MISC: A Multiple Instruction Stream Computer

Gary Tyson
Computer Science Department
University of California, Davis
(tyson@cs.ucdavis.edu)

Matthew Farrens
Computer Science Department
University of California, Davis
(farrens@cs.ucdavis.edu)

Andrew R. Pleszkun
Dept of Electrical and Computer Engineering
University of Colorado-Boulder
(arp@tosca.colorado.edu)

Abstract

This paper describes a single chip Multiple Instruction Stream Computer (MISC) capable of extracting instruction level parallelism from a broad spectrum of programs. The MISC architecture uses multiple asynchronous processing elements to separate a program into streams that can be executed in parallel, and integrates a conflict-free message passing system into the lowest level of the processor design to facilitate low latency intra-MISC communication. This approach allows for increased machine parallelism with minimal code expansion, and provides an alternative approach to single instruction stream multi-issue machines such as SuperScalar and VLIW.

1. The MISC Design

The MISC processor, a direct descendant of the PIPE project [CGKP87, GHLP85] will exploit both the instruction and data parallelism available in a task by combining the capabilities of traditional data parallel architectures with those found in machines designed to exploit instruction level parallelism. Unlike the two processor PIPE design, the MISC system is capable of balancing the processor load of instructions performing *memory access* and *execute* operations among four processors. The characteristics of the MISC design also allow the introduction of a number of new and unique instructions, like the Sentinel and Vector instructions described later in this paper. As its name indicates, MISC is composed of multiple Processing Elements (PEs) which cooperate in the execution of a task. MISC is technically a message passing system where data can be sent between processors as easily as it can be transferred between processor registers.

The MISC system consists of four Processing Elements (PEs), a dual bank-selected data cache (DCache) and a set of internal data paths used to transmit data among PEs and the DCache. The component design of MISC is illustrated in Figure 1. Each PE executes asynchronously from other PEs and the DCache. The internal datapaths are used to facilitate communication between elements (PEs and/or DCache). Each data path is controlled by a single element; for instance, the internal data path labeled PBUS1 is controlled (written) solely by PE1. Each PE has its own such bus (PBUS{1-4}), and the data cache controls two busses (CBUS1 and CBUS2).

In addition to 32 *data* lines, each internal bus also consists of *routing* lines and *busy* lines. The *routing* lines are used to identify the destination of a given message; a *routing* line exists for each possible destination (PE{1-4} and DCache). Collectively, the *routing* lines specify which subset of destinations should receive a message. In order to broadcast a message to all other processors, a Processing Element would therefore assert the *routing* line for each destination, enabling up to 5 data transfers in a single PBUS or CBUS operation. The *busy* lines are asserted by each potential destination to indicate it's inability to accept more data at the present time.

When a processor wants to send a message, it determines the desired *route* and compares this with the *busy* lines associated with it's own bus. If all elements of the destination route are capable of receiving the data, then the data and routing information is placed onto the bus. Transmission of data does not require the communicating PEs to synchronize at the transfer of the message; instead, data buffers (queues) are provided to store messages that have been sent but not yet processed by the destination.

Two separate bank selected I/O channels support the transmission of data between the MISC chip and the rest of the system (main memory). These channels are controlled by the DCache.

1.1. Processing Element Structure

The processing elements are collectively responsible for the execution of a single task. Unlike processor designs which allow for multiple instructions to be issued

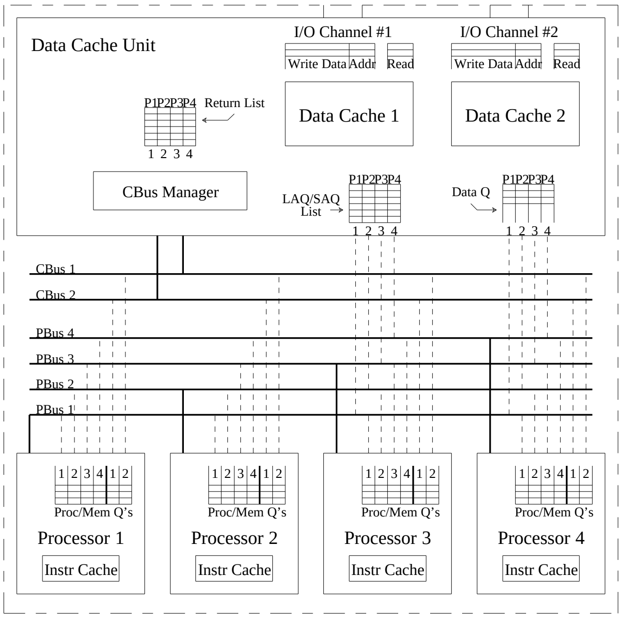

Figure 1: MISC Component Design

from the same instruction stream, MISC PEs each have their own independent instruction streams. Each PE maintains all state information required to function as an independent processor. In fact, the MISC hardware is capable of running 4 completely unrelated tasks in parallel. However, it is assumed that a single task will be partitioned (by the compiler) into 4 instruction streams that cooperate in the execution of a single task. Currently all PE are assumed to be identical in order to allow the compiler maximum flexibility in assigning operations. We intend to relax this restriction after an analysis has been completed of the instruction mix of each PE.

Figure 2 shows a single MISC Processing Element (PE). Each PE contains a basic four-step execution pipeline, including single cycle Instruction Fetch, Instruction Decode, and Instruction Issue phases and a multi-cycle Execution phase. Each of the processing elements (PEs) consists of 32 General Purpose Registers (GPR's) which are available for data storage that persists over multiple references, a FIFO queue (PEQ) for each PE in the system (including itself) to store data transfers between PEs, and 2 memory queues (MQ's) which contain data requested from memory.

The size of each queue (PEQ and MQ) is not given here, however the size of each is an architecturally visible component; the compiler must know the size of each queue (they need not all be the same size) in order to schedule code correctly and avoid deadlocks due to resource depletion.

In addition to the 32 general purpose registers, two special purpose registers are visible to the program. The *Program Counter* (PC - used to schedule the instruction flow) and the *Vector REGister* (VREG - used to store the iteration value for vector and sentinel instructions).

Finally, each PE has its own instruction cache. As one would expect, the use of instruction caches is much more important in MISC than conventional processor designs do to the contention involved in servicing 4 instruction fetches per cycle.

All instructions in MISC are 32 bits in length and have 3 6-bit source operand fields and a 6-bit destination

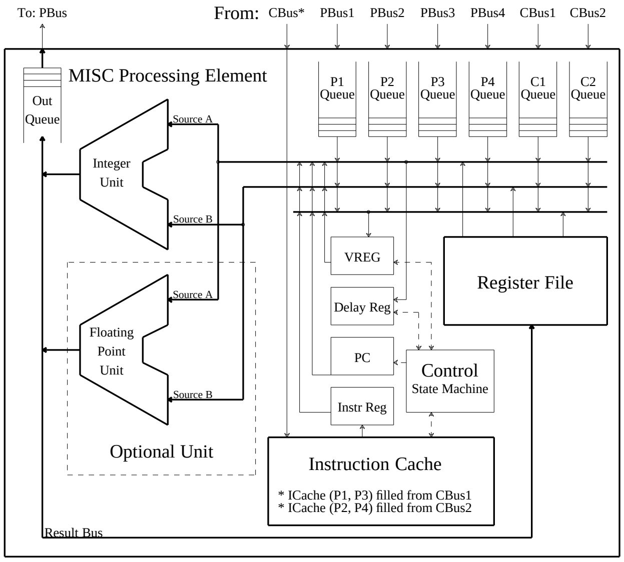

Figure 2: MISC Processing Element

operand specifier. In addition, many instructions allow for two of the source fields to be replaced by a 12 bit constant. Each source field can address any of the 32 GPR's, a PEQ, a MQ, the PC, the VREG, or a small signed constant (-16 to 15). The destination specifier may address a GPR or *routing* information for a data transfer onto the PE's PBUS.

At the instruction issue stage, if a queue is specified as a source input and that queue is currently empty, that instruction is delayed until all required input operands are available.

There are three different types of MISC instructions: Scalars, vector operations and a sentinel operation (which uses a sentinel value to terminate iterations of the instruction).

Scalar ALU/FPU operations and scalar control flow are handled in a fairly conventional manner. For ALU and FPU instructions, the third source field is ignored and the instructions are treated as if they were standard 3-operand instructions. In the case of control flow operations, the *dest* field is used as a constant to determine the number of *delayed branch* slots [FaPl91] be filled. The address of the branch is calculated as the sum of the *src1* and *src2* operands, and the *src3* operand specifies the register to be tested.

Since MISC uses a decoupled approach to memory operations, the load and store request instructions are different than those in a conventional machine. Invoking a memory read operation involves providing the DCache with the memory address to be read, the set of destination PEs which are to receive the data, and which CBUS (and therefore MQ) will receive the data. In a Load Address Queue (LAQ) instruction, the *dest* field will contain the set of PEs that are to receive the data. The *dest* field should not specify a register. The address to be used in the request is the sum of the *src1* and *src2* operands. There are both LAQ and LAQ2 instructions, in order to specify which MQ of the destination PEs should receive the data. The store request (SAQ) instruction operates in a similar manner, except that the *dest* operand specifies a single PE from which the DCache should receive data to be written to memory.

2. Data Cache Structure

The Data Cache unit consists of one or more single access data caches of unspecified size, a Processor Bus (PBUS) request and data interface, a cache manager, a memory interface, and a Cache Bus (CBUS) manager. The design goals for this component of MISC differ somewhat from those for on-chip data caches of other processors. While on-chip caches are generally used to reduce memory access latency, this aspect is less vital in a decoupled machine. However, with four processors on the same die, it is imperative to limit as much as possible the off chip memory activity to avoid overtaxing the I/O capabilities of the IC. The instruction caches located on each processor provide a first-order reduction in I/O traffic; it is the responsibility of the data cache to further reduce off chip traffic to a manageable level.

On each cycle the Data Cache Unit must examine all PBUS's for incoming requests or incoming data from the processing elements. The PBUS request and data interface determines whether the request is a LAQ/SAQ request or a data item and inserts the item onto the appropriate queue (*LAQ/SAQ List* or *Data Q*, in this case). The Cache Manager then examines the *LAQ/SAQ List* to schedule any pending memory requests for data cache lookup. In the event of a cache miss, an external memory reference must occur in order to load the missing cache line.

The memory interface is responsible for dealing with all memory transactions necessary to service a cache miss. Each cache bank has a dedicated memory interface unit which is responsible for maximizing I/O channel utilization. Once read data is available, the CBUS interface is responsible for selecting among the available data in the *return list* and managing activity on the two CBUSs used to send data from the cache unit to the PEs. The CBUS manager determines the availability of each PE to receive data from the cache unit (by examining the busy bit of each CBUS) and transfers the first data item which is ready to be received.

3. Vector and Sentinel Processing

Basic blocks will be very small in MISC. Therefore, a class of vector and sentinel instructions have been specified in order to help reduce the number of branches needed to perform certain tasks.

The vector instructions require one additional source operand specifier (*src3*) which is used as a vector count. When a vector instruction arrives at the issue stage of the instruction pipeline, the vector register (*VREG*) is cleared and a vector count register (*VCOUNT*) is loaded from *src3*. The scalar version of the vector instruction is then executed and the *VREG* is incremented until the contents of *VREG* are equal to *VCOUNT*. Once the *VREG* equals the *VCOUNT* normal instruction pipeline function continues.

In a sentinel instruction, the *src3* field specifies a register whose contents are compared to the sentinel value (assumed to be zero in the initial design). If the contents of the register do not match the sentinel, the scalar version of the instruction is allowed to issue. This pattern of (compare, issue) is repeated until the comparison produces a match. This is a prime example of an instruction that only makes sense in a queue-based multiprocessor environment. By specifying an input queue in the *src3* field, and having another processor feeding that queue, we have a case of an instruction with the fixed semantic "*while (src3) dest = src1 op src2*" producing variable results.

One ramification of having several processors cooperating on the execution of a task is that many more branch instructions are required in order to keep the instruction streams synchronized. This duplication of branch instructions in each stream can lead to a significant increase in the total number of instruction required to perform a task. The *vector loop* (VLOOP) instruction uses the vector register in conjunction with a *Delay Register* (DREG) to achieve a very simple branch hiding (or zero cycle branch) instruction. It is used in cases where the number of times a basic block will execute is known at the initial entry into the loop (either as a constant or register variable). When a VLOOP instruction is issued the DCOUNT register is loaded with the delay count field of the instruction (this is the same as conventional branch instructions). Then, the DREG and VREG registers are cleared, the VCOUNT register is loaded from *src3* of the instruction, and the jump address (JUMPPC reg) is set to the first delayed instruction (i.e. PC + 1)

At this point instruction flow continues as in a regular branch instruction; DCOUNT instructions are executed from the delay slots, and once all delayed instructions have been issued *VREG* is incremented and compared to *VCOUNT*. If *VCOUNT* iterations have not yet been completed (*VREG ≠ VCOUNT*) then the *JUMPPC* is moved into the *PC*, which initiates the next iteration of the loop. When the specified number of iterations have finally been executed (*VREG = VCOUNT*), execution continues with the next sequential instruction.

Conventional wisdom holds that instruction level implementations of higher level semantics seldom leads to performance improvements because of the complexity of implementation and the scarcity of application. We agree that this is true for a standard sequential processor. However, based on our experience with the design and implementation of the PIPE processor, we believe the vector and sentinel instructions defined in MISC can be implemented with minimal hardware modification to the issue logic.

Furthermore, the existence of multiple instruction streams leads to a greater potential for application of these instructions. Often the application of a complex construct (e.g. a *while* loop) is virtually impossible in a single instruction stream design because of the complexity of evaluating the test condition. The need to both evaluate a complex test condition and perform the control flow operation cannot be reduced to a single instruction. However, in a multiple instruction stream machine such as MISC, a single PE can evaluate the test condition and broadcast the boolean result to all PEs, increasing the number of simple boolean tests evaluated during the execution of these loop semantics and allowing for a significant increase in the application of these instructions.

4. Example Code

In order to analyze the performance potential of the MISC system, several small benchmarks were hand compiled and optimized. Two examples have been chosen to illustrate some of the abilities of the MISC design. Of the two programs presented here, one contains highly vectorizable code and the other displays no data parallelism.

The Lawrence Livermore Loops [McMa84] provide a convenient source for small vectorizable loops found in scientific code. Although originally extracted from FORTRAN code, these loops have been translated to C for purposes of this analysis. The code was then hand compiled and optimized using available vectorizing techniques [BeDa91]. The MISC object code generated consists of 4 instruction streams.

PE1 is first responsible for relating loop control to

C Source (*LLL #3* **)**	**MISC Assembly Code**
main() { float z[1000],x[1000],q; int k;	[1] add {P1-4} ← #0, #1000 [2] add R1 ← #0, _x [3] vlaq {P3} ← R1, VREG, PQ1
q = 0.0;	[4] add R1 ← #0, _z [5] vlaq2 {P3} ← R1,VREG,PQ1
for (k=0; k<1000 ;k++)	[6] vfmul {P4} ← MQ1,MQ2,PQ1
q = q + z[k] * x[k]; }	[7] fadd R1 ← #0, #0 [8] vfadd R1 ← R1, PQ3, PQ1

all other PEs. In this loop, the number of iterations is constant (1000), so PE1 prepares all PEs for vector operations by sending the loop count in line [1]. After initializing a register (R1) to contain the base address of vector X (line [2]), PE1 initiates a VLAQ instruction to fetch the vector (with stride 1) from memory with the destination being PE3. The loop count for this operation is received from PQ1 (containing the value 1000 sent in line [1]). Similarly, PE2 initiates a vector memory fetch of Z. Again the loop count is received from its PQ1. PE2 uses a LAQ2 instruction (line [5]) to allow the destination (PE3) to distinguish the elements from the vector X and Z; all elements from X will arrive in PE3's MQ1, while all elements from Z arrive in PE3's MQ2.

In line [6], PE3 performs the floating point multiplication of the vectors (X and Z) provided by the DCache as a result of the load instructions. Again, MQ1 represents elements from X, MQ2 contains elements of Z, and PQ1 contains the vector count specified by PE1 in line [1]. The result of the multiplication of each pair of elements is sent to PE4. This initiates a chain of data allowing PE4 to calculate the sum of multiplied vector elements in register R1 (line [8]). Total execution time is 1022 cycles with a total of 4004 instructions issued leading to a rate of 3.9178 instructions per clock cycle.

The second example program is taken from the C language library *strcpy()* function. This function displays little parallelism due to the dependency found in the loop control variable. This loop is not amenable to vectorizing optimizations nor to techniques to increase instruction parallelism such as loop unrolling. However, a certain amount of instruction parallelism exists in memory address calculation and data manipulation.

PE1 performs the load requests for *str2*. Line [1] simply initializes a register to allow for reading of str2. Lines [2] and [3] jointly perform the load request and transmit the loop control information to the relevant PEs. Store requests are generated by PE2 after initializing R1 (in line [4]) by executing the sentinel SSAQ instruction in line [5]. This generates all store request up to (but not including) the EOS mark. Line [6] uses the scalar version of SAQ to generate the store request for the EOS marker (Note the use of the final vector count (VREG)). Finally, the actual data movement is accomplished by PE3 in line [7]. Characters are transmitted from MQ1 back to the DCache ({M}) to be paired with the store requests generated by PE2. In line [8] the EOS character is written. Due to the data dependency found in line [3], the execution time of an iteration of the loop is dependent on the memory access time. Assuming a latency of 4, execution takes 66 cycles in this example. A total of 50 instructions are issued achieving an issue rate of 1.32 instructions per cycle.

C Source (*strcpy()* **)**	**MISC Assembly Code**

main() {
 	
char str1[15],
 	
*str2 = "example string";
 	
while (*str1++ = *str2++);
 	
}
	[1] add R1 ← #0, _str1+1
	[2] laq {P1-3} ← R1, #-1
	[3] slaq {P1-3} ← R1, VREG, MQ1
	
	[4] add R1 ← #0, _str2
	[5] ssaq P3 → R1, VREG, MQ1
	[6] saq P3 → R1, VREG
	
	[7] sadd {M} ← MQ1, #0, MQ1
	[8] add {M} ← #0, #0

5. Future Work

In this paper we have described an architecture designed to exploit both instruction-level and data level parallelism. An outgrowth of the PIPE processor, the MISC processor features multiple independent processing elements communicating with one another via architectural queues. A number of new instructions have been developed to take advantage of this partitioning of a processing task, such as the Sentinel and Vector instructions described in the paper.

Based on our experiences with the implementation of PIPE, we feel strongly that this approach is an excellent way to use the increasing transistor densities available in VLSI, because it does not require a synchronous distributed clock to be spread over the entire die. In addition, each processing element is relatively simple, reducing the design time necessary and allowing the highest possible clock rate.

We are currently working on developing a C compiler for the MISC machine, in order to fully evaluate the various new approaches used in this design. We are confident that a detailed analysis will demonstrate the potential effectiveness of a multiple instruction-stream approach.

References

[BeDa91] M. E. Benitez and J. W. Davidson, ''Code Generation for Streaming: an Access/Execute Mechanism'', *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA (April 8-11, 1991), pp. 132-141.

[CGKP87] G. L. Craig, J. R. Goodman, R. H. Katz, A. R. Pleszkun, K. Ramachandran, J. Sayah and J. E. Smith, ''PIPE: A High Performance VLSI Processor Implementation'', *Journal of VLSI and Computer Systems,* vol. 2 (1987).

[FaPl91] M. Farrens and A. Pleszkun, ''Overview of the PIPE Processor Implementation'', *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, Kapaa, Kauai (January 9-11, 1991), pp. 433-443.

[GHLP85] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter and H. C. Young, ''PIPE: a VLSI Decoupled Architecture'', *Proceedings of the Twelveth Annual International Symposium on Computer Architecture*(June 1985), pp. 20-27.

[McMa84] F. H. McMahon, *LLNL FORTRAN KERNELS: MFLOPS*, Lawrence Livermore Laboratories, Livermore, California, (March 1984).