Computer Architecture: Y86-64 Sequential Implementation

CSci 2021: Machine Architecture and Organization
October 31st, 2018

Your instructor: Stephen McCamant

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

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 expression;
- int A = int-exp expression;

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

Instruction Format

- Instruction byte: icode/ifun
- Optional register byte: rA:rB
- Optional constant word: valC

Executing Arith./Logical Operation

- OPq rA, rB
- Fetch: Read 2 bytes
- Decode: Read operand registers
- Execute: Perform operation
- Memory: Do nothing
- Write back: Update register
- PC Update: Increment PC by 2

Stage Computation: Arith/Log. Ops

- OPq rA, rB: Read instruction byte
- icode/ifun ← M[PC]: Read register byte
- rA:rB ← M[PC+1]: Compute next PC
- valP ← PC+2: Read operand A
- valA ← R[rA]: Read operand B
- valB ← R[rB]: Perform ALU operation
- valE ← valA OP valB: Set condition code register
- Addr ← valE: Write back result
- valM ← M[PC+2]: Update PC

Stage Computation: rmmovq

- rmmovq rA, D[rB]: Read instruction byte
- icode/ifun ← M[PC]: Read register byte
- rA:rB ← M[PC+1]: Read displacement D
- valA ← M[PC+2]: Read operand A
- valB ← M[PC+10]: Read operand B
- valC ← valA: Compute effective address
- valE ← valB + valC: Write value to memory
- Addr ← valE: Update PC

Stage Computation: rmmovq

- rmmovq rA, D[rB]: Read instruction byte
- icode/ifun ← M[PC]: Read register byte
- rA:rB ← M[PC+1]: Read displacement D
- valA ← M[PC+2]: Read operand A
- valB ← M[PC+10]: Read operand B
- valC ← valA: Compute effective address
- valE ← valB + valC: Write value to memory
- Addr ← valE: Update PC

- Use ALU for address computation
Executing popq

**Stage Computation: popq**

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Read 2 bytes</th>
<th>Memory</th>
<th>Read from old stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>Read stack pointer</td>
<td>Write back</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td>Increment stack pointer by 8</td>
<td>PC Update</td>
<td>Write result to register</td>
</tr>
</tbody>
</table>

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

Stage Computation: Cond. Move

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Read instruction byte</th>
<th>Read register byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>Compute next PC</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td>Read operand A</td>
<td>Increment stack pointer</td>
</tr>
</tbody>
</table>

Execute Conditional Moves

**Stage Computation: cond. move**

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Read 2 bytes</th>
<th>Memory</th>
<th>Do nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>Read operand registers</td>
<td>Write back</td>
<td>Update register (or not)</td>
</tr>
<tr>
<td>Execute</td>
<td>If cond, then set destination register to 0xF</td>
<td>PC Update</td>
<td>Increment PC by 2</td>
</tr>
</tbody>
</table>

Execute Jumps

**Stage Computation: Jumps**

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Read 9 bytes</th>
<th>Increment PC by 9</th>
<th>Memory</th>
<th>Do nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>Do nothing</td>
<td>Write back</td>
<td>Do nothing</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Determine whether to take branch based on jump condition and condition codes</td>
<td>PC Update</td>
<td>Set PC to Dest if branch taken or to incremented PC if not branch</td>
<td></td>
</tr>
</tbody>
</table>

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

- Compute both addresses
- Choose based on setting of condition codes and branch condition
Differ in what gets computed on each step

Increment PC by 1
Read return address
Set PC to destination
Use ALU to increment stack pointer
Set PC to return address
Read 1 byte
valB
M \[Memory read/write\]
valP

Read return address from memory
valC
Read stack pointer
valE
M \[Memory read/write\]
Read operand stack pointer
valE
9
Update stack pointer
valB
valM
Dest
R\[rB\]
valC
CS:APP3e
Write incremented PC to valE
Store incremented PC
PC+2
CS:APP3e
Set PC to return address
Increment stack pointer by 0
valP
rA
M
R\[rA\]
M
Differ in what gets computed on each step
All instructions follow same general pattern
Differ in what gets computed on each step
All instructions follow same general pattern
**Computed Values**

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Execute</th>
<th>Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>opcode</strong></td>
<td>valE</td>
<td>srcA</td>
</tr>
<tr>
<td><strong>ifun</strong></td>
<td>Cnd</td>
<td>srcB</td>
</tr>
<tr>
<td><strong>ra</strong></td>
<td></td>
<td>dstE</td>
</tr>
<tr>
<td><strong>rb</strong></td>
<td></td>
<td>dstM</td>
</tr>
<tr>
<td><strong>valC</strong></td>
<td></td>
<td>valA</td>
</tr>
<tr>
<td><strong>valP</strong></td>
<td></td>
<td>valB</td>
</tr>
</tbody>
</table>

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

**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

**Predefined Blocks**
- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
- Signal invalid address
- Split: Divide instruction byte into iopcode and ifun
- Align: Get fields for ra, rb, and valC

---

**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

---

**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 iopcode and ifun
- Align: Get fields for ra, rb, and valC

---

**Fetch Control Logic in HCL**

```c
# Determine instruction code
int iopcode = [
    imm_error: INOP;
    1: imm_iopcode;
];

# Determine instruction function
int ifun = [
    imm_error: FNONE;
    1: imm_ifun;
];
```

```c
bool need_regids = iopcode in { IMMFOVQ, IOPOQ, IPUSHQ, IPPOPQ, 
    IMMFOVQ, IMMFOVQ, IMMFOVQ, CS:APP3e
};

bool instrs_valid = iopcode in { INOP, IHALT, IMMFOVQ, IMMFOVQ, IMMFOVQ, IMMFOVQ, CS:APP3e
};```
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

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

Memory Address

- **OPq RA, RB**
  - No operation
- **rmmovq RA, D(rB)**
  - Write value to memory
- **popq RA**
  - Read from stack
- **valA = M[valA]**
  - Read return value on stack
- **valM = M[valA]**
  - Read return address

Memory Read

- **OPq RA, RB**
  - No operation
- **rmmovq RA, D(rB)**
  - Write value to memory
- **popq RA**
  - Read from stack
- **call Dest**
  - No operation
- **valA = M[valA]**
  - Read return value on stack
- **valM = M[valA]**
  - Read return address

PC Update Logic

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

PC Update

- **Update PC**
- **Update PC**
- **Update PC**
- **Update PC**
- **Set PC to destination**
- **Set PC to return address**

## Determine instruction status

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

## Instruction status

```c
int mem_addr = 
  code in { IMRMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
  code in { IPOPQ, IRET } : valA;
  # Other instructions don't need address
};
```

## Memory Read

```c
bool mem_read = code in { IMRMOVQ, IPOPQ, IRET };
```
SEQ Operation

State
- PC register
- Cond. Code register
- Data memory
- Register file
All updated as clock rises

Combination 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 irmovq 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 jne 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