PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

100-ft Ring Road, Bengaluru – 560 085, Karnataka, India

UE20EC302 - RISC-V Architecture

Report on

" Design of a 16 bit RISC-V Processor in Verilog"

Submitted by

AMOGH ASV (PES1UG20EC002)

ANIRUDHHAN R (PES1UG20EC029)

Aug - Dec 2022

Course Instructor: Prof. VANAMALA H

Design of a 16 bit RISC V Processor in Verilog

Aim:

The aim of this open ended project is to design a 16 bit RISC processor in verilog and simulate the functions offered

Abstract:

The Reduced Instruction Set Computer or RISC is a microprocessor design principle that favours a smaller and simpler set of instructions that all take the same amount of time to execute. RISC architecture is used across a wide range of platforms from cellular phones to supercomputers. The processor supports 16 instructions with three addressing modes. It has 16 general purpose registers. Each register can store 16-bit data. The processor has 16-bit ALU capable of performing 11 arithmetical and logical operations. The processor also incorporates a flag register which indicates carry, zero and parity status of the result. All the modules in the design are coded in Verilog. The individual modules are designed and tested at each level of implementation and finally integrated in a top level module by appropriate mapping.

INTRODUCTION:

When the controller design became more complex in CISC and the performance was also not up to expectations, people started looking at some other alternatives. It has been found that when a processor talks to the memory the speed gets killed. So the one improvement on CPI was to keep the instruction set very simple. Simple in not the way it works but the way it looks. That’s why there are very few instructions in any typical RISC architecture where the processor asks data from memory, probably not other than Load and Store. At the end the pipelining added a new dimension in the speed just with the help of some additional registers, which increases throughput by reducing CPI. Hence the instruction can be executed effectively in one clock cycle.

Over the years, RISC instruction sets have grown in size and today many of them have a larger set of instructions than many CISC CPUs. The term "Reduced" in that phrase was intended to describe the fact that the amount of work any single instruction accomplishes is reduced at most a single data memory cycle compared to the "complex instructions" of CISC CPUs.

RISC architecture greatly boosts computer speed by using simplified machine instructions for frequently used functions. The following features are typically found in RISC based systems.

1) ***Pre-fetching***: The process of fetching next instruction or instructions into an event queue before the current instruction is complete is called pre-fetching.
2) ***Pipelining***: Pipelining allows issuing an instruction prior to the completion of the currently executing one.
3) ***Superscalar operation***: Superscalar operation refers to a processor that can issue more than one instruction simultaneously.

ARCHITECTURE

The objective of the project is to design a 16-bit RISC processor. The architecture of the proposed 16-bit Processor is shown in Fig.1. The processor incorporates 16-bit ALU capable of performing 11 arithmetical and logical operations, 16-bit program counter, 24-bit Instruction register, Sixteen 16-bit general purpose registers, 3-bit flag register to indicate carry, zero and parity. The processor has four states: idle, fetch, decode and execute. The control unit provides necessary signal interaction to perform expected function in all the states. The 16-bit program counter indicates the address of memory location from which the instruction is to be fetched. After the execution of current instruction, the program counter is incremented by one unless 'JUMP' instruction is encountered. When the 'JUMP' instruction is executed the program counter is incremented or decremented by the amount indicated by the offset

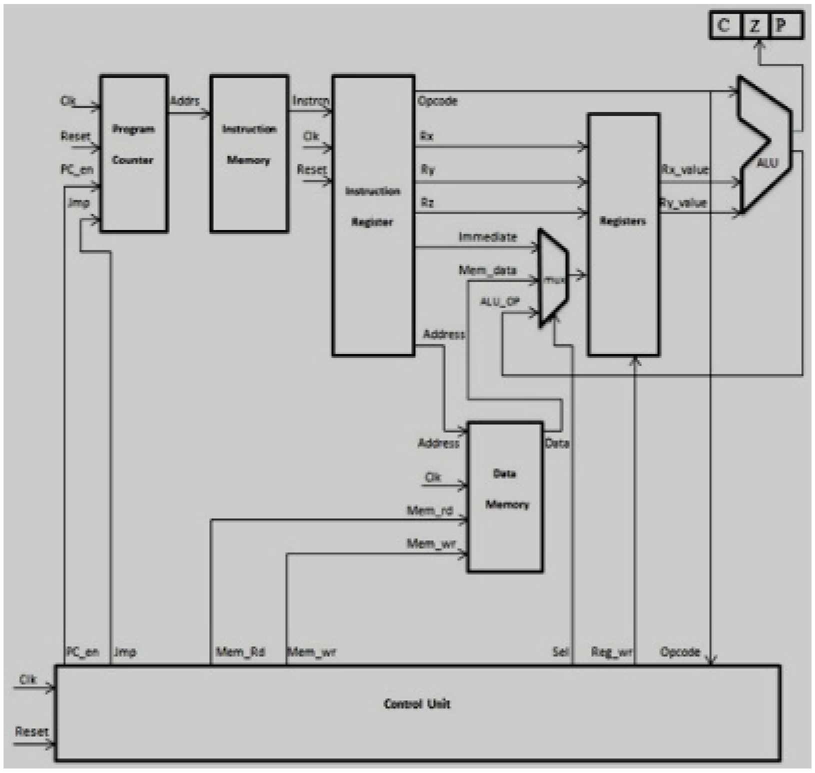


Fig.1 - Architecture

1) ***Fetch state :*** The function of the fetch unit is to obtain an instruction from the instruction memory using the current value of the program counter and increment the program counter value for the next instruction. 16-bit program counter and instruction memory together form a fetch unit.
2) ***Decode state :*** The function of the instruction decode unit is to use the 24- bit instruction provided from the previous fetch unit to index the

register file and obtain the register data values. The instruction register, control unit and registers together form a decode unit.

3) ***Execution state :*** The execution unit of the processor consists of the arithmetic logic unit (ALU) which performs the operation specified by the opcode.

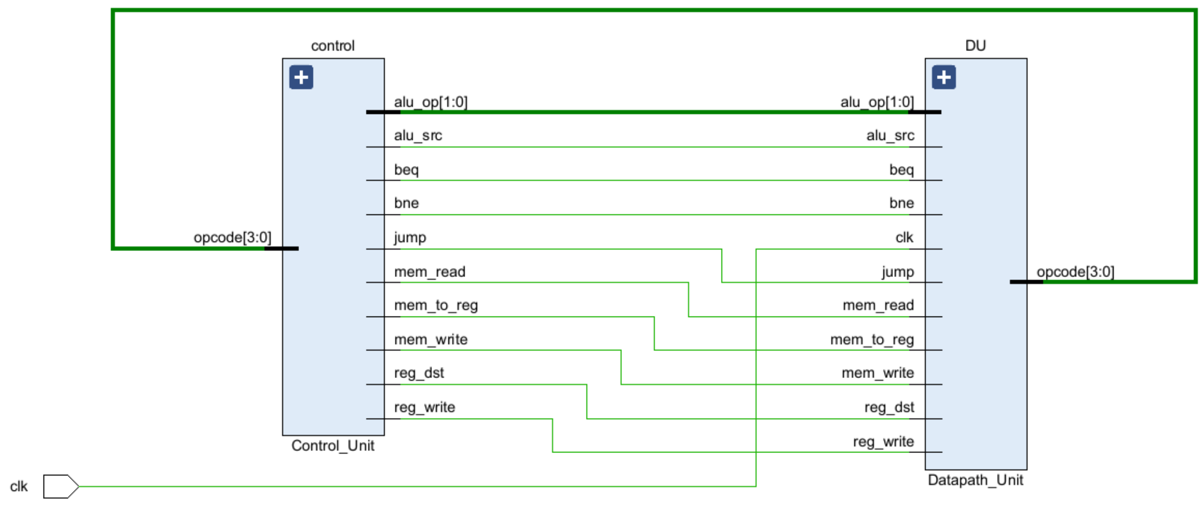


Fig.2 - RTL Schematic

INSTRUCTION SET:

OPCODE	FUNCTION	HEXADECIMAL
0	STOP	0
1	ADD	1
2	SUB	2
3	MUL	3
4	AND	4
5	OR	5
6	XOR	6
7	NOT	7
8	SLLI	8
9	SLRI	9
10	INC	A
11	DEC	B
12	MVI	C
13	LW	D
14	SW	E
15	JUMP	F

FIG.3 - OPCODES AND CORRESPONDING INSTRUCTIONS

OPERATIONS:

The instructions can be classified into the following five functional categories: data transfer operations, arithmetic operations, logical operations, branching operations and control operations.

1) ***Data Transfer Operations :*** This group of instructions copies data from a location called source to another location called destination without modifying the contents of the source. MVI, LOAD and STORE instructions come under this category.

2) ***Arithmetic Operations*** : These instructions perform arithmetic operations such as addition (ADD), subtraction (SUB), multiplication (MUL), increment (INC) and decrement (DEC).

3) ***Logical Operations :*** These instructions perform various logical operations such as AND, OR, XOR, NOT, SHL, SHR

4) ***Branching Operations*** : The instruction JUMP alters the sequence of program execution

5) ***Control Operations :*** The instruction HLT does not produce any result but stops the program execution.

CODE:

Parameter File

```
`ifndef PARAMETER_H_
`define PARAMETER_H_
`define col 16 // 16 bits instruction memory, data memory
`define row_i 15 // instruction memory, instructions number, this number can be changed. Adding more instructions to verify your design is a good idea.
`define row_d 8 // The number of data in data memory. We only use 8 data. Do not change this number. You can change the value of each data inside test.data file. Total number is fixed at 8.
`define filename "./test/50001111_50001212.o"
`define simulation_time #160
`endif
```

Instruction Memory

```
`include "Parameter.v"
module Instruction_Memory(
 input[15:0] pc,
 output[15:0] instruction
);
 reg [`col - 1:0] memory [`row_i - 1:0];
 wire [3 : 0] rom_addr = pc[4 : 1];
 initial
 begin
  $readmemb("D:/Xilinx/Projects/RISC/test/test.prog", memory,0,14);
 end
 assign instruction =  memory[rom_addr];
Endmodule
```

Register File

```
`timescale 1ns / 1ps
module GPRs(
 input       clk,
 input       reg_write_en,
 input  [2:0] reg_write_dest,
 input  [15:0] reg_write_data,
 input  [2:0] reg_read_addr_1,      //read port 1
 output  [15:0] reg_read_data_1,
 input  [2:0] reg_read_addr_2,      //read port 2
 output  [15:0] reg_read_data_2
);
 reg [15:0] reg_array [7:0];
 integer i;
 // write port
 //reg [2:0] i;
 initial begin
  for(i=0;i<8;i=i+1)
   reg_array[i] <= 16'd0;
 end
 always @ (posedge clk ) begin
   if(reg_write_en) begin
               reg_array[reg_write_dest] <= reg_write_data;
   end
 end
 assign reg_read_data_1 = reg_array[reg_read_addr_1];
 assign reg_read_data_2 = reg_array[reg_read_addr_2];
endmodule
```

ALU Control Unit

```
`timescale 1ns / 1ps
module alu_control( ALU_Cnt, ALUOp, Opcode);
 output reg[2:0] ALU_Cnt;
 input [1:0] ALUOp;
 input [3:0] Opcode;
 wire [5:0] ALUControlIn;
 assign ALUControlIn = {ALUOp,Opcode};
 always @(ALUControlIn)
 casex (ALUControlIn)
   6'b10xxxx: ALU_Cnt=3'b000;
   6'b01xxxx: ALU_Cnt=3'b001;
   6'b000010: ALU_Cnt=3'b000;
   6'b000011: ALU_Cnt=3'b001;
   6'b000100: ALU_Cnt=3'b010;
   6'b000101: ALU_Cnt=3'b011;
   6'b000110: ALU_Cnt=3'b100;
   6'b000111: ALU_Cnt=3'b101;
   6'b001000: ALU_Cnt=3'b110;
   6'b001001: ALU_Cnt=3'b111;
  default: ALU_Cnt=3'b000;
  endcase
Endmodule
```

ALU

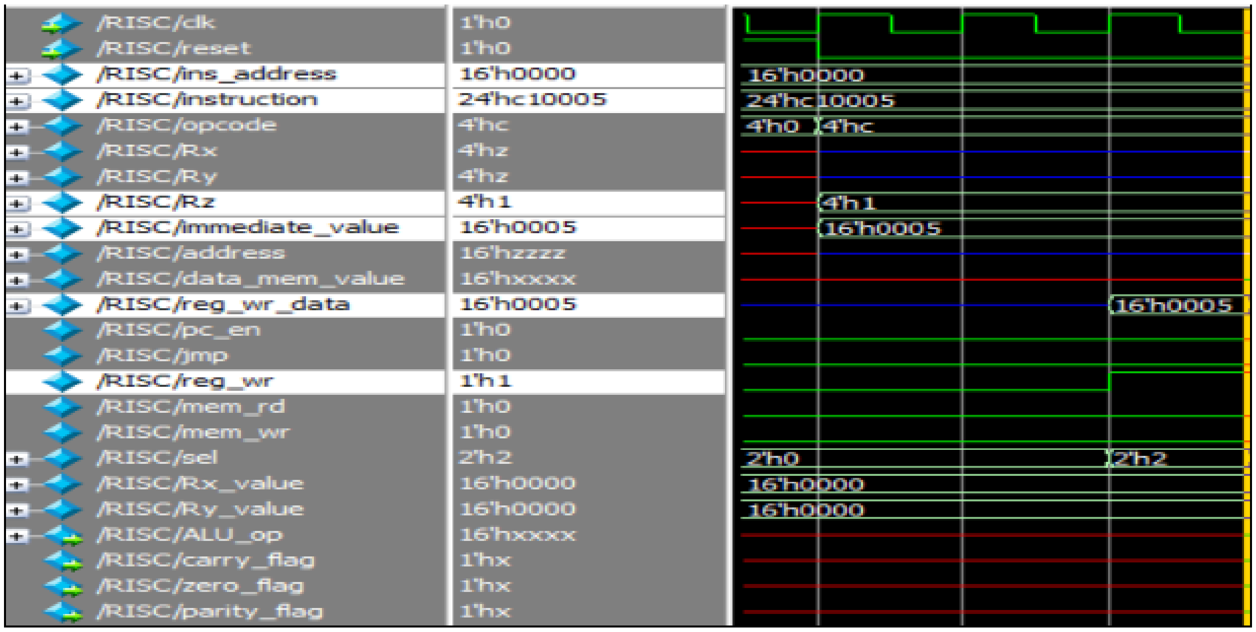
```
module ALU(
 input  [15:0] a,  //src1
 input  [15:0] b,  //src2
 input  [2:0] alu_control, //function sel
  output reg [15:0] result,  //result
 output zero
                 );
always @(*)
begin
 case(alu_control)
 3'b000: result = a + b; // add
 3'b001: result = a - b; // sub
 3'b010: result = ~a;
 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
```

Data Memory

```
`include "Parameter.v"
module Data_Memory(
 input clk,
 input [15:0]   mem_access_addr,
 input [15:0]   mem_write_data,
 input mem_write_en,
 input mem_read,
 output [15:0]   mem_read_data
);
reg [`col - 1:0] memory [`row_d - 1:0];
integer f;
wire [2:0] ram_addr=mem_access_addr[2:0];
initial
 begin
  $readmemb("D:/Xilinx/Projects/RISC/test/test.data", memory);
  f = $fopen(`filename);
  $fmonitor(f, "time = %d\n", $time,
  "\tmemory[0] = %b\n", memory[0],
  "\tmemory[1] = %b\n", memory[1],
  "\tmemory[2] = %b\n", memory[2],
  "\tmemory[3] = %b\n", memory[3],
  "\tmemory[4] = %b\n", memory[4],
  "\tmemory[5] = %b\n", memory[5],
  "\tmemory[6] = %b\n", memory[6],
  "\tmemory[7] = %b\n", memory[7]);
  `simulation_time;
  $fclose(f);
 end
 always @(posedge clk) begin
  if (mem_write_en)
   memory[ram_addr] <= mem_write_data;
 end
 assign mem_read_data = (mem_read==1'b1) ? memory[ram_addr]: 16'd0;
endmodule
```

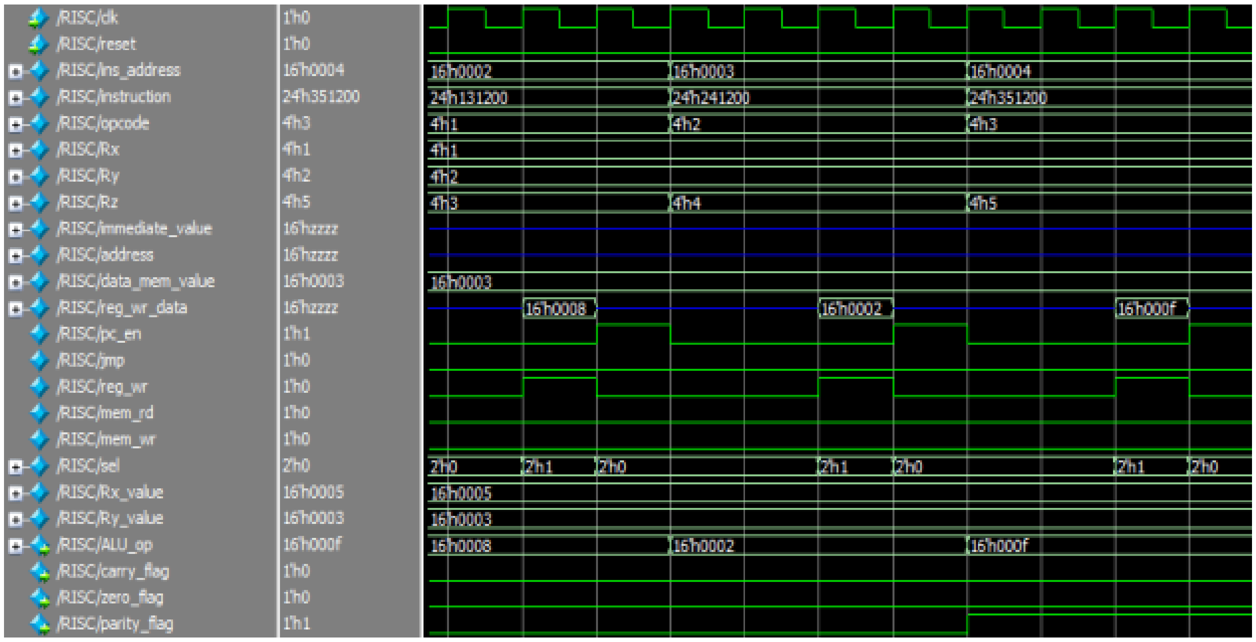
SIMULATION RESULTS:

Move Operation



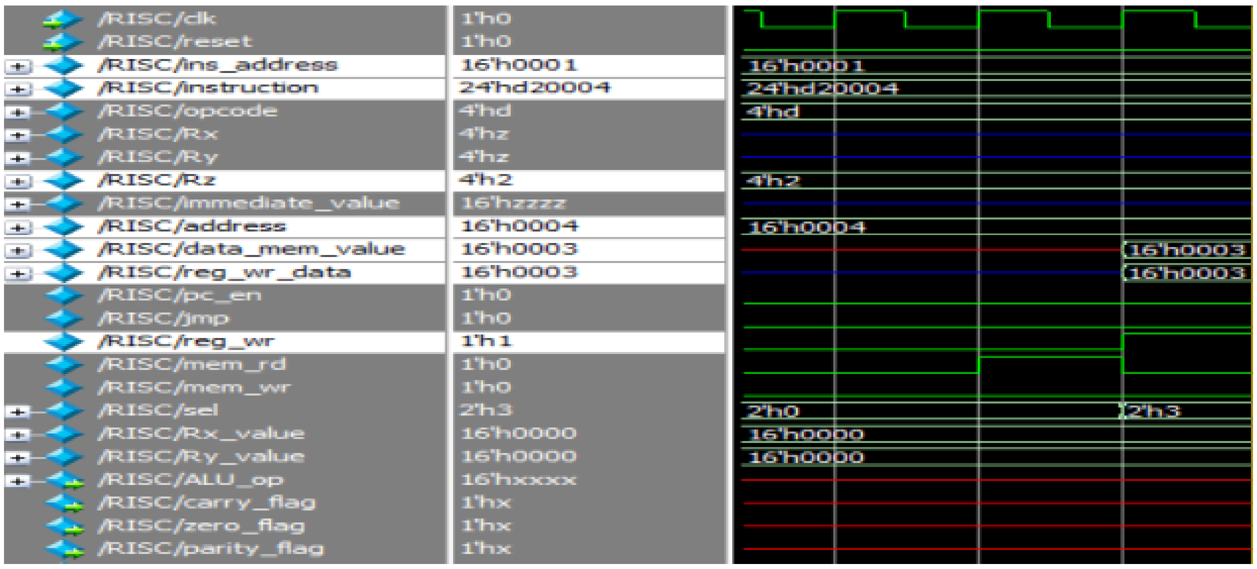
```
/RISC/clk                  1'h0
/RISC/reset                1'h0
/RISC/ins_address          16'h0000
/RISC/instruction          24'hc10005
/RISC/opcode               4'hc
/RISC/Rx                   4'hz
/RISC/Ry                   4'hz
/RISC/Rz                   4'h1
/RISC/immediate_value      16'h0005
/RISC/address              16'hzzzz
/RISC/data_mem_value       16'hxxxx
/RISC/reg_wr_data          16'h0005
/RISC/pc_en                1'h0
/RISC/jmp                  1'h0
/RISC/reg_wr               1'h1
/RISC/mem_rd               1'h0
/RISC/mem_wr               1'h0
/RISC/sel                  2'h2
/RISC/Rx_value             16'h0000
/RISC/Ry_value             16'h0000
/RISC/ALU_op               16'hxxxx
/RISC/carry_flag           1'hx
/RISC/zero_flag            1'hx
/RISC/parity_flag          1'hx
```

Arithmetic Operations



```
/RISC/clk                  1'h0
/RISC/reset                1'h0
/RISC/ins_address          16'h0004
/RISC/instruction          24'h351200
/RISC/opcode               4'h3
/RISC/Rx                   4'h1
/RISC/Ry                   4'h2
/RISC/Rz                   4'h5
/RISC/immediate_value      16'hzzzz
/RISC/address              16'hzzzz
/RISC/data_mem_value       16'h0003
/RISC/reg_wr_data          16'hzzzz
/RISC/pc_en                1'h1
/RISC/jmp                  1'h0
/RISC/reg_wr               1'h0
/RISC/mem_rd               1'h0
/RISC/mem_wr               1'h0
/RISC/sel                  2'h0
/RISC/Rx_value             16'h0005
/RISC/Ry_value             16'h0003
/RISC/ALU_op               16'h000f
/RISC/carry_flag           1'h0
/RISC/zero_flag            1'h0
/RISC/parity_flag          1'h1
```

Load Operation



```
/RISC/clk                 1'h0
/RISC/reset               1'h0
/RISC/ins_address         16'h0001
/RISC/instruction         24'hd20004
/RISC/opcode              4'hd
/RISC/Rx                  4'hz
/RISC/Ry                  4'hz
/RISC/Rz                  4'h2
/RISC/immediate_value     16'hzzzz
/RISC/address             16'h0004
/RISC/data_mem_value      16'h0003
/RISC/reg_wr_data         16'h0003
/RISC/pc_en               1'h0
/RISC/jmp                 1'h0
/RISC/reg_wr              1'h1
/RISC/mem_rd              1'h0
/RISC/mem_wr              1'h0
/RISC/sel                 2'h3
/RISC/Rx_value            16'h0000
/RISC/Ry_value            16'h0000
/RISC/ALU_op              16'hxxxx
/RISC/carry_flag          1'hx
/RISC/zero_flag           1'hx
/RISC/parity_flag         1'hx
```

CONCLUSION

A 16-Bit RISC-V processor has been implemented using verilog and the output has been demonstrated with respective waveforms.