US 20020061107A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2002/0061107 A1

Tham et al. (43) Pub. Date: May 23, 2002

(54) METHODS AND APPARATUS FOR IMPLEMENTING A CRYPTOGRAPHY ENGINE

(76) Inventors: **Terry K. Tham**, San Jose, CA (US); **Errol Lai**, Mountain View, CA (US)

Correspondence Address:
BEYER WEAVER & THOMAS LLP
P.O. BOX 778
BERKELEY, CA 94704-0778 (US)

(21) Appl. No.: **09/948,203**

(22) Filed: **Sep. 6, 2001**

Related U.S. Application Data

(63) Non-provisional of provisional application No. 60/235,190, filed on Sep. 25, 2000.

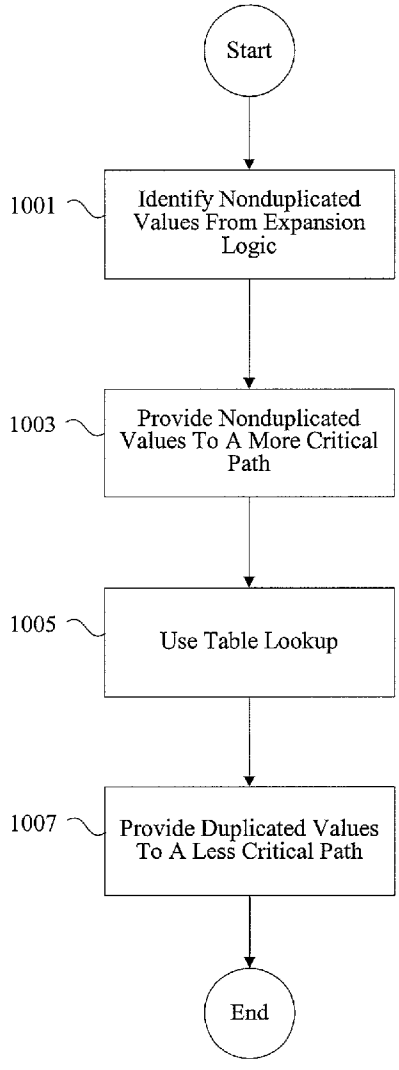
Publication Classification

(51) **Int. Cl.**[7] **H04L 9/00**

(52) **U.S. Cl.** **380/259**

(57) **ABSTRACT**

Methods and apparatus are provided for implementing a cryptography engine for cryptography processing. A variety of techniques are described. A cryptography engine such as a DES engine running at a clock frequency higher than that of surrounding logic can be synchronized with the surrounding logic using a frequency synchronizer. Sbox logic output can be more efficiently determined by intelligently arranging Sbox input.



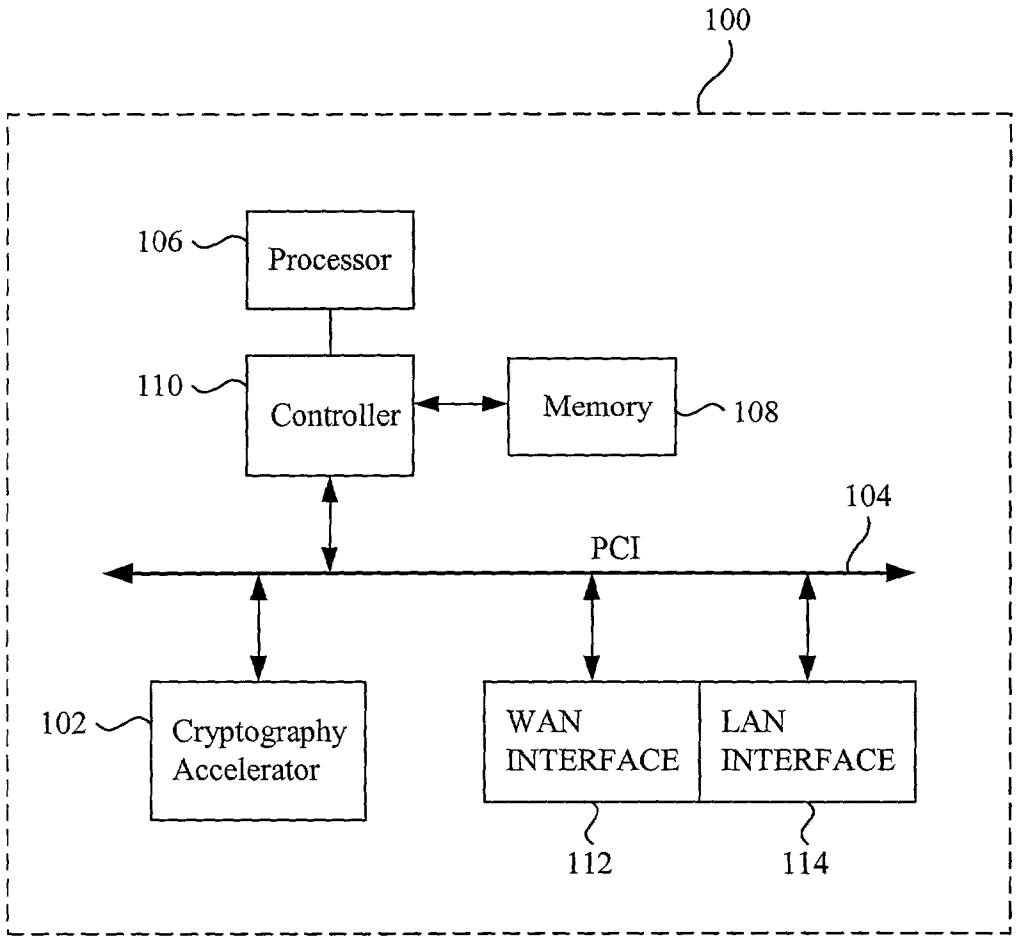

100
106
Processor
110
Controller
Memory
108
104
PCI
102
Cryptography Accelerator
WAN INTERFACE
LAN INTERFACE
112
114

Figure 1

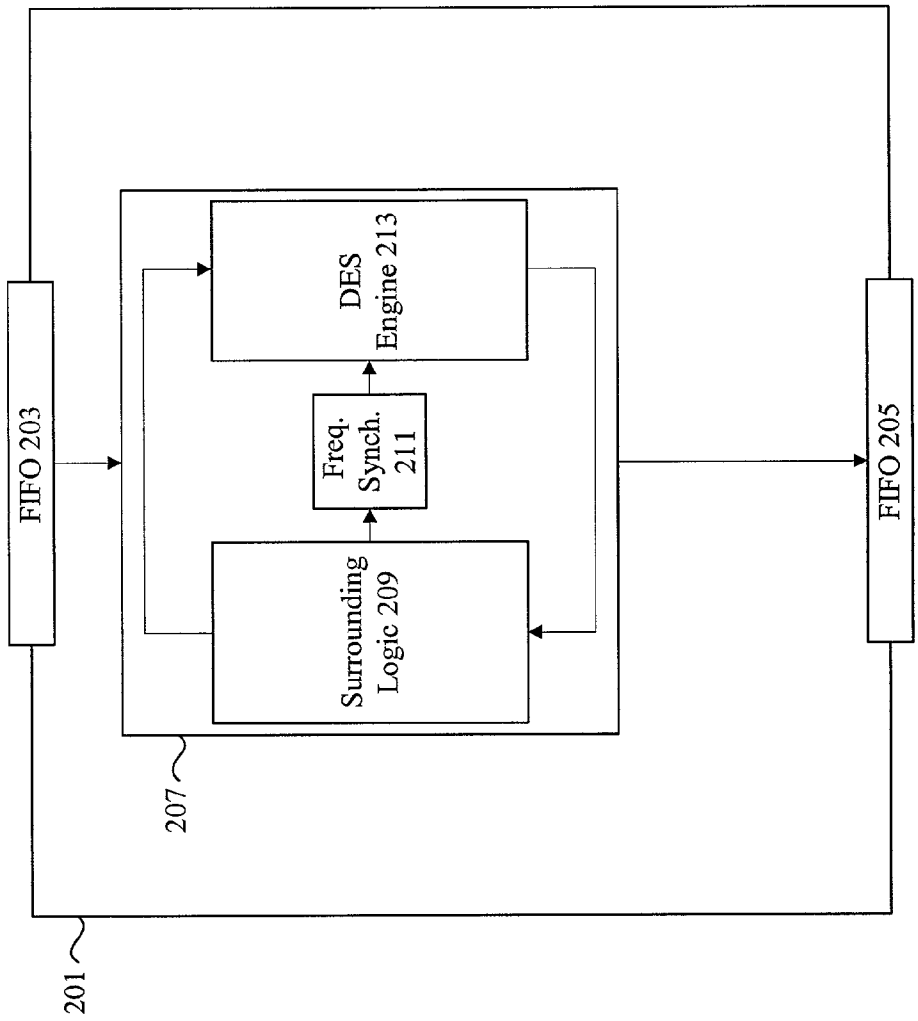

FIFO 203
DES Engine 213
Freq. Synch. 211
Surrounding Logic 209
FIFO 205
207
201

Figure 2

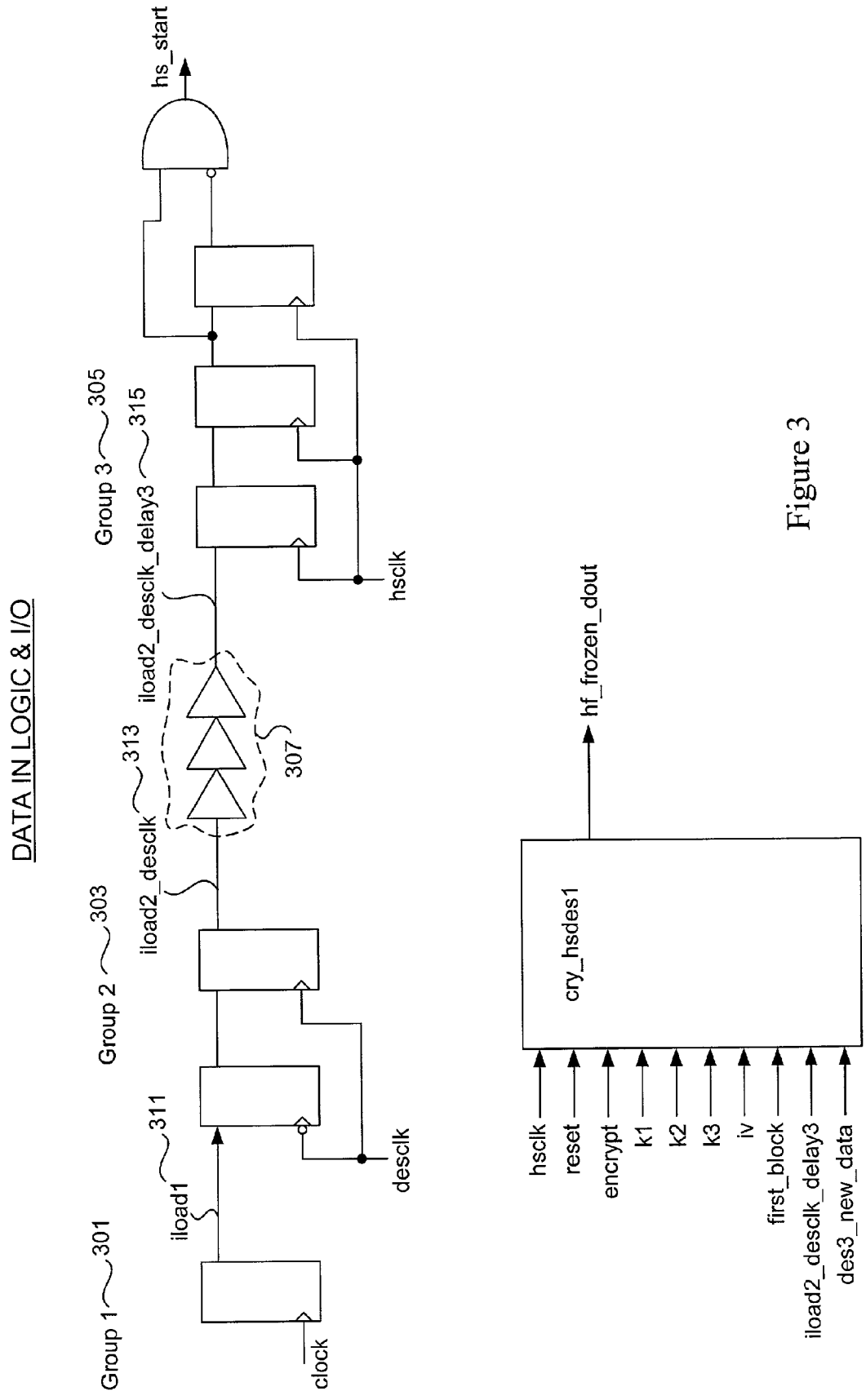

DATA IN LOGIC & I/O
Group 1
301
clock
iload1
311
desclk
Group 2
303
iload2_desclk
313
307
iload2_desclk_delay3
315
hsclk
Group 3
305
hs_start
hf_frozen_dout
cry_hsdes1
hsclk
reset
encrypt
k1
k2
k3
iv
first_block
iload2_desclk_delay3
des3_new_data

Figure 3

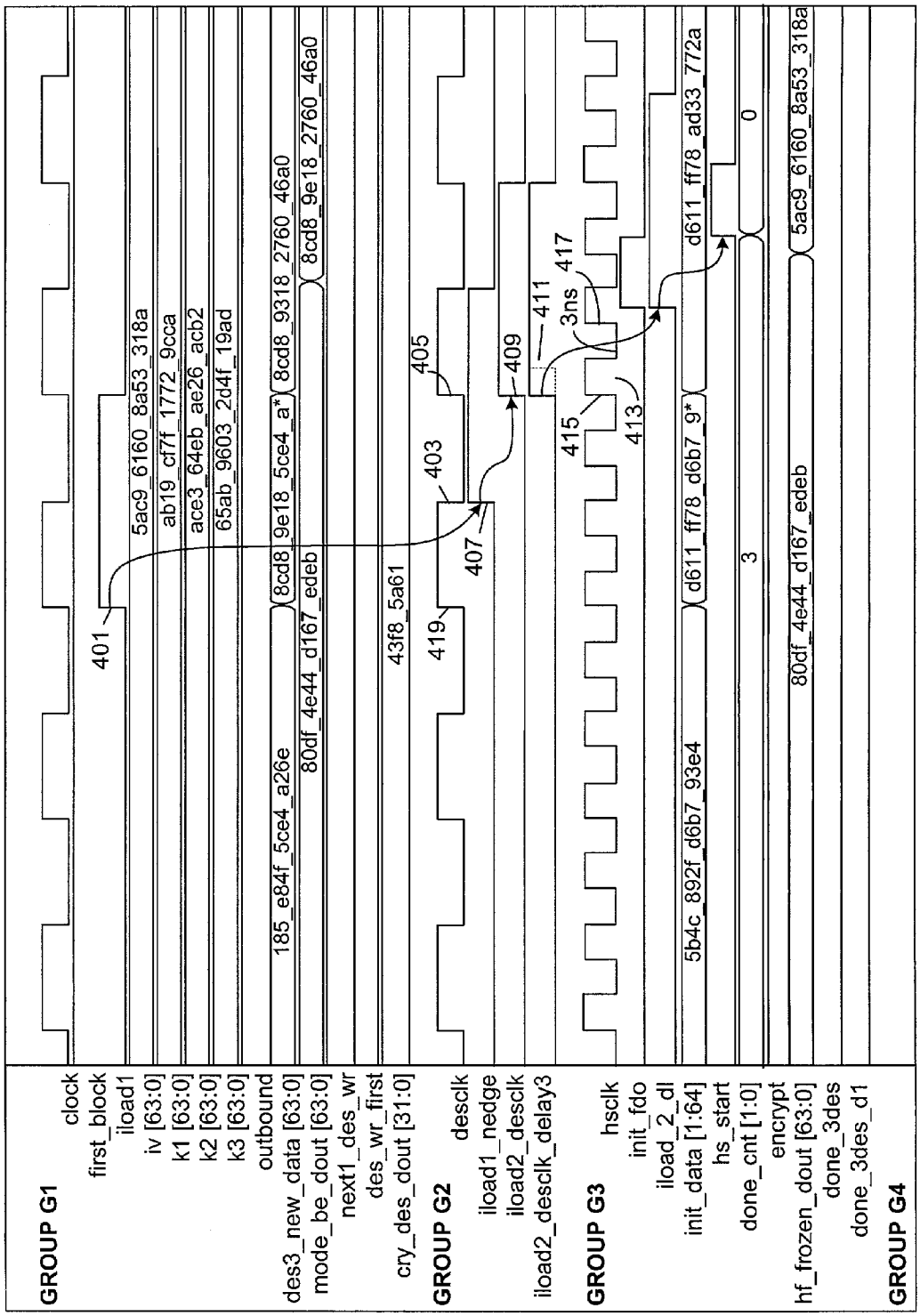

GROUP G1
clock
first_block
iload1
iv [63:0]
k1 [63:0]
k2 [63:0]
k3 [63:0]
outbound
des3_new_data [63:0]
mode_be_dout [63:0]
next1_des_wr
des_wr_first
cry_des_dout [31:0]
GROUP G2
desclk
iload1_nedge
iload2_desclk
iload2_desclk_delay3
GROUP G3
hsclk
init_fdo
iload_2_dl
init_data [1:64]
hs_start
done_cnt [1:0]
encrypt
hf_frozen_dout [63:0]
done_3des
done_3des_d1
GROUP G4
5ac9_6160_8a53_318a
ab19_cf7f_1772_9cca
ace3_64eb_ae26_acb2
65ab_9603_2d4f_19ad
185_e84f_5ce4_a26e
8cd8_9e18_5ce4_a*
8cd8_9318_2760_46a0
80df_4e44_d167_edeb
8cd8_9e18_2760_46a0
43f8_5a61
5b4c_892f_d6b7_93e4
d611_ff78_d6b7_9*
d611_ff78_ad33_772a
3
0
80df_4e44_d167_edeb
5ac9_6160_8a53_318a
401
403
405
407
409
411
413
415
417
419
3ns

Figure 4

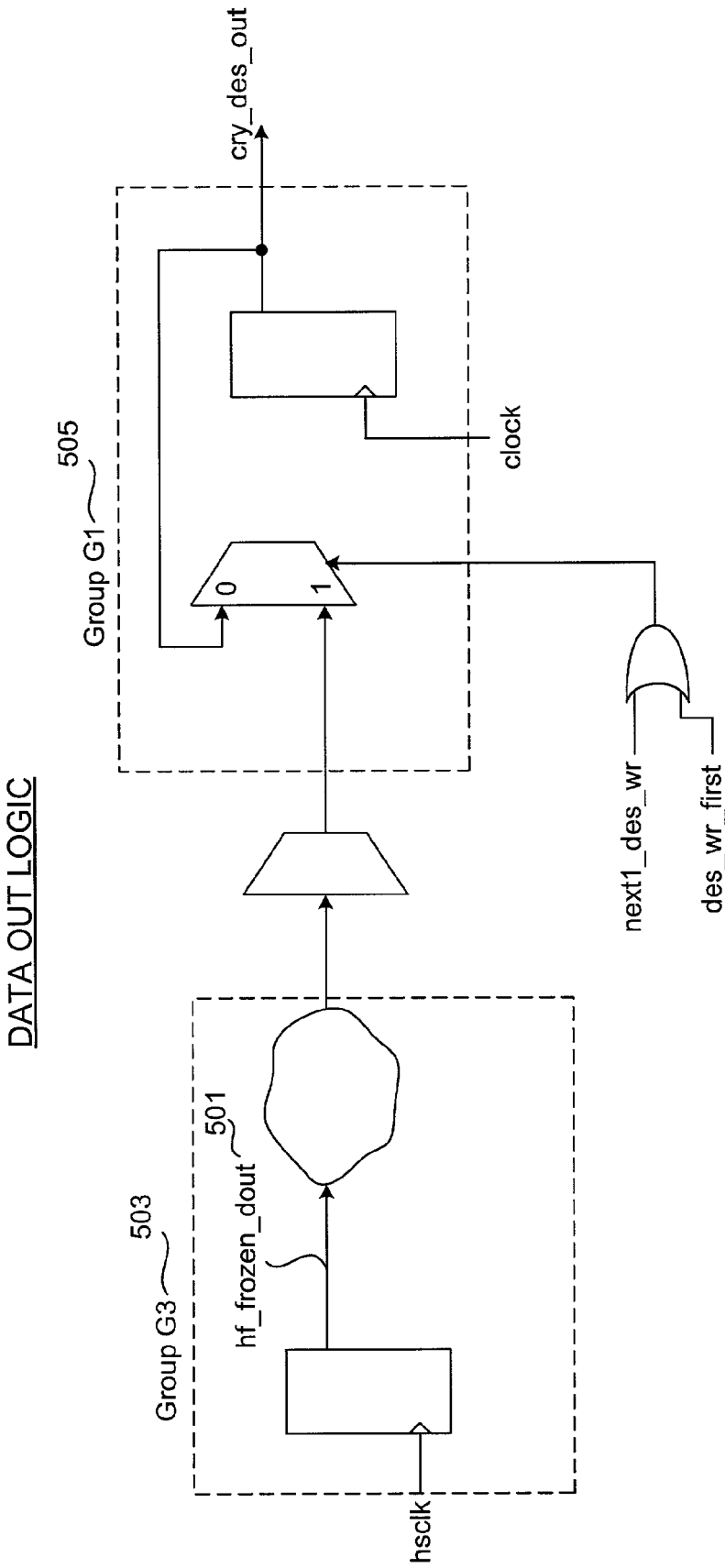

DATA OUT LOGIC
cry_des_out
505
Group G1
clock
0
1
next1_des_wr
des_wr_first
503
Group G3
501
hf_frozen_dout
hsclk

Figure 5

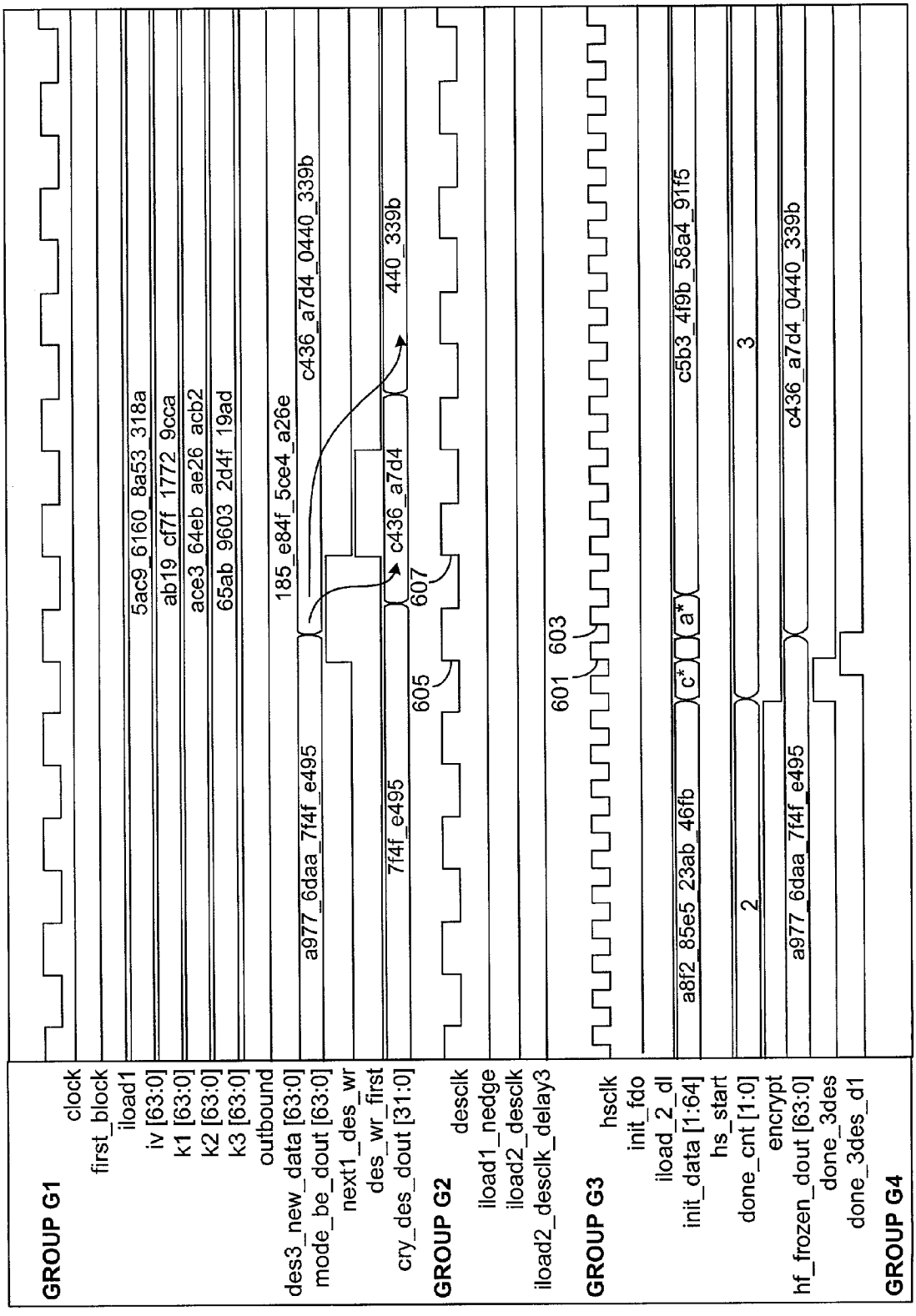

GROUP G1
clock
first_block
iload1
iv [63:0]
k1 [63:0]
k2 [63:0]
k3 [63:0]
outbound
des3_new_data [63:0]
mode_be_dout [63:0]
next1_des_wr
des_wr_first
cry_des_dout [31:0]
GROUP G2
desclk
iload1_nedge
iload2_desclk
iload2_desclk_delay3
GROUP G3
hsclk
init_fdo
iload_2_dl
init_data [1:64]
hs_start
done_cnt [1:0]
encrypt
hf_frozen_dout [63:0]
done_3des
done_3des_d1
GROUP G4
5ac9_6160_8a53_318a
ab19_cf7f_1772_9cca
ace3_64eb_ae26_acb2
65ab_9603_2d4f_19ad
185_e84f_5ce4_a26e
a977_6daa_7f4f_e495
c436_a7d4_0440_339b
7f4f_e495
c436_a7d4
440_339b
605
607
601
603
a8f2_85e5_23ab_46fb
c*
a*
c5b3_4f9b_58a4_91f5
2
3
a977_6daa_7f4f_e495
c436_a7d4_0440_339b

Figure 6

Figure 7

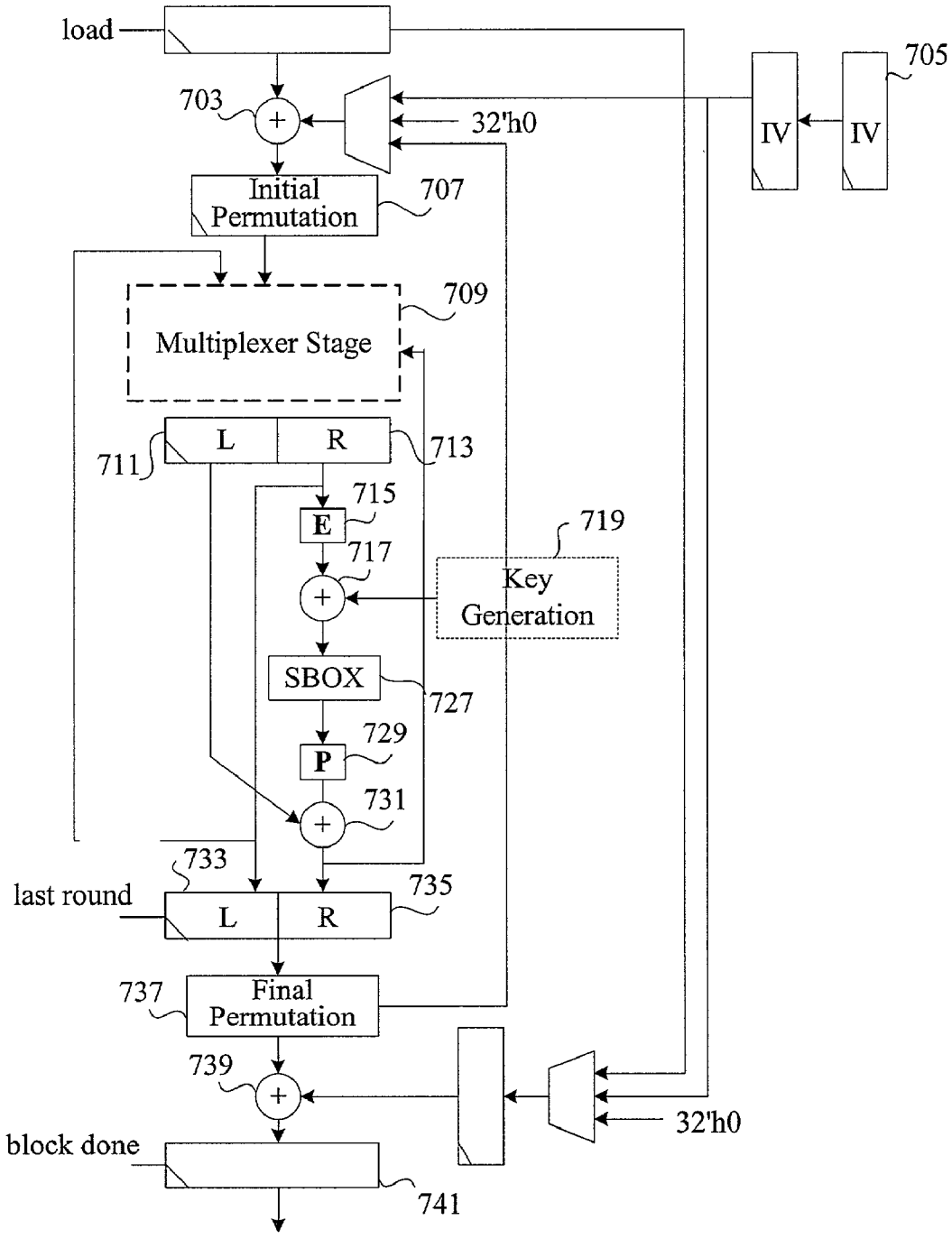

load
703
32'h0
705
IV
IV
Initial Permutation
707
709
Multiplexer Stage
711
L
R
713
715
E
717
719
Key Generation
SBOX
727
729
P
731
733
last round
735
L
R
737
Final Permutation
739
32'h0
block done
741

Figure 8

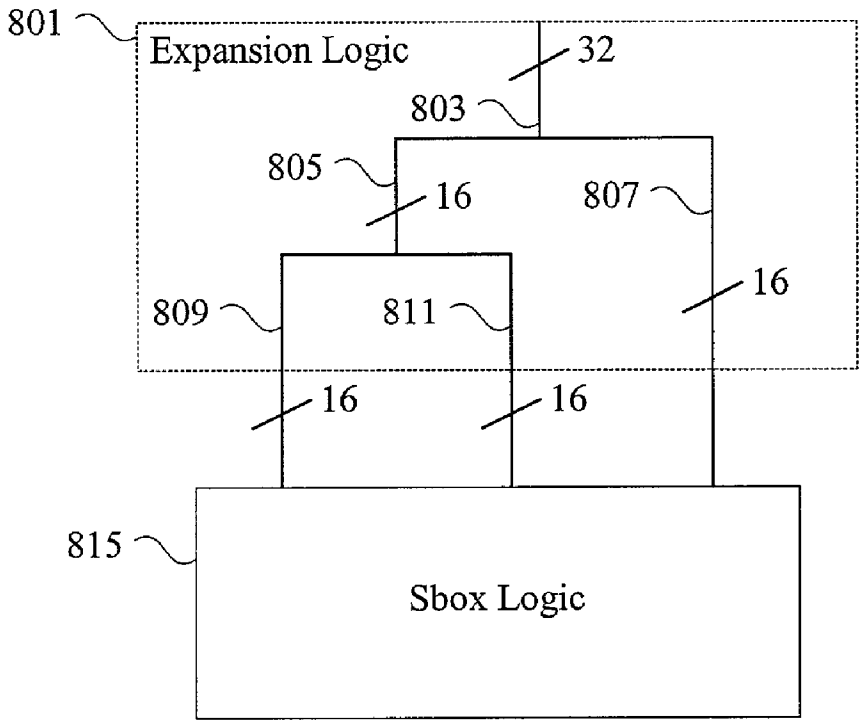

801
Expansion Logic
32
803
805
16
807
16
809
811
16
16
815
Sbox Logic

Figure 9

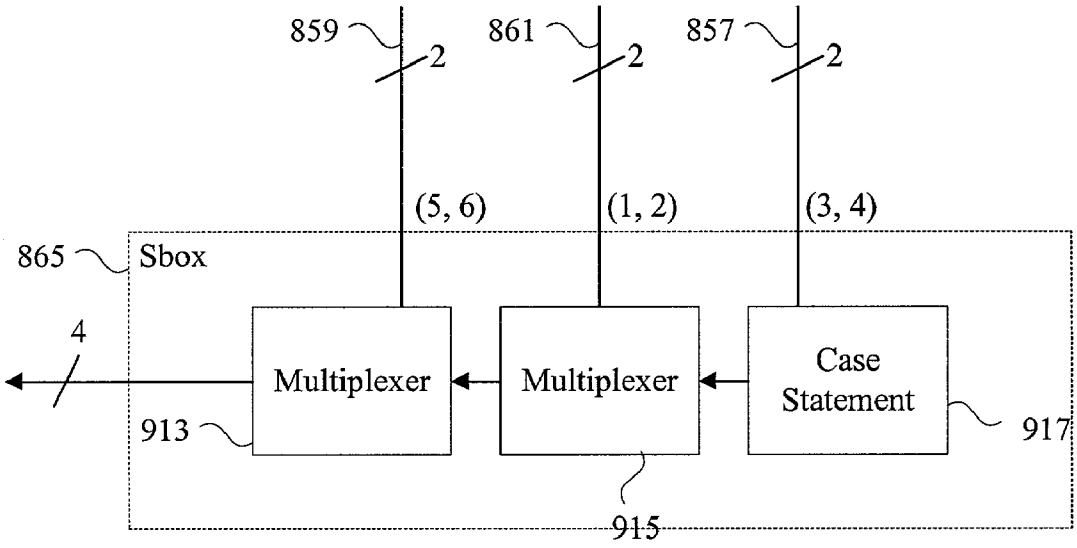
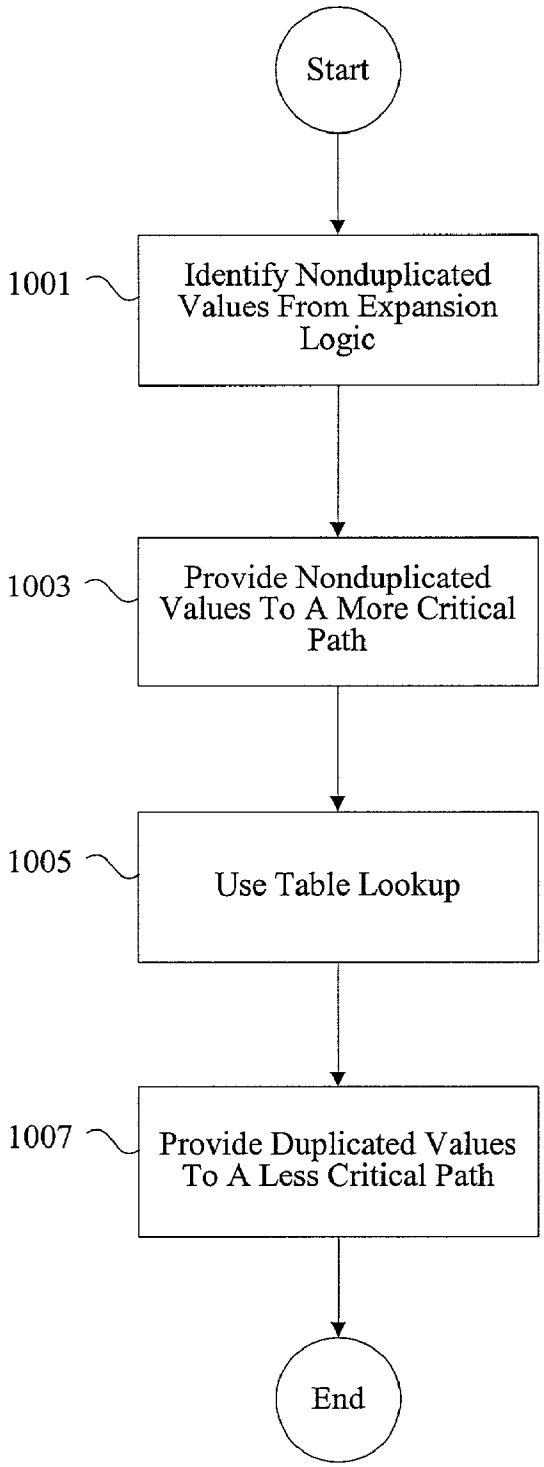

859
2
861
2
857
2
(5, 6)
(1, 2)
(3, 4)
865
Sbox
4
Multiplexer
Multiplexer
Case Statement
913
917
915

Figure 10


Start
1001
Identify Nonduplicated Values From Expansion Logic
1003
Provide Nonduplicated Values To A More Critical Path
1005
Use Table Lookup
1007
Provide Duplicated Values To A Less Critical Path
End

METHODS AND APPARATUS FOR IMPLEMENTING A CRYPTOGRAPHY ENGINE

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority under U.S.C. 119(e) from U.S. Provisional Application No. 60/235,190, entitled "E-Commerce Security Processor," as of filing on Sep. 25, 2000, the disclosure of which is herein incorporated by reference for all purposes.

BACKGROUND OF THE INVENTION

[0002] 1. Field of the Invention

[0003] The present invention relates to implementing a cryptography engine. More specifically, the present invention relates to methods and apparatus for efficient performance of a cryptography algorithm such as a DES algorithm.

[0004] 2. Description of Related Art

[0005] Conventional software and hardware designs for implementing various cryptography algorithms including the DES and the triple DES algorithms have significant performance limitations. Many designs for performing cryptography processing are well known in the art and are discussed, for example, in Applied Cryptography, Bruce Schneier, John Wiley & Sons, Inc. (ISBN 0471128457), incorporated by reference in its entirety for all purposes. In order to improve the speed of cryptography processing, specialized cryptography accelerators have been developed that typically out-perform similar software implementations. Examples of such cryptography accelerators include the BCM™ 5805 manufactured by Broadcom, Inc. of San Jose, Calif.

[0006] Although specialized hardware cryptography accelerators can often outperform software written to perform the same tasks, conventional hardware cryptography accelerators have significant performance limitations for implementing cryptography algorithms including the DES and the triple DES algorithms. Some performance bottlenecks include limitations related Sbox logic and clock synchronization.

[0007] It is therefore desirable to provide methods and apparatus for improving the implementation of cryptography algorithms with respect to some or all of the performance limitations noted above.

SUMMARY OF THE INVENTION

[0008] Methods and apparatus are provided for implementing a cryptography engine for cryptography processing. A variety of techniques are described. A cryptography engine such as a DES engine running at a clock frequency higher than that of surrounding logic can be synchronized with the surrounding logic using a frequency synchronizer. Sbox logic output can be more efficiently determined by intelligently arranging Sbox input.

[0009] In one embodiment, a cryptography engine for performing cryptographic operations on a data block is provided. The cryptography engine includes expansion logic configured to expand a first bit sequence into a second bit sequence by moving and duplicating selected bits in the first bit sequence to form a second bit sequence having a length greater than the length of the first bit sequence. The resulting bit sequence has duplicated bits and nonduplicated bits corresponding to a portion of the data block. The cryptography engine also includes Sbox logic coupled to the output of expansion logic, wherein nonduplicated bits are selected and provided as inputs to a critical path of the Sbox logic to perform cryptographic operations on the portion of the data block.

[0010] The Sbox logic can comprise a plurality of stages. The stages can be components such as case statements, table lookup components, or multiplexers.

[0011] In another embodiment, a method for performing cryptographic operations on a data block is provided. A first bit sequence is provided to expansion circuitry, where the expansion circuitry is configured to move and duplicate bits in the first bit sequence to output a second bit sequence having a length greater than the length of the first bit sequence. Nonduplicated bits output by the expansion circuitry are identified. Nonduplicated bits are provided to a first stage of Sbox circuitry. Duplicated bits output by the expansion circuitry are identified. The duplicated bits correspond to bits in a first bit sequence that are replicated to produce a second bit sequence. Duplicated bits and the output of the first stage of Sbox circuitry is provided to a second stage of Sbox circuitry, wherein duplicated bits are provided to the second stage of Sbox circuitry after nonduplicated bits are provided to the first stage of Sbox circuitry.

[0012] In another embodiment, a method for performing cryptographic operations on a data block is provided. Nonduplicated and duplicated bits resulting from expansion circuitry are identified. Nonduplicated bits are provided to a first stage of Sbox circuitry. Duplicated bits as well as data resulting from the first stage is provided to a second stage of Sbox circuitry.

[0013] In another embodiment, a cryptography accelerator for performing cryptography operations is provided. The cryptography accelerator includes a DES engine, a frequency synchronizer coupled to the DES engine, and surrounding logic coupled to the DES engine and the frequency synchronizer, wherein the DES engine operates at a first clock rate and the surrounding logic operates at a second clock rate different from the first clock rate.

[0014] The frequency synchronizer can use a reference clock associated with the surrounding logic to synchronize a 1×clock and a higher multiple clock associated with the DES engine. In one embodiment, the cryptography accelerator uses the negative edge of the 1×clock to catch a start signal associated with the reference clock to allow consideration of skew.

[0015] These and other features and advantages of the present invention will be presented in more detail in the following specification of the invention and the accompanying figures, which illustrate by way of example the principles of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0016] The invention may best be understood by reference to the following description taken in conjunction with the accompanying drawings, which are illustrative of specific embodiments of the present invention.

[0017] **FIG. 1** is a diagrammatic representation of a system that can use the techniques of the present invention.

[0018] **FIG. 2** is a diagrammatic representation of a cryptography engine having a frequency synchronizer, surrounding logic, and a DES engine.

[0019] **FIG. 3** is a diagrammatic representation of DES engine start logic.

[0020] **FIG. 4** is a graphical representation of a timing diagram depicting the clock synchronization for reading data from the surrounding logic.

[0021] **FIG. 5** is a diagrammatic representation of DES engine output logic.

[0022] **FIG. 6** is a graphical representation of a timing diagram depicting the clock synchronization for reading data from the DES engine.

[0023] **FIG. 7** is diagrammatic representation of a DES engine in accordance with one embodiment of the present invention.

[0024] **FIG. 8** is a diagrammatic representation of expansion logic providing data to an Sbox.

[0025] **FIG. 9** is a diagrammatic representation of expansion logic providing data to an Sbox having a plurality of stages.

[0026] **FIG. 10** is a process flow diagram showing the expansion logic providing bits to the Sbox logic.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

[0027] Reference will now be made in detail to some specific embodiments of the invention including the best modes contemplated by the inventors for carrying out the invention. Examples of these specific embodiments are illustrated in the accompanying drawings. While the invention is described in conjunction with these specific embodiments, it will be understood that it is not intended to limit the invention to the described embodiments. On the contrary, it is intended to cover alternatives, modifications, and equivalents as may be included within the spirit and scope of the invention as defined by the appended claims. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. The present invention may be practiced without some or all of these specific details. In other instances, well known process operations have not been described in detail in order not to unnecessarily obscure the present invention.

[0028] Techniques are provided for enabling the implementation of efficient cryptographic processing, such as DES and triple DES processing. DES specifies encrypting individual 64-bit data blocks. A 64-bit data block of unencrypted data is provided to the DES engine, combined with a key and an initial vector, and output as a 64-bit data block of encrypted data. The key used for DES processing is typically a 56-bit number, although the key can be expressed as a 64-bit number. DES describes breaking up a 64-bit block of data into a right half and a left half, each 32-bits long. As will be appreciated by one of skill in the art, sixteen rounds of cryptographic operations are then performed. In each round, operations on the right half of the data include expansion, permutation, Sbox operations, and combination with a round key. A round key can be determined based on the round number of DES processing is referred to herein as a round key. The round key can be derived by applying permutation and shift functions to all 56 bits of the original key. The round number of DES determines the shift amount.

[0029] An XOR function is used to combine the right half of the data with a version of the key based on the round. The result can then be combined with the left half also by way of an XOR function. The unprocessed right half of the data becomes the left half of the data for the next round. Triple DES specifies performing three 16 round DES operations consecutively using three different keys. Typical hardware implementations for DES or triple DES perform one round calculation per clock cycle. The performance of DES or triple DES engines is therefore related to the clock rate. However, a variety of hardware design constraints have traditionally limited the clock rate of cryptographic processing engines.

[0030] One design constraint that has traditionally limited the clock rate of cryptographic processing engines is the clock rate of circuitry surrounding the cryptographic processing unit. Surrounding logic typically has a slower clock rate than the DES engine. Methods and apparatus are described for providing a synchronizer to bridge the clock rate differences between the DES engine and the surrounding logic. The bridged DES engine and surrounding logic allow flexibility in setting different clock ratios. The flexibility allows the DES engine to be designed to operate at a much higher clock rate than the surrounding logic.

[0031] Other design constraints relate to Sbox processing. As noted above, expansion logic takes a 32-bit block, moves and replicates certain bits, and outputs a 48-bit block. Each Sbox associated with Sbox logic takes a six bit portion of the 48-bit block and maps the six bits to a four bit block. The expansion logic, however, is not able to provide all bits of the 48-bit block at the same time. In one example, replication of certain bits takes time and expansion logic can provide nonduplicated bits before it can provide duplicated bits. Each individual Sbox, however, typically waits for a complete six bit block before processing the block to output a four bit block. To optimize Sbox processing, the techniques of the present invention enables the processing of nonduplicated bits first. In other words, bits from expansion logic can be processed as soon as they become available. In one embodiment, bits three and four in each six bit sequence are processed as the nonduplicated bits by an Sbox before the duplicated bits one, two, five, and six.

[0032] Each Sbox can have a variety of components. Multiplexers can be used to determine what it to select based upon an input control line. In one embodiment, the techniques of the present invention contemplates using a table lookup component to process the nonduplicated bits from expansion logic. A table lookup component can be implemented more efficiently than a multiplexer. The Sbox can use the nonduplicated bits and the table lookup component to begin the process of mapping the six bit input to a four bit output. The Sbox can then use the duplicated bits to complete the process of mapping.

[0033] The techniques of the present invention can be implemented in a variety of contexts. **FIG. 1** describes the present invention in terms of a cryptographic accelerator system that can be implemented in a number of ways, such

as for example, as a stand alone integrated circuit, as embedded software, or as a subsystem included in, for example, a server computer used in a variety of Internet and Internet related activities. It should be noted, however, that the invention is not limited to the described embodiments and can be used in any system where data encryption is desired.

[0034] **FIG. 1** is a diagrammatic representation of one example of a cryptographic processing system **100** in accordance with an embodiment of the invention. As shown in **FIG. 1**, the present invention may be implemented in a stand-alone cryptography accelerator **102** or as part of the system **100**. In the described embodiment, the cryptography accelerator **102** is connected to a bus **104** such as a PCI bus via a standard on-chip PCI interface. The processing system **100** includes a processing unit **106** and a system memory unit **108**. The processing unit **106** and the system memory unit **108** are coupled to the system bus **104** via a bridge and memory controller **110**. Although the processing unit **106** may be the central processing unit or CPU of a system **100**, it does not necessarily have to be the CPU. It can be one of a variety of processors in a multiprocessor system, for example. A LAN interface **114** can couple the processing system **100** to a local area network (LAN) to receive packets for processing and transmit processed packets. Likewise, a Wide Area Network (WAN) interface **112** connects the processing system to a WAN (not shown) such as the Internet and manages in-bound and out-bound packets, providing automatic security processing for IP packets.

[0035] A cryptography accelerator **102** can perform many cryptography algorithms including the DES and the triple DES algorithms. It should be noted that although the techniques of the present invention will be described in the context of DES and triple DES for clarity, one of skill in the art will appreciate that the techniques can also apply to other cryptography algorithms including variations to the DES and triple DES algorithms.

[0036] **FIG. 2** is a diagrammatic representation of a cryptographic processing unit in accordance with one embodiment of the present invention. The cryptographic processing unit **201** has input and output buffers **203** and **205**. The input and output buffers **203** and **205** may be coupled to an external processor as shown in **FIG. 1**. The cryptographic processing unit **201** contains a cryptography engine **207**. The cryptographic processing unit **201** can also contain an authentication engine. As will be appreciated by one of skill in the art, other components including decoding logic may also be components in the cryptographic processing unit **201**. The cryptography engine **207** contains surrounding logic **209** and DES engine **213**. The DES engine **213** is coupled with the surrounding logic **209**. However, the DES engine **213** may operate at a much higher clock rate than the surrounding logic **209**. A frequency synchronizer **211** coupled to both the DES engine **213** and the surrounding logic **209** can manage data transmissions between DES engine **213** and surrounding logic **209**. The frequency synchronizer can coordinate data transmissions to the DES engine **213** to indicate when processing in the DES engine **213** should begin and notify surrounding logic **209** when data should be read from DES engine **213**.

[0037] The surrounding logic **209** can parse the data to allow processing of portions of the data by the DES engine **213**. The DES engine **213** can process a data block whenever a frequency synchronizer **211** sends a start signal. The frequency synchronizer **211** can compensate for the rate mismatch between the surrounding logic **209** and the DES engine **213**. The frequency synchronizer **211** provides flexibility for setting different clock ratios between DES engine **213** and surrounding logic **209** and allows reuse of the same DES engine **213** design in other processor configurations. In one example, the clock ratio between the DES engine **213** and the surrounding logic **209** is 3 to 1. In this example, the DES engine can operate at about 300 MHz while the surrounding logic can operate at about 100 MHz.

[0038] While this embodiment of the invention is described with respect to a DES engine, the invention may be applied more generally to other cryptography engines such as AES.

[0039] **FIG. 3** is a diagrammatic representation of logic for synchronizing the DES engine **213** and the surrounding logic **209**. The logic can be implemented in a frequency synchronizer **211**. **FIG. 3** will be described with reference to the timing diagram shown in **FIG. 4**. In the cryptography engine **207**, the DES engine **213** can be operated at a high clock rate in relation to surrounding logic **209** in order to optimize cryptography processing. In one embodiment, the DES engine operates at 300 MHz while the surrounding logic operates at 100 MHz. In various embodiments, the surrounding logic can also be operated at 300 MHz to build a fully synchronous cryptography engine. However, operating the surrounding logic at 300 MHz uses more on chip resources that can be used to improve the speed of the DES engine. To allow surrounding logic operating at a slower clock frequency to interact with a DES engine operating at a higher clock frequency, several mechanisms can be used. One mechanism is the use of asynchronous FIFOs. Asynchronous FIFOs are described in related U.S. patent application Ser. Nos. 09/892,240, 09/892,310, 09/892,242 all entitled "Methods And Apparatus For Implementing A Cryptography Engine," as of their filing date on Jun. 26, 2001, the entireties of which are herein incorporated by reference for all purposes.

[0040] Another mechanism to allow operation of surrounding logic clocked at a slower frequency and a DES engine clocked at a higher frequency is to use a frequency synchronizer. One problem associated with running a DES engine at a higher clock frequency than surrounding logic relates to skew. It should be noted that other problems such as jitter are also contemplated. Skew refers to the timing difference between design and actual elements. Skew can result from differences in signal paths and can reach a full nanosecond. Jitter may result from indeterministic variations in clock periods. In one example, a 100 MHz clock may not have a period of exactly 10 ns during every cycle. It may have a period of 10.005 ns in one cycle and 9.5 ns in another cycle.

[0041] Surrounding logic may expect that the DES engine will read a signal every 10 ns. Consequently, surrounding logic may provide a signal such as a start signal to begin DES processing when this 10 ns interval occurs. However, the high-speed clock associated with the DES engine may have skew associated with it. In one example, the DES engine will read a start signal when this 10 ns interval arrives. However, due to skew, the DES engine may attempt

to read a start signal after 9.995 ns. In other words, the DES engine may attempt to read a start signal before the start signal is available from the surrounding logic. Skew may cause the DES engine to miss the start signal.

[0042] In the example where the DES engine operates at three times the clock rate of the surrounding logic, frequency synchronization logic shown in **FIG. 3** having timing characteristics shown in **FIG. 4** can be used. Group GI **301** provides a signal iloadl to Group G**2303** at **401**. The signal iloadl **311** is read upon the occurrence of a negative edge **403** of a 1×clock. The negative edge **403** of the 1×clock instead of the rising edge **419** of the 1×clock is used so that the iloadl signal will not be missed because of skew. The signal is then read again using rising edge **405** to provide iload**2**_desclk **313**. A delay element **307** is then introduced so that the iload**2**_desclk **313** signal will not be read at 3×clock leading edge **415** but instead at 3×clock leading edge **417**. Various delays are contemplated. The delay amount can factor in the clock frequency of the DES engine, the clock frequency of the surrounding logic, and the amount of expected skew. The amount of delay can vary from a fraction of a nanosecond to several nanoseconds. In one embodiment where the 3×clock is 300 MHz and the 1×clock is 100 MHz, the amount of delay is 1.5 ns. Using the delay element **307**, group G**3305** reads the start signal at rising edge **417** and provides the start signal to the DES engine.

[0043] The DES engine can then perform DES processing. It should be noted that the surrounding logic will be able to anticipate the number of clock cycles needed to complete DES processing. As will be appreciated by one of skill in the art, a typical implementation of DES uses a predetermined number of clock cycles. The surrounding logic can then read data from the DES engine after the predetermined number of clock cycles has elapsed.

[0044] **FIG. 5** is a diagrammatic representation of data out logic and will be described with reference to the timing diagram shown in **FIG. 6**. In one embodiment shown in **FIGS. 5 and 6**, the surrounding logic waits one clock cycle of the high speed clock before reading the data from the DES engine. Group G**3503** provides a signal hf_frozen_dout **501** to Group G**1505**. Group G**1505** reads the signal not on the 1×clock rising edge **605** corresponding to rising edge **601** of the 3×clock but on the second rising edge **603** of the 3×clock. By waiting one clock cycle, the risk of missing the signal due to skew can be reduced.

[0045] As will be appreciated by one of skill the art, a variety of components such as registers, flip-flops, and delay elements can be used to implement a frequency synchronizer. Furthermore, it will be appreciated that the frequency synchronizer can be associated with edge triggering, level triggering, or other triggering mechanisms.

[0046] **FIG. 7** is a diagrammatic representation of a DES engine **213** as shown in **FIG. 2**. The DES engine can be coupled to a frequency synchronizer **211** to allow the DES engine **213** to operate at a different clock rate than surrounding logic. According to various embodiments, a 64-bit data block is combined with an initialization vector from initialization vector block **705**. Initialization vectors (IV) are described in RFC 2405 titled The ESP DES-CBC Cipher Algorithm With Explicit IV, the entirety of which is incorporated by reference for all purposes. The 64-bit block then undergoes an initial permutation at **707**. The initial permutation occurs before round **1**. It should be noted that in some variations to DES, to which the present invention is applicable, initial permutation and final permutation operations are not performed. The 64-bit block can then be passed to a multiplexer stage **709**.

[0047] The multiplexer stage **709** contains logic for determining whether to load initial data, swap data from the previous round, or not swap data from the previous round. In one embodiment, the multiplexer uses a 3-to-1 multiplexer to select either the initial data, the swapped feedback data, or the non-swapped feedback data. Initial data is loaded in the first round of DES processing. Data is swapped between rounds of DES processing. Data is not swapped in triple DES between the completed 16 rounds of DES processing. Control logic (not shown) can track the round number in order to determine what signals to send to the multiplexers.

[0048] Registers **711** and **713** receive the initial data or the feedback data. It should be noted that the registers referenced in **FIG. 7** can be clocked at a high speed clock rate different than that of surrounding logic. Register **711** contains the last half of this initial 64-bit block in round **1**. Register **713** contains the right half of the 64-bit block in round **1**. Registers **711** and **713** both typically hold 32-bits of data. The 32-bit data block contained in register **713** is provided to both expansion stage **715** and to register **711** through multiplexer stage **709** for the next round. Control signals in multiplexer stage **709** are configured to provide the 32-bit data block contained in register **713** to register **711** in the next round of DES processing. The 32-bit data block is provided to expansion logic **715**.

[0049] As will be appreciated by one of skill in the art, the expansion logic **715** changes the order of the bits in the 32-bit at block and also repeats certain bits. The expansion logic **715** uses the 32-bit block to generate a 48-bit block. The expansion logic improves the effectiveness of the encryption process and also makes the 32-bit block into a 48-bit block that corresponds to the size of the key. The 48-bit block can then be combined with an XOR with the 48-bit round key at **717**.

[0050] Keys are provided by key generation logic or key scheduler circuitry **719**. A version of the key for cryptography processing of the original 64-bit block is provided by key scheduler **719**. Key scheduler **719** can provide a different version of the original key for every round by applying permutation and shift functions to all 56 bits of the original key. The 48-bit block resulting from the XOR at **717** is provided to Sbox stage **727**. As will be appreciated by one of skill the art, each Sbox in Sbox stage **727** converts a six-bit input into a four-bit output. According to various embodiments eight Sboxes are provided in Sbox stage **727**. Sboxes, expansion logic, and other cryptography operations are described in Applied Cryptography, Bruce Schneier (ISBN 0471128457), the entirety of which is incorporated by reference for all purposes.

[0051] The 32-bit output of Sbox stage **727** is provided to permutation stage **729**. A permutation stage **729** maps input bits in certain positions to different output positions. In one example, bit **1** is moved to position **16** of the output, bit **2** is moved to position **7** of the output, and bit **3** is moved to position **20** of the output, etc. The 32-bit output of permutation stage **729** is combined with an XOR with the value in

register **711** at **731**. The result of the XOR is provided to the register **711** through multiplexer stage **709** for the next round of DES processing. The same process occurs for the subsequent rounds of DES processing. That is, the right half is expanded, combined with an XOR function with a version of the key, provided to an Sbox stage, permuted, and combined with an XOR with the left half. After the last round, the outputs are written to register **733** and register **735**. The output can then undergo a final permutation at **737**. The result of a final permutation at **737** is combined by way of an XOR with an initialization vector as noted above when the DES engine is used to decrypt data. Otherwise, the result of the final permutation at **737** can remain unchanged by combining by way of an XOR with a sequence of zeros.

[**0052**] For triple DES, the outputs at **733** and **735** are passed back to multiplexer stage **709**. Control circuitry determines how to pass the data back to register **711** and **713** for a next **16** rounds of DES processing. The processed data resulting from the DES or triple DES algorithm is provided to output register **741**.

[**0053**] **FIG. 8** is a diagrammatic representation of expansion logic **801** coupled to Sbox logic **815**. As will be appreciated by one of skill in the art, expansion logic rearranges and duplicates bits in a 32-bit block to output a 48-bit data block. The 48-bit data block can then be XORed with a round key and provided to Sbox logic **815**. The 32-bit data block along path **803** is divided into a 16-bit data block along path **807** and a 16-bit data block along path **805**. The 16-bit data block along path **805** is duplicated and provided along data paths **811** and **809** as 16-bit data blocks. However, the bit duplication process takes time. Consequently, the 16-bit data blocks on paths **811** and **809** are generated after the 16-bit data block on path **807** is output to Sbox logic **815**.

[**0054**] In conventional implementations, the Sbox logic **815** waits for the entire 48-bit data block from the expansion logic **801** before proceeding with Sbox processing. However, waiting for the entire 48-bit bit data block does not benefit from the fact that the 16-bit data block along path **807** can be provided at a slightly earlier time. By beginning processing of the 16-bit data block **807** before processing the 16-bit data blocks on path is **811** and **809**, Sbox logic can be optimized to process data as soon as it is available. Generally, the data block that is most quickly output from expansion logic **801** is referred to herein as the nonduplicated block or nonduplicated bits. A less quickly output data block, typically delayed because of replication logic, is referred to herein as a duplicated block or duplicated bits. According to various embodiments, the 16-bit data block on path **807** has nonduplicated bits while the 16-bit bit data blocks provided on paths **811** and **809** has duplicated bits.

[**0055**] It should be noted that duplicated and nonduplicated bits can be provided to each individual Sbox as well. As will be appreciated by one of skill in the art, Sbox logic **815** is commonly implemented using eight separate individual Sboxes. Each individual Sbox receives a six-bit input and maps the six-bit input to the corresponding four-bit output. The expansion logic provides each Sbox with a six-bit input. Expansion logic rearranges and duplicates bits in a four-bit data block to output a six-bit data block. The six-bit data block can then be XORed with a key and provided to Sbox logic **815**.

[**0056**] **FIG. 9** is a diagrammatic representation showing expansion logic providing data to Sbox logic having multiple components. Sbox logic can be implemented using a variety of components. As noted above, and Sbox maps a six-bit input to a four-bit output. In other words, and Sbox can map a value ranging from 000000 to 111111 to a value ranging form 0000 to 1111. In one implementation, a table with the 64 entries ranging from 000000 to 111111 can be referenced in order to determine the four-bit output ranging from 0000 to 1111. The six-bit input can be referenced to the 64 entries to locate the corresponding four-bit entry. However, referencing a large table using six bits can be inefficient. Furthermore, referencing a 64 entry table using a six-bit input does not benefit from the nonduplicated bits being provided more quickly to the Sbox logic **865**.

[**0057**] According to various embodiments, the Sbox **865** uses nonduplicated bits from expansion logic when they become available. A table lookup component **917** can use the nonduplicated bits provided on path **857** to narrow the 64 possible bit values to 16 possible bit values. In one example, bits three and four are the nonduplicated bits on path **857**. Depending on whether bits three and four correspond to (0, 0), (0, 1), (1, 0), or (1, 1), 16 possible bit values are provided to multiplexer stage **915**. Code for implementing the mapping of six bit values on to four bit values using Sbox components is provided in Table 1 below. After a table lookup component **917** has narrowed the possible values from 64 to 16 using bits three and four, the 16 possible values are provided to multiplexer stage **915**. According to various embodiments, the component **917** can be referred to herein as a first stage. The first stage can be a component such as a table lookup component, a case statement, or a multiplexer. The first stage **917** can pass data to a second stage **915** that may comprise a multiplexer. The multiplexer stage **915** uses duplicated bits, such as bits one and two or bits five and six to narrow the 16 possible values to four possible values depending on whether the duplicated bits correspond to (0, 0), (0, 1), (1, 0), or (1, 1). Before possible values are provided to multiplexer stage **913** which outputs a single four bit values based on duplicated bits provided on path **859**. It should be noted that multiplexer stage **913** can be referred to herein as a third stage.

[**0058**] It should be noted in various embodiments, multiplexer stage **913** cannot process bits on path **859** until provided information from multiplexer stage **915**. A multiplexer stage **915** similarly cannot process bits on path **861** until provided information from table lookup stage **917**. Consequently, table lookup stage **917** is associated with the critical path of Sbox **865**. A multiplexer stage is **913** and **915** can only process after table lookup component **917** has completed processing. The multiplexer stages **913** and **915** can be referred to as less critical paths in Sbox **865**. It is beneficial to allow table lookup component **917** to begin operations as soon as possible, as multiplexer stages **915** and **913** can not perform processing until table lookup component **917** is finished. One way of allowing the table lookup component **917** to begin operations quickly is to provide nonduplicated bits from expansion logic **851**. As noted above, nonduplicated bits can be provided from expansion logic **851** before duplicated bits.

[**0059**] According to other embodiments, table lookup component **917** can be implemented as a multiplexer stage. However, by using a table lookup component **917** instead of

a multiplexer stage, the processing of the bits on path **857** can be performed more efficiently. A multiplexer stage **913** uses inputs from path **859** and from multiplexer **915**. A multiplexer stage **915** uses inputs from path **861** and from component **917**. According to various embodiments, component **917** can provide outputs using input only from path **857**. Thus, a table lookup component can be used instead of a multiplexer at **917**.

[**0060**] **FIG. 10** is a process flow diagram showing the interaction of expansion logic and Sbox logic. At **1001**, nonduplicated values from expansion logic are identified. The nonduplicated values are typically bits three and four in a six bit block provided from expansion logic to a single Sbox. At **1003**, the nonduplicated values are provided to the most critical path of the Sbox. According to various embodiments, the most critical path is the table lookup component the provides input to a subsequent multiplexer stage as shown in **FIG. 9**. At **1005**, a table lookup component can be used to process bits in the most critical path. At **1007**, the duplicated values or delayed values are provided to bless critical path of the Sbox. The duplicated values may be provided in two bit blocks to separate multiplexer stages or another example, the duplicated values may be provided as a four bit block to another table lookup component. As will be appreciated by one of skill in the art, a variety of different components can be used in a critical paths of an Sbox. For example, a variety of 2 to 1, 3 to 1, and 4 to 1 multiplexers can be used along with table lookup components.

[**0061**] While the invention has been particularly shown and described with reference to specific embodiments thereof, it will be understood by those skilled in the art that changes in the form and details of the disclosed embodiments may be made without departing from the spirit or scope of the invention. For example, embodiments of the present invention may be employed with a variety of encryption algorithms and should not be restricted to the ones mentioned above. Therefore, the scope of the invention should be determined with reference to the appended claims.

TABLE 1

Sbox Implementation Source Code

```
module cry_hs_sbox1 (key_data, right_data, sbox1);
input [1:6]     key_data;
input [1:6]     right_data;
output [1:4]    sbox1;
/*************************************************/
wire    [1:6] sbox1_sel;
reg     [1:4] sbox1a,sbox1b,sbox1c,sbox1d,sbox1e,sbox1f,sbox1g,sbox1h;
reg     [1:4] sbox1i,sbox1j,sbox1k,sbox1l,sbox1m,sbox1n,sbox1o,sbox1p;
wire    [1:4] sbox1w, sbox1x, sbox1y, sbox1z;
wire    [1:4] sbox1;
/*************************************************/
assign sbox1_sel = key_data ^right_data;
/****************/
always @(sbox1_sel)
    begin
        // case ({chunk[3:4, 1:2, 5:6]})
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*00 00*/: sbox1a[1:4]=4'd14;
            2'b01/*00 00*/: sbox1a[1:4]=4'd13;
            2'b10/*00 00*/: sbox1a[1:4]=4'd2;
            2'b11/*00 00*/: sbox1a[1:4]=4'd11;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*01 00*/: sbox1b[1:4]=4'd3;
            2'b01/*01 00*/: sbox1b[1:4]=4'd6;
```

TABLE 1-continued

Sbox Implementation Source Code

```
            2'b10/*01 00*/: sbox1b[1:4]=4'd5;
            2'b11/*01 00*/: sbox1b[1:4]=4'd0;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*10 00*/: sbox1c[1:4]=4'd4;
            2'b01/*10 00*/: sbox1c[1:4]=4'd14;
            2'b10/*10 00*/: sbox1c[1:4]=4'd13;
            2'b11/*10 00*/: sbox1c[1:4]=4'd2;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*11 00*/ : sbox1d[1:4]=4'd15;
            2'b01/*11 00*/ : sbox1d[1:4]=4'd9;
            2'b10/*11 00*/ : sbox1d[1:4]=4'd3;
            2'b11/*11 00*/ : sbox1d[1:4]=4'd5;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*00 01*/: sbox1e[1:4]=4'd0;
            2'b01/*00 01*/: sbox1e[1:4]=4'd7;
            2'b10/*00 01*/: sbox1e[1:4]=4'd14;
            2'b11/*00 01*/: sbox1e[1:4]=4'd13;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*01 01*/: sbox1f[1:4]=4'd10;
            2'b01/*01 01*/: sbox1f[1:4]=4'd12;
            2'b10/*01 01*/: sbox1f[1:4]=4'd9;
            2'b11/*01 01*/: sbox1f[1:4]=4'd3;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*10 01*/: sbox1g[1:4]=4'd15;
            2'b01/*10 01*/: sbox1g[1:4]=4'd8;
            2'b10/*10 01*/: sbox1g[1:4]=4'd4;
            2'b11/*10 01*/: sbox1g[1:4]=4'd1;
        endcase
        case (sbox1_1:4]sel[3:4]) // synopsys full_case parallel_case
            2'b00/*11 01*/: sbox1h[1:4]=4'd5;
            2'b01/*11 01*/: sbox1h[1:4]=4'd3;
            2'b10/*11 01*/: sbox1h[1:4]=4'd10;
            2'b11/*11 01*/: sbox1h[1:4]=4'd6;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*00 10*/: sbox1i[1:4]=4'd4;
            2'b01/*00 10*/: sbox1i[1:4]=4'd1;
            2'b10/*00 10*/: sbox1i[1:4]=4'd15;
            2'b11/*00 10*/: sbox1i[1:4]=4'd8;
        endcase
        case (sbox1_sel[3:4])// synopsys full_case parallel_case
            2'b00/*01 10*/: sbox1j[1:4]=4'd10;
            2'b01/*01 10*/: sbox1j[1:4]=4'd12;
            2'b10/*01 10*/: sbox1j[1:4]=4'd9;
            2'b11/*01 10*/: sbox1j[1:4]=4'd7;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*10 10*/: sbox1k[1:4]=4'd1;
            2'b01/*10 10*/: sbox1k[1:4]=4'd8;
            2'b10/*10 10*/: sbox1k[1:4]=4'd6;
            2'b11/*10 10*/: sbox1k[1:4]=4'd11;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*11 10*/: sbox1l[1:4]=4'd12;
            2'b01/*11 10*/: sbox1l[1:4]=4'd7;
            2'b10/*11 10*/: sbox1l[1:4]=4'd10;
            2'b11/*11 10*/: sbox1l[1:4]=4'd0;
        endcase
        case (sbox1_sel[3:4])// synopsys full_case parallel_case
            2'b00/*00 11*/: sbox1m[1:4]=4'd15;
            2'b01/*00 11*/: sbox1m[1:4]=4'd4;
            2'b10/*00 11*/: sbox1m[1:4]=4'd2;
            2'b11/*00 11*/: sbox1m[1:4]=4'd1;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*01 11*/: sbox1n[1:4]=4'd6;
            2'b01/*01 11*/: sbox1n[1:4]=4'd11;
```

TABLE 1-continued

Sbox Implementation Source Code

```
            2'b10/*01 11*/: sbox1n[1:4]=4'd5;
            2'b11/*01 11*/: sbox1n[1:4]=4'd8;
        endcase
        case (sbox1_sel[3:4]) // synopsys full_case parallel_case
            2'b00/*10 11*/: sbox1o[1:4]=4'd12;
            2'b01/*10 11*/: sbox1o[1:4]=4'd2;
            2'b10/*10 11*/: sbox1o[1:4]=4'd9;
            2'b11/*10 11*/: sbox1o[1:4]=4'd7;
        endcase
        case (sbox1_sel[3 :4])// synopsys full_case parallel_case
            2'b00/*11 11*/: sbox1p[1:4]=4'd11;
            2'b01/*11 11*/: sbox1p[1:4]=4'd14;
            2'b10/*11 11*/: sbox1p[1:4]=4'd0;
            2'b11/*11 11*/: sbox1p[1:4]=4'd13;
        endcase
end
/*********************/
cry_mux4to1x2_4b u_sbox1w (.sel(sbox1_sel[1:2]),
                           .i0(sbox1a[1:4]), .i1(sbox1b[1:4]),
                           .i2(sbox1c[1:4]), .i3(sbox1d[1:4]),
                           .out(sbox1w[1:4])
                          );
cry_mux4to1x2_4b u_sbox1x (.sel(sbox1_sel[1:2]),
                           .i0(sbox1e[1:4]), .i1(sbox1f[1:4]),
                           .i2(sbox1g[1:4]), .i3(sbox1h[1:4]),
                           .out(sbox1x[1:4])
                          );
cry_mux4to1x2_4b u_sbox1y (.sel(sbox1_sel[1:2]),
                           .i0(sbox1i[1:4]), .i1(sbox1j[1:4]),
                           .i2(sbox1k[1:4]), .i3(sbox1l[1:4]),
                           .out(sbox1y[1:4])
                          );
cry_mux4to1x2_4b u_sbox1z (.sel(sbox1_sel[1:2]),
                           .i0(sbox1m[1:4]), .i1(sbox1n[1:4]),
                           .i2(sbox1o[1:4]), .i3(sbox1p[1:4]),
                           .out(sbox1z[1:4])
                          );
cry_mux4to1x2_4b u_sbox1 (.sel(sbox1_sel[5:6]),
                           .i0(sbox1w[1:4]), .i1(sbox1x[1:4]),
                           .i2(sbox1y[1:4]), .i3(sbox1z[1:4]),
                           .out(sbox1[1:4])
                          );
/*********************************************************/
endmodule // cry_sbox1
```

What is claimed is:

1. A cryptography engine for performing cryptographic operations on a data block, the cryptography engine comprising:

expansion logic configured to expand a first bit sequence into a second bit sequence by moving and duplicating selected bits in the first bit sequence to form a second bit sequence having a length greater than the length of the first bit sequence, the resulting bit sequence having duplicated bits and nonduplicated bits corresponding to a portion of the data block;

Sbox logic coupled to the output of expansion logic, wherein nonduplicated bits are selected and provided as inputs to a critical path of the Sbox logic to perform cryptographic operations on the portion of the data block.

2. The cryptography engine of claim 1, wherein the Sbox logic comprises a plurality of stages.

3. The cryptography engine of claim 2, wherein nonduplicated bits are selected as inputs to a first stage of the Sbox logic.

4. The cryptography engine of claim 3, wherein the first stage of the Sbox logic performs a table lookup.

5. The cryptography engine of claim 2, wherein the first stage of the Sbox logic is a multiplexer.

6. The cryptography engine of claim 5, wherein the second stage of the Sbox logic is a multiplexer.

7. The cryptography engine of claim 6, wherein duplicated bits are provided as inputs to the second stage of the Sbox logic.

8. The cryptography engine of claim 1, wherein bits **3** and **4** are provided to the critical path of the Sbox logic.

9. The cryptography engine of claim 1, wherein the first bit sequence is 6 bits in length.

10. The cryptography engine of claim 1, wherein the second bit sequence is 4 bits in length.

11. A method for performing cryptographic operations on a data block, the method comprising:

providing a first bit sequence to expansion circuitry, the expansion circuitry configured to move and duplicate bits in the first bit sequence to output a second bit sequence having a length greater than the length of the first bit sequence;

identifying nonduplicated bits output by the expansion circuitry;

providing nonduplicated bits to a first stage of Sbox circuitry;

identifying duplicated bits output by the expansion circuitry, wherein duplicated bits correspond to bits in a first bit sequence that are replicated to produce a second bit sequence;

providing duplicated bits and the output of the first stage of Sbox circuitry to a second stage of Sbox circuitry, wherein duplicated bits are provided to the second stage of Sbox circuitry after nonduplicated bits are provided to the first stage of Sbox circuitry.

12. The method of claim 11, wherein the Sbox circuitry comprises three stages.

13. The method of claim 12, wherein the first stage of Sbox circuitry is a case statement.

14. The method of claim 12, wherein the first stage of Sbox circuitry performs a table lookup.

15. The method of claim 12, wherein the first stage of Sbox circuitry is a multiplexer.

16. The method of claim 15, wherein the second stage of Sbox circuitry is a multiplexer.

17. The method of claim 11, wherein bits **3** and **4** of a 6 bit block are provided to the first stage of Sbox circuitry.

18. A cryptography accelerator for performing cryptographic operations on a data block, the cryptography accelerator comprising:

means for providing a first bit sequence to expansion circuitry, the expansion circuitry configured to move and duplicate bits in the first bit sequence to output a second bit sequence having a length greater than the length of the first bit sequence;

means for identifying nonduplicated bits output by the expansion circuitry;

means for providing nonduplicated bits to a first stage of Sbox circuitry;

means for identifying duplicated bits output by the expansion circuitry, wherein duplicated bits correspond to bits in a first bit sequence that are replicated to produce a second bit sequence;

means for providing duplicated bits and the output of the first stage of Sbox circuitry to a second stage of Sbox circuitry, wherein duplicated bits are provided to the second stage of Sbox circuitry after nonduplicated bits are provided to the first stage of Sbox circuitry.

19. The cryptography accelerator of claim 18, wherein the Sbox circuitry comprises three stages.

20. The cryptography accelerator of claim 19, wherein the first stage of Sbox circuitry is a case statement.

21. The cryptography accelerator of claim 19, wherein the first stage of Sbox circuitry performs a table lookup.

22. The cryptography accelerator of claim 19, wherein the first stage of Sbox circuitry is a multiplexer.

23. The cryptography accelerator of claim 22, wherein the second stage of Sbox circuitry is a multiplexer.

24. The cryptography accelerator of claim 18, wherein bits **3** and **4** of a 6 bit block are provided to the first stage of Sbox circuitry.

25. A cryptography accelerator for performing cryptography operations, the cryptography accelerator comprising:

a DES engine;

a frequency synchronizer coupled to the DES engine;

surrounding logic coupled to the DES engine and the frequency synchronizer, wherein the DES engine operates at a first clock rate and the surrounding logic operates at a second clock rate different from the first clock rate.

26. The cryptography accelerator of claim 25, wherein the first clock rate is faster than the second clock rate.

27. The cryptography accelerator of claim 25, wherein the frequency synchronizer signals the DES engine to begin performing cryptography operations.

28. The cryptography accelerator of claim 25, wherein the frequency synchronizer signals the surrounding logic to read processed data from the DES engine after cryptography operations are performed.

29. The cryptography accelerator of claim 25, wherein the frequency synchronizer uses a reference clock associated with the surrounding logic to synchronize a 1×clock and a higher multiple clock associated with the DES engine.

30. The cryptography accelerator of claim 25, wherein the negative edge of the 1×clock is used to catch a start signal associated with the reference clock to allow consideration of skew.

31. The cryptography accelerator of claim 25, wherein the second positive edge of the 3×clock is used to catch the start signal

* * * * *