LAB # 7:
Follow the steps to reproduce the Fetch module of MIPS 32-bit microprocessor using VHDL and implementation on FPGA

Objective

- To design and implement the Fetch module of a single-cycle MIPS 32-bit processor

Pre-Lab

Fetch Module (fetch.vhd)

Program Counter ***(PC)*** provides the address of an instruction to be fetched from the instruction memory. For simplicity, we will implement the instruction memory using an array of 32-bit words, which means that your PC will use word addresses, instead of byte addresses as in a real MIPS processor. For this lab, we will limit the size of the instruction memory to 16 words, which would result in using only the least significant 4-bits of the address bus.

The address of an instruction memory word is selected based on three options:

1) Normal PC+1,
2) Branch address, or
3) Jump address.

To incorporate the three options of the PC choices, the fetch module includes branch and jump addresses as a part of its inputs, as illustrated in Figure 7.1. The jump address is supplied by the decoding unit, while the branch address is supplied by the execution unit, which would be implemented in the future labs. The ***PC_out*** output is determined based on two input signals; that is, ***branch_decision*** and ***jump_decision***. If both of these signals are zero, normal PC+1 is sent to ***PC_out***. Otherwise, ***PC_out*** would be the ***branch*** or ***jump target address*** plus one.

The ***PC_out*** is later used by the execution unit (execute.vhd) to compute the branch address. Since ***branch_addr*** is computed from the execute module, it is not a relative address but an absolute address. For this lab, it is only used to test and confirm instruction fetch operations.

We also need a ***Reset*** signal that should set the PC to 0, so the computer can perform initialization of the system. In order to fetch one instruction per clock, a ***clock*** signal (generated by a button) is used as one of the inputs of the fetch module.

Summarizing these needs, a block diagram of the fetch module is shown in Figure 7-1 which includes all required inputs and outputs.

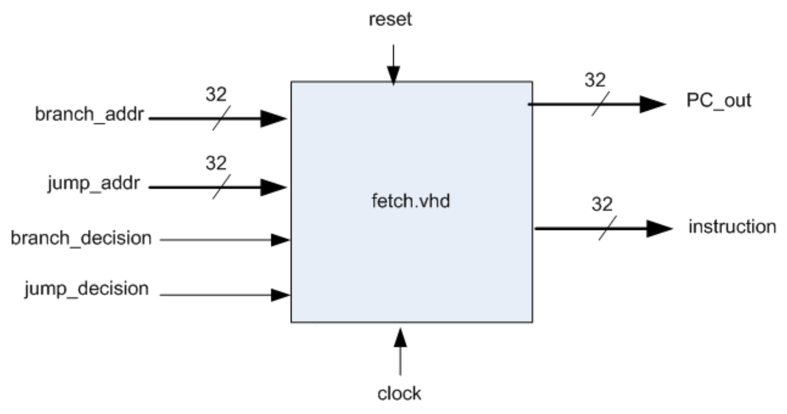

FIGURE 7-1: ENTITY FOR FETCH MODULE

For implementation, the PC is implemented as an internal variable. The internal ***PC*** is normally incremented by one and then outputted to ***PC_out***, i.e., ***PC_out <= PC+1***. For branch and jump instructions, the ***branch_addr*** or ***jump_addr*** is loaded to ***PC*** when ***jump_decision=1*** or ***branch_decision=1***, respectively. The instruction fetched from the PC location is sent out to the instruction output, while PC+1 is sent to the ***PC_out*** for the next instruction fetch.

Pre-Lab Task

Task 1: Entity of Fetch Module

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

```
entity fetch is
port (
    PC_out                                          : out STD_LOGIC_VECTOR (31 downto 0);
    instruction                                     : out STD_LOGIC_VECTOR (31 downto 0);
    branch_addr, jump_addr                          : in STD_LOGIC_VECTOR (31 downto 0);
    branch_decision, jump_decision, reset, clock    : in std_logic
);
end fetch;
```

Task 2: Writing the Fetch Module

```
architecture bhv of fetch is
--instruction memory is created as an array of 32bits and has 16 locations
type mem_array is array(0 to 15) of std_logic_vector(31 downto 0);
begin
process
variable mem: mem_array :=  ( --initialize the instruction memory
            X"8c220000",        --L:    lw $2, 0($1)      -- make your own machine codes here
            X"8c640001",        --      lw $4, 1($3)          -- to check the PC changes
            X"00822022",        --      sub $4, $4, $3
            X"ac640000",        --      sw $4, 0($3)
            X"1022fffa",        --      beq $1, $2, L
            X"00612064",        --      and $4, $3, $1
            X"08000000",        --      j L
            X"00000000".
            …
            );                  --
variable PC : std_logic_vector(31 downto 0);        -- PC is defined here
variable index : integer := 0;
begin
```

```
--- fetch process code begins here.
--- below describes the logic, not the code. It is your responsibility to code it.
--- wait until start of a cycle, i.e., wait until (clock'event and clock='1')
--- if reset = '1' then
--- set the PC to zero, i.e., PC := x"00000000";
--- set instruction to zero, i.e., instruction <= x"00000000"
--- this takes care of the reset
--- Check if PC should be normal increment, branch, or jump address
--- if (branch_decision = '1') then PC := branch_addr;
--- if (jump_decision ='1') then PC := jump_addr
--- Since we are using only four least significant bits,
index := to_integer(PC(3 downto 0));
--- Increment the PC by one, i.e.,
PC := PC + X"1";
remember that your PC uses word address in our design
-- Please remember that variables retain their values until the next time this process is executed.
--- At the end of the process, you need to output the results.
PC_out <= PC;
instruction <= mem(index); -- Output the fetched instruction
end process;
end bhv;
```

Note from the sample MIPS assembly code that the registers are now expressed simply \$0, \$1, ..\$7. For simplicity, we will only implement eight registers, and the registers are simply expressed as \$0 - \$7. This syntax is accepted by the Mars assembler. Therefore, you can still use the Mars assembler to generate the machine codes for this MIPS implementation, except that the PC values have to be modified to word addresses.

In-Lab Tasks

You may start implementing the **fetch.vhd** as the top module to debug it, but it must be turned to a module later to test interfaces. Once debugging is completed, an interface test module must be written as a top module to test and verify the operations of the **fetch.vhd**.

Lab Task 1: Write codes to test the functionality

- Design a wrapper file that calls (port map) fetch module created in pre-lab tasks.
- Set 8 seven segments to display the data (we will call it display unit).
- Connect ***branch_decision, jump_decision***, and ***reset*** to slide switches for testing.
- Connect the ***clock*** signal to one of the push buttons
- Depending on the conditions, show the respective PC value and instruction on display unit
- Test and verify whether ***PC_out*** increments by one for normal (i.e., ***branch_decision=0*** and ***jump_decision=0***) instructions when a clock period is applied. Verify if the correct instruction is fetched to the instruction output, and if **PC** retains the correct value.
- Show the operations of branch and jump. If ***branch_decision='1'*** or ***jump_decision='1'***, the ***PC*** display (LEDs) should show the correct address changes. Later, the control unit should be designed.

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: _____________________