Last time in Lecture 4

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

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

- Increases because of pipeline bubbles
- Reduces because fewer logic gates on critical paths between flip-flops

• Pipelining of 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 and write back results)
  - Bypass (transfer value from older instruction to newer instruction as soon as available somewhere in machine)
  - Speculate (guess effect of earlier instruction)
Control Hazards

• 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 other instructions
    » Opcode and PC
      • have to know it’s not one of above
PC Calculation Bubbles
(assuming no branch delay slots for now)

\(\text{time}\)
\(t_0\) \(t_1\) \(t_2\) \(t_3\) \(t_4\) \(t_5\) \(t_6\) \(t_7\) \(\ldots\)

\(I_1\) \(r_1 \leftarrow (r_0) + 10\)
\(I_2\) \(r_3 \leftarrow (r_2) + 17\)

\(I_3\)
\(I_4\)

\(\text{Resource Usage}\)
\(\text{IF}\) \(I_1\) nop \(I_2\) nop \(I_3\) nop \(I_4\)
\(\text{ID}\) \(I_1\) nop \(I_2\) nop \(I_3\) nop \(I_4\)
\(\text{EX}\) \(I_1\) nop \(I_2\) nop \(I_3\) nop \(I_4\)
\(\text{MA}\) \(I_1\) \(I_2\) \(I_3\) \(I_4\)
\(\text{WB}\) \(I_1\) \(I_2\) \(I_3\) \(I_4\)

\(\text{nop} \Rightarrow \text{pipeline bubble}\)
Speculate next address is PC+4

A jump instruction kills (not stalls) the following instruction

How?
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

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

February 2, 2010  CS152, Spring 2010
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>IF</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td>...</td>
</tr>
<tr>
<td>EX</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td>I9</td>
<td>...</td>
</tr>
<tr>
<td>MA</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td>I9</td>
<td>I10</td>
<td>...</td>
</tr>
<tr>
<td>WB</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
<td>I9</td>
<td>I10</td>
<td>I11</td>
<td>...</td>
</tr>
</tbody>
</table>

Jump Pipeline Diagrams

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

Resource Usage

nop ⇒ pipeline bubble
Pipelining Conditional Branches

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  BEQZ r1 +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
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

| I_1  | 096  | ADD  |
| I_2  | 100  | BEQZ r1 +200 |
| I_3  | 104  | ADD  |
| I_4  | 304  | ADD  |
New Stall Signal

\[
stall = ( (rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D \\
) . !((opcode_E = BEQZ).z + (opcode_E = BNEZ).!z)
\]

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

Instruction at the decode stage is invalid
## Control Equations for PC and IR Muxes

<table>
<thead>
<tr>
<th>PCSrc = Case opcode&lt;sub&gt;E&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ.z, BNEZ.!z ⇒ br</td>
</tr>
<tr>
<td>... ⇒</td>
</tr>
<tr>
<td>Case opcode&lt;sub&gt;D&lt;/sub&gt;</td>
</tr>
<tr>
<td>J, JAL ⇒ jabs</td>
</tr>
<tr>
<td>JR, JALR ⇒ rind</td>
</tr>
<tr>
<td>... ⇒ pc+4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IRSrc&lt;sub&gt;D&lt;/sub&gt; = Case opcode&lt;sub&gt;E&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ.z, BNEZ.!z ⇒ nop</td>
</tr>
<tr>
<td>... ⇒</td>
</tr>
<tr>
<td>Case opcode&lt;sub&gt;D&lt;/sub&gt;</td>
</tr>
<tr>
<td>J, JAL, JR, JALR ⇒ nop</td>
</tr>
<tr>
<td>... ⇒ IM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IRSrc&lt;sub&gt;E&lt;/sub&gt; = Case opcode&lt;sub&gt;E&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ.z, BNEZ.!z ⇒ nop</td>
</tr>
<tr>
<td>... ⇒ stall.nop + !stall.IR&lt;sub&gt;D&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7  \\
(I_1) & 096: \text{ADD} & IF_1 & ID_1 & EX_1 & MA_1 & WB_1 \\
(I_2) & 100: \text{BEQZ +200} & IF_2 & ID_2 & EX_2 & MA_2 & WB_2 \\
(I_3) & 104: \text{ADD} & IF_3 & ID_3 & \text{nop} & \text{nop} & \text{nop} & \text{nop} \\
(I_4) & 108: & IF_4 & \text{nop} & \text{nop} & \text{nop} & \text{nop} & \text{nop} \\
(I_5) & 304: \text{ADD} & IF_5 & ID_5 & EX_5 & MA_5 & WB_5 \\
\end{array}
\]

**Resource Usage**

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

\[\text{nop } \Rightarrow \quad \text{pipeline bubble}\]
Reducing Branch Penalty 
(resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage.
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.

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

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

\( I_1 \) 096: ADD

\( I_2 \) 100: BEQZ +200

\( I_3 \) 104: ADD

\( I_4 \) 304: ADD

Resource Usage

\[ \begin{array}{cccccccc}
\text{IF} & I_1 & I_2 & I_3 & I_4 \\
\text{ID} & I_1 & I_2 & I_3 & I_4 \\
\text{EX} & I_1 & I_2 & I_3 & I_4 \\
\text{MA} & I_1 & I_2 & I_3 & I_4 \\
\text{WB} & I_1 & I_2 & I_3 & I_4 \\
\end{array} \]
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 (pipeline interlocks added in hardware)
    » MIPS:“Microprocessor without Interlocked Pipeline Stages”

• 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

- PS1/Lab1 due start of class Thursday Feb 11
- Quiz 1, Tuesday Feb 16
**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)

[Courtesy Mark Smotherman]
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 $I_i$, completing all the instructions up to $I_{i-1}$ (*precise interrupt*)
  - It saves the PC of instruction $I_i$ 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 the case of a system call trap, 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?

Asynchronous Interrupts
Exception Handling 5-Stage Pipeline

PC
Inst. Mem
D
Decode
E
M
Data Mem
W

Illegal Opcode
Overflow
Data address Exceptions

PC address Exception

Illegal Opcode

PC address Exception

Illegal Opcode

Illegal Opcode

Select Handler PC

Kill F Stage

Kill D Stage

Kill E Stage

Asynchronous Interrupts

Writeback

EPC

Kill

Commit Point
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

• Prediction mechanism
  – Exceptions are rare, so simply predicting no exceptions is very accurate!

• Check prediction mechanism
  – 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

(time) 
| t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . |

(I_1) 096: ADD  
IF | IF_1 | ID | ID_1 | EX | EX_1 | MA | MA_1 | nop | overflow!  

(I_2) 100: XOR  
IF | IF_2 | ID | ID_2 | EX | EX_2 | nop | nop  

(I_3) 104: SUB  
IF | IF_3 | ID | ID_3 | nop | nop | nop | nop  

(I_4) 108: ADD  
IF | IF_4 | ID | ID_4 | nop | nop | nop | nop | nop | nop |

(I_5) Exc. Handler code  
IF | IF_5 | ID | ID_5 | EX | EX_5 | MA | MA_5 | WB | WB_5

Resource Usage

(time) 
| t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . |

IF | I_1 | I_2 | I_3 | I_4 | I_5 |  

ID | I_1 | I_2 | I_3 | I_4 | I_5 |  

EX | I_1 | I_2 | I_3 | I_4 | I_5 |  

MA | I_1 | I_2 | I_3 | I_4 | I_5 |  

WB | I_1 | I_2 | I_3 | I_4 | I_5 |  

Overflow
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