Pipeline Processor (3)

Hung-Wei Tseng
The “life” of a dynamic instruction?

- Instruction Fetch (IF)
  - Fetch the instruction pointed by PC from memory
- Instruction Decode (ID)
  - Decode the instruction for the desired operation and operands
  - Reading source register values
- Execution (EX)
  - ALU instructions: Perform ALU operations
  - Conditional Branch: Determine the branch outcome (taken/not taken)
  - Memory instructions: Determine the effective address for data memory access
- Data Memory Access (MEM) — Read/write data memory
- Write Back (WB) — Present ALU result/read value in the target register
- Update PC
  - If the branch is taken — set to the branch target address
  - Otherwise — advance to the next instruction — current PC + 4
add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9, $10, $11
sw $1, 0($12)
Recap: Pipelining

After this point, we are completing an instruction each cycle!
Recap: Three pipeline hazards

- Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
- Control hazards — the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction depending on a result that’s not yet generated or propagated when the instruction needs that
Recap: Can we get them right?

Given a simple pipelined MIPS processor that we discussed so far, how many of the following code snippets can be executed with expected outcome?

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>b</td>
<td>lw $4, 0($1)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
</tr>
<tr>
<td>c</td>
<td>sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>sub $9, $1, $10</td>
<td>sub $6, $7, $8</td>
</tr>
<tr>
<td>d</td>
<td>sub $9,$10,$11</td>
<td>sub $9,$10,$11</td>
<td>sub $9,$10,$11</td>
<td>sub $9,$10,$11</td>
</tr>
<tr>
<td>e</td>
<td>sw $1, 0($12)</td>
<td>sw $11, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
</tr>
</tbody>
</table>

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

Data Hazard
Structural Hazard
Control Hazard

b cannot get x1 produced by a before WB
both a and d are accessing x1 at the 5th cycle
We don't know if d & e will be executed or not until c finishes
Recap: Structural Hazards

- Stall can address the issue — but slow
- Compiler optimization — what if the hardware design changes?
- Improve the pipeline unit design to allow parallel execution
  - Write first, read later register file
  - All instructions need to go through exactly the same number of stages
Recap: Data hazards

• An instruction currently in the pipeline cannot receive the \textquotedblleft logically\textquotedblright{} correct value for execution

• Data dependencies
  • 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
Recap: How many dependencies do we have?

- How many pairs of data dependences are there in the following MIPS instructions?

```mips
lw $t0,0($a0)
add $t0,$t0, $t2
sw $t0,0($a0)
addi $a0,$a0, 4
bne $a0,$t1, LOOP
```

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
Recap: How many of data hazards?

- How many pairs of instructions in the following MIPS instructions will result in data hazards/stalls in a basic 5-stage MIPS pipeline?

lw $t0,0($a0)  
add $t0,$t0, $t2  
sw $t0,0($a0)  
addi $a0,$a0, 4  
bne $a0,$t1, LOOP

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
Recap: Solution 2: Data forwarding

- Add logics/wires to forward the desired values to the demanding instructions
- In our five stage pipeline — if the instruction entering the EXE stage consumes a result from a previous instruction that is entering MEM stage or WB stage
  - A source of the instruction entering EXE stage is the destination of an instruction entering MEM/WB stage
  - The previous instruction must be an instruction that updates register file
Pipelined processor with Data Forwarding
How many pairs of instructions in the following MIPS instructions will result in data hazards/stalls in a basic 5-stage MIPS pipeline with “full” data forwarding?

1. `lw   $t0,0($a0)`
2. `add  $t0,$t0, $t2`
3. `sw   $t0,0($a0)`
4. `addi $a0,$a0, 4`
5. `bne  $a0,$t1, LOOP`

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

• Data Hazard (cont.)
• Control Hazard and Branch Prediction (cont.)
Pipelined processor with Data Forwarding

**Instruction Memory**
- Instruction Address [31:0]
- Instruction [31:0]

**Register File**
- Read Reg 1
- Read Data 1
- Read Reg 2
- Read Data 2
- Write Reg
- Write Data

**Data Memory**
- Data Address [31:0]
- Data[31:0]
- Write Data[31:0]

**Control**
- Instruction [31-26]
- Instruction [25-21]
- Instruction [20-16]
- Instruction [15-11]
- Instruction [15-0]

**Forwarding**
- ForwardA
- ForwardB

**Hazard Detection**
- PCWrite
- IF/IDWrite
- Control
- Branch
- MemoryRead
- MemoryWrite
- ALUOp
- ALUSrc
- MemoryRead
- MemoryWrite
- RegWrite
- MemToReg

**IF/ID**
- Instruction Memory
- Instruction [31:0]

**ID/EX**
- Instruction [25-21]
- Instruction [20-16]
- Instruction [15-11]
- Instruction [15-0]

**EX/MEM**
- Instruction [5-0]
- Instruction [5-0]

**MEM/WB**
- Instruction [5-0]
Problems with data forwarding

- What if our pipeline gets deeper? — Considering a newly designed pipeline where memory stage is split into 2 stages and the memory access finishes at the 2nd memory stage. By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

1. \texttt{lw} $t0,0($a0)
2. \texttt{add} $t0,$t0, $t2
3. \texttt{sw} $t0,0($a0)
4. \texttt{addi} $a0,$a0, 4
5. \texttt{bne} $a0,$t1, LOOP

We are not making progress
• Both version A and B produces the same output. Without compiler optimization, which version of code would have significantly better performance?
  A. Version A
  B. Version B
  C. They are about the same (less than 10% difference)

— the data dependency on data[index] prevents pipeline parallelism
The effect of code optimization

• By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

1. lw $t0,0($a0)
2. add $t0,$t0, $t2
3. sw $t0,0($a0)
4. addi $a0,$a0, 4
5. bne $a0,$t1, LOOP

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered
The effect of code optimization

• By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

1. lw  $t0,0($a0)
2. add  $t0,$t0, $t2
3. sw  $t0,0($a0)
4. addi  $a0,$a0, 4
5. bne  $a0,$t1, LOOP

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered
The effect of code optimization

• By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered
Control Hazards
The impact of control hazards

• Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards. When there is a branch, we disable the instruction fetch and insert no-ops until we can determine the PC. 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
The impact of control hazards

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards. When there is a branch, we disable the instruction fetch and insert no-ops until we can determine the PC. 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
The impact of control hazards

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards. When there is a branch, we disable the instruction fetch and insert no-ops until we can determine the PC. 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

\[
1 + 20\% \times 2 = 1.4
\]
Why can’t we proceed without stalls/no-ops?

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
  ① The target address when branch is taken is not available for instruction fetch stage of the next cycle
  ② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
  ③ The branch outcome cannot be decided until the comparison result of ALU is not out
  ④ The next instruction needs the branch instruction to write back its result
A. 0
B. 1
C. 2
D. 3
E. 4
Why can’t we proceed without stalls/no-ops?

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
  ① The target address when branch is taken is not available for instruction fetch stage of the next cycle
  ② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
  ③ The branch outcome cannot be decided until the comparison result of ALU is not out
  ④ The next instruction needs the branch instruction to write back its result

A. 0
B. 1
C. 2
D. 3
E. 4
Why can’t we proceed without stalls/no-ops?

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
  ✓ The target address when branch is taken is not available for instruction fetch stage of the next cycle
  ✓ The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
  ✓ The branch outcome cannot be decided until the comparison result of ALU is not out
  ✓ The next instruction needs the branch instruction to write back its result

A. 0
B. 1
C. 2
D. 3
E. 4
Static Prediction
Why can’t we proceed without stalls/no-ops?

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
  1. The target address when branch is taken is not available for instruction fetch stage of the next cycle
  2. The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
  3. The branch outcome cannot be decided until the comparison result of ALU is not out
  4. The next instruction needs the branch instruction to write back its result

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

Let’s predict “not-taken” since we don’t need the target address
Statically predict not-taken

- Always predict the next PC is PC+4

```assembly
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)  
```

flush the instructions fetched incorrectly
What's the overall branch prediction (include both branches) accuracy for this nested for loop?

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

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%
What's the overall branch prediction (include both branches) accuracy for this nested for loop?

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

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%
What's the overall branch prediction (include both branches) accuracy for this nested for loop?

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

(assume all states started with 00)

A. \(~25\%\)  
B. \(~33\%\)  
C. \(~50\%\)  
D. \(~67\%\)  
E. \(~75\%\)  

\[
1 + 75\% \times (20\% \times 2) = 1.3
\]

For branch Y, almost 0%,  
For branch X, only 50%
Dynamic Branch Prediction
Why can’t we proceed without stalls/no-ops?

- How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
  - The target address when branch is taken is not available for instruction fetch stage of the next cycle
  - The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
  - The branch outcome cannot be decided until the comparison result of ALU is not out
  - The next instruction needs the branch instruction to write back its result

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
A basic dynamic branch predictor

Branch Target Buffer

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
<td>10</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
<td>11</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
<td>00</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
<td>01</td>
</tr>
</tbody>
</table>
2-bit/Bimodal local predictor

- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
<td>10</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
<td>11</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
<td>00</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
<td>01</td>
</tr>
</tbody>
</table>
2-bit local predictor

\[ i = 0; \]
\[ \text{do } \{ \]
\[ \quad \text{sum} += a[i]; \]
\[ \} \text{ while}(++i < 10); \]

<table>
<thead>
<tr>
<th>i</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4-9</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

90% accuracy!

\[ CPI_{\text{average}} = 1 + 20\% \times 10\% \times 2 = 1.04 \]
2-bit local predictor

• What's the overall branch prediction (include both branches) accuracy for this nested for loop?

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

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%
2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

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

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%
2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

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

(assume all states started with 00)

- A. ~25%
- B. ~33%
- C. ~50%
- D. ~67%
- E. ~75%

For branch Y, almost 100%, For branch X, only 50%
What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```java
i = 0;
do {  
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y
(assume all states started with 00)

Can we do a better job?

(For branch Y, almost 100%, For branch X, only 50%)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%
Announcements

• Plan your time carefully! — Time management is a skill that could be more useful than all other things you learned from CSE141/L
• Assignment #2 — due this evening
• Assignment #3 — due this Wednesday
• Midterm
  • Thursday during the lecture — make sure that you’re available, we will use Zoom
  • The lecture on Wednesday will provide a review, highlight, sample midterm
• All lab deadlines are “soft”, except for Lab 5/6
Computer Science & Engineering

Now playing — Everywhere (Michelle Branch)

つづく