Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

– CO2007 Computer Architecture – Spring 2019 –

Design Single Clock Cycle MIPS Processor

Ngo Duc Tuan - 1710364
Ho Le Thuc Quyen - 1752454
Truong Le Vinh Khoa - 1752298

June 11, 2019

Contents

1 Introduction

1.1 Product Introduction

In this project, we designed a single clock cycle MIPS processor of 32-bits architecture based on Verilog HDL and synthesized on DE2i-150 board.

1.2 Platform

Figure 1: ModelSim and De2i-150 board

2 Design

2.1 Function

Our MIPS processor can execute the following instructions:

1. Arithmetic instruction (R-format): add, sub, and, or, slt, mul.
2. Arithmetic instruction (I-format): addi.
3. Data transfer instruction (I-format): lw, sw.
4. Condition branch instruction (I-format): beq.
5. Jump instruction (J-format): jump.

Besides, to handle with exceptions, our circuit have an EPC module that can stall the system when exception occurs.

2.2 Inputs

- Instruction:
  To test the function of this circuit, we create a piece of MIPS code including all types of instruction above and write them to module Instruction Memory before synthesize.

- Clock:
  When simulate on ModelSim, we created a clock signal and test on the waveform. When simulate on FPGA board, we used a button to create clock, every time it is pressed and released, a single clock is generated.

- Reset:
  We create a reset signal to initialize the system, reset PC to 0.

2.3 Outputs

- 8 7-segments LEDs:
  Display 32-bits output of sub-modules depends on four switches. Every combinations of these switches is corresponding to an output of a specified module.

- 10 red LEDs:
  Display the control signals of each instruction.

- LCD:
  Display 32-bits output of sub-modules on the first row.
  Display value of program counter (PC) on the second row.

2.4 Schematic

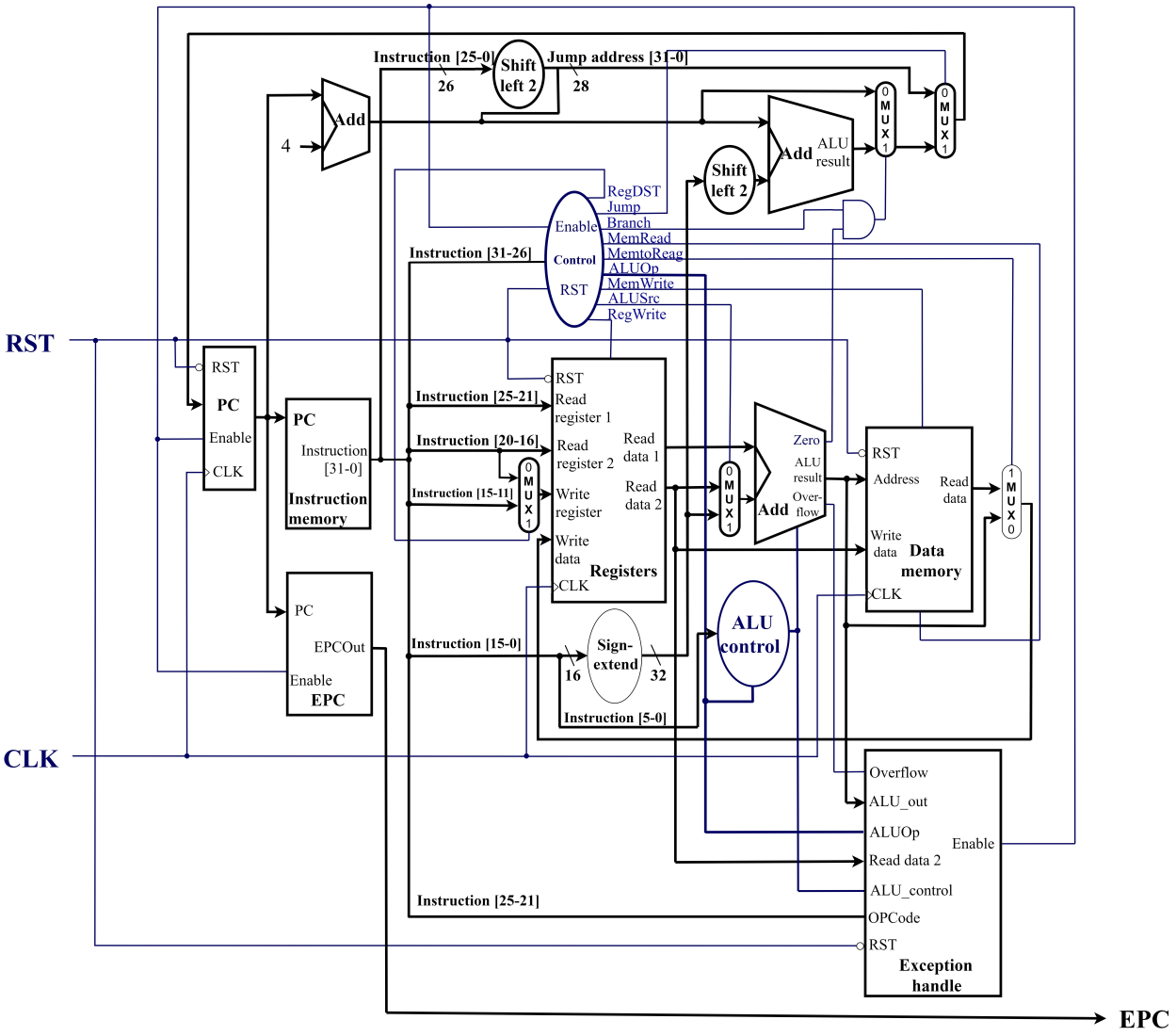

Figure 2: Schematic

3 Implementation

3.1 PC

- **Purpose**
  The main function of this module is to return the **PC_out** which is used to fetch the current instruction.

- **Operation**
  When **PC_out** just releases this module, it is transfered to Instruction Memory module to fetch the instruction. Simultaneously, it is added by 4 **PC_in** and return to the

PC module as its input and wait for the next positive clock to fetch next instruction. In addition, this module will be halted whenever there is an exception handle signal and is initialized to -4 to fetch the first instruction.

3.2 Instruction Memory

- **Purpose**
  Based on the value of **PC_out**, the address of instruction,this module will return the current instruction to execute.

- **Operation**
  Since each instructions is separated by the unit of 4 address, the address of each instruction is the number that can be divided by 4.
  Therefore, with the value of input, in order to get the offset of instruction, we divide it by 4.Then use this offset to load instructions from register.

3.3 Register

- **Purpose**
  Read and load the data in registers to execute or store new data, the result of executed instructions to it.

- **Operation**
  Initially, the **rst** signal will reset all the registers to its initial state.

  The value of register rs and rt will be read and transmit to **Read data 1** and **Read data 2** simultaneously whenever there is an instruction comes to this module. After that with the next positive edge clock, new instruction will be fetched and simultaneously, depending on the signal **RegWrite** from the previous instruction, whether the module will store the **Write data** to the **Write Register** or not.

3.4 Sign extend

- **Purpose**
  Extend the first 16 bits in instructions to 32 bits.

- **Operation**
  First, we check whether the bit 16 of input instruction is 0 or 1. if this bit is 0 it means that this value is positive then we fill the last 16 bits by 0. Otherwise, this value is negative and we will fill the last 16 bits by 1.

3.5 EPC

- **Purpose**
  this module will save the address of instruction which causes the exception handle.

- **Operation**
  This module take **PC_out** and **Enable** signal, a signal will be active high when there is an exception handle as its inputs. Whenever there an **Enable** signal is active high, this module will return the **PC_out**.

3.6 Exception handle

- **Purpose**
  Detect all the cases that cause the program be error.
  There are three cases that can raise the exception handle signal.

  1. The dividend is 0.
  2. Invalid memory address. Since in this architecture, we just consider load and store word which deals with the address can be divided by 4. Therefore, if we meet offsets which is not the divisor of 4, it will be cause the loading wrong word.
  3. Overflow when we deal with some arithmetics add and multiply between 2 values.

- **Operation**
  First, in order to solve the problem with divisor is 0. We check both **read data 2** and **ALU Control**. If the read data2 is equal 0 and the **ALU Control** is 4'b1111, it means that this arithmetic is division and the dividend is 0, the exception handle signal will be active high.

  Second, for the invalid address, we check **ALU Op** and the first 2 bits of ALU_out, whether both the first 2 bits of ALU_out is equal 0, it means this number is the divisor of 4, and **ALU Op** is for load and store word or not. If it meet these cues, the module will return Enable high.

  And the last is wait for the overflow signal from **ALU_Cal** to conclude whether there is an overflow or not.

3.7 Main Control

- **Purpose**
  Decode instruction to create control signals for submodules of the processor.

- **Operation**
  It function as a decoder that takes the opcode (instruction[31-26]) as input then generate control signals: **RegDST, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite**.

3.8 ALU Control

- **Purpose**
  Decide which operation is calculated in ALU_Cal module
- **Operation**
  The combinational condition between two input, **ALUOp** from Control module and **function code** from instruction, results in the 4-bit output **ALU Control** corresponding to the operation.

3.9 ALU Cal

- **Purpose**
  Execute the operation corresponding to instruction.
- **Operation**
  Take two 32-bit **data inputs** and **ALU Control** to execute and the results is stored in 32-bit output **ALU_out**. The **Zero** output shows whether the result equals to 0 or not. The **Overflow** output checks whether the results exceed 32-bits or not.
- **Submodules**
  Adder_32bits, Two_Complement, Multiply.

3.10 Data Memory

- **Purpose**
  Execute data transfer instructions: read data from memory and write data from register to memory.
- **Operation**
  We create 20 32-bits memory words to execute two instruction: lw and sw. When **MemWrite** is 1, it will write data to memory at positive edge of clock signal and when **MemRead** is 1, it will assign data from memory to output.

3.11 LCD

DE2i-150 board control LCD display through **LCD_DATA, LCD_EN, LCD_RW, LCD_RS** signals. Every time we load a new instruction or change the switches, the LCD will reload and update new screen. Therefore, it will write 37 9-bits data, including 5 commands and $16x2 = 32$ characters every time the display changes.
The process to write each character or command is described as diagram below:

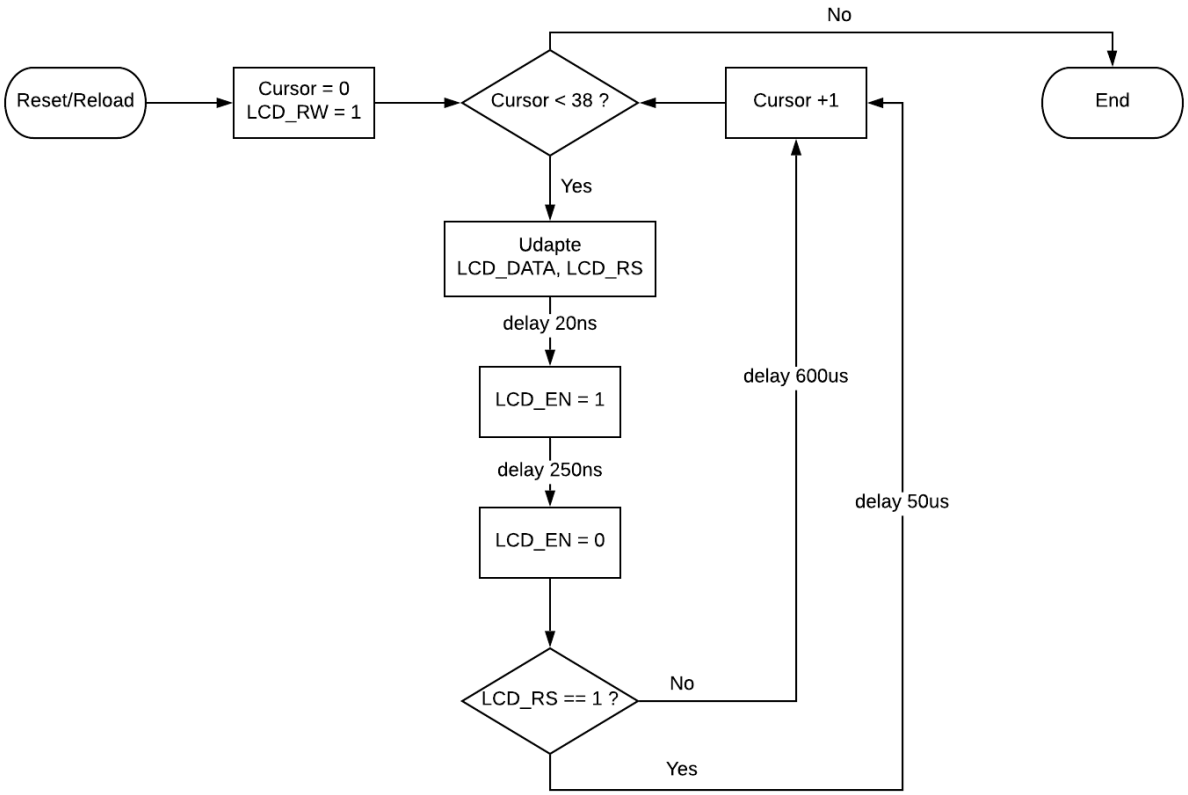

Figure 3: LCD process

4 Results

4.1 Simulation in ModelSim

Initially, after pressing reset signal, all the values of registers in is declared to be its register' number. For instance, the value of registers $s0, $s1, $s2, $s3, $s4, $s5 is 0, 1, 2, 3, 4, 5, respectively.

And for the memory, we have about 20 slots, and each slot is a word with 4 bytes. Therefore,

from the slot 0 to 9, each value is the number of its memory's slot. And for the slot 10 to the rest, all of them will be declare as 0.

No.	Instruction	No.	Instruction
1	add $t0, $s1 , $s2	9	slt $t5, $s0, $s1
2	sub $t1, $t0 , $s2	10	beq $s1, $s2, instruction 1
3	and $t0, $s1 , $s2	11	jump to instruction 14
4	or $t0, $s1 , $s2	12	add $t0, $s1 , $s2
5	lw $t0, 6($s2)	13	sub $t0, $s1 , $s2
6	sw $s5, 39 ($s1)	14	mul $t0, $s3, $s2
7	lw $t3, 37($s3)	15	beq $s1, $s1, instruction 1
8	addi $s2, $s1 ,15		

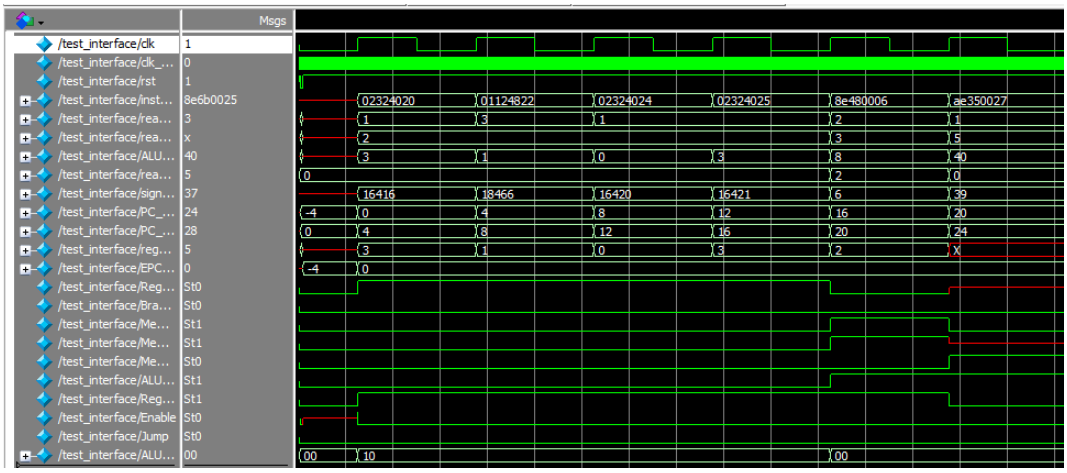

Figure 4: Instructions 1-6

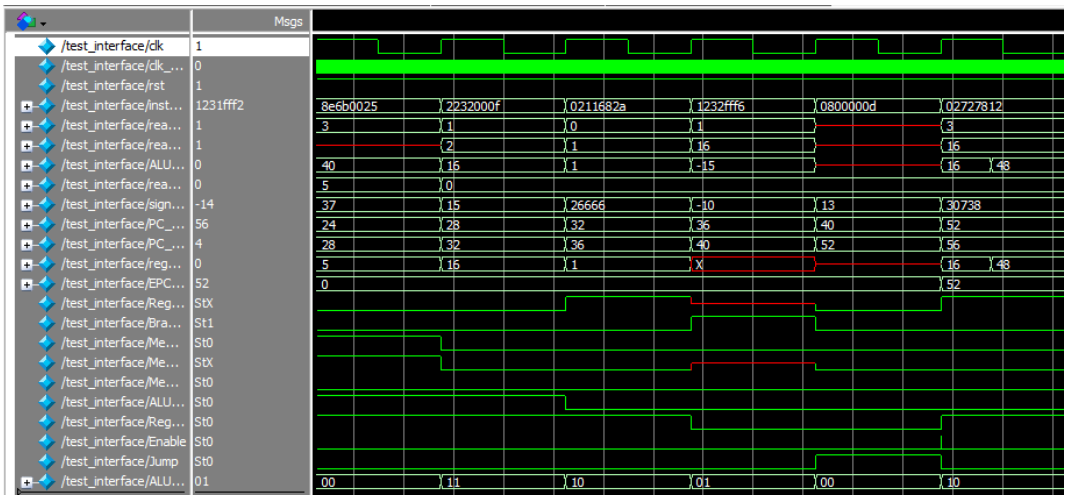

Figure 5: Instructions 7-11 and from instruction 11 jump to instruction 14

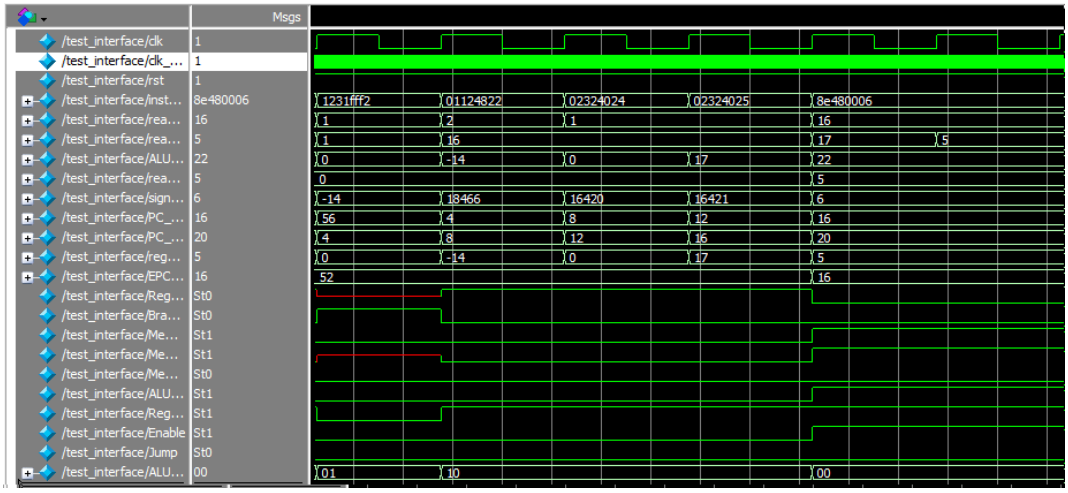

Figure 6: Jump again to instruction 1

4.2 Test on board De2i-150

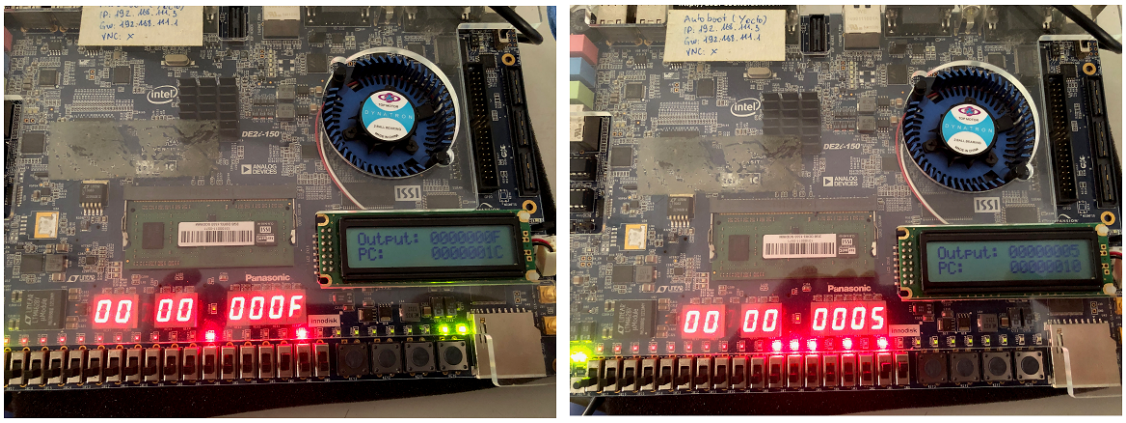

Figure 7: Result on board

5 Discussion

5.1 Conclusion

After 6 weeks working together, we have finished design a single clock cycle MIPS processor based on Verilog HDL. Our processor can execute 11 basic instructions (R,J,I-format) including basic arithmetic operator, data transfer, conditional and unconditional branch. To verify the result, we have created testbench on ModelSim to check the waveform as well as synthesize on De2i-150 board and use 7-segment LEDs, LCD to check the outputs.

5.2 Future work

- Create pipeline model of MIPS processor.
- Modify the ALU to execute more operations.