ECE 4750 Computer Architecture

Topic 4: Pipelining
Control Hazards

Christopher Batten
School of Electrical and Computer Engineering
Cornell University

http://www.csl.cornell.edu/courses/ece4750
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
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

How?
Pipelining Jumps

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

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
Pipelining Conditional Branches

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

| I₁ | 096 | ADD |
| I₂ | 100 | BEQZ r1 +200 |
| I₃ | 104 | ADD |
| I₄ | 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

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

I₁  096  ADD
I₂  100  BEQZ r1 +200
I₃  104  ADD
I₄  304  ADD
New Stall Signal

\[
\text{stall} = ( \ (rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D \\
+ ( (rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D \\
) . !((\text{opcode}_E = \text{BEQZ}).z + (\text{opcode}_E = \text{BNEZ}).!z)
\]

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

Instruction at the decode stage is invalid

Might stall F+D, but we don’t want to stall, we want to squash an update the PC – if branch waits for stall to finish we have deadlock
Control Equations for PC and IR Muxes

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

\[
\text{IRSrc}_D = \text{Case opcode}_E
\]
\[
\begin{align*}
&\text{BEQZ.z, BNEZ.!z} \implies \text{nop} \\
&\ldots \implies \text{Case opcode}_D \\
&\quad \text{J, JAL, JR, JALR} \implies \text{nop} \\
&\ldots \implies \text{IM}
\end{align*}
\]

\[
\text{IRSrc}_E = \text{Case opcode}_E
\]
\[
\begin{align*}
&\text{BEQZ.z, BNEZ.!z} \implies \text{nop} \\
&\ldots \implies \text{stall.nop} + \text{!stall.IR}_D
\end{align*}
\]

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction.
Reducing Branch Penalty
(resolve in decode stage)

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

Pipeline diagram now same as for jumps
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.

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

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
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)*
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
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?
Exception Handling 5-Stage Pipeline

PC address Exception → PC
Illegal Opcode → Decode
Overflow → M
Data address Exceptions → EPC
Asynchronous Interrupts → EPC
Select Handler PC → Kill F Stage
Kill D Stage
Kill E Stage
Asynchronous Interrupts → Writeback
EPC
Kill Writeback
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
- Pipeline diagram for various exception scenarios on board
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
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 & CS152