CSE141: Introduction to Computer Architecture

**Instructor:** Pramod V. Argade (p2argade@cs.ucsd.edu)
Office Hour:
- Tue. 7:30 - 9:00 (Center 105)
- Wed. 5:00 - 6:00 PM (HSS 1330)
- By Appointment

**TAs:**
- Chengmo Yang: c5yang@cs.ucsd.edu
- Wenjing Rao: wrao@cs.ucsd.edu

**Lecture:** Mon/Wed. 6:00 - 8:50 PM, HSS 1330

**Textbook:** Computer Organization & Design
The Hardware Software Interface, 3\textsuperscript{rd} Edition.
Authors: Patterson and Hennessy

**Web-page:** [http://www.cse.ucsd.edu/classes/su05/cse141](http://www.cse.ucsd.edu/classes/su05/cse141)
Announcements

• **Reading Assignment**
  – Chapter 5. The Processor: Datapath and Control
    Sections 5.1 - 5.5

• **Homework 4: Due Mon., July 18th in class**
  5.1, 5.3, 5.8, 5.9, 5.10, 5.11, 5.13, 5.20, 5.22, 5.28
  5.32, 5.33, 5.34, 5.35, 5.36, 5.41, 5.42

• **Quiz**
  **When:** Wed., July 13th, First 10 minutes of the class
  **Topic:** Single Cycle CPU, Chapter 5  **Need:** Paper, pen
<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Time</th>
<th>Room</th>
<th>Topic</th>
<th>Quiz topic</th>
<th>Homework Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mon. 6/27</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Introduction, Ch. 1 ISA, Ch. 2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Wed. 6/29</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Arithmetic, Ch. 3</td>
<td>ISA Ch. 2</td>
<td>#1</td>
</tr>
<tr>
<td></td>
<td>Mon. 7/4</td>
<td>No Class</td>
<td></td>
<td>July 4th Holiday</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Wed. 7/6</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Performance, Ch. 4 Single-cycle CPU Ch. 5</td>
<td>Arithmetic Ch. 3</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>Mon. 7/11</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Single-cycle CPU Ch. 5 Cont. Multi-cycle CPU Ch. 5</td>
<td>Performance Ch. 4</td>
<td>#3</td>
</tr>
<tr>
<td>5</td>
<td>Tue. 7/12</td>
<td>7:30 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Multi-cycle CPU Ch. 5 Cont. (July 4th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Wed. 7/13</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Single and Multicycle CPU Examples and Review for Midterm</td>
<td>Single-cycle CPU Ch. 5</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Mon. 7/18</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Mid-term Exam Exceptions</td>
<td>-</td>
<td>#4</td>
</tr>
<tr>
<td>8</td>
<td>Tue. 7/19</td>
<td>7:30 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Pipelining Ch. 6 (July 4th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Wed. 7/20</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Hazards, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Mon. 7/25</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Memory Hierarchy &amp; Caches Ch. 7</td>
<td>Hazards Ch. 6</td>
<td>#5</td>
</tr>
<tr>
<td>11</td>
<td>Wed. 7/27</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Virtual Memory, Ch. 7 Course Review</td>
<td>Cache Ch. 7</td>
<td>#6</td>
</tr>
<tr>
<td>12</td>
<td>Sat. 7/30</td>
<td>TBD</td>
<td>TBD</td>
<td>Final Exam</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Multi-cycle CPU: Control FSM

start

Instruction fetch

Decode and Register Fetch

Memory instructions
R-type instructions
Branch instructions
Jump instruction
State 1. Instruction Fetch Cycle
(Instruction Independent)

IR = Memory[PC]

PC = PC + 4  *(may not be final value of PC)*
State 2. Instruction Decode and Reg. Fetch Cycle
(Instruction Independent)

A = Register[IR[25-21]]
B = Register[IR[20-16]]

ALUOut = PC + (sign-extend (IR[15-0]) << 2)  [May not be used]
First two states of the FSM

**Instruction Fetch, state 0**
IR = Memory[PC]
PC = PC + 4

**Instruction Decode/ Register Fetch, state 1**
A = Register[IR[25-21]]
B = Register[IR[20-16]]
ALUOut = PC + (sign-extend (IR[15-0]) << 2)
State 3. R-type Execution

\[ \text{ALUOut} = A \text{ op } B \]
State 4. R-type completion

Reg[IR[15-11]] = ALUout
States for R-type Instructions

from state 1

Execution

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

Completion

RegDst = 1
RegWrite
MemtoReg = 0

To state 0
State 3. Branch completion

if (A == B)  PC = ALUOut
State for BEQ Instruction

From state 1

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

To state 0
State 3. Memory address computation

\[ \text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]) \]
State 4. Memory access

Load: \( MDR = \text{Memory}[\text{ALUout}] \)

Store: \( \text{Memory}[\text{ALUout}] = B \) [Completes in this cycle]
State 5. Memory Write-Back

Reg[IR[20-16]] = MDR  [Completes in this cycle]
States for Memory Instructions

From state 1

Address

Computation

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

MemRead
IorD = 1

MemWrite
IorD = 1

RegWrite
MemtoReg = 1
RegDst = 0

write-back

To state 0

Memory

Access

Pramod Argade
UCSD CSE 141, Summer Session II, 2005
Slide 5-17
State 2: Jump Instruction Completion

PC = PC[31-28] | (inst[25-0] <<2)
JMP Instruction

From state 1

PCWrite
PCSource = 10

To state 0
Complete FSM

Instruction fetch
- PCSource = 10
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 01
- PCWriteCond

Instruction decode/register fetch
- PCSource = 01
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 10
- RegDst = 1
- RegWrite
- MemtoReg = 0

Instruction decode/ register fetch
- PCSource = 00
- ALUSrcA = 0
- ALUSrcB = 11
- ALUOp = 00
- MemRead
- IorD = 1
- IRWrite

Memory address computation
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00

Execution
- (Op = 'LW') or (Op = 'SW')
- (Op = 'R-type')
- (Op = 'BEQ')
- (Op = 'J')

Branch completion
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 01
- PCWriteCond
- PCSource = 01

Jump completion
- PCWrite
- PCSource = 10

Memory access
- MemRead
- IorD = 1

Memory access
- MemWrite
- IorD = 1
- MemtoReg = 0

R-type completion
- RegDst = 1
- RegWrite

Write-back step
- RegWrite
- MemtoReg = 1
Simple Questions

- How many cycles will it take to execute this code?

```
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label  #assume not taken
add $t5, $t2, $t3
sw $t5, 8($t3)
Label: ...
```

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of $t2 and $t3 takes place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?
Simple Questions

● How many cycles will it take to execute this code? 21 clocks

5  lw $t2, 0($t3)
5  lw $t3, 4($t3)
3  beq $t2, $t3, Label  #assume not taken
4  add $t5, $t2, $t3
4  sw $t5, 8($t3)
Label: ...

● What is going on during the 8th cycle of execution?
  – Phase 3 of second lw, memory address computation

● In what cycle does the actual addition of $t2 and $t3 takes place?
  – 3rd phase of add: 16th clock cycle

● Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?
  – CPI = 0.2*5 + 0.1*4 + 0.5*4 + 0.2*3 = 4.0
Multicycle CPU Key Points

- Performance gain achieved from variable-time instructions
  - Reduces the cycle time

- Execution Time = #Instructions * CPI * cycle time
  - Average CPI is reduced and so is cycle time

- Required very few new state elements

- More complex control signals

- Control requires FSM
### Logic Equations for Control Unit

<table>
<thead>
<tr>
<th>Output</th>
<th>Current State</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCWrite</td>
<td>state0 + state9</td>
<td></td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>state8</td>
<td></td>
</tr>
<tr>
<td>IorD</td>
<td>state3 + state5</td>
<td></td>
</tr>
<tr>
<td>MemRead</td>
<td>state0 + state3</td>
<td></td>
</tr>
<tr>
<td>MemWrite</td>
<td>state5</td>
<td></td>
</tr>
<tr>
<td>IRWrite</td>
<td>state0</td>
<td></td>
</tr>
<tr>
<td>MemtoReg</td>
<td>state4</td>
<td></td>
</tr>
<tr>
<td>PCSource1</td>
<td>state9</td>
<td></td>
</tr>
<tr>
<td>PCSource0</td>
<td>state8</td>
<td></td>
</tr>
<tr>
<td>ALUOp1</td>
<td>state6</td>
<td></td>
</tr>
<tr>
<td>ALUOP0</td>
<td>state8</td>
<td></td>
</tr>
<tr>
<td>ALUSrcB1</td>
<td>state1 + state2</td>
<td></td>
</tr>
<tr>
<td>ALUSrcB0</td>
<td>state0 + state1</td>
<td></td>
</tr>
<tr>
<td>ALUSrcA</td>
<td>state2 + state6 + state8</td>
<td></td>
</tr>
<tr>
<td>RegWrite</td>
<td>state4 + state7</td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>state7</td>
<td></td>
</tr>
<tr>
<td>NextState0</td>
<td>state4 + state5 + state7 + state8 + state9</td>
<td></td>
</tr>
<tr>
<td>NextState1</td>
<td>state0</td>
<td></td>
</tr>
<tr>
<td>NextState2</td>
<td>state1</td>
<td>(Op='lw') + (Op='sw')</td>
</tr>
<tr>
<td>NextState3</td>
<td>state2</td>
<td>(Op='lw')</td>
</tr>
<tr>
<td>NextState4</td>
<td>state3</td>
<td>(Op='sw')</td>
</tr>
<tr>
<td>NextState5</td>
<td>state2</td>
<td>(Op='sw')</td>
</tr>
<tr>
<td>NextState6</td>
<td>state1</td>
<td>(Op='R-type')</td>
</tr>
<tr>
<td>NextState7</td>
<td>state6</td>
<td>(Op='beq')</td>
</tr>
<tr>
<td>NextState8</td>
<td>state1</td>
<td>(Op='jmp')</td>
</tr>
<tr>
<td>NextState9</td>
<td>state1</td>
<td>(Op='jmp')</td>
</tr>
</tbody>
</table>
Multicycle CPU: Control

```
If (State == Instruction Fetch)
{
    IRWrite = 1;
    // All other signals are 0;
    State = Operand Fetch;
}

If (State == Execute &&
    InstructionOpCode == BEQ )
{
    // Do your thing..
}
```

ControlOutput = f(State, OpCode)
NextState = f(State, OpCode)
Finite State Machine for Control

Control logic

Outputs

Inputs

Instruction register
Opcode field

State register

Op5 Op4 Op3 Op2 Op1 Op0

PCWrite Cond
IorD
MemRead
MemWrite
IRWrite
MerrtoReg
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst

NS3 NS2 NS1 NS0

Pramod Argade
UCSD CSE 141, Summer Session II, 2005
FSM: ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address $2^m$ entries in the ROM.
  - ROM outputs are the bits of data that the address points to
  - ROM outputs are the control and next state signals

$2^m$ is the "height", and n is the "width"
Implementing a control FSM with ROM

Each line in the ROM contains control signal outputs (an operation), and next-state outputs (branch destination)
FSM: ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., $2^{10} = 1024$ different addresses)

- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs

- ROM is $2^{10} \times 20 = 20K$ bits  (and a rather unusual size)

- Very wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored
FSM: Programmed Logic Array (PLA)

Examples:
PCWrite = state0 + state9
PCWriteCond = state8
IorD = state3 + state5
The Problem with FSMs as control sequencers

- They get unmanageable quickly as they grow.
  - Hard to specify
  - Impractical to manipulate
  - Error prone
  - Difficult to visualize

- MIPS-32 instruction set contains over 100 instructions!
  - In one implementation instructions take 1 - 20 cycles
  - Graphical representation would be impractical

- Solution: Microprogramming
  - Uses ideas from programming
  - Think of the set of control signals that must be asserted in a state as an instruction to be executed by the data path.
  - These low level instructions are called “microinstructions”
Implementing a control FSM with a microprogram

Each line in the ROM is now a microprogram instruction, corresponding to a FSM state, with an operation (control signals) and branch destination (next state info).
Microprogramming

- Each microinstruction typically specifies control information
  - Must also specify sequencing information
    - What micro-instruction should be executed next?
- Control signals are grouped into “fields”
  - Make it impossible to write inconsistent microinstruction
- Microcode subroutines reuse code
  - Return address stack is provided in the control unit
- Translated by a program to control logic
- If a microprogram is fundamentally the same as the FSM, what’s the big deal?
  - Easier to specify (program), visualize, and manipulate.
  - Allows us to think about the control symbolically
  - Easier to maintain, fix bugs
  - Attractive choice for large and complex control
# Microinstruction Fields

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALUOp = 00</td>
<td></td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td>Subt</td>
<td>ALUOp = 01</td>
<td></td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td></td>
<td>Use the instruction's function code to determine ALU control.</td>
</tr>
<tr>
<td><strong>SRC1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>ALUSrcA = 0</td>
<td></td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcA = 1</td>
<td></td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td><strong>SRC2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcB = 0</td>
<td></td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td>4</td>
<td>ALUSrcB = 01</td>
<td></td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<td>Extend</td>
<td>ALUSrcB = 10</td>
<td></td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td>Extshft</td>
<td>ALUSrcB = 11</td>
<td></td>
<td>Use the output of the shift-by-two unit as the second ALU input.</td>
</tr>
<tr>
<td><strong>Read</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Write ALU</strong></td>
<td>RegWrite,</td>
<td></td>
<td>Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td>RegDst = 1,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MemtoReg = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Write MDR</strong></td>
<td>RegWrite,</td>
<td></td>
<td>Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.</td>
</tr>
<tr>
<td></td>
<td>RegDst = 0,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MemtoReg = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read PC</td>
<td>MemRead,</td>
<td></td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td></td>
<td>lorD = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemRead,</td>
<td></td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td>lorD = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite,</td>
<td></td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
</tr>
<tr>
<td></td>
<td>lorD = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PC write control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>PCSource = 00</td>
<td></td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td></td>
<td>PCWrite</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUOut-cond</td>
<td>PCSource = 01</td>
<td></td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
</tr>
<tr>
<td></td>
<td>PCWriteCond</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump address</td>
<td>PCSource = 10</td>
<td></td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td></td>
<td>PCWrite</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sequencing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>AddrCtl = 11</td>
<td></td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td>Fetch</td>
<td>AddrCtl = 0</td>
<td></td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
<td></td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
<td></td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>

First six fields control the datapath & the Sequencing field specifies how to select the next microinstruction.

Pramod Argade

UCSD CSE 141, Summer Session II, 2005
Microprogramming

• Selection of next address of microinstruction
  – Sequential (Seq): Increment the current address
  – Branch (Fetch): Branch to the microinstruction that begins next MIPS instruction
  – Dispatch: Select next instruction based on control unit input
    ✔ Implemented by a table indexed by control unit input
    ✔ There may be multiple dispatch tables
A Microprogram

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU control</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch:</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td>Dispatch 1</td>
<td></td>
</tr>
<tr>
<td>Mem1:</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td></td>
<td></td>
<td>Dispatch 2</td>
<td></td>
</tr>
<tr>
<td>LW2:</td>
<td></td>
<td></td>
<td></td>
<td>Read ALU</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>SW2:</td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Rformat1:</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>BEQ1:</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut-cond</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>JUMP1:</td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
</tbody>
</table>

Dispatch Table 1 is used to select one of four microinstruction sequences
- OPCODE field is used to dispatch
- Mem1, Rformat1, BEQ1, JUMP1

Dispatch Table 2 is used to select one of two microinstruction sequences
- OPCODE field is used to dispatch
- LW2, SW2
State 1. Instruction Fetch Cycle (Instruction Independent)
Microprogram Implementation
Multi-cycle CPU: Summary

- Instructions take variable number of cycles to complete
  - Reduces (CPI * Cycle time)
  - Improves performance
- Efficient use of HW elements
  - Blocks are reused
- Control is harder
- Possible to make “common case” faster
  - Improved implementation techniques help
  - Unlike single-cycle CPU
- Microprogramming can simplify (conceptually) CPU control generation
  - A microprogram is a small program inside the CPU that executes the individual instructions of the “real” program.
Announcements

- **Reading Assignment**
  - Chapter 5. The Processor: Datapath and Control
    Sections 5.1 - 5.5

- **Homework 4: Due Mon., July 18th in class**
  5.1, 5.3, 5.8, 5.9, 5.10, 5.11, 5.13, 5.20, 5.22, 5.28
  5.32, 5.33, 5.34, 5.35, 5.36, 5.41, 5.42

- **Quiz**
  **When:** Wed., July 13th, First 10 minutes of the class
  **Topic:** Single Cycle CPU, Chapter 5
  **Need:** Paper, pen