Handling Data Hazards in an Alpha Pipeline
March 17, 1998

Topics
- Hazards in ALU instructions
- Handling by stalling
- Handling by forwarding
- Data hazards with other instruction types
- Systematic testing of hazard-handling logic
# Alpha ALU Instructions

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

<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):** $rc \leftarrow ra \text{ funct } ib$

<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**

- $ib$ 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
ID/EX
EX/MEM
MEM/WB

PC
Instr.
Mem.

Instr.
Mem.

Reg.
Array

aluA
aluB

Adata

ALUout

datIn
Data Mem.
datOut
addr

datW

datA

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

Wdata

Wdest

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
```
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]
**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
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
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

```
<table>
<thead>
<tr>
<th>$2</th>
<th>$3</th>
<th>$4</th>
<th>$5</th>
<th>$6</th>
</tr>
</thead>
</table>
```

```
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

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

Data Generated by ALU

ALU-Store (or Load) Addr

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

ALU-Store Data

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

Not a concern for us
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
  - PC
  - Instr Mem.
  - +4
  - IncrPC

- ID/EX
  - Xtnd
  - regA
  - regB
  - regW
  - datA
  - datB
  - ALU
    - 25:21
    - 20:16
    - +4
  - ALUout
    - Adata
    - ALUout

- EX/MEM
  - Wdest
  - Wdata
  - MEM-MEM
    - datIn
    - Data Mem.
    - datOut
    - addr
  - ALUout
  - MEM-EX

- MEM/WB
  - MEM-EX
  - MEM-EX
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

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

- demo5.O

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

Single Remaining Unsolved Hazard Class

- Load followed by ALU operation
  - Including address calculation

Load-ALU

\[
\begin{align*}
\text{ldq } & \$1, \ 8(\$2) \\
\text{addq } & \$2, \$1, \$2
\end{align*}
\]

Just Forward?

\[
\begin{array}{c|c|c|c|c|c}
\text{IF} & \text{ID} & \text{EX} & \text{M} & \text{WB} \\
\hline
\text{ldq } & \$1, \ 8(\$2) \\
\hline
\text{IF} & \text{ID} & \text{EX} & \text{M} & \text{WB} \\
\hline
\text{addq } & \$2, \$1, \$2 \\
\hline
\end{array}
\]

Value not available soon enough!

With 1 Cycle Stall

\[
\begin{array}{c|c|c|c|c|c}
\text{IF} & \text{ID} & \text{EX} & \text{M} & \text{WB} \\
\hline
\text{ldq } & \$1, \ 8(\$2) \\
\hline
\text{IF} & \text{ID} & \text{ID} & \text{EX} & \text{M} & \text{WB} \\
\hline
\text{addq } & \$2, \$1, \$2 \\
\hline
\end{array}
\]

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):

distance 2 hazard:
RR.rc/RR.ra/2

addq $31, 63, $2
addq $31, $2, $3
addu $2, $31, $4

distance 1 hazard:
RR.rc/RR.rb/1
### Enumerating data hazards

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>RR.rc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI.rc</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>
</tr>
</tbody>
</table>

**distance = 1**

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>RR.rc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI.rc</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>
</tr>
</tbody>
</table>

**distance = 2**

### 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`

```
0x0: 43e21402 addq r31, 0x10, r2 $2 = 0x10
0x4: 47ff041f bis r31, r31, r31
0x8: 47ff041f bis r31, r31, r31
0xc: 43e20405 addq r31, r2, r5 # $5 = 0x10
0x10: 43e50401 addq r31, r5, r1 # $1 = 0x10
0x14: 47ff041f bis r31, r31, r31
0x18: 47ff041f bis r31, r31, r31
0x1c: 47ff041f bis r31, r31, r31
0x20: 44221803 xor r1, 0x10, r3 # $1 should == 0
0x24: 47ff041f bis r31, r31, r31
0x28: 47ff041f bis r31, r31, r31
0x2c: e4600006 beq r3, 0x48 # Should take
0x30: 47ff041f bis r31, r31, r31
0x34: 47ff041f bis r31, r31, r31
0x38: 00000000 call_pal halt # Failure
0x3c: 47ff041f bis r31, r31, r31
0x40: 47ff041f bis r31, r31, r31
0x44: 47ff041f bis r31, r31, r31
0x48: 00000001 call_pal cflush # Success
demo7.O
```