

(19) World Intellectual Property Organization
International Bureau



(43) International Publication Date
9 April 2009 (09.04.2009)

PCT

(10) International Publication Number
WO 2009/043918 A2

(51) International Patent Classification: Not classified

(21) International Application Number:
PCT/EP2008/063259

(22) International Filing Date: 2 October 2008 (02.10.2008)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/977,036 2 October 2007 (02.10.2007) US

(71) Applicant (for all designated States except US): **INTERUNIVERSITAIR MICROELECTRONICA CENTRUM vzw** [BE/BE]; Kapeldreef 75, B-3001 Leuven (BE).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **PRIEWASSER, Robert** [AT/AT]; Villacher Strasse 63/1, A-9020 Klagenfurt (AT). **BOUGARD, Bruno** [BE/BE]; rue Jean Rey 8, B-1370 Jodoigne (BE). **NAESENS, Frederik** [BE/BE]; Cederstraat 3, B-8800 Roeselare (BE).

(74) Agent: **BIRD, William E.**; Klein Dalenstraat 42A, B-3020 Winksele (BE).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

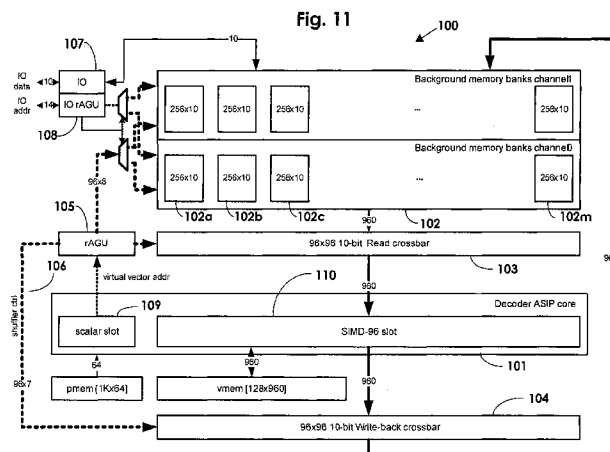
Declaration under Rule 4.17:

— of inventorship (Rule 4.17(iv))

Published:

— without international search report and to be republished upon receipt of that report

(54) Title: AN ASIP ARCHITECTURE FOR DECODING AT LEAST TWO DECODING METHODS



(57) Abstract: The present invention provides a system for execution of a decoding method, the system being capable of executing at least two data decoding methods which are different in underlying coding principle, whereby at least one of said data decoding methods requires data shuffling operations on said data. A system according to embodiments of the present invention comprises: - at least one application specific processor having an instruction set comprising arithmetic operators excluding multiplication, division and power, the processor being selected for execution of approximations of each of said at least two data decoding methods, - at least a first memory unit, e.g. background memory, for storing data, - a transfer means for transferring data from the first memory unit towards said at least one programmable processor, said transfer means including a data shuffler, and - a controller for controlling the data shuffler independent from the processor.

An ASIP architecture for decoding at least two decoding methods

Technical field of the invention

The present invention relates to high-speed and low power data decoding in communication systems, and to decoders providing such decoding. The present invention relates to semi-automated methods for exploration of data decoding methods, to approximations for data decoding methods and/or instruction sets delivered by such methods, to ASIPs capable of executing instructions of such instruction sets, to systems for execution of decoding methods and to methods for executing decoding methods.

Background of the invention

In digital communication systems, reliable transmission is achieved by means of channel coding, a class of Forward Error Correction (FEC) techniques. Coding the information means adding redundancy to the bit stream at the transmitter side, so that it can be properly reproduced at the receiver side. Nowadays, mobile devices are integrating an increasing variety of wireless communication and connectivity standards, each depicting a multitude of operation modes. This diversity, combined with the increasing cost of semiconductor, e.g. silicon, implementation claims for flexible implementations wherever possible.

A Software Defined Radio (SDR) system is a radio communication system where components, such as e.g. mixers, filters, amplifiers, modulators/demodulators, detectors etc., that have typically been implemented in hardware are instead implemented using software on a processor of a personal computer or other embedded computing device. Significant amounts of signal processing are handed over to the processor, rather than being done using special-purpose hardware. The Tier-2 SDR approach, where the whole baseband functionality is run on a programmable architecture, is an attractive way to obtain the above flexibility. Great advances were recently booked in SDR for mobile applications. Multiple SDR processors have been proposed, covering most of the baseband processing with satisfactory performance and

energy efficiency. In some cases, forward error correction is implemented on the same processor as the rest of the baseband.

More recent developments have been tackling high throughput communication standards such as IEEE 802.11n and 3GPP LTE. However, as it typically depicts a magnitude higher computation load, forward error correction (FEC) has been excluded from the scope of most of these high throughput solutions and lead to separate dedicated hardware accelerators. Lots of research is being carried out about efficient implementation of such hardware accelerators.

- 10 The number of advanced FEC options for broadband communication standard is, however, also growing rapidly. Besides 3GPP where convolutional turbo codes are used for a while, the IEEE adopted LDPC coding schemes as optional for the third WLAN generation (IEEE 802.11n) as well as for fixed broadband wireless access (IEEE 802.16a) and their mobility extension (IEEE 15 802.16e). IEEE 802.16 also consider convolutional and block turbo codes as other optional schemes. Advanced forward error correction is also getting popular in broadcast standards. For instance, LDPC is used in DVB-S2/T2. Although these applications have a lower channel throughput, high throughput implementations are still required to enable multi-channel viewing and 20 recording in set-top-boxes.

- Flexibility is hence required for advanced FEC too. To address this need, recent contributions were looking at application specific instruction set processor (ASIP) implementations. These solutions enable running different kind of turbo codes as well as Viterbi decoding on the same processor. In case 25 of LDPC, advances were presented in offering flexibility by combining checknode processing with an interconnection network.

However, no application specific architecture has been presented yet that allows one to map both turbo and LDPC decoding at high throughput, with full memory architecture sharing, full data-path reuse and high energy efficiency.

30 **Summary of the invention**

It is an object of embodiments of the present invention to present the implementation of an application specific programmable architecture

addressing in a unified way the emerging turbo- and LPDC coding requirements of 3GPP-LTE, IEEE802.11n, IEEE802.16(e). It is an advantage of embodiments of the present invention that extension to DVB-S2 is also possible.

- 5 The above objective is accomplished by a device and a method according to the present invention.

In one aspect, the present invention provides a semi-automated method for exploration of data decoding methods whereby at least one of said data decoding methods requires data shuffling operations on said data. The method according to embodiments of the present invention comprises:

- 10 - loading at least two decoding methods which are different in underlying coding principle, such as for example one type of LDPC decoding method and one type of turbo decoding method,
- loading a template of an ASIP device having a memory structure and an
15 ASIP core with a pre-defined instruction set,
- changing the codes of each of the decoding methods into one or more fixed point approximations making use of the pre-defined instruction set,
- selecting for each of the decoding methods one approximation and scheduling of the data accesses to the memory structure, the selecting and the
20 scheduling being by evaluating the performance of the execution of each of said approximations on the device, the performance including throughput and power consumption by the ASIP core and the power consumption due to data accesses to said memory structure, and
- delivering the selected approximations of said at least two decoding methods
25 and providing the selected scheduling.

In a semi-automated method according to embodiments of the present invention, the approximations may comprise arithmetic operations excluding multiplication, division and power. Hence only simple operations are used, no complex operations as floating point operations, multiplication, division, power.

- 30 The arithmetic operations may comprise instructions selected from the group consisting of (max, min, sum, subtraction).

A method according to embodiments of the present invention may furthermore comprise exploration of different templates of an ASIP device. This involves

first loading a first template of an ASIP device having a first memory structure and a first ASIP core with a pre-defined first instruction set, changing the codes of each of the decoding methods into one or more fixed point approximations making use of the pre-defined first instruction set, then loading
5 at least a second template of an ASIP device having a second memory structure and a second ASIP core with a second pre-defined instruction set, at least one of the second memory structure and the second ASIP core being different from the first memory structure and the first ASIP core, respectively, changing the codes of each of the decoding methods into one or more fixed
10 point approximations making use of the pre-defined second instruction set, selecting a template of an ASIP device common for both decoding methods, and selecting for each of the decoding methods one approximation, and scheduling of the data accesses to the memory structure of the selected ASIP device, the selecting and the scheduling being as indicated above, and
15 delivering the selected template of the ASIP device, the selected approximations of said at least two decoding methods and providing the selected scheduling.

In another aspect, the present invention provides an approximation for a data decoding method, obtained by a method according to method embodiments of
20 the present invention.

In another aspect, the present invention provides a semi-automated method for exploration of decoding methods and an instruction set suitable therefor.

The method comprises

- loading at least two decoding methods which are different in underlying
25 coding principle,
- loading a template of an ASIP device with a plurality of possible instruction sets,
- changing the codes of each of the decoding methods into a plurality of fixed point approximations making use of the instruction sets,
- 30 - selecting for each of the decoding methods one approximation and the corresponding instruction set by evaluating the performance of the execution of each of said approximations on the device, the selected instruction set being the same for the at least two decoding methods, and

- delivering the selected approximations of said at least two decoding methods and the corresponding instruction set.

In a further aspect, the present invention provides an instruction set delivered by a method according to an embodiment of the present invention. The

5 Instruction set may comprise arithmetic operators excluding multiplication, division and power. The arithmetic operators may comprise any of $\max^*$ or $\min^+$ (min-sum).

In another aspect, the present invention provides an application specific processor capable of executing instructions of an instruction set according to
10 embodiments of the present invention.

In yet another aspect, the present invention provides a system for execution of a decoding method, the system being capable of executing at least two data decoding methods which are different in underlying coding principle, whereby at least one of said data decoding methods requires data shuffling operations
15 on said data. A system according to embodiments of the present invention comprises:

- at least one ASIP core, i.e. an application specific programmable processor, having an instruction set comprising arithmetic operators excluding multiplication, division and power, the ASIP core being selected for execution
20 of approximations of each of said at least two data decoding methods,
- at least a first memory unit, e.g. background memory, for storing data,
- a transfer means for transferring data from the first memory unit towards said at least one ASIP core, said transfer means including a data shuffler, and
- a controller for controlling the data shuffler independent from the ASIP core.

25 It is not obvious for a person skilled in the art to choose to use an ASIP core for decoding at least two data decoding methods which are different in underlying coding principle, whereby at least one of said data decoding methods requires data shuffling operations on said data, and to at the same time separately control, by a separate controller, the data shuffling. According
30 to embodiments of the present invention, this independency between the ASIP core for decoding and the controller for shuffling is selected on purpose.

According to embodiments of the present invention, the controller may comprise an AGU (address generator unit).

The at least two different data decoding methods may include one or more of LDPC or turbo decoding, e.g. one of convolutional or block turbo decoding.

A system according to embodiments of the present invention may comprise a plurality of ASIP cores. This allows parallelisation of decoding, hence an

5 increase in decoding speed.

A system according to embodiments of the present invention may comprise a plurality of first memory units and a plurality of data shufflers. This is particularly advantageous in case LDPC decoding requiring high parallelisation is to be handled. The transfer means may further comprise a data rotation

10 engine. This is also very advantageous for handling LDPC decoding requiring high parallelisation.

In a system according to embodiments of the present invention, the at least one ASIP core may be adapted by provision of dedicated instructions for executing all the at least two different data decoding methods. This adaptation

15 may comprise providing the ASIP with an instruction set according to embodiments of the present invention.

In a system according to embodiments of the present invention, the ASIP core is adapted so that the at least two different data decoding methods use the same datapath.

20 In a further aspect, the present invention provides the use of a system with at least one programmable ASIP core with dedicated instructions for executing a group of decoding methods which are different in underlying coding principle, said executions at least exploiting all the same of said at least one ASIP core. The group of decoding methods may at least comprise turbo decoding

25 methods and LDPC decoding methods.

In yet another aspect, the present invention provides a method for decoding data. The method comprises

- loading an approximation of a decoding method,
- loading coded data,

30 - decoding the data by means of the approximation,
wherein decoding the data comprises executing arithmetic operations on the coded data by means of a programmable ASIP core, and controlling

scheduling of data accesses to a memory structure by a controller separate from the ASIP core.

The loaded approximation may be an approximation according to embodiments of the present invention.

- 5 Decoding the data may comprise executing arithmetic operations on the coded data by means of a plurality of parallel ASIP cores. Controlling scheduling of data accesses may comprise controlling of shuffling of data from a memory structure of the ASIP by a controller separate from the ASIP core. The controller may be adapted for controlling data accesses so as to be collision-
- 10 free. This increases the throughput of the system.

- In a further embodiment, the present invention provides a programmable processor, an ASIP core, capable of executing at least two decoding methods, approximations of said decoding methods being obtained by a method according to embodiments of the present invention, wherein the ASIP core
- 15 stores dedicated instructions for said at least two decoding methods.

- It is an advantage of embodiments of the present invention that a 100+ Mbps throughput is achieved while providing significant area benefits when compared to the combination of dedicated turbo and LDPC solutions.

- Particular and preferred aspects of the invention are set out in the
- 20 accompanying independent and dependent claims. Features from the dependent claims may be combined with features of the independent claims and with features of other dependent claims as appropriate and not merely as explicitly set out in the claims.

- The above and other characteristics, features and advantages of the present
- 25 invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, which illustrate, by way of example, the principles of the invention. This description is given for the sake of example only, without limiting the scope of the invention. The reference figures quoted below refer to the attached drawings.

30 **Brief description of the drawings**

Fig. 1 is a schematic overview of the turbo decoding paradigm.

Fig. 2 compares sliding-window MAP dataflow (b) and baseline MAP dataflow (a).

Fig. 3 illustrates the operation of a parallel MAP decoder with parallel windows.

Fig. 4 illustrates the H matrix (top), bipartite graph (middle) and parity check

5 equations defining an LDPC code.

Fig. 5 illustrates LDPC Layered Decoding.

Fig. 6 is a graph illustrating BER performance for a quantized layered-belief-propagation-based decoding scheme.

Fig. 7 illustrates convergence speed for a quantized layered-belief-

10 propagation-based decoding scheme.

Fig. 8 is a graph illustrating BER performance for a quantized layered-turbo-decoder-based scheme.

Fig. 9 illustrates convergence speed for a quantized layered-turbo-decoder-based scheme.

15 Fig. 10 is a block-schematic overview of the semi-automated toolflow for the definition of the decoder architecture based on decoding algorithmic input.

Fig. 11 is a block-schematic overview of a decoder architecture of a processor with a 96-slot SIMD decoding engine according to embodiments of the present invention.

20 Fig. 12 depicts an example of the usage of a transparent background memory through rAGUs according to embodiments of the present invention.

Fig. 13 illustrates an amended decoder architecture adapted for DVB-S2 support combining the previously defined architecture with a wide rotation engine according to embodiments of the present invention.

25 Fig. 14 is a bloc-schematic illustration of the decoder core architecture according to embodiments of the present invention.

Fig. 15 is a table showing an instruction set according to an embodiment of the present invention.

30 Fig. 16 illustrates data flow and memory organisation for efficient decoding of LDPC-codes when using an optimised min-sum approximation.

Fig. 17 illustrates an interleaving pattern for example I, where in part a, an exemplary architecture-aware parity-check matrix is shown, in b, the cyclic permutation pattern for this matrix and in c, the message organisation for

reading and writing is shown, when exploiting the inherent parallelism of the code.

Fig. 18 illustrates an interleaving pattern for example II, where in part a, an exemplary architecture-aware parity-check matrix is shown, in b, the cyclic permutation pattern for this matrix and in c, the message organisation for reading and writing, when not exploiting the inherent parallelism of the code.

Fig. 19 illustrates a counter arrangement to allow efficient SIMD mapping of interleaved APP messages.

Fig. 20 illustrates an efficient check node to bit node message storage. In a), the messages of one check node are shown in packed form. In b), the unpacked form is depicted.

Fig. 21 illustrates a check node processing unit. Computation is split up in two stages. First, the forward recursion computes the signs and magnitudes of the check node messages in forward direction. Additionally, the two smallest values are stored. In the second stage, the backward and output recursion, update

Fig. 22 illustrates an exemplary memory read and write back organisation. In the different figures, the same reference signs refer to the same or analogous elements.

20 Description of illustrative embodiments

The present invention will be described with respect to particular embodiments and with reference to certain drawings but the invention is not limited thereto but only by the claims. The drawings described are only schematic and are non-limiting. In the drawings, the size of some of the elements may be exaggerated and not drawn on scale for illustrative purposes. The dimensions and the relative dimensions do not correspond to actual reductions to practice of the invention.

Furthermore, the terms first, second, third and the like in the description and in the claims, are used for distinguishing between similar elements and not necessarily for describing a sequence, either temporally, spatially, in ranking or in any other manner. It is to be understood that the terms so used are interchangeable under appropriate circumstances and that the embodiments of

the invention described herein are capable of operation in other sequences than described or illustrated herein.

It is to be noticed that the term “comprising”, used in the claims, should not be interpreted as being restricted to the means listed thereafter; it does not

5 exclude other elements or steps. It is thus to be interpreted as specifying the presence of the stated features, integers, steps or components as referred to, but does not preclude the presence or addition of one or more other features, integers, steps or components, or groups thereof. Thus, the scope of the expression “a device comprising means A and B” should not be limited to
10 devices consisting only of components A and B. It means that with respect to the present invention, the only relevant components of the device are A and B. Reference throughout this specification to “one embodiment” or “an embodiment” means that a particular feature, structure or characteristic described in connection with the embodiment is included in at least one
15 embodiment of the present invention. Thus, appearances of the phrases “in one embodiment” or “in an embodiment” in various places throughout this specification are not necessarily all referring to the same embodiment, but may do so. Furthermore, the particular features, structures or characteristics may be combined in any suitable manner, as would be apparent to one of ordinary
20 skill in the art from this disclosure, in one or more embodiments.

Similarly it should be appreciated that in the description of exemplary embodiments of the invention, various features of the invention are sometimes grouped together in a single embodiment, figure, or description thereof for the purpose of streamlining the disclosure and aiding in the understanding of one
25 or more of the various inventive aspects. This method of disclosure, however, is not to be interpreted as reflecting an intention that the claimed invention requires more features than are expressly recited in each claim. Rather, as the following claims reflect, inventive aspects lie in less than all features of a single foregoing disclosed embodiment. Thus, the claims following the detailed
30 description are hereby expressly incorporated into this detailed description, with each claim standing on its own as a separate embodiment of this invention.

Furthermore, while some embodiments described herein include some but not other features included in other embodiments, combinations of features of different embodiments are meant to be within the scope of the invention, and form different embodiments, as would be understood by those in the art. For
5 example, in the following claims, any of the claimed embodiments can be used in any combination.

Furthermore, some of the embodiments are described herein as a method or combination of elements of a method that can be implemented by a processor of a computer system or by other means of carrying out the function. Thus, a
10 processor with the necessary instructions for carrying out such a method or element of a method forms a means for carrying out the method or element of a method. Furthermore, an element described herein of an apparatus embodiment is an example of a means for carrying out the function performed by the element for the purpose of carrying out the invention.

15 In the description provided herein, numerous specific details are set forth. However, it is understood that embodiments of the invention may be practiced without these specific details. In other instances, well-known methods, structures and techniques have not been shown in detail in order not to obscure an understanding of this description.

20 The invention will now be described by a detailed description of several embodiments of the invention. It is clear that other embodiments of the invention can be configured according to the knowledge of persons skilled in the art without departing from the technical teaching of the invention as defined by the appended claims. The detailed description starts off with an overview of
25 the turbo decoding scheme and the LDPC decoding scheme, as two particular decoding schemes which can be used in an architecture according to the present invention. Of all practical error correction methods known to date, turbo codes and low-density parity-check codes (LDPC) come closest to approaching the Shannon limit, the theoretical limit of maximum information
30 transfer rate over a noisy channel.

TURBO DECODING ALGORITHM EXPLORATION

Introduction

In coding theory, concatenated codes form a class of error-correcting codes which are derived by combining an inner code and an outer code.

- 5 Turbo codes are a class of high-performance error correction codes which are finding use in deep space satellite communications and other applications where designers seek to achieve a maximal information transfer over a limited-bandwidth communication link in the presence of data-corrupting noise. Turbo codes, although not concatenated codes, do typically also utilise two different
- 10 error correction codes in parallel.

What distinguishes Turbo codes from concatenated codes and what makes them so powerful, is the combination of three concepts:

- Combination of codes,
- Separate decoding of constituent codes with exchange of
- 15 information, and
- Iterative decoding.

A schematic overview of these three concepts into the turbo decoding paradigm is depicted on Fig. 1.

- 20 Turbo codes make it possible to increase data rate without increasing the power of a transmission, or they can be used to decrease the amount of power used to transmit at a certain data rate.

- A turbo encoder 10 provides three sub-blocks of bits. The first sub-block is an k-bit block of payload data. The second sub-block comprises $N-k/2$ parity bits for the payload data, computed e.g. using a recursive systematic convolutional code (RSC code). The third sub-block also comprises $N-k/2$ parity bits, but for a known permutation of the payload data, again computed e.g. using an RSC
- 25 convolutional code. This means that two redundant but different sub-blocks of parity bits are provided for the sent payload. The complete block sent has N bits of data with a code rate of k/N . In order to obtain this, a first encoder sub-block 11 and a second encoder sub-block 12 are provided. The permutation of
- 30 the payload data is carried out by a device called interleaver 13.

The decoder 14 is built in a similar way as the above encoder 10: two elementary sub-decoders 15, 16 are interconnected to each other, but in serial

way, not in parallel. The first sub-decoder 15 operates on lower speed, and is intended for the first encoder sub-block 11, and the second sub-decoder 16 is intended for the second encoder sub-block 12.

To represent the messages exchanged between the two sub-decoders 15, 16, it is not feasible anymore to only take the binary information, whether the received symbol is a '0' or a '1' (hard-decision). Not only the polarity of a bit is stored, but also a measurement of how confident this decision would be (soft-decision). Furthermore, these measurements are expressed as probabilities equal to ratios of the probability for the bit being one or zero. For the uncoded bits to be '1' or '0', under the condition that the sequence y was received, this a posteriori ratio can be written as

$$\frac{Pr(u_k = 1 | y)}{Pr(u_k = 0 | y)}.$$

The sequence y is also denoted as $y^N_1 = y_1, y_2, \dots, y_N$. The task of the decoder 14 is to estimate these a posteriori probabilities, where the received sequence $y = y^N_1$ consists of $d = d^k_1$ information bits (systematic part of the payload), $c^{(N-k)/2}_1$ control bits from the first constituent code, and $c^{(N-k)/2}_1$ control bits from the second constituent code. The first sub-decoder 15 thus yields a soft decision which causes a first delay, corresponding to a same delay caused by a delay line in the encoder 11. The operation of the second sub-decoder 16 causes a second delay. An interleaver 17 installed between the first and second sub-decoders 15, 16 is used to scatter error bursts coming from the first sub-decoder output.

The BCJR algorithm [described by L. R. Bahl, F. Cocke, F. Jelinek, and J. Raviv, in "Optimal decoding of linear codes for minimizing symbol error rate", IEEE Trans. on Information Theory, pages 284–287, 1974] is an algorithm for maximum a posterior (MAP or posterior mode) decoding of error correcting codes defined on trellises. The symbol-by-symbol MAP decoding, implemented using the BCJR algorithm, performs optimal decoding by maximizing the a posteriori probabilities. In other words, it takes all possibilities of each bit in the received vector y into account and takes a decision for each bit as a last step (it minimizes the symbol, or bit error probability). When illustrated in a trellis diagram, all paths through the diagram need to be computed and considered

for a decision. This forms an optimal decoding scheme, for the cost of complexity. To efficiently decode Turbo codes by using the BCJR algorithm, it needs to be tailored for this need and additional steps must be taken to allow an efficient implementation of a MAP decoder for Turbo codes. With reference to the trellis diagram, the decoder's task is to estimate the a posteriori probabilities for all states S_k and all transitions between these states for a received vector $y^N_1 = \{y_1, y_2, \dots, y_N\}$ with N elements. Each element in the received vector y corresponds to the set of transitions from time k to time $k + 1$, with $k = 0, \dots, N$. The probability for being in a state m equals

$$Pr(S_k = m | y^N_1) = \lambda_k(m)$$

This represents the probability, that under the condition that the sequence y^N_1 was received, the state at time k is $m_{[FN]}$.

The same can be written for the probability of the transition from state m' to m :

$$Pr(S_{k-1} = m', S_k = m | y^N_1) = \sigma_k(m', m)$$

This is the probability, that under the condition that the sequence y^N_1 was received, the transition will be from state m' at time $k-1$ to state m at time k . This needs to be computed for all possible transitions. This computation can be split up in two parts. In the first part, called the forward, or α recursion, the probabilities of being in a specific state, taking all previous transition probabilities into account, are computed for all states. The same is done for the second part, in the backward, or β recursion, where the same probabilities are computed starting from the last state.

Decoding algorithm approximation

The equality of

$$\begin{aligned} \max(x, y) &= \log \left(\frac{e^x + e^y}{1 + e^{-|x-y|}} \right) \\ &= \log(e^x + e^y) - \log(1 + e^{-|x-y|}) \end{aligned}$$

can be used to reduce the implementation complexity, when computing the forward recursion, backward recursion and output metric. The function known in the literature as max-star, or $\max^*$, is defined as

$$\max^*(x, y) = \log(e^x + e^y) = \max(x, y) + \log(1 + e^{-|x-y|})$$

The equations for the forward and backward recursion, as well as the output metric, can be translated in the log-domain to take advantage of the above formulas.

The max*-operation can efficiently be implemented in hardware, using comparators to build the maximum and a look-up table to provide the correction term. For the cost of a slight performance degradation, the correction factor of the max* can be omitted. The approximation then only needs to evaluate the max-function.

Fixed point refinement in state-of-the-art implementation has shown [Alexandre Giulietti, Bruno Bougard, and Liesbet Van der Perre. "Turbo Codes Desirable and Designable", Kluwer Academic Publishers, 2004] that, provided that normalization is applied, a word-width of 10 bits is sufficient for all data nodes in the algorithm.

15 Parallel decoding using approximation

Decoding a turbo code requires iterating maximum a posteriori (MAP) decoding of the constituent codes and (de)interleaving operations. Due to the high computational complexity, achieving high throughput requires a massively parallel architecture. The parallelization of the MAP can be done by computing in parallel multiple "windows" out of the coded block [as described by B. Bougard et al., in "A scalable 8.7 nJ/bit 75.6 Mb/s parallel concatenated convolutional (turbo-) CODEC", ISSCC, Feb. 2003].

The MAP decoding can be conveniently parallelized based on a derivative of the sliding-window-MAP (SW-MAP) algorithm [described by M. Bickerstaff et al., in "A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless", ISSCC, Feb 2003]. In the original sliding window MAP algorithm, the backward recursion is split in N_w smaller recursions, which are applied to N_w sub-blocks of input/a priori data (windows). The corresponding dataflow is illustrated in Fig. 2(b). The state probabilities at the last element of a window is approximated based on a training sequence applied on the input data of the adjacent window. The computation of the backward-recursion training sequence of the traditional SW-MAP algorithm as illustrated in Fig. 2(a) can be avoided by computing the initial state-metric value for the backward recursion

based on the last state-metric value of the adjacent window, at the previous iteration. This is referred to as next iteration initialization. Such state-metric initialization scheme can be applied to the forward recursion too. In the resulting scheme, the windows can be processed independently, which enables parallelization by a factor equal to the number of windows (N_w).

Fig. 3 depicts the operation of a parallel MAP decoder with parallel windows. The top part (a) recalls the classical sliding windows MAP decoding flow. A forward recursion, given by Eq. (1)

$$\alpha'_k(m) = \max_{m'}^* (\alpha'_{k-1}(m') + \gamma'_k(m', m)) \quad \text{Eq. (1)}$$

is applied to the input symbol metrics (γ). Part of the backward recursion (Eq. (2))

$$\begin{aligned} e(d_k) = & \max_{m, m'}^* (\alpha'_{k-1}(m') + \gamma^{11}_k(m, m') + \beta'_k(m)) \\ & - \max_{m, m'}^* (\alpha'_{k-1}(m') + \gamma^{00}_k(m, m') + \beta'_k(m)) - L_{\text{sys}} - L_{\text{apriori}} \end{aligned} \quad \text{Eq. (2)}$$

is started after a window of M inputs is treated by the forward recursion. The recursion is then initialized either using a training sequence applied on the symbol metrics from the adjacent window, or using backward state metrics (β) (Eq. (3))

$$\beta'_k(m) = \max_{m'}^* (\beta'_{k+1}(m') + \gamma'_{k+1}(m', m)) \quad \text{Eq. (3)}$$

from a previous iteration (next iteration initialization, NII). Based on the state metrics output of the forward (α) and backward (β) recursions, next to the input symbol metrics (γ), the MAP outputs (extrinsic information, e) can be computed. Using NII is preferred as it leads to more regular data flow. That way, as shown in Fig. 3(b), the MAP can easily be parallelized by computing several windows in parallel.

Besides the MAP, interleaving operations must also be parallelized. As

interleaving means permutating the data out of a memory, parallelizing it requires to use multiple memory banks and to handle potential collision in read and write operations. For this, collision free interleaving patterns or collision-resolution architectures were proposed in [Giulietti, A.; van der Perre, L.; Strum, A., "Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements," Electronics Letters, vol.38, no.5, pp.232-234, 28 Feb 2002]. Later, Tarable proved that any interleaving law can be mapped onto any

parallel multi-banked architecture [Tarable, A.; Benedetto, S.; Montorsi, G., "Mapping interleaving laws to parallel turbo and LDPC decoder architectures," IEEE Transactions on Information Theory, vol.50, no.9, pp. 2002-2009, Sept. 2004]. He also proposed an annealing algorithm that is proven to always

5 converge to a valid mapping function.

In embodiments of the present invention, we are capitalizing on the multi-window MAP parallelization combined with Tarable's interleaving law mapping. A single-instruction multiple-data (SIMD) approach is used. The computation of γ s, α s and β s over the branches or states are serialized but multiple windows
10 are computed in parallel to maximize data-level parallelism.

LDPC DECODING ALGORITHM EXPLORATION

Introduction

Low-density parity-check (LDPC) code is an error correcting code, a method of
15 transmitting a message over a noisy transmission channel. LDPC codes in the class of linear block codes, form the counterpart to turbo codes in the class of convolutional codes. The parity-check matrix representing LDPC codes is a sparse matrix, meaning that there are only a few elements not equal to zero, as illustrated in Fig. 4(a). This property has clear advantages which will be
20 addressed later. LDPC codes with an arbitrarily long block length N are performing very close to the theoretical transmission rate limit.

The parity-check matrix defining an LDPC code can be represented as a bipartite graph, a graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to one in the other set; also
25 called tanner graph or factor graph, see Fig. 4(b). This representation is more convenient for understanding the basic algorithm and data flow, it will therefore be used in later explanations. The graph contains two types of nodes: check nodes c and bit nodes b . Bit nodes are also referred to as variable nodes. Check nodes are corresponding to the rows M of H and therefore each parity-
30 check equation forms one check node. Bit nodes are corresponding to the columns N of H and therewith each bit of the received codeword y is assigned to one column. In Fig. 4(a), the first row of H , for example, forms the check equation c_1 and therewith the check node c_1 , as illustrated in Fig. 4(c). Check

nodes c and bit nodes b are connected by edges, where a check node and a bit node are connected over an edge if the corresponding entry in the parity-check matrix H is equal to 1. Due to the low density or sparseness and to the randomness of the parity-check matrix, there are very limited, but quasi-

5 random connections between nodes. The degrees of check and bit nodes are defined by the number of edges which are connecting them with neighboring bit or check nodes and are denoted d_c and d_v , for check and bit node degree, respectively. An LDPC code is called regular, when the degrees d_c and d_v are constant for all check nodes and all bit nodes. In case of irregular LDPC

10 codes, the degrees are varying among the nodes. The parity-check matrix depicted in Fig. 4(a) represents an irregular code, what can be seen on the imbalanced number of connections between check nodes and bit nodes. Due to the constant check node and bit node degrees d_c and d_v , regular codes are typically easier to encode and enable more structured decoder

15 architectures. Irregular codes however, are outperforming regular ones and are approaching very close to the Shannon bound. These improvements in decoding performance are due to the faster convergence of nodes with higher degrees, which then assist others with lower degrees, by providing them with more reliable information.

20 Decoding of LDPC codes is done iteratively. Messages are passed between nodes, which implements the basic concept of iterative decoding. In the first iteration, no a priori information is available to refine the probabilities of the bit nodes. After each iteration, a more accurate estimate of the received vector y can be provided. Decoding is stopped, when a valid codeword is found.

25 As in the case of turbo codes, LDPC decoding can be expressed and approximated in the logarithmic domain. For the check node computation, the bit node computation, as well as for the output computation which involve a large amount of multiplications, this transformation has clear advantages. Furthermore, for every bit node message and every check node message, two

30 corresponding probabilities need to be stored, which becomes unpractical for large block length. Consequently, the same approach as applied to the Viterbi and BCJR algorithms, a representation in the logarithmic domain has several

advantages and will therefore be highlighted for LDPC codes in the next paragraph.

Decoding algorithm approximation

- 5 The multiplicative nature of the LDPC decoding algorithm in the probabilistic domain and the need for the non-linear function in the logarithmic domain makes it impractical for hardware implementation. Feasible, reduced-complexity approximations must be found, to enable efficient, low-power and high-throughput implementations. The throughput and decoding complexity of
- 10 LDPC codes depend on five major parameters:

1. The length N of the codeword (block length),
2. The ratio of number of information bits to block length (code rate),
3. The complexity of the computations at each check node,
4. The complexity of the interconnect network between check and bit nodes,
- 15 and
5. The number of needed iterations to produce a valid codeword.

In order to tackle these five bottlenecks, optimizations at different levels need to be performed. In order to achieve sufficient bit error ratio (BER, the ratio between the number of incorrect bits transmitted to the total number of bits

20 transmitted) performance, the block length needs to be held at a predetermined length. The code rate needs to be adapted to the environment in order to guarantee reliable communication and can therefore not be increased as desired.

- For the three remaining bottlenecks, namely the check node and interconnect
- 25 network complexity, as well as the number of iterations, solutions or partial solutions have been proposed in the literature and will be presented in the following order. First the min-sum ($\min^+$) algorithm [as described by Marc P. C. Fossorier, Miodrag Mihaljevic, and Hideki Imai, in "Reduced complexity iterative decoding of low-density parity check codes based on belief
- 30 propagation", T-COMM, 47(5):673–680, 1999; and by Farhad Zarkeshvari and Amir Banihashemi, in "On implementation of minsum algorithm for decoding low-density parity-check (LDPC) codes", Technical report, Dept. of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive,

Ottawa, Canada K1S5B6, 2002] can be mentioned, approximating the sum-product algorithm, as presented in [Robert G. Gallager, "Low-density parity-check codes", IRE Transactions on Information Theory, pages 21–28, 1962]. This optimization reduces the check node complexity. On the other hand, the

5 exploitation of the block-based structure of a sub-class of LDPC codes can be exploited in the layered belief-propagation decoding scheme [D. Hocevar, "A Reduced Complexity Decoder Architecture via Layered Decoding of LDPC Codes", IEEE Workshop on Signal Processing Systems, pages 107-112, 2004], which reduces the interconnect overhead, as well as the number of

10 iterations needed. Finally, the turbo decoder-based message passing algorithm [Mohammad M. Mansour and Naresh R. Shanbhag, "Turbo decoder architectures for low-density parity-check codes", Technical report, iCIMS Research Center, Co-ordinated Sciences Laboratory, University of Illinois at Urbana-Champaign, IL 61801, 2002], which enables the exploitation of

15 potential hardware reuse in case the decoder architecture may be used for turbo codes and offers an efficient way to calculate check node messages, in case of decoding LDPC codes.

Parallel LDPC decoding using approximation

20 Considering a LDPC decoding, previous work has shown how layered decoding, which reduces the number of iterations needed to converge, can be efficiently implemented on a SIMD architecture. In layered decoding [D. Hocevar, "A Reduced Complexity Decoder Architecture via Layered Decoding of LDPC Codes", IEEE Workshop on Signal Processing Systems, pages 107-

25 112, 2004] the parity check matrix H of size $(n-k) \times n$, where n equals the total codeword size and k equals the number of data bits. needs to be decomposable into m arbitrary chosen sub-matrices with size $z \times n$, with z equal to $(n-k)/m$, also called super-codes (Fig. 5). For every super-code, the position of 1s is restricted such that, per column in a super-code, only zero or

30 one non-zero element is allowed. For the standards in scope this property is automatically met, through the fact that the sub-matrices within the super-codes are either zero matrices or cyclic permuted identity matrices. This is not the case with the DVB-S2 code. However, in this case, the parity check

equations can be reordered (as well as the received parity metrics respectively) in order to define a layered matrix satisfying the aforementioned property.

Within a layer, the check node and corresponding bit node updates can then
 5 be done in parallel since there is only one edge from every bit node towards a check node in a layer. Intrinsically this enables z parallel processing threads each performing check node updates and message node updates corresponding to a row within the layer of the parity check matrix. As the same operation has to be applied at the same time in z threads, SIMD
 10 implementation is again very efficient.

Performance results

From the previously mentioned decoding algorithm approximation techniques, both the layered-belief-propagation-based (using the min-sum approximation)
 15 and the turbo-based decoding (using the max* approximation) are taken into account for performance comparison. The performance comparison is based on BER performance and on required iterations using a fixed point refined code.

Due to the lack of floating-point arithmetic support in hardware of embodiments
 20 of the present invention, which would lead to a significant area overhead, a more feasible fixed-point number representation is chosen.

A fixed-point scheme, where a virtual decimal point is introduced, enables an efficient way to work with fractional values. This representation is preferable for hardware implementations, in terms of area overhead and energy efficiency.

25 Denote as $Z_{I,Q}$ the fixed-point counterpart of a floating-point value Z , where I and Q digits are used for the integer and fractional part of the value, respectively. The range which can be represented by this quantized value is given by

$$\left[-2^{I-1}, +2^{I-1} - 2^{-Q} \right] \quad \text{Eq. (4)}$$

30 where the Most Significant Bit (MSB) is used to store the sign of the message. The resolution of the fixed-point number is then given by

$$\epsilon = 2^{-Q}$$

The overall word length, needed to represent a number $Z_{l,Q}$, is given by $l+Q$.

The Log-Likelihood Ratio (LLR) messages to be represented in fixedpoint arithmetic, can take values in the range of $[-\infty, \infty]$. In their fixedpoint depiction, this dynamic range is saturated to the bounds shown in Eq. (4). Overflows

5 occurring during fixed-point arithmetic operations are avoided by applying a saturation scheme. The quantization of the messages involved in the decoding process, has been split up in three parts. First, the input messages passed to the decoder are assumed to be represented by $Z_{l,Q}$, e.g. $Z_{4,2}$. This results in an overall word length of 6 bits, where Q bits, e.g. 2 bits, are spent for the

10 resolution of the message and l bits, e.g. 4 bits for the integer part. Two bits for the fractional part of a message leads to a resolution of $Q = 2^{-2} = 0.25$. In practical systems, a quantization width of 3 to 6 bits for the soft-values received from the channel, has shown to be accurate enough to deliver near-optimum performance.

15 Secondly, the data path and therewith the internal message representation has been varied for different experiments, where the word length ranges from 11 bits down to 7 bits, with a resolution held at 2 bits each.

Finally, a normalization factor γ has been introduced for all check to bit node messages ($\mu_{cj \rightarrow bi}$). See Eq. (5) for the min-sum approximation and Eq. (6) for

20 the max $\otimes$ -based approximation.

$$L(\mu_{cj \rightarrow bi}) \approx \gamma \cdot \min_{r \in R(j) \setminus i} |\beta_{jr}| \cdot \prod_{r \in R(j) \setminus i} \alpha_{jr}, \quad \text{Eq. (5)}$$

with $0 < \gamma \leq 1$.

$$\Lambda_i = \gamma \cdot q(\Delta\alpha_{i-1}, \Delta\beta_i) \quad \text{Eq. (6)}$$

25 Comparison of BER performance for the quantized layered-belief-propagation-based decoding scheme in function of the received energy per bit normalized with the noise power (E_b/N_0). can be found on Fig. 6. The algorithms are implementing the min-sum approximation (denoted as layer-belief-propagation-based in contrary to the optimal layer-belief-propagation method) and are

30 quantized with $Z_{8,2}$, $Z_{6,2}$ and $Z_{5,2}$ for the data path. The input values, obtained from the channel, are quantized with $Z_{4,2}$. It is to be noted that a normalization

factor has been introduced for all check to bit node messages. The convergence speed for these modes is depicted on Fig. 7.

What needs to be noticed here is that when comparing the quantized implementation of the min-sum approximation with the one in floating-point, the quantized version delivers the same BER with an required receive energy which can be reduced by a half dB. Such a positive influence of quantization is very rarely observed. Similar effects have been observed in [Farhad Zarkeshvari and Amir Banihashemi, "On implementation of minsum algorithm for decoding low-density parity-check (LDPC) codes", Technical report, Dept. of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, Canada K1S5B6, 2002], where quantization in general also degraded the performance, but clipping could provide improvements compared to the ideal implementation.

Comparison of BER performance for the quantized layered-turbo-decoder-based decoding scheme is depicted on Fig. 8. The algorithm is implementing the max*-based approximation and is quantized with $Z_{9.2}$, $Z_{8.2}$ and $Z_{7.2}$ for the data path. The input values, obtained from the channel, are quantized with $Z_{4.2}$. The convergence speed for the quantized layered-turbo-decoder-based decoding is depicted on Fig. 9.

Even though the max*-based approach delivered near-optimum decoding performance with largely reduced computation complexity when implemented in floating-point, its quantized version involves a performance loss of around half of a dB. The min-sum approximation on the other hand, shows inverted behavior, the quantized version delivers better simulation results than the floating-point implementation.

Comparing the convergence speed, the quantized version of the min-sum algorithm also beats the max*-based approach, even if this could not be foreseen when looking at the floating-point implementations. There, the max* operates very close to the optimum, due to the very accurate approximation of the non-linear function. When comparing Fig. 7 and Fig. 9 at E_b/N_0 point of 1.5 dB, the min-sum approximation needs around 10 iterations less to converge, compared to the max*-based one (both in the fractional format). This can make a significant difference in terms of throughput.

UNIFIED ARCHITECTURE CONCEPT

While the LDPC decoder architecture is heavily dependent on the parity-check matrix structure, the Turbo decoder architecture is dependent on the trellis structure representing the encoder. This makes it very challenging to develop a decoder which is flexible enough to handle LDPC codes and Turbo Codes for all desired communication standards. More generally speaking, it is an object of embodiments of the present invention to provide a decoder which is flexible enough to handle at least two decoding methods which are different in underlying coding principles. Generally speaking, a tradeoff must be found between the compatibility to different code classes (decoding methods which are different in underlying coding principles), the decoder architecture efficiency and the performance in terms of area and energy. This trade-off can be achieved according to embodiments of the present invention by developing an instruction set processor with a unified instruction set, suitable for the at least two decoding schemes.

Depending on the similarities of the core computations needed for the different decoding methods, can this offer a feasible solution.

The discussion hereinafter will focus on the LDPC-specific requirements of a unified decoder architecture according to embodiments of the present invention which will be developed by applying an ASIP-like design methodology. The unified instruction set according to embodiments of the present invention is generally designed for decoding different algorithms for LDPC codes, but can also be applied for decoding Turbo Codes.

Before considering the architectural part of the decoder implementation, where the layered scheme for architecture-aware codes is applied, care must be taken of the fixed-point number representation used for incoming messages and internal computations and storage.

ASIP design methodology

Developing a general purpose processor would offer the maximum level of flexibility, but due to the high-throughput requirements this approach is not applicable. An ASIC design, which would allow very high-throughput

implementations, lacks the flexibility desired for handling at least two decoding methods which are different in underlying coding principles. According to embodiments of the present invention, an ASIP is used, which forms a feasible trade-off in terms of throughput, flexibility and energy efficiency. Hence the architecture development will follow an ASIP-like methodology to derive the unified decoder architecture according to embodiments of the present invention.

The first task to perform when developing an ASIP architecture, is an extensive algorithm exploration and analysis to identify the main computation kernels.

The algorithm exploration will be concluded by performing a fixed-point refinement of the selected decoding algorithms.

The knowledge of the decoding algorithms enables one to define an instruction set suitable for efficient computation of the main kernels. Defining the instruction set of a processor architecture, trades the flexibility and achievable throughput against silicon area and energy efficiency. Hence the selection, which basic operations should be dedicated to an instruction, must carefully be carried out.

With the optimized instruction set at hand, the achievable throughput can be evaluated. By choosing the right amount of parallelism, for instance by applying a Single-Instruction Multiple-Data (SIMD)-based computation scheme, the throughput can be adapted to the system requirements. Limited, of course, by the inherent parallelism given by the targeted algorithms.

The level of parallelism implies the definition of the data path width, which has a direct impact on the memory organization. According to embodiments of the present invention, the memory hierarchy should be properly adjusted, to avoid bottlenecks due to memory latencies.

As a final step, the overall decoding latency of the architecture can be controlled by introducing a pipelining scheme.

The performance number as depicted on Fig. 6, Fig. 7, Fig. 8 and Fig. 9, illustrates that there are trade-offs that can be made in decoding both turbo codes as LDPC. In order to unify multiple decoding methods, such as for example, but not limited thereto, LDPC and turbo decoding, resulting in a novel architecture, a semi-automated toolflow was established according to

embodiments of the present invention. The result of the toolflow is depicted on Fig. 10. As input of the process an algorithmic description, e.g. in matlab or C code, of the decoding algorithm is required, and this for the plurality of decoding algorithms to be combined in the architecture, next to the

5 specification of clock frequency and maximum parallelization.

During the exploration phase the different algorithms specified, with their possible approximations and fixed point refinements, are evaluated for a number of outputs. Some of these outputs are: BER performance, required number of iterations, throughput, latency and power. It is to be noted that the

10 power number at this stage can only be used for a relative comparison.

Based on the output generated in the exploration phase, in the selection phase a selection of the different decoding flavours for which support needs to be provided, can be selected.

During the generation phase, both memory subsystem and processor

15 definition are derived. The memory subsystem includes the memory bank structure combined with interconnect and permutation logic, in the form of combined shuffler and rotation engines. Additionally matlab scripts are generated for both initializations of memory banks as translations for virtual address mappings.

20 The processor definition includes the instruction set list and a template that can be used as input in simulation software, e.g. Coware Lisatek toolsuite. This toolsuite can generate RTL and high-level simulation model that can be used SystemC model.

25 Conceptual overview

A general overview of a decoder architecture 100 according to embodiments of the present invention is shown in Fig. 11. This is a unified application specific instruction set processor architecture template that efficiently fits at least two data decoding methods, e.g. both turbo and LDPC decoding. In the following,

30 the detailed description is oriented towards a combination of at least turbo decoding methods and LDPC decoding; however, this is only for ease of explanation, and the present invention is not limited thereto.

The proposed architecture 100 combines a multi-slot SIMD datapath 101 (implementing dedicated instructions to support both LDPC and turbo decoding) with a distributed background memory structure 102. The background memory structure 102 is such that there are as many single-ported
5 memory banks 102a, 102b, 102c, ... 102m as SIMD slots. These are connected to the core 101 with a read cross-bar 103 and a write-back cross-bar 104.

An idea of embodiments of the present invention is to hide the shuffling needed by the decoding methods, e.g. the scrambling needed by the LDPC
10 and the interleaving required by the turbo-decoding, using reconfigurable address generation units (rAGU) 105. The core SIMD program, written in a low-level programming language such as assembly code, assumes a virtual address space where collisions never happen. The collision free operation allows a continuous mode decoding operation, without delay in data accesses.
15 This virtual address space is mapped to physical background memory addresses in the background memory structure 102, whereby the read and write-back crossbar control 106 uses the rAGUs 105. The rAGUs 105 may be configured with look-up tables (LUT). In case the rAGU needs to drive a rotation engine, the complete LUT could be replaced by a single value e.g. for
20 the first slot an address generators for the other slots, by incrementing the first value.

The detailed decoder architecture according to embodiments of the present invention is depicted in Fig. 11.

A two channel implementation of the background memory structure 102 as
25 illustrated in Fig. 11 allows streaming mode or dual channel processing, but is not required in all embodiments of the present invention. Due to the dual channel implementation of the background memory structure 102, data in one memory channel can be processed while the next set of data is read into the other memory channel. Fig. 11 depicts an architecture with a 96-bit wide SIMD
30 processor 101. The width of the processing engine 101 depends on the level of parallelism present in the decoding algorithms that need to be supported. Alternative decoder architecture templates according to embodiments of the present invention exploit higher parallelism.

Systematic stream, coded streams, extrinsic and a priori information in case of turbo decoding; bit-node information in case of LDPC are stored into the background memory structure 102. The data is made available as a virtual linear memory map to the decoder core 101, hiding the bank addresses and data permutations. In embodiments of the present invention, an IO unit 107,
5 combined with an IO rAGU 108 controls the input/output mapping in one memory bank channel while the core rAGU 105 performs virtual to physical address translation to the other memory bank channel (ping-pong buffers). Simulations have shown that a background memory word size of 10-bit leads
10 to performance close to floating-point reference implementation for both turbo as LDPC decoding (independent of the max* or min-sum approximation used). According to embodiments of the present invention, this architecture 100 can be used in different modes.

In turbo mode, the rAGU 105, e.g. the rAGU LUT, implements the data
15 mapping resulting from the collision free mapping of the turbo-code interleaver law. As an example, Tarable's annealing algorithm [FN2] can be implemented in a tool that generates the rAGU LUT configuration taking as input the interleaver table and the number of parallel slots in the ASIP core used to perform the decoding operations.

20 Hence, received systematic, coded and extrinsic information inputs for each MAP window can be read or written in or from each slot in a collision-free way, either in direct logical order or in interleaved/deinterleaved logical order.

In most of the LDPC modes used within current wireless standards, such as e.g. the 802.16(e) and 802.11n standard, the permutations are rotations. Due
25 to the flexibility provided in the shuffler engine 105, 106 according to embodiments of the present invention, this property is not even required.

An example of the usage of the transparent background memory structure 102 through the rAGUs 105, 108 is depicted on Fig. 12. To operate in streaming mode the input/output can be performed in one memory bank channel 102,
30 during decoding processing on the other. The input/output may be done in serial fashion, where the IO rAGU 108, e.g. the IO rAGU LUT table, specifies the location of both soft information and hard decision output bits.

For decoding processing, a virtual address is provided to the rAGU 105, e.g. the rAGU LUT, which determines addresses for respectively shuffler control 106 and row addresses for memory bank structure 102. Both LUT tables are correlated.

- 5 In case of turbo mode operation, the IO rAGU 108, e.g. IO LUT, is deducted from Tarable's annealing algorithm, while the rAGU 105, e.g. LUT, for decoding operation also requires the interleaving pattern. The decoder core rAGU 105, e.g. LUT, stores address translation both for the direct read / direct write and interleaved read / deinterleaved write phases.
 - 10 For LDPC mode operation, where the parity check matrix consists out of cyclic permuted identity matrices, the structure of the rAGU, e.g. LUT tables, is more uniform. As for the IO rAGU 108, e.g. IO LUT, the memory is filled in linear order, first for row 0 and increasing column, before starting row 1, The rAGU 105, e.g. LUT table, for decoding operation will contain identical row
 - 15 bank addresses for all slots and shuffler addresses will be cyclic shifts depending on the rotation value in the parity check matrix.
- In case support for DVB-S2 LDPC modes needs to be foreseen, the decoder template can be extended to a single bigger processor, e.g. 360 SIMD slot processor, or multiple SIMD processors put in parallel. An example of a Px96
- 20 SIMD slot processor instance is depicted on Fig. 13. P is the number of parallel processors used. In order to have a maximum parallelization of 360, P could be equal to 4. The decoder architecture then comprises P processors each having an N-slot SIMD processor. Besides the SIMD pipeline 111, the processor instances also contain a control unit 112 with allocated program
 - 25 memory 113 and register file 109, vector register file 115 and vector cache.
- The data read from the memory bank 102 is first permuted by the shuffler 103, 104, after which a rotation can be performed over the multiple banks 102 by means of a rotation engine 120. Contrary to the decoding architecture presented in Fig. 11, the vector part of the processor is also equipped with a
- 30 LIFO memory 116. The vector LIFO 116 allows easy storage and access of temporary data elements.

The advantage of the architecture described in Fig. 13, relies on the fact that the P cores can be used separately in case turbo and/or LDPC modes need to

be decoded that require less parallelism than offered with the complete architecture. Since the complexity of the crossbars 103, 104 (also called shufflers) is exponentially increasing with the number of ports, the solution to have P times a 96 shuffler plus a rotation engine 120 over all slots is less
5 expensive (both in power as area) compared to a fully flexible (Px96)-way shuffler (not illustrated).

Datapath

The architecture according to embodiments of the present invention comprises four major parts. First, the processing unit where a configurable amount of parallel processing in a SIMD-based scheme can be applied. Secondly, the APP (A Posteriori Probability) BN (bit node) message memory, also referred to as background memory, which has to be accessed in an interleaved way. Thirdly, the interleaver and deinterleaver to align the APP messages stored in the APP memory. In case of architecture-aware codes, e.g. architecture aware LDPC codes, the interleaving can be performed by simple shift operations, given that the memory word width is sensibly chosen in such a way that no overlapping of memory words or sub-words over sub-block borders occurs. In order to perform the interleaving, the structure of the parity-check matrix must be known. Therefore, a memory initialized with the characteristics of a certain code is available to retrieve the interleaving pattern from. Finally, besides the APP messages, the CN messages need to be stored for each CN.

Unified instruction set

After having investigated the algorithms to decode the codes, e.g. LDPC codes and turbo codes, in floatingpoint and their robustness against fixed-point effects, a choice can be made on which approximations have proven to be feasible to implement. Having made this decision, the basic instruction set needed for decoding the codes can be derived. A unified instruction set for both Turbo Codes and LDPC codes is summarized in the table hereinbelow. The shown instruction set is suitable for all proposed approximations for decoding LDPC codes, since their major differences invoke only a few instructions.

SIMD instructions

add,sub	basic arithmetic operations
and,or,xor	basic logic operations
lsh,lsr	logical shift right/left
asr,asl	arithmetical shift right/left
abs	extract magnitude from message
min	compute minimum of two messages
min ⁺	compute running minimum and sign of message
cs	compare and select operation
max [*]	impl. of log. of sum of exponential functions

vector instructions

vrr,vrl	vector rotate right/left
vpack	pack check node messages in SIMD word
vunpack	extract check node message from SIMD word
vld,vst	load/store a SIMD word from/to memory
vldm,vstm	load/store a SIMD word with modulo addressing
vpush,vpop	store/load a SIMD word to/from stack

scalar instructions

sadd,szub	basic arithmetic operations
sand,sor,sxor	basic logic operations
bz,bnz	conditional jumps (branch zero/branch not zero)
jap	unconditional jump
sld,sldi	scalar load/load immediate

General purpose SIMD instructions

- Basic arithmetical and logical operations have been added to enable basic data manipulation. No multiplications are provided a priori. The only way to perform normalization, whether on received messages from the channel or on messages after processing, is hence by using the shift operations.

Memory access vector instructions

- The instructions interfacing with the memory are in contrast to all other instructions vector instructions. They are manipulating the SIMD word as one

vector. They are tagged with the prefix v. All other instructions are operating on each element of the SIMD word individually.

The vpack and vunpack instruction are used to compress and uncompress check node messages of one check node. The vrl and vrr operation are used
5 for interleaving of SIMD words before writing them back to the APP bit node message storage. The packed check node messages can be transferred to and from the check node memory by employing the vst/vld or vstm/vldm instructions.

For state metrics and intermediate storage of messages, a stack-based
10 memory access scheme is proposed with the vpush and vpop instruction, which could also be implemented making use of the vst and vld instructions, respectively.

Special SIMD instructions

15 The core operations used to compute the different metrics are quite similar for Turbo Codes and LDPC codes. The main operation used in Turbo Codes is the $\max^*$ -operation, which could also be used for decoding LDPC codes.

Besides, a special instruction for efficiently dealing with the signs and magnitudes of messages, namely the $\min^+$ instruction, is proposed for
20 decoding LDPC codes with the min-sum algorithm. This $\min^+$ instruction should keep track of the current sign and the current smallest magnitude, to be computed in the forward and backward recursion at each state.

The cs or compare-select operation is proposed as a special instruction for the optimized min-sum approximation. It is used to find and keep track of the two
25 smallest magnitudes when computing the forward recursion. Due to similarities between the $\min^+$ and the cs operation, some resources or computational effort could be saved by combining these two instructions.

Scalar control instructions

30 Conditional and unconditional jumps as well as basic scalar arithmetical and logical operations are also provided. They are intended to be used for flow control and indexing purposes, respectively. A scalar register file with a minimum word width of for example 12 bits has to be foreseen, in order to be

able to index all message positions. All scalar instructions are tagged with the prefix s.

Pseudo code

Listing 4.1: Pseudo code to demonstrate intended usage of instruction set.

```

1  ;; --- SCALAR registers ---
2  ;; r1...nr. of bit nodes in current check
3  ;; r2...addr. to check node msg. storages for sub-block
4  ;; r3...sub-block base address in app memory
5  ;; r4...offset in current sub-block
6  ;; r5...intermediate constant
7  ;; r8...offset in code specification mem. to data
8  ;;      of current sub-block
9
10 ;; --- VECTOR registers ---
11 ;; v0...bit node msg.
12 ;; v1...check node msg.
13 ;; v2...app msg.
14 ;; v3...running max* value
15
16 ;; --- Pseudo code ---
17
18 ;; initialization
19 sld r1, CODEBASE:(r8) ; load nr. of checks of sub-block
20 sldi r1, #0x0000      ; cnt. to keep track of recursion
21 sldi r5, #0x0001      ; intermediate value
22
23 ;; forward recursion
24 FW:
25 vld v1, 0(r2)+         ; load check node msg.
26 vldm v2, 0(r3):(r4)+   ; load APP msg.
27 sub v0, v2, v1         ; compute bit node msg.
28 vpush @s0, v0          ; save bit node msg. for lw-rec
29 vpush @s1, v10         ; save last result from max* instr.
30 max* v3, v0            ; perform max* approx. on new input
31
32 ssub r1, r1, r5         ; check if recursion is finished
33 baz FW, r1

```

5

In the listing above, a small pseudo code snippet is depicted to illustrate the intended use of the instruction set according to embodiments of the present invention. The operands are specified in such a way that the target of the operation is always stated first. The instruction add v3 ,v1 ,v2 results in the arithmetical operation $v3 := v1 + v2$.

10

Beginning with line 19, some exemplary initialization takes place, where the number of check nodes of the current block-row is read from the code specification memory, with the offset to the current mode and standard

specified by CODEBASE. The scalar register r8 holds the offset to the current block-row.

The sldi instruction loads a immediate value to the specified register.

The SIMD load instruction vld implements a register-indirect memory

- 5 addressing scheme, where the address to read from is stored in register r2. An increment is performed after the read operation. For retrieving the data from the APP message storage, an offset-register-indirect addressing scheme is proposed. See Line 26 in the above listing, where the base address is specified in r3, and the index to the individual message is taken from r4. A post-
 10 increment with modulo-overflow is performed on r4, after the memory access. A special register has to be provided to specify the sub-block dimension, in order to perform the modulo operation.

Decoder core

- 15 The decoder core 101, in embodiments of the present invention, is implemented as a 64-bit very long instruction word (VLIW) ASIP with a scalar slot 109 and a 96-issue vector slot 110. The scalar slot 109 is associated with a 32x32-bit scalar registerfile (SRF) and the vector slot 110 with a 8x960-bit (96x10) vector registerfile (VRF).
- 20 The ASIP 101 is pipelined with a number of stages, up to 10 in the depicted example. The three first stages (p_fetch 130, p_pmem 131 and p_decode 132) take care of instruction fetch, instruction decoding and control flow (branch processing and handling of the external stall signal - *extstall*). Scalar operations and write back are handled in a single scalar execution stage 133
 25 (S_EX). In the vector slot, vector registerfile 110 read and writeback are pipelined to reduce the critical path. Vector operations are handled with 1 to 5 stages 134, 135, 136, 137, 138 (V_EX1, V_EX2, V_EX3, V_EX4, V_EX5) depending on the instructions. Most vector arithmetics are handled with two stages (execute and writeback). The q-equation instruction (for LDPC
 30 decoding according to [M. M. Mansour and N. R. Shanbhab, "Memory-efficient turbo decoder architectures for LDPC codes", IEEE Workshop on Signal Processing Syst. 2002, SIPS '02, pages 159 – 164, Oct 2002]) is pipelined in 7 stages. Read operation from the background memory structure 102

(including address translation) may require 7-stage delay. To minimize the number of vector registers 110, which multiplexing belongs to the most critical paths, a tightly coupled 128-entries 960-bit wide vector memory 139 may be added (2-cycle access latency). Key application specific instructions are

5 v_{maxstar} and v_{minsum} which are used in the kernel of both LDPC and turbo decoding. The SIMD processing is using virtual linear addressing, made possible through the shuffler 103, 104 and rotation engine blocks 120, which can perform permutations on the data stored within the background memory banks 102. Selection of the words within the memory banks 102 and shuffle

10 control is made possible through the rAGU 105. In case of decoding methods such as for example LDPC requiring a high parallelization (like in DVB), a rotation engine 120 connecting all processor cores, allowing a cyclic rotation of data from all background memory banks 102 towards the aligned scratchpads 117, is present. In other cases, the shuffler 103, 104 may be used to permute

15 the data within each N-slot processor.

Datapath and memory architecture proposal

According to embodiments of the present invention, by keeping the central processing units flexible, by means of implementing their functionality as an

20 ASIP with very dedicated instructions, different decoding approaches and methods can be deployed on the same hardware.

According to embodiments of the present invention, not only the main data path is shared for decoding different classes of codes (codes which are different in underlying coding principle, such as for example LDPC and Turbo

25 Codes), but also the major parts of the message storage are shared.

Therefore, in an aspect of the present invention, a unified architecture for multiple FEC schemes is provided.

For LDPC codes, it is clear that one has to go for a layered decoding scheme, since the disadvantage of restricting the decoder to a sub-class of LDPC

30 codes is far outweighed by the advantages gained by applying this scheme.

Two communication standards have been investigated (IEEE 802.11n, IEEE 802.16e) and enable efficient decoding in a layered way, hence the restriction does not seem to become a major problem for future implementations.

The layered scheme also allows splitting the parity-check matrix of the LDPC code into smaller sub-codes, which enables parallel processing of smaller portions of the code and multi-layer memory storage. This also opens possibilities for parallel metric computation when decoding Turbo Codes.

- 5 Simulation results have shown, that an internal message representation with a word length of 8 bits is enough, to deliver near-optimum performance, when applying the min-sum based approach. For the max*-based approach, at least 10 bits are required. Similar results can be shown for Turbo Codes, whereby this word length seems to be a feasible trade-off between decoding
- 10 performance and memory occupation. Reducing the word length also implies a gain in energy efficiency.

For the data path and memory organization, the discussion will now focus on LDPC codes. The generalization of the architecture to also cover Turbo

15 Codes, will not be discussed in detail hereinafter, but lies within the skills of a person skilled in the art. Nevertheless, some indications will be given on which changes are necessary at the architectural level, in order to allow efficient mapping of decoding kernels for Turbo Codes.

Architecture overview

- 20 A top level LDPC decoder architecture 160 is depicted in Fig. 16. The processing is organized in a check-node-centric way, where bit nodes are reduced to message storage and all computations take place in the check node processing unit.

The architecture 160 comprises five major parts:

- 25 1. The processing unit 161, where a configurable amount of parallel processing in a SIMD-based scheme can be applied.
2. The APP bit node message memory 162. Accesses to this memory 162 have to be performed in an interleaved way.
3. The interleaver 163 and deinterleaver 164 to align the APP messages
- 30 stored in the APP memory 162. Interleaving is done by address generators.
4. In order to perform the interleaving, the structure of the parity-check matrix must be known. Therefore, a memory 165 initialized with the characteristics of a certain code is also provided to retrieve the interleaving pattern from.

5. The check node message storage 166 to keep track of all check node messages.

Hereinafter, these parts of the architecture will be highlighted in more detail, starting with the memory organization.

5

Memory organisation

APP memory access interleaving

The interleavers are a part of the check-node-centric LDPC decoder architecture. An effective interleaving pattern must be found to ensure conflict free memory accesses.

10

The interleaver 163 and deinterleaver 164, as depicted in Fig. 16, are denoted π and π^{-1} , respectively. The interleavers/de-interleavers 163, 164 are explicitly shown as functional blocks, although their operation will be covered by the check node processing unit 161. How the interleaving can be performed to allow a SIMD-based processing approach, is shown based on two exemplary setups.

15

Interleaving example I

In Fig. 17, a possible interleaving scheme is depicted, where at part a, an exemplary parity-check matrix with a sub-block width of $Z = 3$ is shown. Each block-row consists of four sub-blocks and three block-rows are drawn. In part c, the compressed definition of the paritycheck matrix HAA is denoted as HDEF, where the cyclic permutation of the identity matrix is specified for each sub-block.

20

Assume that Z messages can be stored in one memory word, this means for this simplified example, three messages are concatenated to form one word. Also assume that the SIMD check node processing unit has three slots and can therewith process three messages from independent check equations per step.

25

As defined by the parity-check matrix, the first check node needs the messages from bit node 1, 5 and 9, the second check node processes the messages from bit node 2, 6, and 7, and so on. The first, second and third check node will be assigned to the first, second and third slot of the SIMD unit.

30

For the first computation step, to fill one SIMD data word, the messages from bit node 1, 2 and 3 have to be transferred to the processing unit in that order.

Assume that the messages stored in the APP memory are already in the correct order, so that for each memory access, one processing step can be performed. For the second SIMD operation, the messages from bit node 5, 6 and 4 have to be transferred to the slots processing check node 1, 2 and 3, respectively.

After having processed all bit node messages of the check equations in that layer, the forward iteration is finished and the backward iteration can start.

- 10 After each computation step in the backward recursion, one output SIMD word is produced. Due to the nature of the backward recursion, the results delivered in reversed order than they were processed in the forward recursion. In this first example, where Z parallel computation slots are available, interleaving and write back of the output SIMD words can start immediately after each output word was computed. To align the updated APP message of each bit node, in order to be able to efficiently read it when decoding the next layer, its position in the SIMD word has to be adapted.

- 15 This is done by rotating each SIMD word by the offset difference between the cyclic permutation index of the sub-block at this layer and the cyclic permutation index of the sub-block in the next layer. As an example, for the first sub-block in layer one, its permutation index is zero, meaning that the bit nodes participating in the check nodes of this layer, are following the pattern of an identity matrix. For the next layer, the same bit nodes are participating according to a shifted version of the identity matrix, where the shift value is two. So, to arrange the bit node messages of the first sub-block in order to enable efficient reading in the next layer, it must be rotated by $0-2 = -2$ before writing it back. A positive value means a rotation to the right, whereas a negative value means a rotation to the left.

- 20 For the scenario depicted in Fig. 17 the inherent parallelism was exploited, which results in an efficient reading, processing, interleaving and write back flow. It might not always be the case that the inherent parallelism is fully exploited, for instance if the data path width gets too wide, or if the throughput requirements can be met with less parallelism. In that case, the interleaving

introduces a significant overhead in terms of delay, as illustrated in the next example.

Interleaving example II

- 5 Consider a parity-check matrix as depicted in Fig. 18, where the sub-block size $Z = 6$ and three sub-blocks are arranged as one block-row. In the example, two block-rows are shown, with their permutation indexes summarized in part b of the figure. Now, instead of exploiting the inherent parallelism of six, only three processing slots are deployed. One memory word again holds three messages and therewith forms a complete SIMD word. As a consequence, decoding of one layer is done in two steps, first for the check equations one to three and afterwards for the check equations 4 until 6.

- Assume the words in the memory are already organized in such a way, that the read words can already be used as input data for the SIMD processing unit. After three processing steps, the first three check node equations have been processed in the forward recursion and while producing the outputs in the backward recursion, the next three input messages, for check node 4 until 6, can be fetched from the memory. This procedure is depicted in Fig. 18 as part c.

- 20 What turns out to be the main bottleneck of this scenario is that before the results from the first decoding round, namely the results of check node 1, 2 and 3, can be written back to the memory, the results of all six check nodes in this layer need to be available. This is illustrated in Fig. 18, as the dashed place holder for the results of the remaining three check nodes.

- 25 After the arrival of all output messages updated by this layer, interleaving by performing rotations can be performed and the rearranged SIMD words, suitable for being read when decoding the next layer, can be written back to the APP message storage.

Code specification memory

- 30 As the parity-check matrix is fully specified by the set of cyclic rotations to perform on the identity matrix with dimension $Z \times Z$, this is the only information needed to be stored by the decoder architecture to have full knowledge about the interconnections between bit nodes and check nodes for a given standard.

As indicated in Fig. 16, this data is stored in two logical parts of a Read-Only Memory (ROM). In the first part, called the counter base ROM 167, parameters like the sub-block dimensions Z , the number of layers and the block length are stored. Additionally, in the counter init ROM 168, a cyclic rotation value for
5 each sub-block and its position in the parity-check matrix is stored. With the rotation index and the position of the sub-blocks in the parity-check matrix, the actual index of a bit node in the data block can be computed. The bit node index for each bit node in one sub-block, can be computed by taking a modulo-counter, initialized by the value specified in the counter init ROM 168 and
10 adding the offset value for the current subblock.

An exemplary arrangement of such modulo-counters to compute the bit node indexes to the APP message storage for the interleaving example I, as detailed before and depicted in Fig. 17, is shown in Fig. 19.

15 Check node message storage

The APP message storage 162 is initialized with the received channel values. These APP of each bit node is updated with a more accurate estimation after each sub-iteration. In addition to the APP messages, every check node message, computed by the check node processing unit, must be stored for
20 each sub-iteration. This means for each check node in one layer, dc messages need to be stored. In case of the (1944,972)-irregular LDPC code standardized in the IEEE 802.11n standard, where the average check node degree is 7, 6966 8-bit messages need to be stored. When assuming the 8-bit, quantization of internal messages from previous simulation results is chosen. Due to the
25 irregularity of the check node degree, the number of messages to be stored for each check node is varying.

To reduce the number of messages to store, the optimized min-sum approximation has been introduced. When this scheme is applied, per check node and per layer, only two 8-bit magnitudes need to be stored. This is
30 enabled by storing the check node messages in a compressed way. The composition of an example of a packed vector is depicted in Fig. 20 in part a. Two magnitudes, the smallest and the second smallest one are stored in one

word. It is to be noted that these values are unsigned values and thus two additional bits can be saved.

Besides the magnitudes of the messages, track must be kept of which magnitude belongs to which individual check node message. There are only two magnitudes to choose from, therewith only one bit per message is needed. The same applies to the sign of the message, which also needs to be stored to reproduce a specific message. How the unpacking of messages can be done, is depicted in part b of Fig. 20. Coming back to the example of the (1944,972)-irregular LDPC code, where the check node degree is either 7 or 8, this results in a maximum word length needed to store all check node messages of one check node in direct form of $8 \times 8 \text{ bit} = 64 \text{ bits}$. When storing the messages in packed form, for one check node $2 \times 7 \text{ bits}$ for the magnitudes + $8 \times 1 \text{ bit}$ for signs + $8 \times 1 \text{ bit}$ for indexes = 30 bits are needed. Which offers the use of a bit width of 32 bits. When comparing the check node storage requirement in packed and in direct form, assuming that no alignment needs to be preserved when storing the messages in the direct form, the memory requirement for the example LDPC code can be reduced by more than 2k of 8-bit bytes, when making use of the optimized scheme. This is a memory footprint reduction of around 30%.

Data path

Check node processing

The detailed data path of a check node processing unit, as introduced in the top level view in Fig. 16, is depicted in Fig. 21 and will be discussed hereafter.

Based on the representative fixed structure when applying the min-sum approximation in combination with the forward-/backward algorithm, a flexible unified instruction set architecture according to embodiments of the present invention is motivated.

Therefore, first the computation of one check node is discussed based on Fig. 21.

Forward recursion

For the computation of one check node, two iterations are involved. In the first step, the forward recursion, a new value can be computed for each incoming APP message, see Fig. 21. Therefore, the following operations are necessary: to get the bit to check node message which is the basis for all computations in the check node, the check node message from the previous iteration stored in packed form, needs to be subtracted from the APP message. This computed bit node message is stored in an intermediate buffer to avoid re-computation of the values in the backward recursion. From the bit node message, the sign is extracted and multiplied with a running product of bit node message signs. The current state of the running product, before it got updated with the current input message, is stored in an intermediate buffer, which will be used in the backward recursion. A similar procedure is applied to the magnitude of the bit node message, where the minimum of all considered bit node messages until now, held in the α register, is stored in the α recursion memory. The magnitude of the current message is checked against the current smallest magnitude, in the α register. If the incoming magnitude is smaller than the stored one, the value in the α register is replaced. These operations on the sign and magnitude of the bit node messages are combined in the min+ instruction of the proposed instruction set. The CS-operation is in parallel comparing the magnitude of the input messages against a local storage, where the two smallest magnitudes are being stored. The CS-operation also updates the local storage, if an incoming message contains a smaller magnitude than the stored ones. The cs-operation is also available as a single instruction to minimize arithmetic operations and conditional jumps during computation of the forward recursion.

Backward and output recursion

After the forward recursion, all signs and minima's at each state of the computation are available in the sign_u and α recursion memory, respectively. Now, in the backward recursion, this information is used to compute on the one hand, the updated APP messages for every bit node having a direct connection to this check node and on the other hand, to compute and pack all check node messages, used to produce the bit node messages in the next

- iteration. The bit node messages, the α recursion messages and the sign_u messages produced in the forward recursion are now consumed in the backward recursion in a stack-based manner where the last produced value is consumed first. Again, the sign of each considered message is multiplied with
- 5 a running product of signs and the smallest magnitude for all messages considered before this step of the recursion, is held in the local register β_i . With the magnitude and sign captured at each state during the forward recursion, combined with the magnitude and sign computed in the backward recursion (β_i and sign_i), an output message can be computed.
- 10 The sign storage sign_u used to capture the intermediate states of the signs in the forward recursion can now be used to store the signs of the output check node messages. The signs from the forward recursion, which have already been considered, can be overwritten. For every output check node message, it also has to be remembered, whether its magnitude is equal to the overall
- 15 smallest one, or the second smallest one. This information is stored in the idx buffer. After finishing the backward recursion, the check node messages used to generate the updated output APP messages, are represented by the updated buffers sign_u , idx and the storage of the updated smallest magnitudes m_0 and m_1 . Before the check node messages are stored in the appropriate
- 20 check node message memory entry, they get packed into a single word, as depicted in Fig. 20.

Unified architecture

- All operations performed by this representative check node processing unit,
- 25 are reflected in the proposed instruction set given above. By having this instruction set available, the above kernels can efficiently be implemented in software. In addition to an register file to store intermediate results, two local stack-based storage buffers for the bit node and α recursion messages are needed.
- 30 Due to the independent, identical computations performed in each check node in one layer, a SIMD-based computation of multiple check nodes is possible. Then, each instruction is operating on messages from independent check nodes in parallel. Where some special instructions, namely all vector

instructions, are operating on the whole SIMD word to enable loading, storing and interleaving of SIMD words. Due to the varying sub-block dimensions (between 24 and 96 in the investigated standards) for different standards and codes, the exploitable parallelism is also varying.

- 5 When a fixed SIMD word width is assumed, not all slots of the SIMD processing unit can always be occupied, when decoding different modes of a communication standard. For example, when a SIMD word width of 27 is implemented and a code from the IEEE 802.16e standard with a sub-block width of 24 is decoded, 3 slots can not be used during processing. Every slot is
10 operating on a different check node equation, hence free slots can not be used to allow faster computations of an individual check node.

- For the proposed interleaving scheme, some further restrictions regarding the APP memory word width must be taken into account. To be compliant to a plurality of decoding methods, e.g. both investigated standards, with a given
15 SIMD word width, the memory architecture needs to make sure that the input SIMD words can efficiently be loaded and stored. The interleaving is aligned on sub-block borders and depending on the current mode (sub-block width), the memory words of messages from different sub-blocks must not overlap. This means that when the sub-block width is for example 32, 32 messages
20 have to be loaded for one decoding step. If the memory word is for instance 28 messages wide, which would make sense for a different mode, no proper alignment of messages from different sub-blocks is possible. As a result, messages must either be collected from different memory words, where certain positions need to be masked, or some storage positions stay empty. It is to be
25 noted that interleaving before write back can only be done on words with all messages of one sub-block available.

- In embodiments of the present invention, a restricting standard such as the IEEE 802.16e standard could be used, where subblock sizes are standardized in intervals of 4. This results in possible subblock sizes of 24, 28, 32, . . . , 96.
30 To enable aligned storage of messages, the maximum message word width is four messages. To build a SIMD word, when for instance 28 parallel slots are available, 7 read operations are necessary. In case multiple read ports are available, the number of read cycles can further be reduced. A possible

memory hierarchy for this example is depicted in Fig. 22. There, the 28 messages are composed of the 7 memory words. Once all memory words have been read from the memory, the SIMD word can be passed to the processing unit. Its output is a SIMD word containing 28 messages, which can be split up again for write back.

If in other standards not all modes of the IEEE 802.16e standard are mandatory to be supported, a wider memory word width could be used. When for instance only modes in an intervall of 8 are supported, the memory width could also be increased to 8 messages per memory word.

The foregoing description details certain embodiments of the invention. It will be appreciated, however, that no matter how detailed the foregoing may appear in text, the invention may be practiced in many ways. It should be noted that the use of particular terminology when describing certain features or aspects of the invention should not be taken to imply that the terminology is being re-defined herein to be restricted to include any specific characteristics of the features or aspects of the invention with which that terminology is associated.

It is to be understood that although preferred embodiments, specific constructions and configurations, as well as materials, have been discussed herein for devices according to the present invention, various changes or modifications in form and detail may be made without departing from the scope of this invention as defined by the appended claims.

CLAIMS

- 1.- A semi-automated method for exploration of data decoding methods whereby at least one of said data decoding methods requires data shuffling operations on said data, the method comprising
- 5 loading at least two decoding methods which are different in underlying coding principle,
loading a template of an ASIP device having a memory structure and an ASIP core with a pre-defined instruction set,
changing the codes of each of the decoding methods into one or more fixed
- 10 point approximations making use of the pre-defined instruction set,
selecting for each of the decoding methods one approximation and scheduling of the data accesses to the memory structure, the selecting and the scheduling being by evaluating the performance of the execution of each of said approximations on the device, the performance including throughput and
- 15 power consumption by the ASIP core and the power consumption due to data accesses to said memory structure, and
delivering the selected approximations of said at least two decoding methods and providing the selected scheduling.
- 2.- A semi-automated method according to claim 1, wherein the
- 20 approximations comprise arithmetic operations excluding multiplication, division and power.
- 3.- A semi-automated method according to claim 2, wherein the arithmetic operations comprise instructions selected from the group (max, min, sum, subtraction).
- 25 4.- A semi-automated method according to any of the previous claims, wherein the method furthermore comprises exploration of different templates of an ASIP device.
- 5.- A semi-automated method for exploration of decoding methods and an instruction set suitable therefor, comprising
- 30 loading at least two decoding methods which are different in underlying coding principle,
loading a template of an ASIP device with a plurality of possible instruction

sets,

changing the codes of each of the decoding methods into a plurality of fixed point approximations making use of the instruction sets,

selecting for each of the decoding methods one approximation and the

- 5 corresponding instruction set by evaluating the performance of the execution of each of said approximations on the device, the selected instruction set being the same for the at least two decoding methods, and
delivering the selected approximations of said at least two decoding methods and the corresponding instruction set.

10 6.- Instruction set delivered by a method according to claim 5.

7.- Instruction set according to claim 6, comprising arithmetic operators excluding multiplication, division and power.

8.- Instruction set according to any of claims 6 or 7 wherein the arithmetic operators comprise any of max* or min sum.

15 9.- A system (100) for execution of a decoding method, the system (100) being capable of executing at least two data decoding methods which are different in underlying coding principle, whereby at least one of said data decoding methods requires data shuffling operations on said data, the system (100) comprising

20 at least one ASIP core (101) having an instruction set comprising arithmetic operators excluding multiplication, division and power, the ASIP core being selected for execution of approximations of each of said at least two data decoding methods,

at least a first memory unit (102) for storing data,

25 a transfer means for transferring data from the first memory unit (102) towards said at least one ASIP core (101), said transfer means including a data shuffler (103), and

a controller for controlling the data shuffler independent from the ASIP core.

10.- A system (100) according to claim 9, wherein the at least two
30 different data decoding methods include LDPC.

11.- A system (100) according to any of claims 9 to 10, wherein the at least two different data decoding methods include turbo decoding.

- 12.- A system (100) according to any of claims 9 to 11, comprising a plurality of ASIP cores.
- 13.- A system (100) according to any of claims 9 to 12, wherein the transfer means further comprises a data rotation engine (120).
- 5 14.- A system according to any of claims 9 to 13, wherein the ASIP core is adapted so that the at least two different data decoding methods use the same datapath.
- 15.- Use of a system with at least one programmable ASIP core with dedicated instructions for executing a group of decoding methods which
10 are different in underlying coding principle, said executions at least exploiting said at least one ASIP core.
- 16.- Use according to claim 15, wherein the group of decoding methods comprises at least turbo decoding and LDPC decoding methods.
- 17.- A method for decoding data, comprising
15 loading an approximation of a decoding method,
loading coded data,
decoding the data by means of the approximation,
wherein decoding the data comprises executing arithmetic operations on the coded data by means of a programmable ASIP core, and controlling
20 scheduling of data accesses to a memory structure by a controller separate from the ASIP core.
- 18.- A method according to claim 17, wherein decoding the data comprises executing arithmetic operations on the coded data by means of a plurality of parallel ASIP cores.
- 25 19.- A method according to claim 18, wherein controlling scheduling of data accesses comprises controlling of shuffling of data from a memory structure by a controller separate from the ASIP core.
- 20.- A method according to claim 19, wherein the controller is adapted for controlling data accesses so as to be collision-free.

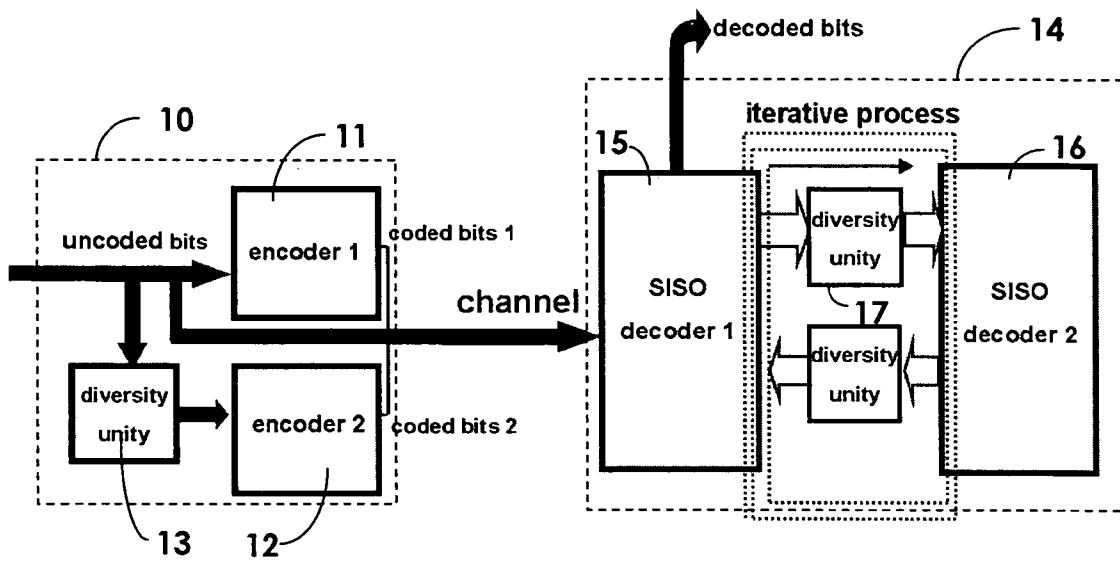


Fig. 1

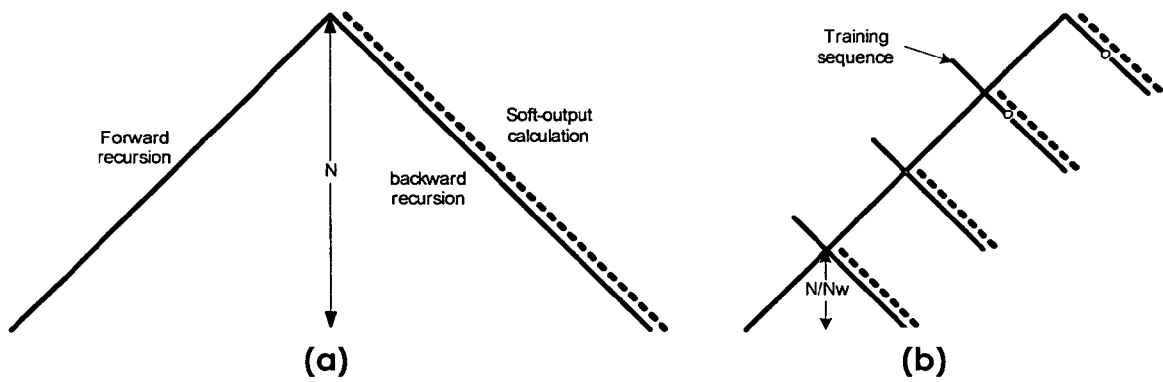


Fig. 2

2/18

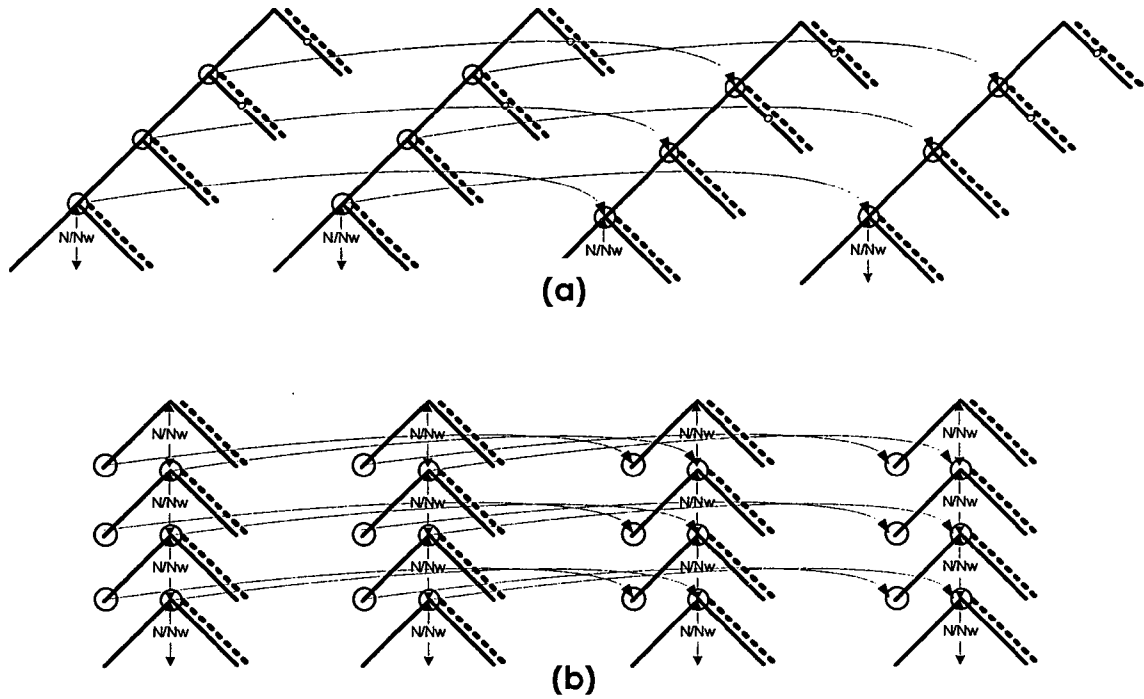
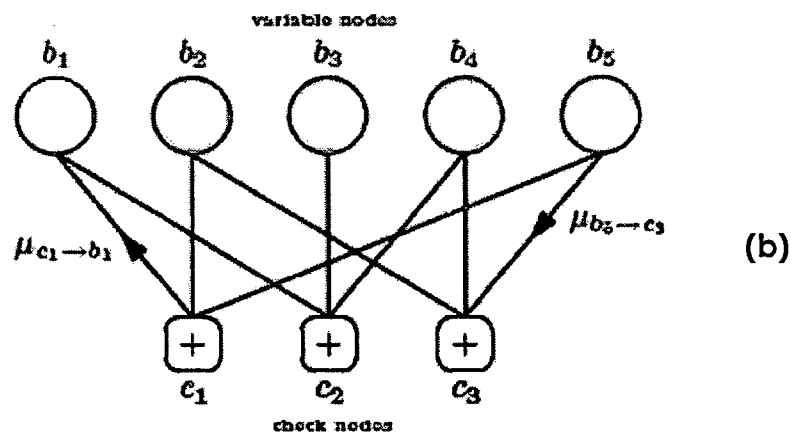


Fig. 3

$$H = \begin{bmatrix} \begin{matrix} b_1 \\ 1 \\ 1 \\ 1 \\ 0 \end{matrix} & \begin{matrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{matrix} & \begin{matrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{matrix} & \begin{matrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{matrix} & \begin{matrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{matrix} \end{bmatrix} \begin{matrix} c_1 \end{matrix}$$

variable node check node

(a)



(b)

$$\begin{aligned} c_1 &= b_1 \oplus b_2 \oplus b_5 \\ c_2 &= b_1 \oplus b_3 \oplus b_4 \\ c_3 &= b_2 \oplus b_4 \oplus b_5 \end{aligned}$$

(c)

Fig. 4

3/18

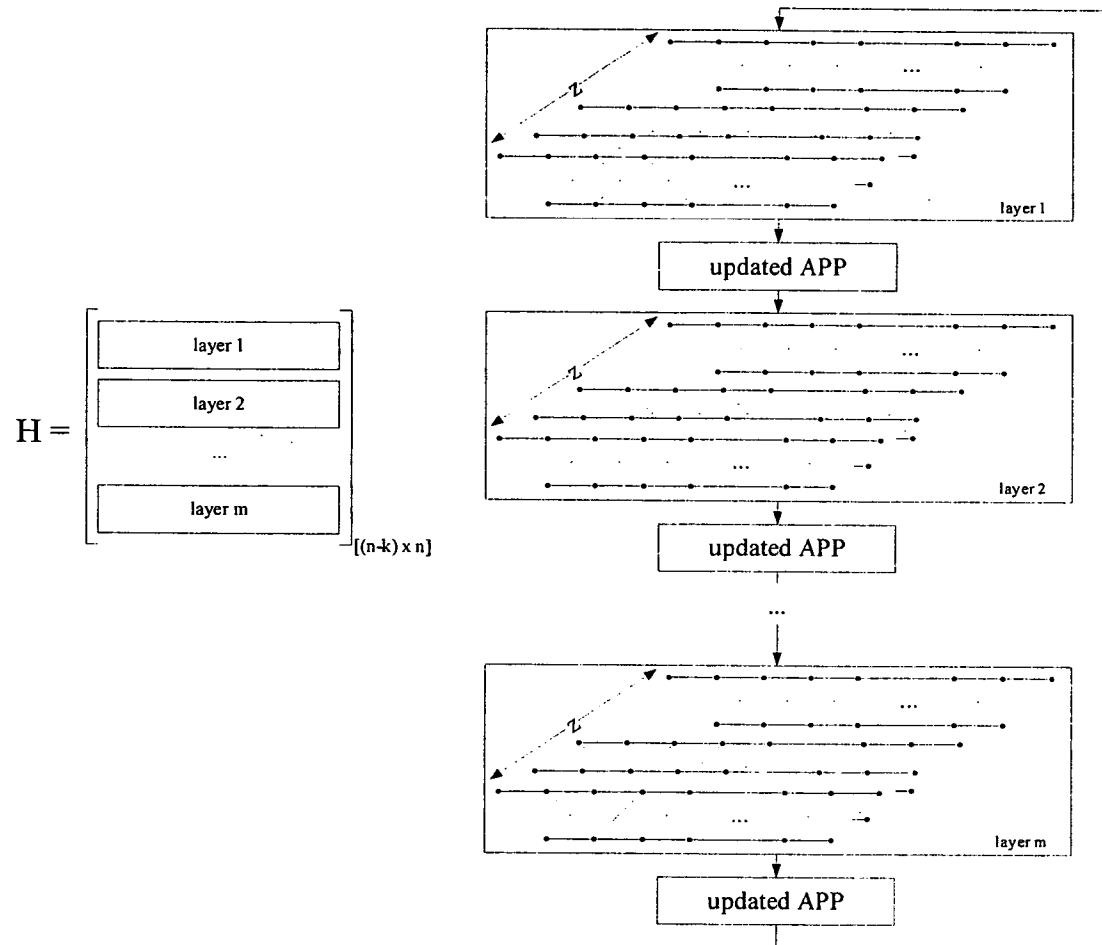


Fig. 5

4/18

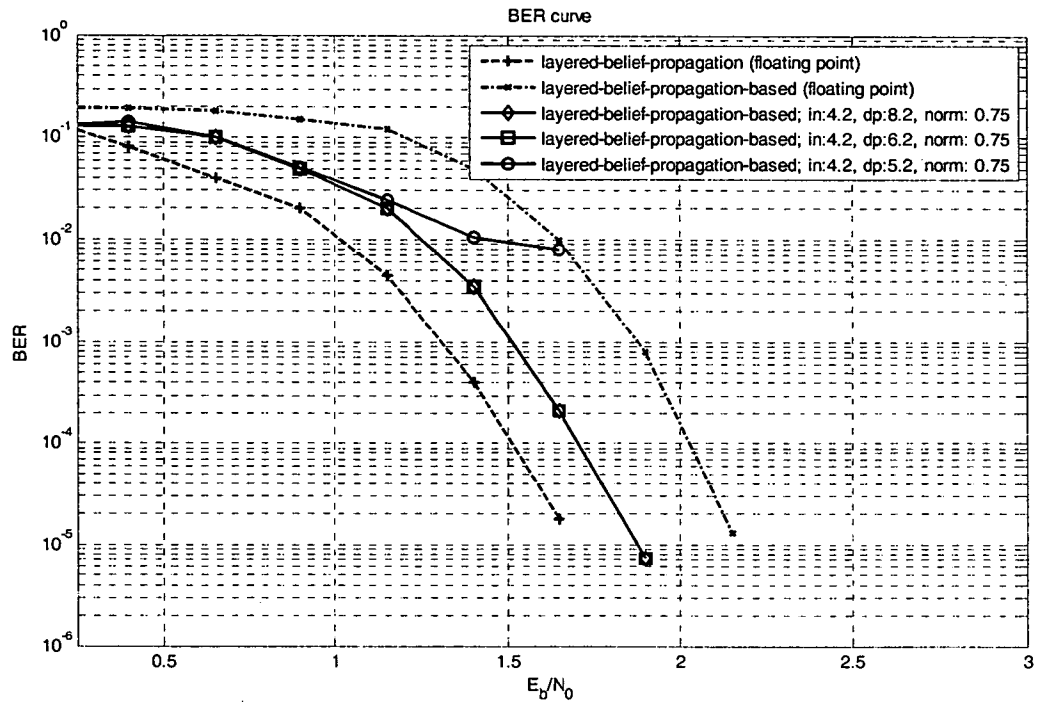


Fig. 6

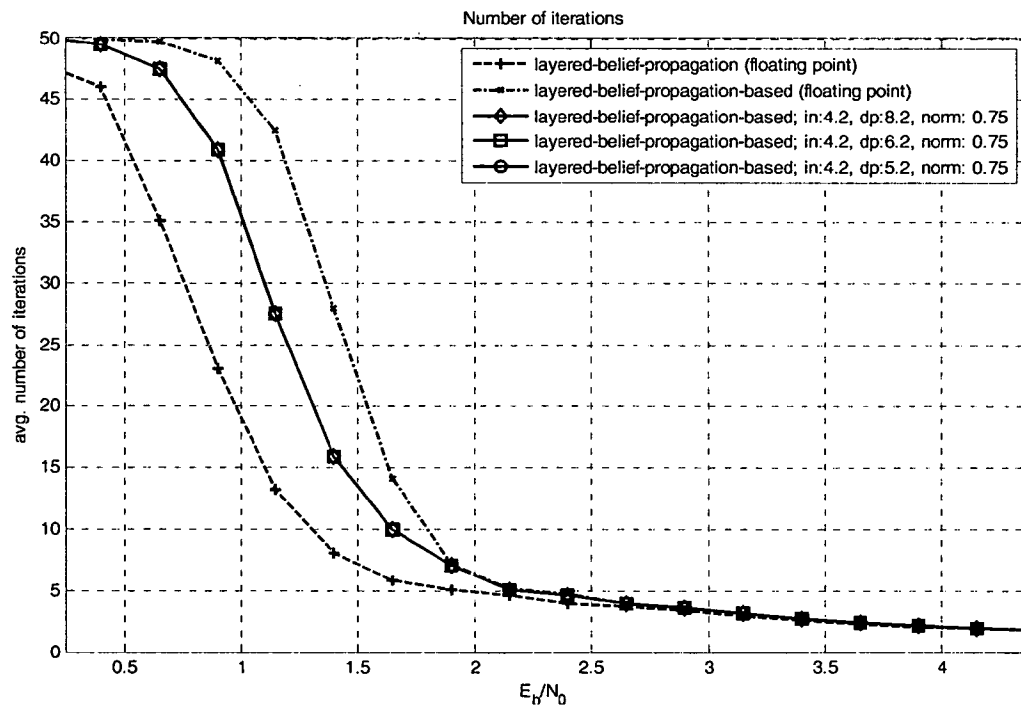


Fig. 7

5/18

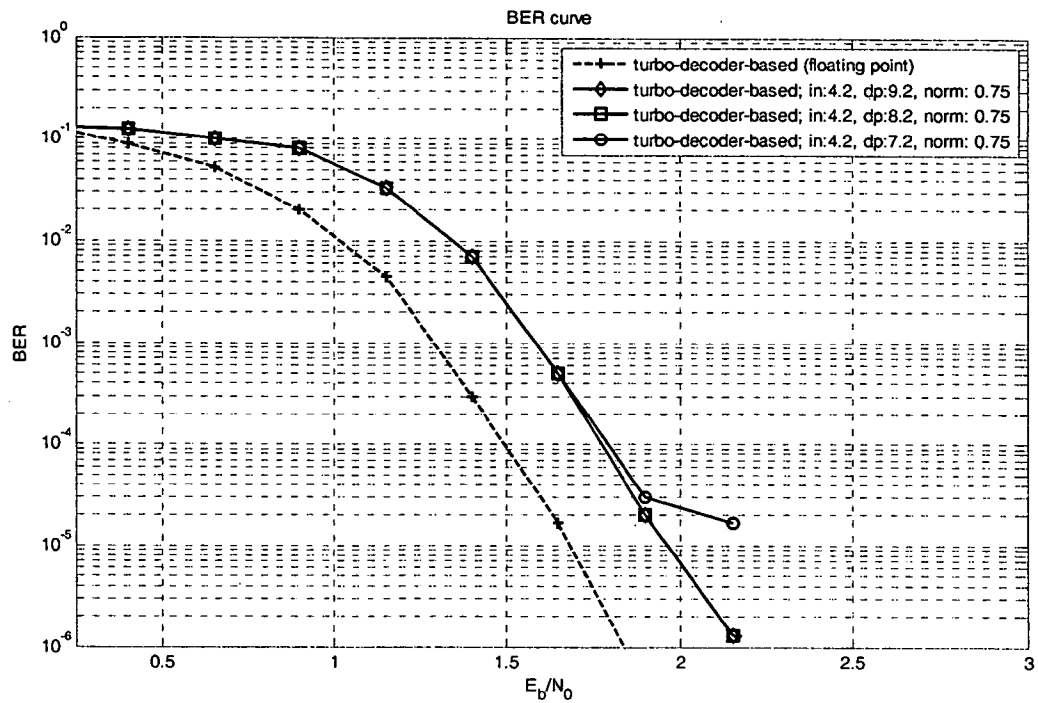


Fig. 8

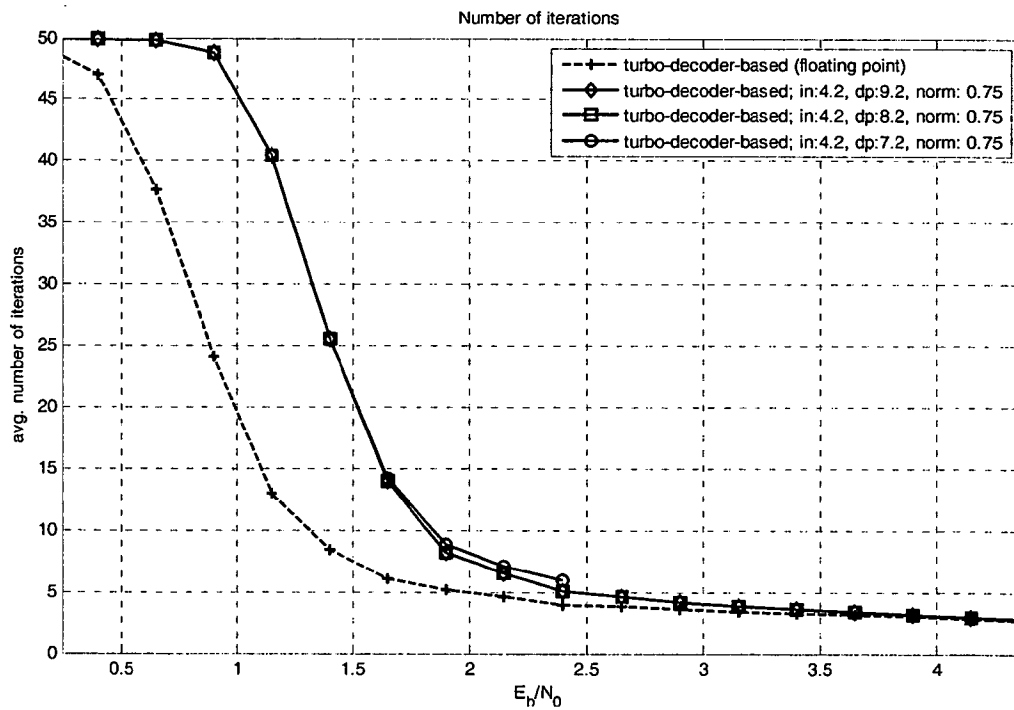


Fig. 9

6/18

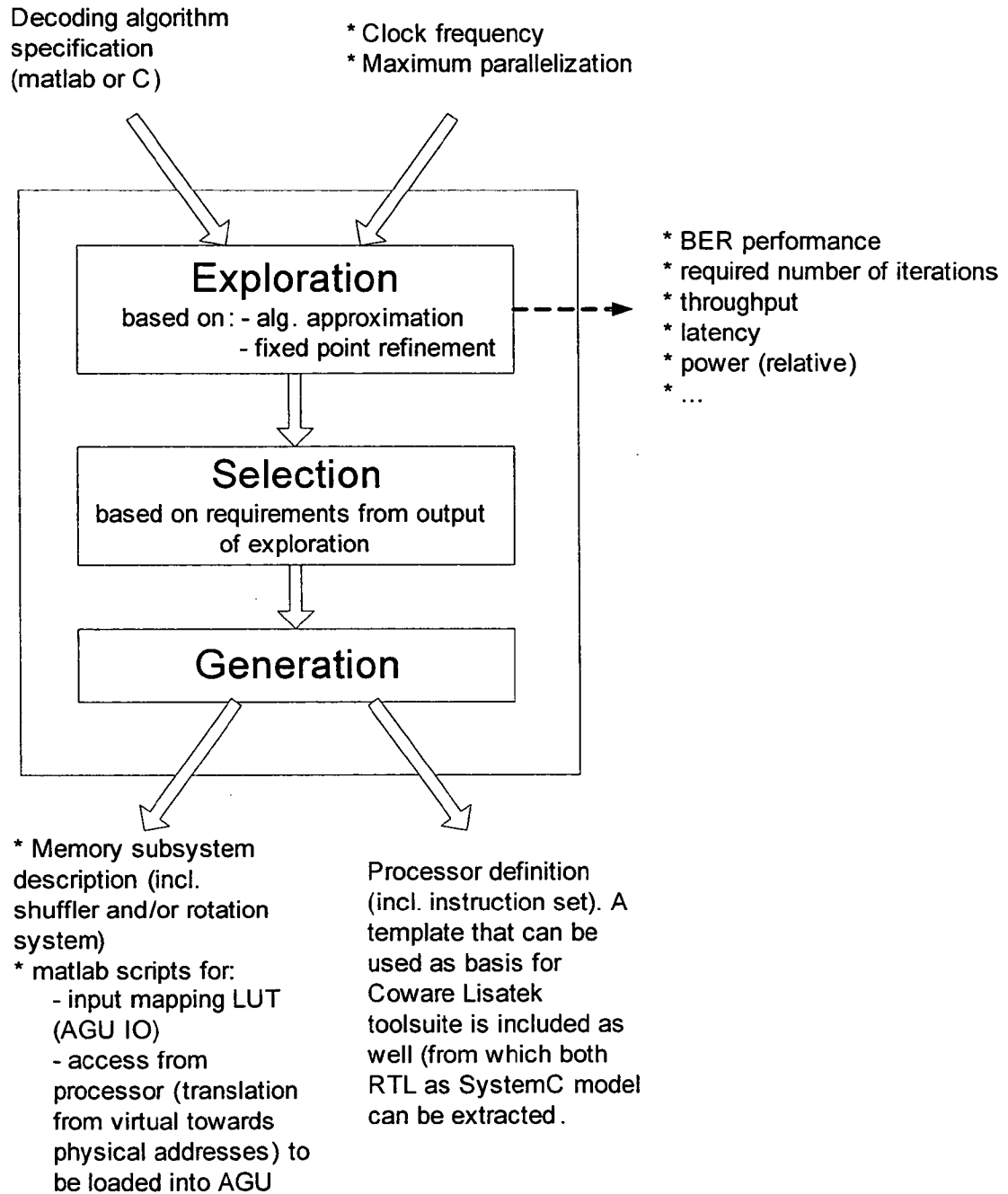


Fig. 10

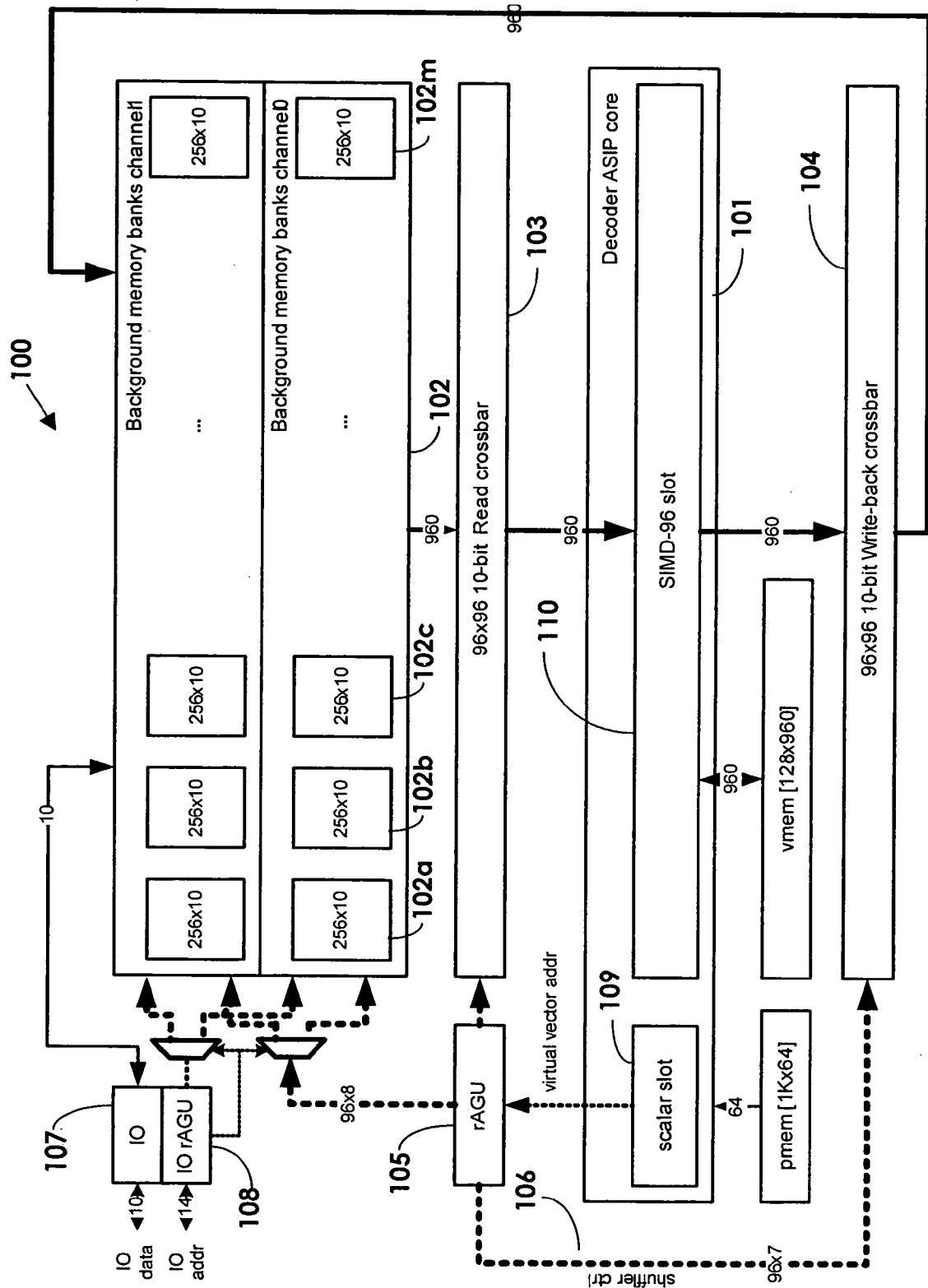


Fig. 11

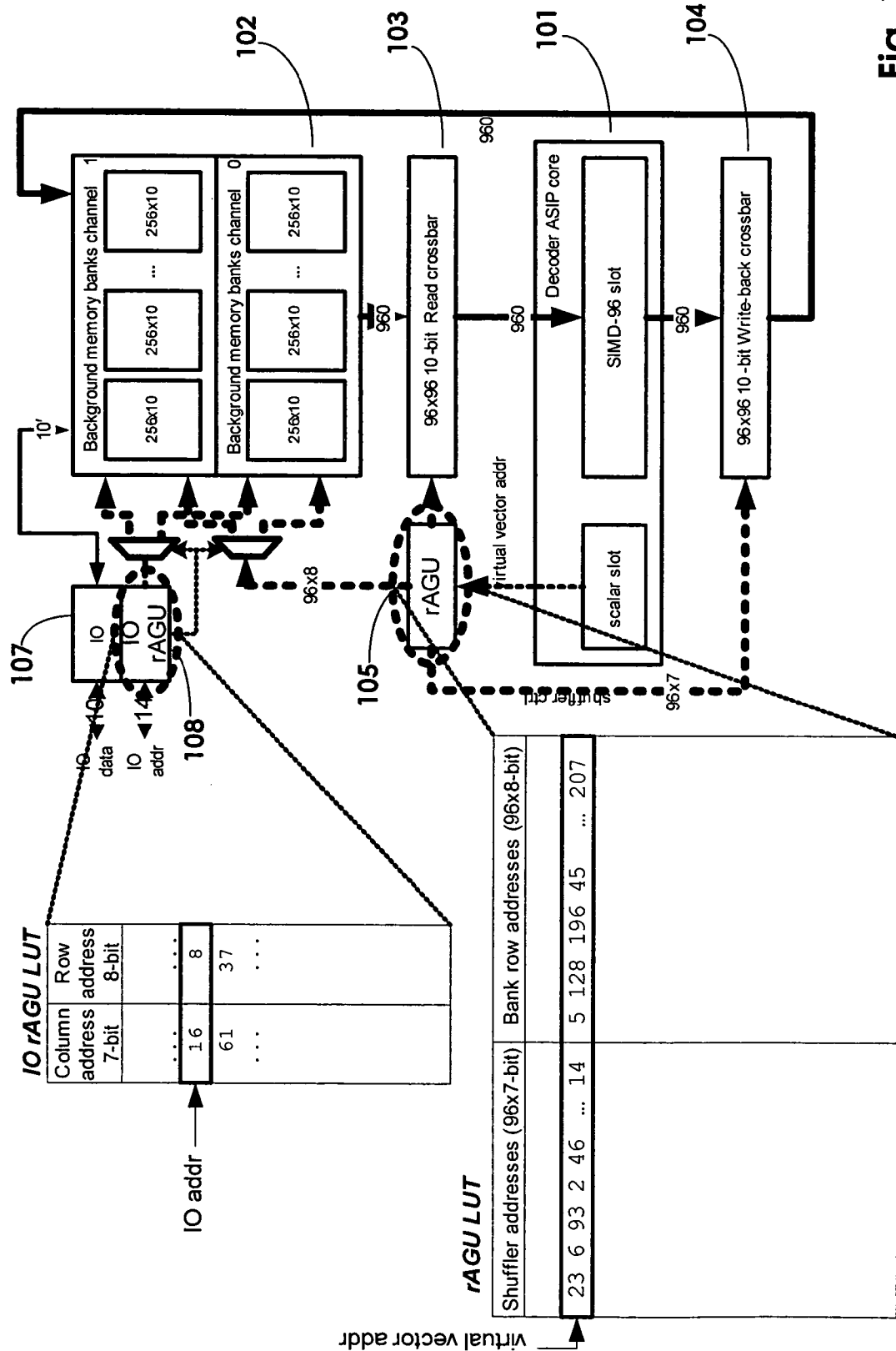


Fig. 12

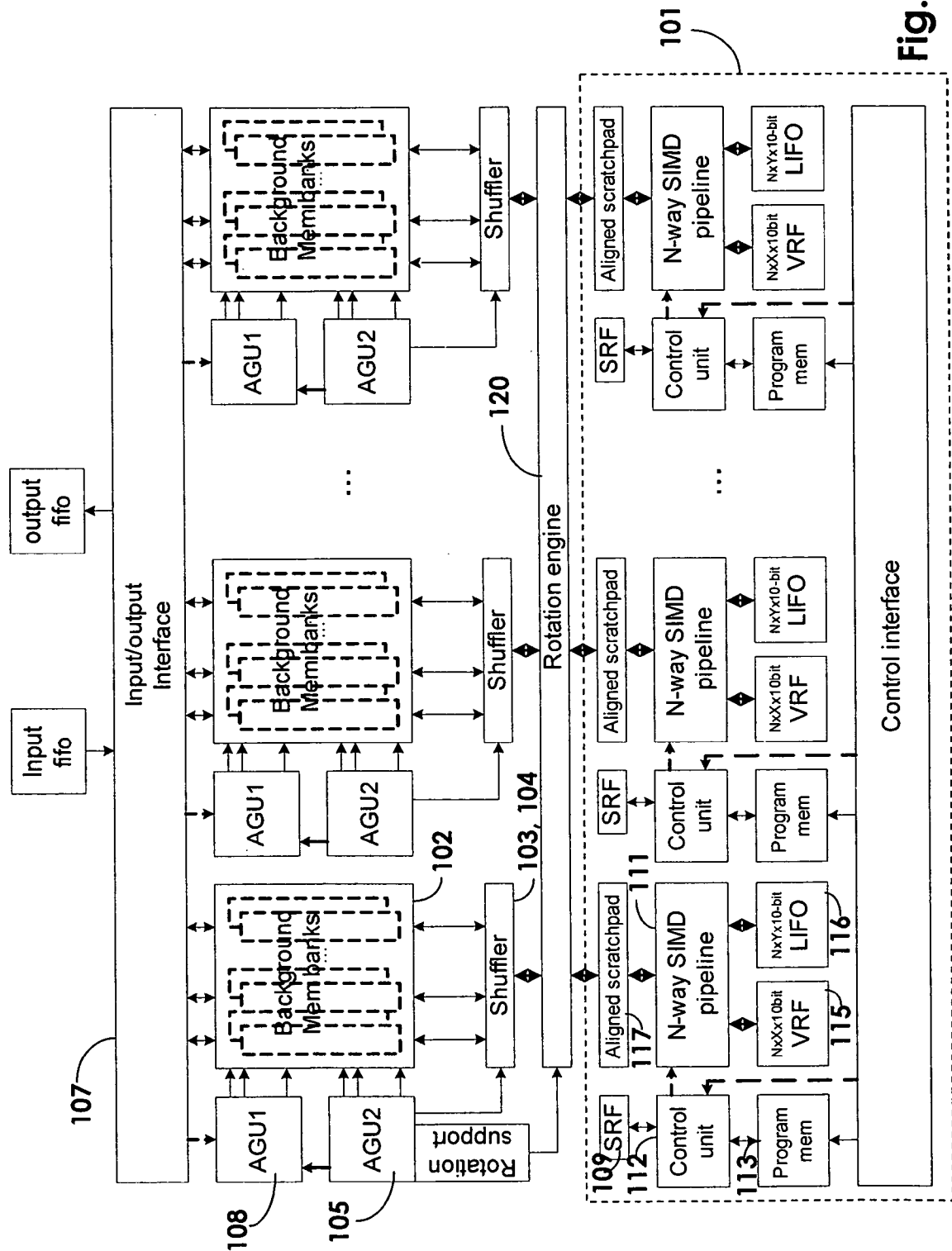


Fig. 13

10/18

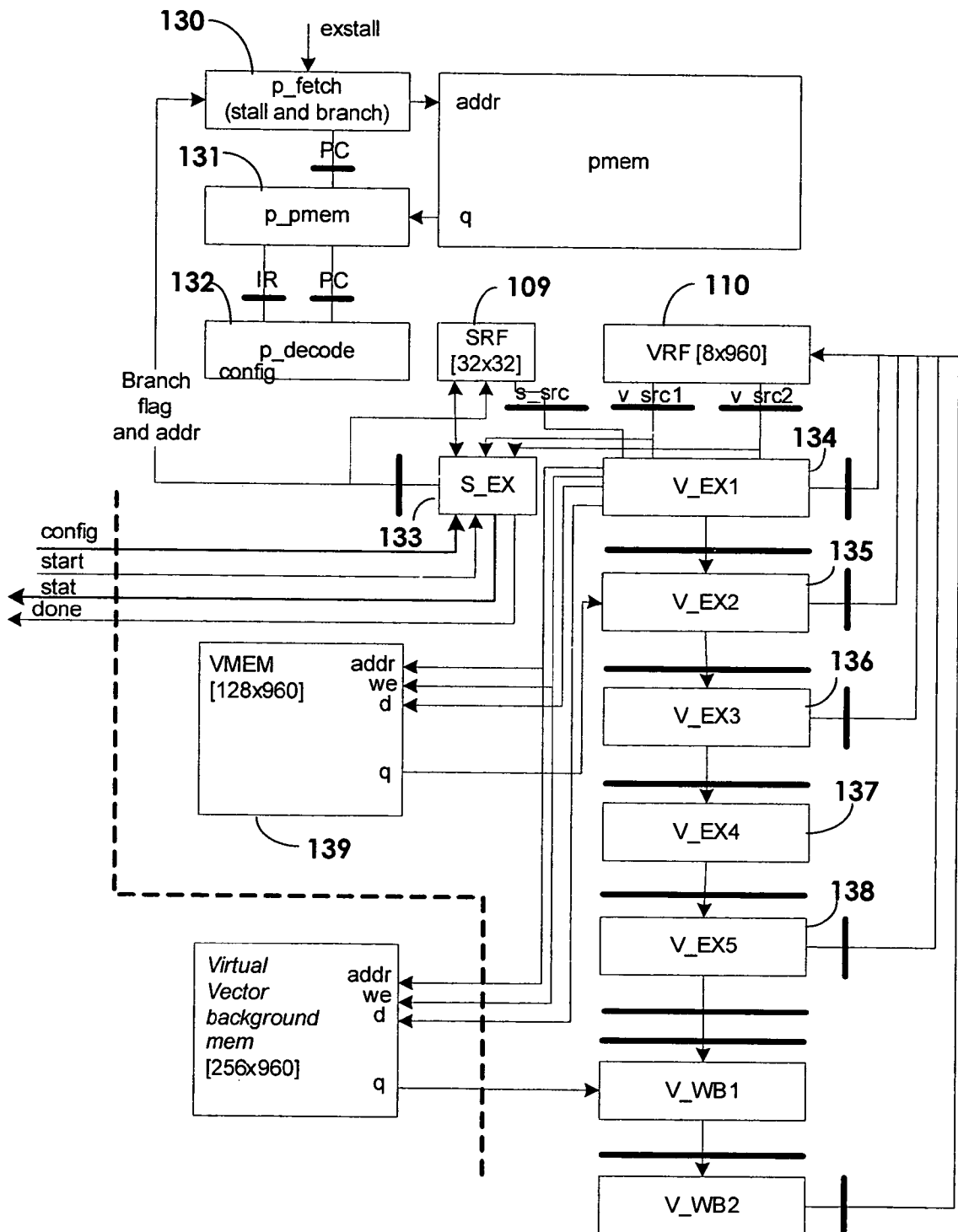


Fig. 14

Table 1 : Instruction set architecture

Slot	Group	Instruction	Usage	Semantic	
scalar	alu	and,or,xor	[INSTR] [src_reg1] [src_reg2] [dest_reg]	_dest_reg = _src_reg1 {& ,^} _src_reg2	Basic logic operations
scalar	alu	andi,ori,xori	[INSTR] [src_reg] [imm16] [dest_reg]	_dest_reg = _src_reg1 {& ,^} _imm16	
scalar	alu	add, sub	[INSTR] [src_reg1] [src_reg2] [dest_reg]	_dest_reg = _src_reg1 {+, -} _src_reg2	Basic arithmetic operations
scalar	alu	addi, subi	[INSTR] [src_reg] [imm16] [dest_reg]	_dest_reg = _src_reg1 {+, -} _imm16	
scalar	alu	mov, not	[INSTR] [src_reg] [dest_reg]	_dest_reg {=, !} _src_reg	
scalar	alu	sl, sr	[INSTR] [src_reg1] [src_reg2] [dest_reg]	_dest_reg = _src_reg1 {<<, >>} _src_reg2	Shift left/right
scalar	alu	sli, sri	[INSTR] [src_reg] [imm16] [dest_reg]	_dest_reg = _src_reg1 {<<, >>} _imm16	
scalar	alu	movli, movhi	[INSTR] [imm16] [reg]	_reg = (_reg & 0xFFFFF0000) _imm16 _reg = (_reg & 0x0000FFFF) (_imm16 << 16)	
scalar	branch	blt, bne	[INSTR] [src_reg1] [src_reg2] [label]	Jump to _label is _src_reg1 {<, !=} _src_reg2	
scalar	branch	jmp	[INSTR] [label]	Jump to _label	
vector	vls	vld	[INSTR] [reg] [imm3] [vreg]	_vreg = mem(virt_addr, _reg, bank=imm3)	
vector	vls	vldi, vldb	[INSTR] [imm16] [imm3] [vreg]	_vreg = mem({virt_addr, addr}=imm16, bank=imm3)	
vector	vls	vst	[INSTR] [reg] [imm3] [vreg]	mem(virt_addr, _reg, bank=imm3) = _vreg	
vector	vls	vsti, vsib	[INSTR] [imm16] [imm3] [vreg]	mem({virt_addr, addr}=imm16, bank=imm3) = _vreg	
vector	vscratch	vld	[INSTR] [reg] [vreg]	_vreg = scratchpad_mem(addr=_reg)	
vector	vscratch	vst	[INSTR] [reg] [vreg]	scratchpad_mem(addr=_reg) = _vreg	
vector	vstack	vpop, vpush	[INSTR] [vreg]		
vector	varith	vadd, vsub, vxor	[INSTR] [src_vreg1] [src_vreg2] [dest_vreg]	_dest_vreg = _src_vreg1 {+, -, ^} _src_vreg2	
vector	varith	vabs	[INSTR] [src_vreg] [dest_vreg]	_dest_vreg = abs(_src_vreg)	
vector	varith	vmax	[INSTR] [src_vreg1] [src_vreg2] [dest_vreg]	_dest_vreg = max(_src_vreg1, _src_vreg2)	
vector	varith	vmaxstar	[INSTR] [src_vreg1] [src_vreg2] [dest_vreg]	max * based approximation for check node computation based on [1] applied on _src_vreg1 and _src_vreg2, result stored in _dest_vreg	
vector	varith	vminsum	[INSTR] [src_vreg1] [src_vreg2] [dest_vreg]	Min-sum based approximation applied on _src_vreg1 and _src_vreg2, result stored in _dest_vreg	
vector	varith	vsli, vsri	[INSTR] [src_vreg] [imm10] [dest_vreg]	_dest_vreg = _src_vreg {<<, >>} _imm16	
vector	vmani	vvsl, vvsr	[INSTR] [vreg]	_vreg = { _vreg(2:end) & 0, 0 & _vreg(1:end-1) }	
vector	vmani	vset, vseti	[INSTR] [{_reg, _imm10}] [vreg]	_vreg(1:end) = { _reg, _imm10 }	
vector	vmani	vedit, vediti	[INSTR] [imm7] [{_reg, _imm10}] [vreg]	_vreg(imm7) = { _reg, _imm10 }	
vector	vmani	vgrep	[INSTR] [reg] [src_vreg] [dest_reg]	_dest_reg = _src_vreg(_reg)	

Fig. 15

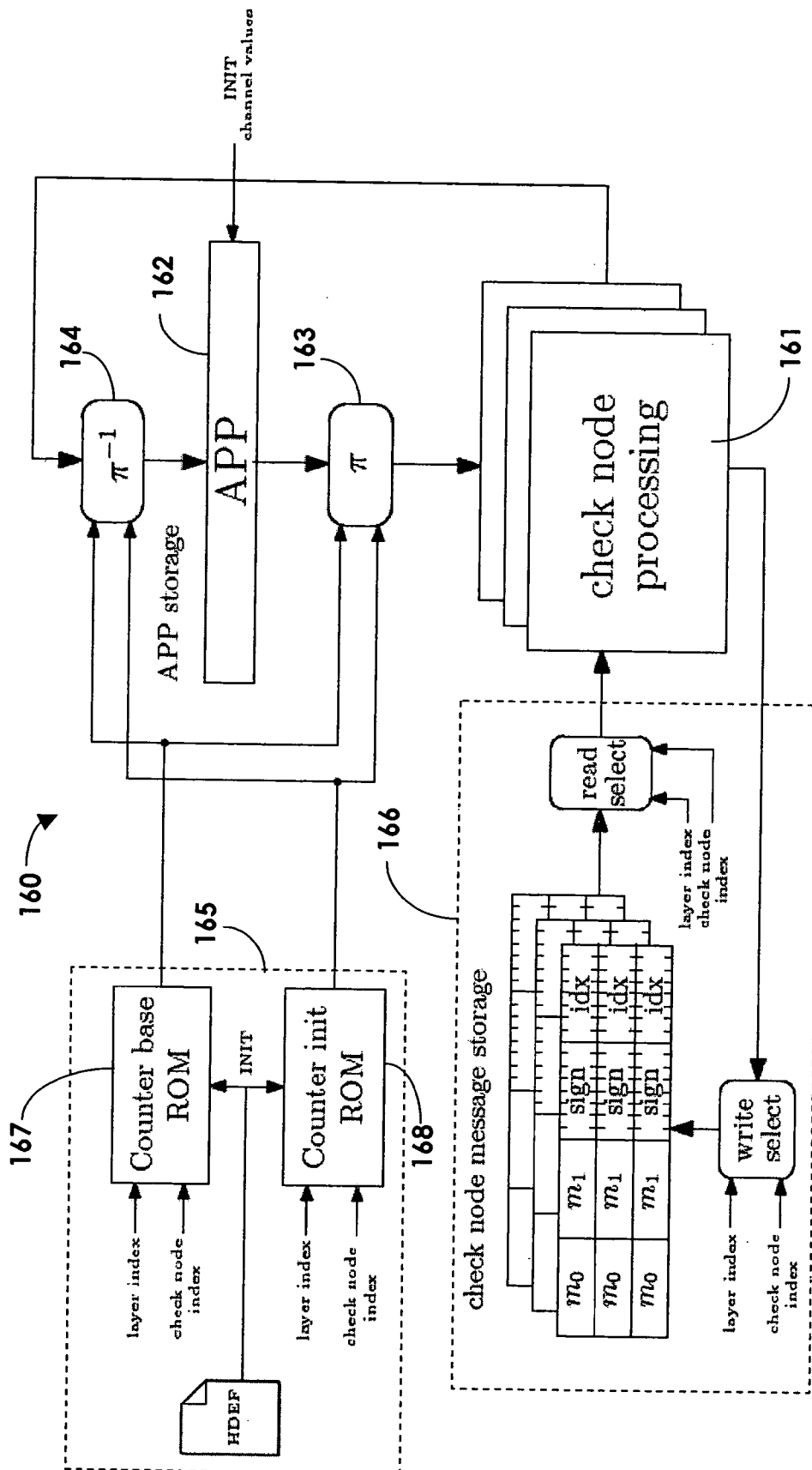


Fig. 16

13/18

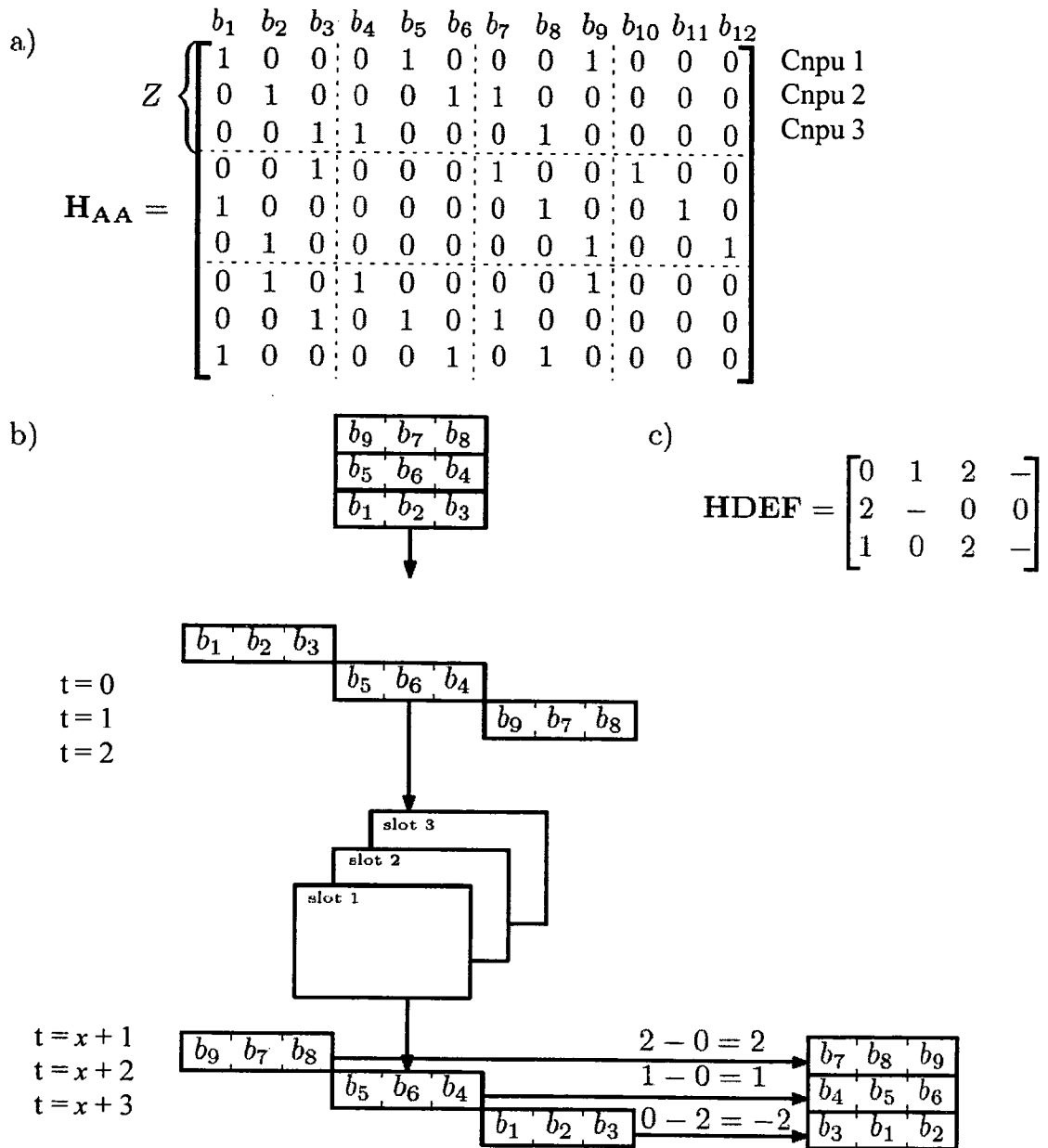


Fig. 17

14/18

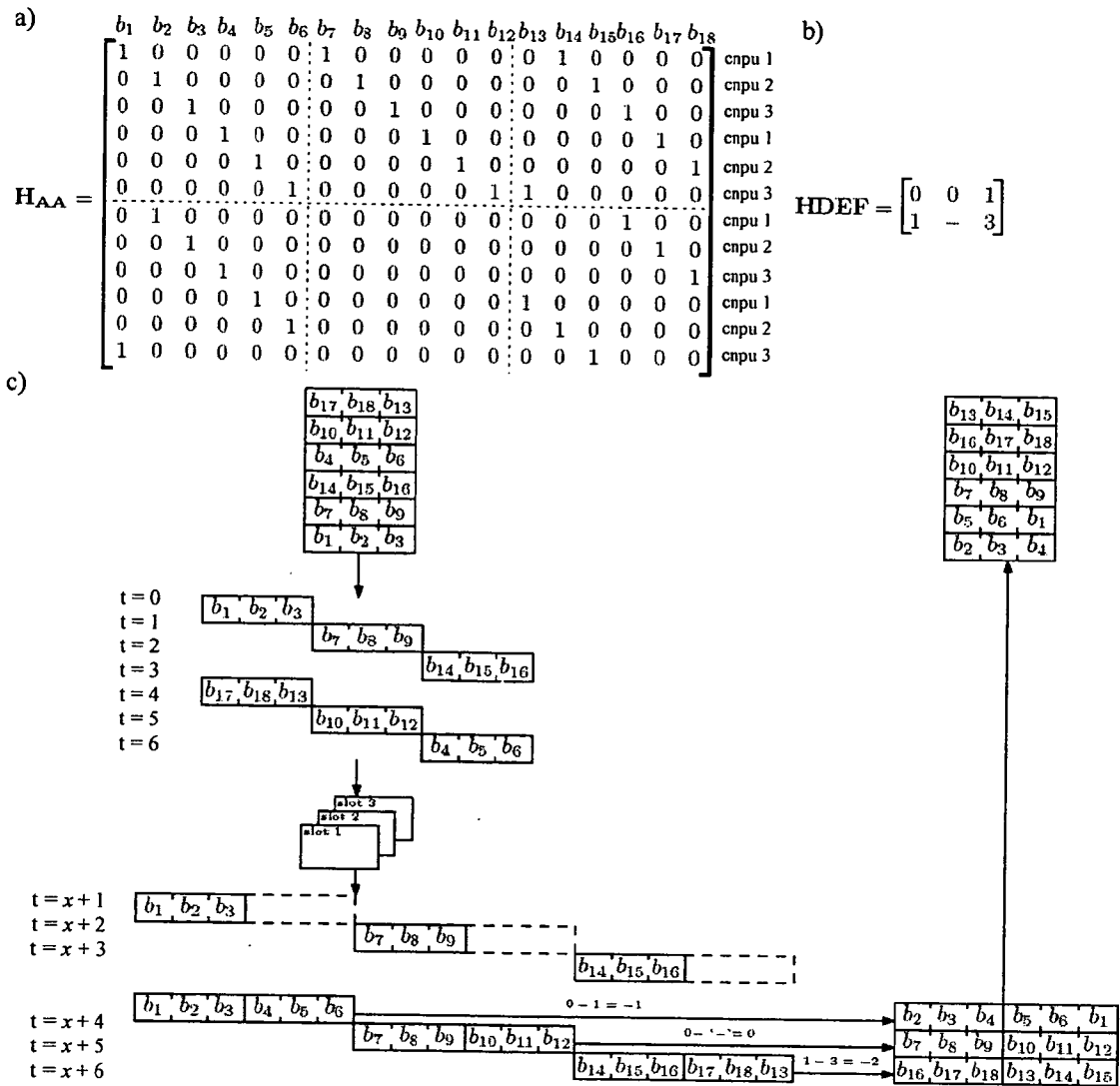


Fig. 18

15/18

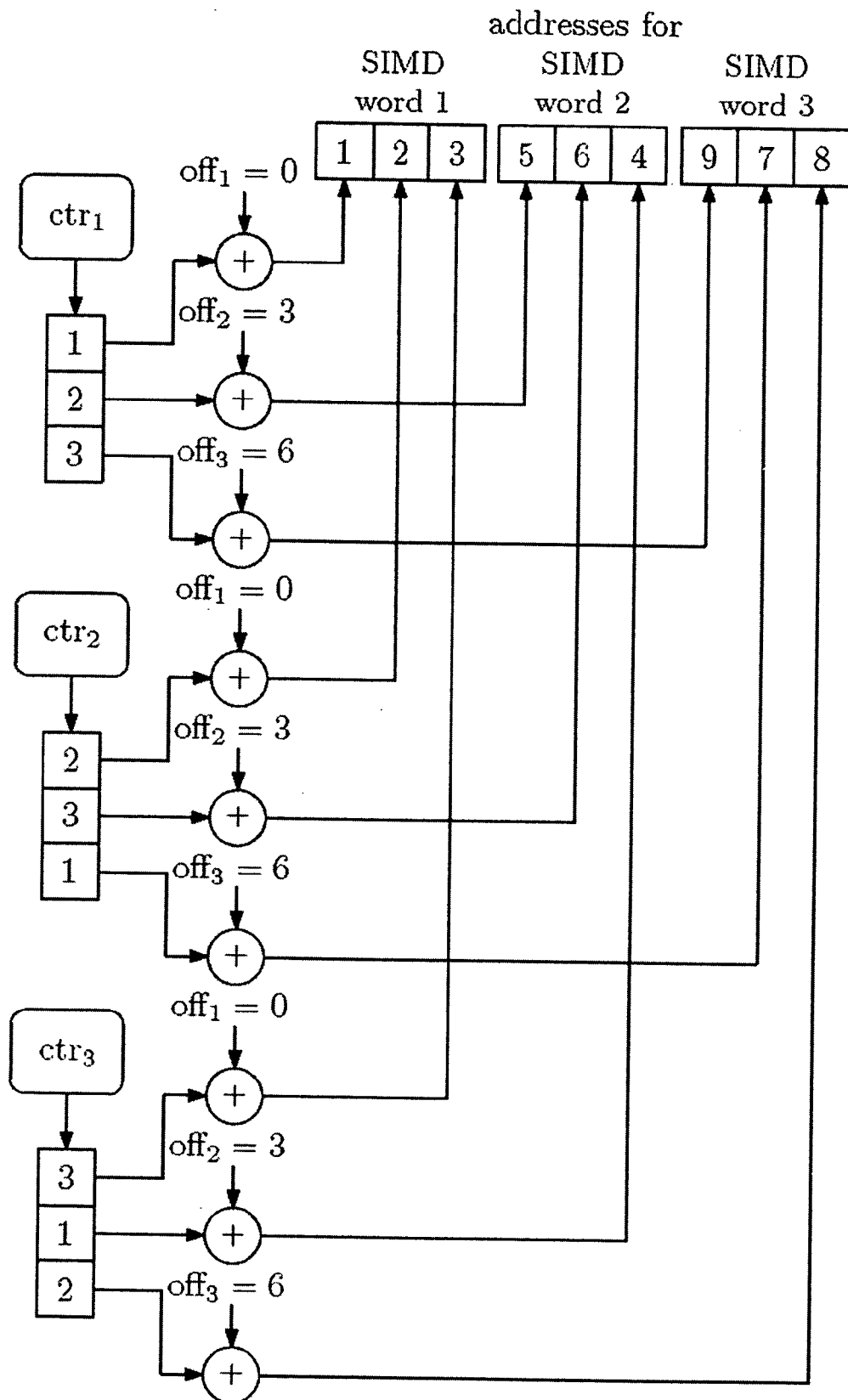


Fig. 19

SUBSTITUTE SHEET (RULE 26)

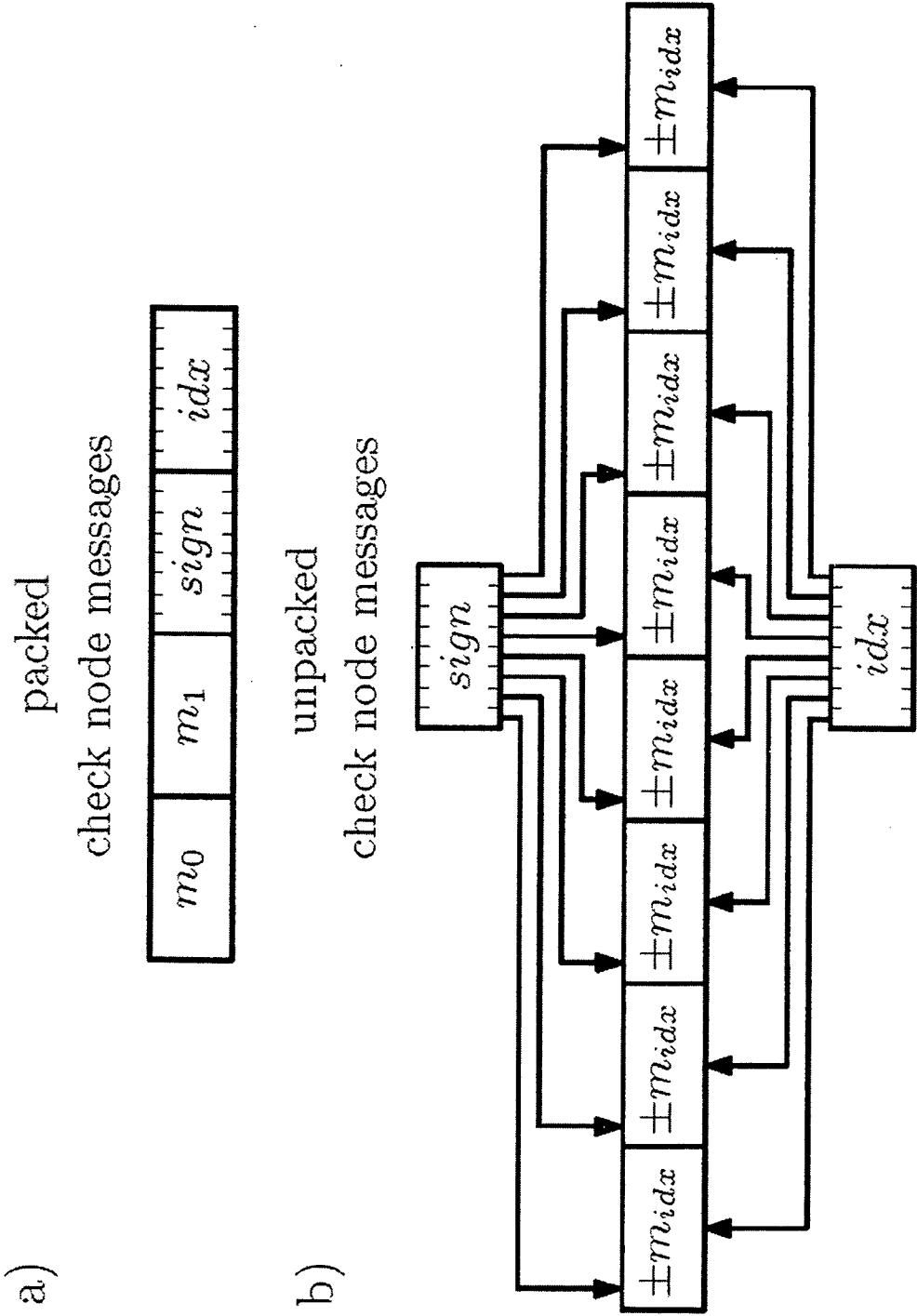


Fig. 20

17/18

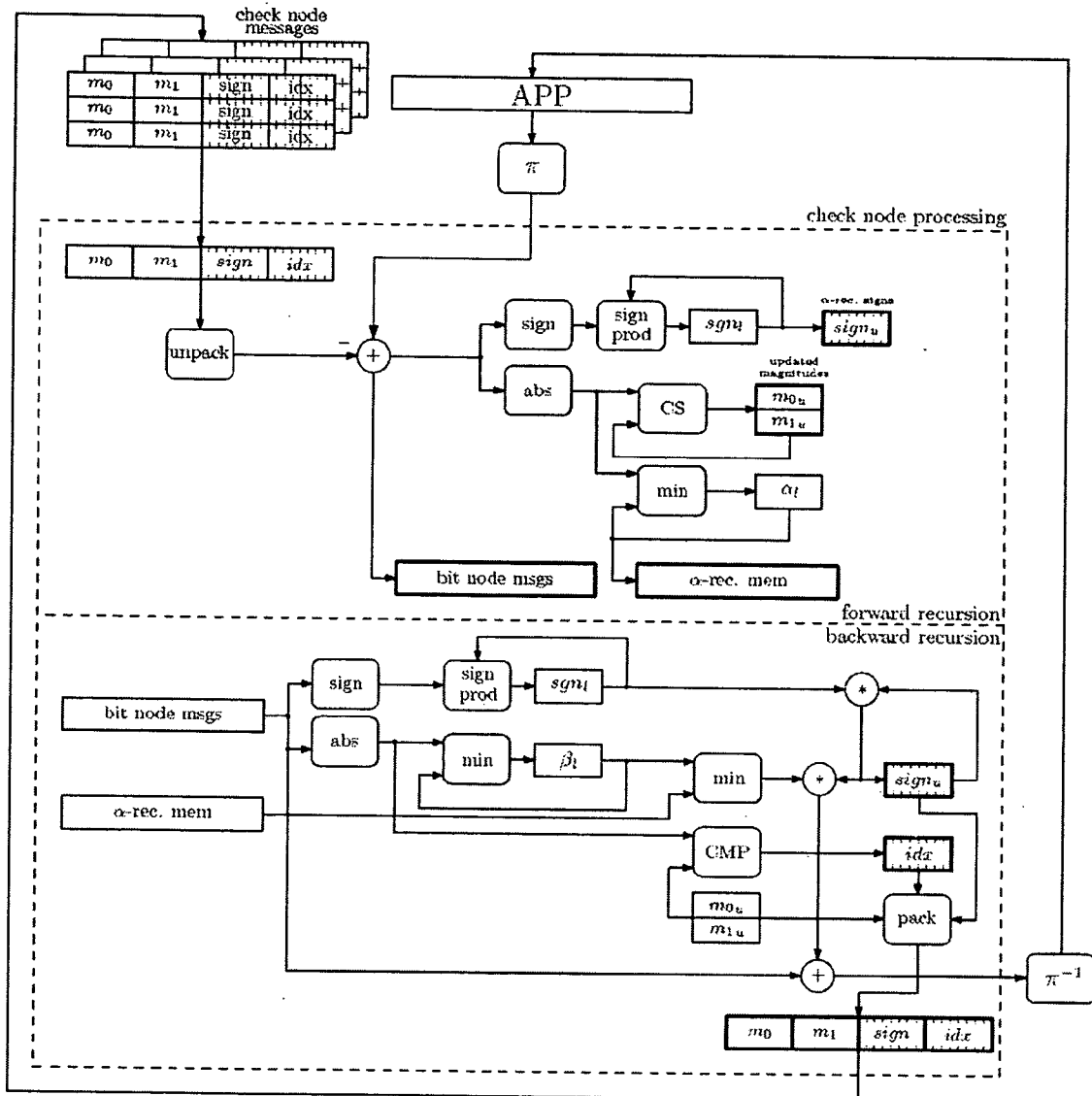


Fig. 21

18/18

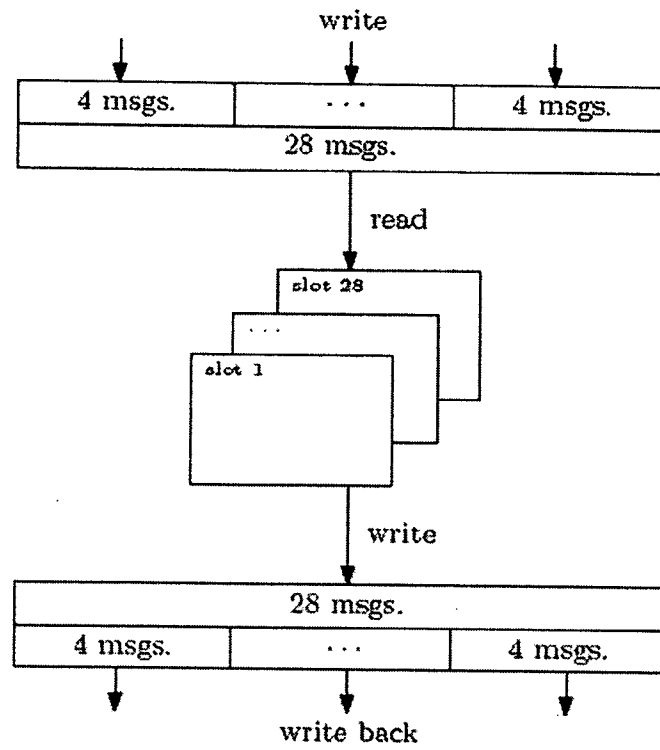


Fig. 22