Pipelining

September 9, 2003

Topics - 1
• Objective
• Instruction formats
• Instruction processing
• Principles of pipelining
• Inserting pipe registers

Topics - 2
• Data Hazards
  – Stalling and Forwarding
  – Systematic testing of hazard-handling logic
• Control Hazards
  – Stalling, Predict not taken
• Exceptions
• Multicycle Instructions

Objective

Design Processor for Alpha Subset
• Interesting but not overwhelming quantity
• High level functional blocks

Initial Design
• One instruction at a time
• Single cycle per instruction
  – Follows H&P Ch. A.1

Refined Design
• 5-stage pipeline
  – Similar to early RISC processors
  – Follows H&P Ch. A.2-3
• Goal: approach 1 cycle per instruction but with shorter cycle time

What Makes it Hard
• Hazards, exceptions (A.4-6)

Alpha Arithmetic Instructions

<p>| RR-type instructions (addq, subq, xor, bis, cmplt): rc &lt;-- ra funct rb |
|-----------------|----|----|---|---|</p>
<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>rb</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>
</tr>
</tbody>
</table>

| RI-type instructions (addq, subq, xor, bis, cmplt): rc <-- ra funct ib |
|-----------------|----|---|---|
| Op | ra | ib | funct | rc |
|-----------------|----|---|---|
| 31-26 | 25-21 | 20-13 | 12 | 11-5 | 4-0 |

Encoding
• ib is 8-bit unsigned literal

Operation | Op field | funct field
---|---|---|
addq | 0x10 | 0x20
subq | 0x10 | 0x29
bis | 0x11 | 0x20
xor | 0x11 | 0x40
cmoveq | 0x11 | 0x24
cmplt | 0x11 | 0x4D

Alpha Load/Store Instructions

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

Encoding
• offset is 16-bit signed offset

Operation | Op field
---|---|
ldq | 0x29
stq | 0x2D
Branch Instructions

Cond. Branch: PC \textless\textgreater\ Cond(Ra) ? 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>

Encoding

• disp is 21-bit signed displacement

Operation Op field Cond
beq 0x39 Ra == 0
bne 0x3D Ra != 0

Branch [Subroutine] (br, bsr): Ra \leftarrow PC + 4; PC \leftarrow PC + 4 + disp*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>

Operation Op field
br 0x30
bsr 0x34

Transfers of Control

jmp, jsr, ret: Ra \leftarrow PC+4; PC \leftarrow Rb

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

Encoding

• High order 2 bits of Hint encode jump type
• Remaining bits give information about predicted destination
• Hint does not affect functionality

Jump Type Hint 15:14
jmp 00
jsr 01
ret 10
call_pal

| Number |
| 31-26 | 25-0 |

• Use as halt instruction

Instruction Encoding

0x0: 40220403 addq r1, r2, r3
0x4: 4487f805 xor r4, 0xf3, r5
0x8: a4c70abc ldq r6, 2748(r7)
0xc: b5090123 stq r8, 291(r9)
0x10: e47ffffb beq r3, 0
0x14: d35ffffa bsr r26, 0(r31)
0x18: 6bfa8001 ret r31, (r26), 1
0x1c: 000abcde call_pal 0xabcde

Object Code

• Instructions encoded in 32-bit words
• Program behavior determined by bit encodings
• Disassembler simply converts these words to readable instructions

Decoding Examples

0x0: 40220403 addq r1, r2, r3

0x8: a4c70abc ldq r6, 2748(r7)

0x10: e47ffffb beq r3, 0

0x18: 6bfa8001 ret r31, (r26), 1

Target = 16 # Current PC
+ 4 # Increment
+ 4 * -5 # Disp
= 0
**Datapath**

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>instruction decode/ register fetch</td>
<td>execute/ address calculation</td>
<td>memory access</td>
<td>write back</td>
</tr>
</tbody>
</table>

**Hardware Units**

**Storage**
- Instruction Memory
  - Fetch 32-bit instructions
- Data Memory
  - Load / store 64-bit data
- Register Array
  - Storage for 32 integer registers
  - Two read ports: can read two registers at once
  - Single write port

**Functional Units**
- +4 PC incrementer
- Xtnd Sign extender
- ALU Arithmetic and logical instructions
- Zero Test Detect whether operand == 0

**RR-type instructions**

RR-type instructions (addq, subq, xor, bis, cmplt):

<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>

**IF: Instruction fetch**
- IR <-- IMemory[PC]
- PC <-- PC + 4

**ID: Instruction decode/register fetch**
- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

**Ex: Execute**
- ALUOutput <-- A op B

**MEM: Memory**
- nop

**WB: Write back**
- Register[IR[4:0]] <-- ALUOutput

**Active Datapath for RR & RI**

**ALU Operation**
- Input B selected according to instruction type
  - datB for RR, IR[20:13] for RI
- ALU function set according to operation type

**Write Back**
- To Rc
Active Datapath for RR & RI

IF: Instruction fetch
• IR <-- IMemory[PC]
• PC <-- PC + 4

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

Ex: Execute
• ALUOutput <-- A op B

MEM: Memory
• nop

WB: Write back
• Register[IR[4:0]] <-- ALUOutput

ALU Operation
• Input B selected according to instruction type
  - datB for RR, IR[20:13] for RI
• ALU function set according to operation type

Write Back
• To Rc

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

IF: Instruction fetch
• IR <-- IMemory[PC]
• PC <-- PC + 4

ID: Instruction decode/register fetch
• B <-- Register[IR[20:16]]

Ex: Execute
• ALUOutput <-- B + SignExtend(IR[15:0])

MEM: Memory
• Mem-Data <-- DMemory[ALUOutput]

WB: Write back
• Register[IR[25:21]] <-- Mem-Data

ALU Operation
• Used to compute address
  - A input set to extended IR[15:0]
• ALU function set to add

Memory Operation
• Read for load, write for store

Write Back
• To Ra for load
• None for store

RI-type instructions

RI-type instructions (addq, subq, xor, bis, cmpq):
rc <-- ra funct ib

Op  ra  ib  l  funct  rc
31-26  25-21  20-13  12  11-5  4-0

IF: Instruction fetch
• IR <-- IMemory[PC]
• PC <-- PC + 4

ID: Instruction decode/register fetch
• A <-- Register[IR[25:21]]
• B <-- IR[20:13]

Ex: Execute
• ALUOutput <-- A op B

MEM: Memory
• nop

WB: Write back
• Register[IR[4:0]] <-- ALUOutput

Active Datapath for Load & Store

ALU Operation
• Used to compute address
  - A input set to extended IR[15:0]
• ALU function set to add

Memory Operation
• Read for load, write for store

Write Back
• To Ra for load
• None for store
**Store instruction**

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>

**IF: Instruction fetch**
- IR <-- IMemory[PC]
- PC <-- PC + 4

**ID: Instruction decode/register fetch**
- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

**Ex: Execute**
- ALUOutput <-- B + SignExtend(IR[15:0])

**MEM: Memory**
- DMemory[ALUOutput] <-- A

**WB: Write back**
- nop

---

**Branch on equal**

beq: PC <-- Ra == 0 ? PC + 4 + disp*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 <-- IMemory[PC]
- incrPC <-- PC + 4

**ID: Instruction decode/register fetch**
- A <-- Register[IR[25:21]]

**Ex: Execute**
- Target <-- incrPC + SignExtend(IR[20:0]) << 2
- Z <-- (A == 0)

**MEM: Memory**
- PC <-- Z ? Target : incrPC

**WB: Write back**
- nop

---

**Active Datapath for Branch and BSR**

**ALU Computes target**
- A = shifted, extended IR[20:0]
- B = IncrPC
- Function set to add

**Zero Test**
- Determines branch condition

**PC Selection**
- Target for taken branch
- IncrPC for not taken

**Write Back**
- Only for bsr and br
- Incremented PC as data

---

**Branch to Subroutine**

Branch Subroutine (bsr): Ra <-- PC + 4; PC <-- PC + 4 + disp*4

<table>
<thead>
<tr>
<th>0x34</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 <-- IMemory[PC]
- IncrPC <-- PC + 4

**ID: Instruction decode/register fetch**
- nop

**Ex: Execute**
- Target <-- incrPC + SignExtend(IR[20:0]) << 2

**MEM: Memory**
- PC <-- Target

**WB: Write back**
- Register[IR[25:21]] <-- incrPC
Jump
jmp, jsr, ret: Ra ← PC+4; PC ← Rb

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

**IF: Instruction fetch**
- IR ← IMemory[PC]
- incrPC ← PC + 4

**ID: Instruction decode/register fetch**
- B ← Register[IR[20:16]]

**Ex: Execute**
- Target ← B

**MEM: Memory**
- PC ← target

**WB: Write back**
- IR[25:21] ← incrPC

---

**Active Datapath for Jumps**

**ALU Operation**
- Used to compute target
  - B input set to Rb
  - ALU function set to select B

**Write Back**
- To Ra
- IncrPC as data

---

**Complete Datapath**

**Pipelining Basics**

**Unpipelined System**
- 30ns
- Delay = 33ns
- Throughput = 30MHz

**Comb. Logic**
- 3ns
- Clock

**Time**
- Op1 → Op2 → Op3 → ...

**Notes:**
- One operation must complete before next can begin
- Operations spaced 33ns apart
3 Stage Pipelining

- Space operations 13ns apart
- 3 operations occur simultaneously

Delay = 39ns
Throughput = 77MHz

Limitation: Non-uniform Pipelining

- Throughput limited by slowest stage
  - Delay determined by clock period * number of stages
- Must attempt to balance stages

Delay = 18 * 3 = 54 ns
Throughput = 55MHz

Limitation: Deep Pipelines

- Diminishing returns as add more pipeline stages
- Register delays become limiting factor
  - Increased latency
  - Small throughput gains

Delay = 48ns, Throughput = 128MHz

Limitation: Sequential Dependencies

- Op4 gets result from Op1!
- Pipeline Hazard
Pipelined datapath

Pipe Registers
- Inserted between stages
- Labeled by preceding & following stage

Pipe Register

Operation
- Current State stays constant while Next State being updated
- Update involves transferring Next State to Current

Pipeline Stage

Operation
- Computes next state based on current
- From/to one or more pipe registers
- May have embedded memory elements
- Low level timing signals control their operation during clock cycle
- Writes based on current pipe register state
- Reads supply values for Next state

Notes
- Each stage consists of operate logic connecting pipe registers
- WB logic merged into ID
- Additional paths required for forwarding
Alpha Simulator

Features
• Based on Alpha subset
  - Code generated by dis
  - Hexadecimal instruction code
• Executable available soon
  - AFS740/sim/solve_tk

Demo Programs
• AFS740/sim/solve_tk/demos

Program Display

Run Controls
• Speed Control
• Mode Selection

Current State
• Pipe Register
• Next State

Register Values
Hex-coded instruction
Pipe Stage
Treated as comment

SIMULATOR ALU Example

IF
• Fetch instruction
ID
• Fetch operands
EX
• Compute ALU result
MEM
• Nothing
WB
• Store result in Rc

# Demonstration of R-R instruction
.set no reorder
mov 3, $2
demo01.O
.mov 4, $3	nop
.addq $2, $3, $4	nop
call_pal 0x0
.set reorder
demo01.s

SIMULATOR STORE/LOAD EXAMPLES

IF
• Fetch instruction
ID
• Get addr reg
• Store: Get data
EX
• Compute EA
MEM
• Load: Read
• Store: Write
WB
• Load: Update reg.

# Demonstration of R-I instruction
.demo02.O

0x0: 43e17402 addq r31, 0xb, r2 # $2 = 0xB
0x4: 43e19403 addq r31, 0xc, r3 # $3 = 0xC
0x8: 43fff404 addq r31, 0xff, r4 # $4 = 0xFF
0xc: 47ff041f bis r31, r31, r31
0x10:47ff041f bis r31, r31, r31
0x14:47ff041f bis r31, r31, r31
0x18:47ff041f bis r31, r31, r31
0x1c:47ff041f bis r31, r31, r31
0x20:47ff041f bis r31, r31, r31
0x24:47ff041f bis r31, r31, r31
0x28:00000000 call_pal halt

SIMULATOR BRANCH EXAMPLES

IF
• Fetch instruction
ID
• Fetch operands
EX
• test if operand = 0
• Compute target
MEM
• Taken: Update PC to target
WB
• Nothing

# Demonstration of B-R instruction
.demo3.O

0x0: 43e07402 addq r31, 0x3, r2 # $2 = 3
0x4: 43e09403 addq r31, 0x4, r3 # $3 = 4
0x8: 47ff041f bis r31, r31, r31
demo01.O
0xc: 47ff041f bis r31, r31, r31
0x10:40430404 addq r2, r3, r4 # $4 = 7
0x14:47ff041f bis r31, r31, r31
0x18:00000000 call_pal halt

0x4: 47ff041f bis r31, r31, r31
0x8: 47ff041f bis r31, r31, r31
0xc: e4400008 beq r2, 0x30 # Don't take
test if operand = 0
Compute target
Taken: Update PC to target
set reorder
demo01.s
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
```

Time
- $2$ written

Control Hazards in Alpha Pipeline

Problem
- Instruction fetched in IF, branch condition set in MEM
- When does branch take effect?
- E.g.: assume initially that all registers = 0

```
beq $0, target
mov 63, $2
mov 63, $3
mov 63, $4
mov 63, $5
mov 63, $6
```

PC Updated
- Target: mov 63, $6

Branch Example

```
Branch Code (demo08.O)
0x0: e7e00005 beq r31, 0x18  # Take
0x4: 43e7f401 addq r31, 0x3f, r1  # (Skip) $1 = 0x3F
0x8: 43e7f402 addq r31, 0x3f, r2  # (Skip) $2 = 0x3F
0xc: 43e7f403 addq r31, 0x3f, r3  # (Skip) $3 = 0x3F
0x10: 43e7f404 addq r31, 0x3f, r4  # (Skip) $4 = 0x3F
0x14: 47ff041f bis r31, r31, r31
0x18: 00000000 call_pal  halt
```

Simulator Data Hazard Example

```
Operation
demo04.O
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
```

Branch Example

```
Branch Code (demo08.O)
0x0: e7e00005 beq r31, 0x18  # Take
0x4: 43e7f401 addq r31, 0x3f, r1  # (Skip) $1 = 0x3F
0x8: 43e7f402 addq r31, 0x3f, r2  # (Skip) $2 = 0x3F
0xc: 43e7f403 addq r31, 0x3f, r3  # (Skip) $3 = 0x3F
0x10: 43e7f404 addq r31, 0x3f, r4  # (Skip) $4 = 0x3F
0x14: 47ff041f bis r31, r31, r31
0x18: 00000000 call_pal  halt
```
Conclusions, So Far

RISC Design Simplifies Implementation
- Small number of instruction formats
- Simple instruction processing

RISC Leads Naturally to Pipelined Implementation
- Partition activities into stages
- Each stage simple computation

We’re not done yet!
- Need to deal with data & control hazards

Pipelined ALU Instruction Datapath

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:

```
<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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]

Detecting Dependencies

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

Implementing Stalls

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

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

Example:
- addq $31, 63, $2
- addq $2, 0, $3
- addq $2, 0, $4
- addq $2, 0, $5
- addq $2, 0, $6
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

**EX-EX**
- 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

IF
ID
EX
M
WB

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

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

Some Hazards with Loads & Stores

Data Generated by Load

<table>
<thead>
<tr>
<th>Load-Store Data</th>
<th>Load-ALU</th>
<th>Load-Store (or Load) Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldq $1, 8($2)</td>
<td>ldq $1, 8($2)</td>
<td>ldq $1, 8($2)</td>
</tr>
<tr>
<td>stq $1, 16($2)</td>
<td>addq $2, $1, $2</td>
<td>stq $2, 16($1)</td>
</tr>
</tbody>
</table>

Data Generated by Store

<table>
<thead>
<tr>
<th>Store-Load Data</th>
<th>ALU-Store Data</th>
<th>ALU-Store (or Load) Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>stq $1, 8($2)</td>
<td>addq $1, $3, $2</td>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>stq $3, 8($2)</td>
<td>stq $3, 8($2)</td>
<td>stq $1, 16($2)</td>
</tr>
</tbody>
</table>

Not a concern for us

Data Generated by ALU
**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**
- `ldq $1, 8($2)`
- `stq $1, 16($2)`

**Complete Bypassing for ALU & L/S**

**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`

With 1 Cycle Stall
- `ldq $1, 8($2)`
- `addq $2, $1, $2`

**Simulator Data Hazard Examples**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Registers</th>
<th>MEM-MEM Forwarding Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td><code>addq r31, 0x3f, r2</code></td>
<td>r31, r2</td>
<td>$2 = 0x3F</td>
</tr>
<tr>
<td>0x4</td>
<td><code>bis r2, r2, r3</code></td>
<td>r2, r3, r3</td>
<td>$3 = 0x3F EX-EX</td>
</tr>
<tr>
<td>0x8</td>
<td><code>addq r31, 0xff, r2</code></td>
<td>r31, r2</td>
<td>$2 = 0xF</td>
</tr>
<tr>
<td>0x10</td>
<td><code>addq r31, 0x8, r3</code></td>
<td>r31, r3</td>
<td>$3 = 8</td>
</tr>
<tr>
<td>0x14</td>
<td><code>addq r31, 0x10, r2</code></td>
<td>r31, r2</td>
<td>$2 = 0x10</td>
</tr>
<tr>
<td>0x18</td>
<td><code>addq r31, 0xf, r2</code></td>
<td>r31, r2</td>
<td>$3 = 0xF MEM-EX</td>
</tr>
<tr>
<td>0x1c</td>
<td><code>addq r31, 0x8, r3</code></td>
<td>r31, r3</td>
<td>$3 = 8</td>
</tr>
<tr>
<td>0x20</td>
<td><code>addq r31, 0x10, r2</code></td>
<td>r31, r2</td>
<td>$2 = 0x10</td>
</tr>
<tr>
<td>0x24</td>
<td><code>addq r31, 0xf, r2</code></td>
<td>r31, r2</td>
<td>$3 = 0xF MEM-EX</td>
</tr>
<tr>
<td>0x28</td>
<td><code>addq r31, 0x8, r3</code></td>
<td>r31, r3</td>
<td>$3 = 8</td>
</tr>
<tr>
<td>0x30</td>
<td><code>addq r31, 0x10, r2</code></td>
<td>r31, r2</td>
<td>$2 = 0x10</td>
</tr>
<tr>
<td>0x34</td>
<td><code>addq r31, 0xf, r2</code></td>
<td>r31, r2</td>
<td>$3 = 0xF MEM-EX</td>
</tr>
<tr>
<td>0x38</td>
<td><code>addq r31, 0x8, r3</code></td>
<td>r31, r3</td>
<td>$3 = 8</td>
</tr>
<tr>
<td>0x3c</td>
<td><code>addq r31, 0x10, r2</code></td>
<td>r31, r2</td>
<td>$2 = 0x10</td>
</tr>
<tr>
<td>0x40</td>
<td><code>addq r31, 0xf, r2</code></td>
<td>r31, r2</td>
<td>$3 = 0xF MEM-EX</td>
</tr>
</tbody>
</table>

Simulator Data Hazard Examples:
- `call_pal halt`
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, $2, $3  RR.rc/RR.rb/1  addu $2, $31, $4

distance 1 hazard:
addq $31, 63, $2

Enumerating data hazards

<table>
<thead>
<tr>
<th>writes</th>
<th>distance = 1</th>
<th>reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR Ra</td>
<td></td>
<td>RR.rb</td>
</tr>
<tr>
<td>RI Ra</td>
<td></td>
<td>LRa</td>
</tr>
<tr>
<td>L Ra</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>writes</th>
<th>distance = 2</th>
<th>reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR Ra</td>
<td></td>
<td>RR.rb</td>
</tr>
<tr>
<td>RR Ra</td>
<td></td>
<td>LRa</td>
</tr>
<tr>
<td>L Ra</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

| 0x0: 43e21402 | addq r31, 0x10, r2 | r2 = 0x10  
| 0x4: 47ff041f | bis r31, r31, r31 |
| 0x8: 47ff041f | bis r31, r31, r31 |
| 0xc: 43e20405 | addq r31, r2, r5 | # r5 = 0x10  
| 0x10: 43e50401 | addq r31, r5, r1 | # r1 = 0x10 |
| 0x14: 47ff041f | bis r31, r31, r31 |
| 0x18: 47ff041f | bis r31, r31, r31 |
| 0x1c: 44221803 | xor r1, 0x10, r3 | # r3 should == 0  
| 0x20: 44221803 | xor r1, 0x10, r3 |
| 0x24: 47ff041f | bis r31, r31, r31 |
| 0x28: 47ff041f | bis r31, r31, r31 |
| 0x2c: 44600006 | beq r3, 0x40 | # 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 |

Branch Instructions

Cond. Branch: PC <-- Cond(Ra) ? 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>

Sources
- PC, Ra

Destinations
- PC

Branch [Subroutine] (br, bsr): Ra <-- PC + 4; PC <-- PC + 4 + disp*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>

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

<table>
<thead>
<tr>
<th>ALU-Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>beq $1, targ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distant ALU-Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>bis $31, $31, $31</td>
</tr>
<tr>
<td>beq $1, targ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load-Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 8($2)</td>
</tr>
<tr>
<td>beq $1, targ</td>
</tr>
</tbody>
</table>

Jump Instructions

<table>
<thead>
<tr>
<th>jmp, jsr, ret: Ra &lt;- PC+4; PC &lt;- Rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources</td>
</tr>
<tr>
<td>- PC, Rb</td>
</tr>
<tr>
<td>Destinations</td>
</tr>
<tr>
<td>- PC, Ra</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sources</th>
<th>Destinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Ra</td>
</tr>
</tbody>
</table>

Jump Instructions

<table>
<thead>
<tr>
<th>Sources</th>
<th>Destinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Ra</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ALU-Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>jsr $26 ($1), 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distant ALU-Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>bis $31, $31, $31</td>
</tr>
<tr>
<td>jmp $31 ($1), 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load-Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $26, 8($sp)</td>
</tr>
<tr>
<td>ret $31 ($26), 1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ALU-Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>jsr $26 ($1), 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distant ALU-Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>bis $31, $31, $31</td>
</tr>
<tr>
<td>jmp $31 ($1), 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load-Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $26, 8($sp)</td>
</tr>
<tr>
<td>ret $31 ($26), 1</td>
</tr>
</tbody>
</table>

Enumerating data hazards

<table>
<thead>
<tr>
<th>BEQ, BNE</th>
<th>Jmp, JSR, Ret</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
### Pipelined datapath

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>instruction decode/register fetch</td>
<td>execute/address calc</td>
<td>memory access</td>
<td>write back</td>
</tr>
</tbody>
</table>

What happens with a branch?

### 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>

### Branch on equal

- **beq: PC <-- Ra == 0 ? PC + 4 + disp*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 <-- IMemory[PC]
- incrPC <-- PC + 4

**ID: Instruction decode/register fetch**
- A <-- Register[IR[25:21]]

**Ex: Execute**
- Target <-- incrPC + SignExtend(IR[20:0]) << 2
- Z <-- (A == 0)

**MEM: Memory**
- PC <-- 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

**Brach Code (demo08.O)**

```assembly
0x0: e7e00005 beq r31, 0x18 # Take
0x4: 43e7f401 addq r31, 0x3f, r1 # (Skip)
0x8: 43e7f402 addq r31, 0x3f, r2 # (Skip)
0xc: 43e7f403 addq r31, 0x3f, r3 # (Skip)
0x10: 43e7f404 addq r31, 0x3f, r4 # (Skip)
0x14: 47ff041f bis r31, r31, r31
0x18: 43e7f405 addq r31, 0x3f, r5 # (Target)
0x1c: 47ff041f bis r31, r31, r31
0x20: 00000000 call_pal halt
```
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: `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

- One cycle later
  - Problem: Will execute 3 extra instructions!

Branch Hazard Pipeline Diagram

- Problem
  - Instruction fetched in IF, branch condition set in MEM

Stall Until Resolve Branch

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

Stall Control
Stalling 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 stalled twice
  -Injects two 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

Taken 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 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

```plaintext
0x0: beq 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

- Branch not taken
- Allow instructions to proceed

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 * (0.67 * 3 + 0.33 * 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

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
  
<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>$31, target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addq $31, 63, $1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addq $31, 63, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addq $31, 63, $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addq $31, 63, $4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>target: addq $31, 63, $5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PC Updated
Non-canceling Branch Pipeline Diagram

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

```
IF ID EX M WB
IF ID EX M WB
IF ID EX M WB
IF ID EX M WB

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)

User Process    Operating System

```
A  ->  exception -> (optional)
C  ->  handler  
```

User Process    Operating System

```
A  ->  event      ->  exception
|       |               | handler
C  ->  exception  | (optional)
```  

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 ctrl-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"

User Process

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 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

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

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 Process

Overflow detected here
the xor instruction completes
flush these instructions
start handler code

Precise vs. Imprecise Exceptions
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)**

0x0: addq r31, 0x3, r3
0x4: stq r3, -4(r31) # bad address
0x8: sll r3, 0x8, r5 # unimplemented
0xc: addq r31, 0xf, r2

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

0x0: addq r31, 0x3, r3
0x4: sll r3, 0x8, r5 # unimplemented
0x8: stq r3, -4(r31) # bad address
0xc: addq r31, 0xf, r2

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

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</th>
</tr>
</thead>
<tbody>
<tr>
<td>add / sub</td>
<td>14%</td>
<td>11%</td>
</tr>
<tr>
<td>multiply</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>divide</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
</tr>
</tbody>
</table>

Pipeline Revisited

Multiply Timing Example

```plaintext
bis $31, 3, $2
bis $31, 7, $3
mulq $2, $3, $4
addq $2, $3, $3  \(\text{IF ID EX M WB}\)
addq $2, $4, $2
```

Stall while Busy

(Not to scale)
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

FP Timing Example

```
addt $f2, $f4, $f1
```

```
divt $f1, $f2, $f6
```
RAW Stall

```
addt $f1, $f2, $f8
```
Structural Stall

```
addt $f1, $f2, $f6
```
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 idiosyncrasies of pipeline structure