Advanced Pipelining
CS740

Sept. 29, 1998

Topics

• Data Hazards
  – Stalling and Forwarding
  – Systematic testing of hazard-handling logic

• Control Hazards
  – Stalling, Predict not taken

• Exceptions

• Multicycle Instructions
# Alpha ALU Instructions

**RR-type instructions (addq, subq, xor, bis, cmplt):** \( \text{rc} \leftarrow \text{ra funct rb} \)

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>000</th>
<th>0</th>
<th>funct</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-13</td>
<td>12</td>
<td>11-5</td>
<td>4-0</td>
</tr>
</tbody>
</table>

**RI-type instructions (addq, subq, xor, bis, cmplt):** \( \text{rc} \leftarrow \text{ra funct ib} \)

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>ib</th>
<th>1</th>
<th>funct</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-13</td>
<td>12</td>
<td>11-5</td>
<td>4-0</td>
</tr>
</tbody>
</table>

**Encoding**

- \( \text{ib} \) is 8-bit unsigned literal

**Operation**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Op field</th>
<th>funct field</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>0x10</td>
<td>0x20</td>
</tr>
<tr>
<td>subq</td>
<td>0x10</td>
<td>0x29</td>
</tr>
<tr>
<td>bis</td>
<td>0x11</td>
<td>0x20</td>
</tr>
<tr>
<td>xor</td>
<td>0x11</td>
<td>0x40</td>
</tr>
<tr>
<td>cmoveq</td>
<td>0x11</td>
<td>0x24</td>
</tr>
<tr>
<td>cmplt</td>
<td>0x11</td>
<td>0x4D</td>
</tr>
</tbody>
</table>
Pipelined ALU Instruction Datapath

**IF**
- instruction fetch

**ID**
- instruction decode/ register fetch

**EX**
- execute

**MEM**
- memory access

**WB**
- write back

**IF/ID**
- Instr

**ID/EX**
- regA
- regB
- datW
- regW
- datA
- ALUout

**EX/MEM**
- Adata
- aluA
- aluB
- ALUout

**MEM/WB**
- datIn
- Data Mem.
- datOut
- addr

**PC**
- +4

**Instr. Mem.**
- incrPC

**Reg. Array**
- 25:21
- 20:16
- 20:13
- 4:0

**Wdata**
- Wdest
- Wdata
- Wdata
Data Hazards in Alpha Pipeline

Problem

• Registers read in ID, and written in WB
• Must resolve conflict between instructions competing for register array
  – Generally do write back in first half of cycle, read in second
• But what about intervening instructions?
• E.g., suppose initially $2$ is zero:

```
addq $31, 63, $2
addq $2, 0, $3
addq $2, 0, $4
addq $2, 0, $5
addq $2, 0, $6
```

$2$ written
Simulator Data Hazard Example

Operation

- Read in ID
- Write in WB
- Write-before-read register file

RAW Data Hazard

- Potential conflict among different instructions
- Due to data dependencies
- "Read After Write"
  - Register $2 written and then read

```
0x0: 43e7f402 addq r31, 0x3f, r2 # $2 = 0x3F
0x4: 40401403 addq r2, 0, r3 # $3 = 0x3F?
0x8: 40401404 addq r2, 0, r4 # $4 = 0x3F?
0xc: 40401405 addq r2, 0, r5 # $5 = 0x3F?
0x10: 40401406 addq r2, 0, r6 # $6 = 0x3F?
0x14: 47ff041f bis r31, r31, r31
0x18: 00000000 call_pal halt
demo04.O
```
Handling Hazards by Stalling

Idea

- Delay instruction until hazard eliminated
- Put “bubble” into pipeline
  - Dynamically generated NOP

Pipe Register Operation

- “Transfer” (normal operation) indicates should transfer next state to current
- “Stall” indicates that current state should not be changed
- “Bubble” indicates that current state should be set to 0
  - Stage logic designed so that 0 is like NOP
  - [Other conventions possible]
Detecting Dependencies

Pending Register Reads
- By instruction in ID
- ID_in.IR[25:21]: Operand A
- ID_in.IR[20:16]: Operand B
  - Only for RR

Pending Register Writes
- EX_in.WDst: Destination register of instruction in EX
- MEM_in.WDst: Destination register of instruction in MEM

CS 740 F ’98
Implementing Stalls

Stall Control Logic

- Determines which stages to stall, bubble, or transfer on next update

Rule:

- Stall in ID if either pending read matches either pending write
  - Also stall IF; bubble EX

Effect

- Instructions with pending writes allowed to complete before instruction allowed out of ID
Stalling for Data Hazards

Operation
- First instruction progresses unimpeded
- Second waits in ID until first hits WB
- Third waits in IF until second allowed to progress

\[
\begin{align*}
&\text{addq } 31, 63, 2 \\
&\text{addq } 2, 0, 3 \\
&\text{addq } 2, 0, 4 \\
&\text{addq } 2, 0, 5 \\
&\text{addq } 2, 0, 6 \\
\end{align*}
\]
Observations on Stalling

Good
- Relatively simple hardware
- Only penalizes performance when hazard exists

Bad
- As if placed NOPs in code
  - Except that does not waste instruction memory

Reality
- Some problems can only be dealt with by stalling
  - Instruction cache miss
  - Data cache miss
- Otherwise, want technique with better performance
Forwarding (Bypassing)

Observation

- ALU data generated at end of EX
  - Steps through pipe until WB
- ALU data consumed at beginning of EX

Idea

- Expedite passing of previous instruction result to ALU
- By adding extra data pathways and control
Forwarding for ALU Instructions

Operand Destinations
- ALU input A
  - Register EX_in.ASrc
- ALU input B
  - Register EX_in.BSrc

Operand Sources
- MEM_in.ALUout
  - Pending write to MEM_in.WDst
- WB_in.ALUout
  - Pending write to WB_in.WDst
Bypassing Possibilities

- From instruction that just finished EX

MEM-EX
- From instruction that finished EX two cycles earlier
Bypassing Data Hazards

Operation

• First instruction progresses down pipeline
• When in MEM, forward result to second instruction (in EX)
  – EX-EX forwarding
• When in WB, forward result to third instruction (in EX)
  – MEM-EX forwarding

```
addq $31, 63, $2
addq $2, 0, $3 # EX-EX
addq $2, 0, $4 # MEM-EX
addq $2, 0, $5
addq $2, 0, $6
```

$2 written

Time
Load & Store Instructions

Load: Ra <-- Mem[Rb + offset]

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
</tr>
</tbody>
</table>

Store: Mem[Rb + offset] <-- Ra

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
</tr>
</tbody>
</table>

ID: Instruction decode/register fetch

• Store: A <-- Register[IR[25:21]]
• B <-- Register[IR[20:16]]

MEM: Memory

• Load: Mem-Data <-- DMemory[ALUOutput]
• Store: DMemory[ALUOutput] <-- A

WB: Write back

• Load: Register[IR[25:21]] <-- Mem-Data
Some Hazards with Loads & Stores

Data Generated by Load

Load-ALU

\[
\text{ldq } \$1, 8(\$2) \\
\text{addq } \$2, \$1, \$2
\]

Load-Store Data

\[
\text{ldq } \$1, 8(\$2) \\
\text{stq } \$1, 16(\$2)
\]

Load-Store (or Load) Addr.

\[
\text{ldq } \$1, 8(\$2) \\
\text{stq } \$2, 16(\$1)
\]

Data Generated by Store

Store-Load Data

\[
\text{stq } \$1, 8(\$2) \\
\text{ldq } \$3, 8(\$2)
\]

ALU-Store Data

\[
\text{addq } \$1, \$3, \$2 \\
\text{stq } \$3, 8(\$2)
\]

ALU-Store (or Load) Addr

\[
\text{addq } \$2, \$3, \$1 \\
\text{stq } \$1, 16(\$2)
\]
Analysis of Data Transfers

Data Sources

• Available after EX
  – ALU Result Reg-Reg Result

• Available after MEM
  – Read Data Load result
  – ALU Data Reg-Reg Result passing through MEM stage

Data Destinations

• ALU A input Need in EX
  – Reg-Reg or Reg-Immediate Operand

• ALU B input Need in EX
  – Reg-Reg Operand
  – Load/Store Base

• Write Data Need in MEM
  – Store Data
Complete Bypassing for ALU & L/S

IF/ID
- Instr
- Xtnd
- regA
- regB
- datW
- regW
- datB
- 15:0
- 25:21
- 20:16
- 20:13
- 20:16
- 4:0
- 25:21

ID/EX
- Reg. Array
- aluA
- aluB
- Adata
- 20:13
- 20:16
- 4:0
- 25:21

EX/MEM
- MEM-MEM
- datIn
- Data Mem.
- datOut
- addr
- 20:16
- 4:0

MEM/WB
- ALUout
- 20:16
- 4:0

Wdata
- EX-EX
- MEM-EX

P C
- IncrPC
- +4

CS 740 F ’98
MEM-MEM Forwarding

Condition

- Data generated by load instruction
  - Register WB_in.WDst
- Used by immediately following store
  - Register MEM_in.ASrc

Load-Store Data

\[
\text{ldq } $1, 8($2) \\
\text{stq } $1, 16($2)
\]

\[
\text{ldq } $1, 8($2) \\
\text{stq } $1, 16($2)
\]
## Simulator Data Hazard Examples

- **demo5.O**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0:</td>
<td>43e7f402 addq r31, 0x3f, r2</td>
<td>$2 = 0x3F</td>
</tr>
<tr>
<td>0x4:</td>
<td>44420403 bis r1, r2, r3</td>
<td>$3 = 0x3F  EX-EX</td>
</tr>
<tr>
<td>0x8:</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0xc:</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0x10:</td>
<td>43e1f402 addq r31, 0xf, r2</td>
<td>$2 = 0xF</td>
</tr>
<tr>
<td>0x14:</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0x18:</td>
<td>44420403 bis r1, r2, r3</td>
<td>$3 = 0xF  MEM-EX</td>
</tr>
<tr>
<td>0x1c:</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0x20:</td>
<td>43e11403 addq r31, 0x8, r3</td>
<td>$3 = 8</td>
</tr>
<tr>
<td>0x24:</td>
<td>43e21402 addq r31, 0x10, r2</td>
<td>$2 = 0x10</td>
</tr>
<tr>
<td>0x28:</td>
<td>b4620000 stq r3, 0(r2)</td>
<td>Mem[0x10] = 8  MEM-EX, EX-EX</td>
</tr>
<tr>
<td>0x2c:</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0x30:</td>
<td>a4830008 ldq r4, 8(r3)</td>
<td>$4 = 8</td>
</tr>
<tr>
<td>0x34:</td>
<td>40820405 addq r4, r2, r5</td>
<td>$5 = 0x18  Stall 1, MEM-EX</td>
</tr>
<tr>
<td>0x38:</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0x3c:</td>
<td>00000000 call_pal halt</td>
<td></td>
</tr>
</tbody>
</table>
Impact of Forwarding

Single Remaining Unsolved Hazard Class

- Load followed by ALU operation
  - Including address calculation

```
Load-ALU

ldq $1, 8($2)
addq $2, $1, $2
```

Just Forward?

```
IF  ID  EX  M  WB
ldq $1, 8($2)
```
```
IF  ID  EX  M  WB
addq $2, $1, $2
```

Value not available soon enough!

With 1 Cycle Stall

```
IF  ID  EX  M  WB
ldq $1, 8($2)
```
```
IF  ID  ID  EX  M  WB
addq $2, $1, $2
```

Then can use MEM-EX forwarding
Methodology for characterizing and Enumerating Data Hazards

The space of data hazards (from a program-centric point of view) can be characterized by 3 independent axes:

3 possible write regs (axis 1):
   RR.rc, RI.rc, Load.ra

6 possible read regs (axis 2):
   RR.ra, RR.rb, RI.ra, Load.ra, Store.ra, Store.rb

A dependent read can be a distance of either 1 or 2 from the corresponding write (axis 3):

\[
\begin{array}{c|c|c}
\text{OP} & \text{writes} & \text{reads} \\
\hline
\text{RR} & \text{rc} & \text{ra, rb} \\
\text{RI} & \text{rc} & \text{ra} \\
\text{Load} & \text{ra} & \text{rb} \\
\text{Store} & & \text{ra, rb}
\end{array}
\]

\text{distance 2 hazard:}
\begin{align*}
\text{RR.rc/RR.ra/2} & : \text{addq} \$31, 63, \$2 \\
& : \text{addq} \$31, \$2, \$3 \\
& : \text{addu} \$2, \$31, \$4
\end{align*}

\text{distance 1 hazard:}
\begin{align*}
\text{RR.rc/RR.rb/1} & : \text{addq} \$31, 63, \$2 \\
& : \text{addq} \$31, \$2, \$3 \\
& : \text{addu} \$2, \$31, \$4
\end{align*}
Enumerating data hazards

Testing Methodology
- 36 cases to cover all interactions between RR, RI, Load, & Store
- Would need to consider more read source and write destinations when add other instruction types
Simulator Microtest Example

- **Tests for single failure mode**
  - ALU Rc -> ALU Ra
  - distance 1
  - RR.rc/RR.ra/1
- **Hits `call_pal 0` when error**
- **Jumps to `call_pal 1` when OK**
- **Error case shields successful case**
- **Grep for ERROR or `call_pal 0`**

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td><code>43e21402 addq r31, 0x10, r2</code></td>
<td>$2 = 0x10</td>
</tr>
<tr>
<td>0x4</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0xc</td>
<td><code>43e20405 addq r31, r2, r5</code></td>
<td># $5 = 0x10</td>
</tr>
<tr>
<td>0x10</td>
<td><code>43e50401 addq r31, r5, r1</code></td>
<td># $1 = 0x10</td>
</tr>
<tr>
<td>0x14</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x1c</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td><code>44221803 xor r1, 0x10, r3</code></td>
<td># $1 should == 0</td>
</tr>
<tr>
<td>0x24</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x28</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x2c</td>
<td><code>e4600006 beq r3, 0x48</code></td>
<td># Should take</td>
</tr>
<tr>
<td>0x30</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x34</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x38</td>
<td><code>00000000 call_pal halt</code></td>
<td># Failure</td>
</tr>
<tr>
<td>0x3c</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x44</td>
<td><code>47ff041f bis r31, r31, r31</code></td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td><code>00000001 call_pal cflush</code></td>
<td># Success</td>
</tr>
</tbody>
</table>
Branch Instructions

Cond. Branch: \( \text{PC} \leftarrow \text{Cond}(\text{Ra}) \ ? \ \text{PC} + 4 + \text{disp} \times 4 \ : \ \text{PC} + 4 \)

\[
\begin{array}{c|c|c}
\text{Op} & \text{ra} & \text{disp} \\
31-26 & 25-21 & 20-0 \\
\end{array}
\]

Sources
- PC, Ra

Destinations
- PC

Branch [Subroutine] (br, bsr): \( \text{Ra} \leftarrow \text{PC} + 4; \ \text{PC} \leftarrow \text{PC} + 4 + \text{disp} \times 4 \)

\[
\begin{array}{c|c|c}
\text{Op} & \text{ra} & \text{disp} \\
31-26 & 25-21 & 20-0 \\
\end{array}
\]

Sources
- PC

Destinations
- PC, Ra
New Data Hazards

Branch Uses Register Data

- Generated by ALU instruction
- Read from register in ID

Handling

- Same as other instructions with register data source
- Bypass
  - EX-EX
  - MEM-EX

ALU-Branch

\[
\text{addq} \hspace{0.5em} 2, \hspace{0.5em} 3, \hspace{0.5em} 1 \\
\text{beq} \hspace{0.5em} 1, \hspace{0.5em} \text{targ}
\]

Distant ALU-Branch

\[
\text{addq} \hspace{0.5em} 2, \hspace{0.5em} 3, \hspace{0.5em} 1 \\
\text{bis} \hspace{0.5em} 31, \hspace{0.5em} 31, \hspace{0.5em} 31 \\
\text{beq} \hspace{0.5em} 1, \hspace{0.5em} \text{targ}
\]

Load-Branch

\[
\text{lw} \hspace{0.5em} 1, \hspace{0.5em} 8(2) \\
\text{beq} \hspace{0.5em} 1, \hspace{0.5em} \text{targ}
\]
Jump Instructions

jmp, jsr, ret: Ra <-- PC+4; PC <-- Rb

<table>
<thead>
<tr>
<th>0x1A</th>
<th>ra</th>
<th>rb</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
</tr>
</tbody>
</table>

Sources
- PC, Rb

Destinations
- PC, Ra

Hint
Still More Data Hazards

Jump Uses Register Data

- Generated by ALU instruction
- Read from register in ID

Handling

- Same as other instructions with register data source
- Bypass
  - EX-EX
  - MEM-EX

ALU-Jump

```
addq $2, $3, $1
jsr $26 ($1), 1
```

Distant ALU-Jump

```
addq $2, $3, $1
bis $31, $31, $31
jmp $31 ($1), 1
```

Load-Jump

```
lw $26, 8($sp)
ret $31 ($26), 1
```
### Enumerating data hazards

#### Cases
- 2 distances (either 1 or 2)
- 5 classes of writer
- 8 classes of readers

#### Testing Methodology
- 80 cases to cover all interactions between supported instruction types

<table>
<thead>
<tr>
<th>writes</th>
<th>RR.ra</th>
<th>RR.rb</th>
<th>RI.ra</th>
<th>L.rb</th>
<th>S.ra</th>
<th>S.rb</th>
<th>BXX.ra</th>
<th>J.rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR.rc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RL.rc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.ra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR.ra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J.ra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelined datapath

What happens with a branch?

IF
instruction
fetch

ID
instruction decode/
register fetch

EX
execute/
address calc

MEM
memory
access

WB
write
back
Conditional Branch Instruction Handling

**beq:** PC <-- Ra == 0 ? PC + 4 + disp*4 : PC + 4

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>disp</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-0</td>
</tr>
</tbody>
</table>

- **Instr. Mem.**
- **Instr.**
- **Reg. Array**
  - regA
  - regB
  - regW
  - datW
  - datA
  - datB
- **ALU**
  - aluA
  - aluB
- **Zero Test**
  - Adata
- **ALUout**
- **Branch Flag**
- **IF/ID**
  - 20:0
  - 25:21
  - 20:16
- **ID/EX**
  - Xtdn << 2
- **EX/MEM**
- **IncrPC**
- **+4**
- **PC**
Branch on equal

\[
\text{beq: } PC \leftarrow \text{Ra} == 0 \ ? \ PC + 4 + \text{disp} \times 4 : PC + 4
\]

<table>
<thead>
<tr>
<th>0x39</th>
<th>ra</th>
<th>disp</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-0</td>
</tr>
</tbody>
</table>

IF: Instruction fetch
- IR \leftarrow \text{IMemory}[PC]
- incrPC \leftarrow PC + 4

ID: Instruction decode/register fetch
- A \leftarrow \text{Register}[IR[25:21]]

Ex: Execute
- Target \leftarrow \text{incrPC} + \text{SignExtend}(IR[20:0]) \times 2
- Z \leftarrow (A == 0)

MEM: Memory
- PC \leftarrow Z \ ? \ Target : incrPC

WB: Write back
- nop
Branch Example

Desired Behavior

• Take branch at 0x00
• Execute target 0x18
  – PC + 4 + disp << 2
  – PC = 0x00
  – disp = 5

Branch Code (demo08.O)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>e7e00005 beq r31, 0x18</td>
<td># Take</td>
</tr>
<tr>
<td>0x4</td>
<td>43e7f401 addq r31, 0x3f, r1</td>
<td># (Skip)</td>
</tr>
<tr>
<td>0x8</td>
<td>43e7f402 addq r31, 0x3f, r2</td>
<td># (Skip)</td>
</tr>
<tr>
<td>0xc</td>
<td>43e7f403 addq r31, 0x3f, r3</td>
<td># (Skip)</td>
</tr>
<tr>
<td>0x10</td>
<td>43e7f404 addq r31, 0x3f, r4</td>
<td># (Skip)</td>
</tr>
<tr>
<td>0x14</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>43e7f405 addq r31, 0x3f, r5</td>
<td># (Target)</td>
</tr>
<tr>
<td>0x1c</td>
<td>47ff041f bis r31, r31, r31</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>00000000 call_pal halt</td>
<td></td>
</tr>
</tbody>
</table>
Branch Hazard Example

0x0: beq r31, 0x18  # Take
0x4: addq r31, 0x3f, r1  # Xtra1
0x8: addq r31, 0x3f, r2  # Xtra2
0xc: addq r31, 0x3f, r3  # Xtra3
0x10: addq r31, 0x3f, r4  # Xtra4
0x18: addq r31, 0x3f, r5  # Target

• With BEQ in Mem stage
Branch Hazard Example (cont.)

0x0: \texttt{beq} r31, 0x18 \# Take
0x4: \texttt{addq} r31, 0x3f, r1 \# Xtra1
0x8: \texttt{addq} r31, 0x3f, r2 \# Xtra2
0xc: \texttt{addq} r31, 0x3f, r3 \# Xtra3
0x10: \texttt{addq} r31, 0x3f, r4 \# Xtra4
0x18: \texttt{addq} r31, 0x3f, r5 \# Target

- One cycle later
- Problem: Will execute 3 extra instructions!
Branch Hazard Pipeline Diagram

Problem

- Instruction fetched in IF, branch condition set in MEM

```
beq $31, target
addq $31, 63, $1
addq $31, 63, $2
addq $31, 63, $3
addq $31, 63, $4
target: addq $31, 63, $5
```
Stall Until Resolve Branch

- Detect when branch in stages ID or EX
- Stop fetching until resolve
  - Stall IF. Inject bubble into ID

Perform when branch in either stage
Stalling Branch Example

```plaintext
0x0: beq r31, 0x18  # Take
0x4: addq r31, 0x3f, r1  # Xtra1
0x8: addq r31, 0x3f, r2  # Xtra2
0xc: addq r31, 0x3f, r3  # Xtra3
0x10: addq r31, 0x3f, r4  # Xtra4
0x18: addq r31, 0x3f, r5  # Target
```

- With BEQ in Mem stage
- Will have stalled twice
  - Injects two bubbles

![Diagram of instruction pipeline showing stalls and bubbles.]
Taken Branch Resolution

- When branch taken, still have instruction Xtra1 in pipe
- Need to flush it when detect taken branch in Mem
  - Convert it to bubble

![Stall Control Diagram]

Perform when detect taken branch
**taken Branch Resolution Example**

\[
\begin{align*}
0x0: & \quad \textbf{beq} \quad r31, 0x18 \quad \# \text{Take} \\
0x4: & \quad \textbf{addq} \quad r31, 0x3f, r1 \quad \# \text{Xtra1} \\
0x8: & \quad \textbf{addq} \quad r31, 0x3f, r2 \quad \# \text{Xtra2} \\
0xc: & \quad \textbf{addq} \quad r31, 0x3f, r3 \quad \# \text{Xtra3} \\
0x10: & \quad \textbf{addq} \quad r31, 0x3f, r4 \quad \# \text{Xtra4} \\
0x18: & \quad \textbf{addq} \quad r31, 0x3f, r5 \quad \# \text{Target}
\end{align*}
\]

- When branch taken
- Generate 3rd bubble
- Begin fetching at target
Taken Branch Pipeline Diagram

Behavior

- Instruction Xtra1 held in IF for two extra cycles
- Then turn into bubble as enters ID

```
beq     $31, target
addq    $31, 63, $1  # Xtra1

target: addq $31, 63, $5  # Target
```
Not Taken Branch Resolution

- Stall two cycles with not-taken branches as well
- When branch not taken, already have instruction Xtra1 in pipe
- Let it proceed as usual
Not Taken Branch Resolution Example

deemo09.O

|-------------|------------|-----|

- Branch not taken
- Allow instructions to proceed

```
0x0:  bne  r31, 0x18  # Don’t Take
0x4:  addq r31, 0x3f, r1  # Xtra1
0x8:  addq r31, 0x3f, r2  # Xtra2
0xc:  addq r31, 0x3f, r3  # Xtra3
0x10: addq r31, 0x3f, r4  # Xtra4
```
Not Taken Branch Pipeline Diagram

Behavior

- Instruction Xtra1 held in IF for two extra cycles
- Then allowed to proceed

```
beq    $31, target
addq  $31, 63, $1  # Xtra1
addq  $31, 63, $2  # Xtra2
addq  $31, 63, $3  # Xtra3
addq  $31, 63, $4  # Xtra4
```
Analysis of Stalling

Branch Instruction Timing

- 1 instruction cycle
- 3 extra cycles when taken
- 2 extra cycles when not taken

Performance Impact

- Branches 16% of instructions in SpecInt92 benchmarks
- 67% branches are taken
- Adds $0.16 \times (0.67 \times 3 + 0.33 \times 2) = 0.43$ cycles to CPI
  - Average number of cycles per instruction
  - Serious performance impact
Fetch & Cancel When Taken

- Instruction does not cause any updates until MEM or WB stages
- Instruction can be “cancelled” from pipe up through EX stage
  - Replace with bubble

Strategy

- Continue fetching under assumption that branch not taken
- If decide to take branch, cancel undesired ones

Perform when detect taken branch
Canceling Branch Example

0x0: `beq` r31, 0x18  # Take
0x4: `addq` r31, 0x3f, r1  # Xtra1
0x8: `addq` r31, 0x3f, r2  # Xtra2
0xc: `addq` r31, 0x3f, r3  # Xtra3
0x10: `addq` r31, 0x3f, r4  # Xtra4
0x18: `addq` r31, 0x3f, r5  # Target

- With BEQ in Mem stage
- Will have fetched 3 extra instructions
- But no register or memory updates
Canceling Branch Resolution Example

0x0: `beq` r31, 0x18 # Take
0x4: `addq` r31, 0x3f, r1 # Xtra1
0x8: `addq` r31, 0x3f, r2 # Xtra2
0xc: `addq` r31, 0x3f, r3 # Xtra3
0x10: `addq` r31, 0x3f, r4 # Xtra4
0x18: `addq` r31, 0x3f, r5 # Target

- When branch taken
- Generate 3 bubbles
- Begin fetching at target
Canceling Branch Pipeline Diagram

Operation

• Process instructions assuming branch will not be taken
• When is taken, cancel 3 following instructions

```
beq $31, target
addq $31, 63, $1
addq $31, 63, $2
addq $31, 63, $3
addq $31, 63, $4
target: addq $31, 63, $5
```
Noncanceling Branch Pipeline Diagram

Operation

- Process instructions assuming branch will not be taken
- If really isn’t taken, then instructions flow unimpeded

```
<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
</tr>
</tbody>
</table>
```

```
bne $31, target
addq $31, 63, $1
addq $31, 63, $2
addq $31, 63, $3
addq $31, 63, $4
target: addq $31, 63, $5
```

PC Not Updated

Time
Branch Prediction Analysis

Our Scheme Implements “Predict Not Taken”
  • But 67% of branches are taken
  • Impact on CPI: $0.16 \times 0.67 \times 3.0 = 0.32$
    – Still not very good

Alternative Schemes
  • Predict taken
    – Would be hard to squeeze into our pipeline
      » Can’t compute target until ID
  • Backwards taken, forwards not taken
    – Predict based on sign of displacement
    – Exploits fact that loops usually closed with backward branches
Exceptions

An exception is a transfer of control to the OS in response to some event (i.e. change in processor state)
Issues with Exceptions

A1: What kinds of events can cause an exception?

A2: When does the exception occur?

B1: How does the handler determine the location and cause of the exception?

B2: Are exceptions allowed within exception handlers?

C1: Can the user process restart?

C2: If so, where?
Internal (CPU) Exceptions

Internal exceptions occur as a result of events generated by executing instructions.

Execution of a CALL_PAL instruction.
  • allows a program to transfer control to the OS

Errors during instruction execution
  • arithmetic overflow, address error, parity error, undefined instruction

Events that require OS intervention
  • virtual memory page fault
External (I/O) exceptions

External exceptions occur as a result of events generated by devices external to the processor.

I/O interrupts
- hitting ^C at the keyboard
- arrival of a packet
- arrival of a disk sector

Hard reset interrupt
- hitting the reset button

Soft reset interrupt
- hitting ctl-alt-delete on a PC
Exception handling (hardware tasks)

Recognize event(s)

Associate one event with one instruction.
  - external event: pick any instruction
  - multiple internal events: typically choose the earliest instruction.
  - multiple external events: prioritize
  - multiple internal and external events: prioritize

Create Clean Break in Instruction Stream
  - Complete all instructions before excepting instruction
  - Abort excepting and all following instructions
    - this clean break is called a “precise exception”
Exception handling (hardware tasks)

Set status registers

- Exception Address: the EXC_ADDR register
  - external exception: address of instruction about to be executed
  - internal exception: address of instruction causing the exception
    » except for arithmetic exceptions, where it is the following instruction
- Cause of the Exception: the EXC_SUM and FPCR registers
  - was the exception due to division by zero, integer overflow, etc.
- Others
  - which ones get set depends on CPU and exception type

Disable interrupts and switch to kernel mode

Jump to common exception handler location
Exception handling (software tasks)

Deal with event

(Optionally) resume execution
  • using special \texttt{REI} (return from exception or interrupt) instruction
  • similar to a procedure return, but restores processor to user mode as a side effect.

Where to resume execution?
  • usually re-execute the instruction causing exception
Precise vs. Imprecise Exceptions

In the Alpha architecture:

- arithmetic exceptions may be *imprecise* (similar to the CRAY-1)
  - motivation: simplifies pipeline design, helping to increase performance
- all other exceptions are precise

Imprecise exceptions:

- all instructions before the excepting instruction complete
- the excepting instruction and instructions after it may or may not complete

What if precise exceptions are needed?

- insert a TRAPB (trap barrier) instruction immediately after
  - stalls until certain that no earlier insts take exceptions

In the remainder of our discussion, assume for the sake of simplicity that all Alpha exceptions are precise.
Example: Integer Overflow

(This example illustrates a precise version of the exception.)

user code

```
and $12, $2, $5
xor $13, $2, $6
addq $1, $2, $1
or $15, $6, $7
ldq $16, 50($7)
```

handler code

```
stq $26, 100($31)
```

Overflow detected here

flush these instructions

start handler code

the xor instruction completes

(This example illustrates a precise version of the exception.)
Exception Handling in pAlpha Simulator

Relevant Pipeline State

- **Address of instruction in pipe stage (SPC)**
- **Exception condition (EXC)**
  - Set in stage when problem encountered
    - IF for fetch problems, EX for instr. problems, MEM for data probs.
  - Triggers special action once hits WB
Alpha Exception Examples

• In directory HOME740/public/sim/demos

**Illegal Instruction (exc01.O)**

```
0x0: sll r3, 0x8, r5  # unimplemented
0x4: addq r31, 0x4, r2  # should cancel
```

**Illegal Instruction followed by store (exc02.O)**

```
0x0: addq r31, 0xf, r2
0x4: sll r3, 0x8, r5  # unimplemented
0x8: stq r2, 8(r31)   # should cancel
```
More Examples: Multiple Exceptions

**EX exception follows MEM exception (exc03.O)**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>addq r31, 0x3, r3</td>
<td></td>
</tr>
<tr>
<td>0x4</td>
<td>stq r3, -4(r31)</td>
<td># bad address</td>
</tr>
<tr>
<td>0x8</td>
<td>sll r3, 0x8, r5</td>
<td># unimplemented</td>
</tr>
<tr>
<td>0xc</td>
<td>addq r31, 0xf, r2</td>
<td></td>
</tr>
</tbody>
</table>

These exceptions are detected *simultaneously* in the pipeline!

**MEM exception follows EX exception (exc04.O)**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>addq r31, 0x3, r3</td>
<td></td>
</tr>
<tr>
<td>0x4</td>
<td>sll r3, 0x8, r5</td>
<td># unimplemented</td>
</tr>
<tr>
<td>0x8</td>
<td>stq r3, -4(r31)</td>
<td># bad address</td>
</tr>
<tr>
<td>0xc</td>
<td>addq r31, 0xf, r2</td>
<td></td>
</tr>
</tbody>
</table>

Which is the excepting instruction?
Final Alpha Exception Example

Avoiding false alarms (exc05.O)

0x0:  beq   r31, 0xc    # taken
0x4:  sll   r3, 0x8, r5  # should cancel
0x8:  bis   r31, r31, r31
0xc:  addq r31, 0x1, r2  # target
0x10: call_pal  halt

Exception detected in the pipeline, but should not really occur.
Implementation Features

Correct

• Detects excepting instruction
  – Furthest one down pipeline = Earliest one in program order
  – (e.g., exc03.O vs. exc04.O)
• Completes all preceding instructions
• Usually aborts excepting instruction & beyond
• Prioritizes exception conditions
  – Earliest stage where instruction ran into problems
• Avoids false alarms (exc05.O)
  – Problematic instructions that get canceled anyhow

Shortcomings

• Store following excepting instruction (exc02.O)
Requirements for Full Implementation

Exception Detection
  • Detect external interrupts at IF
    – Complete all fetched instructions

Instruction Shutdown
  • Suspend if unusual condition in MEM or WB
  • Save proper value of EXC_ADDR
    – Not always same as SPC
  • Rest of control state

Handler Startup
  • Begin fetching handler code
Multicycle instructions

Alpha 21264 Execution Times:

- Measured in clock cycles

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer</th>
<th>FP-Single</th>
<th>FP-Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>add / sub</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>multiply</td>
<td>8-16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>divide</td>
<td>N / A</td>
<td>10</td>
<td>23</td>
</tr>
</tbody>
</table>

H&P Dynamic Instruction Counts:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer Benchmarks</th>
<th>FP Benchmarks Integer</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>add / sub</td>
<td>14%</td>
<td>11%</td>
<td>14%</td>
</tr>
<tr>
<td>multiply</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
<td>13%</td>
</tr>
<tr>
<td>divide</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Pipeline Revisited

Integer Add / Subtract

FP Add / Sub / Mult

Integer Multiply

FP Single-Precision Divide
Multiply Timing Example

bis $31, 3, $2
bis $31, 7, $3
mulq $2, $3, $4
addq $2, $3, $3
bis $4, $31, $5
addq $2, $4, $2

(Not to scale)

Stall while Busy
Floating Point Hardware (from MIPS)

**Independent Hardware Units**
- Can concurrently execute add, divide, multiply
- Except that all share exponent and rounding units
- Independent of integer operations
Control Logic

Busy Flags
- One per hardware unit
- One per FP register
  - Destination of currently executing operation

Stall Instruction in ID if:
- Needs unit that is not available
- Source register busy
  - Avoids RAW (Read-After-Write) hazard
- Destination register busy
  - Avoids WAW hazard

Bypass paths
- Similar to those in integer pipeline

\[
\text{divt}\ f1,\ f2,\ f4
\text{addt}\ f1,\ f2,\ f4
\]
FP Timing Example

\[
\begin{align*}
\text{addt } &\text{ $f2, \ f4, \ f1$} & \text{IF} & \text{ID} & \text{EX} & \text{M} & \text{WB} \\
\text{divt } &\text{ $f1, \ f2, \ f6$} & \text{IF} & \text{ID} & \ldots & \text{EX} & \text{M} & \text{WB} \\
\text{addt } &\text{ $f1, \ f2, \ f8$} & \text{IF} & \text{ID} & \ldots & \text{EX} & \text{M} & \text{WB} \\
\text{addt } &\text{ $f1, \ f2, \ f6$} & \text{IF} & \text{ID} & \ldots & \text{EX} & \\
& & & & & & &
\end{align*}
\]

RAW Stall

Structural Stall

WAW Stall
Conclusion

Pipeline Characteristics for Multi-cycle Instructions

• In-order issue
  – Instructions fetched and decoded in program order

• Out-of-order completion
  – Slow instructions may complete after ones that are later in program order

Performance Opportunities

• Transformations such as loop unrolling & software pipelining to expose potential parallelism

• Schedule code to use multiple functional units
  – Must understand idiosyncracies of pipeline structure