LAB # 8:
Follow the steps to reproduce the Decode module of MIPS 32-bit Microprocessor using VHDL and implementation on FPGA

Objective

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

Pre-Lab

In previous lab, you designed the fetch module (fetch.vhd) and tested PC changes and instruction fetches. In this lab, you will design the decode module that decodes the fetched instruction. The decode module is marked using a blue circle in Figure 9.1. The objective is to design the decode.vhd and test it along the fetch.vhd.

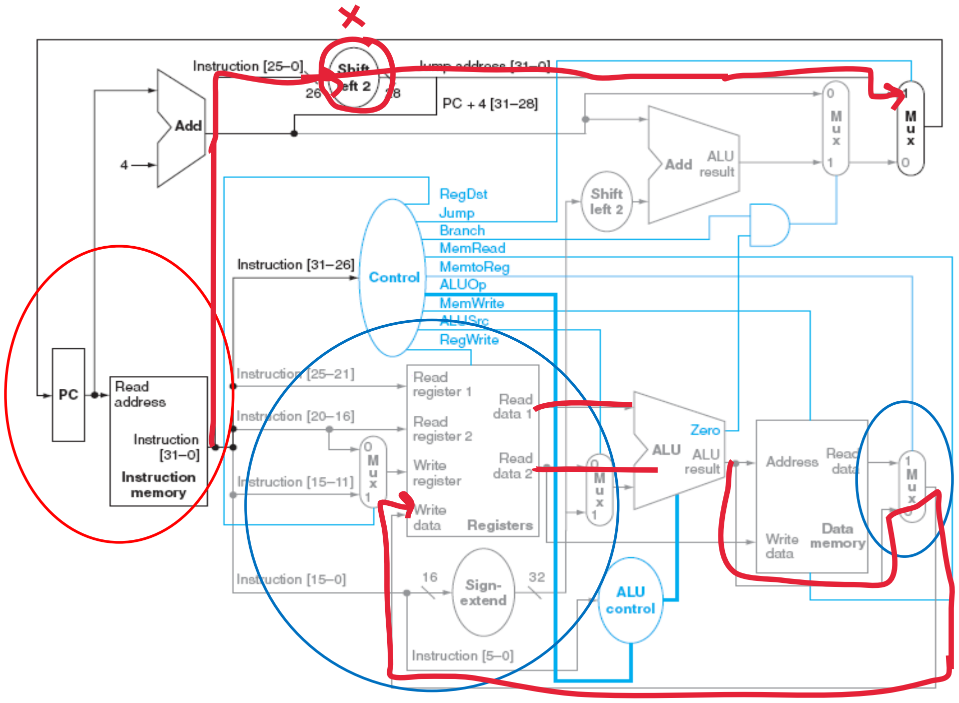

FIGURE 8-1: SINGLE CYCLE MIPS DATAPATH

Decode module (decode.vhd)

The decode module in the overall data path diagram is shown in a blue circle in Figure 8-1. The functional block diagram of the decode module is depicted in Figure 8-2. In this design, the registers will be placed inside the decode module. Note from the diagram that the decode module includes output signals denoted ***register_rs*** and ***register_rt***. These are two register values read from the register file pointed by the rs and rt fields of the instruction, which will be sent to ALU. It also includes ***jump_addr*** and immediate signals that are directly decoded from the instruction. The job of decode module is then to decode the register addresses and immediate values from the fetched instruction and then to send out the values to respective output signals. Since the registers reside inside the decode module, the values of destination register, ***register_rd***, is not allocated as the output.

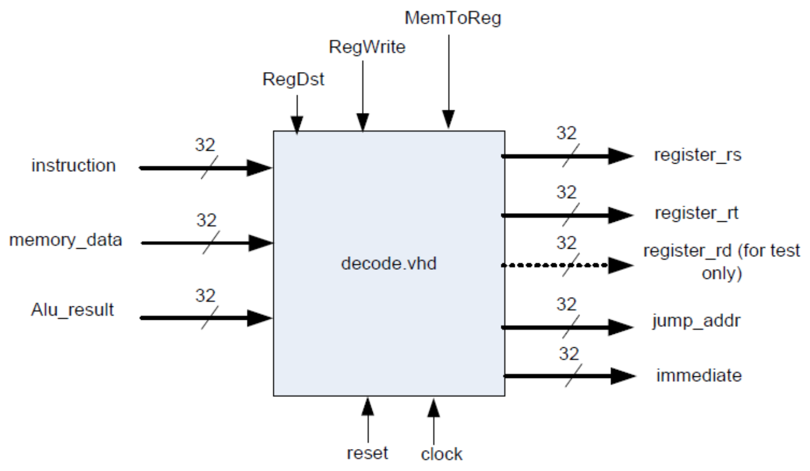

FIGURE 8-2: DECODE MODULE

There are two ways of implementing the internal registers. The first approach is to use an array of 32-bit values, similarly to the memory implementation. The second approach is to implement the register file as signals, i.e.,

signal reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7 : std_logic_vector(31 downto 0);

The second approach makes the code longer and looks not efficient. However, if we look at in a hardware point of view, it should actually run faster since it does not require decoding of the array index. For this lab, either approach should work fine, but we will limit the number of registers to only eight (8). Real MIPS has 32 registers.

The next to consider are the internal multiplexers. Internally there are two multiplexers. The first multiplexer determines the source of the destination register address. For R-type instructions, the destination register is specified in the bits of Instruction[15:11]. However, for the load and store instructions the destination register is specified in Instruction[20:16]. The signal ***RegDst*** controls this choice. Figure 9.3 illustrates implementation of theses relations using a multiplexer.

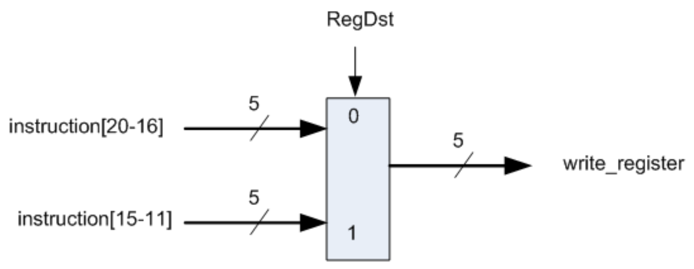

FIGURE 8-3: DESTINATION REGISTER SELECTOR MULTIPLEXER

The second multiplexer determines whether the data to be stored at the destination register is coming from ***Alu_out*** or ***memory_data*** (***read_data*** in Figure 8-1). This relationship is shown in Figure 8-4

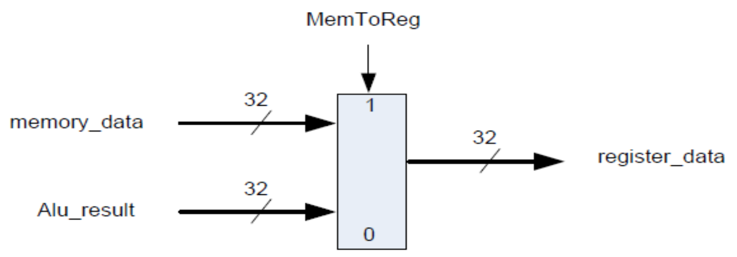

FIGURE 8-4: MULTIPLEXER THAT SELECTS THE SOURCE OF DATA TO BE SENT TO THE DESTINATION REGISTER

Another output must be taken care of is the immediate output. Immediate data is always specified by the bits in Instruction[15:0], but it has to be sign extended to 32 bit. More specifically, Instruction(15) determines the sign and can be coded as:

```
immediate(15 downto 0) <= instruction(15 downto 0);
if instruction(15) = '1' then
    immediate (31 downto 16) <= x"ffff";
else
    immediate (31 downto 16) <= x"0000";
end if;
```

The output ***jump_addr*** is determined from Instruction[25:0]. Since we are using word addresses, concatenation of “00” is not required. We also do not use jump based on the current PC but directly. It should be simply,

```
jump_addr(31 downto 0) <= "000000" & instruction(25 downto 0);
```

Another consideration to simplify the overall program is to create two processes: one for writing registers and another for reading registers. This approach simplifies the overall implementation.

Last, but not least important aspect is that ***reg0*** (or register(0)) must have the constant value 0 because it is called the ***$zero*** register and the value of this register must always be 0.

In-Lab Tasks

Lab Task 1: Entity of Decode Module

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

```
entity decode is
    port (  instruction : in STD_LOGIC_VECTOR (31 downto 0);
    …
    ….
    immediate : out STD_LOGIC_VECTOR (31 downto 0);
    reset : in STD_LOGIC);
    end decode;
```

Lab Task 2: Writing the Decode Module

```
architecture Behavioral of decode is
signal reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7 : std_logic_vector(31 downto 0);
begin
----------------------------------------------------------------------
-- Process to write the register file when required
----------------------------------------------------------------------
reg_write: process(reset, memory_data, alu_result, clock)
variable write_value : std_logic_vector (31 downto 0);
variable addr1, addr2, addr3 : std_logic_vector(2 downto 0);
begin
-- on reset initialize the registers
if reset = '1' then
reg0 <= x"00000000";
--- For testing you may initialize the rest of registers, for example
reg1 <= x"11111111";
….
reg7 <= x"77777777";
else
--- determine the address of the register to be written
--- addr2 := instruction(20 downto 16);
--- addr3 := instruction(15 downto 11);
--- if RegDst = '0' then
…
--- end if;
--- Determine the source operand to be written, i.e., memory or alu result
if RegWrite = '1' then
--- if MemToReg = '1' then
--- write_value := memory_data;
---else
--- write_value := alu_result;
---end if;

--- Store write_value to the destination register (need to disable this when you test other
functions)

--- if register(0) is selected as the destination, simply write 0 to reg0
-- case addr3 is
-- when "000" =>
-- reg0 <= to_stdlogicvector(x"00000000");
-- when "001" =>
-- reg1 <= write_value;
…
-- end case;
end if;
end if;

--- for testing, you may add register_rd in the port and assign rd signal (see the reg_read process
for this).
---However, it should be removed after testing because rd signal does not exist and it is the
register itself.
register_rd <= rd ;
end process reg_write;

-----------------------------------------------------------
-- Process to read register file and pass the operands to the execute module
-----------------------------------------------------------
reg_read: process(instruction)
variable rt, rs, imm : std_logic_vector(31 downto 0);
variable addr1, addr2 : std_logic_vector(2 downto 0);
begin
-- register addresses for reading the registers
addr1 := instruction(25 downto 21);
addr2 := instruction(20 downto 16);
---read the register
---case addr1 is
--- when "000" => rs := reg0;
--- …
--- end case;
--- case addr2 is
--- when "000" => rt := reg0;
--- …
--- end case;
--- access immediate from instruction and perform sign extension
imm(15 downto 0) := instruction(15 downto 0);
if instruction(15) = '1' then
    imm(31 downto 16) := x"ffff";
else
    imm(31 downto 16) := x"0000";
end if;
 --compute the jump address.
 jump_addr(31 downto 0) <= …..;
 --- bring out signals to the ports of the module

 --- register_rs <= rs ;
 --- register_rt <= rt ;
 --- immediate <= imm ;
 --- you may add rd as well for testing
 end process reg_read;
 end Behavioral;
```

Please understand that above codes are only provided as a description or pseudo code, not the actual code.

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: ____________________