27 September – Homework #2 Due
Assignment on web page. Teams of 2-3.
Submit soft copies to Sakai.
Use Piazza for questions.

2 October – Class Discussion
Roughly one reading per class. Do not wait until the day before!

1. Srinivasan et al. “Optimizing pipelines for power and performance”
3. Palacharla et al. “Complexity-effective superscalar processors”
4. Yeh et al. “Two-level adaptive training branch prediction”
Pipelining

Latency = (Instructions / Program) x (Cycles / Instruction) x (Seconds / Cycle)

Performance Enhancement
- Increases number of cycles per instruction
- Reduces number of seconds per cycle

Instruction-Level Parallelism
- Begin with multi-cycle design
- When one instruction advances from stage-1 to stage=2, allow next instruction to enter stage-1.
- Individual instructions require the same number of stages
- Multiple instructions in-flight, entering and leaving at faster rate

<table>
<thead>
<tr>
<th>Multi-cycle</th>
<th>insn0.fetch</th>
<th>insn0.dec</th>
<th>insn0.exec</th>
<th>insn1.fetch</th>
<th>insn1.dec</th>
<th>insn1.exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelined</td>
<td>insn0.fetch</td>
<td>insn0.dec</td>
<td>insn0.exec</td>
<td>insn1.fetch</td>
<td>insn1.dec</td>
<td>insn1.exec</td>
</tr>
</tbody>
</table>
Ideal Pipelining

- All objects go through the same stages
- No resources shared between any two stages
- Equal propagation delay through all pipeline stages
- An object entering the pipeline is not affected by objects in other stages

- These conditions generally hold for industrial assembly lines
- But can an instruction pipeline satisfy the last condition?

Technology Assumptions

- Small, very fast memory (caches) backed by large, slower memory
- Multi-ported register file, which is slower than a single-ported one
- Consider 5-stage pipelined Harvard architecture
Practical Pipelining

Pipeline Overheads
- Each stage requires registers, which hold state/data communicated from one stage to next, incurring hardware and delay overheads
- Each stage requires partitioning logic into “equal” lengths
- Introduces diminishing marginal returns from deeper pipelines

Pipeline Hazards
- Instructions do not execute independently
- Instructions entering the pipeline depend on in-flight instructions or contend for shared hardware resources
Pipelining MIPS

First, build MIPS without pipelining
- Single-cycle MIPS datapath

Then, pipeline into multiple stages
- Multi-cycle MIPS datapath
- Add pipeline registers to separate logic into stages
- MIPS partitions into 5 stages
  - 1: Instruction Fetch (IF)
  - 2: Instruction Decode (ID)
  - 3: Execute (EX)
  - 4: Memory (MEM)
  - 5: Write Back (WB)
IF: IR ← mem[PC]; PC ← PC + 4;
ID: A ← Reg[IR_{rs}]; B ← Reg[IR_{rt}];
EX: Result $\leftarrow A \text{ op}_{IR_{op}} B$;  
MEM: WB $\leftarrow$ Result;  
WB: Reg[IR_{rd}] $\leftarrow$ WB
Visualizing the Pipeline

Time (in clock cycles)

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

Program execution order (in instructions)

IM  Reg  ALU  DM  Reg  IM  Reg  ALU  DM  Reg  IM  Reg  ALU  DM  Reg  IM  Reg  ALU  DM  Reg

© 2007 Elsevier, Inc. All rights reserved.
Hazards and Limits to Pipelining

Hazards prevent next instruction from executing during its designated clock cycle

Structural Hazards
- Hardware cannot support this combination of instructions.
- Example: Limited resources required by multiple instructions (e.g. FPU)

Data Hazards
- Instruction depends on result of prior instruction still in pipeline
- Example: An integer operation is waiting for value loaded from memory

Control Hazards
- Instruction fetch depends on decision about control flow
- Example: Branches and jumps change PC
A single memory port causes structural hazard during data load, instr fetch
Stall the pipeline, creating bubbles, by freezing earlier stages → interlocks
Use Harvard Architecture (separate instruction, data memories)
Data Hazards

Instruction depends on result of prior instruction still in pipeline
Data Hazards

Read After Write (RAW)
- Caused by a dependence, need for communication
- Instr-j tries to read operand before Instr-l writes it
  i: add r1, r2, r3
  j: sub r4, r1, 43

Write After Read (WAR)
- Caused by an anti-dependence and the re-use of the name “r1”
- Instr-j tries to write operand (r1) before Instr-l reads it
  i: add r4, r1, r3
  j: add r1, r2, r3
  k: mul r6, r1, r7

Write After Write (WAW)
- Caused by an output dependence and the re-use of the name “r1”
- Instr-j tries to write operand (r1) before Instr-l writes it
  i: sub r1, r4, r3
  j: add r1, r2, r3
  k: mul r6, r1, r7
Resolving Data Hazards

Strategy 1 – Interlocks and Pipeline Stalls
- Later stages provide dependence information to earlier stages, which can stall or kill instructions
- Works as long as instruction at stage $i+1$ can complete without any interference from instructions in stages 1 through $i$ (otherwise, deadlocks may occur)
Interlocks & Pipeline Stalls

(stalled stages)

<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>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>ID1</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>EX1</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>MA1</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>WB1</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>IF2</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>ID2</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>EX2</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>MA2</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>WB2</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>IF3</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>ID3</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>EX3</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>MA3</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>WB3</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>IF4</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>ID4</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>EX4</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>MA4</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>WB4</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>IF5</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>ID5</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>EX5</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>MA5</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
<tr>
<td>WB5</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l3</td>
<td>l5</td>
</tr>
</tbody>
</table>

Resource Usage

- IF
- ID
- EX
- MA
- WB

(l1) r1 ← (r0) + 10
(l2) r4 ← (r1) + 17
(l3)
(l4)
(l5)
**Interlocks & Pipeline Stalls**

**Example Dependence**

\[ r1 \leftarrow r0 + 10 \]

\[ r4 \leftarrow r1 + 17 \]
Interlock Control Logic

- Compare the source registers of instruction in decode stage with the destination registers of uncommitted instructions.

- Stall if a source register in decode matches some destination register?
  - No, not every instruction writes to a register.
  - No, not every instruction reads from a register.

- Derive stall signal from conditions in the pipeline.
Compare the source registers of the instruction in the decode stage (rs, rt) with the destination register of the uncommitted instructions (ws).
Should we always stall if RS/RT matches some WS? No, because not every instruction writes/reads a register. Introduce write/read enable signals (we/re)

ECE 552 / CPS 550
Source and Destination Registers

R-type: \begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{rd} & \text{func} \\
\end{array}

I-type: \begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate16} \\
\end{array}

J-type: \begin{array}{cccc}
\text{op} & \text{immediate26} \\
\end{array}

<table>
<thead>
<tr>
<th>instruction</th>
<th>source(s)</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU ( \text{rd} \leftarrow (\text{rs}) \text{ func (rt)} )</td>
<td>( \text{rs, rt} )</td>
<td>( \text{rd} )</td>
</tr>
<tr>
<td>ALUi ( \text{rt} \leftarrow (\text{rs}) \text{ op imm} )</td>
<td>( \text{rs} )</td>
<td>( \text{rt} )</td>
</tr>
<tr>
<td>LW ( \text{rt} \leftarrow M[(\text{rs}) + \text{imm}] )</td>
<td>( \text{rs} )</td>
<td>( \text{rt} )</td>
</tr>
<tr>
<td>SW ( M [(\text{rs}) + \text{imm}] \leftarrow (\text{rt}) )</td>
<td>( \text{rs, rt} )</td>
<td></td>
</tr>
<tr>
<td>BZ ( \text{cond (rs)} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>true: ( \text{PC} \leftarrow (\text{PC}) + \text{imm} )</td>
<td>( \text{rs} )</td>
<td></td>
</tr>
<tr>
<td>false: ( \text{PC} \leftarrow (\text{PC}) + 4 )</td>
<td>( \text{rs} )</td>
<td></td>
</tr>
<tr>
<td>J ( \text{PC} \leftarrow (\text{PC}) + \text{imm} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAL ( \text{r31} \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{PC}) + \text{imm} )</td>
<td></td>
<td>( \text{R31} )</td>
</tr>
<tr>
<td>JR ( \text{PC} \leftarrow (\text{rs}) )</td>
<td>( \text{rs} )</td>
<td></td>
</tr>
<tr>
<td>JALR ( \text{r31} \leftarrow (\text{PC}), \text{PC} \leftarrow (\text{rs}) )</td>
<td>( \text{rs} )</td>
<td>( \text{R31} )</td>
</tr>
</tbody>
</table>
Should we always stall if RS/RT matches some RD? No, because not every instruction writes/reads a register. Introduce write/read enable signals (we/re)
Deriving the Stall Signal

Cdest \( \rightarrow \) ws 
Case(\text{opcode})
- ALU: \( \rightarrow \) rd
- ALUi: \( \rightarrow \) rt
- JAL, JALR: \( \rightarrow \) R31

we 
Case(\text{opcode})
- ALU, ALUi, LW \( \rightarrow \) (ws \( \neq \) 0)
- JAL, JALR \( \rightarrow \) 1
- otherwise \( \rightarrow \) 0

Cre 
re1 
Case(\text{opcode})
- ALU, ALUi \( \rightarrow \) 1
- LW, SW, BZ \( \rightarrow \) 1
- JR, JALR \( \rightarrow \) 1
- J, JAL \( \rightarrow \) 0

re2 
Case(\text{opcode})
<< same as re1 but for register rt>>
Deriving the Stall Signal

Notation: [pipeline-stage][signal]
E.g., Drs – rs signal from decode stage
E.g., Ewe – we signal from execute stage

\[
\text{Cstall} \quad \text{stall-1} \leftarrow ( (\text{Drs} == \text{Ews}) \& \text{Ewe} \mid (\text{Drs} == \text{Mws}) \& \text{Mwe} \mid (\text{Drs} == \text{Wws}) \& \text{Wwe}) \& \text{Dre1}
\]

\[
\text{stall-2} \leftarrow ( (\text{Drt} == \text{Ews}) \& \text{Ewe} \mid (\text{Drt} == \text{Mws}) \& \text{Mwe} \mid (\text{Drt} == \text{Wws}) \& \text{Wwe}) \& \text{Dre2}
\]

\[
\text{stall} \leftarrow \text{stall-1} \mid \text{stall-2}
\]
Load/Store Data Hazards

\[ M[(r1)+7] \leftarrow (r2) \]
\[ r4 \leftarrow M[(r3)+5] \]

What is the problem here?
What if \( (r1)+7 == (r3)+5 \)?

Load/Store hazards may be resolved in the pipeline or may be resolved in the memory system. More later.
Resolving Data Hazards

Strategy 2 – Forwarding (aka Bypasses)

- Route data as soon as possible to earlier stages in the pipeline
- Example: forward ALU output to its input

\[
\begin{align*}
(I_1) & \quad r_1 & \leftarrow & \quad r_0 + 10 \\
(I_2) & \quad r_4 & \leftarrow & \quad r_1 + 17 \\
(I_3) & \\
(I_4) & \\
(I_5) & \\
\end{align*}
\]

\[
\begin{align*}
\text{time} & \\
\begin{array}{cccccccc}
(I_1) & \quad r_1 & \leftarrow & \quad r_0 + 10 \\
(I_2) & \quad r_4 & \leftarrow & \quad r_1 + 17 \\
(I_3) & \\
(I_4) & \\
(I_5) & \\
\end{array}
\end{align*}
\]
Deriving Forwarding Signals

This forwarding path only applies to the ALU operations...

\[
\text{Eforward} \quad \text{Case}(\text{Eopcode})
\]

\[
\begin{align*}
\text{ALU, ALUi} & \quad \text{Eforward} \leftarrow (\text{ws} \neq 0) \\
\text{otherwise} & \quad \text{Eforward} \leftarrow 0
\end{align*}
\]

...and all other operations will need to stall as before

\[
\text{Estall} \quad \text{Case}(\text{Eopcode})
\]

\[
\begin{align*}
\text{LW} & \quad \text{Estall} \leftarrow (\text{ws} \neq 0) \\
\text{JAL, JALR} & \quad \text{Estall} \leftarrow 1 \\
\text{otherwise} & \quad \text{Estall} \leftarrow 0
\end{align*}
\]

\[
\text{Asrc} \leftarrow (\text{Drs} == \text{Ews}) \land \text{Dre1} \land \text{Eforward}
\]

Remember to update stall signal, removing case covered by this forwarding path
Multiple Forwarding Paths
Multiple Forwarding Paths
Forwarding Hardware
Forwarding Loads/Stores
LD cannot forward (backwards in time) to DSUB. What is the solution?
Data Hazards and Scheduling

Try producing faster code for
- \( A = B + C; D = E - F; \)
- Assume \( A, B, C, D, E, \) and \( F \) are in memory
- Assume pipelined processor

<table>
<thead>
<tr>
<th>Slow Code</th>
<th>Fast Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Rb, b</td>
<td>Rb, b</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Rc, c</td>
<td>Rc, c</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
</tr>
<tr>
<td>Ra, Rb, Rc</td>
<td>Ra, Rb, Rc</td>
</tr>
<tr>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>a, Ra</td>
<td>a, Ra</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Re e</td>
<td>Re e</td>
</tr>
<tr>
<td>LW</td>
<td>LW</td>
</tr>
<tr>
<td>Rf, f</td>
<td>Rf, f</td>
</tr>
<tr>
<td>LW</td>
<td>SW</td>
</tr>
<tr>
<td>Rd, Re, Rf</td>
<td>Rd, Re, Rf</td>
</tr>
<tr>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>d, RD</td>
<td>d, RD</td>
</tr>
</tbody>
</table>
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)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)