Systems I

Datapath Design I

Topics

- Sequential instruction execution cycle
- Instruction mapping to hardware
- Instruction decoding
Overview

How do we build a digital computer?

- Hardware building blocks: digital logic primitives
- Instruction set architecture: what HW must implement

Principled approach

- Hardware designed to implement one instruction at a time
  - Plus connect to next instruction
- Decompose each instruction into a series of steps
  - Expect that most steps will be common to many instructions

Extend design from there

- Overlap execution of multiple instructions (pipelining)
  - Later in this course
- Parallel execution of many instructions
  - In more advanced computer architecture course
## Y86 Instruction Set

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>halt</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, rB</td>
<td>2</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>rB</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>mrmovl D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>OPl rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Additional Instructions

- addl 6 0
- subl 6 1
- andl 6 2
- xorl 6 3
- jmp 7 0
- jle 7 1
- jl 7 2
- je 7 3
- jne 7 4
- jge 7 5
- jg 7 6

### NOP

- nop 0 0

### HALT

- halt 1 0

### Bitwise Instructions

- rrmovl rA, rB 2 0 rA rB
- irmovl V, rB 3 0 8 rB V
- rrmovl rA, D(rB) 4 0 rA rB D
- mrmovl D(rB), rA 5 0 rA rB D
- OPl rA, rB 6 fn rA rB

### Jump Instructions

- jXX Dest 7 fn Dest
- call Dest 8 0 Dest
- ret 9 0
- pushl rA A 0 rA 8
- popl rA B 0 rA 8
Building Blocks

Combinational Logic
- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control

Storage Elements
- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises
Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

Data Types

- `bool`: Boolean
  - a, b, c, ...
- `int`: words
  - A, B, C, ...
  - Does not specify word size---bytes, 32-bit words, ...

Statements

- `bool a = bool-expr ;`
- `int A = int-expr ;`
HCL Operations

- Classify by type of value returned

Boolean Expressions

- Logic Operations
  - a && b, a || b, !a

- Word Comparisons

- Set Membership
  - A in { B, C, D }
    » Same as A == B || A == C || A == D

Word Expressions

- Case expressions
  - [ a : A; b : B; c : C ]
  - Evaluate test expressions a, b, c, ... in sequence
  - Return word expression A, B, C, ... for first successful test
SEQ Hardware Structure

State
- Program counter register (PC)
- Condition code register (CC)
- Register File
- Memories
  - Access same memory space
  - Data: for reading/writing program data
  - Instruction: for reading instructions

Instruction Flow
- Read instruction at address specified by PC
- Process through stages
- Update program counter
SEQ Stages

**Fetch**
- Read instruction from instruction memory

**Decode**
- Read program registers

**Execute**
- Compute value or address

**Memory**
- Read or write data

**Write Back**
- Write program registers

**PC**
- Update program counter
Instruction Decoding

Instruction Format

- Instruction byte icode:ifun
- Optional register byte rA:rB
- Optional constant word valC
Executing Arith./Logical Operation

**Fetch**
- Read 2 bytes

**Decode**
- Read operand registers

**Execute**
- Perform operation
- Set condition codes

**Memory**
- Do nothing

**Write back**
- Update register

**PC Update**
- Increment PC by 2
- Why?

**OP1 rA, rB**

**6 fn rA rB**
## Stage Computation: Arith/Log. Ops

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>iCode:ifun ← M₁[PC]  [PC+1]</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M₁[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+2</td>
</tr>
<tr>
<td>Decode</td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td></td>
<td>valB ← R[rB]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB OP valA</td>
</tr>
<tr>
<td></td>
<td>Set CC</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td>R[rB] ← valE</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
Executing `rmmovl`

```
rmmovl rA, D(rB)  [4 0 rA rB]  D
```

**Fetch**
- Read 6 bytes

**Decode**
- Read operand registers

**Execute**
- Compute effective address

**Memory**
- Write to memory

**Write back**
- Do nothing

**PC Update**
- Increment PC by 6
## Stage Computation: `rmmovl`

<table>
<thead>
<tr>
<th>Fetch</th>
<th><code>rmmovl rA, D(rB)</code></th>
<th>Read instruction byte</th>
<th>Read register byte</th>
<th>Read displacement D</th>
<th>Compute next PC</th>
<th>Read operand A</th>
<th>Read operand B</th>
<th>Compute effective address</th>
<th>Write value to memory</th>
<th>Update PC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>icode:ifun ← M_1[PC]</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>rA:rB ← M_1[PC+1]</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>valC ← M_4[PC+2]</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>valP ← PC+6</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td><code>valA ← R[rA]</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>valB ← R[rB]</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + valC</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td><code>M_4[valE] ← valA</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td><code>PC ← valP</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Use ALU for address computation**
Executing `popl`

**Fetch**
- Read 2 bytes

**Decode**
- Read stack pointer

**Execute**
- Increment stack pointer by 4

**Memory**
- Read from old stack pointer

**Write back**
- Update stack pointer
- Write result to register

**PC Update**
- Increment PC by 2
**Stage Computation: popl**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode:ifun</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>rA:rB</td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td>valP</td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>valA</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td></td>
<td>valB</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td>valE</td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td>valM</td>
<td>Read from stack</td>
</tr>
<tr>
<td>Write back</td>
<td>R[esp]</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td></td>
<td>valE</td>
<td>Write back result</td>
</tr>
<tr>
<td>PC update</td>
<td>PC</td>
<td>Update PC</td>
</tr>
</tbody>
</table>

- **Use ALU to increment stack pointer**
- **Must update two registers**
  - Popped value
  - New stack pointer
Summary

Today
- Sequential instruction execution cycle
- Instruction mapping to hardware
- Instruction decoding

Next time
- Control flow instructions
- Hardware for sequential machine (SEQ)