

---

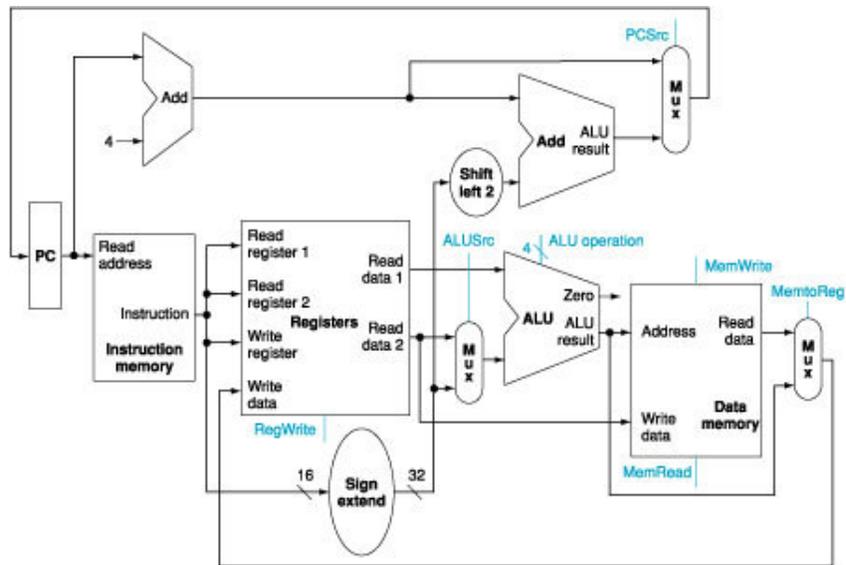
# SI232

## Set #15: Multicycle Implementation (Chapter Five)

1

### Recall – Single Cycle Implementation

---



2

## Evaluation – Single Cycle Approach

---

- **Good:**

- **Bad:**

3

## Multicycle Approach

---

- **Break up the instructions into steps, each step takes a cycle**
  - **balance the amount of work to be done**
  - **restrict each cycle to use only one major functional unit:**

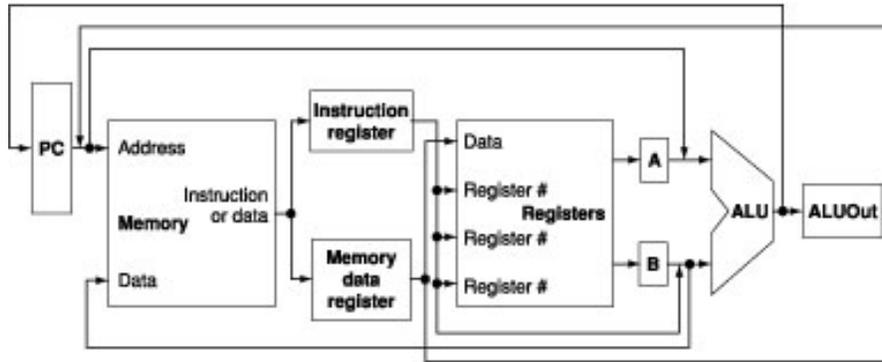
- **At the end of a cycle**
  - **store values for use in later cycles**
  - **introduce additional “internal” registers**

- **Each instruction will take \_\_\_\_\_ cycles to fully execute**

4

## Simplified Multicycle Datapath

---



5

## Breaking down an instruction

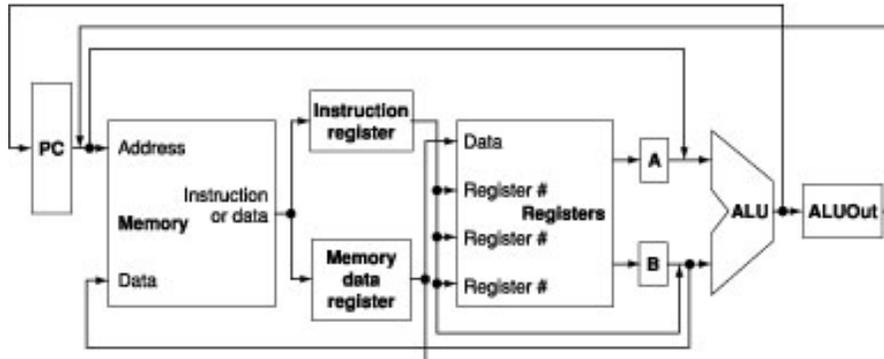
---

- Steps for an R-type instruction:
  - $IR \leftarrow Memory[PC]$
  - $A \leftarrow Reg[IR[25:21]]$
  - $B \leftarrow Reg[IR[20:16]]$
  - $ALUOut \leftarrow A \text{ op } B$
  - $Reg[IR[15:11]] \leftarrow ALUOut$
- What did we forget?
- Above notation is called RTL – Register Transfer Language

6

### Example #1 – sub \$t0, \$s1, \$s2

---

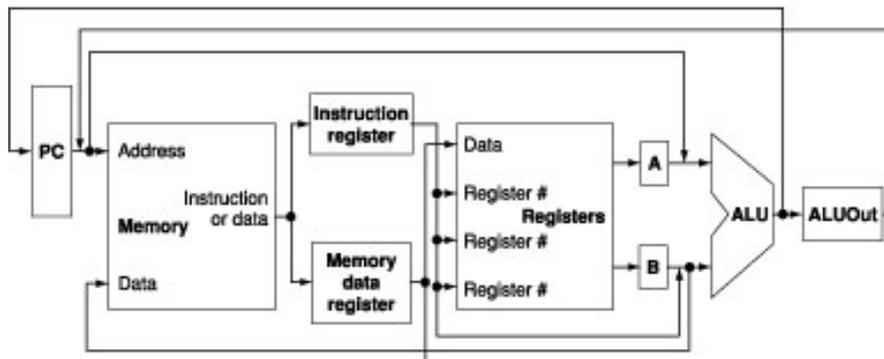


1.  $IR \leftarrow Memory[PC]$
2.  $A \leftarrow Reg[IR[25:21]]$
3.  $B \leftarrow Reg[IR[20:16]]$
4.  $ALUOut \leftarrow A \text{ op } B$
5.  $Reg[IR[15:11]] \leftarrow ALUOut$
6.  $PC \leftarrow PC + 4$

7

### Example #2 – lw \$t0, 8(\$s2)

---



1.  $IR \leftarrow Memory[PC]$
2.  $A \leftarrow Reg[IR[25:21]]$
3.  $ALUOut \leftarrow A + \text{sign-extend}(IR[15:0])$
4.  $MDR = Memory[ALUOut]$
5.  $Reg[IR[20:16]] = MDR$
6.  $PC \leftarrow PC + 4$

8

## How many cycles do we need?

---

In once cycle can do: Register read or write, memory access, ALU

a.) Fill in the cycle number for each task below

<u>Cycle #</u>	<u>Task (for R-type instruction)</u>
	<b>IR &lt;= Memory[PC]</b>
	<b>A &lt;= Reg[IR[25:21]]</b>
	<b>B &lt;= Reg[IR[20:16]]</b>
	<b>ALUOut &lt;= A op B</b>
	<b>Reg[IR[15:11]] &lt;= ALUOut</b>
	<b>PC &lt;= PC + 4</b>

b.) What is the total number of cycles needed?

9

## Exercise #1: How many cycles do we need?

---

In once cycle can do: Register read or write, memory access, ALU

a.) Fill in the cycle number for each task below

<u>Cycle #</u>	<u>Task (for <b>load</b> instruction)</u>
	<b>IR &lt;= Memory[PC]</b>
	<b>A &lt;= Reg[IR[25:21]]</b>
	<b>ALUOut &lt;= A + sign-extend(IR[15:0])</b>
	<b>MDR = Memory[ALUOut]</b>
	<b>Reg[IR[20:16]] = MDR</b>
	<b>PC &lt;= PC + 4</b>

b.) What is the total number of cycles needed?

10

## Exercise #2: How many cycles do we need?

---

In once cycle can do: Register read or write, memory access, ALU

a.) Fill in the cycle number for each task below

<u>Cycle #</u>	<u>Task (for <b>store</b> instruction)</u>
	<b>IR &lt;= Memory[PC]</b>
	<b>A &lt;= Reg[IR[25-21]]</b>
	<b>B &lt;= Reg[IR[20-16]]</b>
	<b>ALUOut &lt;= A + sign-extend(IR[15-0])</b>
	<b>Memory[ALUOut] = B</b>
	<b>PC &lt;= PC + 4</b>

b.) What is the total number of cycles needed?

11

## Exercise #3: How many cycles do we need?

---

In once cycle can do: Register read or write, memory access, ALU

a.) Fill in the cycle number for each task below

<u>Cycle #</u>	<u>Task (for <b>branch</b> instruction)</u>
	<b>IR &lt;= Memory[PC]</b>
	<b>PC &lt;= PC + 4</b>
	<b>A &lt;= Reg[IR[25-21]]</b>
	<b>B &lt;= Reg[IR[20-16]]</b>
	<b>ALUOut &lt;= PC + (sign-extend(IR[15-0]) &lt;&lt; 2)</b>
	<b>if (A ==B) PC = ALUOut</b>

b.) What is the total number of cycles needed?

12

## Exercise #4

---

- The branch instruction from Exercise #3 can't really be executed given our simple datapath – why not?

13

## Multicycle Implementation

---

- Goals:
  - Pack as much work into each step as possible
  - Share steps across different instruction types
- 5 Steps
  1. Instruction Fetch
  2. Instruction Decode and Register Fetch
  3. Execution, Memory Address Computation, or Branch Completion
  4. Memory Access or R-type instruction completion
  5. Write-back step

14

## Step 1: Instruction Fetch

---

```
IR <= Memory[PC];
```

```
PC <= PC + 4;
```

*What is the advantage of updating the PC now?*

15

## Step 2: Instruction Decode and Register Fetch

---

- Read registers rs and rt

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
```
- Compute the branch address

```
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```
- Does this depend on the instruction type?
- Could it depend on the instruction type?

16

### Step 3 (instruction dependent)

---

- ALU function depends on instruction type
- 1. \_\_\_\_\_  
`ALUOut <= A + sign-extend(IR[15:0]);`
- 2. \_\_\_\_\_  
`ALUOut <= A op B;`
- 3. \_\_\_\_\_  
`if (A==B) PC <= ALUOut;`

17

### Step 4 (R-type or memory-access)

---

- Loads and stores access memory  
`MDR <= Memory[ALUOut];`  
or  
`Memory[ALUOut] <= B;`
- R-type instructions finish  
`Reg[IR[15:11]] <= ALUOut;`

*The write actually takes place at the end of the cycle on the edge*

18

## Step 5: Write-back

---

- $\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR};$

*Which instruction needs this?*

19

## Summary:

---

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$\text{IR} \leftarrow \text{Memory}[\text{PC}]$ $\text{PC} \leftarrow \text{PC} + 4$			
Instruction decode/register fetch	$\text{A} \leftarrow \text{Reg}[\text{IR}[25:21]]$ $\text{B} \leftarrow \text{Reg}[\text{IR}[20:16]]$ $\text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[15:0]) \ll 2)$			
Execution, address computation, branch/jump completion	$\text{ALUOut} \leftarrow \text{A op B}$	$\text{ALUOut} \leftarrow \text{A} + \text{sign-extend}(\text{IR}[15:0])$	If (A == B) $\text{PC} \leftarrow \text{ALUOut}$	$\text{PC} \leftarrow \{\text{PC}[31:26], \text{IR}[25:0], 2'b00\}$
Memory access or R-type completion	$\text{Reg}[\text{IR}[15:11]] \leftarrow \text{ALUOut}$	Load: $\text{MDR} \leftarrow \text{Memory}[\text{ALUOut}]$ or Store: $\text{Memory}[\text{ALUOut}] \leftarrow \text{B}$		
Memory read completion		Load: $\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR}$		

**FIGURE 5.30 Summary of the steps taken to execute any instruction class.** Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

20

## 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:   ...
```

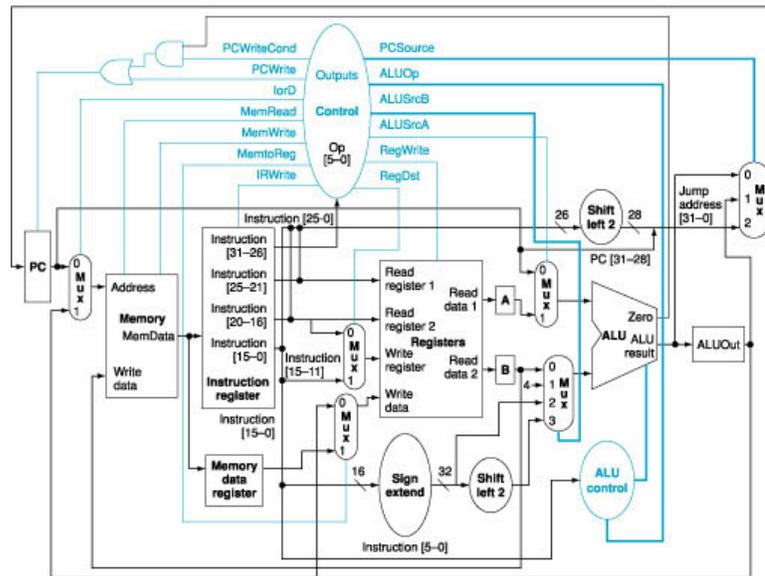
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of \$t2 and \$t3 takes place?

21

---

## Control for Multicycle Implementation

22



Control for “sub \$t0, \$s1, \$s2”

ALUSrcA =

ALUSrcB =

## Multicycle Control

- Control for single cycle implementation was \_\_\_\_\_ ,  
based only on the \_\_\_\_\_
- Control for multicycle implementation will be \_\_\_\_\_ ,  
based on the \_\_\_\_\_ and current \_\_\_\_\_
- We'll implement this control with state machines

## Two Weird Things

1. For enable signals (RegWrite, MemRead, etc.) we'll write down the signal only if it is true.  
For multiplexors (ALUSrcA, IorD, etc.) , we'll always say what the value is. (unless it's a "don't care")
2. Some registers are written every cycle, so no write enable control for them (MDR, ALUOut).  
Others have explicit control (register file, IR)

Random (but useful) Refresher:

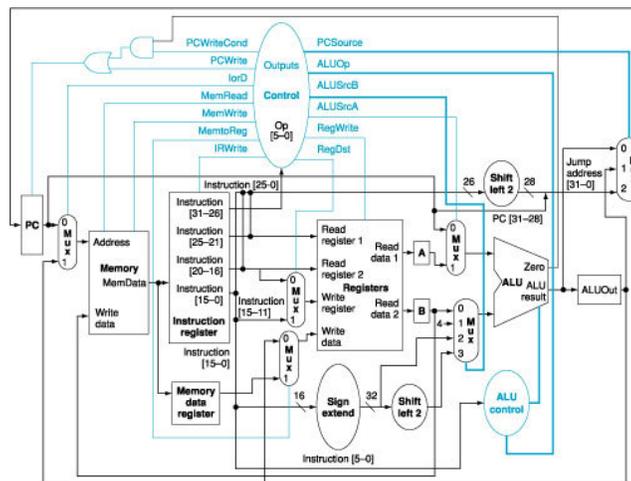
ALUOp = 00 → ALU adds

ALUOp = 01 → ALU subtracts

ALUOp = 10 → ALU uses function field

25

### Example Control

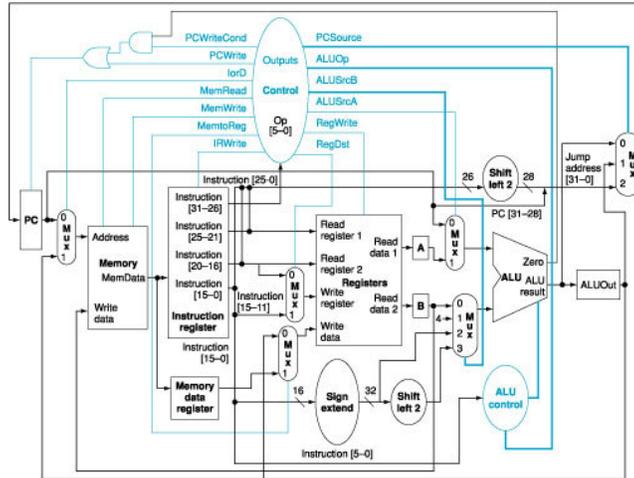


### Step 1: Instruction Fetch

IR  $\leftarrow$  Memory[PC]

PC  $\leftarrow$  PC + 4

## Example Control



## Step 2: Decode/Register Fetch

$A \leftarrow \text{Reg}[\text{IR}[25:21]];$

$B \leftarrow \text{Reg}[\text{IR}[20:16]];$

$\text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[15:0]) \ll 2);$

Exercise #1: Specify control signals needed for a **load** instruction

Step 3:

$\text{ALUOut} \leftarrow A + \text{sign-extend}(\text{IR}[15:0]);$

Step 4:

$\text{MDR} = \text{Memory}[\text{ALUOut}]$

Step 5:

$\text{Reg}[\text{IR}[20:16]] = \text{MDR}$

Exercise #2: Specify control signals needed for a **R-type** instruction

Step 3:

**ALUOut <= A op B**

Step 4:

**Reg[IR[15:11]] <= ALUOut;**

Exercise #3: Specify control signals needed for a **branch** instruction

Step 3:

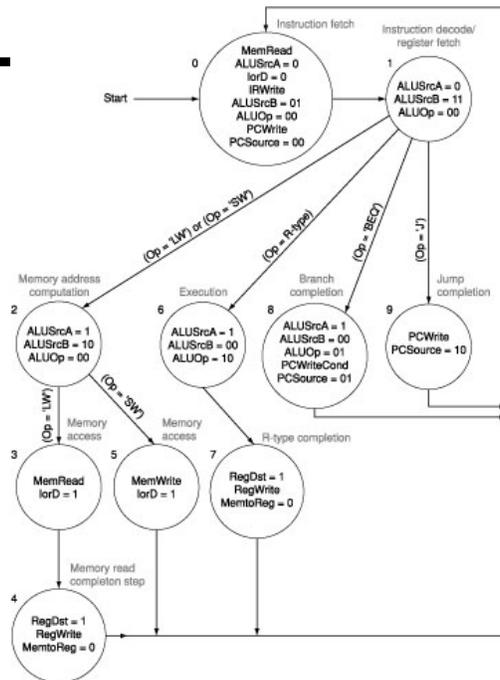
**if (A==B) PC <= ALUOut;**

Exercise #4: Write out steps 3-4 for a **store** instruction and show the control signals needed

Exercise #5: Write out the step(s) (beyond 1 and 2) needed for a “jump” instruction, along with associated control.

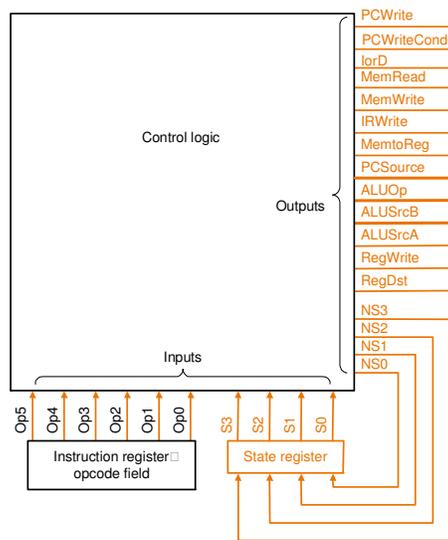
## Graphical Specification of FSM

- How many state bits will we need?



## Finite State Machine for Control

- Implementation:



## Chapter 5 Summary

---

- If we understand the instructions...
  - We can build a simple processor!
- If instructions take different amounts of time, multi-cycle is better
- Datapath implemented using:
  - Combinational logic for arithmetic
  - State holding elements to remember bits
- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation