Advanced Pipelining
CS740
Sept. 22, 2000

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): rc ← 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>

RX-type instructions (addq, subq, xor, bis, cmplt): rc ← 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
• ib is 8-bit unsigned literal

Operation               Op field      funct field
addq                   0x10          0x20
subq                   0x10          0x29
bis                    0x11          0x20
xor                    0x11          0x40
cmovwq                 0x11          0x24
cmplt                  0x11          0x40

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:

$2$
$3$
$4$
$5$
$6$
$2$ written

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

```assembly
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
```

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

```assembly
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
```

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

**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

```assembly
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]
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

IF
ID
EX
M
WB

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

$2 written

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
### Bypassing Possibilities

**EX-EX**
- From instruction that just finished EX

**MEM-EX**
- From instruction that finished EX two cycles earlier

### 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

### 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>Time</th>
<th>$2$ written</th>
</tr>
</thead>
</table>

### 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

**Load-Store Data**
- ldq $1, 8($2)
- stq $1, 16($2)

**Load-ALU**
- ldq $1, 8($2)
- addq $2, $1, $2

**Load-Store (or Load) Addr.**
- ldq $1, 8($2)
- stq $2, 16($2)

#### Data Generated by Store

**Store-Load Data**
- stq $1, 8($2)
- ldq $3, 8($2)

**ALU-Store (or Load) Addr**
- addq $1, $3, $2
- stq $3, 16($2)

**ALU-Store Data**
- addq $2, $3, $1
- stq $1, 16($2)

Not a concern for us

---

### MEM-MEM Forwarding

**Condition**
- Data generated by load instruction
  - Register WQ_in.WDest
- Used by immediately following store
  - Register MEM_in.ASrc

**Load-Store Data**
- ldq $1, 8($2)
- stq $1, 16($2)

**Time**

---

### Complete Bypassing for ALU & L/S

---

### Impact of Forwarding

#### Single Remaining Unsolved Hazard Class
- Load followed by ALU operation
  - Including address calculation

**Just Forward?**
- ldq $1, 8($2)
- addq $2, $1, $2

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

Value not available soon enough!

Then can use MEM-EX forwarding
Simulator Data Hazard Examples

- demo5.O

<table>
<thead>
<tr>
<th>OP</th>
<th>writes</th>
<th>reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>rc</td>
<td>ra, rb</td>
</tr>
<tr>
<td>RI</td>
<td>rc</td>
<td>ra</td>
</tr>
<tr>
<td>Load</td>
<td>ra</td>
<td>rb</td>
</tr>
<tr>
<td>Store</td>
<td>ra</td>
<td>rb</td>
</tr>
</tbody>
</table>

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, RR.ra, 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 1 hazard: RR.rc/RR.ra/1
  - distance 2 hazard: RR.rc/RR.rb/2

Enumerating data hazards

<table>
<thead>
<tr>
<th>reads</th>
<th>distance = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>writes</td>
<td>RR.rc</td>
</tr>
<tr>
<td></td>
<td>RR.rb</td>
</tr>
<tr>
<td>S.rb</td>
<td>S.rb</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reads</th>
<th>distance = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>writes</td>
<td>RR.rc</td>
</tr>
<tr>
<td></td>
<td>RR.rb</td>
</tr>
<tr>
<td>S.rb</td>
<td>S.rb</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

demo7.O

- Tests for single failure mode
  - ALU Rc -> ALU Ro
  - distance 1
  - RR.rc/RR.rb/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
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

ALU-Branch
addq $2, $3, $1
beq $1, targ

Distant ALU-Branch
addq $2, $3, $1
bis $31, $31, $31
beq $1, targ

Load-Branch
lw $1, 8($2)
beq $1, 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

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

Pipelined datapath

Branch on equal

Conditional Branch Instruction Handling
Branch Example

Desired Behavior
- Take branch at 0x00
- Execute target 0x18
  - PC = 0x00
  - disp = 5

Branch Code (demo08.O)

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)
0x20: 00000000 call_pal halt

Displacement

Branch Hazard Example

With BEQ in Mem stage

IF/ID
EX/MEM
INSTR
ALU
XTRAXTRAXTRAXTRA
BEQ
0X10
PC Updated

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

IF/ID
EX/MEM
INSTR
ALU
XTRAXTRAXTRA
BEQ
0X10
PC Updated

Target: addq $31, 63, $5
beq $31, target
addq $31, 63, $1
addq $31, 63, $2
addq $31, 63, $3
addq $31, 63, $4

Time →
Stall Until Resolve Branch

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

Stall Control

Perform when branch in either stage

Stalling Branch Example

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

Stall Bubble Transfer Transfer Transfer

Perform when branch in either stage

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

- 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

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

**Not Taken Branch Resolution Example**

```
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
```

- Branch not taken
- Allow instructions to proceed

**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 * (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

IF ID EX M WB
beq $31, target
addq $31, 63, $1
addq $31, 63, $2
addq $31, 63, $3
addq $31, 63, $4

PC Updated
Time

Noncanceling 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
bne $31, target
addq $31, 63, $1
addq $31, 63, $2
addq $31, 63, $3
addq $31, 63, $4

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

User Process

Operating System

event exception

exception

processing exception

by exception handler

return (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)

- 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 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 Process

```
A
C
?
```

Handler code

```
addq $26, 100($31)
...```

Example: Integer Overflow

```
user code
mov $12, $2, $5
mov $13, $2, $6
addq $1, $2, $1
or $15, $6, $7
ldq $16, 50($7)
stq $26, 100($31)
...
```

Overflow detected here

```
flush these
instructions
start handler
code```

Overflow detected here

```
flush these
instructions
start handler
code```

```
**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**

- **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: `stq r3, -4(r31)` # bad address
  - 0x8: `sll r3, 0x8, r5` # unimplemented
  - 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`

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</th>
<th>FP Benchnarks</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

bis $31, 3, $2
bis $31, 7, $3
mulq $2, $3, $4
addq $2, $3, $3
bis $4, $31, $5
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
    - Avoids RAW hazard
      - divt $f1, $f2, $f6
      - add $f1, $f2, $f4
      - add $f1, $f2, $f4

Bypass paths
- Similar to those in integer pipeline

FP Timing Example

addt $f2, $f4, $f1
divt $f1, $f2, $f6
addt $f1, $f2, $f8
addt $f1, $f2, $f6

raw: $f1, $f4, $f6
RAW Stall

struct: $f1, $f2, $f8
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 idiosyncrasies of pipeline structure