Computer Architecture: Y86-64 Sequential Implementation

CSci 2021: Machine Architecture and Organization
March 23rd-25th, 2020

Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant and Dave O’Hallaron

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>
<th>9</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>
<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>
<td></td>
</tr>
<tr>
<td>cmovX</td>
<td>d, b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>limp</td>
<td>v, d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmovq</td>
<td>v, d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmovq</td>
<td>v(r), m</td>
<td></td>
<td></td>
<td></td>
<td>m</td>
<td>m</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmov</td>
<td>rA, rB</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>V, rB</td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq</td>
<td>rA, D(rB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq</td>
<td>D(rB), rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq</td>
<td>rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>0</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>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>halt</td>
<td>0</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

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
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 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

Stage Computation: rmmovq

- Use ALU for address computation

Executing rmmovq

- Read 10 bytes
- Write to memory
- Do nothing
- Compute effective address
- Increment PC by 10
Executing `popq`

- **Stage Computation: popq**
  - **Fetch**: Read instruction byte, Read register byte
  - **Decode**: Compute next PC, Read stack pointer, Read stack pointer, Increment stack pointer
  - **Execute**: Read from stack, Update stack pointer, Update stack pointer
  - **Memory**: Read from old stack pointer
  - **Write back**: Update stack pointer
  - **PC Update**: Increment PC by 2

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

Executing Conditional Moves

- **Stage Computation: Cond. Move**
  - **Fetch**: Read instruction byte, Read register byte
  - **Decode**: Do nothing
  - **Execute**: Pass value through ALU (Disable register update)
  - **Memory**: Read from stack
  - **Write back**: Write back result
  - **PC Update**: Update PC

Executing Jumps

- **Stage Computation: Jumps**
  - **Fetch**: Read destination address, Fall through address
  - **Decode**: Read operand A
  - **Execute**: Take branch?
  - **Memory**: Do nothing
  - **Write back**: Write back result
  - **PC Update**: Update PC

- **Using ALU to increment stack pointer**
  - **Must update two registers**
    - Popped value
    - New stack pointer
### Executing call

**call Dest**

<table>
<thead>
<tr>
<th>return</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1.png" alt="Image" /></td>
</tr>
</tbody>
</table>

**Fetch**
- Read 9 bytes
- Increment PC by 9

**Decode**
- Read stack pointer

**Execute**
- Decrement stack pointer by 8

- Use ALU to decrement stack pointer
- Store incremented PC
- Read return address from memory
- Read destination address
- Compute return point
- Read stack pointer
- Decrement stack pointer
- Write return addr. on stack
- Update stack pointer
- Set PC to destination

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

### Executing ret

**ret**

| return | ![Image](image2.png) |

**Fetch**
- Read 1 byte

**Decode**
- Read stack pointer

**Execute**
- Increment stack pointer by 8

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

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

### Computation Steps

**call Dest**

<table>
<thead>
<tr>
<th>code/func</th>
<th>PC[RA, RB]</th>
<th>RA</th>
<th>RB</th>
<th>rA,rB</th>
<th>rA,B = M[PC+1]</th>
</tr>
</thead>
</table>

- Read instruction byte
- Read register byte
- Read constant word
- Compute next PC

**ValC**

| valC | ![Image](image3.png) |

- Read destination address
- Compute return point
- Read stack pointer
- Decrement stack pointer
- Write return addr. on stack
- Update stack pointer
- Set PC to destination

**ValB**

| valB | ![Image](image4.png) |

- Read operand stack pointer
- Read operand stack pointer
- Increment stack pointer
- Read return address

**ValE**

| valE | ![Image](image5.png) |

- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write
- Write back ALU result
- Write back memory result
- Update PC

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

**Stage Computation: call**

- Use ALU to decrement stack pointer
- Store incremented PC

**Stage Computation: ret**

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

**Computation Steps**
Computed Values

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Execute</th>
<th>Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>iCode</td>
<td>icode</td>
<td>srcA</td>
</tr>
<tr>
<td>Ifun</td>
<td>ifun</td>
<td>srcB</td>
</tr>
<tr>
<td>rA</td>
<td>rA</td>
<td>dstE</td>
</tr>
<tr>
<td>rB</td>
<td>rB</td>
<td>dstM</td>
</tr>
<tr>
<td>valC</td>
<td>valC</td>
<td>valA</td>
</tr>
<tr>
<td>valP</td>
<td>valP</td>
<td>valB</td>
</tr>
<tr>
<td>Incremented PC</td>
<td>valM</td>
<td>Instruction constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction Register A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction Register B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction Register M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register value A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register value B</td>
</tr>
</tbody>
</table>

Comma       | Comma                    | Comma                    |

Execute
- **valE**: ALU result
- **Cnd**: Branch/move flag
- **Mem**: Value from memory

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
- **Align**: Get fields for rA, rB, and valC

Fetch Logic

Control Logic
- **Instr. Valid**: Is this instruction valid?
- **icode, Ifun**: Generate no-op if invalid address
- **Need regids**: Does this instruction have a register byte?
- **Need valC**: Does this instruction have a constant word?

Fetch Control Logic in HCL

```
# Determine instruction code
int iCode = {
    imem_error: INOP;
    1: imem_iCode;
};

# Determine instruction function
int ifun = {
    imem_error: FNONE;
    1: imem_ifun;
};
```

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: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values
Decide Logic

Register File
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

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

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

A Source

int srcA = {  
    icode in { IRRMOVQ, IOPQ } : rA;  
    icode in { IPUSHQ, IPUSHQ } : rA;  
    1 : RNONE; # Don’t need register
};

ALU A Input

 ALU Operation
- Perform ALU operation
- Pass valA through ALU
- Compute effective address
- Increment stack pointer
- Decrement stack pointer
- No operation

ALU Fun: What function should ALU perform?

Write ports A, B
Read stack pointer
Read operand A
Write ports A, B
Write stack pointer
Decrement stack pointer
Increment stack pointer
Pass

Execute Logic

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

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?
Memory Logic

- 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

```c
int Stat = !imem_error || !dmem_error : SADR;
!instr_valid : SINS;
icode == ISHALT : SHLT:
1 : SAOK;
```

Memory Address

- OPq, rA, rB
- No operation
- rmovq, rA, D(rB)
- Write value to memory
- popq, rA
- Read from stack
- call Dest
- No operation
- M[rA] = MvalP
- Write return value on stack
- ret
- Read return address

```c
int mem_addr = { icode in { IMMOVQ, IPUSHQ, ICALL, INMOVQ } : valE; icode in { IPOPQ, IRET } : valA; icode in { IRMMOVQ } : valE; icode in { IJXX } : valC; icode in { ICALL } : valC; icode in { IRET } : valA; # Other instructions don't need address
};
```

Memory Read

- OPq, rA, rB
- No operation
- rmovq, rA, D(rB)
- Write value to memory
- popq, rA
- Read from stack
- call Dest
- No operation
- M[rA] = MvalP
- Write return value on stack
- ret
- Read return address

```c
bool mem_read = icode in { IMMOVQ, IPUSHQ, ICALL, INMOVQ };
```

PC Update Logic

- New PC
- Select next value of PC

```c
int new_pc = { icode in { ICALL } : valC; icode in { IJXX && Cnd } : valC; icode in { ISHALT } : valM; 1 : valP; }
```
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 irmovq instruction
- Combinational logic starting to react to state changes

SEQ Operation #3

- State set according to second addq instruction
- Combinational logic generates results for addq instruction

SEQ Operation #4

- State set according to addq instruction
- Combinational logic starting to react to state changes

SEQ Operation #5

- State set according to addq instruction
- Combinational logic generates results for 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