c2 8
rd_adr_c2 8
read address generator
dm_advline
advline
advdot
dm_init_index_c[0-3]
dm_lwr_index_c[0-3]
dm_upr_index_c[0-3]
double_line_buf
dm_avail
start_dm_adr
end_dm_adr
line_increment
dm_read_enable
dither_constant
cp0_dither_val
cp1_dither_val
cp2_dither_val
cp3_dither_val

FIG. 236

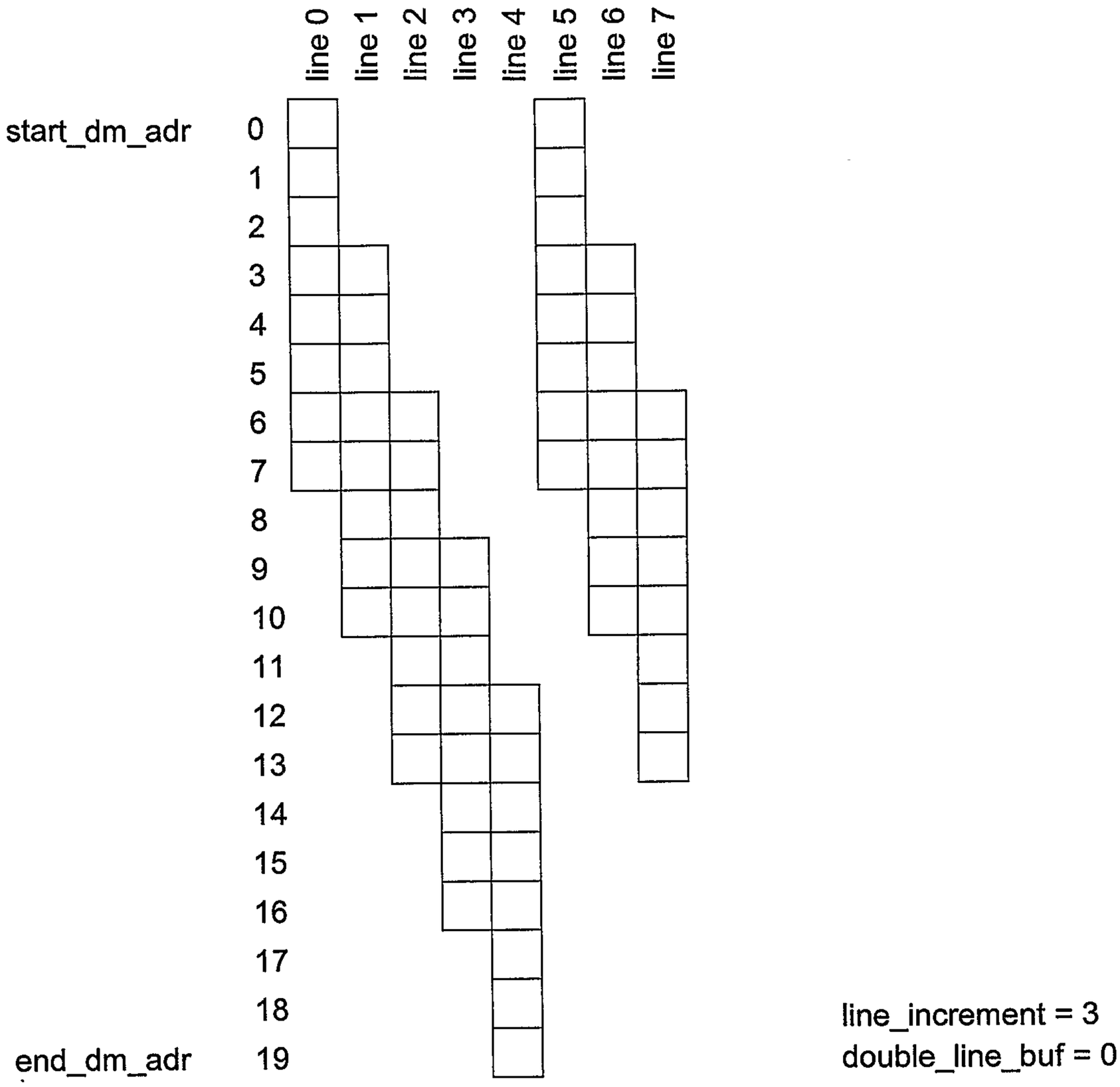

line 0
line 1
line 2
line 3
line 4
line 5
line 6
line 7
start_dm_adr
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
end_dm_adr
19
20
line_increment = 3
double_line_buf = 0

FIG. 237

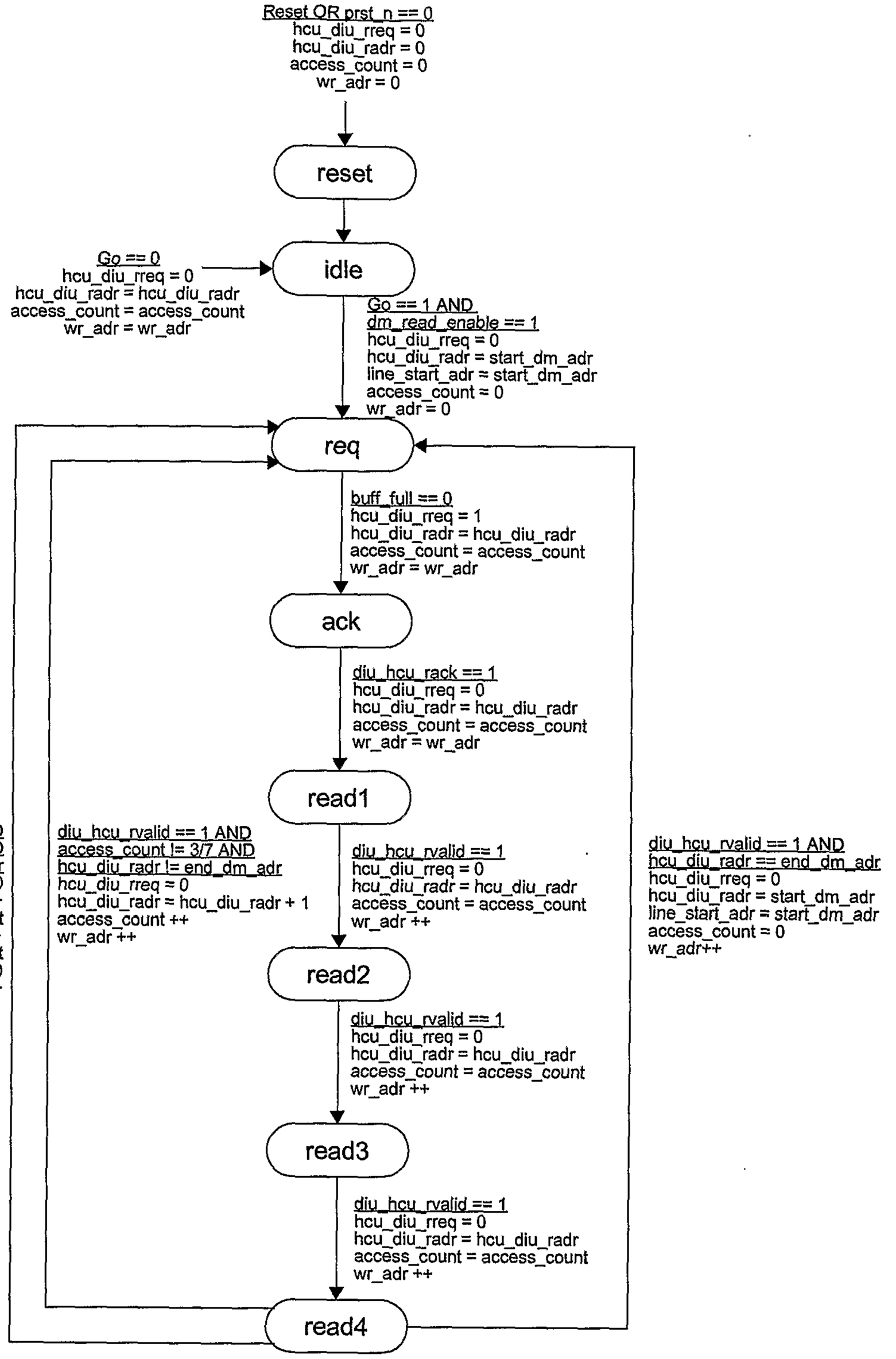

Reset OR prst_n == 0
hcu_diu_rreq = 0
hcu_diu_radr = 0
access_count = 0
wr_adr = 0
reset
idle
Go == 0
hcu_diu_rreq = 0
hcu_diu_radr = hcu_diu_radr
access_count = access_count
wr_adr = wr_adr
Go == 1 AND
dm_read_enable == 1
hcu_diu_rreq = 0
hcu_diu_radr = start_dm_adr
line_start_adr = start_dm_adr
access_count = 0
wr_adr = 0
req
buff_full == 0
hcu_diu_rreq = 1
hcu_diu_radr = hcu_diu_radr
access_count = access_count
wr_adr = wr_adr
ack
diu_hcu_rack == 1
hcu_diu_rreq = 0
hcu_diu_radr = hcu_diu_radr
access_count = access_count
wr_adr = wr_adr
read1
diu_hcu_rvalid == 1 AND
access_count == 3/7 AND
hcu_diu_radr != end_dm_adr
hcu_diu_rreq = 0
hcu_diu_radr = line_start_adr +
line_increment
line_start_adr = line_start_adr +
line_increment
access_count = 0
wr_adr++
diu_hcu_rvalid == 1 AND
access_count != 3/7 AND
hcu_diu_radr != end_dm_adr
hcu_diu_rreq = 0
hcu_diu_radr = hcu_diu_radr + 1
access_count ++
wr_adr ++
diu_hcu_rvalid == 1
hcu_diu_rreq = 0
hcu_diu_radr = hcu_diu_radr
access_count = access_count
wr_adr ++
diu_hcu_rvalid == 1 AND
hcu_diu_radr == end_dm_adr
hcu_diu_rreq = 0
hcu_diu_radr = start_dm_adr
line_start_adr = start_dm_adr
access_count = 0
wr_adr++
read2
diu_hcu_rvalid == 1
hcu_diu_rreq = 0
hcu_diu_radr = hcu_diu_radr
access_count = access_count
wr_adr ++
read3
diu_hcu_rvalid == 1
hcu_diu_rreq = 0
hcu_diu_radr = hcu_diu_radr
access_count = access_count
wr_adr ++
read4

FIG. 238

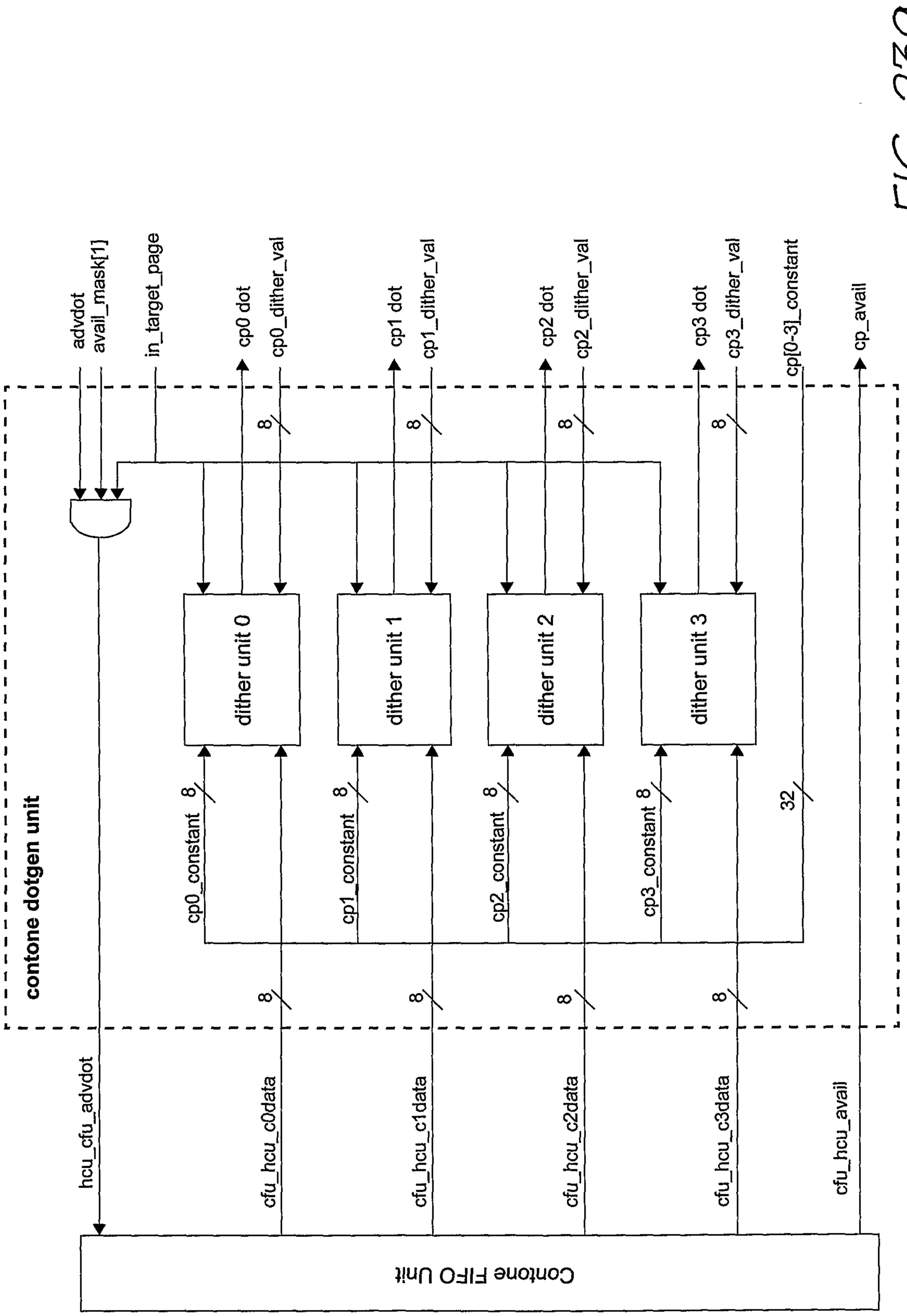

contone dotgen unit
advdot
avail_mask[1]
in_target_page
cp0 dot
cp0_dither_val
cp1 dot
cp1_dither_val
cp2 dot
cp2_dither_val
cp3 dot
cp3_dither_val
cp[0-3]_constant
cp_avail
8
dither unit 0
dither unit 1
dither unit 2
dither unit 3
cp0_constant
cp1_constant
cp2_constant
cp3_constant
32
hcu_cfu_advdot
cfu_hcu_c0data
cfu_hcu_c1data
cfu_hcu_c2data
cfu_hcu_c3data
cfu_hcu_avail
Contone FIFO Unit

FIG. 239

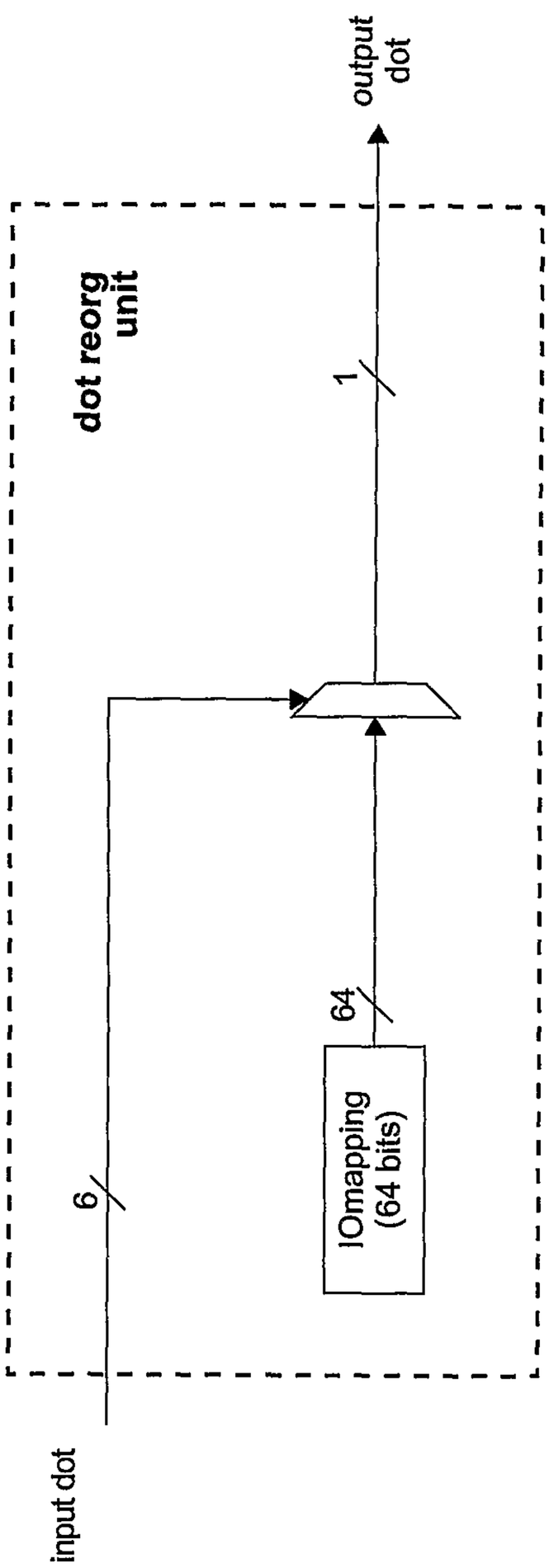
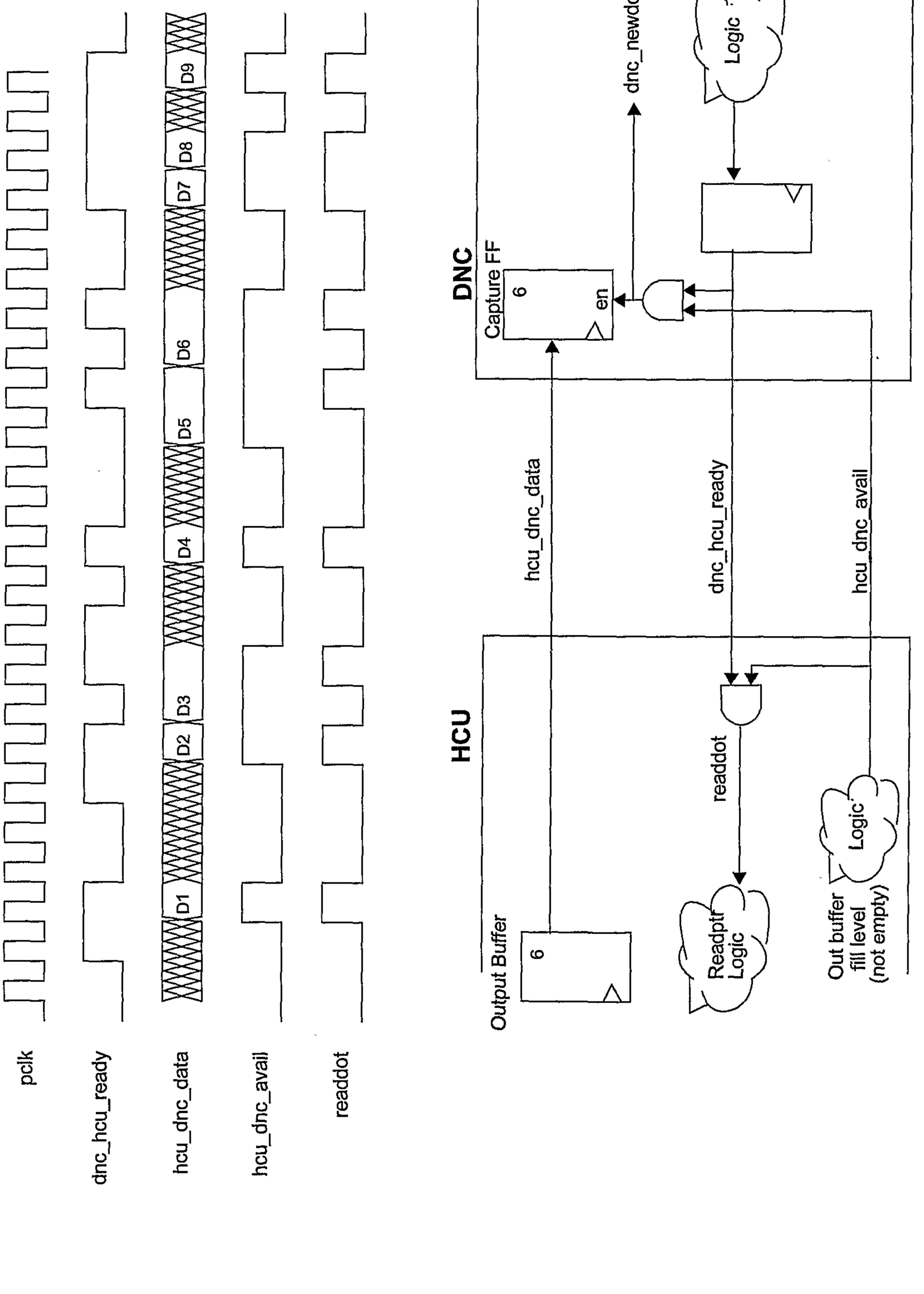


FIG. 240

FIG. 241


pclk
dnc_hcu_ready
hcu_dnc_data
hcu_dnc_avail
readdot
D1
D2
D3
D4
D5
D6
D7
D8
D9
DNC
Capture FF
6
en
dnc_newdot
Logic
HCU
hcu_dnc_data
dnc_hcu_ready
hcu_dnc_avail
readdot
Logic
Output Buffer
6
Readptr Logic
Out buffer fill level (not empty)

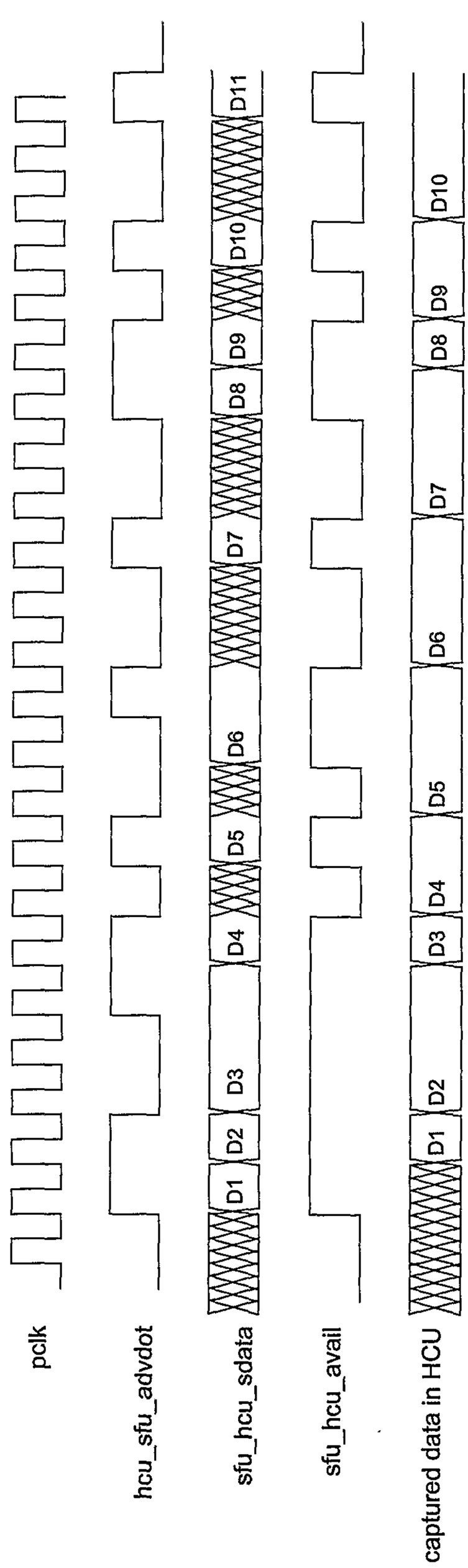

pclk
hcu_sfu_advdot
sfu_hcu_sdata
sfu_hcu_avail
captured data in HCU
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11

FIG. 242

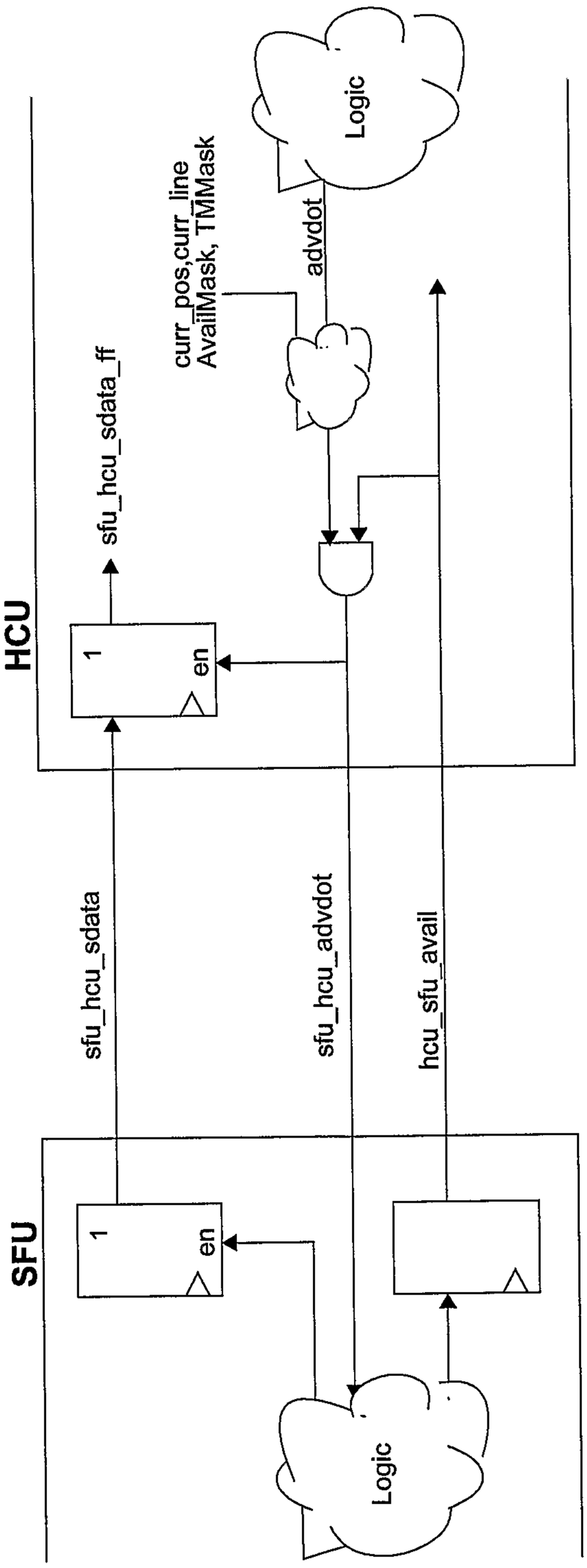

HCU
SFU
sfu_hcu_sdata_ff
curr_pos,curr_line
AvailMask, TMMask
Logic
advdot
1
en
sfu_hcu_sdata
sfu_hcu_advdot
hcu_sfu_avail
1
en
Logic

FIG. 243

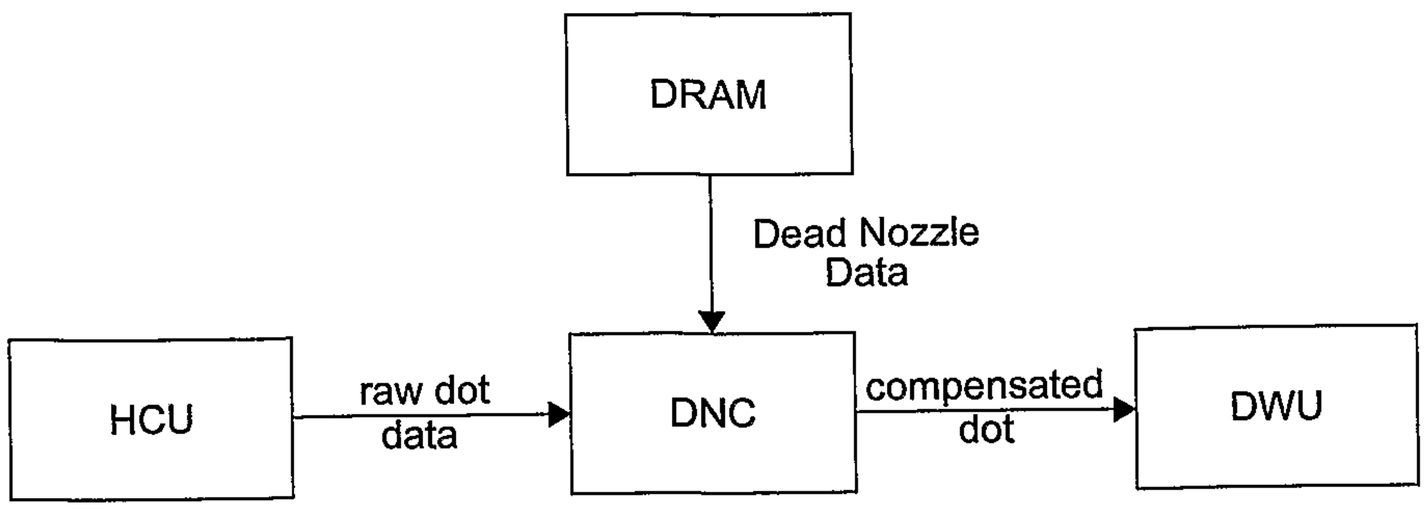


FIG. 244

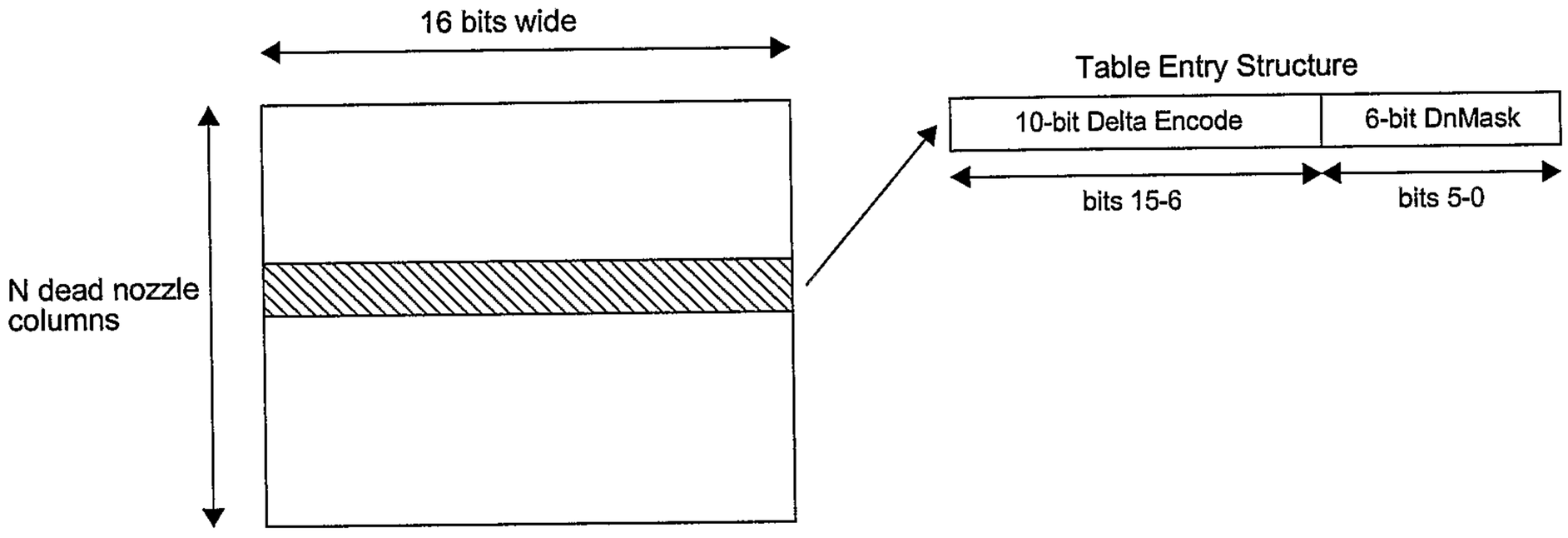


FIG. 245

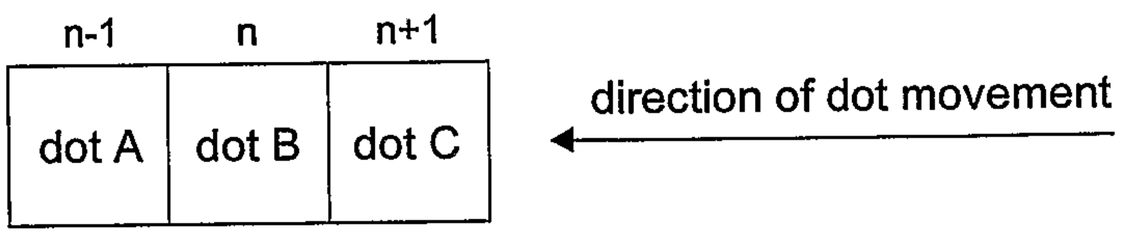


FIG. 246

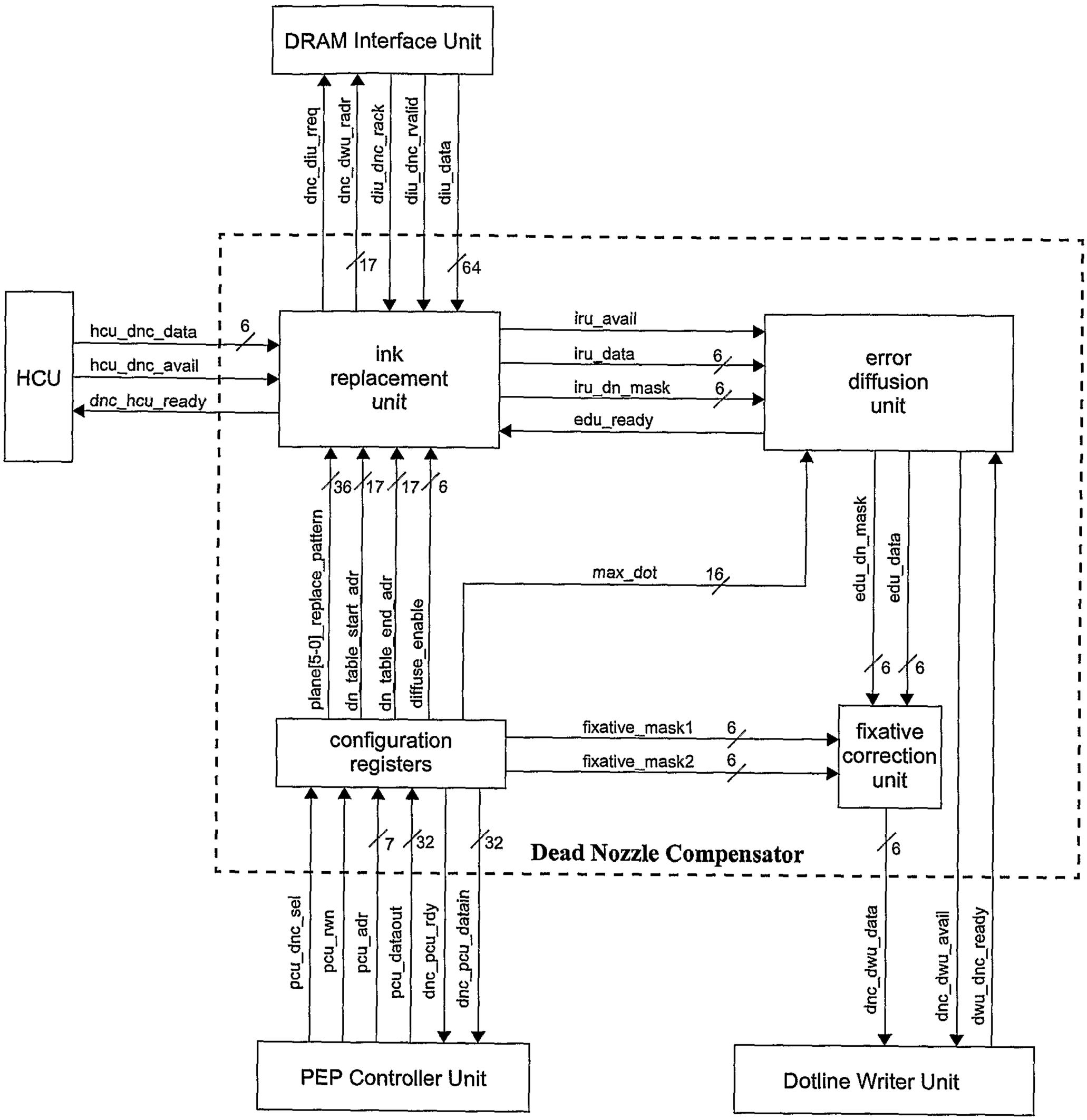

DRAM Interface Unit
dnc_diu_rreq
dnc_dwu_radr
diu_dnc_rack
diu_dnc_rvalid
diu_data
17
64
HCU
hcu_dnc_data
6
hcu_dnc_avail
dnc_hcu_ready
ink replacement unit
iru_avail
iru_data
6
iru_dn_mask
6
edu_ready
error diffusion unit
36
17
17
6
plane[5-0]_replace_pattern
dn_table_start_adr
dn_table_end_adr
diffuse_enable
max_dot
16
edu_dn_mask
edu_data
6
6
configuration registers
fixative_mask1
6
fixative_mask2
6
fixative correction unit
7
32
32
Dead Nozzle Compensator
6
pcu_dnc_sel
pcu_rwn
pcu_adr
pcu_dataout
dnc_pcu_rdy
dnc_pcu_datain
dnc_dwu_data
dnc_dwu_avail
dwu_dnc_ready
PEP Controller Unit
Dotline Writer Unit

FIG. 247

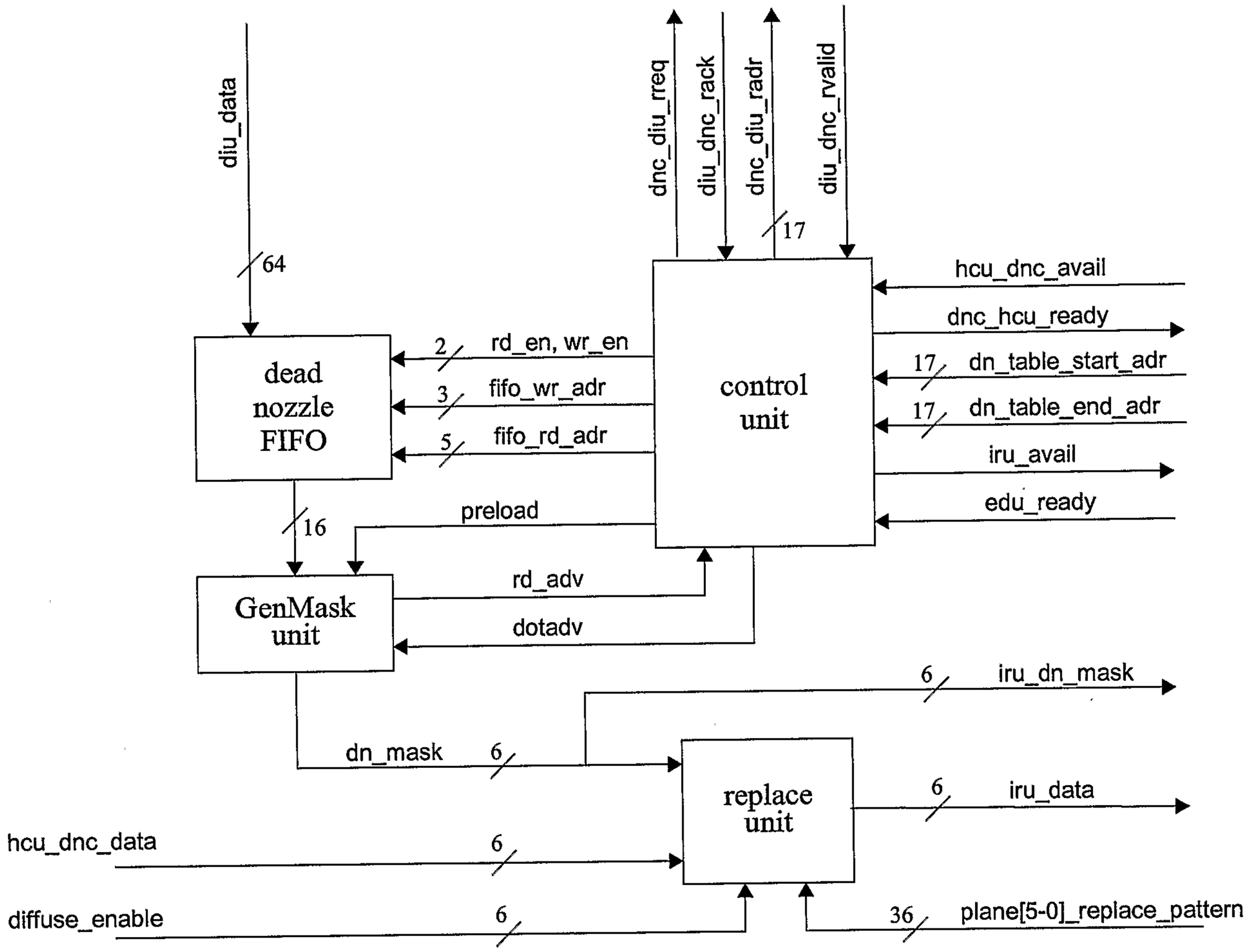

diu_data
64
dnc_diu_rreq
diu_dnc_rack
dnc_diu_radr
diu_dnc_rvalid
17
hcu_dnc_avail
dnc_hcu_ready
17
dn_table_start_adr
17
dn_table_end_adr
iru_avail
edu_ready
dead nozzle FIFO
2
rd_en, wr_en
3
fifo_wr_adr
5
fifo_rd_adr
control unit
16
preload
GenMask unit
rd_adv
dotadv
6
iru_dn_mask
dn_mask
6
replace unit
6
iru_data
hcu_dnc_data
6
diffuse_enable
6
36
plane[5-0]_replace_pattern

FIG. 248

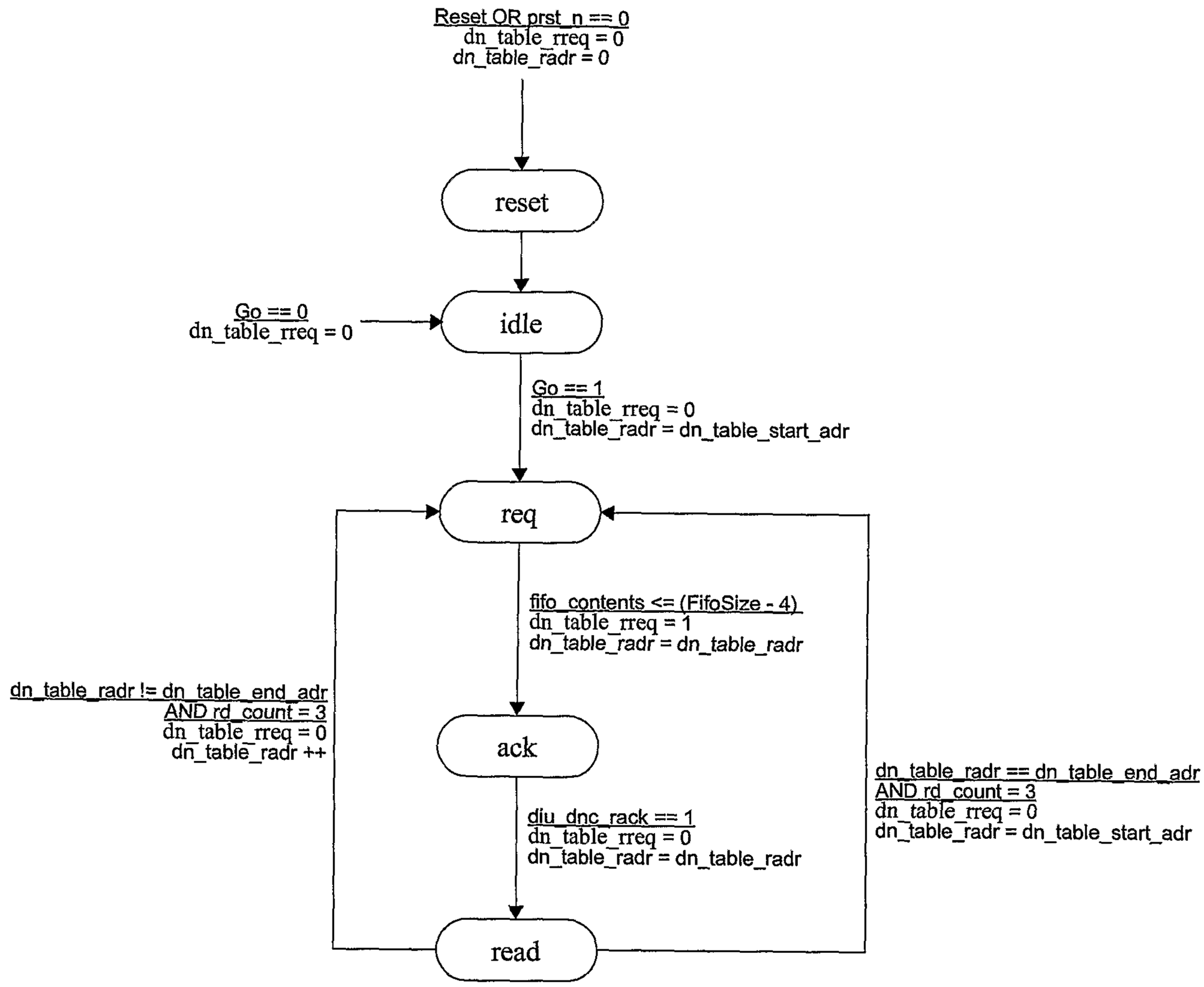

Reset OR prst_n == 0
dn_table_rreq = 0
dn_table_radr = 0
reset
Go == 0
dn_table_rreq = 0
idle
Go == 1
dn_table_rreq = 0
dn_table_radr = dn_table_start_adr
req
fifo_contents <= (FifoSize - 4)
dn_table_rreq = 1
dn_table_radr = dn_table_radr
dn_table_radr != dn_table_end_adr
AND rd_count = 3
dn_table_rreq = 0
dn_table_radr ++
ack
dn_table_radr == dn_table_end_adr
AND rd_count = 3
dn_table_rreq = 0
dn_table_radr = dn_table_start_adr
diu_dnc_rack == 1
dn_table_rreq = 0
dn_table_radr = dn_table_radr
read

FIG. 249

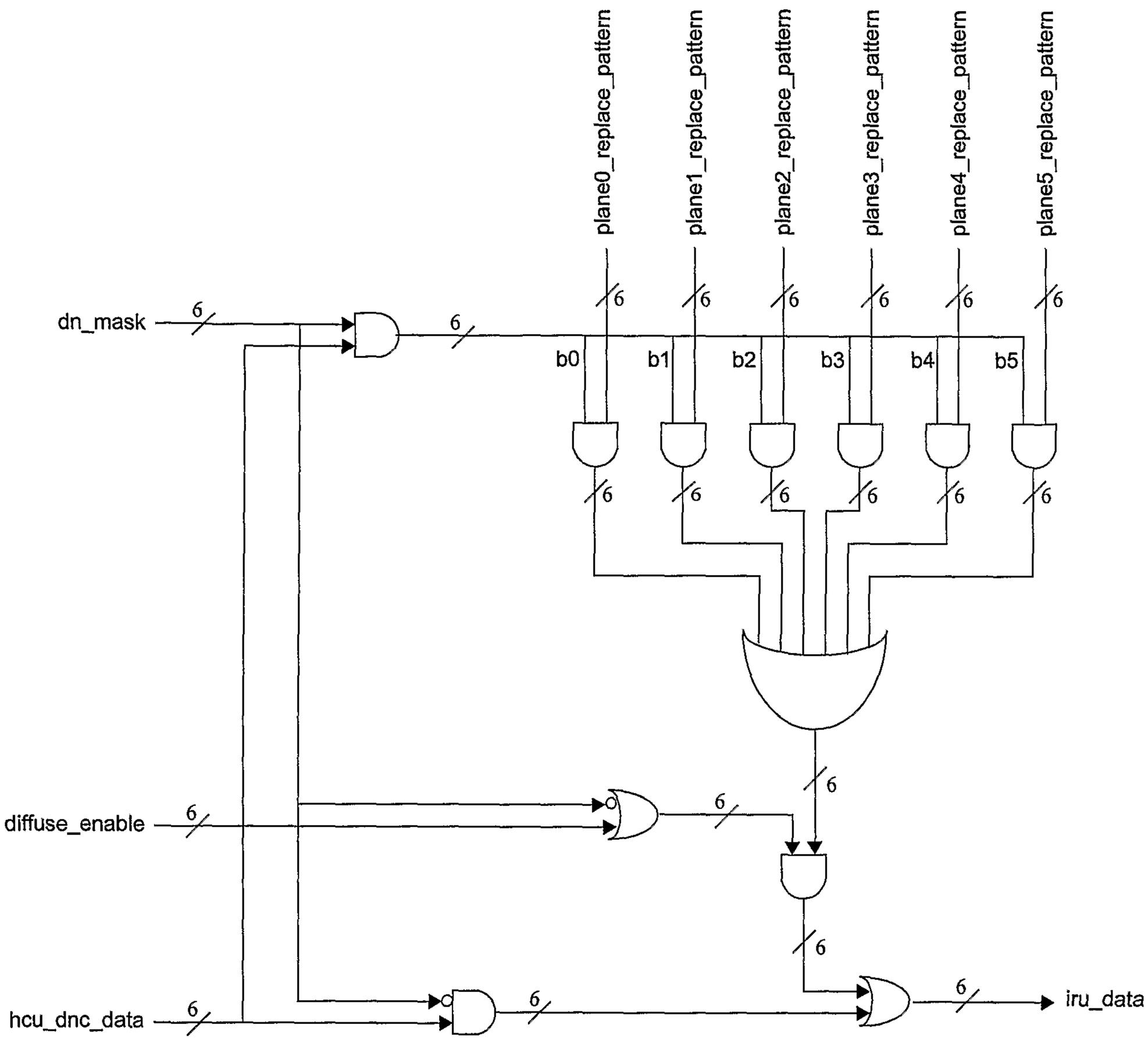

plane0_replace_pattern
plane1_replace_pattern
plane2_replace_pattern
plane3_replace_pattern
plane4_replace_pattern
plane5_replace_pattern
dn_mask
6
b0
b1
b2
b3
b4
b5
diffuse_enable
hcu_dnc_data
iru_data

FIG. 250

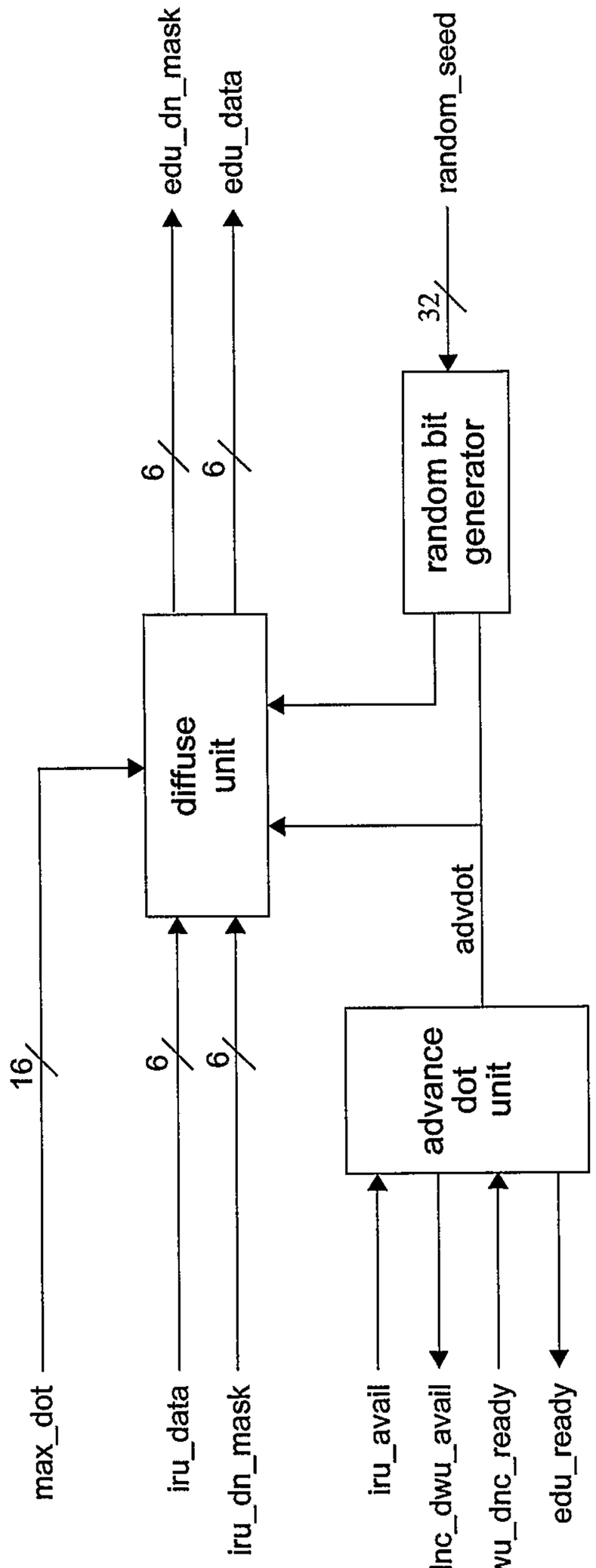

edu_dn_mask
edu_data
random_seed
32
6
6
random bit generator
diffuse unit
advdot
16
6
6
advance dot unit
max_dot
iru_data
iru_dn_mask
iru_avail
dnc_dwu_avail
dwu_dnc_ready
edu_ready

FIG. 251

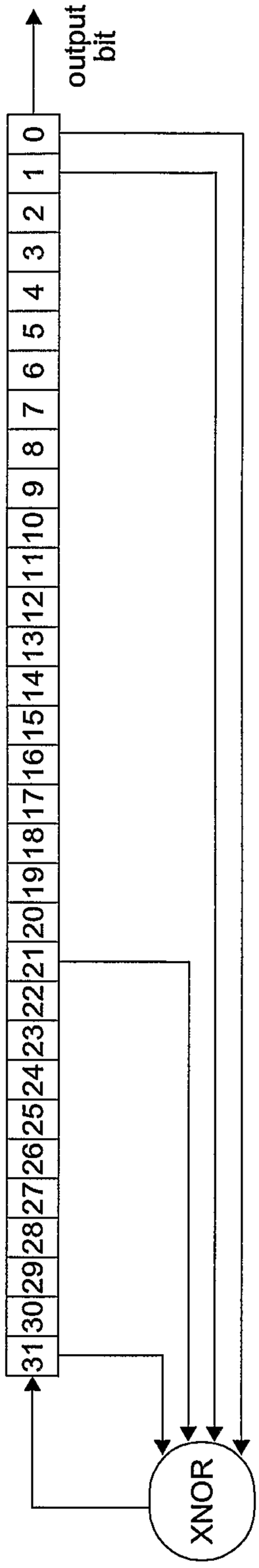


FIG. 252

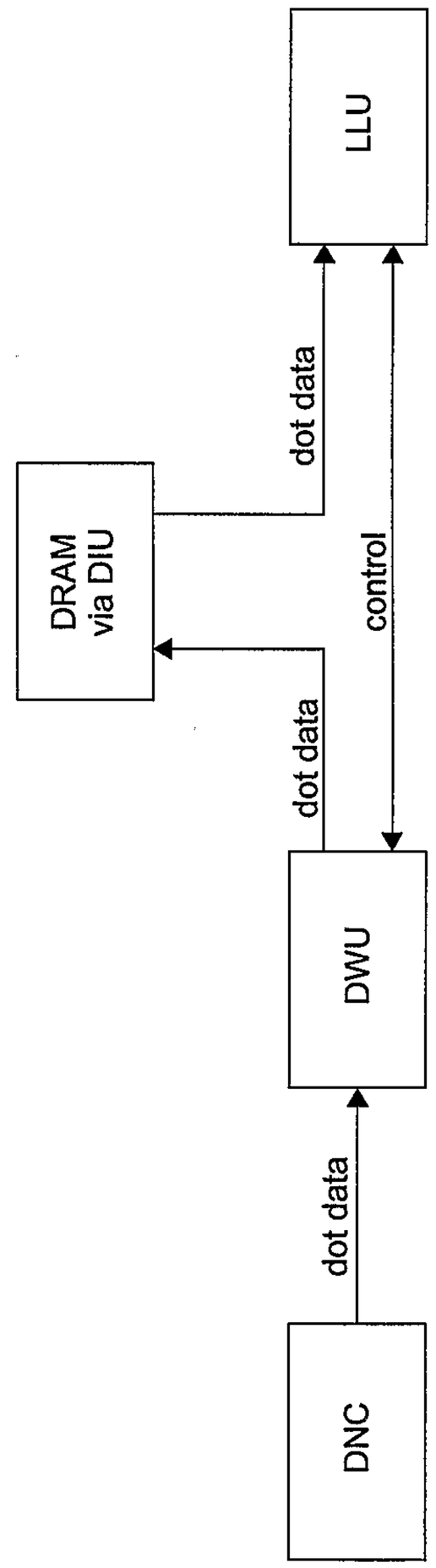


FIG. 253

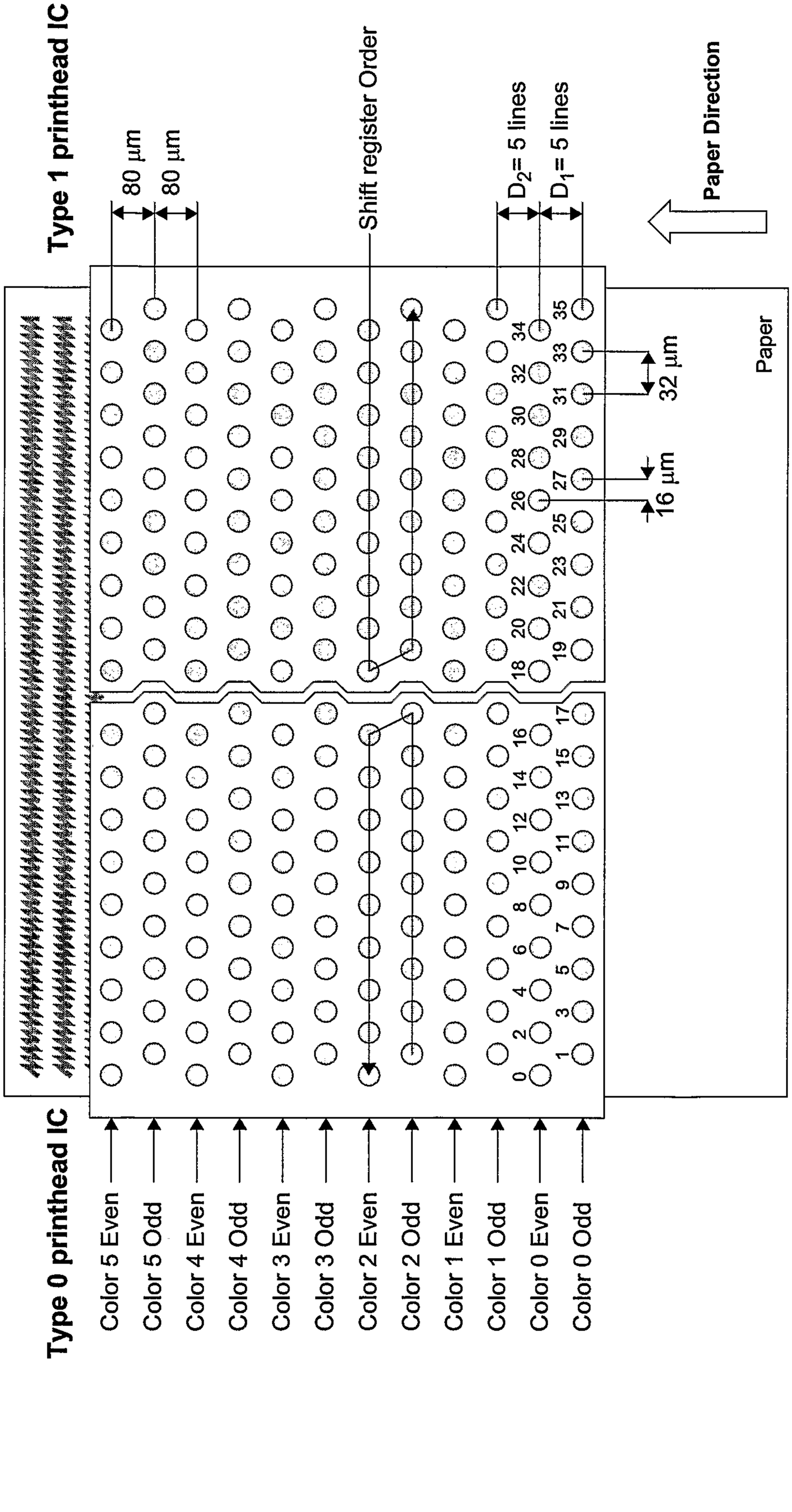

Type 1 printhead IC
80 µm
80 µm
Shift register Order
D_2= 5 lines
D_1= 5 lines
Paper Direction
16 µm
32 µm
Paper
Type 0 printhead IC
Color 5 Even
Color 5 Odd
Color 4 Even
Color 4 Odd
Color 3 Even
Color 3 Odd
Color 2 Even
Color 2 Odd
Color 1 Even
Color 1 Odd
Color 0 Even
Color 0 Odd
Note: Paper passes under printhead

FIG. 254

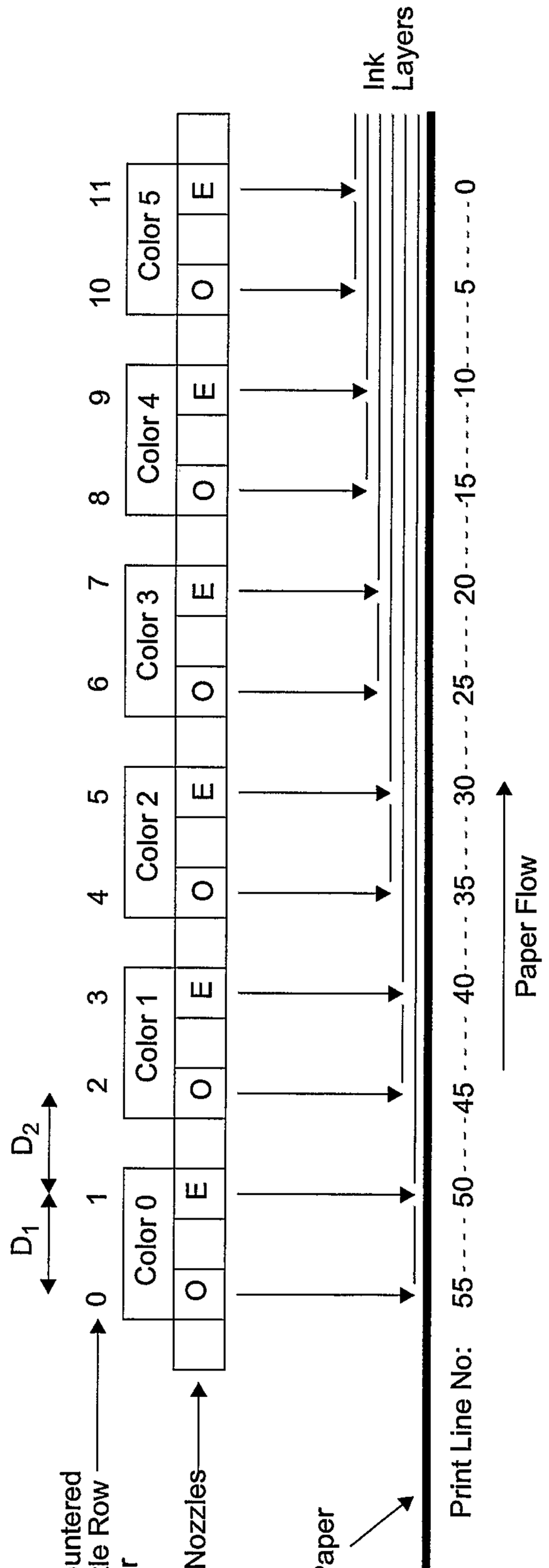

Ink Layers
Color 0
Color 1
Color 2
Color 3
Color 4
Color 5
O
E
D1
D2
0
1
2
3
4
5
6
7
8
9
10
11
5
10
15
20
25
30
35
40
45
50
55
Paper Flow
Encountered Nozzle Row Order
Nozzles
Paper
Print Line No:

FIG. 255

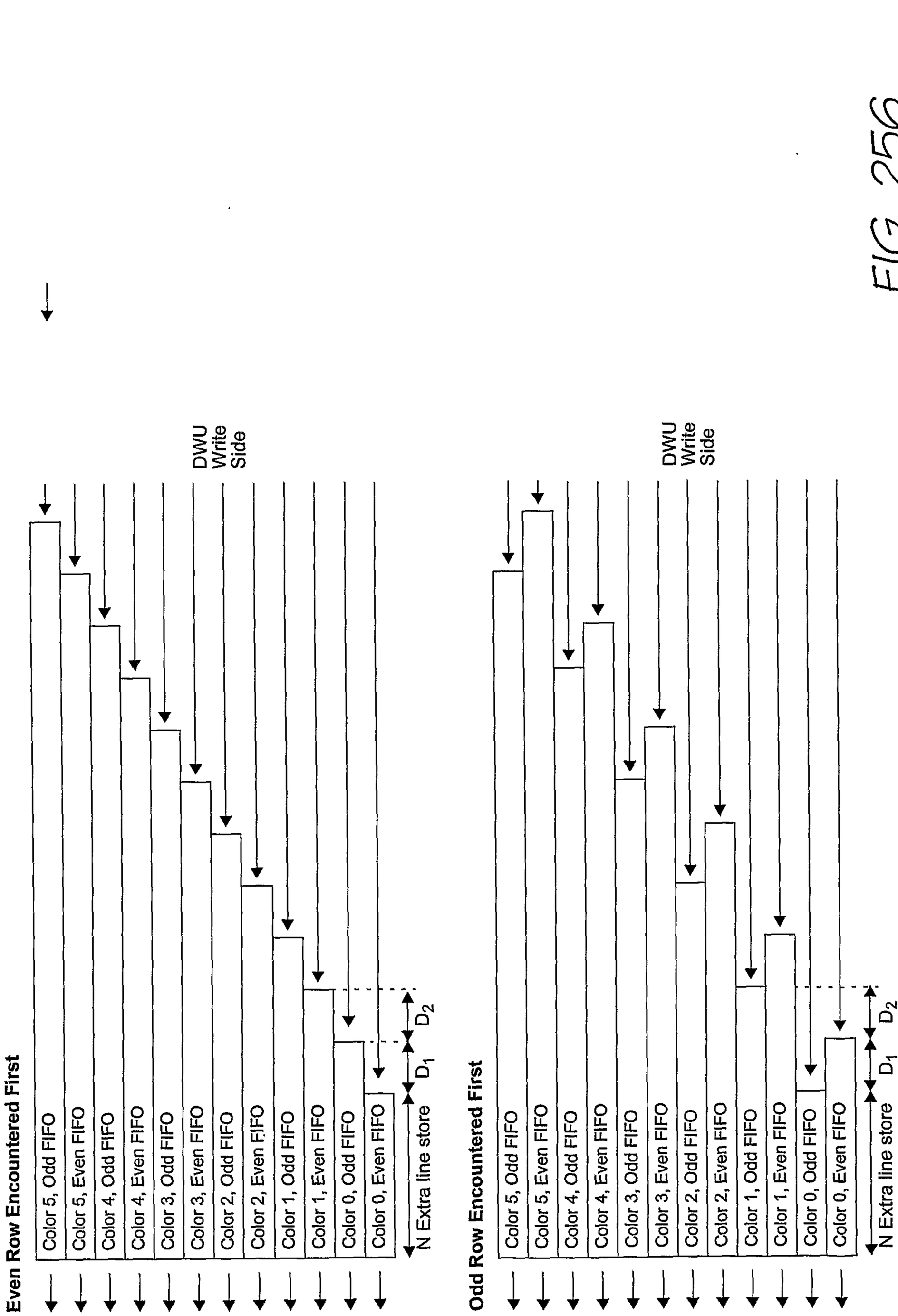

Even Row Encountered First
Color 5, Odd FIFO
Color 5, Even FIFO
Color 4, Odd FIFO
Color 4, Even FIFO
Color 3, Odd FIFO
Color 3, Even FIFO
Color 2, Odd FIFO
Color 2, Even FIFO
Color 1, Odd FIFO
Color 1, Even FIFO
Color 0, Odd FIFO
Color 0, Even FIFO
N Extra line store
LLU Read Side
DWU Write Side
D1
D2
Odd Row Encountered First
Color 5, Odd FIFO
Color 5, Even FIFO
Color 4, Odd FIFO
Color 4, Even FIFO
Color 3, Odd FIFO
Color 3, Even FIFO
Color 2, Odd FIFO
Color 2, Even FIFO
Color 1, Odd FIFO
Color 1, Even FIFO
Color 0, Odd FIFO
Color 0, Even FIFO
N Extra line store
LLU Read Side
DWU Write Side
D1
D2

FIG. 256

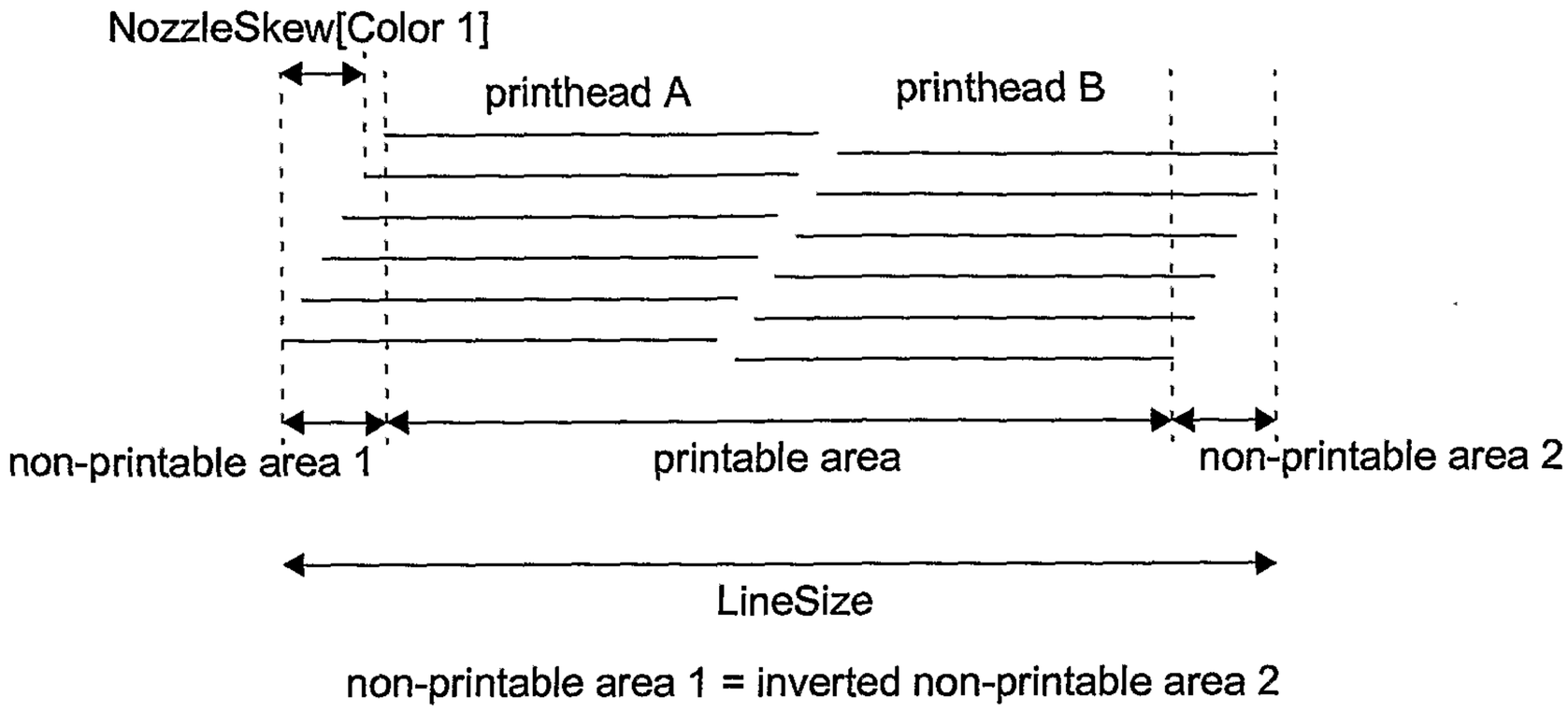


FIG. 257

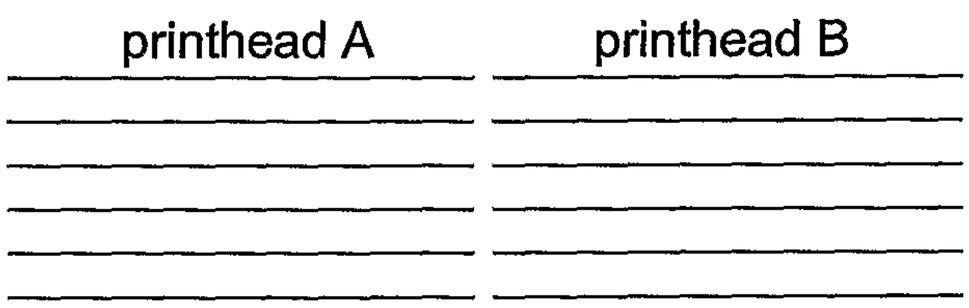


FIG. 258

FIG. 259

x2 Buffering
128 bits
write pt
read pt
x1.5 Buffering
write pt
read pt
FIFO empty writing begins
256 bits full, DRAM request issued
Variable cycles later DRAM request granted
256 bits transferred
256 bits full, DRAM request issued again
Variable cycles later DRAM request granted
256 bits transferred
Process repeated over one line
Final 256 bits full, DRAM request issued
Variable cycles later DRAM final request granted, 256 bits transferred

FIG. 260

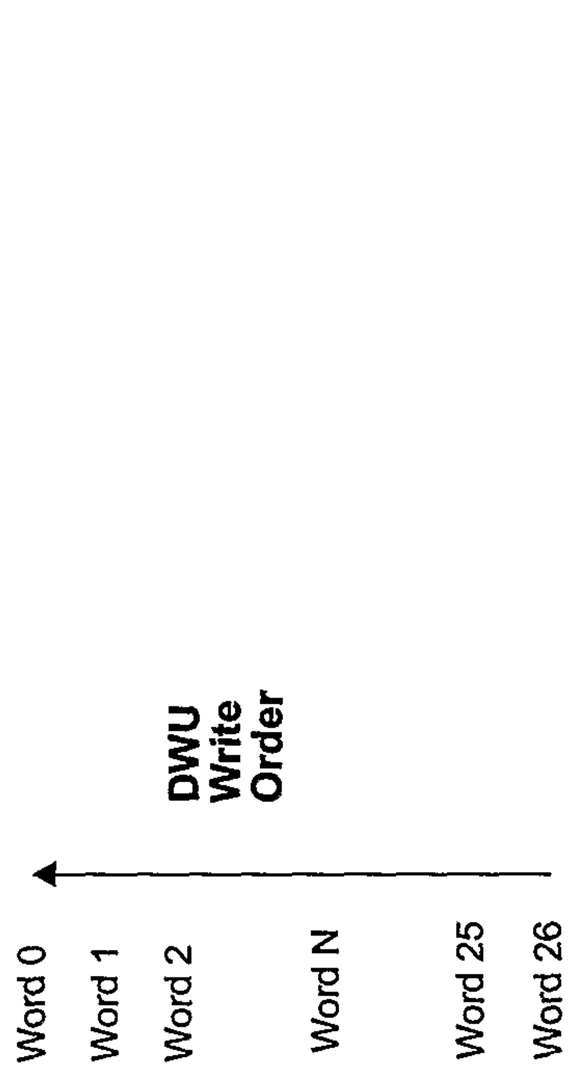
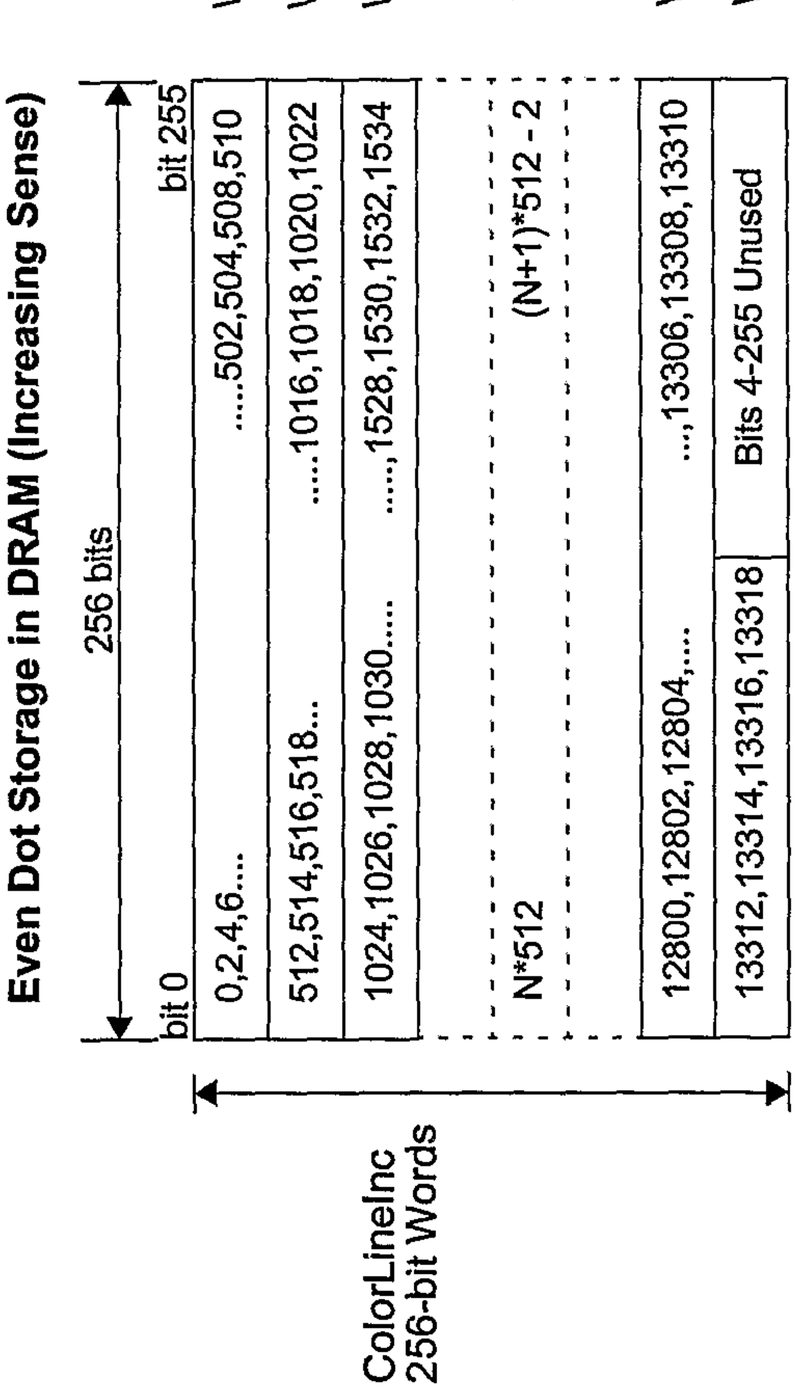
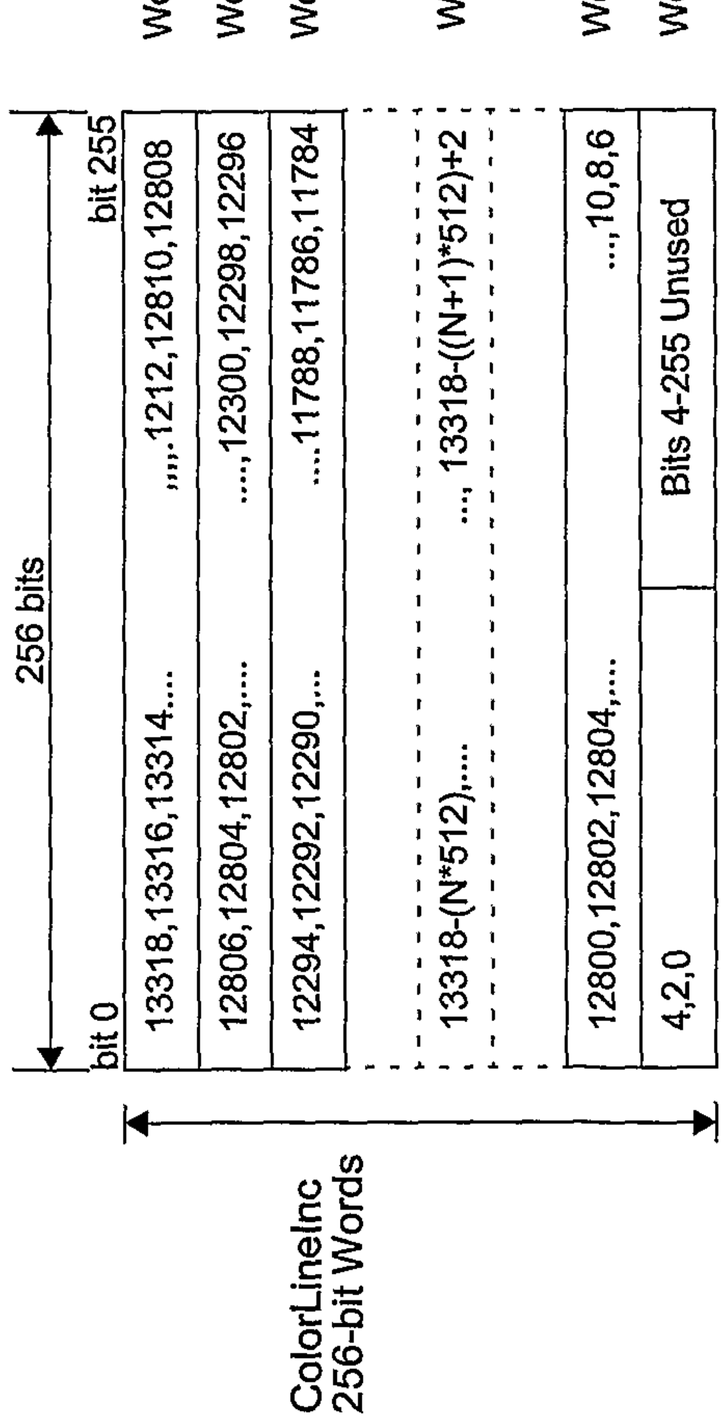
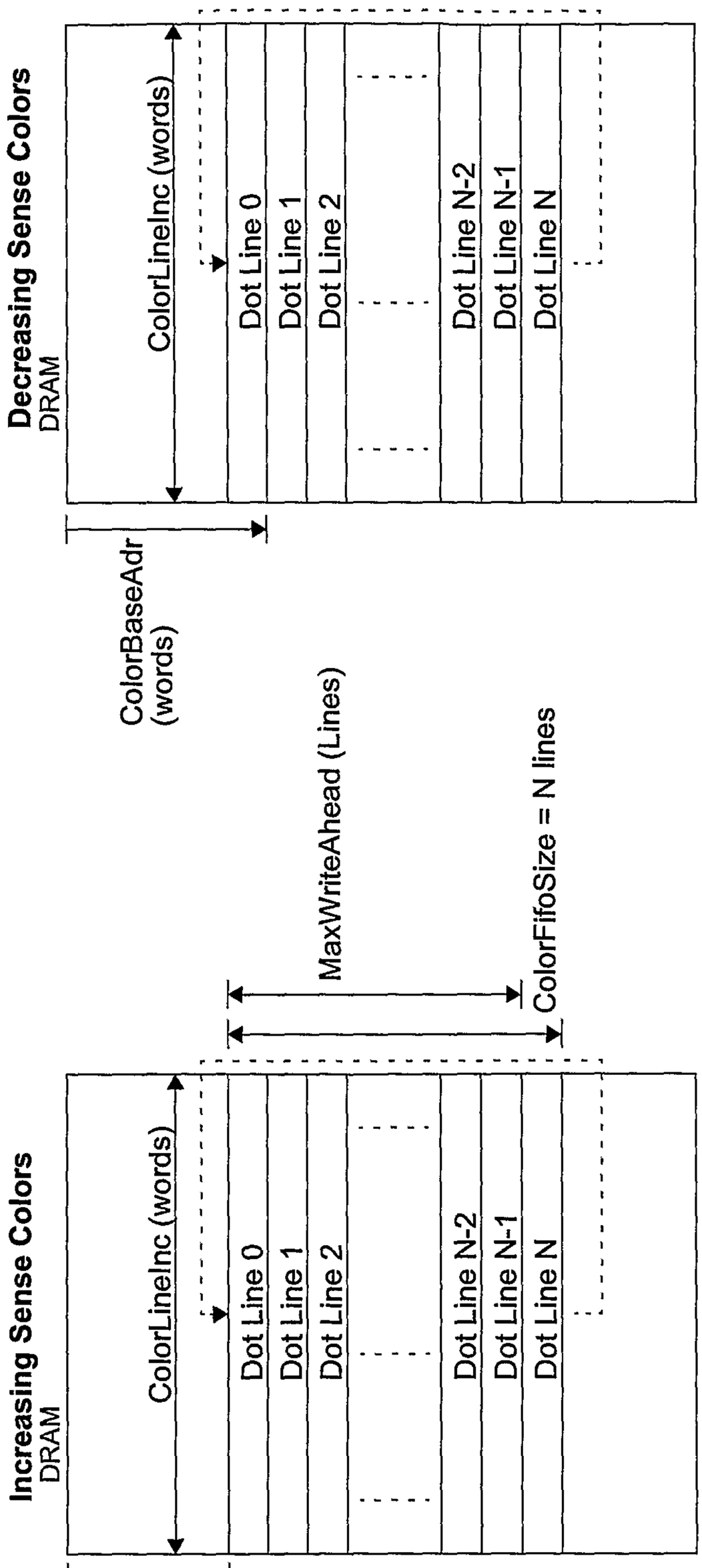


FIG. 261






Decreasing Sense Colors
DRAM
ColorLineInc (words)
Dot Line 0
Dot Line 1
Dot Line 2
Dot Line N-2
Dot Line N-1
Dot Line N
ColorBaseAdr
(words)
MaxWriteAhead (Lines)
ColorFifoSize = N lines
Increasing Sense Colors
DRAM
ColorLineInc (words)
Dot Line 0
Dot Line 1
Dot Line 2
Dot Line N-2
Dot Line N-1
Dot Line N
ColorBaseAdr
(words)

262

FIG. 2

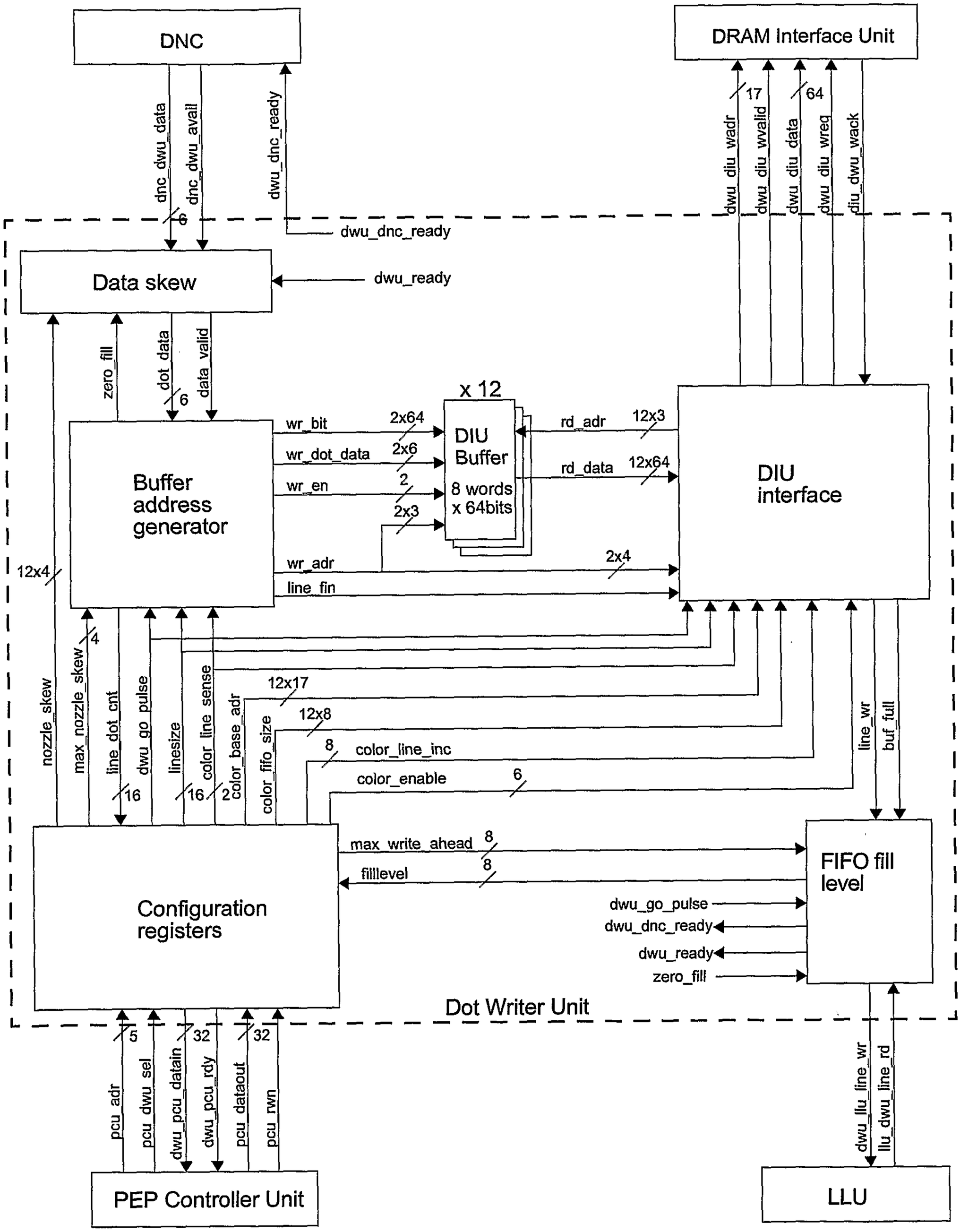

DNC
DRAM Interface Unit
dnc_dwu_data
dnc_dwu_avail
dwu_dnc_ready
6
17
64
dwu diu wadr
dwu diu wvalid
dwu diu data
dwu diu wreq
diu_dwu_wack
dwu_dnc_ready
Data skew
dwu_ready
zero_fill
dot data
data valid
6
x 12
wr_bit
2x64
wr_dot_data
2x6
wr_en
2
2x3
DIU Buffer
8 words x 64bits
rd_adr
12x3
rd_data
12x64
DIU interface
Buffer address generator
12x4
wr_adr
2x4
line_fin
4
nozzle_skew
max_nozzle_skew
line_dot_cnt
dwu_go_pulse
linesize
color_line_sense
color_base_adr
color_fifo_size
12x17
12x8
8
color_line_inc
color_enable
6
16
16
2
line_wr
buf_full
max_write_ahead
8
8
filllevel
FIFO fill level
Configuration registers
dwu_go_pulse
dwu_dnc_ready
dwu_ready
zero_fill
Dot Writer Unit
5
32
32
pcu_adr
pcu_dwu_sel
dwu_pcu_datain
dwu_pcu_rdy
pcu_dataout
pcu_rwn
dwu_llu_line_wr
llu_dwu_line_rd
PEP Controller Unit
LLU

FIG. 263

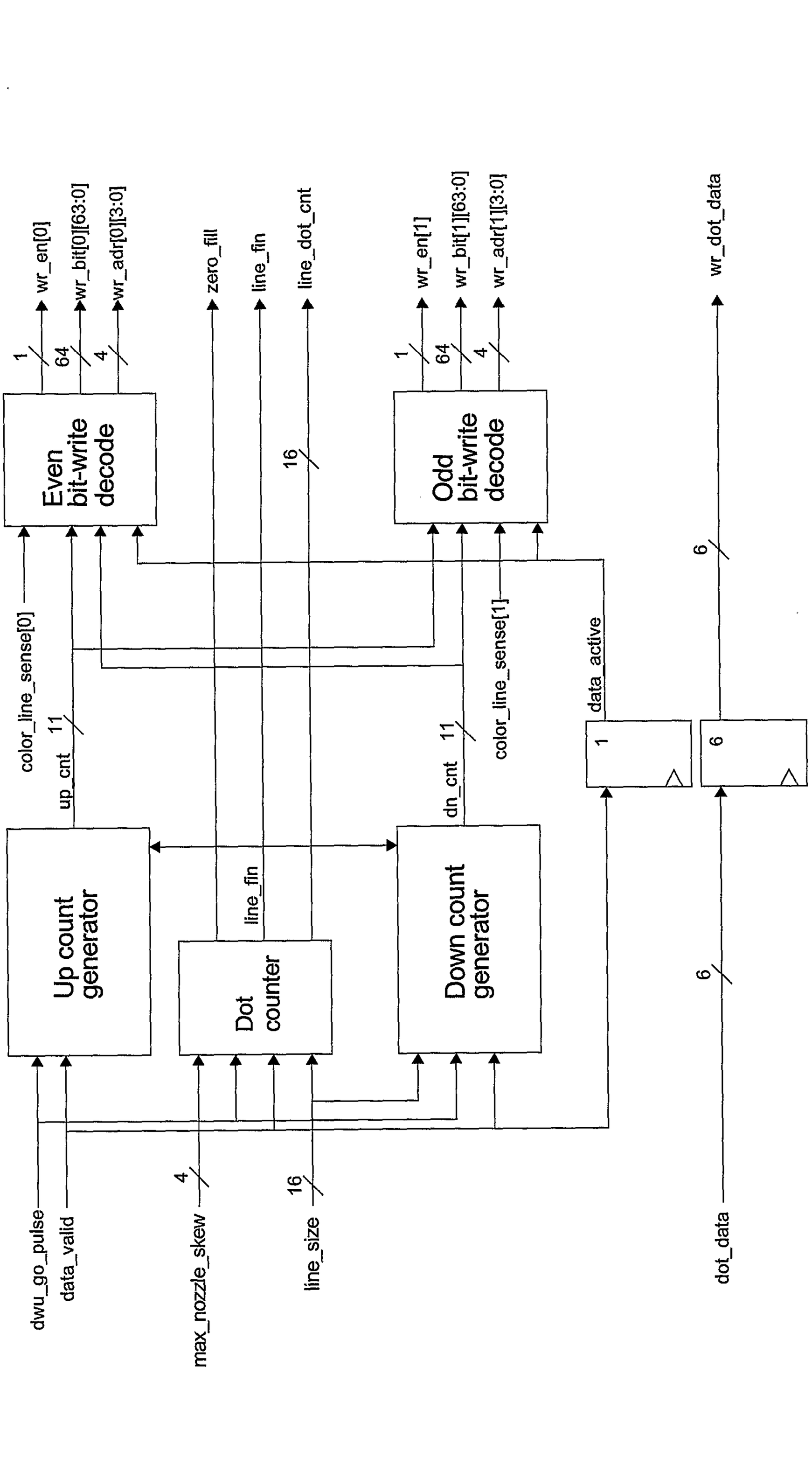

Even bit-write decode
wr_en[0]
wr_bit[0][63:0]
wr_adr[0][3:0]
zero_fill
line_fin
line_dot_cnt
Odd bit-write decode
wr_en[1]
wr_bit[1][63:0]
wr_adr[1][3:0]
color_line_sense[0]
color_line_sense[1]
Up count generator
Dot counter
Down count generator
up_cnt
dn_cnt
data_active
wr_dot_data
dwu_go_pulse
data_valid
max_nozzle_skew
line_size
dot_data

FIG. 264

265

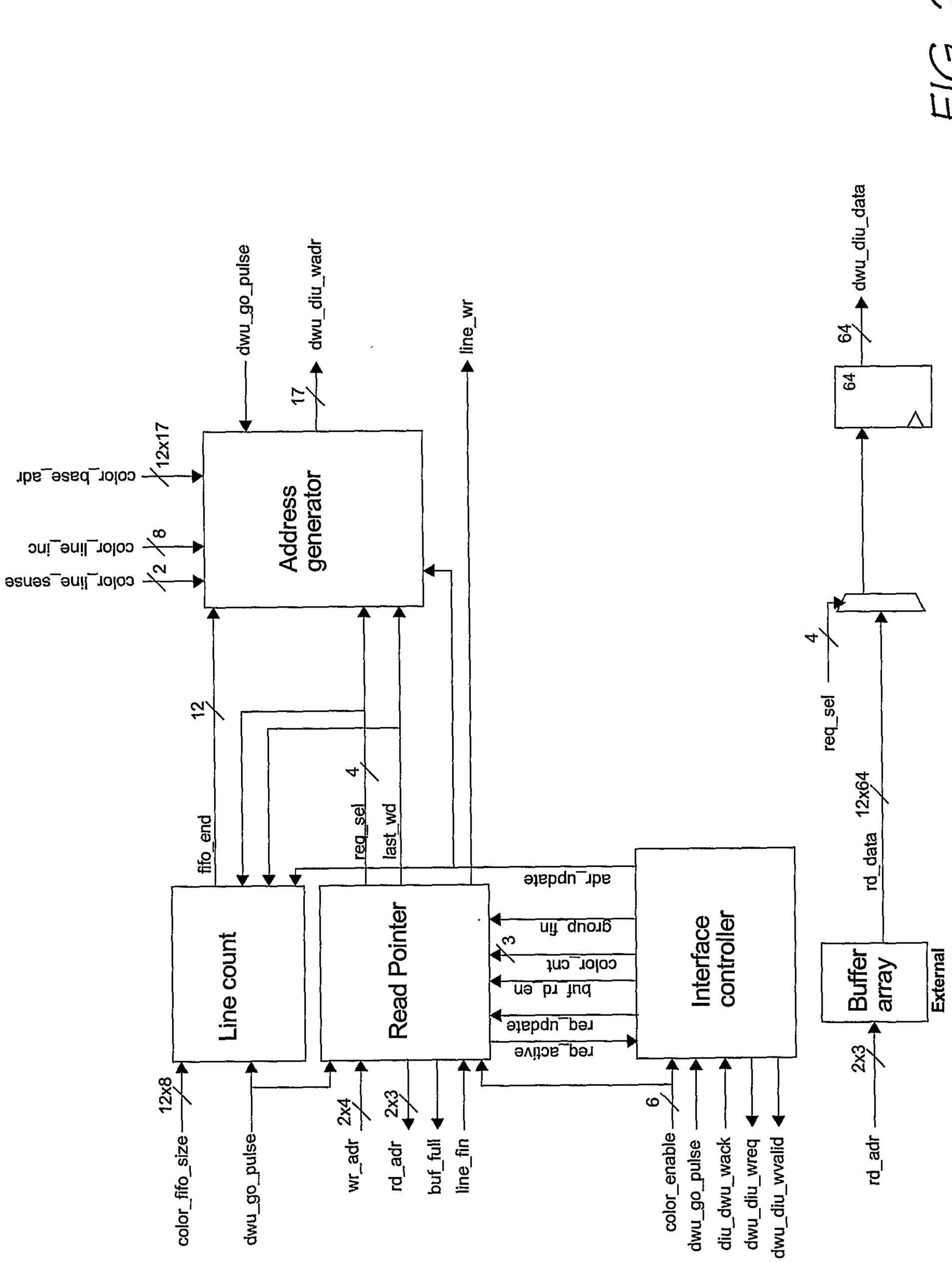

dwu_go_pulse
dwu_diu_wadr
line_wr
12x17
color_base_adr
8
color_line_inc
2
color_line_sense
Address generator
12
fifo_end
4
req_sel
last_wd
adr_update
group_fin
3
color_cnt
buf_rd_en
req_active
req_update
Line count
Read Pointer
Interface controller
12x8
color_fifo_size
dwu_go_pulse
2x4
wr_adr
2x3
rd_adr
buf_full
line_fin
6
color_enable
dwu_go_pulse
diu_dwu_wack
dwu_diu_wreq
dwu_diu_wvalid
64
dwu_diu_data
4
req_sel
12x64
rd_data
Buffer array
External
2x3
rd_adr

FIG. 265

Machine remains in same state by default
All outputs are zero unless otherwise stated

State Description:

Idle: Idle state wait for active request

ColorSelect: Select the color to update before requesting to DIU

Request: Request issued wait for acknowledge

Data0: Data word 0 transfer

Data1: Data word 1 transfer

Data2: Data word 2 transfer

Data3: Data word 3 transfer

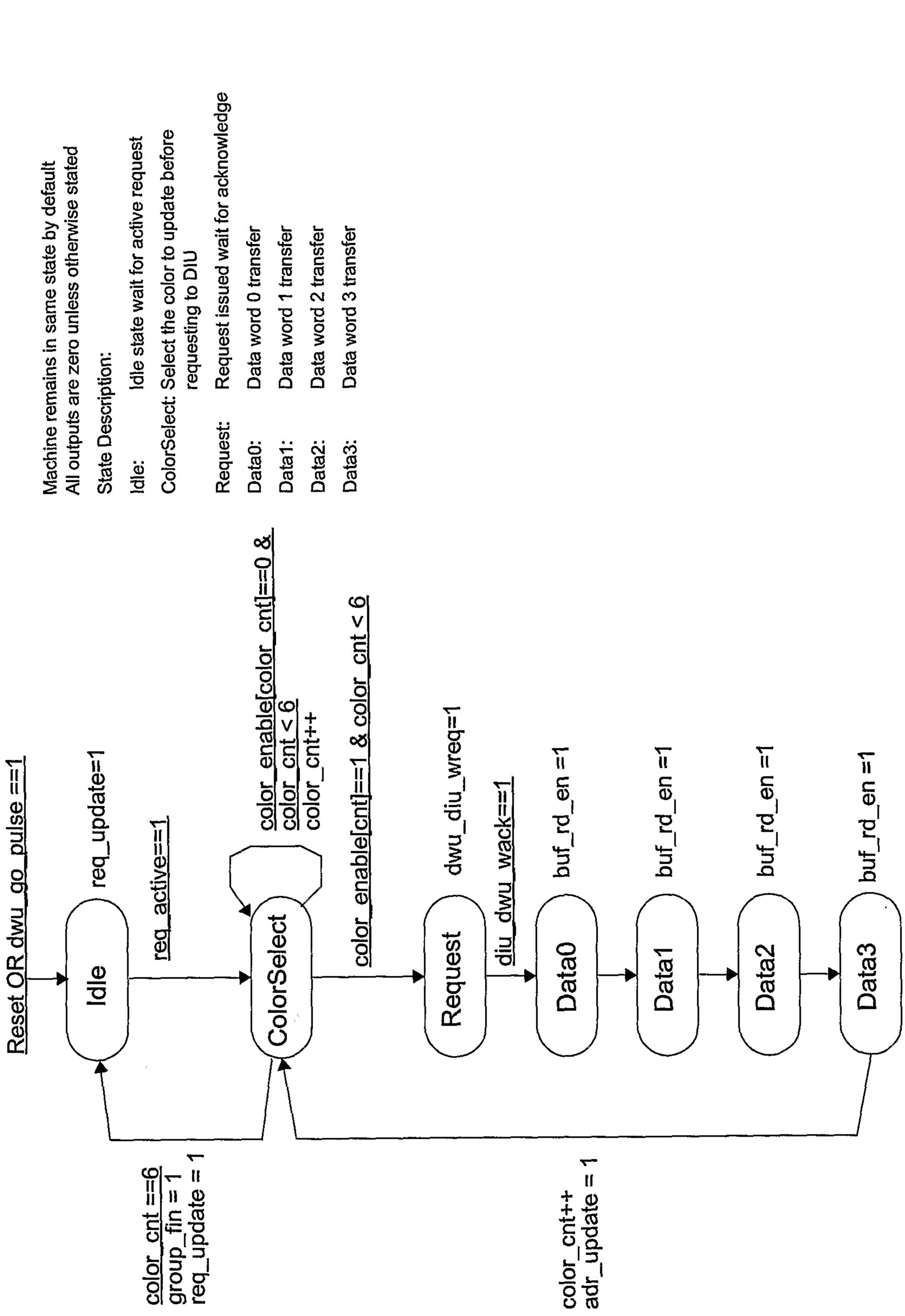


FIG. 266

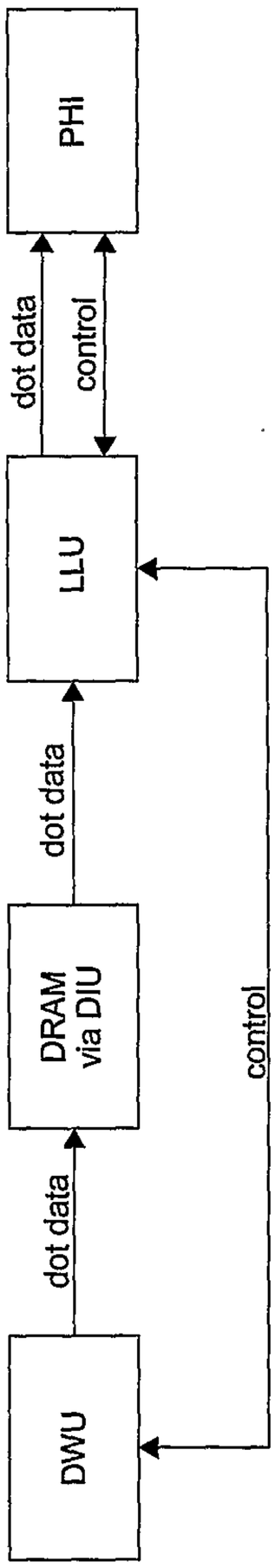


FIG. 267

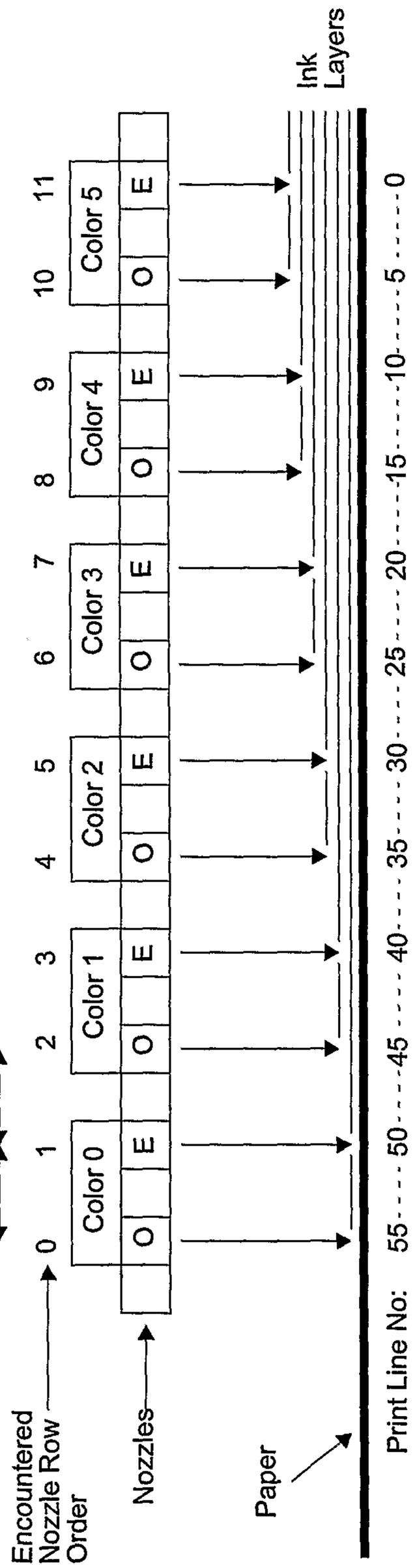


FIG. 268

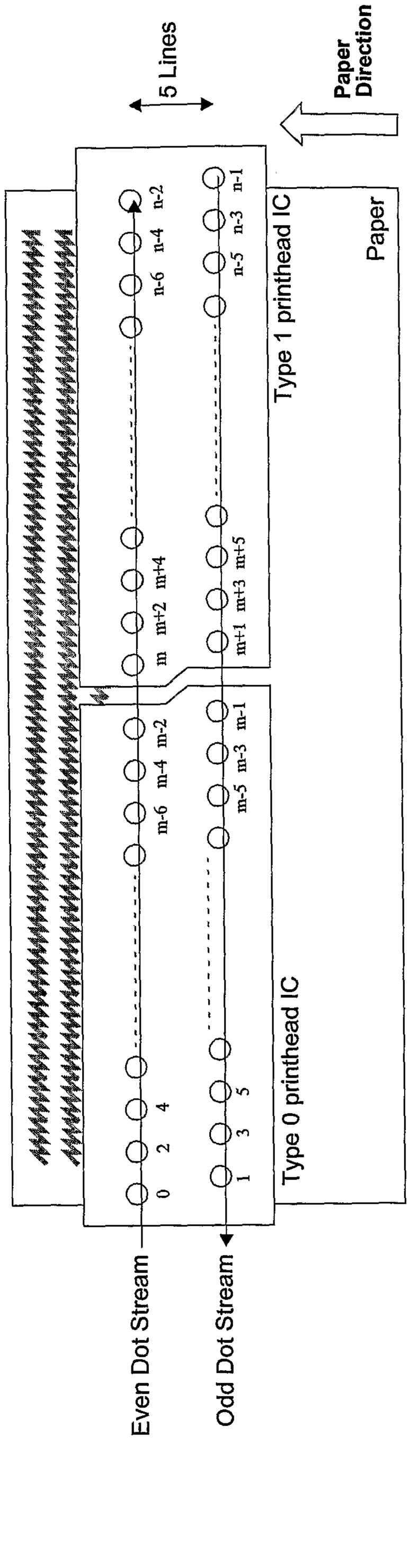

5 Lines
Paper Direction
n-6 n-4 n-2
n-5 n-3 n-1
Type 1 printhead IC
Paper
m m+2 m+4
m-1 m+1 m+3 m+5
m-6 m-4 m-2
m-5 m-3 m-1
Type 0 printhead IC
2 4
1 3 5
0
1
Even Dot Stream
Odd Dot Stream
Note: Paper passing under printhead
M - Midway point in dots
N - Number of dots in a line

FIG. 269

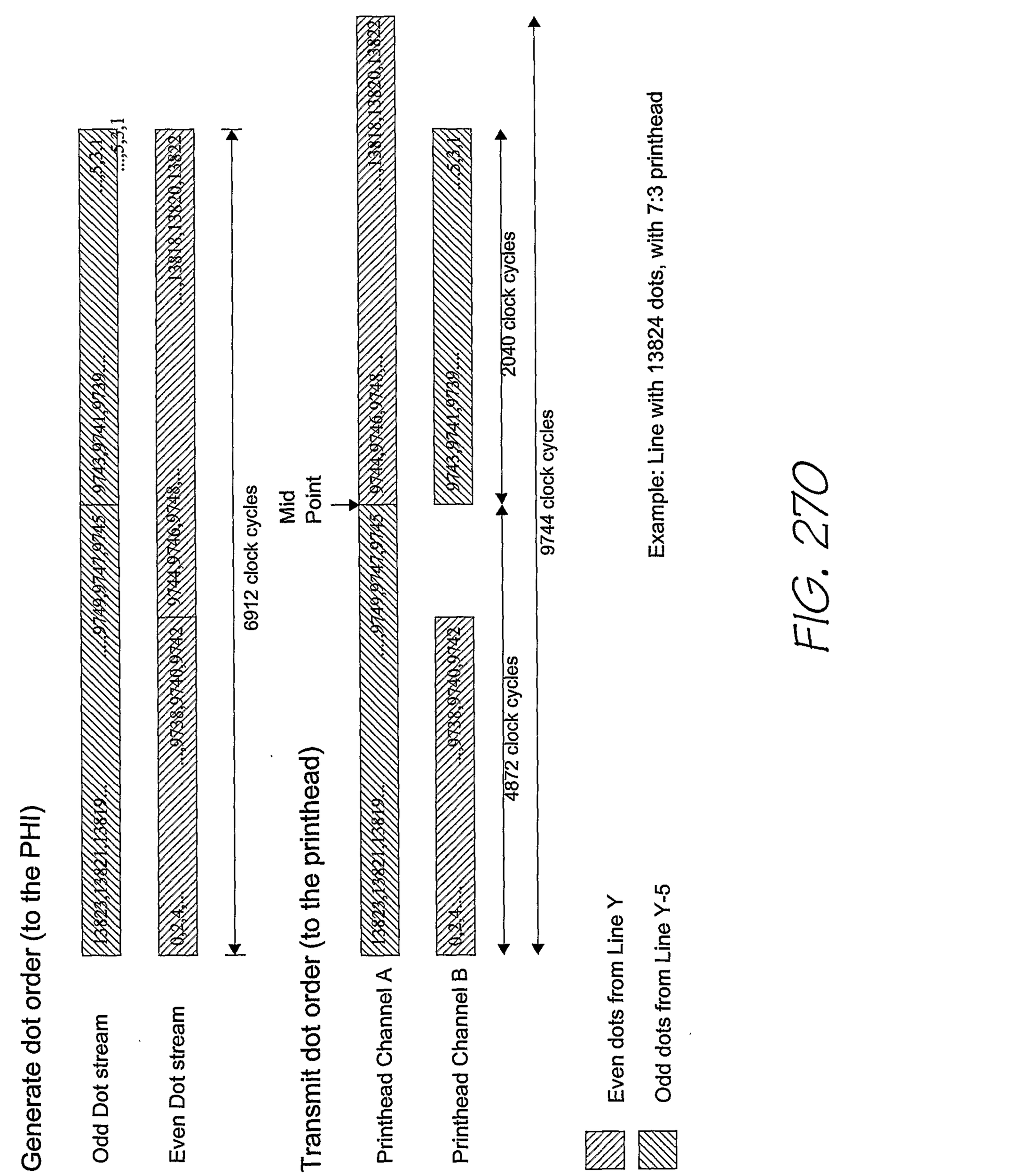

Generate dot order (to the PHI)
Odd Dot stream
Even Dot stream
6912 clock cycles
Mid Point
Transmit dot order (to the printhead)
Printhead Channel A
Printhead Channel B
4872 clock cycles
9744 clock cycles
Even dots from Line Y
Odd dots from Line Y-5
Example: Line with 13824 dots, with 7:3 printhead
2040 clock cycles

FIG. 270

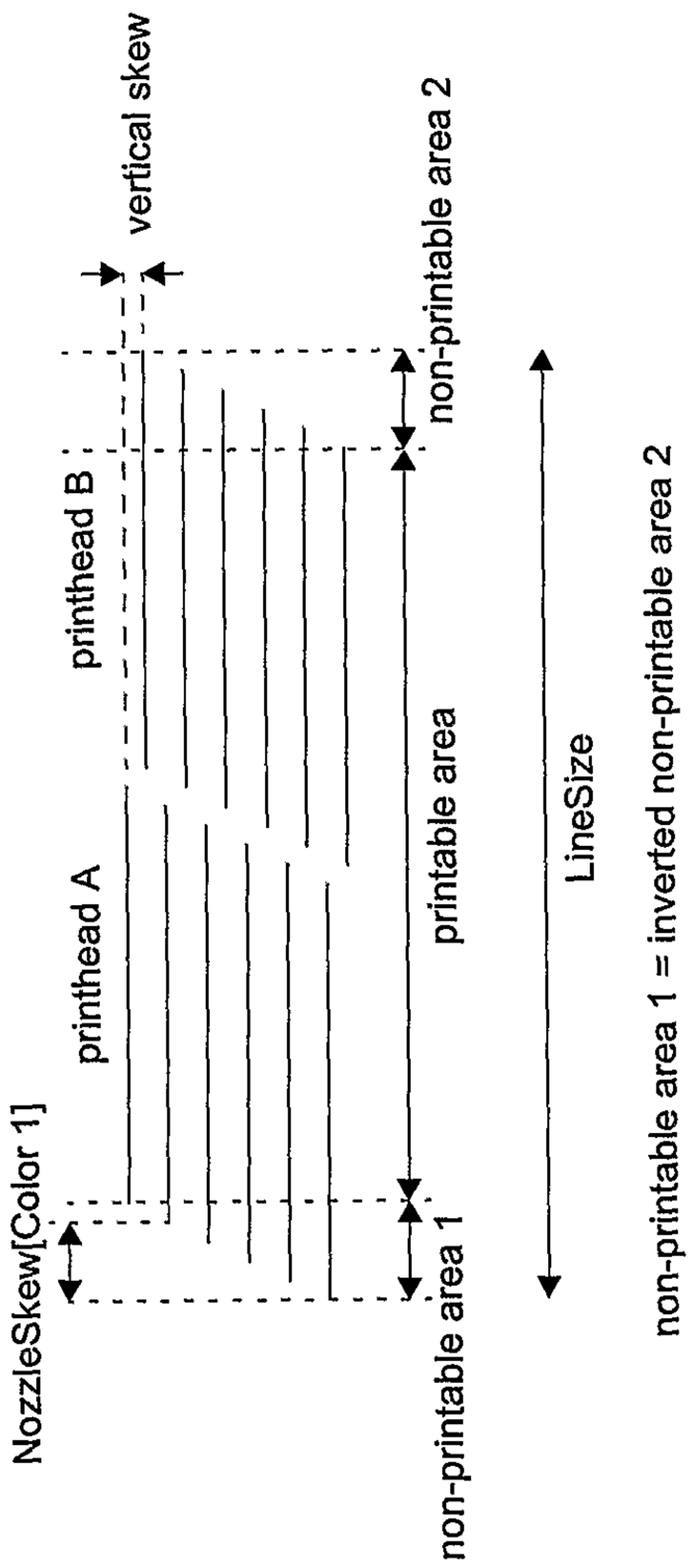

NozzleSkew[Color 1]
printhead A
printhead B
vertical skew
non-printable area 1
printable area
non-printable area 2
LineSize
non-printable area 1 = inverted non-printable area 2

FIG. 271

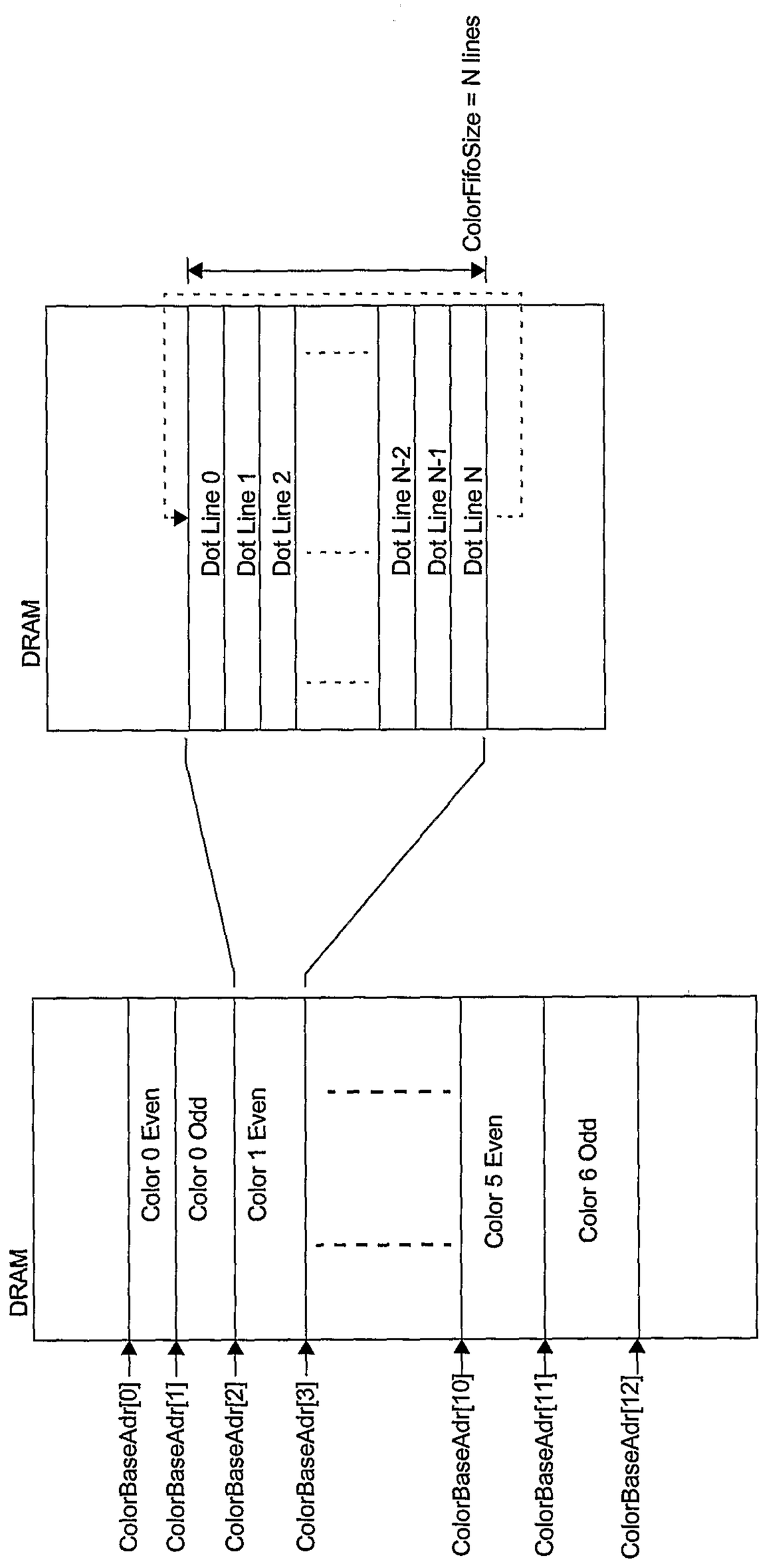

DRAM
Dot Line 0
Dot Line 1
Dot Line 2
Dot Line N-2
Dot Line N-1
Dot Line N
ColorFifoSize = N lines
DRAM
Color 0 Even
Color 0 Odd
Color 1 Even
Color 5 Even
Color 6 Odd
ColorBaseAdr[0]
ColorBaseAdr[1]
ColorBaseAdr[2]
ColorBaseAdr[3]
ColorBaseAdr[10]
ColorBaseAdr[11]
ColorBaseAdr[12]

FIG. 272

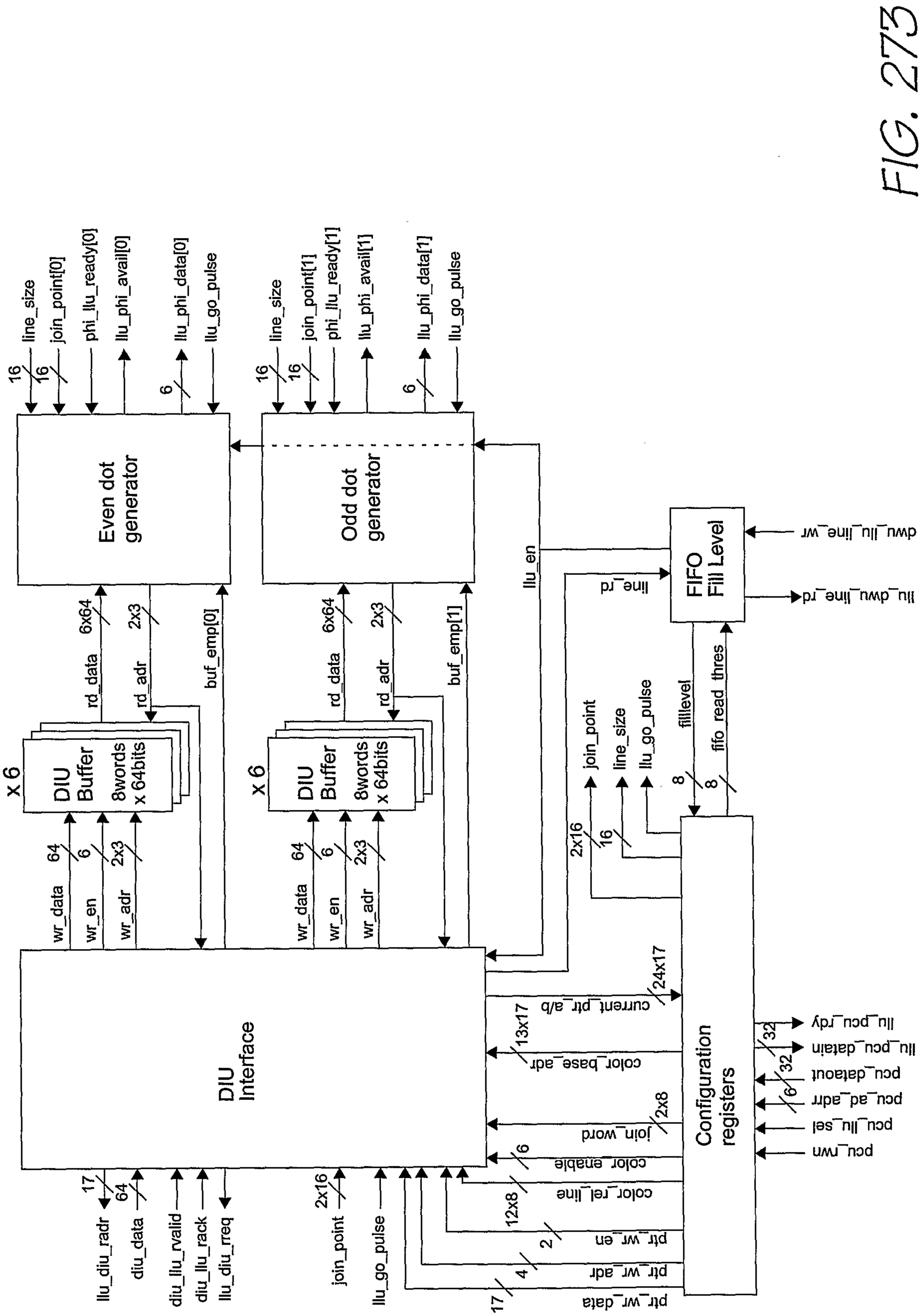

Even dot generator
Odd dot generator
line_size
join_point[0]
phi_llu_ready[0]
llu_phi_avail[0]
llu_phi_data[0]
llu_go_pulse
join_point[1]
phi_llu_ready[1]
llu_phi_avail[1]
llu_phi_data[1]
16
6
6x64
2x3
rd_data
rd_adr
buf_emp[0]
buf_emp[1]
x 6
DIU Buffer
8words x 64bits
wr_data
wr_en
wr_adr
64
2x3
llu_en
line_rd
FIFO Fill Level
dwu_llu_line_wr
llu_dwu_line_rd
join_point
line_size
llu_go_pulse
fillevel
fifo_read_thres
2x16
8
24x17
current_ptr_a/b
13x17
color_base_adr
2x8
join_word
color_enable
color_rel_line
12x8
ptr_wr_en
ptr_wr_adr
ptr_wr_data
DIU Interface
Configuration registers
llu_diu_radr
diu_data
diu_llu_rvalid
diu_llu_rack
llu_diu_rreq
2x16
join_point
llu_go_pulse
17
pcu_rwn
pcu_llu_sel
pcu_ad_adrr
pcu_datain
pcu_datout
llu_pcu_rdy
32

FIG. 273

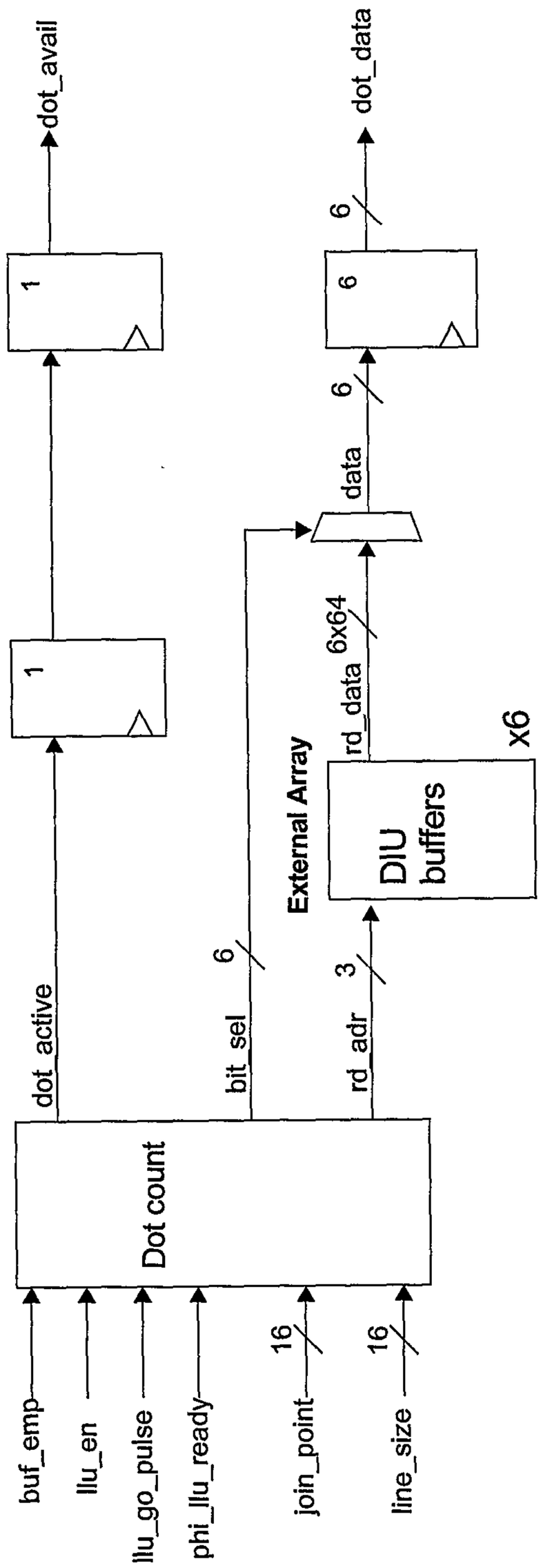

dot_avail
dot_data
1
6
6
6
data
1
dot_active
bit_sel
6
External Array
rd_data
6x64
rd_adr
3
DIU
buffers
x6
Dot count
buf_emp
llu_en
llu_go_pulse
phi_llu_ready
join_point
16
line_size
16

FIG. 274

275

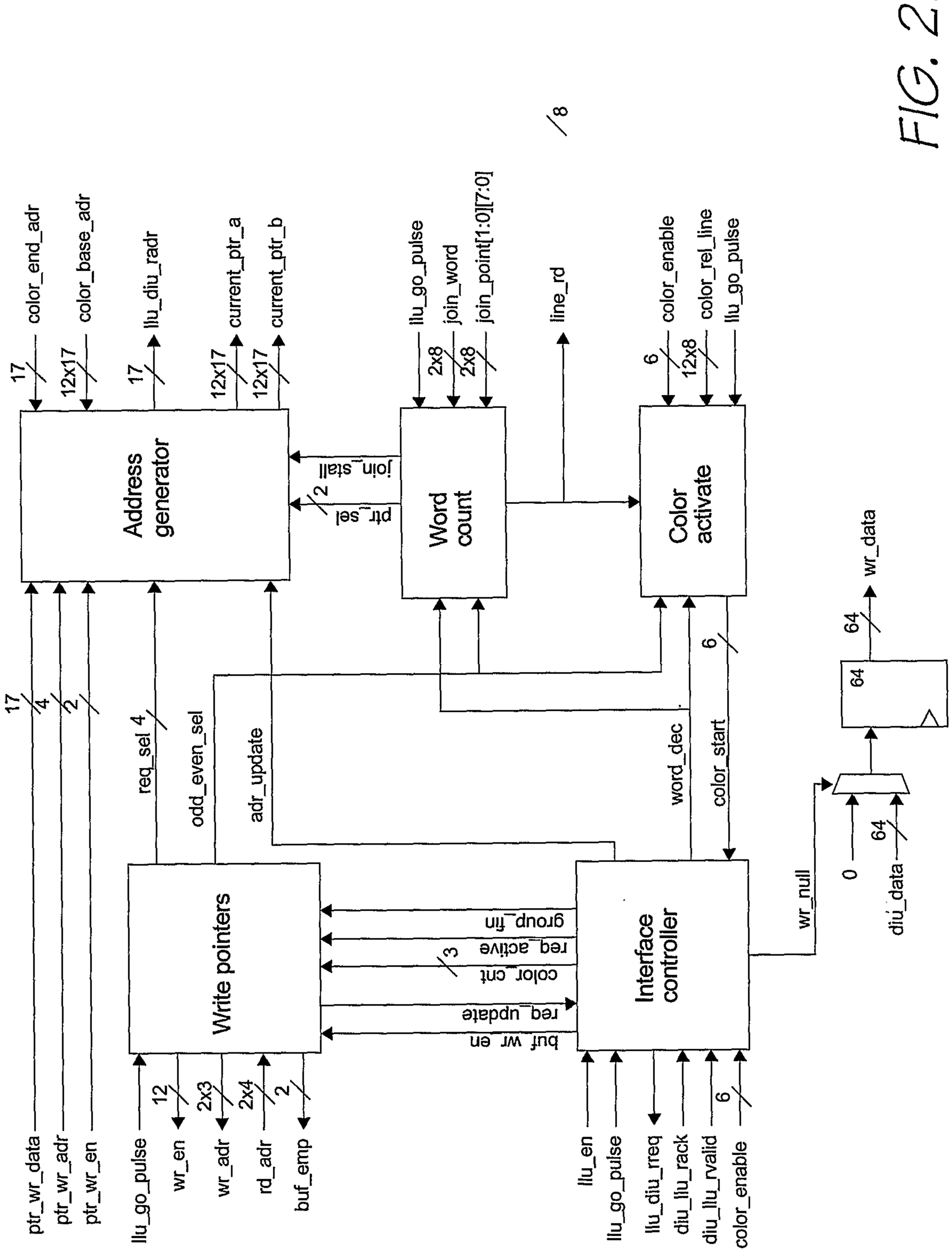

Address generator
Word count
Color activate
Write pointers
Interface controller
color_end_adr
color_base_adr
llu_diu_radr
current_ptr_a
current_ptr_b
join_stall
ptr_sel
llu_go_pulse
join_word
join_point[1:0][7:0]
line_rd
color_enable
color_rel_line
llu_go_pulse
req_sel
odd_even_sel
adr_update
word_dec
color_start
group_fin
req_active
color_cnt
req_update
buf_wr_en
wr_null
wr_data
diu_data
ptr_wr_data
ptr_wr_adr
ptr_wr_en
llu_go_pulse
wr_en
wr_adr
rd_adr
buf_emp
llu_en
llu_go_pulse
llu_diu_rreq
diu_llu_rack
diu_llu_rvalid
color_enable

FIG. 2

Machine remains in same state by default
All outputs are zero unless otherwise stated

State Description:

Idle: Idle state wait for active request

ColorSelect: Select the color to update before requesting to DIU

Request: Request issued wait for acknowledge

Data0: Data word 0 transfer

Data1: Data word 1 transfer

Data2: Data word 2 transfer

Data3: Data word 3 transfer

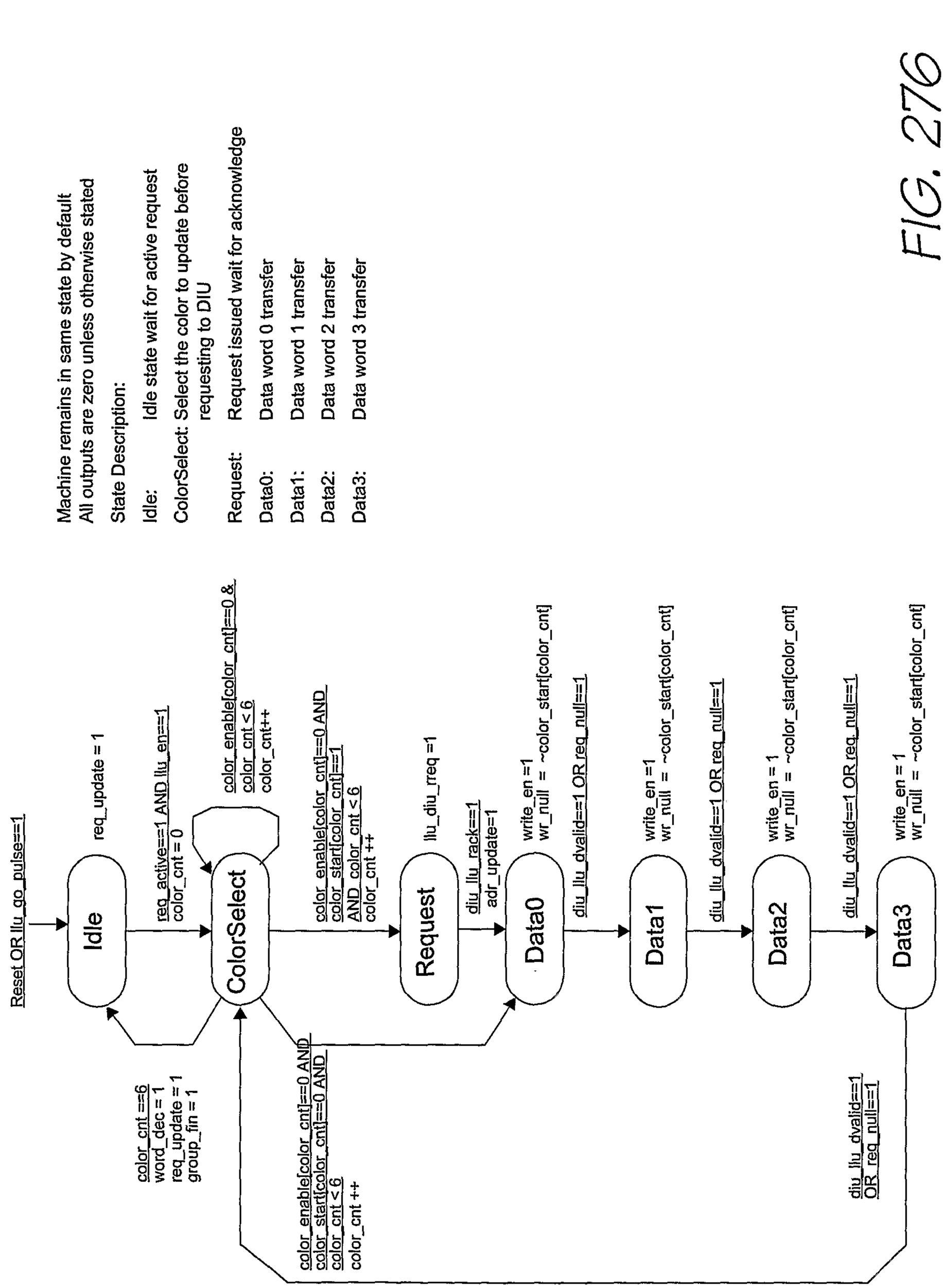


FIG. 276

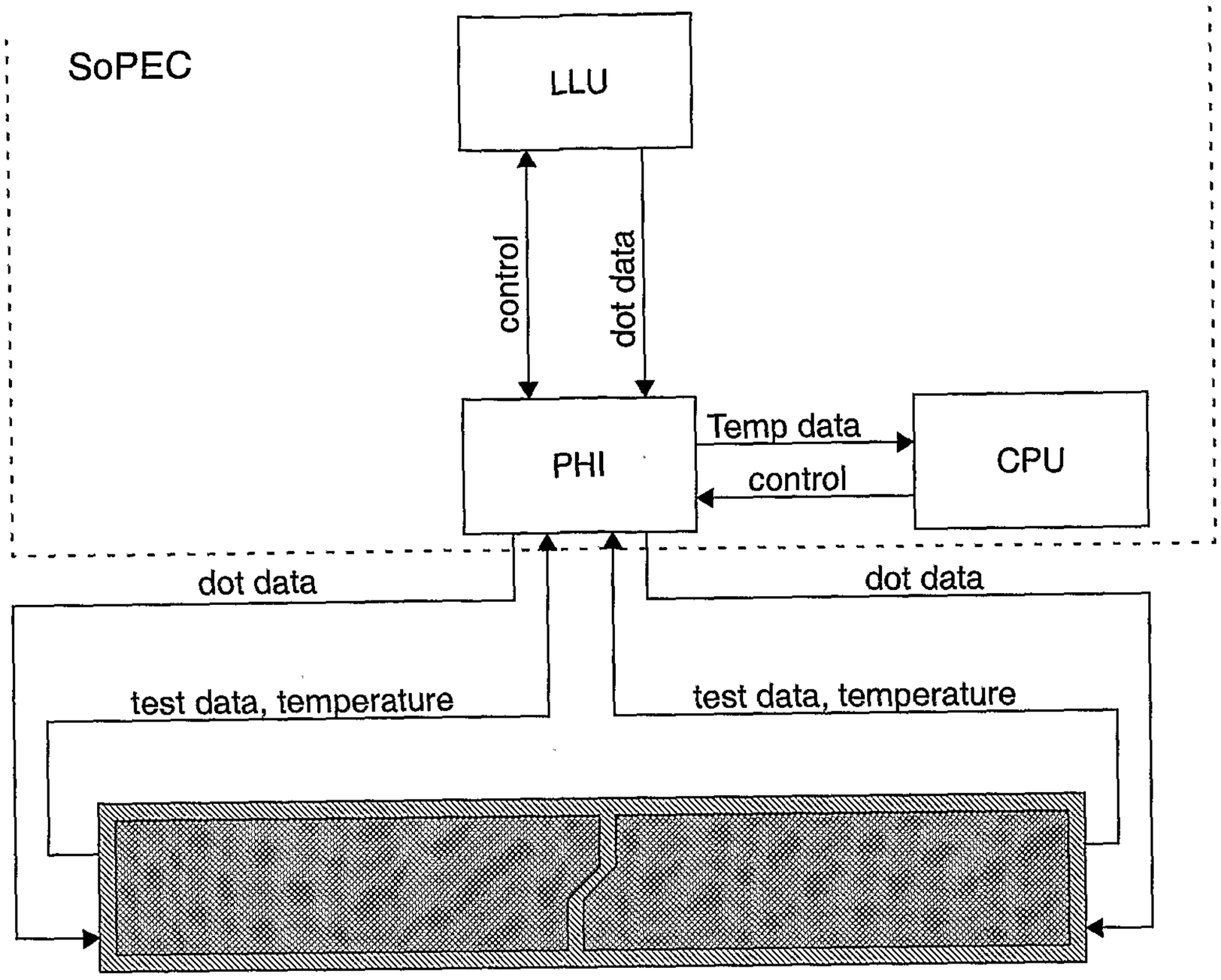


FIG. 277

FIG. 278

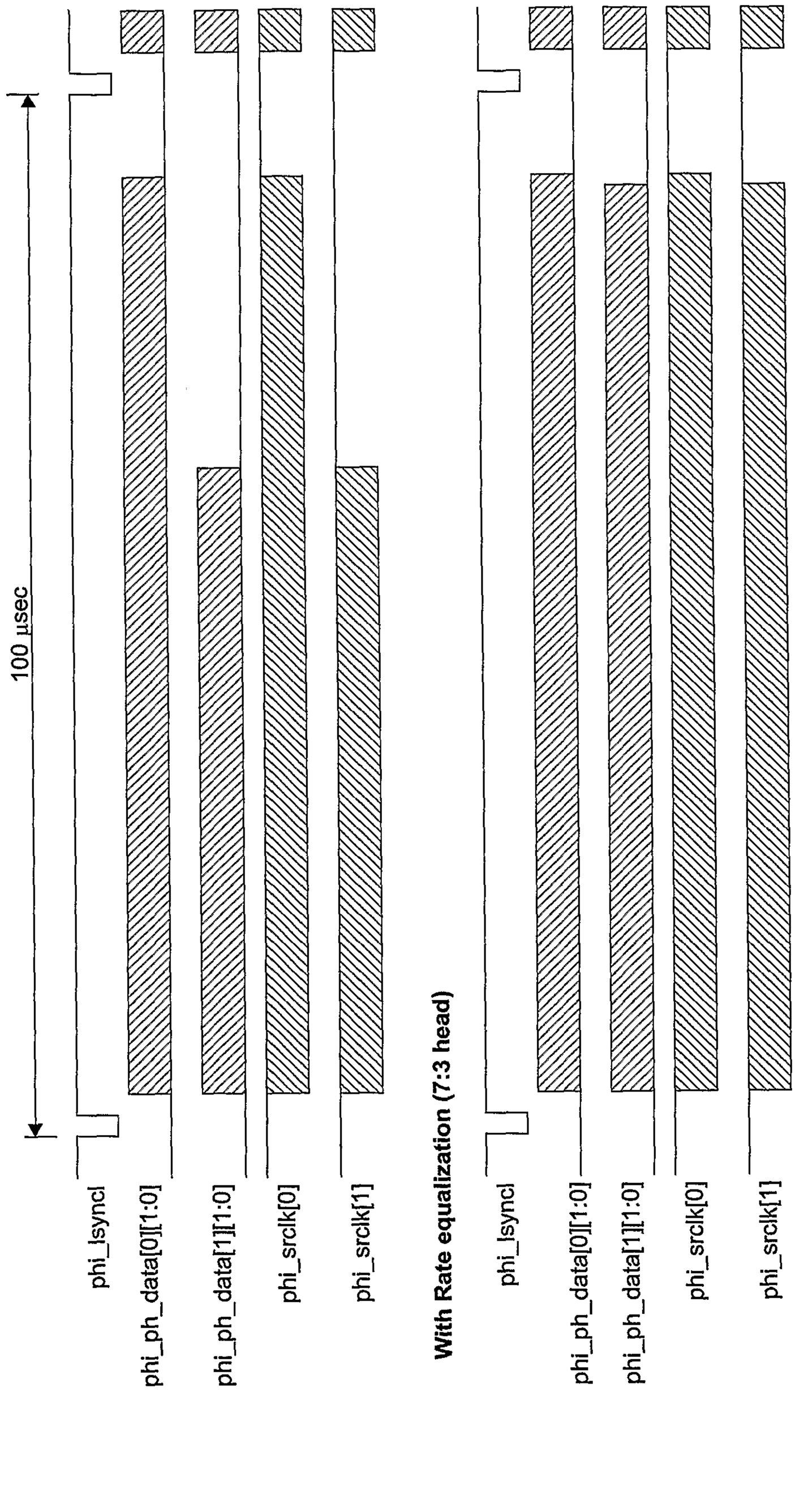

Without Rate equalization (7:3 head)
100 µsec
phi_lsyncl
phi_ph_data[0][1:0]
phi_ph_data[1][1:0]
phi_srclk[0]
phi_srclk[1]
With Rate equalization (7:3 head)
phi_lsyncl
phi_ph_data[0][1:0]
phi_ph_data[1][1:0]
phi_srclk[0]
phi_srclk[1]

FIG. 279

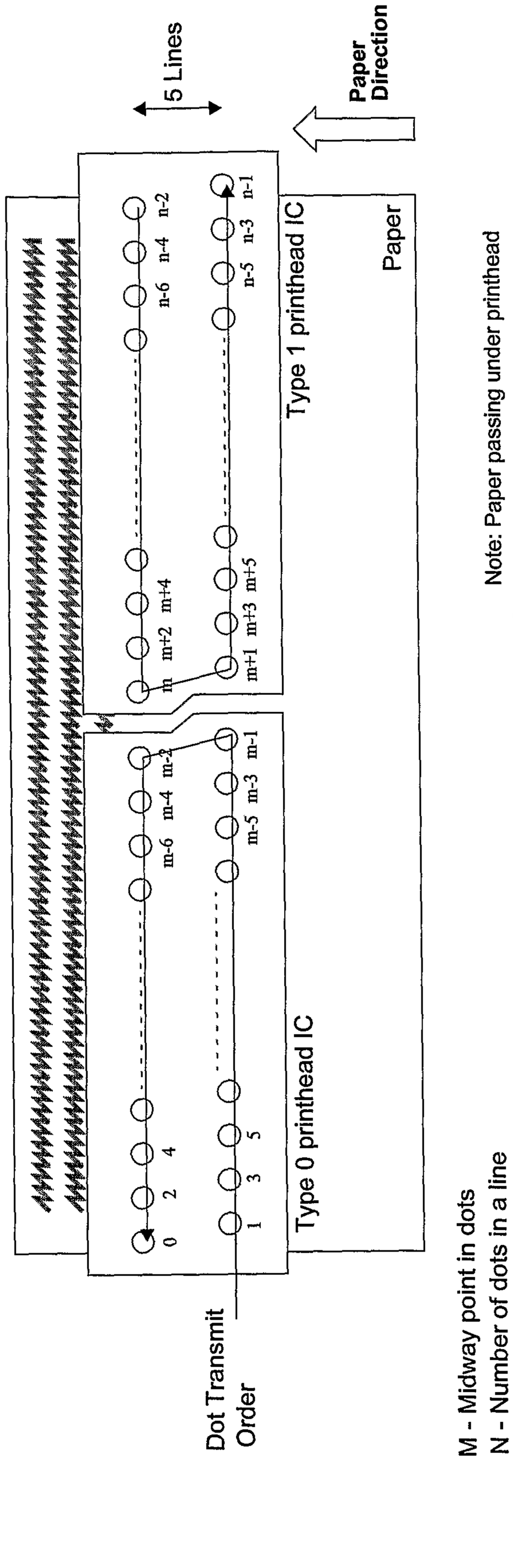

Paper Direction
5 Lines
Type 1 printhead IC
Paper
n-6 n-4 n-2
n-5 n-3 n-1
m-6 m-4 m-2
m-5 m-3 m-1
m m+2 m+4
m+1 m+3 m+5
Type 0 printhead IC
0 2 4
1 3 5
Dot Transmit Order
Note: Paper passing under printhead
M - Midway point in dots
N - Number of dots in a line

FIG. 280

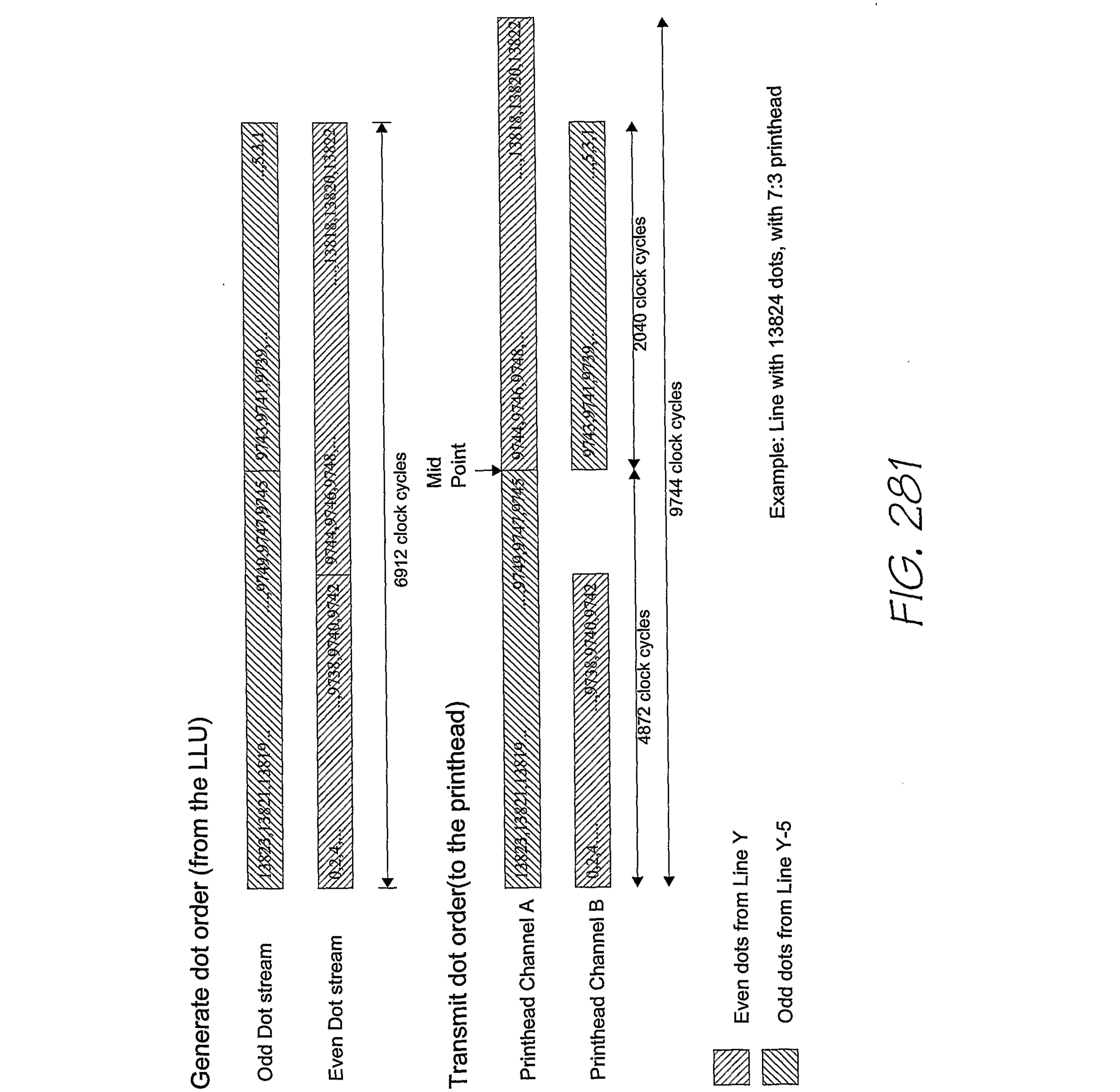

Generate dot order (from the LLU)
Odd Dot stream
Even Dot stream
6912 clock cycles
Mid Point
Transmit dot order(to the printhead)
Printhead Channel A
Printhead Channel B
4872 clock cycles
9744 clock cycles
2040 clock cycles
Example: Line with 13824 dots, with 7:3 printhead
Even dots from Line Y
Odd dots from Line Y-5

FIG. 281


Generate dot order (from the LLU)
Odd Dot stream
9743,9741,9739,...,5,3,1
Even Dot stream
0,2,4,...,9738,9740,9742
4872 clock cycles
Transmit dot order(to the printhead)
Mid Point
Printhead Channel A
9743,9741,9739,...,5,3,1
0,2,4,...,9738,9740,9742
Printhead Channel B
4872 clock cycles
9744 clock cycles
Example: Line with 9744 dots, with 7:0 printhead
Even dots from Line Y
Odd dots from Line Y-5

FIG. 282

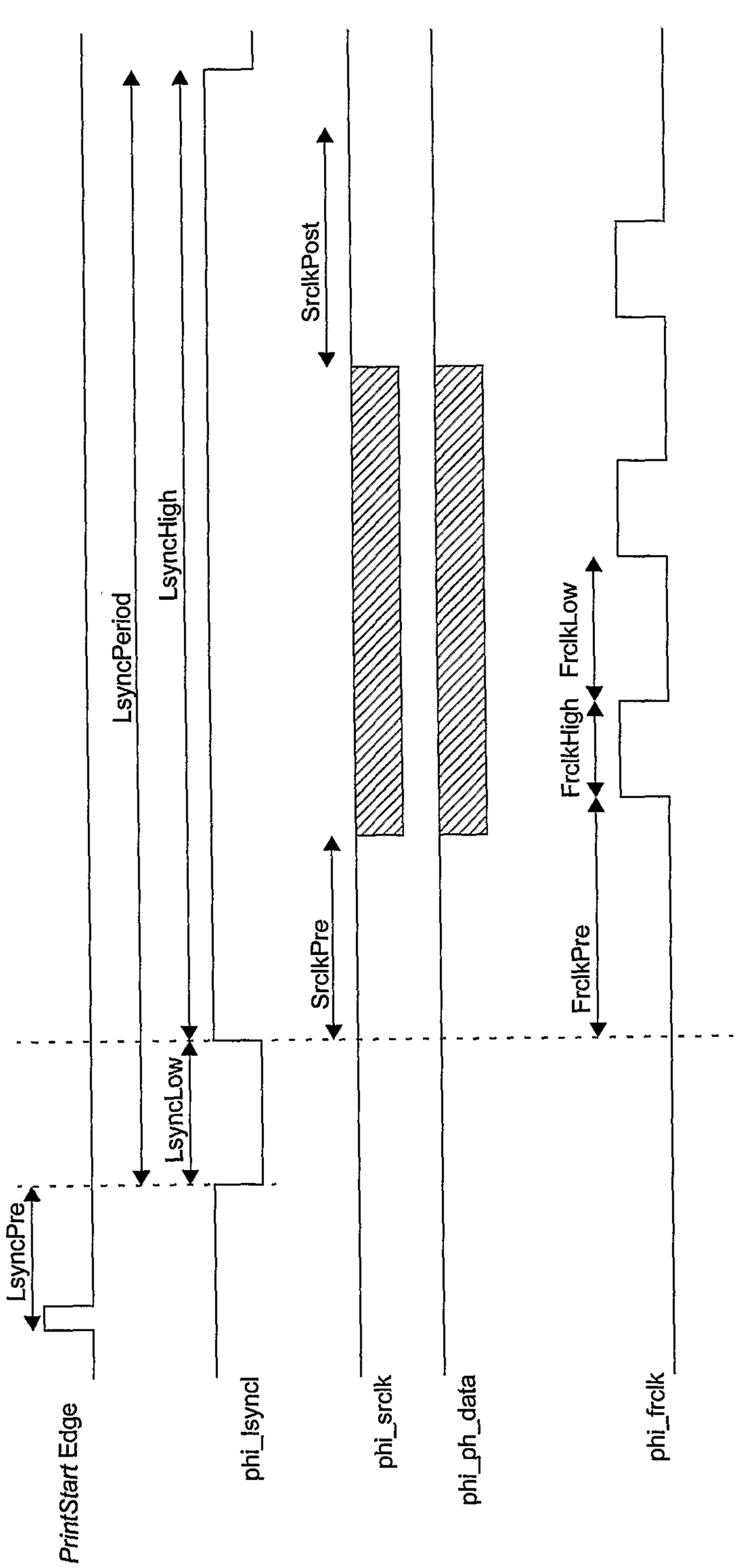

SrclkPost
LsyncHigh
LsyncPeriod
FrclkLow
FrclkHigh
SrclkPre
FrclkPre
LsyncLow
LsyncPre
PrintStart Edge
phi_lsyncl
phi_srclk
phi_ph_data
phi_frclk

FIG. 283

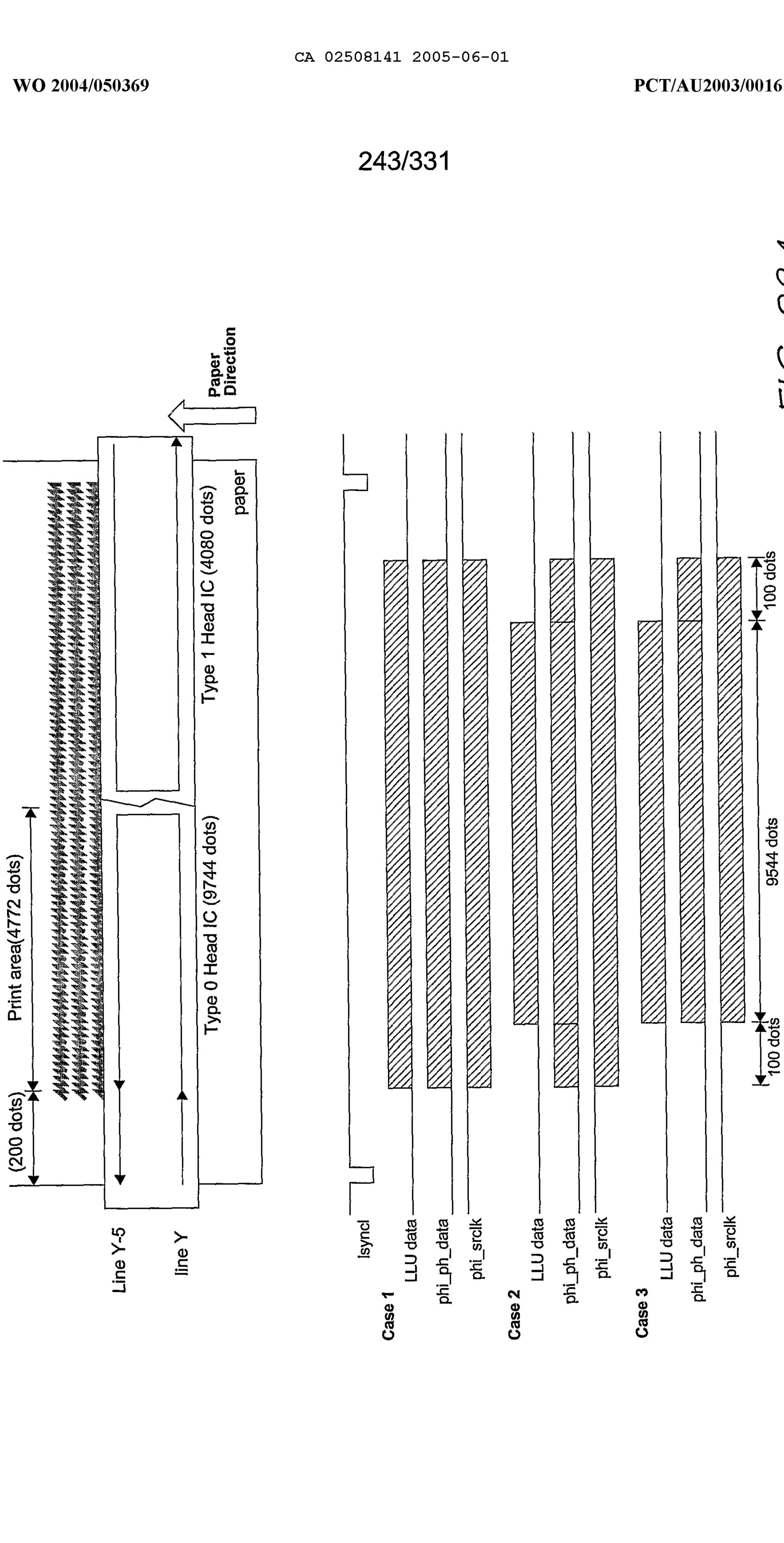

Paper Direction
Type 1 Head IC (4080 dots)
paper
Print area(4772 dots)
(200 dots)
Type 0 Head IC (9744 dots)
Line Y-5
line Y
lsyncl
Case 1
LLU data
phi_ph_data
phi_srclk
Case 2
LLU data
phi_ph_data
phi_srclk
Case 3
LLU data
phi_ph_data
phi_srclk
100 dots
9544 dots
100 dots

FIG. 284

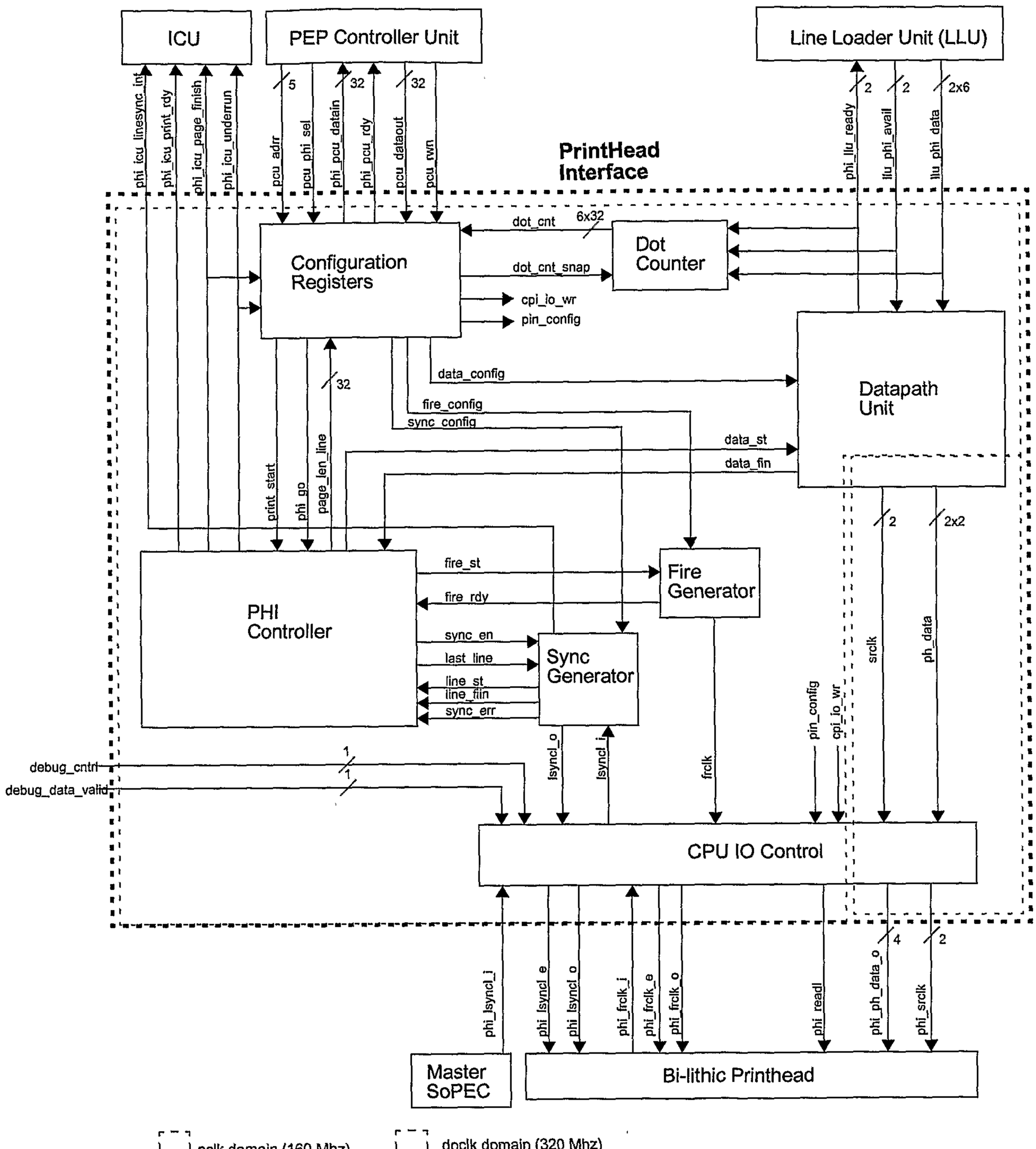

ICU
PEP Controller Unit
Line Loader Unit (LLU)
phi_icu_linesync_int
phi_icu_print_rdy
phi_icu_page_finish
phi_icu_underrun
pcu_adrr
pcu_phi_sel
phi_pcu_datain
phi_pcu_rdy
pcu_dataout
pcu_rwn
phi_llu_ready
llu_phi_avail
llu_phi_data
5
32
32
2
2
2x6
PrintHead Interface
Configuration Registers
dot_cnt
6x32
Dot Counter
dot_cnt_snap
cpi_io_wr
pin_config
data_config
fire_config
sync_config
32
Datapath Unit
data_st
data_fin
print_start
phi_go
page_len_line
2
2x2
fire_st
fire_rdy
Fire Generator
PHI Controller
sync_en
last_line
line_st
line_fiin
sync_err
Sync Generator
lsyncl_o
lsyncl_i
frclk
pin_config
cpi_io_wr
srclk
ph_data
debug_cntrl
debug_data_valid
1
1
CPU IO Control
phi_lsyncl_i
phi_lsyncl_e
phi_lsyncl_o
phi_frclk_i
phi_frclk_e
phi_frclk_o
phi_readl
phi_ph_data_o
phi_srclk
4
2
Master SoPEC
Bi-lithic Printhead
pclk domain (160 Mhz)
doclk domain (320 Mhz)

FIG. 285

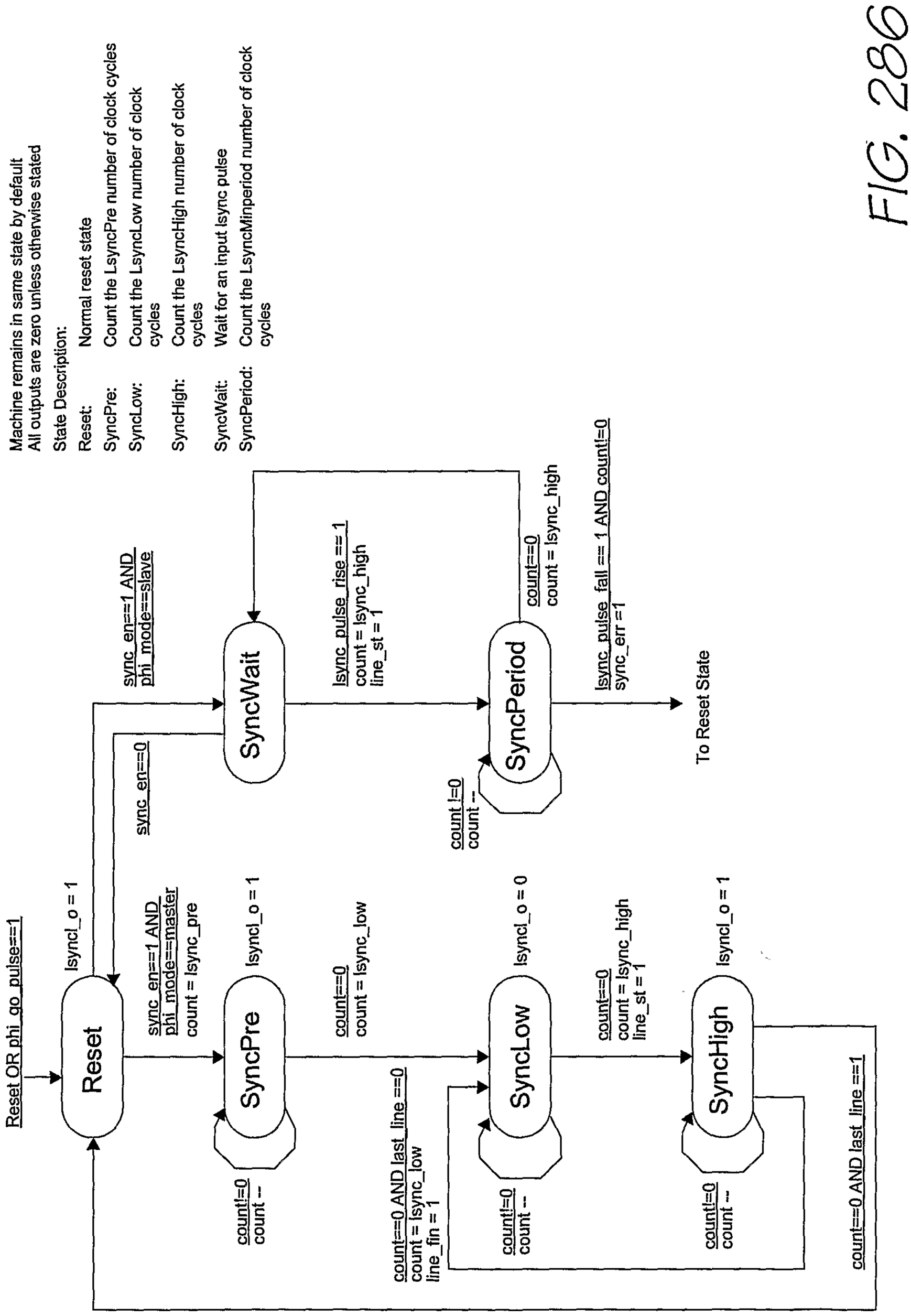

Machine remains in same state by default
All outputs are zero unless otherwise stated
State Description:
Reset: Normal reset state
SyncPre: Count the LsyncPre number of clock cycles
SyncLow: Count the LsyncLow number of clock cycles
SyncHigh: Count the LsyncHigh number of clock cycles
SyncWait: Wait for an input lsync pulse
SyncPeriod: Count the LsyncMinperiod number of clock cycles
Reset
SyncPre
SyncLow
SyncHigh
SyncWait
SyncPeriod
Reset OR phi_go_pulse==1
lsync_o = 1
sync_en==1 AND phi_mode==slave
sync_en==0
sync_en==1 AND phi_mode==master
count = lsync_pre
lsync_o = 1
count==0
count = lsync_low
lsync_o = 0
count==0
count = lsync_high
lsync_o = 1
count==0 AND last_line ==0
count = lsync_low
line_fin = 1
count==0 AND last_line ==1
count!=0
count --
lsync_pulse_rise == 1
count = lsync_high
line_st = 1
count==0
count = lsync_high
lsync_pulse_fall == 1 AND count!=0
sync_err =1
To Reset State

FIG. 286

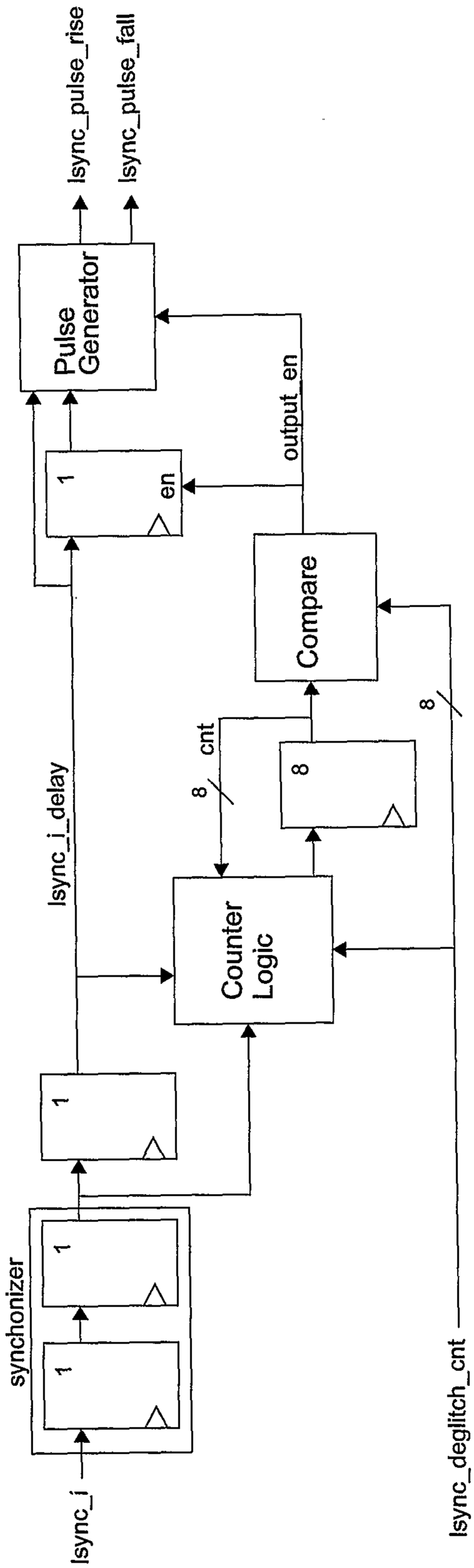

lsync_pulse_rise
lsync_pulse_fall
Pulse Generator
output_en
1
en
Compare
8
cnt
8
8
Counter Logic
lsync_i_delay
1
synchonizer
1
1
lsync_i
lsync_deglitch_cnt

FIG. 287

Machine remains in same state by default
All outputs are zero unless otherwise stated

State Description:

Reset: Normal reset state

FirePre: Count the FrclkPre number of clock cycles, repeat count set to FrclkNum

FireHigh: Count the FrclkHigh number of clock cycles

FireLow: Count the FrclkLow number of clock cycles

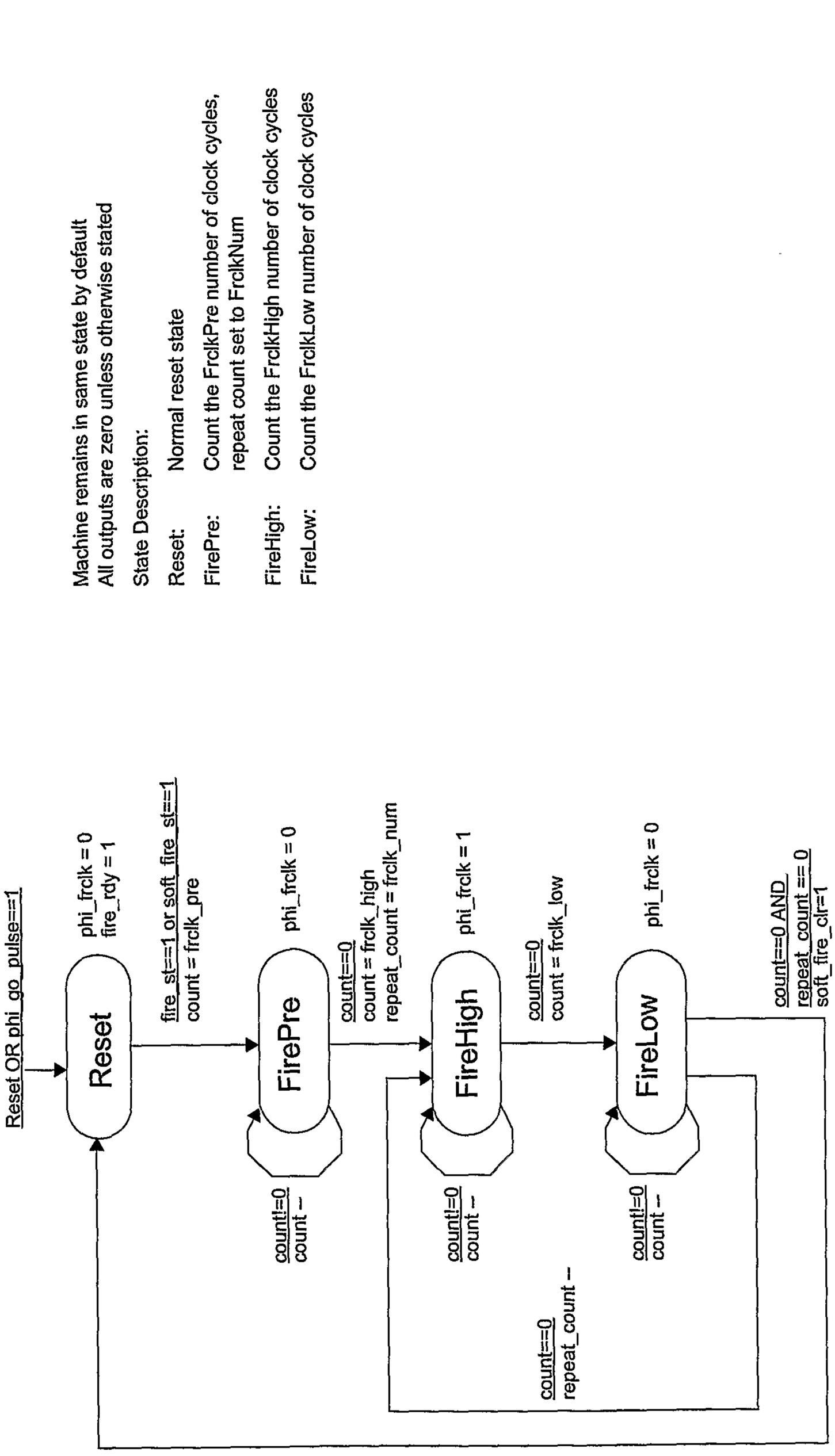


FIG. 288

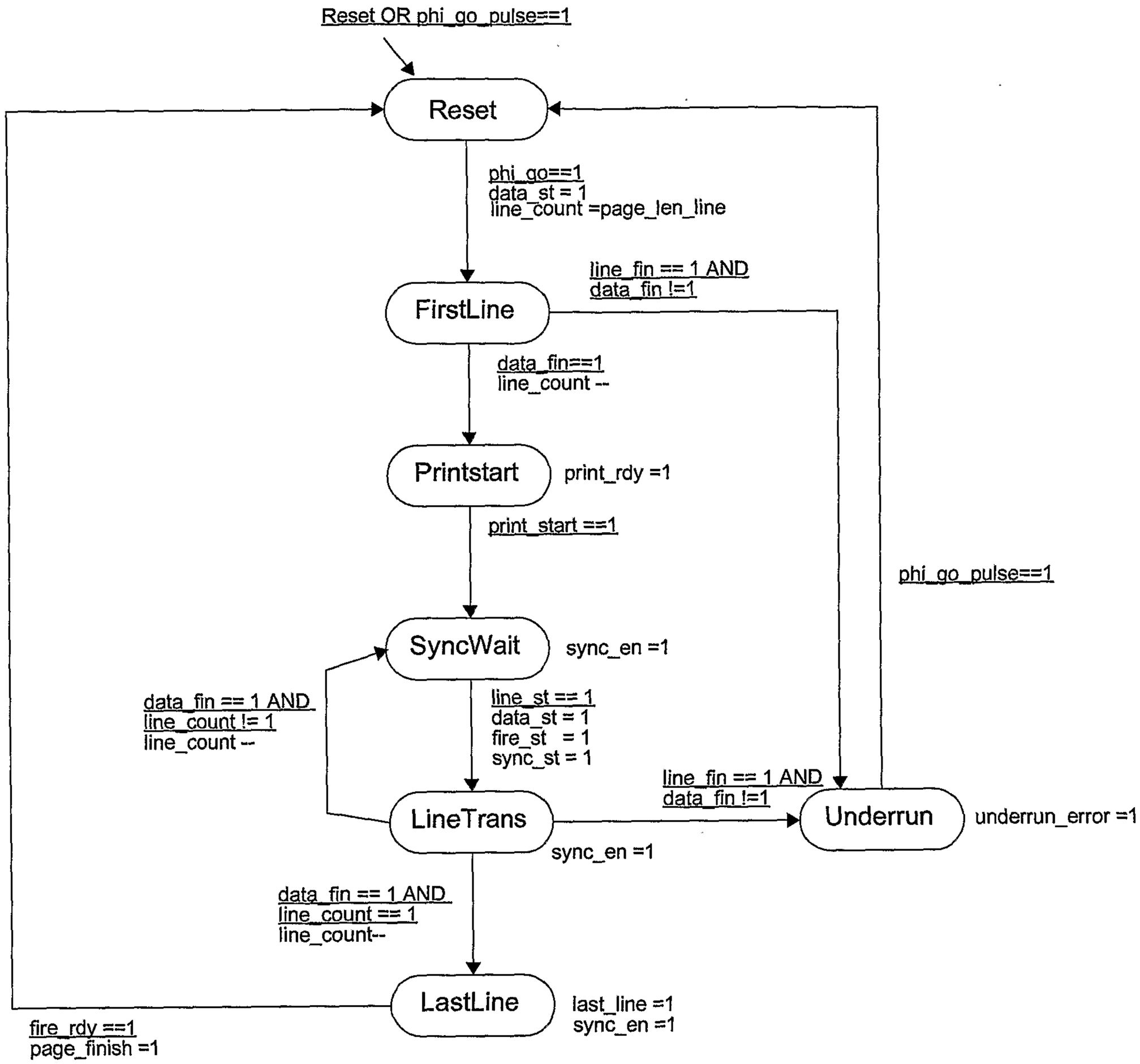

Reset OR phi_go_pulse==1
Reset
phi_go==1
data_st = 1
line_count =page_len_line
FirstLine
line_fin == 1 AND
data_fin !=1
data_fin==1
line_count --
Printstart
print_rdy =1
print_start ==1
phi_go_pulse==1
SyncWait
sync_en =1
data_fin == 1 AND
line_count != 1
line_count --
line_st == 1
data_st = 1
fire_st = 1
sync_st = 1
line_fin == 1 AND
data_fin !=1
LineTrans
Underrun
underrun_error =1
sync_en =1
data_fin == 1 AND
line_count == 1
line_count--
LastLine
last_line =1
sync_en =1
fire_rdy ==1
page_finish =1

FIG. 289

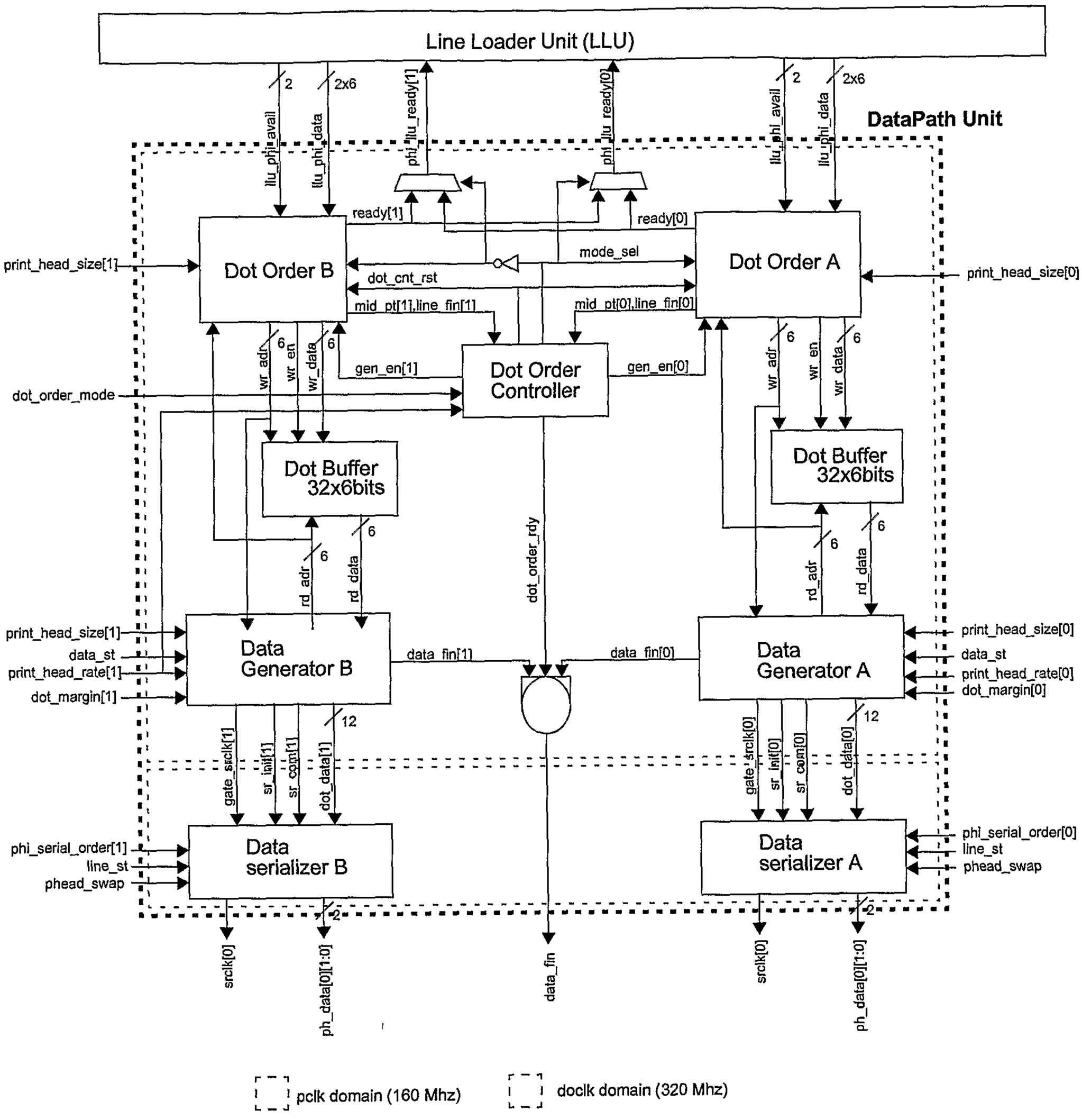

Line Loader Unit (LLU)
DataPath Unit
2
2x6
llu_phi_avail
llu_phi_data
phi_llu_ready[1]
phi_llu_ready[0]
ready[1]
ready[0]
Dot Order B
Dot Order A
print_head_size[1]
print_head_size[0]
mode_sel
dot_cnt_rst
mid_pt[1],line_fin[1]
mid_pt[0],line_fin[0]
6
wr_adr
wr_en
wr_data
gen_en[1]
gen_en[0]
Dot Order Controller
dot_order_mode
Dot Buffer 32x6bits
rd_adr
rd_data
dot_order_rdy
Data Generator B
Data Generator A
data_st
print_head_rate[1]
print_head_rate[0]
dot_margin[1]
dot_margin[0]
data_fin[1]
data_fin[0]
12
gate_srclk[1]
sr_init[1]
sr_com[1]
dot_data[1]
gate_srclk[0]
sr_init[0]
sr_com[0]
dot_data[0]
Data serializer B
Data serializer A
phi_serial_order[1]
phi_serial_order[0]
line_st
phead_swap
srclk[0]
ph_data[0][1:0]
data_fin
pclk domain (160 Mhz)
doclk domain (320 Mhz)

FIG. 290

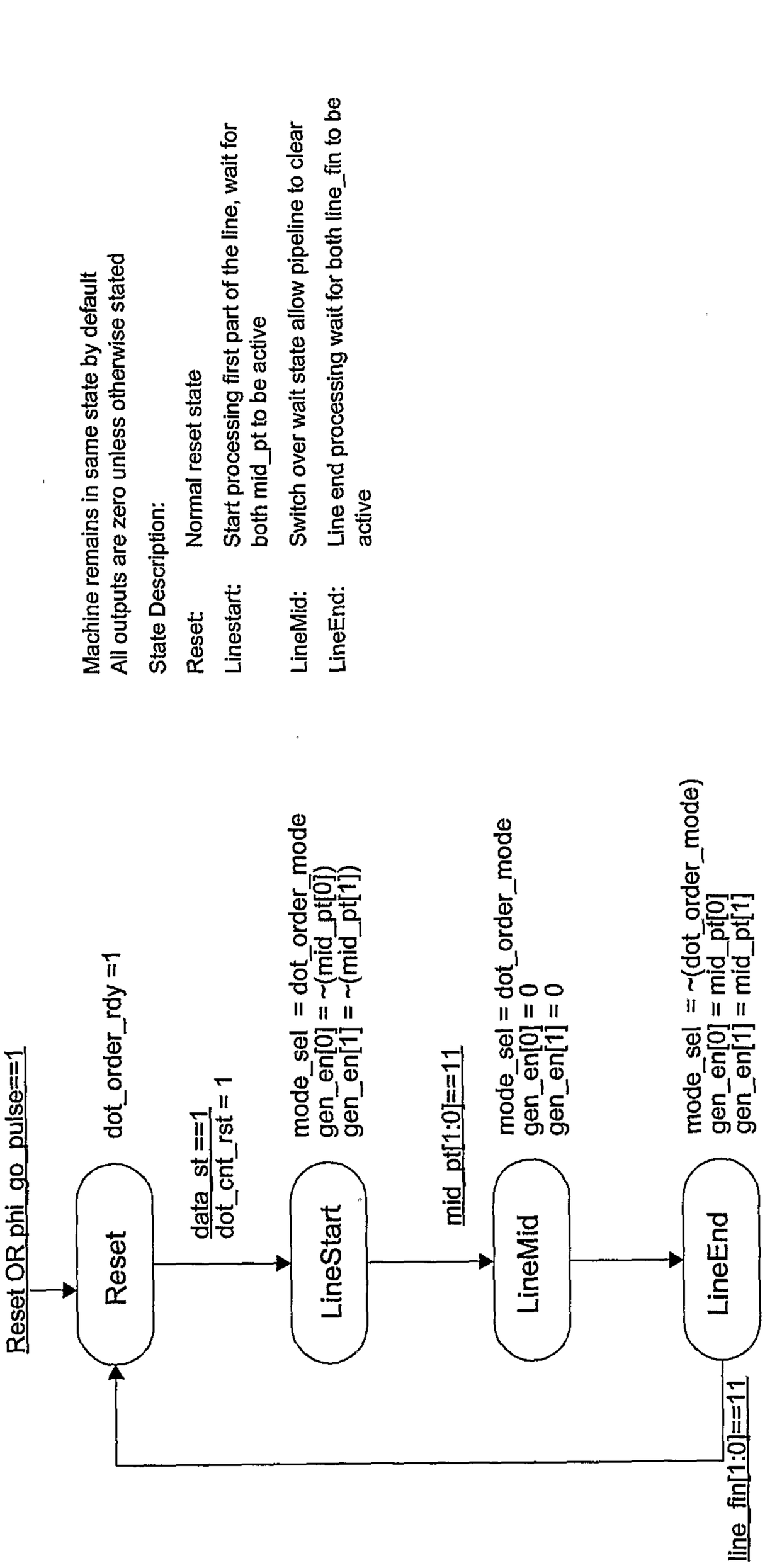

Machine remains in same state by default
All outputs are zero unless otherwise stated
State Description:
Reset: Normal reset state
Linestart: Start processing first part of the line, wait for both mid_pt to be active
LineMid: Switch over wait state allow pipeline to clear
LineEnd: Line end processing wait for both line_fin to be active
Reset OR phi_go_pulse==1
dot_order_rdy =1
mode_sel = dot_order_mode
gen_en[0] = ~(mid_pt[0])
gen_en[1] = ~(mid_pt[1])
data_st ==1
dot_cnt_rst = 1
mid_pt[1:0]==11
mode_sel = dot_order_mode
gen_en[0] = 0
gen_en[1] = 0
mode_sel = ~(dot_order_mode)
gen_en[0] = mid_pt[0]
gen_en[1] = mid_pt[1]
Reset
LineStart
LineMid
LineEnd
line_fin[1:0]==11

FIG. 291

Machine remains in same state by default
All outputs are zero unless otherwise stated

State Description:

Reset: Normal reset state

SrclkPre: Count the SrclkPre number of clock cycles

DataGen1: Read Line Dot data from buffer

DataGen2: Read Line Dot data from buffer

DataGen3: Data gen wait state

Wait1: Data rate control wait state

Wait2: Data rate control wait state

Wait3: Data rate control wait state

MarginGen1:Generate *DotMargin* number of dots

MarginGen2:Generate *DotMargin* number of dots

MarginGen3:Generate margin wait state

SrclkPost: Wait for SrclkPost number of cycles

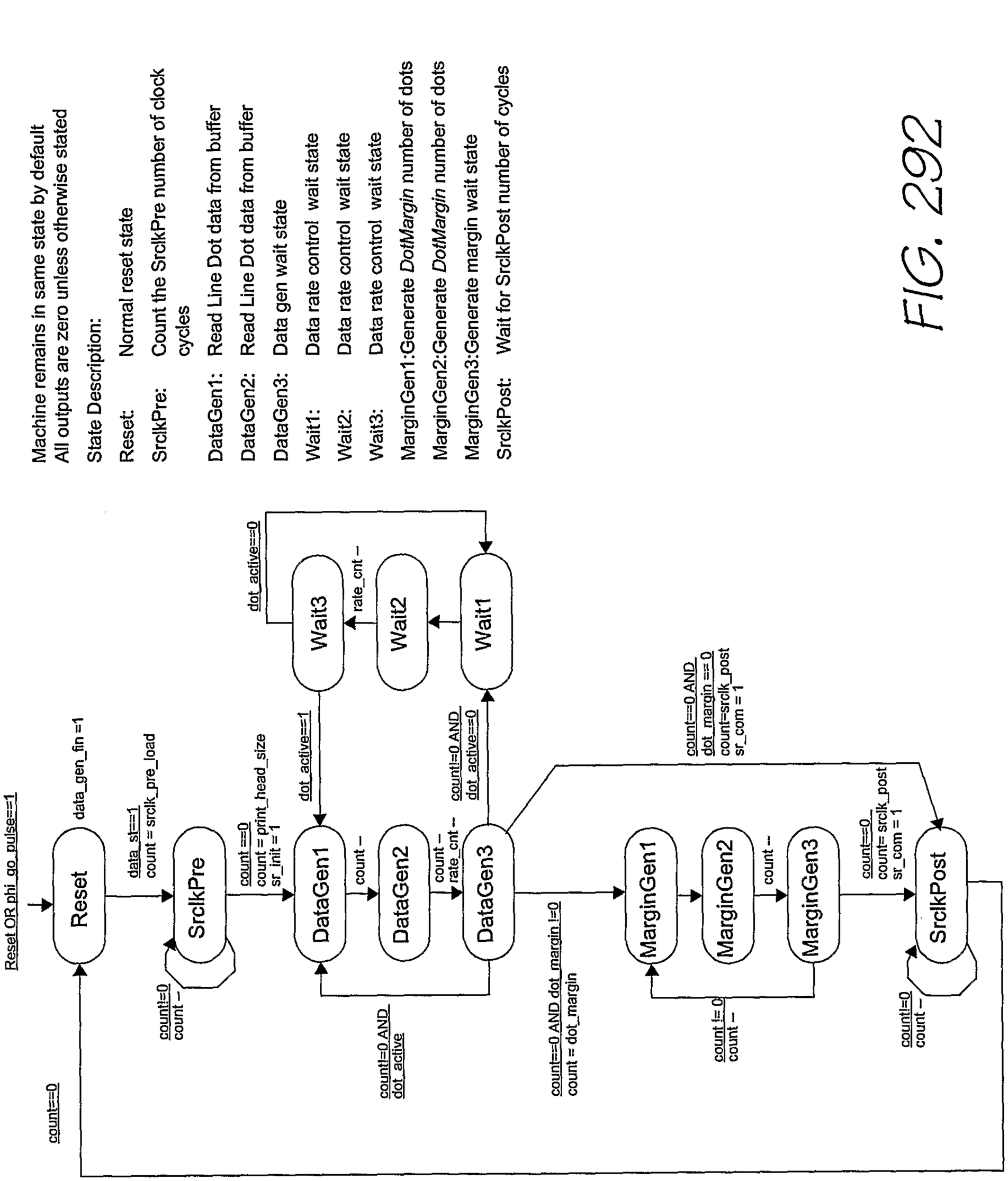


FIG. 292

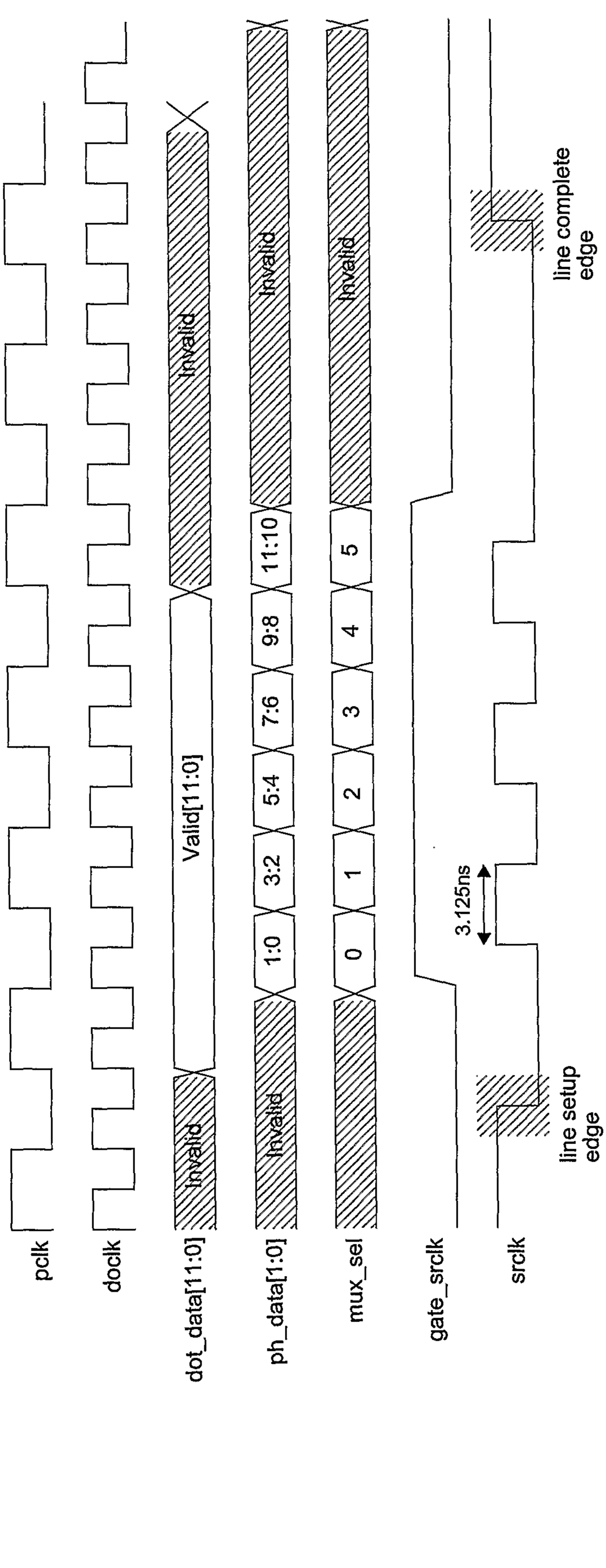

pclk
doclk
dot_data[11:0]
ph_data[1:0]
mux_sel
gate_srclk
srclk
Invalid
Valid[11:0]
1:0
3:2
5:4
7:6
9:8
11:10
0
1
2
3
4
5
3.125ns
line setup edge
line complete edge

FIG. 293

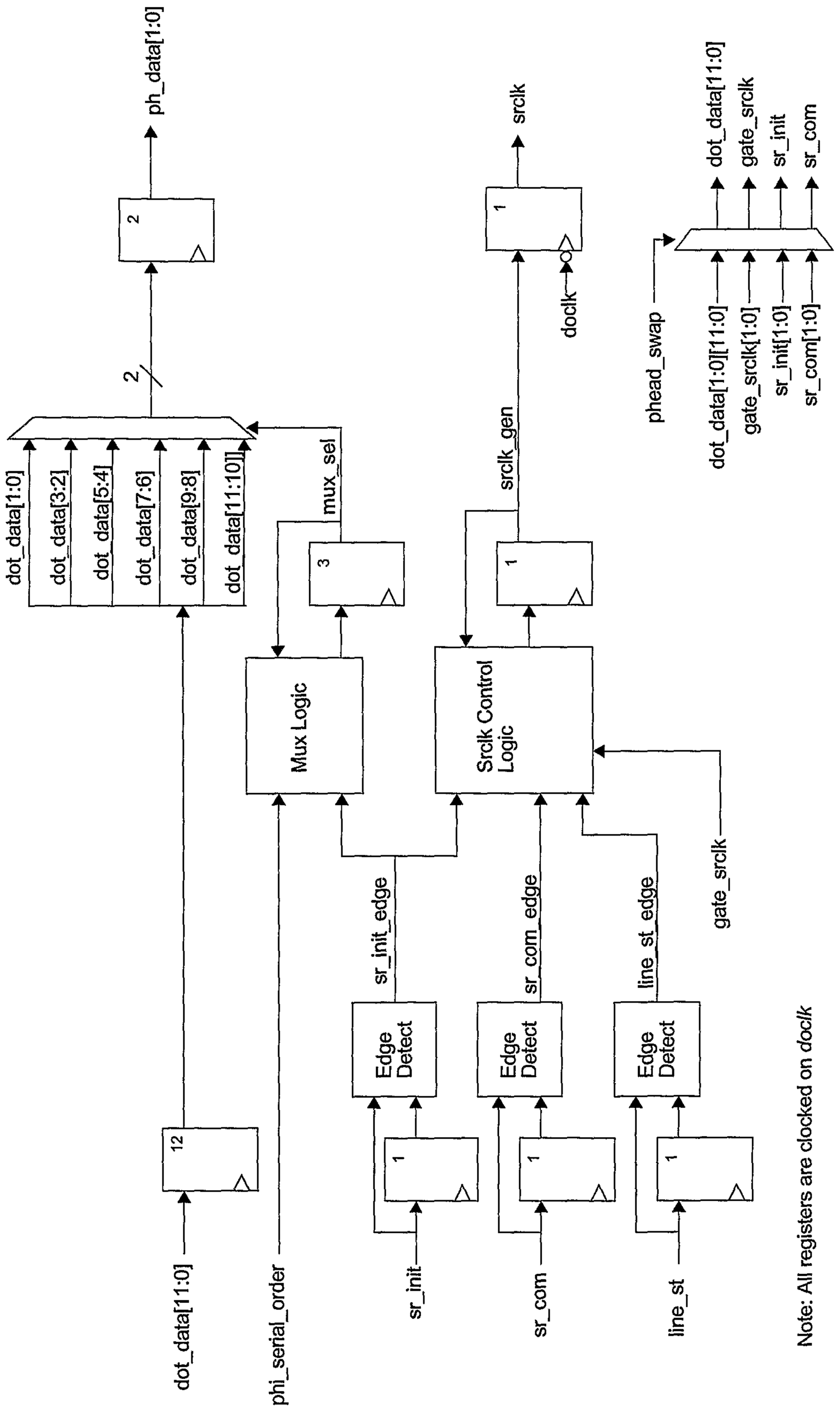

dot_data[1:0]
dot_data[3:2]
dot_data[5:4]
dot_data[7:6]
dot_data[9:8]
dot_data[11:10]]
2
2
ph_dat
mux_sel
3
Mux Logic
srclk_gen
1
srclk
doclk
Srclk Control Logic
phead_swap
dot_data[1:0][11:0]
gate_srclk[1:0]
sr_init[1:0]
sr_com[1:0]
dot_data[11:
gate_srclk
sr_init
sr_com
gate_srclk
sr_init_edge
sr_com_edge
line_st_edge
Edge Detect
Edge Detect
Edge Detect
dot_data[11:0]
phi_serial_order
12
sr_init
sr_com
line_st
Note: All registers are clocked on doclk

FIG. 294

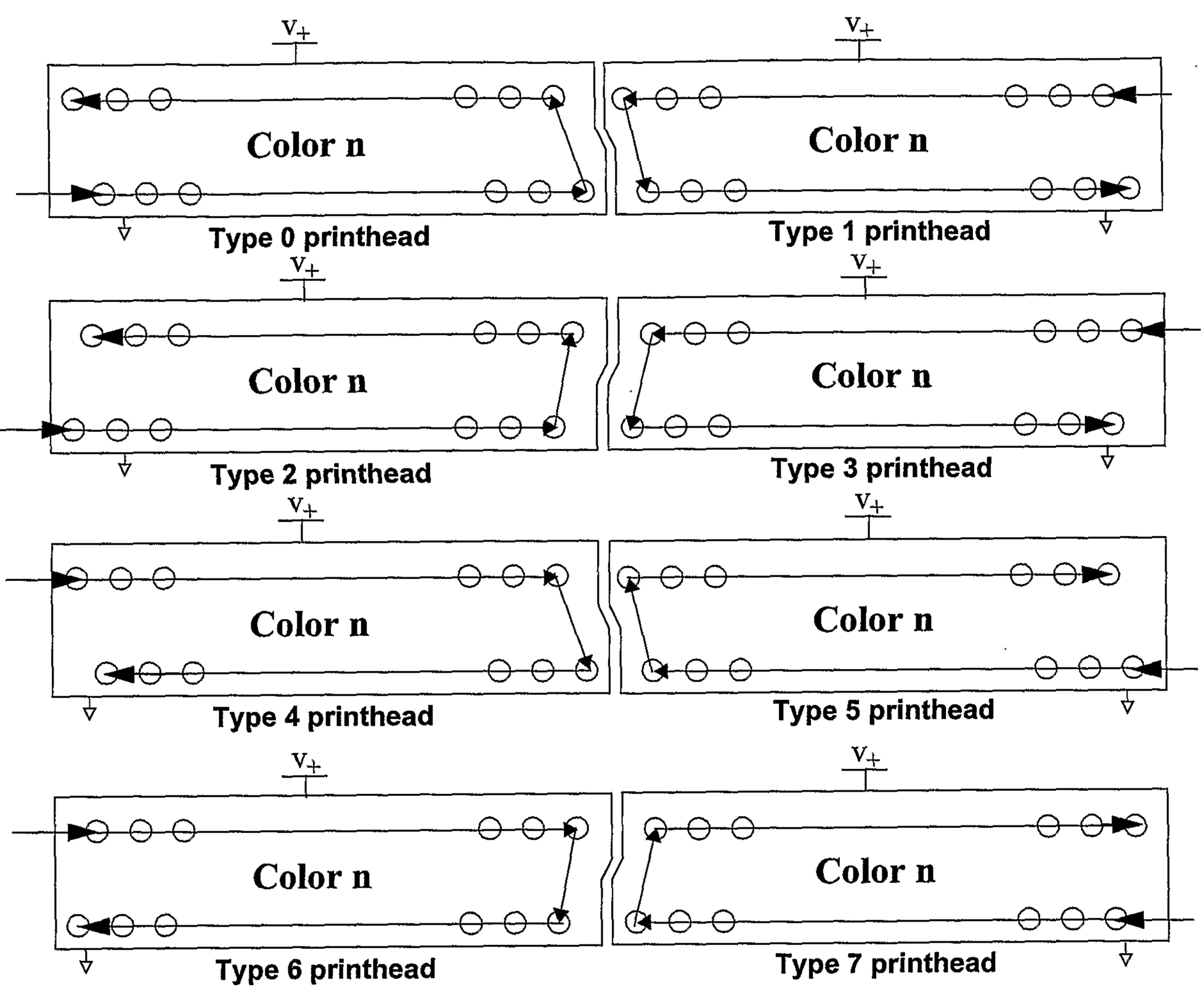

V+
Color n
Type 0 printhead
V+
Color n
Type 1 printhead
V+
Color n
Type 2 printhead
V+
Color n
Type 3 printhead
V+
Color n
Type 4 printhead
V+
Color n
Type 5 printhead
V+
Color n
Type 6 printhead
V+
Color n
Type 7 printhead

FIG. 295

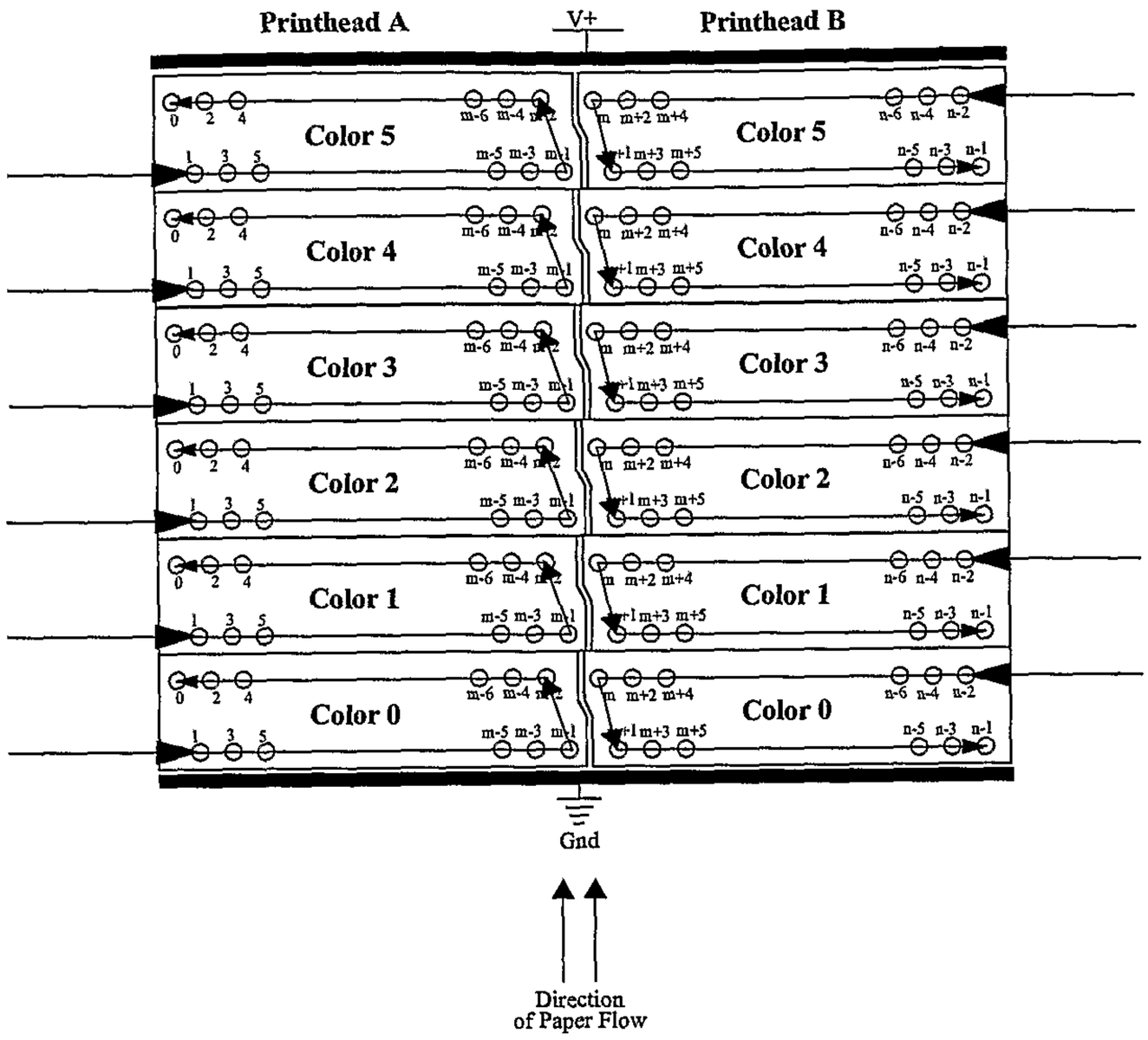

Printhead A
V+
Printhead B
Color 5
Color 4
Color 3
Color 2
Color 1
Color 0
0 2 4
1 3 5
m-6 m-4 m-2
m-5 m-3 m-1
m m+2 m+4
m+1 m+3 m+5
n-6 n-4 n-2
n-5 n-3 n-1
Gnd
Direction
of Paper Flow

FIG. 296

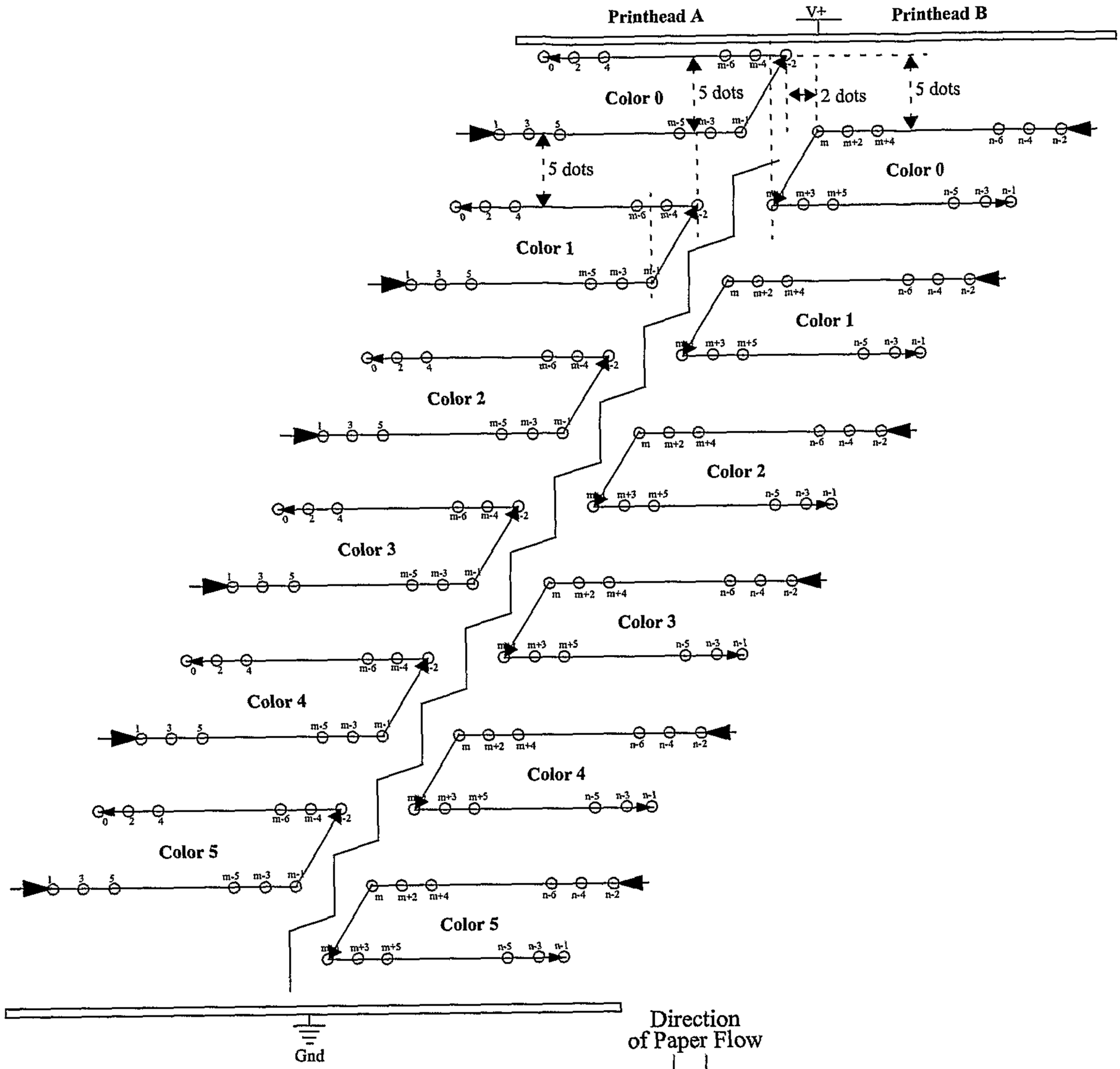

Printhead A
V+
Printhead B
Color 0
5 dots
2 dots
5 dots
5 dots
Color 0
Color 1
Color 1
Color 2
Color 2
Color 3
Color 3
Color 4
Color 4
Color 5
Color 5
Gnd
Direction of Paper Flow
The printheads are facing downwards.
The ink is being shot down onto the page.

FIG. 297

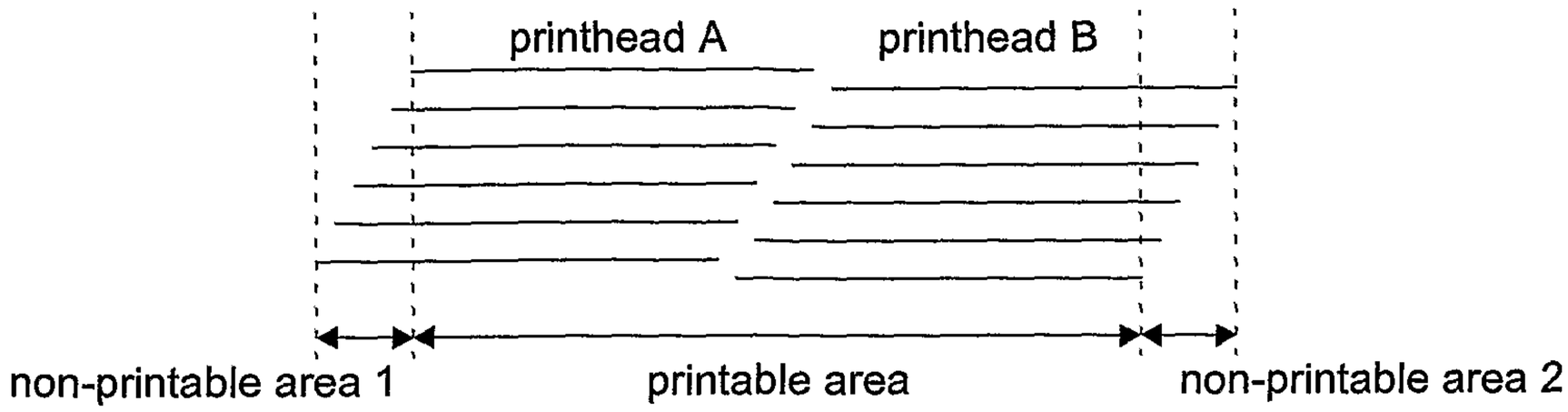

printhead A
printhead B
non-printable area 1
printable area
non-printable area 2

FIG. 298

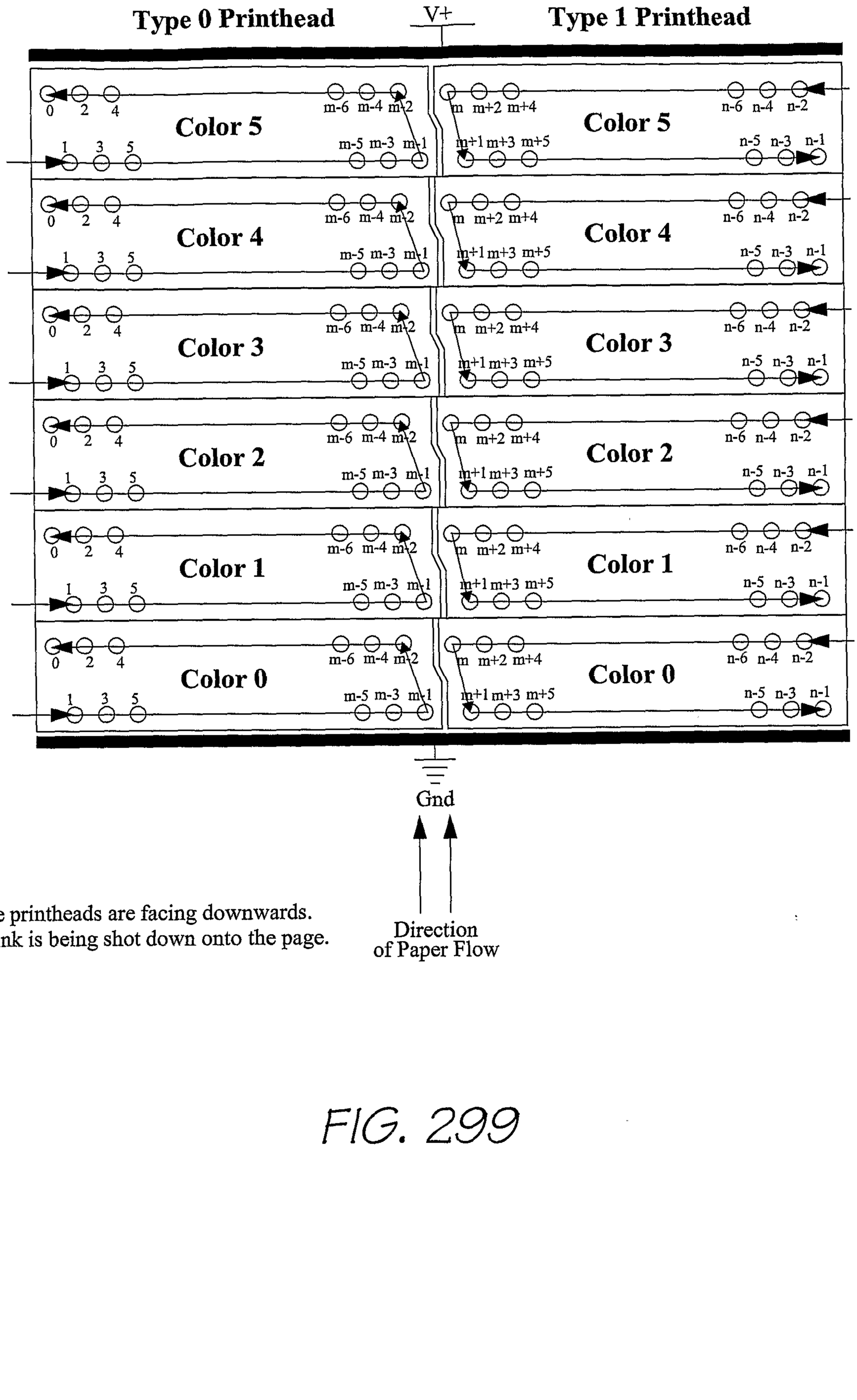

Type 0 Printhead
V+
Type 1 Printhead
Color 5
Color 4
Color 3
Color 2
Color 1
Color 0
0 2 4
1 3 5
m-6 m-4 m-2
m-5 m-3 m-1
m m+2 m+4
m+1 m+3 m+5
n-6 n-4 n-2
n-5 n-3 n-1
Gnd
The printheads are facing downwards.
The ink is being shot down onto the page.
Direction
of Paper Flow

FIG. 299

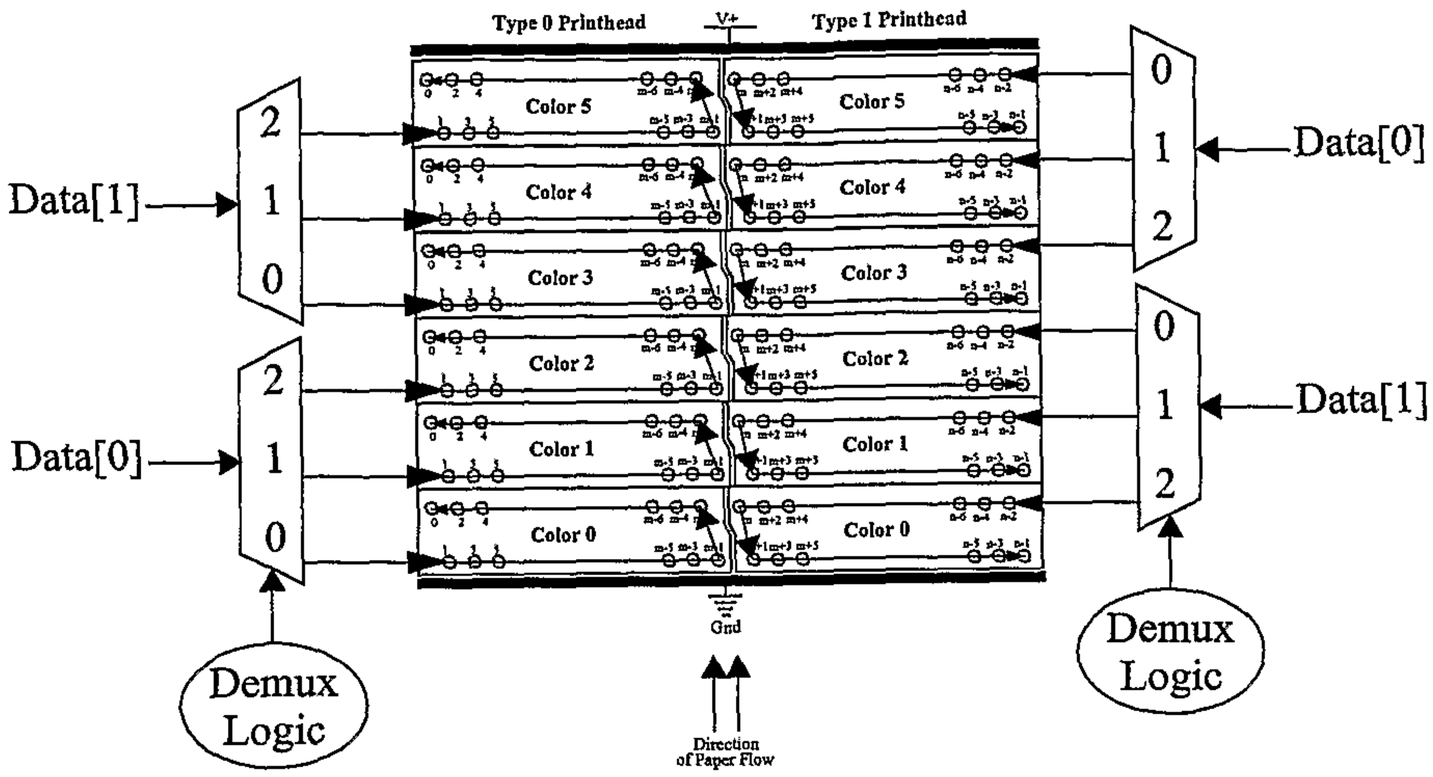


FIG. 300

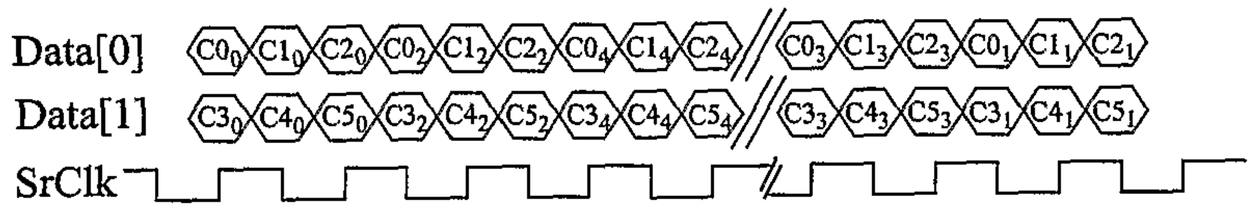


FIG. 301

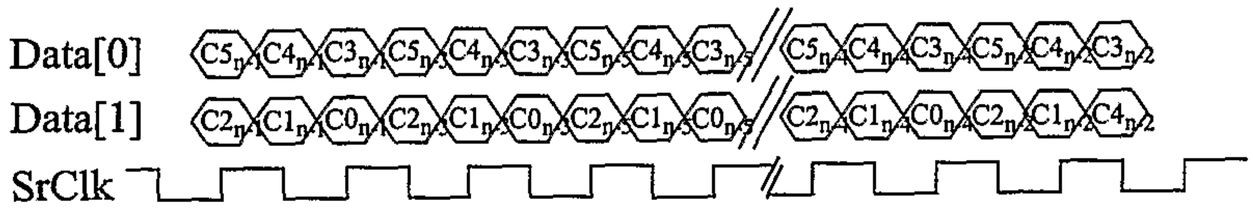


FIG. 302

The printheads are facing downwards.
The ink is being shot down onto the page.

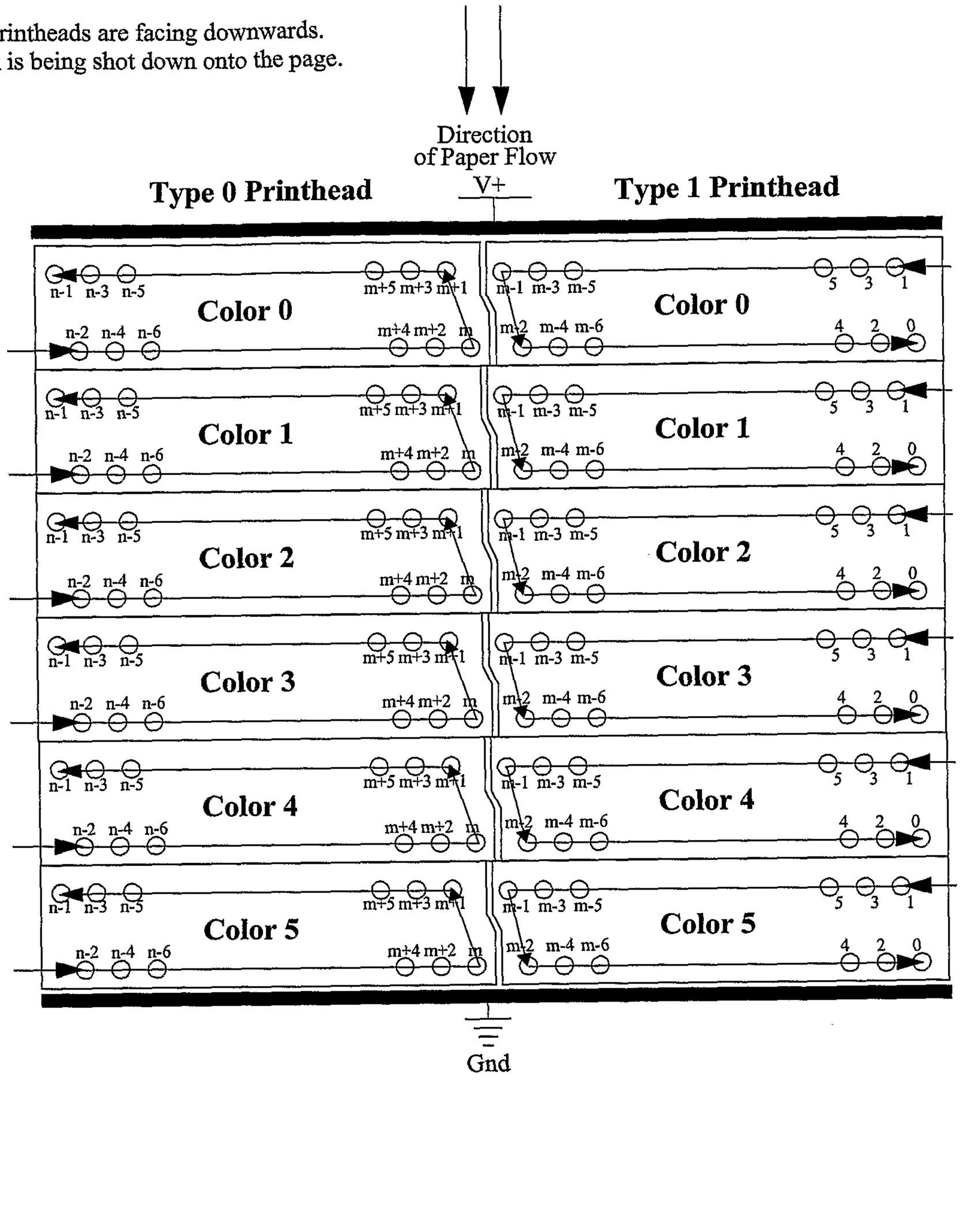


FIG. 303

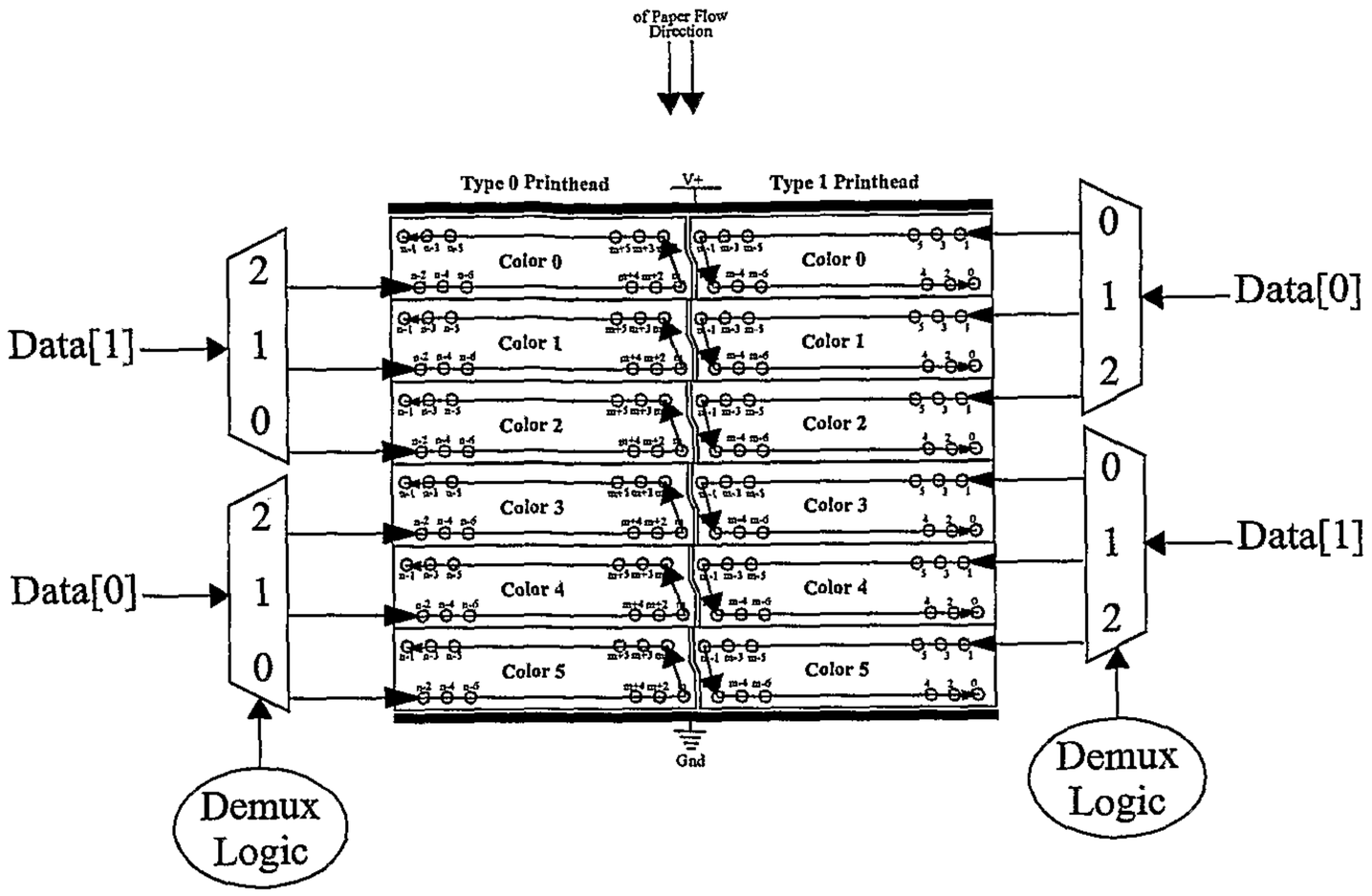


FIG. 304

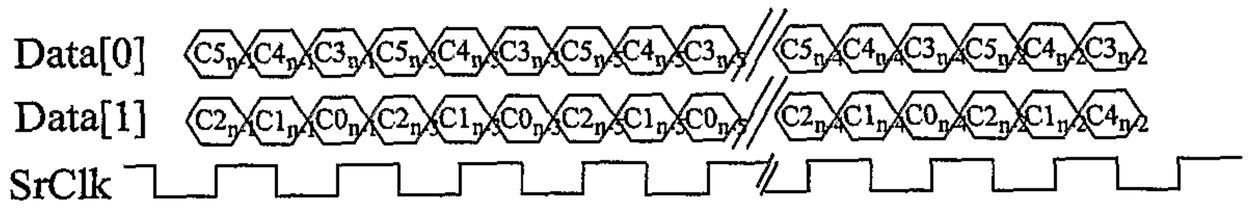


FIG. 305

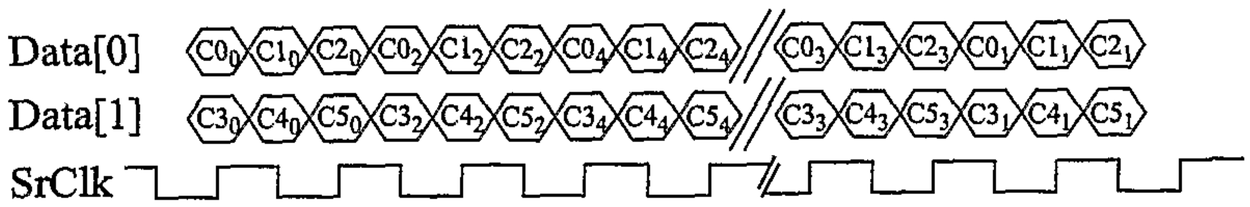


FIG. 306

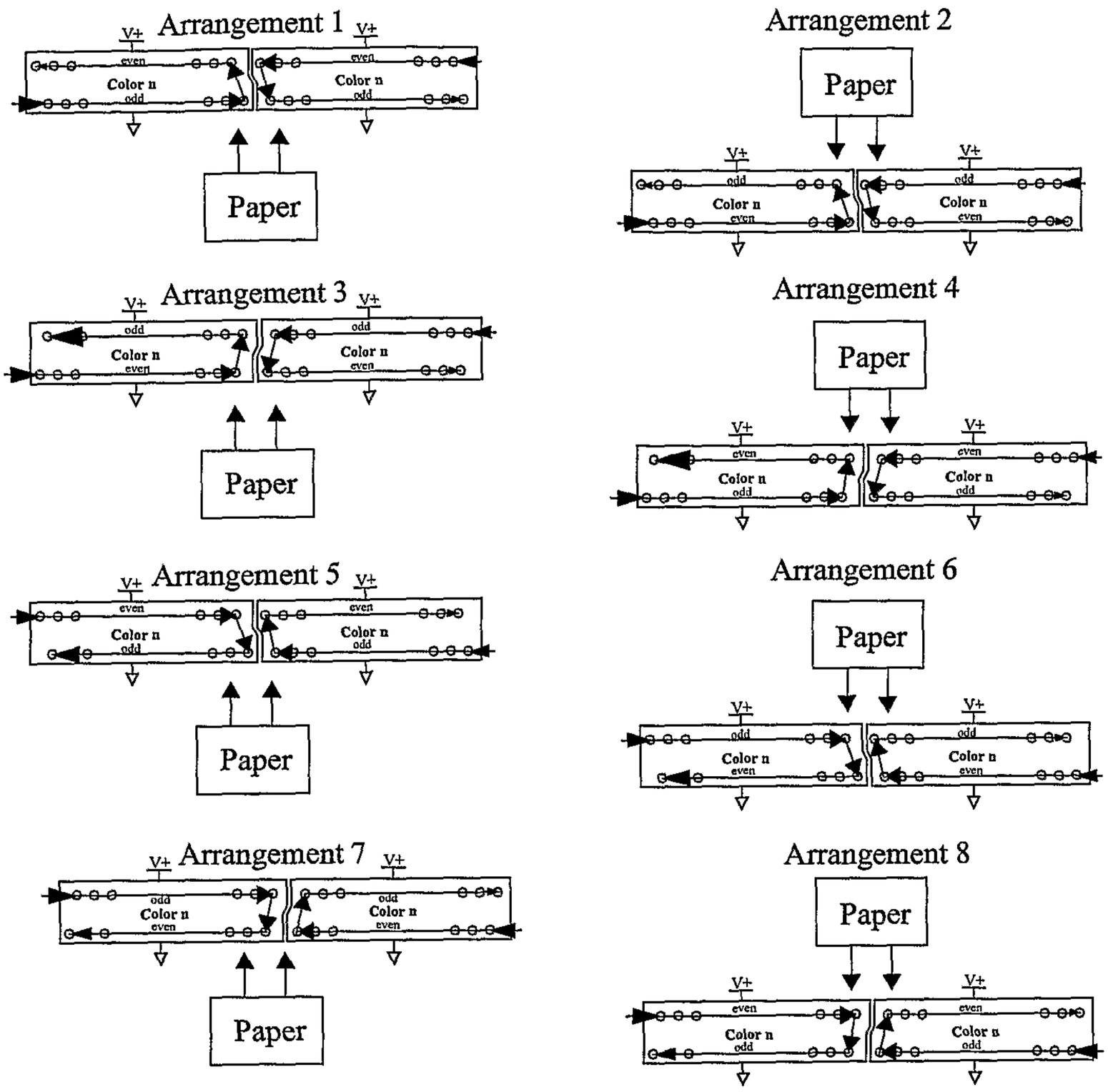

Arrangement 1
V+
even
Color n
odd
Paper
Arrangement 2
Paper
odd
Color n
even
Arrangement 3
odd
Color n
even
Paper
Arrangement 4
Paper
even
Color n
odd
Arrangement 5
even
Color n
odd
Paper
Arrangement 6
Paper
odd
Color n
even
Arrangement 7
odd
Color n
even
Paper
Arrangement 8
Paper
even
Color n
odd

FIG. 307

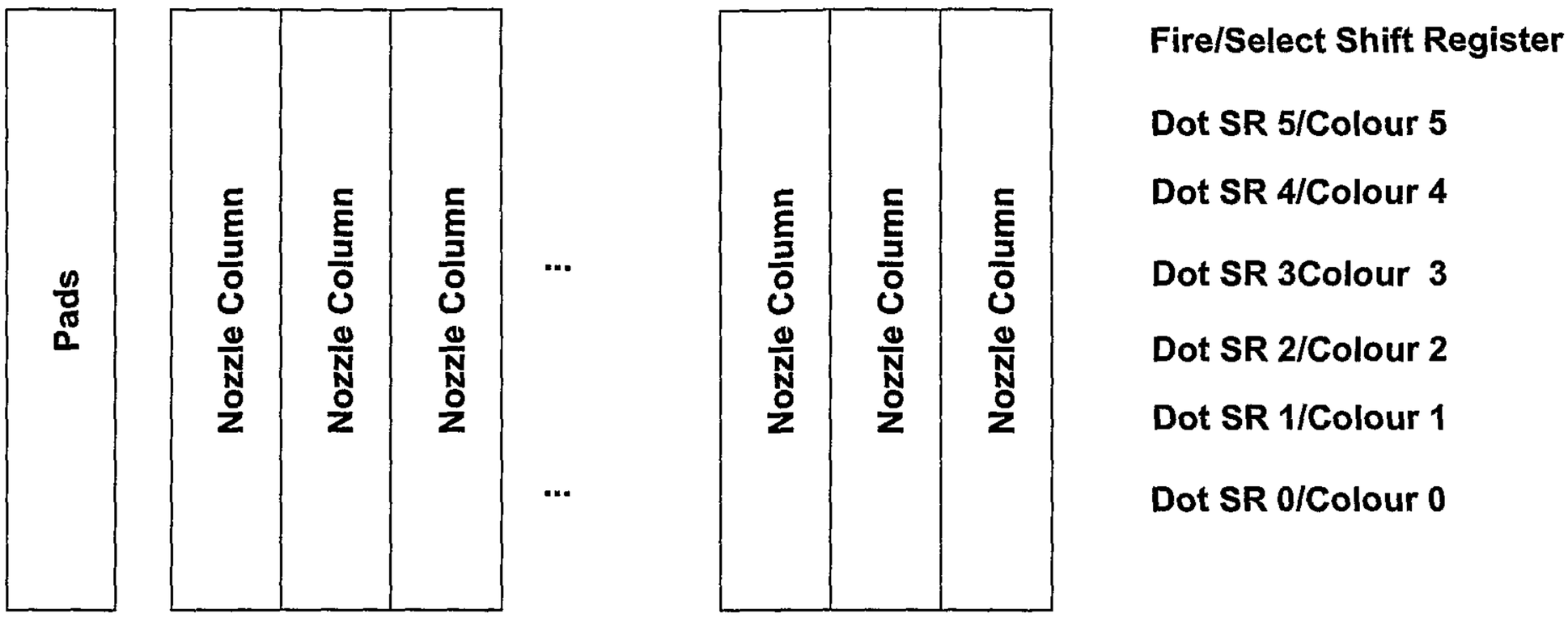

Pads
Nozzle Column
Nozzle Column
Nozzle Column
Nozzle Column
Nozzle Column
Nozzle Column
Fire/Select Shift Register
Dot SR 5/Colour 5
Dot SR 4/Colour 4
Dot SR 3Colour 3
Dot SR 2/Colour 2
Dot SR 1/Colour 1
Dot SR 0/Colour 0

FIG. 308

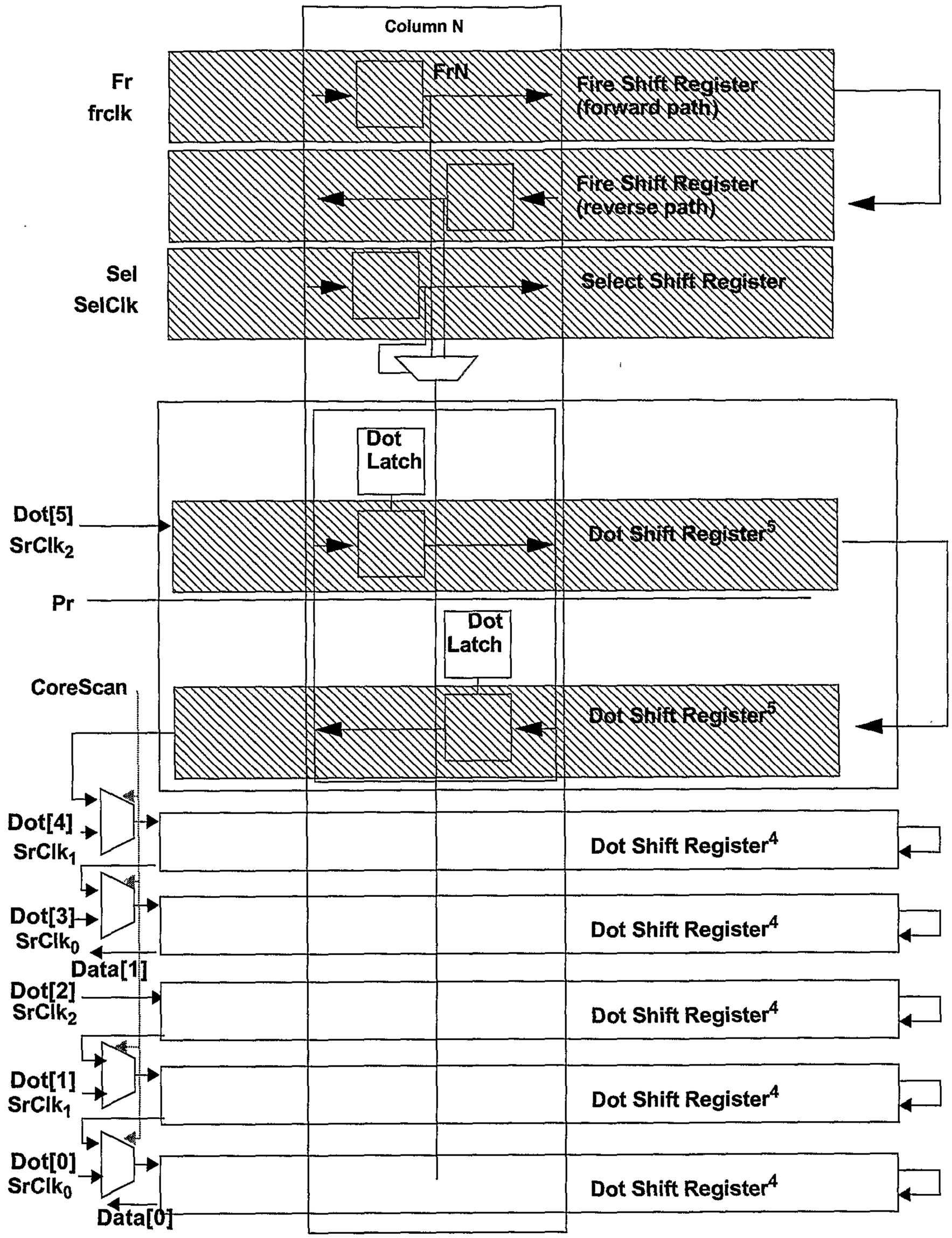
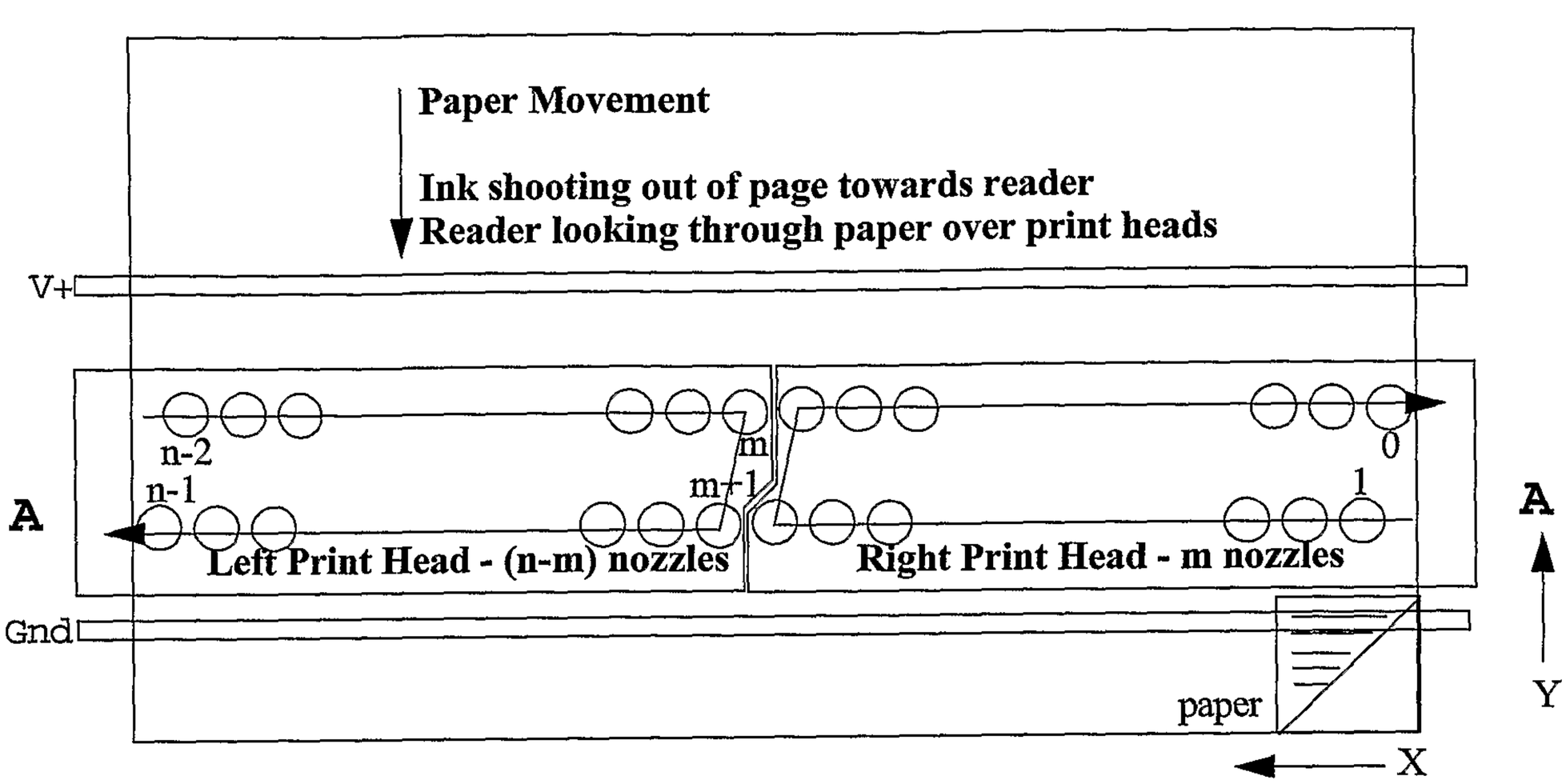

Column N
Fr
frclk
FrN
Fire Shift Register (forward path)
Fire Shift Register (reverse path)
Sel
SelClk
Select Shift Register
Dot Latch
Dot[5]
SrClk2
Dot Shift Register5
Pr
Dot Latch
CoreScan
Dot Shift Register5
Dot[4]
SrClk1
Dot Shift Register4
Dot[3]
SrClk0
Dot Shift Register4
Data[1]
Dot[2]
SrClk2
Dot Shift Register4
Dot[1]
SrClk1
Dot Shift Register4
Dot[0]
SrClk0
Dot Shift Register4
Data[0]

FIG. 309



Section A-A Through Even nozzles

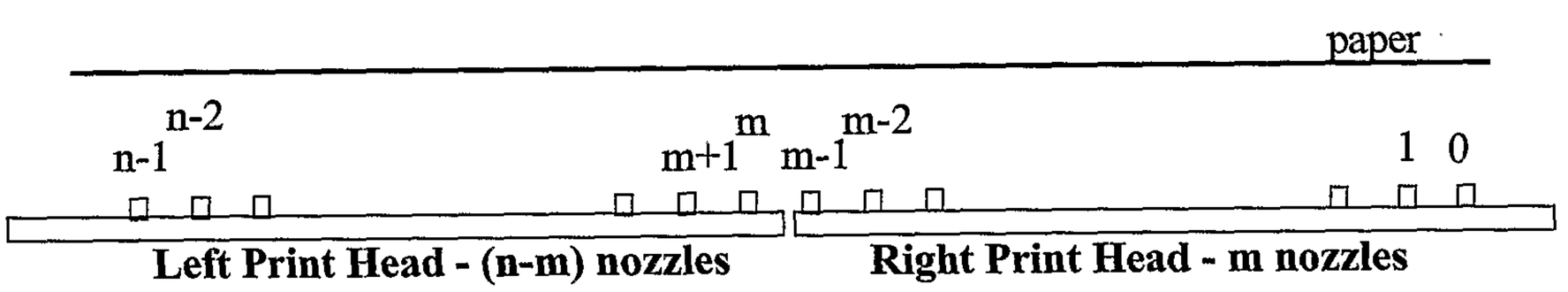


FIG. 310

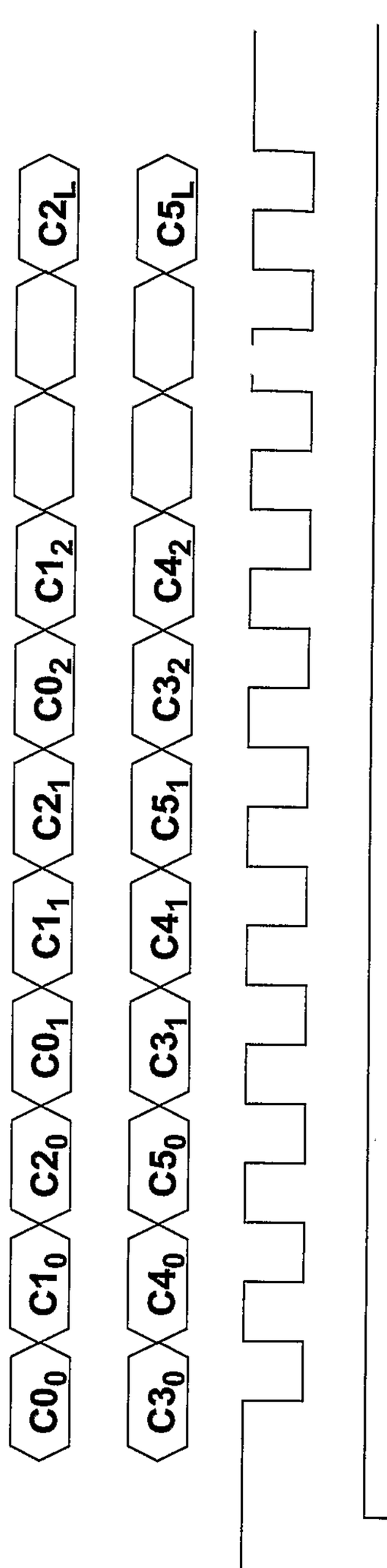

C0₀
C1₀
C2₀
C0₁
C1₁
C2₁
C0₂
C1₂
C2L
C3₀
C4₀
C5₀
C3₁
C4₁
C5₁
C3₂
C4₂
C5L
Data[0]
Data[1]
SrClk
LSyncL

FIG. 311

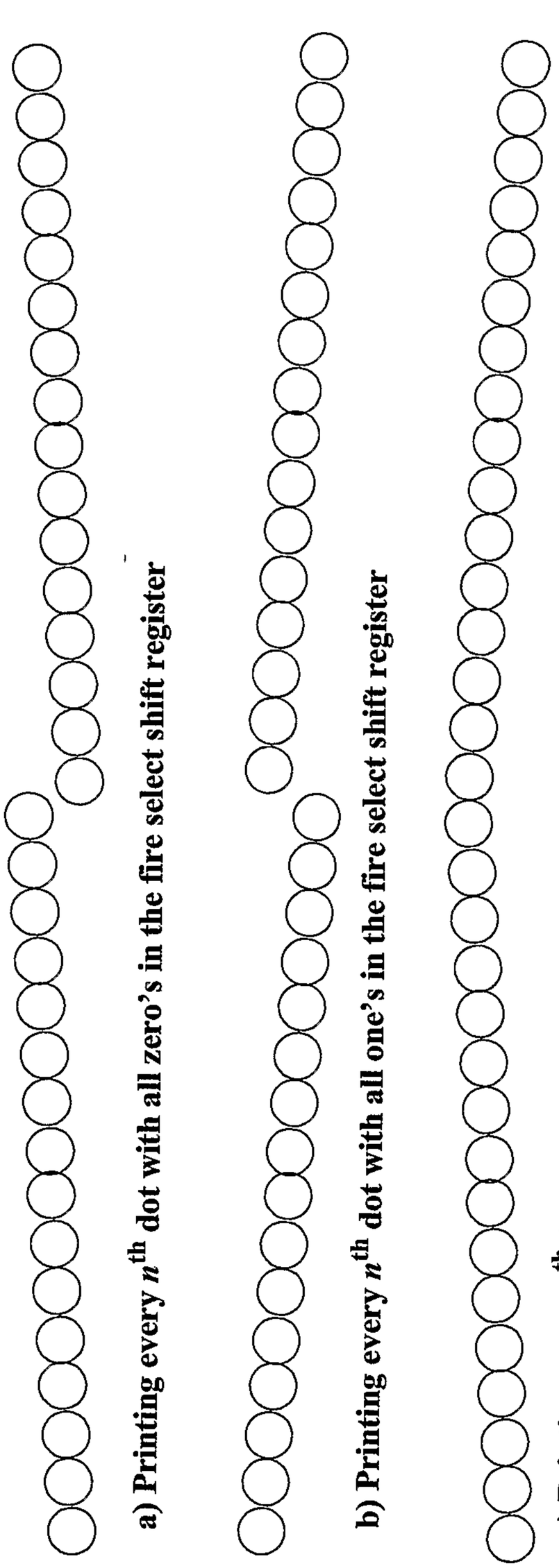

a) Printing every n^{th} dot with all zero's in the fire select shift register
b) Printing every n^{th} dot with all one's in the fire select shift register
c) Printing every n^{th} dot with n zero's then n one's in the fire select shift registers

FIG. 312

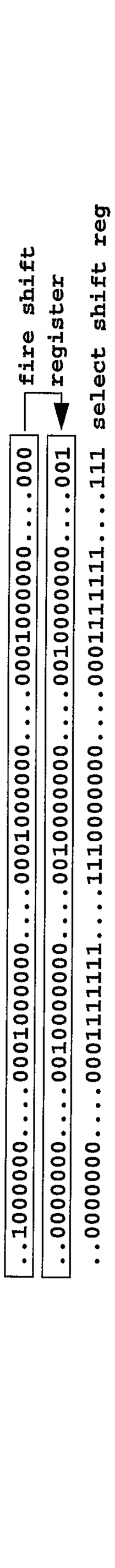


FIG. 313

FIG. 314

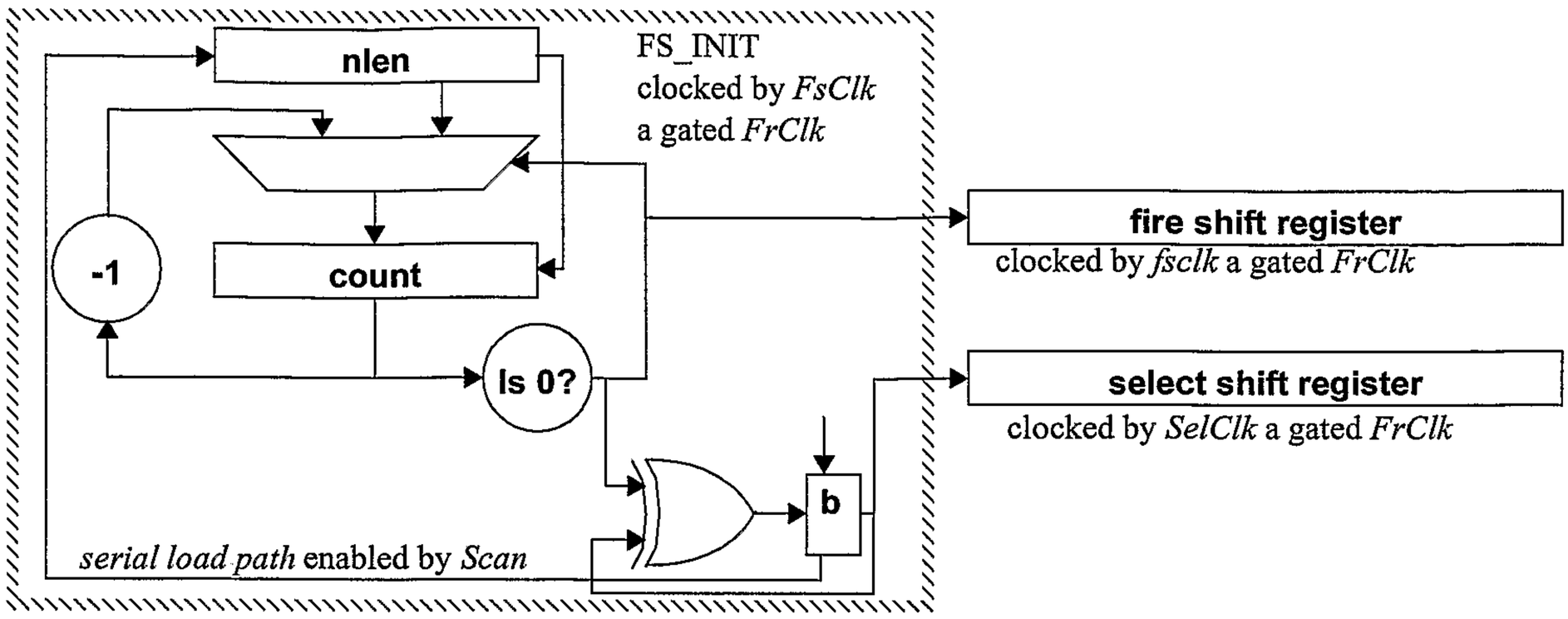


FIG. 315

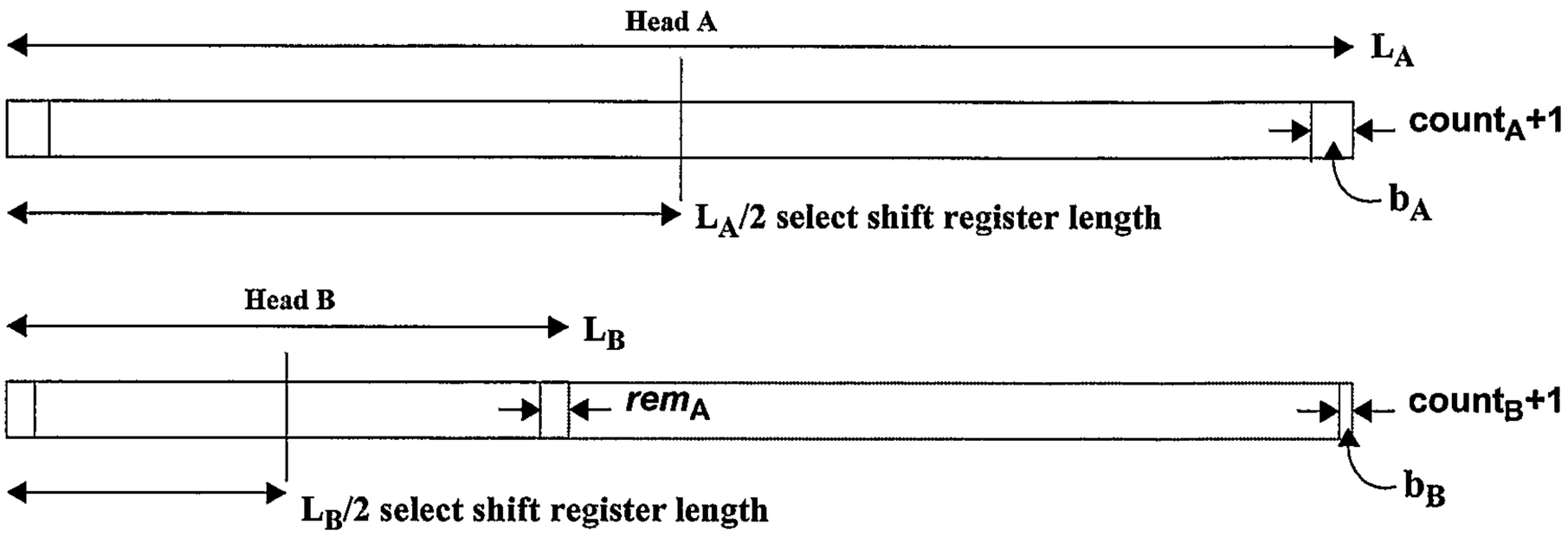


FIG. 316

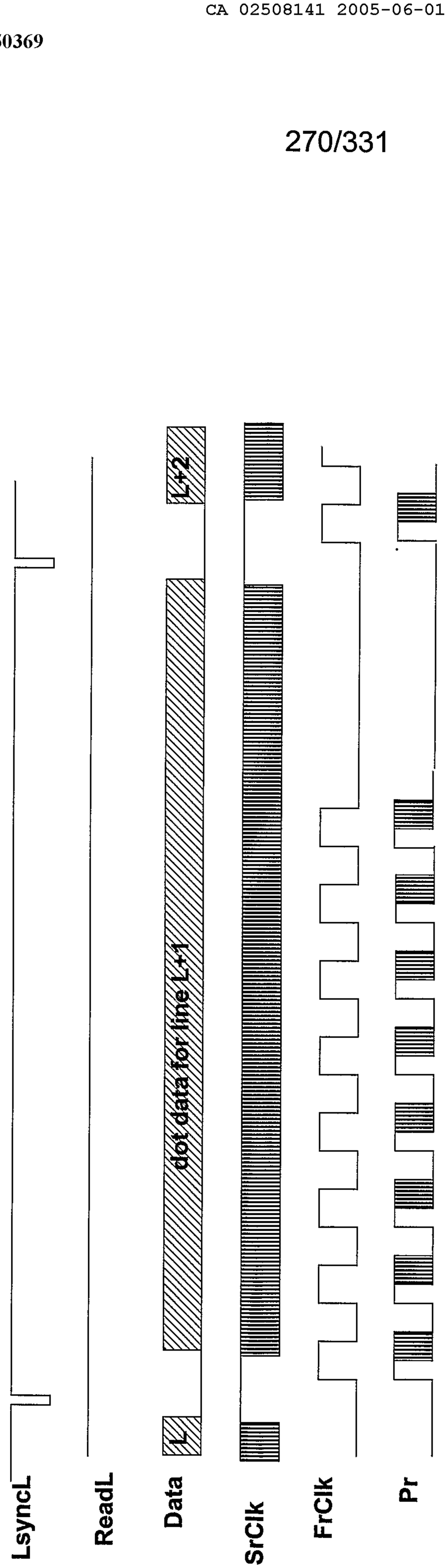


FIG. 317

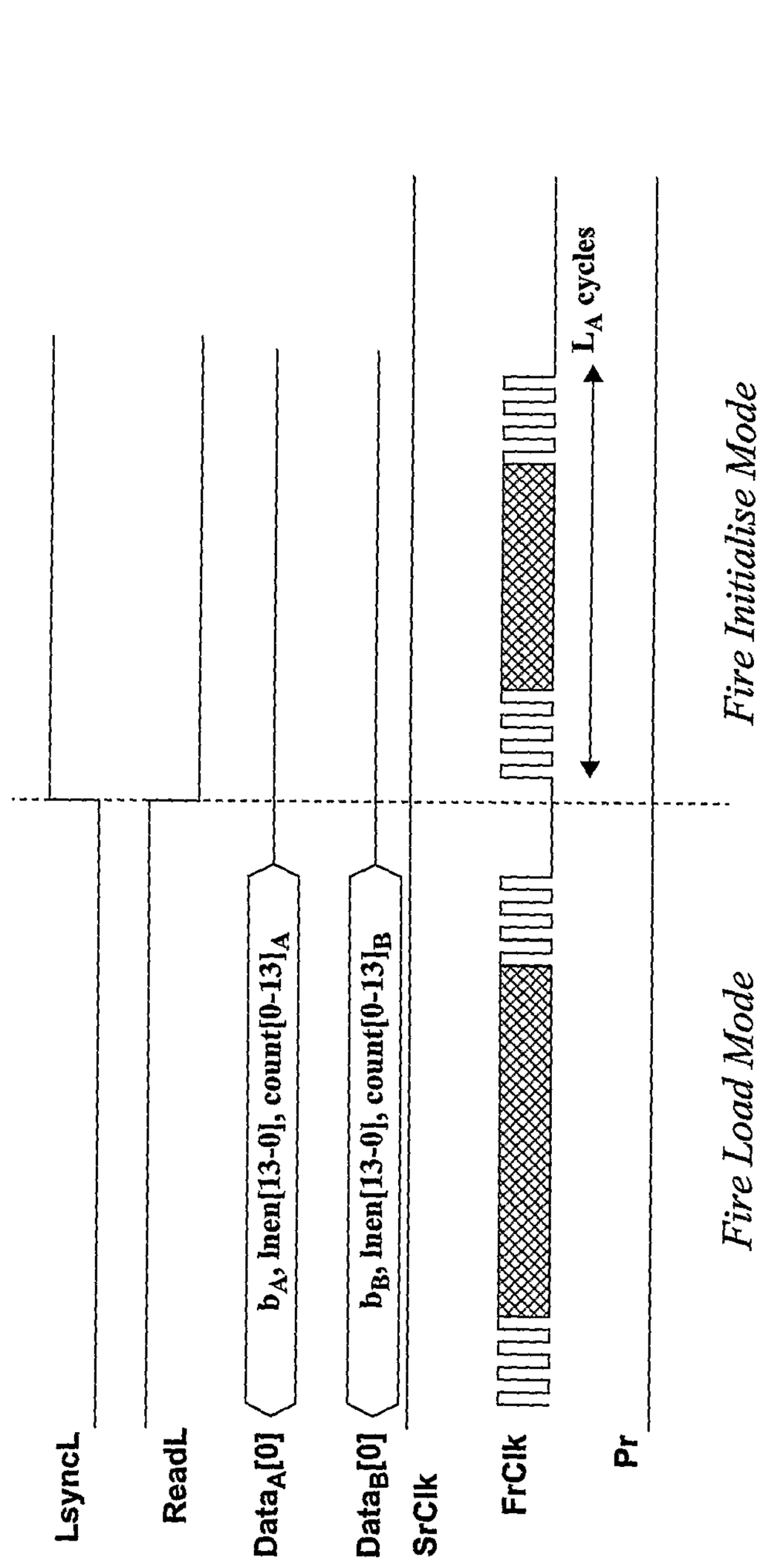

Fire Initialise Mode
Fire Load Mode
L_A cycles
b_A, lnen[13-0], countf[0-13]$_A$
b_B, lnen[13-0], countf[0-13]$_B$
LsyncL
ReadL
Data$_A$[0]
Data$_B$[0]
SrClk
FrClk
Pr

FIG. 318

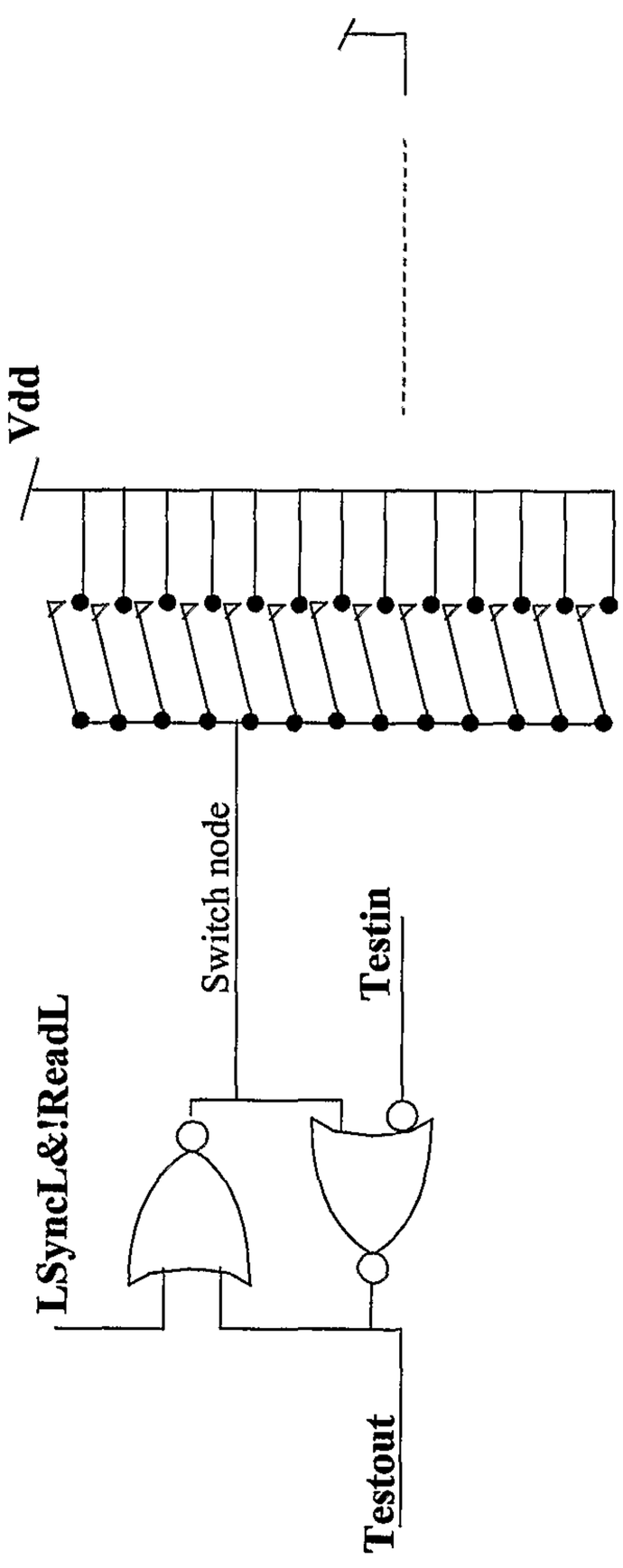


FIG. 319

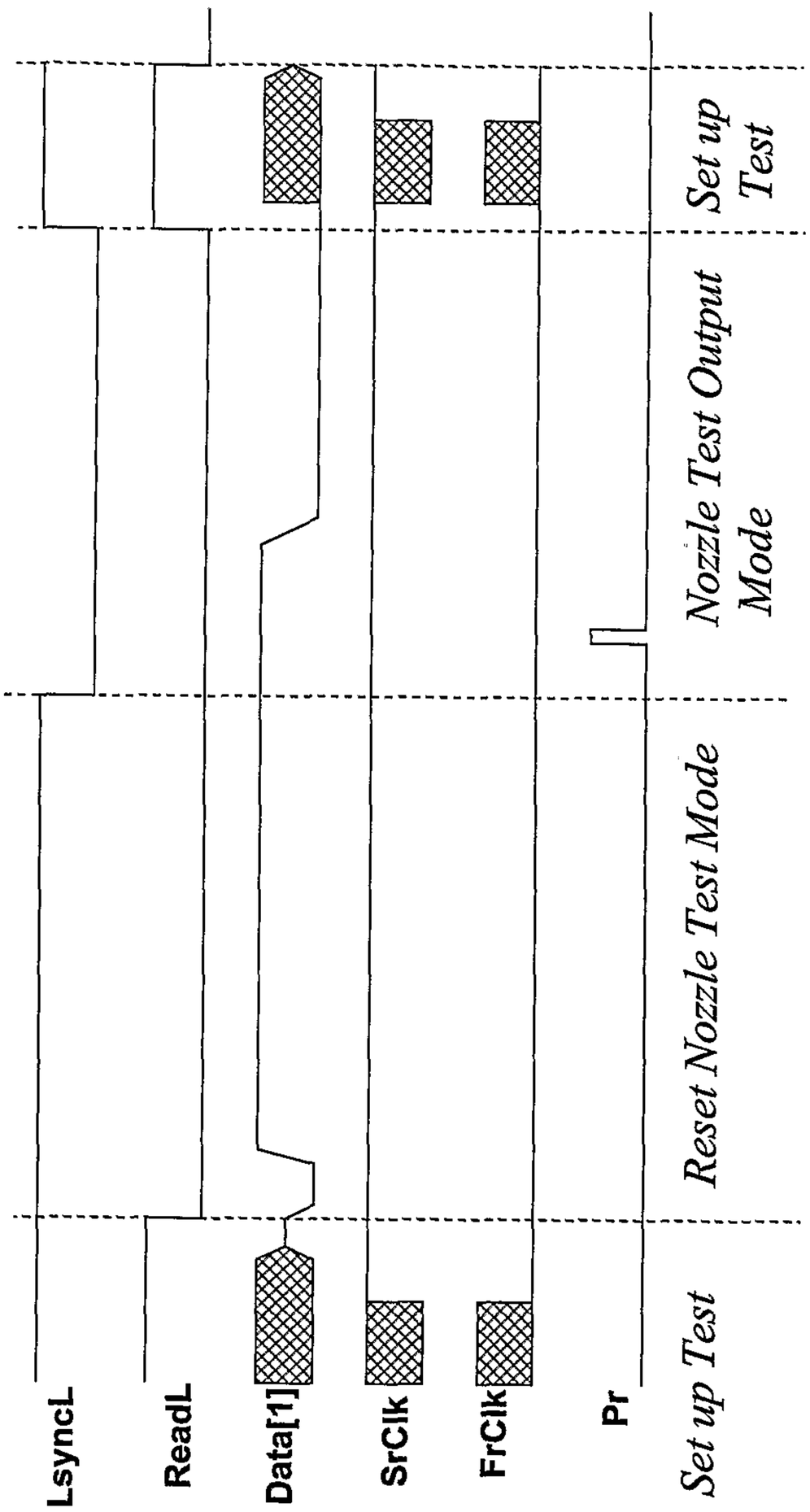

Set up Test
Nozzle Test Output Mode
Reset Nozzle Test Mode
Set up Test
LsyncL
ReadL
Data[1]
SrClk
FrClk
Pr

FIG. 320

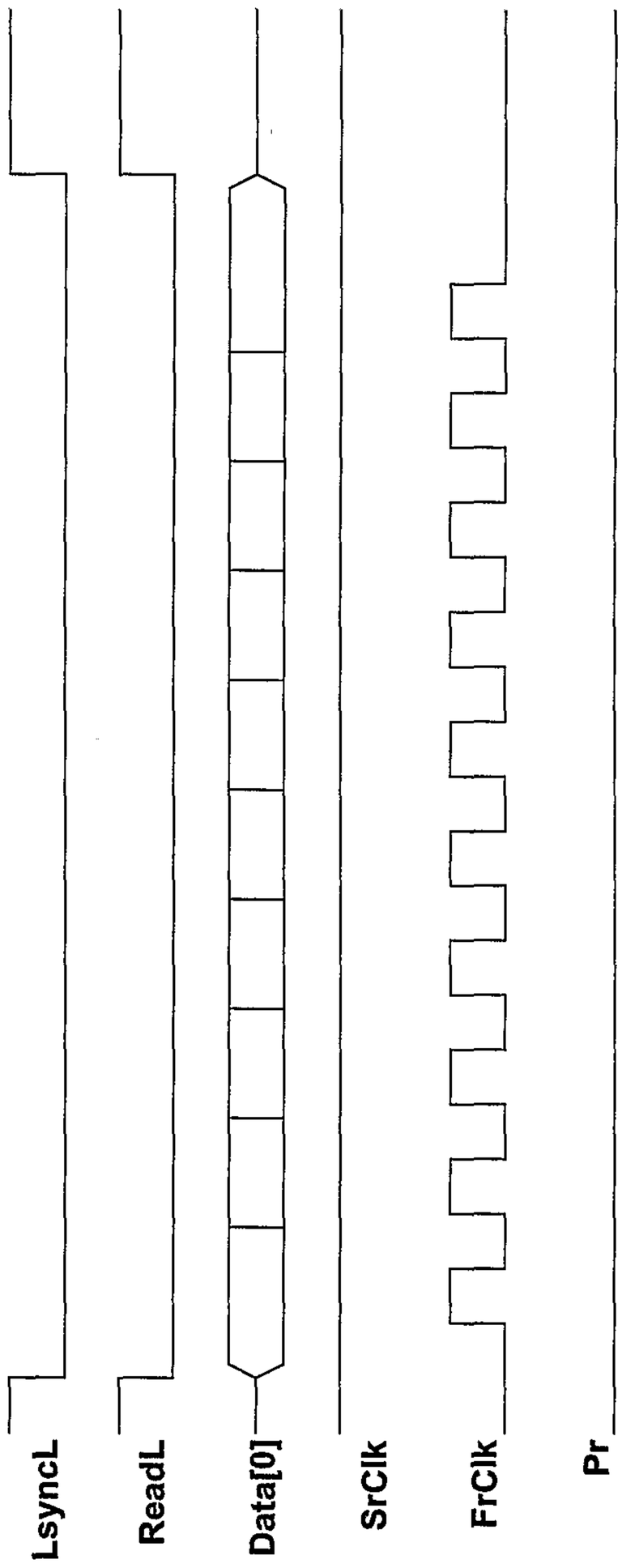


FIG. 321

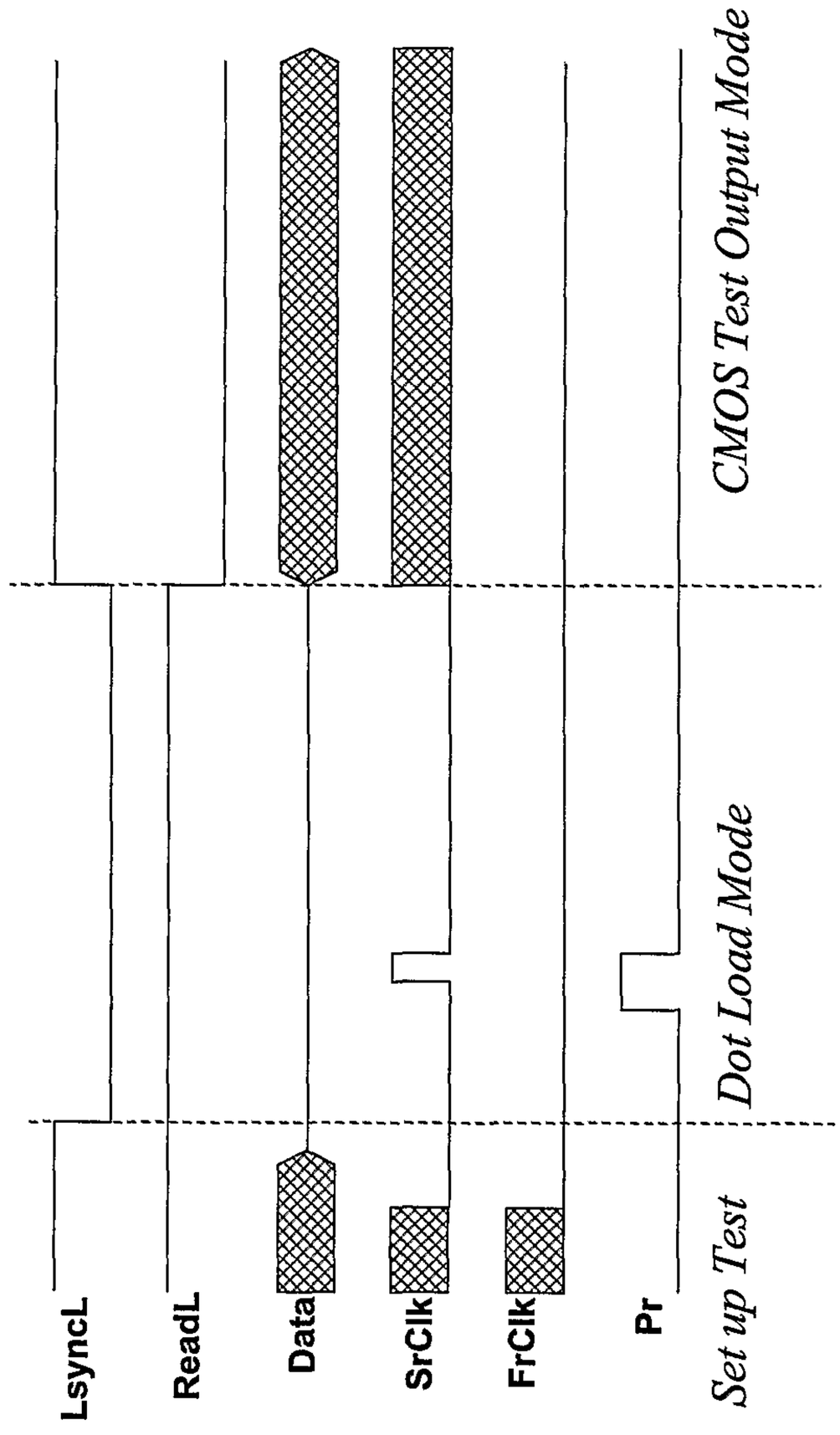

LsyncL
ReadL
Data
SrClk
FrClk
Pr
Set up Test
Dot Load Mode
CMOS Test Output Mode

FIG. 322

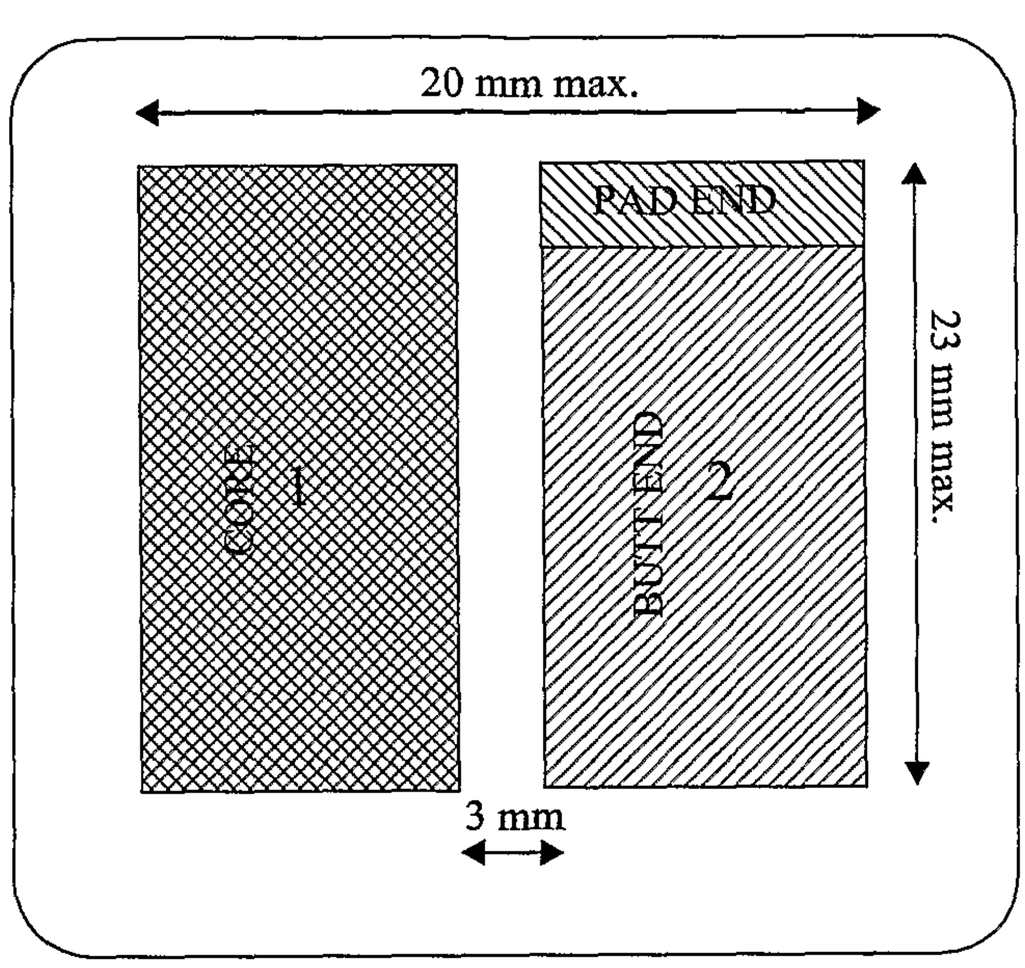


FIG. 323

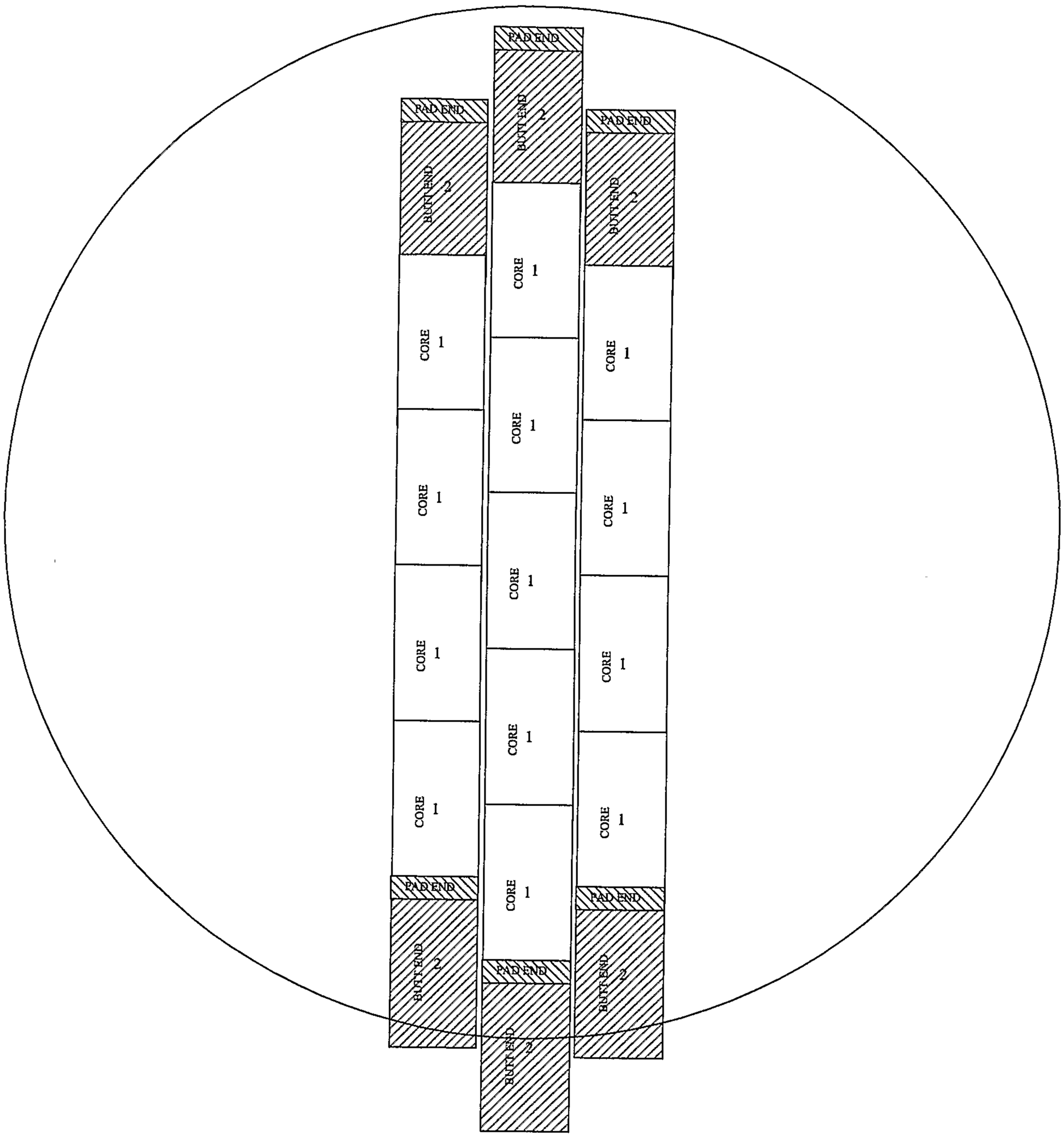


FIG. 324

FIG. 325

FIG. 327

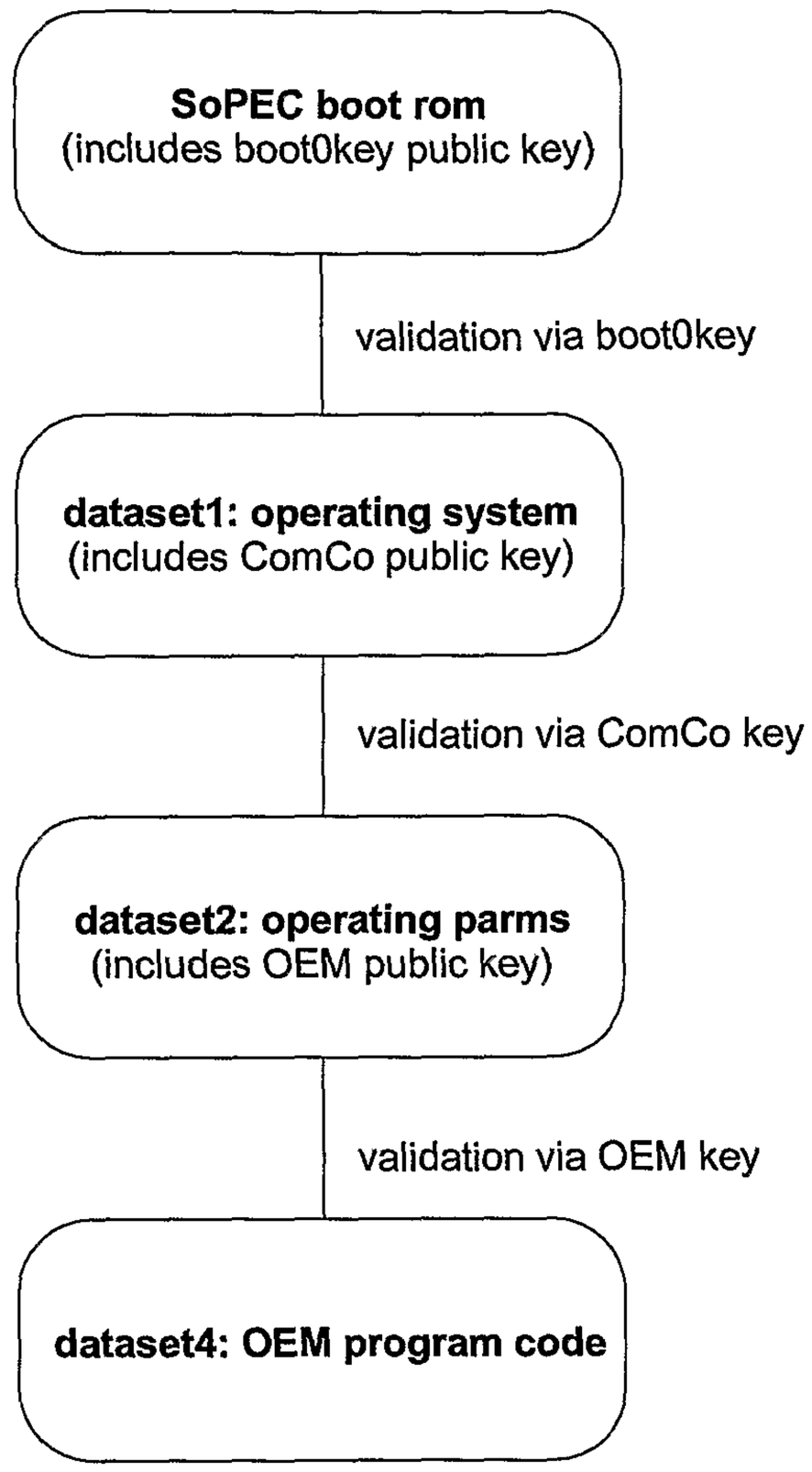

SoPEC boot rom
(includes boot0key public key)
validation via boot0key
dataset1: operating system
(includes ComCo public key)
validation via ComCo key
dataset2: operating parms
(includes OEM public key)
validation via OEM key
dataset4: OEM program code

FIG. 326

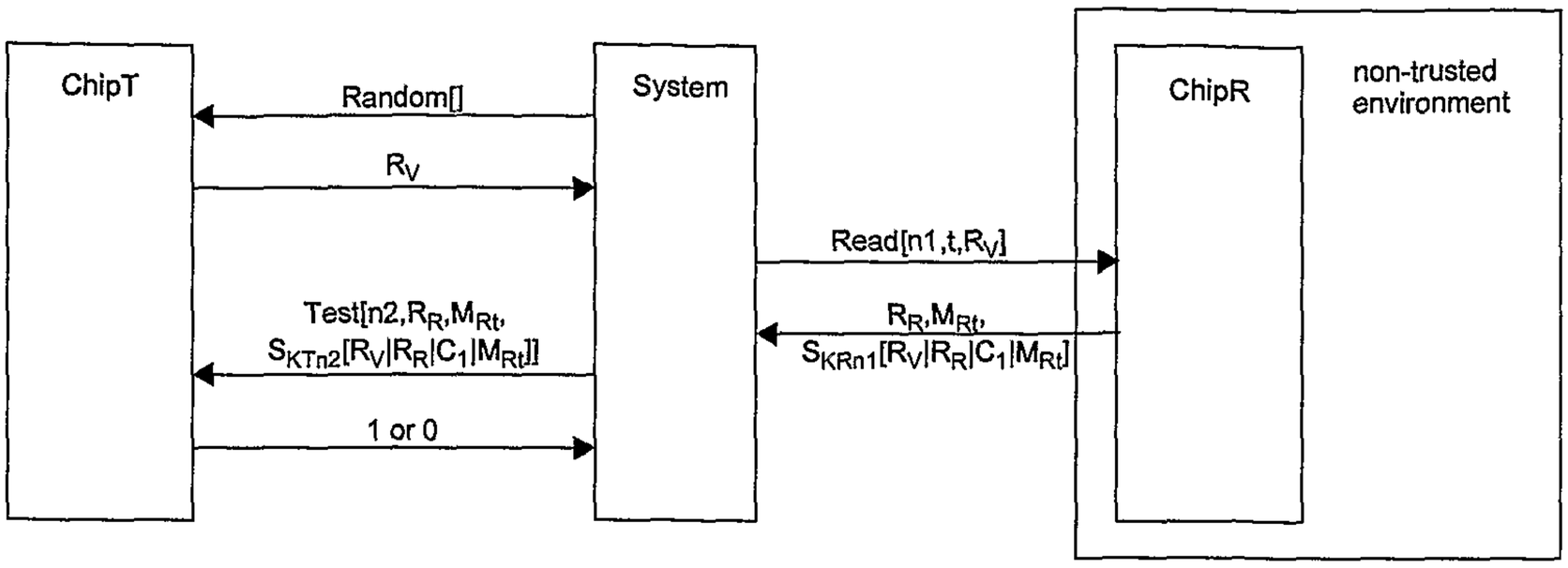


FIG. 328

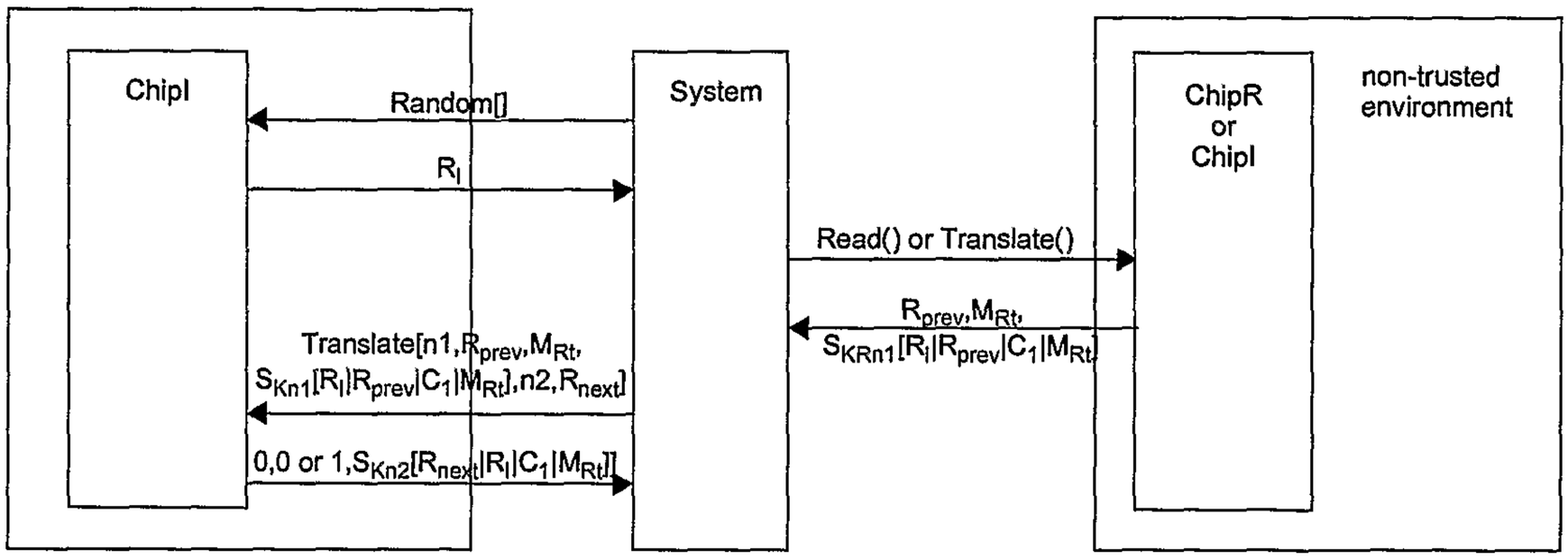


FIG. 329

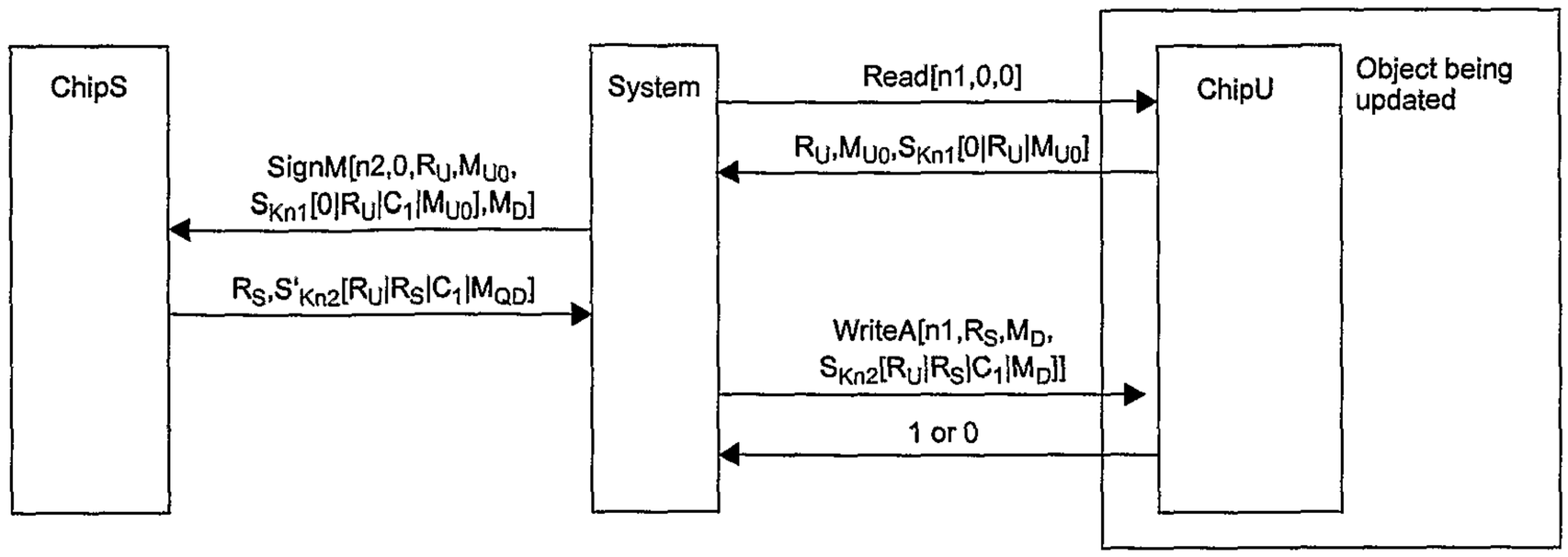


FIG. 330

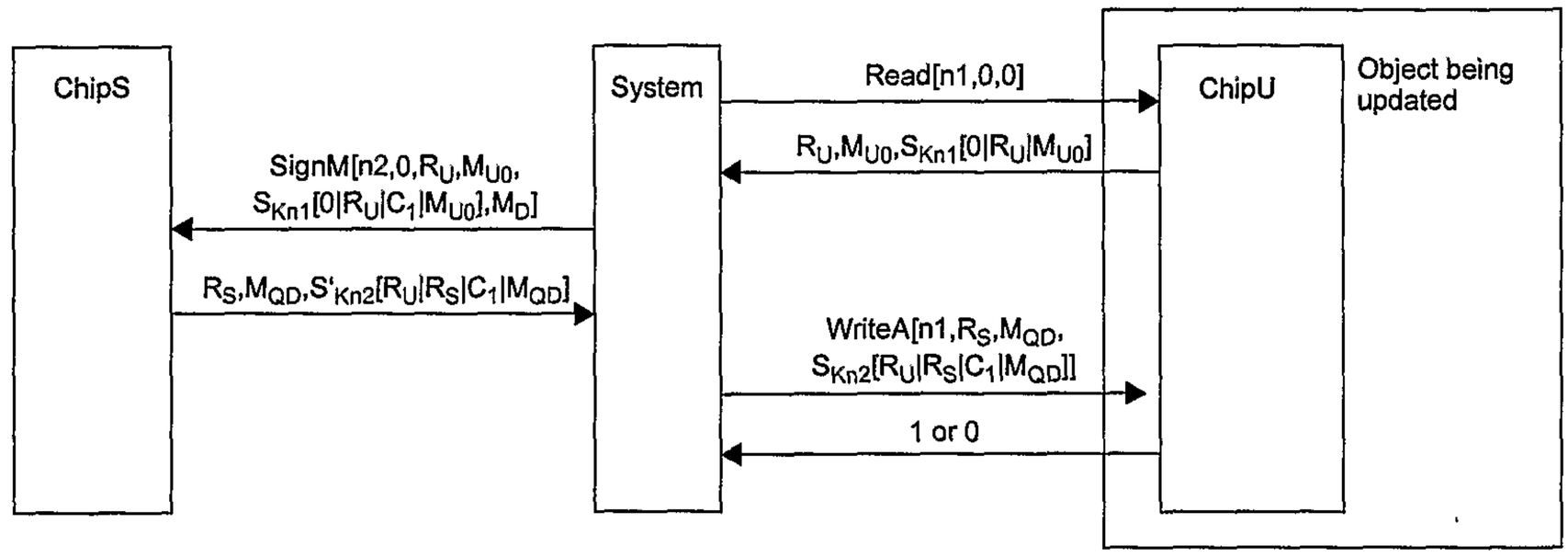


FIG. 331

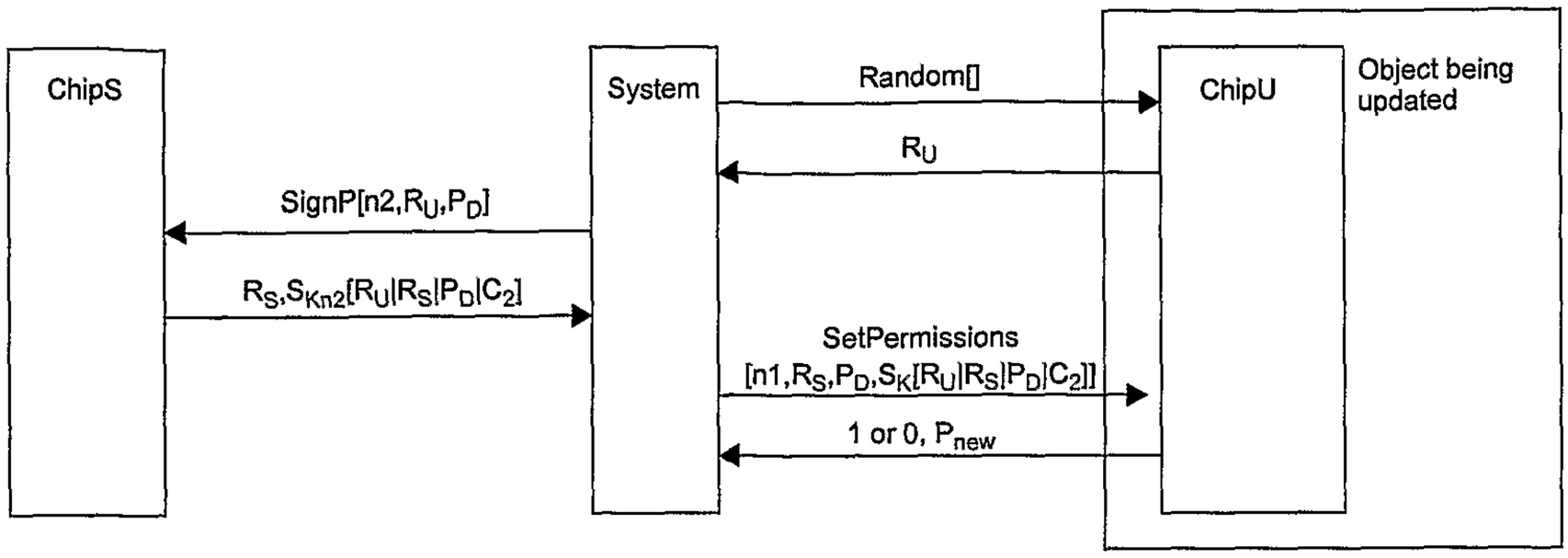


FIG. 332

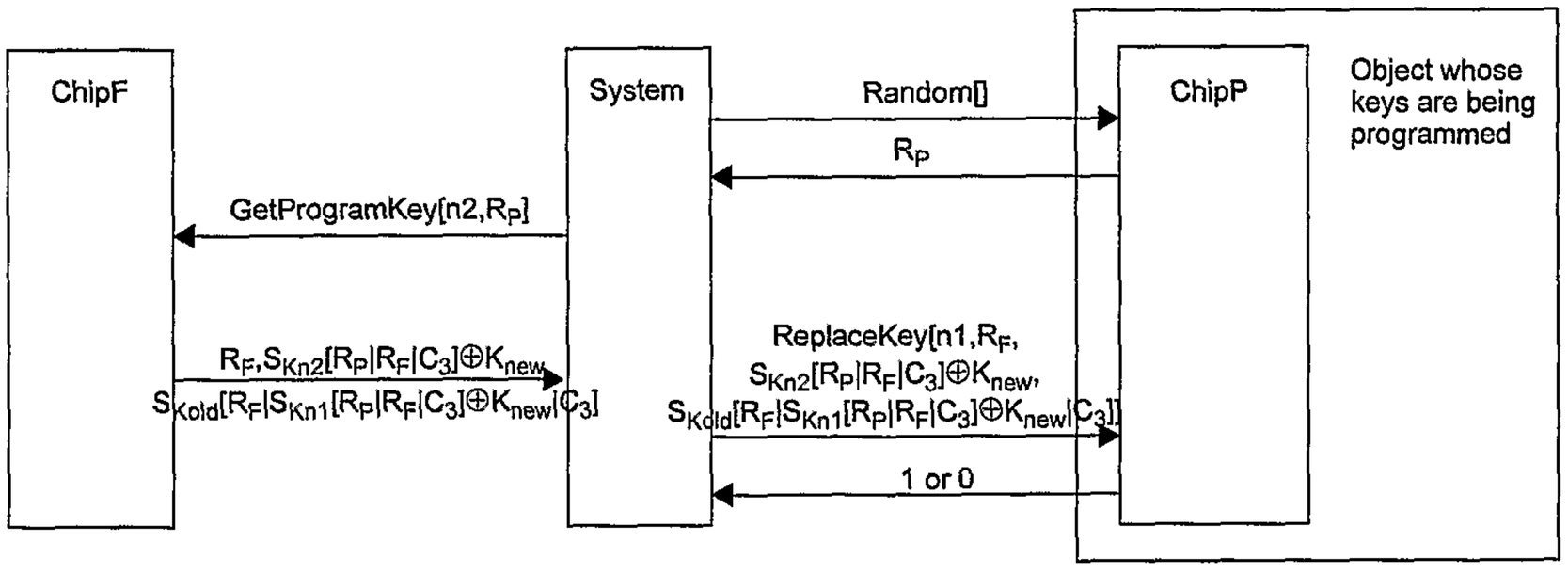


FIG. 333

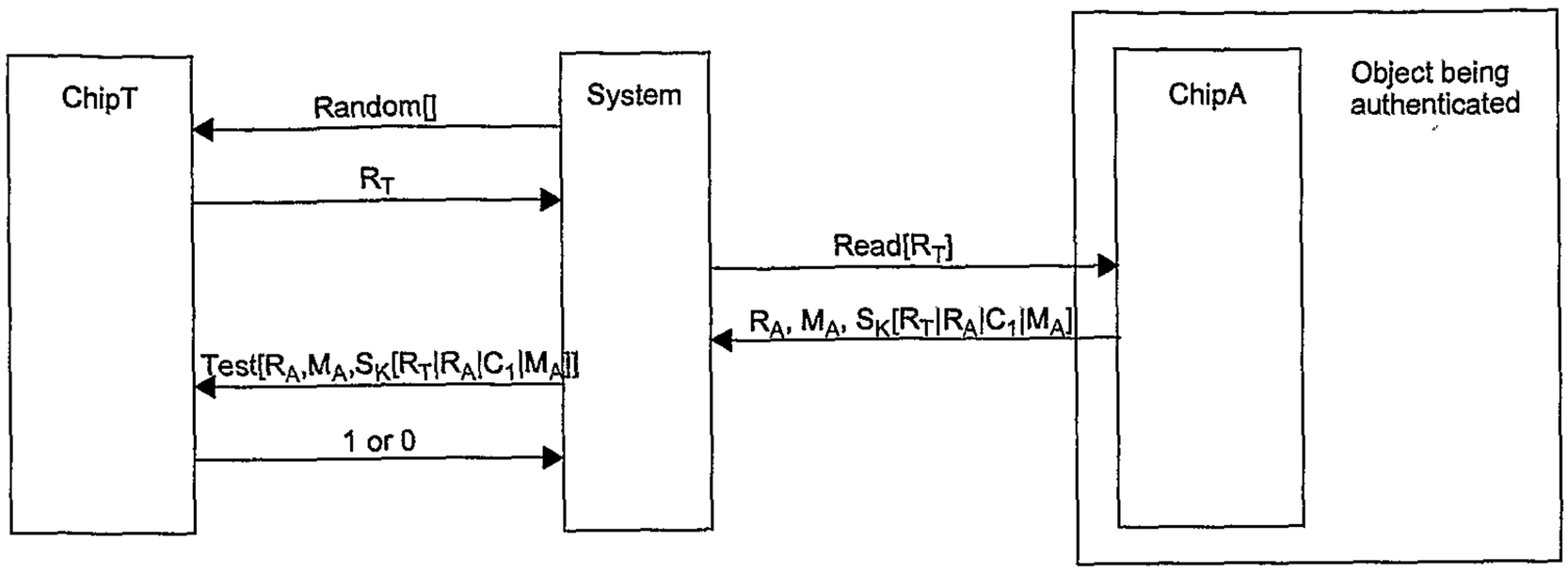


FIG. 334

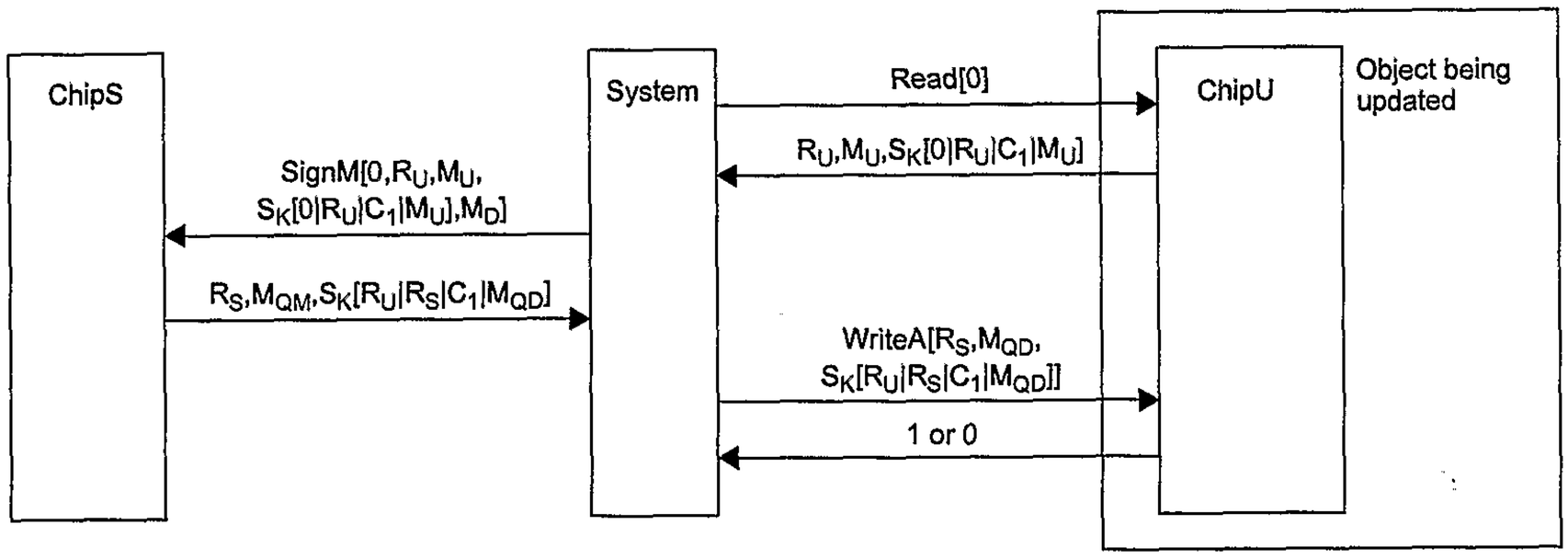


FIG. 335

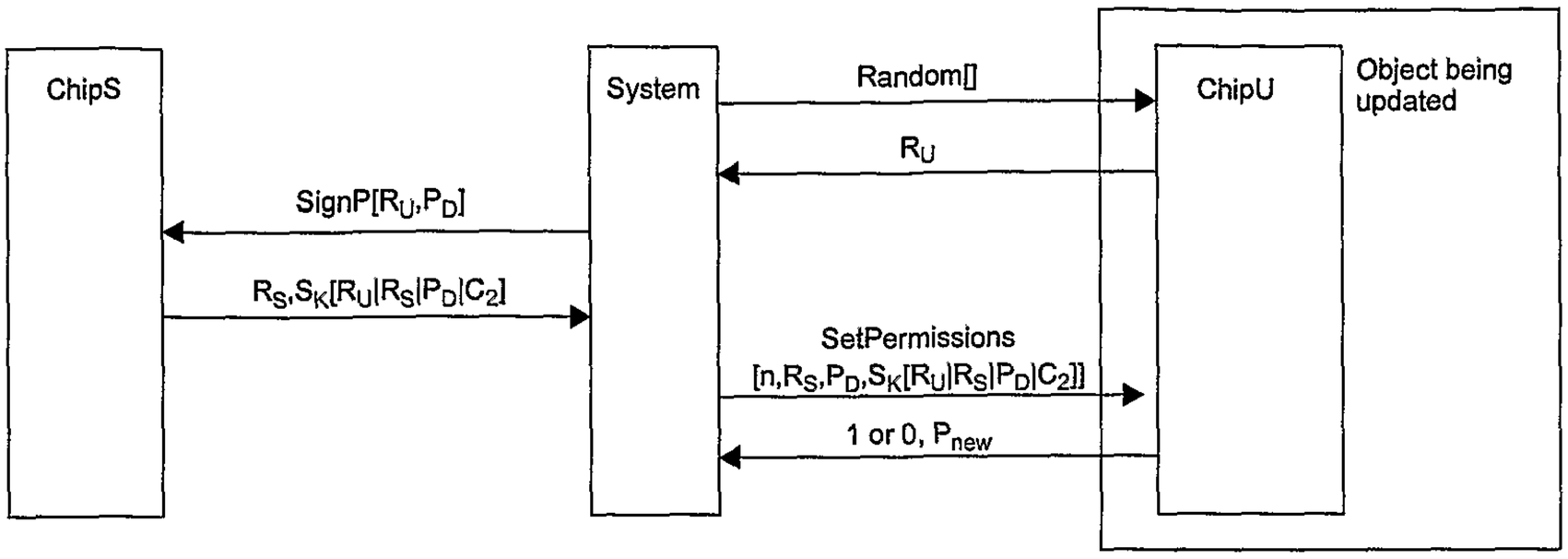


FIG. 336

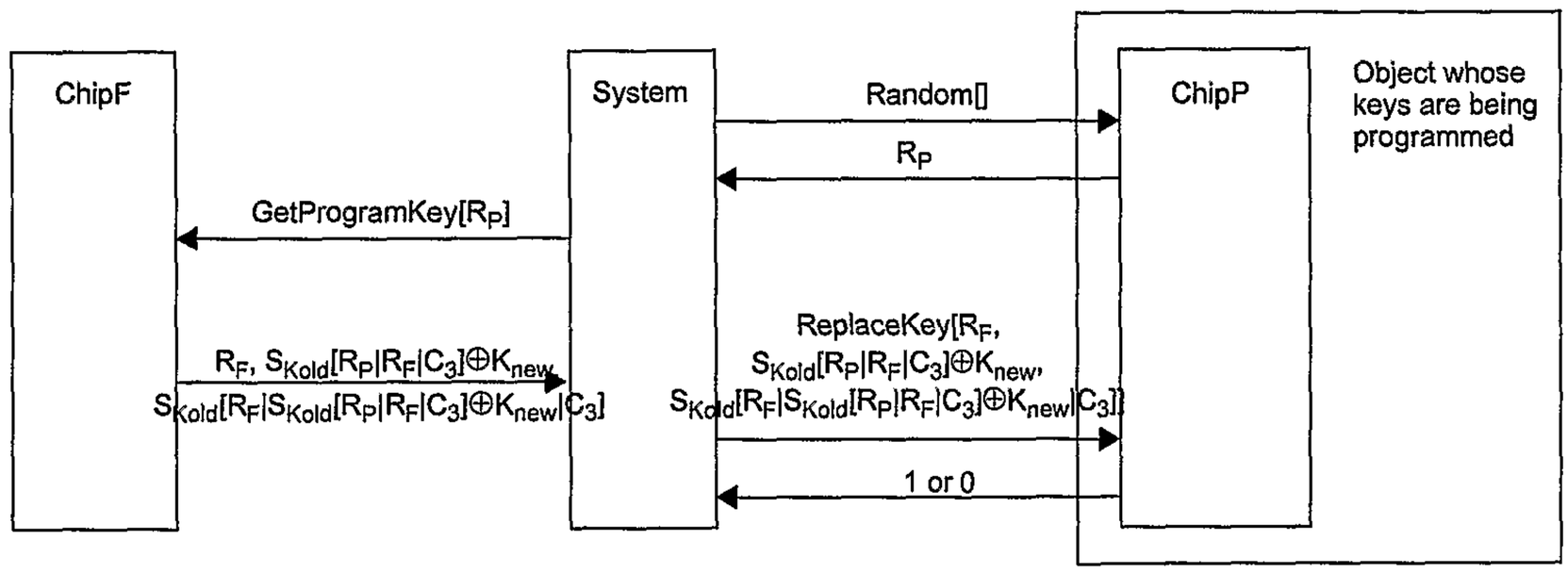


FIG. 337

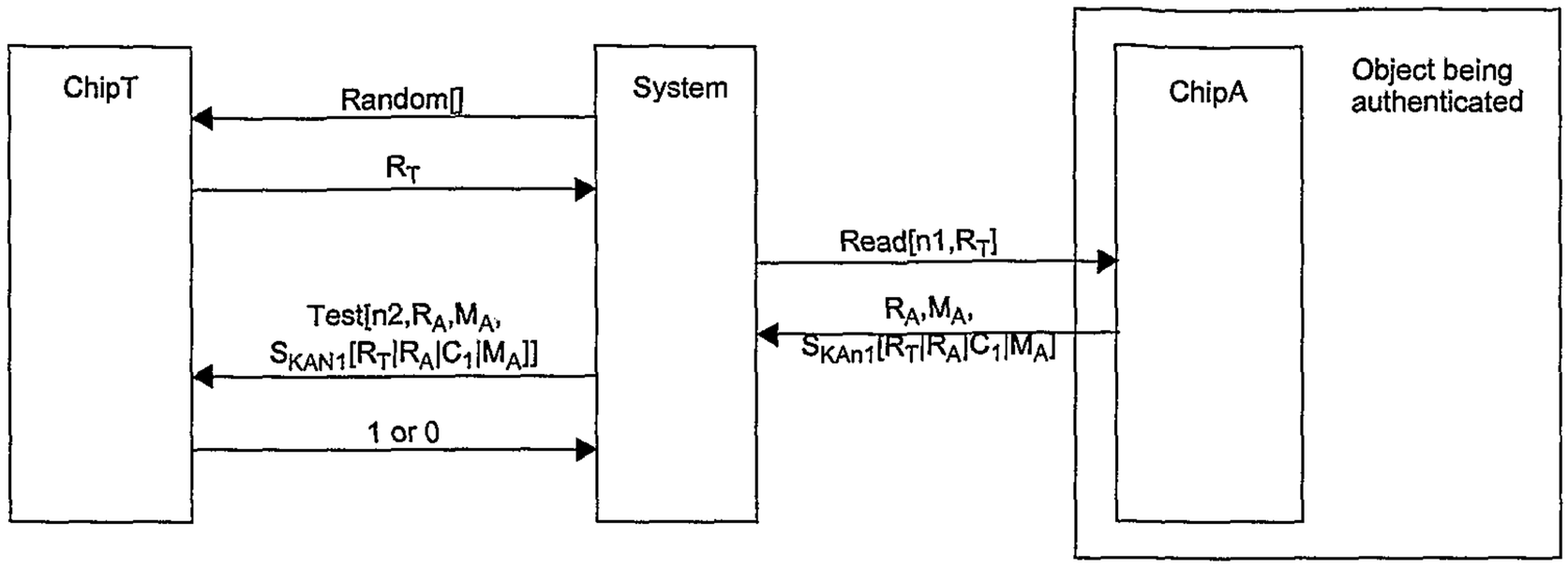


FIG. 338

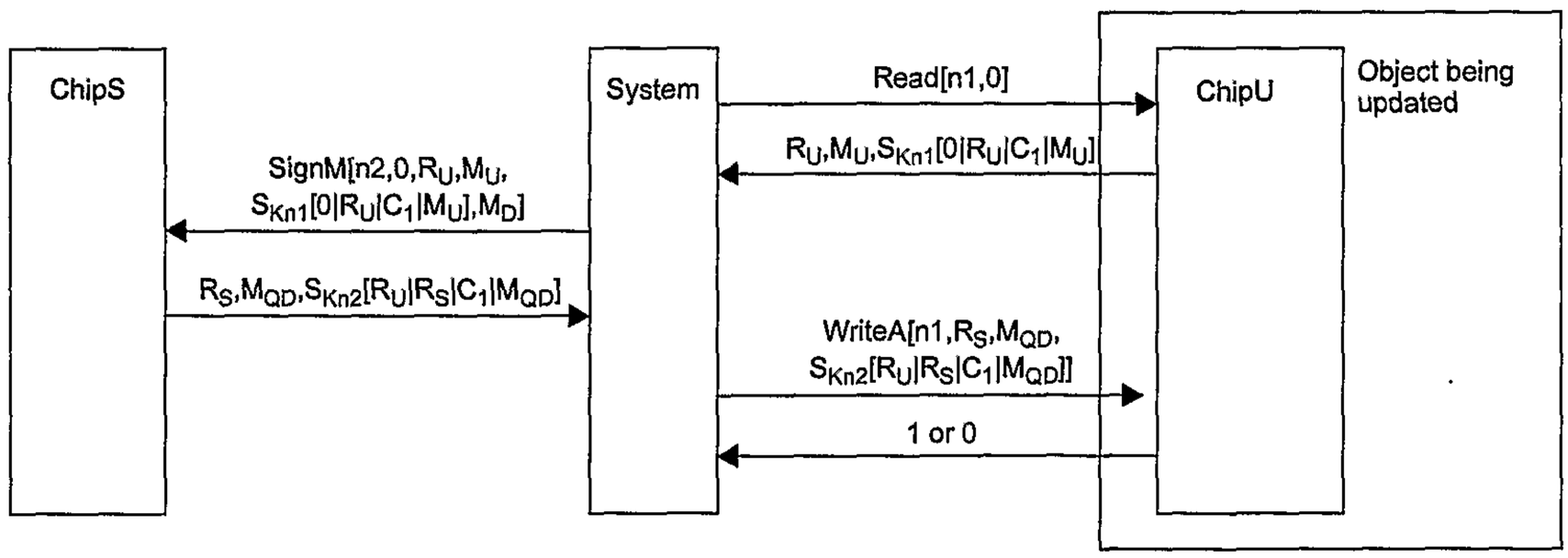


FIG. 339

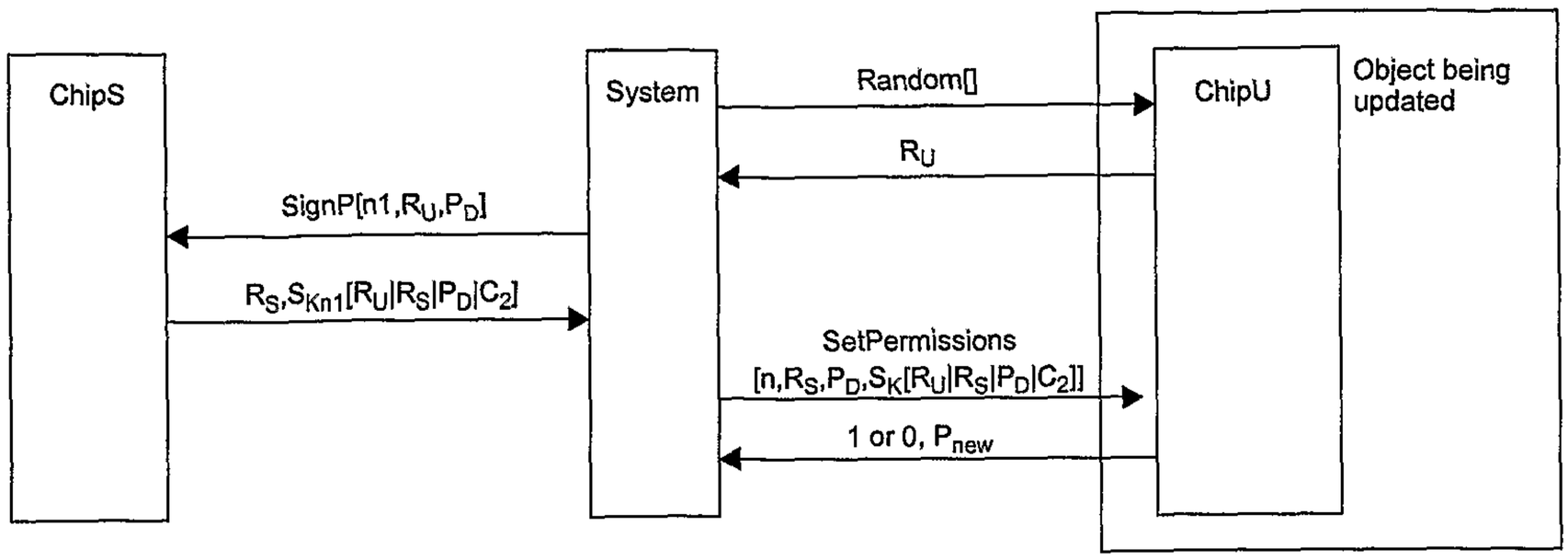


FIG. 340

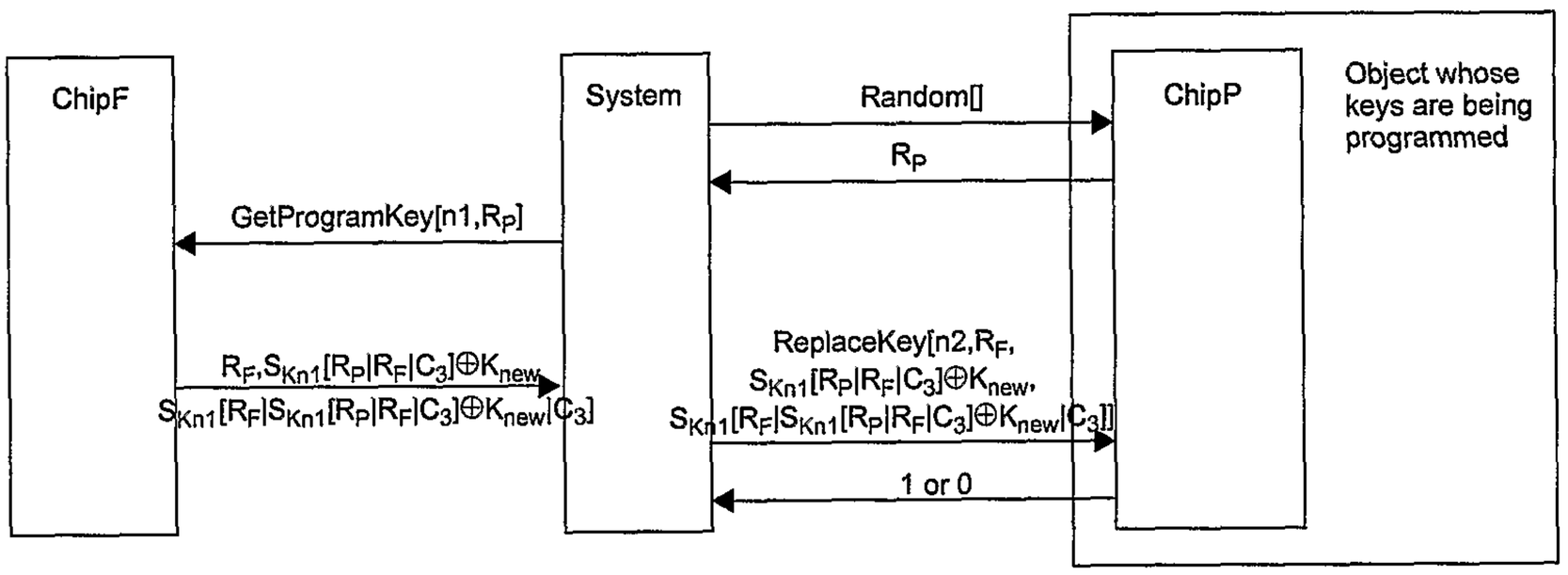


FIG. 341

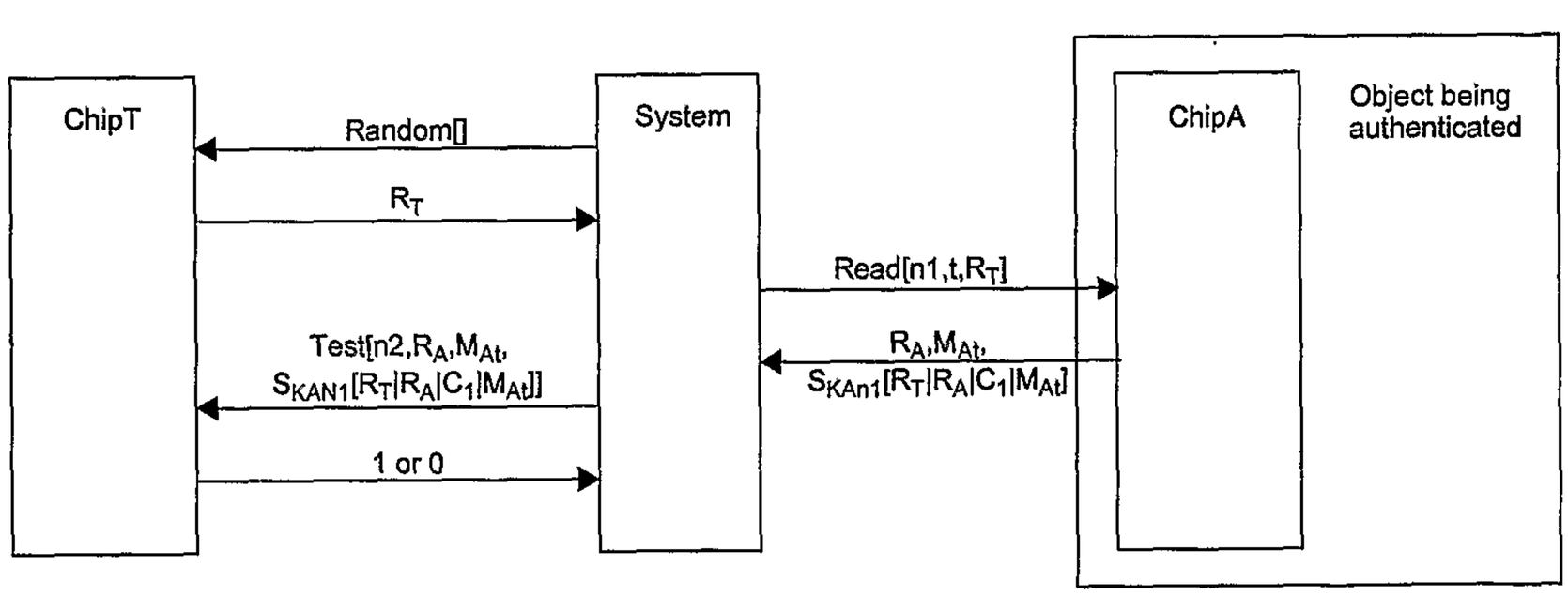


FIG. 342

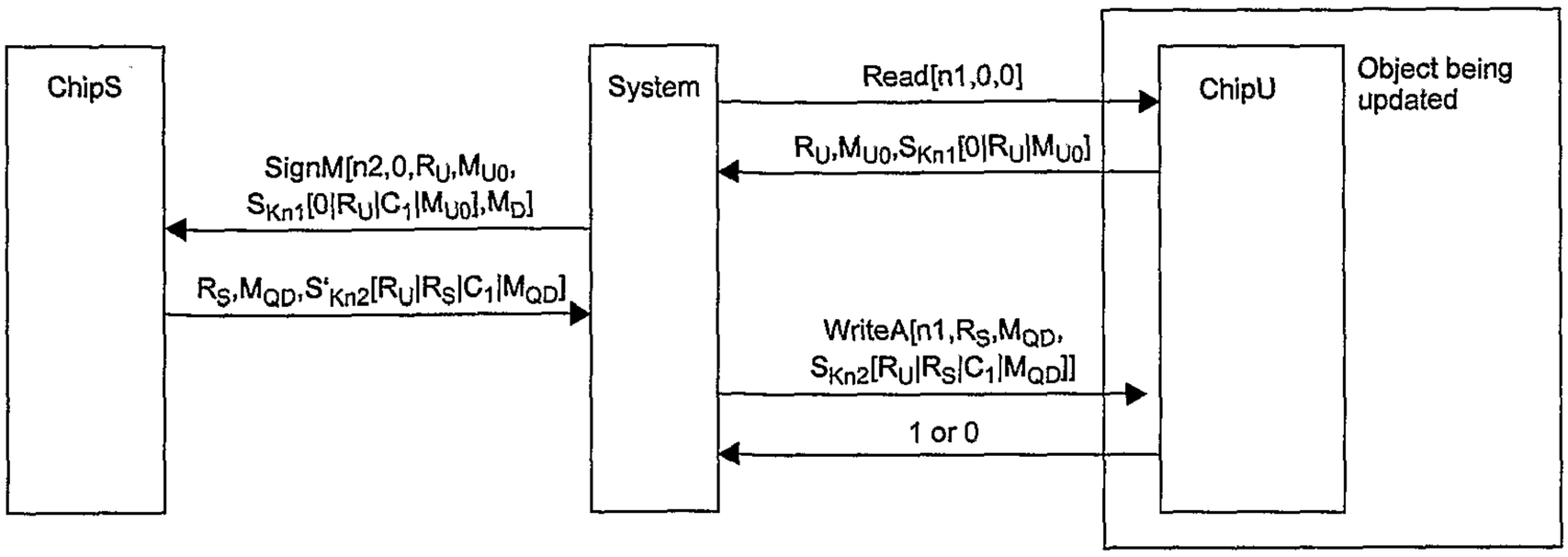


FIG. 343

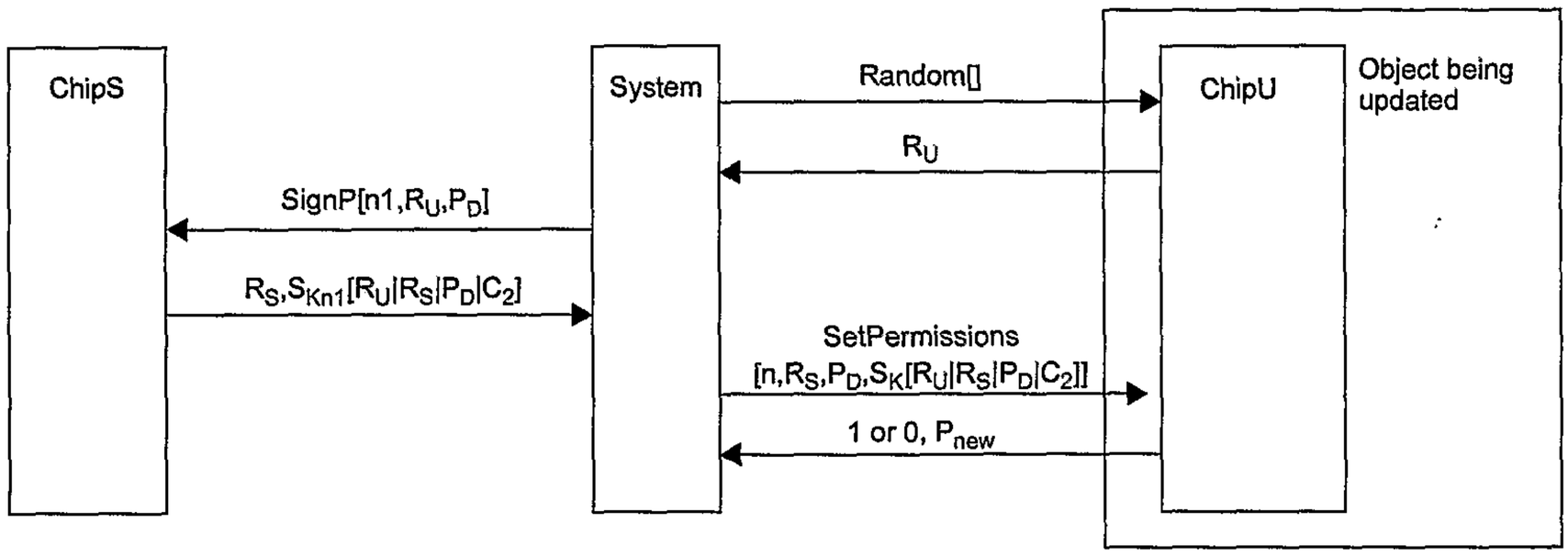


FIG. 344

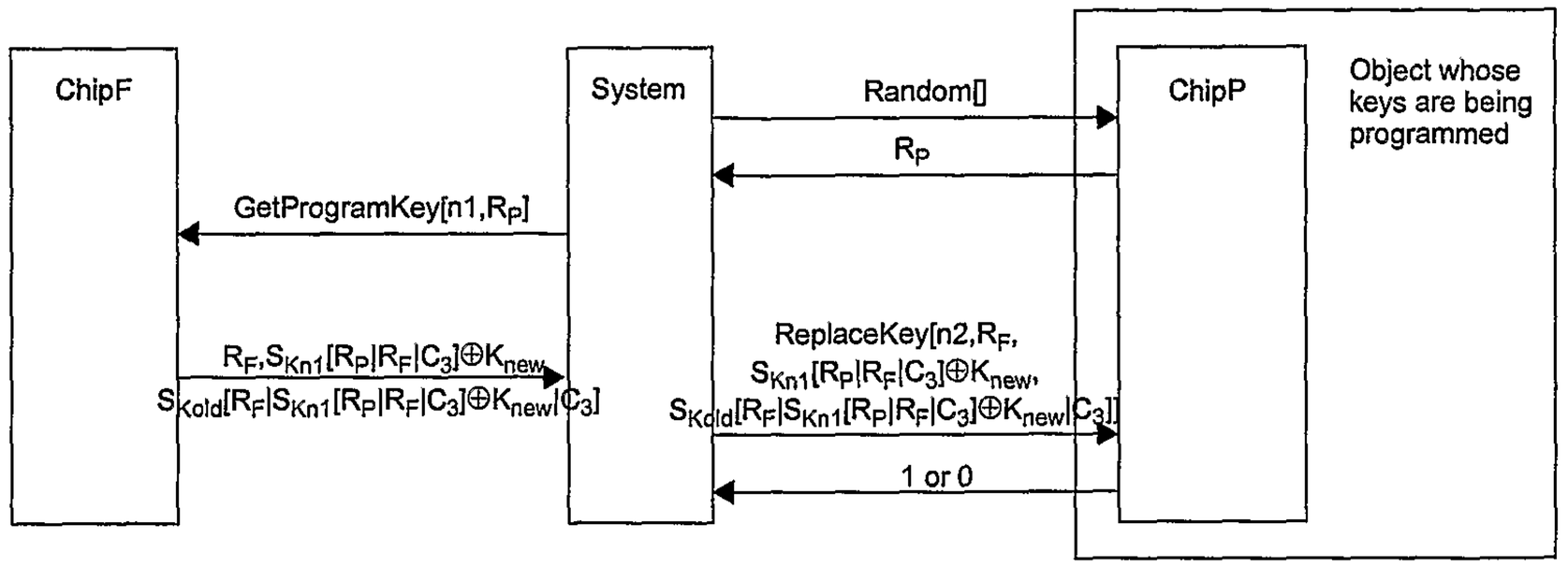


FIG. 345

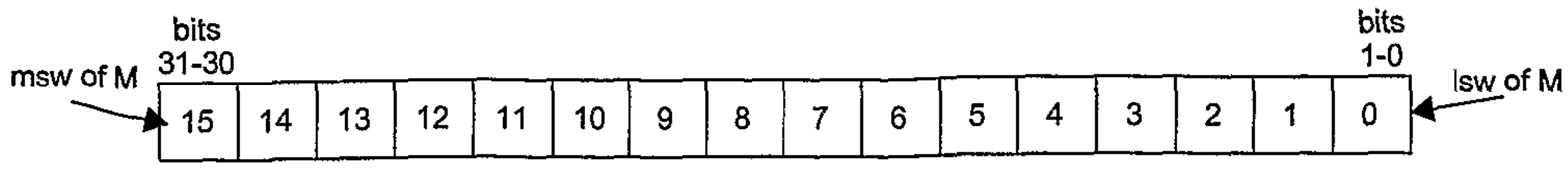


FIG. 346

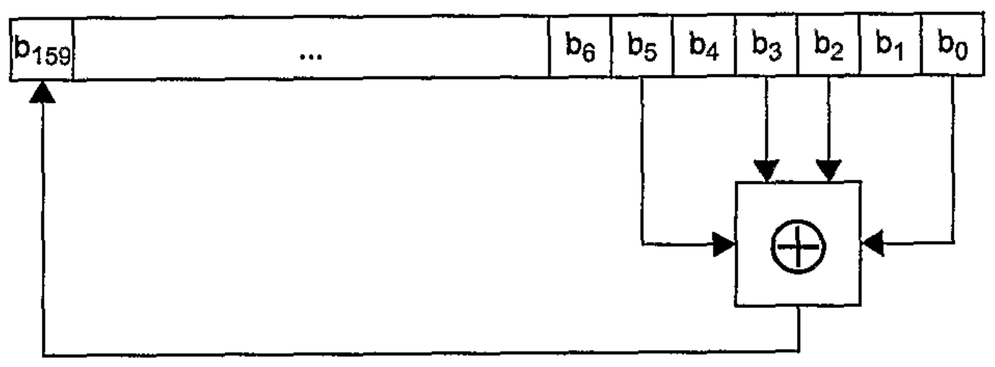


FIG. 347

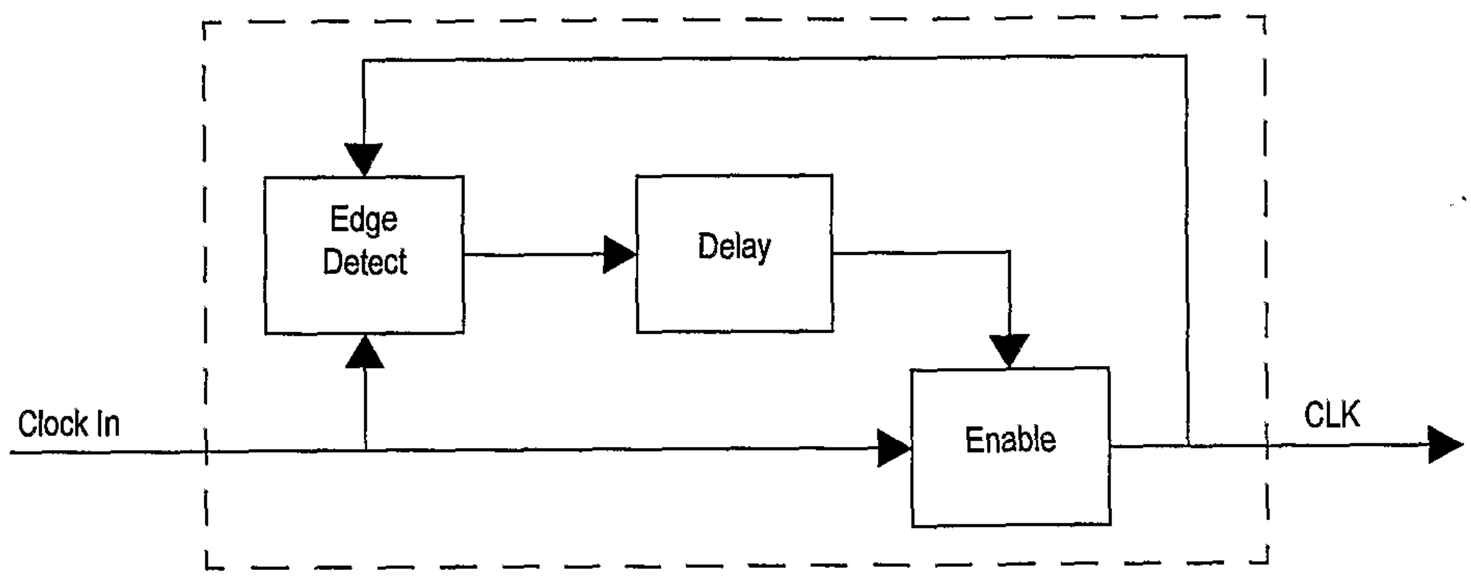


FIG. 348

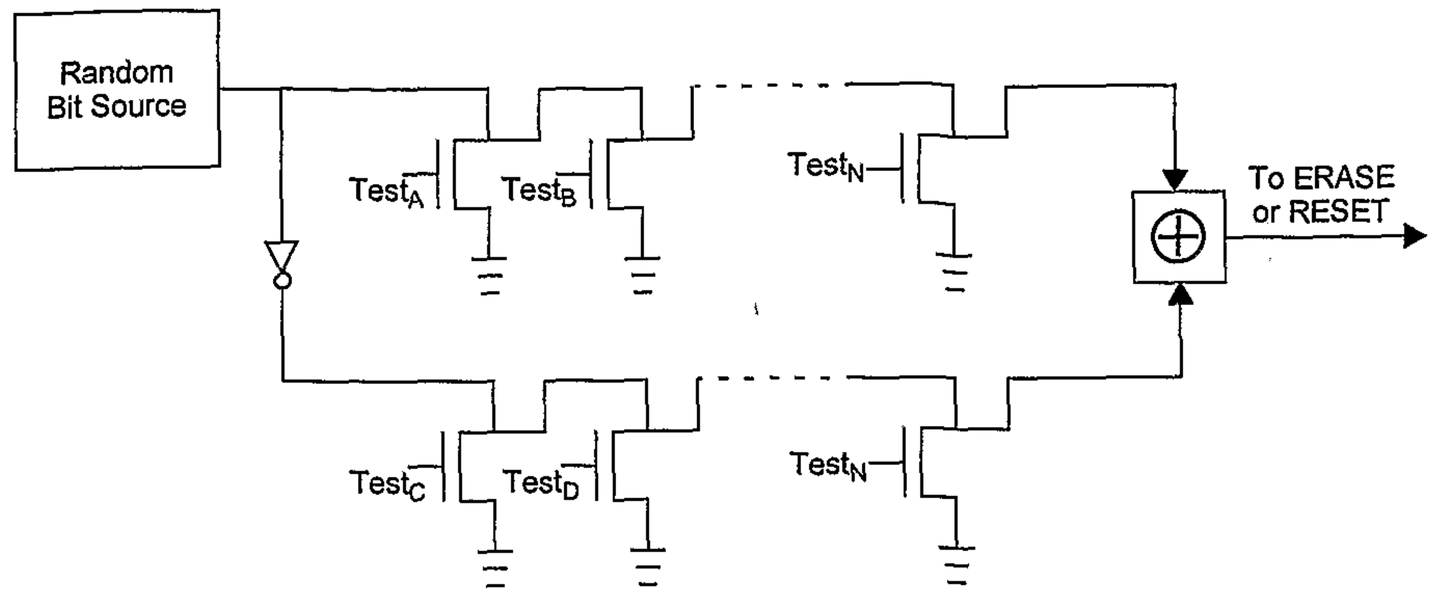


FIG. 349

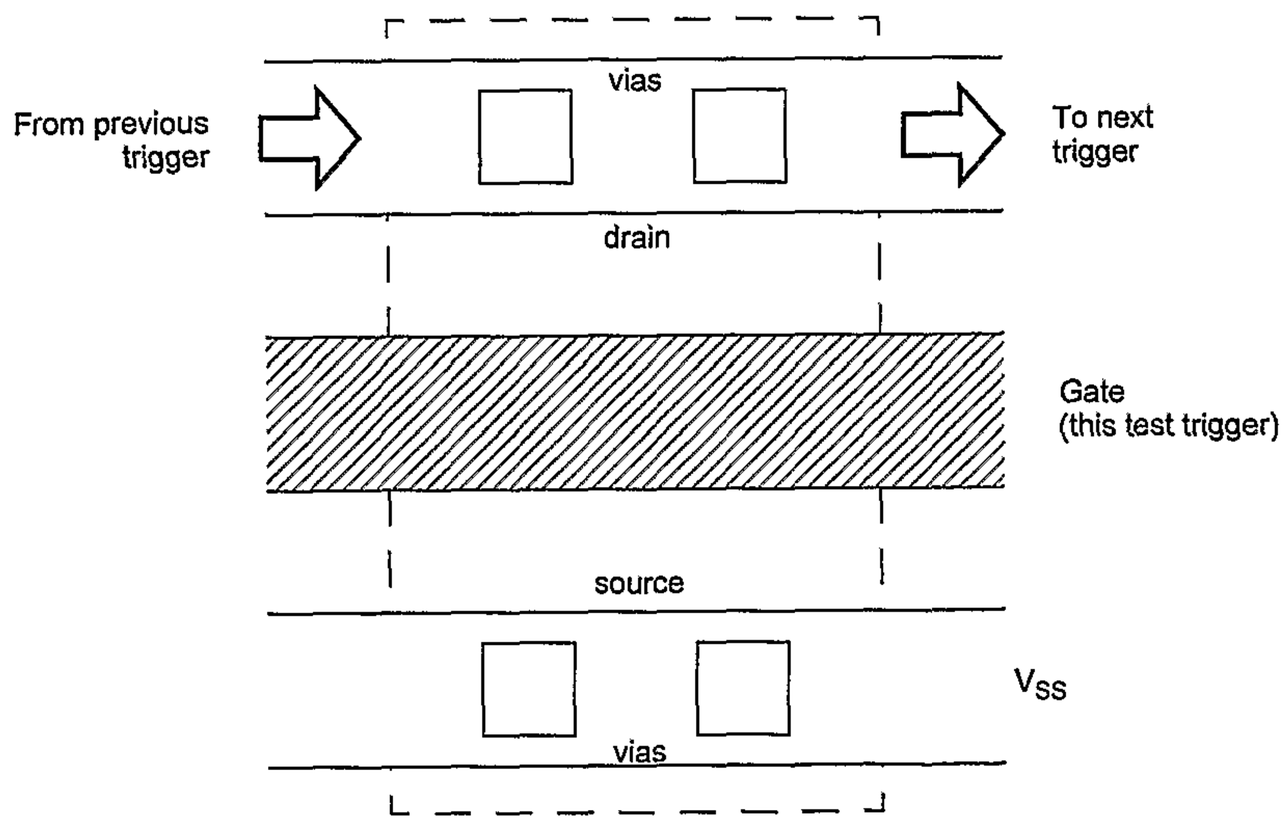


FIG. 350

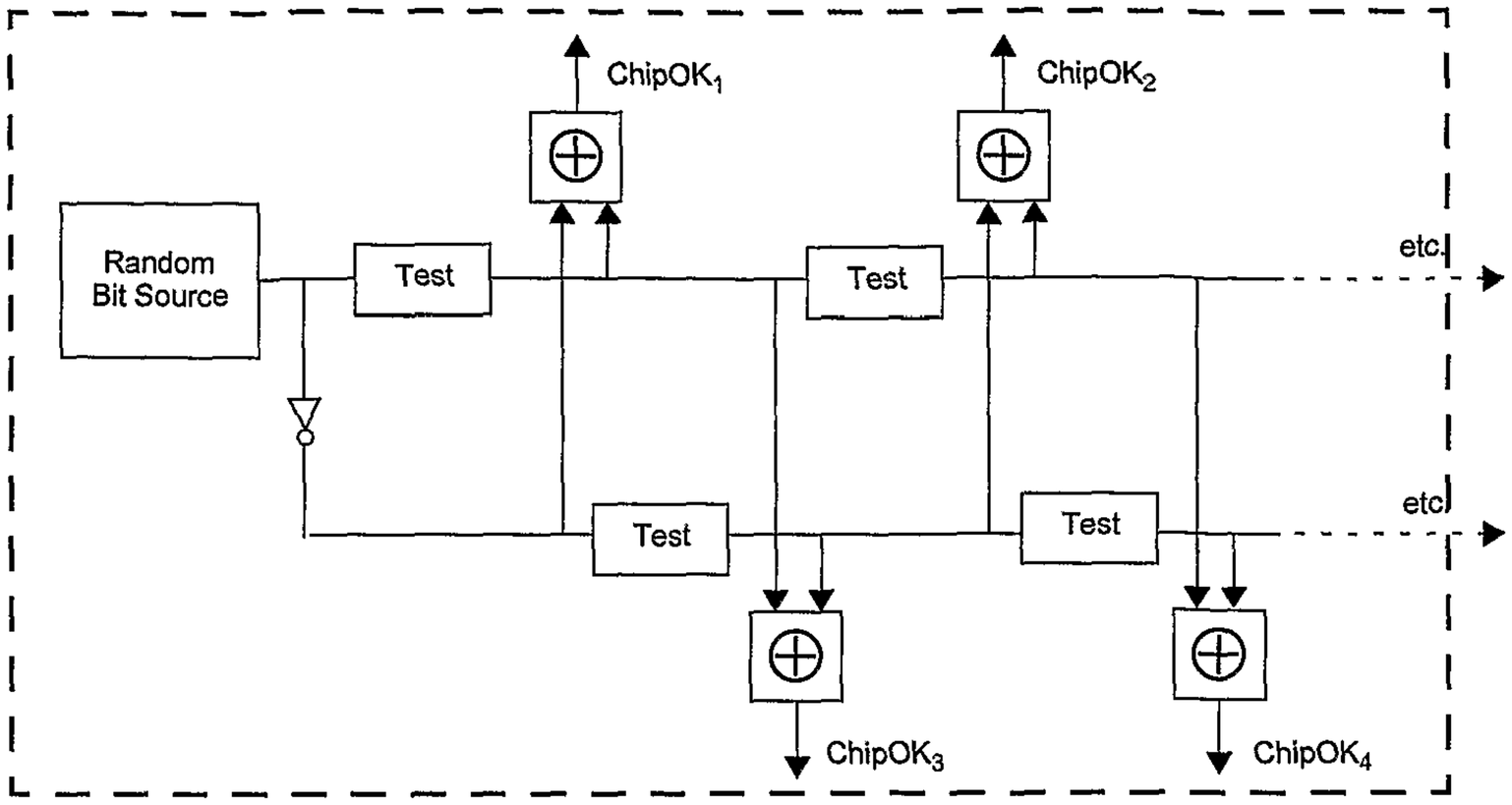


FIG. 351

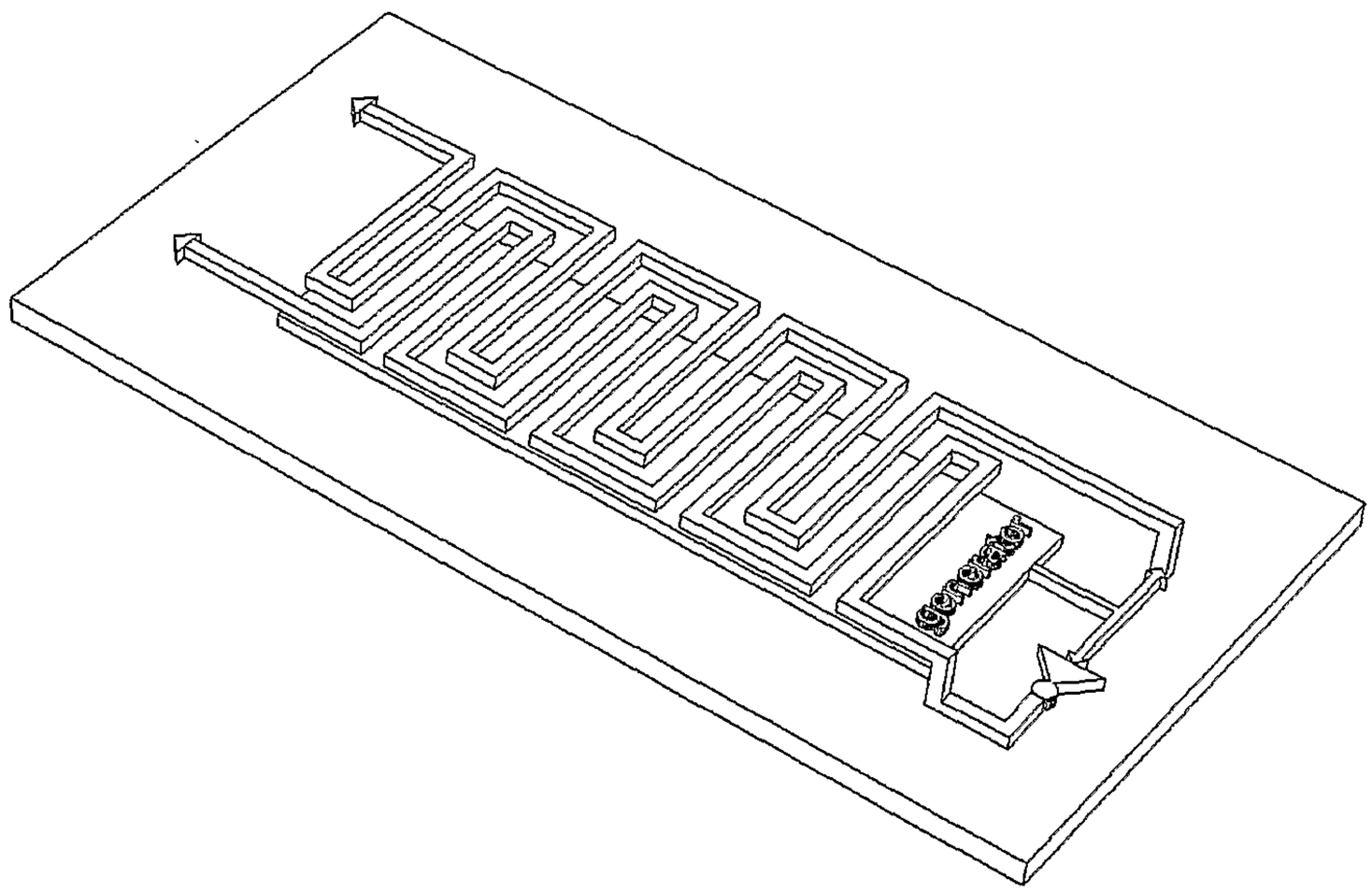


FIG. 352

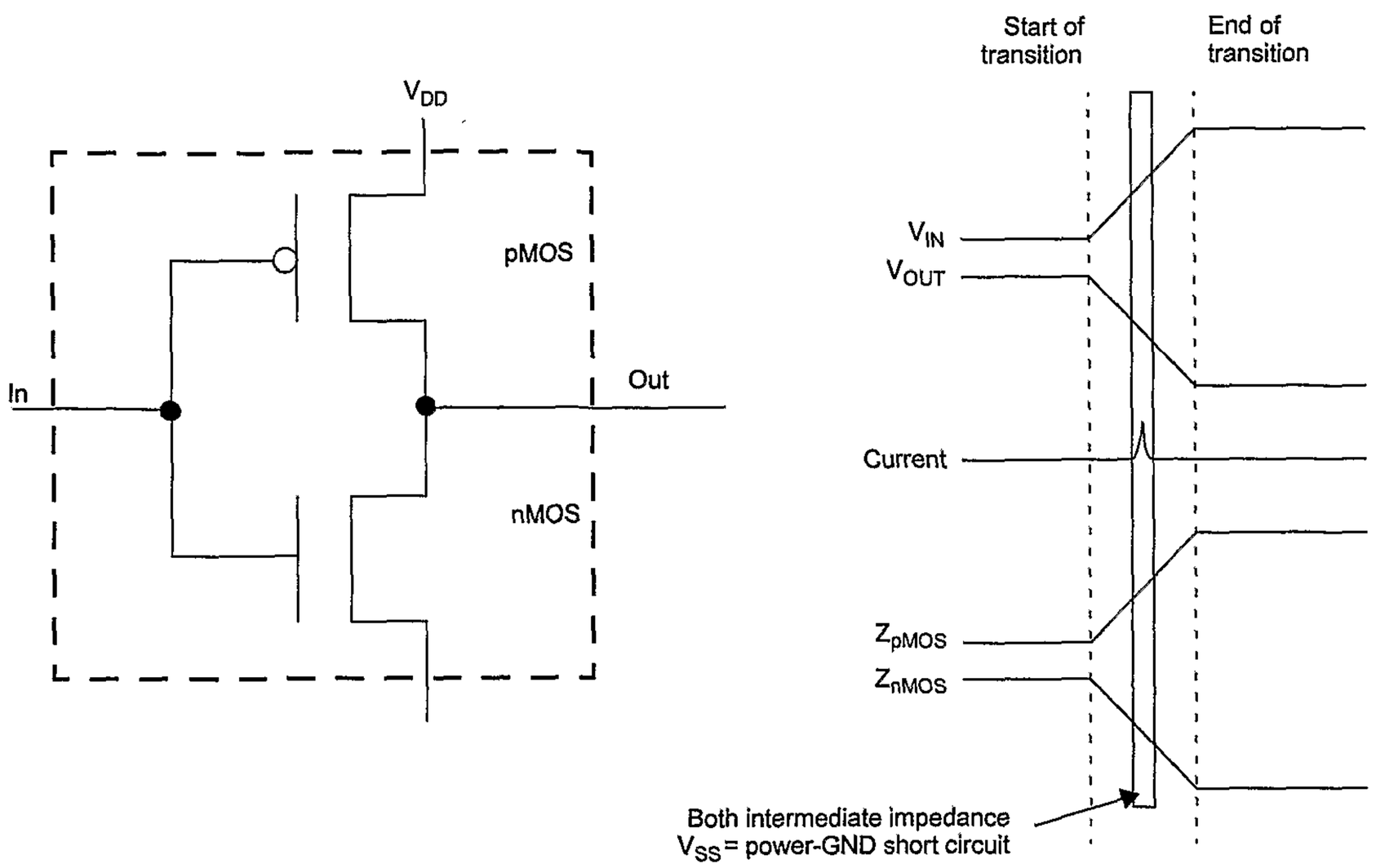


FIG. 353

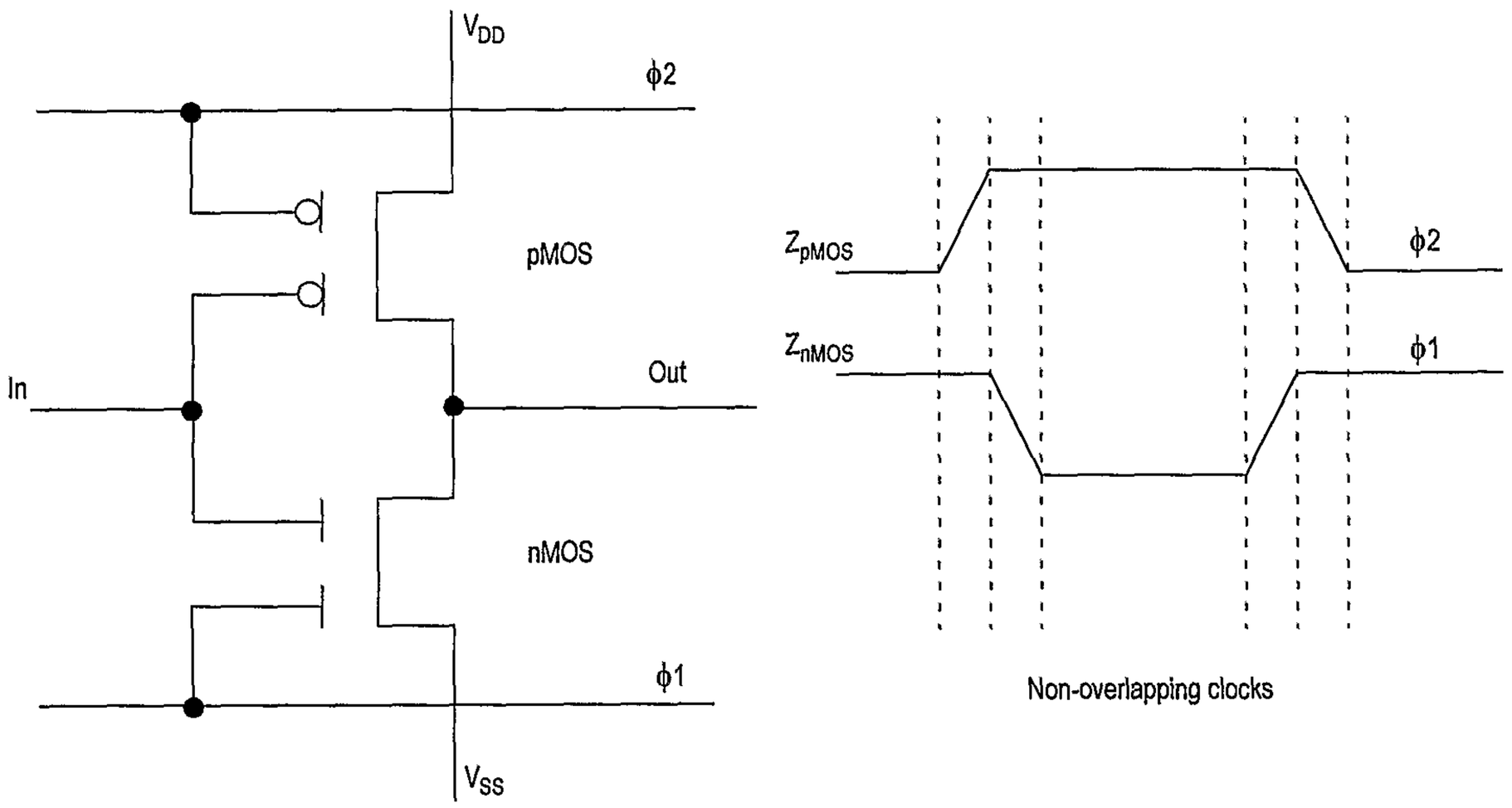


FIG. 354

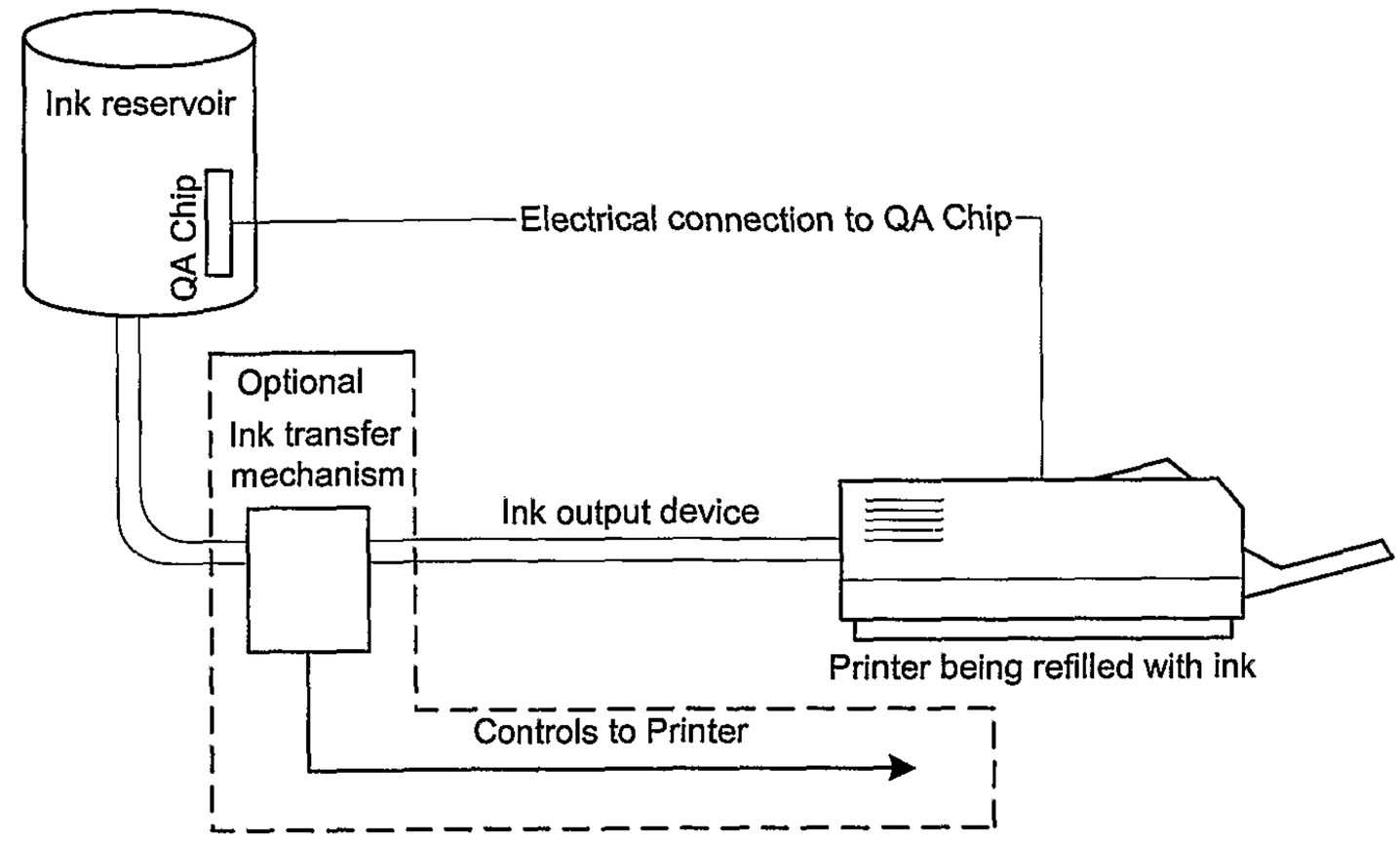


FIG. 355

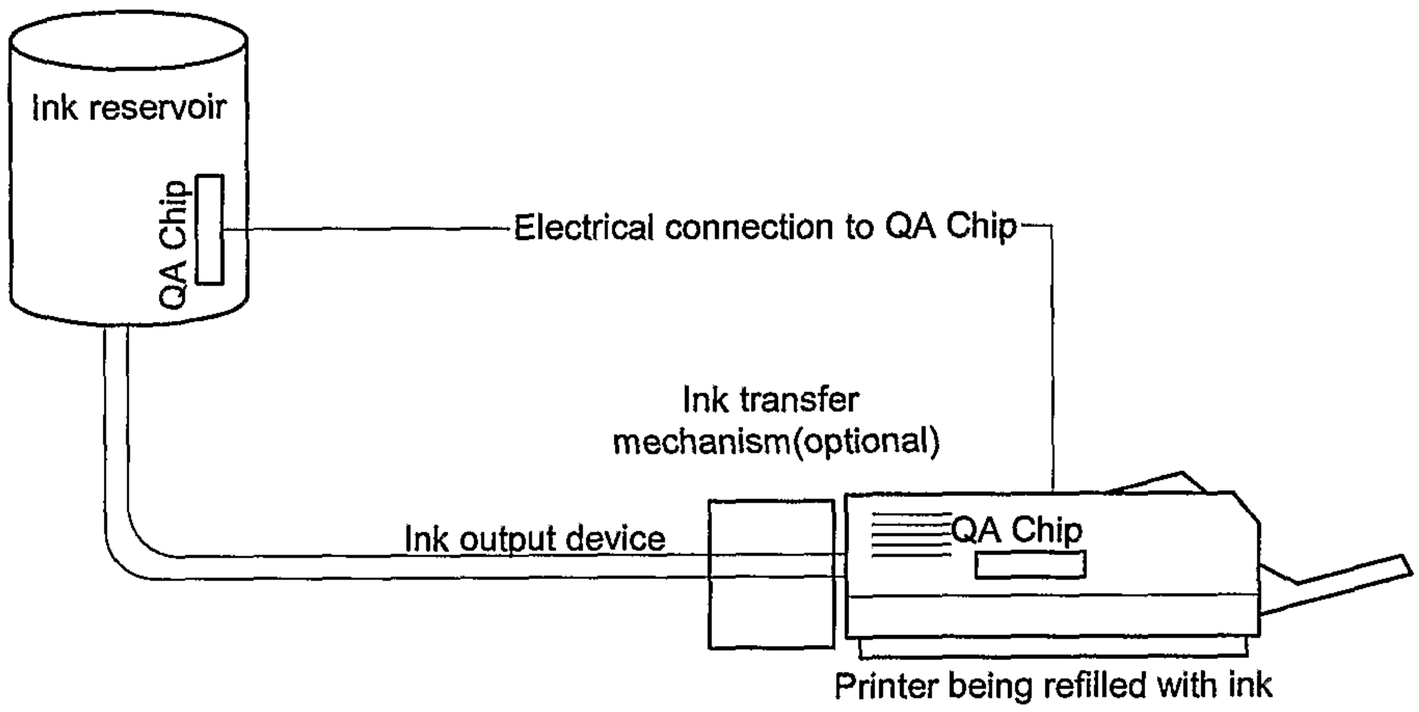


FIG. 356

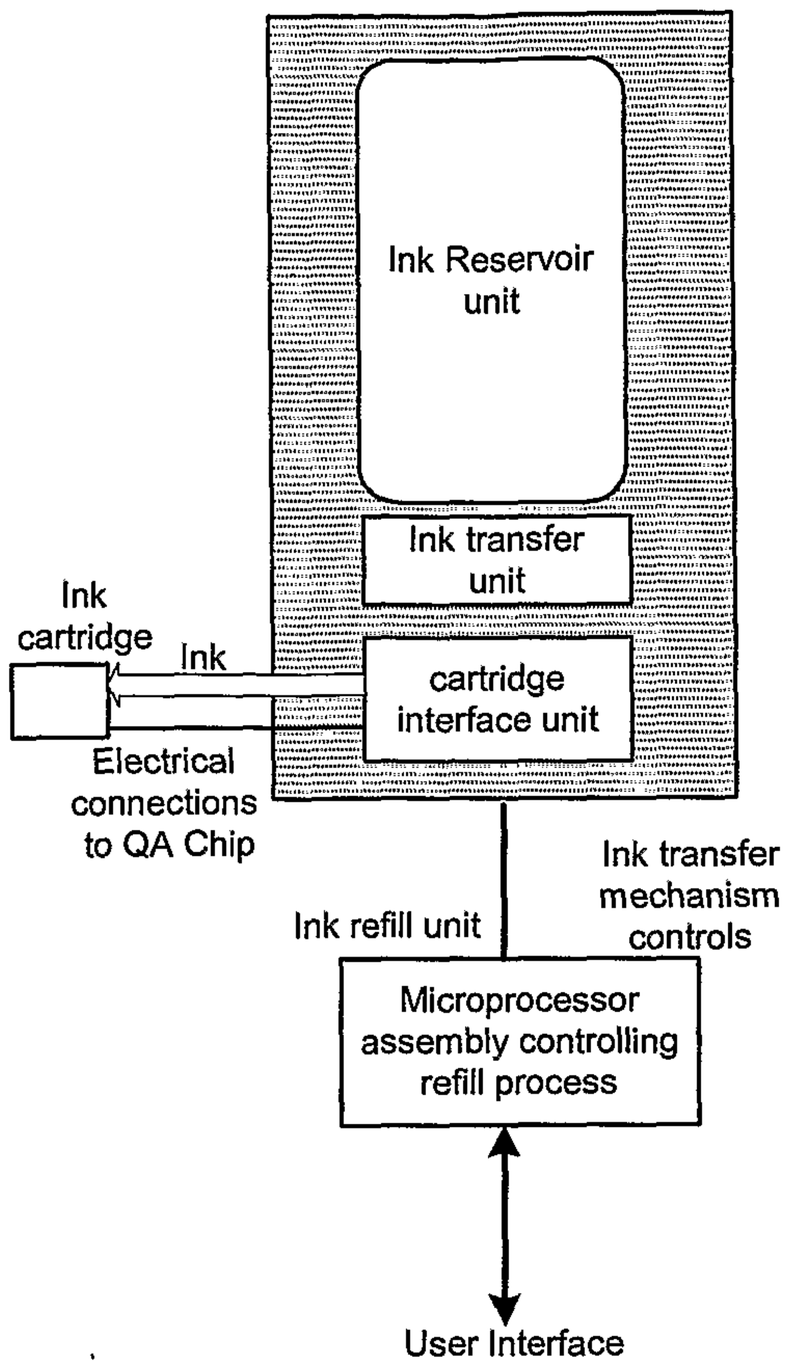


FIG. 357

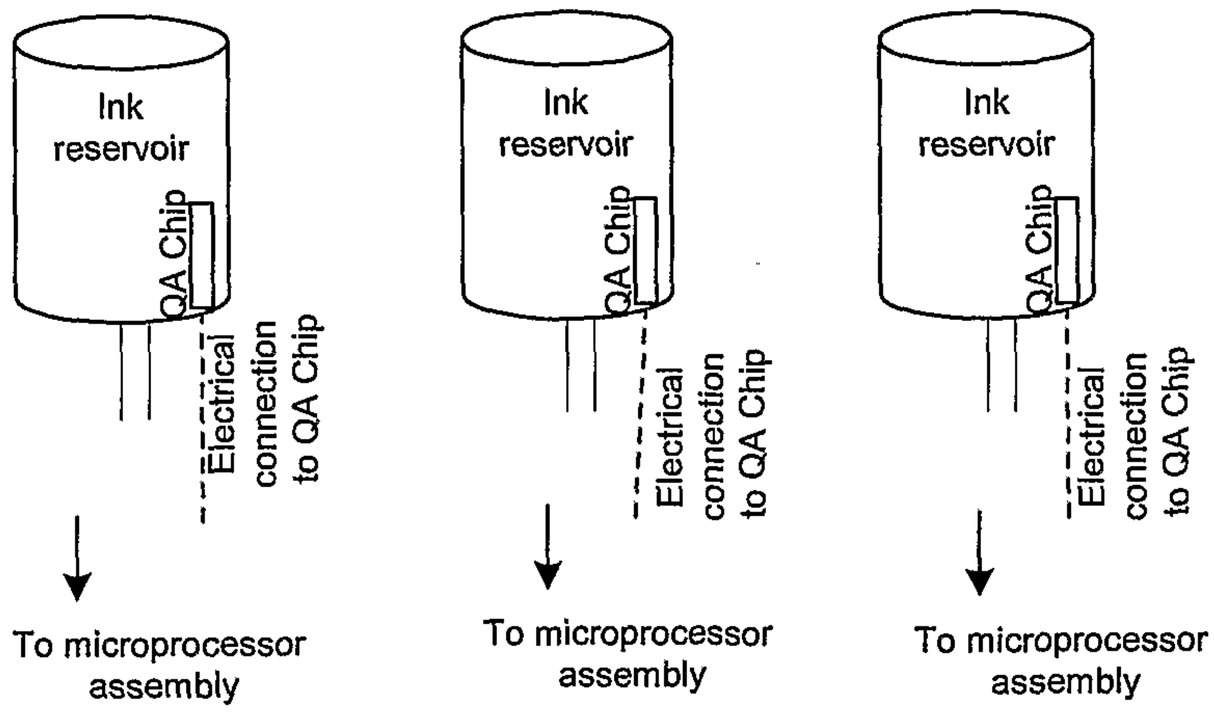


FIG. 358

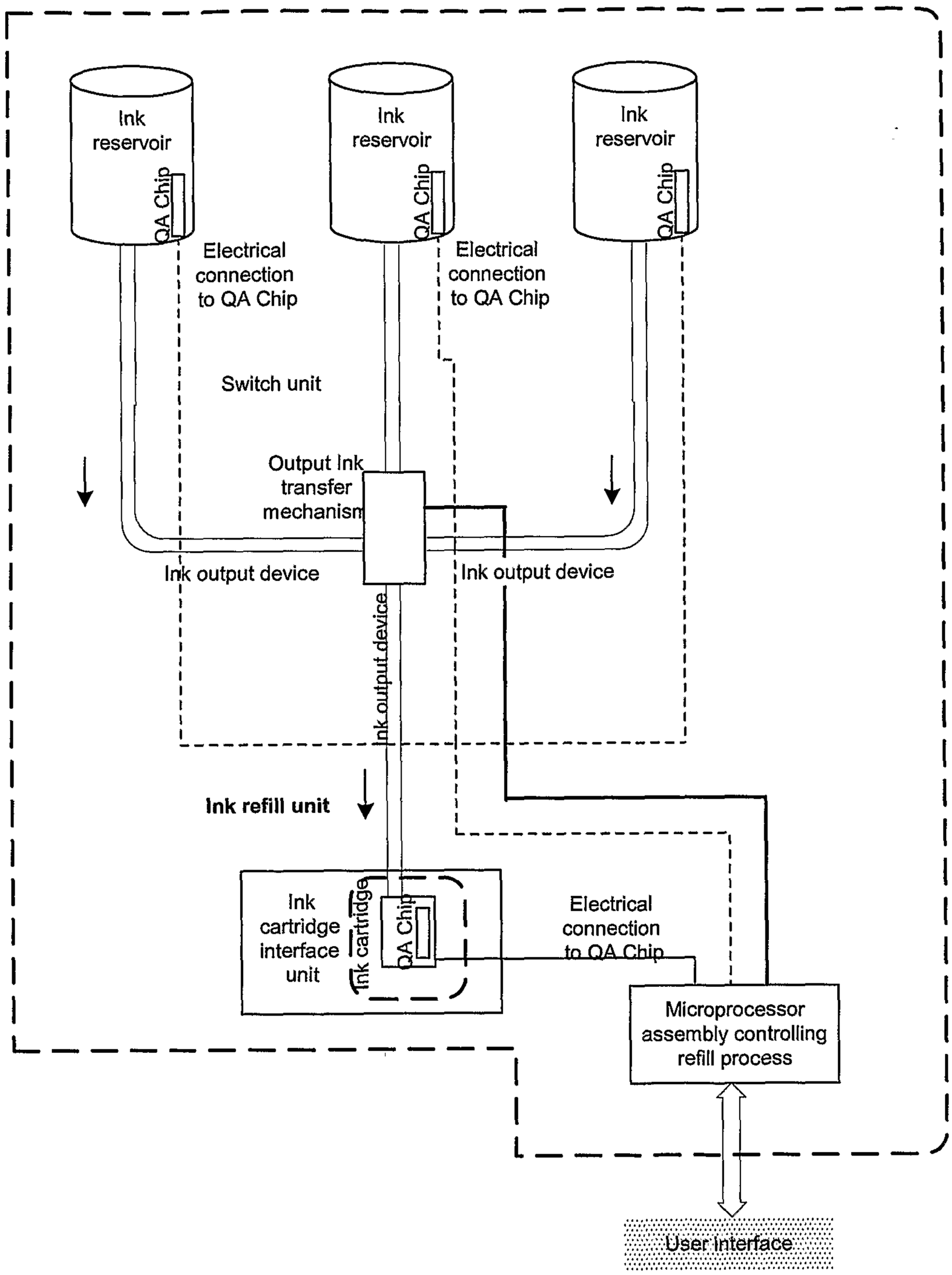

Ink reservoir
QA Chip
Electrical connection to QA Chip
Ink reservoir
QA Chip
Electrical connection to QA Chip
Ink reservoir
QA Chip
Switch unit
Output Ink transfer mechanism
Ink output device
Ink output device
Ink output device
Ink refill unit
Ink cartridge interface unit
Ink cartridge
QA Chip
Electrical connection to QA Chip
Microprocessor assembly controlling refill process
User interface

FIG. 359

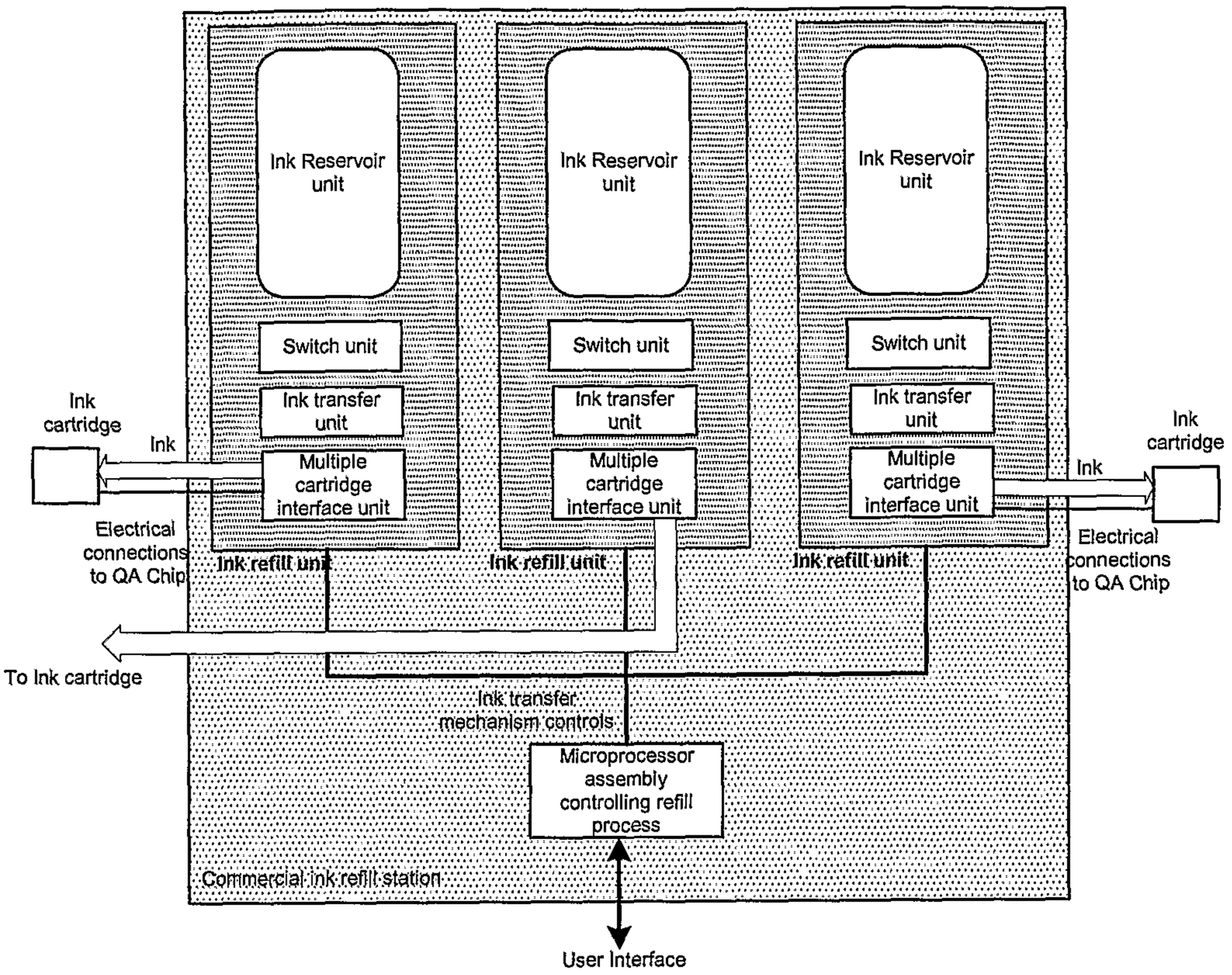


FIG. 360

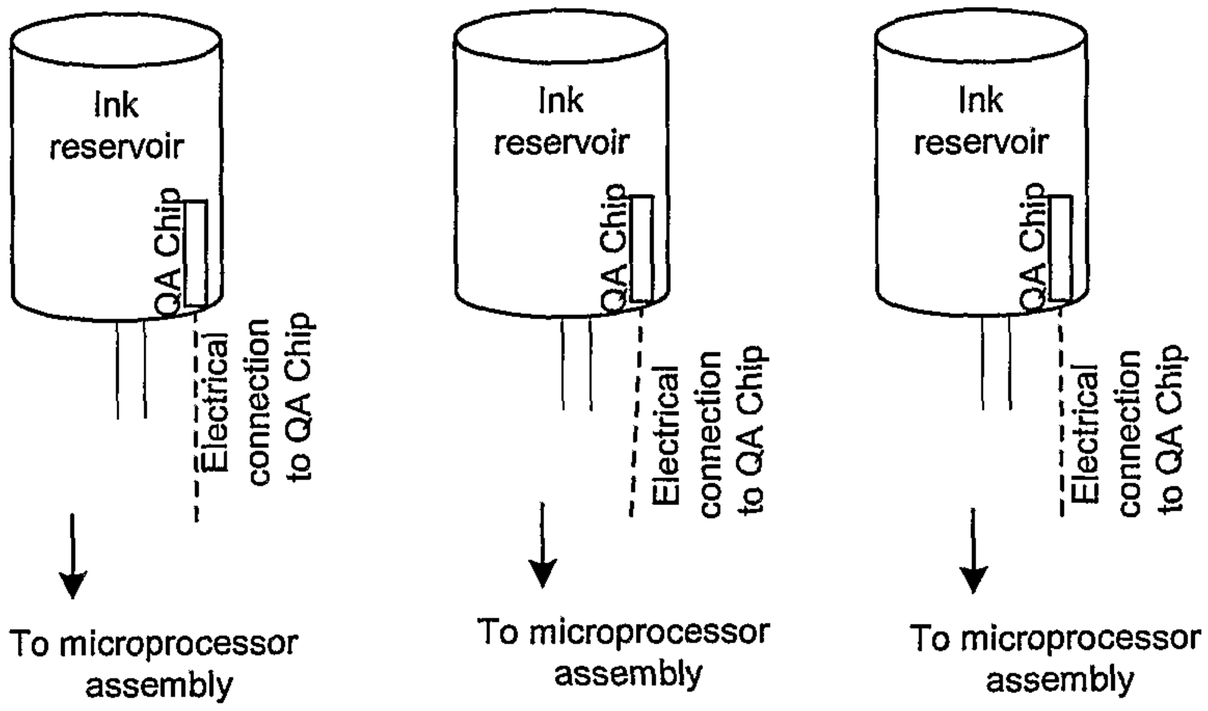


FIG. 361

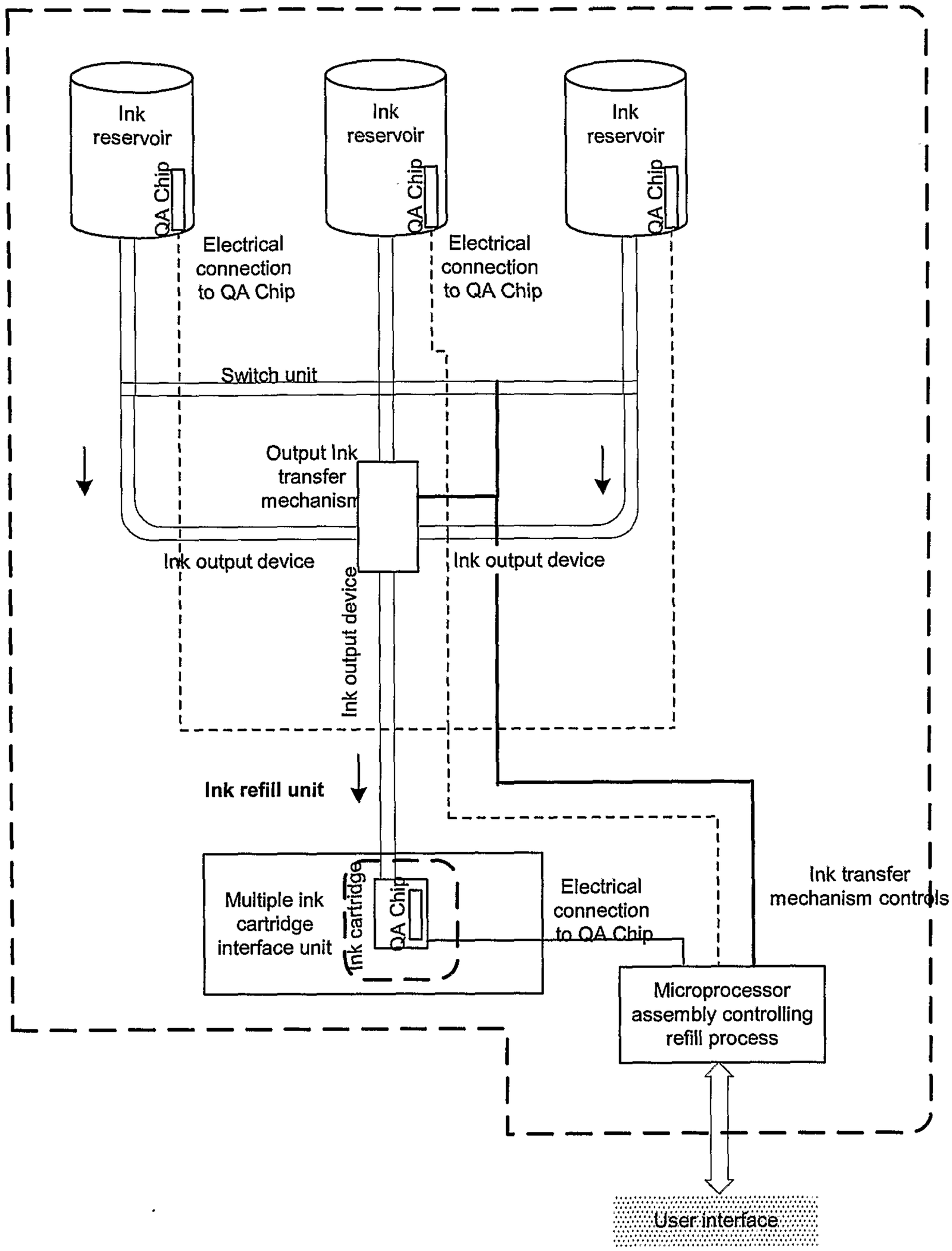

Ink reservoir
QA Chip
Electrical connection to QA Chip
Ink reservoir
QA Chip
Electrical connection to QA Chip
Ink reservoir
QA Chip
Switch unit
Output Ink transfer mechanism
Ink output device
Ink output device
Ink output device
Ink refill unit
Multiple ink cartridge interface unit
Ink cartridge
QA Chip
Electrical connection to QA Chip
Ink transfer mechanism controls
Microprocessor assembly controlling refill process
User interface

FIG. 362

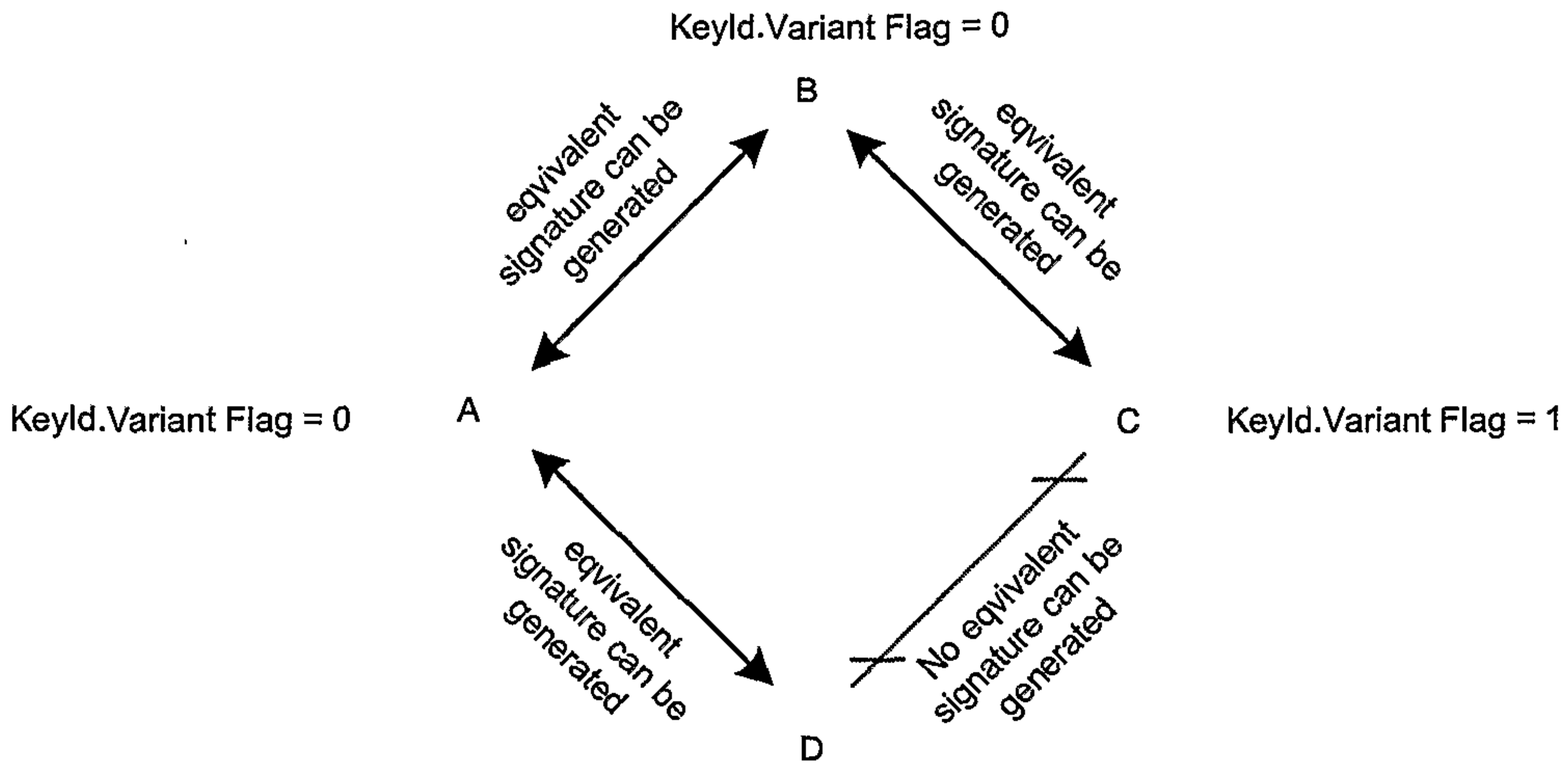


FIG. 363

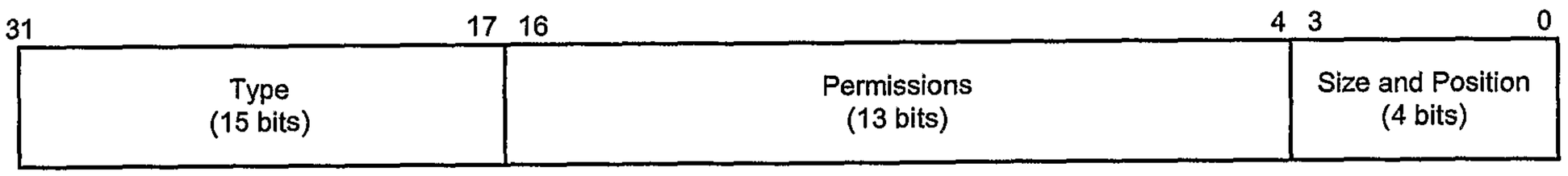


FIG. 364

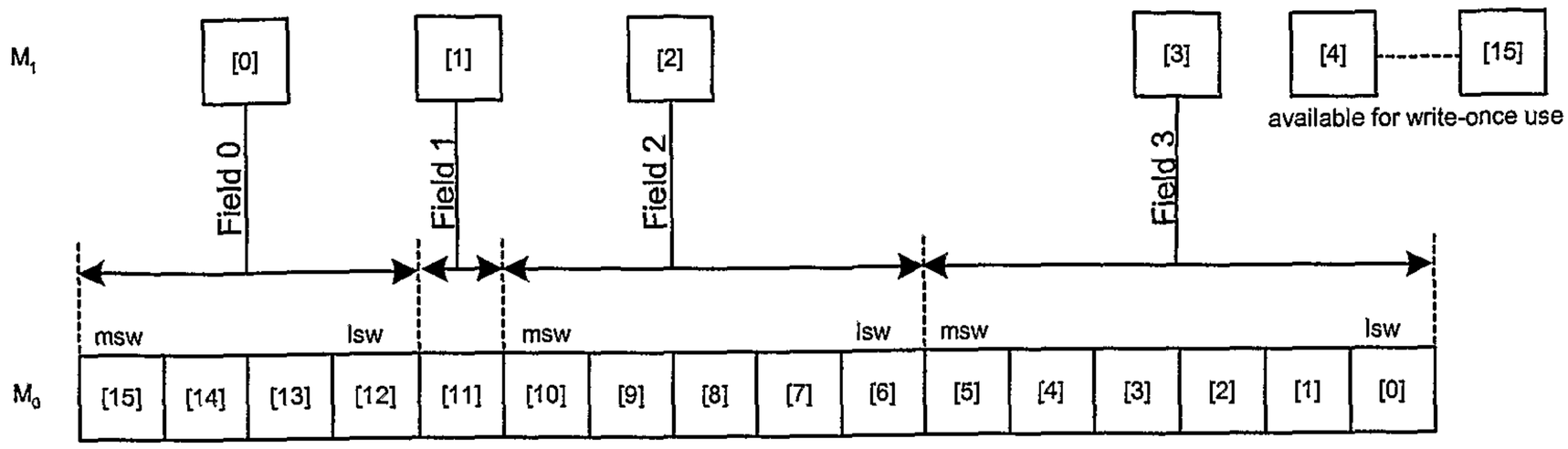


FIG. 365

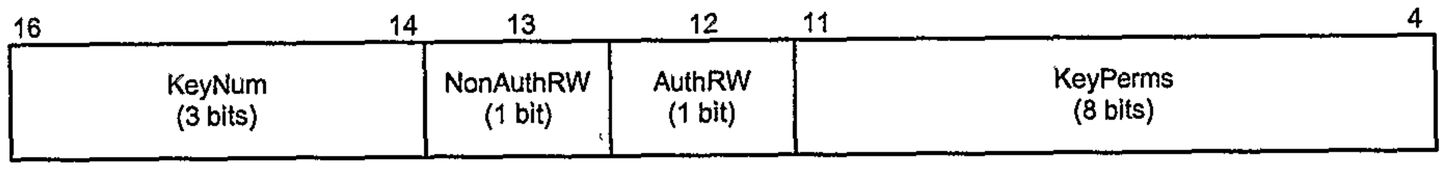


FIG. 366

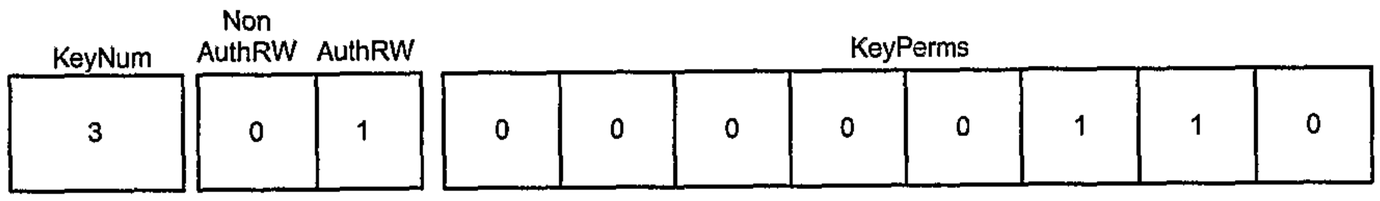


FIG. 367

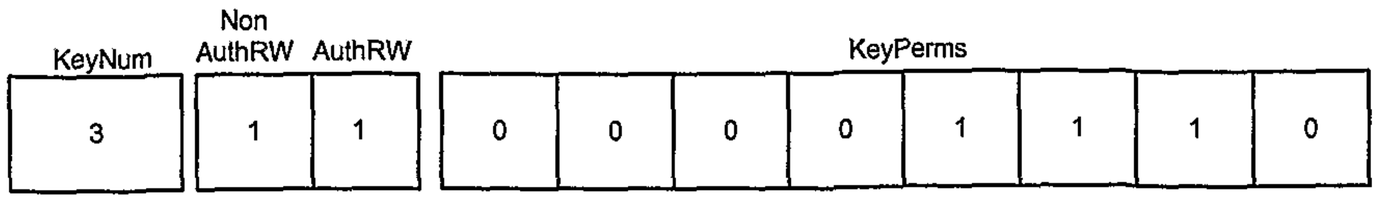


FIG. 368

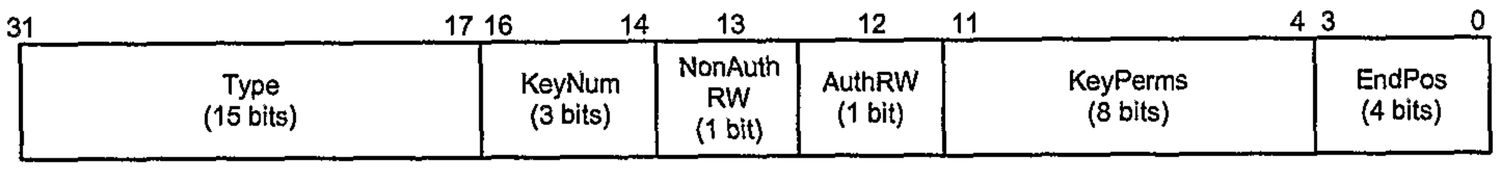


FIG. 369

300/331

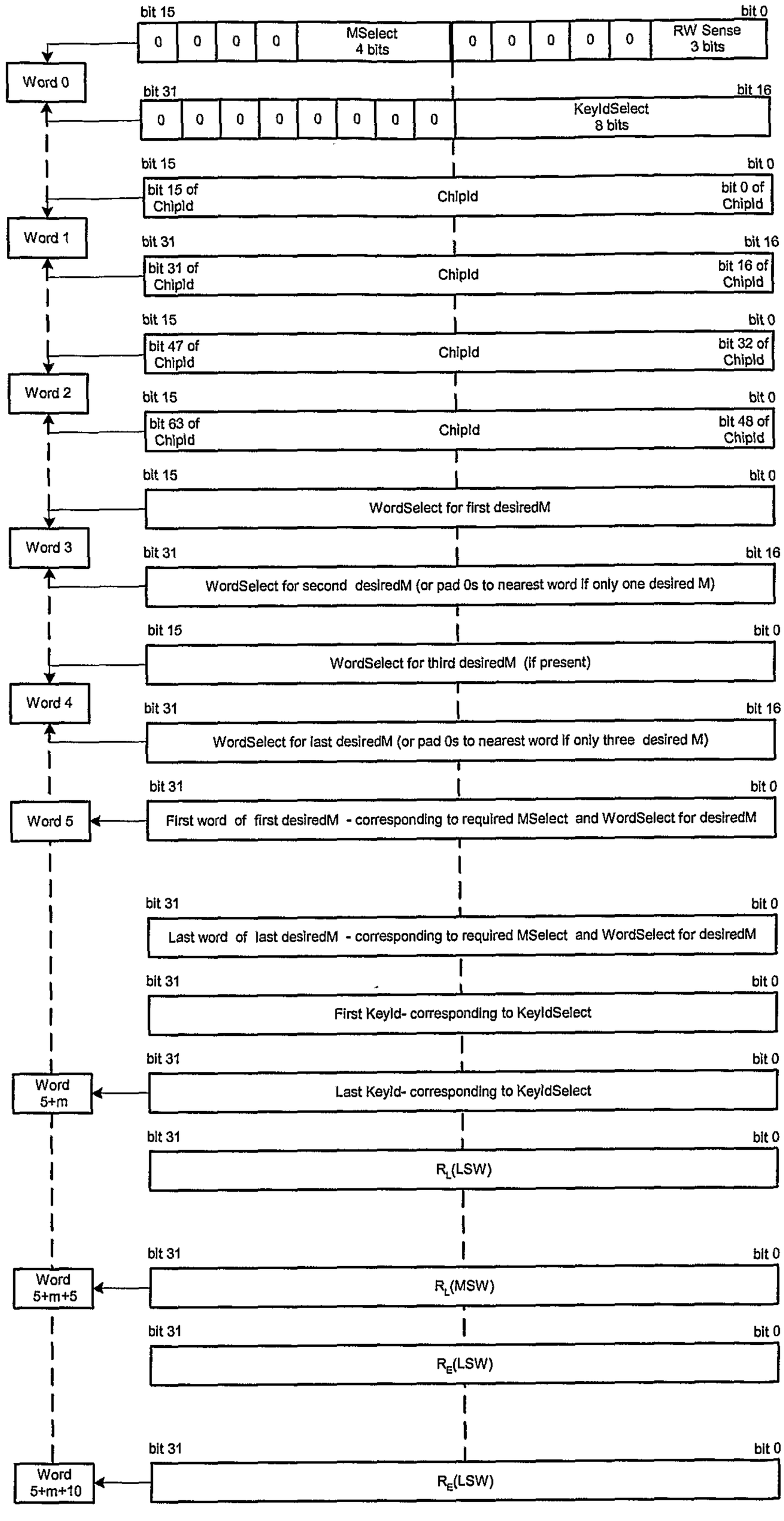


FIG. 370

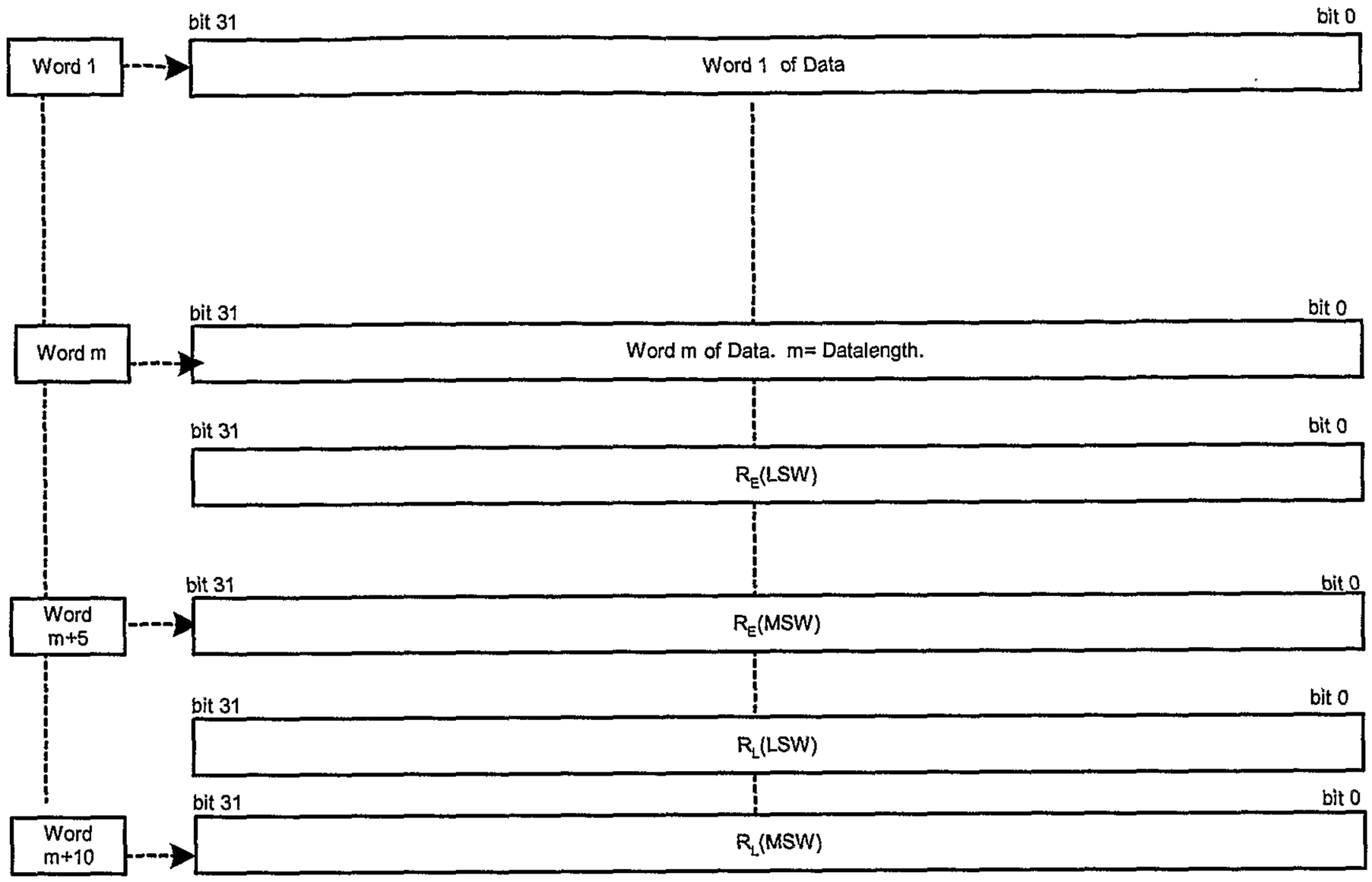

bit 31
bit 0
Word 1
Word 1 of Data
Word m
Word m of Data. m= Datalength.
R_E(LSW)
Word m+5
R_E(MSW)
R_L(LSW)
Word m+10
R_L(MSW)

FIG. 371

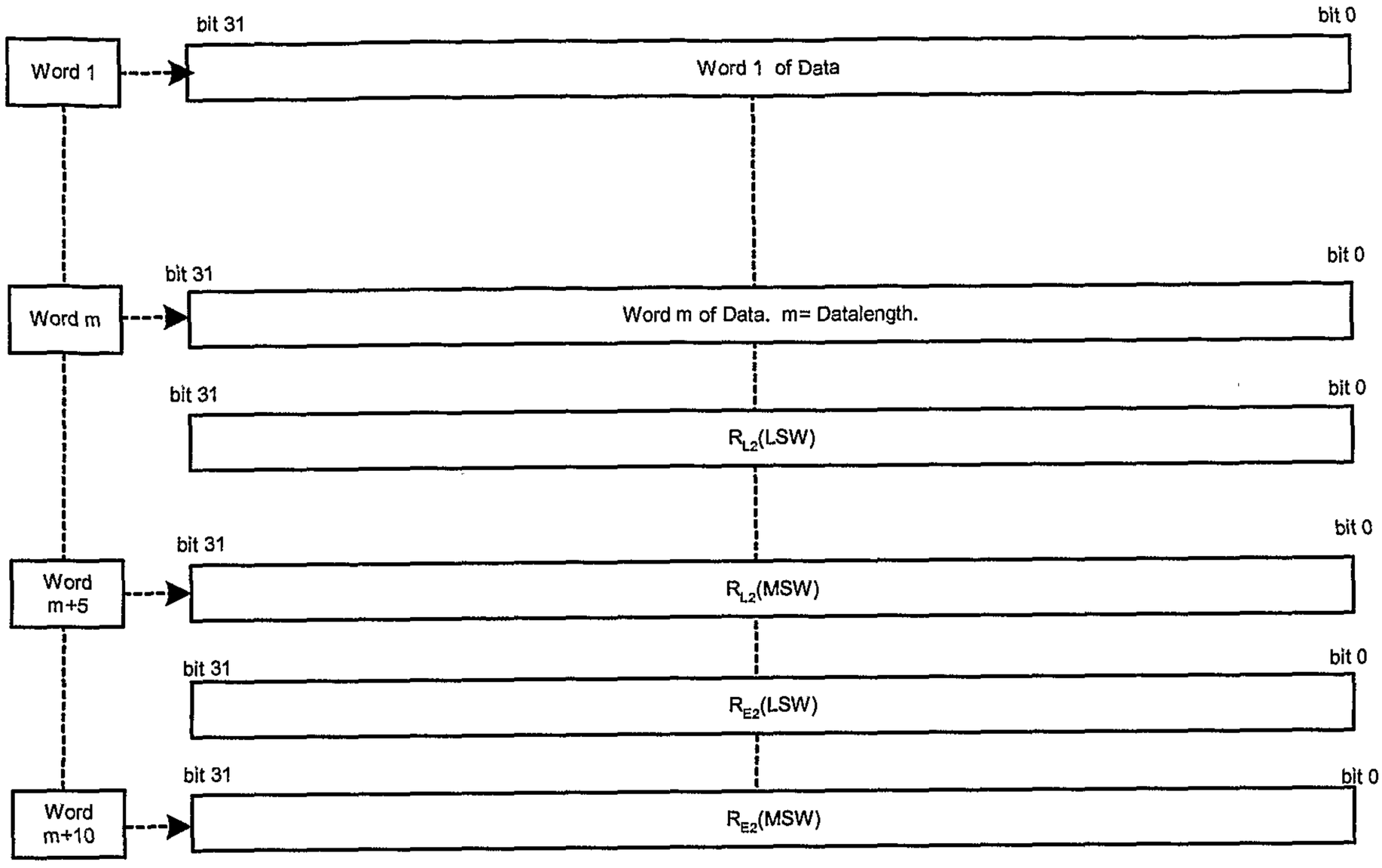

bit 31
bit 0
Word 1
Word 1 of Data
Word m
Word m of Data. m= Datalength.
R_{L2}(LSW)
Word m+5
R_{L2}(MSW)
R_{E2}(LSW)
Word m+10
R_{E2}(MSW)

FIG. 372

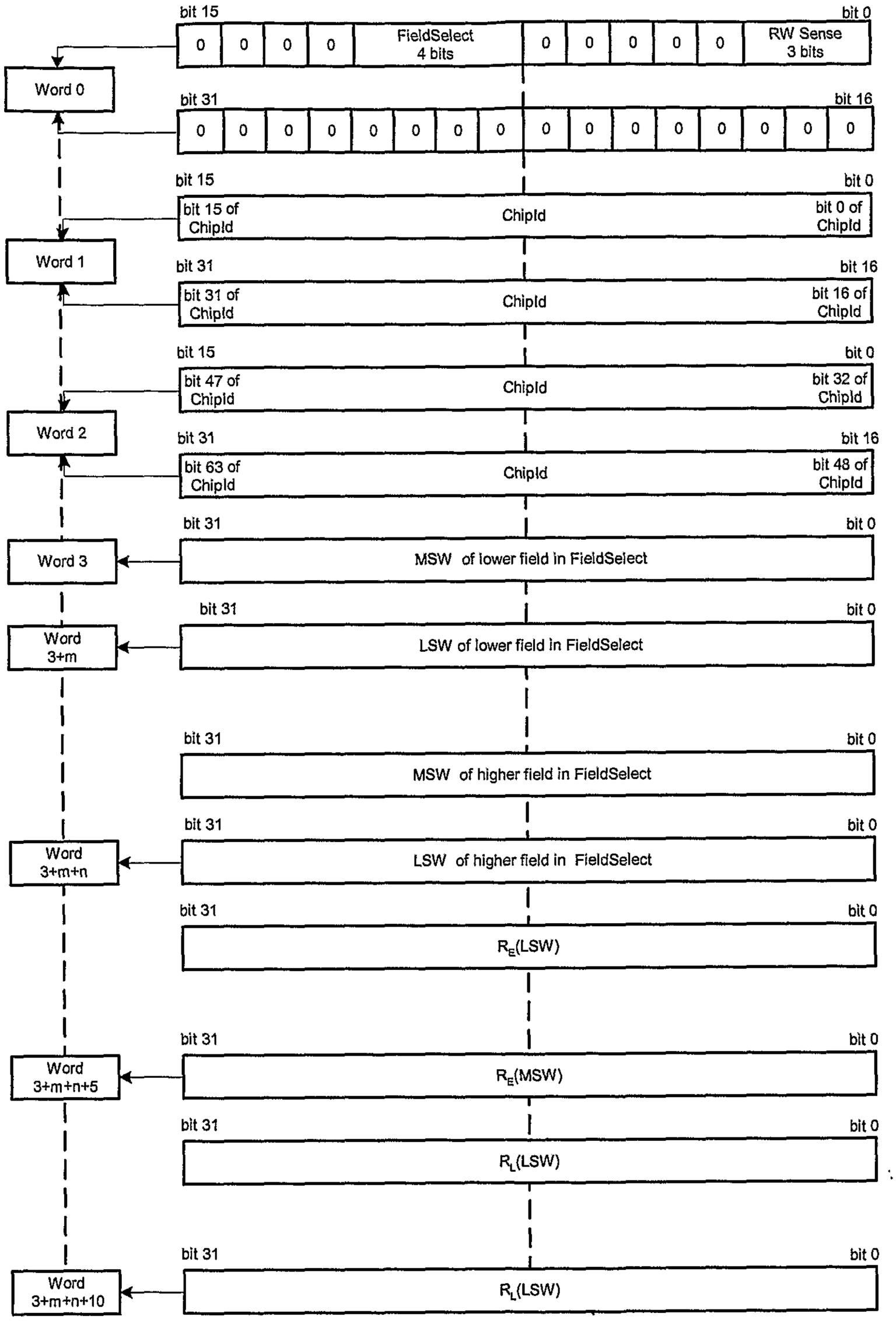

bit 15
bit 0
0 0 0 0 FieldSelect 4 bits 0 0 0 0 0 RW Sense 3 bits
Word 0
bit 31
bit 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
bit 15
bit 0
bit 15 of ChipId
ChipId
bit 0 of ChipId
Word 1
bit 31
bit 16
bit 31 of ChipId
ChipId
bit 16 of ChipId
bit 15
bit 0
bit 47 of ChipId
ChipId
bit 32 of ChipId
Word 2
bit 31
bit 16
bit 63 of ChipId
ChipId
bit 48 of ChipId
bit 31
bit 0
Word 3
MSW of lower field in FieldSelect
bit 31
bit 0
Word 3+m
LSW of lower field in FieldSelect
bit 31
bit 0
MSW of higher field in FieldSelect
bit 31
bit 0
Word 3+m+n
LSW of higher field in FieldSelect
bit 31
bit 0
R_E(LSW)
bit 31
bit 0
Word 3+m+n+5
R_E(MSW)
bit 31
bit 0
R_L(LSW)
bit 31
bit 0
Word 3+m+n+10
R_L(LSW)

FIG. 373

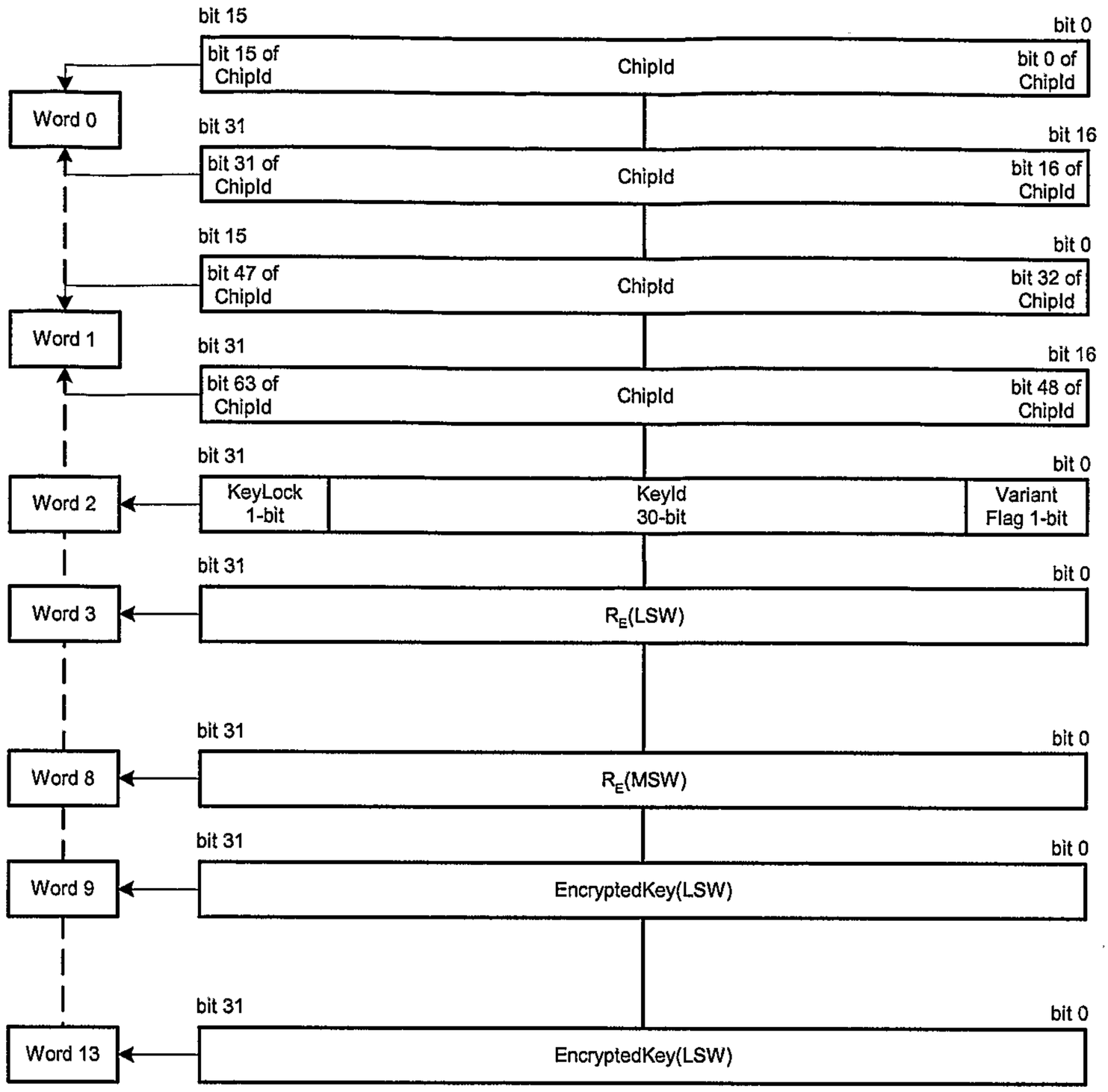

bit 15
bit 0
bit 15 of ChipId
ChipId
bit 0 of ChipId
Word 0
bit 31
bit 16
bit 31 of ChipId
ChipId
bit 16 of ChipId
bit 15
bit 0
bit 47 of ChipId
ChipId
bit 32 of ChipId
Word 1
bit 31
bit 16
bit 63 of ChipId
ChipId
bit 48 of ChipId
bit 31
bit 0
Word 2
KeyLock 1-bit
KeyId 30-bit
Variant Flag 1-bit
bit 31
bit 0
Word 3
R_E(LSW)
bit 31
bit 0
Word 8
R_E(MSW)
bit 31
bit 0
Word 9
EncryptedKey(LSW)
bit 31
bit 0
Word 13
EncryptedKey(LSW)

FIG. 374

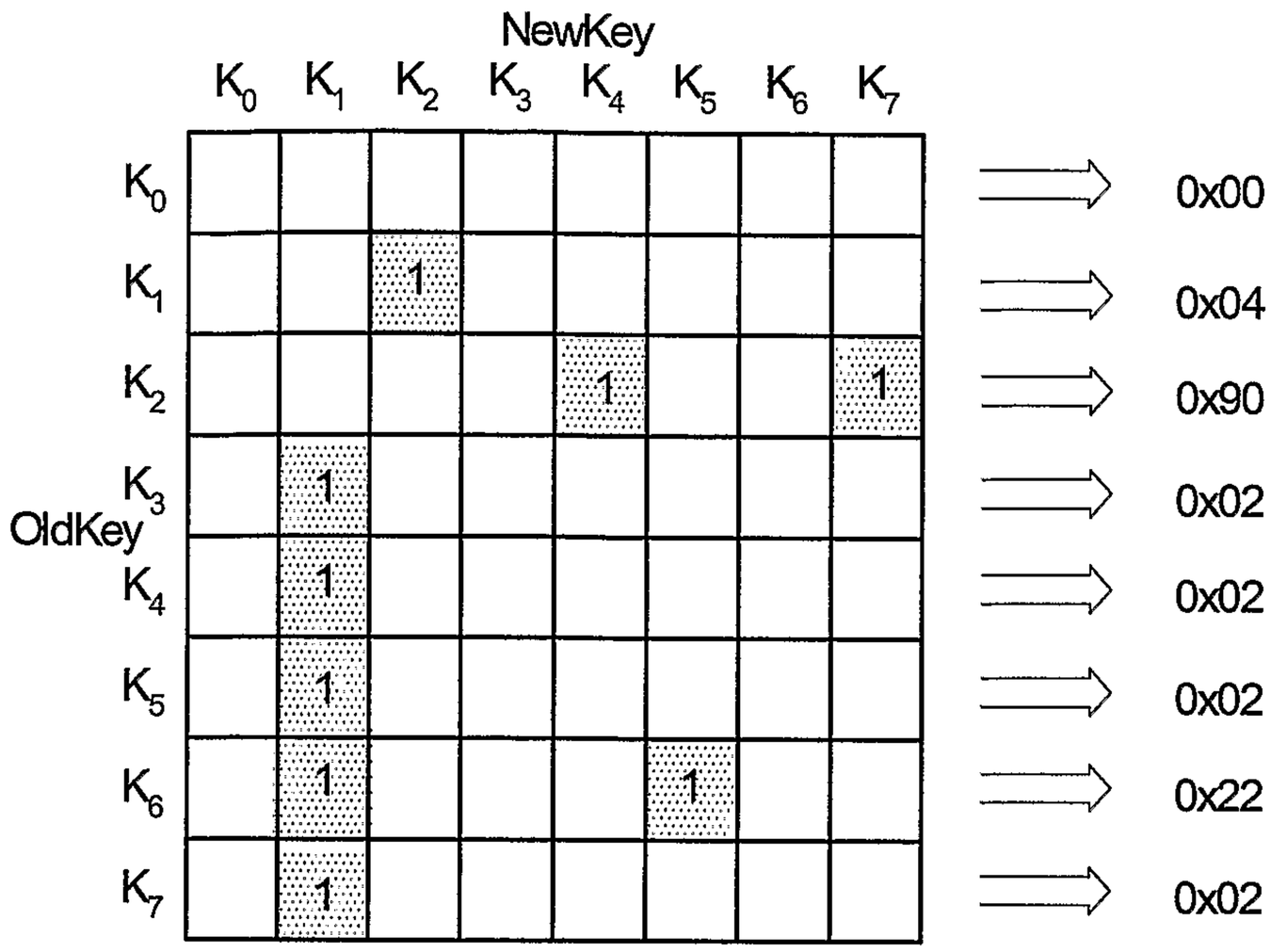


FIG. 375

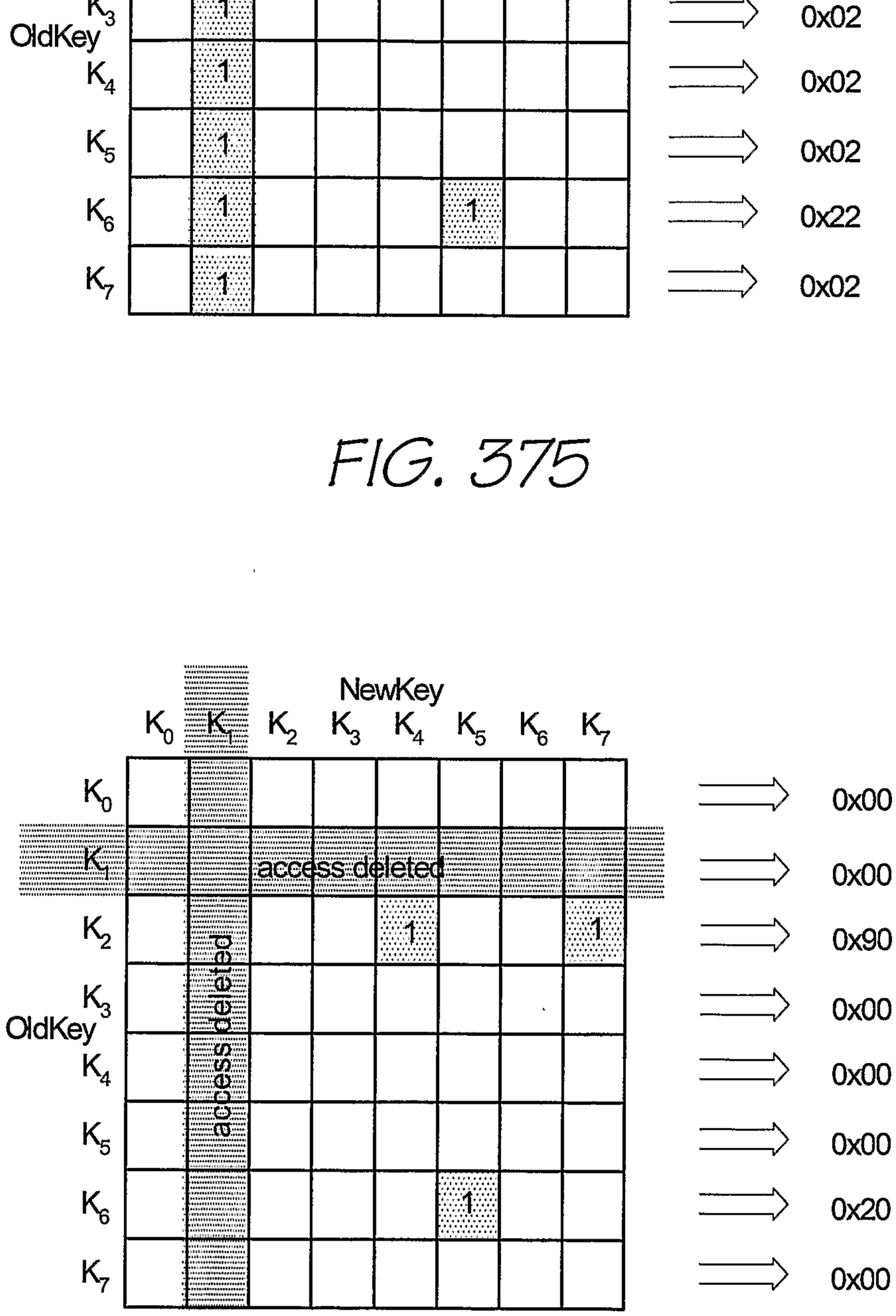


FIG. 376

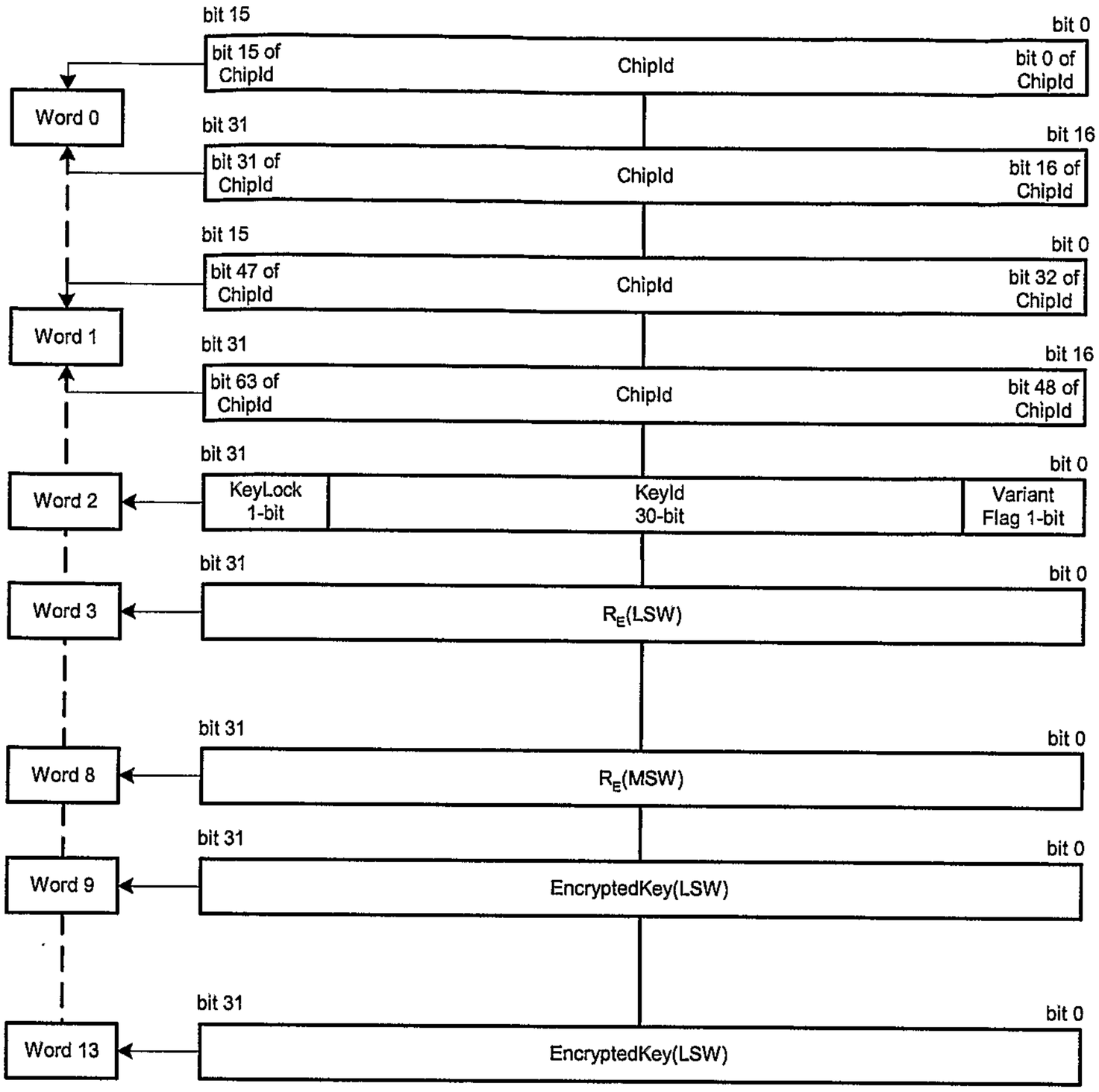

bit 15
bit 0
bit 15 of ChipId
ChipId
bit 0 of ChipId
Word 0
bit 31
bit 16
bit 31 of ChipId
ChipId
bit 16 of ChipId
bit 15
bit 0
bit 47 of ChipId
ChipId
bit 32 of ChipId
Word 1
bit 31
bit 16
bit 63 of ChipId
ChipId
bit 48 of ChipId
bit 31
bit 0
Word 2
KeyLock 1-bit
KeyId 30-bit
Variant Flag 1-bit
bit 31
bit 0
Word 3
R_E(LSW)
bit 31
bit 0
Word 8
R_E(MSW)
bit 31
bit 0
Word 9
EncryptedKey(LSW)
bit 31
bit 0
Word 13
EncryptedKey(LSW)

FIG. 377

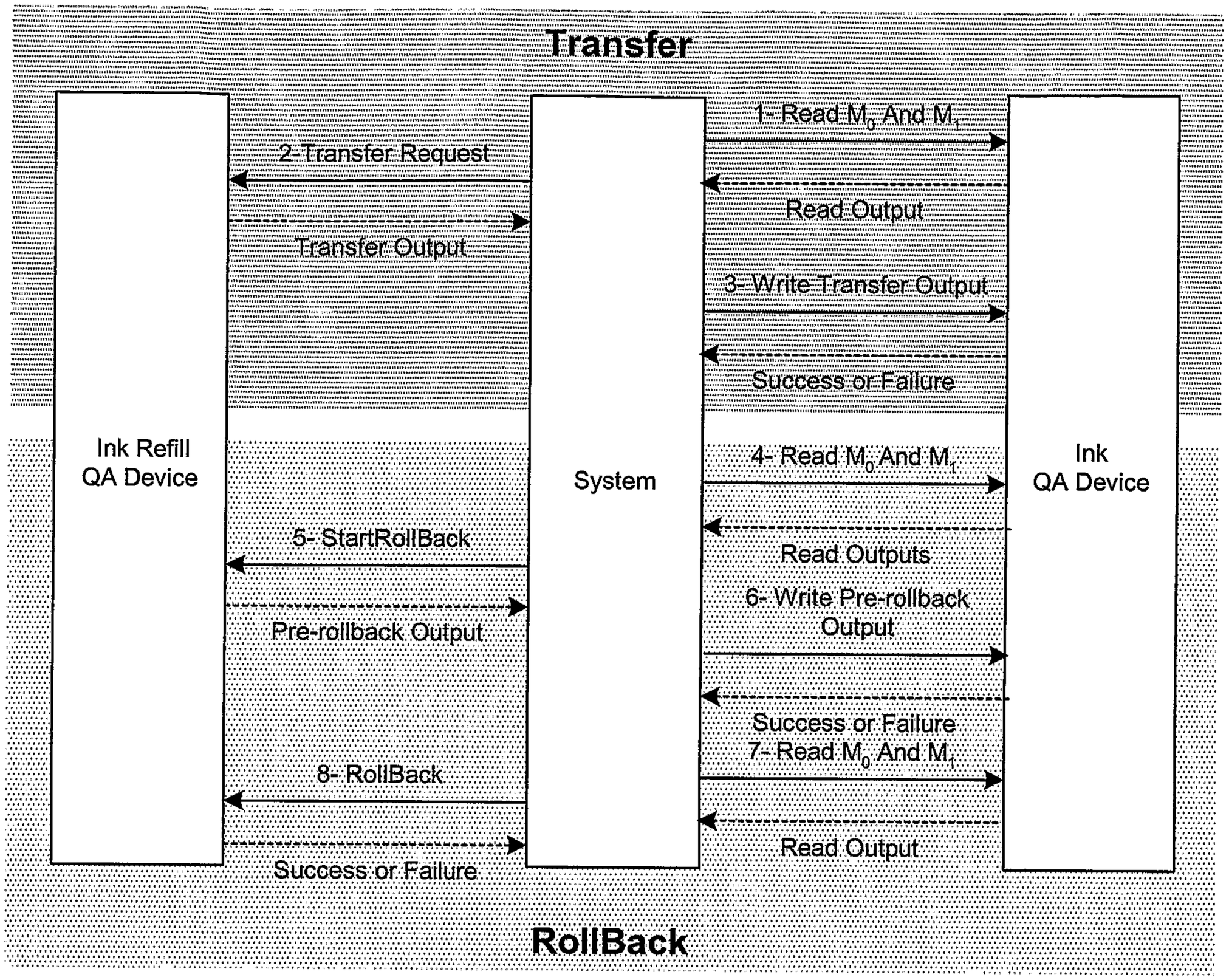

Transfer
Ink Refill QA Device
System
Ink QA Device
1- Read M_0 And M_1
2- Transfer Request
Transfer Output
Read Output
3- Write Transfer Output
Success or Failure
4- Read M_0 And M_1
5- StartRollBack
Pre-rollback Output
Read Outputs
6- Write Pre-rollback Output
Success or Failure
7- Read M_0 And M_1
8- RollBack
Read Output
Success or Failure
RollBack

FIG. 378

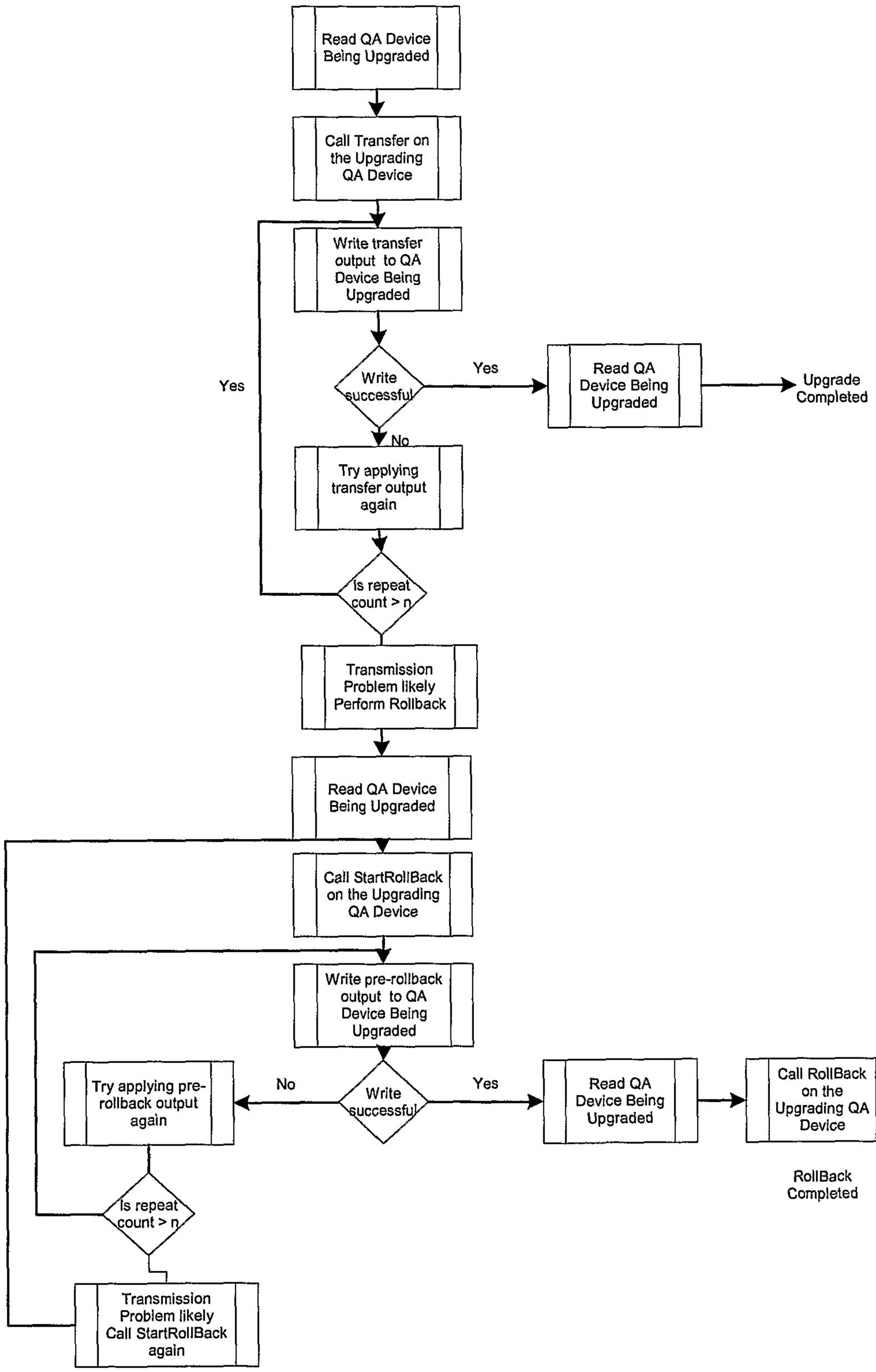

Read QA Device Being Upgraded
Call Transfer on the Upgrading QA Device
Write transfer output to QA Device Being Upgraded
Write successful
Yes
Read QA Device Being Upgraded
Upgrade Completed
No
Try applying transfer output again
Is repeat count > n
Yes
Transmission Problem likely Perform Rollback
Read QA Device Being Upgraded
Call StartRollBack on the Upgrading QA Device
Write pre-rollback output to QA Device Being Upgraded
Write successful
No
Try applying pre-rollback output again
Yes
Read QA Device Being Upgraded
Call RollBack on the Upgrading QA Device
RollBack Completed
Is repeat count > n
Transmission Problem likely Call StartRollBack again

FIG. 379

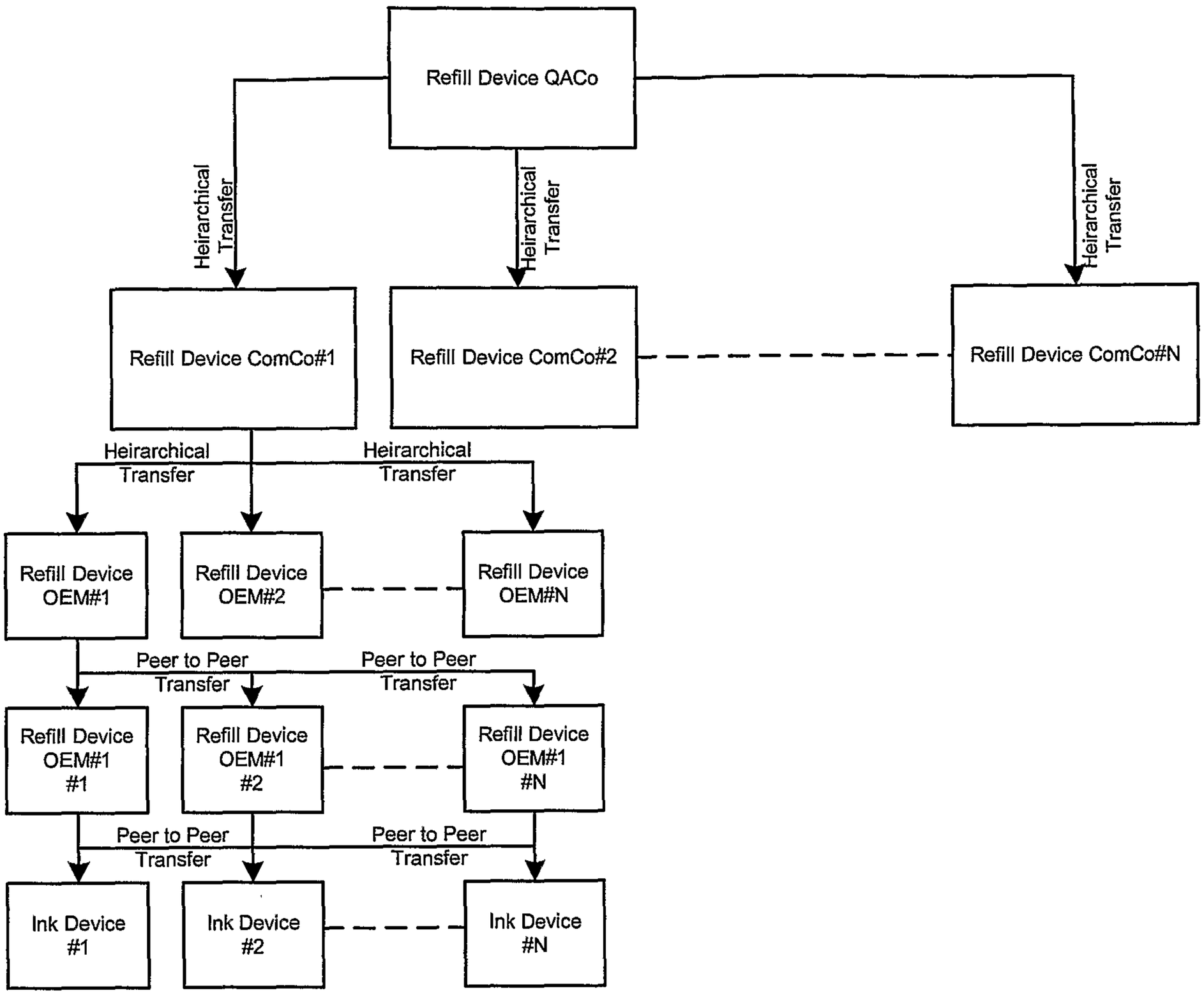

Refill Device QACo
Heirarchical Transfer
Heirarchical Transfer
Heirarchical Transfer
Refill Device ComCo#1
Refill Device ComCo#2
Refill Device ComCo#N
Heirarchical Transfer
Heirarchical Transfer
Refill Device OEM#1
Refill Device OEM#2
Refill Device OEM#N
Peer to Peer Transfer
Peer to Peer Transfer
Refill Device OEM#1 #1
Refill Device OEM#1 #2
Refill Device OEM#1 #N
Peer to Peer Transfer
Peer to Peer Transfer
Ink Device #1
Ink Device #2
Ink Device #N

FIG. 380

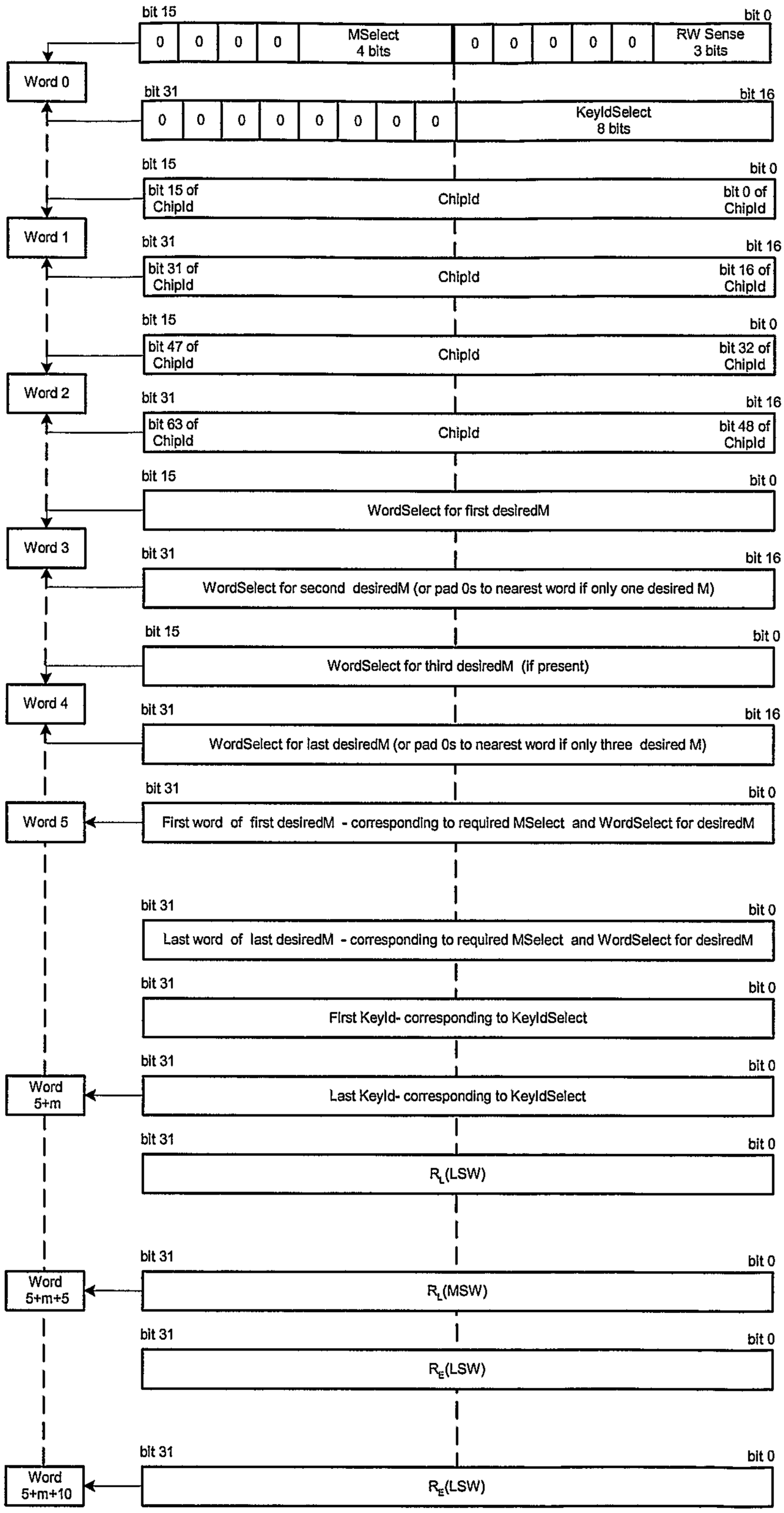

Word 0
bit 15
bit 0
0 0 0 0 MSelect 4 bits 0 0 0 0 0 RW Sense 3 bits
bit 31
bit 16
0 0 0 0 0 0 0 0 KeyIdSelect 8 bits
Word 1
bit 15
bit 0
bit 15 of ChipId
ChipId
bit 0 of ChipId
bit 31
bit 16
bit 31 of ChipId
ChipId
bit 16 of ChipId
Word 2
bit 15
bit 0
bit 47 of ChipId
ChipId
bit 32 of ChipId
bit 31
bit 16
bit 63 of ChipId
ChipId
bit 48 of ChipId
Word 3
bit 15
bit 0
WordSelect for first desiredM
bit 31
bit 16
WordSelect for second desiredM (or pad 0s to nearest word if only one desired M)
Word 4
bit 15
bit 0
WordSelect for third desiredM (if present)
bit 31
bit 16
WordSelect for last desiredM (or pad 0s to nearest word if only three desired M)
Word 5
bit 31
bit 0
First word of first desiredM - corresponding to required MSelect and WordSelect for desiredM
bit 31
bit 0
Last word of last desiredM - corresponding to required MSelect and WordSelect for desiredM
bit 31
bit 0
First KeyId- corresponding to KeyIdSelect
Word 5+m
bit 31
bit 0
Last KeyId- corresponding to KeyIdSelect
bit 31
bit 0
R_L(LSW)
Word 5+m+5
bit 31
bit 0
R_L(MSW)
bit 31
bit 0
R_E(LSW)
Word 5+m+10
bit 31
bit 0
R_E(LSW)

FIG. 381

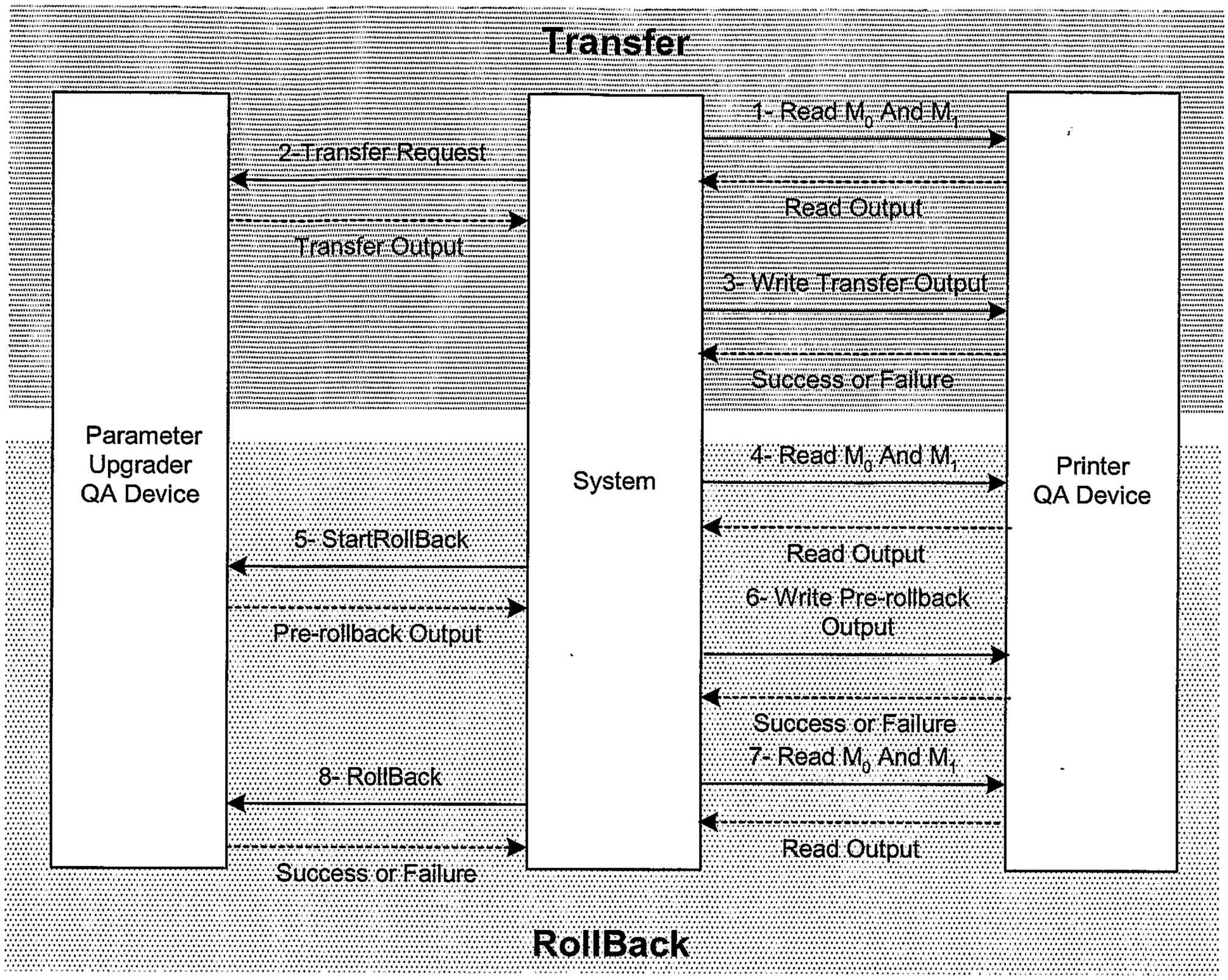

Transfer
Parameter Upgrader QA Device
System
Printer QA Device
1- Read M_0 And M_1
2-Transfer Request
Transfer Output
Read Output
3- Write Transfer Output
Success or Failure
4- Read M_0 And M_1
5- StartRollBack
Pre-rollback Output
Read Output
6- Write Pre-rollback Output
Success or Failure
7- Read M_0 And M_1
8- RollBack
Read Output
Success or Failure
RollBack

FIG. 382

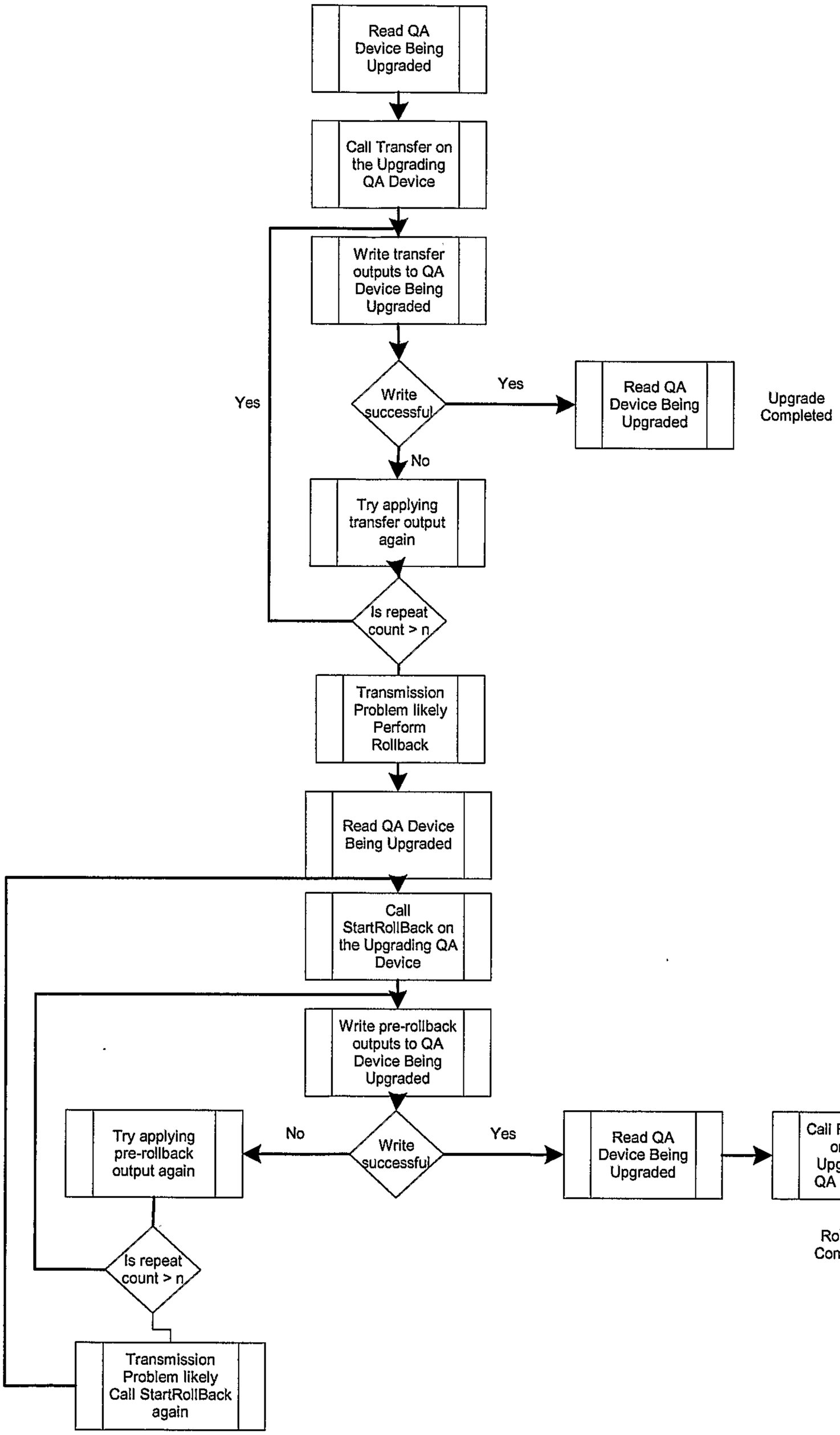

Read QA Device Being Upgraded
Call Transfer on the Upgrading QA Device
Write transfer outputs to QA Device Being Upgraded
Write successful
Yes
Read QA Device Being Upgraded
Upgrade Completed
No
Try applying transfer output again
Is repeat count > n
Yes
Transmission Problem likely Perform Rollback
Read QA Device Being Upgraded
Call StartRollBack on the Upgrading QA Device
Write pre-rollback outputs to QA Device Being Upgraded
Write successful
No
Try applying pre-rollback output again
Is repeat count > n
Yes
Read QA Device Being Upgraded
Call RollBack on the Upgrading QA Device
RollBack Completed
Transmission Problem likely Call StartRollBack again

FIG. 383

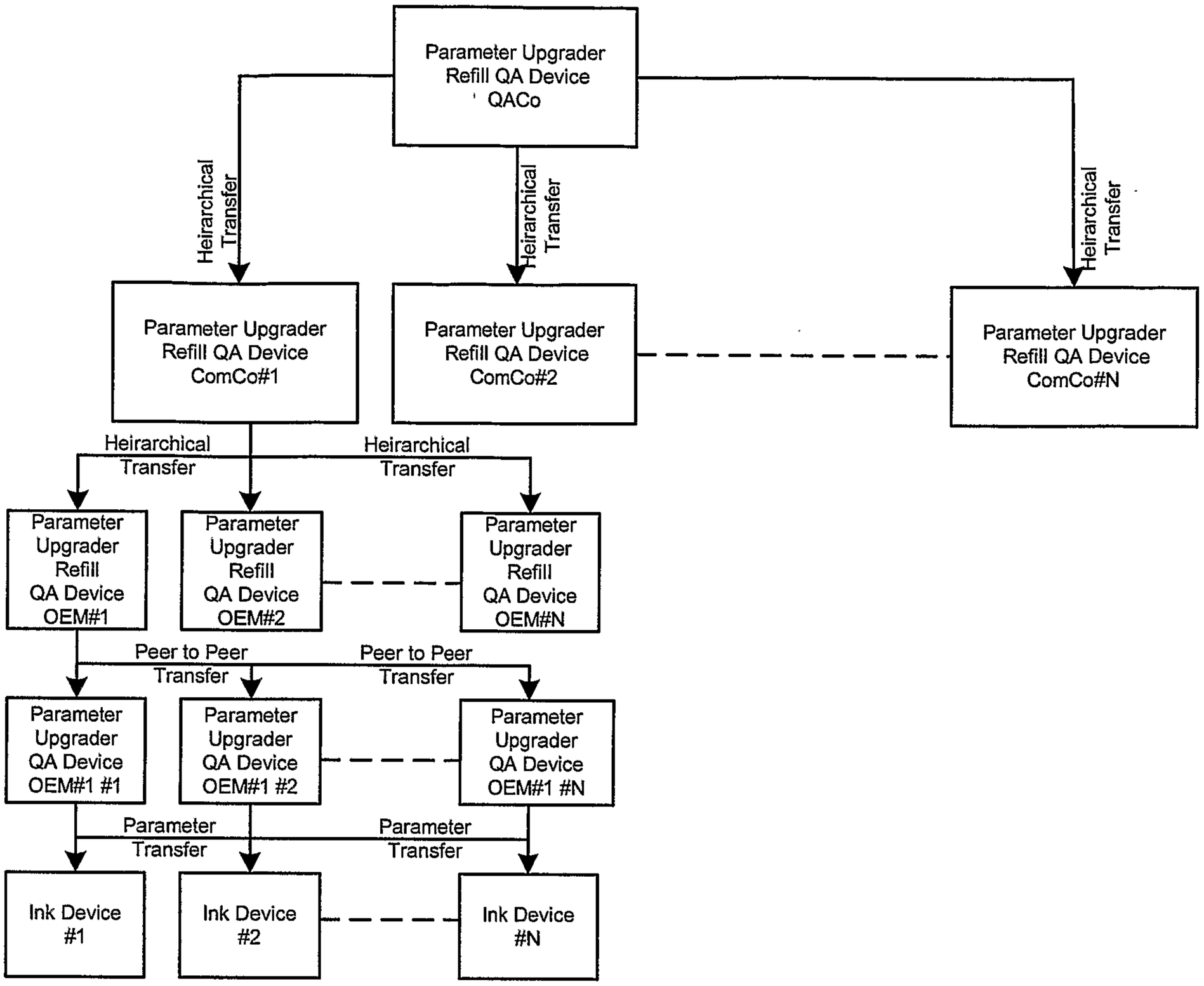


FIG. 384

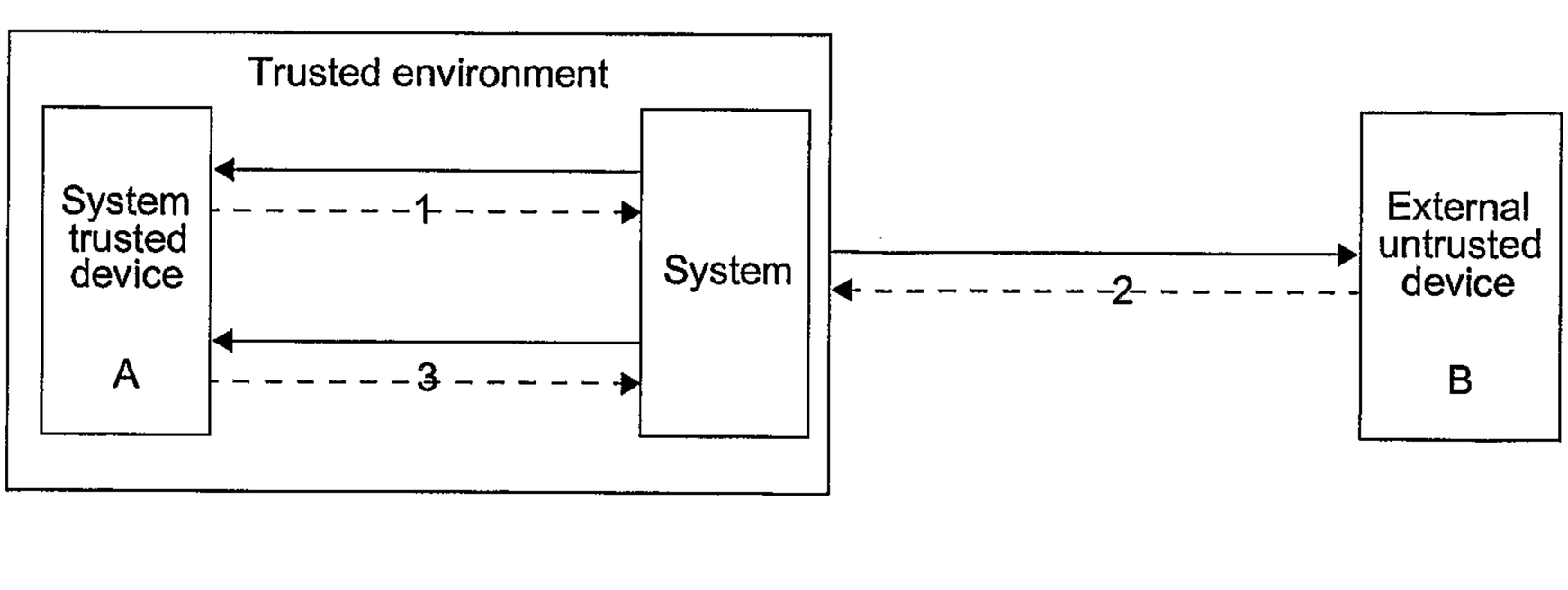


FIG. 385

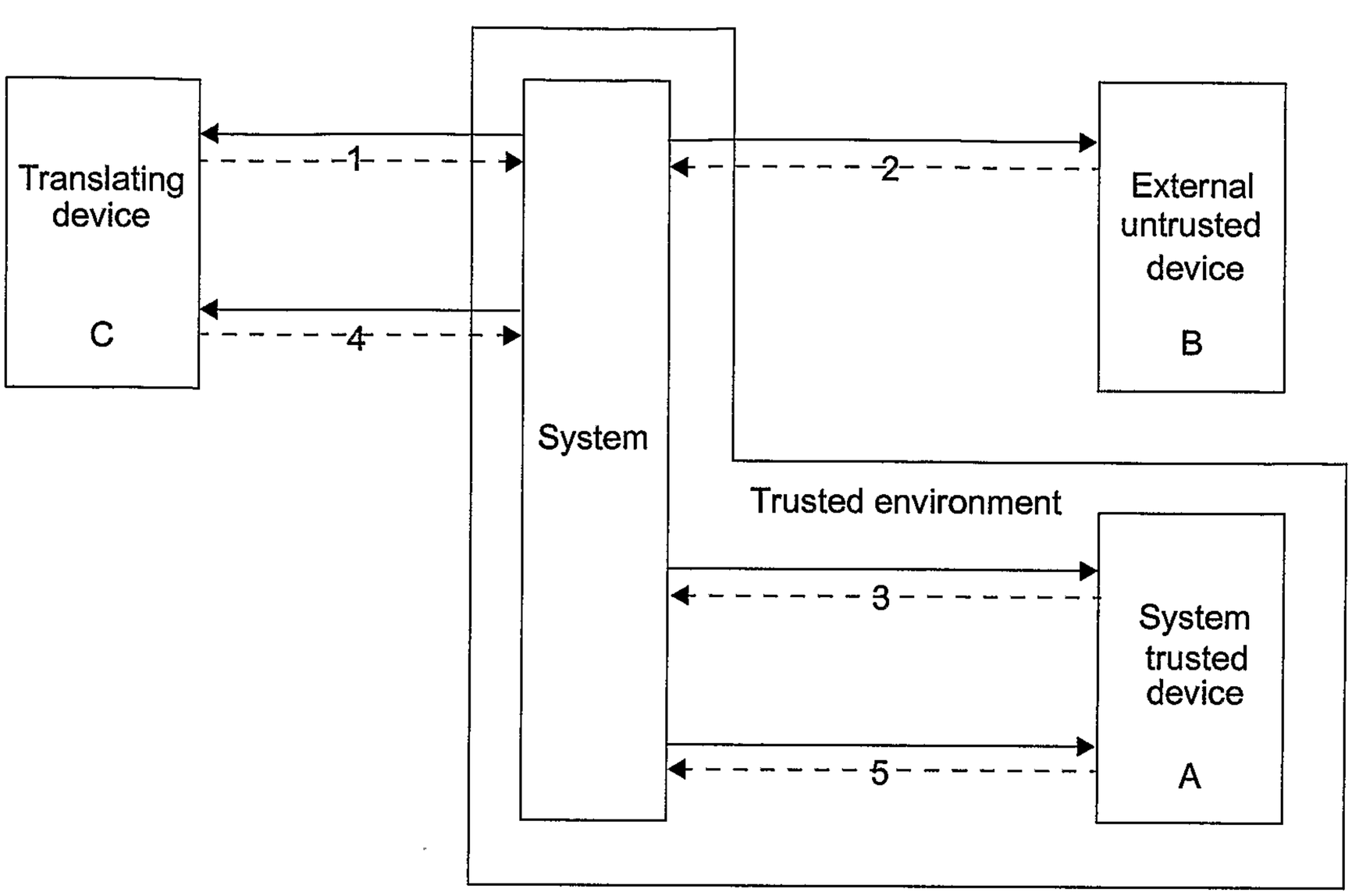

Translating device
C
1
4
System
2
External untrusted device
B
Trusted environment
3
5
System trusted device
A

FIG. 386

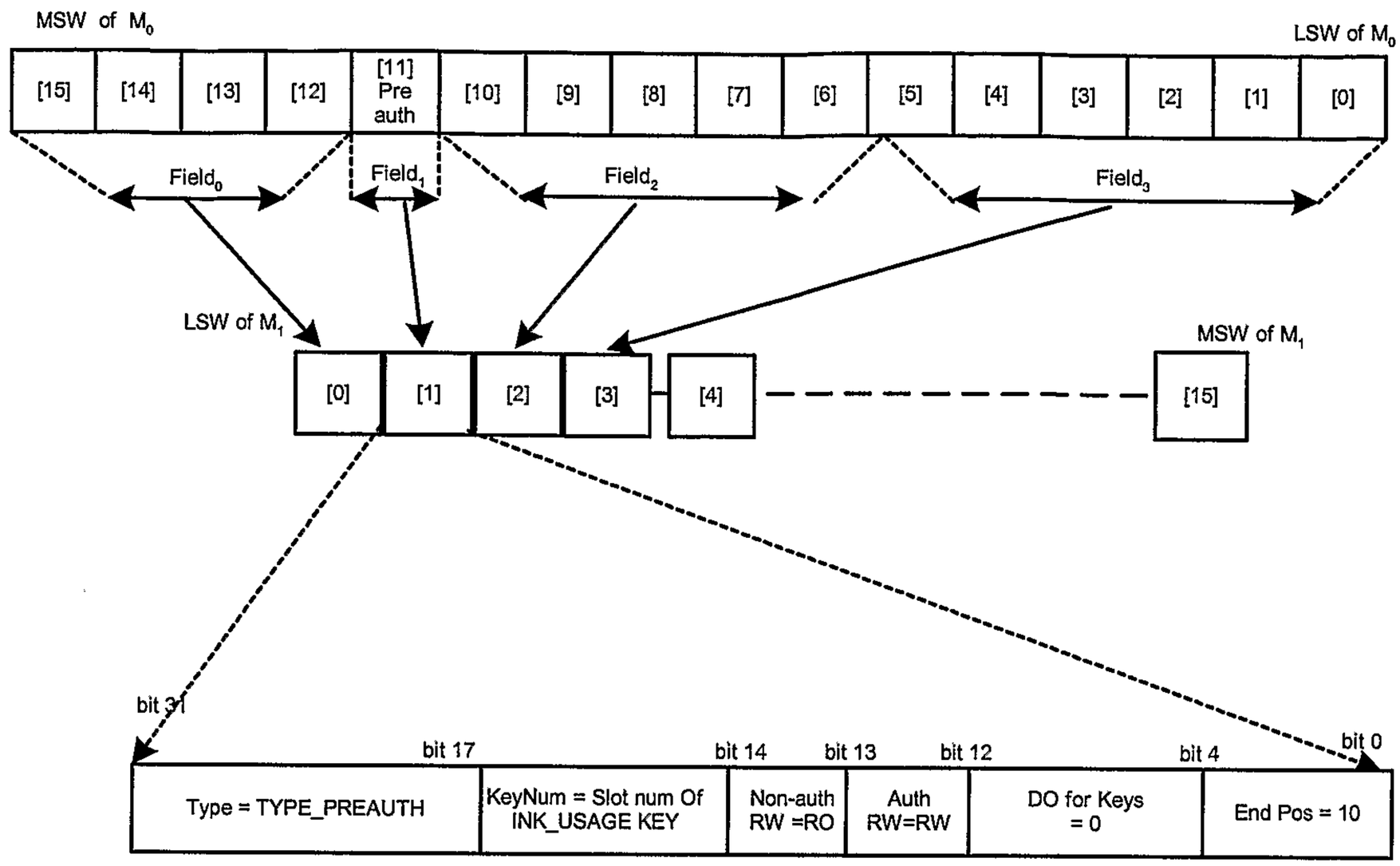


FIG. 387

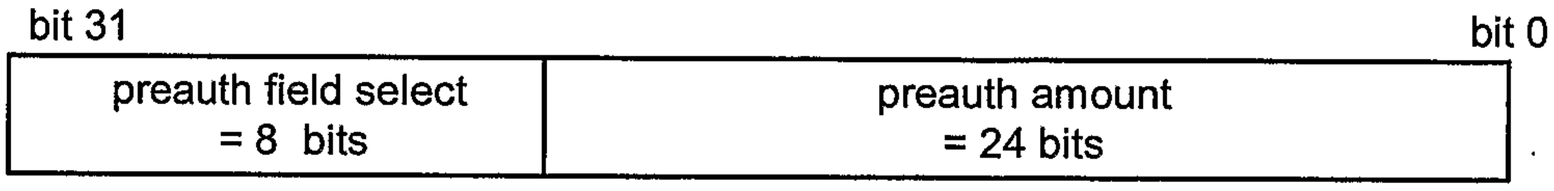


FIG. 388

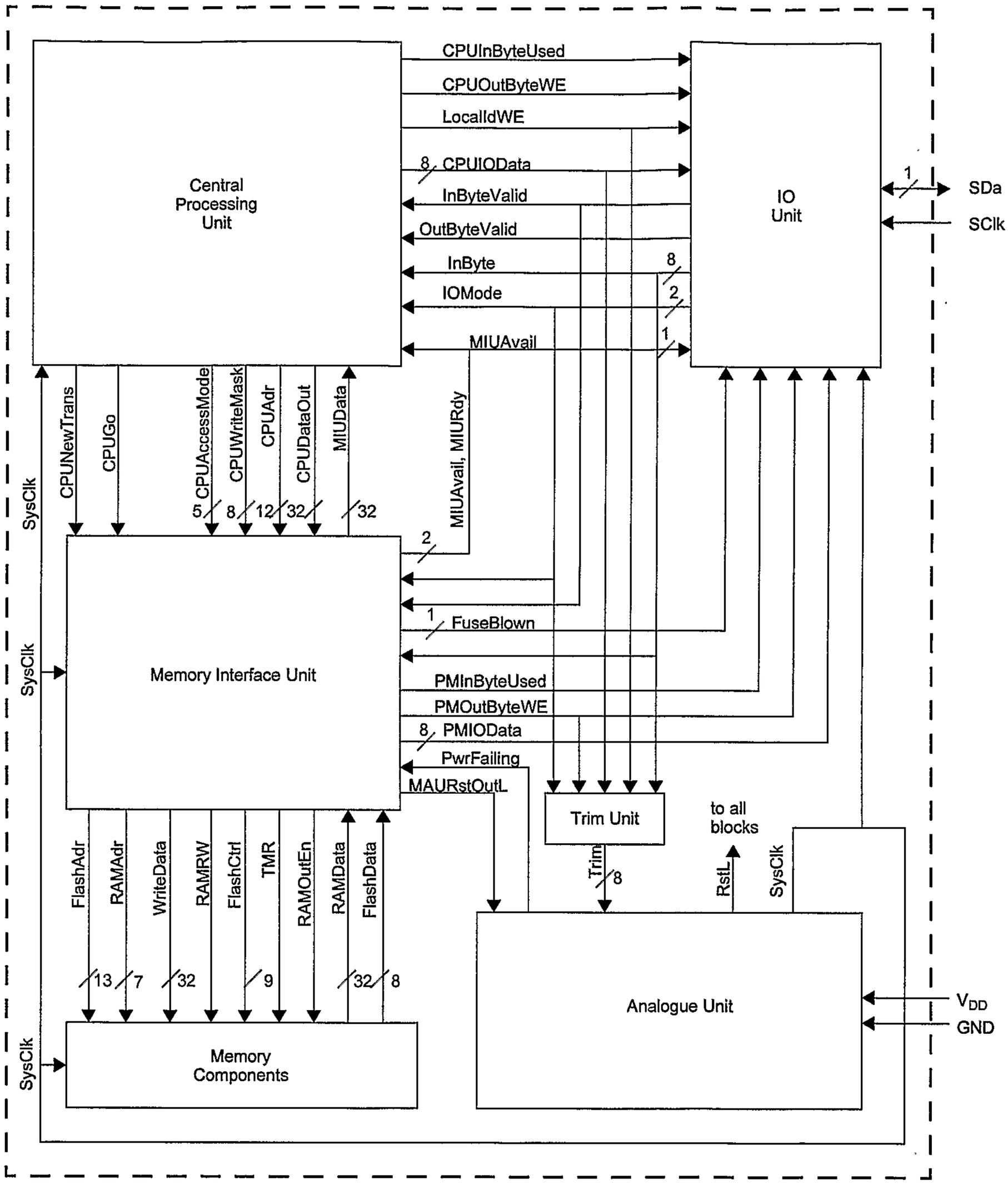

Central Processing Unit
IO Unit
Memory Interface Unit
Trim Unit
Analogue Unit
Memory Components
CPUInByteUsed
CPUOutByteWE
LocalIdWE
8 CPUIOData
InByteValid
OutByteValid
InByte
IOMode
MIUAvail
SDa
SClk
SysClk
CPUNewTrans
CPUGo
CPUAccessMode
CPUWriteMask
CPUAdr
CPUDataOut
MIUData
MIUAvail, MIURdy
5
8
12
32
32
2
1 FuseBlown
PMInByteUsed
PMOutByteWE
8 PMIOData
PwrFailing
MAURstOutL
Trim
8
to all blocks
RstL
FlashAdr
RAMAdr
WriteData
RAMRW
FlashCtrl
TMR
RAMOutEn
RAMData
FlashData
13
7
32
9
32
8
VDD
GND

FIG. 389

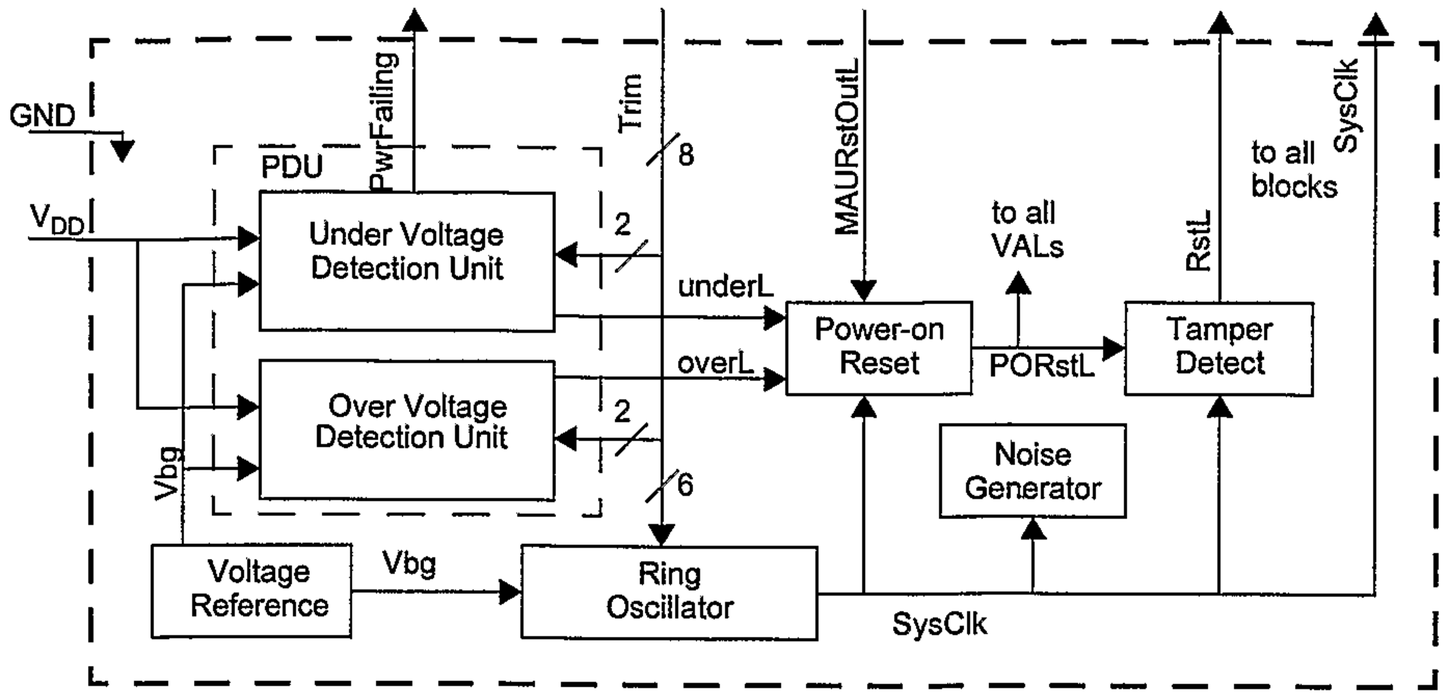


FIG. 390

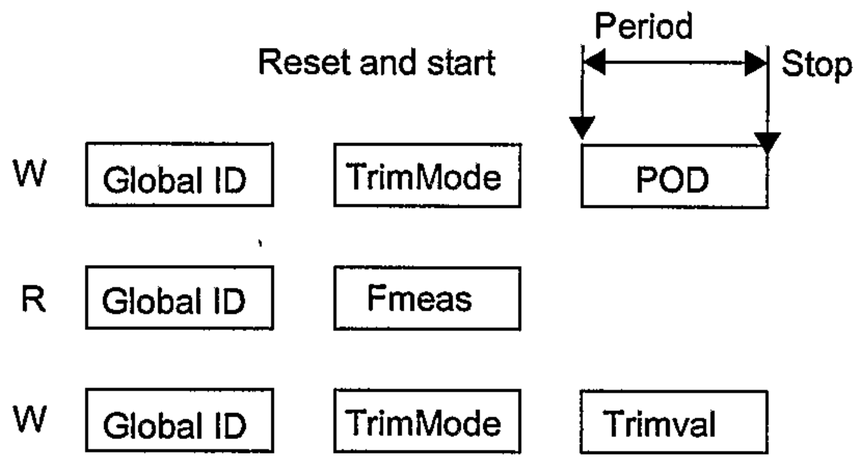


FIG. 391

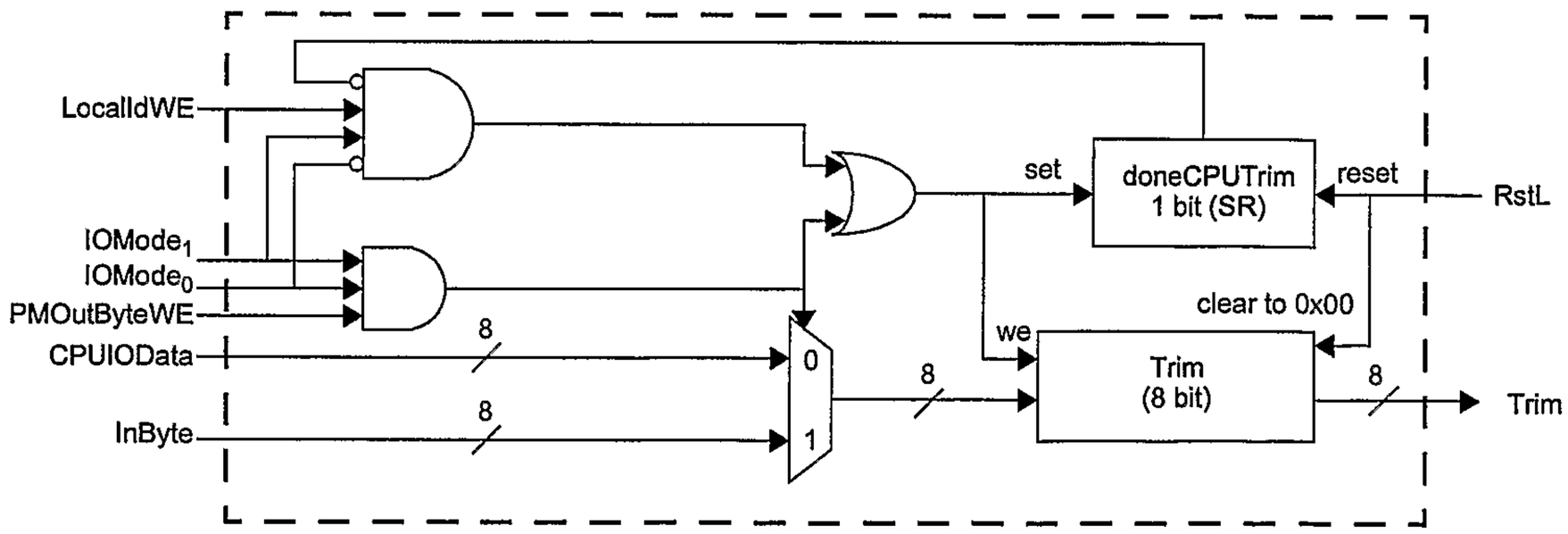


FIG. 392

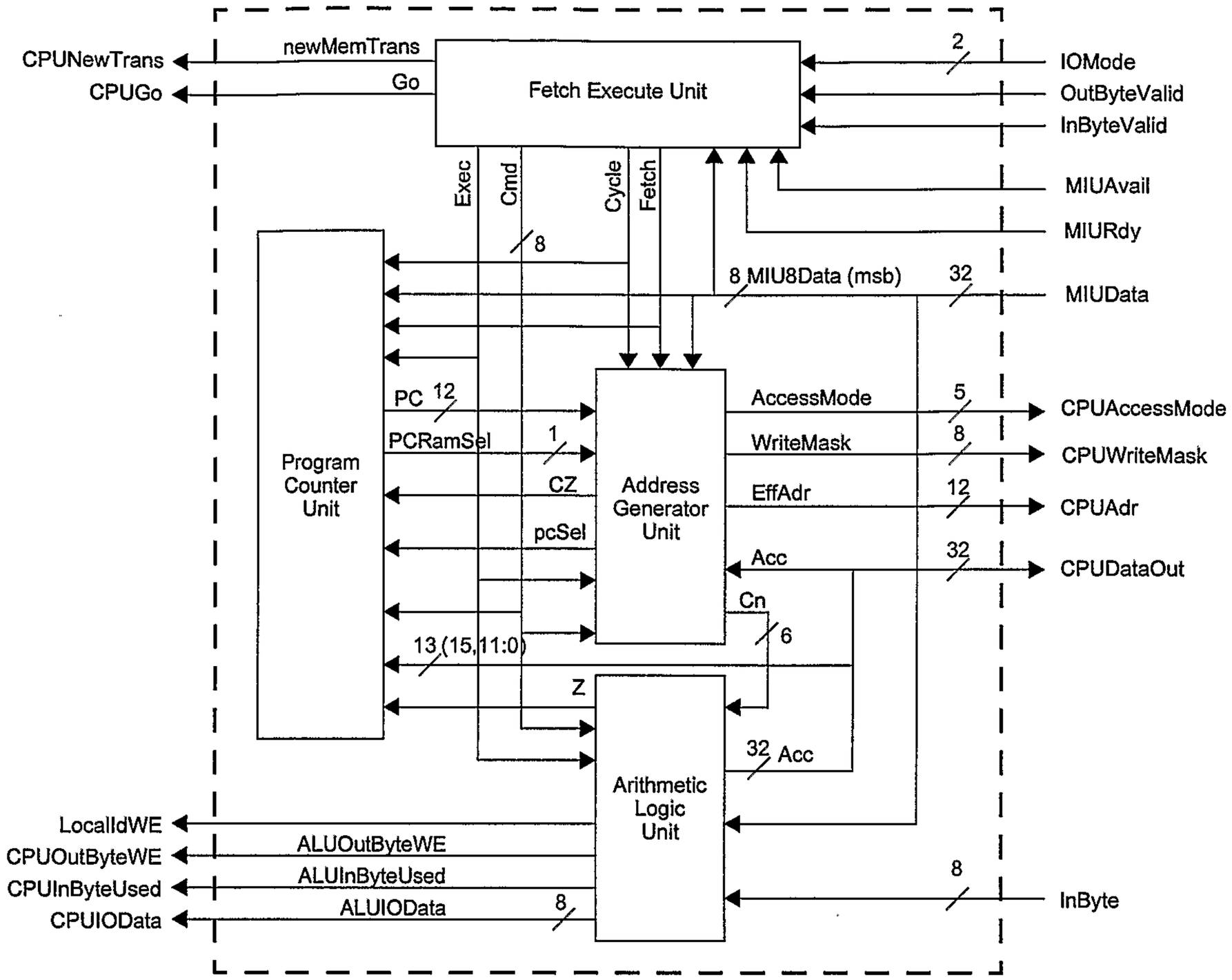

CPUNewTrans
CPUGo
newMemTrans
Go
Fetch Execute Unit
2
IOMode
OutByteValid
InByteValid
MIUAvail
MIURdy
Exec
Cmd
Cycle
Fetch
8
8 MIU8Data (msb)
32
MIUData
PC 12
PCRamSel
1
CZ
pcSel
Program Counter Unit
Address Generator Unit
AccessMode
5
CPUAccessMode
WriteMask
8
CPUWriteMask
EffAdr
12
CPUAdr
Acc
32
CPUDataOut
Cn
6
13 (15,11:0)
Z
32 Acc
Arithmetic Logic Unit
LocalIdWE
CPUOutByteWE
ALUOutByteWE
CPUInByteUsed
ALUInByteUsed
CPUIOData
ALUIOData
8
InByte

FIG. 393

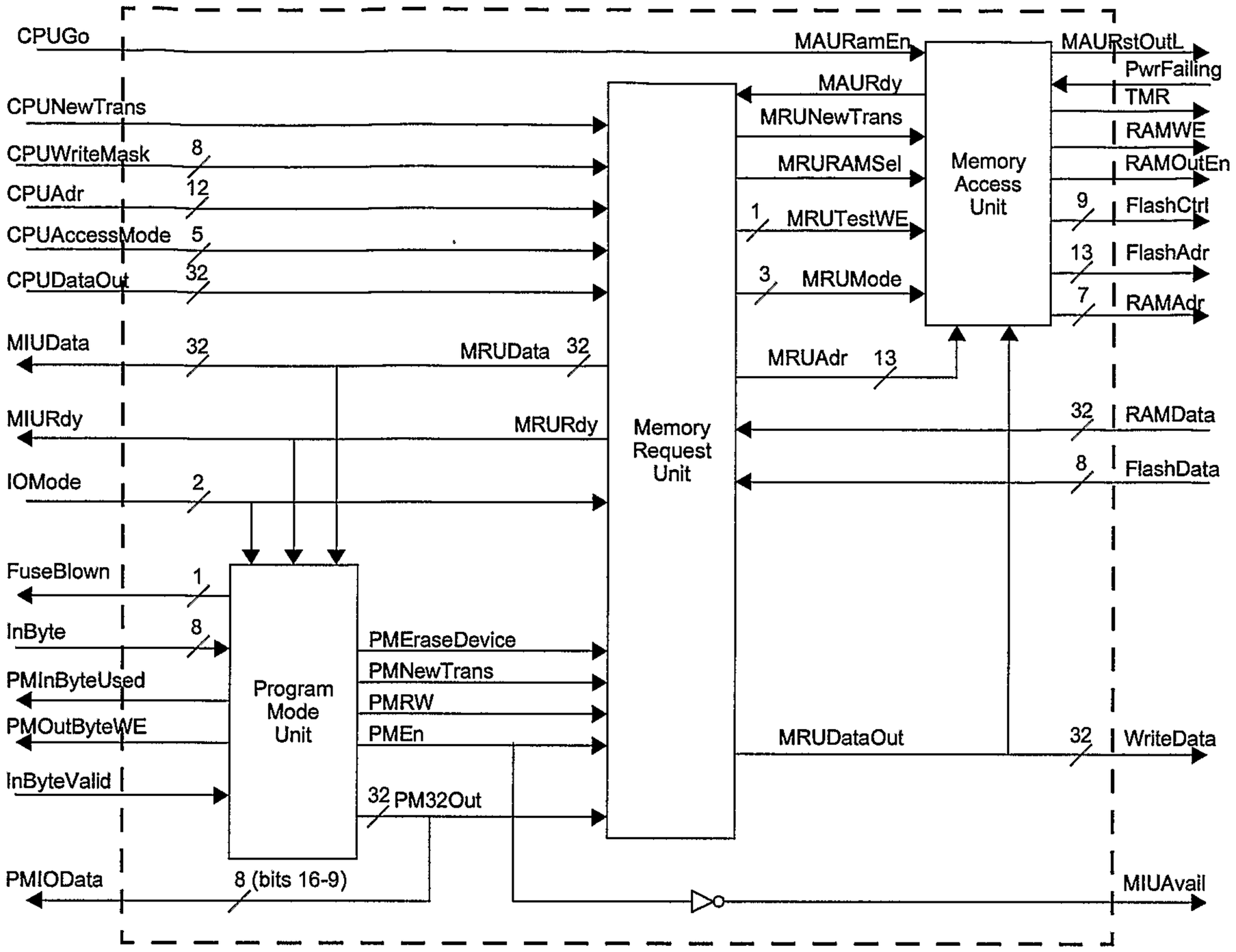


FIG. 394

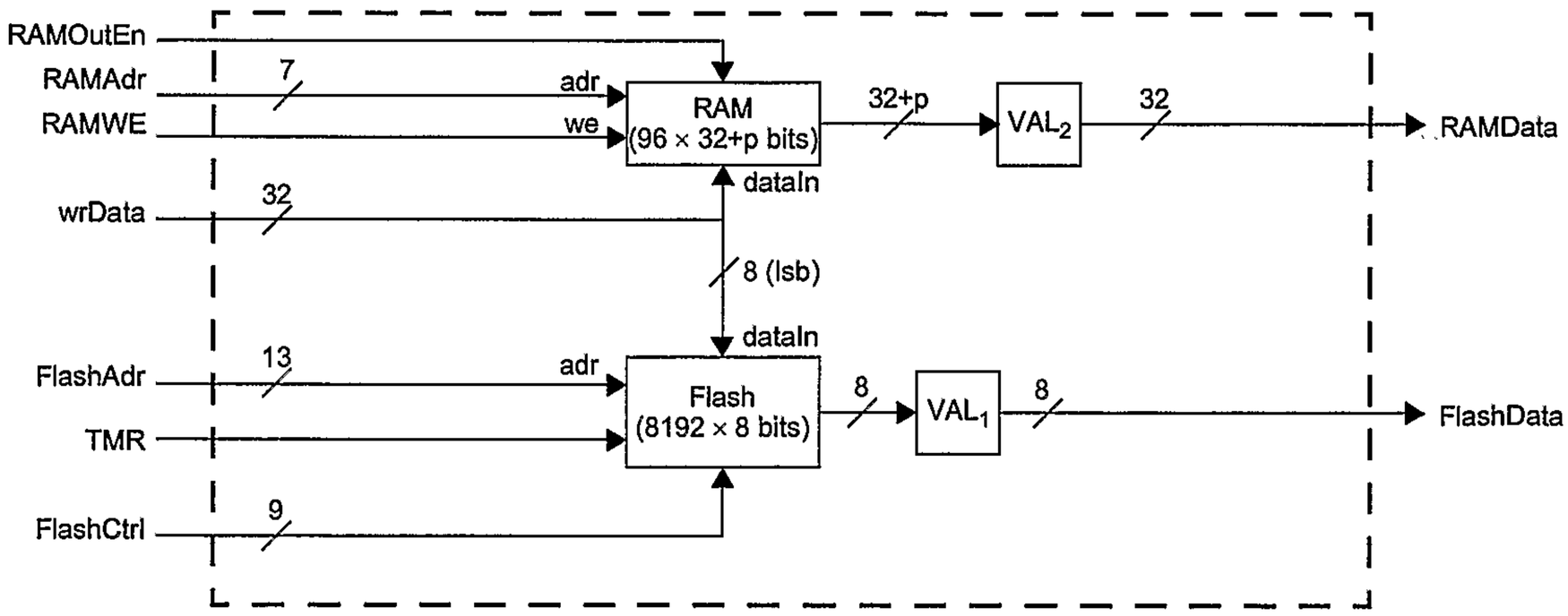


FIG. 395

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PrilD6	PrilD5	PrilD4	PrilD3	PrilD2	PrilD1	PrilD0	R/*W 0 = write 1 = read

FIG. 396

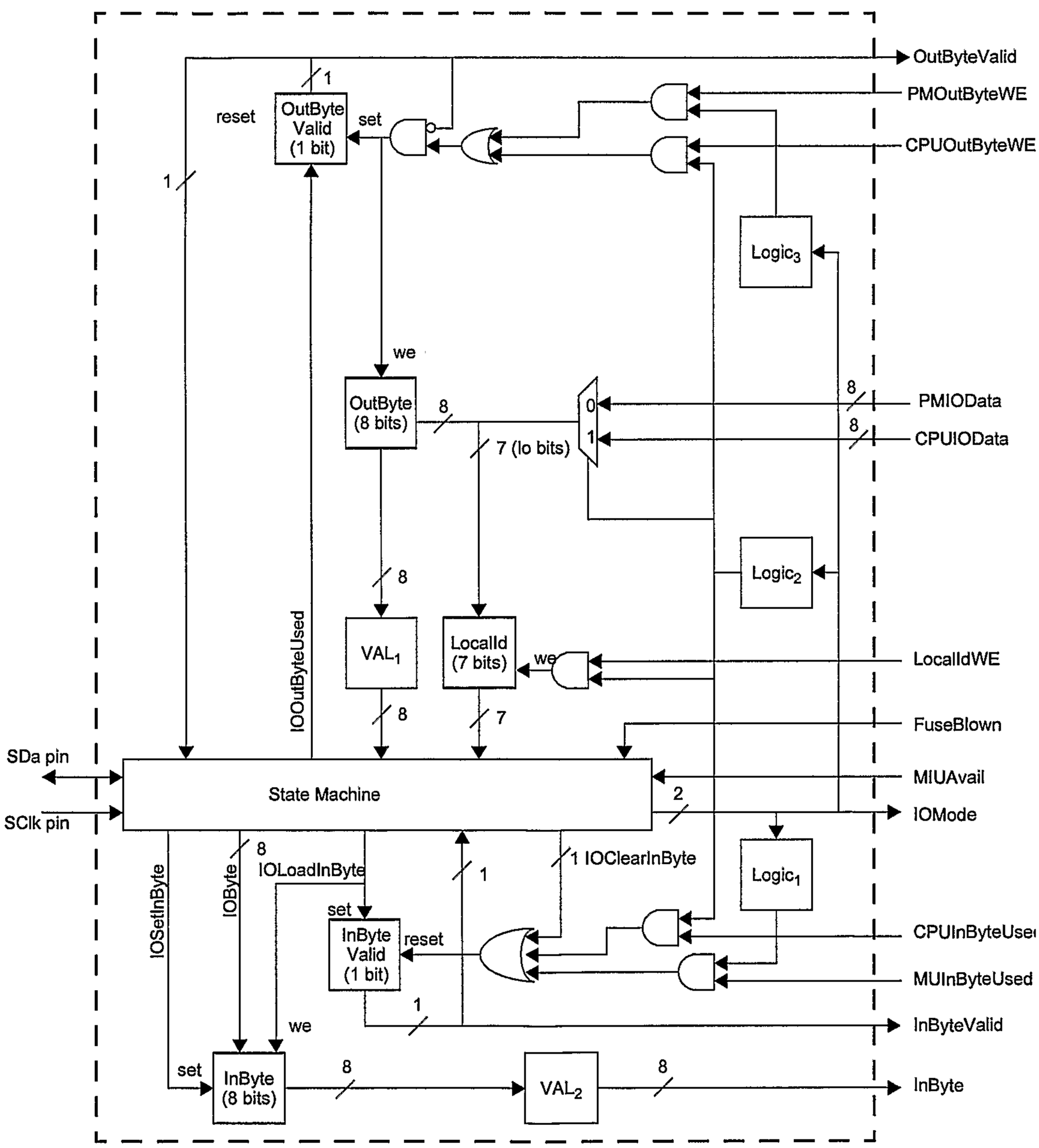


FIG. 397

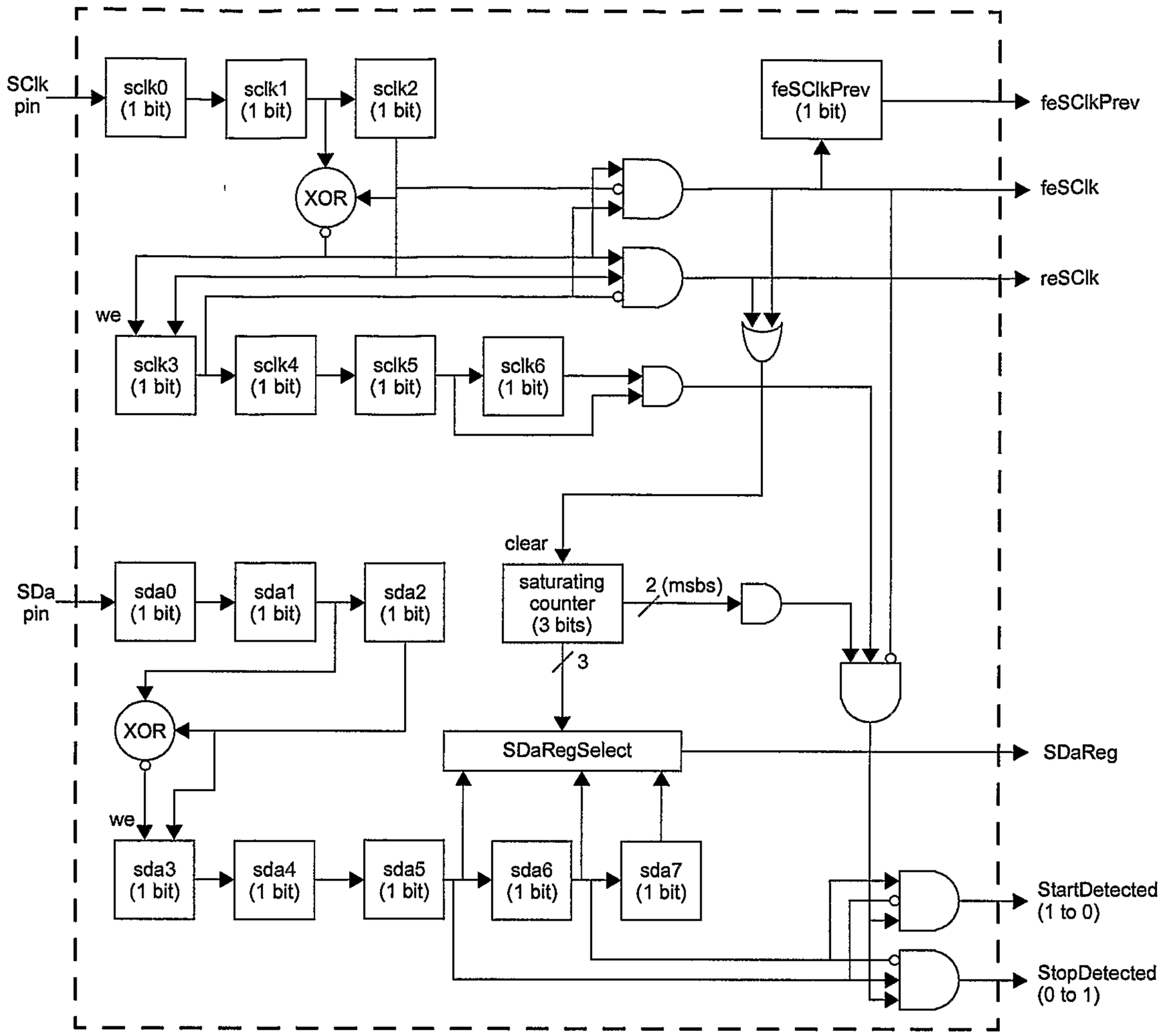

SClk pin
sclk0 (1 bit)
sclk1 (1 bit)
sclk2 (1 bit)
feSClkPrev (1 bit)
feSClkPrev
XOR
feSClk
reSClk
we
sclk3 (1 bit)
sclk4 (1 bit)
sclk5 (1 bit)
sclk6 (1 bit)
clear
SDa pin
sda0 (1 bit)
sda1 (1 bit)
sda2 (1 bit)
saturating counter (3 bits)
2 (msbs)
3
XOR
SDaRegSelect
SDaReg
we
sda3 (1 bit)
sda4 (1 bit)
sda5 (1 bit)
sda6 (1 bit)
sda7 (1 bit)
StartDetected (1 to 0)
StopDetected (0 to 1)

FIG. 398

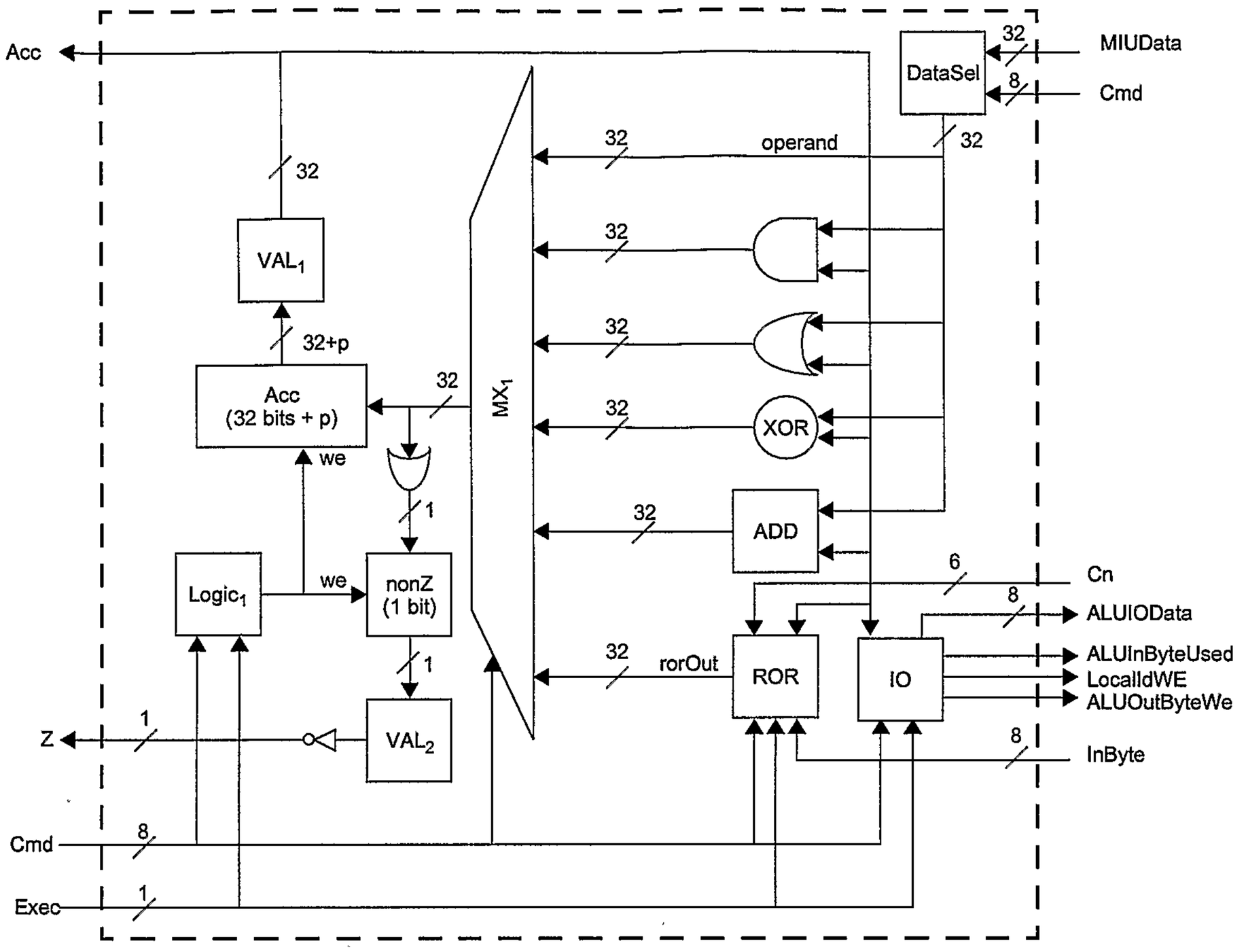


FIG. 399

FIG. 400

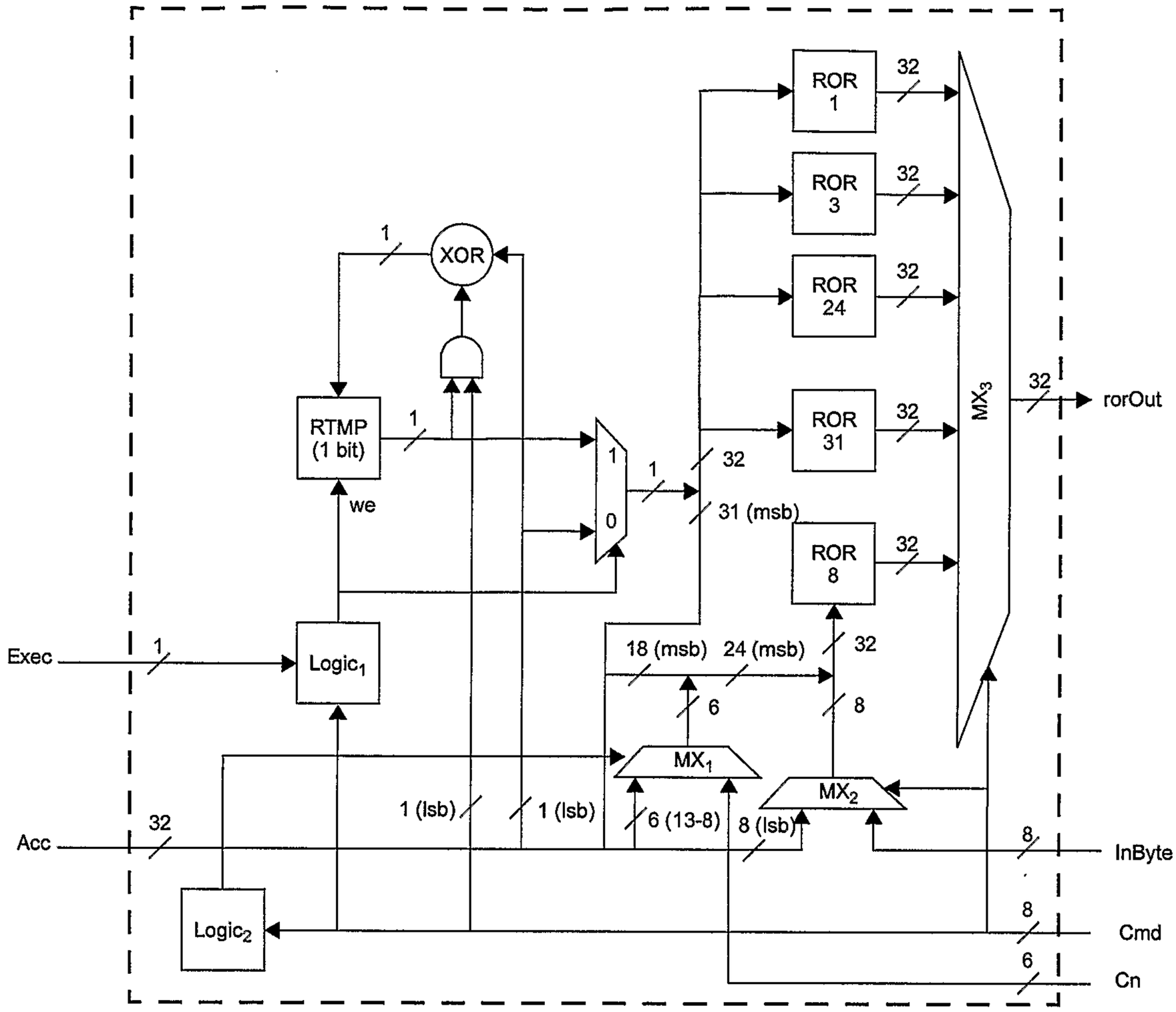

ROR 1
ROR 3
ROR 24
ROR 31
ROR 8
32
XOR
RTMP (1 bit)
we
1
0
31 (msb)
MX3
rorOut
Exec
Logic1
18 (msb)
24 (msb)
6
8
MX1
MX2
1 (lsb)
6 (13-8)
8 (lsb)
Acc
InByte
Logic2
Cmd
Cn

FIG. 401

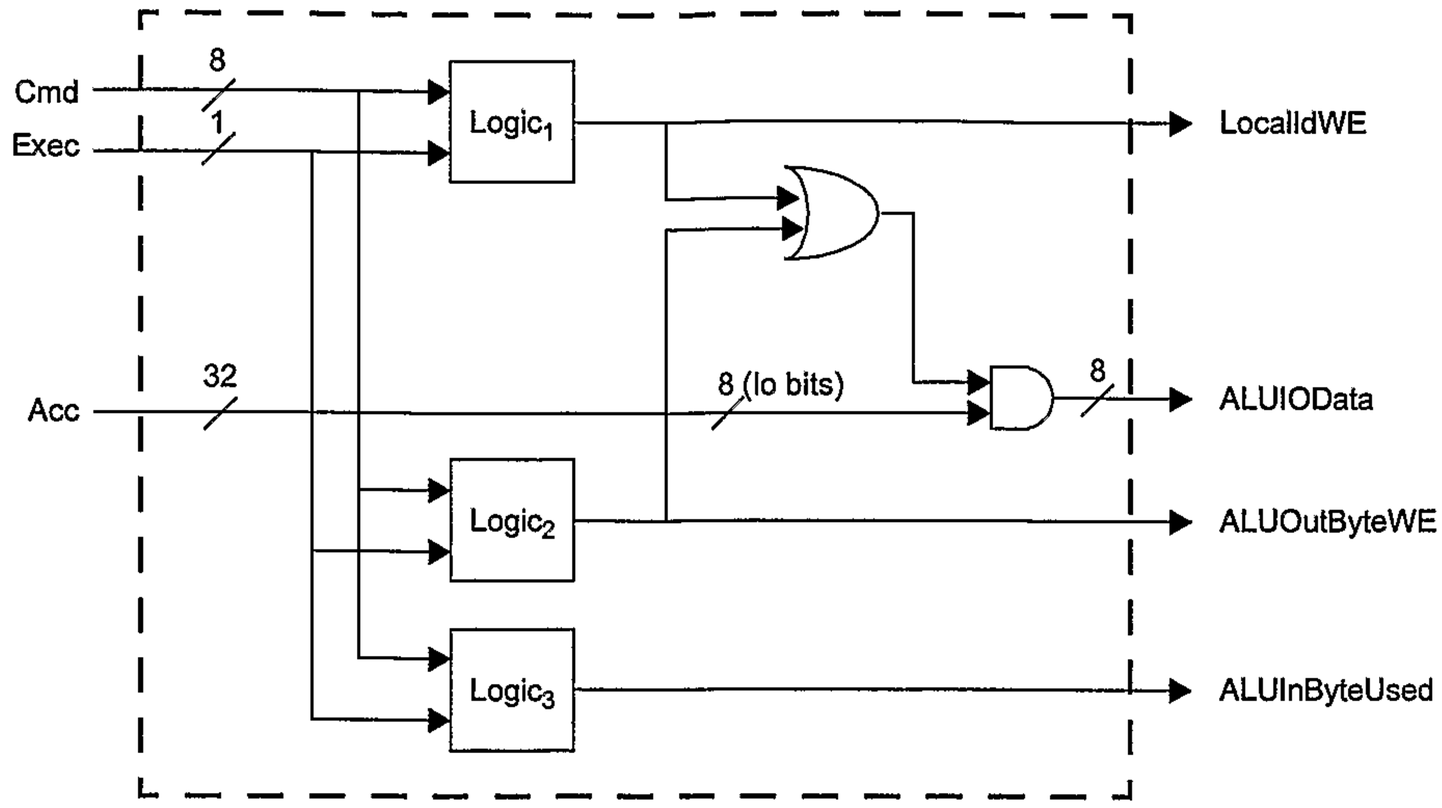

Cmd
Exec
8
1
Logic1
LocalIdWE
Acc
32
8 (lo bits)
8
ALUIOData
Logic2
ALUOutByteWE
Logic3
ALUInByteUsed

FIG. 402

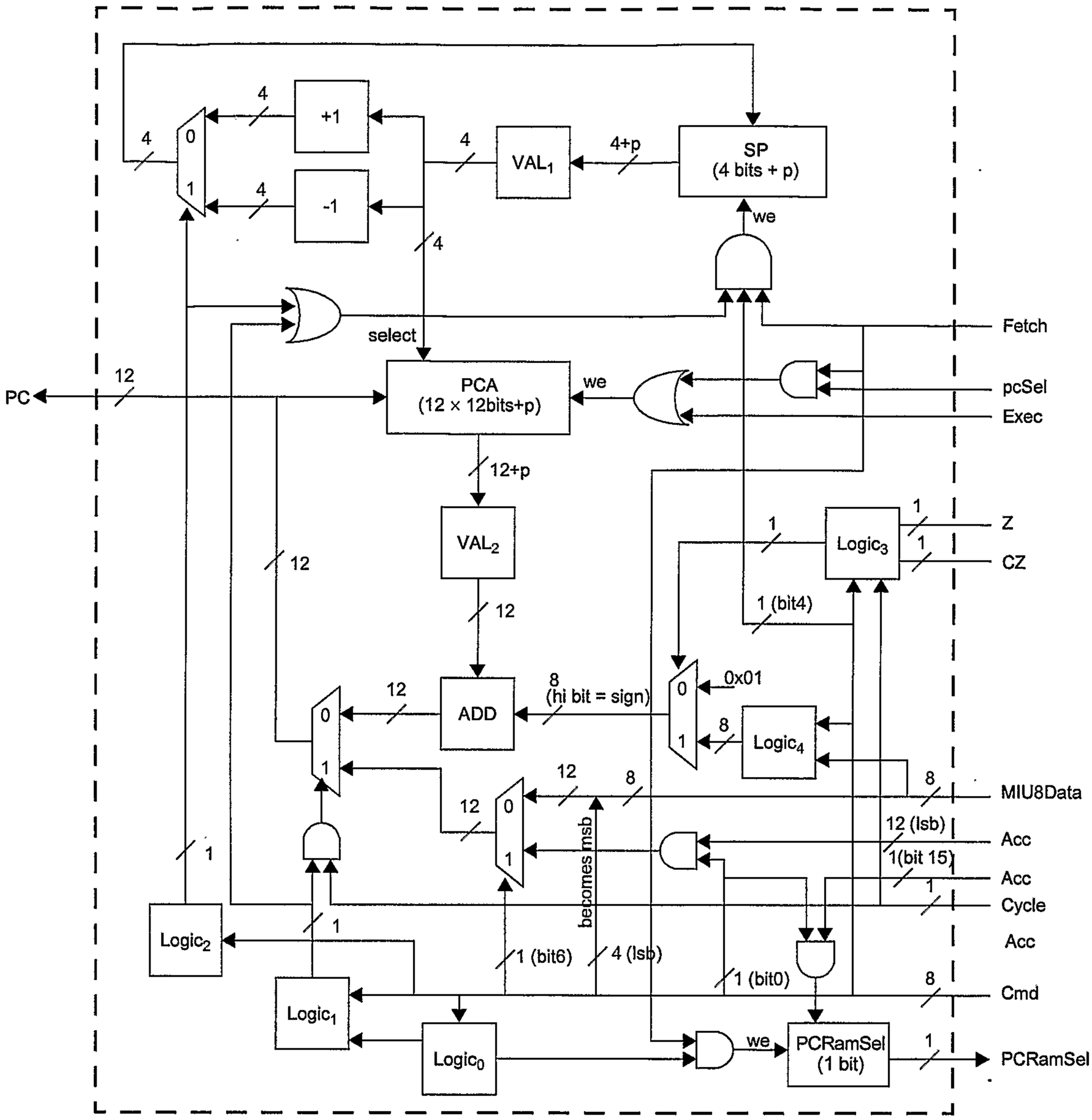

+1
-1
0
1
4
4+p
VAL1
SP
(4 bits + p)
we
Fetch
pcSel
Exec
select
PC
12
PCA
(12 × 12bits+p)
12+p
VAL2
ADD
8
(hi bit = sign)
0x01
Logic3
Logic4
Logic2
Logic1
Logic0
Z
CZ
1 (bit4)
MIU8Data
12 (lsb)
Acc
1(bit 15)
Acc
Cycle
Acc
Cmd
1 (bit6)
4 (lsb)
1 (bit0)
becomes msb
PCRamSel
(1 bit)
PCRamSel

FIG. 403

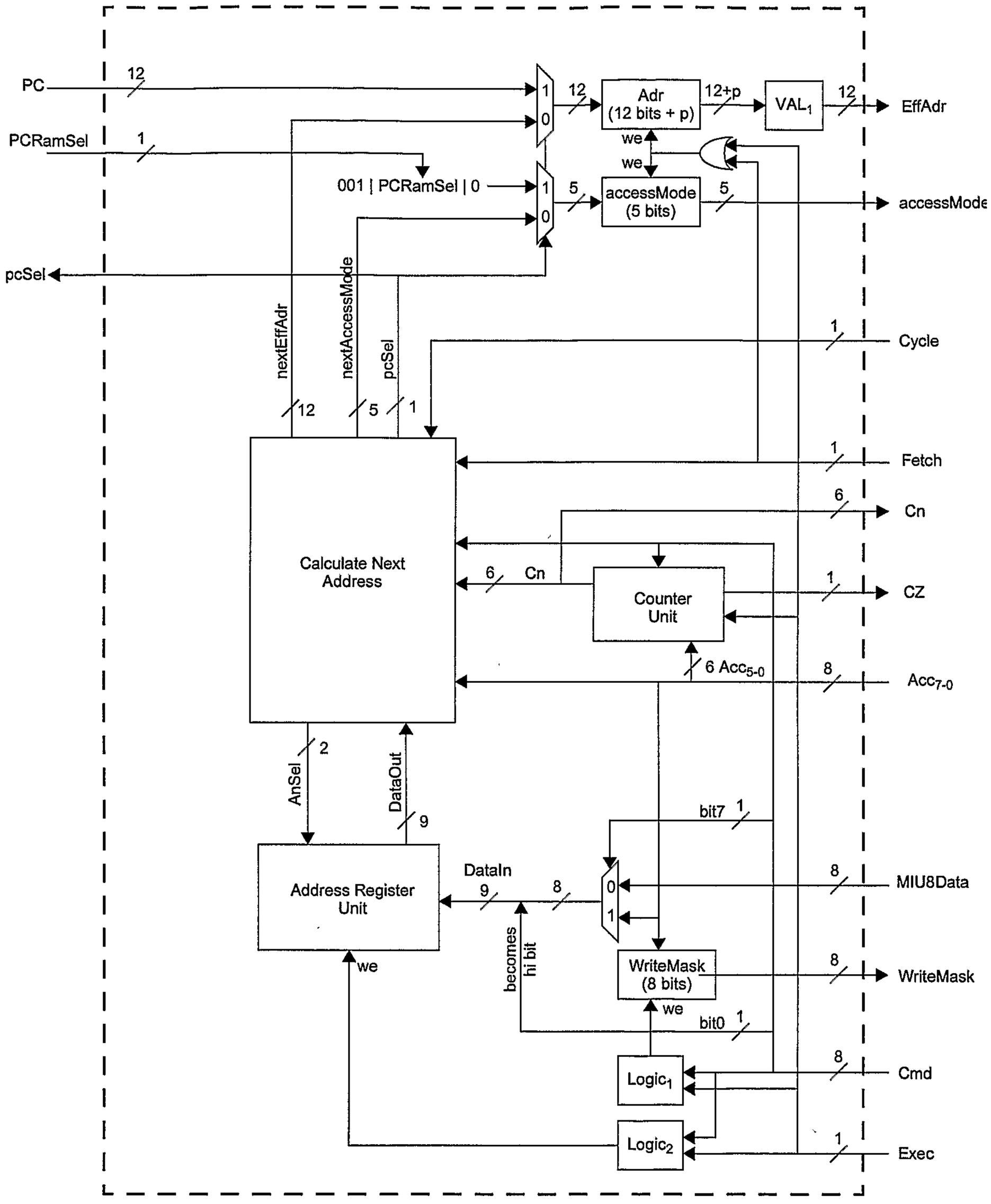

PC
12
PCRamSel
1
001 | PCRamSel | 0
pcSel
nextEffAdr
nextAccessMode
pcSel
12
5
1
Calculate Next Address
2
AnSel
DataOut
9
Address Register Unit
DataIn
9
8
we
becomes hi bit
1
0
12
Adr (12 bits + p)
12+p
VAL1
12
EffAdr
we
we
1
0
5
accessMode (5 bits)
5
accessMode
6
Cn
6
Cn
Counter Unit
1
CZ
6 Acc5-0
8
Acc7-0
1
Cycle
1
Fetch
6
Cn
bit7
1
8
MIU8Data
WriteMask (8 bits)
8
WriteMask
we
bit0
1
Logic1
8
Cmd
Logic2
1
Exec

FIG. 404

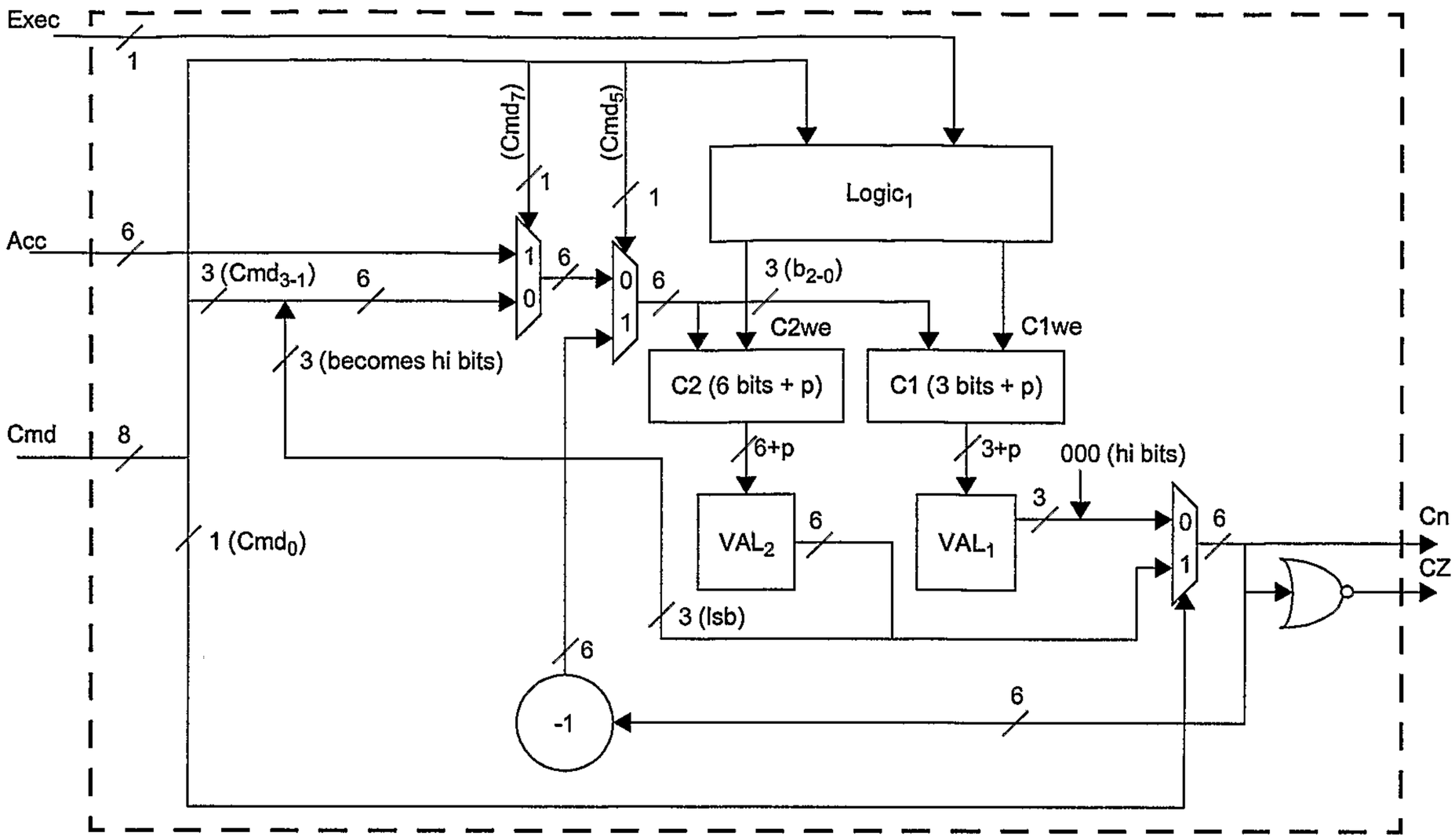


FIG. 405

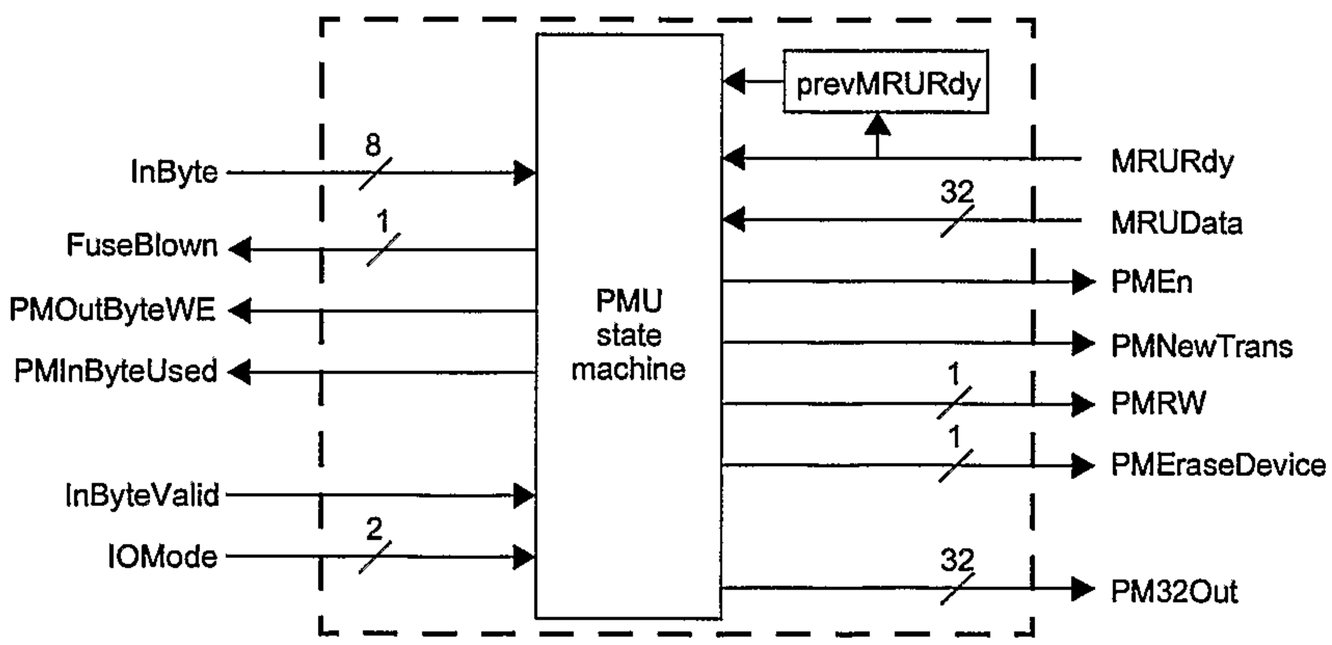


FIG. 406

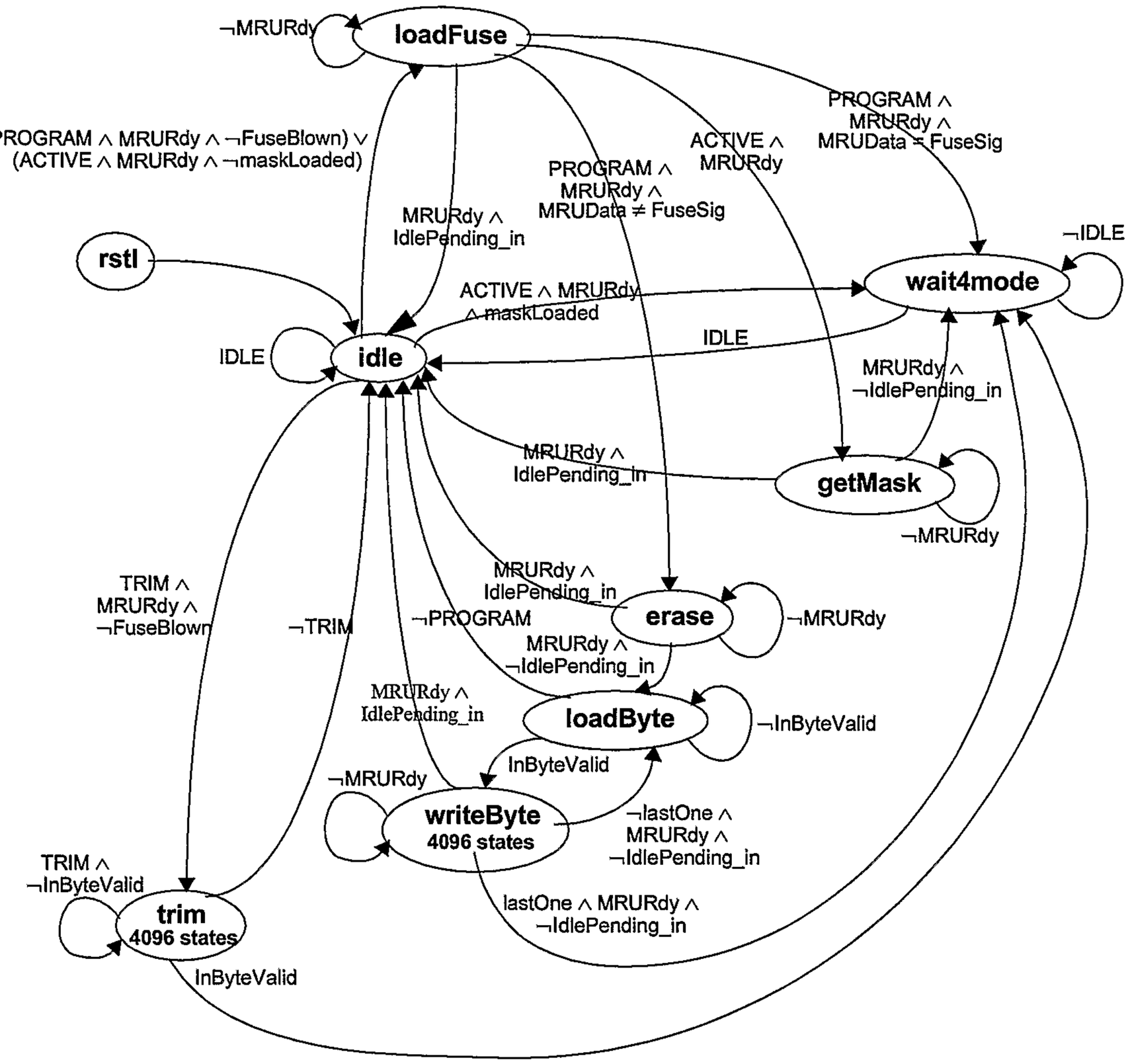

¬MRURdy
loadFuse
(PROGRAM ∧ MRURdy ∧ ¬FuseBlown) ∨
(ACTIVE ∧ MRURdy ∧ ¬maskLoaded)
PROGRAM ∧
MRURdy ∧
MRUData ≠ FuseSig
ACTIVE ∧
MRURdy
PROGRAM ∧
MRURdy ∧
MRUData = FuseSig
MRURdy ∧
IdlePending_in
rstl
¬IDLE
wait4mode
ACTIVE ∧ MRURdy
∧ maskLoaded
IDLE
IDLE
idle
MRURdy ∧
¬IdlePending_in
MRURdy ∧
IdlePending_in
getMask
¬MRURdy
TRIM ∧
MRURdy ∧
¬FuseBlown
¬TRIM
MRURdy ∧
IdlePending_in
¬PROGRAM
erase
¬MRURdy
MRURdy ∧
¬IdlePending_in
MRURdy ∧
IdlePending_in
loadByte
¬InByteValid
InByteValid
¬MRURdy
writeByte
4096 states
¬lastOne ∧
MRURdy ∧
¬IdlePending_in
TRIM ∧
¬InByteValid
trim
4096 states
lastOne ∧ MRURdy ∧
¬IdlePending_in
InByteValid

FIG. 407

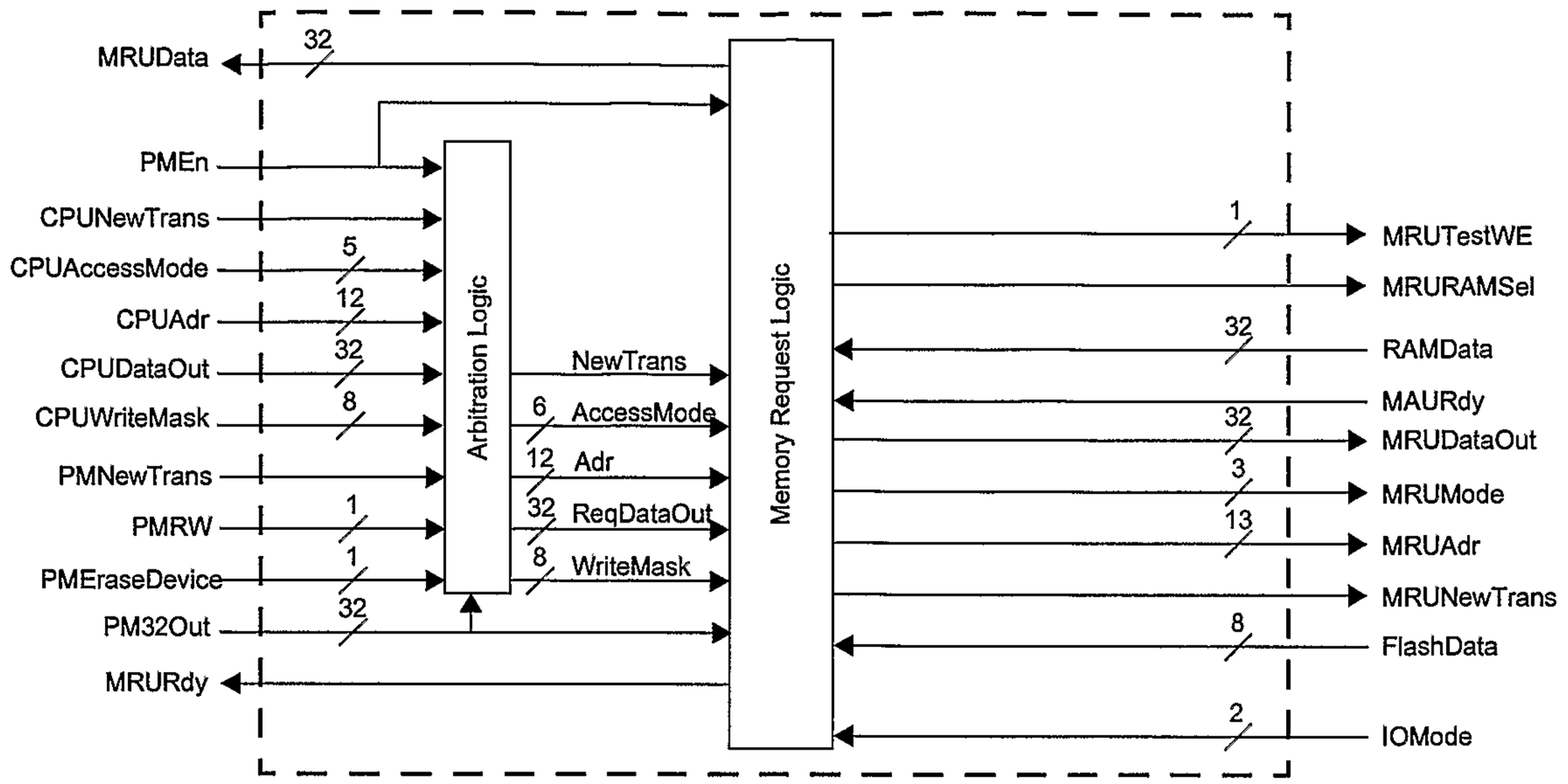

MRUData
32
PMEn
CPUNewTrans
CPUAccessMode
5
CPUAdr
12
CPUDataOut
32
CPUWriteMask
8
PMNewTrans
PMRW
1
PMEraseDevice
1
PM32Out
32
MRURdy
Arbitration Logic
NewTrans
6
AccessMode
12
Adr
32
ReqDataOut
8
WriteMask
Memory Request Logic
1
MRUTestWE
MRURAMSel
32
RAMData
MAURdy
32
MRUDataOut
3
MRUMode
13
MRUAdr
MRUNewTrans
8
FlashData
2
IOMode

FIG. 408

330/331

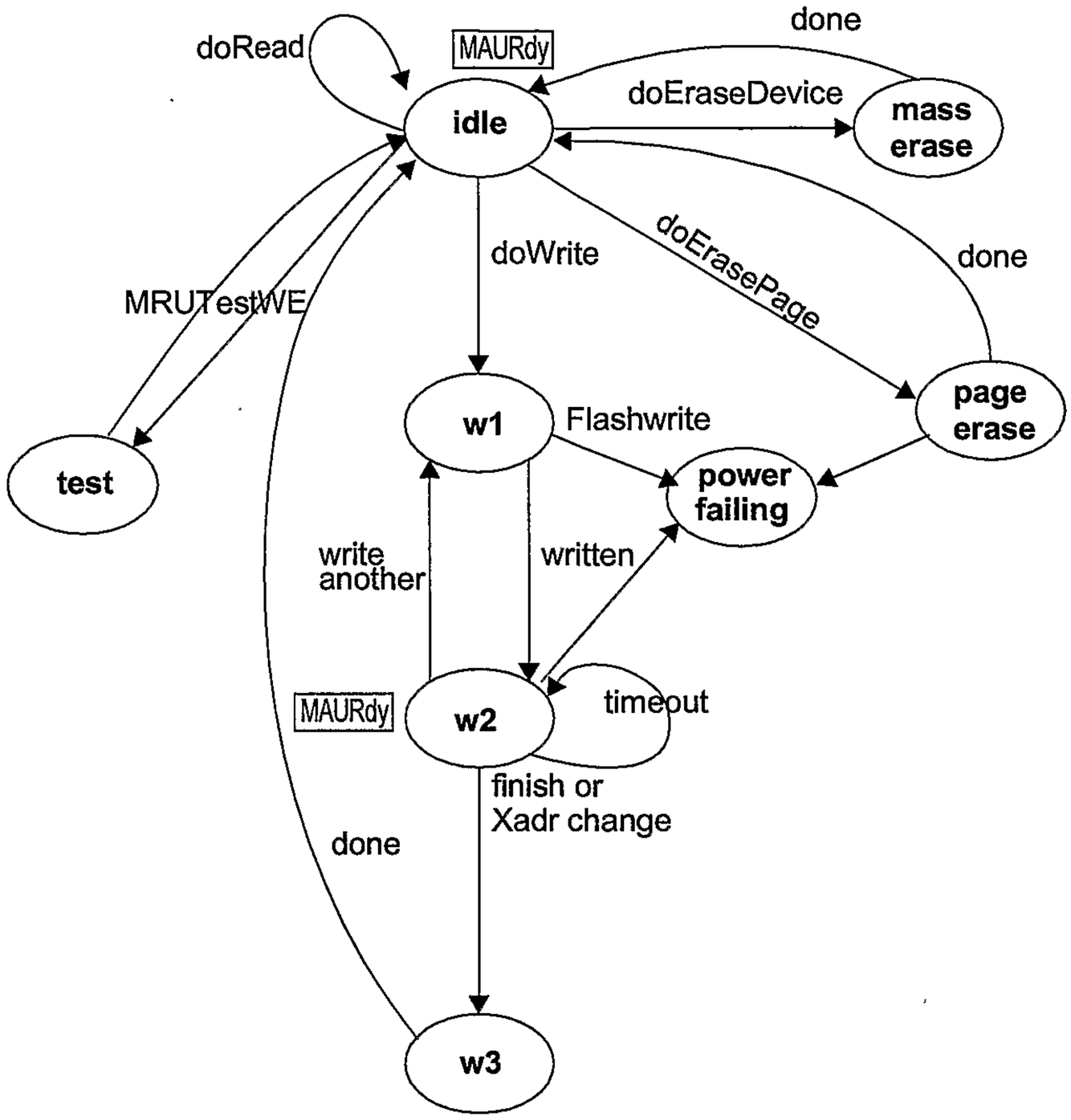


FIG. 409

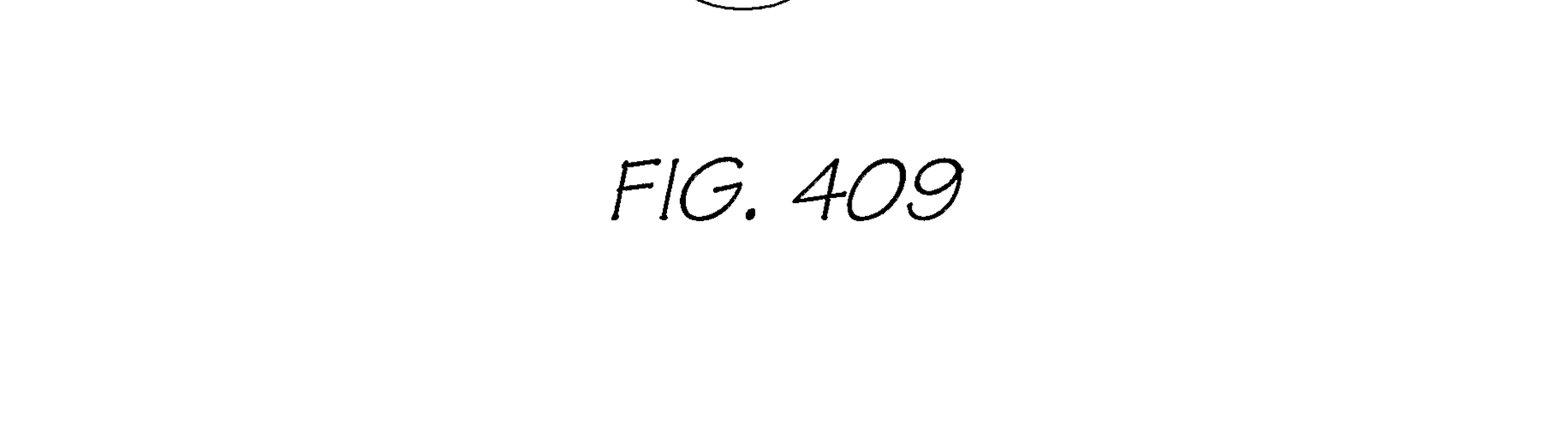


FIG. 410

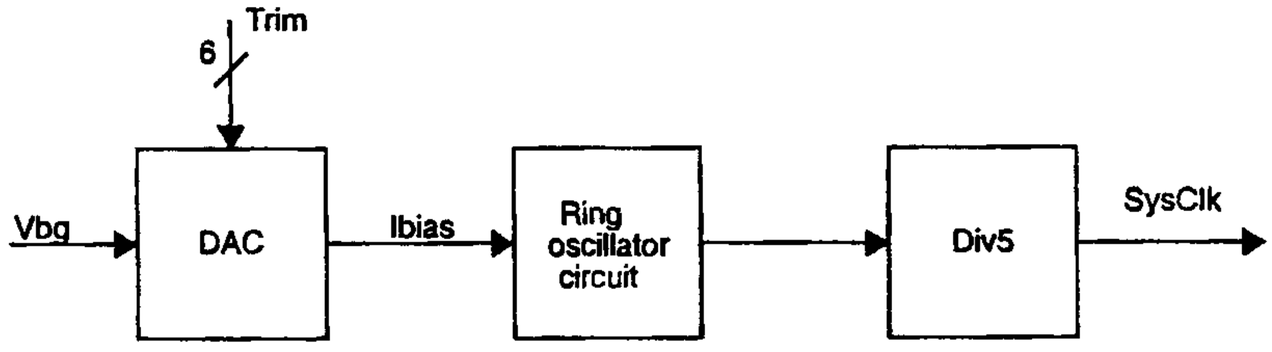


FIG. 411

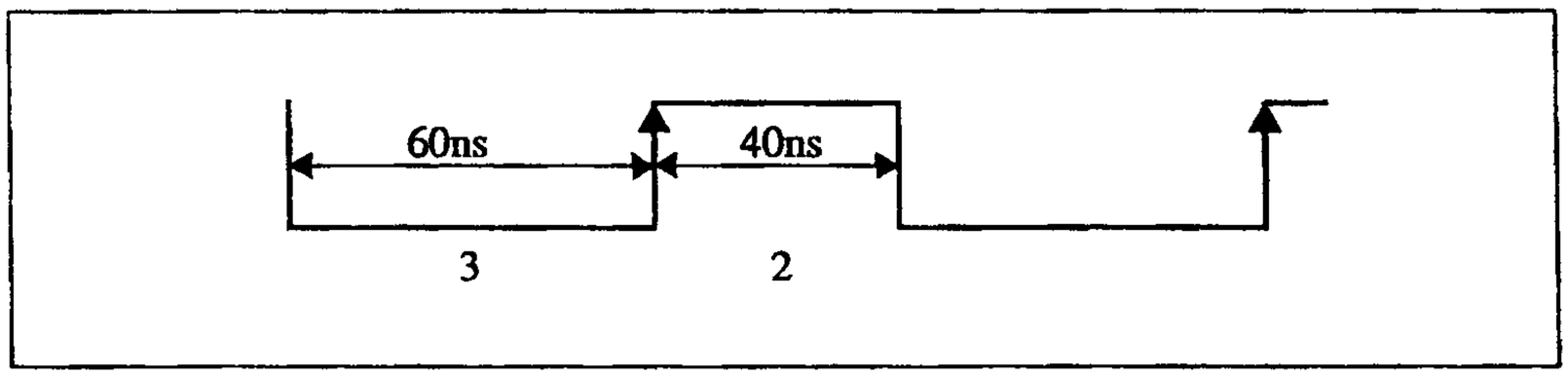


FIG. 412