Project

CMPEN 331 – Computer Organization and Design

Late submission is not accepted and will result in not getting any credits for the project

In the project, you just need to implement what was described in the honor option section in lab 5, with the addition of the implementation and generation of the bit stream without errors.

1. Write a report that contains the following:
   - **i.** Your Verilog design code. Use:
     i. Device: Zyboboard (XC7Z010- -1CLG400C)
   - **ii.** Your Verilog® Test Bench design code. Add "`timescale 1ns/1ps" as the first line of your test bench file.
   - **iii.** The waveforms resulting as requested from item 9 above.
   - **iv.** The design schematics from the Xilinx synthesis of your design. Do not use any area constraints.
   - **v.** Snapshot of the I/O Planning and
   - **vi.** Snapshot of the floor planning
   - **vii.** The design should be free from errors when synthesized, implemented and generated of the bitstream.

The report format will be as follows:

2. REPORT FORMAT: Free form, but it must be:
   a. One report per student.
   b. Have a cover sheet with identification: Title, Class, Your Name, etc.
   c. You have to write an abstract at the beginning of the project report to describe what you are doing in the project.
   d. You should include an introduction for the project explaining with diagrams the connection between all the stages and what would be the benefit of using that architecture in the computer organization field.
   e. Use Microsoft word and your report should be uploaded in word format not PDF. If you know LaTex, you should upload the Tex file in addition to the PDF file.
   f. Single spaced

The following part is not mandatory but any student will choose to do this part in addition to the previous part, will take 3 points extra to the total grade of the whole course:

In this extra points project, the students are implementing a pipeline CPU using the Xilinx design package for FPGAs. You can use any information available in previous labs if needed.

3. **Pipelining**

   As described in lab 3

4. **Circuits of the Instruction Fetch Stage**

   As described in lab 3

5. **Circuits of the Instruction Decode Stage**

   As described in lab 3

6. **Circuits of the Execution Stage**

As described in lab 4

7. Circuits of the Memory Access Stage

As described in lab 4

8. **Circuits of the Write Back Stage**

As described in lab 5

9. Control Hazards and Delayed Branch

The control hazard occurs when a pipelined CPU executes a branch or jump instruction. The jump target address a jump instruction (jr, j, or jal) can be determined in the ID stage and it will be written into PC at the end of the ID stage. But because the pipelined CPU fetches instruction during every clock cycle, the next instruction is being fetched during the ID stage of the jump instruction. The control hazard caused by a conditional branch instruction (beq or bne) becomes more serious than that of a jump instruction because the condition must be evaluated in addition to the calculation of the branch of the target address. Figure 1 shows an example when we calculate the branch target address in the EXE or the ID stage respectively. There are mainly two methods to deal with the instruction(s) next to branch or jump instruction. One method is to cancel it (them). The other is to let it (them) be executed. The second method is called a delayed branch. The position in between the location of a jump or branch instruction and the jump or branch target address are called delay slots. MIPS (microprocessor without interlocked pipeline stages) ISA (instruction set architecture) adopts a one delay slot mechanism: the instruction located in delay slot is always executed no matter wither the branch is taken or not as shown in figure 2. In figure 2 (a) shows the case where the branch is not taken. Figure 2 (b) shows the case where the branch is taken; t is the branch target address. In both cases, the instruction located in a+4 (delay slot) is always executed no matter whether the branch is taken or not. In order to implement the delayed branch with one delay slot, we must let the conditional branch instructions finish the executions in the ID stage. There should be no problem for calculating the branch target address within the ID stage. For checking the condition, we can perform an exclusive OR (XOR) on the two source operands:

```
rsrtequ = ~| (da^db);     // (da == db)
```

where the `rsrtequ` signal indicates where `da` or `db` are equal or not. Both `da` and `db` should be the state of the art data. Referring to figures 3 and 4, we use the outputs of the multiplexers for internal forwarding as `da` and `db`. This is the reason why we put the forwarding to the ID stage instead of to the EXE stage. Because the delayed branch, the return address of the MIPS `jal` instruction is PC+8. Figure 5 illustrates the execution of the `jal` instruction. The instruction located in delay slot (PC + 4) was already executed before transferring control to a function (or a subroutine). The return address should be PC+8, which is written into `$31` register in the WB stage by the `jal` instruction. The return form subroutine can be done by the instruction of `jr $31`. The `jr rs` instruction reads the content of register `rs` and writes it into the PC.

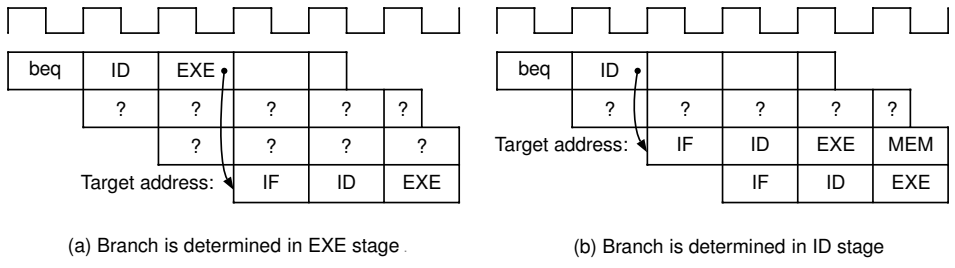

Figure 1 Determining whether a branch is taken or not taken

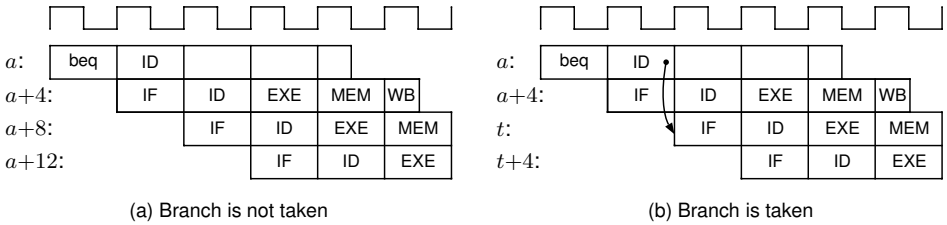

Figure 2 Delayed branch

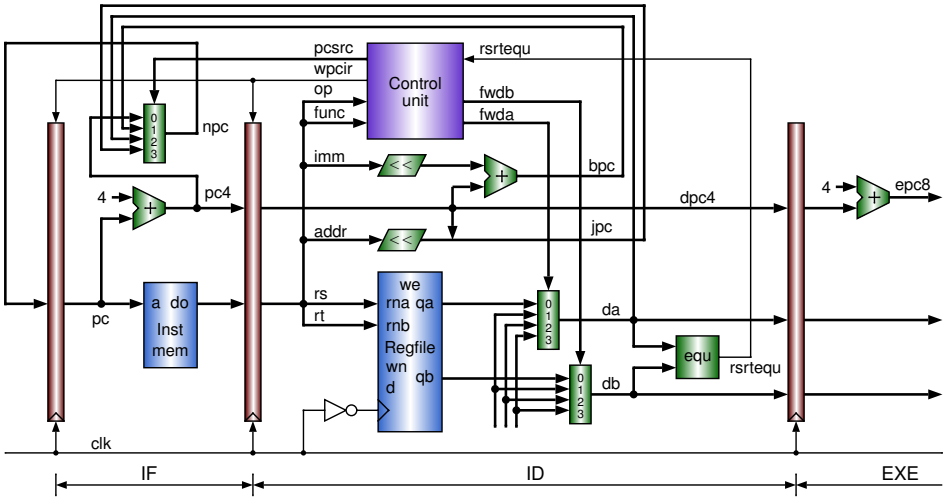

Figure 3 Implementation with delayed branch with one delay slot

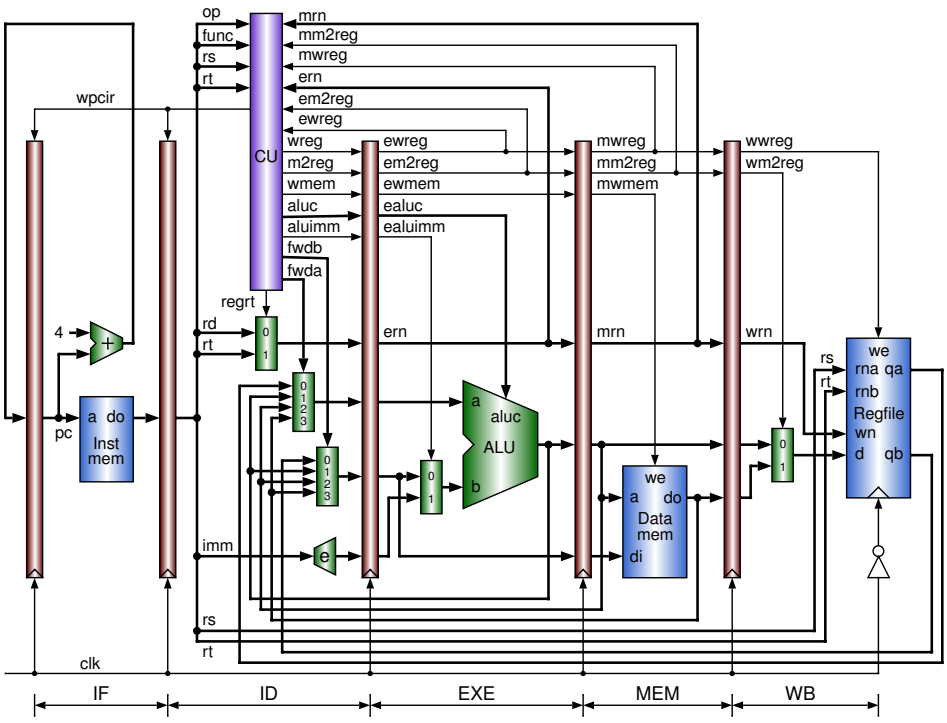

Figure 4 Mechanism of internal forwarding and pipeline stall

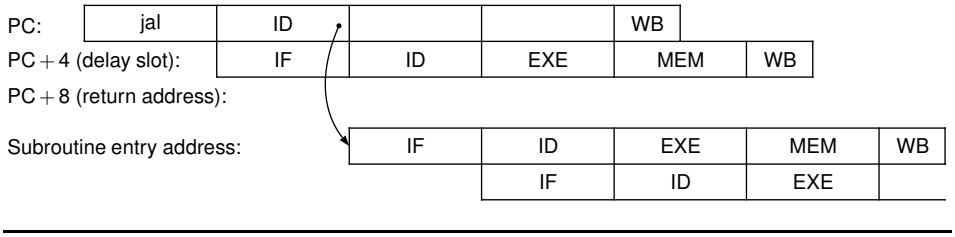

Figure 5 Return address of the function call instruction

For your reference, figure 6 illustrate the detailed circuit of the pipelined CPU, plus instruction memory and data memory. The PC can be considered as the first pipeline located in front of the IF stage, and a register of the register file can be considered as the sixth (last) pipeline register at the end of the WB stage.

In the IF stage, an instruction is fetched from instruction memory, and the PC is incremented by 4 if the instruction in the ID stage is neither a branch nor a jump instruction, and there is no pipeline stall. There are four sources for the next PC:

`pc4`: PC+4
`bpc`: branch target address of a `beq` or `bne` instruction
`da`: target address in register of a `jr` instruction
`jpc`: jump target address of a `j` or `jal` instruction

The selection of the next PC `(npc)` is done by a 32-bit 4-to-1 multiplexer whose selection signal is pcsrc (PC source), generated by the control unit in the ID stage.

In the ID stage, two register operands are read from the register file based on `rs` and `rt`; the immediate `(imm)` is extended and the instruction is decoded based on `op` (and `func`) by the control unit.

The selection signal of the multiplexer for ALU's input e.g. A is named `fwda` (forward A) and the other for ALU's input B, is named fwdb (forward B). if there is no data hazard, the multiplexer selects the data read from the register file. The inverse of the stall signal is used as the write enable for the PC and the IF/ID pipeline register `(wpcir)`. The `stall` signal becomes true when an instruction in the ID stage uses the result of an `lw` instruction which is in the EXE stage. Thus, the stall signal can be generated by the following Verilog HDL code.

```
stall = ewreg & em2reg & (ern!=0) & (i_rs & (ern== rs) | i_rt & (ern
== rt));
```

where `i_rs` and `i_rt` indicate that an instruction uses the contents of the rs register and the rt register respectively.

There is an important thing we must not to forget. The pipeline stall is implemented by prohibiting the updates of the PC and the IF/ID pipeline register. But the instruction that is already in the IF/ID register will be decoded and fed to the next pipeline stage. This will result in an instruction being executed twice. To prvent an instruction from being executed twice, we must cancel the first instruction. Canceling an instruction is easy: prevent it from updating the states of the cpu and memory.

All the control signals that will be used in the following stages are saved into the ID/EXE registers.

In the EXE stage, in addition to the operation performed by the ALU, the PC+8 operation is carried out by an adder for generating the return address for the `jal` instruction. The shift amount `(sa)` for a shift

instruction can be extracted from the immediate field `(eimm)`. If the instruction in the EXE stage is a `jal`, PC+8 is selected and the destination register number `(ern)` is set to 31 (done by `f` component). Otherwise, the ALU output is selected and let ern=ern0 (`rd` or `rt` in the EXE stage).

In the MEM stage, if the instruction is an `sw`, the data `mb` will be written into the data memory addressed by `malu`. If the instruction is an `lw`, the memory data addressed by `malu` is read out. Other instructions do nothing in this stage.

In the WB stage, an instruction is graduated by writing the result, either the ALU result or memory data, into a register file. The destination register number is `wrn` (register number in the WB stage). And the write enable signal is `wwreg` (register write enable in WB stage)

10. Test Program and Simulation Waveform

Write a Verilog code that implement the following instructions to verify the correctness of your pipelined CPU design. The code should be used to initialize the instruction memory block. The register file should be all initialized to zeros. In the test program, it is aimed to check the 20 instructions. The main part of the test program is a subroutine in which four 32-bit memory words are summed by a `for` loop. After returning from the subroutine, the sum is stored in the data memory by a `sw` instruction. A code pattern that causes pipeline stall is also prepared within the loop. Word address is used to assign the content of each word (a 32-bit instruction). The parenthesized hexadecimal number in the center of each line is the byte address (PC).

```
Module instruction_memory (a,inst);          // instruction memory, rom
    input  [31:0] a;                  // rom address
    output [31:0] inst;               // rom content = rom[a]
    wire   [31:0] rom [0:63];         // rom cells: 64 words * 32 bits
    // rom[word_addr] = instruction   // (pc) label   instruction
    assign rom[6'h00] = 32'h3c010000; // (00) main:   lui  $1, 0
    assign rom[6'h01] = 32'h34240050; // (04)         ori  $4, $1, 80
    assign rom[6'h02] = 32'h0c00001b; // (08) call:   jal  sum
    assign rom[6'h03] = 32'h20050004; // (0c) dslot1: addi $5, $0,  4
    assign rom[6'h04] = 32'hac820000; // (10) return: sw   $2, 0($4)
    assign rom[6'h05] = 32'h8c890000; // (14)         lw   $9, 0($4)
    assign rom[6'h06] = 32'h01244022; // (18)         sub  $8, $9, $4
    assign rom[6'h07] = 32'h20050003; // (1c)         addi $5, $0,  3
    assign rom[6'h08] = 32'h20a5ffff; // (20) loop2:  addi $5, $5, -1
    assign rom[6'h09] = 32'h34a8ffff; // (24)         ori  $8, $5, 0xffff
    assign rom[6'h0a] = 32'h39085555; // (28)         xori $8, $8, 0x5555
    assign rom[6'h0b] = 32'h2009ffff; // (2c)         addi $9, $0, -1
    assign rom[6'h0c] = 32'h312affff; // (30)         andi $10,$9,0xffff
    assign rom[6'h0d] = 32'h01493025; // (34)         or   $6, $10, $9
    assign rom[6'h0e] = 32'h01494026; // (38)         xor  $8, $10, $9
    assign rom[6'h0f] = 32'h01463824; // (3c)         and  $7, $10, $6
    assign rom[6'h10] = 32'h10a00003; // (40)         beq  $5, $0, shift
    assign rom[6'h11] = 32'h00000000; // (44) dslot2: nop
    assign rom[6'h12] = 32'h08000008; // (48)         j    loop2
    assign rom[6'h13] = 32'h00000000; // (4c) dslot3: nop
    assign rom[6'h14] = 32'h2005ffff; // (50) shift:  addi $5, $0, -1
    assign rom[6'h15] = 32'h000543c0; // (54)         sll  $8, $5, 15
    assign rom[6'h16] = 32'h00084400; // (58)         sll  $8, $8, 16
    assign rom[6'h17] = 32'h00084403; // (5c)         sra  $8, $8, 16
    assign rom[6'h18] = 32'h000843c2; // (60)         srl  $8, $8, 15
    assign rom[6'h19] = 32'h08000019; // (64) finish: j    finish
    assign rom[6'h1a] = 32'h00000000; // (68) dslot4: nop
```

```
    assign rom[6'h1b] = 32'h00004020;  // (6c) sum:    add  $8, $0, $0
    assign rom[6'h1c] = 32'h8c890000;  // (70) loop:   lw   $9, 0($4)
    assign rom[6'h1d] = 32'h01094020;  // (74) stall:  add  $8, $8, $9
    assign rom[6'h1e] = 32'h20a5ffff;  // (78)         addi $5, $5, -1
    assign rom[6'h1f] = 32'h14a0fffc;  // (7c)         bne  $5, $0, loop
    assign rom[6'h20] = 32'h20840004;  // (80) dslot5: addi $4, $4,  4
    assign rom[6'h21] = 32'h03e00008;  // (84)         jr   $31
    assign rom[6'h22] = 32'h00081000;  // (88) dslot6: sll  $2, $8, 0
    assign inst = rom[a[7:2]];         // use 6-bit word address to read rom
endmodule
```

below is the test data that should be stored in the data memory. Four 32-bit words in the memory will be read by `lw` instructions. The test program will store a word in the location next to the four words.

```
module data_memory (clk,dataout,datain,addr,we); // data memory, ram
    input         clk;                    // clock
    input  [31:0] addr;                   // ram address
    input  [31:0] datain;                 // data in (to memory)
    input         we;                     // write enable
    output [31:0] dataout;                // data out (from memory)
    reg    [31:0] ram [0:31];             // ram cells: 32 words * 32 bits
    assign dataout = ram[addr[6:2]];      // use 5-bit word address
    always @ (posedge clk) begin
        if (we) ram[addr[6:2]] = datain;  // write ram
    end
    integer i;
    initial begin                         // ram initialization
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data          // (byte_addr) item in data array
        ram[5'h14] = 32'h000000a3;        // (50) data[0]   0 +  a3 =  a3
        ram[5'h15] = 32'h00000027;        // (54) data[1]  a3 +  27 =  ca
        ram[5'h16] = 32'h00000079;        // (58) data[2]  ca +  79 = 143
        ram[5'h17] = 32'h00000115;        // (5c) data[3] 143 + 115 = 258
        // ram[5'h18] should be 0x00000258, the sum stored by sw instruction
    end
endmodule
```

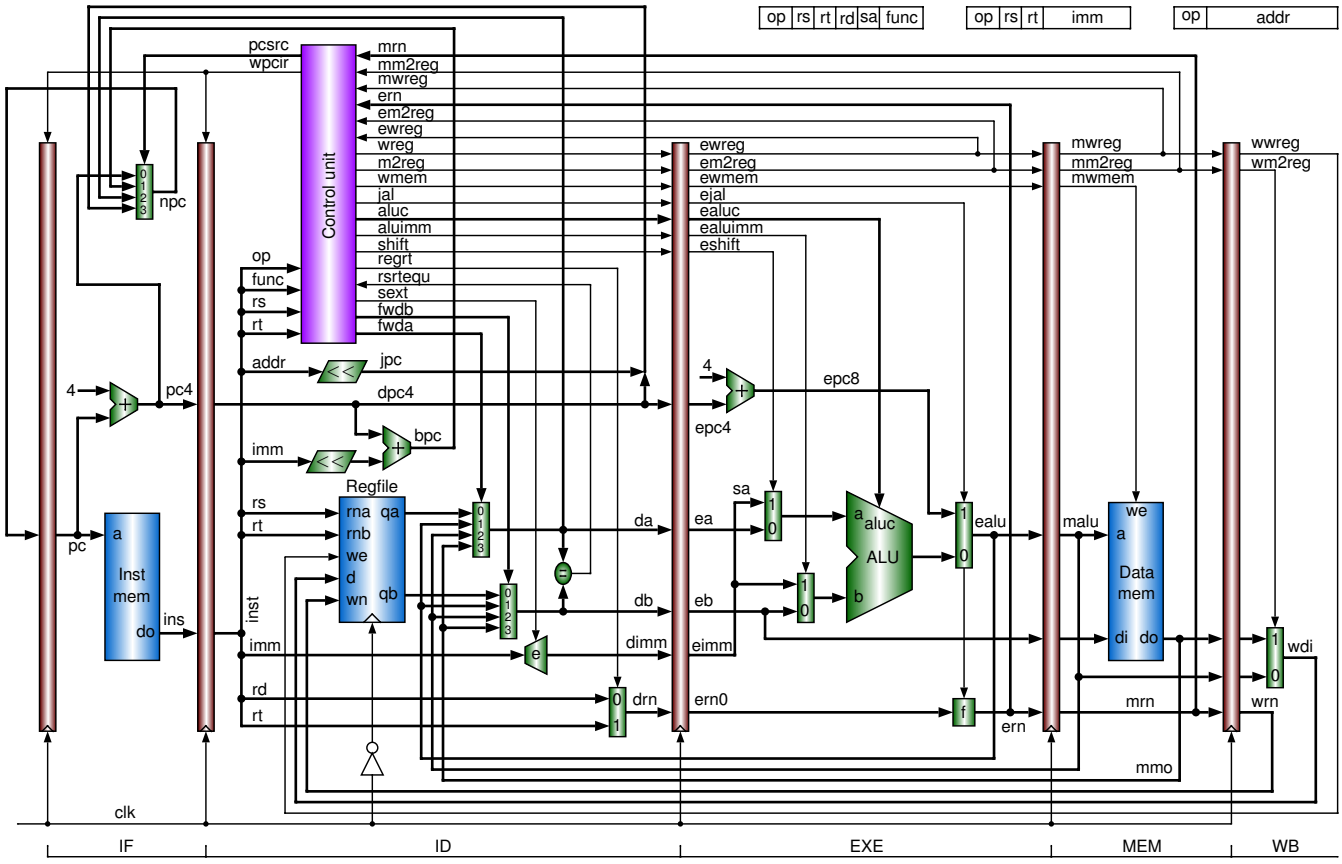

Figure 6 Detailed circuit of the pipelined CPU

Figure 7 illustrates an example of waveforms when the pipelined CPU executes the jal instruction (PC = 0x00000008). The instruction in the delay slot (PC = 0x0000000c) is executed also. The taget address of the jal instruction is 0x0000006c, the entry of a subroutine (sum). The result at the EXE stage of the jal instruction is 0x00000010, which is the return address (from the subroutine).

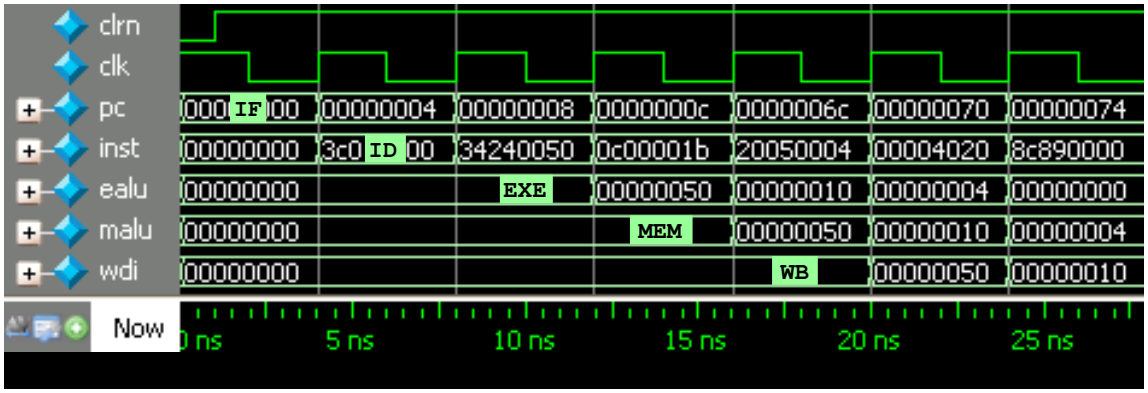

Figure 7 Waveform of the pipelined CPU (call subroutine)

11. Write a Verilog code that implement the instructions shown in item number 8 with the corresponding initialization of data memory using the design shown in **Figure 6**. You need to show your outputs in a similar way as figure 7 with the same signals when the pipelined CPU execute the `lw $9, 0($4)` instruction (PC =

0x00000070) and its follow up, the `add $8, $8, $9` instruction in the fourth (last round) of the `for` loop.

12. Write a report that contains the following:
    - **viii.** Your Verilog design code. Use:
        i. Device: Zyboboard (XC7Z010- -1CLG400C)
    - **ix.** Your Verilog® Test Bench design code. Add "`timescale 1ns/1ps" as the first line of your test bench file.
    - **x.** The waveforms resulting as requested from item 9 above.
    - **xi.** The design schematics from the Xilinx synthesis of your design. Do not use any area constraints.
    - **xii.** Snapshot of the I/O Planning and
    - **xiii.** Snapshot of the floor planning
    - **xiv.** The design should be free from errors when synthesized.

13. **REPORT FORMAT:** Free form, but it must be:
    a. One report per student.
    b. Have a cover sheet with identification: Title, Class, Your Name, etc.
    c. You have to write an abstract at the beginning of the project report to describe what you have done in the project.
    d. Use Microsoft word and it should be uploaded in word format not PDF. If you know LaTex, you should upload the Tex file in addition to the PDF file.
    e. Single spaced

14. You have to upload the whole project design file zipped with the word or LaTex with PDF file.