Computer Architecture: Sequential Implementation

CS:APP3e

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 Format
  - Instruction byte: (icode:ifun)
  - Optional register byte: rA:rB
  - Optional constant word: valC

**Executing Arith./Logical Operation**

- **Fetching**
  - Read 2 bytes

- **Decoding**
  - Read operand registers

- **Execution**
  - Perform operation
  - Set condition codes

**Stage Computation: Arith/Log. Ops**

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

**Executing rmmovq**

- **Fetching**
  - Read 10 bytes

- **Decoding**
  - Read operand registers

- **Execution**
  - Compute effective address
  - Increment PC by 10

**Stage Computation: rmmovq**

- Use ALU for address computation
**Executing popq**

<table>
<thead>
<tr>
<th>Popq rA</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read 2 bytes</td>
<td>Memory</td>
<td>Read from old stack pointer</td>
<td>Write back</td>
</tr>
<tr>
<td>Decode</td>
<td>Read stack pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Increment stack pointer by 8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stage Computation: popq**

- **Fetch**
  - Read instruction byte
  - Read register byte
- **Decode**
  - Compute next PC
  - Read stack pointer
  - Read stack pointer
- **Execute**
  - Print stack pointer
  - Increment stack pointer by 8
- **Memory**
  - Read from stack
  - Update stack pointer
  - Write back
  - Update PC
- **Write back**
  - Update stack pointer
  - Write result to register
- **PC Update**
  - Increment PC by 2

**Executing Conditional Moves**

<table>
<thead>
<tr>
<th>cmovXX rA, rB</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read 2 bytes</td>
<td>Memory</td>
<td>Do nothing</td>
<td>Write back</td>
</tr>
<tr>
<td>Decode</td>
<td>Read operand registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>If cond, then set destination register to 0xF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stage Computation: Cond. Move**

- **Fetch**
  - Read instruction byte
  - Read register byte
- **Decode**
  - Do nothing
- **Execute**
  - Pass valA through ALU
  - Disallow register update
- **Memory**
  - Do nothing
  - Write back
  - Update PC
- **Write back**
  - Write back result
  - Update PC
- **PC Update**
  - Set PC to Dest if branch taken or to incremented PC if not branch

**Executing Jumps**

<table>
<thead>
<tr>
<th>jXX Dest</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read 9 bytes</td>
<td>Memory</td>
<td>Do nothing</td>
<td>Write back</td>
</tr>
<tr>
<td>Decode</td>
<td>Increment PC by 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Determine whether to take branch based on jump condition and condition codes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stage Computation: Jumps**

- **Fetch**
  - Read instruction byte
  - Read destination address
  - Fall through address
- **Decode**
  - Do nothing
- **Execute**
  - Take branch?
- **Memory**
  - Do nothing
- **Write back**
  - Write back result
- **PC Update**
  - Update PC
- **Update PC**
  - Compute both addresses
  - Choose based on setting of condition codes and branch condition

**Conditional Moves**

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

**Jumps**

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
- If condition codes & move condition indicate no move
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
- vaIA: Register value A
- vaIB: Register value B

**Execute**
- vaIE: ALU result
- Cnd: Branch/move flag
- valM: Value from memory

**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 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;
};
```

**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?
**Decode Logic**

- 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

---

**ALU A Input**

- Oper
- Write
- OP
- Write
- Pass valA through ALU
- Compute effective address
- Increment stack pointer
- Decrement stack pointer
- No operation

**ALU Operation**

- OP
- Write
- valA = valB
- Compute effective address
- Increment stack pointer
- Decrement stack pointer
- No operation

int alufun = {
  // ALU instructions
};

---

**A Source**

- Decode
- Read operand A
- Read operand B
- Read stack pointer
- No operand
- Read operand A
- Read stack pointer

int srcA = {
  // ALU sources
};

---

**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

**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

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
```c
int Stat =
  instr_valid || dmem_error ? SADR;
  ! instr_valid ? SINS;
  instr_valid == IHALT ? SHLT;
  1 : SAOK;
```

Memory Address

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

Memory Read

```
bool mem_read = icode in { IMMOVQ, IPOPQ, IRET };
```

PC Update Logic

```
int new_pc =
  icode == ICALL : valC;
  icode == IJXX && Cnd : valC;
  icode == IRET : valM;
  1 : valP;
```

PC Update

```
int new_pc =
  icode == ICALL : valC;
  icode == IJXX && Cnd : valC;
  icode == IRET : valM;
  1 : valP;
```

Data

Mem.
read
write
addr
data
valE
valM
valA
valP
Call
Return
return
Read
Write
PC
Update
set
Cnd
valC
valM
valP
Val
IPush
IRmm
pop
j
IRET
ICALL
IMRMOVQ
IPUSHQ
IMMOVQ
IPOPQ
IRmmovq
rA
rB
D(rB)
D(rA)
rA
rB
rA
rB
rA
rB
rA
rB
rA
rB
rA
rB
rA
rB
rA
rB
rA
rB
rA
rB
rA
rB
Memory reads
0x300
$0x200
0x200
Data
= 0x100

Cycle 4
000
CS:APP3e

Data
Cycle 4 = 0x300
$0x100

Would need to run clock very slowly
# Too slow to be practical

Write
CS:APP3e

Read
%rbx
Write
%rbx
Read
%rbx

# Not taken
state set
Read
Data
Write
Data
0x300
Write
0x200
CS:APP3e

In one cycle, must propagate through instruction memory,
state set
Data
Write
Data
%rbx
PC register
= 0x300
Cycle 4
CS:APP3e

Hardware units only active for fraction of clock cycle

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 *movq* instruction
- combinational logic starting to react to state changes

SEQ Operation #3
- state set according to second *movq* 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