Lab 9

Single Cycle MIPS Datapath

(R-type and Load/Store)

* Pre-Lab

1. Introduction

Our next task is implementing MIPS single cycle datapath shown in figure below. It will take more than one labs. MIPS has 32 registers each 32-bit wide. The instruction size is 32-bit too.

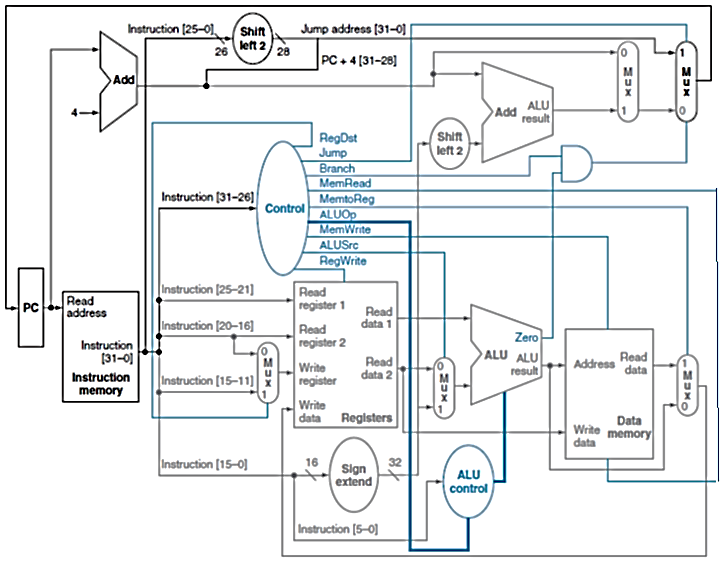
****

Figure 10.1

Common steps in instruction execution are

* Fetching the instruction word from the instruction memory
* Decoding the instruction and reading from the register file or prepare a value from the immediate value (and PC)
* Performing an ALU operation 
* Accessing the data memory (if needed)
* Making a jump (assigning a computed value to PC) (if needed)
* Writing to the register file

1. Implementation.
   1. Supported Instruction set

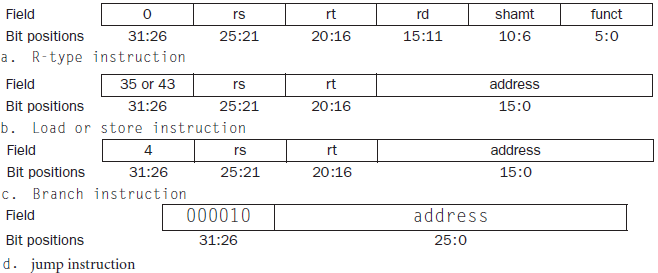
We will implement only four instruction types, R-type, load and store, branch and jump. The R-type instructions which we will implement are add, sub, and, or and slt. Diagram below explains how each format divides the 32-bit instruction in to fields. 

Figure 10.2

1. Building Datapath

The first thing that we need to build the datapath is an instruction memory and a mechanism to get new instruction on each clock cycle. The mechanism can be constructed using PC (Program Counter) and an adder. PC is a 32-bit register which holds the address of the next instruction and the adder increments PC on each clock cycle so that it points to the next instruction. Write (three) separate modules for each of these.

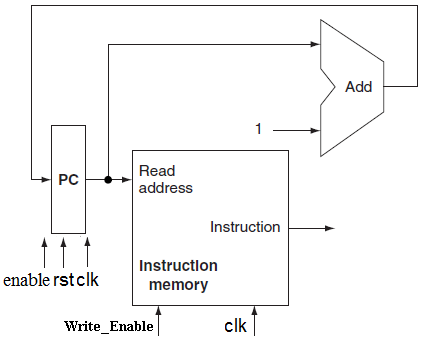


Figure 9.3. Instruction memory and the mechanism to get next instruction on every clock cycle

* 1. R-type

Today we will implement the R-type and load/store instructions only. R-type instructions for example add $rd,$rs,$rt reads two registers from register file ($rs and $rt), forwards them to ALU and writes the result back to the destination register ( $rd). So the components required are a register file and an ALU.

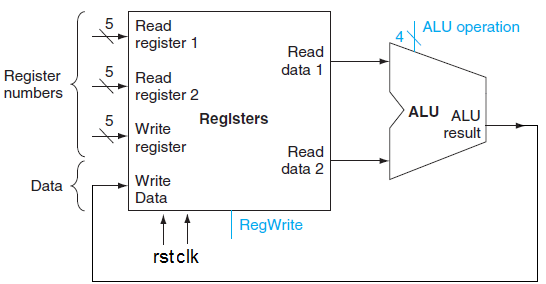


Figure 10.4

Control signals are shown in blue. Data is written only when ‘RegWrite’ is asserted. ‘ALU operation’ control signals are used in deciding the operations performed on operands(values read from registers).

* 1. Memory Unit(for Register file, Instruction and data memory)
     1. **Declaring Memory**

In order to implement a Memory unit you will have to use an array of vectors.

reg [n-1:0] anyname [m-1:0];//Declaring m registers, each n-bit //wide.

For example in order to declare 8 registers each 32-bit wide, the statements used should be,

reg [31:0] reg\_array [7:0];

In order to move the 6th word (1 word=32bits) of reg\_file in another 32-bit vector ‘b’, the statement used should be

b= reg\_file[5]

* + 1. **Output Mechanism**

Data is read from memory asynchronously (independent of clock). It can be accomplished using an assign statement or an always behavior.

assign dataout=reg \_array[address]

or

always @ (address)

dataout= reg \_array[address];

* + 1. **Input Mechanism**

Data is written to memory synchronously (with clock) when write enable signal is 1. It can be accomplished using an always behavior sensitive to positive or negative edge of clock

.

always @ (posedge clock)

if(write\_enable)

reg \_array[address]<=datain;

**Test code with hexadecimal code**

32’bit Hexadecimal value Corresponding Code

018d4820 add $9, $12, $13 # $3 = -1

01cf5020 add $10, $14, $15 # $5 = 1

01494022 sub $8, $10, $9 # $8 = -2

ac080004 sw $8, 4($0) # Store $9 in #DMem[8]

* + 1. **Initializing Memory**

For simulation purposes memory can be initialized using an initial behavior

initial

begin

reg\_array[0]= 8c0c0000;

reg\_array[1]= 8c0d0001;

so on..

..

end

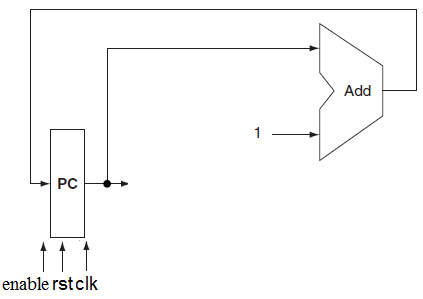
* In-Lab

**Tasks**

* **Write a Verilog HDL module PC Plus adder**

module pcplusadder(**input** clk,rst,enable, **output**[31:0] pc);

At reset pc=0, otherwise if enable=1 it keeps incrementing pc at each clock cycle. If enable=0 pc remains unchanged

****

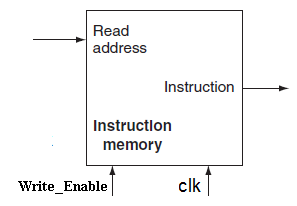
* **Write Verilog module for instruction memory which is an array of registers(15 locations)**

module Instruction\_Memory(**input**[31:0] address,

**input** clock,

**input** enable,

**output** [31:0]instruction);



Initialize memory with following hex values

8c0c0000 ; lw $12, 0($0)

8c0d0001 ; lw $13, 1($0)

8c0e0002 ; lw $14, 2($0)

8c0f0003 ; lw $15, 3($0)

018d4820 ; add $9, $12, $13 # $3 = -1

01cf5020 ; add $10, $14, $15 # $5 = 1

01494022 ; sub $8, $10, $9 # $8 = -2

ac080004 ; sw $8, 4($0) # Store $9 in DMem[8]

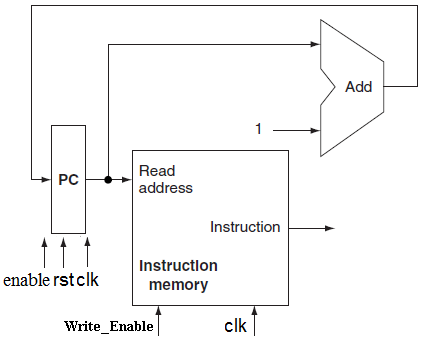
* Post-Lab
* **Combine the two units and name it**

module Get\_Instruction( **input** clock,

input reset,

**input** enable,

**output** [31:0]instruction);



**Submission details**

* **Your lab report, a .doc file, should contain properly commented Post-Lab task code, with Screenshots(of print preview) of Schematic and waveforms, and Critical Analysis.**
* **The report must have a title page in the pescribed format.**
* **Name the .doc file RegNo.docx; eg SP14-BCE-99.docx**
* **Sumbit on portal.**