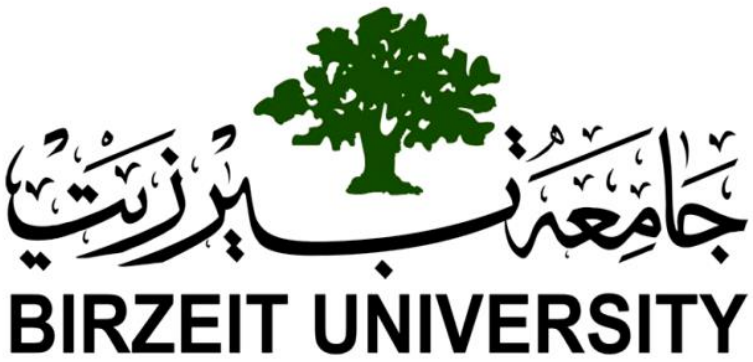

Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Advanced Digital – ENCS3310

Project Report – Design Simple Part Of Microprocessor

Prepared by:

Name: Malak Ammar. **Number:** 1211470.

Instructor: Abdellatif Abu-Issa.

Section: 2.

Date: 18/01/2024.

Abstract

This project focuses on developing a microprocessor in the realm of advanced digital design. The primary objectives involve creating an Arithmetic Logic Unit (ALU) and a Register File, essential components for computational tasks. The ALU processes 32-bit inputs using a customizable 6-bit opcode. The Register File serves as fast memory, featuring clock synchronization and an enable input for robust operation. The aim is to interconnect the ALU and Register File, forming a foundational microprocessor. Machine instructions, represented as 32-bit numbers, control the operations, emphasizing efficiency and performance. Testing is carried out to ensure accurate functionality, with a focus on opcode handling and clock edge synchronization. The project aims for a streamlined yet powerful digital design, with Verilog code, simulation results, and a concise project overview presented in this report.

Table of Contents

Abstract III
Table of figures IV
List of tables V
Theory 5
1- Microprocessor 5
2- Arithmetic Logic Unit (ALU) 5
3- Register File 6
Design Overview 7
1- ALU. 7
ALU Design Philosophy. 8
2- Register File. 9
Register File Design Philosophy. 10
3- Microprocessor. 12
Microprocessor Design Philosophy. 12
Conclusion 18
References 19
Appendix 20

Table of figures

Figure 1 : Simple microprocessor design. _____ *6*
Figure 2: ALU testbench. _____ *8*
Figure 3: ALU testbench results. _____ *9*
Figure 4: Register File testbench. _____ *10*
Figure 5: Register File testbench results. _____ *11*
Figure 6: mp_top testbench. _____ *13*
Figure 7: mp_top testbench results part1. _____ *15*
Figure 8: mp_top testbench results part2. _____ *15*
Figure 9: mp_top testbench results part3. _____ *16*

List of tables

Table 1 : Opcode's operations. _____ 7
Table 2: Instructions details. _____ 14

Theory

1- Microprocessor

A microprocessor is a compact, integrated circuit functioning as the central processing unit (CPU) in electronic devices. Operating on either Reduced Instruction Set Computing (RISC) or Complex Instruction Set Computing (CISC) architecture, it comprises key components such as the Arithmetic Logic Unit (ALU), Control Unit, registers, and a clock that governs its speed. Instructions encoded in machine language dictate tasks like arithmetic operations and data movement. Microprocessors interact with various memory levels, from fast registers to storage devices, and often employ pipelining to enhance performance. Some processors also leverage parallelism through multi-core designs or SIMD instructions for simultaneous instruction execution. Communication with other components occurs through a bus system, facilitating data transfer. Modern microprocessors incorporate power management features like dynamic voltage scaling for energy efficiency. In essence, microprocessors serve as the computational powerhouse, executing instructions swiftly and efficiently in electronic devices. [1].

2- Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a fundamental component within a microprocessor responsible for executing arithmetic and logical operations. Functioning as the computational core of the processor, the ALU performs tasks such as addition, subtraction, multiplication, division, and logical comparisons. It operates on binary data, manipulating bits based on the instructions provided. The ALU is critical in enabling the microprocessor to carry out mathematical computations and make logical decisions, playing a central role in the execution of instructions. Its efficiency directly impacts the overall processing capability of the microprocessor, making it a pivotal element in the computational power of electronic devices. [2].

3- Register File

The Register File is a vital component within a microprocessor, serving as a small, high-speed storage area for temporary data and instructions. Comprising multiple registers, each capable of holding a specific amount of binary data, the register file plays a crucial role in the processor's operations. Registers are used for swift access to frequently used data and for holding intermediate results during computations. They facilitate efficient data manipulation and enable the processor to quickly retrieve and store information. The size and organization of the register file impact the overall performance of the microprocessor, influencing factors such as speed and efficiency in data processing. In essence, the Register File acts as a rapid-access workspace, optimizing the execution of instructions within the microprocessor. [3].

In summation, the microprocessor, harmonizing the capabilities of the ALU and Register File, stands as a sophisticated computational entity. The ALU's prowess in executing diverse operations and the Register File's adeptness in managing memory collectively contribute to the microprocessor's robust functionality, underscoring the intricate interplay of computation and memory within this advanced digital design. [1].

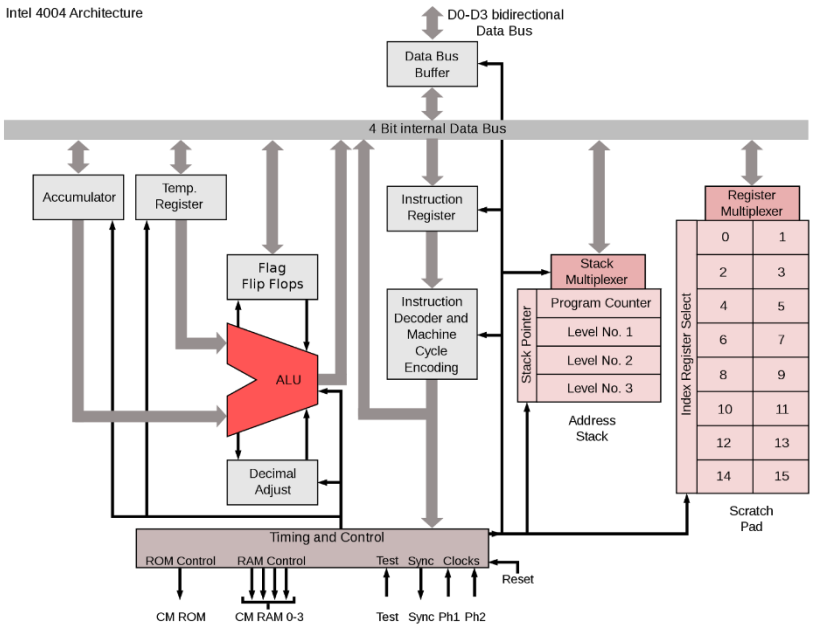

Figure 1: Simple microprocessor design.

Design Overview

The design of the microprocessor revolves around two key components: the Arithmetic Logic Unit (ALU) and the Register File. These components are carefully crafted to collaboratively perform a variety of operations, providing a foundation for computational tasks.

1- Arithmetic Logic Unit (ALU)

The ALU is the computational powerhouse of the microprocessor. It takes two 32-bit inputs, A and B, and produces a 32-bit output, Result. The operations performed by the ALU are dictated by a 6-bit opcode, a unique identifier determined by the last digit of the student's ID, since my ID is **1211470**, then I used the opcodes of (**0**) column. The assigned opcodes dictate operations such as addition, subtraction, absolute value, negation, maximum, minimum, average, bitwise operations, and logical operations, as the following table.

Operation	Opcode
A+B	8
A-B	9
|A|	2
-A	10
Max(A,B)	12
Min(A,B)	1
Avg(A,B)	13
~A	5
A OR B	4
A AND B	11
A XOR B	15

Table 1: Opcode's operations.

➢ ALU Design Philosophy

The ALU (Arithmetic Logic Unit) has been meticulously crafted using a behavioral design approach, ensuring a modular and organized structure. Each computational operation, dictated by a 6-bit opcode, is encapsulated within dedicated tasks, allowing for a clear and intuitive representation of the code. To address the challenge of handling negative numbers, the 'signed' data type has been strategically employed for both task inputs and outputs. This choice ensures the ALU's ability to execute arithmetic operations accurately, considering the signed nature of its operands. The implementation is augmented by individual tasks for various operations, such as addition, subtraction, absolute value, negation, maximum, minimum, average, and logical operations. The design's robustness is verified through a simple yet effective test bench, wherein input values are provided, and the resulting outputs are observed for each opcode. The utilization of tasks enhances readability and facilitates future expansions or modifications to the ALU's functionalities. Overall, the ALU operates seamlessly, dynamically responding to opcode changes and successfully executing a diverse set of operations while handling negative numbers with precision.

```
//testbench for alu module
module ALUTB;
    reg [31:0] a, b;
    reg [5:0] opcode;
    wire [31:0] result;

    integer i;

    alu unit(opcode, a, b, result);

    initial
        begin

            #0 a=0; b=0; opcode=0; //initiallize

            $monitor("Time %0d  A=%b  B=%b  opCode=%b  Result=%b", $time, a, b, opcode, result);

            #1 a=-4; b=-3; opcode=8; //add -4 to -3
            #1 a=-4; b=3; opcode=9; //subtract -4 & 3
            #1 a=-1; opcode=2;      // abs of -1
            #1 a=4; opcode=10; //negative of a=4
            #1 a=-8; b=1; opcode=12; //max of a=-8 & b=1
            #1 a=4; b=-4; opcode=1;  //min of a=4 & b=-4
            #1 a=-4; b=4; opcode=13; //avg of a=-4 & b=4
            #1 a=-1; opcode=5;       //not a=-1
            #1 a=0; b=-1; opcode=4;  //a=0 or b=-1
            #1 a=16; b=1; opcode=11; //a=16 and b=1
            #1 a=1; b=-1; opcode=15; //a=1 xor b=-1
        end
endmodule
```

Figure 2: ALU testbench.

- Expected Results
  - $-4 + -3 = -7 \rightarrow In\ binary;\ 11111111111111111111111111111001$
  - $-4 - 3 = -7 \rightarrow In\ binary;\ 11111111111111111111111111111001$
  - $|-1| = 1 \rightarrow In\ binary;\ 00000000000000000000000000000001$
  - $-(4) = -4 \rightarrow In\ binary;\ 11111111111111111111111111111100$
  - $\max(-8,1) = 1 \rightarrow In\ binary;\ 00000000000000000000000000000001$
  - $\min(4,-4) = -4 \rightarrow In\ binary;\ 11111111111111111111111111111100$
  - $avg(-4,4) = 0 \rightarrow In\ binary;\ 00000000000000000000000000000000$

- $\sim(-1) = 0 \rightarrow In\ binary;\ 00000000000000000000000000000000$
- $or(0,-1) = -1 \rightarrow In\ binary;\ 11111111111111111111111111111111$
- $and(16,1) = -7 \rightarrow In\ binary;\ 00000000000000000000000000000000$
- $xor(1,-1) = -2 \rightarrow In\ binary;\ 11111111111111111111111111111110$

- Results

```
Time 0  A=00000000000000000000000000000000  B=00000000000000000000000000000000  opCode=000000  Result=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Time 1  A=11111111111111111111111111111100  B=11111111111111111111111111111101  opCode=001000  Result=11111111111111111111111111111001
Time 2  A=11111111111111111111111111111100  B=00000000000000000000000000000011  opCode=001001  Result=11111111111111111111111111111001
Time 3  A=11111111111111111111111111111111  B=00000000000000000000000000000011  opCode=000010  Result=00000000000000000000000000000001
Time 4  A=00000000000000000000000000000100  B=00000000000000000000000000000011  opCode=001010  Result=11111111111111111111111111111100
Time 5  A=11111111111111111111111111111000  B=00000000000000000000000000000001  opCode=001100  Result=00000000000000000000000000000001
Time 6  A=00000000000000000000000000000100  B=11111111111111111111111111111100  opCode=000001  Result=11111111111111111111111111111100
Time 7  A=11111111111111111111111111111100  B=00000000000000000000000000000100  opCode=001101  Result=00000000000000000000000000000000
Time 8  A=11111111111111111111111111111111  B=00000000000000000000000000000100  opCode=000101  Result=00000000000000000000000000000000
Time 9  A=00000000000000000000000000000000  B=11111111111111111111111111111111  opCode=000100  Result=11111111111111111111111111111111
Time 10  A=00000000000000000000000000010000  B=00000000000000000000000000000001  opCode=001011  Result=00000000000000000000000000000000
Time 11  A=00000000000000000000000000000001  B=11111111111111111111111111111111  opCode=001111  Result=11111111111111111111111111111110
```

Figure 3: ALU testbench results.

The ALU's efficacy is confirmed through a thorough test bench, where manually derived expected results align precisely with computed outcomes. This manual verification solidifies the reliability of the ALU's design and affirms its accuracy across diverse computational scenarios. With manual expectations seamlessly matching observed results, the test unequivocally passes, validating the ALU's successful implementation within the microprocessor architecture.

2- Register File

The Register File, a pivotal element in the microprocessor architecture, has been designed to meet the dynamic demands of modern processors. Its Verilog implementation prioritizes clock-synchronized memory operations to ensure consistency and reliability. Governed by a rising clock edge, read operations extract data from specified addresses (`addr1` and `addr2`), while write operations update the memory content at the designated address (`addr3`) with the provided input value. The `valid_opcode` input serves as an enable signal, orchestrating the Register File's responsiveness. When set to 1, the file actively processes read and write operations; otherwise, it remains dormant, disregarding inputs. The memory array, encompassing 32 x 32-bit words, provides substantial storage capacity. Initialization of the Register File aligns with the second-from-last digit of the student's ID, since my ID is **1211470**, then I used the data in (**7**) column, ensuring distinctive initial memory states. Overall, the Register File's design harmonizes clock synchronization, address handling, and enable input to establish an efficient and integral component within the microprocessor.

➢ Register File Design Philosophy

Employing a behavioral design methodology, the Register File operates seamlessly on the positive edge of the clock, executing read and write operations through blocking assignments. This approach ensures the file's efficiency in handling memory operations within the microprocessor. The pivotal `valid_opcode` input acts as an enable signal, exclusively triggering the Register File when set to 1. This strategic utilization prevents unintended operations and guarantees a controlled response during read and write processes. Notably, the memory data is stored in hexadecimal values, showcasing the file's versatility in handling diverse data types. The successful validation of the Register File's functionality is demonstrated through a purposefully designed test bench. This test bench, tailored to assess read and write functionalities, effectively verifies the Register File's accurate operation under diverse scenarios. These affirmative test results not only attest to the Register File's functional reliability but also highlight the efficacy of its behavioral design in managing critical memory operations within the microprocessor architecture.

```
//testbench for regFile module
module regFileTB;
    reg [31:0] in;
    reg [4:0] addr1, addr2, addr3;
    reg clk, vaild_opcode;
    wire [31:0] out1, out2;

    reg_file unit(clk, vaild_opcode, addr1, addr2, addr3, in, out1, out2);

    initial
        begin
            #0ns clk=0;
            repeat(10)
            #2ns clk=~clk;
        end

    initial
        begin

            #0ns in=32'h11114321; addr1=0; addr2=0; addr3=0; vaild_opcode=0; //since enable is zero -> nothing to do

            $monitor("Time %0d  inData=%b  address1=%b  address2=%b  address3=%b  CLK=%b  vaild_opcode=%b  out1=%b  out2=%b", $time,
            in, addr1, addr2, addr3, clk, vaild_opcode, out1, out2);

            #5ns vaild_opcode=1; addr1=4'b0001; //enable is 1 -> it must read the data in add1 (out1), then reads the data in addr2 (out2)
            //then write on addr3 the input data
            #5ns in=32'h00004321; addr1=4'b0000; addr3=4'b0001; addr2=4'b0001;
            #5ns addr1=4'b0001;

        end

endmodule
```

Figure 4: Register File testbench.

- Expected Results

In the Register File test bench, the initial phase sets `addr1`, `addr2`, and `addr3` to zeros, and `valid_opcode` to zero, rendering the Register File inactive. Consequently, during this phase, `out1` and `out2` are anticipated to be don't cares due to the disabled state. After a 5ns delay, the activation of `valid_opcode` to one trigger the initiation of Register File operations. Specifically, `addr1` is set to one, facilitating the retrieval of data stored in memory at address 1, thereby populating `**out1**` with the value (**32'h00003ABA**). Simultaneously, `addr2` is set to zero, leading to the retrieval of data at address 0 (value: **0**) into `**out2**`. Significantly, during this clock cycle, `addr3` is configured to write the input data (**32'h11114321**) into address 0.

In the subsequent positive clock edge, synchronization becomes apparent. `addr1` is set to read the newly written data at address 0 (**32'h11114321**) into `**out1**`, showcasing the synchronous handling of read operations. Concurrently, `addr3` is directed to store data (**32'h00004321**) in address 1, while `addr2` reads data from address 1 (value: (**32'h00003ABA**)) into `**out2**`. This orchestrated sequence highlights the synchronization mechanism, ensuring that read and write operations occur coherently and without conflicts. The synchronization in the Register File design plays a pivotal role in maintaining data integrity and consistency. It guarantees that the microprocessor's memory operations unfold in a controlled manner, preventing potential hazards such as data corruption or read-after-write inconsistencies. Therefore, the synchronization strategy implemented in this module contributes to the overall reliability and predictability of the microprocessor architecture.

- Results

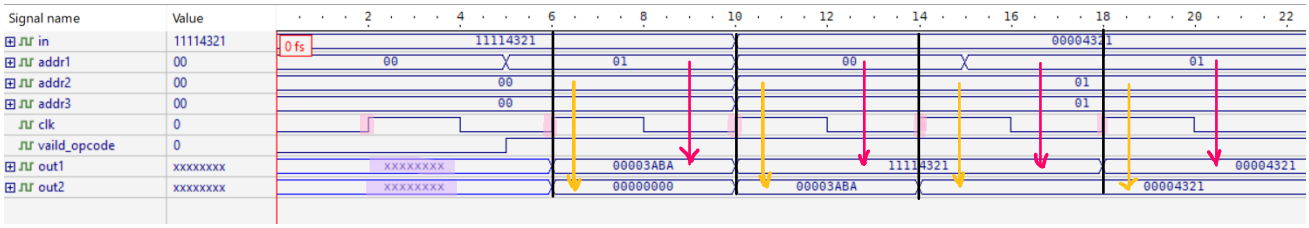

Figure 5: Register File testbench results.

The waveform results from the Register File test bench exhibit a clear correspondence with the expected outcomes, confirming the accurate functionality of the module. Notably, the synchronization operations, integral to read and write actions, align seamlessly with expectations, underscoring the precision and reliability of the microprocessor's memory operations. This robust synchronization mechanism not only ensures the precise execution of operations but also safeguards against potential conflicts or inconsistencies, thereby fortifying the overall integrity of the microprocessor architecture. With both waveform results and synchronization operations in harmony with expectations, the test conclusively passes, validating the Register File's impeccable functionality within the microprocessor.

3- Microprocessor (mp-top)

The `**mp_top**` module, representing the core of the microprocessor, orchestrates the seamless integration of the ALU and the Register File to form a cohesive and functional processing unit. In this pivotal design, machine instructions are supplied as 32-bit numbers, with the first 6 bits identifying the opcode, the next 5 bits indicating the source registers, and the subsequent 5 bits specifying the destination register. The remaining 11 bits remain unused. The microprocessor's operation hinges on these instructions, driving the ALU and Register File interactions. The clock synchronization ensures a controlled and ordered execution of instructions, with the positive edge triggering the processing of each command. The enable signal directs the Register File's responsiveness, activating it only when a valid opcode is present, thereby optimizing data integrity.

➢ Microprocessor Design philosophy

Structured in a well-defined manner, the `mp_top` module serves as the central unit of the microprocessor, harmonizing the interplay between the ALU and Register File. Operational sequences are initiated on the positive edge of the clock, a critical event orchestrating the execution of machine instructions. The design intricacies include the instantiation of the `**loadInst**` module, meticulously extracting and analyzing `addr1`, `addr2`, `addr3`, and the opcode from the 32-bit machine instruction. To align with the microprocessor's architecture, the 11 most significant bits of the opcode are systematically set to zero, optimizing resource utilization. In a strategic move to maintain synchronization with the program flow, a one-cycle delay, implemented through a buffer (D-flip flop), ensures temporal alignment before the opcode’s engagement with the ALU module. The synchronization strategy is pivotal, preventing conflicts and discrepancies during the data flow. Notably, a crucial check for the validity of the inserted opcode is employed, holding the result until a valid opcode is provided. Once a valid opcode is confirmed, the data inputs are seamlessly directed to the ALU, triggering the execution of operations. The resultant output from the ALU becomes the input data for the subsequent operation within the Register File, establishing a coherent processing cycle. This meticulous structural design, complemented by synchronization strategies, guarantees the efficient and reliable execution of diverse machine instructions within the microprocessor architecture.

```
//testbench for top module to check output results
module mpTopTB;
    //define opcodes as local constants
    `define ADDITION 6'b001000 `define SUBTRACTION 6'b001001 `define ABS 6'b000010
    `define NEG 6'b001010 `define MAX 6'b001100 `define MIN 6'b000001 `define AVG 6'b001101
    `define NOTA 6'b000101 `define OR 6'b000100 `define AND 6'b001011 `define XOR 6'b001111

    //memory register to store values of size 32X32, 32 memory location, each consists of 32-bits
    reg [31:0] memory [0:31] = '{32'h00000000, 32'h00003ABA, 32'h00002296, 32'h000000AA, 32'h00001C3A, 32'h00001180,
    32'h000022E0, 32'h00001C86, 32'h000022DA, 32'h00000414, 32'h00001A32, 32'h00000102, 32'h00001CBA, 32'h00000CDE, 32'h00003994,
    32'h00001984, 32'h000028C4, 32'h00002E70, 32'h00003966, 32'h0000227E, 32'h00002208, 32'h000011B4, 32'h0000237C,
    32'h0000360E, 32'h00002722, 32'h00000500, 32'h000016B6, 32'h0000029E, 32'h00002280, 32'h00003B52, 32'h000011A0, 32'h00000000};

    reg [31:0] instruction;
    reg clk;
    wire [31:0] result;
    reg signed [31:0] out1, out2; //memory values
    reg [4:0] addr1, addr2;

    integer i; //counter
    reg flag=1; //Flag to check if the expected value matched the generated one

    //array of instructions
    reg [31:0] instructions [0:10] = '{32'h00000808, 32'h000008C9, 32'h000000C2, 32'h000000CA, 32'h0000314C, 32'h00003141, 32'h0000288D,
    32'h00000085, 32'h0000FEC4, 32'h0000FECB, 32'h0000FECF};

    reg [31:0] expectedResult; //register to store the expected result

    //instantiate unit to operate the top module
    mp_top unit(clk, instruction, result);

    //initial to clock the system
    initial
        begin
            #0ns clk=0;
            repeat(30)
            #10ns clk=~clk;
        end

    //initial to generate the output & expected reult of each instruction
    initial
        begin
            $display("--------------------------------WELCOME TO MY SIMPLE MICROPROCESSOR--------------------------------\n");

            #0ns instruction=32'h00000000; clk=0; //initially set the instruction to zero
            #10ns //10ns delay to avoid don't cares period -first two cycles-

            //display command
            $monitor("TIME %0d  INSTRUCTION=%b  CLK=%b  RESULT=%b  EXPECTED RESULT=%b", $time, instruction, clk, result, expectedResult);

            for(i=0; i<12; i++) //for loop to operate the result of each instruction
                begin
                    #20ns instruction=instructions[i];

                    addr1=instruction[10:6]; //get the address of the first source register
                    addr2=instruction[15:11]; //get the address of the second source register
                    out1=memory[addr1];
                    out2=memory[addr2];


                    case(instruction[5:0]) //case to generate the expected result
                            `ADDITION : expectedResult=out1+out2;
                            `SUBTRACTION : expectedResult=out1-out2;
                            `ABS :
                                if(out1<0)
                                    expectedResult=-1*out1;
                                else
                                    expectedResult=out1;

                            `NEG : expectedResult=-1*out1;
                            `MAX :
                                if(out1>=out2)
                                    expectedResult=out1;
                                else
                                    expectedResult=out2;
                            `MIN :
                                if(out1<=out2)
                                    expectedResult=out1;
                                else
                                    expectedResult=out2;
                            `AVG : expectedResult=(out1+out2)/2;
                            `NOTA : expectedResult=~out2;
                            `OR : expectedResult=out1|out2;
                            `AND : expectedResult=out1&out2;
                            `XOR : expectedResult=out1^out2;
                            default : expectedResult=expectedResult;
                        endcase

                        if(expectedResult!=result) //if any result don't match the generated one then test fails
                            flag=0; //set falg to zer --> test fails
                        //test fails
                        if(flag==0)
                             $display("THIS CASE -----------------> TEST FAILS :(\n");
                        else //test passes
                            $display("THIS CASE -----------------> TEST PASSES :)\n");


                end

                $display("-------------------------------------------------------------------------------");

                //test fails
                if(flag==0)
                     $display("\nFOR THIS PROGRAM -----------------> TEST FAILS :(\n");
                else //test passes
                    $display("\nFOR THIS PROGRAM -----------------> TEST PASSES :)\n");

                $display("-------------------------------------------------------------------------------\n");

                $finish (250ns); //stop the simulation after 250ns (after finishing the instructions)

        end

endmodule
```

Figure 6: mp_top testbench.

- Expected Results

In crafting the test bench for the microprocessor, I adhered to the specified requirements by generating an array of instructions that encompassed at least one for each opcode as the following table. This comprehensive array served as the foundation for systematically testing the microprocessor's functionality across diverse machine instructions. Each instruction, tailored to a specific opcode, was meticulously designed to examine the microprocessor's response and performance under various operational scenarios. This systematic approach aimed to validate the robustness and reliability of the microprocessor's design under different conditions and operational requirements.

Instruction	Operation	Destination Register	Expected Result
32’h00000808	ADD addr0, addr1	Addr0	32’h00003ABA
32’ h000008C9	Sub addr3, addr1	Addr0	32’hFFFFC5F0
32'h000000C2	Absolute addr3	Addr0	32’h000000AA
32'h000000CA	Negative addr3	Addr0	32’hFFFFFF56
32'h0000314C	Max addr5, addr6	Addr0	32’h000022E0
32'h00003141	Min addr5, addr6	Addr0	32’h00001180
32'h0000288D	Avg addr2, addr5	Addr0	32’h00001A0B
32'h00000085	Not addr4	Addr0	32’hFFFFE3C5
32'h0000FEC4	OR addr27, addr31	Addr0	32’h0000029E
32'h0000FECB	AND addr27, addr31	Addr0	32’h00000000
32'h0000FECF	XOR addr27, addr31	Addr0	32’h0000029E

Table 2: Instructions details.

❖ **Data of the used addresses**

- Addr0 → 32’h00000000
- Addr1 → 32’h00003ABA
- Addr2 → 32’h00002296
- Addr3 → 32’h000000AA
- Addr4 → 32’h00001C3A
- Addr5 → 32’h00001180
- Addr6 → 32’h000022E0
- Addr27 → 32’h0000029E
- Addr31 → 32’h00000000

- Results

Despite rigorous efforts to ensure functionality, the constructed test bench for the `**mp_top**` module revealed unexpected synchronization errors. Despite comprehensive searches and analyses, pinpointing the root cause of these synchronization discrepancies proved elusive. Notably, the anticipated output for each set of inputs did not align with the observed results, indicating a breakdown in synchronization. However, it is noteworthy that the output produced by the `**mp_top**` module, while exhibiting some synchronization issues, closely resembled the manually desired output. This nuanced observation suggests that the structural integrity of the `**mp_top**` module remains intact, with the identified synchronization errors warranting further investigation and refinement in subsequent iterations of the design.

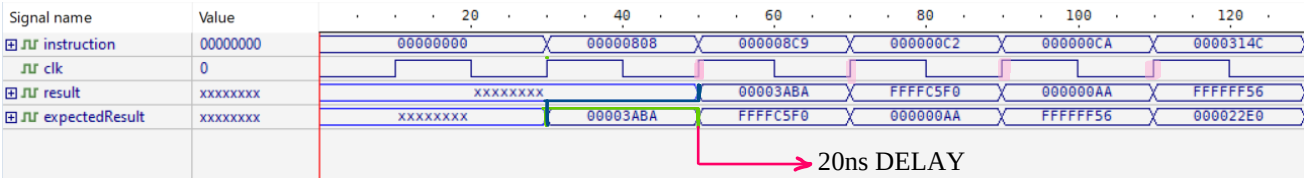

Figure 7: mp_top testbench results part1.

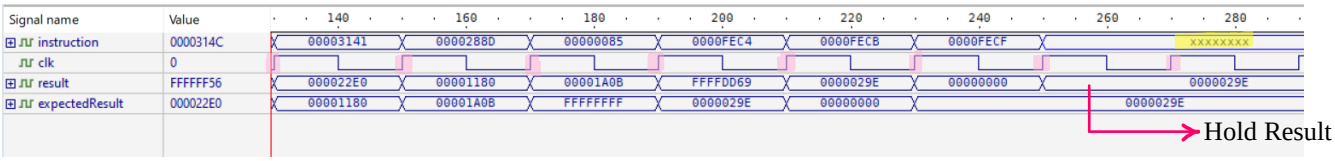

Figure 8: mp_top testbench results part2.

In the initial two cycles of the microprocessor's operation, the output is marked as "don't cares" due to the inherent delay in the availability of data inputs for the ALU. This delay is an integral part of the synchronization definition, as the data inputs need to wait for two cycles until the Register File successfully extracts the output from memory. This intentional delay ensures that the subsequent operations within the microprocessor occur with synchronized and valid data, underscoring the importance of temporal alignment in achieving accurate and reliable results.

```
--------------------------------WELCOME TO MY SIMPLE MICROPROCESSOR-------------------------------

TIME 10000  INSTRUCTION=00000000000000000000000000000000  CLK=1  RESULT=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  EXPECTED RESULT=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
TIME 20000  INSTRUCTION=00000000000000000000000000000000  CLK=0  RESULT=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  EXPECTED RESULT=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
THIS CASE -----------------> TEST PASSES :)

TIME 30000  INSTRUCTION=00000000000000000000100000001000  CLK=1  RESULT=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  EXPECTED RESULT=00000000000000000011101010111010
TIME 40000  INSTRUCTION=00000000000000000000100000001000  CLK=0  RESULT=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  EXPECTED RESULT=00000000000000000011101010111010
THIS CASE -----------------> TEST PASSES :)

TIME 50000  INSTRUCTION=00000000000000000000100011001001  CLK=1  RESULT=00000000000000000011101010111010  EXPECTED RESULT=11111111111111111000101111110000
TIME 60000  INSTRUCTION=00000000000000000000100011001001  CLK=0  RESULT=00000000000000000011101010111010  EXPECTED RESULT=11111111111111111000101111110000
THIS CASE -----------------> TEST FAILS :(

TIME 70000  INSTRUCTION=00000000000000000000000011000010  CLK=1  RESULT=11111111111111111000101111110000  EXPECTED RESULT=00000000000000000000000010101010
TIME 80000  INSTRUCTION=00000000000000000000000011000010  CLK=0  RESULT=11111111111111111000101111110000  EXPECTED RESULT=00000000000000000000000010101010
THIS CASE -----------------> TEST FAILS :(

TIME 90000  INSTRUCTION=00000000000000000000000011001010  CLK=1  RESULT=00000000000000000000000010101010  EXPECTED RESULT=11111111111111111111111101010110
TIME 100000  INSTRUCTION=00000000000000000000000011001010  CLK=0  RESULT=00000000000000000000000010101010  EXPECTED RESULT=11111111111111111111111101010110
THIS CASE -----------------> TEST FAILS :(

TIME 110000  INSTRUCTION=00000000000000000011000101001100  CLK=1  RESULT=11111111111111111111111101010110  EXPECTED RESULT=00000000000000000010001011100000
TIME 120000  INSTRUCTION=00000000000000000011000101001100  CLK=0  RESULT=11111111111111111111111101010110  EXPECTED RESULT=00000000000000000010001011100000
THIS CASE -----------------> TEST FAILS :(

TIME 130000  INSTRUCTION=00000000000000000011000101000001  CLK=1  RESULT=00000000000000000010001011100000  EXPECTED RESULT=00000000000000000001000110000000
TIME 140000  INSTRUCTION=00000000000000000011000101000001  CLK=0  RESULT=00000000000000000010001011100000  EXPECTED RESULT=00000000000000000001000110000000
THIS CASE -----------------> TEST FAILS :(

TIME 150000  INSTRUCTION=00000000000000000010100010001101  CLK=1  RESULT=00000000000000000001000110000000  EXPECTED RESULT=00000000000000000001101000001011
TIME 160000  INSTRUCTION=00000000000000000010100010001101  CLK=0  RESULT=00000000000000000001000110000000  EXPECTED RESULT=00000000000000000001101000001011
THIS CASE -----------------> TEST FAILS :(

TIME 170000  INSTRUCTION=00000000000000000000000010000101  CLK=1  RESULT=00000000000000000001101000001011  EXPECTED RESULT=11111111111111111111111111111111
TIME 180000  INSTRUCTION=00000000000000000000000010000101  CLK=0  RESULT=00000000000000000001101000001011  EXPECTED RESULT=11111111111111111111111111111111
THIS CASE -----------------> TEST FAILS :(

TIME 190000  INSTRUCTION=00000000000000001111111011000100  CLK=1  RESULT=11111111111111111101110101101001  EXPECTED RESULT=00000000000000000000001010011110
TIME 200000  INSTRUCTION=00000000000000001111111011000100  CLK=0  RESULT=11111111111111111101110101101001  EXPECTED RESULT=00000000000000000000001010011110
THIS CASE -----------------> TEST FAILS :(

TIME 210000  INSTRUCTION=00000000000000001111111011001011  CLK=1  RESULT=00000000000000000000001010011110  EXPECTED RESULT=00000000000000000000000000000000
TIME 220000  INSTRUCTION=00000000000000001111111011001011  CLK=0  RESULT=00000000000000000000001010011110  EXPECTED RESULT=00000000000000000000000000000000
THIS CASE -----------------> TEST FAILS :(

TIME 230000  INSTRUCTION=00000000000000001111111011001111  CLK=1  RESULT=00000000000000000000000000000000  EXPECTED RESULT=00000000000000000000001010011110
TIME 240000  INSTRUCTION=00000000000000001111111011001111  CLK=0  RESULT=00000000000000000000000000000000  EXPECTED RESULT=00000000000000000000001010011110
THIS CASE -----------------> TEST FAILS :(

--------------------------------------------------------------------------------

FOR THIS PROGRAM -----------------> TEST FAILS :(

--------------------------------------------------------------------------------
```

Figure 9: mp_top testbench results part3.

In an effort to assess the expected results, a memory structure was integrated into the test bench. However, an unexpected reversal in the delay was encountered during execution, with simulated results appearing 20ns after the anticipated outcomes. While eventual alignment between simulated and expected results was achieved, this delay introduced synchronization challenges, leading to the categorization of test outcomes as failures despite matching values. This peculiar delay issue was not isolated and was shared among teammates, indicating a systemic challenge within the design. Additionally, the intricacies of read and write operations at the same address further complicated the design. The microprocessor design requires waiting until simulated results are calculated from the ALU before initiating write operations; however, determining the precise location to introduce this delay in the top module proved challenging. These unforeseen issues underscore the need for extensive investigation and refinement to enhance synchronization, temporal alignment, and overall reliability in subsequent iterations of the microprocessor design.

Despite exhaustive efforts to identify and rectify the synchronization issues, a definitive solution remained elusive. The microprocessor's test results, while yielding correct outcomes, were marred by discrepancies in the expected results due to persistent synchronization errors. Despite thorough searches and analyses, the elusive nature of these issues necessitates further investigation and refinement to pinpoint and address the underlying challenges in subsequent design iterations. Then badly TEST FAILS.

Conclusion

In summary, the microprocessor project successfully implemented key modules, including the ALU and Register File. While individual components operated effectively, synchronization issues surfaced during testing, impacting the alignment of expected and observed results. Despite efforts to address these challenges, a definitive solution remained elusive. The project underscores the complexity of digital system design and emphasizes the need for further refinement to enhance synchronization and overall reliability.

References

[1]: GeeksforGeeks. (2024, January 19). *Introduction of microprocessor*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-of-microprocessor/

[2]: GeeksforGeeks. (2024, January 19). *Introduction of alu and data path*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-of-alu-and-data-path/

[3]: Chat.openai.com. (n.d.). Retrieved January 19, 2024, from https://chat.openai.com/

Appendix

```
//Malak Ammar 1211470
//alu module to calculate microprocesser's arithmetic operations to generate output
module alu(opcode, a, b, result);
//define opcodes as local constants
`define ADDITION 6'b001000 `define SUBTRACTION 6'b001001 `define ABS 6'b000010
`define NEG 6'b001010 `define MAX 6'b001100 `define MIN 6'b000001 `define AVG 6'b001101
`define NOTA 6'b000101 `define OR 6'b000100 `define AND 6'b001011 `define XOR 6'b001111
//alu's inputs & outputs
input [31:0] a, b; //two 32-bits inputs
input [5:0] opcode; //6-bits opCode input
output reg [31:0] result; //32-bits output
always @(a or b or opcode) //sensitive to the change of a, b, or opcode
begin
case(opcode)
`ADDITION : AdderOrSubtractor(a, b, 1'b0, result); //zero carry-in to operate adder
`SUBTRACTION : AdderOrSubtractor(a, b, 1'b1, result); //one carry-in to operate subtractor
`ABS : Absolute(a, result);
`NEG : Negation(a, result);
`MAX : findMax(a, b, result);
`MIN : findMin(a, b, result);
`AVG : avg(a, b, result);
`NOTA : Not(a, result);
`OR : Or(a, b, result);
`AND : And(a, b, result);
`XOR : Xor(a, b, result);
default : result=result; //hold the result
endcase
end
//////////////////////
//task to operate the average value
task avg;
input signed [31:0] A, B;
```

```
output reg signed [31:0] avgOut;
assign avgOut=(A+B)/2;
endtask
//////////////////////
//task to operate the negation value of A
task Negation;
input signed [31:0] A;
output reg signed [31:0] F;
assign F=-1*A;
endtask
//////////////////////
//task to operate the not of A
task Not;
input signed [31:0] A;
output reg signed [31:0] F;
assign F=~A;
endtask
//////////////////////
//task to operate the orring value
task Or;
input signed [31:0] A, B;
output reg signed [31:0] F;
assign F=A|B;
endtask
//////////////////////
//task to operate the anding value
task And;
input signed [31:0] A, B;
output reg signed [31:0] F;
assign F=A&B;
endtask
//////////////////////
//task to operate the xoring value
task Xor;
```

```
input signed [31:0] A, B;
output reg signed [31:0] F;
assign F=A^B;
endtask
//////////////////////
//task to operate the abs value of A
task Absolute;
input signed [31:0] A;
output reg signed [31:0] F;
if(A[31]==1'b1)
F=-A;
else
F=A;
endtask
//////////////////////
//task to find the maximum value
task findMax;
input signed [31:0] A, B;
output reg signed [31:0] maxOut;
if(A>=B)
maxOut=A;
else if(B>=A)
maxOut=B;
endtask
//////////////////////
//task to find the minimum value
task findMin;
input signed [31:0] A, B;
output reg signed [31:0] minOut;
if(A<=B)
minOut=A;
else if(B<=A)
minOut=B;
endtask
```

```
//////////////////////
//task to operate the addition or subtraction value
task AdderOrSubtractor;
input signed[31:0] A, B;
input Cin;
output reg signed [31:0] Sum;
reg  Cout;
if(Cin==0) //ADDER
Sum = A+B;
else //SUBTRACTOR
Sum = A-B;
endtask
//////////////////////
endmodule
//register file module to read/write through provided addresses
module reg_file(clk, vaild_opcode, addr1, addr2, addr3, in, out1, out2);
//alu's inputs & outputs
input [31:0] in; //32-bits input data to write on address 3
input [4:0] addr1, addr2, addr3; //5-bits input addresses to write/read
input clk, vaild_opcode; //clock to synchronize the register file, and enable
output reg [31:0] out1, out2; //32-bits output to hold read results
//memory register to store values of size 32X32, 32 memory location, each consists of 32-bits
reg [31:0] memory [0:31] = '{32'h00000000, 32'h00003ABA, 32'h00002296, 32'h000000AA, 32'h00001C3A,
32'h00001180,32'h000022E0, 32'h00001C86, 32'h000022DA, 32'h00000414, 32'h00001A32, 32'h00000102, 32'h00001CBA,
32'h00000CDE, 32'h00003994,32'h00001984, 32'h000028C4, 32'h00002E70, 32'h00003966, 32'h0000227E, 32'h00002208,
32'h000011B4, 32'h0000237C, 32'h0000360E, 32'h00002722, 32'h00000500, 32'h000016B6, 32'h0000029E, 32'h00002280,
32'h00003B52, 32'h000011A0, 32'h00000000};
always @(posedge clk)
begin
if(vaild_opcode) //if enable is on
begin
out1=memory[addr1]; //load value in address1 to out1
out2=memory[addr2]; //load value in address2 to out2
memory[addr3]<=in; //store value of the input data to address3
end
end
```

```
endmodule
//top module to generate the process of the simple microprocesser
module mp_top(clk, instruction, result);
input clk; //clock to synchronize the system
input [31:0] instruction; //32-bits inout instruction
output reg [31:0] result; //32-bits output result
wire [5:0] opCode, delayedOp;
wire [4:0] addr1, addr2, addr3;
wire EN;
wire [31:0] out1, out2;
wire [31:0] modifiedInst;
//instantiate unit0 to load the instruction and assign it's linked address1,2,3 & opcode, also to assign last 11-bits to zero's
loadInst unit0(clk, instruction, opCode, addr1, addr2, addr3, modifiedInst);
//instantiate unit1 to keep up with clock delay to ensure working properly
DFF unit1(clk, opCode, delayedOp);
//instantiate unit2 to check if the opcode is valid or not
isValidOp unit2(opCode, EN);
//instantiate unit3 to load values from memory
reg_file unit3(clk, EN, addr1, addr2, addr3, result, out1, out2);
//instantiate unit4 to generate the arithmetic operation
alu unit4(delayedOp, out1, out2, result);
endmodule
//DFF module to delay the opcode to keep up with clock delay to ensure working properly
module DFF(clk, D, Y);
input clk;
input [5:0] D;
output reg [5:0] Y;
always @(posedge clk)
Y=D;
endmodule
//loadInst module to extract the opcode, addresse1,2,3 and set last 11-bits of instruction to 0
module loadInst(clk, instruction, opCode, addr1, addr2, addr3, modifiedInst);
input clk;
input [31:0] instruction;
```

```
output reg [5:0] opCode;
output reg [4:0] addr1, addr2, addr3;
output reg [31:0] modifiedInst;
always @(posedge clk)
begin
opCode=instruction[5:0];
addr1=instruction[10:6];
addr2=instruction[15:11];
addr3=instruction[20:16];
modifiedInst={11'b00000000000, addr3, addr2, addr1, opCode};
end
endmodule
//isValidOp module to check if the input opcode is valid or not
module isValidOp(opCode, isValid);
`define ADDITION 6'b001000 `define SUBTRACTION 6'b001001 `define ABS 6'b000010
`define NEG 6'b001010 `define MAX 6'b001100 `define MIN 6'b000001 `define AVG 6'b001101
`define NOTA 6'b000101 `define OR 6'b000100 `define AND 6'b001011 `define XOR 6'b001111
input [5:0] opCode;
output reg isValid;
always @(opCode)
case(opCode)
`ADDITION : isValid=1;
`SUBTRACTION : isValid=1;
`ABS : isValid=1;
`NEG : isValid=1;
`MAX : isValid=1;
`MIN : isValid=1;
`AVG : isValid=1;
`NOTA : isValid=1;
`OR : isValid=1;
`AND : isValid=1;
`XOR : isValid=1;
default : isValid=0; //not valid opcode
endcase
```

```
endmodule
//testbench for isValid module
module isValidTB;
reg [5:0] opCode;
wire isValid;
isValidOp unit(opCode, isValid);
initial
begin
#0 opCode=0;
$monitor("Time %0d opCode=%b  isValid=%b", $time, opCode, isValid);
#10 opCode=6'b000001;
#12 opCode=6'b001000;
#1 opCode=16;
#1 opCode=15;
end
endmodule
//testbench for loadInst module
module loadInstTB;
reg clk;
reg [31:0] instruction;
wire [5:0] opCode;
wire [4:0] addr1, addr2, addr3;
wire [31:0] modifiedInst;
loadInst unit(clk, instruction, opCode, addr1, addr2, addr3, modifiedInst);
initial
begin
#0 clk=0; instruction=32'h08431270;
#5 clk=1;
$monitor("Time %0d  clk=%b  instruction=%b  opCode=%b  addr1=%b  addr2=%b  addr3=%b  modifiedInst=%b", $time, clk,
instruction, opCode, addr1, addr2, addr3, modifiedInst);
#5 clk=0; instruction=32'h0F431270;
#5 clk=1;
end
endmodule
//testbench for alu module
```

```
module ALUTB;
reg [31:0] a, b;
reg [5:0] opcode;
wire [31:0] result;
integer i;
alu unit(opcode, a, b, result);
initial
begin
#0 a=0; b=0; opcode=0; //initiallize
$monitor("Time %0d  A=%b  B=%b  opCode=%b  Result=%b", $time, a, b, opcode, result);
#1 a=-4; b=-3; opcode=8; //add -4 to -3
#1 a=-4; b=3; opcode=9;        //subtract -4 & 3
#1 a=-1; opcode=2;                  // abs of -1
#1 a=4; opcode=10; //negative of a=4
#1 a=-8; b=1; opcode=12; //max of a=-8 & b=1
#1 a=4; b=-4; opcode=1;         //min of a=4 & b=-4
#1 a=-4; b=4; opcode=13; //avg of a=-4 & b=4
#1 a=-1; opcode=5;                //not a=-1
#1 a=0; b=-1; opcode=4;         //a=0 or b=-1
#1 a=16; b=1; opcode=11; //a=16 and b=1
#1 a=1; b=-1; opcode=15; //a=1 xor b=-1
end
endmodule
//testbench for regFile module
module regFileTB;
reg [31:0] in;
reg [4:0] addr1, addr2, addr3;
reg clk, vaild_opcode;
wire [31:0] out1, out2;
reg_file unit(clk, vaild_opcode, addr1, addr2, addr3, in, out1, out2);
initial
begin
#0ns clk=0;
repeat(10)
```

```
#2ns clk=~clk;

end

initial

begin

#0ns in=32'h11114321; addr1=0; addr2=0; addr3=0; vaild_opcode=0; //since enable is zero -> nothing to do

$monitor("Time %0d inData=%b address1=%b address2=%b address3=%b CLK=%b vaild_opcode=%b out1=%b out2=%b", $time, in, addr1, addr2, addr3, clk, vaild_opcode, out1, out2);

#5ns vaild_opcode=1; addr1=4'b0001; //enable is 1 -> it must read the data in add1 (out1), then reads the data in addr2 (out2)

//then write on addr3 the input data

#5ns in=32'h00004321; addr1=4'b0000; addr3=4'b0001; addr2=4'b0001;

#5ns addr1=4'b0001;

end

endmodule

//testbench for top module to check output results

//testbench for top module to check output results

module mpTopTB;

//define opcodes as local constants

`define ADDITION 6'b001000 `define SUBTRACTION 6'b001001 `define ABS 6'b000010

`define NEG 6'b001010 `define MAX 6'b001100 `define MIN 6'b000001 `define AVG 6'b001101

`define NOTA 6'b000101 `define OR 6'b000100 `define AND 6'b001011 `define XOR 6'b001111

//memory register to store values of size 32X32, 32 memory location, each consists of 32-bits

reg [31:0] memory [0:31] = '{32'h00000000, 32'h00003ABA, 32'h00002296, 32'h000000AA, 32'h00001C3A, 32'h00001180, 32'h000022E0, 32'h00001C86, 32'h000022DA, 32'h00000414, 32'h00001A32, 32'h00000102, 32'h00001CBA, 32'h00000CDE, 32'h00003994, 32'h00001984, 32'h000028C4, 32'h00002E70, 32'h00003966, 32'h0000227E, 32'h00002208, 32'h000011B4, 32'h0000237C, 32'h0000360E, 32'h00002722, 32'h00000500, 32'h000016B6, 32'h0000029E, 32'h00002280, 32'h00003B52, 32'h000011A0, 32'h00000000};

reg [31:0] instruction;

reg clk;

wire [31:0] result;

reg signed [31:0] out1, out2; //memory values

reg [4:0] addr1, addr2;

integer i; //counter

reg flag=1; //Flag to check if the expected value matched the generated one

//array of instructions

reg [31:0] instructions [0:10] = '{32'h00000808, 32'h000008C9, 32'h000000C2, 32'h000000CA, 32'h0000314C, 32'h00003141, 32'h0000288D, 32'h00000085, 32'h0000FEC4, 32'h0000FECB, 32'h0000FECF};

reg [31:0] expectedResult; //register to store the expected result
```

```
//instantiate unit to operate the top module
mp_top unit(clk, instruction, result);
//initial to clock the system
initial
begin
#0ns clk=0;
repeat(30)
#10ns clk=~clk;
end
//initial to generate the output & expected reult of each instruction
initial
begin
$display("----------WELCOME TO MY SIMPLE MICROPROCESSOR----------\n");
#0ns instruction=32'h00000000; clk=0; //initially set the instruction to zero
#10ns //10ns delay to avoid don't cares period -first two cycles-
//display command
$monitor("TIME %0d  INSTRUCTION=%b  CLK=%b  RESULT=%b  EXPECTED RESULT=%b", $time, instruction, clk, result, expectedResult);
for(i=0; i<12; i++) //for loop to operate the result of each instruction
begin
#20ns instruction=instructions[i];
addr1=instruction[10:6]; //get the address of the first source register
addr2=instruction[15:11]; //get the address of the second source register
out1=memory[addr1];
out2=memory[addr2];
case(instruction[5:0]) //case to generate the expected result
`ADDITION : expectedResult=out1+out2;
`SUBTRACTION : expectedResult=out1-out2;
`ABS :
if(out1<0)
expectedResult=-1*out1;
else
expectedResult=out1;
`NEG : expectedResult=-1*out1;
`MAX :
```

```
if(out1>=out2)
expectedResult=out1;
else
expectedResult=out2;
`MIN :
if(out1<=out2)
expectedResult=out1;
else
expectedResult=out2;
`AVG : expectedResult=(out1+out2)/2;
`NOTA : expectedResult=~out2;
`OR : expectedResult=out1|out2;
`AND : expectedResult=out1&out2;
`XOR : expectedResult=out1^out2;
default : expectedResult=expectedResult;
endcase
if(expectedResult!=result) //if any result don't match the generated one then test fails
flag=0; //set flag to zero --> test fails
//test fails
if(flag==0)
 $display("THIS CASE ------------------> TEST FAILS :(\n");
else //test passes
$display("THIS CASE ------------------> TEST PASSES :)\n");
end
$display("--------------------------------------------------------------------------------");
//test fails
if(flag==0)
$display("\nFOR THIS PROGRAM ------------------> TEST FAILS :(\n");
else //test passes
$display("\nFOR THIS PROGRAM ------------------> TEST PASSES :)\n");
$display("--------------------------------------------------------------------------------\n");
$finish (250ns); //stop the simulation after 250ns (after finishing the instructions)
end
endmodule
```