Nalluri Ravi Teja
17XJ1A0528

INSTRUCTION ISSUE LOGIC FOR HIGH PERFORMANCE INTERRUPTIBLE PIPELINED PROCESSORS

ABSTRACT

Pipelined processor performance is severely limited by data dependencies and branch instructions. In order to achieve high performance, a mechanism to decrease the effects of data dependencies must exist. If a pipelined CPU with multiple functional units is to be used in the presence of a virtual memory hierarchy, a mechanism must also exist for determining the state of the machine precisely. In the paper, the author combined the issues of dependency-resolution and preciseness of state. A design for instruction issue logic that resolves dependencies dynamically and. at the same time, guarantees a precise state of the machine, without a significant hardware overhead is presented in the paper. Detailed simulation studies for the proposed mechanism, using the first 14 Lawrence Livermore loops(LLL) as a benchmark, are presented.

Keywords: Data Dependencies, Data dependencies, Branch Instructions, Precise interrupts, out of order execution, Tomasulo's algorithm.

INTRODUCTION

A widely used technique for increasing processing power is pipelining, in which the overall logic of the system is split into several stages with each stage performing a sub-task of a complete task. Pipelined CPUs have major obstacles to their performance like data dependencies and branch instructions. Use of reservation stations for instructions to wait for its operands and allow the other subsequent instructions run or code scheduling techniques to increase dependent instructions are solutions for dependency problems. Reservation Station is a place where the instruction can wait and allow subsequent instructions to proceed thereby allowing instructions to issue out of order. Out of order execution helps the instructions to be executed in an order of availability of data or operands instead of original instructions in the program. The reservation stations for the core of the dependency resolution mechanism. The effects of branch instructions can be alleviated by branch prediction, where the probable execution path of the instruction is determined. Using prediction techniques the path of branch instruction is predicted and instructions are pulled into the instruction buffer or can be executed in conditional mode.

Now coming to precise interrupts, precise interrupts will leave the machine in a well-defined state. The order of instruction execution should be well maintained during precise interrupt handling. Imprecise interrupts are quite the opposite of precise interrupts. In pipelined processors, imprecise interrupts can be caused by instruction-generated traps such as page faults which can leave the machine in an irrecoverable state. Hardware solutions are there for the imprecise interrupts but we are unaware of any software solutions. If at all a software solution exists, the software must 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 a code sequence to overturn the effects of those instructions. So, a software solution is quite difficult.

PROBLEM STATEMENT

Data dependencies and branch instructions are two impediments to the performances of the processor. If the operands of the current instruction are the result of a previous instruction, the instruction must wait till the previous instruction has completed execution because without the operands execution cannot start. This degrades the performance. Branch instructions cause more delay because the instruction must wait for its condition to be known. An additional penalty is also incurred in fetching an instruction from the taken branch path. These effects can be alleviated using delayed branch instructions but it is limited for long pipelines. The major problem that arises in pipeline computer design is imprecise interrupts. This problem is critical in the multi-function unit computers because even though the instructions are in the program order, the instruction execution can be out of program order. This results in degradation of the processors' performance. So, an adequate solution must be developed for

overcoming performance degradation due to imprecise interrupts. Solutions should be developed to overcome the problem of performance degradation caused due to data dependency and branch instructions.

Imprecise interrupts and out of order instruction issues are considered independent by previous researchers. In the paper, imprecise interrupts and out of order instruction issues are treated simultaneously. A precise interrupt mechanism will aggravate dependencies. So, a combined mechanism that implements precise interrupts with an out-of-order instruction issue mechanism so that the aggravated dependencies, as well as other dependencies, can be tolerated is introduced.

SOLUTION DESCRIPTION:

MODEL ARCHITECTURE:

The architecture presented has the same capabilities and executes the same instruction set as CRAY-I. However, there are a few differences between the model and the CRAY-I. First, if conditions are favourable, all the instructions are issued in a single cycle. Secondly, only one function can output data onto the result bus whereas the CRAY-I has separate result buses for the address and functional units. Instruction fetching and decoding and issuing takes place in the instruction fetch and decode and issue unit respectively. The results are written into the register file. Simulations are made by executing the first 14 LLL on the model architecture to check the number of instruction executed, the number of instructions per cycle and number of instruction cycles per program. Assumptions are made for generating the results. No memory bank conflicts, all instruction references are serviced by the instruction buffers and the instructions are already present in the instruction buffers when the program is starts are the assumptions made. The instruction issue rate for the total 14 loops is the harmonic mean of the individual issue rate. From the simulations, the author determined the main reason for the suboptimal performance of the model architecture is data dependencies. Hardware dependency resolution allows the model to tolerate these dependencies as well as implement precise interrupts.

HARDWARE DEPENDENCY RESOlUTION

When an instruction reaches the decode and issue unit, check must be made if all the dependencies are resolved for the instruction. If any operand is not available it waits in the decode and issue unit thereby blocking the subsequent instructions. So, reservation stations must be provided that the waiting instruction steps aside thereby freeing the decode and issue unit.

TOMASULO'S ALGORITHM

The underlying principle under Tomasulo's algorithm is this: If the operands of an instruction are not available, the instruction is moved to the Reservation Station(RS) associated with a functional unit that it will be using until it's operands are available. The instruction in the reservation station monitors the common data bus to resolve the dependency. Source register is busy if it is the destination of an instruction that is still being executed. Each source register is given a busy bit to check if it is busy and a tag to identify which register the result goes into. At instruction issue time, if the source register is busy then the tag of the register is obtained and the instruction is forwarded to the reservation station. When the source register is not busy the operand becomes available and the instruction obtains the destination register tag. The instruction is dispatched to the functional unit and the result proceeds to the result bus which is snooped by the registers and reservation stations which updates their contents. However, this algorithm might be expensive since each register needs to be tagged and each tag needs comparison hardware to match the tags. This may not be possible if there are large no of registers just like the model discussed.

SEPARATE TAG UNIT-EXTENSION OF TOMASULO'S ALGORITHM

Very few registers of all the destination registers are active. So, we can group the tags of all the active registers into a tag unit. When instruction enters the issue stage, if the source register is busy, the tag of the source register is obtained from the TU and forwarded to the reservation station. If the destination register is busy, a new

tag is obtained and the instruction holding the old tag is informed while the register clears it a busy bit when it completes execution. A new bit is associated with each TU entry to check if it is the latest tag of the register and the instruction holding tag has the key to clear the busy bit. The result is sent to the reservation stations as well as TU from which it sends the result to the appropriate register. So registers are updated only by TU and as soon as the register is updated the tag is set free.

EXAMPLE -OPERATION OF TAG UNIT

Consider the execution of an instruction I_1 and puts it in the third register. Consider a TU that has 6 entries(Tabel 1). Each entry in the TU has a bit indicating if the tag is free, a bit indicating if the tag is the latest tag for the register and a field for the number of the destination register. Let the operand registers be S0 and S3 and designation register is S4. When the issue logic decodes the instruction, it gets a new tag for the destination register S4 from the TU since the destination register isn't free and gets tag 3. Since the TU already has a tag for S4. the old tag is updated to indicate that it no longer represents the latest copy of the register. Since S0's and S3's contents are not valid a tag must be obtained from the TU. S0 obtains tag 2 and S3 obtains tag 6. Then a packet is forwarded to the reservation station by issue unit which consists of the tags of S0, S3 and S4. The instruction waits in the reservation station until the tags 2,6 appear on the result bus. After the instruction executes the result will go to all the reservation stations with the matching tag and to Tu as well. The Tu then forwards the result to the corresponding register i.e S4.

Tag No	Register Number	Tag Free	Latest Copy
1	A0	No	Yes
2	S0	No	Yes
3	NIL	Yes	Yes
4	S4	No	Yes
5	S0	No	No
6	S3	No	Yes

Table 1: Tag Unit with 6 entries.

INTERACTING WITH THE MEMORY

A model similar to the tag unit is adopted to handle memory interactions. A set of load registers stores the currently active memory locations. Each load register has a tag just like TU. A load operation requires only memory address whereas the store operation requires both address and data. If the address of an operation is not available, then the subsequent instructions do not proceed. If a memory address is required for load/store then checks are made in the load register for the matches. If it matches it means the address is not free so the operation has to wait. If the operation is load operation and the match is found the tag of the appropriate load register is returned to the reservation station. If the operation is a store and if there is a match, the tag of the load register is updated and returned to the reservation station so that a new memory instance of the memory location is provided. After the operation is finished the reservation station is freed. Also, the load register is also freed depending if the tag matches.

MERGING THE RESERVATION STATIONS

Each functional unit has a set of reservation stations. There is a possibility that the functional unit may run out of reservation stations. While some other reservation stations associated with another functional unit may be

idle. So, we can all combine the reservation stations into a pool and when an instruction is issued go to the common RS pool. As instruction gets ready, it goes to the functional unit. If RS pool is full, the instruction is blocked.

MERGING RS POOL AND TAG UNIT

At any given time there is a one-one correspondence between the entries in TU and the instructions in the RS or functional units. So, we can combine the RS pool and TU into RSTU(RS tag unit). Though a reservation station is wasted if it is associated with an instruction that is in a functional unit, this organization can easily be extended to allow for the implementation of precise interrupts. At the time of the instruction issue, the tag is obtained for RSTU and in doing so a tag is obtained for RS as well. Since reservation stations are merged, a functional identity field is needed to determine to which functional unit the instruction should go.

In order to check the effectiveness of the RSTU mechanism, a simulation analysis is carried out using the first 14 Lawerence Livermore Loops. The RSTU mechanism is able to achieve speed up over simple instruction mechanisms. Also, all the non-branch instructions are able to achieve the limit of 1 instruction per clock cycle. The branch instruction cycles are dead cycles because no instruction is executed. This degradation could be reduced by using delayed branch instructions. 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, it may seem the RSTU mechanism is at a disadvantage when compared to distributed reservation stations. In order to evaluate the effectiveness of multiple data paths between the RSTU and the functional units, an architecture with two 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 are simulated. It didn't make much difference in the presence of the duplicate path.

IMPLEMENTING PRECISE INTERRUPTS

There are several mechanisms to implement precise interrupts: Reorder buffer, reorder buffer with bypass logic, history buffer and a future file. Reorder buffer holds the results of the instruction until they are ready to finish in program order. It commits the instructions in the order of that arrived instructions at the decode and issue unit. The downside of a reorder buffer is the value of the register cannot be read until the reorder buffer updates it. This increases the data dependencies. However, if a bypass logic is associated with the reorder buffer, the instruction directly reads the value from the reorder buffer and can issue. With a bypass mechanism, issue rate not degraded but the mechanism is expensive. History buffer and future file mechanisms achieve the same performance as reorder buffer with bypass logic at the expense of read port for register and duplicating the register file.

MERGING DEPENDENCY RESOLUTION AND PRECISE INTERRUPTS

RSTU can be modified to a reorder buffer by forcing it to update the state of the machine in the order of instruction issue. So, in order to run precise interrupts in architecture with RSTU, the RSTU must be managed as a queue. This logic is called Register Update Unit(RUU).

The RUU is the unit that (i) determines which instruction should be issued to the functional units for execution, reserves the result bus and dispatches the instruction to the functional unit, (ii) determines which instruction can commit (iii) monitors the result bus to resolve dependencies and (iv) provides tags to and accepts new instructions from the decode and issue logic.

In designing the RUU, two things should be noted: (i) it should not involve a large amount of comparison hardware and (ii) it should not affect the clock speed to an intolerable extent. The logic of obtaining source operands and destination register tag is a very important task for managing the RSTU as a queue. Previously, the issue logic searched the TU to obtain the correct tag for the source operand and to update the latest copy field for the destination register. Such a wide associative search may not be acceptable because of the large amount of hardware required. By providing a new instance for a busy destination register, the architecture can process several instructions that write into the same register simultaneously. Unfortunately, disallowing multiple instances of a destination register degrades the performance. However, it is possible to eliminate the associative search and use a counter to provide multiple instances for each register if we can guarantee that results return to the registers in order. This is exactly the

goal of the precise-interrupt mechanism. The implementation of precise interrupts, therefore, simplifies the design of the dependency resolution mechanism.

This scheme uses 2 n-bit counters with each register in the register file and no busy bit. The 2 n-bit registers are 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. If an instruction that writes into the register is issued to the RUU, both NI and LI are incremented. Maximum of 2^n -1 instances of a register can be present in the RUU at any time. If NI for a destination register is 2^n-1, the issue is blocked. The associated NI is decremented when the instruction from RUU updates the value of the register. If NI=0 it means that there is no instruction in RUU that writes into the register.

OPERATIONS OF RUU

RUU resolves the data dependencies. In RUU, each source operand field has a ready bit, a tag subfield and a content sub-field. If the operand is not ready, the tag sub-field snoops the result bus for a matching tag and if a match is detected, the data is forwarded into the content field. In this way, the RUU accomplishes the task of resolving data dependencies The RUU consists of a dispatched field, executed field, counter field. The dispatched field tells if the instruction left for execution to the corresponding functional unit. THe execute field tells if the instruction execution has finished or not. Counter field is used to implement precise interrupts

The RUU determines which instruction should be dispatched to the functional unit by monitoring the ready bits of the source operands. When the operands are ready, the instruction is dispatched. The RUU sets the dispatched bit to indicate that instruction has been dispatched for execution. The RUU accomplishes the task of committing an instruction by monitoring the executed bit. If the executed bit is set, then the result of the destination register can be broadcasted to the register file. The RUU_Head is updated and the NI counter is decremented if the bit is set.

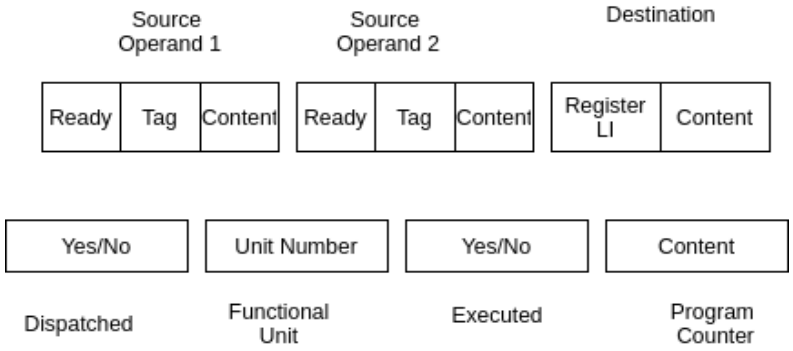

Fig 1: RUU entry

EVALUATION OF RUU

Three RUU organization simulations were made to evaluate the effectiveness of RUU. One with bypass logic, another without bypass logic and the last one limited bypass logic. One of the primary drawbacks of the RUU is that performance may be degraded because instruction issues are blocked if a source register is busy even though its result may be present in the reorder buffer. This performance-degrading problem is easily rectified if bypass logic is provided so that a source operand could be read directly from the RUU before it is written into the register file.

Bypass logic, though simple, is cumbersome and expensive to implement. Sometimes bypass logic is not necessary. Consider two instructions i and j where j is dependent on i. If i completes execution after j is issued, then j can use the result of i when it appears on the result bus since the reservation stations with RUU snoops the result bus. Here the bypass logic is not necessary. Bypass logic is necessary when instruction i has completed execution when instruction j. Instead of bypass logic, the instruction j can wait for i's result to broadcast on the bus to resolve the dependency. The author compared the relative speedups, instruction issues with different sizes of RUU with and

without bypass logic. The speedup that an RUU without bypass logic has significant improvement is a simple instruction issue mechanism and implements precise interrupts at the same time, but it is not as impressive as RUU with bypass logic. The difference is because of the ordering of the code in loops. Due to this the branch instructions are blocked at decode and issue unit until the result of the instruction appears on the result bus. To overcome this problem register files are duplicated, thus creating a limited bypass for the registers. This duplicate register file acts as a future file for the registers. The architectural register file contains a valid copy of all the registers. Instructions that use these registers as source operands fetch the operands from future file. This eliminates waiting for instruction for small size RUU's only. But, for large RUU's the performance isn't satisfactory.

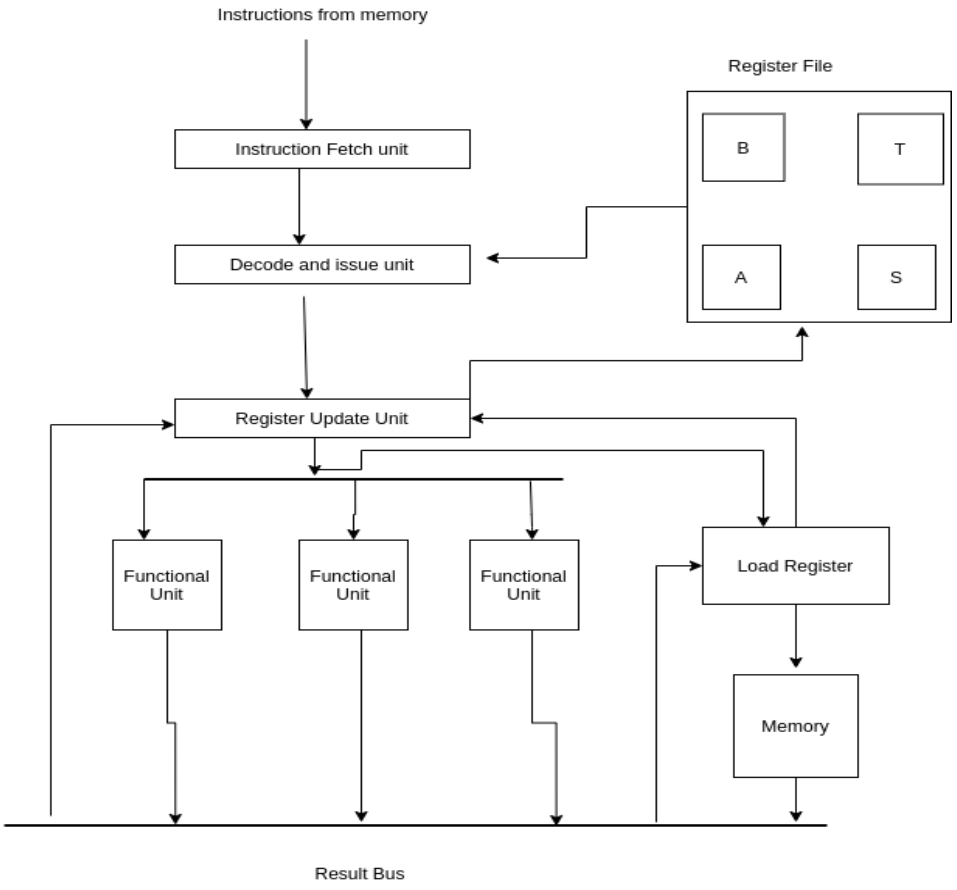

Fig 2:Final Architecture with RUU

BRANCH PREDICTION AND CONDITIONAL INSTRUCTIONS

The detrimental effects of branches can be alleviated by executing instructions conditionally from the predicted path. A hardware mechanism that would help the machine to recover is required if incorrect branch prediction happens during conditional execution of branched instructions. The RUU mechanism nullifies the instructions. These instructions can be valid instructions or instructions executed in the conditional mode. Instructions like valid instructions can be nullified because of data dependency on previous instruction and conditional mode instructions can be nullified if they are from the wrong path of execution. If the decode and issue unit actually finds the branch path and executes in instruction, the recovery from the incorrect branch predictions,if there are any, can be easily done. Such instructions can be identified by adding an extra field in the RUU. and preventing them from committing until they are proven from the correct path. So, the conditional execution of instructions with RUU mechanism is easy.

SUMMARY

In the paper, mechanisms are proposed to alleviate the data dependencies and branch instructions. The author combined issues of hardware dependency resolution and implementation of precise interrupts A scheme to resolve dependencies and allowing out of order execution at low hardware costs is devised. This scheme is extended to integrate the precise interrupts. This integration is made simpler than before using the scheme. The RUU mechanism is able to achieve a substantial increase in speed over the simple instruction issue mechanism. It also implemented precise interrupts with considerable cost in hardware.

BIBLIOGRAPHY

https://cs.nyu.edu/courses/fall02/G22.2243-001/lectures/lect7-4up.pdf
https://user.eng.umd.edu/~blj/RiSC/RiSC-oo.1.pdf
https://courses.cs.washington.edu/courses/cse471/07sp/lectures/Lecture4.pdf
http://nrg.cs.ucl.ac.uk/mjh/3005/2004/19-io.pdf
https://ieeexplore.ieee.org/document/5529339