27 September – Homework #2 Due
- Use blackboard forum for questions
- Attend office hours with questions
- Email for separate meetings

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”
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 ( Rb, b )</td>
<td>LW ( Rb, b )</td>
</tr>
<tr>
<td>LW ( Rc, c )</td>
<td>LW ( Rc, c )</td>
</tr>
<tr>
<td>ADD ( Ra, Rb, Rc )</td>
<td>LW ( Re, e )</td>
</tr>
<tr>
<td>SW ( a, Ra )</td>
<td>ADD ( Ra, Rb, Rc )</td>
</tr>
<tr>
<td>LW ( Re, e )</td>
<td>LW ( Rf, f )</td>
</tr>
<tr>
<td>LW ( Rf, f )</td>
<td>SW ( a, Ra )</td>
</tr>
<tr>
<td>SUB ( Rd, Re, Rf )</td>
<td>SUB ( Rd, Re, Rf )</td>
</tr>
<tr>
<td>SW ( d, RD )</td>
<td>SW ( d, RD )</td>
</tr>
</tbody>
</table>
Compiler Scheduling

Reduce stalls by moving instructions
- Basic pipeline scheduling eliminates back-to-back load-use pairs
- What are the limitations of scheduling?

Scheduling Scope
- Requires an independent instruction to place between load-use pairs
- Little scope for scheduling, 1-add, 3-ld/st

<table>
<thead>
<tr>
<th>Slow Code</th>
<th>Fast Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2, 4(sp)</td>
<td>ld r2, 4(sp)</td>
</tr>
<tr>
<td>ld r3, 8(sp)</td>
<td>ld r3, 8(sp)</td>
</tr>
<tr>
<td>add r3, r2, r1</td>
<td>add r3, r2, r1</td>
</tr>
<tr>
<td>st r1, 0(sp)</td>
<td>st r1, 0(sp)</td>
</tr>
</tbody>
</table>
Number of registers
- Registers hold “live” values
- Example code contains 7 different values, including sp
- Before: max 3 values live $\rightarrow$ 3 registers sufficient
- After: max 4 values live $\rightarrow$ 3 registers insufficient
- Original code re-uses r1 and r2, re-scheduling causes WAR violations

Before

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>r2, 4 (sp)</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>r1, 8 (sp)</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r1, r2, r1</td>
<td>#stall</td>
</tr>
<tr>
<td>st</td>
<td>r1, 0 (sp)</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>r2, 16 (sp)</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>r1, 20 (sp)</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>r2, r1, r1</td>
<td>#stall</td>
</tr>
<tr>
<td>st</td>
<td>r1, 12 (sp)</td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>r2, 4 (sp)</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>r1, 8 (sp)</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>r2, 16 (sp)</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>r1, 20 (sp)</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r1, r2, r1</td>
<td>#WAR</td>
</tr>
<tr>
<td>st</td>
<td>r1, 0 (sp)</td>
<td>#WAR</td>
</tr>
<tr>
<td>sub</td>
<td>r2, r1, r1</td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>r1, 12 (sp)</td>
<td></td>
</tr>
</tbody>
</table>
Compiler Scheduling

Alias Analysis
- Determine whether load/stores reference same memory locations
- Determines if loads/stores can be re-ordered
- Previous Examples: easy, all loads/stores use the same base register (sp)
- New Example: Can compiler tell that r8 == sp?

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2, 4 (sp)</td>
<td>ld r2, 4 (sp)</td>
</tr>
<tr>
<td>ld r3, 8 (sp)</td>
<td>ld r3, 8 (sp)</td>
</tr>
<tr>
<td>add r3, r2, r1 #stall</td>
<td>ld r5, 0 (r8)</td>
</tr>
<tr>
<td>st r1, 0 (sp)</td>
<td>add r3, r2, r1</td>
</tr>
<tr>
<td>ld r5, 0 (r8)</td>
<td>ld r6, 4 (r8)</td>
</tr>
<tr>
<td>ld r6, 4 (r8)</td>
<td>st r1, 0 (sp)</td>
</tr>
<tr>
<td>sub r5, r6, r4 #stall</td>
<td>sub r5, r6, r4</td>
</tr>
<tr>
<td>st r4, 8 (r8)</td>
<td>st r4, 8 (r8)</td>
</tr>
</tbody>
</table>
Control Hazards

Arises when calculating next program counter (PC)
- Pipeline stalls if required values not yet available

Jumps
- Jump (immediate) requires opcode, offset, current PC
- Jump (register) requires opcode, register value

Conditional Branches
- Requires opcode, current PC, register (for condition), offset

Sequential Successor Instructions
- Requires opcode, current PC
Program Counter Calculations

Identify change in control flow during decode
Determine whether \( I_1 \) changes control flow before fetching \( I_2 \)? No.

\[
\begin{array}{ccccccccccc}
\text{Time} & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & \ldots & \ldots \\
(I_1) & r_1 \leftarrow (r_0) + 10 & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 & \\
(I_2) & r_3 \leftarrow (r_2) + 17 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 & \\
(I_3) & \text{IF}_3 & \text{ID}_3 & \text{EX}_3 & \text{MA}_3 & \text{WB}_3 & \\
(I_4) & \text{IF}_4 & \text{ID}_4 & \text{EX}_4 & \text{MA}_4 & \text{WB}_4 & \\
\end{array}
\]

Resource Usage

\[
\begin{array}{ccccccccccc}
\text{time} & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & \ldots & \ldots \\
\text{IF} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 & \\
\text{ID} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 & \\
\text{EX} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 & \\
\text{MA} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 & \\
\text{WB} & I_1 & \text{nop} & I_2 & \text{nop} & I_3 & \text{nop} & I_4 & \\
\end{array}
\]
Speculate PC ← PC + 4

A jump instruction kills (not stalls) the following instruction. How?

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

PCS

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

addr

inst

Memory

addr

inst

Memory

A jump instruction kills (not stalls) the following instruction. How?
Pipelining Jumps

To kill a fetched instruction, add mux before IR to insert “nops”
case(opcodeD)

\[
\begin{align*}
\text{J, JAL:} & \quad \text{IR} \leftarrow \text{nop} \\
\text{otherwise:} & \quad \text{IR} \leftarrow \text{inst}
\end{align*}
\]

<table>
<thead>
<tr>
<th>addr</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>096</td>
<td>ADD</td>
</tr>
<tr>
<td>100</td>
<td>J 304</td>
</tr>
<tr>
<td>104</td>
<td>ADD</td>
</tr>
<tr>
<td>304</td>
<td>ADD</td>
</tr>
</tbody>
</table>

ECE 552 / CPS 550
Pipelining Jumps

\[ t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots \]

\( I_1 \) 096: ADD
\( I_2 \) 100: J 304
\( I_3 \) 104: ADD
\( I_4 \) 304: ADD

**Resource Usage**

- **IF**
  - \( I_1 \)
  - \( I_2 \)
  - \( I_3 \)
  - \( I_4 \)
  - \( I_5 \)

- **ID**
  - \( I_1 \)
  - \( I_2 \)
  - **nop**
  - \( I_4 \)
  - \( I_5 \)

- **EX**
  - \( I_1 \)
  - \( I_2 \)
  - **nop**
  - \( I_4 \)
  - \( I_5 \)

- **MA**
  - \( I_1 \)
  - \( I_2 \)
  - **nop**
  - \( I_4 \)
  - \( I_5 \)

- **WB**
  - \( I_1 \)
  - \( I_2 \)
  - **nop**
  - \( I_4 \)
  - \( I_5 \)
Pipelining Conditional Branches

Branch condition computed in execute stage. What should be done in decode stage?

<table>
<thead>
<tr>
<th>addr</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁</td>
<td>096</td>
</tr>
<tr>
<td>l₂</td>
<td>100</td>
</tr>
<tr>
<td>l₃</td>
<td>104</td>
</tr>
<tr>
<td>l₄</td>
<td>304</td>
</tr>
</tbody>
</table>
Pipelining Conditional Branches

<table>
<thead>
<tr>
<th>addr</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$</td>
<td>096 ADD</td>
</tr>
<tr>
<td>$l_2$</td>
<td>100 BEQZ r1 200</td>
</tr>
<tr>
<td>$l_3$</td>
<td>104 ADD</td>
</tr>
<tr>
<td>$l_4$</td>
<td>304 ADD</td>
</tr>
</tbody>
</table>

If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.
Update Stall Signal

Stall ← <<original stall signal>>

& !( (opcodeE == BEQZ) & zero? # branch condition true
  + (opcodeE == BNEZ) & !zero? # branch condition true
)

Do not stall if branch is taken. Why?
Instruction in the decode stage is invalid.
Kill instruction instead.
Pipelining Conditional Branches

If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.

<table>
<thead>
<tr>
<th>addr</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁</td>
<td>096</td>
</tr>
<tr>
<td>l₂</td>
<td>100</td>
</tr>
<tr>
<td>l₃</td>
<td>104</td>
</tr>
<tr>
<td>l₄</td>
<td>304</td>
</tr>
</tbody>
</table>
Derive PCSrc Signal

Derive mux control signal for PCSrc.

\[
\text{if( (opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z) ), PCSrc} \leftarrow \text{br}
\]

\[
\text{else if ( (opcodeD == J) + (opcodeD == JAL)), PCSrc} \leftarrow \text{jabs}
\]

\[
\text{else if ( (opcodeD == JR) + (opcodeD == JALR)), PCSrc} \leftarrow \text{rind}
\]

\[
\text{otherwise, PCSrc} \leftarrow \text{PC + 4}
\]

Derive mux control signal for IRSrcD.

\[
\text{if( (opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z) ), ICSrcD} \leftarrow \text{nop}
\]

\[
\text{else if ( opcodeD==J ) + (opcodeD==JAL) +}
\]

\[
\text{(opcodeD==JAR) + (opcodeD==JALR) }, ICSrcD \leftarrow \text{nop}
\]

\[
\text{otherwise, ICSrcD} \leftarrow \text{Instr}
\]

Derive mux control signal for IRSrcE.

\[
\text{if( (opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z) ), ICSrcE} \leftarrow \text{nop}
\]

\[
\text{otherwise, IRSrcE} \leftarrow (\text{stall} \& \text{nop}) + (!\text{stall} \& \text{IRD})
\]
Pipelining Conditional Branches

If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.

<table>
<thead>
<tr>
<th>addr</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁</td>
<td>096 ADD</td>
</tr>
<tr>
<td>I₂</td>
<td>100 BEQZ r1 200</td>
</tr>
<tr>
<td>I₃</td>
<td>104 ADD</td>
</tr>
<tr>
<td>I₄</td>
<td>304 ADD</td>
</tr>
</tbody>
</table>
Derive IRSrcD Signal

Derive mux control signal for PCSrc.

\[
\text{if}( \text{opcodeE} == \text{BEQZ} \& z) + (\text{opcodeE} == \text{BNEZ} \& !z), \text{PCSrc} \leftarrow \text{br} \\
\text{else if} ((\text{opcodeD} == \text{J}) + (\text{opcodeD} == \text{JAL})), \text{PCSrc} \leftarrow \text{jabs} \\
\text{else if} ((\text{opcodeD} == \text{JR}) + (\text{opcodeD} == \text{JALR})), \text{PCSrc} \leftarrow \text{rind} \\
\text{otherwise}, \text{PCSrc} \leftarrow \text{PC} + 4
\]

Derive mux control signal for IRSrcD.

\[
\text{if}( \text{opcodeE} == \text{BEQZ} \& z) + (\text{opcodeE} == \text{BNEZ} \& !z), \text{ICSrcD} \leftarrow \text{nop} \\
\text{else if }:(\text{opcodeD} == j) + (\text{opcodeD} == \text{JAL}) + \\
(\text{opcodeD} == \text{JAR}) + (\text{opcodeD} == \text{JALR}) \}, \text{ICSrcD} \leftarrow \text{nop} \\
\text{otherwise}, \text{ICSrcD} \leftarrow \text{Instr}
\]

Derive mux control signal for IRSrcE.

\[
\text{if}( \text{opcodeE} == \text{BEQZ} \& z) + (\text{opcodeE} == \text{BNEZ} \& !z) \}, \text{ICSrcE} \leftarrow \text{nop} \\
\text{otherwise}, \text{ICSrcE} \leftarrow (\text{stall} \& \text{nop}) + (!\text{stall} \& \text{IRD})
\]
Pipelining Conditional Branches

<table>
<thead>
<tr>
<th>addr</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_1</td>
<td>096</td>
</tr>
<tr>
<td>I_2</td>
<td>100</td>
</tr>
<tr>
<td>I_3</td>
<td>104</td>
</tr>
<tr>
<td>I_4</td>
<td>304</td>
</tr>
</tbody>
</table>

If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.
**Derive IRSrcE Signal**

Derive mux control signal for PCSrc.

\[
\text{if( } (\text{opcodeE} == \text{BEQZ} \& \text{z}) + (\text{opcodeE} == \text{BNEZ} \& !\text{z}) \text{ ), } \text{PCSrc} \leftarrow \text{br} \\
\text{else if } ((\text{opcodeD} == \text{J}) + (\text{opcodeD} == \text{JAL})), \text{PCSrc} \leftarrow \text{jabs} \\
\text{else if } ((\text{opcodeD} == \text{JR}) + (\text{opcodeD} == \text{JALR})), \text{PCSrc} \leftarrow \text{rind} \\
\text{otherwise, } \text{PCSrc} \leftarrow \text{PC} + 4
\]

Derive mux control signal for IRSrcD.

\[
\text{if( } (\text{opcodeE} == \text{BEQZ} \& \text{z}) + (\text{opcodeE} == \text{BNEZ} \& !\text{z}) \text{ ), } \text{IRSrcD} \leftarrow \text{nop} \\
\text{else if( } (\text{opcodeD}==\text{J}) + (\text{opcodeD}==\text{JAL}) + \\
(\text{opcodeD}==\text{JAR}) + (\text{opcodeD}==\text{JALR}) \text{ ), } \text{IRSrcD} \leftarrow \text{nop} \\
\text{otherwise, } \text{IRSrcD} \leftarrow \text{Instr}
\]

Derive mux control signal for IRSrcE.

\[
\text{if( } (\text{opcodeE} == \text{BEQZ} \& \text{z}) + (\text{opcodeE} == \text{BNEZ} \& !\text{z}) \text{ ), } \text{IRSrcE} \leftarrow \text{nop} \\
\text{otherwise, } \text{IRSrcE} \leftarrow (\text{stall} \& \text{nop}) + (!\text{stall} \& \text{IRD})
\]
Pipelining Branches

Time:
\[ t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots \]

\[ \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \]
\[ \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \]
\[ \text{IF}_3 \quad \text{ID}_3 \quad \text{EX}_3 \quad \text{MA}_3 \quad \text{WB}_3 \]
\[ \text{IF}_4 \quad \text{ID}_4 \quad \text{EX}_4 \quad \text{MA}_4 \quad \text{WB}_4 \]
\[ \text{IF}_5 \quad \text{ID}_5 \quad \text{EX}_5 \quad \text{MA}_5 \quad \text{WB}_5 \]

Resource Usage:

<table>
<thead>
<tr>
<th>Time</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>l_1</td>
<td>l_1</td>
<td>l_1</td>
<td>l_1</td>
<td>l_1</td>
</tr>
<tr>
<td>t1</td>
<td>l_2</td>
<td>l_2</td>
<td>l_2</td>
<td>l_2</td>
<td>l_2</td>
</tr>
<tr>
<td>t2</td>
<td>l_3</td>
<td>l_3</td>
<td>l_3</td>
<td>l_3</td>
<td>l_3</td>
</tr>
<tr>
<td>t3</td>
<td>l_4</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
</tr>
<tr>
<td>t4</td>
<td>l_5</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
</tr>
<tr>
<td>t5</td>
<td>l_5</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
</tr>
<tr>
<td>t6</td>
<td>l_5</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
</tr>
<tr>
<td>t7</td>
<td>l_5</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
<td>\text{nop}</td>
</tr>
</tbody>
</table>

Program:
\[(l_1) \ 096: \text{ADD} \]
\[(l_2) \ 100: \text{BEQZ 200} \]
\[(l_3) \ 104: \text{ADD} \]
\[(l_4) \ 108: \]
\[(l_5) \ 304: \text{ADD} \]
Solution 1: Resolve Earlier

Large performance impact
- Suppose CPI = 1, 30% branch
- If branch stalls for 2 cycles, new CPI is 1.6

Solution – Branch Computation
- Determine whether branch is taken or not earlier in pipeline (e.g., beq)
- Compute target branch address earlier (e.g., PC addition)

Solution – MIPS
- MIPS branch tests if a register is equal to zero (e.g., beq)
- Move zero test to ID/RF stage
- Introduce adder to calculate new PC in ID/RF stage
- With early branch resolution and kill/stall signals in decode, branches stall for 1 cycle
Add sufficient logic in decode stage to generate “zero?” signal.
Branch is resolved in ID stage instead of EX stage, eliminating one stall cycle.
Solution 2: Predict Condition

Stall until branch direction is clear

Predict Branch Not Taken
- Execute successor instructions in sequence
- “Squash” instructions in pipeline if branch taken
- Advantage: 47% of MIPS branches not taken
- Advantage: PC+4 already calculated for instruction fetch

Predict Branch Taken
- Advantage: 53% of MIPS branches taken
- Disadvantage: Target address not yet calculated, 1-cycle penalty

More sophisticated branch prediction later…
Solution 3: Change ISA Semantics

Delayed Branch

- Change ISA semantics: Instruction after jump/branch always executed.
- Define branch to take place after a following instruction
- Gives compiler flexibility to schedule useful instructions into a branch-induced stall

- Branch delay of length n

  Branch instruction
  Sequential successor 1
  Sequential successor 2
  ...
  Sequential successor n
  Branch target if taken

- MIPS uses n=1 delay slot to calculate branch outcome, target address
(a) Fills delay slot and reduces instruction count, (b) DSUB needs copying and increases instruction count, (c) OR executes if branch fails so issue speculatively
Delayed Branches

Compiler Effectiveness (n=1 branch delay slot)
- Fill about 60% of branch delay slots
- About 80% of instructions executed are useful computation

Disadvantages of Delayed Branches
- As pipelines deepen, branch delay grows and requires more slots
- Less popular than dynamic approaches (e.g., branch prediction)
Pipelining in Practice

Why is IPC < 1?

Full forwarding may be too expensive to implement
- Implement only frequently used forwarding paths
- Implementing infrequently used forwarding paths might impact length of pipeline stage, increase clock period, and reduce IPC forwarding gains

Multi-cycle Instructions (e.g., loads)
- Instruction following a multi-cycle instruction cannot use its results
- MIPS-I defined load-delay slots, a software-visible pipeline hazard.
- Rely on compiler to schedule useful instructions, nops

Conditional Branches
- Without delay slots, kill following instructions
- With delay slots, rely on compiler to schedule useful instructions, nops
Optimal Pipeline Depth

- Performance (BIPS) versus Power (W)

- FO4 is measure of delay: Delay of inverter that is driven by inverter 4x smaller and that is driving inverter 4x larger.
- Quantify amount of logic per pipeline stage in FO4 delays
- (shorter delays → deeper pipelines)
Interrupts alter normal control flow
- Event that needs to be processed by another (system) program
- Event is considered unexpected or rare from program's perspective
Causes of Interrupts

Interrupt
- An event that requests the attention of the processor

Asynchronous Interrupt – External Event
- Input/output device service-request
- Timer expiration
- Power disruptions, hardware failure

Synchronous Interrupt – Internal Event
- Undefined opcode, privileged instruction
- Arithmetic overflow, FPU exception
- Misaligned memory access
- Virtual memory exceptions – page faults, TLB misses, protection violations
- Traps – system calls, jumps into kernel
- Also known as exceptions
Asynchronous Interrupts

Service Request
- An I/O device requests attention by asserting one of the prioritized interrupt request lines

Invoking Interrupt Handler
- Processor decides to process interrupt
- Stops current program at instruction j, completing all instructions up to j-1. Defines a precise interrupt.
- Saves PC of instruction j in a special register (e.g., EPC)
- Disables interrupts and transfers control to designated interrupt handler running in kernel mode.
Interrupt Handler

PC Processing
- Save EPC before re-enabling interrupts, thereby allowing nested interrupts
- Need an instruction to move EPC into general-purpose registers
- Need a way to mask further interrupts until EPC saved

Status Register
- Read status register to determine cause of interrupt
- Executes handler code

Exiting Interrupt Handler
- Use special indirect jump instruction RFE (return-from-exception)
- Enables interrupts
- Restores processor to user mode
- Restores hardware status and control state
Exceptions
- A synchronous interrupt (exception) is caused by a particular instruction

Instruction Re-start
- Generally, instruction cannot be completed without handler
- Instruction needs re-start after exception has been handled
- Processor must undo the effect of partially executed instructions

System Calls
- If the interrupt arises from a system calls (traps), trapping instruction considered complete
- System calls require a special jump instruction and changing into privileged kernel mode
Pipelining and Interrupt Handling

Synchronous: How does the processor handle multiple, simultaneous exceptions in different pipeline stages?

Asynchronous: How does the processor handle external interrupts?
Pipelining and Interrupt Handling

PC

Inst. Mem

D

Decode

E

+

M

Data Mem

W

PC address Exception

Illegal Opcode

Overflow

Data address Exceptions

Asynchronous Interrupts

Select Handler PC

Kill F Stage

Kill D Stage

Kill E Stage

Asynchronous Interrupts

Commit Point

Writeback

EPC Cause

Kill
Pipelining and Interrupt Handling

Propagate exception flags through pipeline until commit point

Internal Interrupts
- An instruction might generate multiple exception flags
- For a given instruction, exceptions in earlier pipe stages over-ride those in later pipe stages, thereby prioritizing exceptions earlier in time

Inject external interrupts at commit point
- External interrupts over-ride internal interrupts

Check exception flags at commit point
- If exception flagged, update cause and EPC register
- Kill instructions in all pipeline stages
- Inject handler PC into fetch stage
Speculating about Exceptions

Predict
- Exceptions are rare. Predicting that no exceptions occurred is accurate.

Check Prediction
- Exceptions detected at end of pipeline (commit point). Invoke special hardware for various exception types

Recovery Mechanism
- Architectural state modified at end of pipeline (commit point).
- Discard partially executed instructions after an exception
- Launch exception handler after flushing pipeline
Pipelining and Exceptions

![Diagram showing pipeline stages for different instructions]

- **(I₁) 096: ADD**
  - IF₁
  - ID₁
  - EX₁
  - MA₁: `nop` overflow!

- **(I₂) 100: XOR**
  - IF₂
  - ID₂
  - EX₂: `nop` `nop`

- **(I₃) 104: SUB**
  - IF₃
  - ID₃
  - EX₃: `nop` `nop` `nop`

- **(I₄) 108: ADD**
  - IF₄
  - ID₄: `nop` `nop` `nop` `nop`
  - EX₄
  - MA₄
  - WB₄

- **(I₅) Exc. Handler code**
  - IF₅
  - ID₅
  - EX₅
  - MA₅
  - WB₅

---

**Resource Usage**

- **EX**
  - t0: l₁
  - t1: l₂
  - t2: l₃
  - t3: l₄
  - t4: l₅

- **MA**
  - t0: l₁
  - t1: l₂
  - t2: l₃
  - t3: `nop` l₅

- **WB**
  - t0: l₁
  - t1: l₂
  - t2: `nop` l₃
  - t3: `nop` l₅
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)