Raul Salazar UFID 6187-6765 Section 17H1 Wed P7

HW4

PART 1

Vocabulary

1.1 Pipelined Datapath: instructions share a datapath, with 5-stage pipeline, up to 5 instructions will be in execution during single clock cycle. Used to speed up execution of programs.

1.2 WAR Dependency: "write-after-read" dependency which occurs when an instruction tries to write to a reg which has not yet been read by a previously issued, but as yet uncompleted instruction

1.3 Pipeline Data Hazard: occurs when previous instructions are computing values needed for current instructions. 3 kinds: WAW, RAW, WAR

1.4 Branch Prediction: a method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome

1.5 Control Hazard in Pipeline Datapath: occurs when the pipeline makes wrong decisions on branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded (aka branch hazard)

2) Explanation

The Branch Delay Slot is the slot directly ~~after~~ a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch. Compilers & assemblers try to place an instruction that always executes in this slot. When ~~executing~~ executing a branch, the instructions in fetch & decode are tossed then we spend time filling up the pipeline again. The delay slot allows the instruction right after the branch to execute regardless and in so, minimizing the amount of time we lose throughout this process.

# Part 2

## 3) Pipelining Schedule

L1: add $s1, $s1, $s2
L2: sub $s2, $s1, $s1

3.1) hazards
- RAW because L2 reads from $s1 after L1 writes its result to it
- WAR because L2 writes to $s2 and L1 might not have read from it yet

3.2)

| | C1 | C2 | C3 | C4 | C5 |
|---|---|---|---|---|---|
| L1 add $s1, $s1, $s2 | F | D | E | M | W |
| L2 sub $s2, $s1, $s1 | | F | D | E | M | W |

// data forwarding from L1(e) to L2(e)

3.3)

| | C1 | C2 | C3 | C4 | C5 |
|---|---|---|---|---|---|
| L1 add $s1, $s1, $s2 | F | D | E | M | W |
| L1b slt $s3, $s4, $s2 | | F | D | E | M | W |
| L2 sub $s2, $s1, $s1 | | | F | D | E | M | W |

## 4) More Pipelining

4.1) hazards
- RAW - L2 reads from $s1 after L1 writes to it
- WAW - L1 & L2 both write to $s1
- RAW - L3 reads from $s1 after L2 writes to it

schedule

| | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 |
|---|---|---|---|---|---|---|---|---|
| L1 addi $s1, $s2, 37 | F | D | E | M | W | | | |
| L2 subi $s1, $s1, X | | F | D | E | M | W | | |
| L3 bne $s1, $s2, Exit | | stall | F | D | E | M | W | |

// data forwarding from L1(e) to L2(e)

↘ rule: if branch comparison registers are dest. reg. for previous instruction we stall once

4.2) CPI for 4.1

there are 8 cycles, and 3 instructions

so -> CPI = 8/3 = 2.66667 = CPI

PART III - EXTRA CREDIT

5 - Branching and Pipelines

5.1

(Assuming instructions take one cycle)

The idea behind branch prediction is that if we can correctly predict whether branches are taken or not, we can reduce the stalls due to control hazards. The performance of the pipeline is affected when a branch is taken: each such branch requires a new address to be loaded into the program counter, which may invalidate all the instructions that are either already in the pipeline or prefetched in the buffer so we must flush out the pipeline. This draining and refilling of the pipeline takes a toll on the performance of the system and causes a much higher number of stalls/ penalty-cycles. Now when it comes to static branch prediction, we can set the computer to always predict in one direction or another.

According to the New Jersey Institute of technology, "About 60% of the forward conditional branches are taken, while because of the prevalence of program loops approximately 85% of the backward conditional branches are taken." This data helps us see that, at least for backward conditional branches, it is a good idea to implement a static prediction of 'branch taken' whereas we might need a more complex prediction system for forward branching. Furthermore, according to the university of Washington's computer science division, when a prediction is incorrect, it is essentially the same as not having a prediction system at all in terms of stalls / cycle-penalties.

Now let's recap: if the prediction is not taken & it is correct, there is no penalty, A branch not taken allows the continued sequential flow of uninterrupted instructions to the pipeline. On the other hand, if the prediction is taken and it is correct, then there is a one cycle penalty because we have a head start on loading the correct instructions rather than continuing to load the instructions coming after the branch had it not been taken. We also know that when a prediction is wrong it is the same penalty as not having a prediction at all, which is definitely a higher penalty than the worst penalty (in this case 1 cycle) possible when predictions are correct.
This is why I believe that it is much more advantageous as a whole to predict taken over not-taken: you will be right most of the time, speeding up the computer's processes; and when you aren't, you will suffer the same consequences as you would had you not implemented branch prediction at all.

5.2

For integer unit, there is no special "Divide By Zero" exception. Instead, the "div" instruction checks the operands for a "divide by zero" condition, and if there is one, uses a trap instruction to set the exception code to 7. The trap causes the CPU to enter exception mode. After that, there are two ways in which MIPS can handle this situation.

The first option is to generate code to take a break exception rather than a trap exception when an error is detected, which is the default.

The other option is that it can automatically macro expands certain division and multiplication instructions to check for overflow and division by zero. This option causes to generate code to take a trap exception rather than a break exception when an error is detected.