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
Question of the Day

- Why a five stage pipeline?
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>OpSel</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>IType12</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>IType12</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>BsType12</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQ&lt;sub&gt;true&lt;/sub&gt;</td>
<td>BrType12</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQ&lt;sub&gt;false&lt;/sub&gt;</td>
<td>BrType12</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>yes</td>
<td>PC</td>
<td>rd</td>
<td>jabs</td>
<td></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  
WBSel = ALU / Mem / PC
PCSel = pc+4 / br / rind / jabs

Correction since L3: “J” has been removed
Pipelined Datapath

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

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

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

Time = Instructions * Cycles

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

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

Lecture 2
Lecture 3
Lecture 4
CPI Examples

Microcoded machine

7 cycles

5 cycles

10 cycles

Inst 1

Inst 2

Inst 3

Time

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

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 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)

$D$ecode, $R$eg. $F$etch (ID)

$E$xecute (EX)

$M$emory (MA)

$W$rite-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></td>
<td>IF$_1$</td>
<td>IF$_2$</td>
<td>IF$_3$</td>
<td>IF$_4$</td>
<td>IF$_5$</td>
</tr>
<tr>
<td></td>
<td>ID$_1$</td>
<td>ID$_2$</td>
<td>ID$_3$</td>
<td>ID$_4$</td>
<td>ID$_5$</td>
</tr>
<tr>
<td></td>
<td>EX$_1$</td>
<td>EX$_2$</td>
<td>EX$_3$</td>
<td>EX$_4$</td>
<td>EX$_5$</td>
</tr>
<tr>
<td></td>
<td>MA$_1$</td>
<td>MA$_2$</td>
<td>MA$_3$</td>
<td>MA$_4$</td>
<td>MA$_5$</td>
</tr>
<tr>
<td></td>
<td>WB$_1$</td>
<td>WB$_2$</td>
<td>WB$_3$</td>
<td>WB$_4$</td>
<td>WB$_5$</td>
</tr>
</tbody>
</table>

$\text{addr}$

$\text{rdata}$

$\text{PC}$

$0x4$

$\text{Addr}$

$\text{Inst. Memory}$

$\text{IR}$

$\text{Decode, Reg. Fetch (ID)}$

$\text{Execute (EX)}$

$\text{Memory (MA)}$

$\text{Write-Back (WB)}$
5-Stage Pipelined Execution

Resource Usage Diagram

I-Fetch (IF)

Decode, Reg. Fetch (ID)

Execute (EX)

Memory (MA)

Write-Back (WB)

Resources

IF

ID

EX

MA

WB

Time

t0

t1

t2

t3

t4

t5

t6

t7

Instructions

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
</tr>
<tr>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
</tr>
<tr>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
</tr>
<tr>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
</tr>
<tr>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
</tr>
</tbody>
</table>
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 pipeline has no structural hazards by design
  - Thanks to RISC-V ISA, which was designed for pipelining
Data Hazards

\[ x_4 \leftarrow x_1 \ldots \]

\[ x_1 \leftarrow \ldots \]

\[ x_1 \text{ is stale. Oops!} \]
How Would You Resolve This?

- What can you do if your project buddy has not completed their part, which you need to start?

- Three options
  - Wait (stall)
  - Speculate on the value of the deliverable
  - Bypass: ask them for what you need before his/her final deliverable
  - (ask him/her to work harder?)
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

$\ldots$

$x_1 \leftarrow x_0 + 10$

$x_4 \leftarrow x_1 + 17$

$\ldots$
Stalled Stages and Pipeline Bubbles

$t_0$ $t_1$ $t_2$ $t_3$ $t_4$ $t_5$ $t_6$ $t_7$ . . . .

$(I_1) \ x_1 \leftarrow (x_0) + 10$ \hspace{1cm} $IF_1$

$(I_2) \ x_4 \leftarrow (x_1) + 17$ \hspace{1cm} $IF_2$

$(I_3)$

$(I_4)$

$(I_5)$

Resource Usage

- $⇒$ pipeline bubble
Interlock Control Logic

Compare the *source registers* of the instruction in the decode stage with the *destination register* of the uncommitted instructions.
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>Field</th>
<th>Source(s)</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>func7</td>
<td>rs2 rs1</td>
<td>rs1, rs2</td>
</tr>
<tr>
<td>rs2</td>
<td>rs1</td>
<td>rd</td>
</tr>
<tr>
<td>func3</td>
<td>rd</td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>immediate12</td>
<td>rs1</td>
<td>rs1</td>
</tr>
<tr>
<td>rs1</td>
<td>func3</td>
<td>imm</td>
</tr>
<tr>
<td>imm</td>
<td>rs1</td>
<td>rd</td>
</tr>
<tr>
<td>Jump Offset[19:0]</td>
<td>rs1</td>
<td>rd</td>
</tr>
</tbody>
</table>

### ALU
- \( rd = rs1 \text{ func}10 \text{ rs2} \)

### ALUI/LW/JALR
- \( rd = rs1 \text{ op imm} \)

### LW
- \( rd = M[rs1 + \text{ imm}] \)

### SW/Bcond
- \( M[rs1 + \text{ imm}] \leq rs2 \)

### Bcond
- On **true**:
  - \( PC \leq PC + \text{ imm} \)
- On **false**:
  - \( PC \leq PC + 4 \)

### JAL
- \( x1 \leq PC, PC \leq PC + \text{ imm} \)

### JALR
- \( rd \leq PC, PC \leq rs1 + \text{ imm} \)
Deriving the Stall Signal

\[ C_{\text{dest}} \]
\[ \text{ws} = \text{rd} \]
\[ \text{we} = \text{Case opcode} \]
\[ \text{ALU, ALUi, LW, JALR} \Rightarrow \text{on} \]
\[ \ldots \Rightarrow \text{off} \]

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

\[ C_{\text{stall}} \]
\[ \text{stall} = ((\text{rs}_1 \text{D} = \text{ws}_E) \cdot \text{we}_E + \]
\[ (\text{rs}_1 \text{D} = \text{ws}_M) \cdot \text{we}_M + \]
\[ (\text{rs}_1 \text{D} = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_1 \text{D} + \]
\[ ((\text{rs}_2 \text{D} = \text{ws}_E) \cdot \text{we}_E + \]
\[ (\text{rs}_2 \text{D} = \text{ws}_M) \cdot \text{we}_M + \]
\[ (\text{rs}_2 \text{D} = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_2 \text{D} \]
Hazards due to Loads & Stores

Stall Condition

What if \( x_1 + 7 = x_3 + 5 \)?

Is there any possible data hazard in this instruction sequence?

\[
M[x_1 + 7] \leq x_2 \\
x_4 \leq M[x_3 + 5] 
\]
Load & Store Hazards

... M[x1+7] <= x2 x4 <= M[x3+5] ...

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 17 will cover PS1, Lab1, lectures 1-5, and associated readings.
Resolving Data Hazards (2)

Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage → bypass
Bypassing

Each stall or kill introduces a bubble in the pipeline

\[ \Rightarrow \text{CPI} > 1 \]

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

When does this bypass help?

(l1)  x1 <= x0 + 10  yes
(l2)  x4 <= x1 + 17

x1 <= M[x0 + 10]  x4 <= x1 + 17
JAL 500  x4 <= x1 + 17

no  no
The Bypass Signal
Deriving it from the Stall Signal

\[\text{stall} = \left( (\text{rs}_1 = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_1 = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_1 = \text{ws}_W) \cdot \text{we}_W \right) \cdot \text{re}_1 \]

\[+ \left( (\text{rs}_2 = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_2 = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_2 = \text{ws}_W) \cdot \text{we}_W \right) \cdot \text{re}_2 \]

\[\text{ws} = \text{rd} \]

\[\text{we} = \text{Case opcode} \]

ALU, ALUi, LW, JAL, JALR => on
...
=> off

\[\text{ASrc} = (\text{rs}_1 = \text{ws}_E) \cdot \text{we}_E \cdot \text{re}_1 \]

Is this correct?

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

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

Split $w_{E}$ into two components: we-bypass, we-stall

**we-bypass$_{E}$** = \textit{Case opcode$_{E}$} ALU, ALUi => on
... => off

**we-stall$_{E}$** = \textit{Case opcode$_{E}$}
LW, JAL, JALR => on
JAL => on
... => off

\[ \text{ASrc} = (rs_{1_{D}} = ws_{E}).we_{-}\text{bypass}_{E} \cdot \text{re}_{1_{D}} \]

\[ \text{stall} = ((rs_{1_{D}} = ws_{E}).we_{-}\text{stall}_{E} + (rs_{1_{D}} = ws_{M}).we_{M} + (rs_{1_{D}} = ws_{W}).we_{W}). \text{re}_{1_{D}} \]

\[ + ((rs_{2_{D}} = ws_{E}).we_{E} + (rs_{2_{D}} = ws_{M}).we_{M} + (rs_{2_{D}} = ws_{W}).we_{W}). \text{re}_{2_{D}} \]
Is there still a need for the stall signal?

\[
\text{stall} = (rs_{1D} = ws_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (ws_E != 0) \cdot \text{re}_{1D} + (rs_{2D} = ws_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (ws_E != 0) \cdot \text{re}_{2D}
\]
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 $\implies$ do nothing

Guessed incorrectly $\implies$ kill and restart

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

\[ \text{Guess}_\text{zero} = (r_{s1_D} = w_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (w_E \neq 0) \cdot r_{e1_D} \]

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 (and have to know it’s not one of above)
## PC Calculation Bubbles

**Time**

<table>
<thead>
<tr>
<th></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>I₁₃</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>I₁₃</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>I₁₃</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA</td>
<td>I₁₃</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>I₁₃</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resource Usage**

- **pipeline bubble**

(I₁) x₁ = x₀ + 10

(I₂) x₃ = x₂ + 17

(I₃)

(I₄)
Speculate next address is PC+4

A jump instruction kills (not stalls) the following instruction

I_1 096 ADD  
I_2 100 J 304  
I_3 104 ADD  
I_4 304 ADD

I_1 104 Inst Memory

I_2 E IR M

Jump?

bubble

stall

0x4 Add

PCSrc (pc+4 / jabs / rind/ br)

1/28/2016  CS152, Spring 2016
Pipelining Jumps

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

Any interaction between stall and jump?

IRSrc_D = Case opcode_D
JAL => bubble
...
=> IM

I_1 096 ADD
I_2 100 J 304
I_3 104 ADD
I_4 304 ADD

1/28/2016
CS152, Spring 2016
## Jump Pipeline Diagrams

### Time

<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>IF1</td>
<td>IF1</td>
<td>IF2</td>
<td>IF3</td>
<td>IF4</td>
<td>IF4</td>
<td>IF4</td>
<td>IF4</td>
<td>IF4</td>
<td>IF4</td>
</tr>
<tr>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
<td>ID1</td>
</tr>
<tr>
<td>EX1</td>
<td>EX1</td>
<td>EX2</td>
<td>EX2</td>
<td>EX2</td>
<td>EX2</td>
<td>EX2</td>
<td>EX2</td>
<td>EX2</td>
<td>EX2</td>
</tr>
<tr>
<td>MA1</td>
<td>MA1</td>
<td>MA2</td>
<td>MA2</td>
<td>MA2</td>
<td>MA2</td>
<td>MA2</td>
<td>MA2</td>
<td>MA2</td>
<td>MA2</td>
</tr>
<tr>
<td>WB1</td>
<td>WB1</td>
<td>WB2</td>
<td>WB2</td>
<td>WB2</td>
<td>WB2</td>
<td>WB2</td>
<td>WB2</td>
<td>WB2</td>
<td>WB2</td>
</tr>
</tbody>
</table>

### Resource Usage

- IF
- ID
- EX
- MA
- WB

### Pipeline Bubble

- $\Rightarrow$ *pipeline bubble*
Branch condition is not known until the execute stage. What action should be taken in the decode stage?
Pipelining Conditional Branches

If the branch is taken
- kill the two following instructions
- the instruction at the decode stage is not valid \(\Rightarrow\) stall signal is not valid
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

\[ \begin{align*}
I_1 &: 096 \quad \text{ADD} \\
I_2 &: 100 \quad \text{BEQ } x1, x2 +200 \\
I_3 &: 104 \quad \text{ADD} \\
I_4 &: 304 \quad \text{ADD}
\end{align*} \]
Branch Pipeline Diagrams
(resolved in execute stage)

\[ time \]
\[ t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots \]

(I_1) 096: ADD
\[ \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \]

(I_2) 100: BEQ +200
\[ \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \]

(I_3) 104: ADD
\[ \text{IF}_3 \quad \text{ID}_3 \quad - \quad - \quad - \]

(I_4) 108:
\[ \text{IF}_4 \quad - \quad - \quad - \quad - \]

(I_5) 304: ADD
\[ \text{IF}_5 \quad \text{ID}_5 \quad \text{EX}_5 \quad \text{MA}_5 \quad \text{WB}_5 \]

Resource Usage

- \[ \Rightarrow \] pipeline bubble
Question of the Day

- Why a five stage pipeline?
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