CS 152 Computer Architecture and Engineering

Lecture 5 - Pipelining II
(Branches, Exceptions)

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152
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)
Branch Pipeline Diagrams
(branches resolved in decode stage)

\[
time
\]
\[
t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots
\]

(I₁) 096: ADD
IF₁ ID₁ EX₁ MA₁ WB₁

(I₂) 100: BEQZ +200
IF₂ ID₂ EX₂ MA₂ WB₂

(I₃) 104: ADD
IF₃ - - - -

(I₄) 304: ADD
IF₄ ID₄ EX₄ MA₄ WB₄

Resource Usage

\[
\text{IF} \quad I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5
\]
\[
\text{ID} \quad I_1 \quad I_2 \quad - \quad I_4 \quad I_5
\]
\[
\text{EX} \quad I_1 \quad I_2 \quad - \quad I_4 \quad I_5
\]
\[
\text{MA} \quad I_1 \quad I_2 \quad - \quad I_4 \quad I_5
\]
\[
\text{WB} \quad I_1 \quad I_2 \quad - \quad I_4 \quad I_5
\]

- ⇒ pipeline bubble
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.

| I₁  | 096   | ADD   |
| I₂  | 100   | BEQZ r1, +200 |
| I₃  | 104   | ADD   |
| I₄  | 304   | ADD   |

Delay slot instruction executed regardless of branch outcome
Branch Pipeline Diagrams
(branch delay slot)

\[
\begin{align*}
time \\
t0 & \quad t1 & \quad t2 & \quad t3 & \quad t4 & \quad t5 & \quad t6 & \quad t7 \quad . . . \\
(I_1) & 096: ADD & IF_1 & ID_1 & EX_1 & MA_1 & WB_1 \\
(I_2) & 100: BEQZ +200 & IF_2 & ID_2 & EX_2 & MA_2 & WB_2 \\
(I_3) & 104: ADD & IF_3 & ID_3 & EX_3 & MA_3 & WB_3 \\
(I_4) & 304: ADD & IF_4 & ID_4 & EX_4 & MA_4 & WB_4
\end{align*}
\]
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 increase instructions/program!*
## RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>B&lt;cond.&gt;</td>
<td>After Execute</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>
Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PC Generation/Mux</td>
</tr>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

- Branch Target Address Known
- Branch Direction & Jump Register Target Known

Remainder of execute pipeline (+ another 6 stages)
Reducing Control Flow Penalty

Software solutions
- *Eliminate branches - loop unrolling*
  Increases the run length
- *Reduce resolution time - instruction scheduling*
  Compute the branch condition as early as possible (of limited value)

Hardware solutions
- Find something else to do - *delay slots*
  Replaces pipeline bubbles with useful work (requires software cooperation)
- *Speculate - branch prediction*
  *Speculative execution* of instructions beyond the branch
Branch Prediction

Motivation:
Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:
Prediction structures:
• Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:
• Keep result computation separate from commit
• Kill instructions following branch in pipeline
• Restore state to state following branch
Static Branch Prediction

Overall probability a branch is taken is \(~60-70\%\) but:

- **backward** 90%
- **forward** 50%

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
  - `bne0 (preferred taken)`  `beq0 (not taken)`

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
  - typically reported as \(~80\%\) accurate
Dynamic Branch Prediction
learning based on past behavior

Temporal correlation
The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation
Several branches may resolve in a highly correlated manner (a preferred path of execution)
Branch Prediction Bits

- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!

**BP state:**

\[(predict \ take/\neg\ take) \times (last \ prediction \ right/wrong)\]
Branch History Table

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
Exploiting Spatial Correlation
Yeh and Patt, 1992

if (x[i] < 7) then
  y += 1;
if (x[i] < 5) then
  c -= 4;

If first condition false, second condition also false

*History register, H,* records the direction of the last N branches executed by the processor
Two-Level Branch Predictor

*Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)*

Fetch PC

2-bit global branch history shift register

Shift in Taken/¬Taken results of each branch

Taken/¬Taken?
Speculating Both Directions

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- resource requirement is proportional to the number of concurrent speculative executions

- only half the resources engage in useful work when both directions of a branch are executed speculatively

- branch prediction takes less resources than speculative execution of both paths

*With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction!*
Limitations of BHTs

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

Correctly predicted taken branch penalty

Jump Register penalty

UltraSPARC-III fetch pipeline
CS152 Administrivia
Branch Target Buffer

BP bits are stored with the predicted target address.

IF stage: If \((BP=\text{taken})\) then \(nPC=\text{target}\) else \(nPC=PC+4\)

later: check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target = 1032

⇒ kill PC=236 and fetch PC=1032

Is this a common occurrence? Can we avoid these bubbles?
BTB is only for Control Instructions

BTB contains useful information for branch and jump instructions only
⇒ Do not update it for other instructions

For all other instructions the next PC is PC+4!

*How to achieve this effect without decoding the instruction?*
Branch Target Buffer (BTB)

- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate

\[ BHT \text{ in later pipeline stage corrects when BTB misses a predicted taken branch} \]

\[ BTB/BHT \text{ only updated after branch resolves in E stage} \]
Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)
  
  BTB works well if usually return to the same place
  
  ⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?
**Subroutine Return Stack**

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```plaintext
fa() { fb(); }
fb() { fc(); }
fcc() { fd(); }
```

*Push call address when function call executed*

*Pop return address when subroutine return decoded*

- k entries (typically k=8-16)
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. traps or exceptions)
  – undefined opcode, privileged instruction
  – arithmetic overflow, FPU exception
  – misaligned memory access
  – virtual memory exceptions: page faults, TLB misses, protection violations
  – 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?
Exception Handling 5-Stage Pipeline

PC address Exception
Illegal Opcode
Overflow
Data address Exceptions
Asynchronous Interrupts
Commit Point
EPC
Kill Writeback
Select Handler PC
Kill F Stage
Kill D Stage
Kill E Stage

February 2, 2012
CS152, Spring 2012
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

\[
\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
\hline
(I_1) 096: ADD & IF_1 & ID_1 & EX_1 & MA_1 & \rightarrow & \text{overflow!} & \text{overflow!} & \\
(I_2) 100: XOR & IF_2 & ID_2 & EX_2 & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \\
(I_3) 104: SUB & IF_3 & ID_3 & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \\
(I_4) 108: ADD & IF_4 & IF_5 & ID_5 & EX_5 & MA_5 & WB_5 & \rightarrow & \\
(I_5) Exc. Handler code & & & & & & & & \\
\end{array}
\]

Resource Usage

\[
\begin{array}{cccccccc}
\text{Resource} & \text{Usage} & \text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
\hline
IF & I_1 & I_2 & I_3 & I_4 & I_5 & \rightarrow & \rightarrow & \rightarrow & \\
ID & I_1 & I_2 & I_3 & \rightarrow & I_5 & \rightarrow & \rightarrow & \rightarrow & \\
EX & I_1 & I_2 & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \\
MA & I_1 & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \\
WB & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \\
\end{array}
\]
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