Computer Architecture
Sequential Implementation

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

With Slides from Bryant, O’Hallaron and Zhai

Y86-64 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>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86-64 Instruction Set #1

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86-64 Instruction Set #2

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Y86-64 Instruction Set #3

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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
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
  - 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

Stage Computation: Arith/Log. Ops

- **Fetch**
  - Read instruction byte
  - Read register byte
  - `icode:ifun = M(PC)`
  - `rA, rB = M(PC+1)`
- **Decode**
  - `valA = R(rA)`
  - `valB = R(rB)`
- **Execute**
  - `valE = valA OF valB`
  - Set CC
- **Memory**
  - `valM = R(rB)`
- **Write back**
  - Write back result
- **Update PC**
  - Increment PC by 2

Executing `rmmovq`

- **Fetch**
  - Read 10 bytes
- **Decode**
  - Write to memory
- **Execute**
  - Do nothing
- **Memory**
  - Write back
- **PC Update**
  - Increment PC by 10
Stage Computation: *rmovq*

**rmovq rA, D(rB)**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch | iCode: iIfun ← M[PC]  
 rA: rB ← M[PC+1]  
 valC ← M[PC+2]  
 PC ← PC+10 |
| Decode| valA ← R[rA]  
 valB ← R[rB]  
 valE ← valB + valC |
| Execute| valE ← valE + valC  
 Write | M[valE] ← valA |
| Memory| Write to memory |
| Write back | PC update: PC ← valP |

- Use ALU for address computation

Stage Computation: *popq*

**popq rA**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch | iCode: iIfun ← M[PC]  
 rA: rB ← M[PC+1]  
 valC ← M[PC+2]  
 PC ← PC+10 |
| Decode| valA ← R[rA]  
 valB ← R[rB]  
 valE ← valB + 8 |
| Execute| valE ← valE + 8  
 Write | R[rB] ← valE |
| Memory| Write to memory |
| Write back | PC update: PC ← valP |

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

Stage Computation: Cond. Move

**cmovXX rA, rB**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch | iCode: iIfun ← M[PC]  
 rA: rB ← M[PC+1]  
 valC ← M[PC+2]  
 PC ← PC+10 |
| Decode| valA ← R[rA]  
 valB ← 0  
 IF = Cond(CC) iIfun rB |
| Execute| valE ← valB + valA  
 Write | R[rB] ← valE |
| Memory| Write to memory |
| Write back | PC update: PC ← valP |

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
  - If condition codes & move condition indicate no move

Executing *popq*

**popq rA**

- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
  - Read stack pointer
- Execute
  - Increment stack pointer by 8
  - Update stack pointer
- Memory
  - Read from old stack pointer
  - Update stack pointer
  - Write result to register
  - PC Update
  - Increment PC by 2

Executing Conditional Moves

**cmovXX rA, rB**

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
  - Read operand registers (or not)
- Execute
  - If Cond, then set destination register to 0xF
  - If Cond, then set destination register to 0xF
- Memory
  - Do nothing
  - Write back
  - Update register (or not)
  - PC Update
  - Increment PC by 2

Executing Jumps

**jXX Dest**

- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch
- Memory
  - Do nothing
- Write back
  - Do nothing

- Execute
  - Determine whether to take branch based on jump condition and condition codes
### Stage Computation: Jumps

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>iCode:ifun ← M[PC]</td>
</tr>
<tr>
<td></td>
<td>valC ← M[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+9</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Cnd ← Cond((CC,ifun))</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← Cond ? valC : valP</td>
</tr>
</tbody>
</table>

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

### Stage Computation: Call

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>iCode:ifun ← M[PC]</td>
</tr>
<tr>
<td></td>
<td>valC ← M[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+9</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + 8</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valC</td>
</tr>
</tbody>
</table>

- Use ALU to decrement stack pointer
- Store incremented PC

### Stage Computation: Ret

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>iCode:ifun ← M[PC]</td>
</tr>
<tr>
<td></td>
<td>valA ← R[esp]</td>
</tr>
<tr>
<td></td>
<td>valB ← R[esp]</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + 8</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valM</td>
</tr>
</tbody>
</table>

- Use ALU to increment stack pointer
- Read return address from memory

### Executing Call

- **Fetch**
  - Read 9 bytes
  - Increment PC by 9
- **Decode**
  - Read stack pointer
- **Execute**
  - Decrement stack pointer by 8

### Executing Ret

- **Fetch**
  - Read 1 byte
- **Decode**
  - Read return address from old stack pointer
- **Execute**
  - Increment stack pointer by 8

### Computation Steps

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

- **Fetch**
  - `icode, ifun ← M[PC]`
  - Read instruction byte
  - `valC ← M[PC+1]`
  - Read constant word
  - `valP ← PC+9`
  - Compute next PC

- **Decode**
  - `valA, srcA`
  - Read operand A
  - `valB, srcB`
  - Read operand B

- **Execute**
  - `valE ← valB - 8`
  - Perform ALU operation

- **Memory**
  - `valM ← srcE`
  - Value from memory

- **Write back**
  - `PC ← valC`
  - Write back ALU result

### Computed Values

- **Fetch**
  - `valE` ALU result
  - `valM` Value from memory

- **Decode**
  - `srcA` Register ID A
  - `srcB` Register ID B
  - `dstE` Destination Register E
  - `dstM` Destination Register M

- **Execute**
  - `valA` Register value A
  - `valB` Register value B

### SEQ Hardware

- **Key**
  - Blue boxes: predesigned hardware blocks
  - Gray boxes: control logic
  - White ovals: labels
  - Thick lines: 64-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 10 bytes (PC to PC+9)
  - Signal invalid address
  - Split: Divide instruction byte into `icode` and `ifun`

### Fetch Control Logic in HCL

```hcl
# Determine instruction code
int icode = {
    imem_error: IMOP;
    1: imem_icode;
}:
```

```hcl
# Determine instruction function
int ifun = {
    imem_error: PHONE;
    1: imem_ifun;
}:
```
Fetch Control Logic in HCL

Control Logic
- Set CC: Should condition code register be loaded?
- ALU A: Input A to ALU
- ALU B: Input B to ALU
- ALU Fun: What function should ALU compute?
- ALU B: Input B to ALU
- ALU Fun: What function should ALU compute?

Decode Logic
- Register file:
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 15 (0W) (no access)

Control Logic
- srcA, srcB: read port addresses
- dstE, dstM: data port addresses

Signals
- Cnd: Indicate whether or not to perform conditional move
  - Computed in Execute stage

A Source

Decode
- Read operand A

Decode
- Read operand A

E Destination

Write-back
- Write-back result

Write-back
- Conditionally write back result

Write-back
- None

Write-back
- Update stack pointer

Write-back
- Update stack pointer

Write-back
- Update stack pointer

Execute Logic
- Units:
  - ALU: Implements 4 required functions
    - Generates condition code values
  - CC: Register with 3 condition code bits
    - cond: Computes conditional jump/move flag

ALU A Input

Execute
- ALU inputs: Pass valid through ALU

Execute
- Compute effective address

Execute
- Increment stack pointer

Execute
- No operation

int aluA = [
  icode in { IRMMOVQ, IDQ } : valA;
  icode in { IRMMOVQ, IDQ } : valC;
  icode in { IPOPQ, IPUSHQ } : 0;
  icode in { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ } : 0;
];

int aluB = [
  icode in { IRMMOVQ, IDQ } : valA;
  icode in { IRMMOVQ, IDQ } : valC;
  icode in { IPOPQ, IPUSHQ } : 0;
  icode in { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ } : 0;
];

int srcA = [
  icode in { IRRMOVQ, IRMOVQ, IDQ, IPUSHQ } : ra;
  icode in { IPOPQ, IRET } : RSP;
  1 : RNONE; # Don't need register
];

int srcB = [
  icode in { IRRMOVQ, IRMOVQ, IDQ, IPUSHQ } : rb;
  icode in { IPOPQ, IRET } : RSP;
  1 : RNONE; # Don't write any register
];

int dstE = [
  icode in { ICALL, IPUSHQ } : -8;
  icode in { IPOPQ, IRET } : 8;
  in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : 0;
  in { IPOPQ, IRET } : 8;
];

int dstM = [
  icode in { ICALL, IPUSHQ } : 0;
  icode in { IPOPQ, IRET } : 8;
  in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : 8;
  in { IPOPQ, IRET } : 8;
];
### ALU Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OP1 rA, rB</code></td>
<td>Execute <code>valE ← valB OP valA</code></td>
</tr>
<tr>
<td><code>OPXX rA, rB</code></td>
<td>Pass <code>valA</code> through ALU</td>
</tr>
<tr>
<td><code>removi rA, D(rB)</code></td>
<td>Compute effective address</td>
</tr>
<tr>
<td><code>popq rA</code></td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td>No operation</td>
</tr>
<tr>
<td><code>ret</code></td>
<td>Increment stack pointer</td>
</tr>
</tbody>
</table>

### Instruction Status

- **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

### 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

### Memory Read

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OPq rA, rB</code></td>
<td>No operation</td>
</tr>
<tr>
<td><code>removi rA, D(rB)</code></td>
<td>Write value to memory</td>
</tr>
<tr>
<td><code>popq rA</code></td>
<td>Read from stack</td>
</tr>
<tr>
<td><code>jXX Dest</code></td>
<td>No operation</td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td>Write return value on stack</td>
</tr>
<tr>
<td><code>ret</code></td>
<td>Read return address</td>
</tr>
</tbody>
</table>

### Memory Address

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OPq rA, rB</code></td>
<td>No operation</td>
</tr>
<tr>
<td><code>removi rA, D(rB)</code></td>
<td>Write value to memory</td>
</tr>
<tr>
<td><code>popq rA</code></td>
<td>Read from stack</td>
</tr>
<tr>
<td><code>jXX Dest</code></td>
<td>No operation</td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td>Write return value on stack</td>
</tr>
<tr>
<td><code>ret</code></td>
<td>Read return address</td>
</tr>
</tbody>
</table>

### PC Update Logic

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

- Update PC
- Instruction execution
- Control logic
- Memory reads
- Instruction memory
- Register
- Data memory

SEQ Operation

- State
  - PC register
  - Cond. Code register
  - Data memory
  - Register file
  - All updated as clock rises
- Combinational Logic
  - ALU
  - Control logic
  - Memory reads
  - Instruction memory
  - Register file
  - Data memory

SEQ Operation #2

- State set according to second \( \text{imovq} \) instruction
- Combinational logic starting to react to state changes

SEQ Operation #3

- State set according to second \( \text{imovq} \) instruction
- Combinational logic generates results for \( \text{addq} \) instruction

SEQ Operation #4

- State set according to \( \text{addq} \) instruction
- Combinational logic starting to react to state changes

SEQ Operation #5

- State set according to \( \text{addq} \) instruction
- Combinational logic generates results for \( \text{je} \) instruction
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