Electrical and Computer Engineering Department

ENCS3310

ADVANCED DIGITAL SYSTEMS DESIGN

Project report

Student name: Kareem Qutob

ID: 1211756

Instructor name: Dr. Abdallatif Abuissa

section: 1

Date: **2024/1/14**

Abstract

In this project, a microprocessor, consisting of a Register File and an Arithmetic Logic Unit (ALU), is designed and implemented. Based on a 6-bit opcode that is determined by the final digit of my student's ID number, the ALU operates on 32-bit inputs. Operating as quick RAM, the Register File stores operands whose initial values are determined by the ID's second-to-last digit. The Register File introduces clock synchronization to guard against data hazards when read and write operations are occurring simultaneously. To adjust the Register File's sensitivity, an enable input is added. After that, the ALU and Register File are incorporated into a microprocessor that is controlled by 32-bit machine instructions, making it possible to verify the accuracy of the design using a specific test bench. The project provides a hands-on exploration of microprocessor design, emphasizing clock synchronization and opcode handling for efficient digital system implementation.

Contents

Table of figures

Table of tables

Theory

1. ALU

The Arithmetic Logic Unit (ALU) is a fundamental component of a computer's central processing unit (CPU) that performs arithmetic and logic operations on binary numbers. The ALU's arithmetic capabilities include addition and subtraction, and in more complex designs, it might also be able to handle multiplication and division. The ALU can also perform logical operations, such as bitwise AND, OR, XOR, and NOT, which are necessary for data manipulation. The ALU also frequently includes bitwise operations, such as rotating and shifting, which allow bits to be moved and rearranged within binary numbers. Moreover, the ALU is essential for carrying out comparison operations and figuring out whether two binary values are equal or not. All things considered, the ALU acts as the CPU's computational engine, allowing the execution of various tasks essential to a computer system's operation.

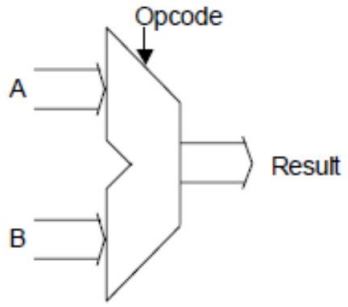


Figure 1: ALU block diagram

2. Opcode

The term "operation code," or "opcode," refers to a binary code segment found in machine language instructions in computer architecture that directs the central processing unit (CPU) to carry out a particular command or operation. Opcodes, which define the basic tasks a CPU is capable of performing, are the cornerstone of instruction sets and are typically expressed in machine code. These include data transfers, logical operations, and arithmetic computations. The exact nature of the operation to be performed is determined by the opcode, which is frequently supplemented by additional bits that indicate operands or parameters. A given instruction set architecture's diversity of opcodes enables computers to perform a broad range of tasks, providing the basis for computing systems' programmability and computational versatility.

Figure 2:basic instruction format

3. Register file

A register file, which is a crucial component of a central processing unit (CPU), is a fast and small way to store volatile data. It is made up of registers that can store binary values. These registers, which are usually arranged in a matrix arrangement, are used to temporarily store data while machine instructions are being executed. The register file is necessary to enable smooth communication with the Arithmetic Logic Unit (ALU) and quick access to operands needed for arithmetic and logic operations. It is essential for increasing CPU efficiency because it facilitates quick data retrieval and manipulation while instructions are being executed. Register files are made up of special-purpose registers that are used for specific purposes, such as program counters or status flags, as well as general-purpose registers that serve a variety of purposes. Together, these registers help to increase the speed and efficiency of computational tasks.

ALU design

The ALU used in the project is an ALU with two 32 bits inputs named A and B , 32-bit output RESULT with 6-bit input OPCODE that will determine the operation performed on the inputs from the operation in the table below, opcode will be based on my last ID digit which is 6

Operation	OPCODE
A+B	**000100**
A-B	**001010**
|A|	**000011**
-A	**001100**
Max (A, B)	**000111**
Min (A, B)	**000010**
AVG (A, B)	**000110**
Not A	**001101**
A or B	**001110**
A and B	**001011**
A xor B	**001000**

Table 1:Opcode table

The code for it is quite simple which is a case statements that change the output whenever the value of the opcode changes as shown in the code below:

```
module alu (opcode, a, b, result );
input signed [5:0] opcode;
input signed [31:0] a, b;
output reg signed  [31:0] result;

always @(*)
    case(opcode)
        // op codes will be based on the last digit of my id which is 6
        6'b000100: result=a+b;
        6'b001010: result=a-b;
        6'b000011: if(a<0) result= -a; else result = a;   // to get absolute value if input less than 0
we will make it positve by puting a negative sign before it
        6'b001100: result=-a;
        6'b000111: if(a>b) result =a ;else result=b;
        6'b000010: if(a<b) result =a ;else result=b;
        6'b000110: result=(a+b)>>1; // the right shift will divide by 2 which will divide t=by 2 ti get
the average
        6'b001101: result=~a;
        6'b001110: result= a|b;
        6'b001011: result = a &b;
        6'b001000: result= a ^ b;

    endcase

endmodule
```

Figure 3:ALU code

The implementation is quite simple and straightforward the absolute value was implemented as an if-else statement if the input is higher than zero it will stay the same else we will negate it, A similar algorithm is used in min and max functions, and for the average I used shift right which will divide by 2, I used signed inputs and outputs because the answer may be negative and arithmetic operations should operate on signed inputs.

Register file design

In our processor design, there is a small amount of memory that holds the operands of the ALU which is called a register file which is a very small and fast RAM that holds 32 x 32-bit words which will require 5-bit addresses (2^5=32), our register will process 3 addresses, the first two are for reading stored data in the register and the third one is used for writing the entered input in the register, to avoid overwriting the register must be synchronized by a clocking signal that will make the register work whenever the clock has a rising edge with non blocking assignment.

The register will hold values that will be based on the second to last digit of my ID which is 5 which are listed in the table below

Address	content
0	0
1	11930
2	5348
3	7308
4	15684
5	12346
6	9716
7	7820
8	5190
9	14702
10	5630
11	2352
12	15424
13	2670
14	4172
15	4300
16	4744
17	1286
18	8122
19	4558
20	8534
21	13340
22	6918
23	11700
24	10722
25	3346
26	3300
27	2386
28	11212
29	3504
30	8712
31	0

Table 2: memory table

Register code:

```
module reg_file (clk,valid_opcode, addr1, addr2, addr3, in , out1, out2);
input clk;
input valid_opcode;
input [4:0] addr1, addr2, addr3;
input [31:0] in;
output reg [31:0] out1, out2;
reg [31:0] memory [31:0];
initial
    begin       /// initializing memory based on my second to last number in
my id which is 5
        memory[0]=0;
        memory[1] =11930;
        memory[2] =5348;
        memory[3] =7308;
        memory[4] =15684;
        memory[5] =12346;
        memory[6] =9716;
        memory[7] =7820;
```

```
        memory[8] =5190;
        memory[9] =14702;
        memory[10] =5630;
        memory[11] =2352;
        memory[12] =15424;
        memory[13] =2670;
        memory[14] =4172;
        memory[15] =4300;
        memory[16] =4744;
        memory[17] =1286;
        memory[18] =8122;
        memory[19] =4558;
        memory[20] =8534;
        memory[21] =13340;
        memory[22] =6918;
        memory[23] =11700;
        memory[24] =10722;
        memory[25] =3346;
        memory[26] =3300;
        memory[27] =2386;
        memory[28] =11212;
        memory[29] =3504;
        memory[30] =8712;
        memory[31] =0;
    end


always @(posedge clk)
    begin
        if(valid_opcode) // work as enable
            begin
            out1 <= memory[addr1];
            out2 <= memory[addr2];
                memory[addr3] <= in;
            end
    end
endmodule
```

Figure 4:Register code

The first part of the code is an initial block that is used to initialize the memory with the data from the table before.

The always block contains the synchronized part of the code, on every rising edge of the clock the register will read the first two addresses and write in the third one, this register is enabled by a bit named valid, if this bit is not 1 the register will not work, this is used to check if given opcode is a valid one, active bit generation will be discussed in details in the later part of the report.

Microprocessor design

Our microprocessor will be built structurally from the ALU and register we built before

The code should be connected in a way to implement this block diagram

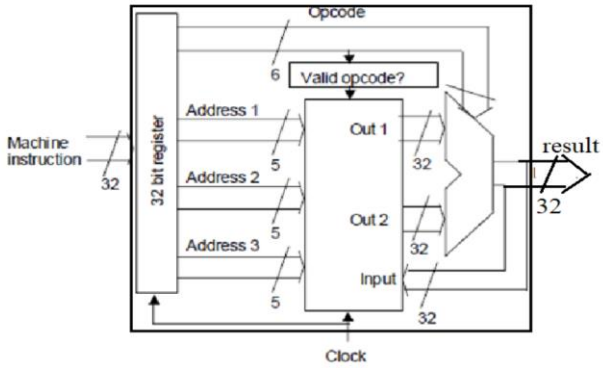


Figure 5: Microprocessor block diagram

The design will take an instruction of length 32 bit and will divide it as the following:

• The first 6 bits identify the opcode

• The next 5 bits identify the first source register

• The next 5 bits identify the second source register

• The next 5 bits identify the destination register

• The final 11 bits are unused

The addresses will be the input of the register that will provide the operands if the checked opcode is valid, this may cause some problems because the opcode is sent to both register and ALU in case the opcode is not valid this will cause some problems to ensure this does not happen the opcode must go on a buffer after entering register so the that when it reaches the ALU the opcode will be already checked if valid and operands will be provided to the alu,

My buffer code:

```
module Buffer (
  input wire [5:0] in,
  input wire clk,
  output wire [5:0] out
);

  reg [5:0] buffer_out;

  always @(posedge clk) begin
    buffer_out <= in;
  end

  assign out = buffer_out;

endmodule
```

Figure 6:Buffer code

My buffer is a simple buffer that will hold the opcode for one clock cycle (positive edge triggered) until the register finishes its work.

The processor code will look like this:

```
module mp_top (clk, instruction , result );
input clk;
input [31:0] instruction;
output reg [31:0] result;

 wire [4:0]addr1,addr2,addr3;
 wire [5:0]opcode;
 reg valid=0;   // will work as an enable bit for register
 wire [32:0]out1,out2;
 assign opcode=instruction[5:0];     // dividing the instruction based on bits
to opcode and the used adresess
 assign addr1=instruction[10:6];
 assign addr2=instruction[15:11];
 assign addr3=instruction[20:16];
 wire [5:0]opcode_alu;


reg [5:0] opcode_array [10:0];
initial
    begin           // initialising valid opcodes in an array so we can check
incoming ones
        opcode_array[0]=4;
        opcode_array[1]=10;
        opcode_array[2]=3;
        opcode_array[3]=12;
        opcode_array[4]=7;
        opcode_array[5]=2;
        opcode_array[6]=6;
        opcode_array[7]=13;
        opcode_array[8]=14;
        opcode_array[9]=11;
        opcode_array[10]=8;
    end
    int  i;
    always@(opcode)     // this loop will check if entered opcode is one of
the valid ones
        begin
            for(int i=0;i<11;i=i+1)
            begin
            if(opcode==opcode_array[i])
                valid=1;
            end
        end
    reg_file register(clk,valid,addr1,addr2,addr3,result,out1,out2);

    Buffer bf(opcode,clk,opcode_alu);   // opcode will go into a buffer to
prevent worng output
    alu ALU(opcode_alu,out1,out2,result) ;


endmodule
```

Figure 7:Top module code

In the top module, the instruction will be divided into parts with the last 11 bits ignored, the opcode bits will go through a loop that will check if the opcode is valid by iterating over an array of all of the valid opcode, if the opcode is found the valid bit will be one else it will remain a zero, the valid bit will work as an enable for the register, while this happens the opcode will go to a buffer before entering the ALU to make sure that receive inputs and to be synchronized with the rest of the design.

Test Cases

To run a test bench I must write instructions based on the previous Instruction specifications the table below shows the instructions used

No.	Instruction code	Instruction function	Expected results
1	32'h000d5004	ADD MEM[0] , MEM[10]	5630
2	32'h000e484a	SUB MEM[1] , MEM[9]	-2772
3	32'h000f4083	ABS MEM[2]	5348
4	32'h001038cc	SUB 0 , MEM[3]	-7308
5	32'h00113107	MAX MEM[4] , MEM[6]	15684
6	32'h00122942	MIN MEM[5] , MEM[5]	12346
7	32'h00132186	AVG MEM[6] , MEM[4]	12700
8	32'h001419cd	NEG MEM[7]	-7820
9	32'h0015120e	ORR MEM[8] , MEM[2]	5350
10	32'h00160a4b	AND MEM[9] , MEM[1]	10250
11	32'h00170288	XOR MEM[10] , MEM[0]	5630
12	32'h00000000	Invalid opcode	Same as previous inst.
13	32'h00011044	ADD MEM[2], MEM[1]	17278

Table 3: test cases table

The instruction functions are written in ARM assembly style to simplify the formatting, these cases use all of the ALU functions so we can test them properly, the second to last instruction is an invalid one that will not affect the output because the enable will mark it as an invalid opcode and the ALU will not do anything the last instruction will test if read and write addresses will overlap or not

Test bench

Using the test cases from the previous table I will build a test bench that will take these instructions and create an instance of the top module and compare the output with the expected output for every instruction if one test fails the whole test will fail

Test bench code:

```
module TestMp;
  reg clk;
  reg [31:0] instruction;
  wire [31:0] result;
  reg [31:0] instructions [12:0];   // arrays of used instruction to test
  reg [31:0] expected_answers[12:0];  //excpected outputs
  reg valid;
  reg [3:0] counter;    // this counter is essintial to the checking of validation because we only want
to test the output when it is stable
  reg [31:0] answer;
  // becuae my code use 2 clock cycle the output will not change immeadiatly
  mp_top Processor(clk, instruction, result);         // the instance we want to test

  initial
```

```
    begin
        // initialising instructions
      instructions[0] = 32'h000d5004;
      instructions[1] = 32'h000e484a;
      instructions[2] = 32'h000f4083;
      instructions[3] = 32'h001038cc;
      instructions[4] = 32'h00113107;
      instructions[5] = 32'h00122942;
      instructions[6] = 32'h00132186;
      instructions[7] = 32'h001419cd;
      instructions[8] = 32'h0015120e;
      instructions[9] = 32'h00160a4b;
      instructions[10] = 32'h00170288; // this instructions is invalid
      instructions[11] = 0;
      instructions[12]= 32'h00011044;    // this instruction to check if there are no overlab between
reading and writing
      expected_answers[0] = 5630;
      expected_answers[1] = 4294964524;
      expected_answers[2] = 5348;
      expected_answers[3] = 4294959988;
      expected_answers[4] = 15684;
      expected_answers[5] = 12346;
      expected_answers[6] = 12700;
      expected_answers[7] = 4294959475;
      expected_answers[8] = 5350;
      expected_answers[9] = 10250;
      expected_answers[10] = 5630;
      expected_answers[11] = 5630;
      expected_answers[12] = 17278;
      clk = 0;
      valid = 1;

      counter = 0;
       instruction = instructions[0];
      $display("Time   clk    instruction      result   excpected results");
```

```
      for (int i = 0; i < 13; i = i + 1)  // this loop will iterate over instructions
        begin

          #10ns clk = ~clk;
          instruction = instructions[i];
          answer= expected_answers[i] ;
          #10ns clk = ~clk;
          #10ns;

            $monitor("%0t   %b    %h      %d    %d", $time, clk, instruction, result,answer);
          // Check validity every two clock cycles
          if (counter == 1)
            begin
              if (result != expected_answers[i]) // if wrong output the test will fail
                valid = 0;

            end

          counter = counter + 1;
          if (counter == 2)
            counter = 0;
        end

      if (valid)         // after ending the test cases we check if it was a succesfull run
        $display("PASS");
      else
        $display("FAIL");

      $finish;
    end
endmodule
```

Figure 8:test bench code

Note that I needed to use a counter to check if the output is valid because my code is clock based and will change only on the positive edge this means that the output will not change immediately when a new instruction is added it will change when the clock go from 0 to 1, in my code I initialized instructions with their expected answers and the loop will check if every instruction has the expected output value, and after finishing the valid bit will decide if the test failed or not

Test bench simulation:


```
# KERNEL: Time    clk    instruction      result excpected results
# KERNEL: 30000    0    000d5004           5630       5630
# KERNEL: 40000    1    000e484a      4294964524 4294964524
# KERNEL: 50000    0    000e484a      4294964524 4294964524
# KERNEL: 70000    1    000f4083           5348       5348
# KERNEL: 80000    0    000f4083           5348       5348
# KERNEL: 100000    1    001038cc      4294959988 4294959988
# KERNEL: 110000    0    001038cc      4294959988 4294959988
# KERNEL: 130000    1    00113107          15684      15684
# KERNEL: 140000    0    00113107          15684      15684
# KERNEL: 160000    1    00122942          12346      12346
# KERNEL: 170000    0    00122942          12346      12346
# KERNEL: 190000    1    00132186          12700      12700
# KERNEL: 200000    0    00132186          12700      12700
# KERNEL: 220000    1    001419cd      4294959475 4294959475
# KERNEL: 230000    0    001419cd      4294959475 4294959475
# KERNEL: 250000    1    0015120e           5350       5350
# KERNEL: 260000    0    0015120e           5350       5350
# KERNEL: 280000    1    00160a4b          10250      10250
# KERNEL: 290000    0    00160a4b          10250      10250
# KERNEL: 310000    1    00170288           5630       5630
# KERNEL: 320000    0    00170288           5630       5630
# KERNEL: 340000    1    00000000           5630       5630
# KERNEL: 350000    0    00000000           5630       5630
# KERNEL: 370000    1    00011044          17278      17278
# KERNEL: 380000    0    00011044          17278      17278
# KERNEL: PASS
```

Figure 9:Test bench results

The results from the test bench are identical to the expected answers from the table before so that the test was successful and Pass is printed; note that the output changes every clock cycle on the positive edge

*note that results like 4294959475 is a negative number stored in signed binary so when it is converted to decimal it will look like this

The test was a success which mean my design is ready to be used in real applications

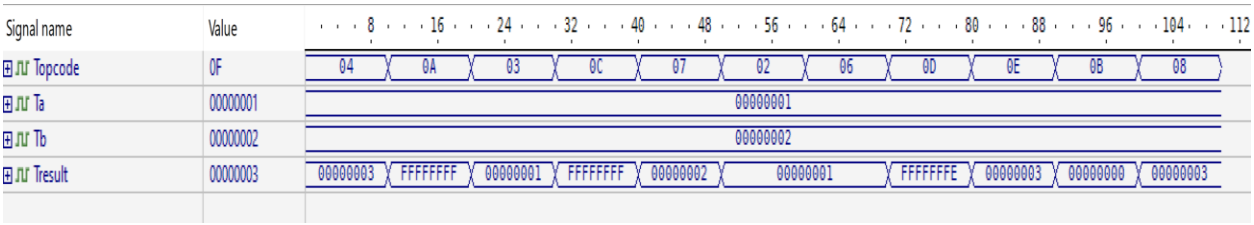
Conclusion

In conclusion, by building a customized microprocessor, this project showed how digital design principles can be used in real-world applications. Using distinct student IDs to customize the opcode brought a creative element and demonstrated a deeper comprehension of Verilog coding. Through the integration of advanced functions like enable signals and clock synchronization, the project attempted to imitate a real microprocessor environment. The focus on opcode functionality, extensive testing, and result validation enhanced the overall educational process. Beyond the initial goal, the project was a great way to practice advanced Verilog coding techniques and gained insight into the complexities of microprocessor architecture.

Appendix I

This appendix will contain screenshots and simulations for the parts of the code without the main test bench

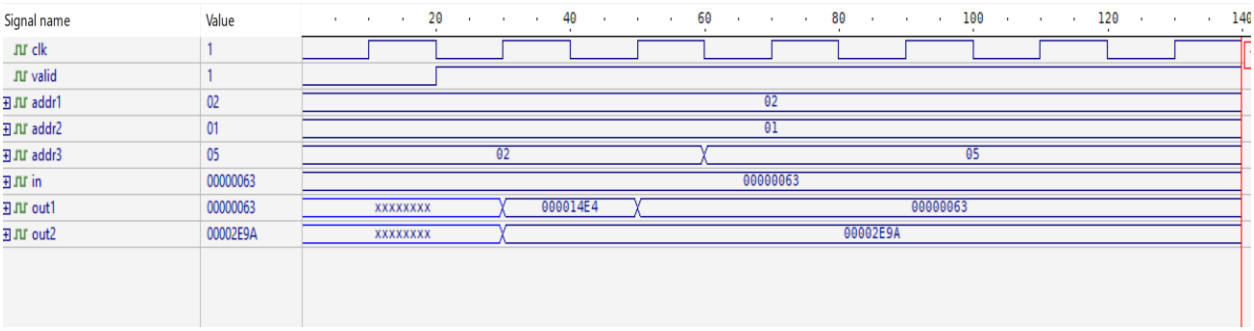
ALU:




```
# KERNEL: Time    opcode    A    B    result
# KERNEL: 0     4                 1              2              3
# KERNEL: 10    10                 1              2   4294967295
# KERNEL: 20     3                 1              2              1
# KERNEL: 30    12                 1              2   4294967295
# KERNEL: 40     7                 1              2              2
# KERNEL: 50     2                 1              2              1
# KERNEL: 60     6                 1              2              1
# KERNEL: 70    13                 1              2   4294967294
# KERNEL: 80    14                 1              2              3
# KERNEL: 90    11                 1              2              0
# KERNEL: 100     8                  1              2              3
# KERNEL: 110    15                  1              2              3
# KERNEL: Simulation has finished. There are no more test vector
```

Figure 10: ALU test bench results

Register file:




```
# KERNEL: Time   clk   valid   addr1  addr2 addr3 in out1  out2
# KERNEL: 0     0     0      2       1        2              99          x             x
# KERNEL: 10     1     0      2       1        2              99          x             x
# KERNEL: 20     0     1      2       1        2              99          x             x
# KERNEL: 30     1     1      2       1        2              99       5348         11930
# KERNEL: 40     0     1      2       1        2              99       5348         11930
# KERNEL: 50     1     1      2       1        2              99         99         11930
# KERNEL: 60     0     1      2       1        5              99         99         11930
# KERNEL: 70     1     1      2       1        5              99         99         11930
```

Figure 11: Register test bench results