Basic Pipelining

October 30, 2012

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>000</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>
</tr>
</tbody>
</table>

RI-type instructions (addq, subq, xor, bis, cmplt): \( rc \leftarrow ra \text{ funct } ib \)

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>ib</th>
<th>1</th>
<th>funct</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-21</td>
<td>20-13</td>
<td>12</td>
<td>11-5</td>
<td>4-0</td>
</tr>
</tbody>
</table>

Encoding
• ib is 8-bit unsigned literal

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

Alpha Load/Store Instructions

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

<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
• offset is 16-bit signed offset

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

- **Cond** Branch: 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Op field</th>
<th>Cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>0x39</td>
<td>Ra == 0</td>
</tr>
<tr>
<td>bne</td>
<td>0x3D</td>
<td>Ra != 0</td>
</tr>
</tbody>
</table>

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>

**Instruction Encoding**

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

- **Object Code**

Decomposition of 32-bit words

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**

<table>
<thead>
<tr>
<th>Hint 15:14</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
</tr>
<tr>
<td>jsr</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

Decoding Examples

<table>
<thead>
<tr>
<th>0x0: 40220403</th>
<th>0x1A: e4c70abc 1dq r6, 2748(r7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 0 2 2 0 O 4 0 3</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0100 0100 0000 0101</td>
<td></td>
</tr>
<tr>
<td>29 06 07 0abc</td>
<td></td>
</tr>
<tr>
<td>274810</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x10: e47ffffb beq r3, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x18: 6bfa8001 ret r31, (r26), 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x0: 40220403</th>
<th>0x1A: e4c70abc 1dq r6, 2748(r7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 0 2 2 0 O 4 0 3</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0100 0100 0000 0101</td>
<td></td>
</tr>
<tr>
<td>29 06 07 0abc</td>
<td></td>
</tr>
<tr>
<td>274810</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x10: e47ffffb beq r3, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x18: 6bfa8001 ret r31, (r26), 1</td>
</tr>
</tbody>
</table>

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

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

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>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>

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

Active Datapath for RR & RI
RI-type instructions

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

<table>
<thead>
<tr>
<th>Op</th>
<th>ra</th>
<th>ib</th>
<th>func</th>
<th>pc</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

---

**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

---

**Load instruction**

**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>

**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

---

**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

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 ← Target

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
- One operation must complete before next can begin
- Operations spaced 1.3 ns apart

Delay = 1.3 ns
Throughput = 0.77 GHz

3 Stage Pipelining
- 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 = \( \frac{1}{2.1 \text{ ns}} \) = 1.43 GHz

Limitation: Deep Pipelines

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

Delay = 1.8 ns
Throughput = 3.33 GHz

Limitation: Sequential Dependencies

- Op4 gets result from Op1
- Pipeline Hazard

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

Pipelined Datapath

- Various stages such as PC, Instr. Mem., Mem., ALU, etc.
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
  - Reads 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:

<table>
<thead>
<tr>
<th>Time</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>$2$</td>
<td>$3$</td>
<td>$4$</td>
<td>$5$</td>
<td>$6$</td>
</tr>
<tr>
<td></td>
<td>addq $31, 63, $2$</td>
<td>addq $2, 0, $3$</td>
<td>addq $2, 0, $4$</td>
<td>addq $2, 0, $5$</td>
<td>addq $2, 0, $6$</td>
</tr>
</tbody>
</table>

$2$ 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

```
<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beq $0, target</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov 63, $2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov 63, $3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov 63, $4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov 63, $5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov 63, $6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- PC Updated
- Time
- 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.ID[25:21]: Operand A
  - ID_in.ID[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
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

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

- 37 - 740 F12

- 38 - 740 F12

- 39 - 740 F12

- 40 - 740 F12
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

Some Hazards with Loads & Stores

Data Generated by Load

Data-Store Data
1dq $1, 8($2)
stq $1, 16($2)

Load-ALU
1dq $1, 8($2)
addq $2, $1, $2

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

Data Generated by Store

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

Not a concern for us

Data Generated by ALU

ALU-Store Data
addq $2, $3, $2
stq $3, 8($2)

ALU-Store (or Load) Addr.
addq $1, $3, $2
stq $3, 8($2)

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
1dq $1, 8($2)
stq $1, 16($2)

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

$2 written
Complete Bypassing for ALU & L/S

Impact of Forwarding

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

Load-ALU
1dq $1, 8($2)
addq $2, $1, $2

Just Forward?

Value not available soon enough!

With 1 Cycle Stall
1dq $1, 8($2)
addq $2, $1, $2

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
jr $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

What happens with a branch?

Conditional Branch Instruction Handling

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

Branch Example

Branch Code (demo08.O)
0x0: 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

- 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
  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

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 Example**

- When branch taken
- Generate 3rd bubble
- Begin fetching at target

```
0x0: beq r31, 0xl8 # Take
0x4: addq r31, 0x3f, r1 # Xtra1
0x8: addq r31, 0x3f, r2 # Xtra2
0x0c: addq r31, 0x3f, r3 # Xtra3
0x10: addq r31, 0x3f, r4 # Xtra4
0x18: addq r31, 0x3f, r5 # 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 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Register</th>
<th>ALU</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Transfer</td>
<td>Bubble</td>
<td>Bubble</td>
<td>Bubble</td>
<td>Transfer</td>
</tr>
</tbody>
</table>

Perform when detect taken branch

Canceling Branch Example

- With BEQ in Mem stage
- Will have fetched 3 extra instructions
- But no register or memory updates

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Register</th>
<th>ALU</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0: beq r31, 0x18</td>
<td># Take</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4: addq r31, 0x3f, r1</td>
<td># Xtra1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8: addq r31, 0x3f, r2</td>
<td># Xtra2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xc: addq r31, 0x3f, r3</td>
<td># Xtra3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10: addq r31, 0x3f, r4</td>
<td># Xtra4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18: addq r31, 0x3f, r5</td>
<td># Target</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Canceling Branch Resolution Example

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Register</th>
<th>ALU</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0: beq r31, 0x18</td>
<td># Take</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4: addq r31, 0x3f, r1</td>
<td># Xtra1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8: addq r31, 0x3f, r2</td>
<td># Xtra2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xc: addq r31, 0x3f, r3</td>
<td># Xtra3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10: addq r31, 0x3f, r4</td>
<td># Xtra4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18: addq r31, 0x3f, r5</td>
<td># Target</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Canceling Branch Pipeline Diagram

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Register</th>
<th>ALU</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq $31, target</td>
<td></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, 55</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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