# MISC: A Multiple Instruction Stream Computer

**Gary Tyson**
**Matthew Farrens**

**Computer Science Department**
**University of California, Davis**
**Davis, CA 95616**
**(tyson@cs.ucdavis.edu)**
**(farrens@cs.ucdavis.edu)**

**Andrew R. Pleszkun**

**Department of Electrical and**
**Computer Engineering**
**University of Colorado-Boulder**
**Boulder, CO 80309-0425**
**(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. Introduction

The goal of most high-performance computers is to maximize the amount of work that can be done per unit time. This quantity (work) can be expressed by the following equation [HePa90]:

$$work = clock\ rate \times \frac{1}{instruction\ count} \times \frac{1}{Clocks\ per\ Instruction}$$

A number of different approaches are used to increase this quantity. CRAY-class computers [Inc82] focus on driving up the clock rate, while CISC machines like the VAX [ClLe82] attempt to reduce the number of instructions required to complete a task. Most current single-chip processors try to both maximize the clock rate and minimize the number of clocks per instruction (CPI) at the expense of an increase in the number of instructions required.

While the clock rate achievable is *primarily* a function of technology, the CPI is not. By exploiting the inherent parallelism available in a task, the CPI term can be reduced dramatically. This parallelism comes in two forms: Instruction level parallelism and data parallelism.

Instruction level parallelism is a characteristic of the programming model, which separates a task into a number of different "subtasks" (execution control flow, memory I/O activity, data manipulation, etc). In many cases these operations are completely independent (e.g. the calculation of an address used to fetch data often has nothing to do with the eventual manipulation of that data) and are therefore prime candidates for parallel execution.

Data parallelism, on the other hand, is a property of the task at hand. Some tasks contain data manipulations that are completely independent from one another. These tasks can be thought of as containing explicit parallelism that can be exploited regardless of the programming model employed. Many architectures exist which exploit data parallelism (such as vector processors, dataflow machines, SIMD and MIMD designs, and many others [AlGo89]).

The key difference between instruction level parallelism and data parallelism is how closely coupled the *independent* operations are. With instruction level parallelism, this coupling is very close, necessitating frequent bi-directional communication between operation units. This high level of communication can often be avoided in data parallel applications.

The MISC processor, a direct descendant of the PIPE project [CGKP87, FaPl91b, 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. For example, MISC supports multiple instruction issue without sacrificing clock rate or increasing code size. 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. Furthermore, the regularity and simplicity of the MISC design should allow for faster implementation and increased scalability.

## 2. Background

Any attempt to exploit parallelism must adequately deal with the various types of dependencies that exist between instructions. These dependencies can be due to either the semantics of the task or from the limitations imposed by the architecture. *Data* dependencies (or *flow* dependencies) are caused by the sequential nature of a program's data. This type of dependency occurs when an item is created (written) by an instruction before it is evaluated (read) by a succeeding instruction. Also knowns as Read After Write (RAW) hazards, these dependencies originate outside of the architecture and are not susceptible to elimination by clever architectural designs.

A different type of dependency is created by the model of storage used by most computer architectures. These are often called *anti*-dependencies or *output* dependencies, and they occur when an attempt is made to reuse a storage location. If an instruction attempts to read a location and a succeeding instruction attempts to write to the same location, then an anti-dependency between those instructions is created. (This is also known as a Write After Read (WAR) hazard.) If we allowed the instructions to issue on the same clock cycle, it may be impossible to guarantee that the instruction performing the read receives the correct value. Similarly, an output dependency (or Write After Write (WAW) hazard) may exist between two instructions writing to the same location. Anti-dependencies and to a lesser extent output dependencies create problems for multi-issue architectures, such as SuperScalar [ Johnson 1990 ] and VLIW [Fish83], which use shared registers to facilitate communication between the separately issued instructions. These architectures must employ hardware techniques like register renaming [Kell75] to alleviate this problem. Since MISC does not use shared registers, additional RAW and WAW hazards are avoided.

If a dependency exists between two instructions, then a certain amount of time must elapse between the execution of the two instructions. If an instruction is waiting on a memory read, for example, the access time of memory will determine how many cycles must elapse before the instruction can issue. Faster memories can reduce this delay; this is one of the major benefits of memory caching.

An alternate approach to reducing memory latency is to maximize the distance between the read operation and the instruction requiring the data. This is done in the hope that such a separation will allow the read operation to precede the dependent instruction by enough cycles to hide

the latency (by allowing the processor to schedule unrelated work during the latency period). Decoupled Architectures [Smit82, SmWP86] use this method to mask high memory latencies without the use of data caches. Various studies have shown that the decoupling of memory access functions from the processing of the data can lead to a significant increase in processor performance.

Latency can also be found in the functional units of a processor. If the result of a floating point multiplication instruction is required by a succeeding instruction, for example, the value will remain unavailable for a number of clock cycles while the calculation proceeds - necessitating the delay in issue of the dependent instruction. As in the case of memory latency, this operational latency can be hidden if unrelated instructions can be scheduled between the instructions with the data dependency. *Loop unrolling* is a compiler optimization that can often provide these unrelated instructions in tasks exhibiting some data parallelism. *ALU chaining*, on the other hand, is a hardware technique used in the CRAY1 to both increase the parallelism of the machine and subsume the latency of the deeply pipelined (and therefore high operational latency) ALUs by allowing the result of one ALU operation to be used as an input of another without storing the intermediate result in a register. MISC will utilize a similar feature called *processor chaining* to achieve a more powerful method of chaining.

## 3. The MISC Design

The MISC system, an outgrowth of the PIPE project [CGKP87, FaPl91b, GHLP85], is being developed at the University of California, Davis to study the characteristics of a broad range of C programs in order to determine the feasibility of efficiently extracting instruction level parallelism and data parallelism from a single chip (scalable) MIMD processor design. 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 a processor and its registers. The following sections describe the MISC component structure and the design of the processing elements and the the data cache.

### 3.1. MISC Component Structure

The MISC system described in this paper 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

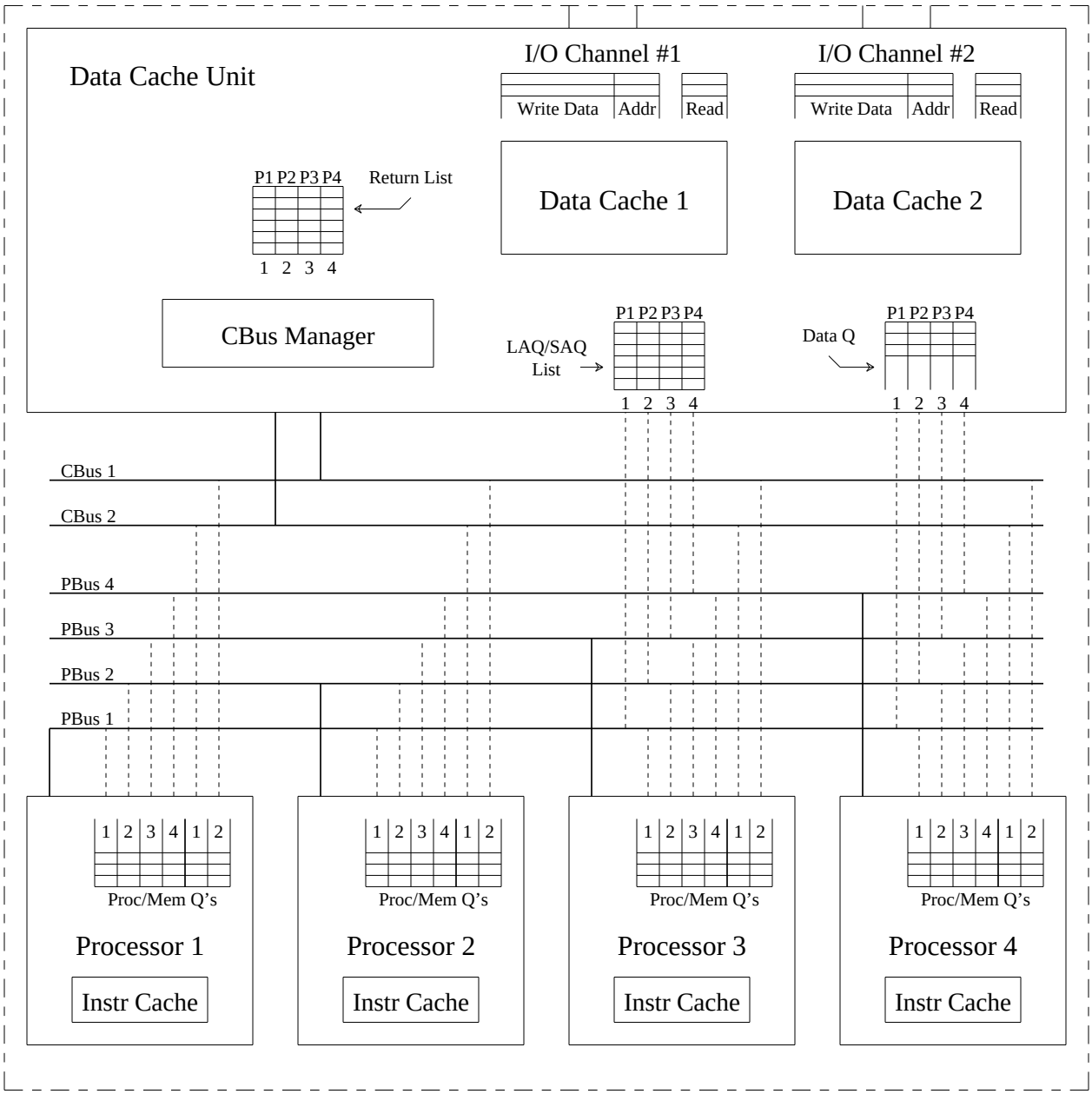


**Figure 1**

**MISC Component Design**

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 the 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.

### 3.2. Processor 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 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 independent instruction streams that cooperate in the execution of a single task. Currently all PE are assumed 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 the following components:

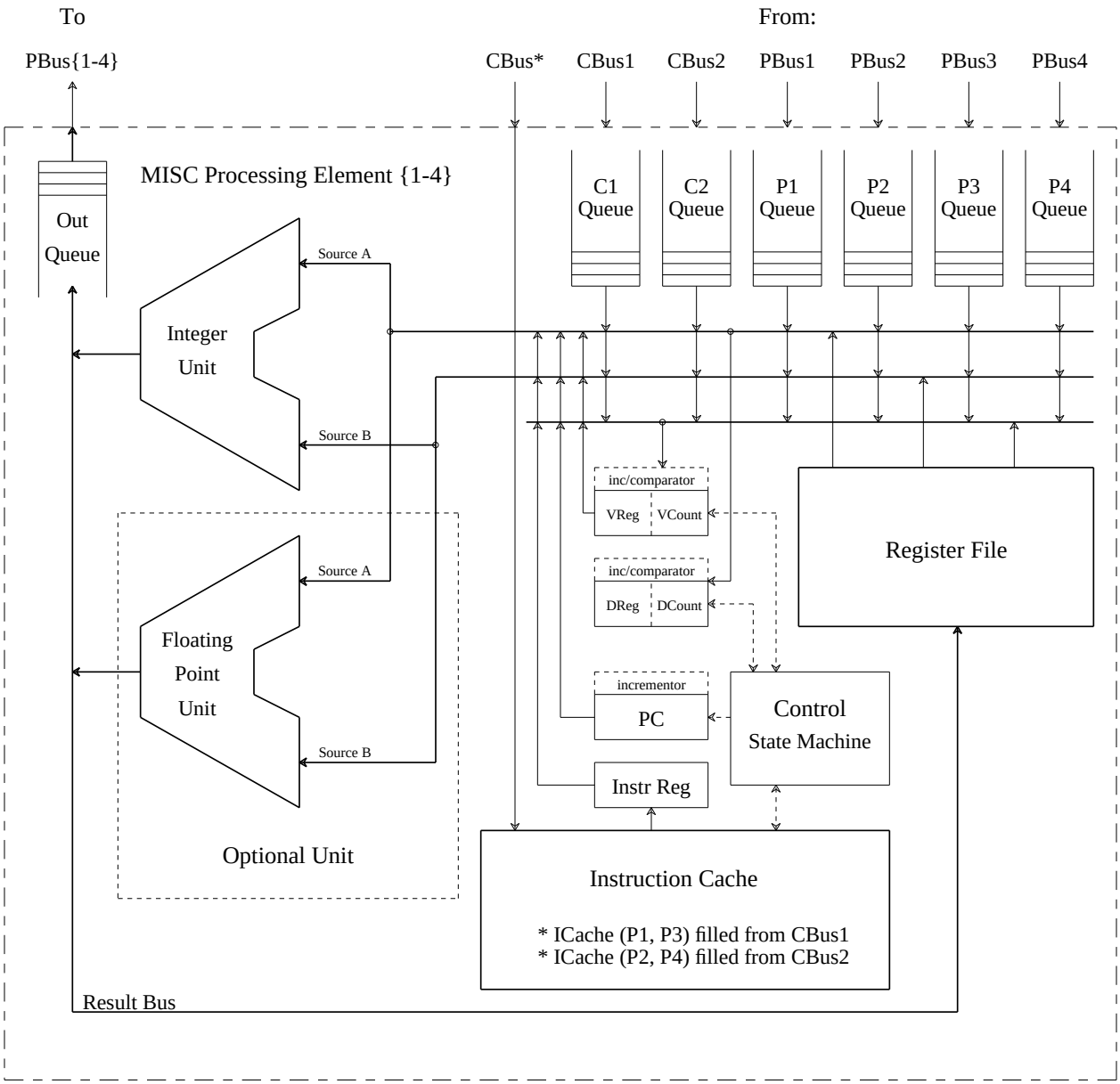


**Figure 2**

**MISC Processing Element**

32 General Purpose Registers
4 PE Data Queues (PEQ's)
2 Memory Data Queues (MQ's)
Integer ALU unit (ALU)
Floating Point Arithmetic Unit (FPU)
(GPR's) Program Counter (PC)
Vector Register (VREG)
Result Queue (OutQ)
Delay Register (DREG)
Instruction Cache (ICache)

The 32 GPR's are available for data storage which persists over multiple references, as well as software controlled environment variables (e.g. stack pointer, argument pointer, etc.). Therefore, in a 4 PE MISC system, 128 registers ($4 \times 32$) are available to store program data. Each PE also contains a FIFO queue (PEQ) for each PE in the system (including itself) to store data transfers between PEs. In addition, each PE has 2 memory queues (MQ's) which contain data requested from memory. While a single MQ would be sufficient (there is only 1 memory system), the frequency of requiring two independent memory streams is great enough that an additional MQ has been allocated (as an architectural feature) to improved performance and simplify code scheduling.

The size of each queue (PEQ and MQ) is not given here, but 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.

The Result Queue (OutQ) contains execution results that are scheduled to leave the PE (heading for memory and/or PEs). This queue may seem unnecessary since there is no contention for the transmission bus (remember, each PE has a dedicated write bus); however, its use simplifies the issue logic of the PE by avoiding a pipeline stall at the end of the execution phase due to a *busy* line assertion. (The operation of the Result Queue (OutQ) is invisible to the program.)

In addition to the 32 general purpose registers, two special purpose registers are visible to the program. The *Program Counter* is used to schedule the instruction flow and is automatically incremented after each instruction, and can also be modified explicitly by the execution of a branch instruction. The *vector register* is used to store the iteration value for vector instructions (see section 3.2.2). Both the *PC* and the *VREG* are available as source inputs to all instructions in the same manner as the GPRs and the various queue elements.

The Delay Register is used to determine the number of delay slots that should be unconditionally executed following a branch instruction, similar to the approach used in the PIPE processor [FaPl91b]. It also contains the size of a *vector loop* in the VLOOP instruction discussed in section 3.2.3.

Finally, each PE has its own instruction cache. Simulations have shown that 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.

### 3.2.1. Instruction Format

All instructions in MISC are 32 bits in length, and are of two distinct formats (see Figure 5). Instructions conforming to the four operand format may have up to three source operand specifiers and a destination specifier. Immediate instructions have a single source specifier, a 12 bit signed constant and a destination specifier.

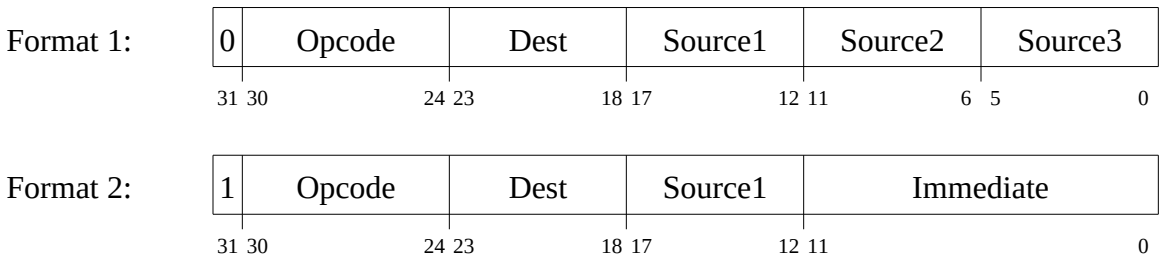


**Figure 5. MISC Instruction Formats**

As can be seen in figure 5, a MISC instruction uses 6 bits to specify a source or destination operand. With 32 GPRs, only 5 bits are required to uniquely identify each general purpose register. As shown in figure 6, if the first bit is a 0, then the other 5 bits are used to address one of these 32 GPRs.

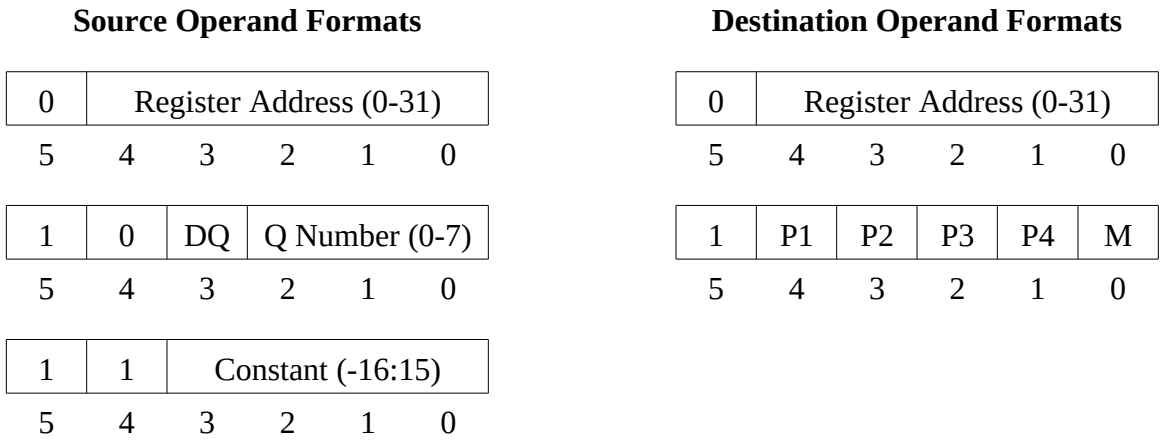


**Figure 6. MISC Operand Formats**

In the case of a destination specifier, if the first bit is set, the remaining 5 bits are used as *routing* information for a data transfer onto the PE's PBUS. For a source specifier, each operand can reference one of the input queues in a destructive or non-destructive manner. If the DQ bit is set, then the item read from the queue is de-queued; if the DQ bit is not set, then that item remains at the head of the queue. Also, the two special registers (*PC* and *VREG*) can be read in

this manner. Access to the *PC* allows for PC-relative addressing; access to *VREG* is often useful for removing the loop induction variable. See figure 7 for a list of queue specifiers.

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. Each source operand can also specify a small constant value. The ability to support a small constant for each operand in a three operand instruction provides considerable flexibility in code scheduling.

| Source Operand Q Number Specification | | | |
|---|---|---|---|
| 4 | 5 | 6 | Input Item |
| 0 | 0 | 0 | Processor Queue #1 |
| 0 | 0 | 1 | Processor Queue #2 |
| 0 | 1 | 0 | Processor Queue #3 |
| 0 | 1 | 1 | Processor Queue #4 |

| Source Operand Q Number Specification | | | |
|---|---|---|---|
| 4 | 5 | 6 | Input Item |
| 1 | 0 | 0 | Input Memory Queue #1 |
| 1 | 0 | 1 | Input Memory Queue #2 |
| 1 | 1 | 0 | Induction Variable (VREG) |
| 1 | 1 | 1 | Program Counter (PC) |

**Figure 7. Queue Specification for Source Operands**

There are four different types of MISC instructions: Scalars, immediate scalars, vector operations and a sentinel operation (which uses a sentinel value to terminate iterations of the instruction). A more complete description of these instructions is provided in Appendix A.

### 3.2.2. Scalars and Immediate Scalars

Scalar and immediate scalar instructions include ALU/FPU operations, load and store request instructions, control flow instructions and some special purpose instructions used to control the execution model.

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 [FaPl91a] 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. In addition to scalar branch instructions, MISC supports a vector loop instruction and eliminates the need for branching in some circumstances with vector and sentinel operations (these instructions will be described in more detail later).

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, for example, the *dest* field will contain the set of PEs that are to receive the data. The *dest* field should not specify a register, nor should it specify the memory (M) *routing* line. 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. A separate SAQ2 instruction unnecessary as no data is sent on the CBUS.

Each PE also has a number of special instructions used to provide efficient interrupt capabilities, debugging support and independent PE execution. A description of these instructions is beyond the scope of this paper.

### 3.2.3. Vector

Basic blocks will be very small in MISC. Therefore, a class of vector instructions have been specified in order to help reduce the number of branches needed to perform certain tasks. Three types of vector operations have been defined: ALU/FPU, LAQ/SAQ, and Vector Loop.

### 3.2.4. Vector ALU/FPU

These vector instructions require one additional source operand specifier (*src3*) (which is used as a vector count), and work as follows: When a vector instruction arrives at the issue stage of the instruction pipeline, the vector register (*VREG*) is cleared and the 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.

### 3.2.5. Vector LAQ/SAQ

The vector load and store instructions differ slightly from the ALU/FPU vector instructions, because there are now five pieces of information required (dest, src1, src2, count and stride) and only four fields in an instruction. There are two ways to approach this - the fields in a vector load/store instruction can be interpreted differently than in other instructions, or the interpretations can remain the same and two processors can essentially cooperate to create a vector load.

In order to maintain orthogonality, we decided to follow the second course of action. Therefore, as in the ALU/FPU vector instructions, the vector load instruction adds *src1* to *src2* and places the result in the *dest* register, repeating this action *src3* times.

Two PE's must act in concert in order to generate a vector load with a stride of more than one. For example, one PE would execute a vector add of the previous offset + stride, and direct it's output to the input queue of a different PE. This other PE would be executing a vector load of the base address and the input queue containing the correct offset. However, if the stride of a vector is one, then using the base address as one source input and *VREG* as the second performs a vector load without requiring any assistance from any other PE's. (The purpose of allowing the *VREG* to be a source input can now be seen). Since a vector with a stride of one is quite common, and the PE's are designed for ease of communication anyway, we feel the decision to keep the instruction set orthogonal is the correct one. (It is worth noting that the four operand scalar instructions are in fact a degenerate case of vector instructions, with a constant value 1 as the third source operand.)

### 3.2.6. Vector Loop

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 instruction uses the vector register in conjunction with the delay register 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 following activities occur: First, 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. See Appendix B for a more complete description of the vector instructions.

### 3.2.7. Sentinel

Vectors are the most common organization of variable length arrays. However, a significant alternate mechanism is the use of a trailing sentinel (special) value to mark the termination of a string of data. (This is the mechanism used by the C libraries to manipulate character string data.) MISC supports the efficient handling of string (and other sentinel) "types" by incorporating the sentinel mechanism into a second class of vector-like instructions: the Sentinel instructions. 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). This instruction works as follows: The contents of the specified register are compared to the sentinel value. If they do not match, the scalar version of the sentinel 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.

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 (a *while* loop in this instance) 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. See Appendix C for a further description of the sentinel instructions.

## 3.3. 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. Finally, the CBUS interface is responsible for 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. Further discussion of DCache operation is beyond the scope of this paper.

## 4. Analysis

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 C source for LLL #3 follows:

```
main() {
    float z[1000], x[1000], q;
    int k;

    q = 0.0;
    for ( k=0 ; k < 1000 ; k++ )
        q = q + z[ k ] * x[ k ];
}
```

The MISC object code generated consists of 4 instruction streams. Processing element 1 is responsible for relating loop control information and generating the load requests for vector X. Processing element 2 will generate the load requests for the vector Z. Processing elements 3 and 4 will cooperate in establishing a processor chain to perform the calculation, with PE3 performing the multiplication and PE4 calculating the sum. What follows is the annotated object code of each of the 4 PEs:

| | | | | |
|---|---|---|---|---|
| **PE1** | | | **CONTROL and ACCESS processor (#1)** | |
| [1] | add | {P1,P2,P3,P4} | <- #0, #1000 | ; send loop count to all PE |
| [2] | add | R1 | <- #0, _x | ; R1 = base address of x |
| [3] | vlaq | {P3} | <- R1, VREG, PQ1 | ; fetch x[0] - x[999] for PE3 MQ1 |
| **PE2** | | | **ACCESS processor (#2)** | |
| [4] | add | R1 | <- #0, _z | ; R1 = base address of z |
| [5] | vlaq2 | {P3} | <- R1, VREG, PQ1 | ; fetch z[0] - z[999] for PE3 MQ2 |
| **PE3** | | | **EXECUTE processor (#3)** | |
| [6] | vfmul | {P4} | <- MQ1, MQ2, PQ1 | ; multiply vectors X and Z for P4 |
| **PE4** | | | **EXECUTE processor (#4)** | |
| [7] | fadd | R1 | <- #0, #0 | ; q = 0.0; |
| [8] | vfadd | R1 | <- R1, PQ3, PQ1 | ; sum q with vector from P3 |

PE1 is first responsible for relating loop control to 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]).

The first noticeable characteristic of this code is that there are no explicit branch instructions; all "branching" has been removed in favor of vector operations (lines [3],[5],[6], and [8]). Perhaps the second most striking characteristic is the amount to which this code resembles a data-flow representation of the program. Data flow patterns are establish from PE1 and PE2 (through the memory system) to PE3, and from PE3 to PE4. Decoupling the read requests from the data manipulation allows processors PE1 and PE2 to continue execution without dependency (as long as PE3's MQs do not fill). This lack of dependencies allow the memory system to be exercised to its full capacity.

The operational latency of the *vfmul* instruction in the code for PE3 does not greatly effect the execution time because there exists no dependency in P3 on the result of the calculation (PE4 is delayed only on the first element of the vector). This demonstrates a simple form of processor chaining; removing circular data dependencies which reduce processor performance. Since the *vfmul* instruction executing on PE3 depends only on the vector X and Z (supplied by PE1 and PE2 through memory), successive multiplications can proceed on successive cycles independent of the addition taking place with the result of the multiplication. The performance of this loop is determined by the maximum I/O throughput of the processor. In the system described, assuming a data cache hit ratio of 1, an iteration of the loop takes 1 clock cycle. Including some startup costs (initiation instructions and latencies) the total execution time is 1022 cycles. There were a total of 4004 instructions issue from the PEs leading to a CPI rate of 0.25524, or reciprocally 3.9178 instructions were allowed to issue each 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 (*str1). 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.

```
main() {
    char *str1 = "example string", str2[15];
    while (*str2++ = *str1++) ;
}
```

The code generated utilized 3 of the 4 MISC processing elements (the forth PE can process any independent task if applicable). As in the LLL3 example, PE1 is responsible for distributing the loop control information to all PEs cooperating in the execution of this loop. Unlike the previous example, the loop control information cannot be concisely conveyed in a single transmission. The condition of the loop must be re-evaluated after each iteration depending on the value of str1 fetched from memory. Therefore, PE1 is also responsible for performing the load requests of the characters in str1. PE2 generates the store requests for character placement in str2 and PE3 performs the actual character manipulation (movement).

| | | | | |
|---|---|---|---|---|
| **PE1** | | | **CONTROL and ACCESS processor (#1)** | |
| [1] | add | R1 | <- #0, _str1+1 | ; R1 = base address of str1 + 1 |
| [2] | laq | {P1, P2, P3} | <- R1, #-1 | ; pre-read first character for PE1-3 |
| [3] | slaq | {P1, P2, P3} | <- R1, VREG, MQ1 | ; read str1[1] - end of string (EOS) |
| | | | | |
| **PE2** | | | **ACCESS processor (#2)** | |
| [4] | add | R1 | <- #0, _str2 | ; R1 = base address of str2 |
| [5] | ssaq | P3 | -> R1, VREG, MQ1 | ; generate SAQ for str2 until EOS |
| [6] | saq | P3 | -> R1, VREG | ; generate SAQ for str2 EOS marker |
| | | | | |
| **PE3** | | | **EXECUTE processor (#3)** | |
| [7] | sadd | {M} | <- MQ1, #0, MQ1 | ; move each character in str1 to str2 |
| [8] | add | {M} | <- #0, #0 | ; move EOS in str1 to str2 |

PE1 contains the most involved code in this example and it has been optimized to demonstrate the decoupled nature of this machine. As stated above, PE1 performs two inter-related tasks: loop control and load request. Line [1] simply initializes a register to allow for reading of str1. Lines [2] and [3] jointly perform the load request and transmit the loop control information to the relevant PEs. Line [3] generates a load request for a character as long as the previous character was not the End of String (EOS) sentinel. The function of line [2] is to generate the

load request for the first character of the string; this cannot be performed by the instruction at line [3] due to the dependency on MQ1 (the previous character). 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. This sentinel instruction continues until the EOS character is reached. It is worth noting that the src1 and src3 operands specify the same queue. This is allowed in MISC and each operand is evaluated to the same (front) element of the queue. All processing is now completed except for the movement of the EOS character, which occurs in line [8].

Though the code generated for this loop is the same length as LLL3, the execution time is determined by the memory latency since a dependency exists in PE1's PQ1 (the test of *str1). Each iteration of the *slaq* instruction in PE1 must wait for the *Src3* item to arrive in MQ1. Since this instruction will be responsible for generating the memory request for the next MQ1 item, the full memory latency period must occur between iterations of this instruction. Furthermore, since both PE2 and PE3 have dependencies resulting from this same instruction, those PE are also stalled until the data is received from the memory system. Assuming a cache hit ratio of 1 and a DCache latency of 4, execution takes 66 cycles in this example. A total of 50 instructions are issued, translating into a CPI of 0.7576, or 1.32 instruction issues per cycle. Achieving a CPI less that one for this example is an excellent result.

## 5. Conclusions and 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**


[AlGo89] G. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, (1989).

[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.

[ClLe82] D. W. Clark and H. M. Levy, ‘‘Measurement and Analysis of Instruction Use in the VAX-11/780’’, *Proceedings of the Ninth Annual International Symposium on Computer Architecture*, Austin, Texas (April 1982), pp. 9-17.

[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).

[FaPl91a] 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.

[FaPl91b] M. Farrens and A. Pleszkun, ‘‘Implementation of the PIPE Processor’’, *Computer*(January 1991), pp. 65-70.

[Fish83] J. A. Fisher, ‘‘Very Long Instruction Word Architectures and the ELI-512’’, *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, Stockholm, Sweden (June 13-17, 1983), pp. 140-149.

[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.

[HePa90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Mateo, California, (1990).

[Inc82] C. R. Inc, ‘‘Cray-1 Computer Systems ’’, *S Series Mainframe Reference Manual (HR-0029)*(1982).

[Kell75] R. M. Keller, ‘‘Look-ahead Processors’’, *ACM Computing Surveys,* vol. 7, no. 4 (December 1975), pp. 177-195.

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

[Smit82] J. E. Smith, ‘‘Decoupled Access/Execute Computer Architectures’’, *Proceedings of the Ninth Annual International Symposium on Computer Architecture*, Austin, Texas (April 26-29, 1982), pp. 112-119.

[SmWP86] J. E. Smith, S. Weiss and N. Y. Pang, ‘‘A Simulation Study of Decoupled Architecture Computers’’, *IEEE Transactions on Computers,* vol. C-35, no. 8 (August 1986), pp. 692-702.