In this lab you will complete the MIPS-32 processor design by adding the final two modules: the execution and control module. The execution module is marked using a red polygon in Figure 1. The control module and signals are shown using green color.

Figure 1: Single-cycle implementation of data path
1. The execution module (execute.vhd)

The execution module takes its input from the decode module and performs ALU operations. It consists of seven inputs which are register_rs (32bits), register_rt (32bits), immediate (32bits), ALUOp (2 bits), ALUSrc (1bit), and beq_control (1 bit). These inputs produce three outputs: alu_result (32 bits), brach_addr (32 bits), and brach_decision (1 bit). The block diagram is shown in Figure 2.

Figure 2. The execute module

For the design, understanding of ALU control is imperative. Please refer to the textbook. Assume that you defined a temporary local variables alu_output 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 be set to the variable zero.
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.
```

For undefined or unimplemented instructions (when others), write the error indications as,

```
alu_output := x"fffffffe";
```

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.
2. The control module (control.vhd)

The control module simply generates the control signals based on the instruction op code.

![Figure 3. The control module](image)

The implementation of the control module basically follows the table given in textbook page 323. It is relatively simple to implement this module. Below gives an implementation example (partial):

```vhdl
process (instruction, reset)
variable rformat, lw, sw, beq, jmp, addi : std_logic;  -- 1 if the instruction is identified
variable opcode : std_logic_vector(5 downto 0);
begin
if reset = '1' then
    RegDst <= '0';
    ALUSrc <= '0';
    MemToReg <= '0';
    RegWrite <= '0';
    MemRead <= '0';
    MemWrite <= '0';
    ALUOp <= "00";
    Jump <= '0';
    beq_control <= '0';
else
    opcode(5 downto 0) := instruction(31 downto 26);
    -- identify each instruction type from instruction
    if opcode = "000000" then rformat := '1';
    else rformat := '0'; end if;
    if opcode = "100011" then lw := '1';
    else lw := '0'; end if;
```

4
--- do yourself the rest
--- The instructions you should implement are
--- lw, rformat, sw, beq, jmp, and addiu
--- you would be able to program virtually anything with this instruction set

--- implement all of the control output signals here.
--- see the table in page 327 and implement all of the signals
  RegDst <= rformat;
  ALUSrc <= (lw or sw or addi);
  .
  .
  -- do the rest

end if;
end process;

3. Data Memory module

Data memory module implementation is nearly identical to the instruction memory. Please refer to your previous program. The block diagram is shown in Figure 4 and implemented using an array of 32bits. For your reference, an example implementation is shown in Appendix-B.

![Data Memory Module Diagram](image)

Figure 4: Data memory module
Testing the functions

Testing will be done by connecting all of the modules you have constructed. Your understanding on the data path is important for this lab, and it is recommended that you depict your own diagram and follow the data path.

Step 1: Make module-to-module connections as shown

For all connections, a signal should be defined and used for the connections. This allows testing of the module to module functionality by connecting them to LEDs and Hyperterminal.

Figure 4. Module-to-module connection diagram

This diagram does not show the control module and reset signals 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.

Step 2: Connections to switches, LEDs, and hyperterminal

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
SW0 =1 displays **PC_out** from fetch.vhd to LEDs
SW1 =1 displays **register_rs** from decode.vhd to Hyperterminal
SW2 =1 displays **register_rt** from decode.vhd to Hyperterminal
SW3 =1 displays **register_rd** from decode.vhd to Hyperterminal
SW4 =1 displays **immediate** from decode.vhd to Hyperterminal
SW5 =1 displays **alu_result** from execute.vhd to Hyperterminal
SW6 =1 displays **read_data** from memory.vhd to Hyperterminal

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

**Step 3:** 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"102f1fbb", -- 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"); --
```

Modify the code, so that you can test execution of every type of instructions implemented.

**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

Test the following four lines of codes and provide a demo:

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

- Objective of this lab and intro.
- Your implementation
- Your conclusion
- Attach your VHDL source code

Appendix A – A partial example port map

```vhdl
entity mips32 is
  Port ( clock : in  STD_LOGIC;
        reset : in  STD_LOGIC);
end mips32;
---
--- global 32bit signals
signal s_branch_addr, s_PC_out : std_logic_vector(31 downto 0);
signal s_immediate, s_instruction, s_alu_result : std_logic_vector(31 downto 0);
signal s_register_rs, s_register_rt, s_register_rd : std_logic_vector(31 downto 0);
signal s_memory_data, s_jump_addr : std_logic_vector(31 downto 0);
---
--- global single bit signals
signal s_RegDst, s_MemRead, s_MemWrite, s_MemToReg, s_ALUSrc : std_logic;
signal s_RegWrite, s_branch_decision, s_jump_decision, s_beq_control : std_logic;
signal s_ALUOp : std_logic_vector(1 downto 0);
begin
  FE: fetch port map (PC_out=>s_PC_out, instruction=>s_instruction,
                      branch_addr=>s_branch_addr, jump_addr=>s_jump_addr,
                      branch_decision=>s_branch_decision, jump_decision=>s_jump_decision,
                      reset=>reset, clock=>clock);
  ID: decode port map (instruction=>s_instruction, memory_data=>s_memory_data,
                       alu_result=>s_alu_result, RegDst=>s_RegDst,
                       RegWrite=>s_RegWrite, MemToReg=>s_MemToReg,
                       register_rs=>s_register_rs, register_rt=>s_register_rt, register_rd=>s_register_rd,
                       jump_addr=>s_jump_addr, immediate=>s_immediate,
                       reset=>reset);
  EX: execute port map (PC4=>s_PC_out, register_rs=>s_register_rs,
                        register_rt=>s_register_rt, immediate=>s_immediate,
                        ALUOp=>s_ALUOp, ALUSrc=>s_ALUSrc,
                        beq_control=>s_beq_control, branch_addr=>s_branch_addr,
                        alu_result=>s_alu_result, branch_decision=>s_branch_decision);
```

Appendix B -- An example of data memory module implementation

entity memory is
  generic (module_delay: time := 10 ns);
  Port ( address : in  STD_LOGIC_VECTOR (31 downto 0);
         write_data : in  STD_LOGIC_VECTOR (31 downto 0);
         MemWrite, MemRead : in std_logic;
         read_data : out  STD_LOGIC_VECTOR (31 downto 0));
end memory;

architecture Behavioral of memory is
  type mem_array is array(0 to 7) of std_logic_vector(31 downto 0);
begin
  ReadWrite1: process(address, write_data)
    variable data_mem : mem_array := (X"00000000",    --initialize the data memory to test the content
                                      X"11111111",
                                      X"22222222",
                                      X"33333333",
                                      X"44444444",
                                      X"55555555",
                                      X"66666666",
                                      X"77777777");
    variable addr: integer;
    variable mem_content : std_logic_vector(31 downto 0);
  begin
    addr := conv_integer(address(2 downto 0));    --since there are only 8 words
    mem_content := write_data;
    if MemWrite = '1' then
      data_mem(addr) := mem_content;
    elsif MemRead = '1' then
      mem_content := data_mem(addr);
      read_data <= mem_content after module_delay;
    end if;
  end process ReadWrite1;
end Behavioral;