Advanced Pipelining
15-740/18-740

October 28, 2009

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): \( rc \leftarrow ra \text{ funct } rb \)

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

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>ib</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>
</tr>
</tbody>
</table>

Encoding
  • ib is 8-bit unsigned literal

Operation Op field funct field

<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>0x40</td>
</tr>
</tbody>
</table>

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:

$2 \rightarrow IF \rightarrow ID \rightarrow EX \rightarrow MEM \rightarrow WB$

\[
\begin{align*}
\text{addq } &\text{ $31, 63, } \text{ $2} \\
\text{subq } &\text{ $2, 0, } \text{ $3} \\
\text{bis } &\text{ $2, 0, } \text{ $4} \\
\text{xor } &\text{ $2, 0, } \text{ $5} \\
\text{cmoveq } &\text{ $2, 0, } \text{ $6}
\end{align*}
\]

$\text{\$2 written}$
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 RS

**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

```
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-WB forwarding

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2$</td>
<td></td>
<td></td>
<td>$2$</td>
<td></td>
</tr>
<tr>
<td>$3$</td>
<td></td>
<td></td>
<td></td>
<td>$3$</td>
</tr>
<tr>
<td>$4$</td>
<td></td>
<td></td>
<td>$4$</td>
<td></td>
</tr>
<tr>
<td>$5$</td>
<td></td>
<td></td>
<td></td>
<td>$5$</td>
</tr>
<tr>
<td>$6$</td>
<td></td>
<td></td>
<td>$6$</td>
<td></td>
</tr>
</tbody>
</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

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
  - Available after MEM
  - Read Data
  - ALU Data
- Reg-Reg Result
- Load result
- 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>Data Generated by Store</th>
</tr>
</thead>
</table>
| 1dq $1, 8($2)  | Store-Load Data
| 1stq $1, 16($2)| 1dq $3, 8($2) |

Data Generated by ALU

<table>
<thead>
<tr>
<th>Load-ALU</th>
<th>Data Generated by ALU (Load) Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1dq $1, 8($2)</td>
<td>addq $1, $3, $2</td>
</tr>
<tr>
<td>addq $2, $1, $2</td>
<td></td>
</tr>
</tbody>
</table>

Data Generated by ALU (Load) Addr

<table>
<thead>
<tr>
<th>Load-Store (or Load) Addr</th>
<th>Data Generated by ALU (Load) Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1dq $1, 8($2)</td>
<td>addq $2, $3, $1</td>
</tr>
<tr>
<td>1stq $2, 16($1)</td>
<td>addq $2, 0, $6</td>
</tr>
</tbody>
</table>
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
- \texttt{ldq $1, 8($2)}
- \texttt{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

Just Forward?
- \texttt{ldq $1, 8($2)}
- \texttt{addq $2, $1, $2}

With 1 Cycle Stall
- \texttt{ldq $1, 8($2)}
- \texttt{addq $2, $1, $2}

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:
  - RR.rc/RR.rb/2
  - addq $31, 63, $2

- Distance 1:
  - RR.rc/RR.rb/1
  - addq $31, 63, $2
Enumerating data hazards

distance = 1

<table>
<thead>
<tr>
<th>reads</th>
<th>distance = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>writes</td>
<td>RR ra, RR rb, RR rc, RI ra, L rb, S ra, 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 ra, RR rb, RR rc, RI ra, L rb, S ra, 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

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 $3l, $3l, $3l |
| 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>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-16</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

<table>
<thead>
<tr>
<th>ALU-Jump</th>
<th>Distant ALU-Jump</th>
<th>Load-Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $2, $3, $1</td>
<td>addq $2, $3, $1</td>
<td>lw $26, 8($sp)</td>
</tr>
<tr>
<td>jsr $26 ($1), 1</td>
<td>bis $31, $31, $31</td>
<td>ret $31 ($26), 1</td>
</tr>
</tbody>
</table>

Enumerating data hazards

reads: BEQ, BNE

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

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>

Pipelined datapath

IF instruction fetch
ID instruction decode/ register fetch
EX execute/ address calc
MEM memory access
WB write back

Conditional Branch Instruction Handling

<table>
<thead>
<tr>
<th>Zero IF/ID</th>
<th>ID/EX</th>
<th>EX/MEM</th>
</tr>
</thead>
</table>

What happens with a branch?
Branch on equal

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 = 0x00
- disp = 5

Branch Code (demo08.O)

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>beq r31, 0x18</td>
</tr>
<tr>
<td>0x4</td>
<td>addq r31, 0x3f, r1</td>
</tr>
<tr>
<td>0x8</td>
<td>addq r31, 0x3f, r2</td>
</tr>
<tr>
<td>0xc</td>
<td>addq r31, 0x3f, r3</td>
</tr>
<tr>
<td>0x10</td>
<td>addq r31, 0x3f, r4</td>
</tr>
<tr>
<td>0x18</td>
<td>addq r31, 0x3f, r5</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

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 | M | WB
---|----|----|---|---
beq | $31, target
addq | $31, 63, $1
addq | $31, 63, $2
addq | $31, 63, $3
addq | $31, 63, $4

PC Updated
Time

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

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

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

```assembly
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**

```assembly
deemo09.O
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

<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</td>
<td>$31, 63, 1$ # Xtra1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addq</td>
<td>$31, 63, 2$ # Xtra2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addq</td>
<td>$31, 63, 3$ # Xtra3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addq</td>
<td>$31, 63, 4$ # Xtra4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PC Not Updated

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

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

Canceling Branch Example
- 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

Noncanceling Branch Pipeline Diagram

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

Branch Prediction Analysis

Our Scheme Implements “Predict Not Taken”
- But 67% of branches are taken
- Impact on CPI: 0.16 * 0.67 * 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

event -> exception -> exception processing by exception handler

exception 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 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
- this clean break is called a “precise exception”

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
- 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.)

<table>
<thead>
<tr>
<th>User code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and $12, $2, $5</code></td>
</tr>
<tr>
<td><code>xor $13, $2, $6</code></td>
</tr>
<tr>
<td><code>addq $1, $2, $1=</code></td>
</tr>
<tr>
<td><code>or $15, $6, $7</code></td>
</tr>
<tr>
<td><code>ldq $16, 50($7)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Handler code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stq $26, 100($31)</code></td>
</tr>
</tbody>
</table>

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

Pipeline Revisited

Multiply Timing Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>EX</th>
<th>ID</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis $31, 3, $2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>bis $31, 7, $3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>mulq $2, $3, $4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>addq $2, $3, $3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>bis $4, $31, $5</td>
<td>IF</td>
<td>ID</td>
<td>...</td>
<td>EX</td>
</tr>
<tr>
<td>addq $2, $4, $2</td>
<td>IF</td>
<td>...</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Stall while Busy
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