CSE 141 – Computer Architecture
Summer Session 1, 2005
Lectures 9

Pipelining Hazards

Pramod V. Argade
July 20, 2005
CSE141: Introduction to Computer Architecture

Instructor: Pramod V. Argade (p2argade@cs.ucsd.edu)
Office Hour:
  Tue. 7:30 - 9:00 (Center 105)
  Wed. 5:00 - 6:00 PM (HSS 1330)
  By Appointment

TAs:
  Chengmo Yang: c5yang@cs.ucsd.edu
  Wenjing Rao: wrao@cs.ucsd.edu

Lecture: Mon/Wed. 6:00 - 8:50 PM, HSS 1330

Textbook: Computer Organization & Design
  The Hardware Software Interface, 3rd Edition.
  Authors: Patterson and Hennessy

Web-page: http://www.cse.ucsd.edu/classes/su05/cse141
<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Time</th>
<th>Room</th>
<th>Topic</th>
<th>Quiz topic</th>
<th>Homework Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mon. 6/27</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Introduction, Ch. 1 ISA, Ch. 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Wed. 6/29</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Arithmetic, Ch. 3 ISA Ch. 2</td>
<td>ISA Ch. 2</td>
<td>#1</td>
</tr>
<tr>
<td></td>
<td>Mon. 7/4</td>
<td>No Class</td>
<td></td>
<td>July 4th Holiday</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Wed. 7/6</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Performance, Ch. 4 Single-cycle CPU Ch. 5</td>
<td>Arithmetic Ch. 3</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>Mon. 7/11</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Single-cycle CPU Ch. 5 Cont. Multi-cycle CPU Ch. 5</td>
<td>Performance Ch. 4</td>
<td>#3</td>
</tr>
<tr>
<td>5</td>
<td>Tue. 7/12</td>
<td>7:30 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Multi-cycle CPU Ch. 5 Cont. (July 4th make up class)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Wed. 7/13</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Single and Multicycle CPU Examples and Review for Midterm</td>
<td>Single-cycle CPU Ch. 5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Mon. 7/18</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Mid-term Exam Exceptions</td>
<td></td>
<td>#4</td>
</tr>
<tr>
<td>8</td>
<td>Tue. 7/19</td>
<td>7:00 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Pipelining Ch. 6 (July 4th make up class)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Wed. 7/20</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Hazards, Ch. 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Mon. 7/25</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Memory Hierarchy &amp; Caches Ch. 7</td>
<td>Hazards Ch. 6</td>
<td>#5</td>
</tr>
<tr>
<td>11</td>
<td>Wed. 7/27</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Virtual Memory, Ch. 7 Course Review</td>
<td>Cache Ch. 7</td>
<td>#6</td>
</tr>
<tr>
<td>12</td>
<td>Sat. 7/30</td>
<td>TBD</td>
<td>TBD</td>
<td>Final Exam</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Announcements

- **Reading Assignment**
  - Chapter 5. The Processor: Datapath and Control
    Sections 5.6
  - Chapter 6. Enhancing Performance with Pipelining
    Sections 6.1 - 6.10

- **Homework 5: Due Mon., July 25th in class**
  5.49, 5.50
  6.1, 6.2, 6.4, 6.6, 6.15, 6.17, 6.20, 6.22, 6.23, 6.31, 6.32

- **Quiz**
  **When:** Mon., July 25th
  **Topic:** Pipelining and Hazards
Announcements

• **Reading Assignment**
  – Chapter 5. The Processor: Datapath and Control
    Sections 5.6
  – Chapter 6. Enhancing Performance with Pipelining
    Sections 6.1 - 6.10

• **Homework 5: Due Mon., July 25th in class**
  5.49, 5.50
  6.1, 6.2, 6.4, 6.6, 6.15, 6.17, 6.20, 6.22, 6.23, 6.31, 6.32

• **Quiz**
  **When:** Mon., July 25th
  **Topic:** Pipelining and Hazards
Would our pipeline design work in any case?

- What happens when...
  - add $3, $10, $11
  - lw $8, 1000($3)
  - sub $11, $8, $7
Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

- Result of SUB instruction not available until CC5 or later!
Software Solutions to Data Hazards

- Have compiler guarantee no hazards
  - Rearrange code to remove hazard
    - Not possible every time

- Insert “nops”
  - Where do we insert the “nops”?  
    - `sub $2, $1, $3`
    - `and $12, $2, $5`
    - `or $13, $6, $2`
    - `add $14, $2, $2`
    - `sw $15, 100($2)`

- Problem: Data hazards are very common!
  - “nops” really slows us down!
Hardware Solutions to Data Hazards

- Stall the pipeline (insert bubbles)
  - Data hazards are too common
    - Same as “nops”
  - Severe performance hit

- Forward the data as soon as it is available
  - Modify the pipeline to forward (bypass data)
Forwarding

- Use temporary results, don’t wait for them to be written
Forwarding

a. No forwarding

b. With forwarding

Forwarding unit
Reducing **EX** Data Hazards Through Forwarding
Reducing **EX** Data Hazards Through Forwarding

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

Pramod Argade

CSE 141, Summer Session I, 2005
Reducing **MEM** Data Hazards Through Forwarding

[Diagram of a pipeline with forwarding unit]
Reducing **MEM** Data Hazards Through Forwarding

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.Register**Rd** = ID/EX.Register**Rs**)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.Register**Rd** = ID/EX.Register**Rt**)) ForwardB = 01
Simultaneous EX/MEM Forwarding

- Consider following code

\[
\begin{align*}
\text{add} &\quad & \text{$1$, $1$, $2$} \\
\text{add} &\quad & \text{$1$, $1$, $3$} \\
\text{add} &\quad & \text{$1$, $1$, $4$} \\
\text{...} &
\end{align*}
\]
Simultaneous EX/MEM Forwarding

- Consider following code
  
  \[
  \text{add } $1, $1, $2 \\
  \text{add } $1, $1, $3 \\
  \text{add } $1, $1, $4 \\
  \ldots
  \]

- Must forward from MEM stage

- Disable WB stage forwarding
  
  \[
  \begin{align*}
  \text{if (MEM/WB.RegWrite} & \text{ and (MEM/WB.RegisterRd } \neq 0) \\
  \text{and } ?? & \text{ and (MEM/WB.RegisterRd } = \text{ ID/EX.RegisterRs}) \text{ ForwardA } = 01 \\
  \text{if (MEM/WB.RegWrite} & \text{ and (MEM/WB.RegisterRd } \neq 0) \\
  \text{and } ?? & \text{ and (MEM/WB.RegisterRd } = \text{ ID/EX.RegisterRt}) \text{ ForwardB } = 01
  \end{align*}
  \]
Forwarding in Action

sub $1, $12, $3 and $12, $3, $4  add $3, $8, $11  Memory Access  Write Back

sub $1, $12, $3 and $12, $3, $4  add $3, $8, $11  Memory Access  Write Back
Forwarding in Action

Instruction Fetch

sub $1, $12, $3 and $12, $3, $4 add $3, $8, $11 Write Back
Forwarding in Action

Instruction
Fetch

sub $1, $12, $3  and $12, $3, $4  add $3, $8, $11  Write Back

sub $1, $12, $3
add $3, $8, $11
and $12, $3, $4

Pramod Argade  CSE 141, Summer Session I, 2005
Forwarding in Action

sub $1, $12, $3 and $12, $3, $4 add $3, $9, $11
Forwarding in Action

sub $1, $12, $3 and $12, $3, $4 add $3, $9, $11
Data Hazard: Load followed by Store

lw $2, 10($1)

st $2, 0x1000($5)
or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

This forwarding can be done but is there a forwarding path?
M⇒M Forwarding for LW⇒SW
(HW Exercise 6.20 in the Textbook)
Forwarding does not eliminate Data Hazard in all cases

- Consider this code:

```assembly
lw $2, 10($1)
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```
Data Hazard: Load followed by R-type

lw $2, 10($1) and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Eliminating Data Hazards via Forwarding and stalling

- lw $2, 10($1) and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
Pipeline Interlocks

- Not all data hazards can be handled by forwarding

- Pipeline Interlock or Hazard Detection Unit
  - detects a hazard and stalls the pipeline until the hazard is clear

- A stall creates a pipeline bubble:
  - Preventing the IF and ID stages from proceeding
    - don’t write the PC (PCWrite = 0)
    - don’t rewrite IF/ID register (IF/IDWrite = 0)
  - Inserting “nops”
    - set all control signals propagating to EX/MEM/WB to zero (inserts a no-op instruction)
Hazard Detection Unit

- Keeping an instruction in the same stage is called pipeline stall

```java
if (ID/EX.MemRead and 
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or 
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    then stall the pipeline ➔ PCWrite = 0, IF/IDWrite = 0, EX/M/WB = 0
```
Hazard Detection Unit

and $4, $2, $5

$w\,\$2, \,20($1)$
Hazard Detection Unit

and $4, S2, S5

bubble

lw $2, 20($1)
Hazard Detection Unit

\[ \text{and } \#4, \#2, \#5 \quad \text{bubble} \quad \text{\texttt{lw } \#2, 20(\#1)} \]
Data Hazard Key Points

- Pipelining provides high throughput
- Data dependencies cause *data hazards*
- Data hazards can be solved by:
  - Software (nops)
  - Hardware data forwarding
  - Hardware pipeline stalling
- Our processor, and indeed all modern processors, use a combination of forwarding and stalling
Announcements

● **Reading Assignment**
  – Chapter 5. The Processor: Datapath and Control
    Sections 5.6
  – Chapter 6. Enhancing Performance with Pipelining
    Sections 6.1 - 6.10

● **Homework 5: Due Mon., July 25th in class**
  5.49, 5.50
  6.1, 6.2, 6.4, 6.6, 6.15, 6.17, 6.20, 6.22, 6.23, 6.31, 6.32

● **Quiz**
  **When:** Mon., July 25th
  **Topic:** Pipelining and Hazards
Control Hazards
Conditional Branches in a Pipeline

- In a program flow, data computed by certain instructions is used to determine next instruction to execute
  - using conditional branches

- In a pipelined processor, conditional branches result in control hazards

```
sub $2, $2, $5
and $6, $2, $4
beq $6, $8, L9

L9: and $x, $y, $z
sub $p, $q, $r
```
Pipelined Datapath and Control

Decision about whether to branch doesn’t occur until the MEM pipeline stage
Impact of a Branch Instruction on the Pipeline

Program execution order (in instructions)

40 beq $1, $3, 7

44 and $12, $2, $5

48 or $13, $6, $2

52 add $14, $2, $2

72 lw $4, 50($7)

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9

IM Reg

IM DM

IM DM

IM DM

IM DM

IM DM

Decision about whether to branch doesn’t occur until the MEM pipeline stage

Can you explain why the PC of the target instruction is 72?
Dealing With Branch Hazards

● Software
  – Insert nops,
  – Insert instructions that get executed either way (delayed branch).

● Hardware
  – Stall until you know which direction
    ➢ 3 cycles wasted for every branch
  – Guess which direction
    ➢ assume not taken (easiest)
    ➢ more educated guess based on history (requires that you know it is a branch before it is even decoded!)
  – Ignore the branch for a cycle (branch delay slot)
Branch Hazards

When we decide to branch, other instructions are in the pipeline!
Stalling for Branch Hazards: Assume the branch is taken

beq $4, $0, there

and $12, $2, $5

or ...

add ...

sw ...

Wastes cycles if branch is not taken
Assume Branch *Not Taken*

- Same performance as stalling when you’re wrong
- This is preferred approach
There is a 3 cycle penalty if a branch is taken
How could we reduce this penalty?
Resolving Branch in ID Stage, and Flushing if Branch is Taken

Note: Forwarding paths and muxes have to be added before registers are compared in ID stage
- Forwarding path from EX/MEM to IF/ID
- Forwarding path from MEM/WB to IF/ID
Resolving Branch in ID Stage, and Flushing if Branch is Taken

Note: Forwarding paths and muxes have to be added before registers are compared in ID stage
- Forwarding path from EX/MEM to IF/ID
- Forwarding path from MEM/WB to IF/ID

Problem: What if instruction immediately preceding branch writes to the required register?
- The pipeline must be stalled for one clock cycle in ID stage
Reducing the delay of branches

- Resolve the branch in ID stage
  - Move register compare in ID stage
  - Provide data forwarding
    - Ensure that most recent register values are used in ID stage
    - Add necessary forwarding muxes and paths

- Implement faster logic to compare registers
  - Current ALU approach
    - Subtract the two registers and check whether the result is zero
    - Slow!
  - Faster approach
    - Exclusive OR the two registers. OR result bits to check whether the result is zero
    - Fast, since no carry propagation
Flush Instructions in the Pipe if a Branch is Taken

- Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC).
  - Similar to a pipeline bubble...
  - The difference is that we need to be able to insert those bubbles later in the pipeline

- Flushing an instruction on a taken branch
  - Must flush the instruction being fetched in IF stage using IF.Flush signal, which changes all instruction fields to zero
    - SLL $0, $0, 0 is equivalent to NOP
  - Let the instruction fields percolate through the pipeline
Branch is Taken

36 sub $10, $4, $8
40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
...

72 lw $4, 50($7)

Branch stall reduced from 3 cycles to 1 cycle!
Eliminating the 1 Cycle Branch Stall

- There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?

- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.

- The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not!

- This works great for this implementation of the architecture, but becomes a permanent part of the ISA.

- What about the MIPS R10000, which has a 5-cycle branch penalty, and executes 4 instructions per cycle?
Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.
Scheduling Branch Delay Slot

The branch delay slot is only useful if you can find something to put there. If you can’t find anything, you must put a *nop* to insure correctness.

For b and c, $t4$ must be an unused temporary register.
Importance of efficient processing of branches

- Our implementation assumes “branch not taken”
  - Move branch resolution to ID stage
  - Flush instruction in IF stage if the branch is taken
- 15 to 20% of all instructions are branches
- MIPS
  - branch stall of 1 cycle, 1 instruction issued per cycle
  - delayed branch
- Recent processors
  - 3-4 cycle hazard, 1-2 instructions issued per cycle
  - cost of branch mis-prediction goes up
- Pentium Pro
  - 12+ cycle misprediction penalty, 3 instructions issued per cycle
  - HUGE penalty for mispredicting a branch
  - 36+ issue slots wasted
Predicting Branch Direction

● Easiest
  – always not taken, always taken
  – forward not taken, backward always taken
    ➢ Appropriate for loops
  – compiler predicted (branch likely, branch not likely)

● Save history of branch outcome
  – If the history is available, use it in the fetch stage
    ➢ Change PC accordingly
  – In the decode stage verify that the prediction was correct
    ➢ If not, set correct PC, flush pipeline, update history

● Next easiest
  – Record 1-bit history of whether the branch was taken or not
    ➢ 1-bit predictor
1-bit Pattern History Table (PHT)

- Uses low bits of branch address to choose an entry
- The entry has 1 branch prediction bit
- Size is small, e.g. 1 bits by N (e.g. 4K)

- Why not use all bits of branch address?
- What happens when the table is too small?

Prediction is incorrect twice for loops

```
Loop: lw $t0, 0($s1)    # $t0 = array element
   addu $t0, $t0, $s2   # add scalar in $s2
   sw $t0, 0($s1)      # store result
   addi $s1, $s1, -4    # decrement pointer
   bne $s1, $zero, Loop # branch $s1 !=0
```
Branch History Table

Note: Branch prediction logic is implemented in the IF stage
- Check PC in PHT if the instruction is a branch
- Modify next PC accordingly
- Validate branch prediction in ID stage
- If prediction was incorrect
  - Invalidate instructions in the pipeline
  - Start execution at correct PC
2-bit Branch Prediction Scheme

Branch prediction has to be incorrect twice before it is changed
Control Hazards -- Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.

- Control hazards are detected in hardware.

- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - branch prediction
  - branch delay slots
Exceptions
Exception Handling in the Pipeline

- Consider arithmetic overflow exception
  - add $1, $2, $1

- Extra hardware
  - Note: add is in EX stage
  - Flush instructions that follow add
    - In IF stage, assert IF.flush
    - In ID stage, use mux added for stall (OR ID.flush)
    - In EX stage, use EX.flush signal
  - **Prevent the ADD instruction from writing to $1**
    - Enables programmer to see the value of $1 that caused exception
  - Transfer control to PC = 0x8000 0180
  - Save PC in EPC (**need to subtract 4 from it before saving**)
  - Save exception cause in Cause Register
Datapath and Control for Exceptions
Exception Handling in a Pipeline

0x40 sub $11, $2, $4
0x44 and $12, $2, $5
0x48 or $13, $2, $6
0x4c add $1, $2, $1
0x50 slt $15, $6, $7
0x54 lw $16, 50($7)

0x80000180 sw $25, 1000($0)
0x80000184 sw $26, 1004($0)

Note: ALU overflow signal is input to the control unit

Pramod Argade
Issues in Handling an Exception

- Five instructions are active in the pipeline
- Multiple exceptions may occur
  - Earliest instruction that generated exception is interrupted
- Exceptions are detected in different stage of the pipeline
  - Undefined instruction is discovered in ID stage
  - Overflow is detected in EX stage
  - Kernel call (i.e. OS call) is detected in EX stage
- Precise exception
  - EPC saves PC of the instruction that caused exception
  - This is required for virtual memory
- Imprecise exception
  - EPC may not save PC of the instruction that caused exception
    - For ease of implementation
Summary: Hazards & Exceptions

- **Data Hazard**
  - Operand forwarding
  - Stall when forwarding cannot be done

- **Control Hazards**
  - Stall on branch
  - Predict branch

- **Exceptions**
  - May be detected in different stages of the pipeline
  - Must flush the offending instructions and all following it
  - Save context
  - Jump to exception handler
Announcements

● **Reading Assignment**
  – Chapter 5. The Processor: Datapath and Control
    Sections 5.6
  – Chapter 6. Enhancing Performance with Pipelining
    Sections 6.1 - 6.10

● **Homework 5**: Due Mon., July 25th in class
  5.49, 5.50
  6.1, 6.2, 6.4, 6.6, 6.15, 6.17, 6.20, 6.22, 6.23, 6.31, 6.32

● **Quiz**
  **When**: Mon., July 25th
  **Topic**: Pipelining and Hazards