Validating Word-oriented Processors for Bit and Multi-word Operations

Ruby B. Lee, Xiao Yang, and Zhijie Jerry Shi

Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Princeton University
{rblee, xiaoyang, zshi}@princeton.edu

Abstract. We examine secure computing paradigms to identify any new architectural challenges for future general-purpose processors. Some essential security functions can be provided by different classes of cryptography algorithms. We identify two categories of operations in these algorithms that are not common in previous general-purpose workloads: bit operations within a word and multi-word operations. Both challenge the basic word orientation of processors. We show how very complex bit-level operations, namely arbitrary bit permutations within a word, can be achieved in O(1) cycles, rather than O(*n*) cycles as in existing RISC processors. We describe two solutions: one using only microarchitecture changes, and another with Instruction Set Architecture (ISA) support. We generalize our solutions to define *datarich execution* with MOMR (Multi-word Operands Multi-word Result) functional units. This can address both challenges, leveraging available resources in typical processors with minimal additional cost. Thus we validate the basic word-orientation of processor architectures, since they can also provide superior performance for both bit and multi-word operations needed by cryptographic processing.

1. Introduction

The dependence on the public Internet and wireless networks in modern society poses a growing need for secure communications, computations and storage. To provide basic security functions like data confidentiality, data integrity, and user authentication, different classes of cryptographic algorithms can be used with security protocols at network, system or application levels. Not only network transactions need to be protected, all data and programs may also need these security functions. As secure computing paradigms become more pervasive, it is likely that such cryptographic computations will become a major component of every processor's workload. Understanding the new requirements of secure information processing is critical for the design of all future processors, whether general-purpose, application-specific or embedded. In this paper, we especially target the needs of high performance microprocessors.

Basic security functions include confidentiality, integrity and authentication. Confidentiality of messages transmitted over the public networks, and of data stored in disks, can be achieved by encrypting the data, using symmetric-key cryptography algorithms such as DES [1], and AES [2]. Data integrity, where data is not changed in transit or in storage, can be accomplished with one-way hash functions such as SHA and MD-5 [1]. Authenticating users and devices remotely across the Internet can be accomplished with public-key cryptographic algorithms such as Diffie-Hellman and RSA [1]. They also allow digital signatures and the exchange of a shared secret key across the Internet.

We observe two categories of new requirements imposed by these three classes of cryptographic algorithms: bit-oriented operations and multi-word operations. Both challenge the basic word-orientation of modern processors. Symmetric-key cryptography introduces a new requirement: bit-level permutations. Previously, the bit-oriented operations in general-purpose workloads were SHIFT instructions and logical operations like AND, OR, XOR and NOT. These are supported efficiently by simple single-cycle instructions. Public-key cryptography introduces the other new requirement: multi-word arithmetic. While multiword integer arithmetic has been a requirement in previous high-precision integer computations, its need remained relatively low since the basic word size in general-purpose processors has increased from 16 to 32 to 64 bits. Frequent use of public-key cryptography algorithms may significantly increase the need for multi-word arithmetic. For example, multiplication of two 1024-bit operands in RSA involves two 16-word operands, if each word is 64 bits. If a hardwired multiply instruction operates on two words, many such 64-bit multiply instructions are needed, as well as add operations to accumulate the result. 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. These requirements have more impact on performance than defining a new instruction or special-purpose functional unit for accelerating a particular cryptographic primitive. They also challenge the atomic word-orientation of processors, since they emphasize bit operations within a word, and operations requiring operands much larger than a word.

A second contribution is showing how arbitrary bit-level permutations can be accomplished very efficiently in only 1 or 2 cycles.

A third contribution is a generalized architectural solution that allows high-performance processors to support *datarich* operations with flexible, multi-word operands and multi-word result (MOMR) functional units. Our generalized solution supports both high performance bit permutations and multi-word operations. Hence, the basic word-orientation of processors is still a good design choice, since both bit-oriented and multi-word oriented operations can also be supported very efficiently.

In Section 2, we describe past work on permutation instructions, including how our recent past work has reduced the time taken to achieve any n-bit permutation down from $O(n)$ to $O(\log(n))$ instructions and cycles. In Section 3, we propose two new architectural methods for further bringing this down to $O(1)$ cycles. One method is purely micro-architectural, and the other involves new ISA. In Section 4, we describe the changes in the datapath and control path needed to implement our two methods. In Section 5, we generalize these two methods to solve the second challenge of achieving multi-word operations efficiently in word-oriented processors. In Section 6, we discuss performance, and conclude in Section 7.

2. Past Work

Past work in accelerating cryptographic processing included many hardware ASIC (Application Specific Integrated Circuit) implementations of specific ciphers. For programmable solutions, new instructions were proposed to accelerate symmetric-key ciphers in general-purpose processors [3], and in cryptographic coprocessors like Cryptomaniac [4] for ciphers used in secure networking protocols. In contrast, we do not propose any specific new instructions in this paper, but rather new methodologies for bringing more data to the functional units. This *datarich* computation is done with very low overhead, utilizing the datapaths and control already provided for superscalar execution found in most microprocessors, including out-of-order superscalar machines.

The datarich methodology allows us to accelerate both bit permutations used for symmetric-key ciphers, and multi-word operations used in public-key ciphers. It allows us to achieve one of our major contributions: performing arbitrary bit permutations in 1 or 2 cycles – a significant improvement over our recent past work achieving O(log(n)) instructions and cycles [5], which we describe further below.

Performing bit-level permutation has been a hard problem for word-oriented general-purpose processors. Previously, processors 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. While some n-bit permutations can be achieved with fewer instructions, allowing arbitrary, data-dependent n-bit permutations is very slow. Conventional logical and shift instructions take O(n) cycles to achieve any one of n! permutations [5]. Alternatively, table lookup methods can be used, but this is limited to a few fixed permutations due to the high memory space requirement, and cache misses cause performance degradation.

More recently, permutation instructions have been introduced into certain microprocessors as multimedia ISA extensions to handle the re-arrangement of subwords packed in registers. Examples are MIX and PERMUTE in HP's MAX-2 [6], VPERM in Motorola's AltiVec [7], and MIX and MUX in IA-64 [8]. However, these instructions can only handle subword sizes down to 8 bits. They do not provide a general solution for performing arbitrary bit-level permutations efficiently.

Very recently, researchers have tackled the general bit permutation problem, and defined new permutation instructions that can achieve any n-bit permutation with only log(n) instructions. Several approaches were proposed. The CROSS [9] and OMFLIP [10] permutation instructions each performs the equivalent function of two stages of a "virtual" interconnection network. A sequence of log(n) CROSS or OMFLIP instructions can build a 2log(n)-stage virtual network that can achieve any one of the n! permutations. Another approach was the GRP instruction [11], which partitions the data bits into two groups. At most log(n) GRP instructions are sufficient to achieve any one of n! permutations [11]. A third approach involves specifying the order of the indices of the source bits in the permuted result. Examples are PPERM[11], and SWPERM and SIEVE [12]. The XBOX instruction [3] is similar to PPERM.

A comparison of CROSS, OMFLIP, GRP, and PPERM is presented in [5]. CROSS, OMFLIP and GRP all achieve arbitrary 64-bit permutations in 6 instructions. PPERM and SWPERM with SIEVE require more than log(n) instructions, but can be executed in as few as 4 cycles on a 4-way superscalar machine. Unfortunately, CROSS, OMFLIP and GRP cannot achieve speedup with superscalar machines, due to the strict data dependency between the sequence of log(*n*) permutation instructions. Below, we show how this data dependency can be overcome, so that arbitrary 64-bit permutations can be achieved in 1 or 2 cycles, rather than log(*n*) = 6 cycles.

This paper extends the concepts we presented in [13] with new work on the detailed ISA or microarchitectural changes required, and the detailed implementation in an out-of-order processor.

3. Achieving Arbitrary 64-bit Permutations in 1 or 2 Cycles

The reason log(*n*) instructions are needed to achieve any permutation of *n* bits is because *n*log(*n*) configuration bits are needed to specify an arbitrary *n*-bit permutation [5][11]. Since a typical instruction reads up to 2 source operands and produces 1 result, a permutation instruction uses one source operand for the data and the other for *n* bits of 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 supply the *n*log(*n*) configuration bits and the data to be permuted, to achieve any *n*-bit permutation [5]. If all *n*log(*n*) configuration bits and the *n* data bits to be permuted can be specified by a single instruction, then it may be possible to execute any arbitrary *n*-bit permutation in 1 instruction. Hence, the main performance limiter is the ISA instruction format and the datapaths that support only two *n*-bit operands per instruction, and a design goal of not having to save states between permutation instructions. This is a reasonable goal since it reduces context-switch and operating system overhead.

Suppose that the latency through the permutation functional unit is not a cycle-time limiter. Then, the problem reduces to the following: how can *n*(log(*n*)+1) bits be sent from general registers to a permutation functional unit (PU) in a single instruction? If each register is *n* bits, this means sending (log(*n*)+1) register values, or operands, to a functional unit. We propose two methods to solve this problem. Method 1 identifies instruction groups dynamically with microarchitecture techniques; method 2 employs ISA techniques to identify instruction groups statically.

3.1. Datapath, MOMR and Instruction Groups

We first define some new architectural terms: 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* 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 in one cycle. In general, a *k*-way multi-issue processor has a (2*k*,*k*) datapath, supporting the simultaneous execution of *k* standard (2,1) functional units each cycle.

A *datarich or MOMR (Multi-word Operands Multi-word Result) functional unit* in a word-oriented processor is a functional unit that requires more than the standard two word-sized operands and one word-sized result.

A sequence of consecutive instructions is called an *instruction group* if the instructions can be executed simultaneously by a datarich (or MOMR) functional unit.

Emerging secure computing paradigms may require extensive execution of algorithms where performance can be improved by the use of datarich functional units. Sometimes datarich functional units improve the performance, other times they improve the cost-performance. Our thesis is that a *k*-way multi-issue processor with a (2*k*,*k*) datapath can accommodate different types of datarich functional units, with relatively minor changes to the pipeline control logic. In the rest of Section 3 and Section 4, we illustrate two methods for datarich MOMR execution, using bit permutation as the example. In Section 5, we generalize datarich MOMR execution to operations with multi-word operands.

3.2. Method 1: microarchitecture group detection

A permutation instruction is defined as follows:

```
PERM rs,rc,rd
```

where rs contains data source, rc contains configuration bits and rd is the result. For example, PERM can be either a CROSS or OMFLIP instruction. Fig. 1(a) shows a 64-bit permutation specified with a sequence of 6 PERM instructions in 2 groups of 3 instructions each. Each group provides the data source and 3 configuration words for a (4,1) permutation unit, PU. The group is dynamically detected, and then its 3 instructions are transformed into 2 "internal" instructions. The first one supplies the data source and one configuration word, and the second one provides the other 2 configuration words. When all the operands are ready, the two “internal” instructions are issued for execution simultaneously on one (4,1) PU. Hence, the 6 instructions can be transformed into 4 internal instructions in 2 groups and executed over 2 cycles. If there are two or more (4,1) PUs, we can pipeline the executions of the 2 groups and achieve a throughput of one permutation per cycle.

3.3. Method 2: new ISA for group identification

In the second method, we enhance the conventional RISC instruction encoding with 2 new subop bits, gs and gc, for identifying instructions which start a group (gs=1) or continue a group (gc=1). The meanings of these 2 bits are shown in .

The permutation instruction is defined as:

```
PERM,subop   rs1,rs2,rd
```

where subop contains the gs and gc bits. If gs is set, the instruction is the first in a 2-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 2 configuration words for the (4,1) PU. We specify this 2-instruction group as follows:

```
PERM,gs   rs,rc1,rd
PERM,gc   rc2,rc3,rd
```

Unlike method 1, method 2 does not need the dynamic group detection and instruction transformation. It also helps reduce static code size, since only 4 permutation instructions (rather than 6) are required in the program. Similar to method 1, when all the source operands for the 2 grouped instructions are ready, they are issued for execution together on one (4,1) PU.

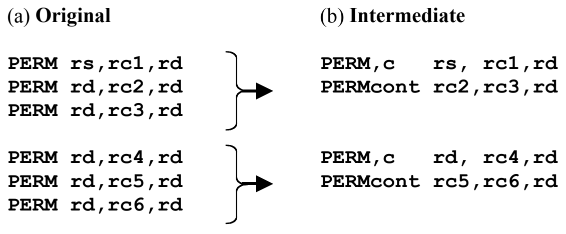

Fig. 1. Instruction transformation for method 1

Table 1. Meanings of 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

4. Microarchitectural Changes

In this section, we show how either of the two above methods can leverage the resources already present in a superscalar processor, with minimal additional cost. We first describe a typical superscalar processor in Section 4.1, then detail changes that must be made to its datapath and control path in Sections 4.2 and 4.3, respectively.

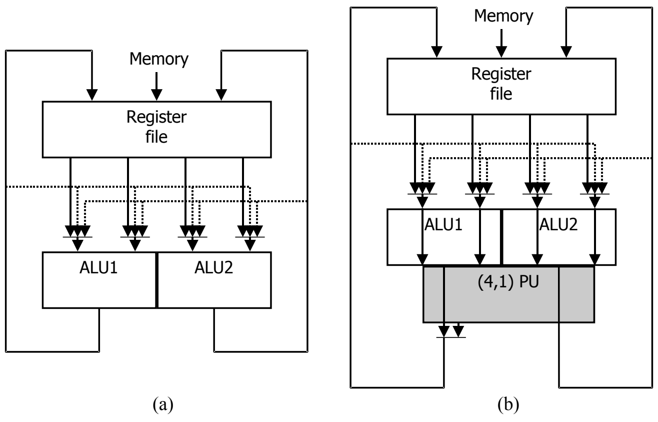

Fig. 2. (a) Standard 2-way superscalar processor datapath; (b) with a (4,1) PU added

4.1. Baseline microarchitecture

Fig. 2(a) shows a standard 2-way superscalar RISC processor with a (4,2) datapath, i.e., 4 register read ports, 2 write ports and associated data buses and bypass paths.

Fig. 3 shows the pipeline frontend of a generic out-of-order superscalar processor [14]. A block of instructions is fetched from the instruction cache. These instructions are then decoded and their operands renamed (to physical registers to eliminate register-name dependencies) before entering the issue window. They will be issued for execution when all their source operands and required functional units become available (wakeup and select stage). Certain stages of the pipeline may take multiple cycles. For an in-order issue processor, there are no rename or select stages.

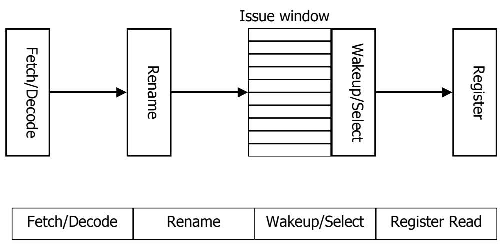

Fig. 3. Generic out-of-order superscalar processor pipeline frontend

4.2. Changes to the datapath

Fig. 2(b) shows a (4,1) permutation unit (PU) added to a standard (4,2) datapath of a 2-way superscalar processor. Fig. 4 shows an implementation of the PU based on the butterfly network. There are 2 separate PUs, one contains a 6-stage butterfly network and the other contains an inverse butterfly network. In a 2-way processor, we have both of them in the datapath, but only one PU is used at a time, resulting in a 2-cycle latency and a throughput of one permutation per 2 cycles. In a 4-way or wider processor, we can use both of them in parallel. Then we can pipeline the permutation operation and achieve one permutation per cycle throughput (Fig. 5).

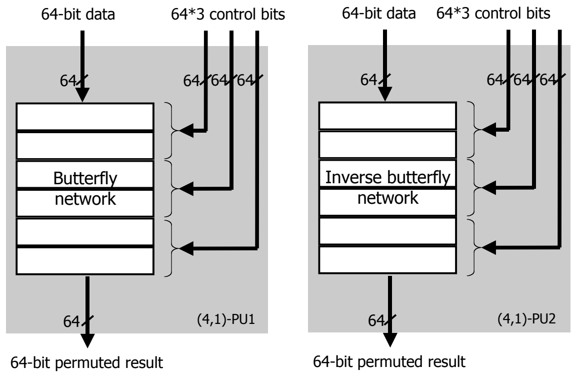

Fig. 4. One implementation of (4,1) permutation FU

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

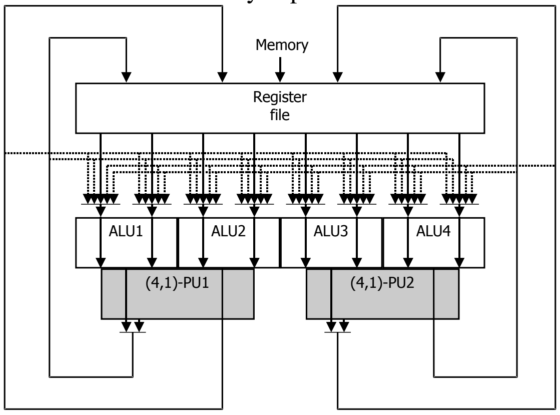

Fig. 5. Two (4,1) permutation FUs in 4-way superscalar processor

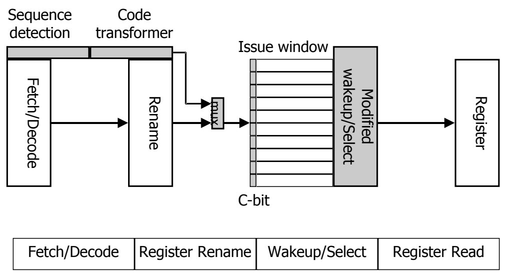

Fig. 6. Modified superscalar processor pipeline frontend

4.3. Changes to the control path

We now show that even the required control path changes are minimal. Method 1, which uses the microarchitecture to detect a sequence of dependent instructions that can be executed together, requires some modifications to the pipeline control front-end as shown in Fig. 6. The sequence detection unit detects sequences of 3 permutation instructions. These sequences are then transformed to groups of 2 instructions by the code transformer. The muxes pick the correct inputs to the issue window between the original instructions and the transformed instructions. A 1-bit field reserved for a C-bit is added to each entry in the instruction window to denote whether the corresponding instruction and the following one are in a group. The wakeup/select logic is also modified so that the 2 grouped instructions can be woken up and executed together. For method 2, since instruction groups are explicitly identified by instruction subop bits, the sequence detection unit, code transformer and muxes are not needed. The rest of the control path is the same as for method 1.

Group sequence detection. Dynamic instruction group detection is needed only by method 1. The group sequence detection unit (Fig. 7) recognizes 3 consecutive permutation instructions in a fetch block that satisfy the following two criteria: they have the same opcodes and the data source operand in a permutation instruction is the result of the previous permutation instruction. It then sets C-bits for the first instruction of the detected group sequence. For simplicity, sequences residing in two fetch blocks are not recognized to avoid keeping additional states.

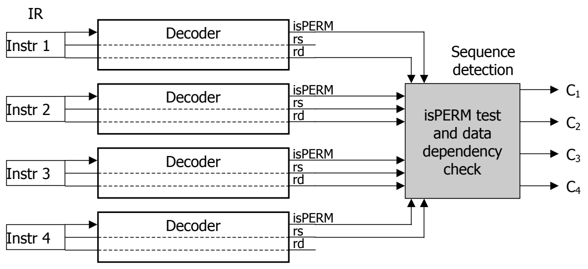

Fig. 7. Functions of sequence detection unit

Instruction transformation. The code transformer is also needed only by method 1. It transforms the group of 3-instruction sequences into 2-instruction sequences (see Fig. 1). The 2 new instructions are generated according to the C-bits produced by the sequence detection unit and the renamed operands of the original 3 instructions. The code transformer replaces the data operand in the second instruction with the configuration operand from the third instruction before discarding the third instruction. Then, it updates the C-bits in the newly generated instructions. An instruction that has its C-bit set starts a group. Grouped instructions are adjacent in the issue window. Fig. 8 shows the functions of the code transformer.

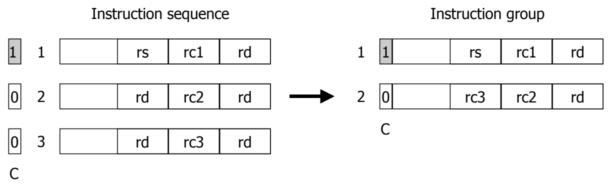

Fig. 8. Code transformer transforms 3-instruction sequence to 2-instruction group

Instruction wakeup. Fig. 9 shows the modified wakeup logic needed by both methods to wake up the 2 grouped instructions together. This is necessary because otherwise the 2 instructions might be issued separately, producing the wrong result. Previously, an instruction is ready to issue when both of its source operands are ready. The modified wakeup logic ensures that grouped instructions become ready only when all the source operands in the group are ready.

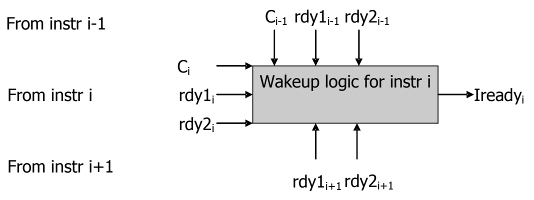

Fig. 9. Modified instruction wakeup logic

Instruction select. We can modify the select logic for ALU1 and ALU2 to handle the permutation unit as well. This is achieved by adding C-bit propagation to the original select logic for ALU1 and ALU2 and 2 small control units, as shown in Fig. 10(a). Assume the select logic for ALU1 selects instruction i. The control unit 1 tests the C-bit of i. If i's C-bit is set, then grant both instruction i and i+1 and bypass the select logic for ALU2. Otherwise grant i and proceed to select logic for ALU2. Suppose instruction j is selected for ALU2. The control unit 2 then tests the C-bit of j. We grant instruction j only if j's C-bit is not set.

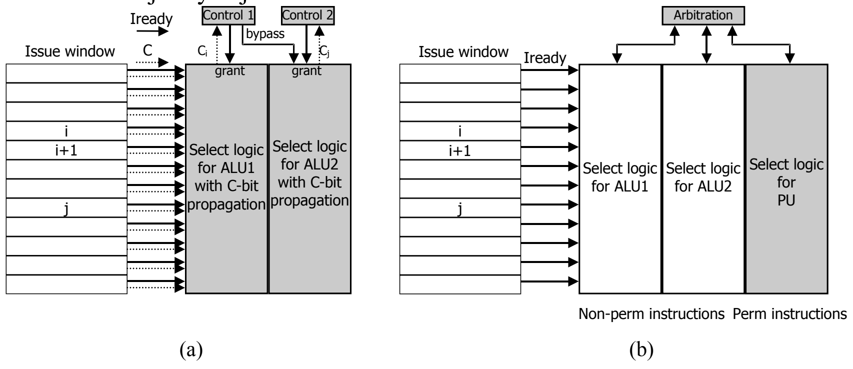

Fig. 10. (a) Select logic with modifications on the original select logic for ALU1 and ALU2; (b) Select logic with new set of logic for PU

Alternatively, we can add a new set of select logic for the PU, which deals only with instructions with C-bits set, while the select logic for ALU1 and ALU2 deals with normal instructions. The arbitration unit picks the result of either the select logic for ALU1 and ALU2 or the new select logic. (see Fig. 10(b)).

If there are multiple issue queues, such as proposed in [14], we can devise an instruction steering method so that the 2 permutation instructions in a group are dispatched to the same queue. This is easy to achieve because the 2 grouped instructions are adjacent. If the C-bit of the instruction at the head of a queue is set, we grant this instruction together with the following one.

4.4. Complexity and delay of control path modifications

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 21264 processor, a 4-way superscalar RISC processor, contains about 141000 transistors [15], 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, our modifications may increase the cycle time. However, many methods have been proposed to reduce the latency of issue logic by either simplifying the instruction issue logic [14][16][17][18], or breaking wakeup/select to multiple stages [19][20] in order to achieve fast instruction scheduling. By incorporating these methods, we can integrate our modifications without affecting the processor cycle time.

5. Generalization to Multi-word Operations

We define *multi-word operations* as operations that use more than 2 word-sized operands and produce more than 1 word-sized result, i.e., they are operations that could use datarich MOMR functional units. Arbitrary bit permutation is one example of multi-word operations since the configuration bits span multiple words. Other multi-word operations include the multiplication of two 16-word operands in a 64-bit processor, for a public key algorithm like RSA using 1024-bit keys. If larger hardware multipliers can be accommodated within a high performance microprocessor, we can speed up the multiword multiplication by producing longer (and fewer) partial products with each instruction, resulting in fewer instructions needed to accumulate the partial products to get the final result. In particular, if the implementation can afford larger multipliers, we want to eliminate the ISA restriction of only performing the multiplication of two word-sized operands per instruction.

5.1. Multiplication of Multi-word Operands

The use of MOMR methods for speeding up *n*-bit permutations was described in Sections 3 and 4. We now describe how MOMR methods may be used to accelerate the multiplication of multi-word operands. Let the original multiply instructions be:

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

Two 64-bit registers ra and rb are multiplied together to generate the low and high 64 bits of the result in rc and rd, in successive instructions. Actually, both halves of the product are generated by the same hardware multiplier at the same time, and it is only because of the ISA restriction of one word-sized result per instruction that two separate instructions have to be used to generate the double-word result. If a (2,2) instruction were available, then these two instructions can be executed together on one multiplier simultaneously. Method 1 can recognize this case at run-time. 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, in method 2 as follows:

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

The first two MUL,L instructions would be executed together as a group on a 128-bit multiplier to generate the low 128 bits of the result. The next two MUL,H instructions generate the high 128 bits of the result. To get the equivalent 256-bit product using only 64-bit multipliers and conventional (2,1) instructions, we need to do 8 multiply's and 5 add's. A 128-bit multiplier can also be used for (4,4) MOMR execution, where all 4 instructions above belong to the same group and are executed together. Larger multipliers can also be used, for even further speedup.

5.2. Datarich MOMR execution

Table 2. Architectual methods for MOMR execution

Steps	Method 1: microarchitecture detected groups	Method 2: ISA specified groups
Group detection: Recognize a small set of pre-defined groups of instructions that can be executed together on a MOMR functional unit in the same cycle.	Need to consult a table like Table 3 to check multiple combinations of opcodes and data dependencies so as to recognize all the supported multi-word operations.	Although groups are already defined in the ISA, still need to consult a table similar to that for method 1 to determine whether gs and gc define legitimate (and complete) groups.
Instruction transformation: Transform the instructions in a group to fewer instructions with some operand register re-packaging, if necessary.	For different multi-word operations, different transformations are needed.	Set C-bits in microarchitecture according to gs and gc bits in instructions.
Wakeup and select: Wake up and select the instructions to be executed on one MOMR functional unit together	Similar to permutation-only case. The logic shown in Section 4 deals with simultaneous executions of up to 2 instructions. It can be extended if 3 or more instructions are to be issued together. This may result in increased latency and more stages for wakeup and select.	

We now define generalized MOMR or datarich execution. In Table 2 and Table 3, method 1 achieves a microarchitecture solution for MOMR execution, while method 2 yields an ISA solution, where the MOMR operations are explicitly specified with new gs and gc bits in the instruction encoding. The steps in these 2 methods are listed in Table 2. Method 1 requires a more complex group detection unit to recognize all the supported multi-word operations. For method 2, a similar unit is also necessary, but to check the correctness of groups. Examples of the criteria for recognizing or checking instruction groups are given in Table 3. The first 3 columns specify the instructions in the instruction stream, and the last 2 specify the data dependencies they must satisfy.

Table 3. Examples of Group Definitions

	Instr i	i+1	i+2	Dependency criteria	Remarks
Method 1	PERM	PERM	PERM	$rd_i=rs1_{i+1}$ & $rd_{i+1}=rs1_{i+2}$ & $rd_i!=rs2_{i+1}$ & $rd_i!=rs2_{i+2}$ & $rd_{i+1}!=rs2_{i+2}$	Serial dependency No RAW hazard*
	MUL,L PMIN	MUL,H PMAX		$rs1_i=rs1_{i+1}$ & $rs2_i=rs2_{i+1}$ & $rd_i!=rd_{i+1}$ & $rd_i!=rs1_{i+1}$ & $rd_i!=rs2_{i+1}$	Same sources Diff dest for H, L No RAW hazard
Method 2	PERM,gs	PERM,gc		$rd_i!=rs1_{i+1}$ & $rd_i!=rs2_{i+1}$	No RAW hazard
	MUL,L,gs PMIN,gs	MUL,H,gc PMAX,gc		$rs1_i=rs1_{i+1}$ & $rs2_i=rs2_{i+1}$ & $rd_i!=rd_{i+1}$ & $rd_i!=rs1_{i+1}$ & $rd_i!=rs2_{i+1}$	Same source Diff dest for H, L No RAW hazard

* Since a PERM group is composed of 3 serially dependent instructions, there must be data dependencies (RAW hazards) between adjacent PERM instructions. No RAW hazard here means no additional data dependencies other than those required for a serial chain. For multi-word operations (all the rest), no RAW hazard means no RAW data dependencies.

In order to simplify the architectural solution, we require that instructions to be executed together as a group be consecutive in sequential program order. That is, we are not trying to look through the whole program to find instructions that may be far apart which can be executed together in the same cycle. Rather, we target programs which can be re-compiled, or new programs, so that instructions that can be "grouped" for simultaneous execution are next to each other.

The ISA cost of method 2 is that we must define the gs bit for every instruction that can serve as the start of a multi-word operation and the gc bit for all instructions that can act as continuation instructions in a group. Encoding space may be tight in existing ISAs and one or two unused bits per instruction may not be available.

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 (last 2 rows in Table 3). Therefore, method 2 can support a broader scope of multi-word operations.

6. Performance

We test two distinct aspects of our new architecture: support for fast bit permutations and for multi-word operations. Table 4 illustrates the performance of our architecture.

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 then test DES using an enhanced ISA that has an OMFLIP permutation instruction [10] added to it (columns c and d). For multi-word operations, we test integer Diffie-Hellman (column e).

We implement these programs using a generic 64-bit RISC processor. First, we obtain the execution time, in cycles, of the programs running on 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 multiplier (for column e). Second, the same programs are executed on a standard 2-way superscalar processor with two sets of (2,1) functional units. The speedup is shown in the first row of Table 5.

Then, we simulate the programs on an enhanced 2-way superscalar processor with one MOMR functional unit. For DES, the MOMR unit is a (4,1) Butterfly permutation unit as detailed in Sections 3 and 4. For DH (columns e), the MOMR unit is a (4,2) multiplier as described in Section 5. This is a 128-bit multiplier, which we are now able to utilize, but could not previously because of ISA limitations in a standard 64-bit superscalar processor. We assume a latency of 3 cycles for a 64-bit multiplier, and 5 cycles for the 128-bit multiplier. Either method 1 or 2 can be used in the DES programs (columns a-d). Method 2 is used for the DH program which is re-coded using the new ISA features to specify grouped instructions with the gs and gc bits. 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 4. Speedup of execution time

	a. DES enc	b. DES key	c. DES enc	d. DES key	e. Integer DH	f. Binary ecDH
2-way vs. 1-way	1.49	1.04	1.50	1.19	1.81	1.97
2-way MOMR vs. 1-way	1.89	17.64	1.70	1.42	3.63	2.96
2-way MOMR vs. 2-way	1.27	17.04	1.13	1.19	2.00	1.50

The second row of Table 4 shows the speedup of our enhanced 2-way processor with a MOMR functional unit over a single-issue machine. In all cases, our new architecture achieves greater speedup over single-issue execution than the standard 2-way superscalar processor (first row). The third row illustrates the additional speedup provided by our 2-way MOMR architecture over standard 2-way superscalar processors. For DES, the performance gain is very pronounced for key generation (17X 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 introduction of new permutation instructions (CROSS or OMFLIP) in columns c and d already yields huge speedup over the table lookup method (in columns a and b). The number of instructions for a 64-bit permutation is reduced from over 20 to at most 6, and most of the memory accesses are also eliminated, resulting in much fewer cache misses. Even then, our MOMR execution achieves an additional speedup of 13% to 19% by further reducing the cycles needed for a 64-bit permutation from 6 to 2 cycles.

For the integer DH, there is significant additional speedup of 2X over the standard 2-way superscalar processors. This is because our MOMR architecture allows the inclusion of wider functional units such as 128-bit multipliers. This reduces the overall number of instructions and cycles needed to complete a 1024 by 1024-bit multiplication, which is a primitive operation in the exponentiation function needed by public-key cryptography algorithms.

7. Conclusions

This paper makes several new contributions. First, we identify two categories of bit and multi-word operations as new challenges for word-oriented processor architecture for high-performance cryptographic processing. This insight is more useful from a broad architectural perspective than just picking out special-purpose operations to accelerate.

Second, we present two architectural solutions for achieving arbitrary 64-bit permutations in O(1) cycles. This is a significant result since previously arbitrary n-bit bit permutations took O(n) cycles. Even with our recent proposals of permutation instructions [5][9][10][11][12], this took at least O(log(n)) cycles. We show how a different 64-bit dynamically–specified permutation can be achieved *every cycle* by a 4-way superscalar processor with datarich MOMR execution. Our software solution for achieving permutations is much more powerful than a hardware solution – the latter can only achieve a few statically-defined permutations, while our solution can achieve all possible dynamically-defined permutations. Furthermore, the incremental cost is minimal, since we leverage common microarchitecture trends like superscalar processors. Our result is also significant because it implies that word-oriented processors have no problem supplying very high performance (1 or 2 cycles) for even extremely challenging bit-oriented processing like arbitrary bit permutations. Cryptographers can use bit permutations freely in their new algorithms if microprocessor architectures include these bit permutation instructions.

Third, we define the concepts of datarich MOMR (Multi Operands Multi Result) execution and instruction groups. MOMR functional units can achieve extremely high performance for bit-level permutations as well as multi-word operations, with a single coherent architectural solution. The MOMR feature 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 on a 2-way superscalar machine with minimal changes. Our proposal to base MOMR implementations on the ($2k$,k) datapath of a k-way superscalar processor gives us the flexibility of supporting all MOMR functional unit sizes covered by these existing datapath resources. We have also shown the control path modifications needed to support MOMR; these are minimal when compared to the complex pipeline control in typical superscalar, out-of-order machines.

Finally, a fourth contribution is the validation of the word as the atomic unit upon which a processor is optimized, since we show how both bit and multi-word operations can be achieved with MOMR execution for either superior performance or enhanced cost-performance.

8. References

[1] B. Schneier, *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] J. Burke, J. McDonald and T. Austin, "Architectural support for fast symmetric-key cryptography", *Proceedings of ASPLOS 2000*, pp. 178-189. November 2000.

[4] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: a fast flexible architecture for secure communication", *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 110-119, June 2001.

[5] R. B. Lee, Z. Shi, and X. Yang, "Efficient permutation instructions for fast software cryptography", *IEEE Micro,* vol. 21, no. 6, pp. 56-69, December 2001.

[6] R. B. Lee, "Subword parallelism with MAX-2", *IEEE Micro*, Vol. 16, No. 4, pp. 51-59, August 1996.

[7] K. Diefendorff et al, "AltiVec extension to PowerPC accelerates media processing", *IEEE Micro*, Vol. 20, No. 2, pp. 85-95, March/April 2000.

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

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

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

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

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

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

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

[15] J. A. Farell and T. C. Fischer, "Issue logic for a 600-mhz out-of-order execution microprocessor", *IEEE Journal of Solid-State Circuits*, Vol. 33, Issue 5, pp. 707-712, May 1998.

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

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

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

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

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