Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

ENCS3310 - Advanced Digital Design

Course Project Report :

Simple Microprocessor

Sarah Hassouneh - 1210068

Instructor: Dr. Abdellatif Abu-Issa

Section: 2

Date: 20 Jan 2024

Abstract

In this project we designed a simple microprocessor system. The purpose of the microprocessor is to execute simple instructions including arithmetic instruction and logical instructions and output the correct result. The microprocessor consists of two main units: the ALU and the register file. The ALU is the functional unit that executes the specific operation specified in the instruction on the specific operand s also denoted in the instruction. The reg-file is a memory module that has holds the operands we want to work on. This register file can perform two read operations and one write operation and is used to fetch operand and save results. In this project we examine how these modules were built, tested and how efficient are they.

Table of Contents

Table of Figures

Introduction

The project we want to build here demonstrates the importance hardware description languages and why we use them. We are designing a digital circuit but with these languages we are able to design, test and improve.

The main design in our system will be referred to as top module or (mp_top)in the Verilog code. This design main purpose is to receive instructions and execute them. It consists of the ALU and a reg-file unit. The design is a synchronous logic with a clock input. It also has the main instruction input and the result output.

The ALU unit performs the arithmetic and logical operations that are basically the main functions of any CPU or computer in general. Tn reg file is a small internal memory of this microprocessor. It is similar to some extent to the RAM but different in that the reg_file specified in our design has 3 addresses inputs and outputs and one read input. It is unlike normal RAM in that it can process three addresses at the Output 1 produces the item within the register file that is address by Address 1. Similarly Output 2 produces the item within the register file that is address by Address 2. Input is used to supply a value that is written into the location addressed by Address 3.The initial values in the memory were specifies by the provided table.

The procedure that we took was design each of these modules and test it on its own and then combining them and creating a main testbench. This report gives an explanation of the method used, design procedure, challenges and an efficiency overview al last .

Procedure

To build the simple microprocessor, first, the two main components : ALU unit and Register file unit were built and tested. Then we combined instances of them in our top design. This section will explain how the components were built and tested.

Part 1: The ALU

The ALU module was built using behavioral Verilog description. It is a combinational circuit and our design calls for two 32-bit inputs, A and B, and a 32-bit output Result. My id student ends in 8, so the opcodes shown in (
Table 1) were used, we have 11 different operations in the system. For any unary operations operand A was used, otherwise both operands A and B were used. The logical operations are done bitwise between the two operands.

Add	Subtract	Absolute value	Negative	Max	Min	Avg	Not	Or	And	Xor
1	6	13	8	7	4	11	15	3	5	2

Table 1: Opcodes

Design notes

1. **Parameters:**

The code for the ALU module is shown in (Figure 6), for better configurability, parameters were used for both data width (which represent the size of inputs A and B), and opcode length (which is the size of opcode input). This provides a way to make the code more flexible, reusable, and easy to maintain especially when thinking about upscaling our current design. Parameters were also used to reference the opcodes, the opcodes we are using in the design are constants, i.e. if (add) has an opcode of 1, then it will stay 1 for that instance through the entire design. Parameters allow us to provide meaningful names for those constants, and if we want to change opcode for any operation later it can be easily changed. The use of parameters can be seen through the whole design and has been effectively used in all modules.

2. **Finding average:**

A note for finding the average, we basically summed the two values and shifted it left by 1, this is equivalent for dividing by 2 and taking the result to be integer part of that division.

Testing ALU

To test our ALU module, a simple test bench was created, for efficient testing we had to test all 11 opcodes. (Figure 7) shows part of the test bench and test case 1, we defined parameters similar to the ALU module, inputs were put inside registers and wires for the outputs, we also defined an expected register to compare the result to. Test case 1 shows how the (add) operation was tested, operands A and B were set, the expected value of the addition was saved in expected, then it was compared with the result ,and then prints if pass or fail. (Figure 8) shows the result of executing all test cases both in waveform and on console. It shows that in all cases, the expected output matches the result. So, we can conclude that ALU module is correct. Please note that the output in waveform are in signed decimal, but in the console we printed hexadecimal for logical operations and decimal for arithmetic for easier checking.

Part 2: The Register File

The Register file we want to create in our design is like a very small fast RAM. For our system we designed 32 x 32-bit words, so we used a 5-bit address lines and 32-bit data lines. This register file can process three addresses at the same time, two are read operations, and one is written to. Output 1 produces the item within the register file that is address by Address1, Output 2 produces the item within the register file that is address by Address 2, and Input is used to supply a value that is written into the location addressed by Address 3.

Design notes

1. **The clock:**

If we design register file as a combinational circuit without clock, it may cause problems when the write address is the same as one of the read addresses (race condition). This problem can be solved by synchronizing the register file to a clock. This means that on the rising edge of the clock the two outputs will be produced, and the input will be written. We designed this in reg_file simple(Figure 9), it can be seen that on the rising edge of the clock the out1 and out2 are provided based on the corresponding addresses, and (in) is saved in the memory in the provided address (addr3), note that all of this is done in one clock cycle.

Now, did we solve the issue of reading and writing when the address for the input is the same as the address of one of the outputs?

No, with this current implementation of the register file(simple reg file) the issue is not solved. We created a test bench to test that (will be explained in detail in next part), and the test bench showed having a race condition when the input address is same as output address. Simply adding the clock has not changed the issue, because still on the rising edge the system is trying to read and write from same position at the same time. **So how can this be solved?** After some through reading and research, we concluded that it is inevitable to have a race condition if I am trying to read and write at the same exact time. So one approach that I used is creating a synchronous two state machine. We split reading in a state and writing in the following state to prevent race condition. The reg-file now needs two clock cycles to complete reading and writing, the first clock cycles outputs are read from the memory and the second clock cycle (in) is saved in the right address.

Now, this design is powerful for two reasons. First, it allows us to read and write at the same addresses. Second, it means that the outputs out1 and out2 (that will later be used as operands) are ready(fetched) by the end of the first cycle and writing can be done at the following clock cycle with no issues, the improvement in code is shown in (Figure 10).

2. **Enabling the register file**

The register is sensitive to changes in any of inputs (addresses and in) and so we want to create a way to control that through an enable signal. This is important especially in controlling writing because we don't want garbage values to be written in unintended places in memory , therefore losing the data.

The enable signal in our design in denoted (valid_opcode) because its value is dependent on the opcode value as will be seen in top module .

3. **Task to initialize memory**

Tasks can be used in organizing and structuring code and creating better modular designs. Here a simple task was used to initialize memory (shown in Figure 10), which basically

assigns memory with initial register values (as per given in the provide table). This allows for easier access and modification for any value initial value in memory, later.

Testing Register file

To test the register file, we basically want to test 4 things: the reading, the writing, the enable signal (valid_opcode) and reading and writing to same address. The test cases in the test bench that was designed is shown in (Figure 11). We basically designed four test cases:

1) In the first test case we set valid code to 1 to enable the reg file and then read from a two known addresses (0,1) and wrote on address(2). The out1 and out2 outputs were checked to ensure that their values match the actual addresses that were saved in the memory inside reg file. It was found that it matches so the test case passed. Note, that here we always test outputs after 2 clock cycles because a complete writing and reading action that was designed in this reg-file takes two clock cycles as was explained earlier.
2) In the second test case we wanted to ensure that a proper writing action was done in the first test case. So again, we read from the same position we wrote on (that is address number 2). We compared the output and found that it matches what we saved in, so test case passed.
3) In the third test case, we wanted to test the enable signal(valid opcode), to test that we set valid opcode to 0, then changed other inputs. We want to compare if any of the outputs change or if they remain stable. In our test case they remain unchanged and so that is correct and test case passed.
4) In the fourth test case, we mainly wanted to test race condition. From our design we know that this could mainly happen when the write address is same as read. So, we set addr2 and addr3 to be 15 at the same time and set in to be 100. We then tried to read the value at 15, we are expecting to get 3600 but if race condition happened we would get 100. In our test case we got 3600, so there was no race condition. Furthermore, to ensure correctness we also re-read the value at 15 again(after two clock cycles) and it was 100 which is the value we wrote in (in).

   (Figure 12) shows the result of running the testbench in waveform and console.

   It is important to note that the test bench that was created here was used to also test our original implementation of register file(reg_file_simple that reads and write in one clock cycle). While testing the simple register file, Test case 4 (that test the race condition would always fail even though we have a clock in the register (we would get 100 instead

of 3600). **So, for building the top module, reg_file that uses two clock cycles was used to ensure correctness and prevent race condition.**

Part 3: The Top module

The top module we designed here represent the core of the microprocessor, we essentially used the ALU and the register file and connected them in a similar manner to what is shown in (Figure 1).

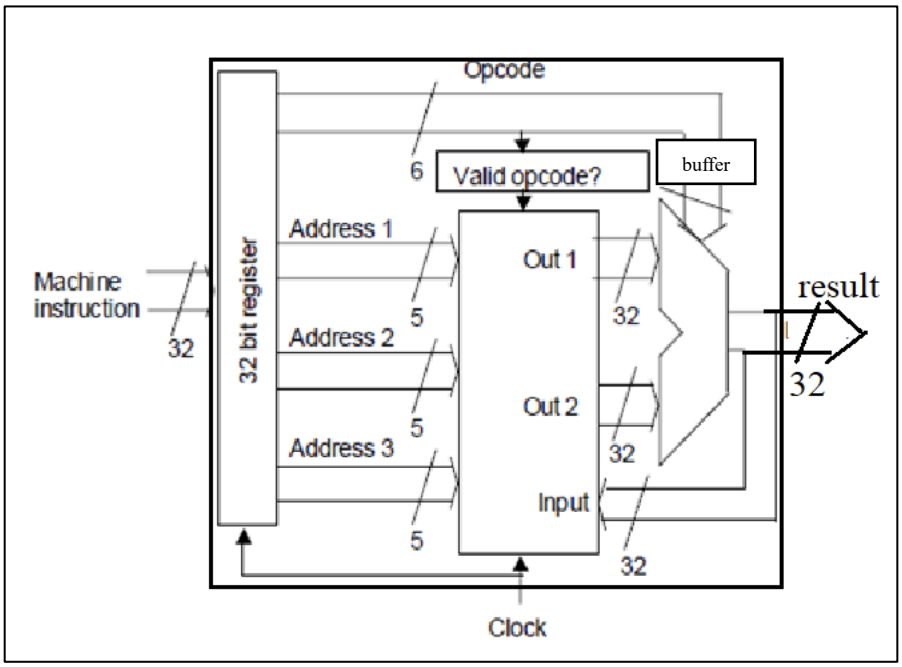

Figure 1: Top Module Design

Our top module is responsible for handlining different machine instruction. It a general manner it follows the instruction life cycle, it first fetch the instruction, decodes it , fetches operands, perform operation (calculate output), then it writes result to memory.

The code for the top module is shown in(Figure 13). We can say that the system that was designed consists of four main components as follows:

1) The ALU unit : An instance from the ALU module we built in part1. It is responsible for the actual execution of the instruction and outputting result.
2) The register file: An instance from the reg_file module we built in part2. It represents main memory where the operands will be fetched from. The register file is synchronized with the clk input of the top module, so when there is a rising edge in the top module, there is the same rising edge in the reg_file.

3) Internal Register: The registers are necessary for connecting all the components together and ensure correct flow of execution . Registers include (opcode, addr1, addr2, addr3, is_valid, alu_output, out1, out2, buffer). As for the register_file its inputs are is_valid, addr1, addr2, addr3 and alu_output (as the (in) argument). For the ALU, its inputs are out1, out2 (that represents operands A, B respectively), the buffer that is connected to the opcode input of ALU and the alu_output register is connected to the result output of ALU.
4) Buffer : Buffer can be seen simply as a register. The importance of it lies in synchronizing the components, especially the ALU combinational circuit with the register file sequential circuit. In the ALU unit, as will be seen in the next section, the opcode usually arrive before the register values (operands), thus we added buffer (with one clock cycle delay) for the opcode.

The result of top module is assigned to be the (alu_output) at all times.

A demonstration of how it works

The system we designed here is a sequential circuit that has two states. The first state is the read state where we fetch the operands from memory to perform calculation on. The second state is the write state where we write the result to the memory.

To demonstrate how the system work, we will take this example of performing a simple instruction to add elements at 1 and 2 and store output in 3. The following waveform (Figure 2) shows the changes in the values at each clock cycle:

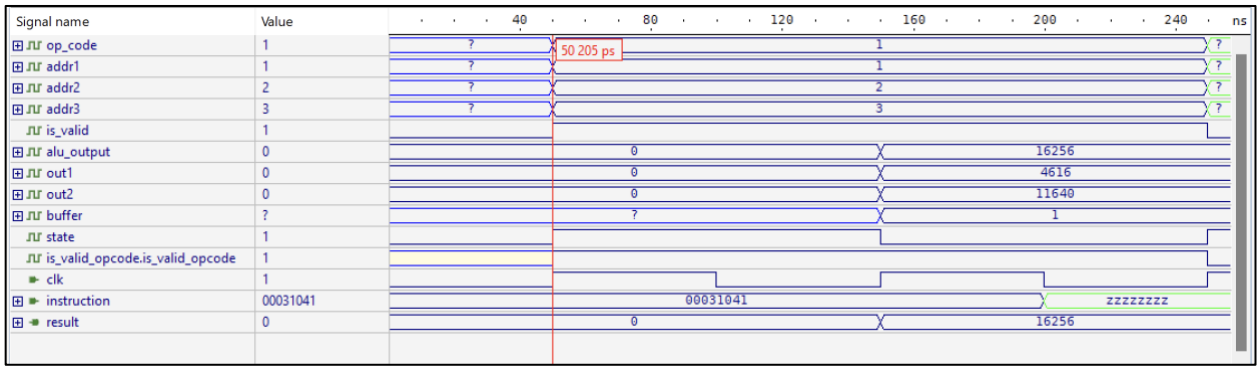

Figure 2: Demonstration of add operation

At (state =0) : At the first rising edge of the clock cycle (at 50ns), we notice that the instruction is read and decoded (using a task that was created- discussed in next section). As a result of decoding, the opcode register now holds value of 1 (equivalent to add opcode), addr1 has the address of first operand, addr2 has the address of second operand and addr3 has the address of where the result would be saved. Now the opcode is checked if it is valid or not, in this case it is so the valid opcode reg is set to 1. Setting this to 1 means that it enables the register file to start its reading process. The register file starts reading form memory and by the end of the first clock cycle / start of the next the operands are fetched and ready. Now state is set to 1. Now note hardware is inherently parallel, so all the latter is happening at the same time (so state 0 changes to 1 as soon as the posedge clk is encountered, it doesn't wait for decoding the instruction or checking opcode).

At (state =1) At the second rising edge of the clock cycle (at 150ns), we notice that the operands are ready in out1 and out2. In this state we set the value of buffer(which is the register connected to ALU opcode input) to the actual value of the opcode. Since ALU is a combinational circuit we can notice as soon as the buffer value is set to 1(add opcode), the result shows in the alu_output and thus the result. Since the reg_file is synchronized which the top_module, in the second clock cycle the reg_file starts writing the result to the memory and by the end of the second clock cycle the output is written to the reg_file memory. And now the state is reset to 0.

Design notes

1. **The use of tasks and functions:**

Tasks and functions in behavioral modeling provide more readability and easier code management. We know tasks are more general than functions and that functions can only model purely combinational calculations. We utilized that in our design by creating a task to decode instruction and a function to determine if the opcode is valid or not. These are shown in the (Figure 3) below:

```
547
548     // Task to decode; fill addresses registers and opcode register from instruction
549     task decode_instruction;
550
551         op_code  = instruction[5:0];
552         addr1       = instruction[10:6];
553         addr2       = instruction[15:11];
554         addr3       = instruction[20:16];
555
556         // remaining bits are unused
557     endtask
558
559     // function to find if valid opcode or not
560     function is_valid_opcode;
561             input [5:0] op_code;
562
563         // parameter of alu opcodes to check if valid
564         parameter add=1, sub=6,abs=13, negative_a=8,max=7, min=4,avg=11, not_a=15, or_ab=3, and_ab=5, xor_ab=2;
565
566         // Check if the opcode matches using a case statement
567         case (op_code)
568             add:        is_valid_opcode = 1;
569                 sub:        is_valid_opcode = 1;
570             abs:        is_valid_opcode = 1;
571                 negative_a: is_valid_opcode = 1;
572                 max:        is_valid_opcode = 1;
573                 min:        is_valid_opcode = 1;
574                 avg:        is_valid_opcode = 1;
575                 not_a:      is_valid_opcode = 1;
576                 or_ab:      is_valid_opcode = 1;
577                 and_ab:     is_valid_opcode = 1;
578                 xor_ab:     is_valid_opcode = 1;
579
580                 default:    is_valid_opcode = 0; //  zero for all other codes
581         endcase
582
583     endfunction
584
585
```

Figure 3: Decode Instruction task and Valid Opcode function

The decode instruction task basically fills the 4 registers from the instruction array (op_code, addr1, addr2, addr3) based on the specified bits that was provided in design specification. On the otherhand, the is_valid_opcode function has one input which is the opcode(that was decoded) and return a value of 1 or 0 depending if it is valid or not.

Part 4: Testing The Top module

An important aspect of this project is the testing of the design. A testbench was created it contains an instance of the (module mp_top) that contains an instance register file and an instance of the ALU.

Machine instructions were supplied to the (mp_top) module are in the following format:

• The first 6 bits identify the opcode

• The next 5 bits identify first source register

• The next 5 bits identify second source register

• The next 5 bits identify destination register

• The final 11 bits are unused

(Figure 14) shows a snippet of the testbench code. First, we defined registers (clk and instruction) as inputs to the (mp_top) instance and wire for the result output of the instance. We then defined integers :

1) **i** which is a loop control variable
2) **number_of_testcases**
3) **passed** variable to count how many passed
4) **failed** variable to count how many failed.
5) **decimal** which is a flag to decide if result of operation is displayed in decimal or hexadecimal. For easier reading, instructions that resulted in arithmetic results were printed in decimal format and instructions with logical results were printed in hexadecimal format.

Also, we defined **reg expected** that has the expected value for each instruction. The purpose is to try and make the testbench more generic.

We supplied different machine instructions and tested output inside a loop. At the start of the loop, we first wrote to the console the time (when the instruction arrived) and we wrote the test case number. Then we have used a case statement to specialize each test case (to supply different machine instruction). For the example below(Figure 4), we set the instruction to be the subtraction of element2 from element 1 and put the result in elemnt1. We then set the expected value to the result that we manually calculated to compare with later. We then wrote to the console what the instruction does and formatted it (see Figure 4).

```
632
633         //make a loop for test cases --> I tried to make it generic here using a loop
634         for (i = 1; i < num_test_cases+1; i = i + 1) // note i started from 1
635             begin
636                 // display test case number
637                 $write("\tTime: %5dns >>", $time);
638                 $write("\tTest Case %2d >>", i);
639 
640 
641                 //please note many of the instructions I tested depend on each other, so I used the result of the past operation.
642                 //This is to better check results and ensure correct reading and writing.
643 
644                 //set instruction and print description for each instruction, this part is specilized for each instruction, so I used case statement
645                 case (i)
646                         1:begin
647                             instruction =   32'h11046; // sub: element1 = element1- element2
648                             expected= -32'd7024;
649                             $write( $sformatf ("%-60s", " (sub: element1 = element1- element2) >>")); // write description, sformatf here used to make description align
650                             //print output as decimal
651                             decimal=1;
652                         end
653
```

Figure 4: Test case 1 in top module and loop definition

After the end case statemen, we first wrote the expected output in decimal or hex depending on flag value. We then checked if result matches the expected (that we set for each instruction), if it does we increment passed and if not we increment failed (see Figure 5). The testbench I created has 16 test cases or instructions that were supplied and tested, the description of each and how they were encoded is showed in appendix B.

```
791 
792             // after case statement:
793 
794             // display expeted output and actual output
795             #20ns // instruction needs two clock cycles to execute , so output will be ready after 20 ns
796             if(decimal)
797                 $write(" Expected: %8d , Output: %8d >>  " , $signed(expected), $signed(result));
798             else
799                 $write(" Expected: %8h , Output: %8h >>  " ,expected, result);
800 
801             if(expected == result)
802                 begin
803                     $write(" Test case passed! \n");
804                     passed =passed+1;
805                 end
806             else
807                 begin
808                     $write(" Test case failed! \n");
809                     failed =failed+1;
810                 end
811 
812         end // end for loop
813 
```

Figure 5: Comparing result with expected

At the end of the test bench we printed the summary, **all the test cases that were tried passed successfully**. Note that each instruction in our design takes two clock cycles and the result was compared by the end of the **two clock cycles**.

Design Features

To evaluate any design there are two important questions:

Did we meet design specifications?

To answer the first question, we have designed an efficient top module that has an instruction input, clk input and has a result output. We can clearly see that through the test bench and wave forms that all test cases passed. This means that design is working correctly and efficiently; it takes an instruction, decodes it, successfully calculate result and saves that result in memory. We also made sure to deal with possible race conditions and allowed reading and writing result at the same memory address.

And how efficient is the design?

There are two aspects we can use to answer this : time and system complexity. As for the time, we refer to time of executing each instruction. In our system a full instruction needs two clock cycles to execute, to successfully read and write. This is relatively efficient, two clock cycles is not a heavy or large time, especially considering that we are reading from a memory like module, no pipelining is used and also race condition is prevented. As for the complexity, the system is manageable. It was designed in a modular way, so each component is designed and tested on its own and then they were used to construct the top-module. The top module itself is not complicated because it consists of multiple small components and it is clear to test, view and assemble.

Other good feature is design

The top module and all other modules in design were carefully designed and were made to be flexible. Modifications and scalability were thoughtfully considered . That's why we can see efficient use of parameters and variables. Building the design in a modular way and the testing at each stage allowed us to ensure correctness and create a clear design. Also, the efficient use of tasks and functions allowed the design to be more flexible, readable, and adjustable, especially for future uses and modifications for example using a 64-bit instruction, or changing to 8 addresses bits , or using 25 as opcode for add rather than 1.

Conclusion

In conclusion, this project gave us an overview of the whole design process start to finish. It demonstrated the importance of using hardware description languages to test our design and ensure correctness before implementing the actual hardware design. It was also evident throughout the project how design requirements influence design decisions, for example when designing the register file, we had to design it with the specific inputs and outputs provided. We were able also to investigate potential issues that arise like race condition and tried to find sensible ways to deal with these issues. Additionally, using sequential logic to execute different machine instructions gave us a clearer way of how this process is actually done in a real microprocessor or computer.

Throughout the project we were able to use the powerful feature especially the behavioral Verilog logic and the testbenches to implement different modules and test them. The use of tasks, functions and parameters were also efficiently used in the design. All of this allowed us to eventually create a correct design, that meets the design specifications.

Appendix A : Screen Shots

The ALU:

```
24  /*
25      The ALU full module.
26  */
27  module alu (opcode, a, b, result );
28
29      // define parameters for lengths
30      parameter data_width = 32,opcode_length=6;
31
32      input signed [opcode_length-1:0] opcode;  // inputs and outpust are signed here
33      input signed  [data_width-1:0] a, b;
34      output reg signed  [data_width-1:0] result;
35
36      //define parametrs for operations
37      parameter add=1, sub=6,abs=13, negative_a=8,max=7, min=4,avg=11, not_a=15, or_ab=3, and_ab=5, xor_ab=2;
38
39      //perform correct operation
40       always @(*)
41      begin
42           case (opcode)
43              add:        result = a+b;
44              sub:        result = a-b;
45              abs:        result = (a < 0) ? -a : a;
46              negative_a: result = -a;
47              max:        result = (a > b ) ? a : b;
48              min:        result = (a < b ) ? a : b;
49              avg:        result = (a+b) >> 1; // a+b/2 , we used shift to take only integer part
50              not_a:      result = ~a;
51              or_ab:      result =  a | b; // bitwise operations
52              and_ab:     result = a & b;
53              xor_ab:     result = a ^ b;
54
55              default:    result = 0;
56          endcase
57      end
58
59  endmodule
60
```

Figure 6: ALU code

```
65  module alu_tb;
66
67      // parameters
68      parameter data_width = 32, opcode_length = 6;
69
70      //define parametrs for operations
71      parameter add=1, sub=6,abs=13, negative_a=8,max=7, min=4,avg=11, not_a=15, or_ab=3, and_ab=5, xor_ab=2;
72
73
74      // inputs   of alu
75      reg signed [opcode_length-1:0] opcode;
76      reg signed [data_width-1:0] a, b;
77
78      // outputs of alu
79      wire signed [data_width-1:0] result;
80
81      reg signed [data_width-1:0] expected=0;
82      // Instantiate alu module
83      alu unit (opcode, a , b, result);
84
85      // Test cases
86      initial
87          begin
88                  // Test case 1
89              opcode = add;
90              a = 10;
91              b = 5;
92              expected = 32'd15;
93              #10ns;
94              $write("Time:%0d, Operation: %4d (add), A:%4d, B:%4d, Expected: %4d,  Result: %4d", $time, opcode, a, b,expected, result);
95              if(expected == result)
96                  $display("\t>> Test case passed!");
97              else
98                  $display("\t>> Test case failed!");
99              #10ns;
100
```

Figure 7: ALU testbench-testcase1

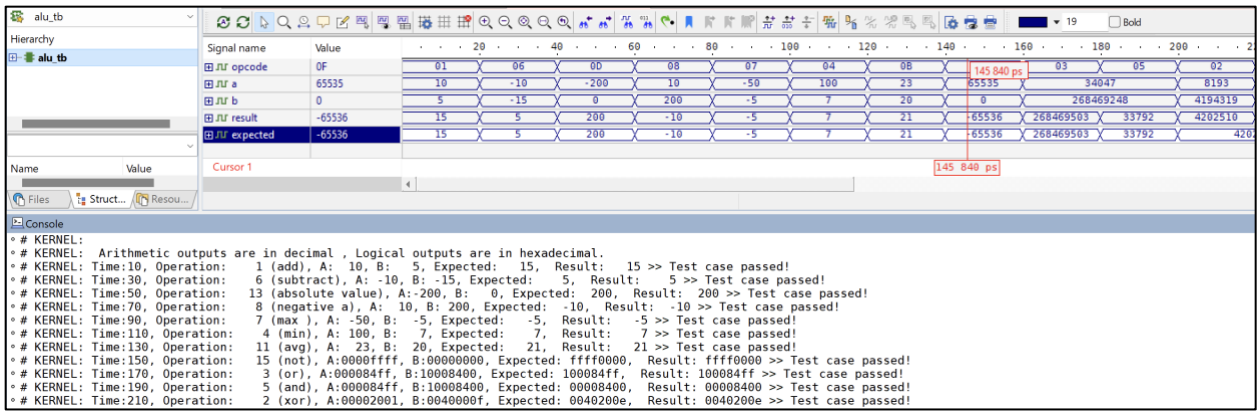

Figure 8: Test Bench for ALU waveform

The reg_file:

```
268 module reg_file_simple (clk, valid_opcode, addr1, addr2, addr3, in , out1, out2);
269
270     // define parameters for the memory
271     parameter data_width = 32, address_width = 5;
272
273     //define inputs in terms of parameters
274     input clk;
275     input valid_opcode;// this is like an enable
276     input [(address_width-1):0] addr1, addr2, addr3;
277     input signed [(data_width -1):0] in;
278     output reg signed [(data_width -1):0] out1, out2;
279
280
281     // define memory in terms of parameters
282     reg [(data_width-1):0] memory [ 0:((1<<address_width)-1)];// equivalent to: reg [31:0] memory [0:31];
283
284
285     initial
286         begin
287                 initialize_memory; // this is a task to intailize memory with given values
288             out1=0;
289             out2=0;
290
291             end
292
293     always @(posedge clk)
294         if(valid_opcode) // if enabled
295             begin
296                 // one clock cycle , handle 3 addresses, note : there is an unexpected behavior(race condition) if addr3 == addr1 or addr3 == addr2
297                 out1 <= memory[addr1];
298                 out2 <= memory[addr2];
299                 memory[addr3] <= in;
300             end
301
```

Figure 9: reg_file_simple code

```
339
340     // we will make read- write operation a two cycle proces, to address issue of read-write on same place in memory
341     reg state=0;
342
343
344     initial
345         begin
346             initialize_memory; // this is a task to intailize memory from given values
347             out1=0;
348             out2=0;
349
350             end
351
352     always @(posedge clk)
353         if(valid_opcode) // if enabled
354             begin
355
356                 if(state==0) // read state
357                     begin
358                         // first clock cycle output operands
359                         out1 <= memory[addr1];
360                         out2 <= memory[addr2];
361                         // go to next state
362                         state=1;
363                     end
364
365                 else     // write state
366                     begin
367                         // second clock cycle write in to memory
368                         memory[addr3] <= in;
369                         // go to intial state
370                         state=0;
371                     end
372
373             end
374
375
376     // Task to initialize memory
377     task initialize_memory; // memory intilized based on given values (look at 6)
378        memory = '{32'd0, 32'd4616, 32'd11640, 32'd11254, 32'd6786, 32'd6784, 32'd12432, 32'd13548,
379                   32'd13462, 32'd13454, 32'd11780, 32'd13170, 32'd2982, 32'd8096, 32'd514, 32'd3600,
380                   32'd10870, 32'd12528, 32'd9860, 32'd6166, 32'd4520, 32'd14436, 32'd12136, 32'd5134,
381                   32'd11958, 32'd7688, 32'd5258, 32'd12420, 32'd3560, 32'd1248, 32'd8724, 32'd0};
382     endtask
```

Figure 10: reg_file two state code

```
415
416     // Start Test cases
417     initial
418         begin
419
420             // Test case 1 : test reading
421             valid_opcode<=1;
422             addr1 <= 0;
423                 addr2 <= 1;
424                 addr3 <= 2;
425             in <= 32'd50;
426             #20ns; // after two clock cycles
427             $display("Time:%0d ns, Valid Opcode: %1d, Addr1: %-2d, Addr2: %-2d, Addr3: %-2d, In: %-8d, Out1: %-8d, Out2: %-8d", $time,
428             valid_opcode, addr1, addr2, addr3, in, out1, out2);
429             if(out1== 32'd0 && out2==32'd4616)
430                   $display("readig correct");
431             else
432                  $display("readig incorrect");
433
434             // Test case 2: test writing
435             valid_opcode<=1;
436             addr1 <= 2; //now out 1 should be 50
437                 addr2 <= 3;
438                 addr3 <= 4;
439             in <= 32'd10;
440             #20ns; // after two clock cycles
441             $display("Time:%0d ns, Valid Opcode: %1d, Addr1: %-2d, Addr2: %-2d, Addr3: %-2d, In: %-8d, Out1: %-8d, Out2: %-8d", $time,
442             valid_opcode, addr1, addr2, addr3, in, out1, out2);
443              if(out1== 32'd50 )
444                   $display("writing correct ");
445             else
446                  $display("writing incorrect");
447
448
449             // Test case 3 : test valid opcode
450             valid_opcode<=0;
451             // check if values don't change with change of addresses and valid is zero
452             addr1 <= 10;
453                 addr2 <= 11;
454                 addr3 <= 12;
455             #20ns; // after two clock cycles
456             $display("Time:%0d ns, Valid Opcode: %1d, Addr1: %-2d, Addr2: %-2d, Addr3: %-2d, In: %-8d, Out1: %-8d, Out2: %-8d", $time,
457             valid_opcode, addr1, addr2, addr3, in, out1, out2);
458              if(out1== 32'd50 &&  out2== 32'd11254 )
459                   $display("valid opcode is correct ");
460              else
461                  $display("valid opcode is incorrect");
462
463
464              // Test case 4 : test reading and writing at same place
465             valid_opcode<=1;
466             // check if values don't change with change of addresses and valid is zero
467             addr1 <= 10;
468             addr2 <= 15;// has 3600
469             addr3 <= 15;
470             in <= 32'd100;
471             #20ns; // after two clock cycles
472             $display("Time:%0d ns, Valid Opcode: %1d, Addr1: %-2d, Addr2: %-2d, Addr3: %-2d, In: %-8d, Out1: %-8d, Out2: %-8d", $time,
473             valid_opcode, addr1, addr2, addr3, in, out1, out2);
474              if(out2== 32'd3600)
475                   $display("correct from same place ");
476             else
477                $display("race condition");
478
479             // read value at 15 again
480             #20ns; // after two clock cycles
481             $display("Time:%0d ns, Valid Opcode: %1d, Addr1: %-2d, Addr2: %-2d, Addr3: %-2d, In: %-8d, Out1: %-8d, Out2: %-8d", $time,
482             valid_opcode, addr1, addr2, addr3, in, out1, out2);
483             if(out2== 32'd100)
484                   $display("correct from same place");
485             else
486                 $display("race condition");
487
488
489             #100;
490             $finish; // terminate simulation
491
492         end
493
```

Figure 11: reg_file test cases

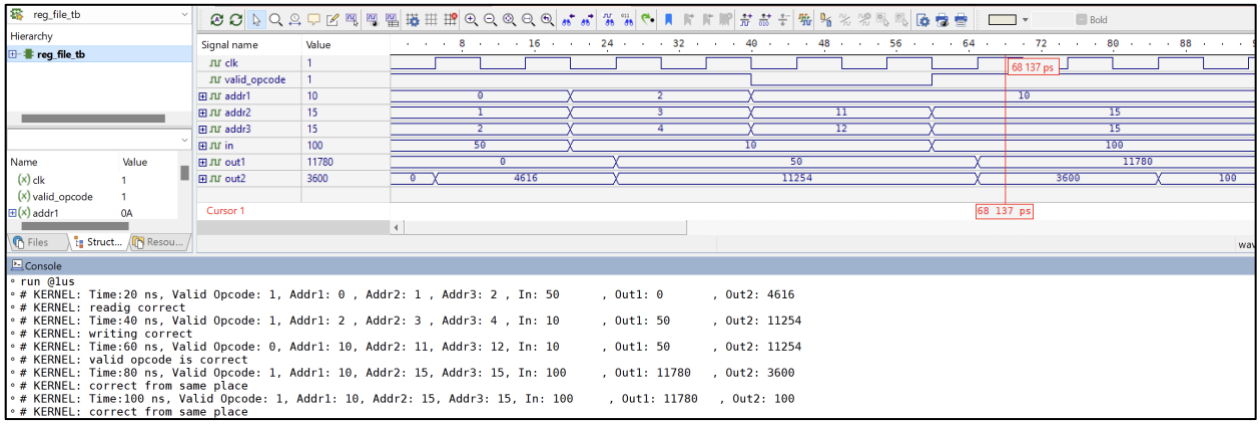

Figure 12: reg_file testbench waveform

The top module:

```
499 module mp_top (clk, instruction , result );
500
501     // define parameters
502     parameter data_width = 32, address_width = 5, opcode_length = 6;
503
504
505     input clk;
506     input [data_width-1:0] instruction; // this is the instruction register
507     output reg [data_width-1:0] result;
508
509     //define these values to connect components together
510     reg [opcode_length-1:0] op_code; // 6 bits for opcode
511     reg [address_width-1:0] addr1, addr2, addr3; // 5 bits for each address
512
513     //inistintiate reg file
514     reg isValid =0;
515     reg [data_width-1:0] alu_output;
516     reg [data_width-1:0] out1, out2;
517     reg_file reg_unit (clk, isValid, addr1, addr2, addr3, alu_output , out1, out2);
518
519     //inistintiate ALU
520     reg [5:0] buffer;// a buffer for the opcode before it goes to ALU
521     alu alu_unit(buffer, out1, out2, alu_output);
522     assign result = alu_output; // the result of alu is always the result
523
524     // make it into a 2 state machine, one state for reading and one state for writing
525     reg  state=0;
526
527
528
529      always @(posedge clk)
530         begin
531
532             if(state==0) // fetch operands state
533                 begin
534                     decode_instruction; // this is a task to decode the instruction
535                     isValid <= is_valid_opcode(op_code); // use this function to check if opcode is valid
536                     state=1; // next state
537                  end
538
539             else if(state==1) // compute the output and write result state
540                 begin
541                     buffer<= op_code;// buffer is the register connected to alu (opcode is also a register )
542                     state=0;// return to intial state
543                 end
544
545         end
546
547
```

Figure 13: top_module code

```
/************** main test bench for top module start here *******/

module tb_mp_top;


    // reg for inputs
    reg clk;
    reg [31:0] instruction;

    // wires for output
    wire [31:0] result;

    // get instance from top
    mp_top top_module (clk, instruction, result);

    // set the clk change values with time
     always #5ns clk = ~clk; // a full clock cycle is 10ns


     // Initial statements
    initial
      begin
                clk = 0;
                instruction = 0;
      end

    // define variable
    integer i=1, passed=0, failed=0,num_test_cases =16;
    reg [31:0] expected;

    integer decimal=0; // this is just a flag for printing

    // Run Test Cases
    initial
        begin
            #5ns;
            instruction =   32'h0;
            #20ns;
            $display("\n Arithmetic outputs are in decimal , Logical outputs are in hexadecimal. ");
            $display("\n Time printed is time when instruction arrived (every instruction needs two clock cyles which equal to 20ns in this testbench).Result is compare

        //make a loop for test cases --> I tried to make it generic here using a loop
        for (i = 1; i < num_test_cases+1; i = i + 1) // note i started from 1
            begin
                // display test case number
                $write("\tTime: %5dns >>", $time);
                $write("\tTest Case %2d >>", i);


                //please note many of the instructions I tested depend on each other, so I used the result of the past operation.
                //This is to better check results and ensure correct reading and writing.

                //set instruction and print description for each instruction, this part is specilized for each instruction, so I used case statement
                case (i)
                        1:begin
                            instruction =   32'h11046; // sub: element1 = element1- element2
                            expected= -32'd7024;
                            $write( $sformatf ("%-60s", " (sub: element1 = element1- element2) >>")); // write description, sformatf here used to make description align
                            //print output as decimal
                            decimal=1;
                        end

                        2:begin
                            instruction = 32'h001F004D; // abs: element 31 = abs value of  element1
                            expected= 32'd7024;
                            $write(  $sformatf ("%-60s"," (abs: element 31 = abs value of  element1 ) >>")); // write description
                             //print output as decimal
                            decimal=1;

                        end
```

Figure 14: top module testbench code snippet

Appendix B : Machine instructions

description	Unused bits (MSB)	addr3 in decimal	addr3 in binary	addr2 in decimal	addr2 in binary	addr1 in decimal	add1 in binary	opcode in decimal	opcode in binary	in binary	in hex	Expected output in decimal	Expected output in hexadesimal
sub: element1 = element1- element2	00000000000	1	00001	2	00010	1	00001	6	000110	00000000000000010001000001000110	00011046	-7024	
abs: element 31 = abs value of element1	00000000000	31	11111	0	00000	1	00001	13	001101	00000000001111100000000001001101	001F004D	7024	
max: element 31 = max(element31,element30)	00000000000	31	11111	30	11110	31	11111	7	000111	00000000001111111111011111000111	001FF7C7	8724	
add: element 23 = element23+ element23	00000000000	23	10111	23	10111	23	10111	1	000001	00000000001011110111011110000001	0017BDC1	10268	
min: element 20 = min(element21, element23)	00000000000	20	10100	23	10111	21	10101	4	000100	00000000001010010111101010000100	0014BD44	10268	
avg: element 14 = avg(element6, element5)	00000000000	14	01110	5	00101	6	00110	11	001011	00000000000111000101001100010011	000E298B	9608	
negative: element 18 = - element18	00000000000	18	10010	0	00000	18	10010	8	001000	00000000001001000000010010001000	00120488	-9860	
and: element 10 = element0 & element11	00000000000	10	01010	11	01011	0	00000	5	000101	00000000000010100101100000000101	000A5805	0	0
or : element 27 = element12 | element11	00000000000	27	11011	11	01011	12	01100	3	000011	00000000000110110101101100000011	001B5B03	15350	3BF6
xor : element 24 = element26 ^ element25	00000000000	24	11000	25	11001	26	11010	2	000010	00000000000110001100111010000010	0018CE82	2690	A82
not: element 2 = ~ element 3	00000000000	2	00010	0	00000	3	00011	15	001111	00000000000000100000000011001111	000200CF	-11255	FFFFD409
sub: element17 = element16- element15	00000000000	17	10001	15	01111	16	10000	6	000110	00000000001000101111100000000110	00117C06	7270	
and: element 6 = element4 & element 5	00000000000	6	00110	5	00101	4	00100	5	000101	00000000000001100010100100000101	00062905	6784	1A80
xor : element 25 = element26 ^ element27	00000000000	25	11001	27	11011	26	11010	2	000010	00000000001100111011110100000010	0019DE82	12156	2F7C
add: element 13 = element7+ element8	00000000000	13	01101	8	01000	7	00111	1	000001	00000000000110101000001110000001	000D41C1	27010	
min: element 16 = min(element15, element13)	00000000000	16	10000	13	01101	15	01111	4	000100	00000000001000001101011110000100	00106BC4		
add: element 3 = element1+ element2	00000000000	3	00011	2	00010	1	00001	1	000001	00000000000000110001000001000001	00031041	16256	

Table 2: Machine instructions used