Basic Pipelining

October 27, 2009

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 ← ra 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 ← ra 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 | 0x40

Alpha Load/Store Instructions

Load: Ra ← Mem[Rb + offset]
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>

Encoding
- offset is 16-bit signed offset

Operation | Op field
---|---
ldq | 0x29
stq | 0x20
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 1 = 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

jmp, jsr, ret: Ra ← PC+4; PC ← Rb

<table>
<thead>
<tr>
<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>
</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

<table>
<thead>
<tr>
<th>0x00</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>25-0</td>
</tr>
</tbody>
</table>

- Use as halt instruction

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

Instruction Encoding

0x0: 40220403 addq r1, r2, r3
0x4: 4487f805 xor r4, 0x3f, r5
0x8: a4c70abc ldiq r6, 2748(r7)
0xc: b5090123 stq r8, 291(r9)
0x10: e47ffffb beq r3, 0
0x14: d35ffffa bsr r26, 0(r31)
0x18: 6bfa8001 ret r31, (r26), 1
0x1c: 000abcde call_pal 0xabcde

Decoding Examples

0x0: 40220403 addq r1, r2, r3
0x4: 4487f805 xor r4, 0x3f, r5
0x8: a4c70abc ldiq r6, 2748(r7)
0xc: b5090123 stq r8, 291(r9)
0x10: e47ffffb beq r3, 0
0x14: d35ffffa bsr r26, 0(r31)
0x18: 6bfa8001 ret r31, (r26), 1
0x1c: 000abcde call_pal 0xabcde

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

- Program behavior determined by bit encodings
- Disassembler simply converts these words to readable instructions
### 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, cmpq): 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>

#### 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
  - Register[IR[4:0]] <-- ALUOutput

### Active Datapath for RR & RI

- **Instr. Mem.**
- **Data Mem.**
- **Reg. Array**
- **ALU**
- **Instr.**
- **Data Out**
- **IncrPC**
- **Wdest**
- **Wdata**
- **datIn**
- **addr**
- **datA**
- **datB**
- **aluA**
- **aluB**
- **Zero Test**
- **Write Back**
- **Datapath**
- **IF**
- **ID**
- **EX**
- **MEM**
- **WB**
**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>11-5</td>
<td>4-0</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[20:13]

**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**
- 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

---

**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**

<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 ← A op B

**MEM: Memory**
- Mem-Data ← DMemory[ALUOutput]

**WB: Write back**
- Register[IR[25:21]] ← Mem-Data
- 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
- 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

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

ID: Instruction decode/register fetch
- nop

Ex: Execute
- Target ← B

MEM: Memory
- PC ← target

WB: Write back
- Register[IR[25:21]] ← incrPC

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
- nop

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**
  - 1.2ns
  - 0.1ns
  - Delay = 1.3ns
  - Throughput = 0.77GHz

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

**3 Stage Pipelining**

- **Delay = 1.5ns**
- **Throughput = 2.0GHz**

- **Space operations 0.5ns 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 = 0.476GHz

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

Pipe Registers

- Inserted between stages
- Labeled by preceding & following stage
Notes
- Each stage consists of operate logic connecting pipe registers
- WB logic merged into ID
- Additional paths required for forwarding

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

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:

```plaintext
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

```
beq $0, target
mov 63, $2
mov 63, $3
mov 63, $4
```

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