Processor Design — Pipelined Processor (3)

Hung-Wei Tseng
Announcement

• Midterm this Thursday
  • We will talk about the detail next Wednesday
  • No cheat-sheet allowed/required
  • You need to bring a calculator
Recap: Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
  - Each pipeline registers is clocked
  - Each pipeline stage takes one cycle
  - Pipeline registers only updates at the end of a clock
- Each stage acts on a different instruction simultaneously
- Each stage can only work on an instruction once
- States/control signals of instructions are hold in pipeline registers
Recap: Pipeline hazards

• Even though we perfectly divide pipeline stages, it’s still hard to achieve CPI == 1.

• Pipeline hazards:
  • Structural hazard
    • The hardware does not allow two pipeline stages to work concurrently
  • Data hazard
    • A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
  • Control hazard
    • The processor is not clear about what’s the next instruction to fetch
Recap: data hazard

• When an instruction in the pipeline needs a value that is not available

• Data dependences
  • The output of an instruction is the input of a later instruction
  • May result in data hazard if the later instruction that consumes the result is still in the pipeline

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```
Recap: stall for data hazard

add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)

15 cycles! CPI == 3
(If there is no stall, CPI should be just 1!)
Recap: Data Forwarding

- Take the values, where ever they are!
- Full forwarding allows a later instruction to consume the value as soon as the value is generated

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

10 cycles! CPI == 2 (Not optimal, but much better!)
Hazard detection in hardware
Data hazard revisited

• How many cycles it takes to execute the following code?
• Draw the pipeline execution diagram
  • assume that we have full data forwarding.

```
lw  $t1, 0($a0)
lw  $a0, 0($t1)
bne $a0, $zero, 0
```

9 cycles
Data hazards on a different pipeline design

- If we split the “MEM” stage into two stages, ME1 and ME2, and data is available after ME2, how many cycles it takes to execute the following code?

```
lw   $t1, 0($a0)  
lw   $a0, 0($t1)  
bne  $a0, $zero
```

A. 9  
B. 10  
C. 11  
D. 12  
E. 13
Control hazard
Control hazard

• Consider the following code and the pipeline we designed

LOOP: lw $t3, 0($s0)
     addi $t0, $t0, 1
     add $v0, $v0, $t3
     addi $s0, $s0, 4
     bne $t1, $t0, LOOP
     sw $v0, 0($s1)

How many cycles the processor needs to stall before we figure out the next instruction after “bne”?

A. 0
B. 1
C. 2
D. 3
E. 4
Control hazard

• Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards, what’s the average CPI if we execute this program on the 5-stage MIPS pipeline?

A. 1
B. 1.2
C. 1.4
D. 1.6
E. 1.8
Control hazard

- The processor cannot determine the next PC to fetch

```plaintext
LOOP: lw $t3, 0($s0)
    addi $t0, $t0, 1
    add $v0, $v0, $t3
    addi $s0, $s0, 4
    bne $t1, $t0, LOOP
    lw $t3, 0($s0)
```

7 cycles per loop
Solution I: Delayed branches

- An agreement between ISA and hardware
  - “Branch delay” slots: the next N instructions after a branch are always executed
  - Compiler decides the instructions in branch delay slots
    - Reordering the instruction cannot affect the correctness of the program
  - MIPS has one branch delay slot

- Good
  - Simple hardware

- Bad
  - N cannot change
  - Sometimes cannot find good candidates for the slot
Solution 1: Delayed branches

```
LOOP: lw $t3, 0($s0)
    addi $t0, $t0, 1
    add  $v0, $v0, $t3
    addi $s0, $s0, 4
    bne $t1, $t0, LOOP
```

6 cycles per loop
Solution II: always predict not-taken

- Always predict the next PC is PC+4

```
LOOP:  lw  $t3, 0($s0)
       addi $t0, $t0, 1
       add  $v0, $v0, $t3
       addi $s0, $s0, 4
       bne $t1, $t0, LOOP
       sw  $v0, 0($s1)
       add  $t4, $t3, $t5
       lw  $t3, 0($s0)
```

If branch is not taken: no stalls!
If branch is taken: no hurt!

7 cycles per loop
Solution III: always predict taken

- Using the pipeline we have, how many cycles we have to stall if we predict branch taken?

LOOP: lw $t3, 0($s0)  
      addi $t0, $t0, 1  
      add  $v0, $v0, $t3  
      addi $s0, $s0, 4  
      bne $t1, $t0, LOOP  
      lw $v0, 0($s0)

A. 0  
B. 1  
C. 2  
D. 3  
E. 4  

We still don’t know the target address until EX stage completes
Solution III: always predict taken

- When is the earliest stage that we can calculate the target address?
  A. IF  
  B. ID  
  C. EX  
  D. MEM  
  E. WB
Solution III: always predict taken
Solution III: always predict taken

Still have to stall 1 cycle
Solution III: always predict taken

Consult BTB in fetch stage
Branch Target Buffer

PC

branch PC

Branch Target Buffer

target address or target instruction
Solution III: always predict taken

- Always predict taken with the help of BTB

```
LOOP: lw   $t3, 0($s0)  
  addi $t0, $t0, 1    
  add  $v0, $v0, $t3  
  addi $s0, $s0, 4    
  bne $t1, $t0, LOOP
  lw   $t3, 0($s0)    
  addi $t0, $t0, 1    
  add  $v0, $v0, $t3
```

5 cycles per loop
(CPI == 1 !!!)

But what if the branch is not always taken?
Consider the following code:

```c
i = 0;
do {
    if (i % 3 != 0) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while (++i < 100) // Branch X
```

What is the prediction accuracy of branch Y if we use static always predict taken predictor? (Choose the closest one)

A. 0%
B. 33%
C. 67%
D. 100%
Static branch predictions

• How many of the following about static branch prediction method is correct?
  • Comparing with stalls, static branch prediction mechanisms are never doing worse in our current MIPS 5-stage pipeline
  • A static branch prediction mechanism never changes the prediction result during program execution
  • “Flush” occurs only after the processor detects an incorrect branch prediction
  • “Always predict taken” cannot fetch a taken instruction during the ID stage of the branch instruction without the help of BTB

A. 0  B. 1  C. 2  D. 3  E. 4
Dynamic branch prediction
1-bit counter

- Predict this branch will go the same way as the result of the last time this branch executed
- 1 for taken, 0 for not taken

PC = 0x400420

Branch Target Buffer

<table>
<thead>
<tr>
<th>PC</th>
<th>Target Address</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>1</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>1</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>0</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>1</td>
</tr>
</tbody>
</table>
Accuracy of 1-bit counter

• Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.

Assume unlimited BTB entries.

A. 0%
B. 33%
C. 67%
D. 100%