Instructions with Indirect Register Indexing for Multiple and Exponential Operations to Improve Performance

Nagi Mekhiel
Department of Electrical, Computer and Biomedical Engineering
Ryerson University
Toronto, ON, CANADA M5B 2K3
Email: nmekhiel@ee.ryerson.ca

***Abstract*—The current multi-core systems cannot be made scalable to large number of processors due to limited memory bandwidth needed to deliver instructions and data to many processors at the required rate.**

We propose a method that is capable of enhancing the power of a single instruction to perform multiple or exponential number of operations on data in registers that maps to any location in memory. The enhanced features of instruction are hidden from pipeline until the execution stage, creating a virtual effect of executing single instruction to simplify the microarchitecture and reduces the demands for fetching multiple instructions or accessing data elements. The method uses multiple function units that exist in any conventional processor, to perform these operations referenced by indirect register indexing in single cycle at the same time.

I. INTRODUCTION

The steady improvement in the speed of CMOS circuits that has been achieved by shrinkage, allowed processor designers to implement and use advanced architectures [3,7]. These new architectures depend on delivering maximum numbers of instructions per cycle, through some form of speculative , VLIW, multithread, superscalar or multi-core, which made the performance more dependent on memory bandwidth and clock speed. Memory systems, even with large caches cannot deliver the needed bandwidth for future technology [6].

Unfortunately, the improvement in processor performance is reaching a diminishing return. Designing architectures that fully utilize advancements in technology to meet the growing demands for more computing power has become very challenging [4,6].

Future architectures must be simple to run fast and benefit from advancement in technology. They also need to be compatible with the current applications. These architectures have to deliver more computing power without increasing memory bandwidth requirements.

The main objectives in our method is to design special instructions with enhanced computing power capabilities. These instructions could be used to replace multiple number of instructions. The performance improvement therefore, comes from decreasing the number of instructions per task and the ability to use multiple and exponential number of data in registers which reduces dependency on memory bandwidth and power consumption without complicating the design.

This method could be added to any conventional processor without increasing its complexity. It hides the enhancement from pipeline stages to create virtual effect of running a single instruction in the pipeline.

Section 2 gives summary of related work. In Section 3, we explain the motivations and concept. Section 4 presents the new instructions. In Section 5, we describe the Data Path for proposed method. Section 6 gives analysis of the performance potential. In Section 7, we give conclusions and future work.

II. RELATED WORK

Processor designers use latest available technology to enhance the computing power of processors. Some changes in the existing microarchitectures were needed to improve the performance of popular applications as in multimedia applications that use large sets of data and benefit from vector operations [5,8].

Intel added SIMD type instructions to the instruction set [9] to improve the performance of multimedia applications. These changes required the addition of SSE and MMX extensions to instruction set, and some modifications in the microarchitecture as:-

- It uses eight 64/128 bit special registers and adds special instructions to pack or unpack data in these registers.
  These special registers might not be enough for applications with many data streams.
- It adds MMX special instructions to ISA, thus complicating the design making it harder to scale with future technology.
- Executing MMX instructions requires extension to pipeline stages.
  This increases complexity and adds more pipeline stalls.
- All operands must fit in one register, thus limiting the performance gain of having SIMD applied to operands in many registers.

Vector processor use large vector registers that fit large array of data. If the array is smaller than the vector register, then portion of the vector register is not utilized [3]. The vector operations use highly pipelined function units, thus depends on having faster clock which increases power consumption and is limited by the wire delay.Each vector operation takes N number of clock cycles where N is the number of elements in the vector registers. Our method executes enhanced instructions on multiple elements mapped to registers using available multiple function units in parallel in one clock cycle.

Accessing registers indirectly to improve performance has been proposed in [1] by implementing pointers in instructions. This method uses separate dereferenceable register file DRF that is referenced by pointers from single instruction. The separate DRF adds overhead to complexity as it requires separate register pointer file RP that must have these instructions with pointers to allow the indirection to access DRF.

Accessing DRF through RP file increases the latency of instructions and could affect the clock cycle time. When accessing data that maps to both RF and DRF but might have dependency, the forwarding of data to eliminate hazards will be complicated and cannot be done in the same cycle because accessing time of data in DRF takes longer time.

The size of DRF must be small to run fast and each register in DRF requires a pointer in PR file, which increases power consumption and size of silicon and complicates the design of control unit due to executing operations on registers referenced by pointers differently than operations to registers in RF.

Our proposed method enhances the power of instruction without the need to use special register units as in [1], or separate pipeline or add stages to the pipeline as in [9]. The enhanced power of instructions ,in our method, are invisible to control and pipeline until the execution stage. This creates the effect of having a simple microarchitecture to execute the enhanced instructions the same as the other instructions.

III. Motivations and Concept

A. Motivations

Processor performance is measured in execution time. Execution time depends on the number of instructions, the number of clock cycles per instruction, and the clock cycle time. Parallel instructions ILP has been used in superscalar to improve performance and increase IPC. Increasing IPC comes at a cost of increasing the hardware complexity. As complexity of the system increases, the scalability decreases.

Multi core is now used to improve performance by executing multiple parallel instructions at the same clock cycle. This requires complicated multi-level memory hierarchy to deliver instructions to multiple cores. Memory cannot supply these processors with instructions at the required rate.

What is needed to increase system performance, is an enhancement method that is visible to the software so that it can use it, but invisible (virtual) to the microarchitecture so that it will not complicate it. Our new method has a direct impact on performance by reducing the number of required instructions. This is achieved by special instructions with enhanced power.

B. The Concept

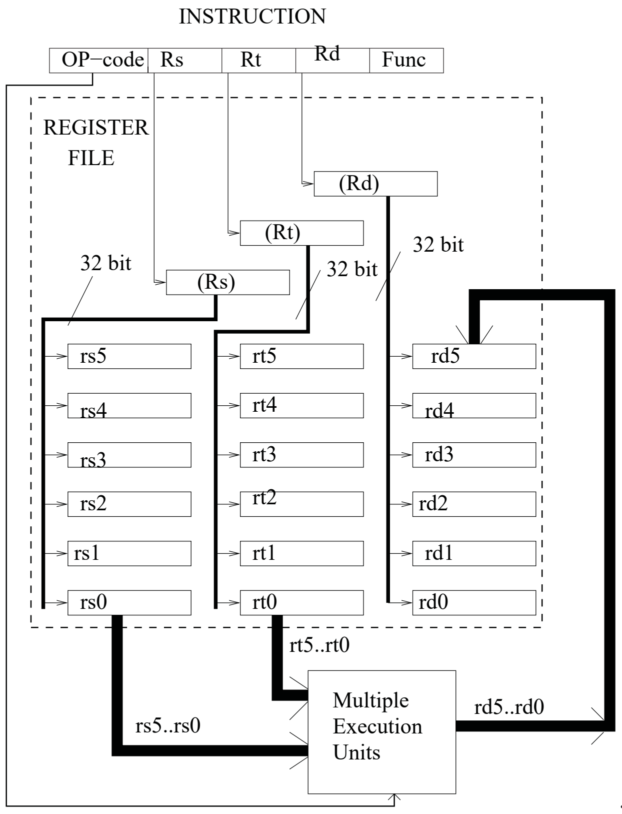
The method has the following features:

- Capable of producing multiple operations from a single instruction in the execution stage. It uses indexes in instructions to perform operations on multiple registers. This reduces the number of instructions required, improves performance, and reduces memory bandwidth requirements. Memory bandwidth cannot satisfy future processors requirement [5].
- An instruction could execute a single operation on single data elements as in conventional processors ,to maintain compatibility with the existing applications, or it executes multiple operations on multiple or exponential number of data elements to enhance performance.
- It uses the microarchitecture of any conventional, pipelined, processor. We assume MIPS processor architecture [2,3,7] to explain this concept.
- The control unit is simple and supports the multiple and exponential operations for the enhanced instructions by duplicating the control signals (it does not need extra states for the state machine of control unit).
- The new enhanced instructions have indexes or indirect indexes to access multiple or exponential number of operands from a single instruction. Each field in the enhanced instruction sources and destination register is used to index multiple registers. Dividing the referenced (source/destination) registers into a number of fields ,in which each field indexes a source or a destination register, provides multiple operands from the single instruction.
- Multiple execution units are available for the multiple operations. The operations in the execution units are performed in parallel on the operands that are referenced by the register indexes from the single enhanced instruction.
- The operation (defined by the opcode of the single instruction) is applied to all execution units in SIMD style. The results are written back to registers that are referenced by the single instruction with indexes to destination register.
- The pipeline starts with fetching a single instruction that goes through the pipeline as any simple instruction, until it reaches the execution phase, the control unit will then execute it as an instruction with enhanced power for multiple operations. The execution of the enhanced power instruction is thus hidden from the microarchitecture.

Figure (1), shows the concept block diagram of the new enhanced instruction. This single instruction is capable of referencing multiple operands.

Contents of Rs and Rt are divided into 6 fields (5 bits each for a 32 registers) to reference multiple operands; in registers rs5 to rs0 and rt5 to rt0.

Multiple execution units (6 units) use the operands rs5 to rs0 and rt5 to rt0 and perform a SIMD operation between them. The execution units write the results of the multiple operations to the destination registers rd5 to rd0. All of these operations are performed in parallel as if one instruction is



.8

Fig. 1. New Instruction with Indirect Register Indexing

being executed. The destination registers are selected by the indexes in Rd register of the single instruction.

IV. The INSTRUCTION SET ARCHITECTURE for The Enhanced Instructions

A. *Instruction Set Format for Single Operations*

In this study, we used MIPS instruction set as an example [2,3,7].

The three different types of instructions are:

- Instructions for Arithmetic and Logic operations:
  add $3,$2,$1;
  Meaning register1 is added to register2 and results is stored in register3 so ($3)= ($1) + ($2)
- Instructions for Data transfer:
  lw $1, 100($2);
  Meaning ($1)=Memory[100+($2)] : load content of Memory at address (100 + content of $2) and transfer it to register $1.
- Control flow instructions:
  beq $1, $2, 28;
  Meaning if content of register1 is the same as register2 then branch to next program counter plus offset of 28. if(($1)==($2)) PC=PC+4+28 else PC=PC+4.

Figure (2) shows the Instruction Format for the single operation mode.

op: operation, rs: first register source operand, rt: second register source operand, rd: register destination operand,

Arithmetic and Logic Instructions Format

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Data Transfer and control Flow Instructions Format:

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

.8

Fig. 2. Instruction Format for The Single Operation Mode

shamt: shift amount, funct: variant of operation in op field, address: address field [2].

B. *The Instruction Format for Multiple Operations Mode*

The enhanced Instructions for multiple operations are similar to single instruction except we added m to opcode as:

- The instructions for multiple Arithmetic and Logic operations are:
  addm ($3),($2),($1); add multiple.
  Executing the above will result in 6 simultaneous ADD multiple operations as:
  (($3i))= (($1i)) + (($2i)) ; content of a register that is indexed by content of $3 i field equal the sum of the content of a register that is indexed by the content of $1 i field plus the content of a register that is indexed by the content of $2 i field. The value of i changes from 0 to 5 for 6 multiple operations in parallel.
- Instructions for Data transfer:
  lwm ($1), 100(($2)) ; load multiple.
  Executing the above instruction will result in 6 simultaneous load operations as:

  (($1i))=Memory[100+(($2i))] content of the register that is indexed by the content of destination $1 i field equal the content of Memory at address (100 + content of the register that is indexed by the content of source $2 i field).
- Control flow instructions:
  beq $1, $2, 28 ; branch if equal.
  Meaning if(($1)==($2)) PC=PC+4+28 else PC=PC+4.
  It uses the same single instruction as MIPS control flow instructions.

Figure (3) shows the Instruction Format for the Enhanced multiple operations Mode.

op: operation

(rs): index to first source register that has indexes to source operands

(rt):index to second source register that has indexes to second source operands

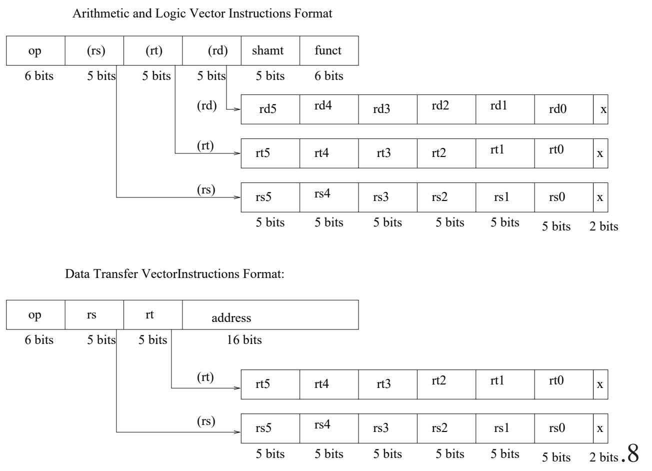


Fig. 3. Enhanced Instruction Format for multiple operations Mode

(rd): index to register destination that has indexes to destination operands.

shamt; shift amount, funct: variant of operation in op field, address: address field.
rs5..rs0; indexes to first register source operands
rt5..rt0; indexes to second register source operands
rd5..rd0; indexes to register destination operands.

C. *The Enhanced Instruction set for Exponential Power Operation Mode*

The enhanced Instructions for exponential power operations uses same opcode for single instruction with added ex as:

- Instructions for exponential Arithmetic and Logic operations:
  addex (($3)),(($2)),(($1)); Add with exponential power
  The operands of the first source are obtained from the indirect indexing of register $1 by dividing it into 6 fields, each field is then used to index a register that is also divided to 6 fields resulting in 36 operands (5 bits each) using 6 registers.
  Four indexes equal zero to make total number of referenced registers equal the maximum number of 32 available registers. The 36 fields are used to access 32 registers (assuming processor has 32 registers) for the first source operands list.
  Similarly the content of register $2 is used to index 32 registers for the second source operands.
  The content of register $3 is used to collect the results of applying the operations to the 32 operand pairs in 32 destination registers, indexed by the indirect referencing of register $3.
- Instructions for Data transfer:
  lwex (($1)), 100((($2))); Load with exponential power.
  First source operands are obtained from the content of register $2 which is divided to 6 fields, each indexes a register, that is also divided to 6 fields resulting in 36 indexes (each of 5 bits fields in the 6 registers). Four indexes equal zero to make the total number of referenced registers equal the 32 maximum number of available registers. The 36 fields are then used to index a 32 first source operands.
  The offset 100 is added to the content of each source operand in the 32 registers, then used to access the memory and load its content to the 32 destination registers referenced by the indirect indexes in register $1.

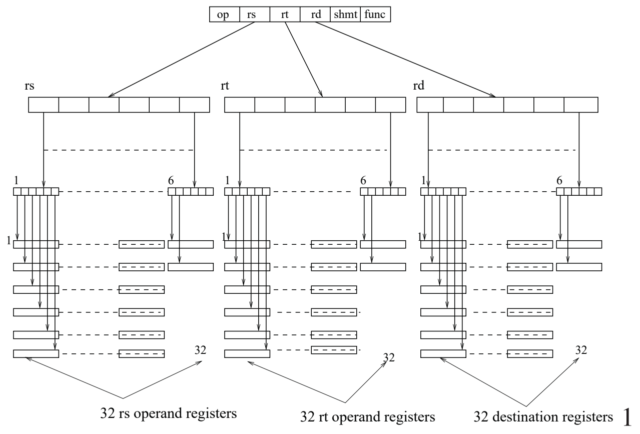


Fig. 4. Enhanced Instruction Format for exponential power operation Mode

Figure (4) shows the Instruction Format for the exponential power operations.
The number of registers in the register file must increase to support the exponential operations (it needs 21 registers to hold the indexes for the 96 operand registers). It is important to note that the use of these registers is different from the elements stored in vector register that must be accessed in specific order in the same vector register or must be access RP file first to get data from DRF in [1].

V. DATA PATH FOR PROPOSED METHOD

Figure (5) shows a block diagram for the Data Path of proposed method. The Data Path consists of the following stages:

- IF: Instruction Fetch from the Instruction cache.
- ID: Instruction Decode and operand fetching.
- EX: execute the effective address, arithmetic and logic operations using multiple execution units.
- MEM: memory access (load/store) to data cache.
- WB: Write back to register file.

This is based on MIPS and has the following additions:
1-**The Register File**: has expanded input and output ports to support multiple operands transfer between RF and memory. Each port is used to access a register that is referenced by one of the indexes from the indexed register in the instruction. Adding multiple ports to the register file will increase the cost of the register file implementation but not the latency as these registers are accessed in parallel.

The number of registers needed to support multiple or expenential operations must include extra registers (overhead) used as indexes to the register operands.

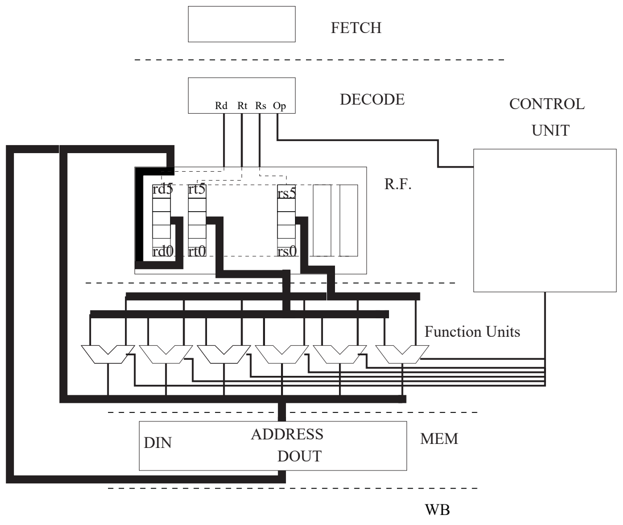


Fig. 5. Datapath Block Diagram for Proposed Method

This method could easily supports variable number of operands, if we assign zeros to some register indexes to point to R0.

2-**Expanded multiplexers**: The input multiplexers to the register file need to be expanded to select the output of multiple execution units, or select multiple data streams from the data cache, or from data forwarding.

3-**Execution Units**: uses multiple execution units to support multiple operations. The inputs to the execution units come from the register file of multiple operands (rt5..rt0 and rs5..rs0). Each execution unit perform the same operation and is connected to the same control signals of single instruction by duplicating them for all the execution units.

4-**Data Memory**: It uses a data cache with large block size to match the number of operands in the instruction (6 for multiple operations and 32 for exponential operations).

VI. ANALYSIS OF THE PERFORMANCE POTENTIAL FOR PROPOSED METHOD

We give the following example for using the enhanced instruction set as compared to the conventional processor.

The application is to add the elements of array A[1000] and B[1000], then store the results to array C[1000]. Each element is assumed to be 32 bit. Data elements and instructions are all assumed to be in Cache. Processor cycle time and pipeline length is assumed to be same as in the conventional processor.

We assume that there are enough registers and execution units enough to support multiple or exponential number of operations without slowing the processor.

A. The Performance of Conventional Processor

If we assume that A[0] is at location 10000, B[0] is at location 20000 and C is at location 30000, register $15=1000 and $6=0.

Using MIPS, as an example, the code is:

```
LOOP: lw $8, 10000($6);     #$8=Mem[10000+i]=A[i]
      lw $9, 20000($6);     #$9=Mem(20000+i]=B[i]
      add $10, $8,$9;       #$10=$8+$9 or A[i]+B[i]
      sw $10, 30000($6);    #Mem[30000+i]=C[i]=A[i]+B[i]
      addi $6,$6,4;         #i=i+1
      bne $6, $15, LOOP
```

This program takes 1000*6=6000 cycles to complete and executes 6000 instructions.

B. Using the Enhanced Instructions with Multiple Operations

We assume the following:

Assume that register $7= % 00101 00100 00011 00010 00001 00000 00, which contains indexes to registers $5,$4,$3,$2,$1,$0.
Assume that $5=5, $4=4, $3=3, $2=2, $1=1 and $0=0.
Assume that $15=1000.
Assume that register $8 =% 10101 10100 10011 10010 10001 10000 00, which contains indexes to $21, $20, $19, $18, $17 and $16.
Assume that register $9 =% 11101 11100 11011 11010 11001 11000 00, which contains indexes to registers $29, $28, $27, $26, $25 and $24.
Assume register $10 =% 01101 01100 01011 10110 10111 11110 00, which contains indexes to registers $13, $12, $11, $22, $23 and $30.

The code for the enhanced instructions for multiple operations is:

```
LOOP: lwm ($8), 10000(($7));      #$21=Mem[10000+5],..
$16=Mem[100000+0].
       lwm ($9), 20000(($7));     #$29=Mem[20000+5],..
$24=Mem[200000+0].
        addm ($10), ($8), ($9);     #$13= $21+$29,...
$30=$16+$24.
       swm ($10), 30000(($7));   #Mem[30000+5]=$13,...
Mem[30000+0]=$30.
      addmi ($7), ($7), 24;   #i=i+6
      bne $21, $15, LOOP
```

This program takes 167*6=1000 cycles to complete and executes 1002 instructions. The performance gain compared to conventional processor is 600%.

C. Using the Enhanced Instructions with Exponential Operations

To run the same application, using the exponential operations, we assume the following code:

Assume that register $7 contains indexes to 6 registers, that are indexing 32 registers.
Assuming that register $15=1000.
Assume that register $8 contains indexes to 6 registers that are indexing 32 source 1 operand registers.
Assume that register $9 contains indexes to 6 registers that are indexing 32 source 2 operand registers.
Assume register $10 contains indexes to 6 registers that are

indexing 32 destination registers.
The number of registers needed to support this mode can exceed the number of available registers in MIPS. To maintain the architecture simple and the same as MIPS, we assume some registers are used for both the sources and destinations. There are other registers needed to hold the indexes. These index registers could be implemented with special registers in the register files, that are used only when the enhanced power instructions are executed.
The following is the code for the Enhanced Instructions:

```
LOOP: lwex (($8)), 10000((($7)))    ; load first 32 source operands.
      lwex (($9)), 20000((($7)))    ; load second 32 source operands.
      addex (($10)), (($8)), (($9))  ; perform 32 add operations in parallel.
      swex (($10)), 30000((($7)))   ; store results of 32 destination registers.
      addmi ($7), ($7), 128    ; #i=i+32
      bne $21, $15, LOOP
```

This program takes 32*6=192 cycles to complete and only 192 instructions to execute. The performance improvements = 3200%

D. Performance Potential

In the above examples; a conventional processor completes the code in 6000 cycles and executes 6000 instructions.

Enhanced instruction method takes 1000 cycles ,using the multiple operation mode, and executes 1000 instructions. This reduces the execution time by 6 times and offers 6 times reduction in memory bandwidth requirements compared to the conventional processor.

Enhance instruction method takes 192 cycles with the exponential operation mode, and executes 192 instructions. This gives a reduction in execution time by 32 times and offers 32 times reduction in memory bandwidth requirements compared to the conventional processor.

This improvements come with the addition of more execution units, expansion of multiplexers, and increasing the number of registers in register file, but with limited addition to the complexity of executing these enhanced instructions as they are kept invisible to the microarchitecture until the execution stage. This virtual effect keeps the complexity of the control circuit and most of the pipeline stages the same as in conventional processor.

VII. CONCLUSIONS

Our method enhances the computing power of simple instructions without adding complexity to the microarchitecture.

It offers performance gain using simple instructions with enhanced power to support future growing demands of computing power. It also offers compatibility with the existing applications.

It uses indexes in registers that are visible to programmer but not to most of the microarchitecture. Scattered data elements in memory could be grouped with these indexes and transferred to registers then operate on them in parallel at same time to reduce memory dependency.

Future work will include the implementation of this method in FPGA using a simple processor then evaluate the performance of the system and compare it to a conventional system implemented with same processor using same technology.

REFERENCES

[1] JongSoo Park, Sung-Boem Park, James Balfour, David Black-Schaffer, Christos Kozyrakis, William Dally Register Pointer Architecture for Efficient Embedded Processors. 2007 Design, Automation and Test in Europe Conference and Exposition, April 16-20, 2007, Nice, France.

[2] J. Hennessey and D. Patterson, Computer Organization and Design: The HardwareSoftware Interface," Morgan Kaufmann, San Mateo, Calif. 1994.

[3] J. Hennessey and D. Patterson, Computer Architecture : A Quantitative Approach," 2nd ed. Morgan Kaufmann, San Mateo, Calif. 2003.

[4] D. Matzke, Will Physical Scalability Sabotage Performance Gains?," IEEE Computer, September 1997, pp. 37-39.

[5] K. Diefendorff and P. Durbey, How Multimedia Workloads Will Change Processor Design," IEEE Computer, September 1997, pp. 43-45.

[6] D. Burger, J. R. Goodman, and A. Kagi, Memory Bandwidth Limitations of Future Microprocessors," Proc. 23rd Ann. Int'l Symp. on Computer Architecture, ACM Press, New York, 1996, pp. 78-89.

[7] K. Yeager, The MIPS R10000 Superscalar Microprocessor," IEEE Micro, April 1996, pp. 28-40.

[8] P. M. Johnson, An Introduction to Vector Processing," Computer Design, Feb. 1978, pp. 89-97.

[9] S. Raman, V. Pentkovski, and J. Keshava, Intel Corporation, Implementing Streaming SIMD Extension On The Pentium III Processor," IEEE Micro, July/Aug 2000, pp. 47-57.