INSTRUCTION ISSUE LOGIC FOR PIPELINED SUPERCOMPUTERS

Shlomo Weiss
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, WI 53706

Abstract

Basic principles and design tradeoffs for control of pipelined processors are first discussed. We concentrate on register-register architectures like the CRAY-1 where pipeline control logic is localized to one or two pipeline stages and is referred to as "instruction issue logic". Design tradeoffs are explored by giving designs for a variety of instruction issue methods that represent a range of complexity and sophistication. These vary from the original CRAY-1 issue logic to a version of Tomasulo's algorithm, first used in the IBM 360/91 floating point unit. Also studied are Thornton's "scoreboard" algorithm used on the CDC 6600 and an algorithm we have devised. To provide a standard for comparison, all the issue methods are used to implement the CRAY-1 scalar architecture. Then, using a simulation model and the Lawrence Livermore Loops compiled with the CRAY FORTRAN compiler, performance results for the various issue methods are given and discussed.

1. Introduction

Although modern supercomputers are closely associated with high speed vector operation, it is widely recognized that scalar operation is at least of equal importance, and pipelining [KOGG81] is the predominant technique for achieving high scalar performance. In a pipelined computer, instruction processing is broken into segments and processing proceeds in an assembly line fashion with the execution of several instructions being overlapped. Because of data and control dependencies in a scalar instruction stream, interlock logic is placed between critical pipeline segments to control instruction flow through the pipe. In an register-register architecture like the CDC 6600 [THOR70], the CDC 7600 [BONS69], and the CRAY-1 [CRAY77, CRAY79,RUSS78], most of the interlock logic is localized to one segment early in the pipeline and is referred to as "instruction issue" logic.

It is the purpose of this paper to highlight some of the tradeoffs that affect pipeline control, with particular emphasis on instruction issue logic. The primary vehicle for this discussion is a simulation study of different instruction issue methods with varying degrees of complexity. These range from the simple and straightforward as in the CRAY-1 to the complex and sophisticated as in the CDC 6600 and the IBM 360/91 floating point unit [TOMA67]. Each is used to implement the CRAY-1 scalar architecture, and each implementation is simulated using the 14 Lawrence Livermore Loops [MCMA72] as compiled by the Cray Research FORTRAN compiler (CFT).

1.1. Tradeoffs

We begin with a discussion of design tradeoffs that centers on four principle issues:

(1) clock period,

(2) instruction scheduling,

(3) issue logic complexity, and

(4) hardware cost, debugging, and maintenance.

Each of these issues will be discussed in turn.

Clock Period. In a pipelined computer, there are a number of segments containing combinational logic with latches separating successive segments. All the latches are synchronized by the same clock, and the pipeline is capable of initiating a new instruction every clock period. Hence, under ideal conditions, i.e. no dependencies or resource conflicts, pipeline performance is directly related to the period of the clock used to synchronize the pipe. Even with data dependencies and resource conflicts, there is a high correlation between performance and clock period.

Historically, pipelined supercomputers have had shorter clock periods than other computers. This is in part due to the use of the fastest available logic technologies, but it is also due to designs that minimize logic levels between successive latches.

Scheduling of Instructions. Performance of a pipelined processor depends greatly on the order of the instructions in the instruction stream. If consecutive instructions have data and control dependencies and contend for resources, then "holes" in the pipeline will develop and performance will suffer. To improve performance, it is often possible to arrange the code, or schedule it, so that dependencies and resource conflicts are minimized. Registers can also be allocated so that register conflicts are reduced (register conflicts caused by data dependencies can not be eliminated in this way, however). Because of their close relationship, in the remainder of the paper we will group code scheduling and register allocation together and refer to them collectively as "code scheduling".

There are two different ways that code scheduling can be done. First, it can be done at compile time by the software. We refer to this as "static" scheduling because it does not change as the program runs. Second, it can be done by the hardware at run time. We refer to this as "dynamic" scheduling. These two methods are not mutually exclusive.

Most compilers for pipelined processors do some form of static scheduling to avoid dependencies. This adds a new dimension to the optimization problems faced by a compiler, and occasionally a programmer will hand code inner loops in assembly language to arrive at a better schedule than a compiler can provide.

0194-7111/84/0000/0110$01.00©1984 IEEE

Issue Logic Complexity. By using complex issue logic, dynamic scheduling of instructions can be achieved. This allows instructions to begin execution "out-of-order" with respect to the compiled code sequence. This has two advantages. First, it relieves some the burden on the compiler to generate a good schedule. That is, performance is not as dependent on the quality of the compiled code. Second, dynamic scheduling at issue time can take advantage of dependency information that is not available to the compiler when it does static scheduling. Complex issue logic does require longer control paths, however, which can lead to a longer clock period.

Hardware Cost, Debugging, and Maintenance. Complex issue methods lead to additional hardware cost. More logic is needed, and design time is increased. Complex control logic is also more expensive to debug and maintain. These problems are aggravated by issue methods that dynamically schedule code because it may be difficult to reproduce exact issue sequences.

1.2. Historical Perspective

It is interesting to review the way the above tradeoffs have been dealt with historically. In the late 1950's and early 1960's there was rapid movement toward increasingly complex issue methods. Important milestones were achieved by STRETCH [BUCH62] in 1961 and the CDC 6600 in 1964. Probably the most sophisticated issue logic used to date is in the IBM 360/91 [ANDE67], shipped in 1967. After this first rush toward more and more complex methods, there was a retreat toward simpler instruction issue methods that are still in use today. At CDC, the 7600 was designed to issue instructions in strict program sequence with no dynamic scheduling. The clock period, however, was very fast, even by today's standards. The more recent CRAY-1 and CRAY-XMP [CRAY82] differ very little from the CDC7600 in the way they handle scalar instructions. The CDC CYBER205 [CDC81] scalar unit is also very similar. At IBM, and later at Amdahl Corp., pipelined implementations of the 360/370 architecture following the 360/91 have issued instructions strictly in order.

As for the future, both the debug/maintenance problem and the hardware cost problem may be significantly alleviated by using VLSI where logic is much less expensive and where replaceable parts are such that fault isolation does not need to be as precise as with SSI. In addition, there is a trend toward moving software problems into hardware, and code scheduling seems to be a candidate. Consequently, tradeoffs are shifting and instruction issue logic that dynamically schedules code deserves renewed study.

1.3. Paper Overview

The tradeoffs just discussed lead to a spectrum of instruction issue algorithms. Through simulation we can look at the performance gains that are made possible by dynamic code scheduling. Other issues like clock period and hardware cost and maintenance are more difficult and require detailed design and construction to make quantitative assessments. In this paper, we do discuss the control functions that need to be implemented to facilitate qualitative judgements. Section 2 examines one endpoint of the spectrum: the CRAY-1. The CRAY-1 uses simple issue logic with a fast clock and static code scheduling only. Section 3 examines the other endpoint of the spectrum: Tomasulo's algorithm. Tomasulo's algorithm is capable of considerable dynamic code scheduling via a complex issue mechanism. Sections 4 and 5 then discuss two intermediate points. The first is a variation of Thornton's "scoreboard" algorithm used in the CDC 6600. Thornton's algorithm is also used to implement the CRAY-1 scalar architecture. The second is an algorithm we have devised to allow dynamic scheduling while doing away with some of the associative compares required by the other methods that perform dynamic scheduling. Each of the four CRAY-1 implementations is simulated over the same set of benchmarks to allow performance comparisons. Section 6 contains a further discussion on the relationship between software and hardware code scheduling in pipelined processors, and Section 7 contains conclusions.

2. The CRAY-1 Architecture and Instruction Issue Algorithm

2.1. Overview of the CRAY-1

The CRAY-1 architecture and organization are used throughout this paper as a basis for comparison. The CRAY-1 scalar architecture is shown in Figure 1. It consists of two sets of registers and functional units for (1) address processing and (2) scalar processing. The address registers are partitioned into two levels: eight A registers and sixty four B registers. The integer add and multiply functional units are dedicated to address processing. Similarly, the scalar registers are partitioned into two levels: eight S registers and sixty four T registers. The B and T register files may be used as a programmer-manipulated data cache, although this feature is largely unused by the CFT compiler. Four functional units are used exclusively for scalar processing. In addition, three floating point functional units are shared with the vector processing section (vector registers are not shown in Fig. 1).

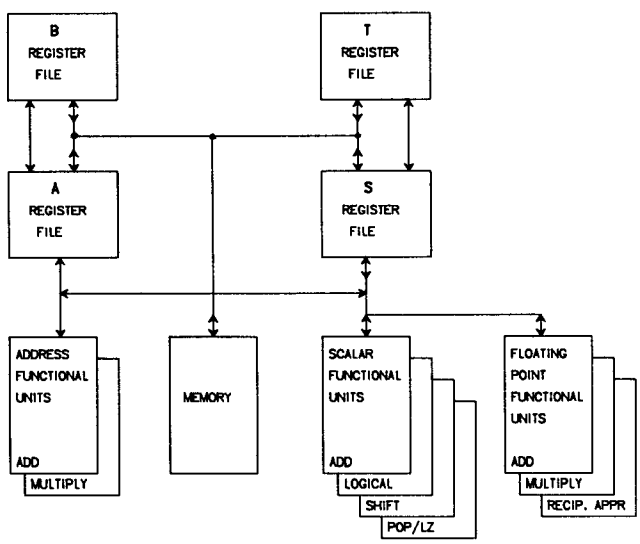


Figure 1 -- The CRAY-1 Scalar Architecture

The instruction set is designed for efficient pipeline processing. Being a register-register architecture, only load and store instructions can access memory. The rest of instructions use operands from registers. Instructions for the B and T registers are restricted to memory access and copies to and from register files A and S, respectively.

The information flow is from memory to registers A (S), or to the intermediate registers B (T). From file A (S) data is send to the functional units, from which it returns to file A (S). Then data can be further processed by functional units, stored into memory or saved in file B (T). Block transfers of operands between memory and registers B and T are also available, thus reducing the number of memory access instructions.

2.2. CRAY-1 Issue Logic

Instructions are fetched from instruction buffers at the rate of one parcel (16 bits) per clock period. Individual instructions are either one or two parcels long. After a clock period is spent for instruction decoding, the issue logic checks interlocks. If there is any conflict, issue is blocked until the conflict condition goes away.

For scalar instructions, the following are the primary interlock checks made at the time of instruction issue:

(1) registers; both the source and destination registers must not be reserved;

(2) result bus; the A and S register files have one bus each over which data can be written into the files. Based on the completion time of the particular instruction, a check is made to determine if the bus will be available at the clock period when the instruction completes.

(3) functional unit; due to vector instructions, a functional unit may be busy when a scalar instruction wishes to use it. Since we are considering scalar performance only, this type of conflict will not occur. The memory system can also be viewed as a functional unit; it can occasionally become busy due to a memory bank conflict, but for scalar code this is a very infrequent occurrence and does not affect performance in any appreciable way.

If all its interlocks pass, an instruction issues and causes the following to take place.

(1) The destination register is reserved; this reservation is removed only when the instruction completes.

(2) The result bus is reserved for the clock period when the instruction completes.

Memory accesses have one further interlock to be checked: memory bank busy. This must be delayed until the indexing register is read and the effective address is computed. Hence, the bank busy check is performed two clock periods after a load or store instruction issues. If the bank happens to be busy, the memory "functional unit" is busied, and no further loads or stores can be issued. Because all loads and stores are two parcels long, they can issue at a maximum rate of one every two clock periods. This means that a bank busy blockage catches a subsequent load or store before it is issued. After a load instruction passes the bank busy check, it places its reservation for the appropriate result bus. A load can only conflict for the bus with a previously issued reciprocal approximation instruction, so the additional interlocking done at that point is minimal.

2.3. CRAY-1 Performance

In this paper performance is measured by simulating the first 14 Lawrence Livermore Loops. These are excerpts from large FORTRAN programs that have been judged to provide a good measure of large scale computer performance. The loops were compiled using the CFT compiler, and instruction trace tapes were generated. These were then simulated with a performance simulator written in C, running on a VAX11/780. With bank busies and instruction buffer misses modeled, the simulator agrees exactly with actual CRAY-1 timings, except when there is a difference in the way a loop fits into the instruction buffers. This particular difference is a function of where the loader chooses to place a program in memory, and for practical use has to be viewed as a nondeterminism.

Since we are interested in scalar performance, the CFT compiler was run with the "vectorizer" turned off so that no vector instructions were produced. When the vectorizer is on, half of the 14 loops contain a substantial amount of vector code, and half remain scalar.

Loop	# instructions executed	# clock cycles
1	7217	18046
2	8448	18918
3	14015	38039
4	9783	22198
5	8347	21707
6	9350	23045
7	4573	10361
8	4031	7841
9	4918	10146
10	4412	10230
11	12002	30011
12	11999	29999
13	8846	18858
14	9915	22391

Table 1 -- CRAY-1 Execution times for the 14 Lawrence Livermore Loops - 1 parcel instructions are issued in 1 cycle, 2 parcel instructions in 2 cycles.

For the simulations reported here, we have made the following simplifications:

(1) There are no memory bank conflicts.

(2) All loops fit in the instruction buffers.

One reason for this simplification was to simplify the simulator design for the alternative CRAY-1 issue methods to be given later; as mentioned earlier our original CRAY-1 simulator is capable of modeling bank conflicts and instruction buffers. Also, this allows us to concentrate on the performance differences caused by issue logic and to filter the "noise" introduced by other factors (e.g. instruction buffer crossings). Table 1 shows the scalar performance of the CRAY-1 for the first 14 Lawrence Livermore Loops.

3. Tomasulo's Algorithm

The CRAY-1 forces instructions to issue strictly in program order. If an instruction is blocked from issuing due to a conflict, all instructions following it in the instruction stream are also blocked, even if they have no conflicts. In contrast, the scheme in this section allows instructions to begin execution out of program order. It is a variation of the instruction issue algorithm first presented in [TOMA67]. Although the original algorithm was devised for the floating point unit of the IBM 360/91, we show how it can be adapted and extended to control the entire pipeline structure of a CRAY-1 implementation.

Figure 2 illustrates the essential elements of a tag-based mechanism for issue of instructions out-of-order. Fig. 3 illustrates the full CRAY-1 implementation.

Each register in the A and S register files is augmented by a *ready* bit (R) and a *tag* field. Associated with each functional unit is a small number of *reservation stations*. Each reservation station can store a pair of operands; each operand has its own tag field and ready bit. A reservation station also holds a *destination tag* (DTG). When an instruction is issued, a new tag is stored into DTG (see Fig. 2).

New destination tags are assigned from a "tag pool" that consists of some finite set of tags. These are associated with an instruction from the time the instruction is issued to a reservation station until the time it produces a result and completes. The tag is returned to the pool when an instruction finishes. In the original Tomasulo's algorithm, the tags were in 1-to-1 correspondence with the reservation stations. This particular way of assigning tags is not essential, however. Any method will work as long as tags are assigned and released to the pool as described above.

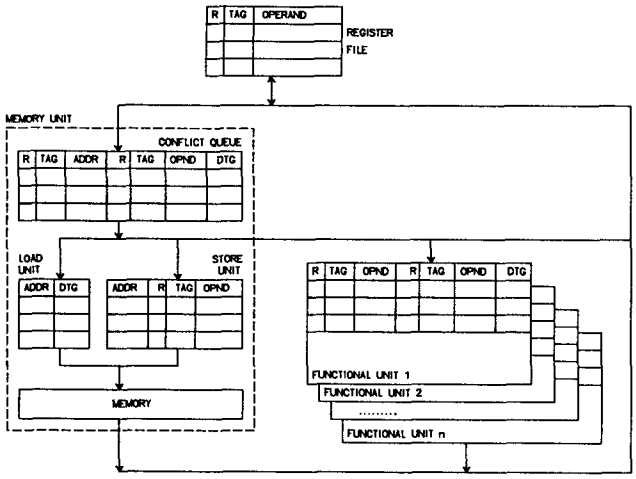


Figure 2 -- Tag based mechanism to issue out-of-order.

We treat the register files B and T as a unit, with one *busy* bit per file, since it is not practical to assign tags to so many registers. When one of these registers awaits an operand, the whole file is set to busy.

To facilitate transfer of operands between the register files, special copy units (AS, SA, AB, and ST) are introduced. These are treated as functional units, with reservation stations and execution time of one clock cycle. These reservation stations (and some others, e.g. reciprocal approximation) have only one operand.

The memory unit appears to the issue logic as a (somewhat more complex) functional unit. Instead of one set of reservation stations, the memory unit has three: *Load* reservation stations, *Store* reservation stations, and a *Conflict Queue*. When a new memory instruction I_i is issued, its effective address (if available) is checked against addresses in the Load and Store reservation stations. If there is a conflict with instruction I_j, I_i is issued and queued in the Conflict Queue. When I_j is eventually processed and the conflict disappears, I_i is transferred from the Conflict Queue to a Load or Store reservation station. If I_i uses an index register that is not ready, the effective address is unknown and there is no way to check for conflicts. In this case, I_i is stored in the Conflict Queue, where it waits for its index register to become ready.

Therefore, instructions from the Load and Store units can be processed asynchronously, since they never conflict with each other (no two instructions in these units have the same effective address). On the other hand, instructions from the Conflict Queue are processed in the order of arrival. This guarantees that two instructions with the same effective address are processed in the right order. The above mechanism takes care of any read after write, write after read or write after write hazards. The Conflict Queue is the only unit in the system in which instructions are strictly processed in the order of their arrival. This is a simpler mechanism that the one employed by the IBM 360/91 Storage System [BOLA67]. The latter has a similar queue for resolving memory conflicts, but instructions stored in this queue can be processed out of order; only two or more requests for a particular address are kept in sequence.

The tag mechanism described above allows decoded instructions to issue to functional units with little regard for dependencies. There are three conditions that must be checked before an instruction can be sent to a functional unit, however.

(1) The requested functional unit must have an available reservation station

(2) There must be an available tag from the tag pool. In Tomasulo's implementation, conditions 1 and 2 are equivalent.

(3) A source register being used by the instruction must not be loaded with a just-completed result during the same clock period as the instruction issues to a reservation station. This hazard condition is often neglected when discussing Tomasulo's algorithm. If it is not handled properly, the source register contents and the instruction that uses the source register will both be transferred during the same clock period. Because the instruction is not in the reservation station at the time of the register transfer, the register's contents will not be correctly sent to the reservation station. When this hazard condition is detected, instruction issue is held for one clock period.

In our simulations, each functional unit had 8 reservation stations. The reason we had a relatively large number of reservation stations was to monitor their usage; in fact most of them were not required. Few functional units need more than one or two reservation stations. Those that need more, usually because they wait for instructions with long latency, such as load or floating point multiply and add, would do very well with 4 reservation stations. Although in our scheme reservation stations are statically allocated to each functional unit, it is possible to reduce their number and optimize their usage by clustering them in a common pool and then allocating as needed.

When an instruction is issued, the following actions take place.

(1) The instruction's source register(s) contents are copied into the requested functional unit's reservation station.

(2) The instruction's source register(s) ready bits are copied into the reservation station.

(3) The instruction's source register(s) tag fields are copied into the reservation station.

(4) A tag allocated from the tag pool is placed in the result register's tag field (if there is a result), the register's ready bit is cleared, and the tag is written into the DTG field of the reservation station.

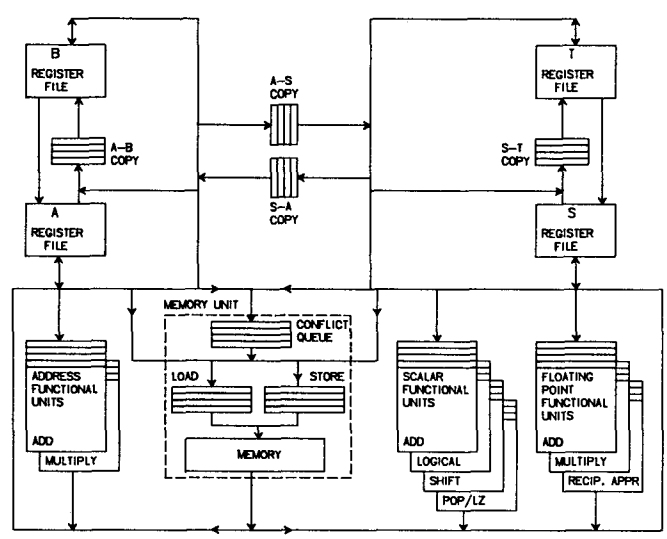


Figure 3 -- A modified CRAY-1 scalar architecture to issue instructions out-of-order.

Thus, if a source register's ready bit is set, the reservation station will hold a valid operand. Otherwise, it will hold a tag that identifies the expected operand.

In order for an instruction waiting in a reservation station to begin execution, the following must be satisfied.

(1) All its operands must be ready.

(2) It must gain access to the required functional unit; this may involve contention with other reservation stations belonging to the same functional unit that also have all operands ready.

(3) It must gain access to the result bus for the clock period when its result will be ready; again, this may involve contention with other instructions issuing to the same or any other functional unit that will complete at the same time.

When an instruction begins execution, it does the following.

(1) It releases its reservation station.

(2) It reserves the result bus for the clock period when it will complete. Reserving the bus in advance avoids the implementation problems of stopping the pipeline if the bus is busy. An alternative is to request the bus a short time before the end of execution and to provide buffering at the output of the functional unit [TOMA67].

(3) It copies its destination tag into the functional unit control pipeline because the destination tag must be attached to the result when the instruction completes.

When an either a load or functional unit instruction completes, its result and corresponding destination tag appear on the result bus. (In a practical implementation, the tag will probably precede the data by one clock period.) The data is stored in all the reservation stations and registers that have the ready bit clear and a tag that matches the tag of the result. Then the ready bit is set to signal a valid operand.

Because instruction issue takes place in two phases (the first moves an instruction to a reservation station and the second moves it on to the functional unit for execution) we assume that each phase takes a full clock period. In the CRAY-1 there is only a one clock period delay in moving an instruction to a functional unit. We recognize the one clock period difference in our simulation model, so that the minimum time for an instruction to complete is one clock period greater than in the CRAY-1. This takes into account some of the lost time due to the more complex control decisions that are required. The primary factor that may lead to longer control paths is the contention that takes place among the reservation stations for functional units and busses when more than one are simultaneously ready to initiate an instruction.

When branches are taken, issue is held for at least 5 clock cycles. Since branches test the contents of register A0 for the condition specified in the branch instruction, A0 should not be busy in the previous 2 cycles. These assumptions are in accord with the CRAY-1 implementation and the assumptions made to produce Table 1.

Loop	# clock cycles on CRAY-1	# clock cycles for Tomasulo's algorithm	speedup
1	18046	10838	1.67
2	18918	14102	1.34
3	38039	30017	1.27
4	22198	16534	1.34
5	21707	16925	1.28
6	23045	15042	1.53
7	10361	6513	1.59
8	7841	6780	1.16
9	10146	8238	1.23
10	10230	7421	1.38
11	30011	20006	1.50
12	29999	20000	1.50
13	18858	12314	1.53
14	22391	12780	1.75
total	281790	197510	1.43

Table 2 -- Performance with Tomasulo's Algorithm; One Parcel Issued per Clock Period

3.1. Performance Results

Table 2 shows the results of simulating the CRAY-1 implemented with Tomasulo's algorithm. The total speedup achieved was 1.43. We recognize that these are in a sense, "theoretical maximum speedups"; any lengthening of the clock period due to longer control paths will diminish this speedup.

Loop	# clock cycles 1 parcel/cp	# clock cycles 1 instr/cp	speedup
1	18046	17244	1.05
2	18918	17717	1.07
3	38039	37037	1.03
4	22198	21163	1.05
5	21707	20709	1.05
6	23045	22045	1.05
7	10361	10241	1.01
8	7841	6874	1.14
9	10146	9744	1.04
10	10230	9430	1.08
11	30011	28013	1.07
12	29999	28001	1.07
13	18858	17957	1.05
14	22391	22087	1.01
total	281790	268262	1.05

Table 3 -- Comparison of the 14 Lawrence Livermore Loops on CRAY-1: one instruction per clock period vs one parcel per clock period.

We noticed while doing the simulations that limiting instruction fetches to the maximum rate of one parcel per clock period appeared to be restricting performance. Hence, we modified the implementation so that a full instruction could be fetched and issued to a reservation station each clock period. This would be slightly more expensive to implement, but for Tomasulo's algorithm it gives a significant performance improvement over one parcel per clock period.

To keep comparisons fair, we went back and modified the original CRAY-1 simulation model so that it, too, could issue instructions at the higher rate. Table 3 gives the results of these simulations, and compares them with the one parcel per clock period results given earlier. Here, the performance improvement is small. This is an interesting result in itself, and shows the wisdom of opting for simpler instruction fetch logic in the original CRAY-1.

Because the higher instruction fetch rate does appear to alleviate a bottleneck that reduces the efficiency of Tomasulo's algorithm, we incorporated it into the model for the studies to follow, and use the CRAY-1 results of Table 3 (1 instruction per clock period) as a basis for further comparisons.

Table 4 shows simulation results for Tomasulo's algorithm with one instruction issued per clock period. The speedup achieved is in the range 1.23 - 2.02 (total 1.58). In three out of the four loops whose speedup was

Loop	# clock cycles on CRAY-1	# clock cycles for Tomasulo's algorithm	speedup
1	17244	8832	1.95
2	17717	11062	1.60
3	37037	28012	1.32
4	21163	15418	1.37
5	20709	16898	1.23
6	22045	12956	1.70
7	10241	5069	2.02
8	6874	5195	1.32
9	9744	6332	1.54
10	9430	5318	1.77
11	28013	17871	1.57
12	28001	16001	1.75
13	17957	9364	1.92
14	22087	11713	1.89
total	268262	170041	1.58

Table 4 -- Performance of Tomasulo's Algorithm; One Instruction Issued per Clock Period

less than 1.5 (i.e. loops 3, 4 and 8) issue was halted due to a busy T file. With an architecture that doesn't have the large number of registers the CRAY-1 has, it would be possible to use tags for all the registers, thus increasing the speedup of the above loops.

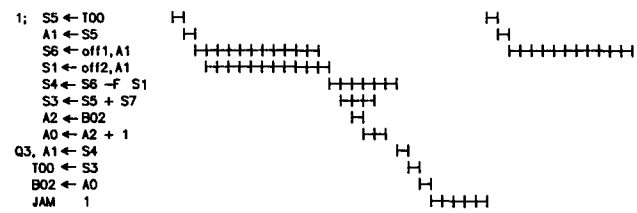
3.2. Example

Figure 4 shows a timing diagram of Tomasulo's algorithm compared with that of CRAY-1, both executing loop 12. On the left appear the instructions as generated by the CFT compiler . The timing for two consecutive loop iterations are shown next to each other. Each "|-|" represents one clock period. From the standpoint of issue logic, when store instructions are initiated they go to memory and are no longer considered. Therefore, stores are shown to execute only for the clock period they are initiated. Solid lines indicate that the respective instruction is in execution. For Tomasulo's algorithm, a dotted line shows that an instruction has been issued to a reservation station, and is waiting for operand(s).

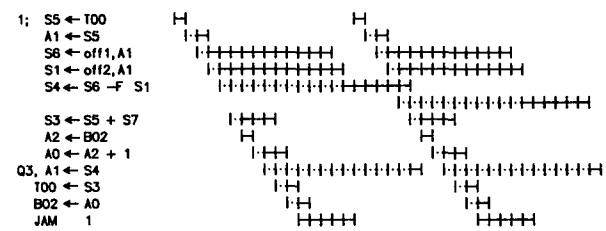
With the original CRAY-1 issue algorithm, all instructions in a loop must begin execution and a loop-terminating conditional branch instruction must complete before the next loop iteration can begin. With Tomasulo's algorithm, loop iterations can be "telescoped"; a second loop iteration can begin after all instructions have been sent to reservation stations and the conditional branch is completed. The instructions belonging to the first loop iteration do not necessarily need to begin execution. One could also view this as dynamic rescheduling of the branch instruction. Obviously, the branch instruction is one instruction that can not be moved earlier in the loop as would have to be the case with static rescheduling. The significant speedup of Tomasulo's algorithm for this loop (1.75, see Table 4) is due mainly to the overlap of the loading of registers S6 and S1 with the floating point difference (-F) which uses S6 and S1 as operands. Although -F cannot be executed until the operands return from memory, it can be issued to a reservation station; this allows following instructions to proceed. On the other hand, the CRAY-1 executes the load of S1 and the floating point difference strictly in order.

```
1:  S5 ← T00          .COPY T00 TO S5
    A1 ← S5           .COPY S5 TO A1
    S6 ← off1,A1      .LOAD S6 (ADDRESS INDEXED BY A1)
    S1 ← off2,A1      .LOAD S1 (ADDRESS INDEXED BY A1)
    S4 ← S6 -F S1     .FLOATING DIFFERENCE OF S6 AND S1 TO S4
    S3 ← S5 + S7      .INTEGER SUM OF S5 AND S7 TO S3
    A2 ← B02          .COPY B02 TO A2
    A0 ← A2 + 1       .INTEGER SUM OF A2 AND 1 TO A0
Q3, A1 ← S4           .STORE S4 (ADDRESS INDEXED BY A1)
   T00 ← S3           .COPY S3 TO T00
   B02 ← A0           .COPY A0 TO B02
   JAM   1            .BRANCH TO LOOP ENTRY
```

(a) LLL 12 in CRAY Assembly Language; arrows inserted for readability.



(b) The CRAY-1



(c) Tomasulo's algorithm

Figure 4 -- Timing Diagrams for Lawrence Livermore Loop 12.

One can see from the timing diagrams that with Tomasulo's algorithm only 2 reservation stations are used for more than 1 clock period. The store instruction needs a reservation station that is released a short period of time before the next store is issued. Another reservation station is used extensively by the floating point difference instruction.

4. Thornton's Algorithm

Tomasulo's algorithm leads to complex issue logic and may be quite expensive to implement. Therefore, in this section and in the next section, we consider ways to reduce the cost. One major cost is the associative hardware needed to match tags. When an operand and its attached tag appear on a bus, register files A and S and all the reservation stations have to be searched simultaneously. The operand is stored in any register or reservation station with a matching tag.

In this section, we implement the CRAY-1 scalar architecture with an issue method that is a derivative of Thornton's "scoreboard" algorithm used in the CDC6600. Here, control is more distributed (there is no global scoreboard), and reservation stations have been added to functional units. The primary difference between Thornton's algorithm and Tomasulo's is that instruction issue is halted when the destination of an instruction is a register that is busy. This simplifies the issue logic hardware in the following ways.

(1) The associative compare with the register file tags is eliminated.

(2) Tag allocation and de-allocation hardware is eliminated because the result register designator acts as the tag.

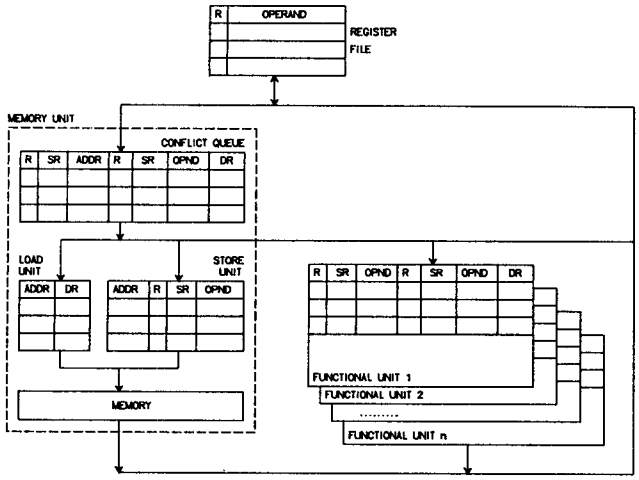


Figure 5 -- Thornton's issue logic.

Figure 5 illustrates the algorithm. Most reservation stations hold two operands (although some need only one, depending on the functional unit) and the address of the destination register (DR). Attached to each operand is the source register designator (SR) and a ready flag (R). Also, attached to each register in the register file is a ready bit (R). (These are the same as the reserved bits used in the original CRAY-1 control.)

The following operations are performed when an instruction is issued:

(1) A reservation station of the requested functional unit, if available, is reserved. Otherwise, issue is blocked.

(2) If a source register is ready, it is copied into the reservation station and the ready bit is set. Otherwise, the ready bit is cleared and the source register's designator is stored into the SR field.

The conditions for moving an instruction from a reservation station to begin execution are the same as given for Tomasulo's algorithm in the previous section. When a functional unit is finished with an instruction the result register designator is matched against all the SR fields in the reservation stations, and the result is written into the reservation stations where there is a match. The corresponding operand ready bits are then set. The result is also stored into the destination register, and it is set ready.

Although the above is derived from Thornton's original algorithm, there are some differences. The functional units of the CDC 6600 do not have reservation stations (one could say that they have reservation stations of depth 1 that are not able to hold operands, only control information). This imposes some additional restrictions. An instruction to be executed on a particular functional unit can be issued even if its source registers are not ready, but a second instruction requiring the same functional unit will block issue until the first one is done. On the other hand, with reservation stations, several instructions can wait for operands at the input of the functional unit. For example,

S1 <- S2 *F S3
S4 <- S5 *F S6

with the CDC 6600 scoreboard the second instruction will block, while with the algorithm here it will be issued.

Another restriction of the scoreboard is the following. In this example,

S1 <- S2 *F S3
S2 <- S4 + 1

we assume that S2 and S4 are ready, while S3 is not (e.g. it awaits an operand from memory). The second instruction will be completed before the first one, but on the CDC 6600 the result cannot be stored into S2 since it serves as a source register for the first instruction. Therefore, the add functional unit will remain busy until the first instruction completes as well. On the other hand, with the algorithm we have given, S2 has been copied into the floating point multiply reservation station when the first instruction was issued, and therefore the second instruction can complete before the first one.

4.1. Performance Results

We originally planned to simulate the scoreboard as designed by Thornton, but decided that it would lead to a more interesting comparison if multiple reservation stations were allowed. The results of the simulation are shown in Table 5. The total speedup is 1.28. We discuss ways this can be improved by static code scheduling in Section 6.

Loop	# clock cycles on CRAY-1	# clock cycles for Thornton's algorithm	speedup
1	17244	12434	1.39
2	17717	13500	1.31
3	37037	28020	1.32
4	21163	19578	1.08
5	20709	17030	1.22
6	22045	18370	1.20
7	10241	8671	1.18
8	6874	6381	1.08
9	9744	8634	1.13
10	9430	7820	1.21
11	28013	18001	1.56
12	28001	16003	1.75
13	17957	14870	1.21
14	22087	20273	1.09
total	268262	209585	1.28

Table 5 -- Performance of Thornton's Algorithm; One Instruction Issued per Clock Period

5. An Issue Method Using a Direct Tag Search

In this section we propose an alternative issue algorithm that is related to Tomasulo's algorithm, but which eliminates the need for associative tag comparison hardware in the reservation stations. This algorithm instead uses a direct tag search (DTS), and will be referred to as the "DTS" algorithm. The DTS algorithm imposes the restriction that a particular tag can be stored only in one reservation station. This is easily implemented by associating with each tag in the tag pool a *used* bit. Whenever a register that is not ready is accessed for the first time, its tag is copied to the respective reservation station and the used bit is set. A second attempt to use the same tag will block issue.

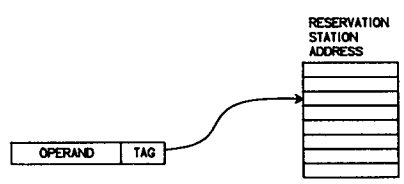


Figure 6 -- Tag Search Table for the DTS Algorithm.

The DTS algorithm allows implementation of the tag search mechanism by a table indexed by tags (Fig. 6), rather than associative hardware. For each tag there is one entry in the table that stores the address of a reservation station. The table is small since there are few tags (we used 5 bits for each tag; there are 32 tags).

5.1. Performance Results

A comparison of the results for the DTS algorithm with Tomasulo's algorithm reveals that for 9 out of the 14 loops the DTS algorithm achieves speedup similar to Tomasulo's algorithm. This shows that the restriction imposed by the DTS algorithm, namely that a tag can be in no more than one reservation station at a given time, has only a limited effect on performance. The reason is that the following pattern is quite common:

```
S1 <- off1,A1
S2 <- off2,A1
S3 <- S1 *F S2
S5 <- S3 +F S4
```

That is, two registers are loaded and are sent to a functional unit whose result is input to another functional unit, and so on. The DTS algorithm is able to process such code at full speed.

On the other hand, if a register is used as an input to a functional unit and at the same time has to be stored (in the memory, or temporarily in the T or B file), then the register, and its tag, have to be used twice, so the DTS algorithm blocks issue. Such cases account for the lower speedup of 5 out of 14 loops.

Loop	# clock cycles on CRAY-1	# clock cycles for the DTS algorithm	speedup
1	17244	8832	1.95
2	17717	11083	1.60
3	37037	28020	1.32
4	21163	20534	1.03
5	20709	17690	1.17
6	22045	17027	1.29
7	10241	5069	2.02
8	6874	5361	1.28
9	9744	6732	1.45
10	9430	8518	1.11
11	28013	17871	1.57
12	28001	16001	1.75
13	17957	14356	1.25
14	22087	17421	1.27
total	268262	194515	1.38

Table 6 -- Performance of DTS Issue Logic.

6. Further Comments on Code Scheduling

For all the various issue logic simulations, we used as input the object code generated by the CRAY-1 optimizing FORTRAN compiler, without any changes. Therefore, the level of code optimization and scheduling is realistic for the CRAY-1. Tomasulo's algorithm is less sensitive to the order of the instructions, since it does a great deal of dynamic scheduling. It is also capable of dynamic register re-allocation so it is not as susceptible to the compiler's register allocation method. On the other hand, static scheduling and register allocation has a significant impact on the performance of the DTS issue logic and Thornton's algorithm. Since we have not reorganized the code for the latter two schemes, many dependencies and resource conflicts that appear in the compiled code contribute to lower performance as compared with Tomasulo's algorithm.

Figure 7 illustrates an example, extracted from Lawrence Livermore Loop 4. The instruction "T02 <- S5" saves register S5 for use during the next pass through the loop. For the DTS issue logic, this instruction has to be blocked, since it attempts to use register S5 as a source for the second time (register S5 is not ready and was used for the first time by the previous instruction). However, neither S5 nor T02 are used before the branch instruction, so this interlock could be postponed by moving instruction "T02 <- S5" down, just before the branch. The 4 clock cycles thus saved, multiplied by the number of iterations through the loop, result in a 4% performance improvement.

```
      do 175 l=7,107,50
      lw = l
      do 4 j=30,870,5
      x(l-1) = x(l-1) - x(lw)*y(j)
    4 lw = lw + 1
      x(l-1) = y(5)*x(l-1)
  175 continue
```

(a) Fortran code for Lawrence Livermore Loop 4 (banded linear equations).

```
LOOP4: S6 <- T00
       S1 <- T01
       A3 <- S1
       A2 <- S6
       S3 <- off2,A3      ; x(lw)
       S5 <- off3,A2      ; y(j)
       S4 <- S5 *R S3
       S3 <- T02
       S5 <- S3 -F S4
       S4 <- S1 + S2
       A1 <- B02
       S3 <- S6 + S7
  off4,A7 <- S5    ; x(l-1)
     T02 <- S5
     T01 <- S4
      A0 <- A1 + 1
     T00 <- S3
     B02 <- A0
     JAM     LOOP4
```

(b) Compiled object code (the inner loop) extracted from Lawrence Livermore Loop 4.

Figure 7 -- Example of interlock for the DTS Issue Logic.

Figure 8 shows another example, extracted from loop 1. Registers S2, S6, S1, S3 and S4 are designated as destination registers for the first time in the upper half of the loop, and then for the second time, almost in the same order, in the lower half of the loop. The second usage of S2 (S2 <- S4 *R S6) as a destination register causes an immediate blockage for Thornton's algorithm, since S2 is still busy from the previous load instruction (S2 <- off1,A1). Any independent instruction inserted just before instruction "S2 <- S4 *R S6" will execute for free. There are 3 such instructions before the branch:

```
A2 <- B02
A0 <- A2 + 1
B02 <- A0
```

This simple rescheduling gives a performance improvement of 10.7%.

```
      q = 0.0
      do 1 k = 1,400
1     x(k) = q + y(k)*(r*z(k+10) + t*z(k+11))
```

(a) Fortran code for Lawrence Livermore Loop 1 (hydro excerpt).

```
LOOP1: S5 <- T00
       A1 <- S5
       S2 <- off1,A1      ; z(k+10)
       S6 <- off2,A1      ; z(k+11)
       S1 <- T02
       S3 <- S1 *R S2
       S4 <- T01
       S2 <- S4 *R  S6
       S6 <- off3,A1
       S1 <- S2 +F S3
       S4 <- S6 *R S1
       S3 <- S5 + S7
       A2 <- B02
       A0 <- A2 + 1
  off4,A1 <- S4
     T00 <- S3
     B02 <- A0
     JAM     LOOP1
```

(b) Assembly code extracted from Lawrence Livermore Loop 1.

Figure 8 -- Example of interlock for Thornton's Algorithm.

Since there is a large gap between Tomasulo's and Thornton's algorithms for loop 1 (1.95 vs 1.39), we tried to see if this gap could be closed by reallocating registers for Thornton's algorithm. The following code does the processing of loop 1, except for index and address calculations:

```
     S1 <- off1,A1
     S2 <- off2,A1
     S4 <- S1 *F S3
     S6 <- S2 *F S5
     S7 <- S4 +F S6
     S8 <- off3,A1
     S9 <- S7 *F S8
off4,A1 <- S9
```

Since registers are not re-used during the same pass through the loop, Thornton's algorithm would run as fast as Tomasulo's. But we need 9 scalar registers, more than available on the CRAY-1. The high speedup achieved by Tomasulo's algorithm demonstrates the importance of its ability to reallocate registers dynamically.

7. Summary and Conclusions

We have discussed design tradeoffs for control of pipelined processors. Performance of a pipelined processor depends greatly on its clock period and the order in which instructions are executed. Simple control schemes allow a short clock period and place the burden of code scheduling on the compiler. Complex control schemes generally require a longer clock period, but are less susceptible to the order of the instructions generated by the compiler. The latter are also able to take advantage of information only available at run time and "dynamically" reschedule the instructions. Additional factors to be considered are hardware cost, debugging and maintenance.

We have presented a quantitative measure of the speedup achievable by sophisticated issue logic schemes. The CRAY-1 scalar architecture is used as a basis for comparison. Simulation results of the 14 Lawrence Livermore Loops executed on 4 different issue logic mechanisms show the performance gain achievable by various degrees of issue logic complexity. Tomasulo's algorithm gives a total speedup of 1.58. The direct tag search (DTS) algorithm, introduced in this paper, allows dynamic scheduling while eliminating the need for associative tag comparison hardware. The DTS issue logic achieves a total speedup of 1.38, and thus retains much of the performance gain of Tomasulo's algorithm. A derivative of Thornton's algorithm gives a total gain of 1.28.

In our model, the large intermediate register files of the CRAY-1 (B and T) are treated as a unit, since it is not practical to assign tags to so many registers. With Tomasulo's algorithm, this was the cause of a relatively low performance improvement for three loops. With an architecture that does not have the large number of registers the CRAY-1 has, it would be possible to use tags for all the registers, thus increasing the speedup of the above loops.

Finally, we have discussed the impact of code scheduling on the simulation results. Tomasulo's algorithm is less sensitive to the order of the instructions, since it does a great deal of dynamic scheduling and register reallocation. On the other hand, static scheduling has a significant impact on the performance of the DTS and Thornton's algorithms. We gave specific examples how the performance of the latter two algorithms can be improved even by simple code scheduling.

8. Acknowledgements

This is material based upon work supported by the National Science Foundation under Grant ECS-8207277.

The authors would like to thank Nick Pang for the CRAY-1 simulators used for generating the results in Tables 1 and 3.

9. References

[ANDE67] D.W. Anderson, F.J. Sparacio, F.M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling", IBM Journal, V 11, Jan 1967

[BOLA67] L.J. Boland, G.D. Granito, A.U. Marcotte, B.U. Messina, J.W. Smith, "The IBM System/360 Model 91: Storage System", IBM Journal, V 11, Jan 1967.

[BONS69] P. Bonseigneur, "Description of the 7600 Computer System," Computer Group News, May 1969.

[BUCH62] W. Bucholz,ed., *Planning a Computer System*, McGraw-Hill, New York, 1962.

[CDC81] "CDC CYBER 200 Model 205 Computer System Hardware Reference Manual," Control Data Corp., Arden Hills, MN, 1981.

[CRAY77] "The CRAY-1 Computing System", Cray Research, Inc., Publication number 2240008 B, 1977.

[CRAY79] "CRAY-1 Computer Systems, Hardware Reference Manual", Cray Research, Inc., Chippewa Falls, WI, 1979.

[CRAY82] "CRAY X-MP Computer Systems Mainframe Reference Manual", Cray Research, Inc., Chippewa Falls, WI,1982.

[KOGG81] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, 1981.

[MCMA72] F. H. McMahon, "FORTRAN CPU Performance Analysis," Lawrence Livermore Laboratories, 1972.

[RUSS78] R.M. Russell, "The CRAY-1 Computer System", Comm. ACM, V 21, N 1, January 1978, pp. 63-72.

[THOR70] J.E. Thornton, *Design of a Computer - The Control Data 6600*, Scott, Foresman and Co., Glenview, IL, 1970

[TOMA67] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal, V 11, Jan 1967