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

| Byte | Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td><code>nop</code></td>
</tr>
<tr>
<td>1 0</td>
<td><code>halt</code></td>
</tr>
<tr>
<td>2 0 rA rB</td>
<td><code>rrmovl</code> rA, rB</td>
</tr>
<tr>
<td>3 0 8 rB</td>
<td><code>irmovl</code> V, rB</td>
</tr>
<tr>
<td>4 0 rA rB</td>
<td><code>rmmovl</code> rA, D(rB)</td>
</tr>
<tr>
<td>5 0 rA rB</td>
<td><code>rmmovl</code> D(rB), rA</td>
</tr>
<tr>
<td>6 fn rA rB</td>
<td><code>OP1</code> rA, rB</td>
</tr>
<tr>
<td>7 fn Dest</td>
<td><code>jXX</code> Dest</td>
</tr>
<tr>
<td>8 0</td>
<td><code>call</code> Dest</td>
</tr>
<tr>
<td>9 0</td>
<td><code>ret</code></td>
</tr>
<tr>
<td>A 0 rA 8</td>
<td><code>pushl</code> rA</td>
</tr>
<tr>
<td>B 0 rA 8</td>
<td><code>popl</code> rA</td>
</tr>
</tbody>
</table>

### Example Instructions

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

### Byte Codes

- `byte 0 0`: Instruction `nop` (no operation)
- `byte 1 0`: Instruction `halt`
- `byte 2 0 rA rB`: Instruction `rrmovl` rA, rB
- `byte 3 0 8 rB`: Instruction `irmovl` V, rB
- `byte 4 0 rA rB`: Instruction `rmmovl` rA, D(rB)
- `byte 5 0 rA rB`: Instruction `rmmovl` D(rB), rA
- `byte 6 fn rA rB`: Instruction `OP1` rA, rB
- `byte 7 fn Dest`: Instruction `jXX` Dest
- `byte 8 0`: Instruction `call` Dest
- `byte 9 0`: Instruction `ret`
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
  - $a \&\& b$, $a \mid \mid b$, $!a$

- Word Comparisons

- Set Membership
  - $A$ in $\{ B, C, D \}$
    - Same as $A == B \mid \mid A == C \mid \mid A == D$

**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
An Abstract Processor

What does a processor do?

Consider a processor that only executes nops.

```c
void be_a_processor(unsigned int pc, unsigned char* mem)
{
    while(1) {
        char opcode = mem[pc];
        assert(opcode == NOP);
        pc = pc + 1;
    }
}
```

Fetch
Decode
Execute
An Abstract Processor

Executes nops and absolute jumps

```c
void be_a_processor(unsigned int pc,
                     unsigned char* mem)
{
    while(1) {
        char opcode = mem[pc];
        switch (opcode) {
            case NOP: pc++;
            case JMP: pc = *(int*)&mem[(pc+1)];
        }
    }
```

Missing execute and memory access
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   \text{icode:ifun}
- Optional register byte   \text{rA:rB}
- Optional constant word   \text{valC}
Executing Arith./Logical Operation

- **OP1 rA, rB**

**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?

12
### Stage Computation: Arith/Log. Ops

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch         | Read instruction byte
                | Read register byte                                                         |
|               | Compute next PC                                                            |
| Decode        | Read operand A                                                             |
|               | Read operand B                                                             |
| Execute       | Perform ALU operation                                                      |
|               | Set condition code register                                                |
| Memory        | Write back result                                                          |
| Write back    | Update PC                                                                  |
| PC update     |                                                                            |

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

- **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

The assembly instruction `rmmovl rA, D(rB)` is shown with its fields: 4 0 | rA rB | D.
# Stage Computation: \texttt{rmmovl}

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>\texttt{icode:ifun} \leftarrow M_1[PC] \br \texttt{rA:rB} \leftarrow M_1[PC+1] \br \texttt{valC} \leftarrow M_4[PC+2] \br \texttt{valP} \leftarrow PC+6</td>
<td>Read instruction byte \br Read register byte \br Read displacement D \br Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>\texttt{valA} \leftarrow R[rA] \br \texttt{valB} \leftarrow R[rB]</td>
<td>Read operand A \br Read operand B</td>
</tr>
<tr>
<td>Execute</td>
<td>\texttt{valE} \leftarrow \texttt{valB} + \texttt{valC}</td>
<td>Compute effective address</td>
</tr>
<tr>
<td>Memory</td>
<td>\texttt{M_4[valE]} \leftarrow \texttt{valA}</td>
<td>Write value to memory</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>\texttt{PC} \leftarrow \texttt{valP}</td>
<td>Update PC</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>Operation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>popl rA</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>icode:ifun ← M_1[PC]</code></td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td><code>rA:rB ← M_1[PC+1]</code></td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td><code>valP ← PC+2</code></td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td><code>valA ← R[.esp]</code></td>
<td>Read stack pointer</td>
</tr>
<tr>
<td></td>
<td><code>valB ← R[esp]</code></td>
<td>Read stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + 4</code></td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td><code>valM ← M_4[valA]</code></td>
<td>Read from stack</td>
</tr>
<tr>
<td>Write back</td>
<td><code>R[esp] ← valE</code></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td></td>
<td><code>R[rA] ← valM</code></td>
<td>Write back result</td>
</tr>
<tr>
<td>PC update</td>
<td><code>PC ← valP</code></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)