LAB # 11:

Follow the steps to reproduce the Execution unit of MIPS 32-bit Microprocessor using VHDL and implementation on FPGA

Objective

- To design the Execution module of a single-cycle MIPS 32-bit processor

Pre-Lab

In this lab you will complete the MIPS-32 processor design by adding the final module: the execution unit. The execution module is marked using a red polygon in Figure 11.1.

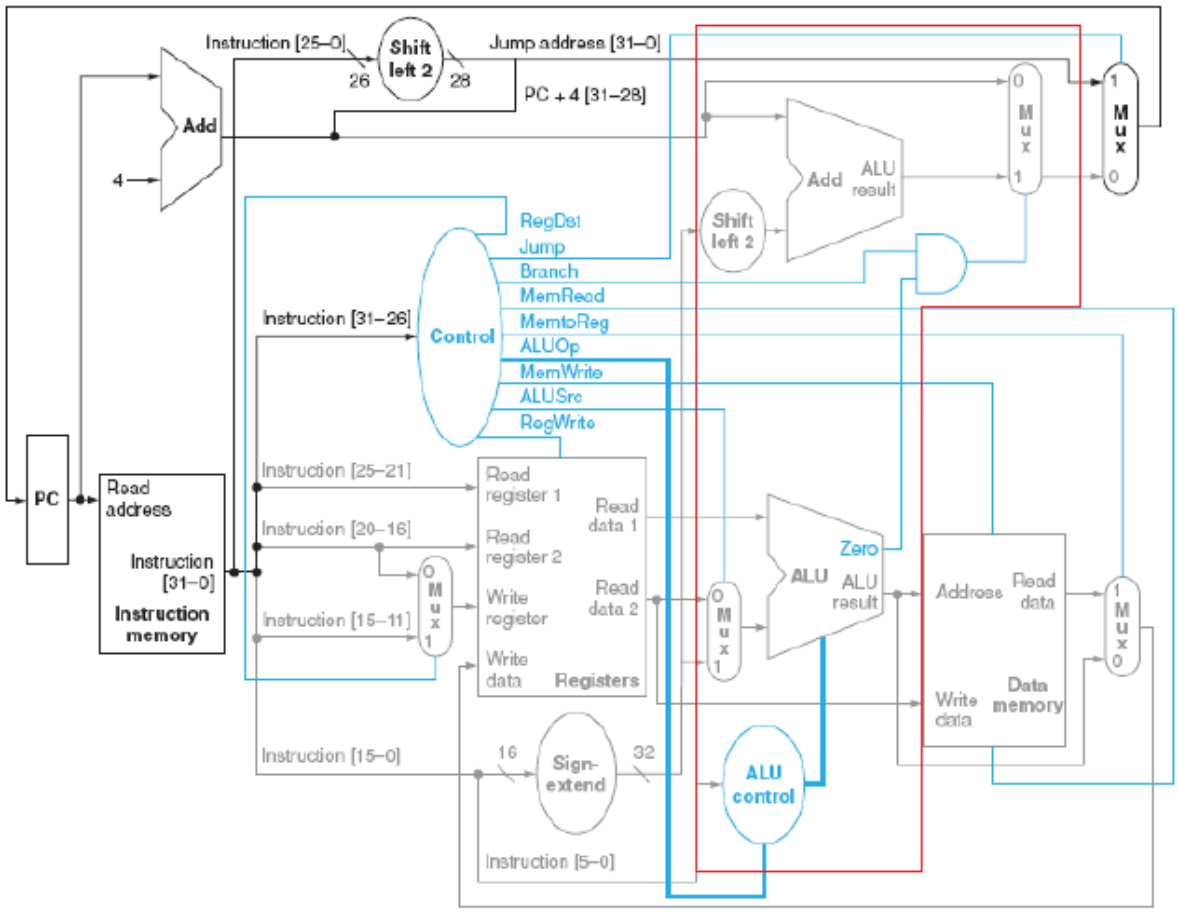

Figure 11.1 Execution Unit

The execution module (execute.vhd)

The execution module takes its inputs from the decode module and performs ALU operations. It consists of six inputs which are

1. register_rs (32bits)
2. register_rt (32bits)
3. immediate (32bits)
4. ALUOp (2 bits)
5. ALUSrc (1bit)
6. beq_control (1 bit).

These inputs produce three outputs:

1. alu_result (32 bits)
2. branch_addr (32 bits)
3. branch_decision (1 bit).

The block diagram is shown in Figure 11.2

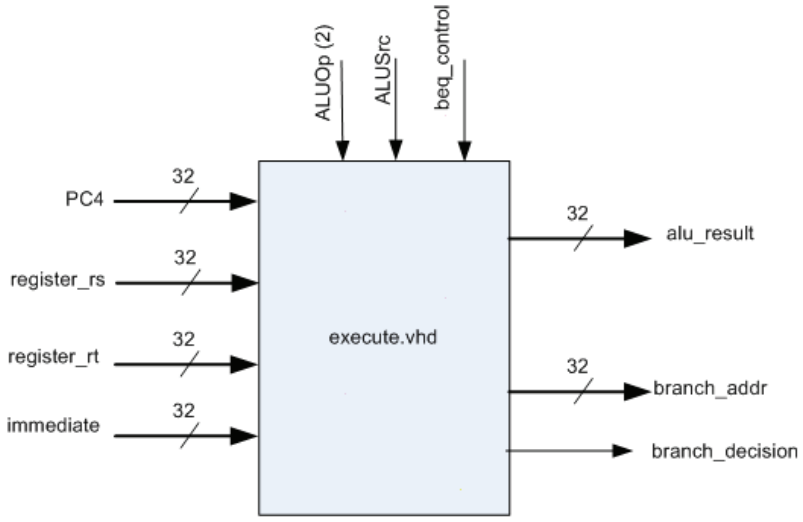

Figure 11.2 The execute module

For the design, understanding of the ALU control is imperative. Please refer to the textbook. Assume that you defined two temporary local variables ***alu_output***(32bit) and zero (1 bit). The zero variable is later passed to ***branch_decision*** output. The function of ALU is controlled by two bits of ALUOp as summarized below.

```
When ALUOp = "00"
--This is a memory instruction, so memory address is computed and set to the alu_output,
--i.e.,
--      alu_output := register_rs + immediate
-- In addition, zero must be set according to the condition.
When ALUOp = "01"
-- This is a branch instruction.
-- alu_output := register_rs - register_rt
-- The result of this operation, that is whether it is 0 or not,
-- must set the variable named zero to 0 or 1.
When ALUOp = "10"
-- This is an r-type instruction, so the operation is determined by the 6 least
-- significant bits in the instruction. A couple examples are:
case immediate(5 downto 0) is
    when "100000" => --add
        alu_output := register_rs + register_rt;
    when "100010" => -- subtract
        alu_output := register_rs - register_rt.
--      and so on.

 --      and so on.
 -- For undefined or unimplemented instructions (when others), write the error indications as,
        alu_output := x"ffffffff";
 -- The branch address is computed using:
        branch_offset := immediate;
 -- recall that we are using a word address, so no need for shift by 2
        temp_branch_addr := PC4 + branch_offset;
 -- The final synchronized outputs are then given as:
        branch_decision <= (beq_control and zero);
        branch_addr <= temp_branch_addr;
        alu_result <= alu_output;
 --The execute module is implemented as a process in a vhdl module.
```

Pre-Lab Task

Task 1: Entity of Execute Module

The entity of ***execute.vhd*** according to the above block diagram is:

```
entity execute is
    port( register_rs, register_rt: in std_logic_vector(31 downto 0);
          PC4, immediate: in std_logic_vector(31 downto 0);
          ALUOp: in std_logic_vector(1 downto 0);
          ALUSrc: in std_logic;
          beq_control,clock: in std_logic;
          alu_result, branch_addr: out std_logic_vector(31 downto 0);
          branch_decision: out std_logic);
end execute;
```

Task 2: Writing the Execute Module

```
architecture Behavioral of execute is
begin
    process
    variable alu_output: std_logic_vector(31 downto 0);
        variable zero: std_logic;
        variable branch_offset, temp_branch_addr: std_logic_vector(31 downto 0);
    begin
            if(clock'event and clock='1') then
                case ALUOp is
                    when "00" =>
                        --logic comes here
                    when "01" =>
                        --logic comes here
                    when "10" =>
                        --logic comes here

                end case;
                branch_offset:=immediate;
                temp_branch_addr:=PC4+branch_offset;
                branch_decision <= (beq_control and zero);
                branch_addr <= temp_branch_addr;
                alu_result <= alu_output;
            end if;
    end process;
end Behavioral;
```

Lab Tasks

Testing will be done by connecting all the modules you have constructed. Your understanding on the data path is important for debugging.

Lab Task 2: Make module-to-module connections as shown

For all connections, a signal should be defined. This allows testing of the module-to-module functionality to be done conveniently by connecting them to LEDs, 7-segment displays, and the PC Hyper terminal.

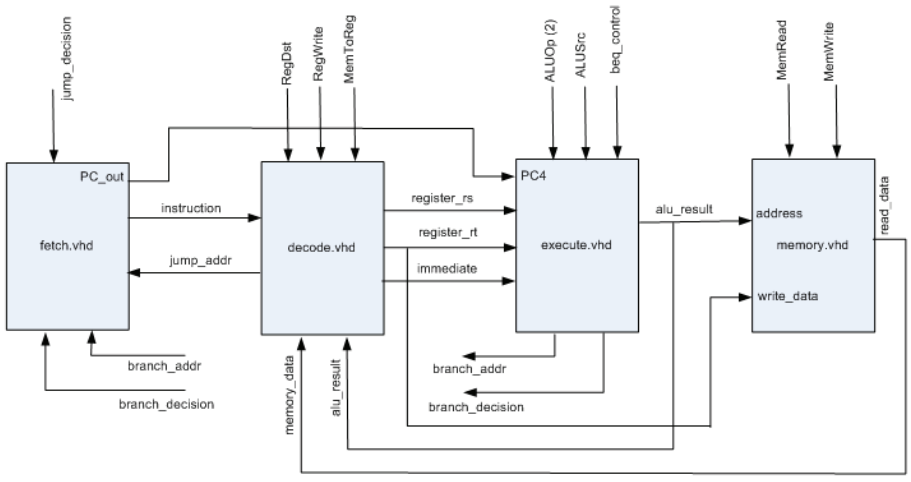

Figure 11.3 Module to Module connection

This diagram does not show the control module, neither the reset signal, to avoid the clutter. It is not shown but the 32bit instruction must be fed to the control module. All of the outputs of the control module are connected to the specified locations in the diagram.

Lab Task 3: Connections to switches, LEDs, and 7-segment

From the decode module, assign register_rd as an additional output port and modify the code so that its value can be displayed. Assign the switches as

Switch status	To Display
0001	Displays **PC_out** from fetch.vhd to LEDs
0010	Displays **register_rs** from decode.vhd to 7-segment
0011	Displays **register_rt** from decode.vhd to 7-segment
0100	Displays **register_rd** from decode.vhd to 7-segment
0101	Displays **Immediate** from decode.vhd to 7-segment
0110	Displays **alu_result** from execute.vhd to 7-segment
0111	Displays **read_data** from memory.vhd to 7-segment
Otherwise	Displays **Instruction** from fetch.vhd to 7-segment

Connect the clock to a push button as before and debounce it.

Lab Task 4: Examine the registers, PC, alu_result, and memory values instruction-by-instruction

```
--instruction memory

variable mem: mem_array := (

X"8c220000",          --L: lw $2, 0($1) ; $2 <= mem[0+$1]

X"8c640001",          -- lw $4, 1($3) ; $4 <= mem[1+$3]

X"00622022",          -- sub $4, $3, $2 ; $4 <= $3 - $2

X"ac640000",          -- sw $4, 0($3) ; mem[0+$3] <=$4

X"1022fffb",          -- beq $1, $2, L ; if ($1=$2), branch_addr<=L

X"00612064",          -- and $4, $3, $1 ; $4 <= $3 and $4

X"08000000",          -- j L ; jump_addr <= L

X"00000000"); --
```

Lab Task 5: Demo

- Show the registers, PC, immediate, read_data, alu_result of the following test code. This code simply tests loading of two values, add them, and store the result.
- Initialize the register and memory values as:
  reg0=0, reg1=1, reg2=2, reg3=3, reg4=4, reg5=5, reg6=6, reg7=7

  mem[0]=0, mem[1]=1, mem[2]=2, mem[3]=3, mem[4]=4, mem[5]=5, mem[6]=6, mem[7]=7

- Test the following five lines of codes and provide a demo:Show the four outputs of the decode module using buttons, switches, and the Hyperterminal for each type of instruction.

	Machine Code	Assembly Code	Register, imm, ALU, and mem signals, i.e., contents
0	8c22 0000	lw $2,0($1)	rs=1, rt=1, rd=, imm=0, ALU=1, mem=1
1	8c23 0005	lw $3,5($1)	rs=1, rt=6, rd=, imm=5, ALU=6, mem=6
2	0062 2020	add $4,$3,$2	rs=6, rt=1, rd=7, imm=2020, ALU=7, mem=6 (x)
3	ac24 0000	sw $4, 0($1)	rs=1, rt=7, rd= , imm=0, ALU=1, mem=6 (x)
4	8c22 0000	lw $2, 0($1)	rs=1, rt=7, rd= , imm=0, ALU=1, mem=7

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date: _____