INSTRUCTION ISSUE LOGIC FOR HIGH-PERFORMANCE, INTERRUPTIBLE, MULTIPLE FUNCTIONAL UNIT, PIPELINED COMPUTERS

by

Gurindar S. Sohi

Computer Sciences Technical Report #704

July 1987

INSTRUCTION ISSUE LOGIC FOR HIGH-PERFORMANCE, INTERRUPTIBLE, MULTIPLE FUNCTIONAL UNIT, PIPELINED COMPUTERS

Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706

July, 1987

Abstract

The performance of pipelined processors is limited by data dependencies and branch instructions. In order to achieve high performance, mechanisms must exist to alleviate the effects of data dependencies and branch instructions. If the processor is to support virtual memory, it is essential that a precise state of the machine be recoverable at all times, i.e., interrupts must be precise. In multiple functional unit pipelined processors where the instructions can complete and update the state of the machine out of program order, hardware support must be provided to implement precise interrupts. In this paper, we combine the problems of data dependencies and imprecise interrupts. We present a design for a hardware mechanism that resolves dependencies dynamically and, at the same time, guarantees precise interrupts, without a significant hardware overhead. Simulation studies, using the Lawrence Livermore loops as a benchmark, show that by resolving dependencies, the proposed mechanism is able to obtain a significant speedup over a simple instruction issue mechanism and, at the same time, is able to implement precise interrupts. We also discuss how this mechanism could be used to alleviate the effects of branch instructions.

1. INTRODUCTION

The CPUs of most supercomputers consist of several pipelined functional units connected together in some fashion [1,2]. Such multiple functional unit, pipelined machines are able to achieve a considerable overlap in the execution of instructions. Unfortunately, pipelined CPUs have two major impediments to their performance: i) *data dependencies* and ii) *branch instructions*. An instruction cannot begin execution until its operands are available. If an operand is the result of a previous instruction, the instruction must wait till the previous instruction has completed execution, thereby degrading performance. The performance degradation due to branch instructions can be even more severe. Not only must a conditional branch instruction wait for its condition to be known, an additional penalty may be incurred when fetching an instruction from the taken branch path to the instruction decode and issue stage.

Pipelined CPUs suffer from another major problem - an interrupt can be *imprecise* [3-5]. This problem is especially severe in multiple functional unit computers in which instructions can complete execution out of program order even though they are issued in program order [1-3]. For a high-performance, pipelined CPU, an adequate solution must be found for the imprecise interrupt problem and means must be provided for overcoming the performance degradation due to data dependencies and branch instructions.

The detrimental effects of branch instructions can be alleviated by using *delayed* branch instructions. However, the utility of delayed branch instructions is limited for long pipelines. In such cases, other means must exist to alleviate the detrimental effects. A common approach is to use *branch prediction* [6,7]. Using prediction techniques, the probable execution path of a branch instruction is determined. Instructions from the predicted path can then be fetched into instruction buffers or even executed in a *conditional mode*[3,8-11]. While the conditional mode of execution will result in a higher pipeline throughput, especially if the outcome of the branches is predicted correctly, a hardware mechanism must exist which will allow the machine to recover from an incorrect sequence of conditional instructions.

Both hardware and software solutions exist to the data dependency problem. Software solutions use code scheduling techniques (combined with a large set of registers) to increase the distance between dependent instructions and to provide interlocks [12]. Most hardware solutions employ *waiting stations* or *reservation stations* where an instruction can wait for its operands and allow subsequent instructions to proceed [13], thereby allowing instructions to issue out of program order. The reservation stations form the core of a *dependency-resolution* mechanism that must exist in order to preserve program dependencies. In this paper, a dependency-resolution mechanism is synonymous with an out-of-order instruction issue mechanism. Note the difference between out-of-order instruction issue (also called out-of-order instruction execution) and out-of-order instruction completion. Instructions can complete out of program order even though they were issued in program order.

In a pipelined machine, imprecise interrupts can be caused by instruction-generated traps such as arithmetic exceptions and page faults. An imprecise interrupt can leave the machine in an irrecoverable state. While the occurrence of arithmetic exceptions is rare, the occurrence of page faults in a machine that supports virtual memory is not. Therefore, if virtual memory is to be used with a pipelined CPU, it is crucial that interrupts be precise. Several hardware solutions to the problem are described in [5]. We are unaware of any software solutions to the imprecise interrupt problem for multiple functional unit computers. A software solution will be extremely difficult, if not impossible. Not only must the software allow for the worst-case execution time for any instruction, it must also keep track of instructions that have completed out of program order and generate the appropriate code sequence to undo the effects of those instructions. In any case, some hardware support must be provided to maintain run time information.

In this paper, we treat the problems of out-of-order instruction issue and imprecise interrupts simultaneously. If interrupts are to be precise, some hardware support is needed. The precise-interrupt mechanism will aggravate dependencies [5]. Why not extend this mechanism to allow out-of-order instruction execution as well so that the aggravated dependencies (as well as other dependencies) can be tolerated?

In section 2, we describe the model architecture that we use throughout this paper. In section 3, we discuss Tomasulo's out-of-order instruction issue algorithm and extend it, giving several variations, so that the cost of implementing it is not prohibitive even for a large number of registers. Simulation results for the proposed dependency resolution mechanism are presented. In section 4, we discuss the problem of imprecise interrupts and present solutions. Section 5 describes a unit, the Register Update Unit (RUU), that resolves dependencies as well as implements precise interrupts. The precise interrupt and out-of-order instruction issue mechanisms mutually aid and simplify each other. An evaluation of the RUU is carried out in section 6. Finally, we discuss how our mechanism might be used to alleviate the degradation due to branch instructions.

2. MODEL ARCHITECTURE

The model architecture that we use for our studies is presented in Figure 1. It has the same capabilities and executes the same instruction set as the scalar unit of the CRAY-1 [1, 14]. The CRAY-1 was chosen because it represents a state-of-the-art scalar unit and its execution can be modeled precisely. The author also had easy access to tools that could be used to generate instruction traces for the CRAY-1 scalar unit [15]. There are a few differences between the CRAY-1 scalar unit and our model architecture. First, in our model architecture, all instructions, whether they are composed of 1 parcel (16 bits) or 2 parcels (32 bits) can issue in a single cycle if issue conditions are favorable. Next, only one function can output data onto the result bus in any clock cycle. In contrast, the CRAY-1 scalar unit has separate result busses for the address and scalar functional units. Instructions are fetched by the *Instruction Fetch Unit* and decoded and issued by the *Decode and Issue Unit*. Once dependencies have been resolved in the decode and issue unit, instructions are forwarded to the functional units for execution. The results of the functional units are written directly into the register file. The register file consists of 8 A, 8 S, 64 B and 64 T registers.

2.1. Benchmark Programs

The benchmark programs used throughout this paper were the first 14 Lawrence Livermore loops [16]. The first 14 loops were chosen because they were readily available. Henceforth, we shall refer to them as LLL1, LLL2, ..., LLL14. The simulations were carried out as follows. The benchmark programs, as compiled by the CFT compiler for the scalar unit, were fed into a CRAY-1 simulator [15]. The CRAY-1 simulator generates an instruction trace for each program. Vector instructions are not used. Each instruction trace was then fed into the appropriate simulator.

2.2. Simulation of the Model Architecture

We simulated the execution of the benchmark programs on the model architecture of Figure 1. The number of instructions executed and the number of clock cycles taken for the

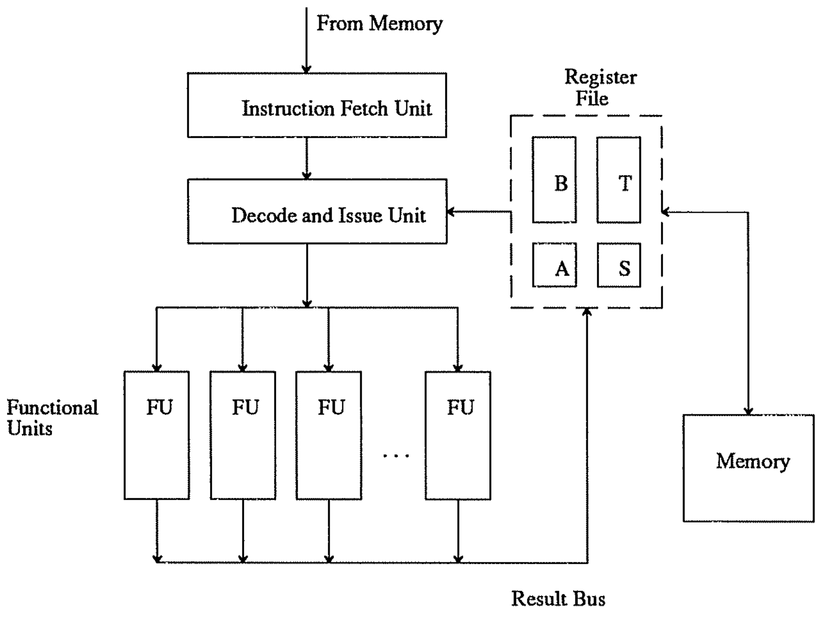

Figure 1. The Model Architecture

execution of each benchmark program and the number of instructions executed per cycle is given in Table 1. In generating the results of Table 1, we assumed that (i) no memory bank conflicts occur, (ii) all instruction references are serviced by the instruction buffers, and (iii) the instructions are already present in the instruction buffers when the program is started. These assumptions do not affect the execution time considerably for the benchmark programs. These assumptions and a difference in the bus structure account for the difference between the data presented in Table 1 and in [17].

Table 1: Statistics for the Benchmark Programs

Benchmark Program	Instructions Executed	Clock Cycles to Execute Program	Instruction Issue Rate
LLL1	7217	17234	0.419
LLL2	8448	17102	0.494
LLL3	14015	36023	0.389
LLL4	9783	20643	0.474
LLL5	8347	20696	0.403
LLL6	9350	22034	0.424
LLL7	4573	10231	0.447
LLL8	4031	8026	0.502
LLL9	4918	10134	0.485
LLL10	4412	9420	0.468
LLL11	12002	28002	0.429
LLL12	11999	27991	0.429
LLL13	8846	17814	0.497
LLL14	9915	23573	0.421
Total	117856	268923	0.438

The instruction issue rate is the average number of instructions that are executed in a cycle, i.e., the total number of instructions executed in the benchmark divided by the total number of cycles to execute the benchmark. The instruction issue rate for the total of all 14 loops is calculated as the total number of instructions for all loops divided by the total number of cycles to execute all loops rather than a average of the individual issue rates. For reasons of brevity, we shall present all subsequent simulation results for a combination of all 14 loops rather than report the detailed breakup for each individual loop.

As we can see from Table 1, the performance of the model machine is far from the theoretical limit of 1 instruction per cycle. From our simulations, we determined that the main reason for this sub-optimal performance is data dependencies. Therefore, we must find some way of alleviating the affects of data dependencies. We have two choices: (i) eliminating the dependencies or (ii) tolerating the dependencies. Data dependencies can be eliminated by software code scheduling techniques. Hardware dependency resolution techniques allow the machine to tolerate dependencies. Since we are mainly concerned with a hardware mechanism that allows the architecture to tolerate dependencies as well as implement precise interrupts, we can restrict our attention to hardware mechanisms for tolerating dependencies.

3. HARDWARE DEPENDENCY RESOLUTION

When an instruction reaches the decode and issue stage in the pipeline, checks must be made to determine if the operands for the instruction are available, i.e., if all dependencies for this instruction have been resolved. If an operand is not available, the instruction must wait in the decode and issue stage. Because the decode and issue stage of the pipeline is busy, subsequent instructions cannot proceed even though they may be ready to execute. Subsequent instructions can proceed if the waiting instruction "steps aside," thereby freeing the decode and issue stage and allowing other instructions to bypass it while it waits for its operands. In order to do so, some form of waiting stations or *reservation stations* must be provided [13]. Other mechanisms also exist in the literature [18]. Since our work is based on the concept of reservation stations, we shall focus our attention on mechanisms that employ reservation stations in some form.

3.1. Tomasulo's Algorithm

Tomasulo's hardware dependency-resolution (or out-of-order instruction issue) algorithm was first presented for the floating-point unit of the IBM 360/91 [13]. An extension of this algorithm for the CRAY-1 scalar unit is presented in [17]. The algorithm operates as follows. An instruction whose operands are not available when it enters the decode and issue stage is forwarded to a *Reservation Station (RS)* associated with the functional unit that it will be using. It waits in the RS until its data dependencies have been resolved, i.e., its operands are available. Once at a reservation station, an instruction can resolve its dependencies by monitoring the Common Data Bus (the Result Bus in our model architecture). When all the operands for an instruction are available, it is dispatched to the appropriate functional unit for execution. The result bus can be reserved either when the instruction is dispatched to the functional unit[17] or before it is about the leave the functional unit [13].

Each source register is assigned a *busy bit.* A register is busy if it is the destination of an instruction that is still in execution. Each destination register (also called a sink register) is assigned a tag which identifies the result that must be written into the register. Since any register in the register file can be a destination register, each register must be assigned a tag. A reservation station has the following fields:

Source Operand 1			Source Operand 2			Destination
Ready	Tag	Contents	Ready	Tag	Contents	Tag

If a source register is busy when the instruction reaches the issue stage, the tag for the source register is obtained and the instruction is forwarded to a reservation station. The appropriate ready bit in the reservation station is set to indicate that the source operand is unavailable. If the source register is not busy, the contents of the register are read into the reservation station and the ready bit is reset to indicate that the source operand is available. The instruction fetches a tag for the destination register, updates the old tag of the destination register and proceeds to a reservation station. When all source operands are available in a reservation station, the instruction is dispatched to a functional unit for execution and the reservation station is released for reuse by future instructions. Memory is treated as a special functional unit. When an instruction has completed execution, the result (along with its tag) appears on the result bus. The registers as well as the reservation stations monitor the result bus and update their contents (and ready/busy bits) when a matching tag is found. Details of the algorithm can be found in [13] and [17].

While this algorithm is straightforward and effective, it is expensive to implement because each register needs to be tagged and each tag needs associative comparison hardware to carry out the tag-matching process. This may not be practical if the number of *possible destination registers*, i.e., the number of registers is large. For our model architecture which has 8 A, 8 S, 64 B and 64 T registers, an implementation of this dependency-resolution mechanism for all the registers would require 144 tag-matching units. The use of such a large number of hardware units may not be practical.

3.2. Extensions to Tomasulo's Algorithm

3.2.1. A Separate Tag Unit

On closer inspection we see that very few of all *possible* destination registers may actually be active, i.e., be waiting for a result at any given time. Therefore, if we associate a tag with *each possible* destination register, a lot of associative tag-matching hardware will be idle at any given time. Why not have a common tag pool and assign a tag only to a *currently active* destination register rather than associating a tag with each possible destination register? In Tomasulo's algorithm, a currently active register is one whose busy bit is on.

We consolidate the tags from all *currently active* registers into a *Tag Unit (TU)*. Each register has only a single busy bit. At instruction issue time, if a source register is busy, the TU is queried for the current tag of the source register and the tag is forwarded to the reservation stations. A new tag is obtained for the destination register of the instruction. If the destination register is not busy, acquiring such a tag from the TU is straightforward. If the destination register is busy, i.e., the TU already holds a tag for the register, a new tag is obtained and the instruction holding the old tag is informed that, while it may update the register, it may not *unlock* the register, i.e., clear the busy bit when it completes execution. In order to do so, we associate another bit with each TU entry. This bit indicates if the tag is the latest tag for the register and if the instruction has a *key* to *unlock* the register, i.e., clear the busy bit. Instruction issue blocks if no tag can be obtained, i.e., the TU is full. The TU has the following fields:

Register Number	Tag Free	Latest Copy

The reservation stations are modified so that the result can be forwarded to the appropriate slot in the TU. The new reservation station has the following fields:

Source Operand 1			Source Operand 2			Destination
Ready	Tag	Contents	Ready	Tag	Contents	Slot in TU

As before, the instruction along with its associated tags/operands is forwarded to a reservation station where it waits for its operands to become ready. The result from a functional unit (along with its tag) is broadcast to all reservation stations and is also forwarded to the TU. Reservation stations monitor the result bus and gate in the result if the tag of the data on the result bus matches the tag stored in the reservation station. The TU forwards the result to the register specified in the appropriate slot of the TU. All registers are, therefore, updated only by the TU when their data is available and no direct connection is needed between the functional units and the register file. When the register has been updated by the TU, the corresponding tag is released and is marked free in the TU. The modified architecture that incorporates a Tag Unit and reservation stations associated with each functional unit is shown in Figure 2.

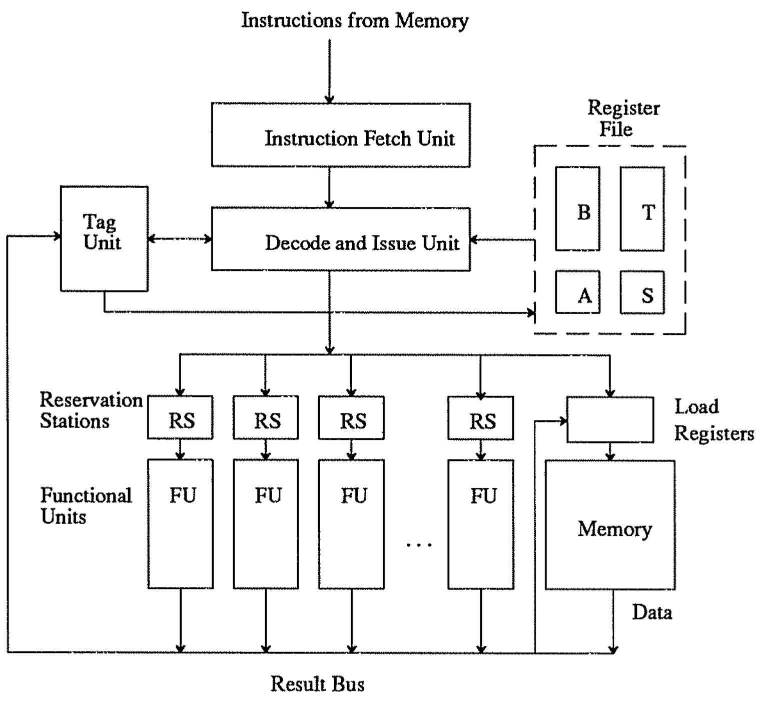

Figure 2. The Model Architecture with a Tag Unit and Distributed Reservation Stations

3.2.1.1. Example

The operation of the Tag Unit is best illustrated by an example. Consider a TU that has 6 entries as shown in Figure 3. Each entry in the TU has a bit indicating if the tag is free, i.e., available for use by the issue logic, a bit indicating if the tag is the latest tag for the register and a field for the number of the destination register.

The TU is indexed by the tag number. Consider the execution of an instruction I_1 that adds the contents of registers S0 and S7 and put the result in S4. Assume that the state of the TU is as shown in Figure 3 and that S7 is free (indeed a register must be free if it does not have an entry in the TU). When the issue logic decodes I_1, it attempts to get a *new* tag for the destination register S4 from the TU and obtains tag 3. Since the TU already has a tag for S4, the old tag (4) is updated to indicate that it no longer represents the latest copy of the register. Since S7's contents are valid, they can be read from the register file and forwarded to the reservation stations directly. However, since the contents of S0 are not valid, the latest tag for S0 (tag 2) must be obtained from the TU. The issue unit forwards a packet to the reservation station associated with the add functional unit. The packet contains the contents of S7, a tag (2) for S0 and a tag (3) for the destination register S4. I_1 waits in the reservation station till that tag 2 appears on the

Tag Number	Register Number	Tag Free	Latest Copy
1	A0	N	Y
2	S0	N	Y
3	NIL	Y	Y
4	S4	N	Y
5	S0	N	N
6	S3	N	Y

Figure 3: A Tag Unit

result bus. At this point, the reservation station reads the value for S0 and I_1 is ready to execute. When I_1 completes execution, i.e., leaves the add functional unit, the result is forwarded to all reservation stations that have a matching tag (3) and also to the TU. The TU forwards the result to the register file to be written into S4. Since tag 3 is the latest tag for S4, S4's busy bit can be reset when the data has been written into S4. Tag 3 is then marked free and is available for reuse by the issue logic.

3.2.1.2. Interactions with Memory

Instructions that interact with the memory, i.e., load/store instructions, are handled in a special manner. A set of *Load Registers* is responsible for resolving dependencies in the memory functional unit. The load registers contain the addresses of "currently active" memory locations. Each load register has tags to allow for multiple instances of a memory address, i.e., multiple outstanding requests to the memory location.

The reservation stations associated with the memory functional unit are managed in a pseudo-queue fashion to satisfy dependencies. If the address of a load/store operation is unavailable, subsequent load/store instructions are not allowed to proceed. If the destination address of a pending store instruction is known, subsequent load instructions can proceed in any order (if their addresses are available) as long as dependencies are not violated. When a load instruction is allowed to proceed, it checks to see if the address for the load operation matches an address stored in the load registers. If a match occurs, and the load register is not free, the load instruction is halted and is not submitted to memory. A match can occur if there is either a pending load or a pending store operation. In either case, the load need not be submitted to memory since the desired data can be obtained when the pending load or store operation completes. If a match occurs for a store instruction, the tag associated with the load register is updated.

If no match occurs for either operation, a free load register is obtained. A load register is free if there are no pending load or store instructions to the memory address held in the load register. The load request is submitted to memory. The corresponding tag is also submitted to memory so that the data supplied by the memory may be read by the appropriate source operands in the reservation station. Issue is blocked if a free load register is not available. The busy bit of the load register is updated when the load/store instruction completes execution.

3.2.2. Merging the Reservation Stations

If each functional unit has a separate set of reservation stations, it is likely that some functional unit will run out of reservation stations while the reservation stations associated with another functional unit are idle. As suggested in [17], we can combine all the reservation stations into a common *RS Pool* rather than having disjoint pools of reservation stations associated with each functional unit. All instructions that were previously issued to distributed reservation stations associated with the functional units now go to the common RS Pool. Instruction issue is blocked if no free reservation station is available, i.e., if the RS Pool is full. As instructions become ready in the RS Pool, they are issued to the functional units. All the other functions are as before.

3.2.3. Merging the RS Pool and the Tag Unit

In the Tag Unit, there is one entry for every instruction that is present in either the RS Pool or in the functional units. Therefore, at any time, there is a one-to-one correspondence between the entries in the TU and the instructions in the reservation stations or the functional units. This suggests that we can combine the RS Pool and the Tag Unit into a single *RS Tag Unit (RSTU).* Of course, a reservation station is wasted if it is associated with an instruction that is in a functional unit. However, as we shall see in section 5, this organization can easily be extended to allow for the implementation of precise interrupts.

In the RSTU, a reservation station is reserved at the same time that a tag is reserved. When an instruction issues, it obtains a tag from the RSTU and in doing so automatically reserves a reservation station. All the other functions, including interactions with the memory are as before. The architecture with a RSTU is given in Figure 4. Each entry in the RSTU is as follows:

Tag Free	Latest Copy	Source Operand 1		
Yes/No	Yes/No	Ready	Tag	Contents

Source Operand 2			Destination
Ready	Tag	Contents	Register

Note that in the RSTU, associative logic is needed for 2 purposes: (i) obtaining tags for source operands and updating the latest copy field and (ii) in the reservation stations for matching tags.

3.2.3.1. Simulation Analysis of the RSTU

In order to evaluate the effectiveness of the RSTU, we carried out a simulation analysis of the RSTU using the first 14 Lawrence Livermore loops as a benchmark. The results obtained for the execution of all 14 loops are presented in Table 2. The relative speedup is the speedup compared to the simple instruction issue mechanism of Table 1 and the instruction issue rate is the average number of instructions issued per cycle. The number of load registers was 6. This guarantees that instruction issue is never blocked because of an unavailable load register.

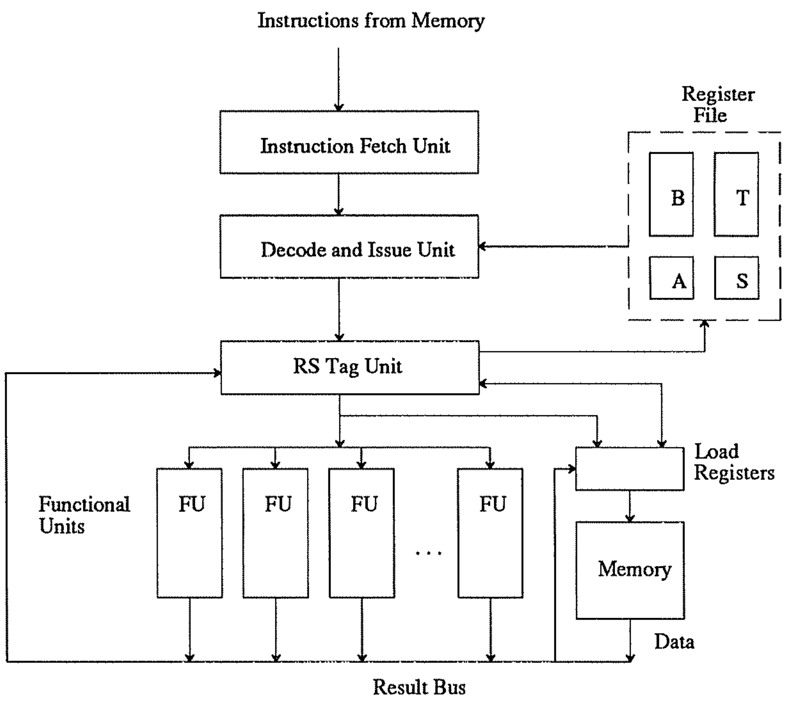

Figure 4. The Model Architecture with a RS Tag Unit

Table 2: Relative Speedup and Issue Rate with a RSTU

Number of Entries in RSTU	Relative Speedup	Instruction Issue Rate
3	0.965	0.423
4	1.140	0.499
5	1.294	0.567
6	1.424	0.624
7	1.479	0.648
8	1.553	0.681
9	1.587	0.696
10	1.642	0.720
15	1.763	0.773
20	1.798	0.788
25	1.820	0.798
30	1.821	0.798

From Table 2, it is quite clear that the RSTU is able to achieve a significant speedup over a simple instruction issue mechanism with a reasonable amount of hardware. The RSTU is also quite close to achieving the theoretical issue limit of 1 instruction per clock cycle. Indeed, all non-branch instructions are able to achieve the limit of 1 instruction per cycle. The only cycles in which no useful instruction instruction is executed are the dead cycles following each branch instruction. The degradation due to such cycles could be reduced by using delayed branch instructions or by executing instructions in a conditional mode. The results presented in Table 2 compare favorably with the results presented in [17]. Because the RSTU can implement the dependency-resolution mechanism for the B and T register files, it can achieve a better speedup than a mechanism that is somewhat restricted as in [17].

At first glance, it may seem that an organization with merged reservation stations (such as the RSTU of Figure 4) has a distinct disadvantage over an organization with distributed reservation stations (such as Figure 2) since only one instruction can issue from the reservation stations to the functional units in a clock cycle unless multiple paths are provided between the RSTU and the functional units. On the other hand, a better use of the reservations stations results since the reservation stations can be shared amongst several functional units. In order to evaluate the effectiveness of multiple data paths between the RSTU and the functional units, we simulated an architecture with 2 paths from the RSTU to the functional units, but only a single issue unit, a single result bus and single path from the RSTU to the register file. The results are presented in Table 3.

Table 3: Relative Speedup and Issue Rate with a RSTU and 2 Data Paths

Number of Entries in RSTU	Relative Speedup	Instruction Issue Rate
3	0.976	0.428
4	1.155	0.506
5	1.310	0.574
6	1.442	0.632
7	1.515	0.664
8	1.586	0.695
9	1.634	0.716
10	1.667	0.730
15	1.796	0.787
20	1.832	0.803
25	1.843	0.808
30	1.845	0.809

As is evident from Table 3, the presence of a duplicate path from the RSTU to the functional units makes a small difference. This result is not counter-intuitive. We use an argument based on instruction flow to convince the reader. The RSTU is essentially a reservoir of instructions that is filled by the decode and issue logic and drained by the functional units. Since the decode and issue logic can fill this reservoir at a maximum rate of 1 instruction per cycle, having a drain that is capable of draining more than 1 instruction per cycle will not be very useful in a steady state.

4. IMPLEMENTATION OF PRECISE INTERRUPTS

We now address the issue of precise interrupts. A complete description of several schemes that implement precise interrupts is given in [5]. The basic idea behind all these schemes is that the instructions must update the state of the machine in program order even though they may complete execution out of program order (they may even begin execution out of program order). In order to achieve this, a mechanism must be provided to reorder the instructions after they have completed execution. This reorder mechanism could be a simple *reorder buffer* or a more complex reorder buffer with *bypass logic*, a *history buffer* or a *future file* [5].

The simple reorder buffer allows instructions to finish execution out of order but updates the state of the machine (registers, memory, etc.), i.e., *commits* the instructions in the order that the instructions were present in the program, thereby assuring that a precise state of the machine is recoverable at any time. By forcing an ordering of commitment amongst the instructions, the reorder buffer aggravates data dependencies - the value of a register cannot be read till it has been updated by the reorder buffer, even though the instruction that computed a value for the register may have already completed and the new value is in the reorder buffer. If bypass logic is associated with the reorder buffer, an instruction does not have to wait for the reorder buffer to update a source register; it can fetch the value from the reorder buffer (if it is available) and can issue. With a bypass mechanism, the issue rate of the machine is not degraded considerably if the size of the buffer is reasonably large [5]. However, a bypass mechanism is expensive to implement since it requires a search capability and additional data paths for each buffer entry. A history buffer has the same performance as a reorder buffer with bypass logic. It does not need bypass logic but the register file needs another read port. A future file achieves the same performance as a reorder buffer with bypass logic at the expense of duplicating the entire register file.

5. MERGING DEPENDENCY RESOLUTION AND PRECISE INTERRUPTS

We note that the RSTU of section 3.2.3 can be modified to behave like a reorder buffer if it is forced to update the state of the machine in the order that the instructions are encountered. This is easily accomplished by managing the RSTU as a queue. Therefore, all that we have to do to implement precise interrupts in an architecture with a RSTU is to manage the RSTU like a queue. We call the modified logic the *Register Update Unit (RUU)*. The RUU is essentially the RSTU constrained to commit instructions in the order that the instructions were received by the decode and issue logic (and consequently by the RUU). The functional units remain unchanged. The modified architecture that uses a RUU to execute instructions out of program order and to ensure a precise state of the machine is given in Figure 5.

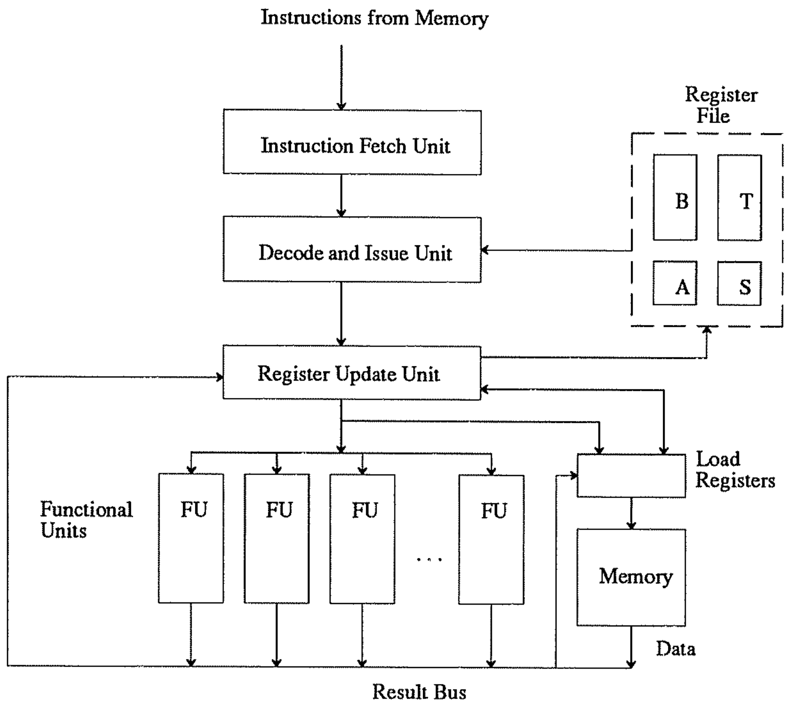

Figure 5. The Model Architecture with a RUU

5.1. The Register Update Unit (RUU)

The RUU performs four major functions: (i) it determines which instruction should be issued to the functional units for execution, reserves the result bus and dispatches an instruction to the functional unit, (ii) it determines if an instruction can commit, i.e., update the state of the machine, (iii) it monitors the result bus to resolve dependencies and (iv) it provides tags to and accepts new instructions from the decode and issue logic. The RUU is managed like a queue using RUU_Head and RUU_Tail pointers. RUU slots that do not lie between RUU_Head and RUU_Tail are free. If RUU_Head = RUU_Tail, the RUU is full and cannot accept any more instructions from the decode and issue logic. RUU_Tail points to the slot that will be used by the decode and issue logic and RUU_Head points to the next instruction that must commit to ensure a precise state. In designing the RUU, we keep in mind that (i) it should not involve a large amount of hardware and (ii) it should not affect the clock speed to an intolerable extent.

Managing the RSTU like a queue has a very important side effect - the logic for obtaining tags for source operands and generating tags for destination operands is greatly simplified. Recall that in the RSTU, the issue logic needed to search the RSTU associatively to obtain the correct tag for the source operand and to update the latest copy field for the destination register. If multiple instances of the same destination register are disallowed, i.e., instruction issue is blocked if the destination register is busy, no associative logic is necessary since the register number itself serves as the tag. An *instance* of a register is a new copy of the register. By providing multiple instances of a destination register, the architecture can process several instructions with the same destination register simultaneously. Disallowing multiple instances of a destination register can degrade performance [17]. As noted in [13], it is possible to eliminate the associative search and use a counter to provide multiple instances and source operand tags for each register *if we can guarantee that results return to the registers in order.* This is precisely the situation in the RUU. The implementation of precise interrupts, therefore, simplifies the out-of-order instruction issue mechanism.

The scheme we use to provide multiple instances of a destination register and to provide source operand tags associates 2 n-bit counters (control bits) with each register in the register file (this includes the B and T register files). There is no busy bit. The counters, the *Number of Instances (NI)* and the *Latest Instance (LI)*, represent the number of instances of a register in the RUU and the number of the latest instance, respectively. When an instruction with a destination register Ri is issued to the RUU, both NI and LI associated with Ri are incremented. LI is incremented modulo n. Up to 2^n-1 instances of a register can be present in the RUU at any time; issue is blocked if NI for a destination register is 2^n-1. When an instruction leaves the RUU and updates the value of Ri, the associated NI is decremented. A register is free if NI = 0, i.e., there is no instruction in the RUU that is going to write into the register.

When an instruction is decoded, the issue logic requests an entry in the RUU. If no free entry is available, i.e., the RUU is full, instruction issue is blocked. If an entry is available, the issue logic obtains the position of the entry (using the RUU_Tail pointer). It then forwards the contents of the source registers (if they are available) or a register identifier (the register number appended with the LI counter to be used as a tag) to the selected reservation station in the RUU. The LI and NI counters for the destination register are updated and the new identifier for the destination register forwarded to the RUU.

The register tag sent to the RUU consists of the register number Ri appended with the LI counter. This guarantees that future instructions access the latest instance, i.e., obtain the latest copy of the register contents and that instructions already present in the RUU get the correct version of the data. In our experiments, each of these counters was 3 bits wide. This allowed upto 7 instantiations of a destination register. A 3-bit counter ensured that, for our benchmark programs, an instruction never blocked in the decode and issue stage because an instance of a register was unavailable. Since we had a total of 144 registers, the tag field was 11 (8+3) bits wide.

Each source operand field in the RUU has a ready bit, a tag sub-field and a content sub-field. If the operand is not ready, the tag sub-field monitors the result bus for a matching tag. If a match is detected, the data on the bus is gated into the content field. There is no need for a Latest Copy field in the RUU and no associative search logic is needed in the RUU to generate and maintain the tags. However, associative comparison logic is still needed for all the reservation stations in the RUU so that they can gate in the value of source operands when available. Each entry in the RUU is as follows:

Source Operand 1			Source Operand 2			Destination	
Ready	Tag	Content	Ready	Tag	Content	Register,LI	Content

Dispatched	Functional Unit	Executed	Program Counter
Yes/No	Unit Number	Yes/No	Content

The Dispatched field indicates if the instruction has been dispatched for execution to the functional unit specified in the Functional Unit field. The Executed field indicates if the instruction has finished execution and is ready to update the register file. The Program Counter field is needed for the implementation of precise interrupts [5]. For the sake of brevity, we have omitted the details of extra information that must be carried around with each instruction. The details of such information are obvious.

When the operands of an instruction in the RUU are ready, the instruction can issue to the functional units. Priority is first given to load/store instructions and then to an instruction which entered the RUU earlier. The RUU reserves the result bus when it issues an instruction to the functional units. When the instruction at the head of the RUU has finished execution, its results are forwarded to the register file. The associated NI counter is decremented. As is obvious from the above discussion, each of the tasks of the RUU can be carried out in parallel unless the hardware configuration is such that resource conflicts occur.

Instructions that interact with the memory, i.e., load/store instructions, are handled as in section 3.2.1.2. The reservation stations for the memory are provided by the RUU. Note that decode and issue unit logic needs to search the load registers associatively for memory addresses. However, the hardware needed for this comparison is not very great for a small number of load registers. In our simulations, we used 6 load registers though 4 were sufficient for most cases.

6. EVALUATION OF THE RUU

In order to evaluate the effectiveness of the RUU, we simulated three RUU organizations, (i) a RUU with bypass logic for source operand values, (ii) a RUU without bypass logic and (iii) a RUU with a limited bypass logic. The results presented in this section differ slightly from results presented previously [19]. The main reason for the difference is a different pipeline structure and a different issue mechanism for load and store instructions.

6.1. The RUU with Bypass Logic

Recall that the RUU forces the results to return to the registers in program order. In doing so, it aggravates data dependencies. Such a degradation could be eliminated if bypass logic for source operands was provided in some form. The simplest form could be associative comparison hardware with the destination field of each RUU entry. If a source operand for instruction I_j is provided by I_i and the destination operand of I_i is ready in the RUU, the operand can be read from the RUU and I_j allowed to proceed with execution. Note that the history buffer and the future file[5] are alternate forms for bypass logic. The relative speedups (compared to the simple instruction issue mechanism of Table 1) and the corresponding instruction issue rate for different sizes of a RUU with bypass logic are presented in Table 4.

Table 4: Relative Speedups and Issue Rate for a RUU with Bypass Logic

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.853	0.374
4	0.937	0.411
6	1.077	0.472
8	1.246	0.546
10	1.378	0.604
12	1.502	0.658
15	1.597	0.700
20	1.668	0.731
25	1.713	0.751
30	1.755	0.769
40	1.780	0.780
50	1.786	0.783

The results of Table 4 are quite promising. An RUU with a reasonable number of entries (10-12) not only speeds up execution but also provides precise interrupts. Moreover, for somewhat larger RUU sizes, the RUU is able to achieve a speedup that is quite similar to the RSTU. Note that the RSTU was not constrained to implement precise interrupts.

6.2. The RUU without Bypass Logic

Since bypass logic is expensive to implement, we decided to evaluate a RUU without any bypass logic. Before we present the results, let us see where bypass logic is helpful.

Consider an instruction I_j that uses the result of a previous instruction I_i. Recall that the reservation stations associated with the RUU already have the capability to monitor the result bus. Therefore, if I_i completes execution *after* I_j is issued to the RUU, I_j can gate in the result from I_i when it appears on the result bus. In this case, no bypass logic is needed. If I_i has completed execution but has not committed, i.e., updated the register file, when I_i is issued to the RUU, then we must extend the monitoring capabilities of the reservation stations to monitor both the result bus and the bus between the RUU and the register file. This is necessary to prevent deadlock since I_j's dependency on I_i would never be resolved if the bus between the RUU and the register file is not monitored (or bypass logic provided). Extending the monitoring capabilities of the reservation stations can be accomplished without a substantial increase in hardware.

Bypass logic is helpful only in the cases where I_i has completed execution when I_j is issued. Rather that providing bypass logic for this case, we wait for the result of I_i to come out on the bus between the RUU and the register file in order to resolve I_j's dependency on I_i. If I_j is issued to the RUU before I_i completes, I_j's dependency on I_i can be resolved when I_i's result appears on the result bus.

Table 5: Relative Speedups and Issue Rate for a RUU without Bypass Logic

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.825	0.361
4	0.906	0.397
6	1.030	0.451
8	1.070	0.469
10	1.102	0.483
12	1.190	0.522
15	1.212	0.531
20	1.291	0.566
25	1.337	0.586
30	1.365	0.598
40	1.447	0.634
50	1.475	0.646

Table 5 presents the relative speedups and instruction issue rates for a RUU without bypass logic. From Table 5, we see that a RUU without any bypass logic at all is still able to achieve a substantial increase in speed over a simple instruction issue mechanism and implement precise interrupts at the same time. The speedup however, is not as impressive as the speedup obtained if bypass logic were used. The difference arises mainly because of the ordering of code in the loops. Let us illustrate the problem with an example.

Consider the following section of code:

```
I_i     A2 <- A1 + A3
             .
             .
I_j      A0 <- A2 + 1
             .
             .
I_k     JAM  loopstart
```

Conventional compilation techniques try and increase the distance between instructions I_i and I_j and instructions I_j and I_k so that when instructions I_j and I_k reach the issue stage, their respective operands are ready. Such an increase in dependency distance is in fact harmful to a RUU without bypass logic. If I_j was issued sufficiently before I_k and completed execution before I_k reached the decode and issue stage, I_k would be forced to wait till I_j left the RUU. If, on the other hand, I_j was issued soon before I_k, I_k could resolve its dependency on I_j when the result of I_j was available on the functional unit result bus. In our simulations, no attempt was made to reorder the code for this special case.

6.3. The RUU with Limited Bypass Logic

Because of the problem illustrated above, we found that branch instructions were blocked for a long period of time in the decode and issue stage since the contents of the A0 register could not be read from the RUU (or were unavailable because of a dependency chain aggravated as above). The branch instruction has to wait in the decode and issue unit until the value of A0 appears on a bus. In order to eliminate this problem, we duplicated the A register file, effectively creating a limited bypass path for the A registers. The duplicate A register file acts as a future file for the A registers. The entire A register file (8 registers) was duplicated to prevent the unnecessary increase in the length of the dependency chain that affects the conditional branch instruction. All other functions are as before. Specifically, there is only 1 copy of the B, S and T register files and there is no bypass logic in the RUU. As functions that affect the A registers are completed and appear on the result bus, the result is forwarded to the RUU and also the the A future file. The architectural register file contains a valid copy of registers at all time for recovering a precise state. Instructions that use A registers as source operands, fetch the data from the A future file, if it is available, and proceed. The results for a RUU with limited bypass logic is presented in Table 6.

Table 6: Relative Speedups and Issue Rate for a RUU with Limited Bypass Logic

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.846	0.371
4	0.928	0.407
6	1.064	0.466
8	1.115	0.489
10	1.266	0.555
12	1.303	0.571
15	1.420	0.622
20	1.448	0.635
25	1.484	0.651
30	1.505	0.660
40	1.518	0.665
50	1.547	0.678

An RUU with limited bypass logic is able to overcome a significant portion of the performance penalty paid for eliminating bypass logic especially for small RUU sizes. For larger RUU sizes, however, the performance is not as good. This is because instructions that transfer data from a B register to an A register are still held up in the RUU (no bypass logic for the B register file). Since the destination A register of such transfer instructions eventually affects the branch condition (most branch instructions in the benchmark programs tested the value of the A0 register), instruction issue is blocked for longer periods of time. We are confident that the performance of a RUU without bypass logic and a RUU with limited bypass logic could be improved considerably and would come close to the speedups with bypass logic if the code was modified accordingly.

7. BRANCH PENALTY AND CONDITIONAL INSTRUCTIONS

As mentioned earlier, the performance degradation due to branches can be reduced by conditionally executing instructions from a predicted branch path. Several architectures employ this approach [3, 9, 11, 20]. To allow conditional execution of instructions, a hardware mechanism is needed that would allow the machine to recover from an incorrect branch prediction.

The RUU provides a very powerful mechanism for *nullifying* instructions, be the instructions valid instructions or instructions that executed in a conditional mode. Valid instructions may be nullified because of an interrupt caused by a previous instruction; conditionally executed instructions may be nullified if they are from an incorrect execution path. Therefore, the conditional execution of instructions with a RUU is very easy. If the decode and issue unit predicts the outcome of branches and actually executes instructions from a predicted path in a conditional mode, recovery from incorrect branch predictions can be achieved very easily without duplicating the register file. We can identify such instructions through the use of an additional field in the RUU and prevent them from being committed until they are proven to be from a correct path. Furthermore, there is no hard limit to the number of branches that can be predicted; the RUU can provide multiple instances of a register for the different paths. Extending the RUU to accommodate branch prediction and conditional execution is a topic for future research.

8. CONCLUSION

In this paper, we have combined the issues of hardware dependency-resolution and implementation of precise interrupts. We devised a scheme that can resolve dependencies and thereby allow out-of-order instruction execution without associating tag-matching hardware with each register. Such a scheme can, therefore, be used even in the presence of a large number of registers without a substantial hardware cost. Then we extended the scheme to incorporate precise interrupts. The precise interrupt and the dependency-resolution mechanisms mutually aid and simplify each other. We evaluated the performance of the resulting hardware that allows out-of-order instruction execution and also implements precise interrupts using several Livermore loops as the benchmark. The results are quite encouraging. The combined mechanism, called the RUU, is able to implement precise interrupts and is able to achieve a significant performance improvement over a simple instruction issue mechanism without a substantial cost in hardware. We noted that this mechanism can easily be extended to support conditional execution of instructions from a predicted branch path.

Acknowledgments

This work was supported in part by the University of Wisconsin Graduate Research Committee. The author would like to thank Jim Goodman, Andy Pleszkun, Jim Smith for their useful discussions during this research. The author would also like to thank Sriram Vajapeyam for his help with the simulations.

References

[1] R. M Russel, "The CRAY-1 Computer System," *CACM*, vol. 21, pp. 63-72, January 1978.

[2] "CDC Cyber 200 Model 205 Computer System Hardware Reference Manual," *Control Data Corporation, Arden Hills, MN*, 1981.

[3] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, pp. 8-24, January 1967.

[4] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

[5] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Annual Symposium on Computer Architecture*, pp. 36-44, June 1985.

[6] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[7] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, vol. 17, pp. 6-22, January 1984.

[8] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," *Proc. 13th Annual Symposium on Computer Architecture*, pp. 386-395, June 1986.

[9] A. Pleszkun, J. Goodman, W. C. Hsu, R. Joersz, G. Bier, P. Woest, and P. Schecter, "WISQ: A Restartable Architecture Using Queues," in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 290-299, June, 1987.

[10] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," in *Proc. 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, pp. 396-304, June, 1986.

[11] P. Chow and M. Horowitz, "Architectural Tradeoffs in the Design of MIPS-X," in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 300-308, June, 1987.

[12] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill, "Hardware/Software Tradeoffs for Increased Performance," *Proc. Int. Symp. on Arch. Support for Prog. Lang. and Operating Sys.*, pp. 2-11, March 1982.

[13] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, pp. 25-33, January 1967.

[14] *CRAY-1 Computer Systems, Hardware Reference Manual*. Chippewa Falls, WI: Cray Research, Inc., 1982.

[15] N. Pang and J. E. Smith, "CRAY-1 Simulation Tools," Tech. Report ECE-83-11, University of Wisconsin-Madison, Dec. 1983.

[16] F. H. McMahon, *FORTRAN CPU Performance Analysis*. Lawrence Livermore Laboratories, 1972.

[17] S. Weiss and J. E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," *Proc. 11th Annual Symposium on Computer Architecture*, pp. 110-118, June 1984.

[18] R. D. Acosta, J. Kjelstrup, and H. C. Torng, ''An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors,'' *IEEE Trans. on Computers*, vol. C-35, pp. 815-828, September 1986.

[19] G. S. Sohi and S. Vajapeyam, ''Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors,'' in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 27-36, June, 1987.

[20] W. Hwu and Y. N Patt, ''HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality,'' *Proc. 13th Annual Symposium on Computer Architecture*, pp. 297-307, June 1986.