CS 152 Computer Architecture and Engineering

Lecture 4 - Pipelining

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152
Last time in Lecture 3

• Microcoding became less attractive as gap between RAM and ROM speeds reduced
• Complex instruction sets difficult to pipeline, so difficult to increase performance as gate count grew
• Load-Store RISC ISAs designed for efficient pipelined implementations
  – Very similar to vertical microcode
  – Inspired by earlier Cray machines (more on these later)
• Iron Law explains architecture design space
  – Trade instructions/program, cycles/instruction, and time/cycle
An Ideal Pipeline

- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

*These conditions generally hold for industrial assembly lines, but instructions depend on each other!*
Pipelined RISC-V

To pipeline RISC-V:

- First build RISC-V without pipelining with CPI=1

- Next, add pipeline registers to reduce cycle time while maintaining CPI=1
Lecture 3: Unpipelined Datapath for RISC-V
## Lecture 3: Hardwired Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ImmSel</th>
<th>Op2Sel</th>
<th>FuncSel</th>
<th>MemWr</th>
<th>RFWen</th>
<th>WBSel</th>
<th>WASel</th>
<th>PCSel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>*</td>
<td>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUi</td>
<td>IType$_{12}$</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>LW</td>
<td>IType$_{12}$</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>SW</td>
<td>BsType$_{12}$</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQ$_{true}$</td>
<td>BrType$_{12}$</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQ$_{false}$</td>
<td>BrType$_{12}$</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>J</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>jabs</td>
</tr>
<tr>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>X1</td>
<td>jabs</td>
</tr>
<tr>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>rd</td>
<td>rind</td>
</tr>
</tbody>
</table>

- Op2Sel = Reg / Imm
- WASel = rd / X1
- WBSel = ALU / Mem / PC
- PCSel = pc+4 / br / rind / jabs
Pipelined Datapath

Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \quad (= t_{DM} \text{ probably}) \]

However, CPI will increase unless instructions are pipelined
“Iron Law” of Processor Performance

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Microarchitecture</th>
<th>CPI</th>
<th>Cycle time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Microcoded</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>3</td>
<td>Single-cycle unpipelined</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>4</td>
<td>Pipelined</td>
<td>1</td>
<td>short</td>
</tr>
</tbody>
</table>
CPI Examples

Microcoded machine

7 cycles, 5 cycles, 10 cycles

Inst 1, Inst 2, Inst 3

3 instructions, 22 cycles, CPI=7.33

Unpipelined machine

Inst 1, Inst 2, Inst 3

3 instructions, 3 cycles, CPI=1

Pipelined machine

Inst 1, Inst 2, Inst 3, Inst 3

3 instructions, 3 cycles, CPI=1

5-stage pipeline CPI≠5!!!
Technology Assumptions

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

Thus, the following timing assumption is reasonable

\[ t_{IM} \approx t_{RF} \approx t_{ALU} \approx t_{DM} \approx t_{RW} \]

A 5-stage pipeline will be the focus of our detailed design

- some commercial designs have over 30 pipeline stages to do an integer add!
5-Stage Pipelined Execution

- **I-Fetch (IF)**
- **Decode, Reg. Fetch (ID)**
- **Execute (EX)**
- **Memory (MA)**
- **Write-Back (WB)**

<table>
<thead>
<tr>
<th>time</th>
<th>instruction1</th>
<th>instruction2</th>
<th>instruction3</th>
<th>instruction4</th>
<th>instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>IF&lt;sub&gt;1&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;2&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;3&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;4&lt;/sub&gt;</td>
<td>IF&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>t1</td>
<td>ID&lt;sub&gt;1&lt;/sub&gt;</td>
<td>ID&lt;sub&gt;2&lt;/sub&gt;</td>
<td>ID&lt;sub&gt;3&lt;/sub&gt;</td>
<td>ID&lt;sub&gt;4&lt;/sub&gt;</td>
<td>ID&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>t2</td>
<td>EX&lt;sub&gt;1&lt;/sub&gt;</td>
<td>EX&lt;sub&gt;2&lt;/sub&gt;</td>
<td>EX&lt;sub&gt;3&lt;/sub&gt;</td>
<td>EX&lt;sub&gt;4&lt;/sub&gt;</td>
<td>EX&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>t3</td>
<td>MA&lt;sub&gt;1&lt;/sub&gt;</td>
<td>MA&lt;sub&gt;2&lt;/sub&gt;</td>
<td>MA&lt;sub&gt;3&lt;/sub&gt;</td>
<td>MA&lt;sub&gt;4&lt;/sub&gt;</td>
<td>MA&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>t4</td>
<td>WB&lt;sub&gt;1&lt;/sub&gt;</td>
<td>WB&lt;sub&gt;2&lt;/sub&gt;</td>
<td>WB&lt;sub&gt;3&lt;/sub&gt;</td>
<td>WB&lt;sub&gt;4&lt;/sub&gt;</td>
<td>WB&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5-Stage Pipelined Execution

Resource Usage Diagram

- **I-Fetch (IF)**
  - PC
  - Address
  - Instruction Memory

- **Decode, Reg. Fetch (ID)**
  - IR
  - Decoder
  - Register File

- **Execute (EX)**
  - ALU
  - Data Memory
  - GPRs

- **Memory (MA)**
  - Address
  - Read/Write

- **Write-Back (WB)**

**Resources**
- IF: I₁, I₂, I₃, I₄, I₅
- ID: I₁, I₁, I₂, I₃
- EX: I₁, I₁, I₂, I₃
- MA: I₁, I₁, I₂, I₃
- WB: I₁, I₁, I₂, I₃

**Time**
- t₀: I₁
- t₁: I₂
- t₂: I₃
- t₃: I₄
- t₄: I₅
- t₅: I₁
- t₆: I₂
- t₇: I₃
Pipelined Execution:
ALU Instructions

Not quite correct!

We need an Instruction Reg (IR) for each stage
Pipelined RISC-V Datapath

without jumps

Control Points Need to Be Connected
Instructions interact with each other in pipeline

• An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*

• An instruction may depend on something produced by an earlier instruction
  – Dependence may be for a data value → *data hazard*
  – Dependence may be for the next instruction’s address → *control hazard (branches, exceptions)*
Resolving Structural Hazards

- Structural hazard occurs when two instructions need same hardware resource at same time
  - Can resolve in hardware by stalling newer instruction till older instruction finished with resource

- A structural hazard can always be avoided by adding more hardware to design
  - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory

- Our 5-stage pipe has no structural hazards by design
  - Thanks to RISC-V ISA, which was designed for pipelining
Data Hazards

\[
x_4 \leftarrow x_1 \ ...
\]

\[
x_1 \leftarrow ... \\
\]

... 
\[
x_1 \leftarrow x_0 + 10 \\
x_4 \leftarrow x_1 + 17 \\
...
\]

\[
x_1 \text{ is stale. Oops!}
\]
Resolving Data Hazards (1)

Strategy 1:

Wait for the result to be available by freezing earlier pipeline stages → interlocks
Feedback to Resolve Hazards

• Later stages provide dependence information to earlier stages which can stall (or kill) instructions

• Controlling a pipeline in this manner works provided the instruction at stage \( i+1 \) can complete without any interference from instructions in stages 1 to \( i \) (otherwise deadlocks may occur)
Interlocks to resolve Data Hazards

Stall Condition

... x1 ← x0 + 10
x4 ← x1 + 17
...

January 31, 2012
CS152, Spring 2012
Stalled Stages and Pipeline Bubbles

\[
\begin{align*}
(I_1) & \ x_1 \leftarrow (x_0) + 10 \text{ IF}_1 \\
(I_2) & \ x_4 \leftarrow (x_1) + 17 \\
(I_3) & \\
(I_4) & \\
(I_5) & \\
\end{align*}
\]

\[\text{time}\]
\[t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots \]

\[\begin{array}{cccccccc}
(\text{IF}_1) & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 \\
(\text{IF}_2) & \text{IF}_2 \\
(\text{IF}_3) & \text{IF}_3 \\
(\text{IF}_4) & \text{IF}_4 \\
(\text{IF}_5) & \text{IF}_5 \\
\end{array}\]

\[\text{stalled stages}\]

\[
\begin{align*}
(I_1) & \ x_1 \leftarrow (x_0) + 10 \\
(I_2) & \ x_4 \leftarrow (x_1) + 17 \\
(I_3) & \\
(I_4) & \\
(I_5) & \\
\end{align*}
\]

\[\text{Resource Usage}\]

\[\text{time}\]
\[t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots \]

\[\begin{array}{cccccccc}
\text{IF} & I_1 & I_2 & I_3 & I_3 & I_3 & I_4 & I_5 \\
\text{ID} & I_1 & I_2 & I_2 & I_2 & I_3 & I_4 & I_5 \\
\text{EX} & I_1 & - & - & - & I_2 & I_3 & I_4 & I_5 \\
\text{MA} & I_1 & - & - & - & I_2 & I_3 & I_4 & I_5 \\
\text{WB} & I_1 & - & - & - & I_2 & I_3 & I_4 & I_5 \\
\end{array}\]

\[\Rightarrow \text{pipeline bubble}\]
Interlock Control Logic

Compare the source registers of the instruction in the decode stage with the destination register of the uncommitted instructions.
Interlock Control Logic

ignoring jumps & branches

Should we always stall if an rs field matches some rd?

not every instruction writes a register ⇒ we
not every instruction reads a register ⇒ re
## Source & Destination Registers

<table>
<thead>
<tr>
<th>Source(s)</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>rd ← rs1 func10 rs2</td>
</tr>
<tr>
<td>ALUI</td>
<td>rd ← rs1 op imm</td>
</tr>
<tr>
<td>LW</td>
<td>rd ← M [rs1 + imm]</td>
</tr>
<tr>
<td>SW</td>
<td>M [rs1 + imm] ← rs2</td>
</tr>
<tr>
<td>Bcond</td>
<td>rs1, rs2</td>
</tr>
<tr>
<td></td>
<td>true: PC ← PC + imm</td>
</tr>
<tr>
<td></td>
<td>false: PC ← PC + 4</td>
</tr>
<tr>
<td>J</td>
<td>PC ← PC + imm</td>
</tr>
<tr>
<td>JAL</td>
<td>x1 ← PC, PC ← PC + imm</td>
</tr>
<tr>
<td>JALR</td>
<td>rd ← PC, PC ← rs1 + imm</td>
</tr>
</tbody>
</table>
Deriving the Stall Signal

\[ C_{\text{dest}} \]
\[
ws = \text{Case opcode}
\]
\[
\text{JAL} \Rightarrow X1
\]
\[
\text{else} \Rightarrow \text{rd}
\]
\[
\text{we} = \text{Case opcode}
\]
\[
\text{ALU, ALUi, LW, JALR} \Rightarrow (ws \neq 0)
\]
\[
\text{JAL} \Rightarrow \text{on}
\]
\[
... \Rightarrow \text{off}
\]

\[ C_{\text{re}} \]
\[
\text{re1} = \text{Case opcode}
\]
\[
\text{ALU, ALUi, LW, SW, Bcond, JALR} \Rightarrow \text{on}
\]
\[
\text{J, JAL} \Rightarrow \text{off}
\]
\[
\text{re2} = \text{Case opcode}
\]
\[
\text{ALU, SW, Bcond} \Rightarrow \text{on}
\]
\[
... \Rightarrow \text{off}
\]

\[ C_{\text{stall}} \]
\[
\text{stall} = (rs_{1D} = ws_E).we_E +
(rs_{1D} = ws_M).we_M +
(rs_{1D} = ws_W).we_W . re1_D +
((rs_{2D} = ws_E).we_E +
(rs_{2D} = ws_M).we_M +
(rs_{2D} = ws_W).we_W ) . re2_D
\]

This is not the full story!
Hazards due to Loads & Stores

**Stall Condition**

Is there any possible data hazard in this instruction sequence?

What if $x_1 + 7 = x_3 + 5$?

$M[x_1 + 7] ← x_2$

$x_4 ← M[x_3 + 5]$

January 31, 2012
Load & Store Hazards

However, the hazard is avoided because our memory system completes writes in one cycle!

Load/Store hazards are sometimes resolved in the pipeline and sometimes in the memory system itself.

More on this later in the course.
**CS152 Administrivia**

- Quiz 1 on Feb 14 will cover PS1, Lab1, lectures 1-5, and associated readings.
- Section on Friday will review pipelining.
Resolving Data Hazards (2)

Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage \(\rightarrow\) bypass
Bypassing

Each *stall or kill* introduces a bubble in the pipeline

⇒ *CPI* > 1

A new datapath, i.e., *a bypass*, can get the data from the output of the ALU to its input

\[
\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
(I_1) x1 & \leftarrow & x0 + 10 & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 \\
(I_2) x4 & \leftarrow & x1 + 17 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 \\
(I_3) & & & \text{ID}_3 & \text{ID}_3 & \text{IDX}_3 & \text{ID}_3 & \text{ID}_3 \\
(I_4) & & & & \text{ID}_3 & \text{ID}_3 & \text{EX}_3 & \text{MA}_3 \\
(I_5) & & & & & \text{ID}_3 & \text{EX}_3 & \text{MA}_3 \\
\end{array}
\]
Adding a Bypass

When does this bypass help?

(I_1) \( x_1 \leftarrow x_0 + 10 \)  yes

(I_2) \( x_4 \leftarrow x_1 + 17 \)  yes

\( x_1 \leftarrow M[x_0 + 10] \)  no

\( x_4 \leftarrow x_1 + 17 \)  no

JAL 500  no

\( x_4 \leftarrow x_1 + 17 \)  no
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = ( (\text{rs}_1 = \text{ws}_E).\text{we}_E + (\text{rs}_1 = \text{ws}_M).\text{we}_M + (\text{rs}_1 = \text{ws}_W).\text{we}_W).\text{re}_1_D \\
+ ( (\text{rs}_2 = \text{ws}_E).\text{we}_E + (\text{rs}_2 = \text{ws}_M).\text{we}_M + (\text{rs}_2 = \text{ws}_W).\text{we}_W).\text{re}_2_D)
\]

\[
\text{ws} = \text{Case opcode} \\
\text{JAL} \Rightarrow X1 \\
\text{else} \Rightarrow \text{rd}
\]

\[
\text{we} = \text{Case opcode} \\
\text{ALU, ALUi, LW, JALR} \Rightarrow (\text{ws} \neq 0) \\
\text{JAL} \Rightarrow \text{on} \\
\text{...} \Rightarrow \text{off}
\]

\[
\text{ASrc} = (\text{rs}_1 = \text{ws}_E).\text{we}_E.\text{re}_1_D \quad \text{Is this correct?}
\]

No because only ALU and ALUi instructions can benefit from this bypass

Split \text{we}_E into two components: we-bypass, we-stall
Bypass and Stall Signals

Split $we_E$ into two components: $we$-bypass, $we$-stall

$we$-bypass$_E = \text{Case opcode}_E$
- $\text{ALU}, \text{ALUi} \Rightarrow (ws \neq 0)$
- $\ldots \Rightarrow \text{off}$

$we$-stall$_E = \text{Case opcode}_E$
- $\text{LW, JALR} \Rightarrow (ws \neq 0)$
- $\text{JAL} \Rightarrow \text{on}$
- $\ldots \Rightarrow \text{off}$

$\text{ASrc} = (rs_1_D = ws_E).we$-bypass$_E . \text{re}_1_D$

$\text{stall} = ((rs_1_D = ws_E).we$-stall$_E +$
- $(rs_1_D = ws_M).we_M + (rs_1_D = ws_W).we_W). \text{re}_1_D$
- $+((rs_2_D = ws_E).we_E + (rs_2_D = ws_M).we_M + (rs_2_D = ws_W).we_W). \text{re}_2_D$
Is there still a need for the stall signal?

\[
\text{stall} = (rs1_D = ws_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (ws_E \neq 0) \cdot \text{re1}_D \\
+ (rs2_D = ws_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (ws_E \neq 0) \cdot \text{re2}_D
\]
Pipeline CPI Examples

Measure from when first instruction finishes to when last instruction in sequence finishes.

3 instructions finish in 3 cycles
CPI = 3/3 = 1

3 instructions finish in 4 cycles
CPI = 4/3 = 1.33

3 instructions finish in 5 cycles
CPI = 5/3 = 1.67
Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

Guessed correctly ➔ do nothing

Guessed incorrectly ➔ kill and restart

.... We’ll later see examples of this approach in more complex processors.
Speculation that load value=zero

Speculation is based on the assumption that the load value is zero. The circuitry for speculation includes the following steps:

1. Load value is speculated to be zero.
2. The speculated zero value is compared against the actual load value.
3. If the load value is not zero, the pipeline is flushed.

The equation for speculation is:

Guess_zero = (rs1_D = ws_E) \cdot (opcode_E = LW_E) \cdot (ws_E \neq 0) \cdot re1_D

Also need to add circuitry to remember that this was a guess and flush pipeline if load not zero!

Not worth doing in practice – why?
Control Hazards

• What do we need to calculate next PC?
  – For Jumps
    » Opcode, PC and offset
  – For Jump Register
    » Opcode, Register value, and PC
  – For Conditional Branches
    » Opcode, Register (for condition), PC and offset
  – For all other instructions
    » Opcode and PC
      • have to know it’s not one of above
**PC Calculation Bubbles**

\[
\begin{align*}
(I_1) & \ x1 \leftarrow x0 + 10 \\
(I_2) & \ x3 \leftarrow x2 + 17 \\
(I_3) & \\
(I_4) & \\
\end{align*}
\]

\[
\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
\text{IF} & I_1 & - & I_2 & - & I_3 & - & I_4 & \\
\text{ID} & I_1 & - & I_2 & - & I_3 & - & I_4 & \\
\text{EX} & I_1 & - & I_2 & - & I_3 & - & I_4 & \\
\text{MA} & I_1 & - & I_2 & - & I_3 & - & I_4 & \\
\text{WB} & I_1 & - & I_2 & - & I_3 & - & I_4 & \\
\end{array}
\]

- ⇒ pipeline bubble
Speculate next address is PC+4

A jump instruction kills (not stalls) the following instruction

I1  096  ADD
I2  100  J 304
I3  104  ADD
I4  304  ADD

How?
Pipelining Jumps

To kill a fetched instruction -- Insert a mux before IR

Any interaction between stall and jump?

\[ \text{IRSrc}_D = \text{Case} \; \text{opcode}_D \]
\[ \text{J, JAL} \Rightarrow \text{bubble} \]
\[ \ldots \Rightarrow \text{IM} \]

I_1  096  ADD
I_2  100  J 304
I_3  104  ADD
I_4  304  ADD
Jump Pipeline Diagrams

Resource Usage

<table>
<thead>
<tr>
<th>time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>I_1</td>
<td>I_2</td>
<td>I_3</td>
<td>I_4</td>
<td>I_5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ID</td>
<td>I_1</td>
<td>I_2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>I_4</td>
<td>I_5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EX</td>
<td>I_1</td>
<td>I_2</td>
<td>I_1</td>
<td>I_2</td>
<td>-</td>
<td>I_4</td>
<td>I_5</td>
<td>-</td>
<td>I_5</td>
</tr>
<tr>
<td>MA</td>
<td>I_1</td>
<td>I_2</td>
<td>I_1</td>
<td>I_2</td>
<td>-</td>
<td>I_4</td>
<td>I_5</td>
<td>I_5</td>
<td>-</td>
</tr>
<tr>
<td>WB</td>
<td>I_1</td>
<td>I_2</td>
<td>I_1</td>
<td>I_2</td>
<td>-</td>
<td>I_4</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
</tr>
</tbody>
</table>

- \rightarrow \text{pipeline bubble}

(I_1) 096: ADD
(I_2) 100: J 304
(I_3) 104: ADD
(I_4) 304: ADD
Branch condition is not known until the execute stage

what action should be taken in the decode stage?

I_1  096  ADD
I_2  100  BEQ x1,x2 +200
I_3  104  ADD
I_4  304  ADD
Pipelining Conditional Branches

If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ stall signal is not valid

I₁  096  ADD
I₂  100  BEQ x₁, x₂ +200
I₃  104  ADD
I₄  304  ADD
Pipelining Conditional Branches

If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ stall signal is not valid

I₁: 096  ADD
I₂: 100  BEQZ x₁, x₂ +200
I₃: 104  ADD
I₄: 304  ADD

January 31, 2012  CS152, Spring 2012
Branch Pipeline Diagrams
(resolved in execute stage)

\[
\begin{array}{cccccccc}
\text{time} & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 \\
(I_1) & 096: \text{ADD} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 \\
(I_2) & 100: \text{BEQZ} +200 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 \\
(I_3) & 104: \text{ADD} & \text{IF}_3 & \text{ID}_3 & - & - & - \\
(I_4) & 108: & \text{IF}_4 & - & - & - & - \\
(I_5) & 304: \text{ADD} & \text{IF}_5 & \text{ID}_5 & \text{EX}_5 & \text{MA}_5 & \text{WB}_5 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Resource} & \text{Usage} & \text{time} & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 \\
\text{IF} & I_1 & I_2 & I_3 & I_4 & I_5 \\
\text{ID} & I_1 & I_2 & I_3 & - & I_5 \\
\text{EX} & I_1 & I_2 & - & - & I_5 \\
\text{MA} & I_1 & I_2 & - & - & I_5 \\
\text{WB} & I_1 & I_2 & - & - & I_5 \\
\end{array}
\]

- \Rightarrow \text{pipeline bubble}
Reducing Branch Penalty
(resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage – issues?
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252