Pipeline Optimization by Out-of-Order Execution and Register Renaming

Thu Nandar
Technological University of Hmawbi, Yangon, MYANMAR
thunandar@gmail.com

Abstract

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exits among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs. Instructions in the pipeline can depend on one another, which prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. This paper will resolve this problem by using Out-of-Order execution and register renaming method. In this design, instructions may be issued out of order and may be retired out of order as well. The MIPS microprocessor is used as a running example to demonstrate our method. In this paper, it will represent hardware design to be able to execute out-of-order and the implementation is simulated by software using Visual Basic 6.0.

1. Introduction

Throughput is the number of operations processed by a circuit in a given time (usually per clock period). The concept of throughput applies to any type of circuit. For the throughput of microprocessors, the operations that we are concerned with are instructions. Pipelining is commonly used to optimize throughput, by partitioning the function of the circuit into stages, so that multiple instructions can be processed concurrently. The partitioning of instruction processing is done by introducing state-holding elements, called pipeline registers (delays), into the pipeline. We demonstrate this process using the MIPS. The MIPS is decomposed into 5 functional units. They are Instruction Fetch , Instruction Decode, Instruction Execution, Memory Access and Write Back. A functional unit is a portion of the circuitry that performs a task, which contributes to the overall objective of processing instructions.

Pipelining may introduce hazard situations, which occur when the overlapping of execution stages of instructions causes incorrect output. These hazard situations must be detected and resolved, in order to ensure correct output. Hazards arise as a result of data dependencies, instructions that change the pc, and resource conflicts. There are three types of hazards. They are structural hazards, data hazards and control hazards.

For a pipeline to process instructions correctly, hazards must be resolved using control circuitry (bypass, stall or kill hardware). The addition of this control circuitry increases the cost of the stages to which it is added, which means that pipelining may actually decrease clock period or instruction throughput, rather than increase it.

2. Pipeline Design Techniques

Pipelining has different techniques depending on the nature of design environment. The key contribution of pipelining is that it provides a way to start a new task before an old one has been completed. Hence the completion rate is not a function of the total processing time, but rather of how soon a new process can be introduced [4].

2.1. Performance of Pipelined Computers

Computer performance can be defined in several different ways; especially response time—the time between the start and completion of a task—also referred to as execution time and throughput—the total amount of work done in a given time. To maximize performance, it is needed to minimize response time or execution time for some task [1].

2.2. Pipeline Types

There are two types of pipeline. They are arithmetic pipeline and instruction pipeline. An arithmetic pipeline divides an arithmetic operation into sub-operations for execution in the pipeline segment. Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed point number, vector operations, and similar computations encountered in scientific problems. An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phase of the instruction cycle. Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This cause the instructions fetch and execute phases to overlap and perform simultaneously operations [3].

3. The MIPS Architecture

In this paper, a simple 64-bit load-store architecture called MIPS is described. Like most recent computers, MIPS emphasizes

- A simple load-store instruction set
- Design for pipelining efficiency, including a fixed instruction set encoding
- Efficiency as a compiler target

In this paper, we will use a subset of what is now called MIPS64, which will often abbreviate to just MIPS. MIPS64 has 32 64-bit general-purpose registers(GPRs). Additionally, there is a set of 32 floating point registers (FPRs), which can hold 32 single-precision (32-bit)values or 32 double-precision (64-bit) values. The data types are 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit double words for integer data and 32-bit single precision and 64-bit double precision for floating point. MIPS memory is byte addressable with a 64-bit address. Since MIPS has just two addressing modes, these can be encoded into the opcode. MIPS has three instruction types. They are I-type instruction, R-type instruction and J-type instruction [2].

4. Design and Simulation for Pipeline Optimization

4.1. Out-of-Order Execution and Register Renaming with Scoreboard

Scoreboarding is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard. After decoding an instruction, the decode unit has to decide whether or not it can be issued immediately. To make this decision, the decode unit needs to know the status of all registers. If the current instruction needs a register whose value has not yet been computed, the current instruction cannot be issued and the CPU must stall.

We will keep track of register use with a device called a scoreboard. In real machine, the scoreboard also keeps track of functional unit usage, to avoid issuing an instruction for which no functional unit is available. For simplicity, we will assume there is always a suitable functional unit available, so we will not show the functional units on the scoreboard. To make the example realistic, we will assume that our MIPS machine will allow the decode unit to issue up two instructions per clock cycle.

To see whether or not an instruction can be issued, the following rules: RAW dependence, WAR dependence and WAW dependence are applied. [5]

4.2. System Design to be able to Execute Out-of-Order

There has been added Instruction Queue, Issue or not control unit, Scoreboard and one 32 64_bit Secret register file to the original MIPS machine. Figure.1 shows what the processor looks like.

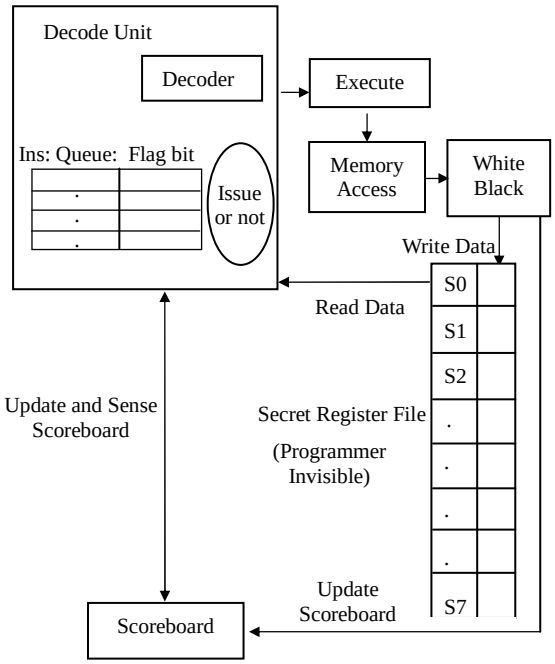

Figure. 1. The Basic Structure of a MIPS Processor with a Scoreboard

The scoreboard function is to control instruction issue. The scoreboard has a small counter for each register telling how many times that register is in use as a source by currently executing instructions. If a maximum of, say, 15 instructions may be executing at once, then a 4-bit counter will do. When an instruction is issued, the scoreboard entries for its operand registers are incremented. When an instruction is retired, the entries are decremented.

The scoreboard also has a counter for each register to keep track of registers being used as destinations. Since only one write at a time is allowed, these counters can be 1-bit wide.

To resolve WAR and WAW hazards, one 32 64_bits secret register file, named S0.…S31, is added to the original MIPS machine. In consequently, the scoreboard also keeps track of original register number to secret register number.

There will be required to know whether the instructions in the queue of the decode unit are stalled or not. Thus, one bit flag register is added for each instruction in the queue. If there is a stall, the flag bit for this instruction will be 1, otherwise it will be 0. When the Write Back unit receives the result, it notifies scoreboard and then the scoreboard will be updated.

4.3. Algorithm for Issue or Not Control Unit

In this paper, It is assumed that our MIPS machine is a dual pipeline superscalar processor. So it can fetch and decode 2 instructions simultaneously. Every clock cycle undergoes as shown in Figure. 2.

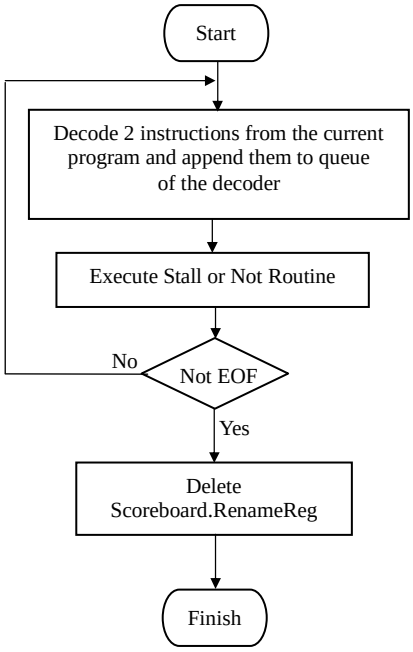

Figure. 2 . Algorithm for Stall or Not Control Unit

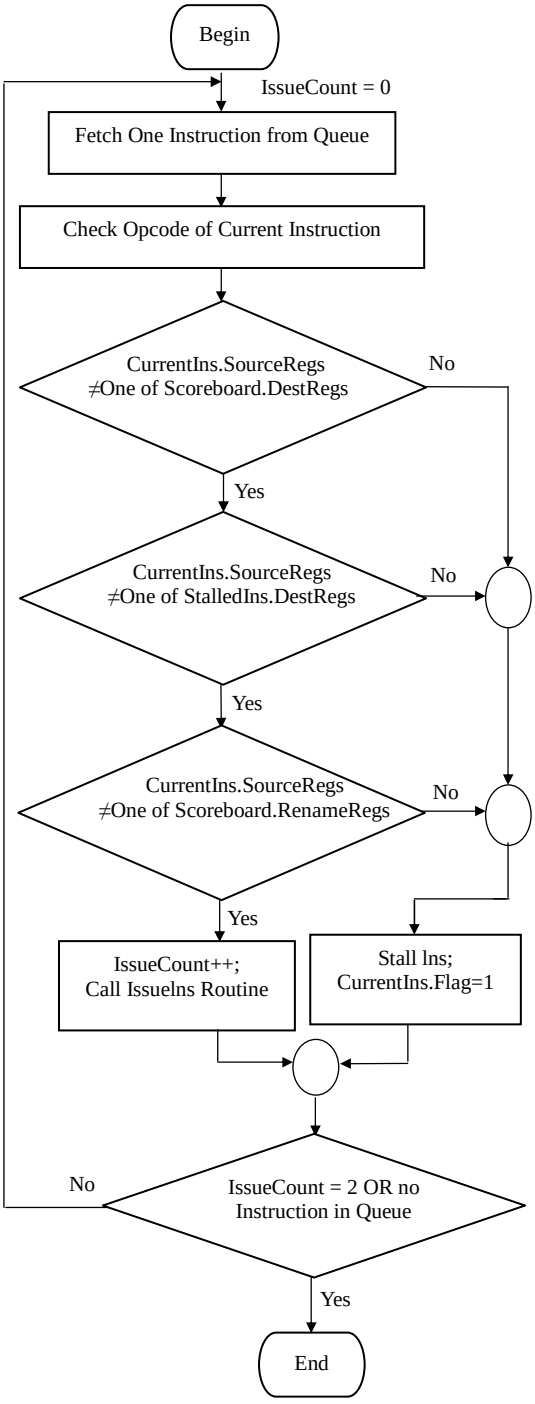

Figure. 3. Algorithm for Stall or Not Routine

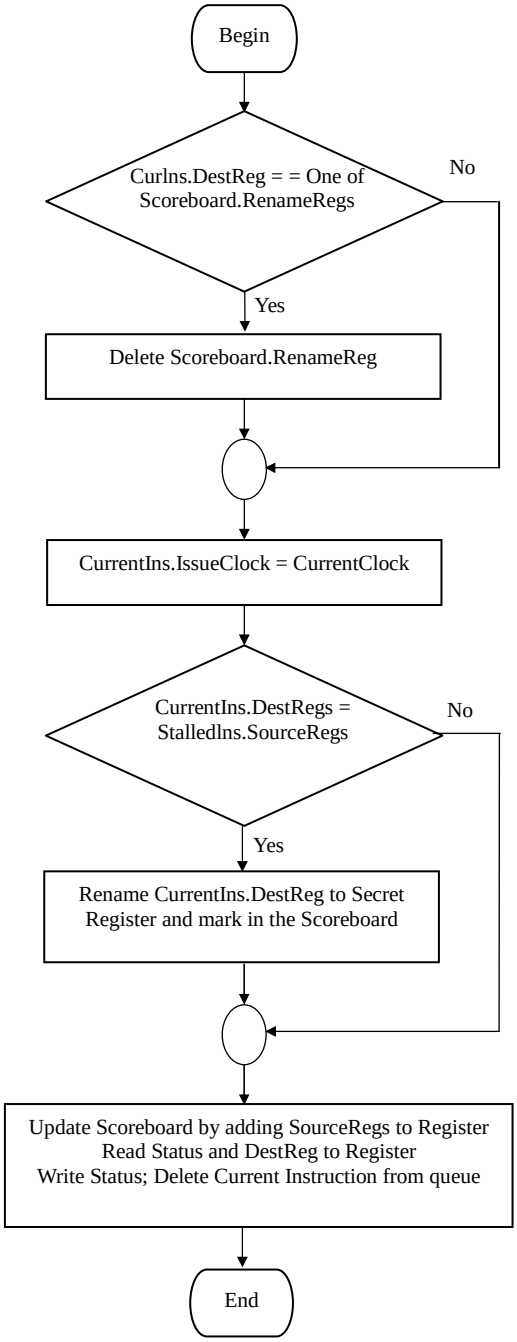

Figure. 4. Algorithm for Issue Instruction Routine

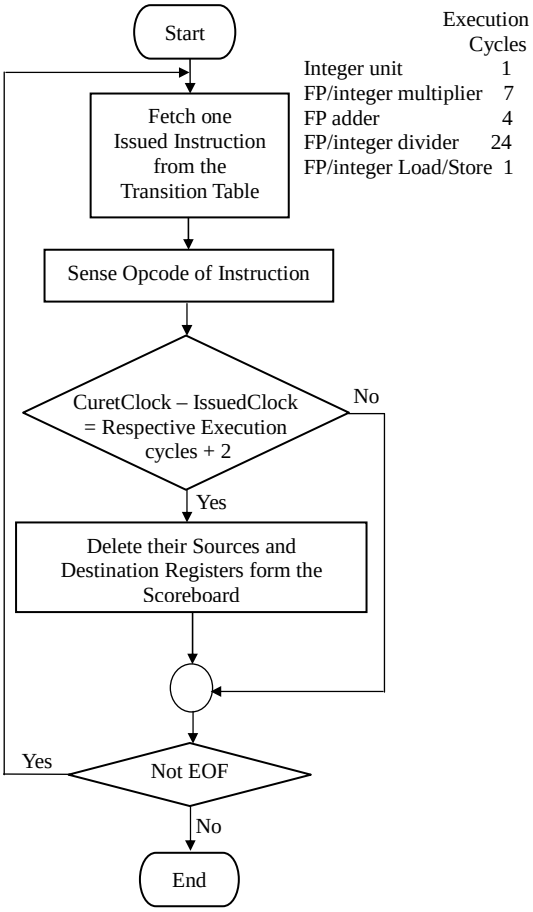

Figure. 5. Algorithm for Finished or Not Routine

In the hardware design, when the issued instruction arrive the write back and memory access units, there can be known from the register write control signal for R-type instructions and memory read or memory write instruction for store word and load word instructions. The scoreboard could use these control signals and then can update.

But the scoreboard need to sense whether the issued instructions finish or not in our simulator in every clock cycle. Thus, there has been created a transition table for the issued instructions (currently executing instructions) and all are stored in this table. Finish or Not Routine is used in the simulator design.

4.4. In Order Issue and In Order Completion

To illustrate the nature of the problem, there has been started with a machine that always issues instructions in program order and also requires them to complete execution in program order.

Here is a loop in C:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h ) goto Loop;
```

Assume that A is an array of 100 elements .

The corresponding MIPS machine code corresponding to this C loop is

```
Add R9, R19, R19
Add R9, R9, R9
Add R9, R9, R21
Lw  R8, 0, R9
Add R17, R17, R8
Add  R19, R19, R20
Bne R19, R18, Loop
```

This program to repeat 3 times execution takes 69 clock cycles.

4.5. Out-of-Order Issue and Out-of-Order Completion

In this design, instructions may be issued out of order and may be retired out of order as well.

Program will appear in the source code table. Two instructions are decoded and only first instruction is issued in the instruction table. The instruction 2 depends on instruction 1 and thus it is stalled. The sources and destination register numbers of this instruction are stored in the scoreboard.

Next two instructions are decoded and they are stalled because one of it’s source registers depend on the destination registers of instruction 2.

Even though instruction 2,3.4 and 5 are stalled the simulator decode and issue instruction 6 since it does not conflict with any pending instruction.

Instruction 1 is finished at clock cycle 4 and so instruction 2 is issued at clock cycle 5. At that time instruction 6 is at memory stage.

The next instruction after branch (Beq and Bne) is decoded at the time of write back of the previous branch instruction because the zero control pin is at the execute stage . Then it takes 1 clock for instruction fetch.

A branch to see if two registers are not equal will give us the effect of branching to label LOOP. But our simulator couldn't know the value of these two registers. So we need to input 1 for branch taken and 0 for otherwise.

Instruction 3 is still computing a value in register 9 that will be used by instruction 4 stalled now. At the same time, instruction 1 for second iteration start to issue and it can overwrite Register 9. There is no real reason to use register 9 as the place to hold the result of instruction 1 for second iteration. Register renaming solves for this problem. The wise stall or not control unit changes the use of register 9 in instruction 1 for second iteration to a secret register, S9, not visible to programmer. This technique can often eliminate WAR and WAW dependencies. There are many secret registers and a table called secret that maps the registers visible to the programmer onto the secret registers. Thus the real register being used for, say, R0 is located at entry 0 of this mapping table. In this way, there is no real register R0, just in binding between the name R0 and one of the secret registers. This binding changes frequently during execution to avoid dependencies.

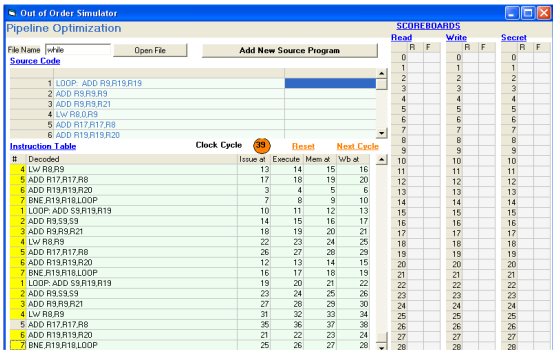

Figure. 6. Clock Cycle 39 of Simulation

Notice that in Figure.6 when reading down the "Issue at" column and "Wb at"column of "Instruction Table", the instructions have not been in order. Nor they have been retired in order. The conclusion of this example is simple: using out-of-order execution and register renaming there has been able to speed up the computation by close to a factor of two.

5. Conclusion

Out of order execution is one method for improving performance in a multiple instruction issue processor. When applied to a superscalar processor, out-of-order execution has the traditional benefit of boosting performance in the face of data hazards. MIPS64 pipeline is used for basic machine to optimize by out-of-order execution for this paper.

RAW dependence occur when an instruction needs to use as a source a result that previous instruction has not yet produced.

A feature to even more increase the performance gain of out-of-order execution is register renaming. In the determination whether an instruction is executable or not it is trivial that a producer must have finished before a consumer of the same data can use it. This is called Read after Write (RAW).

The only remaining dependency in a perfectly working register renaming out of order system is the read after write dependency.

References

[1] A. Klauser, A. Paithankar & D. Grunwald , "*Selective Eager Execution on the PolyPath Architecture*" IEEE, 1998.

[2] A. Patterson, David & L. Hennessy, John, "*Computer Architecture: A Quantitative Approach*", Morgan Kaufmann Publishers Inc, San Francisco, California.

[3] M. Morris, Mano, "*Computer System Architecture*", Prentice-Hall Inc,Third Edition, 1993.

[4] S. Stone, Harold, "*High-Performance Computer Architecture*", Addison-Wesley Publishing Company, Inc, Third Edition, 1993.

[5] S. Tanenbaum, Andrews, "*Structured Computer Organization*", Prentice-Hall, Inc, Fourth Edition, 1999.