(54) Title: PARTIAL BITWISE PERMUTATIONS
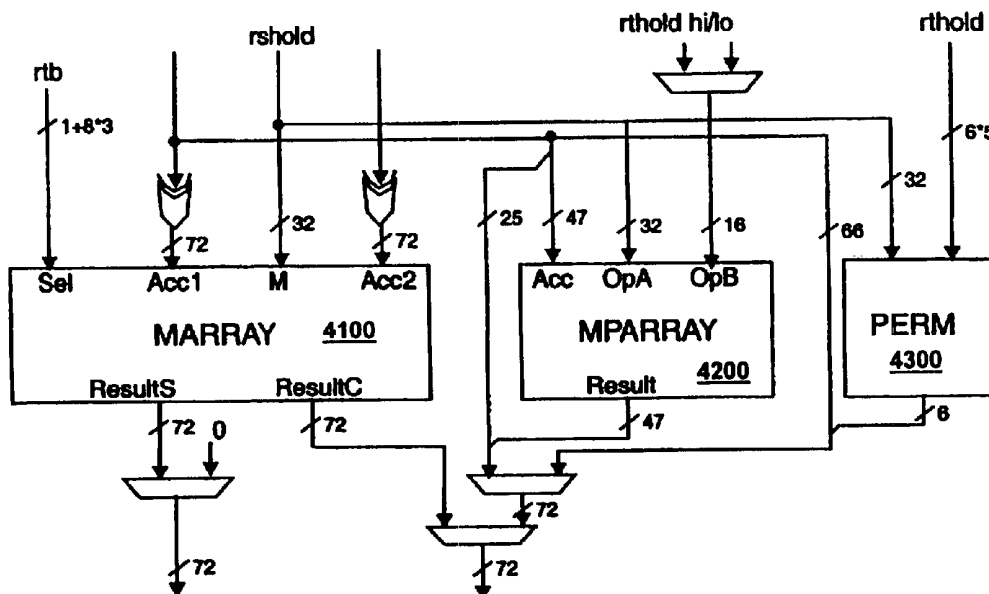
(57) Abstract: Partial bitwise permutation instructions are provided in a microprocessor or microcontroller. Partial bitwise permu-
tations may be specified by one or more of the following: a destination specifier, a previous partial value source, a destination subset
specifier, and a control specifier. The array unit (3030) within the microprocessor or microcontroller includes two parallel multipliers
(Marray 4100 and MParray 4200) and permutation logic (4300). The first array, Marray (4100), performs arithmetic multiplication.
Marray (4100) uses Acc1 (3031) Acc2 (3032), M (3033) and sel (3034) as inputs and produces a ResultC and ResultS as outputs.
The second array MParray 4200 performs binary polynomial multiplication. Permutation logic (4300) is used to perform various
permutations on low order bits of RShold based on the value in RThold (3012).

PARTIAL BITWISE PERMUTATIONS

TECHNICAL FIELD

This invention relates to a technique for performing partial bitwise permutations in a microprocessor.

BACKGROUND

5   Reduced instruction set computer (RISC) architectures were developed as industry trends tended towards larger, more complex instruction sets. By simplifying instruction set designs, RISC architectures make it easier to use techniques such as pipelining and caching, thus increasing system performance.

RISC architectures usually have fixed-length instructions (e.g., 16-bit, 32-bit, 10   or 64-bit), with few variations in instruction format. Each instruction in an instruction set architecture (ISA) may have the source registers always in the same location. For example, a 32-bit ISA may always have source registers specified by bits 16-20 and 21-25. This allows the specified registers to be fetched for every instruction without requiring any complex instruction decoding.

15   SUMMARY

Cryptographic systems ("cryptosystems") are increasingly used to secure transactions, to encrypt communications, to authenticate users, and to protect information.

Many secret-key cryptosystems, such as the Digital Encryption Standard 20   (DES), are relatively simple computationally and frequently reducible to hardware solutions performing sequences of XORs, rotations, and permutations on blocks of data.

In one general aspect, an instruction for performing partial bitwise permutations is provided in an instruction set architecture. The instruction includes 25   an opcode identifying the instruction as a partial permutation instruction, and a permutation operation specification. The permutation operation specification includes a destination specifier identifying a destination register, a previous partial value source specifier, a destination subset specifier, and a control specifier. The destination

1

subset specifier identifies one or more destination bits of the destination register, and the control specifier identifies a source for each of the identified destination bits. The instruction is processed by performing a partial bitwise permutation defined by the permutation operation specification.

5   Implementations may include a destination specifier that either implicitly or explicitly identifies the destination register. The destination register may be specified as an accumulator within a multiply/divide unit of a microprocessor and/or a general-purpose register. Likewise, the partial value source specifier may implicitly or explicitly identify a previous partial value source register. The previous partial value 10   source register may be specified as an accumulator and/or a general-purpose register. Additionally, the destination register and the previous partial value source register may be identified as the same register.

In some implementations, the destination subset specifier identifies a contiguous block of bits within the destination register. The contiguous block of bits 15   may include the least significant bit of the destination register. Various implementations may include 1-6 or more bits within the contiguous block of bits.

The control subset specifier may include one or more source bit identifiers, and may include a mask and a default bit. Additionally, the control subset specifier may be stored as a field within an instruction or may be stored in a general-purpose 20   register.

Implementations may include providing a partial permutation instruction within a RISC instruction set.

The details of one or more implementations are set forth in the accompanying drawings and the description below. Other features and advantages will be apparent 25   from the description and drawings, and from the claims.

DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram of an exemplary five-stage pipeline that may be used in a RISC architecture.

FIG. 2 is a block diagram of a processor core including an execution unit and a 30   multiply/divide unit.

FIG. 3 is a diagram of data paths in an implementation of a multiply/divide unit supporting binary polynomial arithmetic.

FIG. 4 is a block diagram of multiplier arrays supporting arithmetic and binary polynomial multiplication in one implementation.

FIG. 5 is a block diagram of an arithmetic multiplier array that may be used in the implementation shown in FIG. 4.

FIG. 6 is a block diagram of a binary polynomial multiplier array that may be used in the implementation shown in FIG. 4.

FIG. 7A is a timing diagram showing the operation of 32-bit by 16-bit multiplies in one implementation.

FIG. 7B is a timing diagram showing the operation of 32-bit by 32-bit multiplies in one implementation.

FIG. 7C is a timing diagram showing the operation of divisions in one implementation.

FIG. 8 is a finite state machine implementing steps for performing multiply instructions.

FIG. 9 is a finite state machine implementing steps for performing division instructions.

FIG. 10A is an instruction encoding of an exemplary partial permutation instruction having four operands.

FIG. 10B is an exemplary control word encoding format for use as an operand to a partial permutation instruction such as shown in FIG. 10A.

FIGS. 11A and 11B are instruction encodings of exemplary partial permutation instructions not using a control word operand.

FIG. 12A is an instruction encoding of an exemplary partial permutation instruction that implicitly uses registers of a multiply/divide unit.

FIG. 12B is an exemplary control word encoding format for use as an operand to a partial permutation instruction such as shown in FIG. 12A.

FIG. 13A is an instruction encoding of another partial permutation instruction that implicitly uses registers of a multiply/divide unit.

FIG. 13B is an exemplary control word encoding format for use as an operand to a partial permutation instruction such as shown in FIG. 13A.

DETAILED DESCRIPTION

Some cryptographic operations, such as the Digital Encryption Standard (DES) (as well as several of the candidates for the Advanced Encryption Standard (AES) to replace DES), perform some degree of bitwise permutation of data values. These operations, which map poorly to conventional microprocessor instruction set architectures, often may be implemented efficiently in hardware. However, the National Institute of Standards and Technology is in the process of creating the new AES standard. Therefore, it is uncertain which algorithms will be used in the future. For at least this reason, it is desirable to provide a microprocessor with support for bitwise permutations that may be used to increase the performance of cryptographic algorithms such as block ciphers.

Referring to FIG. 1, an exemplary microprocessor architecture that may be used to implement polynomial multiplication includes a five-stage pipeline in which each instruction is executed in a fixed amount of time, such as, for example, four clock cycles. The execution of each instruction is divided into five stages: instruction fetch (IF) stage 1001, register read (RD) stage 1002, arithmetic/logic unit (ALU) stage 1003, memory (MEM) stage 1004, and write back (WB) stage 1005. In the IF stage 1001, a specified instruction is fetched from an instruction cache. A portion of the fetched instruction is used to specify source registers that may be used in executing the instruction. In the read registers (RD) stage 1002, the system fetches the contents of the specified source registers. These fetched values may be used to perform arithmetic or logical operations in the ALU stage 1003. In the MEM stage 1004, an executing instruction may read/write memory in a data cache. Finally, in the WB stage 1005, values obtained by the execution of the instruction may be written back to a register.

4

Because some operations, such as floating point calculations and integer multiply/divide, cannot be performed in a single clock cycle, some instructions merely begin execution of an instruction. After sufficient clock cycles have passed, another instruction may be used to retrieve a result. For example, when an integer

5 multiply instruction takes five clock cycles, one instruction may initiate the multiplication calculation, and another instruction may load the results of the multiplication into a register after the multiplication has completed. If a multiplication has not completed by the time a result is requested, the pipeline may stall until the result is available.

10 Referring to FIG. 2, an exemplary RISC architecture is provided by way of example. The processor core 2000 (also referred to as a "microprocessor core") includes the following: an execution unit 2010, a multiply/divide unit (MDU) 2020, a system control coprocessor (CP0) 2030, a memory management unit 2040, a cache controller 2050, and a bus interface unit (BIU) 2060. In FIG. 2, MDU 2020 is a

15 combined multiply/divide unit; however, other implementations provide separate multiply and divide units.

Execution unit 2010 is the primary mechanism for executing instructions within processor core 2000. Execution unit 2010 includes a register file 2011 and an arithmetic logic unit (ALU) 2012. In one implementation, the register file 2011

20 includes 32 32-bit general-purpose registers that may be used, for example, in scalar integer operations and address calculations. The register file 2011 includes two read ports and one write port and may be fully bypassed to minimize operation latency in the pipeline. ALU 2012 supports both logical and arithmetic operations, such as addition, subtraction, and shifting.

25 The MDU 2020 includes three registers (ACX 2021, HI 2022, and LO 2023) that may be used for various operations. In accordance with one implementation, these three registers may be used together to hold up to a 72-bit value. In one implementation, LO register 2023 and HI register 2022 are each 32 bits wide and function as dedicated output registers of MDU 2020. In one implementation, ACX

30 register 2021 provides 8 bits of additional integer precision beyond those provided by the HI/LO register pair. The precise number of bits is implementation dependent, with

the preferred minimum size being 2 bits. For processors with 32 bit data paths, the preferred maximum size of the ACX register is 32 bits. In contrast, for processors with 64 bit data paths, the preferred maximum size of the ACX register is 64 bits. Hence, in a processor with 32-bit wide HI and LO registers, the combination of ACX/HI/LO can hold a 64-or-more-bit concatenated value. MDU 2020 may be used to perform various operations including some or all of the following instructions: DIV, DIVU, MADD, MADDU, MFHI, MFLO, MSUB, MSUBU, MTHI, MTLO, MUL, MULT, MULTU, MFLHXU, MTLHX, MADDP, MULTP, and PPERM.

The instructions MUL, MULT, and MULTU may be used to multiply two 32-bit numbers together. The result is stored in a specified register for MUL, and in the HI/LO registers for MULT and MULTU. For example, "MUL $7, $6, $5" multiplies the contents of registers $6 and $5 together and stores the result in register $7. The instruction "MULT $6, $5" multiplies the contents of registers $6 and $5 together and stores the result in the HI/LO registers. The MULTU instruction performs the same operation as MULT with MULTU applying to unsigned operands and MULT applying to signed operands. Additionally, the MULTU instruction clears the ACX register to all zeros.

The instructions DIV and DIVU perform division operations and store the results in the ACX/HI/LO registers. For example, "DIV $6, $5" divides the contents of register $6 by the contents of register $5 and stores the resulting remainder and quotient in the HI/LO registers. The DIVU instruction performs the same operation on unsigned operands.

The instructions MSUB, MSUBU, MADD, and MADDU may be used to multiply the contents of two registers and then add or subtract the resulting product with the contents of the ACX/HI/LO registers. For example, "MSUB $6, $5" multiplies the contents of registers $6 and $5 together, subtracts the result of the multiplication from the contents of the ACX/HI/LO registers, and then stores the resulting value in the ACX/HI/LO registers. The MADD instruction similarly multiplies the contents of two registers, adds the result to the ACX/HI/LO registers, and stores the result in the ACX/HI/LO registers. The MSUBU and MADDU perform the analogous operations to unsigned operands. In some implementations,

6

the ACX register is not used for some operations and the contents of the ACX register following such operations may be undefined.

The MFHI, MFLO, MTHI, MTLO, MFLHXU, and MTLHX are used to move data between the ACX/HI/LO registers and general purpose registers. The first instruction, MFHI, loads the contents of the HI register into a general purpose register. For example, "MFHI $5" loads the contents of the HI register into register $5. Similarly, MFLO loads the contents of the LO register into a general purpose register. Conversely, the instructions MTHI and MTLO are used to load the contents of a general purpose register into the HI or LO registers. For example, "MTHI $5" loads the contents of register $5 into the HI register.

In one implementation, the content of the ACX register is not directly accessible. To indirectly access the ACX register, the values stored in the ACX/HI/LO registers may be shifted to the left or right. For example, "MFLHXU $5" shifts contents of the ACX, HI, and LO registers to the right by one register position, loading the contents of the LO register into register $5. Thus, after performing the operation, the ACX register is zero, the HI register contains the previous contents of the ACX register, the LO register contains the previous contents of the HI register, and register $5 contains the previous contents of the LO register. Because the contents of the 8-bit ACX register are loaded into a 32-bit register, the 8-bit value may be zero-extended to 32-bits before loading the HI register.

The MTLHX performs the inverse operation. For example, "MTLHX $5" loads the ACX register with the previous contents of the HI register, loads the HI register with the previous contents of the LO register, and loads the LO register with the contents of register $5.

The PPERM operation performs permutations as specified in a register, storing the result in the ACX/HI/LO registers. For example, "PPERM $5, $6" causes the ACX/HI/LO registers to be shifted 6-bits to the left. Then, low-order six bits are selected from register $5 as specified by register $6. In particular, the 32-bit contents of register $6 are used to select which bits of register $5 will be used to fill the low-order bits of the ACX/HI/LO registers. Since there are 32 bits in register $5, 5 bits are needed to specify a specific one of the 32 bits. For example, "01101" is binary for

the number 13. Thus, these five bits may specify bit 13. Similarly, "00000" is binary for 0 and "11111" is binary for 31. Thus, any one of the 32 bits may be specified using a 5-bit specifier, and 6 bits may be specified using 30 bits (i.e., 6 5-bit specifiers).

5   Register $6 may specify the bits of register $5 used to fill the low-order bits of ACX/HI/LO as follows: bits 0-4 are used to specify the source of bit 0, bits 5-9 are used to specify bit 1, bits 10-14 are used to specify bit 2, bits 15-19 are used to specify bit 3, bits 20-24 are used to specify bit 4, and bits 25-29 are used to specify bit 5. The remaining bits, 30-31, may be unused. Thus, the instruction is performed using the
10   specifiers as described to fill the lowest 6 bits of the LO register with the specified bits from the register $5.

         Finally, MULTP may be used to perform binary polynomial multiplication and MADDP may be used to perform binary polynomial multiplication with the result added to the ACX/HI/LO registers. These operations are analogous to MULT and
15   MADD, but operate on binary polynomial operands.

         The polynomial operands of MULTP and MADDP are encoded in 32-bit registers with each bit representing a polynomial coefficient. For example, the polynomial "$x^4 + x + 1$" would be encoded as "10011" because the coefficients of x^3 and x^2 are "0" and the remaining coefficients are "1". The MULTP instruction
20   performs binary polynomial multiplication on two operands. For example,

$$(x^4 + x + 1)(x + 1) = x^5 + x^4 + x^2 + 2x + 1.$$

         Reducing the polynomial modulo two, yields $x^5 + x^4 + x^2 + 1$. If the polynomials are encoded in the binary representation above, the same multiplication may be expressed as $(10011)(11) = 110101$.

25   The MADDP instruction performs multiplication just as MULTP, and then adds the result to the ACX/HI/LO registers. Polynomial addition may be performed using a bitwise XOR. For example, the binary polynomial addition $(x^4 + x + 1) + (x + 1)$ yields $x^4 + 2x + 2$. Reducing the coefficients modulo 2 yields x^4, which may be expressed as "10000".

Referring to FIG. 3, MDU 2020 receives two 32-bit operands, RS and RT. Using these operands, MDU 2020 performs a requested operation and stores a result in registers ACX 2021, HI 2022, and LO 2023. Major data paths that may be used to perform these operations are shown in FIG. 3. The RShold register 3010 and

5   the RThold register 3012 are used to hold the RS and RT operands. Multiplexers 3020, 3022, and 3024 are used to select whether to use the RS and RT operands directly or to use the values stored in the RShold register 3010 and the RThold register 3012. Additionally, multiplexer 3022 may be used to select between the low-order and high-order bits of RT or the value stored in the RThold register 3012.

10   The RThold register 3012 is connected to multiplexer 3022. Multiplexer 3022 produces a 16-bit result by selecting the high-order bits of RThold 3012, the low-order bits of RThold 3012, the high-order bits of the RT operand, or the low-order bits of the RT operand. The output from multiplexer 3022 is processed by Booth recoder 3040 and stored in register RTB 3042. Booth recoding is a technique

15   that permits the multiplier array to treat signed and unsigned operands the same. The output of register RTB 3042 becomes the input SEL 3034 to array unit 3030.

Array unit 3030 is used to perform arithmetic and binary polynomial multiplication as described below with reference to FIG. 4. Array unit 3030 takes as inputs ACC1 3031, ACC2 3032, M 3033, SEL 3034, and RThold 3012. Inputs ACC1

20   3031 and ACC2 3032 are accumulated results used for operations that perform a multiplication and add or subtract the resulting value from an accumulated result. The inputs SEL 3034 (determined by register RTB 3042) and M 3033 (determined by register RShold 3010) form the operands for arithmetic operations. The inputs RThold 3012 (or the high-order or low-order bits of RThold 3012) and M 3033

25   (determined by RShold 3010) form operands for polynomial operations and permutations. Combinations of these inputs are used to perform various calculations as described in detail below.

Array unit 3030 also includes two outputs, ResultC 3035 and ResultS 3036. In performing arithmetic operations, carry-save adders (CSAs) may be used to

30   build a multiplication array. Carry-save adders calculate sums and carries separately to produce two outputs. Thus, ResultC 3035 and ResultS 3036 represent,

respectively, the carry and the sum outputs of a CSA multiplier array. In one implementation, ACC1 3031, ACC2 3032, ResultC 3035, and ResultS 3036 are each 72 bits long and the remaining inputs are at most 32 bits long. Inputs ACC1 3031 and ACC2 3032 may be selected using multiplexers 3037 and 3038.

5   Multiplexers 3050 and 3052 are used to select values as inputs to registers CPAA 3054 and CPAB 3056. For example, multiplexer 3050 may be used to select between ResultC 3035, the output of CPA 3058, or the output of multiplexer 3020 (i.e, operand RS or the output of RShold 3010). Similarly, multiplexer 3052 may be used to select between ResultS 3036, the value 0, and the output of multiplexer 10   3024 (i.e., operand RT or the output of RThold 3012). Registers CPAA 3054 and CPAB 3056 store the inputs to carry propagate adder (CPA) 3058. CPA 3058 may be used to complete multiplication operations (multiplies) and to perform iterative division operations (divides) as discussed below.

Register RDM 3060 stores the result of CPA 3058. Finally, multiplexers 3070 15   and 3072 select which values form the result to be loaded into registers ACX, HI, and LO. Multiplexer 3070 may be used to select the ACX/HI/LO registers, RDM 3060, or the result of CPA 3058. Multiplexer 3072 may be used to instead load various permutations of the result selected by multipexer 3070. Multiplexer 3072 is used to perform various rotations and loads of the ACX/HI/LO registers by permitting 20   selection of the following values (forming 72-bit values when concatenated): (1) ahl, the 72-bit output of multiplexer 3070; (2) arl, the 8 high-order bits of multiplexer 3070, the contents of RShold 3010, and the 32 low-order bits of multiplexer 3070; (3) ahr, the 40 high-order bits of multiplexer 3070 and the contents of RShold 3010; (4) hlr, the 40 low-order bits of multiplexer 3070 and the contents of RShold 3010; and 25   (5) 0ah, the 40 high-order bits of multiplexer 3070 (with 32 leading zeros).

Some operations cause the values stored in the result registers ACX, HI, and LO to be overwritten. For this reason, a separate result register 3080 may be provided to store the high-order and low-order result without the accumulator ACX.

The data path described below includes six major parts: (1) input 30   registering and selection; (2) Booth recoding; (3) multiplier arrays and permutation

10

logic; (4) a carry propagate adder; (5) result registering and selection; and (6) a separate 32-bit output register for presenting results.

Input registering and selection is performed using the RShold and RThold registers to hold the RS and RT operands. Multiplexers select whether to use these 5 operands directly or to use the registered versions. Booth recoding is performed on half of the RT operand at a time to provide inputs to the multiplier arrays and permutation logic.

Booth recoding is a technique that permits the multiplier array to treat signed and unsigned operands the same. This technique "recodes" operands as a subtraction 10 from the next highest power of two. For example, 7 may be Booth recoded as follows: $8 - 1 = 1000_2 - 0001_2 = 100\overline{1}$, wherein $\overline{1}$ represents -1. Hennessy and Patterson describe Booth recoding in Appendix A of "Computer Architecture: A Quantitative Approach," which is incorporated by reference in its entirety for all purposes.

15 One array of array unit 3030 performs arithmetic multiplication and one array of array unit 3030 performs binary polynomial multiplication. In one implementation, both arrays are 32 bits by 16 bits (32x16) and are used once or twice depending on the size of the RT operand (i.e., an appropriate array is used once when RT is 16 bits long and twice when RT is 32 bits long). The CPA may be used to complete multiplies and 20 to perform iterative divides. Other implementations may include faster mechanisms for performing divides.

The arithmetic multiplication array may be implemented using any of the techniques described by Hennessy and Patterson in the incorporated "Computer Architecture: A Quantitative Approach." For example, Appendix A of Hennessy and 25 Patterson describes several ways to speed up arithmetic multipliers. Any of the described techniques may be used as a basis for the polynomial multiplication extensions described below.

Referring to FIG. 4, array unit 3030 includes two parallel multipliers (Marray 4100 and MParray 4200) and permutation logic 4300. The first array, Marray 4100, 30 performs arithmetic multiplication as described below with reference to FIG. 5.

11

Marray 4100 uses ACC1 3031, ACC2 3032, M 3033, and SEL 3034 as inputs and
produces a ResultC and a ResultS as outputs. The second array, MParray 4200,
performs binary polynomial multiplication as described below with reference to FIG.
6. MParray 4200 uses the low-order bits of RThold 3012 or the high-order bits of

5   RThold 3012 (as selected by a multiplexer 4305), RShold 3010, and ACC1 3031 as
inputs, and produces a Result as an output. Finally, permutation logic 4300 is used to
perform various permutations on the low-order bits of RShold 3010 based on the
value stored in RThold 3012.

A multiplexer 4310 selects between the ResultS output of Marray 4100 and a

10   zero to produce ResultS 3036. Multiplexers 4315 and 4320 select between the
ResultC produced by Marray 4100, the combination of 25 bits of ACC1 and the 47
bits of the Result produced by MParray 4200, and the results produced by permutation
logic 4300 to produce ResultC 3035.

Referring to FIG. 5, Marray 4100 is a 32-bit by 16-bit Wallace tree multiplier

15   array that has been modified to support the addition of two 72-bit wide operands
ACC1 and ACC2. The ACC1 and ACC2 operands hold a carry-save representation
of a 72-bit value. Because additions are already performed to carry out
multiplications (i.e., by the carry-save adders (CSAs)), an additional adder may be
included to allow ACC1 and ACC2 to be added to intermediate results of

20   multiplications. Marray 4100 generates a 72-bit wide result in a carry-save
representation. Since 32x16 bits are processed per cycle, two passes through the array
are required for 32x32 bit multiplies.

Marray 4100 is implemented as a Wallace tree built from arrays of CSAs. The
width of these arrays may vary. This design may be implemented using an automated

25   place and route rather than using data path style. Because the accumulate value from
the previous array pass is input late into the array, the accumulate value does not need
to come directly from a register. Booth recoding is performed using the method of
overlapping triplets to more efficiently process multiplications. The output of Booth
recoding tells whether to add operand M multiplied by -2, -1, 0, 1, or 2 for each power

30   of 4. The multiplexers on the top-level CSA inputs are used to select the
corresponding multiple of M.

Marray 4100 accumulates eight products from the Booth recoding plus one special partial product. The latter may be used for 32-bit unsigned calculations using the "0" and "1x" choices from the multiplexers. Within the Wallace tree, operands may be sign-extended to properly accumulate 2's complement results.

5   Referring to FIG. 6, binary polynomial-based multiplication operations are processed similarly to corresponding unsigned arithmetic operations. In one implementation, MParray 4200 is a 32x16 bit array that also performs an addition using exclusive-or (XOR) on an operand, for example, ACC1. As with Marray 4100, 32x16 bits are processed per cycle and two passes through the array may be used for

10   32x32 multiplies. In the first cycle, ACC1 is zero (for a MULTP operation) or the previous result (for a MADDP operation). In a second cycle, ACC1 is the high order bits of the output from the first cycle.

MParray 4200 multiplies two operands (e.g., OpA and OpB) using an array with each row formed by taking the AND of OpA and a bit of OpB. For example, the

15   first row is the logical AND of OpA and bit 0 of OpB. Row two is the logical AND of OpA and bit 1 of OpB. The result of each successive row is shifted one bit to the left. The final result is formed by taking the exclusive-or (XOR) of each column. Because a bitwise XOR may be used to perform addition in binary polynomial arithmetic, an accumulator row may be added to array MParray 4200 to support

20   instructions such as MADDP.

Referring again to FIG. 1, MDU 2020 starts a computation in the first cycle of the execute stage of the pipeline 1003. If the calculations complete before the instruction has moved past the memory stage 1004 in the pipeline, then the result is held at that point. If the operation completes when the instruction has been moved

25   past the memory stage 1004 in the pipeline, then the instruction has been committed and the results are written directly to the ACX/HI/LO registers.

The MDU 2020 is decoupled from the environment pipeline; it does not stall with the environment. That is to say the MDU 2020 will continue its computation during pipeline stalls. In this way, multi-cycle MDU operations may be partially

30   masked by system stalls and/or other, non-MDU instructions.

13

FIG. 7A shows the pipeline flow through MDU 2020 for 32x16 bit multiplies. RS and RT arrive late, so the first cycle may be used for Booth recoding. The second cycle is where the array is run and the third cycle is where the CPA 3058 completes the computation. Because the results are always accessible to reads by MFxx instructions, 32x16 multiplies may be run without stalls. A 32x16 MUL, which returns the result directly to a general purpose register (GPR), may stall for one cycle.

Referring to FIG. 7B, for 32x32 bit multiplies, the array is used twice, which adds one extra clock cycle to the 32x16 bit multiplications. As the first array pass is completing for the first portion of operand RT, Booth recoding is performed on the second portion of the operand. Thus, the Booth recoded portion of RT is available to begin the second pass through the array immediately after the first pass is complete. The multiplication result is then calculated using CPA 3058.

Referring to FIG. 7C, a simple non-restoring division algorithm may be used for positive operands. The first cycle is used to negate RS, if needed. For timing reasons, this cycle is taken even if RS is positive. Following that, 32, 25, 18, or 10 cycles of iterative add/subtract operations are performed. The actual number is based on the amount of leading zeros on the positive RS operand. A final remainder adjust may be needed if the remainder was negative. For timing reasons, this cycle is taken even if the remainder adjust is not needed. Finally, sign adjustment is performed if needed on the quotient and/or the remainder. If both operands are positive, this cycle may be skipped.

In one implementation, target applications demand fast division. Many techniques may be used to increase the performance of division. For example, the Sweeney, Robertson, and Tocher (SRT) algorithm or some variation thereof may be used.

Referring to FIG. 8, multiplication operations are implemented using a finite state machine. Multiplication begins in IDLE state 8010. The multiplier stays in the idle state until the start signal is asserted. Then, the multiplier transitions to either the ARR1 state 8020 or the ARR2A state 8030 depending on whether operand RT contains a 32-bit or 16-bit value. If a 16-bit value is stored in RT, then the system transitions to state ARR2A 8030 where the first array pass is run. The multiplier then

transitions to state ARR2B 8040 where the second array pass is run. If a 16-bit value is stored in operand RT, the multiplication is run through the array unit in state ARR1 8020.

In this implementation, the multiplier is pipelined. One multiplication may be run through the array unit and another through the CPA. Thus, the multiplier either transitions from ARR1 8020 or ARR2B 8040 to state CPA 8050 if there is no additional multiplication to perform, or begins a second multiplication. If no additional multiplication is needed, the multiplier is run through CPA 8050 and then either returns to IDLE 8010 or begins a new multiplication as discussed above.

If a second multiplication is ready to be performed when the first multiplication is ready to be run through the CPA, then the multiplier either transitions to CPA1 8060 (for a 32x16 multiplication) or CPA2A 8070 (for a 32x32 multiplication). In state CPA1 8060, the first multiplication is run through the CPA and the second multiplication is run through the array unit. The multiplier then transitions to state CPA 8050 to finalize the second multiplication.

If the second multiplication is a 32-bit multiplication, then in state CPA2A 8070 the first multiplication is run through the CPA and the second multiplication is run through the array unit. The multiplier then transitions to state ARR2B 8040 to complete the 32x32 multiplication. This pipelined approach allows 32x16 multiplications to be issued every clock cycle, with a two-cycle latency. Also, 32x32 multiplications may be issued every other clock cycle, with a three-cycle latency.

Referring to FIG. 9, iterative division operations may be implemented using a finite state machine. In one implementation, the MDU begins in IDLE state 9010. When a signal is received to begin a division operation, the MDU either transitions to DIV1 9020 if the operation is signed or DIV1U 9030 if the operation is unsigned. States DIV1 9020 and ERLY 9040 are used to prepare signed operands for division, and adjust the signs as necessary. States DIV1U 9030 and ERLYU 9050 are used to prepare an unsigned division operation. In states ERLY 9040 and ERLYU 9050, leading zeros are detected in operand RS to adjust the number of division iterations necessary.

Iterative division is performed in states DIV 9060 and DIVU 9070. Division may be performed by using a series of iterative add/subtracts and shifts. Finally, the remainders are finalized in states REM 9080 and REMU 9090. If either of the operands is negative, sign adjustment is performed in state SGN 9100.

5   Referring again to Fig. 4, in one implementation, permutation logic 4300 is used to support the PPERM instruction described above. Permutation logic 4300 consists of 6 single bit 32:1 selectors that may be used to select any of the 32 bits of RShold 3010 based on the value of RThold 3012. This logic may be implemented directly in the data path module.

10   For example, permutation logic 4300 may be used to execute the instruction "PPERM $5, $6". Permutation logic 4300 uses 6 5-bit selectors determined by RThold 3012 to identify which bits to include as output from RShold 3010. For example, if register $5 contains the low-order bits "010101", then the selector "00010" (corresponding to the low-order bits of register $6) would choose bit 2 (i.e.,

15   the third bit from the right) containing "1". If RThold 3012 contains the low-order bits "0001000011" (corresponding to the 10 low-order bits of register $6), then bit 2 (containing a "1") and bit 3 (containing a "0") will be selected to yield "10". Using this method, permutation logic 4300 may select bits from RShold 3010 to generate 6 bits based on RThold 3012. The resulting 6 bits are concatenated to the 66 low-order

20   bits of ACC1 to form the result. This effectively shifts the 66 low-order bits of ACC1 six bits to the left and replaces the 6 low-order bits with the output of the permutation logic 4300.

The PPERM instruction discussed above is one technique that may be used to provide bitwise permutation support in hardware. In a strict mathematical sense, a

25   bitwise permutation consists of an arbitrary re-ordering of an ordered group of bits within a register or a memory location, as a one-to-one mapping. Permutations as described herein may be more general operations in which one-to-many and one-to-none mappings are also possible. If enough hardware is used, any permutation may be performed in a single clock cycle. However, for anything other than a fixed

30   permutation, a significant amount of state must be established before the permutation may be performed. For example, if bits from a 32-bit value are permuted into an

expanded 48-bit value, each of the 48 destination bits requires 5 bits of data to indicate the corresponding source bit. Thus, 240 bits of state are needed to fully specify the operation.

The amount of state required to specify a permutation may be reduced by reducing the number of destination bits that may be permuted. These partial permutation operations permit extensive permutations to be completed over multiple clock cycles while providing increased performance relative to shift-and-mask algorithms that may be used with unaugmented instruction sets. Partial permutation instructions may be provided that take inputs such as the following: (1) a subset of destination bits to permute into; (2) a description of the source of each bit in the subset of destination bits to permute into; (3) a previous partial value; and (4) a destination register.

Destination bits may be specified in several ways, with varying degrees of economy. For example, destination bits may be specified in a free-form format with each destination bit using at least a 5-bit value to specify its position. Destination bits also may be specified as a contiguous group starting at an explicitly controlled bit, requiring at least 5 bits per instruction. Additionally, destination bits may be specified as a contiguous group starting at an implicitly controlled bit, with a full permutation operation being performed as a canonical instruction sequence.

The PPERM instruction provides a hardware implementation of partial bitwise permutations in a microprocessor multiply or multiply/divide unit. In addition to the PPERM instruction discussed above, several alternative implementations of partial bitwise permutations may be desirable.

Referring to FIG. 10A, a partial permutation operation taking four operands may be specified. The *rd* operand specifies the destination register to store the result. The *rs* operand specifies the input word used as the source bits for performing a partial permutation. The *rt* operand identifies a register storing a previous partial value. Finally, the *ru* operand is used as a control input to specify the partial permutation that is to be performed.

Referring to FIG. 10B, the control register specified by operand *ru* may be

implemented using four 5-bit values to specify the source bits, a mask specifying

which of the four destination bits to permute, a default bit, and a destination nibble.

The destination nibble may be used to specify a particular 4-bit field within the

5   destination register to place the resulting permuted bits. For example, if the

destination nibble is 0, the low-order four bits of the destination register are replaced.

The four source bit identifiers specify which bits of the input word (specified

by *rs*) are used to replace the corresponding bits in the destination register (specified

by *rd*). These source bit identifiers form a 4-bit field to be placed as specified by the

10   destination nibble. For example, if source bit 0 is "00010", then bit 2 of *rs* is the low

order bit of the 4-bit field. If *rs* contains "1010", then a "0" (bit 2 is the third bit from

the right) forms the low-order bit of the destination nibble.

The mask is used to specify whether to permute the corresponding source bit.

For example, a mask of "00111" will only perform the permutations specified by

15   source bits 0, 1, and 2. If the corresponding mask bit is 0, the default bit is used in the

destination nibble. In this example, the mask bit corresponding to source bit 3 is a

"0", therefore the value of the default bit is used for the high-order bit of the

destination nibble. The use of masks and default bits, wherein the default bit may be

zero, one, or an indication that the value of destination bits unselected for permutation

20   by the mask is to remain unchanged, is useful where the desired result value is the

permutation of bits gathered from multiple source words. The value of all nibbles of

the destination register not selected for the operation are copied from the previous

partial value specified by operand *rt*.

Referring to FIG. 11A, in another implementation of partial permutations,

25   fewer registers are used in specifying partial permutation instructions. In this

implementation, a destination operand *rd* and a source operand *rs* are specified.

Instead of using a control register, as discussed above, two source bit specifiers are

included in the instruction. Using this instruction format, instructions specify fewer

bits to permute. However, the instruction repeat rate may be higher.

30   Because there is a mask or starting specifier used in this instruction format, the

source register of the previous partial permutation is implicitly the destination register

18

and the instruction implicitly performs a shift or rotate by two bits on the previous value before merging in the two additional bits from the source. The absence of a mask for bits whose values are not derivable from the current source (*rs*) register may be handled by using explicit shift/rotate instructions, and by using a single-bit partial permutation instruction.

Referring to FIG. 11B, a single-bit partial permutation instruction includes an analogous format to the two-bit format discussed with reference to FIG. 11A. In this implementation, a single bit is specified in the instruction field and an implicit one-bit shift is performed before replacing the low-order bit with the selected source bit.

Referring to FIG. 12A, partial permutations also may be performed using the accumulator of a multiply/divide unit. For example, the instruction format shown in FIG. 12A includes a source register operand *rs*, a control word operand *rt*, and a destination quibble (5-bit field identifier). The HI/LO registers of MDU 2030 may be used implicitly as the source data register and the previous partial value source.

Referring to FIG. 12B, a control word may contain a default bit, a mask, and source bits as described above with reference to FIG. 10B. However, in this implementation, the destination quibble (as opposed to nibble) is specified in the instruction, leaving space for an additional source bit identifier in the control word. Instructions in this format operate on the HI/LO register pair and may be used to make expansion permutations more efficient (e.g., 32-bit values expanded to 48-bit values).

Referring to FIG. 13A, partial permutations also may be performed using the accumulator of a multiplier or multiply/divide unit and no mask. For example, the instruction format shown in FIG. 13A includes a source register operand *rs* and a control word operand *rt*. The HI/LO registers of MDU 2030 may be used implicitly as the previous partial value source. Referring to FIG. 13B, a control word may contain a description of six source bits as described above with reference to FIG. 10B.

Additional partial permutation implementations may increase the number of bits specified by using more than one control word operand. For example, one implementation uses an instruction encoding such as that described in FIG. 10A and a

19

control word encoding such as that described in FIG. 10B, except that two control

words may be specified. This implementation uses operand *rt* to specify a second

control word operand instead of the previous partial value source, which is implicitly

specified as another register, for example, the HI/LO register. Additional

5   implementations allow (i) permutation of 12 bits by modifying the instruction coding

of FIG. 13A to provide two control word operands, such as that described in FIG.

13B; and (ii) permutations of 10 bits using two control word operands as described

with reference to FIG. 12B.

        In addition to multiplier implementations using hardware (e.g., within a

10   microprocessor or microcontroller), implementations also may be embodied in

software disposed, for example, in a computer usable (e.g., readable) medium

configured to store the software (i.e., a computer readable program code). The

program code causes the enablement of the functions or fabrication, or both, of the

systems and techniques disclosed herein. For example, this can be accomplished

15   through the use of general programming languages (e.g., C, C++), hardware

description languages (HDL) including Verilog HDL, VHDL, AHDL (Altera HDL)

and so on, or other available programming and/or circuit (i.e., schematic) capture

tools. The program code can be disposed in any known computer usable medium

including semiconductor, magnetic disk, optical disk (e.g., CD-ROM, DVD-ROM)

20   and as a computer data signal embodied in a computer usable (e.g., readable)

transmission medium (e.g., carrier wave or any other medium including digital,

optical, or analog-based medium). As such, the code can be transmitted over

communication networks including the Internet and intranets.

        It is understood that the functions accomplished and/or structure provided by

25   the systems and techniques described above can be represented in a core (e.g., a

microprocessor core) that is embodied in program code and may be transformed to

hardware as part of the production of integrated circuits. Also, the systems and

techniques may be embodied as a combination of hardware and software.

Accordingly, other implementations are within the scope of the following claims.

30

WHAT IS CLAIMED IS:

1.   In an instruction set architecture, an instruction for performing partial bitwise permutations, the instruction being part of the instruction set architecture and including:

5   an opcode identifying the instruction as a partial permutation instruction; and

a permutation operation specification including:

a destination specifier identifying a destination register;

a partial value source specifier;

10   a destination subset specifier identifying one or more destination bits of the destination register; and

a control specifier identifying a source for each of the one or more destination bits identified by the destination subset specifier;

wherein the instruction is processed by performing a partial bitwise

15   permutation defined by the permutation operation specification.

2.   The instruction of claim 1 wherein the destination specifier implicitly identifies the destination register.

3.   The instruction of claim 2 wherein the destination register comprises a multiply unit accumulator.

20   4.   The instruction of claim 1 wherein the destination specifier explicitly specifies a general-purpose register.

5.   The instruction of claim 1 wherein the partial value source specifier implicitly identifies a partial value source register.

6.   The instruction of claim 5 wherein the partial value source register

25   comprises a multiply unit accumulator.

7.   The instruction of claim 1 wherein the partial value source specifier explicitly specifies a general-purpose register.

8.   The instruction of claim 1 wherein the destination specifier and the

partial value source specifier each identify the same register.

9.   The instruction of claim 8 wherein the identified register comprises a general-purpose register.

10.   The instruction of claim 8 wherein the identified register comprises a multiply unit accumulator.

11.   The instruction of claim 1 wherein the destination subset specifier identifies a contiguous block of bits within the destination register.

12.   The instruction of claim 11 wherein the contiguous block of bits includes the least significant bit of the destination register.

13.   The instruction of claim 12 wherein the contiguous block of bits includes 12 or fewer bits.

14.   The instruction of claim 1 wherein the control subset specifier includes one or more source bit identifiers.

15.   The instruction of claim 14 where the control subset specifier further includes:

            a mask selecting bits to receive permutation data; and

            a default bit identifying a default value to be assigned to bits not selected by the mask to receive permutation data.

16.   The instruction of claim 14 wherein each of the one or more source bit identifiers is a field within the instruction.

17.   The instruction of claim 14 wherein each of the one or more source bit identifiers are stored in a control register, the control register identified by the control specifier.

18.   The instruction of claim 1 wherein the instruction set comprises a RISC instruction set.

19.   A method for performing partial bitwise permutations using an instruction, the instruction including:

            fetching an instruction to perform an operation from a data store;

reading one or more registers;

performing the operation specified by the instruction, the instruction including:

an opcode identifying the instruction as a partial permutation

5   instruction; and

a permutation operation specification including:

a destination specifier identifying a destination register;

a previous partial value source specifier;

a destination subset specifier identifying one or more

10   destination bits of the destination register; and

a control specifier identifying a source for each of the one or more destination bits identified by the destination subset specifier;

wherein the instruction is processed by performing a partial bitwise permutation defined by the permutation operation specification.

15   20.   The method of claim 19 wherein the destination specifier implicitly identifies the destination register.

21.   The method of claim 20 wherein the destination register comprises a multiply unit accumulator.

22.   The method of claim 19 wherein the destination specifier explicitly

20   specifies a general-purpose register.

23.   The method of claim 19 wherein the previous partial value source specifier implicitly identifies a partial value source register.

24.   The method of claim 23 wherein the partial value source register comprises a multiply unit accumulator.

25   25.   The method of claim 19 wherein the previous partial value source specifier explicitly specifies a general-purpose register.

26.   The method of claim 19 wherein the destination specifier and the previous partial value source specifier identify each identify the same register.

27. The method of claim 26 wherein the identified register comprises a general-purpose register.

28. The method of claim 26 wherein the identified register comprises a multiply unit accumulator.

29. The method of claim 19 wherein the destination subset specifier identifies a contiguous block of bits within the destination register.

30. The method of claim 29 wherein the contiguous block of bits includes the least significant bit of the destination register.

31. The method of claim 30 wherein the contiguous block of bits includes 12 or fewer bits.

32. The method of claim 19 wherein the control subset specifier includes one or more source bit identifiers.

33. The method of claim 32 where the control subset specifier further includes:

a mask selecting bits to receive permutation data; and

a default bit identifying a default value to be assigned to bits not selected by the mask to receive permutation data.

34. The method of claim 32 wherein each of the one or more source bit identifiers is a field within the instruction.

35. The method of claim 32 wherein each of the one or more source bit identifiers are stored in a control register, the control register identified by the control specifier.

36. The method of claim 19 wherein the instruction is part of an instruction set, and the instruction set comprises a RISC instruction set.

37. A computer-readable medium comprising a microprocessor core embodied in software, the microprocessor core including an instruction for performing partial bitwise permutations, the instruction including:

an opcode identifying the instruction as a partial permutation

24

instruction; and

a permutation operation specification including:

a destination specifier identifying a destination register;

a previous partial value source specifier;

a destination subset specifier identifying one or more destination bits of the destination register; and

a control specifier identifying a source for each of the one or more destination bits identified by the destination subset specifier;

wherein the instruction is processed by performing a partial bitwise permutation defined by the permutation operation specification.

38.   The computer-readable medium of claim 37 wherein the destination specifier implicitly identifies the destination register.

39.   The computer-readable medium of claim 38 wherein the destination register comprises a multiply unit accumulator.

40.   The computer-readable medium of claim 37 wherein the destination specifier explicitly specifies a general-purpose register.

41.   The computer-readable medium of claim 37 wherein the previous partial value source specifier implicitly identifies a partial value source register.

42.   The computer-readable medium of claim 41 wherein the partial value source register is a multiply unit accumulator.

43.   The computer-readable medium of claim 37 wherein the previous partial value source specifier explicitly specifies a general-purpose register.

44.   The computer-readable medium of claim 37 wherein the destination specifier and the previous partial value source specifier identify each identify the same register.

45.   The computer-readable medium of claim 44 wherein the identified register comprises a general-purpose register.

46.   The computer-readable medium of claim 44 wherein the identified

register comprises a multiply unit accumulator.

47.   The computer-readable medium of claim 37 wherein the destination subset specifier identifies a contiguous block of bits within the destination register.

48.   The computer-readable medium of claim 47 wherein the contiguous block of bits includes the least significant bit of the destination register.

49.   The computer-readable medium of claim 48 wherein the contiguous block of bits includes 12 or fewer bits.

50.   The computer-readable medium of claim 37 wherein the control subset specifier includes one or more source bit identifiers.

51.   The computer-readable medium of claim 50 where the control subset specifier further includes:

a mask selecting bits to receive permutation data; and

a default bit identifying a default value to be assigned to bits not selected by the mask to receive permutation data.

52.   The computer-readable medium of claim 50 wherein each of the one or more source bit identifiers is a field within the instruction.

53.   The computer-readable medium of claim 50 wherein each of the one or more source bit identifiers are stored in a control register, the control register identified by the control specifier.

54.   The computer-readable medium of claim 37 wherein the instruction is part of an instruction set, and the instruction set comprises a RISC instruction set.

55.   In a microprocessor containing a first general purpose register, a second general purpose register and an extended-precision accumulator, a method for performing a partial permutation comprising:

shifting contents of the extended-precision accumulator to produce a predetermined number of open bit positions;

selecting bits for filling the open bit positions with information contained in the first general purpose register; and

26

filling the open bit positions with bits retrieved from the second general

purpose register, wherein the shifting, selecting and filling occur in response to a

single instruction.

56.   The method of claim 55 wherein the single instruction specifies the

first and second general purpose registers.

57.   The method of claim 55 wherein the predetermined number of open bit

positions are six least significant bits of the extended-precision accumulator.

58.   In a microprocessor, an instruction for performing partial bitwise

permutations, the instruction being part of the instruction set architecture and

including:

an opcode identifying the instruction as a partial permutation

instruction; and

a permutation operation specification including:

a destination specifier identifying a destination register;

a previous partial value source specifier;

a destination subset specifier identifying one or more destination bits

of the destination register; and

a control specifier identifying a source for each of the one or more

destination bits identified by the destination subset specifier;

wherein the instruction is processed by performing a partial bitwise

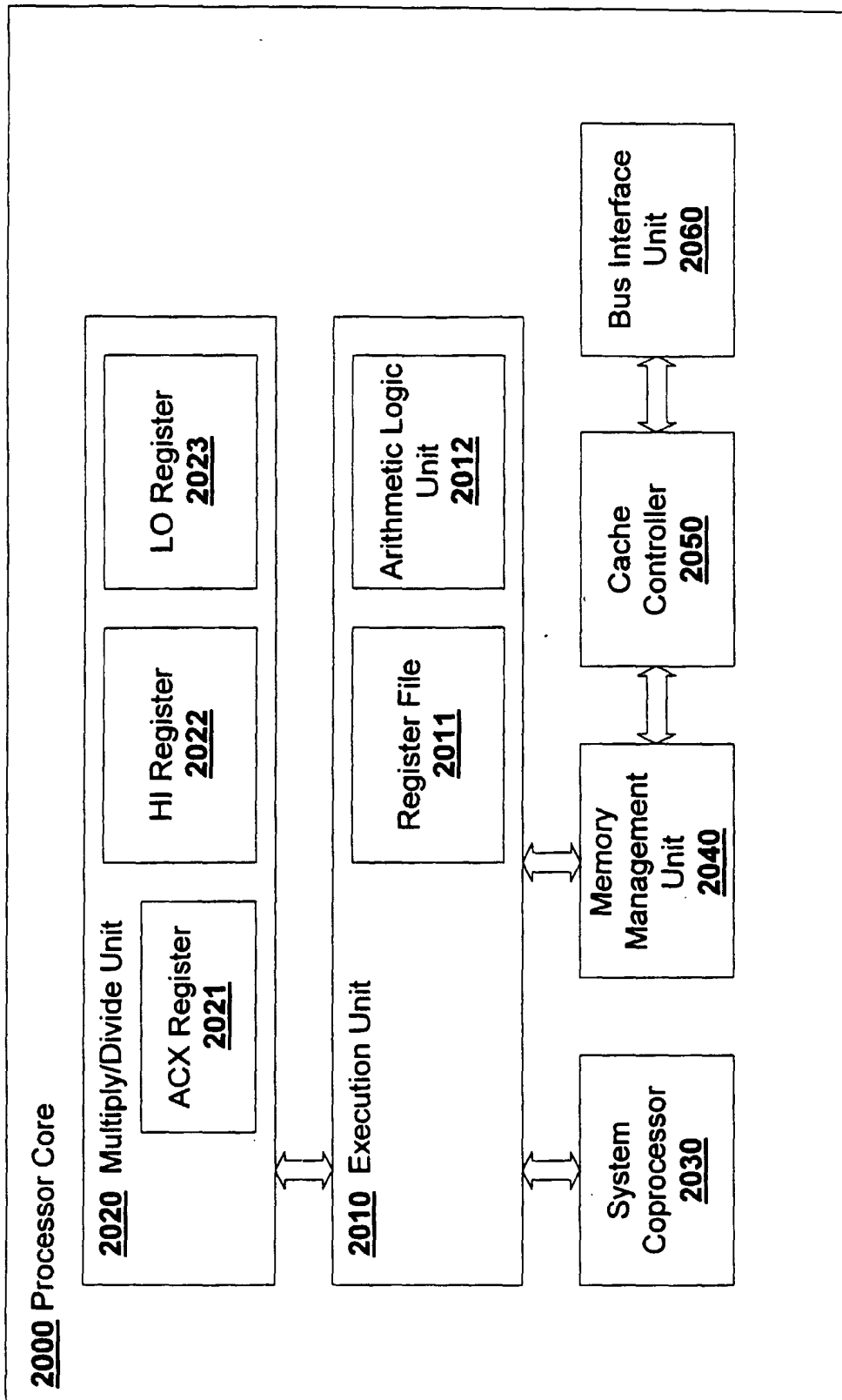permutation defined by the permutation operation specification.
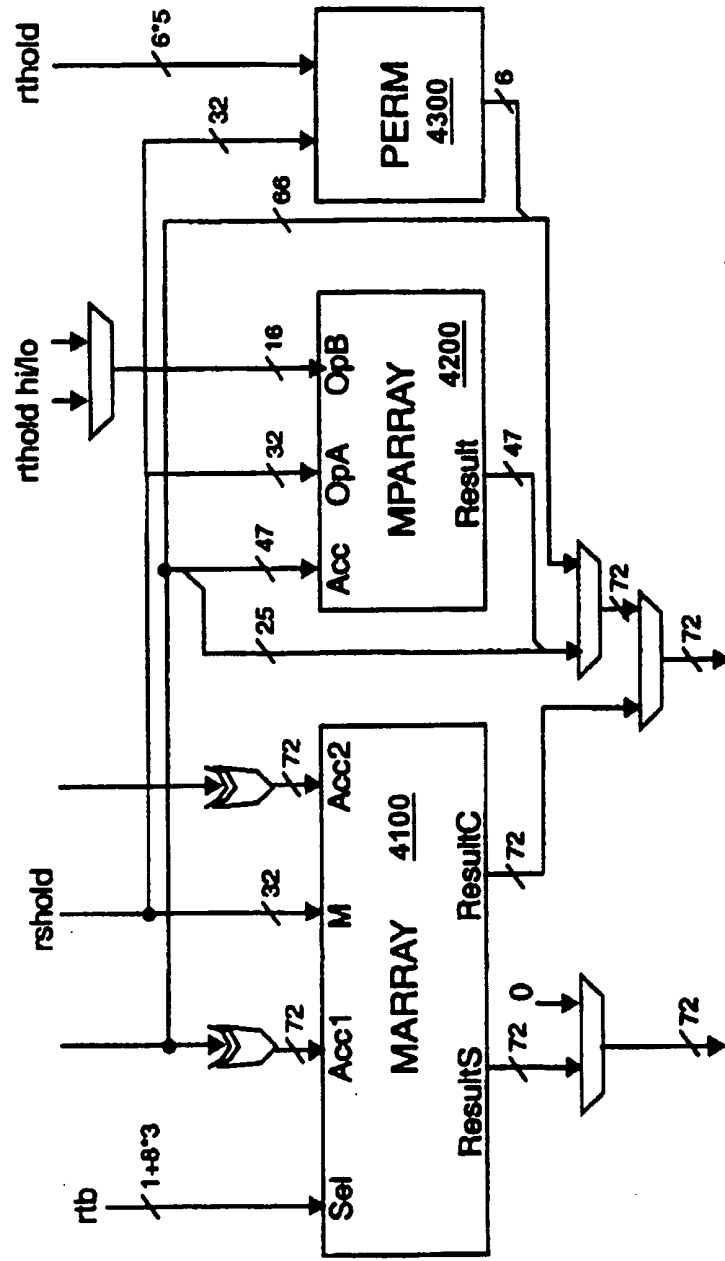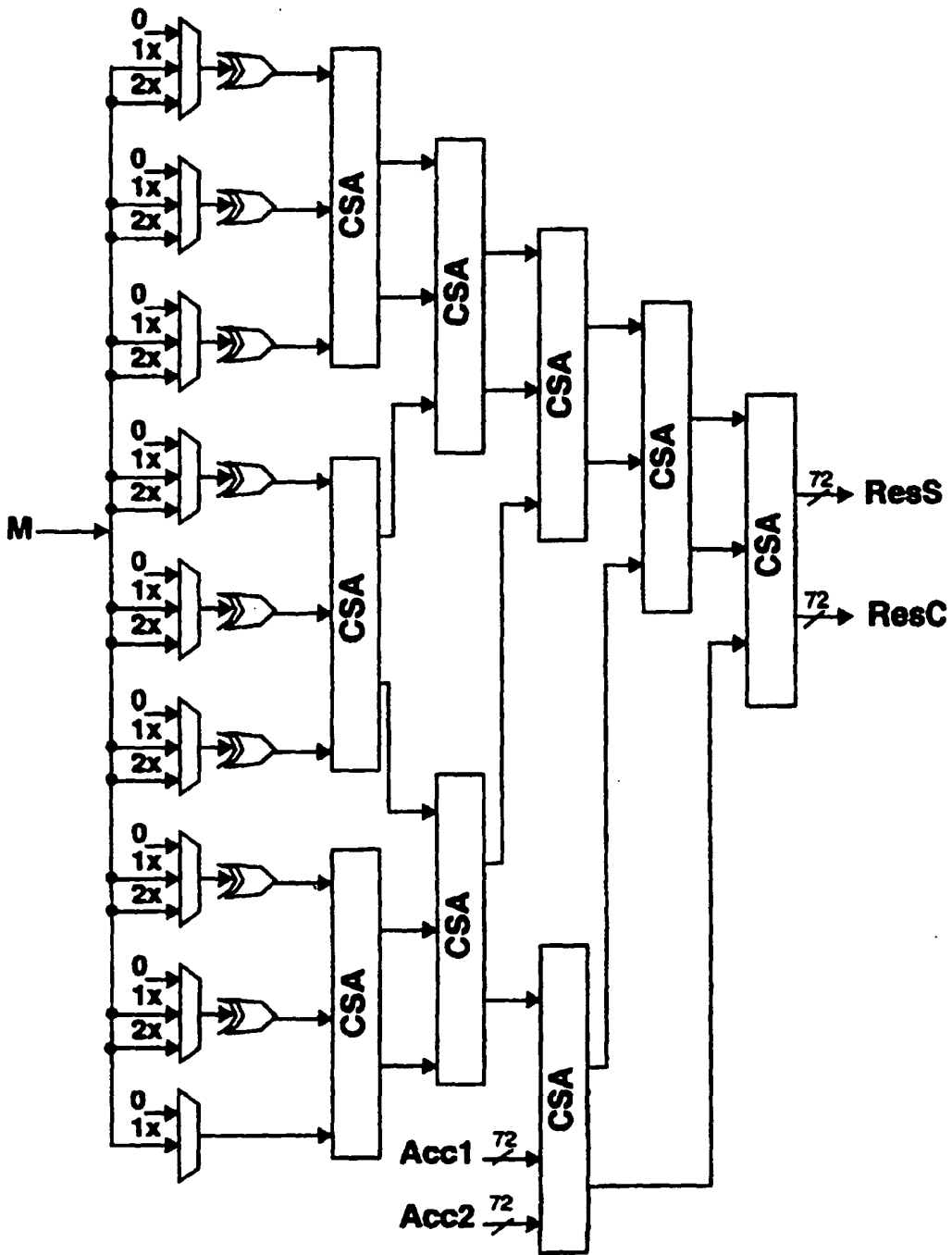
FIG. 1

FIG. 2

FIG. 3

FIG. 4

FIG. 5
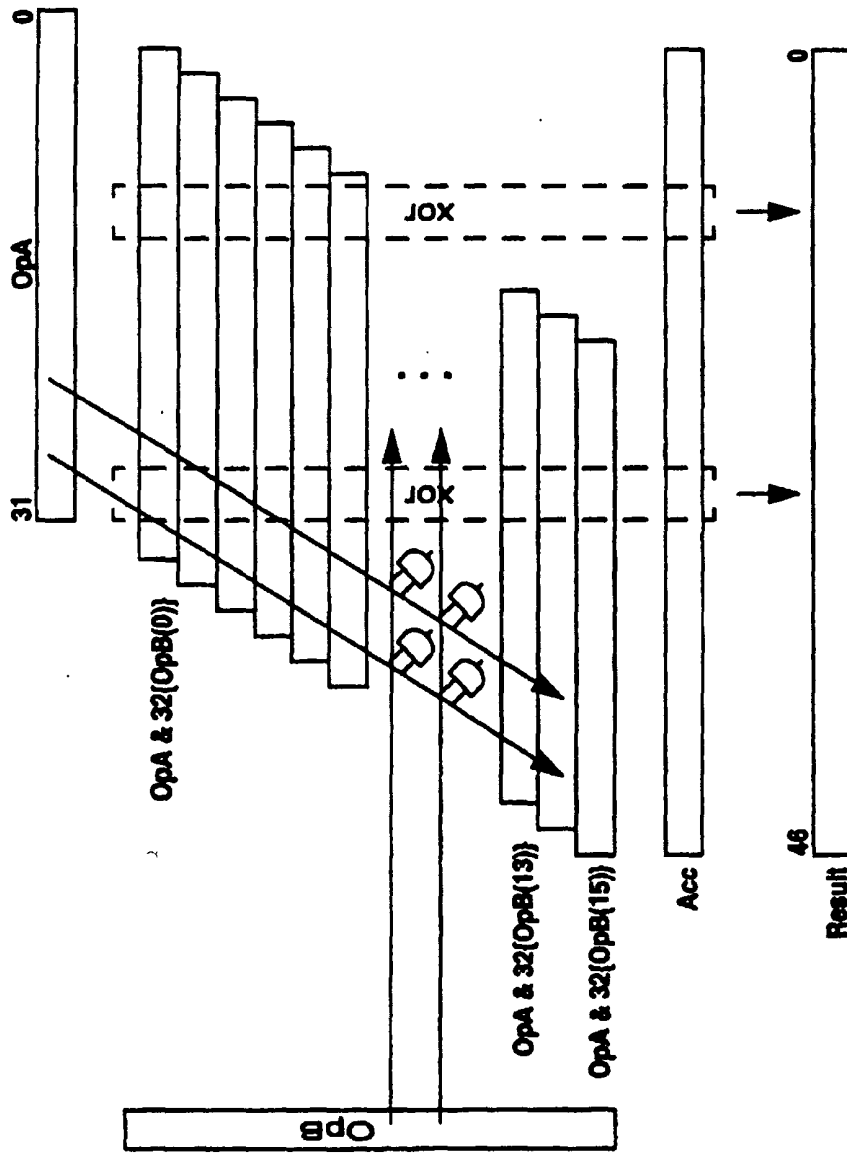
FIG. 6
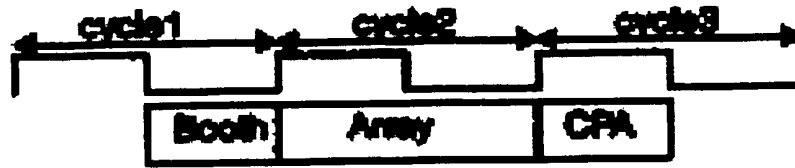
FIG. 7A

FIG. 7B

FIG. 7C

FIG. 8

FIG. 9

| SPEC2 | rd | rs | rt | ru | PPERM |

FIG. 10A

3 bits	3 bits	1 bit	5 bits	5 bits	5 bits	5 bits	5 bits
unused	dest nibble	default bit	mask	src bit 3	src bit 2	src bit 1	src bit 0

FIG. 10B

SPBC2	rd	rs	src bit 1	src bit 0	PPERM2

FIG. 11A

SPBC2	rd	rs	null	src bit 0	PPERM1

FIG. 11B

| SPBC2 | 0 | rs | rt | 0 | dest. "quibble" | PPERM |

FIG. 12A

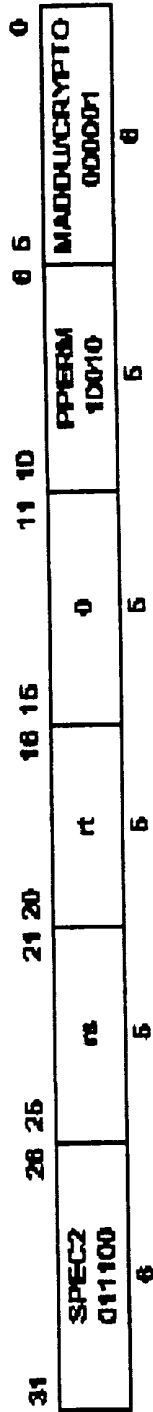| 1 bit | 1 bit | 5 bits | 5 bits | 5 bits | 5 bits | 5 bits | 5 bits |
| unused | default bit | mask | src bit 4 | src bit 3 | src bit 2 | src bit 1 | src bit 0 |
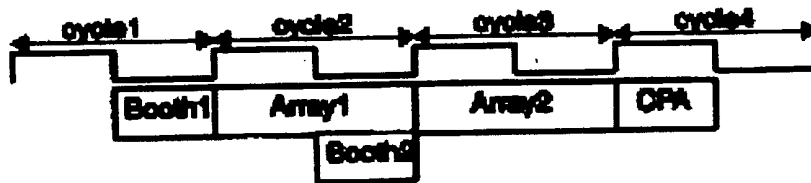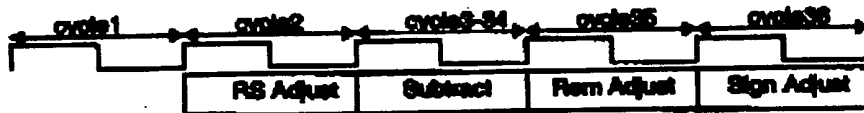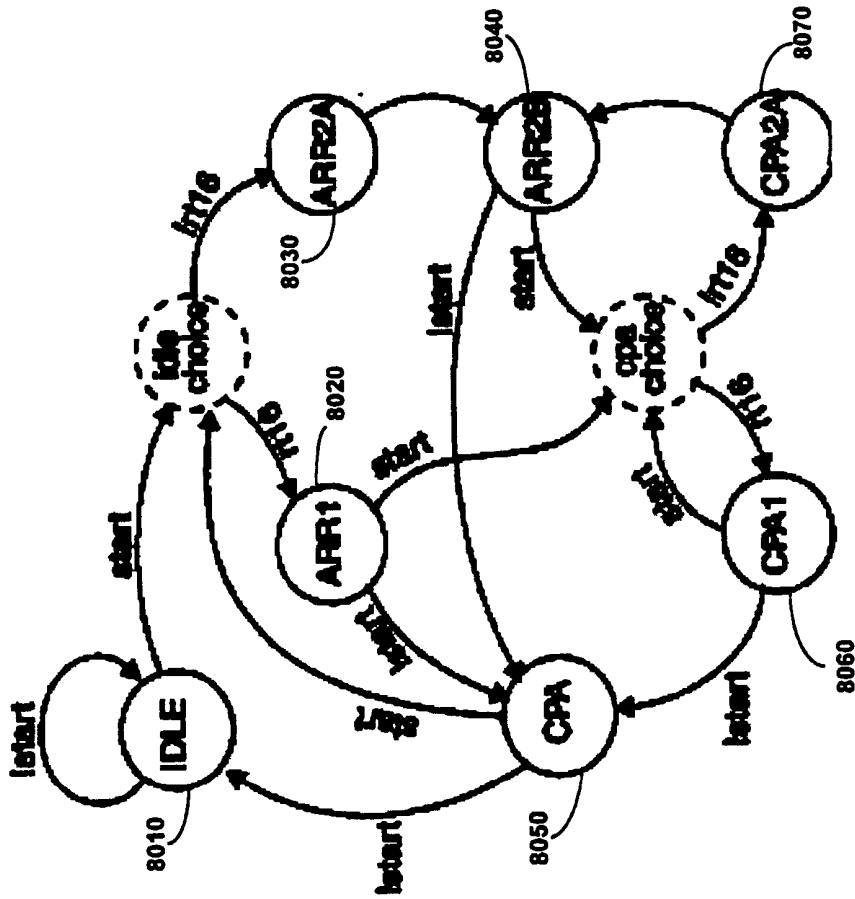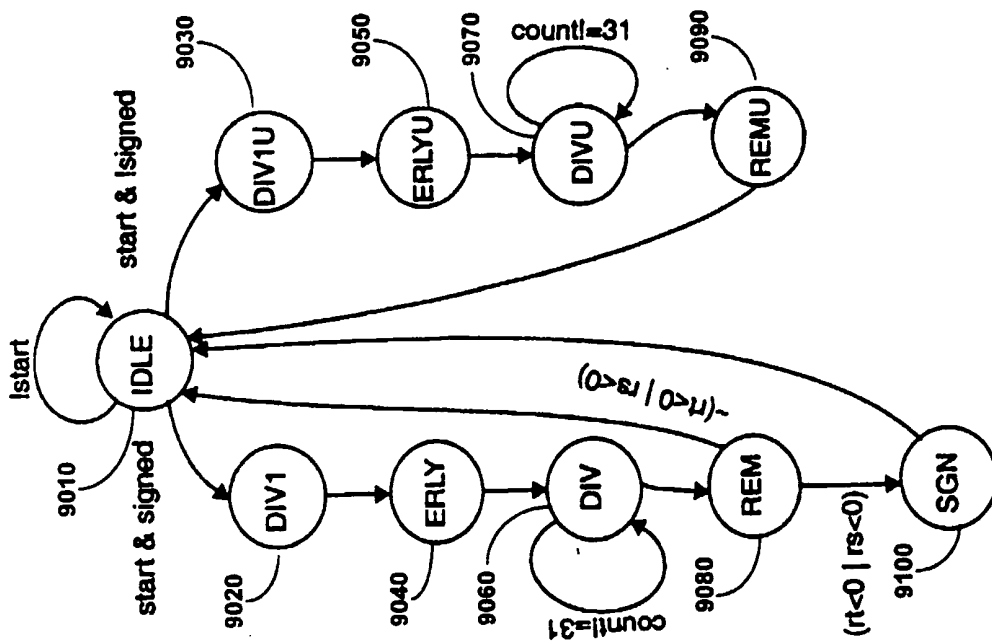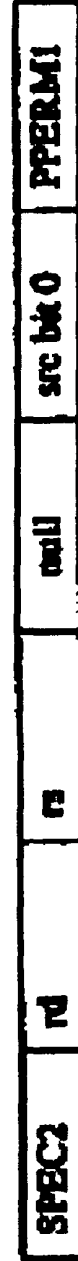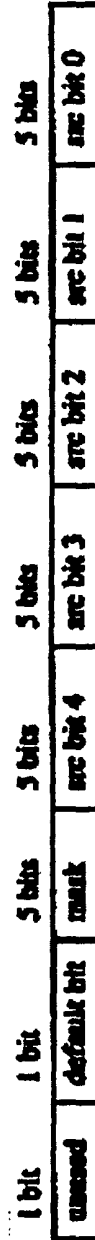
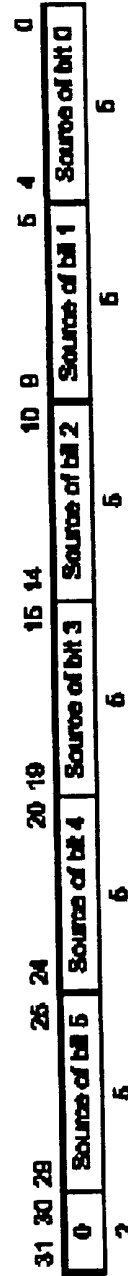**FIG. 12B**

FIG. 13A

FIG. 13B

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US02/04427

A. CLASSIFICATION OF SUBJECT MATTER
IPC(7) : G06F 7/38; 9/30
US CL : 712/41, 223, 300
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
  U.S. : 712/41, 223, 300; 707/101; 708/168,230,400,490

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
Please See Continuation Sheet

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US, 5,838,986 A(GARG et al)17 November 1998 (17.11.1998), figures 1,4,7, column 4, line 33-column 5, line 18.	1-58
Y	US 5,781,457 A(COHEN et al) 14 July 1998 (14.07.1998), figures 2,3, 4,5, 6, 7, 8, 9, column 2, line 44-column 3, line 52.	1-58
Y	US 5,673,407 A(POLAND et al) 30 September 1997 (30.09.1997),figs. 11,13,17,18,19,20,21,22,23, column 25, line 1-column 33, line 38.	1,19,37,55,58
Y	US 4,949,250 A(BHANDARKAR et al) 14 August 1990(14.08.1990), figs. 3,4a-4d,5a,5b,8a-8f, column 7, line 1-column 8, line 60.	1,19,37,55,58
A	US 5,850,452 A (SOURGEN et al) 15 December 1998 (15.12.1998), column 3, lines 3-column 4, line 62.	1,19,37,55,58
A	US 5,768,172 A (DERBY) 16 June 1998 (16.06.1998) column 2, lines 7-26.	1,19,37,55,58
A	US 5,819,117 A (HANSEN) 06 October 1998 (06.10.1998), column 3, lines 16-66.	1,19,37,55,58
A,P	US 6,199,087 A (BLAKE et al)06 March 2001(06.03.2001), column 2, line 50-column 3, line 65.	1,19,37,55,58

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance		
"E"	earlier application or patent published on or after the international filing date	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed	"&"	document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
29 April 2002 (29.04.2002)	2 8 MAY 2002

Name and mailing address of the ISA/US
  Commissioner of Patents and Trademarks
  Box PCT
  Washington, D.C. 20231
Facsimile No. (703)305-3230
	Authorized officer ~
	
	Eric Coleman
	
	Telephone No. (703)-305-3900

Form PCT/ISA/210 (second sheet) (July 1998)

Continuation of B. FIELDS SEARCHED Item 3:
EAST
search terms: permutations, instruction, operation, byte,upper,higher, lower , half, boolean, risc,operations,code