Last time in Lecture 4

- Pipelining increases clock frequency, while growing CPI more slowly, hence giving greater performance

<table>
<thead>
<tr>
<th>Time</th>
<th>Instructions</th>
<th>Cycles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Program</td>
<td>* Instruction</td>
<td>* Cycle</td>
</tr>
</tbody>
</table>

- Pipelining instructions is complicated by HAZARDS:
  - Structural hazards (two instructions want same hardware resource)
  - Data hazards (earlier instruction produces value needed by later instruction)
  - Control hazards (instruction changes control flow, e.g., branches or exceptions)

- Techniques to handle hazards:
  - Interlock (hold newer instruction until older instructions drain out of pipeline)
  - Bypass (transfer value from older instruction to newer instruction as soon as available somewhere in machine)
  - Speculate (guess effect of earlier instruction)

- Speculation needs predictor, prediction check, and recovery mechanism
Instruction to Instruction Dependence

- What do we need to calculate next PC:
  - For Jumps
    » Opcode, offset and PC
  - For Jump Register
    » Opcode and Register value
  - For Conditional Branches
    » Opcode, PC, Register (for condition), and offset
  - For all others
    » Opcode and PC

PC Calculation Bubbles

- For Jumps
  - Opcode, offset and PC
- For Jump Register
  - Opcode and Register value
- For Conditional Branches
  - Opcode, PC, Register (for condition), and offset
- For all others
  - Opcode and PC

IJ1, r1 ← (r0) + 10
IJ2, r3 ← (r2) + 17

Resource Usage

<table>
<thead>
<tr>
<th></th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t0</td>
</tr>
<tr>
<td>IF</td>
<td>I1</td>
</tr>
<tr>
<td>ID</td>
<td>I1</td>
</tr>
<tr>
<td>EX</td>
<td>I1</td>
</tr>
<tr>
<td>MA</td>
<td>I1</td>
</tr>
<tr>
<td>WB</td>
<td>I1</td>
</tr>
<tr>
<td></td>
<td>t1</td>
</tr>
<tr>
<td>IF</td>
<td>nop</td>
</tr>
<tr>
<td>ID</td>
<td>nop</td>
</tr>
<tr>
<td>EX</td>
<td>nop</td>
</tr>
<tr>
<td>MA</td>
<td>nop</td>
</tr>
<tr>
<td>WB</td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td>t2</td>
</tr>
<tr>
<td>IF</td>
<td>I2</td>
</tr>
<tr>
<td>ID</td>
<td>I2</td>
</tr>
<tr>
<td>EX</td>
<td>I2</td>
</tr>
<tr>
<td>MA</td>
<td>I2</td>
</tr>
<tr>
<td>WB</td>
<td>I2</td>
</tr>
<tr>
<td></td>
<td>t3</td>
</tr>
<tr>
<td>IF</td>
<td>nop</td>
</tr>
<tr>
<td>ID</td>
<td>nop</td>
</tr>
<tr>
<td>EX</td>
<td>nop</td>
</tr>
<tr>
<td>MA</td>
<td>nop</td>
</tr>
<tr>
<td>WB</td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td>t4</td>
</tr>
<tr>
<td>IF</td>
<td>I3</td>
</tr>
<tr>
<td>ID</td>
<td>I3</td>
</tr>
<tr>
<td>EX</td>
<td>I3</td>
</tr>
<tr>
<td>MA</td>
<td>I3</td>
</tr>
<tr>
<td>WB</td>
<td>I3</td>
</tr>
<tr>
<td></td>
<td>t5</td>
</tr>
<tr>
<td>IF</td>
<td>nop</td>
</tr>
<tr>
<td>ID</td>
<td>nop</td>
</tr>
<tr>
<td>EX</td>
<td>nop</td>
</tr>
<tr>
<td>MA</td>
<td>nop</td>
</tr>
<tr>
<td>WB</td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td>t6</td>
</tr>
<tr>
<td>IF</td>
<td>I4</td>
</tr>
<tr>
<td>ID</td>
<td>I4</td>
</tr>
<tr>
<td>EX</td>
<td>I4</td>
</tr>
<tr>
<td>MA</td>
<td>I4</td>
</tr>
<tr>
<td>WB</td>
<td>I4</td>
</tr>
<tr>
<td></td>
<td>t7</td>
</tr>
<tr>
<td>IF</td>
<td>nop</td>
</tr>
<tr>
<td>ID</td>
<td>nop</td>
</tr>
<tr>
<td>EX</td>
<td>nop</td>
</tr>
<tr>
<td>MA</td>
<td>nop</td>
</tr>
<tr>
<td>WB</td>
<td>nop</td>
</tr>
</tbody>
</table>

\[ \text{nop} \Rightarrow \text{pipeline bubble} \]
Speculate next address is PC+4

A jump instruction kills (not stalls) the following instruction

Pipelining Jumps

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

Any interaction between stall and jump?

IRSrc$_D$ = Case opcode$_D$
J, JAL  ⇒ nop
...  ⇒ IM
Jump Pipeline Diagrams

(time) 096: ADD  
100: J 304  
104: ADD  
304: ADD

Resource Usage

IF  ID  EX  MA  WB

(I1) 096: ADD  
(I2) 100: J 304  
(I3) 104: ADD  
(I4) 304: ADD

Pipelining Conditional Branches

branch 0x4 / jabs / rind / br

branch condition is not known until the execute stage
what action should be taken in the decode stage?

096: ADD  
100: BEQZ r1 +200  
104: ADD  
304: ADD
**Pipelining Conditional Branches**

If the branch is taken

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁ 096 ADD</td>
<td>- kill the two following instructions</td>
</tr>
<tr>
<td>I₂ 100 BEQZ r1 +200</td>
<td>- the instruction at the decode stage is not valid</td>
</tr>
<tr>
<td>I₃ 104 ADD</td>
<td>⇒ <em>stall signal is not valid</em></td>
</tr>
<tr>
<td>I₄ 304 ADD</td>
<td></td>
</tr>
</tbody>
</table>

2/7/2008 CS152-Spring’08
New Stall Signal

\[
\text{stall} = ( (rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re_1_D \\
+ (rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re_2_D \\
). !(opcode_E = BEQZ).z + (opcode_E = BNEZ).!z)
\]

Don’t stall if the branch is taken. Why?

Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

\[
\text{PCSrc} = \text{Case opcode}_E \\
\quad \text{BEQZ}.z, \text{BNEZ}.!z \quad \Rightarrow \text{br} \\
\quad \ldots \quad \Rightarrow \text{Case opcode}_D \\
\quad \text{J, JAL} \quad \Rightarrow \text{jabs} \\
\quad \text{JR, JALR} \quad \Rightarrow \text{rind} \\
\quad \ldots \quad \Rightarrow \text{pc}+4
\]

\[
\text{IRSrc}_D = \text{Case opcode}_E \\
\quad \text{BEQZ}.z, \text{BNEZ}.!z \quad \Rightarrow \text{nop} \\
\quad \ldots \quad \Rightarrow \text{Case opcode}_D \\
\quad \text{J, JAL, JR, JALR} \quad \Rightarrow \text{nop} \\
\quad \ldots \quad \Rightarrow \text{IM}
\]

\[
\text{IRSrc}_E = \text{Case opcode}_E \\
\quad \text{BEQZ}.z, \text{BNEZ}.!z \quad \Rightarrow \text{nop} \\
\quad \ldots \quad \Rightarrow \text{stall.nop} + \text{!stall.IR}_D
\]

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction
Branch Pipeline Diagrams
(resolved in execute stage)

Resource Usage

Reducing Branch Penalty
(resolve in decode stage)

- One pipeline bubble can be removed if an extra
  comparator is used in the Decode stage
**Branch Delay Slots**

*expose control hazard to software*

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>096</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_2$</td>
<td>100</td>
<td>BEQZ r1 +200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_3$</td>
<td>104</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_4$</td>
<td>304</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Delay slot instruction*

*executed regardless of branch outcome*

- Other techniques include more advanced branch prediction, which can dramatically reduce the branch penalty... *to come later*

---

**Branch Pipeline Diagrams**

*(branch delay slot)*

```
<table>
<thead>
<tr>
<th>time</th>
<th>I0</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I6</th>
<th>I7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
</tr>
</tbody>
</table>

(I1) 096: ADD

(I2) 100: BEQZ +200

(I3) 104: ADD

(I4) 304: ADD
```

*Resource Usage*

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td>$I_4$</td>
<td>$I_1$</td>
</tr>
</tbody>
</table>
Why an Instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined load delay slots, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler. NOPs not counted in useful CPI (alternatively, increase instructions/program)

CS152 Administrivia

- Krste, office hours, Monday 1-3pm, 645 Soda
  - email for alternate time
- Henry office hours, 511 Soda
  - 9:30-10:30AM Mondays
  - 2:00-3:00PM Fridays
- First lab and practice problems this evening
- In-class quiz dates:
  - Q1: Tuesday February 19 (ISAs, microcode, simple pipelining)
  - Q2: Tuesday March 4 (memory hierarchies)
Breaking News from ISSCC 2008
(International Solid-State Circuits Conference)

Sun Rock Processor

- 16 cores in 4 clusters
- Each core runs 2+2 threads
  - Each thread pair has one thread “scouting” ahead of main thread to find upcoming cache misses
- Transactional memory support
  - Atomically mutate up to 32 locations in memory
- 2.3GHz
- 396mm$^2$ in 65nm CMOS
- 250W!!!
**Intel Quad Core Itanium**

- 4 cores
- 2.0 GHz
- 698mm² in 65nm CMOS!!!!
- 170W
- Over 2 billion transistors

---

**Intel Silverthorne, low-power x86**

- 1 core
- 2.0 GHz
- 25mm² in 45nm CMOS
- 0.5-2W
- 47 million transistors
- In-order dual-issue superscalar pipeline
- Two-way multithreading
**Interrupts**: altering the normal flow of control

An external or internal event that needs to be processed by another (system) program. The event is usually unexpected or rare from program’s point of view.

---

**Causes of Interrupts**

**Interrupt**: an event that requests the attention of the processor

- **Asynchronous**: an external event
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure

- **Synchronous**: an internal event (a.k.a. exceptions)
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - virtual memory exceptions: page faults, TLB misses, protection violations
  - traps: system calls, e.g., jumps into kernel
**History of Exception Handling**

- First system with exceptions was Univac-I, 1951
  - Arithmetic overflow would either
    » 1. trigger the execution a two-instruction fix-up routine at address 0, or
    » 2. at the programmer’s option, cause the computer to stop
  - Later Univac 1103, 1955, modified to add external interrupts
    » Used to gather real-time wind tunnel data

- First system with I/O interrupts was DYSEAC, 1954
  - Had two program counters, and I/O signal caused switch between two PCs
  - Also, first system with DMA (direct memory access by I/O device)

---

**DYSEAC, first mobile computer!**

- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

[Courtesy Mark Smotherman]
Asynchronous Interrupts:invoking the interrupt handler

• An I/O device requests attention by asserting one of the *prioritized interrupt request lines*

• When the processor decides to process the interrupt
  – It stops the current program at instruction $l_1$, completing all the instructions up to $l_{n-1}$ (*precise interrupt*)
  – It saves the PC of instruction $l_i$ in a special register (EPC)
  – It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

Interrupt Handler

• Saves EPC before enabling interrupts to allow nested interrupts ⇒
  – need an instruction to move EPC into GPRs
  – need a way to mask further interrupts at least until EPC can be saved

• Needs to read a *status register* that indicates the cause of the interrupt

• Uses a special indirect jump instruction RFE (*return-from-exception*) which
  – enables interrupts
  – restores the processor to the user mode
  – restores hardware status and control state
Synchronous Interrupts

• A synchronous interrupt (exception) is caused by a particular instruction

• In general, the instruction cannot be completed and needs to be restarted after the exception has been handled
  – requires undoing the effect of one or more partially executed instructions

• In case of a trap (system call), the instruction is considered to have been completed
  – a special jump instruction involving a change to privileged kernel mode

Exception Handling 5-Stage Pipeline

• How to handle multiple simultaneous exceptions in different pipeline stages?
• How and where to handle external asynchronous interrupts?
Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)

- Exceptions in earlier pipe stages override later exceptions for a given instruction

- Inject external interrupts at commit point (override others)

- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage
Speculating on Exceptions

- **Predict**
  - Exceptions are rare, so simply predicting no exceptions is very accurate!

- **Check prediction**
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types

- **Recovery mechanism**
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline

- **Bypassing allows use of uncommitted instruction results by following instructions**

---

Exception Pipeline Diagram

\[
\begin{array}{cccccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
(I_1) & 096: \text{ADD} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{nop} & \text{overflow!} \\
(I_2) & 100: \text{XOR} & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{nop} & \text{nop} \\
(I_3) & 104: \text{SUB} & \text{IF}_3 & \text{ID}_3 & \text{EX}_3 & \text{nop} & \text{nop} & \text{nop} \\
(I_4) & 108: \text{ADD} & \text{IF}_4 & \text{ID}_4 & \text{EX}_4 & \text{MA}_4 & \text{WB}_4 & \text{Exc. Handler code} \\
(I_5) & \text{Exc. Handler code} & \text{IF}_5 & \text{ID}_5 & \text{EX}_5 & \text{MA}_5 & \text{WB}_5 \\
\end{array}
\]
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