Basic Pipelining

September 20, 2000

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
  - Follows H&P Ch. 3.1 (Chs. 5.1--5.3 in undergrad version of text)

Refined Design
• 5-stage pipeline
  - Similar to early RISC processors
  - Follows H&P Ch. 3.2 (Chs. 6.1--6.7 in undergrad version of text)
• Goal: approach 1 cycle per instruction but with shorter cycle time

Alpha Arithmetic Instructions

| RR-type instructions (addq, subq, xor, bis, cmplt): rc ← ra funct rb |
|-----------------|-----|-----|-----|-----|-----|
| Op  | ra | rb | 000  | funct | rc |
| 31-26 | 25-21 | 20-16 | 15-13 | 12 | 11-5 | 4-0 |

| RI-type instructions (addq, subq, xor, bis, cmplt): rc ← ra funct ib |
|-----------------|-----|-----|-----|-----|-----|
| Op  | ra | ib | 1 | funct | rc |
| 31-26 | 25-21 | 20-13 | 12 | 11-5 | 4-0 |

Encoding
• ib is 8-bit unsigned literal

Alpha Load/Store Instructions

<table>
<thead>
<tr>
<th>Load: Ra ← Mem[Rb + offset]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store: Mem[Rb + offset] ← Ra</td>
</tr>
</tbody>
</table>

<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 |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ldq</td>
<td>0x29</td>
</tr>
<tr>
<td>stq</td>
<td>0x20</td>
</tr>
</tbody>
</table>
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>

Transfers of Control

Jmp, jsr, ret: Ra ← PC + 4; PC ← Rb

<table>
<thead>
<tr>
<th>0xA</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
- call_pal

Instruction Encoding

Object Code

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

Decoding Examples

0x0: 40220403 addq r1, r2, r3
0x4: 4487f805 xor r4, 0x3f, r5
0x8: a4c70abc ldq r6, 2748(r7)
0xc: b5090123 stq r8, 291(r9)
0x10: e47ffffb beq r3, 0
0x14: d35ffffff beq r26, 0(r31)
0x18: 6bfa8001 ret r31, (r26), 1
0x1c: 000abcde call_pal 0xabcde

0x0: 40220403 addq r1, r2, r3
| 4 | 0 | 2 | 2 | 4 | 3 |
| 100 | 0000 | 0100 | 0010 | 0000 | 0000 |
| 1000 | 0110 | 0111 | 0000 | 0100 | 0111 |

0x8: a4c70abc ldq r6, 2748(r7)
| 4 | 0 | 2 | 2 | 4 | 3 |
| 1000 | 0000 | 0100 | 0010 | 0000 | 0000 |
| 1000 | 0110 | 0111 | 0000 | 0100 | 0111 |
| 2748 |

0x10: e47ffffb beq r3, 0
| 4 | 0 | 2 | 2 | 4 | 3 |
| 1000 | 0000 | 0100 | 0010 | 0000 | 0000 |
| 1000 | 0110 | 0111 | 0000 | 0100 | 0111 |
| 1a |

0x18: 6bfa8001 ret r31, (r26), 1
| 4 | 0 | 2 | 2 | 4 | 3 |
| 1000 | 0000 | 0100 | 0010 | 0000 | 0000 |
| 1000 | 0110 | 0111 | 0000 | 0100 | 0111 |
| 1a |

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

**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 <-- 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>

**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>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**

**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**
- A <-- Register[IR[25:21]]
- B <-- 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**

```
<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

beg: PC ← Ra == 0 ? PC + 4 + disp*4 : PC + 4

0x39    ra   disp
31-26  25-21  20-0

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

Jump

jmp, jsr, ret: Ra ← PC+4; PC ← Rb

0xA    ra   rb
31-26  25-21  20-16  15-0

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

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

Ex: Execute
  - Target ← B

MEM: Memory
  - PC ← Target

WB: Write back
  - IR[25:21] ← incrPC

Branch to Subroutine

Branch Subroutine (bsr): Ra ← PC + 4; PC ← PC + 4

0x34    ra   disp
31-26  25-21  20-0

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]] ← oldPC
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

3 Stage Pipelining
- Space operations 13ns apart
- 3 operations occur simultaneously

Pipelining Basics

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

Op1 Op2 Op3 ... Op4

Time

One operation must complete before next can begin
Operations spaced 33ns apart

3 Stage Pipelining
- Delay = 39ns
- Throughput = 77MHz
Limitation: Nonuniform Pipelining

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

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

Limitation: Deep Pipelines

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

Delay = 48ns, Throughput = 128MHz

Limitation: Sequential Dependencies

- Op4 gets result from Op1
- Pipeline Hazard

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

- 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

### Alpha Simulator

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

**Demo Programs**
- AFS740/sim/solve_tk/demos

**Program Display**
- Hex-coded instruction
- Pipe Stage
- Treated as comment
- Reg. File
- Next State
- Current State
**Simulator ALU Example**

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

**Simulator Store/Load Examples**

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

**Simulator Branch Examples**

- **IF**
  - Fetch instruction
- **ID**
  - Fetch operands
- **EX**
  - Test if operand 0
  - Compute target
- **MEM**
  - Taken: Update PC to target
- **WB**
  - Nothing

**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:

```plaintext
<table>
<thead>
<tr>
<th>Time</th>
<th>$2$ written</th>
<th>$3$ written</th>
<th>$4$ written</th>
<th>$5$ written</th>
<th>$6$ written</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$2$</td>
<td>$3$</td>
<td>$4$</td>
<td>$5$</td>
<td>$6$</td>
</tr>
<tr>
<td>1</td>
<td>$2$</td>
<td>$3$</td>
<td>$4$</td>
<td>$5$</td>
<td>$6$</td>
</tr>
<tr>
<td>2</td>
<td>$2$</td>
<td>$3$</td>
<td>$4$</td>
<td>$5$</td>
<td>$6$</td>
</tr>
<tr>
<td>3</td>
<td>$2$</td>
<td>$3$</td>
<td>$4$</td>
<td>$5$</td>
<td>$6$</td>
</tr>
</tbody>
</table>
```
Simulator Data Hazard Example

Operation
- Read in ID
- Write in WB
- Write-before-read register file

```
demo04.O

0x0: 43e7f402 addq r31, 0x3f, r2 # $2 = 0x3F
0x4: 40401403 addq r31, 0x3f, r3 # $3 = 0x3F?
0x8: 40401404 addq r31, 0x3f, r4 # $4 = 0x3F?
0xc: 40401405 addq r31, 0x3f, r5 # $5 = 0x3F?
0x10: 40401406 addq r31, 0x3f, r6 # $6 = 0x3F?
0x14: 47ff041f bis r31, r31, r31
0x18: 00000000 call_pal halt
```

Control Hazards in Alpha Pipeline

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

```
beq $0, target
```

```
mov 63, $2
mov 63, $3
mov 63, $4
mov 63, $5
call_pal halt
```

Branch Example

```
Branch Code (demo08.O)

0x0:  07e00005 beq  r31, 0x18    # Take
0x4:  43e7f401 addq r31, 0x3f, r1    # (Skip) $1 = 0x3F
0x8:  43e7f402 addq r31, 0x3f, r2    # (Skip) $2 = 0x3F
0xc:  43e7f403 addq r31, 0x3f, r3    # (Skip) $3 = 0x3F
0x10: 43e7f404 addq r31, 0x3f, r4    # (Skip) $4 = 0x3F
0x14: 47ff041f bis  r31, r31, r31
0x18: 43e7f405 addq r31, 0x3f, r5    # (Target) $5 = 0x3F
0x1c: 47ff041f bis  r31, r31, r31
0x20: 00000000 call_pal halt
```

Conclusions

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

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

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