Department of Electronic and Telecommunication Engineering

University of Moratuwa

32 bit non-pipelined RISC-V processor using Micropramming with 3 bus structure

G.I. Deshapriya

200118X

Table of Contents

1.Introduction

The purpose of this report is to provide an in-depth analysis of resource utilization within the context of the design and implementation of a 32-bit non-pipelined RISC-V processor. The processor is based on the RV32I instruction set architecture and has been extended to incorporate custom instructions, MEMCOPY and MUL, to enhance its functionality. While the primary objective of this project was to develop a functional processor, it is equally crucial to assess the allocation of resources to achieve optimal performance and efficiency. Resource utilization is a critical aspect of any hardware design, particularly for Field-Programmable Gate Array (FPGA) implementations. FPGA-based systems are widely used across a range of applications, and efficient resource allocation is essential to minimize costs and power consumption while maximizing performance.

2.Background

In the realm of digital design and FPGA (Field-Programmable Gate Array) development, the efficient allocation of resources holds a position of paramount importance. FPGAs are extraordinary devices that provide reconfigurable hardware, allowing designers to craft custom solutions for specific tasks. However, the effectiveness of these solutions hinges on the judicious utilization of the available resources, a factor that profoundly influences the cost-effectiveness and performance of a design. Resource allocation in FPGA-based systems is akin to assembling a puzzle: each logic element, memory block, and interconnect resource must be carefully placed and interconnected to form a cohesive and optimized whole. Every piece of this puzzle affects the overall functionality and efficiency of the final design, making it a central concern for FPGA designers.

This report delves into the crucial realm of resource utilization within the context of developing a 32-bit non-pipelined RISC-V processor. The project not only seeks to create a functional processor but also to assess and optimize the allocation of vital resources to ensure that the design strikes a harmonious balance between performance and efficiency. In this digital design journey, the FPGA becomes our canvas, and the efficient allocation of resources, our brushstrokes. To fully appreciate the significance of these brushstrokes, we must first understand the canvas and the tools at our disposal.

This report begins by establishing the groundwork for the intricate art of FPGA-based design and resource utilization. It is here that we lay the foundation for the intricate analysis and findings to follow, exploring how FPGA technology enables the construction of custom solutions and the role of efficient resource allocation in this endeavor. As we embark on this exploration, we find ourselves at the intersection of creativity and engineering, where each resource allocation decision contributes to the symphony of a well-orchestrated digital system. This journey invites us to appreciate the delicate balance between design choices and FPGA resources that ultimately shapes the destiny of our RISC-V processor and, by extension, the landscape of FPGA-based systems.

3.Key Resource Types

Resource utilization in FPGA development encompasses various key resource types:

a. Logic Utilization: This refers to the efficient use of logic elements, such as lookup tables (LUTs), in FPGA designs.

b. Memory Utilization: FPGA devices include memory blocks for RAM and ROM, and their efficient use is a key concern.

c. DSP Blocks: For applications involving signal processing, digital signal processing (DSP) blocks are a critical resource.

d. I/O Resources: Effective utilization of input and output pins and buffers is vital for interfacing with external systems.

e. Clock Resources: Proper management of clock resources, including PLLs and clock dividers, ensures synchronous operation.

f. Routing Resources: Efficient use of routing resources, such as interconnects and switches, is essential for reduced routing congestion.

4. Importance of Efficient Resource Utilization

Efficient resource utilization offers several benefits:

a. Cost-Effectiveness: Minimizing resource use can lead to lower hardware costs.

b. Performance: It can improve performance by reducing delays and allowing higher clock frequencies.

c. Flexibility: Efficient resource use allows FPGA devices to be used for multiple purposes within a single design.

d. Power Efficiency: Proper utilization reduces power consumption, which is particularly important for portable and low-power applications.

5. Strategies for Optimization

Achieving efficient resource utilization in FPGA designs involves employing various strategies:

a. Careful Logic Design: Optimize the logic design by minimizing logic redundancy and ensuring that LUTs are used effectively.

b. Memory Hierarchy: Consider the memory hierarchy and choose the appropriate type and size of memory block.

c. DSP Optimization: Configure DSP blocks for the specific signal processing tasks to be performed.

d. I/O Planning: Plan the use of input and output pins efficiently to meet interfacing requirements.

e. Clock Domain Management: Properly manage clock domains and employ PLLs as needed to achieve synchronous operation.

f. Routing Considerations: Be mindful of routing requirements to avoid congestion and delays.

6.Derivation of important logics

1) Overflow detection logic

Signed overflow

a[31]	b[31]	result[31]	O
0	0	0	0
0	0	1	1
0	1	X	0
1	0	X	0
1	1	0	1
1	1	1	0

O = ((~result[31] & a[31] & b[31]) | (result[31] & ~a[31] & ~b[31]))

Unsigned overflow

Subtract -> only underflow happen -> c=0 indicate underflow has happened ;c=carry bit

Addition -> only overflow happen -> c=1 indicate overflow has happened

2) Overflow handled Set less than logic

Sum[31] / Negative	O (overflow)	Result / slt
0	0	0
0	1	1
1	0	1
1	1	1

Result = O ^ Sum[31]

3) Branch logic

-- beq/bne

funct3[0]	Zero flag	Branch
0 (beq)	0	0
0 (beq)	1	1
1 (bne)	0	1
1 (bne)	1	0

Branch = funct3[0]^Zero_flag

-- blt/bge

funct3[0]	Valid_negative	Branch
0 (blt)	0	0
0 (blt)	1	1
1 (bge)	0	1
1 (bge)	2	0

Negative_flag	Overflow_flag	Valid_negative
0	0	0
0	1	1
1	0	1
1	1	0

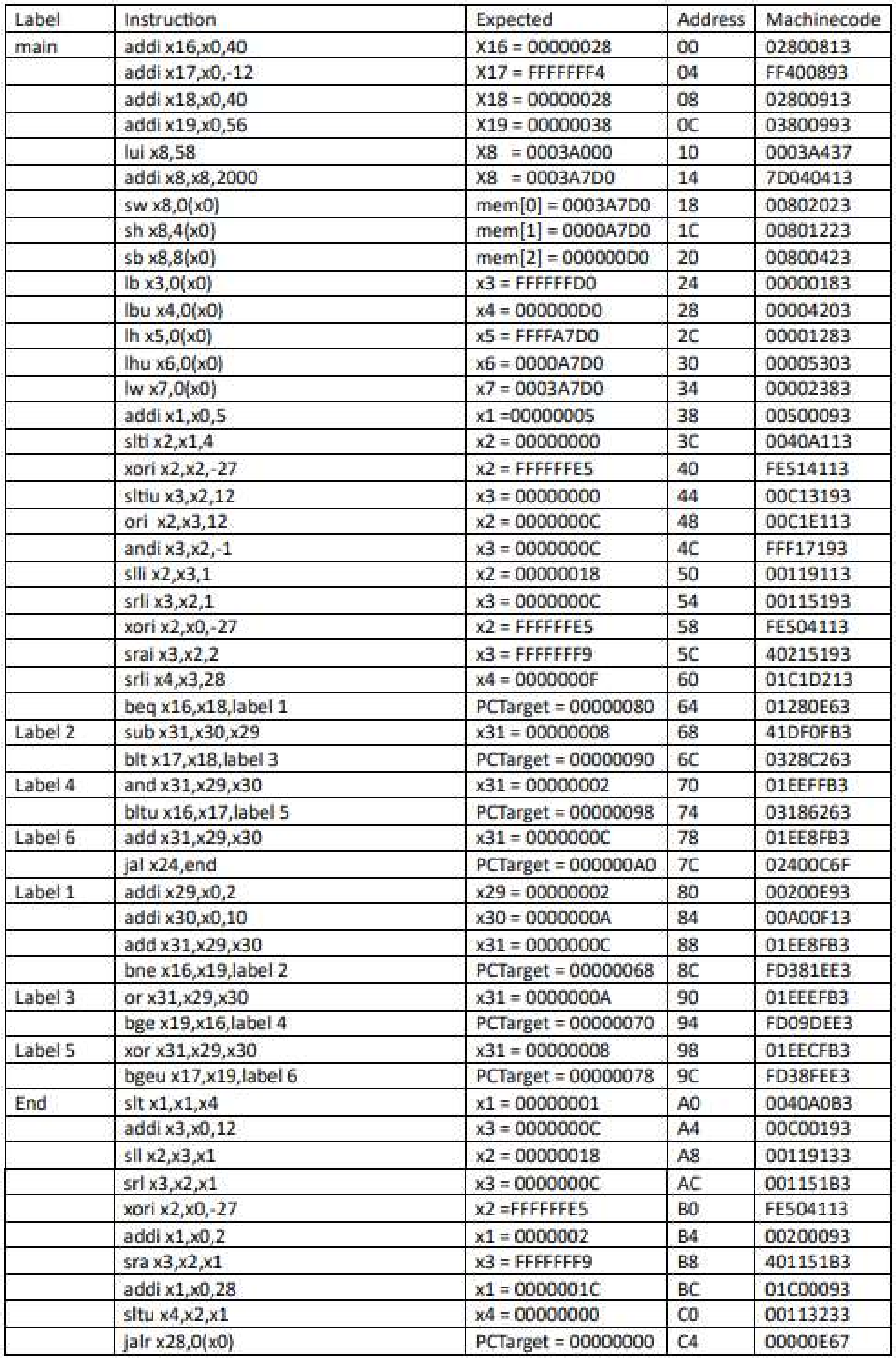
Branch = funct3[0]^(negative_flag^overflow_flag)

-- bltu/bgeu

funct3[0]	Carry_flag	~Carry_flag	Branch
0 (bltu)	0	1	1
0 (bltu)	1	0	0
1 (bgeu)	0	1	0
1 (bgeu)	1	0	1

Branch = funct3[0]^ (~Carry_flag)

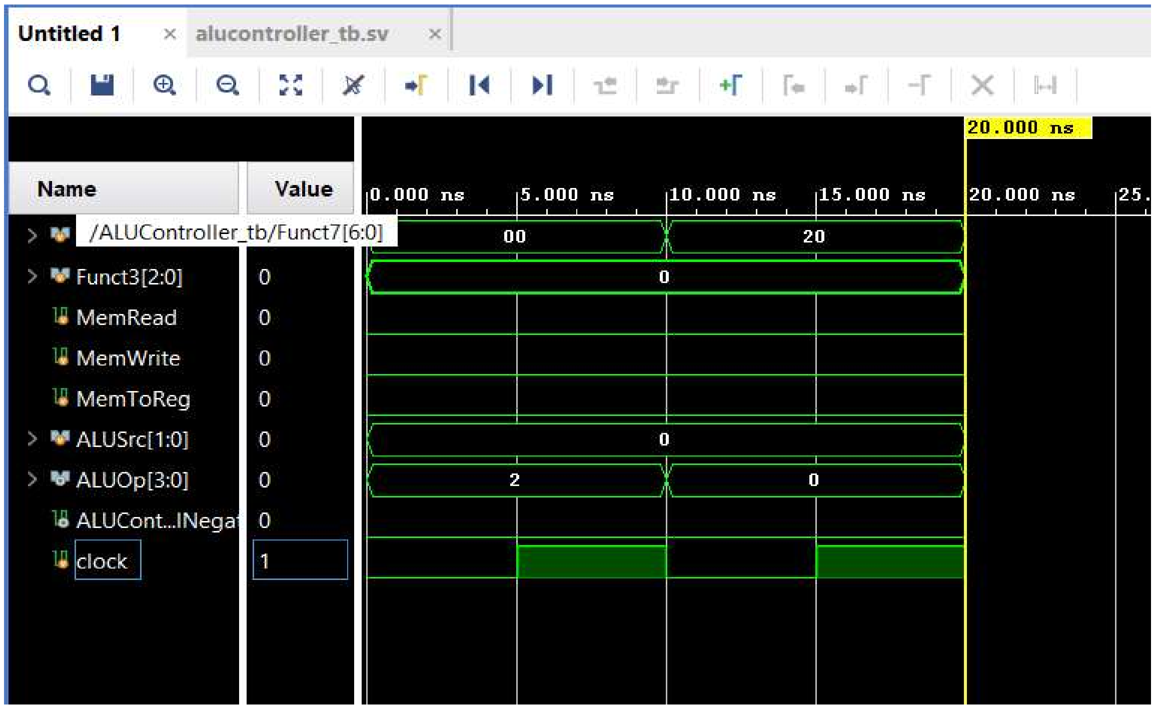
7. Program to check the processor



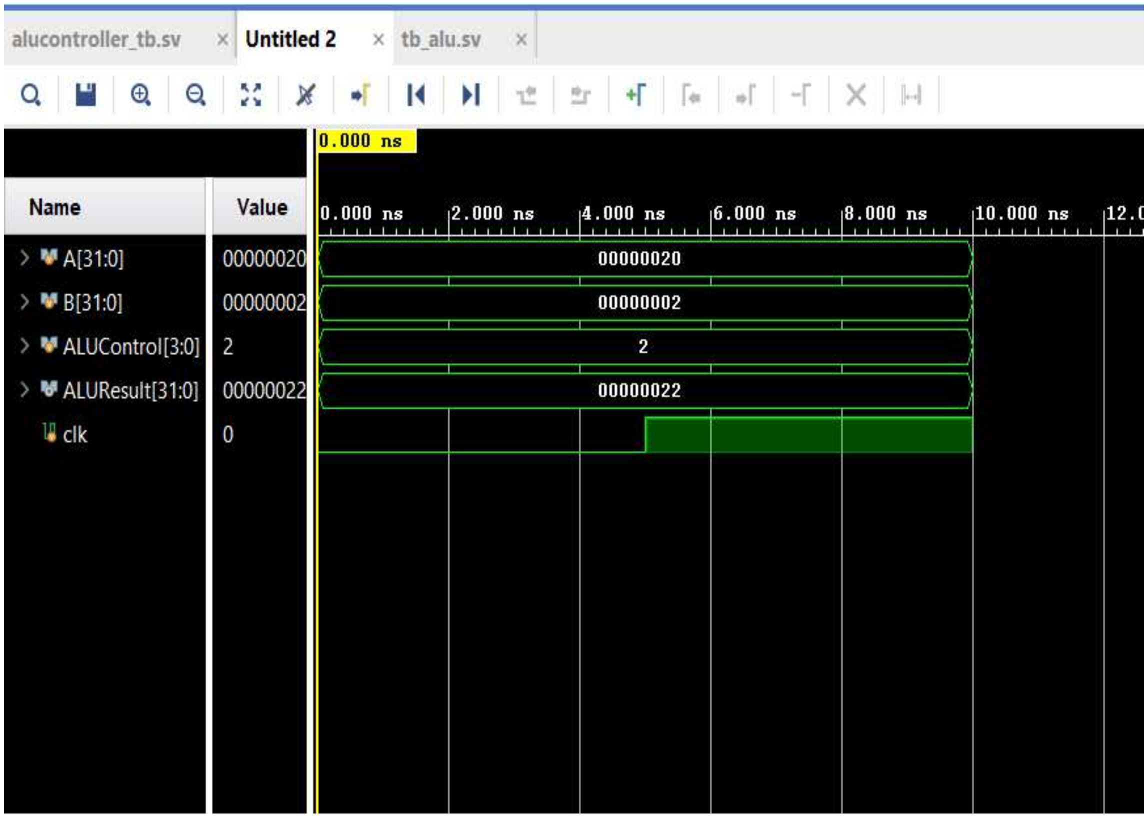
Label	Instruction	Expected	Address	Machinecode
main	addi x16,x0,40	X16 = 00000028	00	02800813
	addi x17,x0,-12	X17 = FFFFFFF4	04	FF400893
	addi x18,x0,40	X18 = 00000028	08	02800913
	addi x19,x0,56	X19 = 00000038	0C	03800993
	lui x8,58	X8 = 0003A000	10	0003A437
	addi x8,x8,2000	X8 = 0003A7D0	14	7D040413
	sw x8,0(x0)	mem[0] = 0003A7D0	18	00802023
	sh x8,4(x0)	mem[1] = 0000A7D0	1C	00801223
	sb x8,8(x0)	mem[2] = 000000D0	20	00800423
	lb x3,0(x0)	x3 = FFFFFFD0	24	00000183
	lbu x4,0(x0)	x4 = 000000D0	28	00004203
	lh x5,0(x0)	x5 = FFFFA7D0	2C	00001283
	lhu x6,0(x0)	x6 = 0000A7D0	30	00005303
	lw x7,0(x0)	x7 = 0003A7D0	34	00002383
	addi x1,x0,5	x1 =00000005	38	00500093
	slti x2,x1,4	x2 = 00000000	3C	0040A113
	xori x2,x2,-27	x2 = FFFFFFE5	40	FE514113
	sltiu x3,x2,12	x3 = 00000000	44	00C13193
	ori x2,x3,12	x2 = 0000000C	48	00C1E113
	andi x3,x2,-1	x3 = 0000000C	4C	FFF17193
	slli x2,x3,1	x2 = 00000018	50	00119113
	srli x3,x2,1	x3 = 0000000C	54	00115193
	xori x2,x0,-27	x2 = FFFFFFE5	58	FE504113
	srai x3,x2,2	x3 = FFFFFFF9	5C	40215193
	srli x4,x3,28	x4 = 0000000F	60	01C1D213
	beq x16,x18,label 1	PCTarget = 00000080	64	01280E63
Label 2	sub x31,x30,x29	x31 = 00000008	68	41DF0FB3
	blt x17,x18,label 3	PCTarget = 00000090	6C	0328C263
Label 4	and x31,x29,x30	x31 = 00000002	70	01EEFFB3
	bltu x16,x17,label 5	PCTarget = 00000098	74	03186263
Label 6	add x31,x29,x30	x31 = 0000000C	78	01EE8FB3
	jal x24,end	PCTarget = 000000A0	7C	02400C6F
Label 1	addi x29,x0,2	x29 = 00000002	80	00200E93
	addi x30,x0,10	x30 = 0000000A	84	00A00F13
	add x31,x29,x30	x31 = 0000000C	88	01EE8FB3
	bne x16,x19,label 2	PCTarget = 00000068	8C	FD381EE3
Label 3	or x31,x29,x30	x31 = 0000000A	90	01EEEFB3
	bge x19,x16,label 4	PCTarget = 00000070	94	FD09DEE3
Label 5	xor x31,x29,x30	x31 = 00000008	98	01EECFB3
	bgeu x17,x19,label 6	PCTarget = 00000078	9C	FD38FEE3
End	slt x1,x1,x4	x1 = 00000001	A0	0040A0B3
	addi x3,x0,12	x3 = 0000000C	A4	00C00193
	sll x2,x3,x1	x2 = 00000018	A8	00119133
	srl x3,x2,x1	x3 = 0000000C	AC	001151B3
	xori x2,x0,-27	x2 =FFFFFFE5	B0	FE504113
	addi x1,x0,2	x1 = 0000002	B4	00200093
	sra x3,x2,x1	x3 = FFFFFFF9	B8	401151B3
	addi x1,x0,28	x1 = 0000001C	BC	01C00093
	sltu x4,x2,x1	x4 = 00000000	C0	00113233
	jalr x28,0(x0)	PCTarget = 00000000	C4	00000E67

8.Simmulataion results of modules

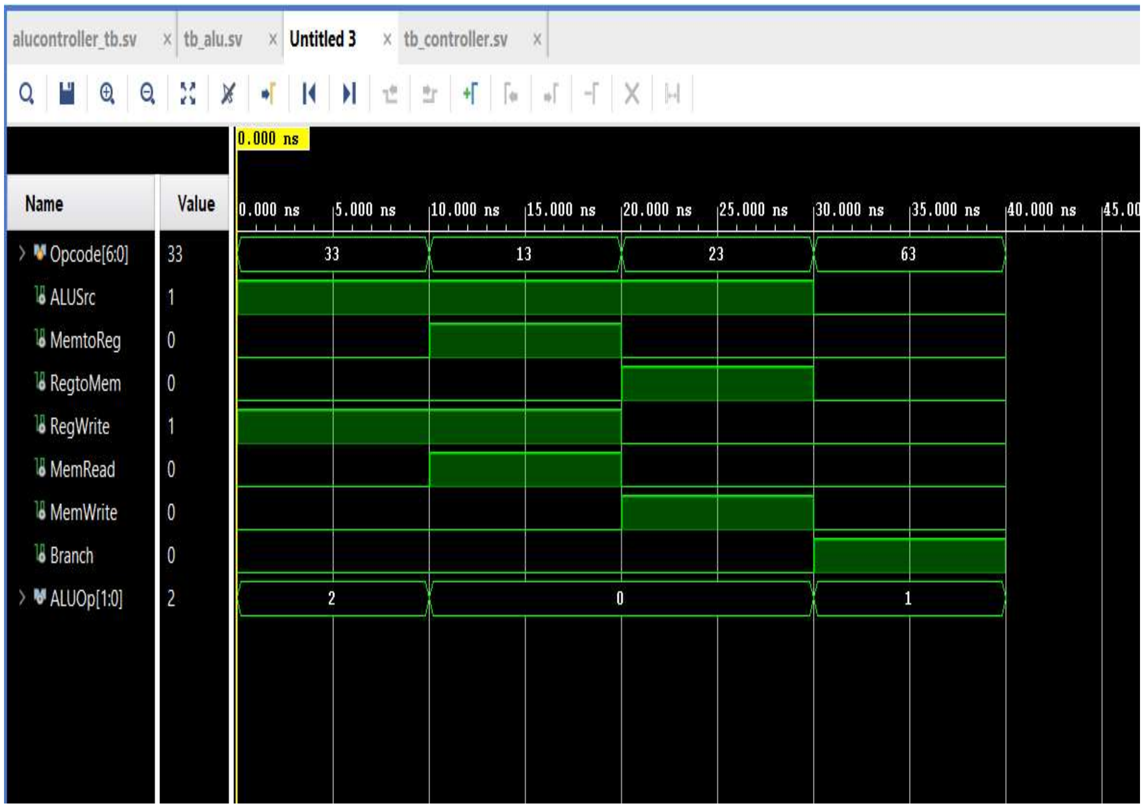
Alu controller



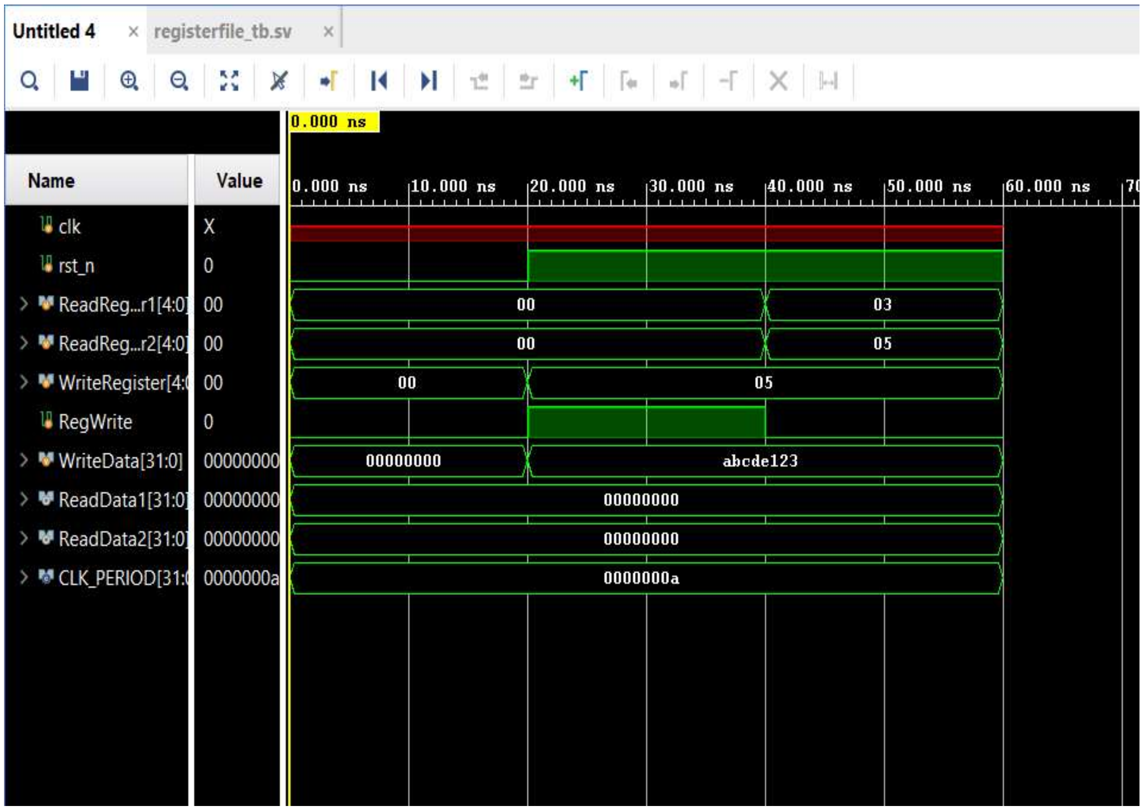
Alu



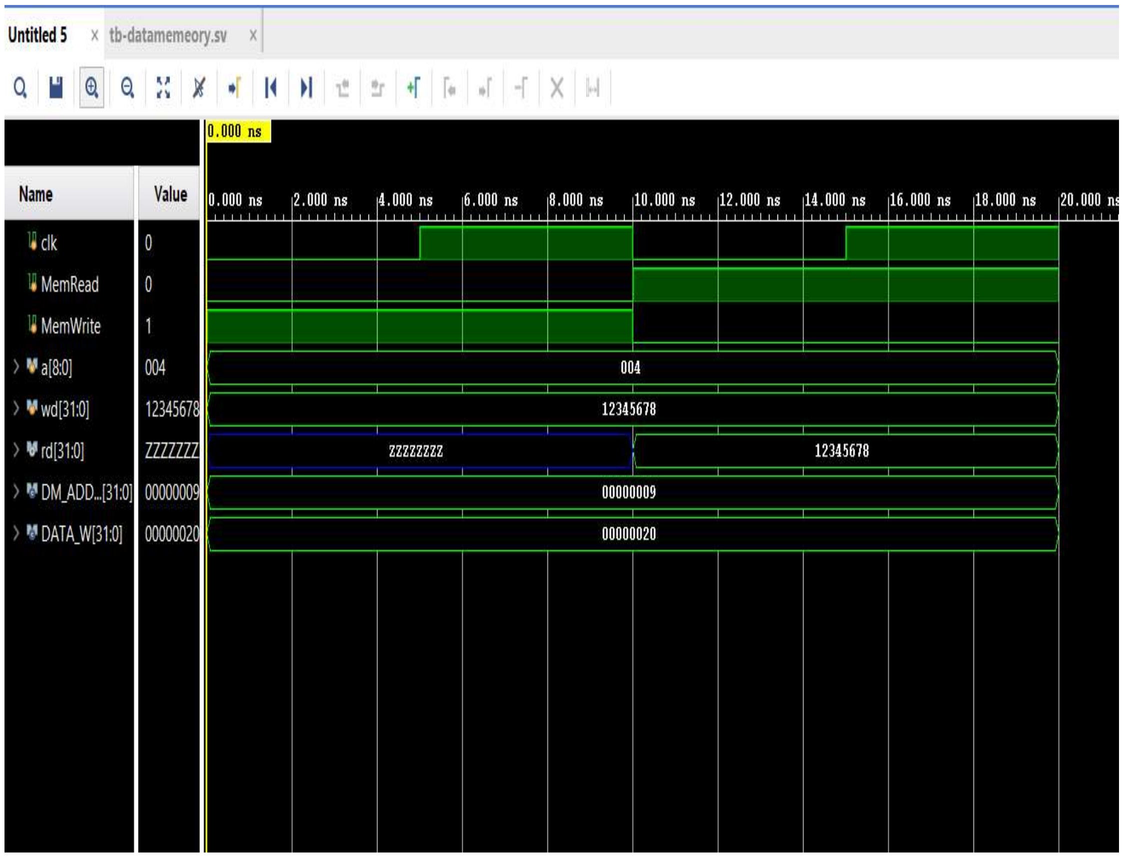
controller



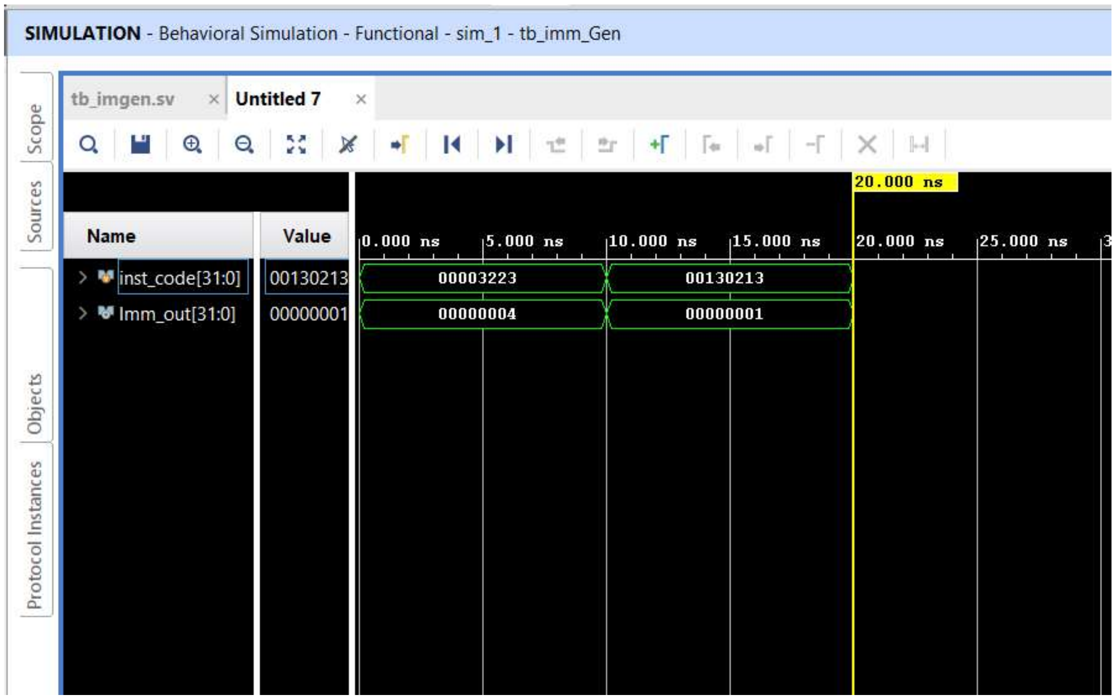
Registerfile



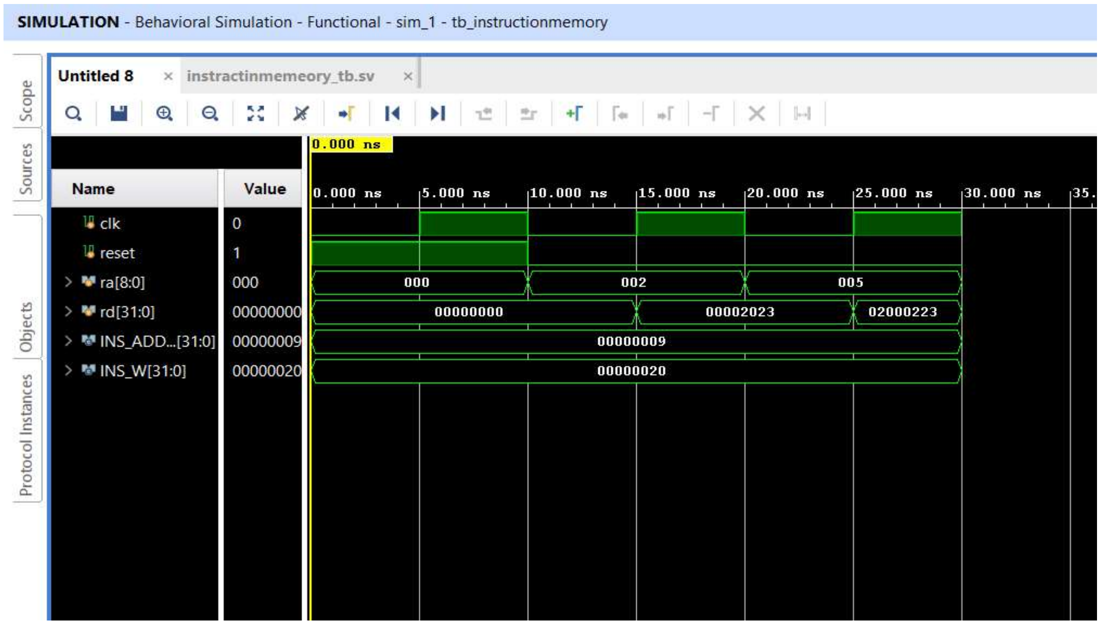
Datamemory



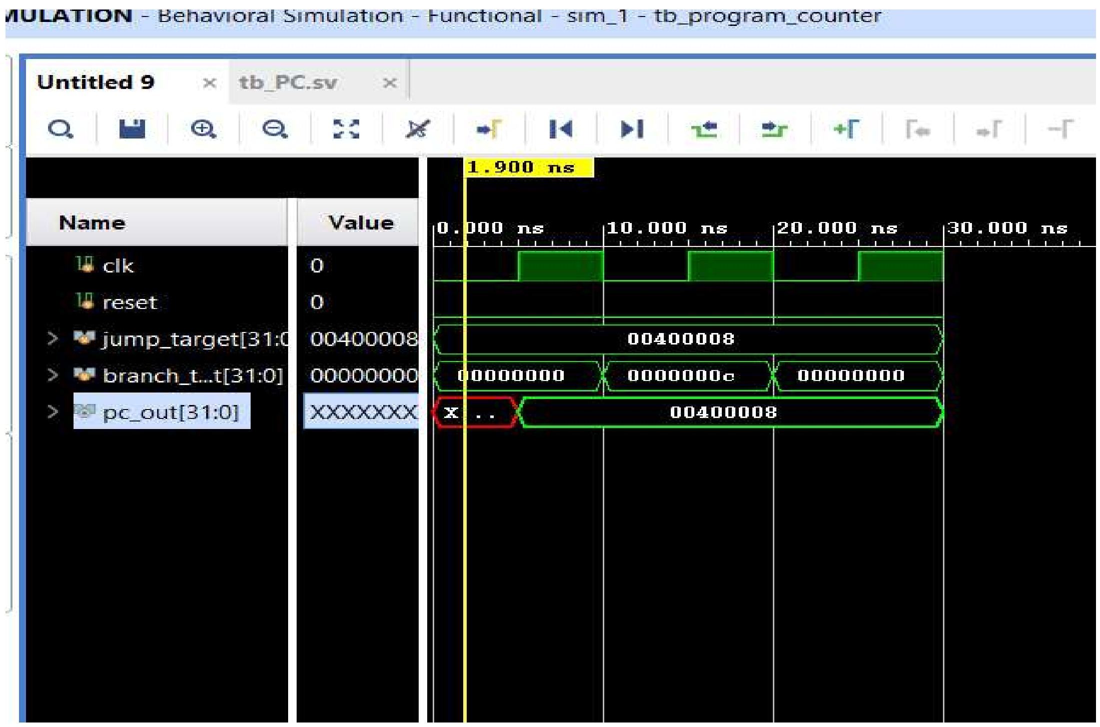
Immediate generator



Instruction memory



Program counter



My test bench code ,

```
module tb_final_data_path;

  reg clk;
  reg reset;
  wire [31:0] ALUResult;

  // Instantiate the final_data_path module
  final_data_path dut (
    .clk(clk),
    .reset(reset),
    .ALUResult(ALUResult)
  );

  // Define initial values for signals
  reg [7:0] pc_input;

  initial begin
    // Initialize clock and reset
    clk = 0;
    reset = 0;

    // Test case 1: and r0, r0, r0
    pc_input = 8'h00;
    #100;
    $display("Test Case 1: ALUResult = %h", ALUResult);

    // Test case 2: addi r1, r0, 1
    pc_input = 8'h04;
    #100;
    $display("Test Case 2: ALUResult = %h", ALUResult);

    // Test case 3: add r2, r0, 2
    pc_input = 8'h08;
    #100;
    $display("Test Case 3: ALUResult = %h", ALUResult);
```

```
// Test case 4: sub r3, r1, 3

pc_input = 8'h0C;

#100;

$display("Test Case 4: ALUResult = %h", ALUResult);


// Test case 5: sll r4, r1, 4

pc_input = 8'h10;

#100;

$display("Test Case 5: ALUResult = %h", ALUResult);


// Test case 6: slt r5, r2, 5

pc_input = 8'h14;

#100;

$display("Test Case 6: ALUResult = %h", ALUResult);


// Test case 7: sltu r6, r2, 6

pc_input = 8'h18;

#100;

$display("Test Case 7: ALUResult = %h", ALUResult);


// Test case 8: xor r7, r3, 7

pc_input = 8'h1C;

#100;

$display("Test Case 8: ALUResult = %h", ALUResult);


// Test case 9: srl r8, r1, r2

pc_input = 8'h20;

#100;

$display("Test Case 9: ALUResult = %h", ALUResult);


// Test case 10: sra r9, r8, r4

pc_input = 8'h24;

#100;

$display("Test Case 10: ALUResult = %h", ALUResult);


// Test case 11: and r10, r2, r3
```

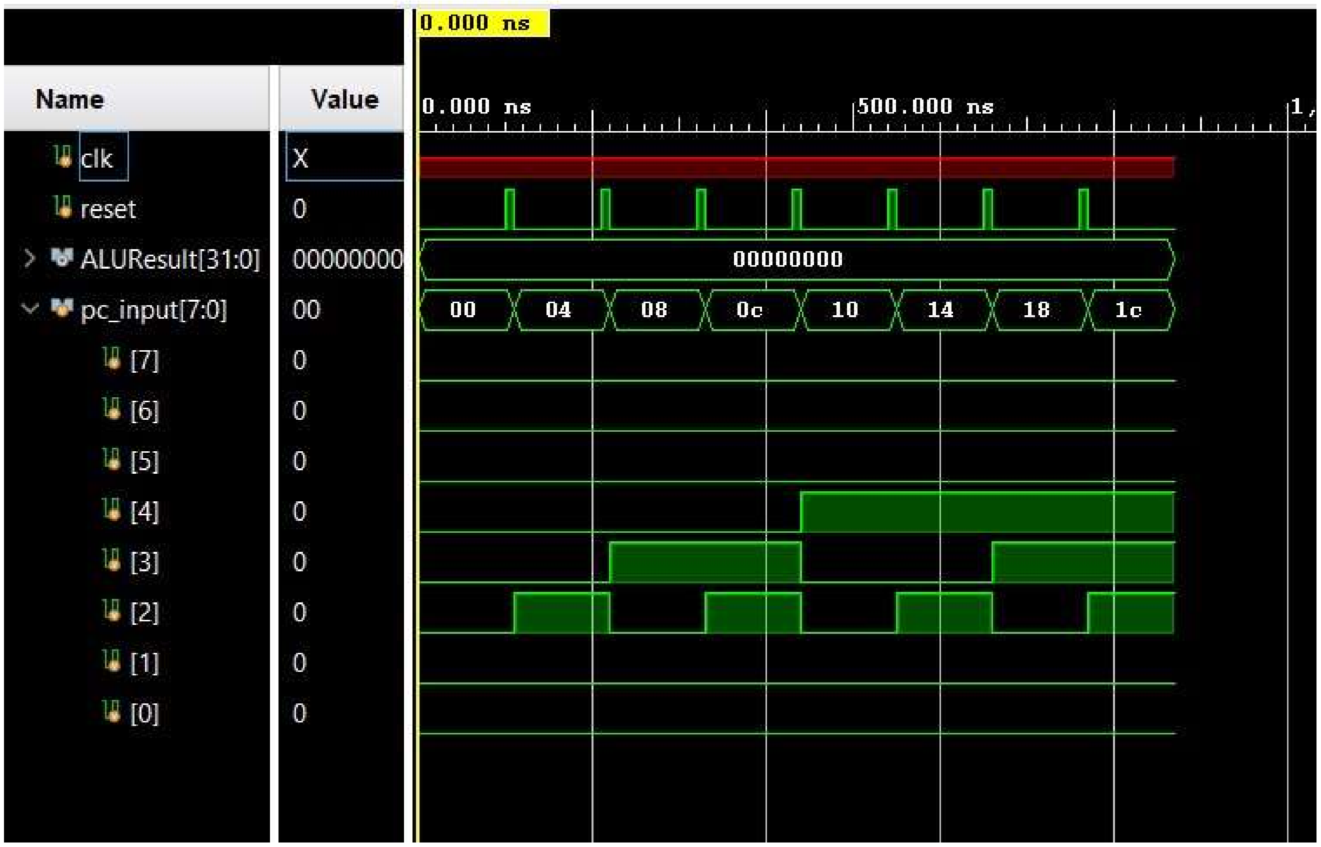
```
pc_input = 8'h28;

#100;

$display("Test Case 11: ALUResult = %h", ALUResult);


// Test case 12: or r11, r3, r4

pc_input = 8'h2C;

#100;

$display("Test Case 12: ALUResult = %h", ALUResult);


// Test case 13: mul

pc_input = 8'h30;

#100;

$display("Test Case 13: ALUResult = %h", ALUResult);


// Test case 14: slti

pc_input = 8'h34;

#100;

$display("Test Case 14: ALUResult = %h", ALUResult);


// Test case 15: sltiu

pc_input = 8'h38;

#100;

$display("Test Case 15: ALUResult = %h", ALUResult);


// Test case 16: xori

pc_input = 8'h3C;

#100;

$display("Test Case 16: ALUResult = %h", ALUResult);


// Test case 17: ori

pc_input = 8'h40;

#100;

$display("Test Case 17: ALUResult = %h", ALUResult);


// Test case 18: andi

pc_input = 8'h44;

#100;
```

```
$display("Test Case 18: ALUResult = %h", ALUResult);


// Test case 19: sw

pc_input = 8'h48;

#100;

$display("Test Case 19: ALUResult = %h", ALUResult);


// Test case 20: sb

pc_input = 8'h4C;

#100;

$display("Test Case 20: ALUResult = %h", ALUResult);


// Test case 21: sh

pc_input = 8'h50;

#100;

$display("Test Case 21: ALUResult = %h", ALUResult);


// Test case 22: lb

pc_input = 8'h54;

#100;

$display("Test Case 22: ALUResult = %h", ALUResult);


// Test case 23: lh

pc_input = 8'h58;

#100;

$display("Test Case 23: ALUResult = %h", ALUResult);


// Test case 24: lbu

pc_input = 8'h5C;

#100;

$display("Test Case 24: ALUResult = %h", ALUResult);


// Test case 25: lb

pc_input = 8'h60;

#100;

$display("Test Case 25: ALUResult = %h", ALUResult);
```

```
// Test case 26: beq

pc_input = 8'h64;

#100;

$display("Test Case 26: ALUResult = %h", ALUResult);


// Test case 27: lb

pc_input = 8'h68;

#100;

$display("Test Case 27: ALUResult = %h", ALUResult);


// Test case 28: bne

pc_input = 8'h6C;

#100;

$display("Test Case 28: ALUResult = %h", ALUResult);


// Test case 29: lh

pc_input = 8'h70;

#100;

$display("Test Case 29: ALUResult = %h", ALUResult);


// Test case 30: blt

pc_input = 8'h74;

#100;

$display("Test Case 30: ALUResult = %h", ALUResult);


// Test case 31: lh

pc_input = 8'h78;

#100;

$display("Test Case 31: ALUResult = %h", ALUResult);


// Test case 32: bgt

pc_input = 8'h7C;

#100;

$display("Test Case 32: ALUResult = %h", ALUResult);


// Test case 33: lh

pc_input = 8'h80;
```

```
        #100;
        $display("Test Case 33: ALUResult = %h", ALUResult);

        // Test case 34: jal
        pc_input = 8'h84;
        #100;
        $display("Test Case 34: ALUResult = %h", ALUResult);

        // Test case 35: lh
        pc_input = 8'h88;
        #100;
        $display("Test Case 35: ALUResult = %h", ALUResult);

        // Test case 36: lh
        pc_input = 8'h8C;
        #100;
        $display("Test Case 36: ALUResult = %h", ALUResult);

        $finish;
    end

    always begin
        #5 clk = ~clk;
    end

endmodule
```

Finally I got in simulation,



(all instruction are not include above simulation)

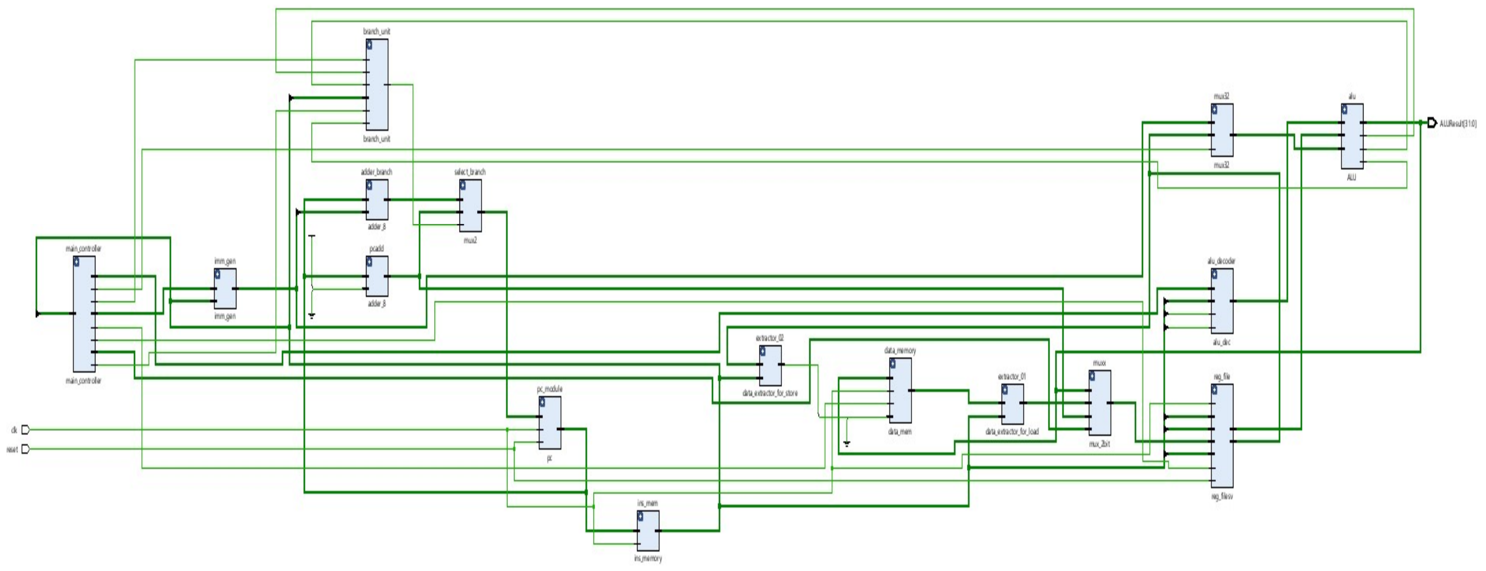
9.How to get data parth

In my processor design, the data path is a crucial component responsible for the execution of instructions. It consists of several functional units, each contributing to the processing of instructions. The program counter (PC) keeps track of the current instruction address, while an adder increments it by 4 in preparation for the next instruction. The instruction memory (ins_memory) fetches the instruction from memory based on the PC address. A register file (reg_file) stores and retrieves data as needed for operations.

The Arithmetic Logic Unit (ALU) is responsible for executing arithmetic and logic operations, with control signals influenced by the main controller and ALU decoder. The immediate generator (imm_gen) produces immediate values used in instructions. A multiplexer (mux32) selects data for ALU operations, which is often sourced from register values. A data memory unit (data_mem) reads from and writes to memory when necessary.

The main controller, together with the ALU decoder, determines the control signals for various operations, including ALU operations and data memory writes. Data extraction units help extract specific data from instruction fields, facilitating operations. The branch unit is responsible for handling branch instructions, adjusting the PC accordingly. Our data path is designed to efficiently execute a variety of instructions and support the core functionalities of our processor.

10.Schmetic Diagram



11.Extending the processor to support

1) MUL Operation:

The MUL (Unsigned Multiplication) instruction can be implemented as an R-type instruction with the following instruction encoding: 0100000 | rs2 | rs1 | 111 | rd | 0110011. This instruction performs the multiplication of two source registers, rs1 and rs2, and stores the least significant 32 bits of the 64-bit result in the destination register rd. To enable the MUL operation in the datapath, the ALUdecoder ROM table needs to be updated to generate the appropriate ALU control signals for multiplication in the ALU. It's important to note that the ALU generates a 64-bit result during multiplication, but we only consider the least significant 32 bits due to the 32-bit bus width of the processor. Therefore, this operation has a limitation: it cannot multiply numbers with a total bit length exceeding 32 bits.

2) MEMCOPY Operation:

The MEMCOPY instruction is conceptually similar to an S-type instruction but requires different control signal generation compared to typical S-type instructions. It involves three operands:

i. Source Address

ii. Destination Address

iii. N (Length of the data to be copied in bytes)

The MEMCOPY instruction copies the content of N bytes from the source address and replaces the first N bytes at the destination address. To support these requirements, the data memory should be designed as a two-port memory, with both ports being readable and writable. Both memory ports are utilized only when the MEMCOPY instruction is executed. This enables simultaneous reading from the source address and writing to the destination address, making MEMCOPY an efficient memory operation.

12.Final Data parth

```
module final_data_path
   (

     input logic clk,reset,
                  output logic [31:0] ALUResult
     );

                   logic [31:0] write_data;
                  logic Con_BGT;
                  logic Con_BLT;
                   logic  ALUsrc;
                  logic [1:0] MemtoReg;
                  logic rg_wrt_en;
                  logic [1:0] aluop;
                  logic [3:0] Operation;
                   logic [1:0] imm_src;
                   logic we;
                   logic [7:0] pc_input;
                  logic [7:0] pc_plus_4;
                  logic PC_src;
                  logic [7:0] pc_output;
                   logic [7:0] ext_imm;
                   logic [31:0] rg_rd_data1;
      logic [31:0] rg_rd_data2;
                   logic [31:0] Imm_out;
                   logic [31:0] mux_out;
                   logic [31:0] extractor_out;
                   logic [31:0] read_data;
```

```
                 logic zero;

                 logic [31:0] instruction;

                 logic jump;

                 logic branch;



                pc pc_module (

                .a(pc_input),

                .clk(clk),

                .reset(reset),

                .y(pc_output)

                );


                adder_8  pcadd (

                .a(pc_output),

                .b(8'b00000100),

                .y(pc_plus_4)

                );


ins_memory ins_mem(

                .clk(clk),

                .address_in(pc_output),

                .instruction_out(instruction)

                );


                reg_filesv reg_file(

                .clk(clk),

                .rst(reset),

                .rg_wrt_en(rg_wrt_en),
.rg_wrt_dest(instruction[11:7]),
.rg_rd_addr1(instruction[19:15]),
.rg_rd_addr2(instruction[24:20]),
.rg_wrt_data(write_data),
.rg_rd_data1(rg_rd_data1),
.rg_rd_data2(rg_rd_data2)
);

ALU alu(
.SrcA(rg_rd_data1),
.SrcB(mux_out),
.Operation(Operation),
.ALUResult(ALUResult),
.Con_BLT(Con_BLT),
.Con_BGT(Con_BGT),
.zero(zero)
);

imm_gen imm_gen(
.imm_src(imm_src),
.inst_code(instruction),
.Imm_out(Imm_out)
);


mux32 mux32(
.d0(Imm_out),
.d1(rg_rd_data2),
.s(ALUsrc),
                .y(mux_out)
                );

                data_mem data_memory(
                .we(we),
.clk(clk),
.write_data(store_extractor),
.address(ALUResult),
.read_data(read_data)
                );

                mux_2bit muxx(
                .a(ALUResult),
                .b(extractor_out),
                .c(pc_plus_4),
                .sel(MemtoReg),
                .y(write_data)
                );



                main_controller main_controller(
                .Opcode(instruction[6:0]),
                .ALUSrc(ALUsrc),
                .Result_Src(MemtoReg),
                .RegWrite(rg_wrt_en),
                .Imm_Src(imm_src),
                .MemWrite(we),
                .ALUOp(aluop),
.jump(jump),
.Branch(branch)
);

alu_dec alu_decoder(
.f7b5(instruction[30]),
.op5(instruction[5]),
.func3(instruction[14:12]),
.aluop(aluop),
.alucontrol(Operation)
);

data_extractor_for_load extractor_01(
.inst(instruction[31:0] ),
.data(read_data),
.y(extractor_out)
);

data_extractor_for_store extractor_02(
.inst(instruction[31:0] ),
.data(rg_rd_data2),
.y(store_extractor)
);

adder_8 adder_branch(
.a(pc_output),
.b(Imm_out[7:0]),
.y(ext_imm)
);
```

```
		mux2 select_branch(
		.d0(ext_imm),
		.d1(pc_plus_4),
		.s(PC_src),
		.y(pc_input)
		);

		branch_unit branch_unit(
		.funct3(instruction[14:12]),
		.Branch(branch),
		.zero(zero),
		.jump(jump),
		.Con_BLT(Con_BLT),
		.Con_BGT(Con_BGT),
		.PC_Src(PC_src)
		);

endmodule
```

13. Conclusion

Efficient resource utilization stands as one of the cornerstones in the realm of FPGA (Field-Programmable Gate Array) design. It's not just a desirable trait; it's a fundamental necessity. In the world of digital design, where flexibility and customization are paramount, the ability to make the most of available resources is the linchpin for achieving cost-effective, high-performance, and power-efficient designs. The strategies expounded upon in this comprehensive report serve as a roadmap for designers seeking to harness the full potential of FPGA resources while ensuring that the functional requirements of their designs are not only met but exceeded. With the implementation of the insights contained herein, designers can navigate the intricate landscape of FPGA design, ensuring that every logic element, memory block, and interconnect resource is meticulously orchestrated to serve the overarching goals of their projects. In doing so, they pave the way for cutting-edge solutions that are not only resource-efficient but also capable of pushing the boundaries of performance and efficiency.