Design and Implementation of a 16bit-RISC CPU using Verilog

Anish De – B22194
Anirudh Vijan – B22193
Amuel – B22192
Anshul – BB22196

Abstract

This project is about building a 16-bit RISC CPU on verilog/VHDL and also implementing it on FPGA board. We implemented an Instruction Set Architecture (ISA) which can perform simple programmes such as arithmetic operations, logic and control of programs. The CPU is consists of one memory interface, the datapath, and the control unit. It is further tested on the Vivado simulation tool and implemented on a Xilinx FPGA. This report tells about the project in detail, like how the ISA was made, how the datapath and control unit work in the model.

Project Description

Our main objective was to design a 16-bit RISC CPU that meets the following specifications:
1.The ISA should Supports no more than 15 instructions, each encoded within 16 bits.
2.The ISA is to support linear addressing of 8K, 16-bit words memory.
3.The model should has some programmable registers and a 16-bit datapath.

This processor executes instructions using a modular design comprising the datapath and control unit, verified in simulations and implemented on FPGA hardware.

A. Instruction Set Architecture (ISA)

1. Registers

Registers are small high speed storage units present in the cpu itself to assit the processor during instructions exectution.

R0-R7: General-purpose registers.

- **PC**: Program Counter is responsible for keeping an account on the sequence of instructions.
- **IR**: Instruction Register stores the instructions fetched from the memory.

2. Instruction Formats

The CPU uses a fixed 16-bit instruction format:

- **Opcode**: 4 bits.
- **Operands**: 12 bits (register/memory addresses or immediate values).

3. Instruction Set

Mnemonic	**Operation**	**Opcode**	**Description**
ADD	Add	0000	R0 = R0 + Rx
SUB	Subtract	0001	R0 = R0 - Rx
AND	Logical AND	0010	R0 = R0 & Rx
OR	Logical OR	0011	R0 = R0
NOT	Logical NOT	0100	R0 = ~R0
LOAD	Load from memory	0101	R0 = M[Addr]
STORE	Store to memory	0110	M[Addr] = R0
JMP	Unconditional jump	0111	PC = Addr
JNZ	Jump if not zero	1000	if R0 != 0 then PC = Addr
HALT	Halt execution	1111	Stop the CPU

OP Code (Binary)	**Operation**
0000	**Load Word**
0001	**Store Word**
0010	**Add**
0011	**Subtract**
0100	**Invert (1's complement)**
0101	**Logical Shift Left**

OP Code (Binary)	**Operation**
0110	**Logical Shift Right**
0111	**Bitwise AND**
1000	**Bitwise OR**
1001	**Set on Less Than**
1010	**Hamming Distance**
1011	**Branch on Equal**
1100	**Branch on Not Equal**
1101	**Jump**
1111	**Halt**

Instruction Format

Memory Access: Load

Field	**Op**	**Rs1**	**Ws**	**Offset**
Bit Width	4	3	3	6

Memory Access: Store

Field	**Op**	**Rs1**	**Rs2**	**Offset**
Bit Width	4	3	3	6

Data Processing

Field	**Op**	**Rs1**	**Rs2**	**Ws**	**Useless**
Bit Width	4	3	3	3	3

Branch

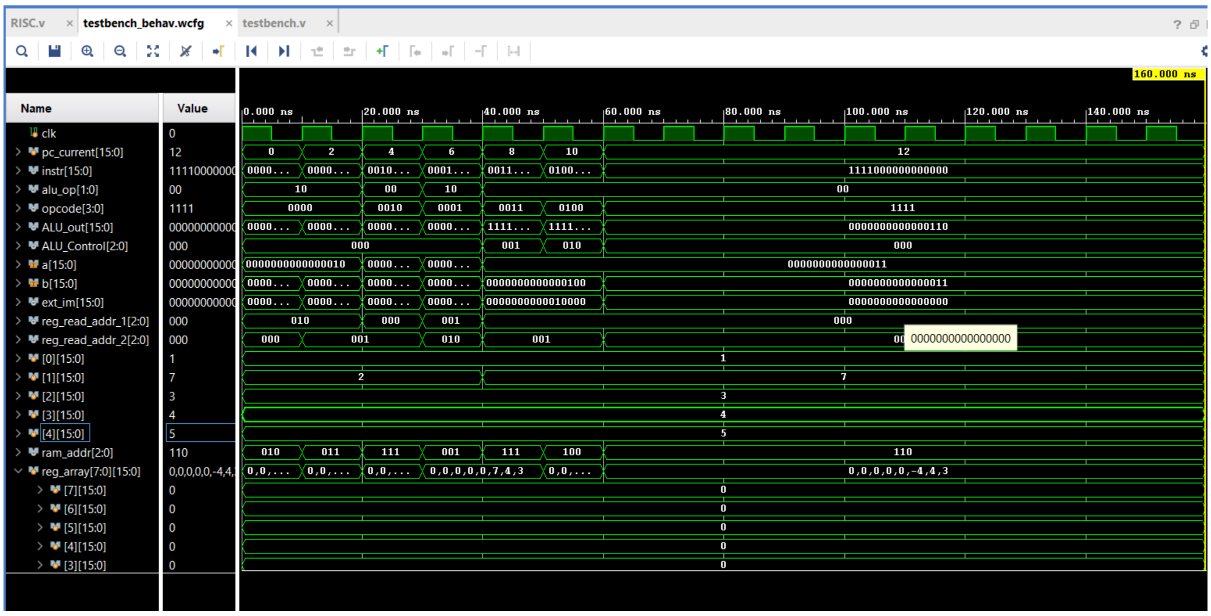
Field	**Op**	**Rs1**	**Rs2**	**Offset**
Bit Width	4	3	3	6

Jump

Field	Op	Offset
Bit Width	4	12

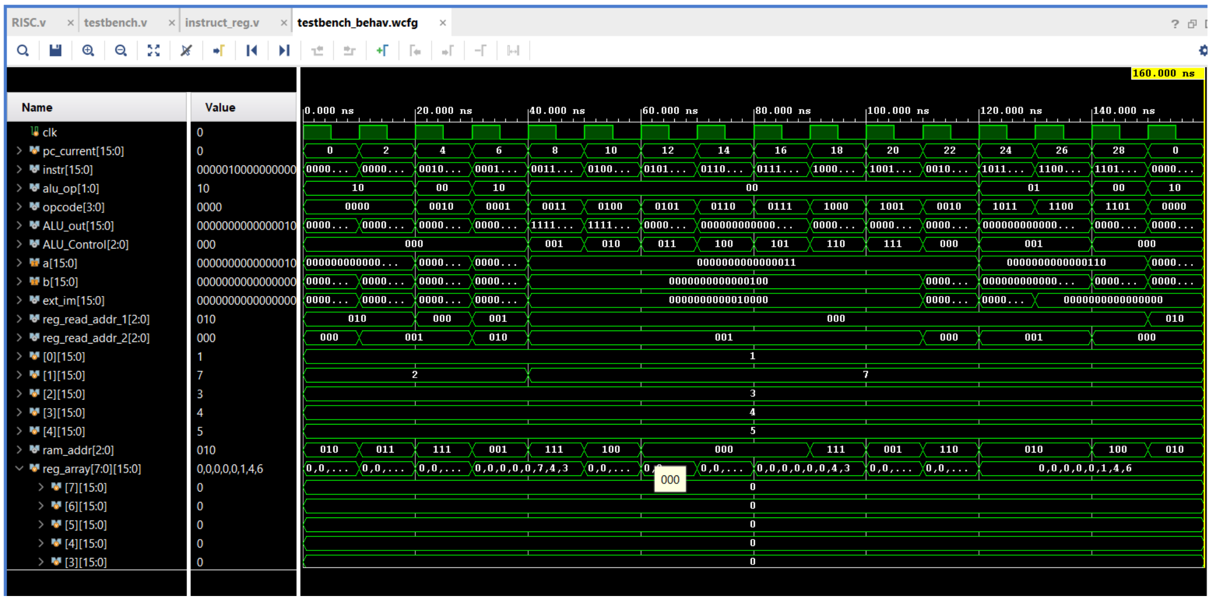
Processor Control Unit Design:

Control signals									
Instruction	Reg Dst	ALUSrc	Memto Reg	Reg Write	MemRead	Mem Write	Branch	ALUOp	Jump
Data-processing	1	0	0	1	0	0	0	00	0
LW	0	1	1	1	1	0	0	10	0
SW	0	1	0	0	0	1	0	10	0
BEQ, BNE	0	0	0	0	0	0	1	01	0
J	0	0	0	0	0	0	0	00	1



Name	Value
clk	0
pc_current[15:0]	12
instr[15:0]	11110000000
alu_op[1:0]	00
opcode[3:0]	1111
ALU_out[15:0]	00000000000
ALU_Control[2:0]	000
a[15:0]	00000000000
b[15:0]	00000000000
ext_im[15:0]	00000000000
reg_read_addr_1[2:0]	000
reg_read_addr_2[2:0]	000
[0][15:0]	1
[1][15:0]	7
[2][15:0]	3
[3][15:0]	4
[4][15:0]	5
ram_addr[2:0]	110
reg_array[7:0][15:0]	0,0,0,0,0,-4,4,
[7][15:0]	0
[6][15:0]	0
[5][15:0]	0
[4][15:0]	0
[3][15:0]	0

Full simulation. With halt

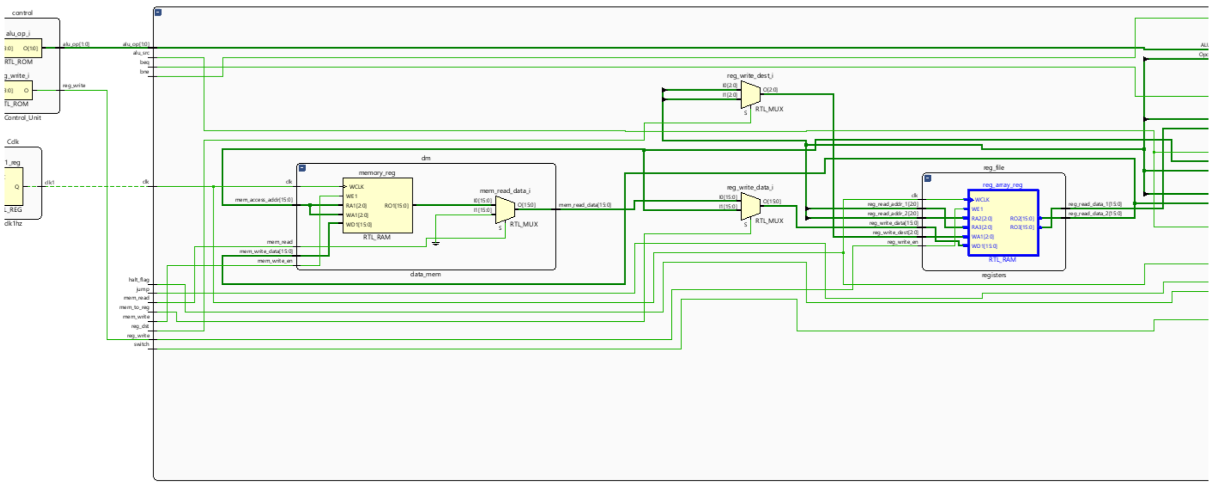


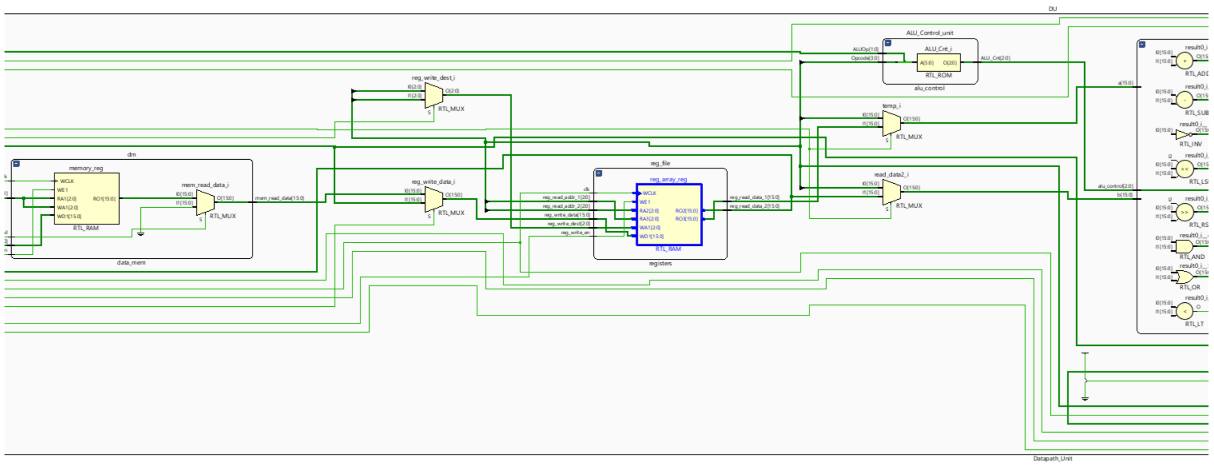
Name	Value
clk	0
pc_current[15:0]	0
instr[15:0]	0000010000000000
alu_op[1:0]	10
opcode[3:0]	0000
ALU_out[15:0]	0000000000000010
ALU_Control[2:0]	000
a[15:0]	0000000000000010
b[15:0]	0000000000000000
ext_im[15:0]	0000000000000000
reg_read_addr_1[2:0]	010
reg_read_addr_2[2:0]	000
[0][15:0]	1
[1][15:0]	7
[2][15:0]	3
[3][15:0]	4
[4][15:0]	5
ram_addr[2:0]	010
reg_array[7:0][15:0]	0,0,0,0,0,1,4,6
[7][15:0]	0
[6][15:0]	0
[5][15:0]	0
[4][15:0]	0
[3][15:0]	0

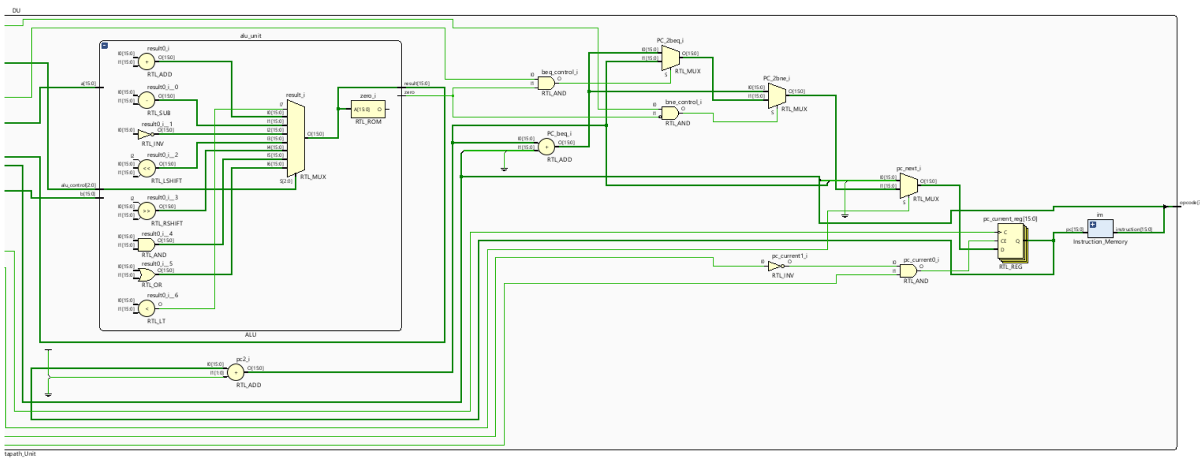
Full simulation. Without halt

B. Datapath Architecture

1. Block Diagram







2. Component Descriptions

- **ALU**: Performs arithmetic and logical operations as per the opcode.
- **Register File**: Stores general-purpose registers (R0-R7).
- **Program Counter (PC)**: Holds the address of the next instruction.

- **Instruction Register (IR)**: Temporarily holds the current instruction.
- **Memory**: Stores instructions and data.

3. Instruction Execution

Stage	**Register Transfer**
Fetch	IR = M[PC], PC = PC + 1
Decode	Decode the opcode and determine control signals
Execute	Perform the required operation (e.g., R0 = R0 + Rx for ADD)

1. **Design Decisions**

Cost vs. Speed Tradeoffs

- A multi-cycle design was chosen while considering hardware usage and cost. It reduces hardwire requirement by reusing same component multiple times accross each clock size. A single-cycle design works faster for simpler instructions, would increase cost due to duplicated hardware and wasted cycles for complex instructions.

Single-Cycle vs. Multi-Cycle Design

A single-cycle design executes every instruction in one clock cycle. This architecture has the following characteristics:

- **Simpler Control Logic**: All instructions are completed in one clock cycle, leading to straightforward control logic without the need for additional states or control signals for instruction execution.
- **Higher Throughput for Simple Instructions**: Since every instruction completes in one cycle, the throughput for simple instructions can be higher, making it suitable for workloads dominated by less complex operations.

Shared vs. Dedicated Components

- Copies such as ALU and memory access units were used for several cycles to reduce the hardware cost. Critical parallel activities were assigned dedicated hardware resources, like the Program Counter (PC), and the Register File, while not sacrificing efficiency by potential bottlenecks.

Edge-Triggered vs. Latching Registers

- On the basis of critical control timing and prevention of data hazards; edge triggered registers were chosen for a multi-cycle design. This helps in having synchronous activities avoiding errors due to latching level sensitivity.

Other Design Considerations

- Adopted Linear addressing memory to reduce control logics. Modular design makes easy to add new instruction in the future. Further grouping of control signals was logically done for easy debugging, and clock gating has been considered for efficient low power while idle cycles.

C. Datapath Verification

1. Simulation Results

Simulation of the CPU was conducted in VIVADO to verify:

1. Correct fetch, decode, and execute operations.
2. Accurate register and memory updates.

2. Control Signal Table

Signal	Fetch	Decode	Execute
ALU_op	0	0	1
MemRead	1	0	0
MemWrite	0	0	1

3. Sample Code

Example Verilog code for the clock divider used in FPGA implementation:

```
module clk1hz(
    input clk,
    output reg clk1
    );
reg [26:0] temp = 0;
always @(posedge clk)
begin
    if (temp == 27'd99_999_999)
    begin
        temp <= 0;
        clk1 <= ~clk1;
    end
    else
        temp <= temp + 1;
end
```

```
endmodule




module ALU(
 input  [15:0] a,  //src1
 input  [15:0] b,  //src2
 input  [2:0] alu_control, //function sel

 output reg [15:0] result,  //result
 output zero
    );

always @(*)
begin
 case(alu_control)
 3'b000: result = a + b; // add
 3'b001: result = a - b; // sub
 3'b010: result = ~a;
 3'b011: result = a<<b;
 3'b100: result = a>>b;
 3'b101: result = a & b; // and
 3'b110: result = a | b; // or
 3'b111: begin if (a<b) result = 16'd1;
    else result = 16'd0;
    end
 default:result = a + b; // add
 endcase
end
assign zero = (result==16'd0) ? 1'b1: 1'b0;
```

```
endmodule
```

D. Controller Design

The control unit generates signals to guide datapath operations based on instruction opcodes. It was verified through simulation and integrated with the datapath for testing.

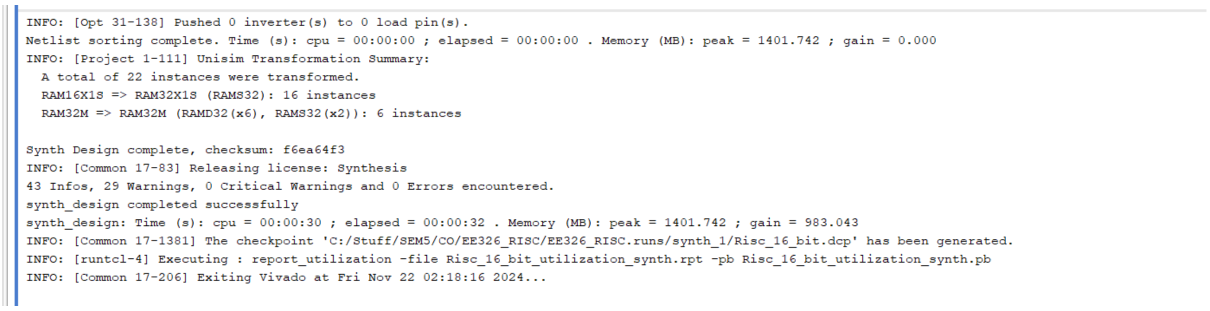
```
`timescale 1ns / 1ps

// Verilog code for RISC Processor

// Verilog code for Control Unit

module Control_Unit(

      input[3:0] opcode,

      output reg[1:0] alu_op,

      output reg
halt_flag,jump,beq,bne,mem_read,mem_write,alu_src,reg_dst,mem_to_reg,reg_write

    );


always @(*)

begin

 case(opcode)

 4'b0000:  // LW

   begin

    reg_dst = 1'b0;

    alu_src = 1'b1;

    mem_to_reg = 1'b1;

    reg_write = 1'b1;

    mem_read = 1'b1;

    mem_write = 1'b0;

    beq = 1'b0;

    bne = 1'b0;

    alu_op = 2'b10;

    jump = 1'b0;

    halt_flag = 1'b0;  // Custom signal to indicate halt
```

Example of one opcode.

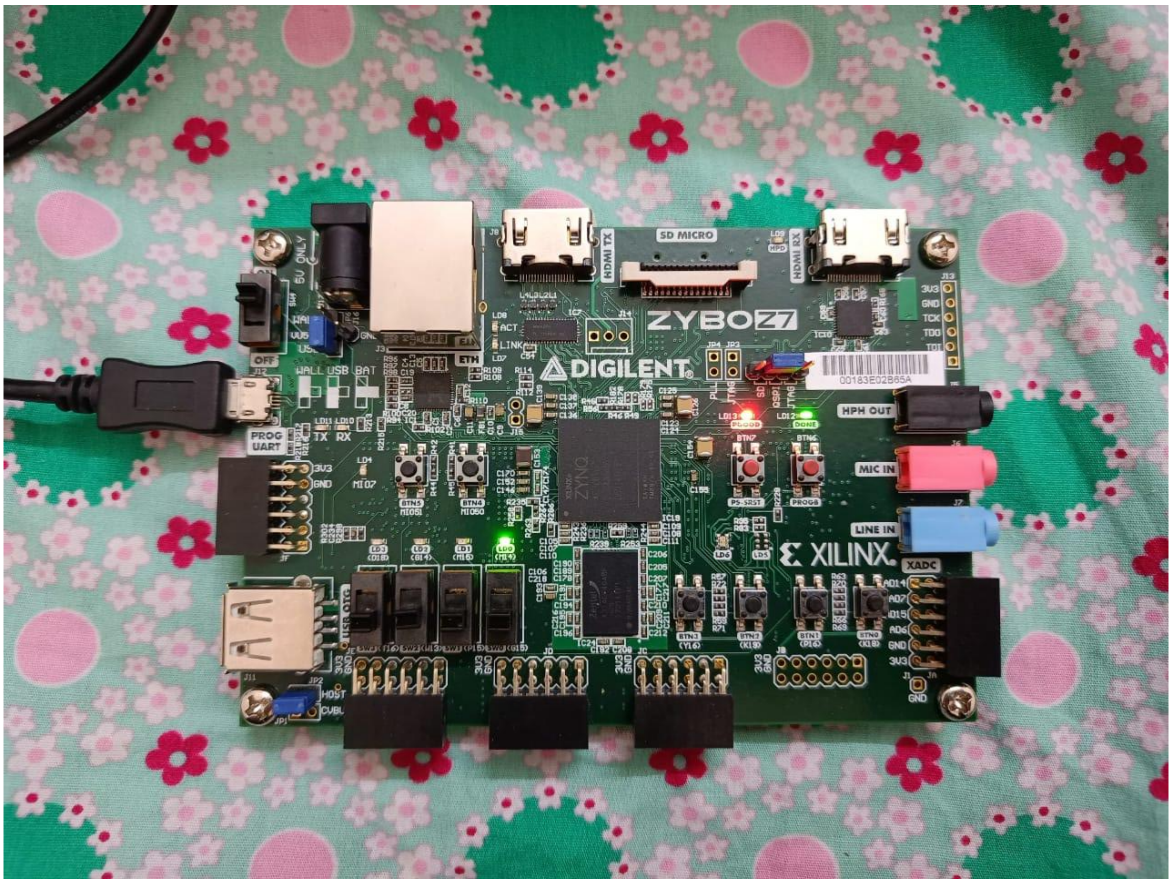
E. FPGA Implementation

The CPU was synthesized and implemented on a Xilinx FPGA. A test program was executed to demonstrate:

1. Arithmetic and logical operations.
2. Control flow via jump instructions.
3. HALT instruction to stop execution.



```
INFO: [Opt 31-138] Pushed 0 inverter(s) to 0 load pin(s).
Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak = 1401.742 ; gain = 0.000
INFO: [Project 1-111] Unisim Transformation Summary:
  A total of 22 instances were transformed.
  RAM16X1S => RAM32X1S (RAMS32): 16 instances
  RAM32M => RAM32M (RAMD32(x6), RAMS32(x2)): 6 instances

Synth Design complete, checksum: f6ea64f3
INFO: [Common 17-83] Releasing license: Synthesis
43 Infos, 29 Warnings, 0 Critical Warnings and 0 Errors encountered.
synth_design completed successfully
synth_design: Time (s): cpu = 00:00:30 ; elapsed = 00:00:32 . Memory (MB): peak = 1401.742 ; gain = 983.043
INFO: [Common 17-1381] The checkpoint 'C:/Stuff/SEM5/CO/EE326_RISC/EE326_RISC.runs/synth_1/Risc_16_bit.dcp' has been generated.
INFO: [runtcl-4] Executing : report_utilization -file Risc_16_bit_utilization_synth.rpt -pb Risc_16_bit_utilization_synth.pb
INFO: [Common 17-206] Exiting Vivado at Fri Nov 22 02:18:16 2024...
```



F. Remarks

(a) What did you learn from this project?

This project provided a comprehensive understanding of processor design, encompassing both theoretical and practical aspects. Key takeaways include:

- Instruction Set Architecture (ISA) Design: Gained insights into encoding instructions, defining registers, and balancing simplicity with functionality.
- Datapath and Control Unit Development: Understood the problems of designing multi-cycle datapaths and the importance of control signal timing.
- Hardware Description Languages (HDL): Improved proficiency in VHDL/Verilog for modeling and simulating digital designs.
- FPGA Implementation: Learned how to map theoretical designs to physical hardware, addressing real-world challenges like resource constraints and clock synchronization and when a design is synthesizable or not.

(b) How can you possibly improve the performance of the designed processor?

Several enhancements could improve the processor's performance:

- Pipelining: Introducing pipelining could significantly increase throughput by executing multiple instructions simultaneously.
- Instruction-Level Optimization: Including additional instructions, like multiplication or shift operations, would reduce the instruction count for certain tasks.
- Cache Integration: Adding an on-chip cache could reduce memory access latency and improve overall speed.
- Clock Rate Adjustment: Optimizing the clock rate by balancing speed with power consumption and thermal limits would enhance performance.

(c) What would you do differently next time?

If given the opportunity to redo the project, the following changes would be made:

- Adopt a Modular Design Approach: This would simplify debugging and allow for easier reuse of components in future designs.
- Use a Simulation-Driven Workflow: Testing components individually in more detail before integration would reduce debugging time during final implementation.
- Focus on Power Efficiency: Integrating features like clock gating and low-power states for unused modules would improve energy efficiency.
- Implement More Complex Control Structures: Using microcode-based control could provide greater flexibility for handling additional instructions or features.

(d) What is your advice to someone who is going to work on a similar project?

- Start with a Clear Plan: Define your ISA, datapath, and control unit early, ensuring that every design decision aligns with your project requirements.

- Test Incrementally: Simulate and verify each component individually before integration. This reduces debugging complexity during the final stages.
- Allocate Time for FPGA Implementation: Hardware implementation often presents unforeseen challenges, so plan sufficient time for debugging and adjustments.
- Try new ways: Explore new techniques or tools to optimize your design and address challenges effectively.