Pipelining and Hazards

Prof. Hakim Weatherspoon
CS 3410, Spring 2015
Computer Science
Cornell University

See P&H Chapter: 4.6-4.8
Announcements

Prelim next week

Tuesday at 7:30.

Go to location based on netid

[a-g]* → MRS146: Morrison Hall 146
[h-l]* → RRB125: Riley-Robb Hall 125
[m-n]*→ RRB105: Riley-Robb Hall 105
[o-s]* → MVRG71: M Van Rensselaer Hall G71
[t-z]* → MVRG73: M Van Rensselaer Hall G73

Prelim reviews

TODAY, Tue, Feb 24 @ 7:30pm in Olin 255
Sat, Feb 28 @ 7:30pm in Upson B17

Prelim conflicts

Contact Deniz Altinbuken <deniz@cs.cornell.edu>
Announcements

Prelim1:

- **Time:** We will start at 7:30pm sharp, so come early
- **Location:** on previous slide
- **Closed Book**
  - Cannot use electronic device or outside material
- **Practice prelims are online in CMS**

- **Material covered everything up to end of this week**
  - Everything up to and including data hazards
  - Appendix B (logic, gates, FSMs, memory, ALUs)
  - Chapter 4 (pipelined [and non] MIPS processor with hazards)
  - Chapters 2 (Numbers / Arithmetic, simple MIPS instructions)
  - Chapter 1 (Performance)
  - HW1, Lab0, Lab1, Lab2, C-Lab0, C-Lab1
Goals for Today
RISC and Pipelined Processor: Putting it all together

Data Hazards

- Data dependencies
- Problem, detection, and solutions
  - (delaying, stalling, forwarding, bypass, etc)
- Hazard detection unit
- Forwarding unit

Next time

- Control Hazards
  What is the next instruction to execute if a branch is taken? Not taken?
MIPS Design Principles

Simplicity favors regularity
  • 32 bit instructions

Smaller is faster
  • Small register file

Make the common case fast
  • Include support for constants

Good design demands good compromises
  • Support for different type of interpretations/classes
### Recall: MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats

<table>
<thead>
<tr>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-type</strong></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
<tr>
<td><strong>I-type</strong></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td></td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>J-type</strong></td>
<td>6 bits</td>
<td></td>
<td></td>
<td>26 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recall: MIPS Instruction Types

Arithmetic/Logical
- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access
- load/store between registers and memory
- word, half-word and byte operations

Control flow
- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute
Recall: MIPS Instruction Types

Arithmetic/Logical
- ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
- ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
- MUL, DIV, MFLO, MTLO, MFHI, MTHI

Memory Access
- LW, LH, LB, LHU, LBU, LWL, LWR
- SW, SH, SB, SWL, SWR

Control flow
- BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
- J, JR, JAL, JALR, BEQL, BNEL, BLEZL, BGTZL

Special
- LL, SC, SYSCALL, BREAK, SYNC, COPROC
Pipelining

Principle:
Throughput increased by parallel execution
Balanced pipeline very important
Else slowest stage dominates performance

Pipelining:
• Identify *pipeline stages*
• Isolate stages from each other
• Resolve pipeline *hazards* (this and next lecture)
Basic Pipeline

Five stage “RISC” load-store architecture

1. Instruction fetch (IF)
   – get instruction from memory, increment PC
2. Instruction Decode (ID)
   – translate opcode into control signals and read registers
3. Execute (EX)
   – perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   – access memory if needed
5. Writeback (WB)
   – update register file
Pipelined Implementation

- Each instruction goes through the 5 stages
  - Each stage takes one clock cycle
    - So slowest stage determines clock cycle time
Time Graphs

Clock cycle 1 2 3 4 5 6 7 8 9

<table>
<thead>
<tr>
<th>add</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lw</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Instruction: add, lw
The pipeline achieves:

A) Latency: 1, throughput: 1 instr/cycle

B) Latency: 5, throughput: 1 instr/cycle

C) Latency: 1, throughput: 1/5 instr/cycle

D) Latency: 5, throughput: 5 instr/cycle

E) None of the above
**Time Graphs**

Latency: 5 cycles  
Throughput: 1 instr/cycle  
Concurrency: 5  

CPI = 1
Pipelined Implementation

• Each instruction goes through the 5 stages
  • Each stage takes one clock cycle
    • So slowest stage determines clock cycle time

• Stages must share information. How?
  • Add pipeline registers (flip-flops) to pass results between different stages
Pipelined Processor

memory

PC

new pc

register file

extend

control

alu

compute jump/branch targets

d_{in}  d_{out}

memory

Fetch

Decode

Execute

Memory

WB

+4
Pipelined Implementation

• Each instruction goes through the 5 stages
  • Each stage takes one clock cycle
    • So slowest stage determines clock cycle time

• Stages must share information. How?
  • Add pipeline registers (flip-flops) to pass results between different stages

And is this it?
Not quite....
Hazards

3 kinds

• Structural hazards
  – Multiple instructions want to use same unit

• Data hazards
  – Results of instruction needed before ready

• Control hazards
  – Don’t know which side of branch to take

Will get back to this
First, how to pipeline when no hazards
Pipelined Processor

Instruction Fetch

Memory

New pc

Instruction Decode

Extend

Control

Instruction File

alu

compute jump/branch targets

Jump/branch targets

Memory

Write-Back

IF/ID

ID/EX

EX/MEM

MEM/WB
Example: Sample Code (Simple)

```
add   r3, r1, r2;
nand  r6, r4, r5;
lw    r4, 20(r2);
add   r5, r2, r5;
sw    r7, 12(r3);
```
Example: Sample Code (Simple)

Assume eight-register machine

Run the following code on a pipelined datapath

\[
\begin{align*}
\text{add} & \quad r3 \quad r1 \quad r2 \quad ; \quad \text{reg} \ 3 = \text{reg} \ 1 + \text{reg} \ 2 \\
\text{nand} & \quad r6 \quad r4 \quad r5 \quad ; \quad \text{reg} \ 6 = \neg(\text{reg} \ 4 \ & \& \ \text{reg} \ 5) \\
\text{lw} & \quad r4 \quad 20 \ (r2) \quad ; \quad \text{reg} \ 4 = \ \text{Mem}[\text{reg}2+20] \\
\text{add} & \quad r5 \quad r2 \quad r5 \quad ; \quad \text{reg} \ 5 = \ \text{reg} \ 2 + \ \text{reg} \ 5 \\
\text{sw} & \quad r7 \quad 12(r3) \quad ; \quad \text{Mem}[\text{reg}3+12] = \ \text{reg} \ 7
\end{align*}
\]
At time 1, Fetch

`add r3 r1 r2`

Initial State

Time: 0

IF/ID | ID/EX | EX/MEM | MEM/WB

Data | dest | dest | dest
Fetch: nand 6 4 5

Time: 3

IF/ID

ID/EX

EX/MEM

MEM/WB
sw 7 12(3)

Fetch:
sw 7 12(3)

Time: 5
IF/ID

ID/EX

EX/MEM

MEM/WB

add 5 2 5
lw 4 20 (2)
nand 6 4 5
add 3 1 2

Data mem
Takeaway

Pipelining is a powerful technique to mask latencies and increase throughput

• Logically, instructions execute one at a time
• Physically, instructions execute in parallel
  – Instruction level parallelism

Abstraction promotes decoupling

• Interface (ISA) vs. implementation (Pipeline)
Hazards

See P&H Chapter: 4.7-4.8
Hazards

3 kinds

• Structural hazards
  – Multiple instructions want to use same unit

• Data hazards
  – Results of instruction needed before

• Control hazards
  – Don’t know which side of branch to take
Next Goal

What about data dependencies (also known as a data hazard in a pipelined processor)?

i.e. $\text{add} \ r3, r1, r2$

$\text{sub} \ r5, r3, r4$

Need to detect and then fix such hazards
Data Hazards

• register file reads occur in stage 2 (ID)
• register file writes occur in stage 5 (WB)
• next instructions may read values about to be written
  – i.e instruction may need values that are being computed further down the pipeline
  – *in fact, this is quite common*
add r3, r1, r2
sub r5, r3, r4
lw r6, 4(r3)
or r5, r3, r5
sw r6, 12(r3)
How many data hazards due to r3 only

add r3, r1, r2
sub r5, r3, r4
lw r6, 4(r3)
or r5, r3, r5
sw r6, 12(r3)

A) 1
B) 2
C) 3
D) 4
E) 5
1. add \( r_3, r_1, r_2 \)
2. sub \( r_5, r_3, r_4 \)
3. lw \( r_6, 4(r_3) \)
4. or \( r_5, r_3, r_5 \)
5. sw \( r_6, 12(r_3) \)

\[ r_3 = 10 \]
\[ r_3 = 20 \]
Clock cycle

- **Clock cycle 1:**
  - `r3 = 10` (Add `r3`, `r1`, `r2`)
  - `r3 = 10` (Subtract `r5`, `r3`, `r4`)

- **Clock cycle 2:**
  - `lw r6, 4(r3)`

- **Clock cycle 3:**
  - `or r5, r3, r5`
  - `r3 = 10`

- **Clock cycle 4:**
  - `r3 = 10`

- **Clock cycle 5:**
  - `or r5, r3, r5`
  - `r3 = 20`

- **Clock cycle 6:**
  - `sw r6, 12(r3)`

- **Clock cycle 7:**
  - `r6 = Mem[r3 + 4]`

- **Clock cycle 8:**
  - `r3 = 20`

- **Clock cycle 9:**
  - `r3 = 20`
Data Hazards

Data Hazards

• register file reads occur in stage 2 (ID)
• register file writes occur in stage 5 (WB)
• next instructions may read values about to be written

i.e. add \( r3, r1, r2 \)
    sub \( r5, r3, r4 \)

How to detect?
Detecting Data Hazards

IF/ID.\(Ra\) ≠ 0 &&
(IF/ID.\(Ra\)==ID/Ex.\(Rd\)
IF/ID.\(Ra\)==Ex/M.\(Rd\)
IF/ID.\(Ra\)==M/W.\(Rd\))
Data Hazards

Data Hazards

• register file reads occur in stage 2 (ID)
• register file writes occur in stage 5 (WB)
• next instructions may read values about to be written

How to detect? Logic in ID stage:

\[
\text{stall} = (\text{IF/ID.Ra} \neq 0 \land (\text{IF/ID.Ra} = \text{ID/EX.Rd} \lor \text{IF/ID.Ra} = \text{EX/M.Rd} \lor \text{IF/ID.Ra} = \text{M/WB.Rd})) \\
\quad \lor (\text{same for Rb})
\]
Detecting Data Hazards

- add r3, r1, r2
- sub r5, r3, r5
- or r6, r3, r4
- add r6, r3, r8

The diagram shows the pipeline stages of the processor with operations like add, sub, and or. The pipeline stages are IF/ID, ID/EX, EX/MEM, and MEM/WB. The detection of hazards occurs between ID/EX and EX/MEM stages.
Takeaway

Data hazards occur when an operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.
Next Goal

What to do if data hazard detected?
Resolving Data Hazards

What to do if data hazard detected?

A) Wait/Stall
B) Reorder in Software (SW)
C) Forward/Bypass
D) All the above
E) None. We will use some other method
Resolving Data Hazards

What to do if data hazard detected?

A) Wait/Stall
B) Reorder in Software (SW)
C) Forward/Bypass
D) All the above
E) None. We will use some other method

Discuss today
Next Goal

What to do if data hazard detected?

Options

• Nothing
  – Change the ISA to match implementation

• Stall
  – Pause current and subsequent instructions till safe

• Forward/bypass
  – Forward data value to where it is needed
Stalling

How to stall an instruction in ID stage

• prevent IF/ID pipeline register update
  – stalls the ID stage instruction

• convert ID stage instr into nop for later stages
  – innocuous “bubble” passes through pipeline

• prevent PC update
  – stalls the next (IF stage) instruction
Detecting Data Hazards

IF/ID
- add r3, r1, r2
- sub r5, r3, r5
- or r6, r3, r4
- add r6, r3, r8

ID/EX
- detect hazard

EX/MEM
- $d_{in}$, $d_{out}$
- mem

MEM/WB
- MemWr=0
- RegWr=0

PC

+4

WE=0
Stalling

time

Clock cycle

1  2  3  4  5  6  7  8

add r3, r1, r2

sub r5, r3, r5

or r6, r3, r4

add r6, r3, r8
### Stalling

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>IF</th>
<th>ID</th>
<th>Ex</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>4</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>5</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>6</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>7</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>8</td>
<td>IF</td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

- \( r_3 = 10 \)
- \( \text{add } r_3, r_1, r_2 \)
- \( r_3 = 20 \)
- \( \text{sub } r_5, r_3, r_5 \)
- \( \text{or } r_6, r_3, r_4 \)
- \( \text{add } r_6, r_3, r_8 \)

Stalls:
- \( 3 \) Stalls

Time:
- \( r_3 = 10 \)
- \( r_3 = 20 \)
NOP = If(IF/ID.rA ≠ 0 && (IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))

(MemWr=0
RegWr=0)

op

add r3,r1,r2

sub r5,r3,r5

or r6,r3,r4

/STALL

Inst mem

+4

PC

data mem

stall

Op WE Rd

IF/ID

Def

EX

M

Mem

Op WE Rd

Inst mem

Inst

inst mem
nop

sub r5, r3, r5

NOP = If(IF/ID.rA ≠ 0
&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))
NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))
r3 = 10
add r3, r1, r2
r3 = 20
sub r5, r3, r5
or r6, r3, r4
add r6, r3, r8

IF ID Ex M W

3 Stalls

Stalls
Stalling

How to stall an instruction in ID stage

• prevent IF/ID pipeline register update
  – stalls the ID stage instruction

• convert ID stage instr into nop for later stages
  – innocuous “bubble” passes through pipeline

• prevent PC update
  – stalls the next (IF stage) instruction
Takeaway

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards.

Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. *Bubbles in pipeline significantly decrease performance.
Next Goal: Resolving Data Hazards via Forwarding

What to do if data hazard detected?

A) Wait/Stall
B) Reorder in Software (SW)
C) Forward/Bypass
Forwarding
Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).

Three types of forwarding/bypass
• Forwarding from Ex/Mem registers to Ex stage (M→Ex)
• Forwarding from Mem/WB register to Ex stage (W→Ex)
• RegisterFile Bypass
Three types of forwarding/bypass

• Forwarding from Ex/Mem registers to Ex stage \((M \rightarrow Ex)\)
• Forwarding from Mem/WB register to Ex stage \((W \rightarrow Ex)\)
• RegisterFile Bypass
Three types of forwarding/bypass
- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W→Ex)
- RegisterFile Bypass
Forwarding Datapath 1

Ex/MEM to EX Bypass

- EX needs ALU result that is still in MEM stage
- Resolve:
  Add a bypass from EX/MEM.D to start of EX

How to detect? Logic in Ex Stage:

```plaintext
forward = (Ex/M.WE && EX/M.Rd != 0 &&
           ID/Ex.Ra == Ex/M.Rd)
|| (same for Rb)
```
Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass
Forwarding Datapath 1

add r3, r1, r2

sub r5, r3, r1
Forwarding Datapath 2
Mem/WB to EX Bypass

- EX needs value being written by WB
- Resolve:
  Add bypass from WB final value to start of EX

How to detect? Logic in Ex Stage:

\[
\text{forward} = (\text{M/WB.WE} \&\& \text{M/WB.Rd} \neq 0 \&\&
\text{ID/Ex.Ra} =\text{M/WB.Rd} \&\&
\text{not (Ex/M.WE} \&\& \text{Ex/M.Rd} \neq 0 \&\&
\text{ID/Ex.Ra} = \text{Ex/M.Rd})
\]

| | (same for Rb)

Check pg. 311
Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass
Forwarding Datapath 2

- add r3, r1, r2
- sub r5, r3, r1
- or r6, r3, r4
Register File Bypass

Register File Bypass

• Reading a value that is currently being written

Detect:

\(((Ra == MEM/WB.Rd) \lor (Rb == MEM/WB.Rd))\)
and (WB is writing a register)

Resolve:

Add a bypass around register file (WB to ID)
Better: (Hack) just negate register file clock
– writes happen at end of first half of each clock cycle
– reads happen during second half of each clock cycle
Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass
Forwarding Example

Clock cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td>IF</td>
<td></td>
<td></td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ex</td>
<td></td>
<td>ID</td>
<td>Ex</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- r3 = 10
- add r3, r1, r2
- r3 = 20
- sub r5, r3, r5
- or r6, r3, r4
- add r6, r3, r8

Example:
- r3 = 10: add r3, r1, r2
- r3 = 20: sub r5, r3, r5

Time

IF ID Ex M W
## Forwarding Example 2

<table>
<thead>
<tr>
<th></th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>add r3, r1, r2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>sub r5, r3, r4</strong></td>
<td></td>
</tr>
<tr>
<td><strong>lw r6, 4(r3)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>or r5, r3, r5</strong></td>
<td></td>
</tr>
<tr>
<td><strong>sw r6, 12(r3)</strong></td>
<td></td>
</tr>
</tbody>
</table>
Three types of forwarding/bypass
• Forwarding from Ex/Mem registers to Ex stage (M→Ex)
• Forwarding from Mem/WB register to Ex stage (W→Ex)
• Register File Bypass
Takeaway

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards. Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Bubbles (nops) in pipeline significantly decrease performance.

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register). Better performance than stalling.
Data Hazard Recap

Stall

• Pause current and all subsequent instructions

Forward/Bypass

• Try to steal correct value from elsewhere in pipeline
• Otherwise, fall back to stalling or require a delay slot

Tradeoffs?