Sequential Implementation of Y86

CSci 2021: Machine Architecture and Organization
Lecture #19, March 6th, 2015

Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O'Hallaron, Antonia Zhai

Y86 Instruction Set #1

<table>
<thead>
<tr>
<th>Byte</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovX a, b</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imovi V, b</td>
<td>B</td>
<td>B</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovi a, D</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmovi D(b), r</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp a, b</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>B</td>
<td>B</td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>B</td>
<td>B</td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl a</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl a</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86 Instruction Set #2

<table>
<thead>
<tr>
<th>Byte</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovX a, b</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imovi V, b</td>
<td>B</td>
<td>B</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovi a, D</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmovi D(b), r</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp a, b</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>B</td>
<td>B</td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>B</td>
<td>B</td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl a</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl a</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86 Instruction Set #3

<table>
<thead>
<tr>
<th>Byte</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovX a, b</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imovi V, b</td>
<td>B</td>
<td>B</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovi a, D</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmovi D(b), r</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp a, b</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>B</td>
<td>B</td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>B</td>
<td>B</td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl a</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl a</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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-exp ;
- int A = int-exp ;
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

Instruction Decoding
- Instruction Format
  - Instruction byte $icode:ifun$
  - Optional register byte $rA:rB$
  - Optional constant word $valC$

Stage Computation: Arith/Log. Ops
- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
### Executing `rmmovl` 

**Stage Computation: rmmovl**

<table>
<thead>
<tr>
<th>rmmovl</th>
<th>Stage Computation: rmmovl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode: ifun $\rightarrow$ M[PC]</td>
</tr>
<tr>
<td></td>
<td>rA:B $\rightarrow$ M[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valC $\rightarrow$ M[PC+2]</td>
</tr>
<tr>
<td>Decode</td>
<td>valA $\rightarrow$ R[A]</td>
</tr>
<tr>
<td></td>
<td>valB $\rightarrow$ R[B]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE $\rightarrow$ valB + valC</td>
</tr>
<tr>
<td>Memory</td>
<td>M[valE] $\rightarrow$ valA</td>
</tr>
<tr>
<td></td>
<td>Write valE to memory</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC $\leftarrow$ valP</td>
</tr>
<tr>
<td>Update PC</td>
<td></td>
</tr>
</tbody>
</table>

- Use ALU for address computation

### Executing `popl` 

**Stage Computation: popl**

<table>
<thead>
<tr>
<th>popl</th>
<th>Stage Computation: popl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode: ifun $\rightarrow$ M[PC]</td>
</tr>
<tr>
<td></td>
<td>rA:B $\rightarrow$ M[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP $\rightarrow$ PC+2</td>
</tr>
<tr>
<td>Decode</td>
<td>valB $\rightarrow$ R[esp]</td>
</tr>
<tr>
<td></td>
<td>valB $\rightarrow$ R[esp+4]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE $\rightarrow$ valB + 4</td>
</tr>
<tr>
<td>Memory</td>
<td>valM $\rightarrow$ valA</td>
</tr>
<tr>
<td></td>
<td>R[rA] $\rightarrow$ valM</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC $\leftarrow$ valP</td>
</tr>
<tr>
<td>Update PC</td>
<td></td>
</tr>
</tbody>
</table>

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

### Executing Jumps 

**Stage Computation: Jumps**

|  || Stage Computation: Jumps |
|---|-------------------------|
| Fetch | icode: ifun $\rightarrow$ M[PC] |
|       | valC $\rightarrow$ M[PC+1] |
|       | valP $\rightarrow$ PC+5 |
| Decode | |
| Execute | Cnd $\rightarrow$ Cond(CC, ifun) |
| Memory | |
| Write back | |
| PC update | PC $\leftarrow$ Cnd ? valC : valP |

- Compute both addresses
- Choose based on setting of condition codes and branch condition

---

**Executing `rmmovl`**

- Read 6 bytes
- Write to memory
- Do nothing
- Compute effective address
- Write to memory
- Do nothing
- Increment PC by 6

**Executing `popl`**

- Read 2 bytes
- Read from old stack pointer
- Update stack pointer
- Write result to register
- Increment stack pointer by 4
- Increment PC by 2

**Executing Jumps**

- Read 5 bytes
- Increment PC by 5
- Do nothing
- Increment PC by 5
- Set PC toDest if branch taken or to incremented PC if not branch

---

**Stage Computation: rmmovl**

- Read instruction byte
- Read register byte
- Read displacement D
- Read operand A
- Compute next PC
- Read operand B
- Compute effective address
- Write value to memory
- Increment PC by 6

**Stage Computation: popl**

- Read instruction byte
- Read register byte
- Compute next PC
- Read stack pointer
- Increment stack pointer
- Increment stack pointer
- Update stack pointer
- Write result to register
- Update PC

**Stage Computation: Jumps**

- Read destination address
- Fall through address
- Take branch?
Executing call

**Fetch**
- Read 5 bytes
- Increment PC by 5

**Decode**
- Read stack pointer

**Execute**
- Decrement stack pointer by 4
- Write incremented PC to new value of stack pointer
- Update stack pointer
- Set PC to Dest

Executing ret

**Fetch**
- Read 1 byte

**Decode**
- Read stack pointer

**Execute**
- Increment stack pointer by 4
- Read return address from old stack pointer
- Update stack pointer
- Set PC to return address

Stage Computation: call

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
<th>PC update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read instruction byte</td>
<td>Read stack pointer</td>
<td>Decrement stack pointer</td>
<td>Read return value on stack</td>
<td>Update stack pointer</td>
<td>Set PC to destination</td>
</tr>
</tbody>
</table>

Stage Computation: ret

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
<th>PC update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read operand A</td>
<td>Read operand B</td>
<td>Perform ALU operation</td>
<td>Read return address</td>
<td>Update stack pointer</td>
<td>Set PC to return address</td>
</tr>
</tbody>
</table>

Computation Steps

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

- All instructions follow same general pattern
- Differ in what gets computed on each step
### Computed Values

**Fetch**
- icode: Instruction code
- ifun: Instruction function
- rA: Instr. Register A
- rB: Instr. Register B
- valC: Instruction constant
- valP: Incremented PC

**Decode**
- srcA: Register ID A
- srcB: Register ID B
- dstE: Destination Register E
- dstM: Destination Register M
- valA: Register value A
- valB: Register value B

**Execute**
- valE: ALU result
- Cnd: Branch/move flag

**Memory**
- valM: Value from memory

### Administrative Break
- Quiz 1 solutions: posted on Moodle, probably late tonight
- Quiz 1 return: on Monday
- Buffer lab: due next Wednesday
- Assignment III: out next Wednesday, due Monday after break

### SEQ Hardware

**Key**
- Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 32-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values

### Fetch Logic

**Predefined Blocks**
- PC: Register containing PC
- Instruction memory: Read 6 bytes (PC to PC+5)
- Signal invalid address
- Split: Divide instruction byte into icode and ifun
- Align: Get fields for rA, rB, and valC

### Fetch Control Logic in HCL

```hcl
# Determine instruction code
int icode = [imem_error: INOP;
1: imem_icode];

# Determine instruction function
int ifun = [imem_error: FNONE;
1: imem_ifun];
```
Fetch Control Logic in HCL

bool need_regs = icode in { IRRMOVL, IOPL, IPUSHL, IPOPL, IIRMOVL, IRMMOVIL, IMRMOVL };
bool instr_valid = icode in { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVIL, IIRMOVL, ISP, IJXX, ICALL, INET, IPUSHL, IPOPL };

A Source

decode_OP(rA, rB) = Read operand A
romovX(rA, rB) = Read operand A
romovI(rA, rB) = Read operand A
impiA(rA) = Read stack pointer
jal Dest = No operand
jal Dest = No operand
jal Dest = No operand
jal Dest = No operand
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointer
jal Dest = Read stack pointe
ALU Operation

- `OPrA, rB` Perform ALU operation
- `valB ← valR OP valA`
- `movXX rA, rB` Write value to ALU
- `valA ← 0` Compute effective address
- `valE ← valB + valC` Increment stack pointer
- `valE ← valB + 4` Increment stack pointer

```
int alufun = {
  icode == IOPL : ifun;
  1 : ALUADD;
};
```

Memory Logic

- **Memory**
  - Reads or writes memory word
- **Control Logic**
  - stat: What is instruction status?
  - Mem. read: Should word be read?
  - Mem. write: Should word be written?
  - Mem. addr.: Select address
  - Mem. data.: Select data

Instruction Status

- **Control Logic**
  - stat: What is instruction status?

```
## Determine instruction status
int stat = {
  instr_error || instr_mem_error : SADR;
  !instr_valid : SINS;
  icode == IHALT : SHLT;
  1 : SAOK;
};
```

Memory Address

- **Memory**
  - OPrA, rB No operation
  - movXX rA, D(rB) Write value to memory
  - popl rA Read from stack
  - rmmovl rA, D(rB) No operation
  - rmmovl rA, M[valA] Write return value on stack
  - ret Read return address

```
int mem_addr = {
  icode in { IMRMOVL, IPOPL, IRET } : valA;
  1 : SADR;
};
```

Memory Read

- **Memory**
  - OPrA, rB No operation
  - movXX rA, D(rB) Write value to memory
  - popl rA Read from stack
  - rmmovl rA, M[valA] No operation
  - rmmovl rA, M[valA] Write return value on stack
  - ret Read return address

```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

PC Update Logic

- **New PC**
  - Select next value of PC

```
new PC
```
int new_pc = {
    icode == ICALL : valC;
    icode == IJXX && valC;
    icode == IBRT : valM;
    1 : valP;
};
## SEQ Summary

### Implementation
- Express every instruction as series of simple steps
- Follow same general flow for each instruction type
- Assemble registers, memories, predesigned combinational blocks
- Connect with control logic

### Limitations
- Too slow to be practical
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- Would need to run clock very slowly
- Hardware units only active for fraction of clock cycle