Single-Cycle Bit Permutations with MOMR Execution

Ruby B. Lee[1], Xiao Yang[1], and Zhijie Jerry Shi[2]

[1]*Department of Electrical Engineering, Princeton University, Princeton, NJ 08544, USA*

[2]*Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA*

E-mail: rblee@princeton.edu; xiaoyang@princeton.edu; zshi@uconn.edu

Received , 200; revised , 200.

Abstract Secure computing paradigms impose new architectural challenges for general-purpose processors. Cryptographic processing is needed for secure communications, storage, and computations. We identify two categories of operations in symmetric-key and public-key cryptographic algorithms that are not common in previous general-purpose workloads: advanced bit operations within a word and multi-word operations. We define *MOMR* (*Multiple Operands Multiple Results*) *execution* or *datarich execution* as a unified solution to both challenges. It allows arbitrary n-bit permutations to be achieved in one or two cycles, rather than $O(n)$ cycles as in existing RISC processors. It also enables significant acceleration of multi-word multiplications needed by public-key ciphers. We propose two implementations of MOMR: one employs only hardware changes while the other uses Instruction Set Architecture (ISA) support. We show that MOMR execution leverages available resources in typical multi-issue processors with minimal additional cost. Multi-issue processors enhanced with MOMR units provide additional speedup over standard multi-issue processors with the same datapath. MOMR is a general architectural solution for word-oriented processor architectures to incorporate datarich operations.

Keywords permutation, bit permutations, cryptography, cryptographic acceleration, security, multi-word operation, datarich execution, MOMR, instruction set architecture, ISA, processor, high performance secure computing

1 Introduction

Secure communications, computations, and storage utilize cryptographic processing in secure protocols to provide data confidentiality, data integrity, and user authentication. As secure computing paradigms increase in importance, cryptographic processing will become a major part of every processor's workload. Hence, understanding the new requirements of fast cryptographic processing by software is critical in the design of all future processors, whether general-purpose, application-specific, or embedded. In this paper, we target especially the needs of high performance microprocessors.

Confidentiality can be achieved by encrypting the data or message, using symmetric-key cryptographic algorithms such as DES[1], and AES [2]. Data integrity, which assures data or messages are not changed in transit or in storage, can be accomplished with secure hash functions such as SHA and MD-5[1]. Authenticating users and devices remotely across the Internet, enabling digital signatures and key exchanges can be accomplished with public-key cryptographic algorithms such as Diffie-Hellman and RSA[1].

We observe two categories of new requirements imposed by these classes of cryptographic algorithms: bit-oriented operations and multi-word operations. Both challenge the basic word-orientation of modern processors.

A highly desirable operation in the construction of symmetric-key block ciphers is *bit permutation*, which allows arbitrary rearrangement of the bits in the block being encrypted. This is currently very slow in microprocessors where only simple logical operations are supported for bit-wise operations.

Public-key cryptography introduces the other new requirement: multi-word arithmetic. While multi-word integer arithmetic has been a requirement in previous high-precision integer computations, its need remained relatively low since the basic word size in microprocessors has increased from 16 to 32, then to 64 bits. Frequent use of public-key cryptography algorithms may significantly increase the need for multi-word arithmetic, such as the multiplication, in RSA, of two 1024-bit operands (or 16-word operands, each 64 bits). Public-key algorithms based on Elliptic Curve Cryptography (ECC) often perform polynomial operations requiring both bit-oriented and multi-word operations.

A key contribution of this paper is the observation that fast cryptographic processing depends on a processor's ability to support both complex bit-level manipulations as well as multiword operations.

A second contribution is a generalized architectural solution that allows high-performance processors to support *datarich* operations with MOMR (Multiple Operands Multiple Results) execution. MOMR is also a unified solution that achieves both high performance bit permutations and multi-word operations needed for cryptographic acceleration of symmetric-key and public-key ciphers.

A third contribution is to show how arbitrary n-bit permutations can be accomplished very efficiently in only one or two cycles with MOMR execution. This previously took $O(n)$ instructions in basic RISC processors. With recently proposed bit permutation

This work was supported in part by the National Science Foundation, U.S.A., under grants CCR-0326372 and CCR-0105677.

instructions[3−7], $O(\log(n))$ ① instructions was the optimal result.

Section 2 defines datarich or MOMR execution and introduces two methods for achieving MOMR execution in microprocessors; one purely micro-architectural, and the other involving new ISA. Section 3 describes how the two MOMR methods can be applied to achieve an arbitrary n-bit permutation every cycle. Section 4 describes MOMR implementation in detail. Section 5 shows the generality of MOMR and Section 6 discusses MOMR performance.

2 Datarich MOMR Execution

We introduce the following new definitions:

An (s, t) functional unit in a word-oriented processor is a functional unit that takes s word-sized operands and produces t word-sized results. A standard functional unit, e.g., an adder, is a $(2, 1)$ functional unit.

An (s, t) *datapath* in a word-oriented processor is a datapath where s source buses and t destination buses are connected to functional units. If the datapath contains a register file, it has s read ports and at least t write ports for the results coming from the functional units and from memory. A k-way multi-issue processor has a $(2k, k)$ datapath, supporting the simultaneous execution of k standard $(2, 1)$ functional units each cycle.

A *MOMR* (*Multiple Operands Multiple Results*) or *datarich functional unit* in a word-oriented processor is a functional unit that requires more than two source operands or generates more than one result, or both. It is an (m, n) functional unit, where either $m > 2$ or $n > 1$. It requires the support of an (m, n) datapath.

An *instruction group* is a sequence of instructions that can be executed simultaneously by a MOMR functional unit.

We describe two methods for detecting instruction groups: Method 1 detects instruction groups dynamically with hardware, while method 2 employs ISA techniques to identify instruction groups statically.

2.1 MOMR Method 1: Hardware Detection of Instruction Group for MOMR Execution

An instruction group can be detected as:

(a) a sequence of dependent instructions with the same or related opcode and at least three different source registers between them, but only one destination register (MO case).

(b) a sequence of instructions with related opcode where the operands in each instruction are the same but the destination registers are different (MR case).

By a related opcode, we mean either that the opcode is the same but the subop is different, or that the opcodes are different but they use the same or similar functional units. These two conditions can be generalized to a small set of destination registers in (a) and only some of the operands being the same in (b).

For example, in Fig.1(a), condition (a) for an instruction group is detected. (The first two registers specify the operand registers while the last register specifies the result register.) This can be implemented as a MOMR functional unit with four operands (Rs1, Rs2, Rs3, and Rs4) and one result Rd. In Fig.1(b), condition (b) for an instruction group is detected. This can potentially be implemented as a MOMR functional unit with two operands (Rs1 and Rs2) and two results (Rd1 and Rd2). In each case, if a MOMR functional unit is implemented in the processor, the whole instruction group can be issued together to it.

```
OPA   Rs1, RS2, Rd        OPB.L   Rs1, Rs2, Rd1
OPA    Rd, Rs3, Rd        OPB.R   Rs1, Rs2, Rd2
OPA    Rd, Rs4, Rd
          (a)                         (b)
```

Fig.1. Examples of instruction groups for MOMR. (a) Multiple operands. (b) Multiple results.

2.2 MOMR Method 2: New ISA for Instruction Group Identification

This method uses two new subop bits, *gs* and *gc*, for identifying instructions which start a group ($gs = 1$) or continue a group ($gc = 1$). These two bits are only needed in instructions that can be part of an instruction group for possible MOMR execution. They are defined in Table 1.

Table 1. *gs* and *gc* Bits

$gs = 0, gc = 0$	Normal instruction, not part of a group
$gs = 1, gc = 0$	First instruction in a group
$gs = 0, gc = 1$	Continuation instruction in a group
$gs = 1, gc = 1$	Reserved

The performance of cryptographic processing and other applications can be significantly improved by the use of MOMR functional units.

3 Achieving Single-Cycle Bit Permutations

MOMR execution enables us to achieve the "gold standard" of adding a new primitive operation to a general-purpose RISC microprocessor (in this case, for fast arbitrary n-bit permutations). It enables an n-bit processor to perform a *different* n-bit permutation *each cycle* with minimal changes to the ISA, microarchitecture, and datapath. In this section, we first describe past work on permutation instructions, then show how a different n-bit permutation can be achieved every cycle using one of our MOMR methods.

①We will use $\log(n)$ to denote $\log_2(n)$ throughout this paper.

3.1 Past Work

Performing arbitrary bit-level permutations has been very slow with word-oriented general-purpose processors. Previously, processor hardware only supported a very restricted subset of bit permutations known as rotations. Here, every bit in the n-bit word is moved by the same shift amount, with wrap-around. There are only n different rotations of an n-bit word, but $n!$ different permutations. While some n-bit permutations can be achieved with fewer instructions, allowing arbitrary, data-dependent, n-bit permutations takes $O(n)$ cycles using shift and logical instructions[3]. Alternatively, table lookup methods can be used, but the memory space required limits this to a few fixed permutations, and cache misses degrade the performance.

From the mid 1980s, subword permutation instructions have been introduced into microprocessors as multimedia ISA extensions to handle the rearrangement of subwords packed in registers. Examples are MIX and PERMUTE in HP's MAX-2[8], VPERM in Motorola's AltiVec[9], and MIX and MUX in IA-64[10]. However, these instructions only handle subword sizes down to eight bits. They do not provide a general solution for performing arbitrary bit-level permutations efficiently.

More recently, our PALMS[11] research group has tackled the general bit permutation problem, defining alternative new permutation instructions for achieving any n-bit permutation with only $\log(n)$ instructions. The CROSS[4] and OMFLIP[5] permutation instructions each performs the equivalent function of two stages in a multi-stage interconnection network. A sequence of $\log(n)$ CROSS or OMFLIP instructions can build a "virtual" $2\log(n)$-stage network, which can achieve any one of the $n!$ permutations. The GRP instruction[6] partitions the data bits into left and right groups, preserving the order of the bits in each group. A third approach specifies the order of the indices of the source bits in the permuted result. Examples are PPERM[3,6], and SWPERM with SIEVE[7,6]. The XBOX instruction[12,13] is similar to PPERM.

A comparison of CROSS, OMFLIP, GRP and PPERM is presented in [3]. CROSS, OMFLIP, and GRP all achieve arbitrary n-bit permutations in $\log(n)$ instructions. PPERM and SWPERM with SIEVE require more than $\log(n)$ instructions. However, for 64-bit permutations, SWPERM with SIEVE can be executed in as few as 4 cycles (less than $\log(64) = 6$ cycles) on a 4-way superscalar 64-bit processor. Unfortunately, CROSS, OMFLIP, and GRP cannot achieve further speedup with superscalar machines, due to the strict data dependency in the sequence of $\log(n)$ permutation instructions used. Below, we show how this data dependency can be overcome with MOMR execution so that arbitrary 64-bit permutations can be achieved in one or two cycles, rather than $\log(n)$ cycles. This paper extends[14,15] with new work on the definition of MOMR architecture, its implementation and performance.

3.2 Using MOMR Method 1 (HW Detection)

We define a permutation instruction as:

`PERM rs, rc, rd`

where `rs` contains the data source, `rc` contains configuration bits and `rd` is the result.

Fig.2a shows a 64-bit permutation specified with a sequence of six dependent PERM instructions.

$$\left.\begin{array}{ll} PERM & rs, rc1, rd \\ PERM & rd, rc2, rd \\ PERM & rd, rc3, rd \end{array}\right\} \qquad \begin{array}{l} \text{PERM, gs rs, rc1, rd} \\ \text{PERM, gc rc2, rc3, rd} \end{array}$$

$$\left.\begin{array}{ll} PERM & rd, rc4, rd \\ PERM & rd, rc5, rd \\ PERM & rd, rc6, rd \end{array}\right\} \qquad \begin{array}{l} \text{PERM, gs rd, rc4, rd} \\ \text{PERM, gc rc5, rc6, rd} \end{array}$$

(a) (b)

Fig.2. Instruction groups for MOMR methods 1 and 2. (a) Method 1. (b) Method 2.

Condition (a) for an instruction group is satisfied for these six instructions with a total of seven source operands (`rs, rc1, rc2, rc3, rc4, rc5, and rc6`) and one result (`rd`). This would require a functional unit with seven 64-bit operands which may be too large and slow. However, a smaller and faster $(4,1)$ functional unit can be used instead. The six permutation instructions can be divided into two instruction groups, each providing the data word and three configuration words to a $(4,1)$ permutation unit (PU). The two instruction groups are executed over two cycles, if there is only one $(4,1)$ PU active each cycle. If two $(4,1)$ PUs can execute each cycle, we can pipeline the executions of the two groups and achieve a throughput of one permutation per cycle.

3.3 Using MOMR Method 2 (ISA Support)

Here, we define the permutation instruction as:

`PERM, subop rs1, rs2, rd`

where subop contains *gs* or *gc* bits.

Assuming the same $(4,1)$ PU as above, we can now specify an instruction group as only two instructions (Fig.2(b)). If *gs* is set, the instruction is the first in an instruction group, supplying the data word and one configuration word to the $(4,1)$ PU. If *gc* is set, it is the second instruction in a group, supplying two configuration words for the $(4,1)$ PU. This method can reduce static code size, since fewer instructions are required.

3.4 Why MOMR Accelerates Bit Permutations

Achieving an arbitrary n-bit permutation requires $n\log(n)$ configuration bits to specify the desired permutation out of $n!$ possible permutations[3,6]. If standard $(2,1)$-datapaths are used and no intermediate states are stored in functional units, then $\log(n)$ instructions are needed just to supply the $n\log(n)$ configuration bits, as in Fig.2(a). A permutation instruction uses one source operand for the data and the other for n bits of the configuration. The intermediate result produced by one permutation instruction is used as the data for the next. Hence, a sequence of $\log(n)$ instructions are needed to provide $n\log(n)$ configuration bits to achieve an n-bit permutation[3]. In total, $\log(n)+1$ operands are specified with this sequence of instructions.

Some permutation circuits can use more than one configuration word to permute the data bits without increasing the latency of the operation. Fig.3 shows 8-bit versions of the butterfly and inverse butterfly network circuits used to implement the CROSS instruction[4]. The latency through an n-bit butterfly circuit is less than that through an n-bit ALU since both require $\log(n)$ stages, but each stage of a butterfly circuit is simpler, being only the time taken by a 2:1 multiplexer and wire propagation.

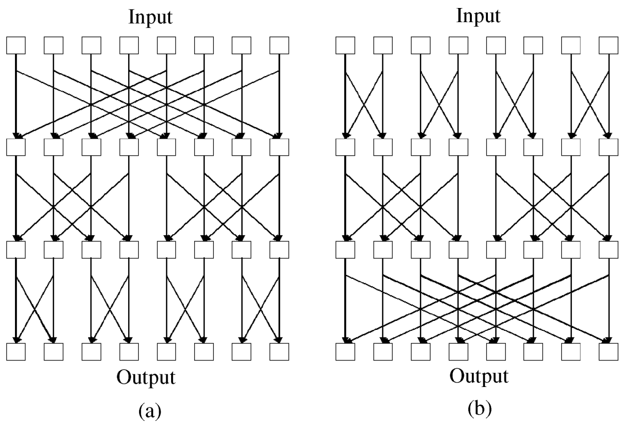

Fig.3. 8-bit butterfly and inverse butterfly networks. (a) Butterfly network. (b) Inverse butterfly network.

However, the CROSS instruction only uses 2 stages of either the butterfly or inverse butterfly network. Since each stage requires $n/2$ configuration bits, only 2 stages can be controlled with the second n-bit operand of the CROSS permutation instruction. Hence, it is the instruction format and datapaths in the ISA and microarchitecture that constrain the maximum performance of arbitrary bit permutations not the latency of the permutation circuitry. MOMR execution removes these constraints.

While a MOMR unit could be designed with $(\log(n)+1)$ operands, the latency of this would be greater than that of an ALU, which we use to define the cycle time. Hence, we choose to use two $(4,1)$ MOMR permutation units, each with single cycle latency comparable to an ALU, to implement the 64-bit versions of the butterfly and the inverse butterfly networks. Since most microprocessors do not have the luxury to change the fundamental ISA and datapath definitions, our MOMR methods allow them to accommodate functional units with more operands and results with minimal changes[15].

4 MOMR Implementation

Our two MOMR methods can leverage the resources already present in a standard multi-issue microprocessor, with minimal additional cost. We first describe a typical superscalar processor, then detail changes that must be made to its datapath and control path. We use the bit permutation example to illustrate the MOMR methods. However the techniques described are applicable to other datarich operations as well.

4.1 Baseline Microarchitecture

Fig.4(a) shows a standard 2-way superscalar RISC processor with a $(4,2)$ datapath, i.e., 4 register read ports, three write ports (one for memory), and associated data buses and bypass paths. Since MOMR implementation in an in-order processor is quite straightforward, we focus on the changes needed in a processor with out-of-order instruction issue and execution.

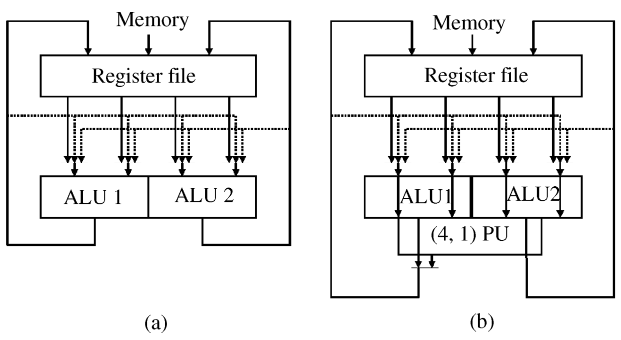

Fig.4. (a) Standard 2-way superscalar processor datapath. (b) with a $(4,1)$ PU added.

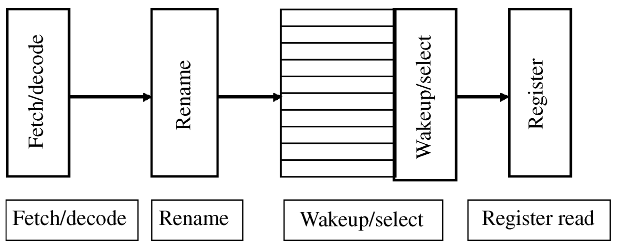

Fig.5. Pipeline front-end of an out-of-order processor.

Fig.5 shows the pipeline front-end of a generic out-of-order superscalar processor[16]. A block of instructions is fetched from the instruction cache and decoded. Their operands are renamed to physical registers, to eliminate register-name dependencies, before entering the issue window. They are issued for execution when all their source operands and required functional units become available (wakeup and select stages). Certain stages of the pipeline may take multiple cycles. An in-order processor does not need rename or select stages.

4.2 Changes to the Datapath

Fig.4b shows a $(4,1)$ permutation unit (PU) added to a standard $(4,2)$ datapath of a 2-way superscalar processor. Fig.6 shows examples of 64-bit $(4,1)$ PUs. One PU implements a 6-stage 64-bit butterfly network; a second PU implements an inverse butterfly network. The $(4,2)$ datapath in a 2-way processor limits the use of only one $(4,1)$ PU per cycle, resulting in a 2-cycle latency for an arbitrary 64-bit permutation, and a throughput of one permutation per two cycles. In a 4-way or wider processor, both PUs can be used in parallel (see Fig.7), resulting in an arbitrary 64-bit permutation every cycle.

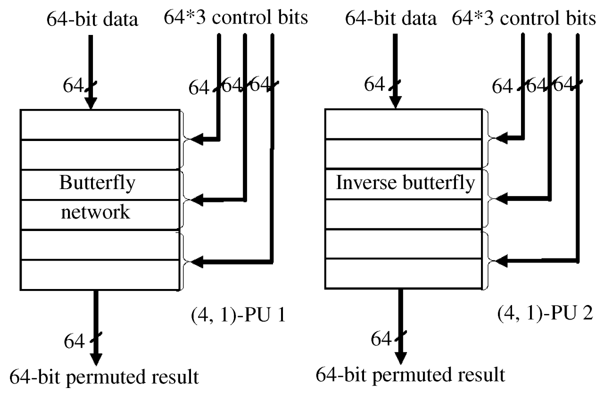

Fig.6. Two $(4,1)$ permutation units.

Adding a $(4,1)$ MOMR functional unit to a 2-way superscalar processor causes minimal datapath overhead of only one additional result multiplexer (Fig.4(b)). All the expensive register ports, data buses, and bypasses have already been provided by the $(4,2)$ datapath. Similarly, adding two $(4,1)$ MOMR units to a 4-way superscalar processor leverages the existing $(8,4)$ datapath. Two $(4,1)$ PUs are sufficient to achieve the ultimate performance of a different 64-bit permutation every cycle. A key benefit of our solution is the leveraging of existing resources in today's microprocessors, almost all of which are at least 2-way multi-issue processors.

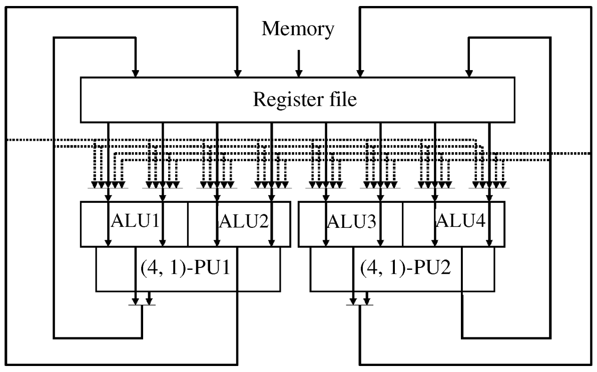

Fig.7. Two $(4,1)$ PUs in a 4-way superscalar processor.

4.3 Changes to the Control Path

We now show that the required control path changes are also minimal. MOMR method 1 requires some modifications to the pipeline control front-end as shown in Fig.8. The sequence detection unit detects an instruction group. The code transformer transforms this to a (smaller) group of internal instructions, if necessary. The multiplexers select either the original or transformed instructions to place in the issue window. A group field, denoted C, is added to each entry in the instruction window to indicate if it and its successor are in an instruction group. The wakeup/select logic is also modified so that the grouped instructions can be woken up and executed together. For MOMR method 2, since instruction groups are explicitly identified in the instructions themselves, the sequence detection unit, code transformer, and multiplexers are not needed. The rest of the control path is the same as in MOMR method 1. Below, each new change is described in detail.

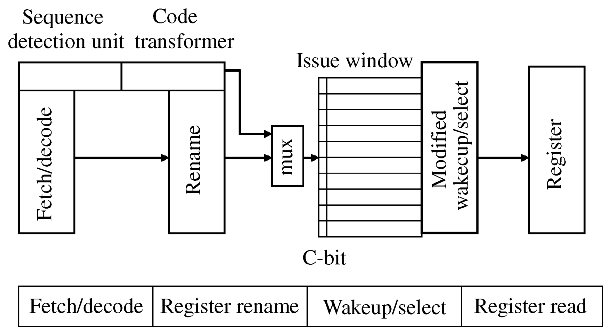

Fig.8. Modified superscalar processor pipeline frontend.

Group sequence detection. Dynamic instruction group detection is needed only by method 1. The group sequence detection unit (Fig.9) recognizes three consecutive permutation instructions that satisfy the following three criteria: 1) they have the same opcode; 2) the data source operand in a permutation instruction is the result of the previous permutation instruction; 3) they have the same destination register. It then sets the C-bits for all the instructions in the group except the last one. The comparison logic required by 2) and 3) is already present in the pipeline data dependency check logic. The gates required for 1) are a few XOR and NOR gates. For simplicity, sequences residing in two fetch blocks are not recognized to avoid keeping additional states.

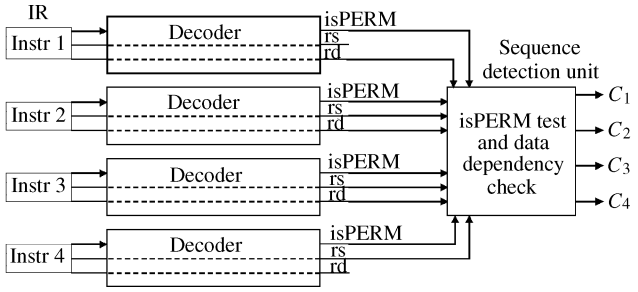

Fig.9. Functions of sequence detection unit.

Instruction transformation. The code transformer is also needed only by method 1. It transforms a group of 3-instruction sequences into internal 2-instruction sequences, by replacing the data operand in the second instruction with the configuration operand from the third instruction before discarding the third instruction (Fig.10). Then, it updates the C-bits in the newly generated internal instructions. Grouped instructions are adjacent in the issue window.

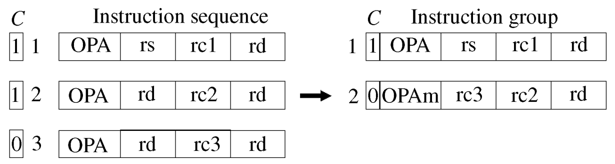

Fig.10. Code transformer transforms a 3-instruction sequence to a 2-instruction group.

Instruction wakeup. The wakeup logic needs to be modified for both methods to wake up two grouped instructions at the same time. This is necessary because otherwise the two grouped instructions might be issued separately and produce wrong results. Previously, when an instruction is ready to issue, it set its request signal to 1. The select logic selects one of the ready instructions, and sends it the grant signal. The modified wakeup logic ensures that grouped instructions request execution only when both instructions are ready. Only the first instruction in a group has its request signal set to 1; other instructions in the group always have their request signal at 0. Suppose R_i and C_i are the request signal and C field for instruction i. Instruction i follows instruction $i-1$, and instruction $i+1$ follows instruction i. The new request signal R_i for instruction i is generated as:

$$R_i' = R_i \cdot \overline{C_{i-1}} \cdot (\overline{C_i} + R_{i+1})$$

R_i and R_{i+1} on the right hand side of the equation refer to the old request signals generated when the operands are ready for instructions i and $i+1$, respectively (before MOMR is added). R_i' on the left represents the new request signal for MOMR that takes grouped instructions into consideration. The equation considers three cases for instruction i. 1) If the preceding instruction $i-1$ starts a group, instruction i never sets its request signal to 1. 2) If instruction i starts a group, it requests execution only when instruction $i+1$ is also ready. 3) Instruction i is not a grouped instruction and sets its request signal when its operands are ready, i.e., $R_i' = R_i$.

Instruction select. Suppose the $(4,1)$ PUs share buses with ALU1 and ALU2. C-bits from the issue window need to be propagated to the select logic for ALU1, ALU2, and a small control unit (see Fig.11(a)).

Assume the select logic for ALU1 selects instruction i. The control unit 1 tests the C-bit of instruction i. If instruction i's C-bit is 1, then grant both instruction i and $i+1$ and bypass the select logic for ALU2. Otherwise grant instruction i only for ALU1. The select logic for ALU2 is modified so that it does not select an instruction with its C-bit set.

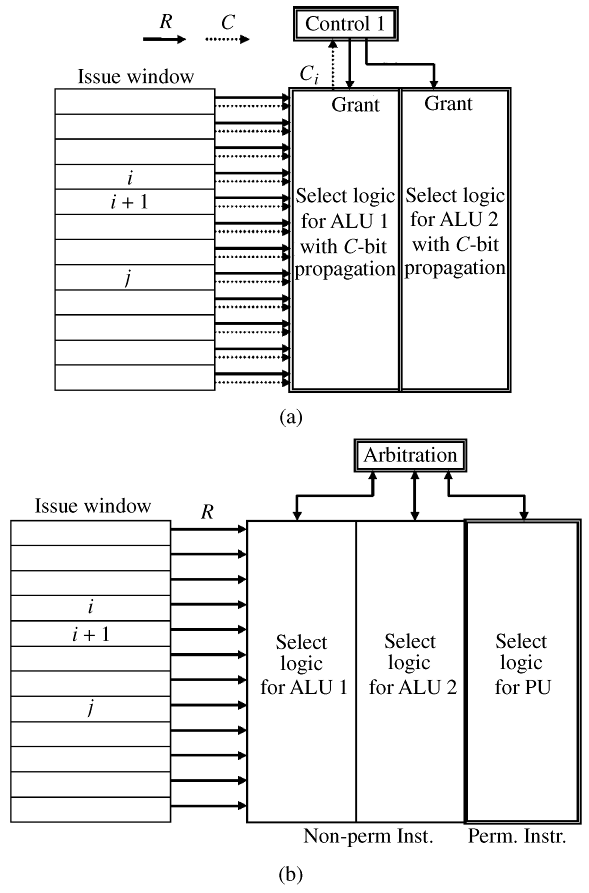

Fig.11. (a) Select logic with modifications to the select logic for ALU1 and ALU2. (b) New select logic for PU.

Alternatively, we can add a new set of select logic for the PU (see Fig.11(b)), which deals only with grouped instructions, while the select logic for ALU1 and ALU2 deals with normal instructions. The arbitration logic picks the result of either the select logic for ALU1 and ALU2 or the new select logic.

If there are multiple issue queues, such as proposed in [16], an instruction steering method can ensure that the instructions in an instruction group are dispatched to the same queue. This is easy to achieve because the two grouped instructions are adjacent.

4.4 Complexity and Delay Estimates

The modifications to the control path consist of a small amount of combinatorial logic, estimated at a few thousand gates for a 4-way superscalar processor. As comparison, the issue logic of the Compaq Alpha 21,264 processor, a 4-way superscalar RISC processor, contains about 141,000 transistors[17], making the complexity of our modifications negligible.

In terms of delay, the sequence detection unit and the code transformer run in parallel with the decode and rename logic. Due to their simple functions, they should have no impact on the processor cycle time.

Since the wakeup and select logic are already in the critical path for back-to-back executions of dependent instructions, there is a small possibility our modifications *may* increase the cycle time slightly. However, many methods have been proposed to reduce the latency of issue logic by either simplifying the instruction issue logic[16,18−20], or breaking wakeup/select to multiple stages[21,22]. Using these methods, we can integrate our modifications without affecting the processor cycle time.

4.5 Discussion

Method 1 requires a more complex group detection unit to recognize all the supported MOMR operations. Although method 2 specifies instruction groups in the instructions themselves, if multiple MOMR units are supported by the processor, it also needs a similar unit to check if *gs* and *gc* pairs define legitimate and correct groups.

In order to simplify MOMR implementations, we require that instructions in a group be consecutive in the program. We are not trying to find instructions that may be far apart in the program to be executed together. Rather, we target programs which can be recompiled, or new programs, where instructions that can be "grouped" for simultaneous execution are next to each other. Compilers need to be modified to recognize grouped instructions and not separate them during optimization. Alternatively, the MOMR instructions can be placed into regions that the compiler's optimization phases do not change.

Method 2 incurs an ISA cost of defining the *gs* and *gc* bits for instructions that can be in an instruction group. Encoding space may be tight in existing ISAs and one or two unused bits per instruction may not be available. Compilers also need to be modified to insert the *gs* and *gc* bits.

When there are different instruction groups, the microarchitecture needed to support method 2 is not significantly simpler than in method 1. However, method 2 can specify MOMR execution opportunities that are too difficult for method 1 to recognize dynamically. For example, it takes a long sequence of 64-bit multiply and add instructions to get the result equivalent to the multiplication of two 128-bit operands. With the *gs* and *gc* bits, method 2 only needs 4 instructions to specify this operation. Hence, method 2 can support a broader scope of multi-word operations.

We showed in detail how MOMR enables a 64-bit processor to achieve single-cycle 64-bit permutations, using two $(4, 1)$ permutation units. MOMR techniques can be generalized to allow n-bit permutations in n-bit processors, for values of n both smaller and larger than 64. MOMR permutation units may have to be larger than $(4, 1)$ for $n > 64$. The designer can still achieve single-cycle pipelined throughput by increasing the number of cycles of latency, the cycle time, or the area.

For example, an arbitrary 128-bit permutation of bits in a 128-bit register can be accomplished on a 128-bit processor, still using only $(4, 1)$ permutation units. The total circuitry needed for 128-bit permutation consists of a 7-stage 128-bit butterfly network followed by a 7-stage 128-bit inverse butterfly network (compare Fig.6). This can be divided into five, five, and four stages, implemented on three $(4, 1)$ MOMR units, If all three units can be pipelined and simultaneously active, then a different 128-bit permutation can be achieved each cycle. A processor that can issue 6 or more instructions per cycle will have sufficient register ports and data buses to do this quite easily.

Alternatively, two $(5, 1)$ MOMR permutation units can be used, one implementing a full 128-bit butterfly circuit and the other a full 128-bit inverse butterfly circuit. This achieves single-cycle pipelined throughput with only two cycles of latency, but a potentially longer cycle time. Other MOMR implementations for 128-bit permutation are also possible.

5 Multi-Word Arithmetic

MOMR is a general technique that can be used to perform any operations with multiple operands and multiple results, as defined in Section 2. MOMR provides either improved performance or cost-effectiveness. We now demonstrate the generality of MOMR execution to other applications, in particular, for multi-word arithmetic operations.

Public-key cryptography algorithms involve modular exponentiation, using keys that are typically 1,024 or 2,048 bits long. The exponentiation is broken down into multiplication operations and the 1,024-bit operands are broken down into sixteen 64-bit words in a 64-bit processor. Typically, two 64×64-bit multiply instructions are needed to produce the lower and higher 64-bit halves of the product, using a standard $(2, 1)$ datapath:

```
MUL.L ra, rb, rc
MUL.H ra, rb, rd
```

Actually, both halves of the product are generated by the same hardware multiplier. Two instructions are needed to generate the double-word result only because the ISA and datapath restrict an instruction to one word-sized result. If a $(2, 2)$ instruction were available, then these two instructions can be executed together on one multiplier simultaneously. MOMR method 1 can recognize this at run-time. MOMR method 2 can specify this at compile time with the *gs* and *gc* bits:

```
MUL.L.gs ra, rb, rc
MUL.H.gc ra, rb, rd
```

A 2-way supercsalar processor with two $(2, 1)$ multipliers can achieve the same performance as a single $(2, 2)$ MOMR multiplier, but with twice the area for two multipliers. Hence, MOMR execution is more cost-effective.

Alternatively, an even higher performance microprocessor may be able to afford a 128-bit multiplier. Implemented as a $(4, 2)$ MOMR multiplier, we can execute 128-bit versions of the Multiply Low and High instructions, using method 2 as follows:

```
MUL.L.gs ra1, rb1, rc1
MUL.L.gc ra2, rb2, rc2
MUL.H.gc ra1, rb1, rd1
MUL.H.gc ra2, rb2, rd2
```

All four instructions can be executed together using 128-bit multipliers with MOMR execution. To get the equivalent 256-bit product using only 64-bit multipliers and conventional $(2, 1)$ instructions, we need eight multiply and five add instructions. Larger multipliers can also be used for even further speedup.

While we described only multi-word integer multiplication used in public-key ciphers like RSA, the same technique can also be applied to polynomial multiplication in Elliptic-Curve Cryptography (ECC)[23].

6 Performance

Table 3 shows the performance of MOMR architecture for different ciphers.

For bit permutation, we test DES encryption (DES enc) and round key generation (DES key) with the fastest software program on existing processors which uses table lookup to perform bit permutations (columns a and b). We also test DES using an enhanced ISA that has a CROSS permutation instruction[5] added to it (columns c and d). For multi-word operations, we test integer Diffie-Hellman (column e) and binary Diffie-Hellman (used in ECC)[23].

Table 3. Speedup with and without MOMR

	a. DES enc (RISC)	*b.* DES key (RISC)	*c.* DEC enc (Opt)	*d.* DES key (Opt)	*e.* Int. DH	*f.* Bin. DH
1-way	1	1	1	1	1	1
2-way	1.49	1.04	1.50	1.19	1.76	1.61
2-way MOMR	1.89	17.64	1.70	1.42	3.02	1.67
4-way	1.73	1.05	1.65	1.31	2.70	2.31
4-way MOMR	2.12	19.81	1.91	1.60	4.87	2.36

We implement these programs using a generic 64-bit RISC processor and simulate their execution time, in cycles, on the following machine configurations:

- a single-issue processor with one set of $(2, 1)$ functional units, including a 64-bit ALU, a 64-bit shifter, a 64-bit permutation unit (for columns c and d), and a 64-bit integer or binary multiplier (for columns e and f)
- a 2-way superscalar processor with two sets of $(2, 1)$ functional units
- an enhanced 2-way superscalar processor with one MOMR functional unit active per cycle
- a 4-way superscalar processor with two sets of $(2, 1)$ functional units
- an enhanced 4-way superscalar processor with two MOMR functional units active per cycle.

For DES, the MOMR units used are $(4, 1)$ butterfly and inverse butterfly permutation units. For DH, the MOMR unit used is a $(4, 2)$ multiplier, i.e., a 128-bit multiplier, which we are now able to utilize, but could not previously because of ISA limitations. We assume a latency of three cycles for a 64-bit integer multiplier, and five cycles for the 128-bit multiplier. MOMR method 1 or 2 can be used for the DES programs. Method 2 is used for the DH programs, which are recoded using the *gs* and *gc* bits to specify instruction groups.

The cache parameters used in the DES simulations are 16 kilobytes L1 data cache and 256 kilobytes L2 unified cache with 10-cycle and 50-cycle miss penalties, respectively.

Table 3 shows that the speedup of our enhanced 2-way MOMR processor over a single-issue machine is greater than that of the standard 2-way multi-issue processor, in every case. This is also true for our 4-way MOMR compared to a standard 4-way processor. In fact, our 2-way MOMR is even faster than a standard 4-way superscalar processor, except for binary DH.

For DES, the performance gain is very pronounced for key generation (17.6X and 19.8X speedup in column b), where permutation operations are more frequent than for encryption. The MOMR speedup is less when compared to the enhanced ISAs (columns c and d) than when compared to existing ISAs (columns a and b). This is because the enhanced ISAs already have fast new permutation instructions (e.g., CROSS or OMFLIP). The number of instructions for a 64-bit permutation has already been reduced from over 20 to at most 6, and most of the memory accesses have also been eliminated, resulting in much fewer cache misses. Even then, our MOMR execution is still 13% to 22% faster.

For the integer DH, MOMR execution provides significant super-linear speedup of 3.02 for a 2-way superscalar processor enhanced with one $(4, 2)$ MOMR unit, and 4.87 for 4-way superscalar enhanced with two $(4, 2)$ MOMR units. This is because MOMR architecture allows the inclusion of wider functional units such as 128-bit multipliers, reducing the number of instructions and cycles needed to complete a 1,024 by 1,024-bit multiplication. The speedup for binary DH is less since the operands are smaller 163-bit polynomials in this ECC public-key cipher.

7 Conclusions

We identify bit and multi-word operations as two new challenges in high-performance cryptographic processing for word-oriented processor architectures. This insight is more useful from a broad architectural perspective than just accelerating a few special-purpose operations.

We propose a unified solution to both these challenges in terms of a generalized MOMR execution model which enables datarich execution. MOMR methods can be used to speed up both symmetric-key block ciphers using arbitrary bit permutation instructions, as well as public-key ciphers using multi-word multiplication. Furthermore, we show that MOMR implementation incurs minimal incremental cost, since it leverages common micro-architecture trends like multi-issue processors, whether superscalar or VLIW.

We also show how any one of $n!$ bit permutations can be achieved each cycle using MOMR permutation functional units. This is a very significant result since previously arbitrary n-bit permutations took $O(n)$ cycles. Even with our recent proposals of bit permutation instructions[3−7], $O(\log(n))$ cycles is needed. We show how a different 64-bit dynamicallyspecified permutation can be achieved *every cycle* by a 4-way superscalar processor with MOMR execution.

Our software solution for achieving permutations is more powerful than a hardware solution the latter can only achieve a few fixed permutations, while our solution can achieve all possible data-dependent permutations. We hope that this will stimulate the design of new ciphers and other algorithms that can take advantage of such a powerful operation – an arbitrary bit permutation that can be done in about the same time as an addition and in less time than a multiplication. Cryptographers can use arbitrary bit permutations freely in their new algorithms if processors include our bit permutation instructions.

MOMR execution enables a very flexible extension of standard ISAs to support datarich operations of many flavors. We do not have to decide whether instruction formats of future processors should support $(3,1)$, $(4,1)$, $(2,2)$, $(3,2)$, or $(4,2)$ functional units, all of which are useful for different operations. They can all be supported as MOMR functional units in a 2-way superscalar machine with minimal changes, as we have shown. Basing MOMR implementations on the $(2k,k)$ datapath of a k-way multiple-issue processor gives the flexibility of supporting all MOMR functional unit sizes covered by these existing datapath resources. Sometimes MOMR functional units improve the performance, and other times they improve the cost-performance

Finally, we show that the fundamental choice of a word as the atomic unit upon which a processor is optimized is still sound. Our MOMR proposal allows even challenging bit and multi-word operations to be achieved efficiently on word-oriented processors, with minimal incremental costs.

Acknowledgements We thank A. Murat Fiskiran for the binary DH performance numbers.

References

[1] Schneier B. Applied Cryptography. 2nd Ed., John Wiley & Sons, Inc., 1996.

[2] NIST (National Institute of Standards and Technology). Advanced Encryption Standard (AES). FIPS Pub. 197, November 2001.

[3] Lee R B, Shi Z, Yang X. Efficient permutation instructions for fast software cryptography. *IEEE Micro,* December 2001, 21(6): 56–69.

[4] Yang X, Vachharajani M, Lee R B. Fast subword permutation instructions based on butterfly networks. In *Proceedings of SPIE 2000,* January 2000, pp.80–86.

[5] Yang X, Lee R B. Fast subword permutation instructions using omega and flip network stages. In *Proceedings of the International Conference on Computer Design,* September 2000, pp.15–22.

[6] Shi Z, Lee R B. Bit permutation instructions for accelerating software cryptography. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors,* July 2000, pp.138–148.

[7] McGregor J P, Lee R B. Architectural enhancements for fast subword permutations with repetitions in cryptographic applications. In *Proceedings of the International Conference on Computer Design,* September 2001, pp.453–461.

[8] Lee R B. Subword parallelism with MAX-2. *IEEE Micro*, August 1996, 16(4): 51–59.

[9] Diefendorff K *et al.* AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, March 2000, 20(2): 85–95.

[10] IA-64 application developer's architecture guide, Intel Corp., May 1999.

[11] Princeton Architecture Lab for Multimedia and Security, http://palms.ee.princeton.edu/.

[12] Burke J, McDonald J, Austin T. Architectural support for fast symmetric-key cryptography. In *Proceedings of ASPLOS 2000,* November 2000, pp.178–189.

[13] Wu L, Weaver C, Austin T. CryptoManiac: A fast flexible architecture for secure communication. In *Proceedings of the 28th International Symposium on Computer Architecture,* June 2001, pp.110–119.

[14] Lee R B, Shi Z, Yang X. How a processor can permute n bits in $O(1)$ cycles. In *Proceedings of Hot Chips 14 – A Symposium on High Performance Chips,* August 2002.

[15] Lee R B, Yang X, Shi Z J. Validating word-oriented processors for bit and multi-word operations. In *Proceedings of the Asia-Pacific Computer Systems Architecture Conference,* September 2004, pp.473–488.

[16] Palacharla S, Jouppi N P, Smith J E. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture,* 1997, pp.206–218.

[17] Farell J A, Fischer T C. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, May 1998, 33(5): 707–712.

[18] Onder S, Gupta R. Superscalar execution with direct data forwarding. In *Proceedings of the 1998 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques,* 1998, pp.130–135.

[19] Henry D S, Kuszmaul B C, Loh G H, Sami R. Circuits for wide-window superscalar processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture,* 2000, pp.236–247.

[20] Canal R, Gonzalez A. A Low-complexity issue logic. In *Proceedings of the 14th international conference on Supercomputing,* 2000, pp.327–335.

[21] Stark J, Brown M D, Patt Y N. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33th Annual ACM/IEEE International Symposium on Microarchitecture,* 2000, pp.57–66.

[22] Brown M D, Stark J, Patt Y N. Select-free instruction scheduling logic. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture,* December 2001, pp.204–213.

[23] Fiskiran A M, Lee R B. Evaluating instruction set extensions for fast arithmetic on binary finite fields. In *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors* (*ASAP*)*,* September 2004, pp.125–136.

Ruby B Lee is the Forrest G. Hamrick Professor of Engineering and Professor of Electrical Engineering at Princeton University, with an affiliated appointment in the Computer Science Department. She is the director of the Princeton Architecture Laboratory for Multimedia and Security (PALMS). Her current research is in designing security and new media support into core computer architecture and designing architectures resilient to Internet-scale epidemics. She is a Fellow of the ACM and the IEEE, Associate Editor-in-Chief of *IEEE Micro* and Editorial Board member of *IEEE Security and Privacy*.

Prior to joining the Princeton faculty in 1998, Dr. Lee served as chief architect at Hewlett-Packard, responsible at different times for processor architecture, multimedia architecture and security architecture. She was a key architect of PA-RISC used in HP workstations and servers, and of multimedia instructions for microprocessors. She was Consulting Professor of Electrical Engineering at Stanford University. She has a Ph.D. in Electrical Engineering and a M.S. in Computer Science, both from Stanford University, and an A.B. with distinction from Cornell University, where she was a College Scholar. She has been granted over 115 United States and international patents.

Xiao Yang is a Ph.D. candidate in the Department of Electrical Engineering at Princeton University. His research area is in computer architecture with special focus on high performance, scalable architecture for 3D graphics processing. He has a M.S. in Physics from Northwestern University and a B.S. in Physics from Peking University, P.R. China.

Z. Jerry Shi is an assistant professor in the Department of Computer Science and Engineering at the University of Connecticut. He received his Ph.D. in Electrical Engineering from Princeton University in 2004, and this work was done while he was a student at Princeton. He received his B.S. and M.S. from the Computer Science and Technology Department at Tsinghua University, China, in 1992 and 1994, respectively. Dr. Shi is a member of the ACM and the IEEE. His research areas are in computer architecture, cryptography, and high performance, secure computer systems.