Lecture 09: RISC-V Pipeline Implementation

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Acknowledgement

- Slides adapted from Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley
Introduction

- **CPU performance factors**
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
  
  \[
  CPU \ Time = \frac{\text{Instructions} \times \text{Cycles} \times \text{Time}}{\text{Program} \ \text{Instruction} \ \text{Cycle}}
  \]

- **Three groups of instructions**
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `jal`, `jalr`, `b*`

- **CPI**
  - Single-cycle, CPI = 1
  - 5 stage unpipelined, CPI = 5
  - 5 stage pipelined, CPI = 1
An Ideal Pipeline

- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

*These conditions generally hold for industrial assembly lines, but instructions depend on each other!*
Review: Unpipelined Datapath for RISC-V
## Review: Hardwired Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ImmSel</th>
<th>Op2Sel</th>
<th>FuncSel</th>
<th>MemWr</th>
<th>RFWen</th>
<th>WBSel</th>
<th>WASel</th>
<th>PCSel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>*</td>
<td>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUi</td>
<td>IType	extsubscript{12}</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>LW</td>
<td>IType	extsubscript{12}</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>SW</td>
<td>BsType	extsubscript{12}</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQ	extsubscript{true}</td>
<td>BrType	extsubscript{12}</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQ	extsubscript{false}</td>
<td>BrType	extsubscript{12}</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>yes</td>
<td>PC</td>
<td>rd</td>
<td>jabs</td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>rd</td>
<td>rind</td>
</tr>
</tbody>
</table>

Op2Sel= Reg / Imm    
WBSel = ALU / Mem / PC
PCSel = pc+4 / br / rind / jabs
Pipelined Datapath

Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} (= t_{DM} \text{ probably}) \]

However, CPI will increase unless instructions are pipelined
Technology Assumptions

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

Thus, the following timing assumption is reasonable

\[ t_{IM} \sim t_{RF} \sim t_{ALU} \sim t_{DM} \sim t_{RW} \]

A 5-stage pipeline will be focus of our detailed design
- some commercial designs have over 30 pipeline stages to do an integer add!
5-Stage Pipelined Execution

I-Fetch (IF)  |  Decode, Reg. Fetch (ID)  |  Execute (EX)  |  Memory (MA)  |  Write-Back (WB)

Instruction1  |  Instruction2  |  Instruction3  |  Instruction4  |  Instruction5

t0  |  t1  |  t2  |  t3  |  t4  |  t5  |  t6  |  t7  |  ...

Time:
IF1  |  ID1  |  EX1  |  MA1  |  WB1  |  IF2  |  ID2  |  EX2  |  MA2  |  WB2  |  IF3  |  ID3  |  EX3  |  MA3  |  WB3  |  IF4  |  ID4  |  EX4  |  MA4  |  WB4  |  IF5  |  ID5  |  EX5  |  MA5  |  WB5
5-Stage Pipelined Execution

Resource Usage Diagram

0x4

Addr

PC

Addr rdata

Inst. Memory

I-Fetch (IF)

Decode, Reg. Fetch (ID)

Execute (EX)

Memory (MA)

Write-Back (WB)

<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>l1</td>
<td>l2</td>
<td>l3</td>
<td>l4</td>
<td>l5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l4</td>
<td>l5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l4</td>
<td>l5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l4</td>
<td>l5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l4</td>
<td>l5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Not quite correct!
We need an Instruction Reg (IR) for each stage
Pipelined RISC-V Datapath

without jumps

Control Points Need to Be Connected
Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → structural hazard

- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value → data hazard
  - Dependence may be for the next instruction’s address → control hazard (branches, exceptions)
Resolving Structural Hazards

• Structural hazard occurs when two instructions need same hardware resource at same time
  – Can resolve in hardware by stalling newer instruction till older instruction finished with resource

• A structural hazard can always be avoided by adding more hardware to design
  – E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory

• Our 5-stage pipeline has no structural hazards by design
  – Thanks to RISC-V ISA, which was designed for pipelining
Data Hazards

\[ x_4 \leftarrow x_1 \ldots \]

\[ x_1 \leftarrow \ldots \]

... 
\[ x_1 \leftarrow x_0 + 10 \]
\[ x_4 \leftarrow x_1 + 17 \]
...

\[ x_1 \text{ is stale. Oops!} \]
How Would You Resolve This?

• Three options
  – Wait (stall)
  – Bypass: ask them for what you need before his/her final deliverable
  – Speculate on values to read
Strategy 1:

*Wait for the result to be available by freezing earlier pipeline stages* ➔ *interlocks*
Interlocks to resolve Data Hazards

Stall Condition

... x1 ← x0 + 10
x4 ← x1 + 17
...
Stalled Stages and Pipeline Bubbles

Resource Usage

<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 I1</td>
<td>I2</td>
<td>I3</td>
<td>I3</td>
<td>I3</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID I1</td>
<td>I2</td>
<td>I2</td>
<td>I2</td>
<td>I2</td>
<td>I2</td>
<td>I2</td>
<td>I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX I1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>I2</td>
<td>I3</td>
</tr>
<tr>
<td>MA I1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>I2</td>
<td>I3</td>
</tr>
<tr>
<td>WB I1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>I2</td>
<td>I3</td>
</tr>
</tbody>
</table>

(\(I_1\) \times 1 \leftarrow (x_0) + 10 \text{IF}_1)

(\(I_2\) \times 4 \leftarrow (x_1) + 17

\(I_3\)

\(I_4\)

\(I_5\)

- \Rightarrow \text{pipeline bubble}
Compare the source registers of the instruction in the decode stage with the destination register of the uncommitted instructions.
Should we always stall if an rs field matches some rd?
not every instruction writes a register => we
not every instruction reads a register => re
In RISC-V Sodor Implementation

```scala
{ // stall for all hazards
    stall := ((exe_reg_wbaddr == dec_rs1_addr) && (dec_rs1_addr != UInt(0)) && exe_reg_ctrl_rf_wen && dec_rs1_oen) ||
              ((mem_reg_wbaddr == dec_rs1_addr) && (dec_rs1_addr != UInt(0)) && mem_reg_ctrl_rf_wen && dec_rs1_oen) ||
              ((wb_reg_wbaddr == dec_rs1_addr) && (dec_rs1_addr != UInt(0)) && wb_reg_ctrl_rf_wen && dec_rs1_oen) ||
              ((exe_reg_wbaddr == dec_rs2_addr) && (dec_rs2_addr != UInt(0)) && exe_reg_ctrl_rf_wen && dec_rs2_oen) ||
              ((mem_reg_wbaddr == dec_rs2_addr) && (dec_rs2_addr != UInt(0)) && mem_reg_ctrl_rf_wen && dec_rs2_oen) ||
              ((wb_reg_wbaddr == dec_rs2_addr) && (dec_rs2_addr != UInt(0)) && wb_reg_ctrl_rf_wen && dec_rs2_oen) ||
              ((exe_inst_is_load) && (exe_reg_wbaddr == dec_rs1_addr) && (exe_reg_wbaddr != UInt(0)) && dec_rs1_oen) ||
              ((exe_inst_is_load) && (exe_reg_wbaddr == dec_rs2_addr) && (exe_reg_wbaddr != UInt(0)) && dec_rs2_oen) ||
              ((exe_reg_is_csr))
}
```
# Source & Destination Registers

<table>
<thead>
<tr>
<th>Func Code</th>
<th>Rs2</th>
<th>Rs1</th>
<th>Func Code</th>
<th>Rd</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td></td>
<td></td>
<td>ALUI/LW/JALR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUI</td>
<td></td>
<td></td>
<td>SW/Bcond</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operands:**

- **source(s):** ALU, ALUI, LW, SW, Bcond, JAL, JALR
- **destination:** rd

**Example Instructions:**

- **ALU:**
  - `rd <= rs1 func10 rs2`
  - Destination: `rs1, rs2`
  - Opcode: `rd`

- **ALUI:**
  - `rd <= rs1 op imm`
  - Destination: `rs1`
  - Opcode: `rd`

- **LW:**
  - `rd <= M [rs1 + imm]`
  - Destination: `rs1`
  - Opcode: `rd`

- **SW:**
  - `M [rs1 + imm] <= rs2`
  - Destination: `rs1, rs2`
  - Opcode: `-`

- **Bcond:**
  - `rs1, rs2`
  - True: `PC <= PC + imm`
  - False: `PC <= PC + 4`
  - Destination: `rs1, rs2`
  - Opcode: `-`

- **JAL:**
  - `x1 <= PC, PC <= PC + imm`
  - Destination: `-`
  - Opcode: `rd`

- **JALR:**
  - `rd <= PC, PC <= rs1 + imm`
  - Destination: `rs1`
  - Opcode: `rd`
Deriving the Stall Signal

$C_{\text{dest}}$
\[ ws = \text{rd} \]
\[ \text{we} = \text{Case opcode} \]
\[ \text{ALU, ALUi, LW, JALR} \Rightarrow \text{on} \]
\[ \ldots \Rightarrow \text{off} \]

$C_{\text{re}}$
\[ \text{re1} = \text{Case opcode} \]
\[ \text{ALU, ALUi, LW, SW, Bcond, JALR} \Rightarrow \text{on} \]
\[ \text{JAL} \Rightarrow \text{off} \]
\[ \text{re2} = \text{Case opcode} \]
\[ \text{ALU, SW, Bcond} \Rightarrow \text{on} \]
\[ \ldots \Rightarrow \text{off} \]

$C_{\text{stall}}$
\[ \text{stall} = ((rs_{1D} == ws_E) \&\& \text{we}_E + (rs_{1D} == ws_M) \&\& \text{we}_M + (rs_{1D} == ws_W) \&\& \text{we}_W) \&\& \text{re1}_D + ((rs_{2D} == ws_E) \&\& \text{we}_E + (rs_{2D} == ws_M) \&\& \text{we}_M + (rs_{2D} == ws_W) \&\& \text{we}_W) \&\& \text{re2}_D \]
Hazards due to Loads & Stores

Stall Condition

What if $x_{1+7} = x_{3+5}$?

Is there any possible data hazard in this instruction sequence?

... $M[x_{1+7}] \leq x_2$

... $x_4 \leq M[x_{3+5}]$
Load & Store Hazards

... M[x1+7] <= x2
x4 <= M[x3+5]
...

However, the hazard is avoided because our memory system completes writes in one cycle!

Load/Store hazards are sometimes resolved in the pipeline and sometimes in the memory system itself.

More on this later in the course.
Resolving Data Hazards (2)

Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage → bypass
Each *stall or kill* introduces a bubble in the pipeline

\[ \Rightarrow CPI > 1 \]

A new datapath, i.e., a *bypass*, can get the data from
the output of the ALU to its input
Hardware Support for Forwarding
Detecting RAW Hazards

• Pass register numbers along pipeline
  – ID/EX.RegisterRs = register number for Rs in ID/EX
  – ID/EX.RegisterRt = register number for Rt in ID/EX
  – ID/EX.RegisterRd = register number for Rd in ID/EX
• Current instruction being executed in ID/EX register
• Previous instruction is in the EX/MEM register
• Second previous is in the MEM/WB register
• RAW Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
Detecting the Need to Forward

• But only if forwarding instruction will write to a register!
  – EX/MEM.RegWrite, MEM/WB.RegWrite

• And only if Rd for that instruction is not R0
  – EX/MEM.RegisterRd ≠ 0
  – MEM/WB.RegisterRd ≠ 0
Forwarding Conditions

• Detecting RAW hazard with Previous Instruction
  – if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01 (Forward from EX/MEM pipe stage)
  – if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01 (Forward from EX/MEM pipe stage)

• Detecting RAW hazard with Second Previous
  – if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10 (Forward from MEM/WB pipe stage)
  – if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10 (Forward from MEM/WB pipe stage)
Adding a Bypass

When does this bypass help?

\( l_1 \) \( x1 \leq x0 + 10 \)
\( l_2 \) \( x4 \leq x1 + 17 \)

yes

\( x1 \leq M[x0 + 10] \)
\( x4 \leq x1 + 17 \)

no

JAL 500
\( x4 \leq x1 + 17 \)

no
The Bypass Signal
Deriving it from the Stall Signal

\[
\text{stall} = \left( (\text{rs}_D = \text{ws}_E) \&\& \text{we}_E + (\text{rs}_D = \text{ws}_M) \&\& \text{we}_M + (\text{rs}_D = \text{ws}_W) \&\& \text{we}_W \right) \&\& \text{re}_1_D \\
+ \left( (\text{rs}_2_D = \text{ws}_E) \&\& \text{we}_E + (\text{rs}_2_D = \text{ws}_M) \&\& \text{we}_M + (\text{rs}_2_D = \text{ws}_W) \&\& \text{we}_W \right) \&\& \text{re}_2_D
\]

\( \text{ws} = \text{rd} \)

\( \text{we} = \text{Case opcode} \)
ALU, ALUi, LW,, JAL JALR => on
... => off

\( \text{ASrc} = (\text{rs}_1_D = \text{ws}_E) \&\& \text{we}_E \&\& \text{re}_1_D \)

Is this correct?

No because only ALU and ALUi instructions can benefit from this bypass

Split \( \text{we}_E \) into two components: we-bypass, we-stall
Bypass and Stall Signals

Split $w_{E}$ into two components: we-bypass, we-stall

we-bypass$ _{E} = Case \ \text{opcode}_{E}$
ALU, ALUi $\Rightarrow$ on
... $\Rightarrow$ off

we-stall$ _{E} = Case \ \text{opcode}_{E}$
LW, JAL, JALR $\Rightarrow$ on
JAL $\Rightarrow$ on
... $\Rightarrow$ off

$ASrc = (rs_{1D} == ws_{E}) \& \& \ \text{we-bypass}_{E} \& \& re_{1D}$

stall = $((rs_{1D} == ws_{E}) \& \& \ \text{we-stall}_{E} +$
(rsn_{1D} == ws_{M}) \& \& we_{M} + (rs_{1D} == ws_{W}) \& \& we_{W}) \& \& re_{1D}$
$+((rs_{2D} == ws_{E}) \& \& we_{E} + (rs_{2D} == ws_{M}) \& \& we_{M} + (rs_{2D} == ws_{W}) \& \& we_{W}) \& \& re_{2D}$
Is there still a need for the stall signal?

stall = (rs1_D == ws_E) && (opcode_E == LW_E) && (ws_E != 0) && re1_D + (rs2_D == ws_E) && (opcode_E == LW_E) && (ws_E != 0) && re2_D
Control Hazards

What do we need to calculate next PC?

• For Jumps
  – Opcode, PC and offset

• For Jump Register
  – Opcode, Register value, and PC

• For Conditional Branches
  – Opcode, Register (for condition), PC and offset

• For all other instructions
  – Opcode and PC (and have to know it’s not one of above)
### PC Calculation Bubbles

<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>-</td>
<td>I₂</td>
<td>-</td>
<td>I₃</td>
<td>-</td>
<td>I₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>I₁</td>
<td>-</td>
<td>I₂</td>
<td>-</td>
<td>I₃</td>
<td>-</td>
<td>I₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>I₁</td>
<td>-</td>
<td>I₂</td>
<td>-</td>
<td>I₃</td>
<td>-</td>
<td>I₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA</td>
<td>I₁</td>
<td>-</td>
<td>I₂</td>
<td>-</td>
<td>I₃</td>
<td>-</td>
<td>I₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>I₁</td>
<td>-</td>
<td>I₂</td>
<td>-</td>
<td>I₃</td>
<td>-</td>
<td>I₄</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Resource Usage

IF: I₁  -  I₂  -  I₃  -  I₄
ID: I₁  -  I₂  -  I₃  -  I₄
EX: I₁  -  I₂  -  I₃  -  I₄
MA: I₁  -  I₂  -  I₃  -  I₄
WB: I₁  -  I₂  -  I₃  -  I₄

-  ⇒  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?

$\text{IR}_{D} = \text{Case opcode}_D$
JAL $\Rightarrow$ bubble
...
$\Rightarrow$ IM

I_1 096 ADD
I_2 100 J 304
I_3 104 ADD  $\textit{kill}$
I_4 304 ADD
Jump Pipeline Diagrams

\[
\begin{array}{cccccccc}
\text{time} & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 \\
(I_1) & 096: \text{ADD} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 & - & - \\
(I_2) & 100: J 304 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 & - & - \\
(I_3) & 104: \text{ADD} & \text{IF}_3 & - & - & - & - & - & - \\
(I_4) & 304: \text{ADD} & \text{IF}_4 & \text{ID}_4 & \text{EX}_4 & \text{MA}_4 & \text{WB}_4 & - & -
\end{array}
\]

Resource Usage

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

- $\Rightarrow$ pipeline bubble
Pipelining Conditional Branches

Branch condition is not known until the execute stage

*what action should be taken in the decode stage?*
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

\[ l_1 \quad 096 \text{ ADD} \]
\[ l_2 \quad 100 \text{ BEQ x1, x2 +200} \]
\[ l_3 \quad 104 \text{ ADD} \]
\[ l_4 \quad 304 \text{ ADD} \]
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
**Branch Pipeline Diagrams**
(resolved in execute stage)

\[time\]

<table>
<thead>
<tr>
<th></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>(I_1)</td>
<td>096: ADD</td>
<td>IF_1</td>
<td>ID_1</td>
<td>EX_1</td>
<td>MA_1</td>
<td>WB_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_2)</td>
<td>100: BEQ +200</td>
<td>IF_2</td>
<td>ID_2</td>
<td>EX_2</td>
<td>MA_2</td>
<td>WB_2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_3)</td>
<td>104: ADD</td>
<td>IF_3</td>
<td>ID_3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>IF_5</td>
<td>ID_5</td>
<td>EX_5</td>
</tr>
<tr>
<td>(I_4)</td>
<td>108:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_5)</td>
<td>304: ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resource Usage**

IF  | I_1  | I_2  | I_3  | I_4  | I_5   |    |    |    |    |
ID  | I_1  | I_2  | I_3  | -    | I_5   |    |    |    |    |
EX  | I_1  | I_2  | -    | -    | I_5   |    |    |    |    |
MA  | I_1  | I_2  | -    | -    | -     | I_5 |    |    |    |
WB  | I_1  | I_2  | -    | -    | -     |    | I_5 |    |    |

- \[\Rightarrow\] pipeline bubble
What If...

- We used a simple branch that compares only one register (rs1) against zero
- Can we do any better?
Use simpler branches (e.g., only compare one reg against zero) with compare in decode stage

<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>(I_1) 096: ADD</td>
<td>IF_1</td>
<td>ID_1</td>
<td>EX_1</td>
<td>MA_1</td>
<td>WB_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_2) 100: BEQZ +200</td>
<td>IF_2</td>
<td>ID_2</td>
<td>EX_2</td>
<td>MA_2</td>
<td>WB_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_3) 104: ADD</td>
<td>IF_3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I_4) 300: ADD</td>
<td>IF_4</td>
<td>ID_4</td>
<td>EX_4</td>
<td>MA_4</td>
<td>WB_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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_1</td>
<td>I_1</td>
<td>I_1</td>
<td>I_1</td>
</tr>
<tr>
<td>I_2</td>
<td>I_2</td>
<td>I_2</td>
<td>I_2</td>
<td>I_2</td>
</tr>
<tr>
<td>I_3</td>
<td>I_3</td>
<td>I_3</td>
<td>I_3</td>
<td>I_3</td>
</tr>
<tr>
<td>I_4</td>
<td>I_4</td>
<td>I_4</td>
<td>I_4</td>
<td>I_4</td>
</tr>
<tr>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
<td>I_5</td>
</tr>
</tbody>
</table>

- \( \Rightarrow \) 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.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>096</td>
<td>ADD</td>
</tr>
<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>300</td>
<td>ADD</td>
</tr>
</tbody>
</table>

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

\[\text{time}\]

\[
\begin{array}{cccccccc}
\text{t0} & \text{t1} & \text{t2} & \text{t3} & \text{t4} & \text{t5} & \text{t6} & \text{t7} \\
(I_1) 096: \text{ADD} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 \\
(I_2) 100: \text{BEQZ} +200 & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 \\
(I_3) 104: \text{ADD} & \text{IF}_3 & \text{ID}_3 & \text{EX}_3 & \text{MA}_3 & \text{WB}_3 \\
(I_4) 300: \text{ADD} & \text{IF}_4 & \text{ID}_4 & \text{EX}_4 & \text{MA}_4 & \text{WB}_4 \\
\end{array}
\]

Resource Usage

\[\text{time}\]

\[
\begin{array}{cccccccc}
\text{t0} & \text{t1} & \text{t2} & \text{t3} & \text{t4} & \text{t5} & \text{t6} & \text{t7} \\
\text{IF} & I_1 & I_2 & I_3 & I_4 & I_4 \\
\text{ID} & I_1 & I_2 & I_3 & I_4 & I_4 \\
\text{EX} & I_1 & I_2 & I_3 & I_4 \\
\text{MA} & I_1 & I_2 & I_3 & I_4 \\
\text{WB} & I_1 & I_2 & I_3 & I_4 \\
\end{array}
\]
Post-1990 RISC ISAs don’t have delay slots

• Encodes microarchitectural detail into ISA
  – C.f. IBM 650 drum layout

• What are the problems with delay slots?

• Performance issues
  – E.g., I-cache miss or page fault on delay slot instruction causes machine to wait, even if delay slot is a NOP

• Complicates more advanced microarchitectures
  – 30-stage pipeline with four-instruction-per-cycle issue

• Complicates the compiler’s job

• Better branch prediction reduced need for delay slots
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
RISC-V Branches and Jumps

• JAL: unconditional jump to PC+immediate

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset[20:1]</td>
<td>dest</td>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• JALR: indirect jump to rs1+immediate

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>dest</td>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Branch: if (rs1 conds rs2), branch to PC+immediate

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset[12,10:5]</td>
<td>src2</td>
<td>src1</td>
<td>BEQ/BNE</td>
<td>offset[11,4:1]</td>
<td>BRANCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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?

• **JAL**: unconditional jump to PC+immediate
• **JALR**: indirect jump to rs1+immediate
• **Branch**: if (rs1 conds rs2), branch to PC+immediate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</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)

- Branch Address Calc/Begin Decode
- Complete Decode
- Steer Instructions to Functional units
- Register File Read
- Integer Execute
- 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 because branches often in critical path through code)

• **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 that following branch
Static Branch Prediction

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

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

- bne0 \((preferred \ taken)\)
- beq0 \((not \ taken)\)
Dynamic Branch Prediction
learning based on past behavior

• Temporal correlation (time)
  – If I tell you that a certain branch was taken last time, does this help?
  – The way a branch resolves may be a good predictor of the way it will resolve at the next execution

• Spatial correlation (space)
  – Several branches may resolve in a highly correlated manner
  – For instance, a preferred path of execution
Dynamic Branch Prediction

• 1-bit prediction scheme
  – Low-portion address as address for a one-bit flag for Taken or NotTaken historically
  – Simple

• 2-bit prediction
  – Miss twice to change
Branch Prediction Bits

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

**BP state:**

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

Fetch PC

I-Cache

Instruction

Opcode

offset

Branch?

Target PC

+  

BHT Index

$2^k$-entry BHT, 2 bits/entry

Taken/¬Taken?

4K-entry BHT, 2 bits/entry, ≈80-90% correct predictions
Exploiting Spatial Correlation

Yeh and Patt, 1992

If first condition false, second condition also false

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

*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!
  – What would you choose with 80% accuracy?
Are We Missing Something?

• Knowing whether a branch is taken or not is great, but what else do we need to know about it?

Branch target address
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
BP bits are stored with the predicted target address.

IF stage: If (BP=taken) then nPC=target else nPC=PC+4
Later: check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb
Address Collisions (Mis-Prediction)

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?*
BTB is only for Control Instructions

• Is even branch prediction fast enough to avoid bubbles?
• When do we index the BTB?
  – i.e., what state is the branch in, in order to avoid bubbles?

• **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?*
• 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
Are We Missing Something? (2)

- When do we update the BTB or BHT?

---

PC $\rightarrow$ addr $\rightarrow$ Inst $\rightarrow$ Memory

Add $\rightarrow$ $0x4$ $\rightarrow$ addr $\rightarrow$ inst $\rightarrow$ IR

Inst $\rightarrow$ Memory

IR $\rightarrow$ $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$

IR $\rightarrow IR$ $\rightarrow IR$ $\rightarrow IR$
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 in later pipeline stage corrects when BTB misses a predicted taken branch

BTB/BHT 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.

```c
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)