CSC 252: Computer Organization
Spring 2020: Lecture 12

Processor Architecture

Substitute Instructor: Sandhya Dwarkadas

Department of Computer Science
University of Rochester
Announcement

• Programming assignment 3 due soon
  • Details: https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment3.html
  • Due on **Feb. 28, 11:59 PM**
  • You (may still) have 3 slip days
Announcement

• Grades for lab2 are posted.
• If you think there are some problems
  • Take a deep breath
  • Tell yourself that the teaching staff like you, not the opposite
  • Email/go to Shuang or Sudhanshu’s office hours and explain to them why you should get more points, and they will fix it for you
Announcement

• Programming assignment 3 is in x86 assembly language. Seek help from TAs.
• TAs are best positioned to answer your questions about programming assignments!!!
• Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.
Overview of Logic Design

Fundamental Hardware Requirements

- Communication
  - How to get values from one place to another
- Computation
- Storage

Bits are Our Friends

- Everything expressed in terms of values 0 and 1
- Communication
  - Low or high voltage on wire
- Computation
  - Compute Boolean functions
- Storage
  - Store bits of information
Combinational Circuits

Acyclic Network of Logic Gates

- Continuously responds to changes on primary inputs
- Primary outputs become (after some delay) Boolean functions of primary inputs
Arithmetic Logic Unit

- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86
- Also computes values for condition codes
Sequential Logic: Memory and Control

Sequential:

■ **Output** depends on the *current* input values and the *previous* sequence of input values.

■ **Are Cyclic:**
  - Output of a gate feeds its input at some future time.

■ **Memory:**
  - Remember results of previous operations
  - Use them as inputs.

■ **Example of use:**
  - Build registers and memory units.
Registers

- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal
Register Operation

- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you’d better compute the input before the C signal rises if you want to store the input data to the register

Output *continuously* produces y after the rising edge unless you cut off power.
Clock Signal

• A special C: periodically oscillating between 0 and 1
• That’s called the **clock** signal. Generated by a crystal oscillator inside your computer
Clock Signal

• Cycle time of a clock signal: the time duration between two rising edges.
• Frequency of a clock signal: how many rising (falling) edges in 1 second.
• 1 GHz CPU means the clock frequency is 1 GHz
  • The cycle time is $1/10^9 = 1$ ns
A register file consists of a set of registers that you can individual read from and write to. To read: give a register file ID, and read the stored value out. To write: give a register file ID, a new value, overwrite the old value. How do we build a register file out of individual registers?
Register File Read

• Continuously read a register independent of the clock signal

```
  Register 0
  Register 1
  Register 2
  Register 3
```

4:1 MUX
Register File Write

- Only write the a specific register when the clock rises. How??

Clock

Write Reg ID

Write Reg ID

Data

4:1 MUX

Register 0

Register 1

Register 2

Register 3

C0

D

Cl

D

C2

D

C3

D
### Decoder

<table>
<thead>
<tr>
<th>W1</th>
<th>W0</th>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W0</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>_ C0</td>
</tr>
<tr>
<td>1</td>
<td>_ C1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W1</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>_ C1</td>
</tr>
<tr>
<td>1</td>
<td>_ C2</td>
</tr>
</tbody>
</table>

C0 = !W1 & !W0
C1 = !W1 & W0
C2 = W1 & !W0
C3 = W1 & W0
Register File Write

- This implementation can read 1 register and write 1 register at the same time: 1 read port and 1 write port
Multi-Port Register File

• What if we want to read multiple registers at the same time?

• This register file has 2 read ports and 1 write port. How many ports do we actually need?
Multi-Port Register File

- Is this correct? What if we don’t want to write anything?
Register File

- Stores multiple registers of data
  - Address input specifies which register to read or write
- Register file is a form of Random-Access Memory (RAM)
- Multiple Ports: Can read and/or write multiple words in one cycle. Each port has separate address and data input/output
State Machine Example

- Accumulator circuit
- Load or accumulate on each cycle
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
Arithmetic and Logical Operations

- Refer to generically as “OPq”
- Encodings differ only by “function code”
  - Low-order 4 bits in first instruction word
- Set condition codes as side effect

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td></td>
</tr>
<tr>
<td>addq rA, rB</td>
<td>6 0 rA rB</td>
</tr>
<tr>
<td>Subtract (rA from rB)</td>
<td></td>
</tr>
<tr>
<td>subq rA, rB</td>
<td>6 1 rA rB</td>
</tr>
<tr>
<td>And</td>
<td></td>
</tr>
<tr>
<td>andq rA, rB</td>
<td>6 2 rA rB</td>
</tr>
<tr>
<td>Exclusive-Or</td>
<td></td>
</tr>
<tr>
<td>xorq rA, rB</td>
<td>6 3 rA rB</td>
</tr>
</tbody>
</table>
Executing an ADD instruction

- How does the processor execute `addq %rax,%rsi`?
- The binary encoding is `60 06`.

![Diagram showing the instruction encoding and fields](Diagram.png)
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is `60 06`.

![Diagram of the processor pipeline](image)

- **Instruction Code**: Add
- **Function Code**: `6 0 rA rB`
- **Memory (Later…)**
- **Register File**
  - Write Reg. ID
  - Read Reg. ID 1
  - Read Reg. ID 2

![Registers](image)
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?
Executing an ADD instruction

- When the rising edge of the clock arrives, the RF/PC/Flags will be written.
- So the following has to be ready: newData, nPC, which means Logic1, Logic2, Logic3, and Logic4 has to finish.
Executing a JLE instruction

- Let’s say the binary encoding for `jle .L0` is 71 0123000000000000
- What are the logics now?

![Diagram showing the execution of a JLE instruction with logics and data flow.]
Executing a JLE instruction

- Logic 1: if \( s0 == 6 \) select = s1;
- Logic 2: if \( s0 == 6 \) Enable = 1; else Enable = 0;
Executing a JLE instruction

- Logic 3??

```c
if (s0 == 6) nPC = oPC + 2;
else if (s0 == 7) {
    if (s1 == 1) { // jLE
        if (Z || (S ^ O)) nPC = Dest; // jump
        else nPC = oPC + 9; // don’t jump, but add 9 (why??)
    } else if (s1 == …) {…}
}
```

```
jle Dest
[7 1]
```
Executing a JLE instruction

- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;
Combinational Logic

- Logic for generating ALU select signal
- Logic for generating new PC value
- Logic for generating new flag value
- Logic for deciding all the enable signal values

Read current\_states;
next\_states = f(current\_states);
When clock rises, current\_states = next\_states;
Executing a MOV instruction

How do we modify the hardware to execute a move instruction?

```
rmmovq rA, D(rB)
```

move rA to the memory address rB + D

```
rmmovq %rsi,0x41c(%rsp)
```

40 64 1c 04 00 00 00 00 00 00 00 00
move rA to the memory address rB + D

rmmovq rA, D(rB)

Need new logic (Logic 6) to select the input to the ALU for Enable.

How about other logics?
How About Memory to Register MOV?

move data at memory address rB + D to rA

mrmovq D(rB), rA

Data to write
Enable
Address

Logic 6
Logic 1
Logic 4
Logic 5
Logic 2
Logic 3
Memory
Register File

Data to write
Enable

PC
nPC
oPC

Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2

newData

[s4…s19]

MUX

RA
RB

Select

Flags [s2…s9]

Address

Clock

4 0 rA rB

D

Move data at memory address rB + D to rA

How About Memory to Register MOV?

Data to write
Enable

Logic 6
Logic 1
Logic 4
Logic 5
Logic 2
Logic 3
Memory
Register File

Data to write
Enable

PC
nPC
oPC

Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2

newData

[s4…s19]

MUX

RA
RB

Select

Flags [s2…s9]

Address

Clock

4 0 rA rB

D

Move data at memory address rB + D to rA

How About Memory to Register MOV?
How About Memory to Register MOV?

move data at memory address \( rB + D \) to \( rA \)

```plaintext
mrmovq  D(rB),  rA
```

![Diagram](image-url)
Microarchitecture (with MOV)

Combinational Logic

Read `current_states`;
`next_states = f(current_states)`;
When clock rises, `current_states = next_states`;
`next_states` has to be ready before the close rises
Microarchitecture Overview

Think of it as a state machine

Every cycle, one instruction gets executed. At the end of the cycle, architecture states get modified.

States (All updated as clock rises)

- PC register
- Cond. Code register
- Data memory
- Register file
state set according to second `irmovq` instruction
combinational logic starting to react to state changes
state set according to second \texttt{irmovq} instruction

combinational logic generates results for \texttt{addq} instruction
### Cycle 1:
0x000: `irmovq $0x100,%rbx` # %rbx <-- 0x100

### Cycle 2:
0x00a: `irmovq $0x200,%rdx` # %rdx <-- 0x200

### Cycle 3:
0x014: `addq %rdx,%rbx` # %rbx <-- 0x300 CC <-- 000

### Cycle 4:
0x016: `je dest` # Not taken

### Cycle 5:
0x01f: `rmmovq %rbx,0(%rdx)` # M[0x200] <-- 0x300

---

**Combinational logic**

- **Data memory**
- **Register file**:
  - %rbx = 0x300
- **PC**:
  - 0x016

**Clock**

- **Cycle 1**
- **Cycle 2**
- **Cycle 3**
- **Cycle 4**
- **Cycle 5**

**State set according to `addq` instruction**

**Combinational logic starting to react to state changes**
state set according to `addq` instruction
combinational logic generates results for `je` instruction

| Cycle 1: | 0x000: irmovq $0x100,%rbx # %rbx <-- 0x100 |
| Cycle 2: | 0x00a: irmovq $0x200,%rdx # %rdx <-- 0x200 |
| Cycle 3: | 0x014: addq %rdx,%rbx # %rbx <-- 0x300 CC <-- 000 |
| Cycle 4: | 0x016: je dest # Not taken |
| Cycle 5: | 0x01f: rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300 |