Basic Pipelining

October 24, 2014

Topics

- Objective
- Instruction formats
- Instruction processing
- Principles of pipelining
- Inserting pipe registers

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

Refined Design
- 5-stage pipeline
  - Similar to early RISC processors
- Goal: approach 1 cycle per instruction but with shorter cycle time

Alpha Arithmetic 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
---|---|---
addq | 0x10 | 0x20
subq | 0x10 | 0x29
bis | 0x11 | 0x20
xor | 0x11 | 0x40
cmoveq | 0x11 | 0x24
cmplt | 0x11 | 0x4D

Alpha Load/Store Instructions

Load: \[ Ra \leftarrow \text{Mem}[Rb + \text{offset}] \]
Store: \[ \text{Mem}[Rb + \text{offset}] \leftarrow Ra \]

<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
- \( \text{offset} \) is 16-bit signed offset

Operation | Op field
---|---
ldq | 0x29
stq | 0x2D
### 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>

**Encoding**
- disp is 21-bit signed displacement

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

*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>

**Operation Op field**
- br 0x30
- bsr 0x34

### Transfers of Control

**Jump, 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>

**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

### Instruction Encoding

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

<table>
<thead>
<tr>
<th>0x0: 40220403</th>
<th>addq r1, r2, r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4: 4487f805</td>
<td>xor r4, 0x3f, r5</td>
</tr>
<tr>
<td>0x8: a4c70abc</td>
<td>ldq r6, 2748(r7)</td>
</tr>
<tr>
<td>0x8: b5090123</td>
<td>stq r8, 291(r9)</td>
</tr>
<tr>
<td>0x10: e47ffffb</td>
<td>beq r3, 0</td>
</tr>
<tr>
<td>0x14: d35ffffa</td>
<td>bsr r26, 0(r31)</td>
</tr>
<tr>
<td>0x18: 6bfa8001</td>
<td>ret r31, (r26), 1</td>
</tr>
</tbody>
</table>

### Decoding Examples

0x0: 40220403 addq r1, r2, r3

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

0x8: a4c70abc ldq r6, 2748(r7)

- Target = 2748_{10}

0x10: e47ffffb beq r3, 0

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

- Target = 31_{10} = 26_{16}
**Datapath**

**IF** instruction fetch
- IR $\leftarrow$ IMemory[PC]
- $PC \leftarrow PC + 4$

**ID** instruction decode/register fetch
- A $\leftarrow$ Register[IR[25:21]]
- B $\leftarrow$ Register[IR[20:16]]

**EX** execute/address calculation
- ALUOutput $\leftarrow$ A op B

**MEM** memory access
- nop

**WB** write back
- Register[IR[4:0]] $\leftarrow$ ALUOutput

---

**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): rc $\leftarrow$ ra 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>11-5</td>
</tr>
</tbody>
</table>

**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

---
RI-type instructions

<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>

**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

---

**Load instruction**

<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**
- 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

---

**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

**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

**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

**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 <-- B

**WB: Write back**
- Register[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
- Delay = 1.3 ns
- Throughput = 0.77 GHz
- One operation must complete before next can begin
- Operations spaced 1.3 ns apart

3 Stage Pipelining
- Delay = 1.5 ns
- Throughput = 2.0 GHz
- Space operations 0.5 ns apart
- 3 operations occur simultaneously
**Limitation: Nonuniform Pipelining**

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

```
Delay = 0.7 * 3 = 2.1 ns
Throughput = 1.43 GHz
```

**Limitation: Deep Pipelines**

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

```
Delay = 1.8ns
Throughput = 3.33GHz
```

**Limitation: Sequential Dependencies**

- Op4 gets result from Op1!
  - Pipeline Hazard

**Pipelined Datapath**

- Inserted between stages
- Labeled by preceding & following stage

Pipe Registers:

- Inserted between stages
- Labeled by preceding & following stage
**Pipeline Structure**

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

**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

**Data Hazards in Alpha Pipeline**

Problem
- Registers read in ID, and written in WB
- Must resolve conflict between instructions competing for registers
  - Generally do writeback 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
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

```
IF ID EX MEM WB
beq $0, target
```
```
IF ID EX MEM WB
mov 63, $2
```
```
IF ID EX MEM WB
mov 63, $3
```
```
IF ID EX MEM WB
mov 63, $4
```
```
IF ID EX MEM WB
mov 63, $5
```

PC Updated

```
target: mov 63, $6
```

Handling Hazards by Stalling

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

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

Detecting Dependencies

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

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

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

```
    IF ID EX M WB
  addq $31, 63, $2
    IF ID EX M WB
  addq $2, 0, $3
     IF ID EX M WB
  addq $2, 0, $4
        IF ID EX M WB
  addq $2, 0, $5
            IF ID EX M WB
  addq $2, 0, $6
```

- $2 written

--- Time ---

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

---

**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>Store-Load Data</th>
<th>Not a concern for us</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldq $1, 8($2)</td>
<td>stq $1, 16($2)</td>
<td></td>
</tr>
<tr>
<td>stq $1, 16($2)</td>
<td>ldq $3, 8($2)</td>
<td></td>
</tr>
<tr>
<td>Load-ALU</td>
<td>ALU-Store (or Load) Addr</td>
<td></td>
</tr>
<tr>
<td>ldq $1, 8($2)</td>
<td>addq $2, $1, $2</td>
<td></td>
</tr>
<tr>
<td>addq $2, $1, $2</td>
<td>stq $2, 16($1)</td>
<td></td>
</tr>
</tbody>
</table>

Data Generated by ALU

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

Complete Bypassing for ALU & L/S

Some Hazards with Loads & Stores

Data Generated by Load

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

Data Generated by Store

<table>
<thead>
<tr>
<th>Store-Load Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>stq $1, 8($2)</td>
</tr>
<tr>
<td>ldq $3, 8($2)</td>
</tr>
</tbody>
</table>

ALU-Store Data

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

Data Generated by ALU

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

Load-ALU

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

Load-Store (or Load) Addr

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

ALU-Store Data

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

ALU-Store (or Load) Addr

| ldq $1, 8($2) |
| addq $1, $3, $2 |

Load-Store Data

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

ALU-Store Data

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

ALU-Store (or Load) Addr

| ldq $1, 8($2) |
| addq $1, $3, $2 |
Impact of Forwarding

Single Remaining Unsolved Hazard Class

- Load followed by ALU operation
  - Including address calculation

```
Load-ALU
idq $1, 8($2)
addq $2, $1, $2
```

**Just Forward?**

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

**With 1 Cycle Stall**

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

```
IF ID ID EX M WB
addq $2, $1, $2
```

Value not available soon enough!

```
Time
```

Then can use MEM-EX forwarding

```
```

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
```

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
```

Pipelined datapath
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>

Desired Behavior
- Take branch at 0x00
- Execute target 0x18
  - PC + 4 + disp << 2
  - PC = 0x00
  - disp = 5

Branch Example

```
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, r3, r3  
0x18: 43e7f405  addq  r31, 0x3f, r5  # (Target)
0x1c: 47ff041f  bis  r31, r3, r3  
0x20: 00000000  call_pal  halt
```

Branch Hazard Example

```
Branch Code (demo08.O)
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 Example (cont.)

```
Branch Code (demo08.O)
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

```
beq $31, target
addq $31, 63, $1
addq $31, 63, $2
addq $31, 63, $3
addq $31, 63, $4
```

target: addq $31, 63, $5

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

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

```
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
```

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

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

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
Behavior

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

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

Analysis of Stalling

Branch Instruction Timing

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

Performance Impact

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

Fetch & Cancel When Taken

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

Strategy

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

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
```

Canceling Branch Example

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

- When branch taken
- Generate 3 bubbles
- Begin fetching at target

```
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
```

Canceling Branch Pipeline Diagram

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

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

Noncanceling Branch Pipeline Diagram

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

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

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 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”

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

Precise vs. Imprecise Exceptions

In the Alpha architecture:

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

Imprecise exceptions:

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

What if precise exceptions are needed?

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

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

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

**User code**

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

**Overflow detected here**

**Handler code**

```
and $12, $2, $5
xor $13, $2, $6
addq $1, $2, $1
or $15, $6, $7
stq $26, 100($31)
```

Multicycle instructions

**Alpha 21264 Execution Times:**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>add / sub</td>
<td>14%</td>
<td>11% 14%</td>
</tr>
<tr>
<td>multiply</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1% 13%</td>
</tr>
<tr>
<td>divide</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1% 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
```

(Not to scale)
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