ASSIGNMENT REPORT

Computer Architecture (3EC503CC24)

TOP – IMPLEMENTING AND SIMULATING RISCV PROCESSOR SINGLE CYCLE CORE AND CACHE

MADE BY – VARUN LUTHARIA AND ARYAN SAHAL

ROLL NUMBER – 22BEC069 AND 23BEC502

BATCH – A2

Implementing and Simulating RISCV Processor and Cache

Varun Lutharia and Aryan Sahal

Department of Electronics & Communication, Institute of Technology

Nirma University, Ahmedabad, Gujarat-382481, India

22bec069@nirmauni.ac.in and 23bec502@nirmauni.ac.in

Abstract *– RISCV processors are widely used in range of devices such as Microcontrollers and IOT, FPGA and Consumer Electronics. It comprises of several unit as such as Program Counter, Program Counter + 4(With help of Adder block),3 Muxes, Register Module, Control Unit, ALU Block, Instruction Memory and several others etc. etc. Firstly, as we know processor always uses to get execute as in fetch, decode, execute so instruction program counter will fetch thereafter it gets executed while being getting executed Program Counter will fetch the instruction present next in Instruction Memory. Some Instruction such as LW, SW are there present in Cache (as Instruction Cache) always instructions which is going to be executed will check that if it's equal to SW or LW then hit occurs, else miss occurs as miss happened instruction needed to be fetched from instruction memory there are Various module present such as PC, PC+4, Mux, Control Unit , Instruction Memory , Register Module , Immediate generator Module , ALU Control , ALU Result , Data Memory, Cache Module etc.*

Keywords – Program Counter, Program Counter + 4, Types of Instruction, Pipelining Mechanism, Immediate Generator Module Block etc.

Literature Survey – With the growing demand for open-source processor architectures, RISC-V has emerged as a widely adopted instruction set architecture (ISA). Efficient cache memory design is crucial for optimizing RISC-V processor performance, reducing memory access latency, and improving power efficiency. Unlike other processors such as 8086, ARM, RISCV is open-source with the help of which one can do as much as modification for project related work and also required instruction which user do wants, simplified decoding making hardware complexity lesser and even Power Consumption Supported by major compilers such as (Spike, Qemu , LLVM, GCC), operating systems (Linux, FreeBSD, Zephyr).

This report mainly discusses how to use Verilog HDL in such a manner such that we can Implement RISCV Processor Cache, Chip used to realize this Project is Cyclone II EP2C35F672C6.

Limitations of the currently available technology – There are few limitations for Implementing and Simulating RISCV Processor and Cache with currently available Technology**.**

1. Limited Standardization in RISC-V Cache Implementations

- Unlike ARM and x86, which have well-established cache architectures present, RISC-V does not have a single standard cache implementation.
- Different RISC-V vendors need to implement their own cache architecture, leading to compatibility issues in simulation and hardware deployment finding that part difficult.

2. FPGA Implementation Constraints

- Implementing a high-performance RISC-V cache on an FPGA is challenging due to limited on-chip memory and bandwidth along with we can’t do performance measure thoroughly we can only do miss or hit rate not like we can do benchmark performance over here only with help of Verilog Code we can’t do so.
- Cache coherence mechanisms for multi-core RISC-V processors on FPGA are still being underdeveloped.

Proposed Solution – Limitations Present can be overcome through Following Work

Cache Subsystem Abstraction Layers:

- Some projects have proposed an abstraction layer between the cache and processor, allowing modular plug-and-play cache architectures.
- This would enable different cache architectures to be integrated without significant software changes.

Hybrid Benchmarking Approaches: Since full benchmarking is difficult on FPGA Several Solutions are listed below:

- **Software-Based Profiling (gem5, Spike)** to test various cache configurations before FPGA deployment.
- **Hardware-Based Miss/Hit Rate Analysis** for verifying cache efficiency in FPGA.
- **Performance Counters Integration** (like RISC-V performance monitoring extensions) to extract meaningful performance metrics from FPGA designs.

BLOCK DIAGRAM

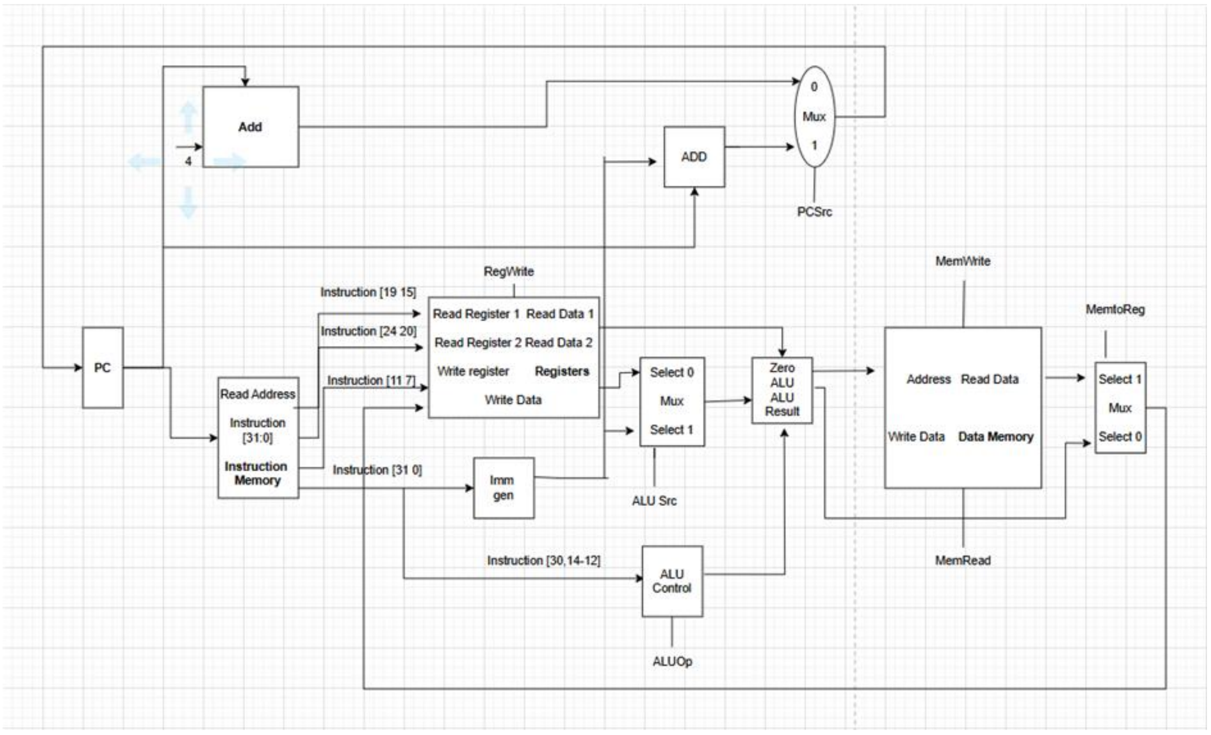


Fig. 1 Block Diagram of RISCV Processor and Cache

SIMULATION RESULTS

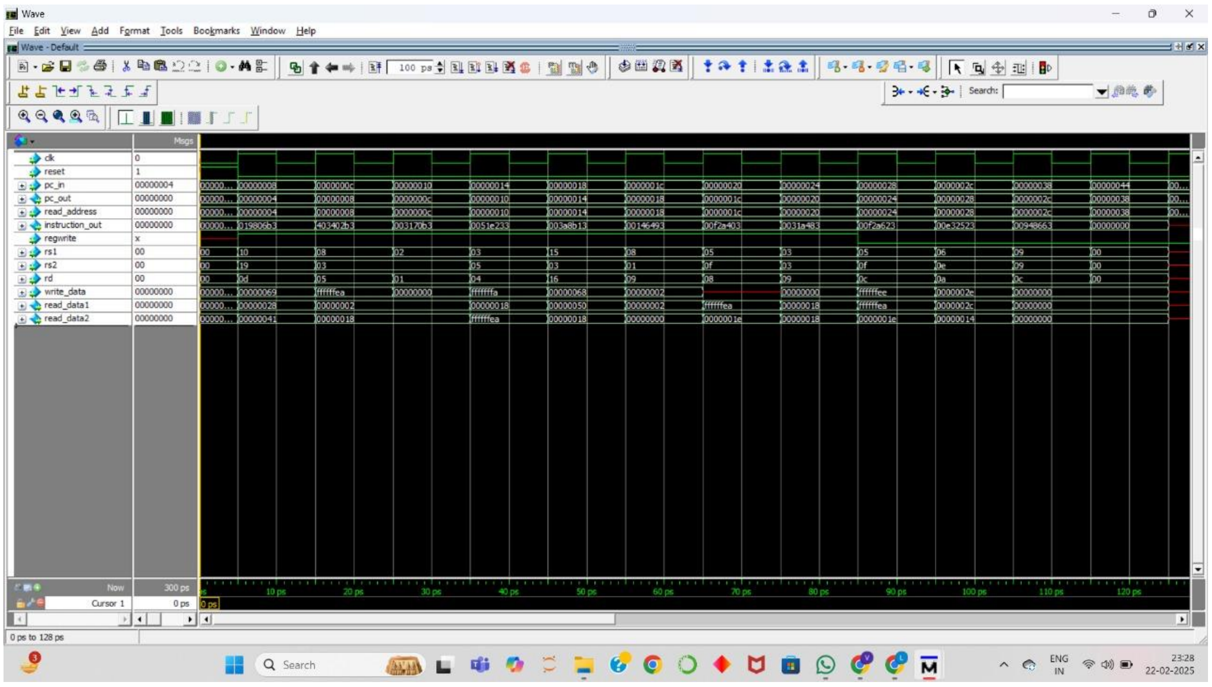
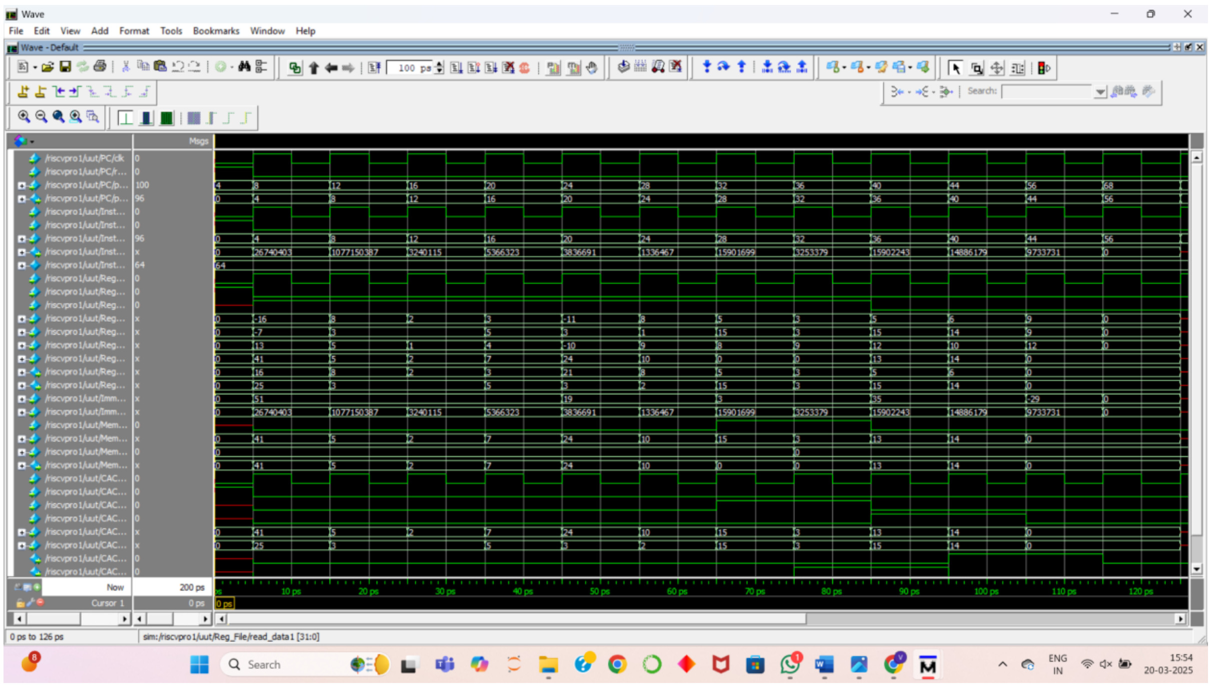


Fig. 1 RTL Simulation Waveform



Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/riscvpro1/uut/PC/clk
Now 200 ps
Cursor 1 0 ps
0 ps to 126 ps
sim:/riscvpro1/uut/Reg_File/read_data1 [31:0]

Fig.2 RTL Simulation Waveform

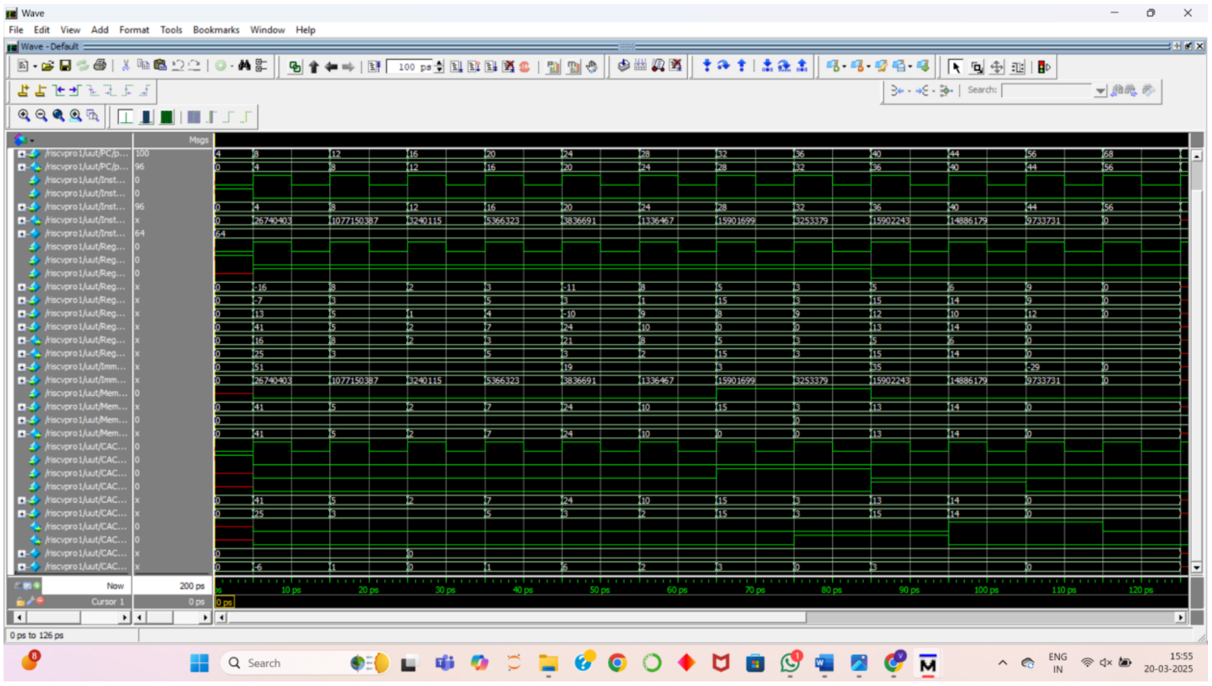


Fig.3 RTL Waveform Simulation

Analysis of the obtained results

So, Values of registers are respectively associated in the code, along with that in code itself we had constructed several instructions (12), in Instruction memory they are written step by step they will get executed with each clock cycle and obtained result aligned with expected results either of any instruction lw, sw, add, sub or any etc.

Any other relevant information

RTL Synthesis of RISCV Processor

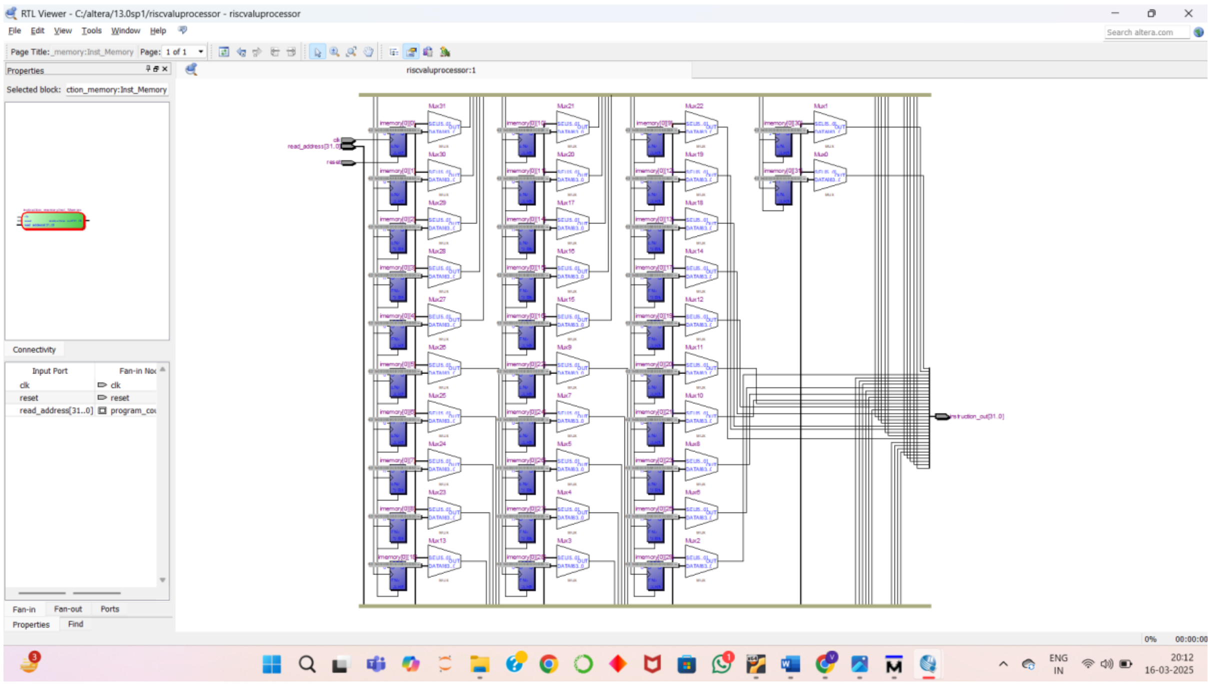

RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor
riscvaluprocessor:1
Selected block: ction_memory:Inst_Memory
Input Port
clk
reset
read_address[31..0]
instruction_out[31..0]

Fig. 3 RTL of Instruction Memory

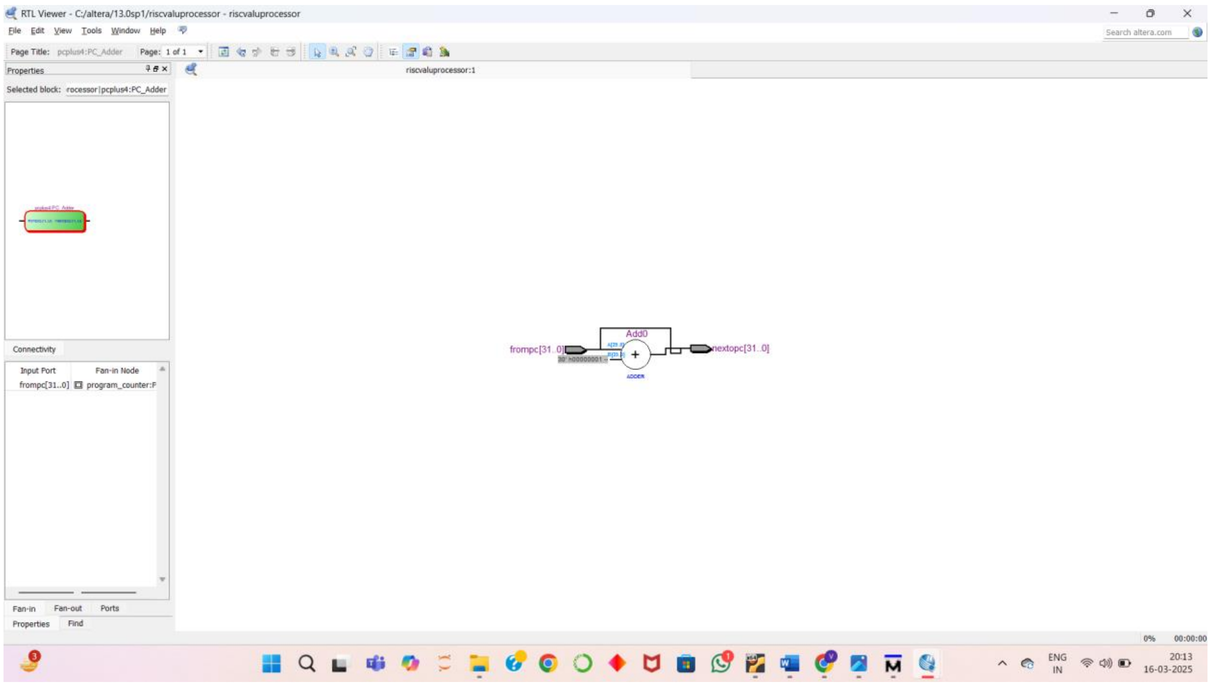


Fig. 4 RTL of PC Adder

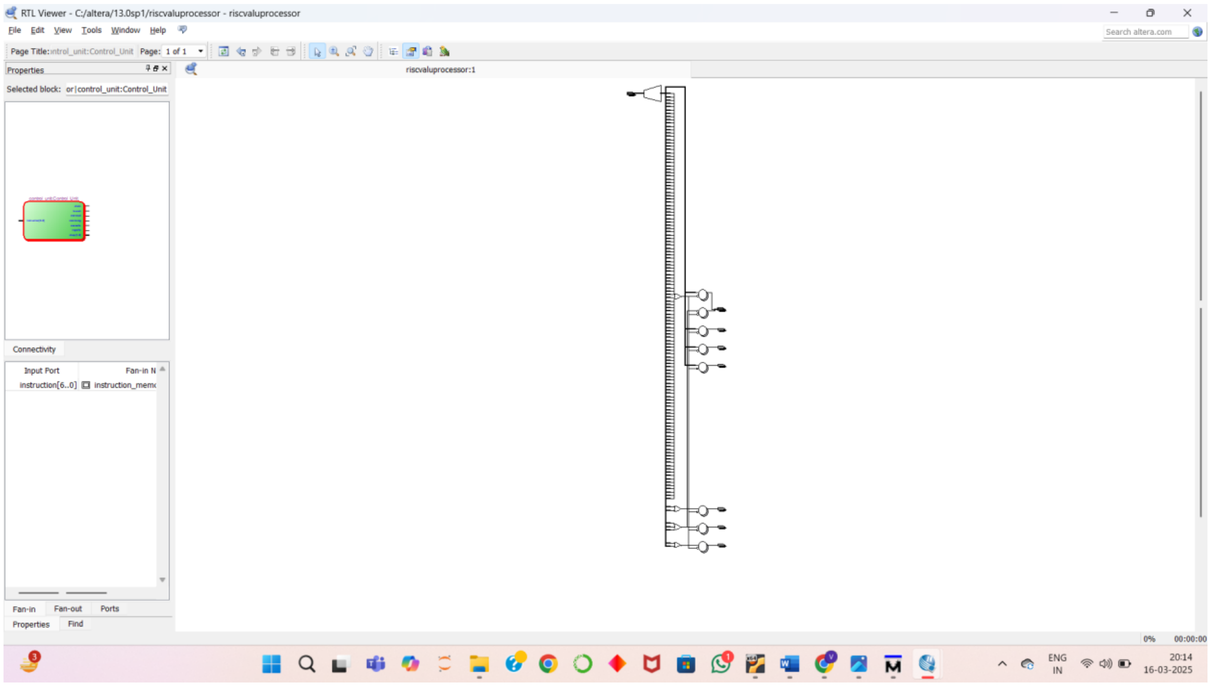
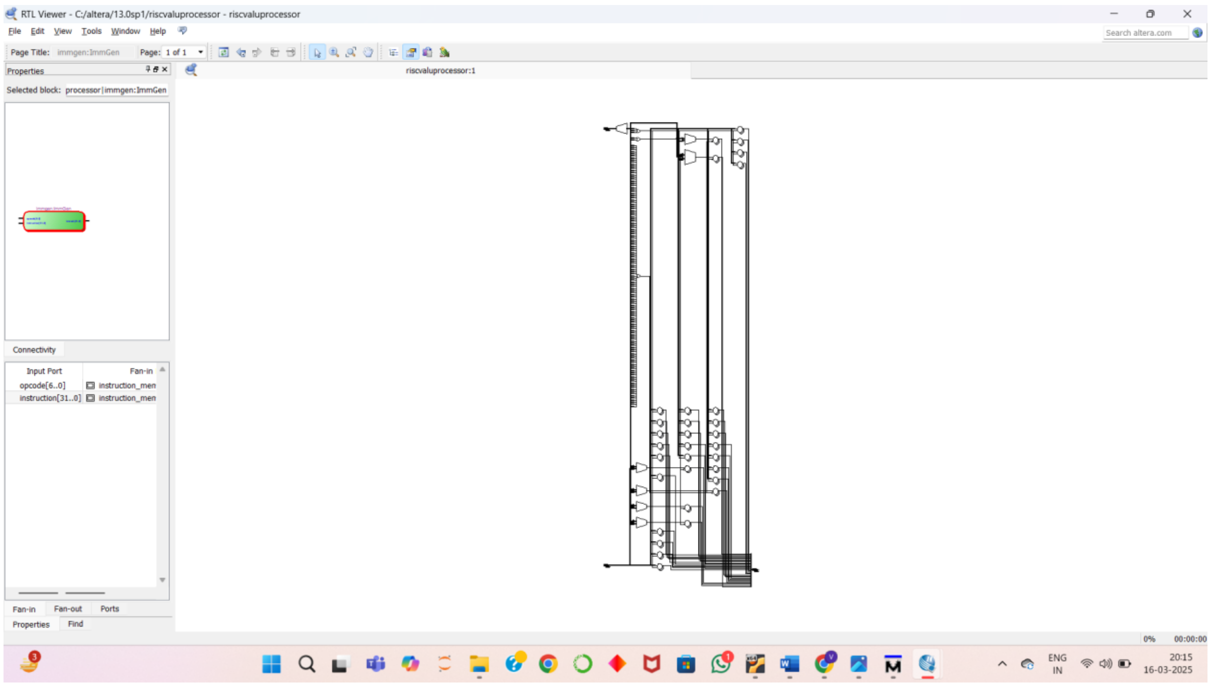
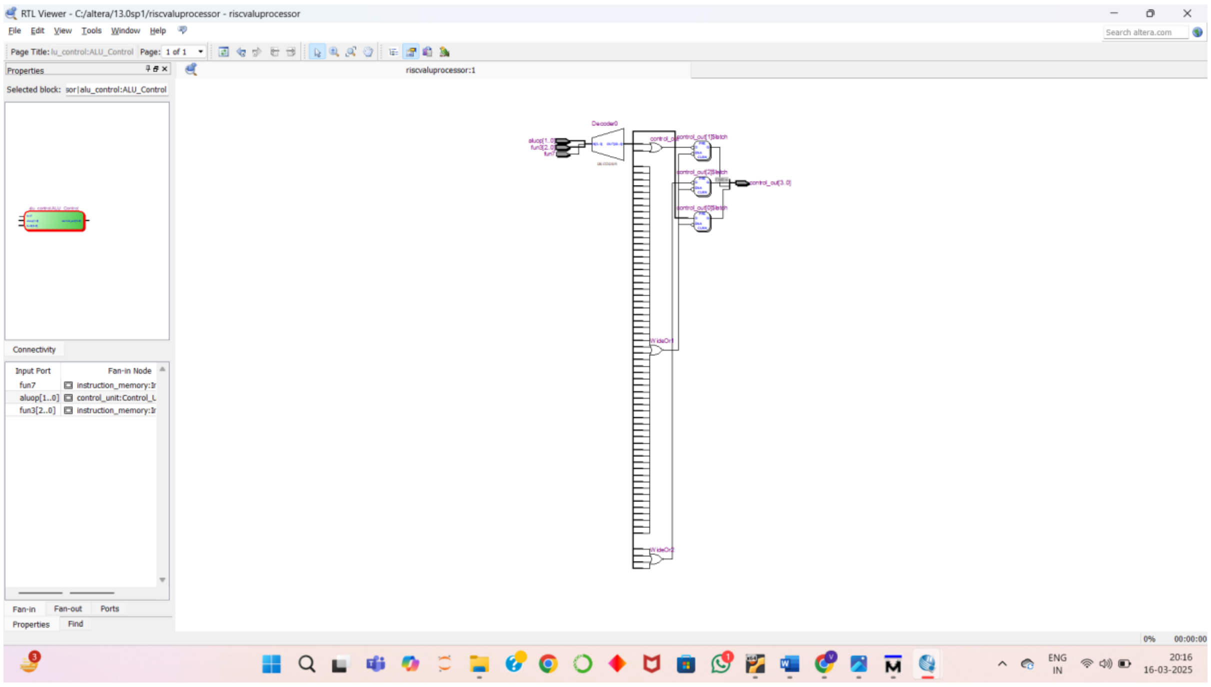

RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor
File Edit View Tools Window Help
Page Title: ntrol_unit:Control_Unit
Page: 1 of 1
Properties
Selected block: or|control_unit:Control_Unit
riscvaluprocessor:1
Connectivity
Input Port
Fan-in N
instruction[6..0]
instruction_mem
Fan-in
Fan-out
Ports
Properties
Find
16-03-2025
20:14

Fig. 5 RTL of Control Unit


RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor
File Edit View Tools Window Help
Page Title: immgen:ImmGen
Page: 1 of 1
Properties
Selected block: processor|immgen:ImmGen
riscvaluprocessor:1
Connectivity
Fan-in Fan-out Ports
Properties Find

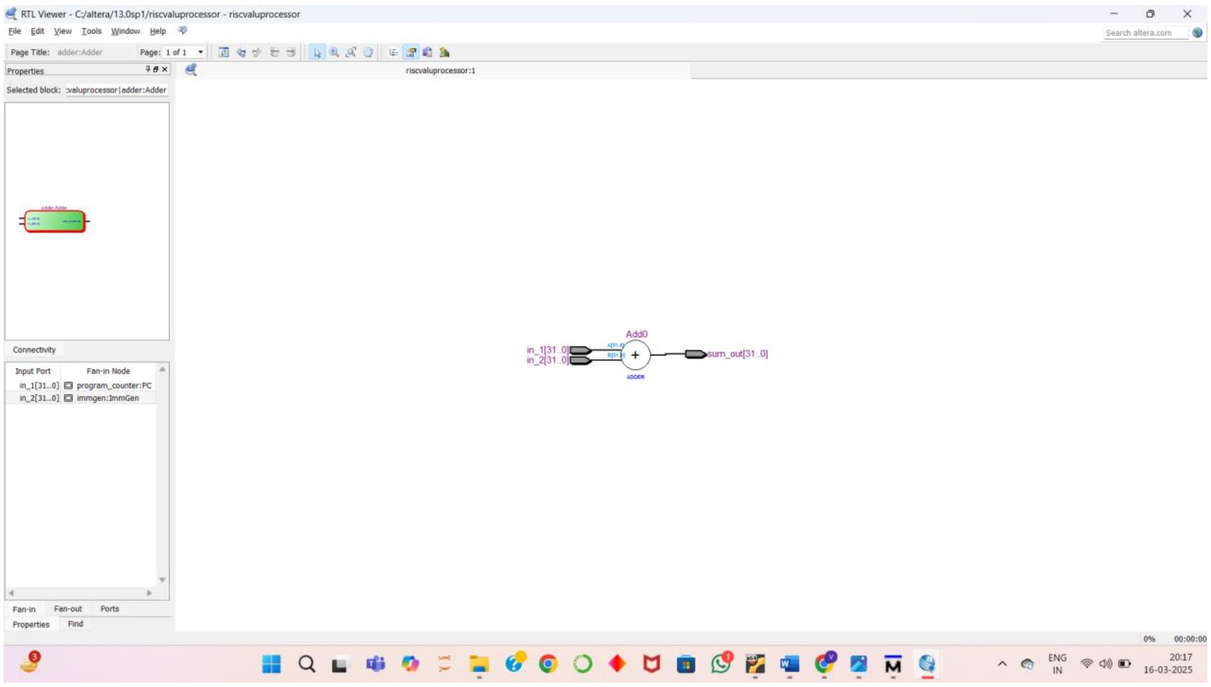
Input Port	Fan-in
opcode[6..0]	instruction_mem
instruction[31..0]	instruction_mem

Fig. 6 RTL of Instruction Memory.



Input Port	Fan-in Node
fun7	instruction_memory:I
aluop[1..0]	control_unit:Control_L
fun3[2..0]	instruction_memory:I

Fig. 7 RTL of ALU Control Unit.


RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor

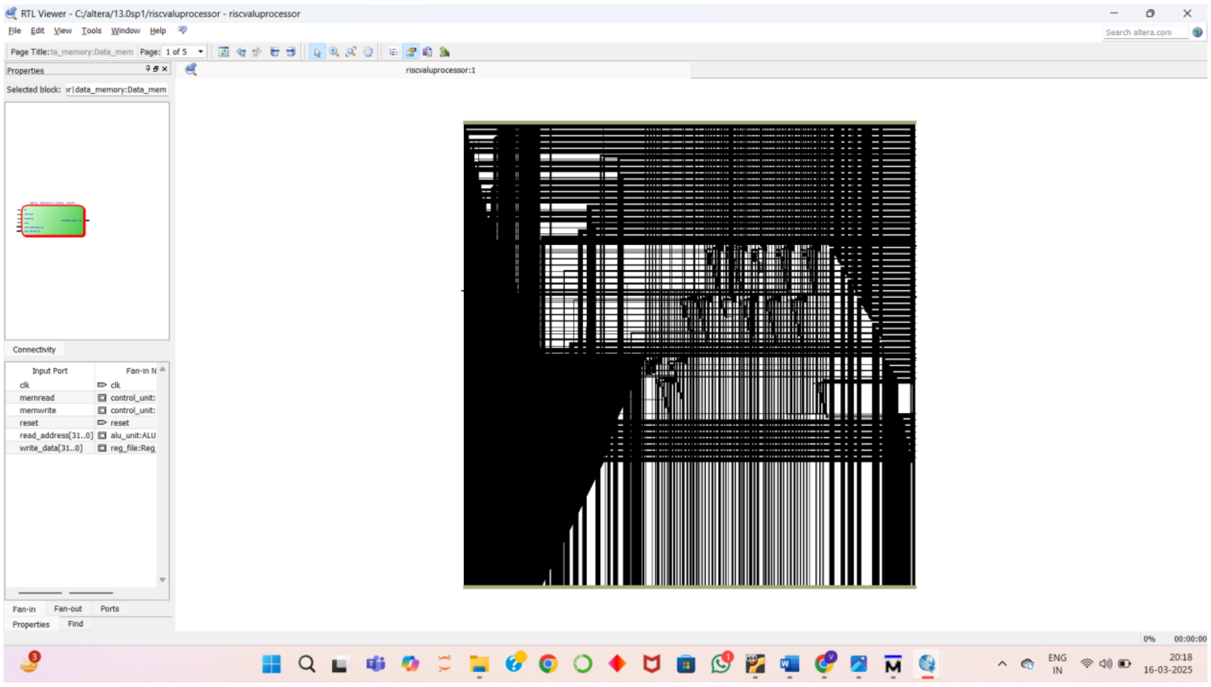
Page Title: adder:Adder Page: 1 of 1

Selected block: :valuprocessor|adder:Adder

Connectivity

Input Port	Fan-in Node
in_1[31..0]	program_counter:PC
in_2[31..0]	immgen:ImmGen

Fig. 8 RTL of Adder.


RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor

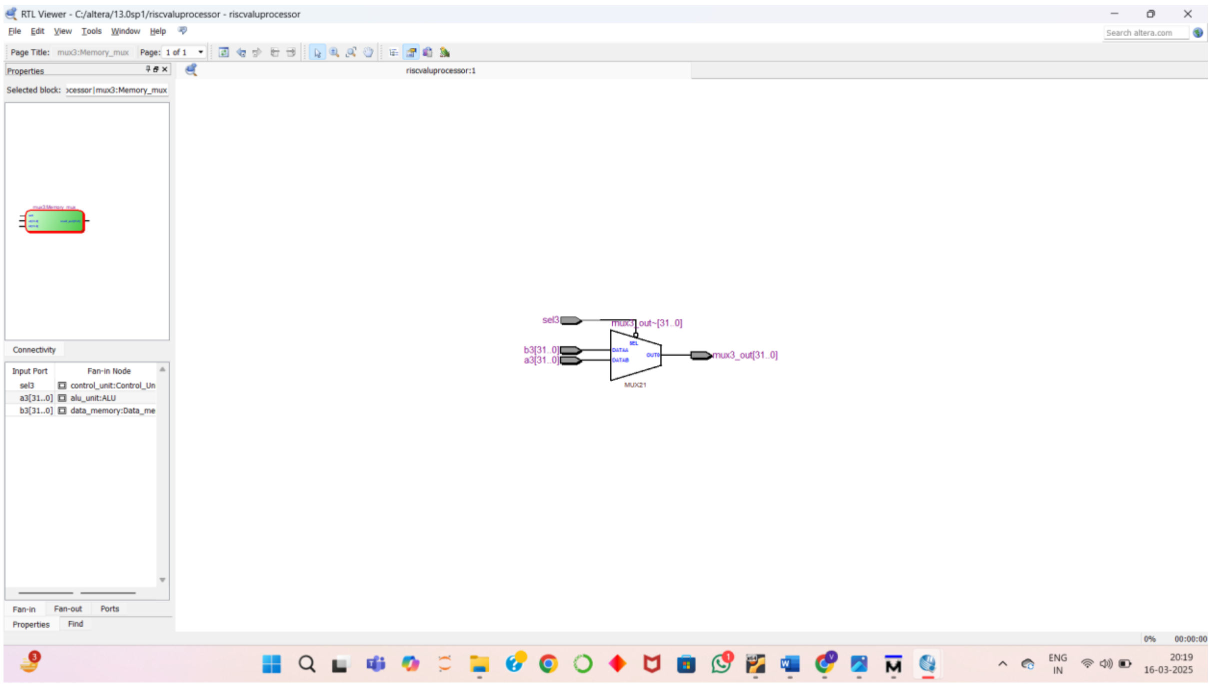
Page Title: ta_memory:Data_mem Page: 1 of 5

Selected block: ›r|data_memory:Data_mem

Connectivity

Input Port	Fan-in N
clk	clk
memread	control_unit:
memwrite	control_unit:
reset	reset
read_address[31..0]	alu_unit:ALU
write_data[31..0]	reg_file:Reg

Fig. 9 RTL of Data Memory.

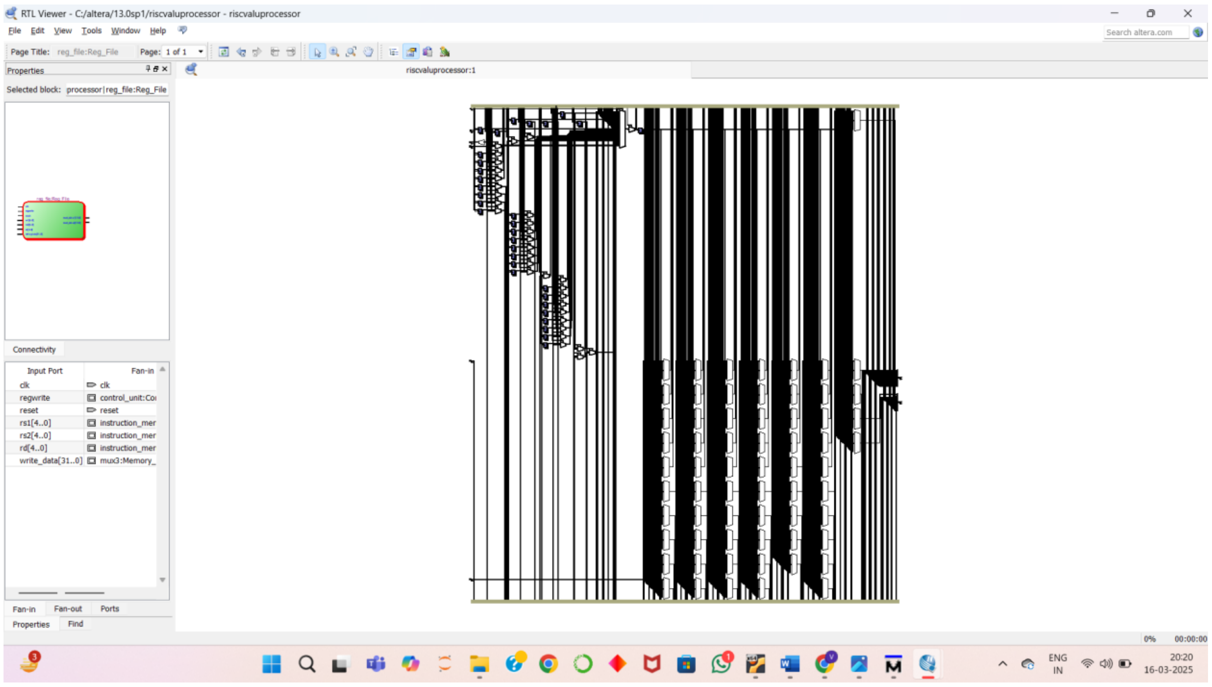


Page Title: mux3:Memory_mux Page: 1 of 1

Selected block: ›cessor|mux3:Memory_mux

Input Port	Fan-in Node
sel3	control_unit:Control_Un
a3[31..0]	alu_unit:ALU
b3[31..0]	data_memory:Data_me

Fig. 10 RTL of Memory Mux.

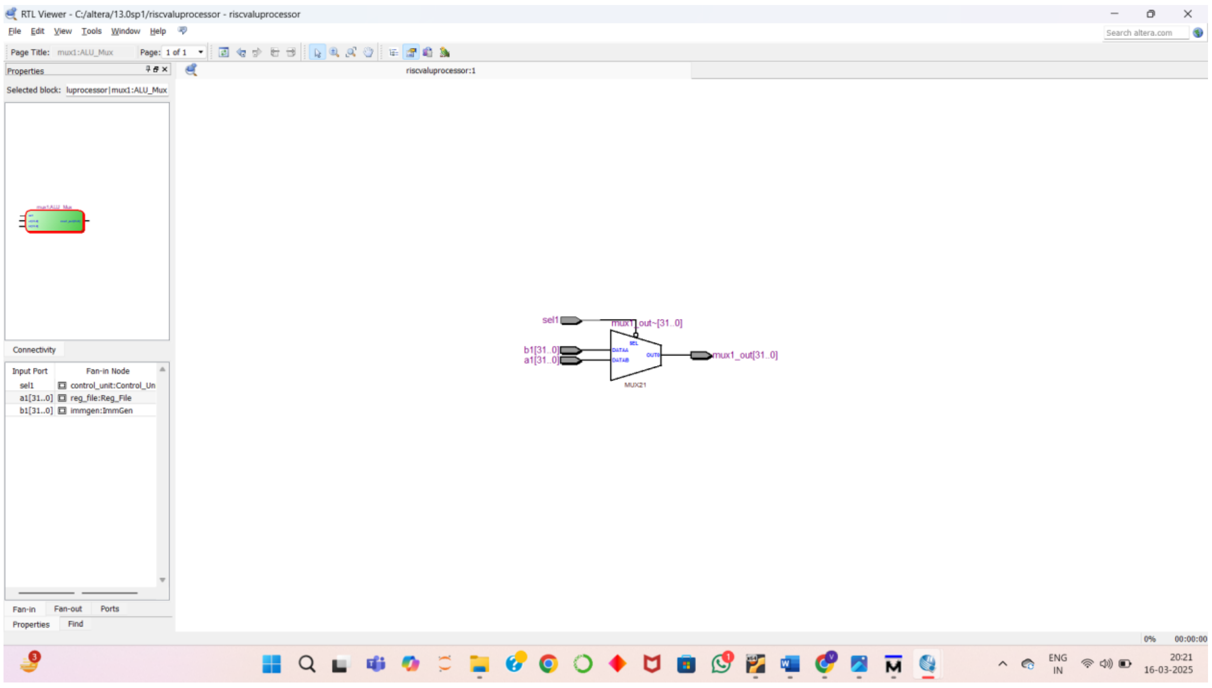


Page Title: reg_file:Reg_File Page: 1 of 1

Selected block: processor|reg_file:Reg_File

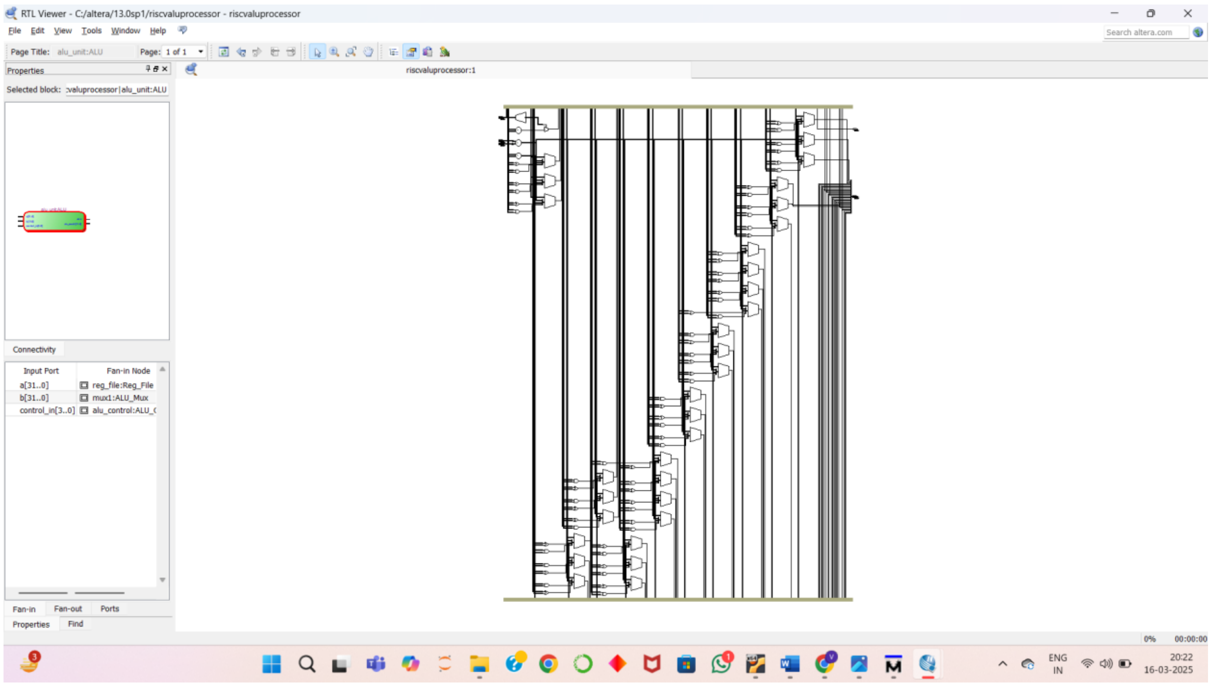
Input Port	Fan-in
clk	clk
regwrite	control_unit:Co
reset	reset
rs1[4..0]	instruction_mer
rs2[4..0]	instruction_mer
rd[4..0]	instruction_mer
write_data[31..0]	mux3:Memory_

Fig. 11 RTL of Register Memory.



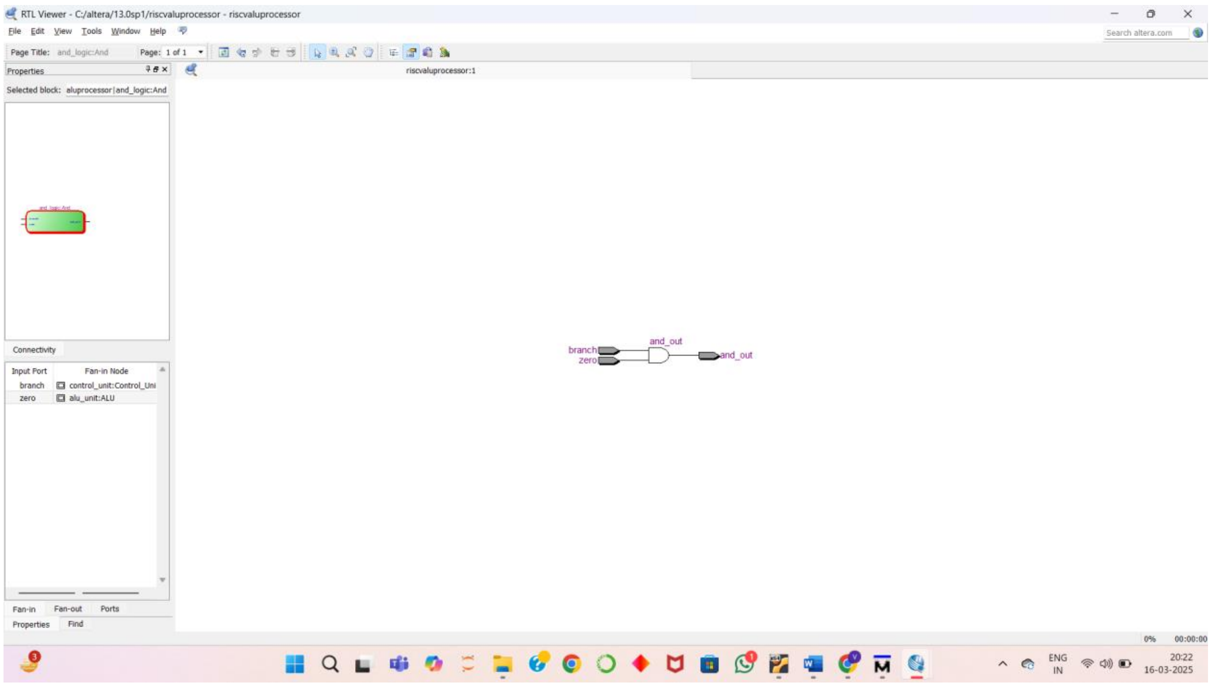
Input Port	Fan-in Node
sel1	control_unit:Control_Un
a1[31..0]	reg_file:Reg_File
b1[31..0]	immgen:ImmGen

Fig. 12 RTL of ALU Mux.


RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor
Page Title: alu_unit:ALU
Page: 1 of 1
riscvaluprocessor:1
Selected block: :valuprocessor|alu_unit:ALU
Connectivity
20:22
16-03-2025

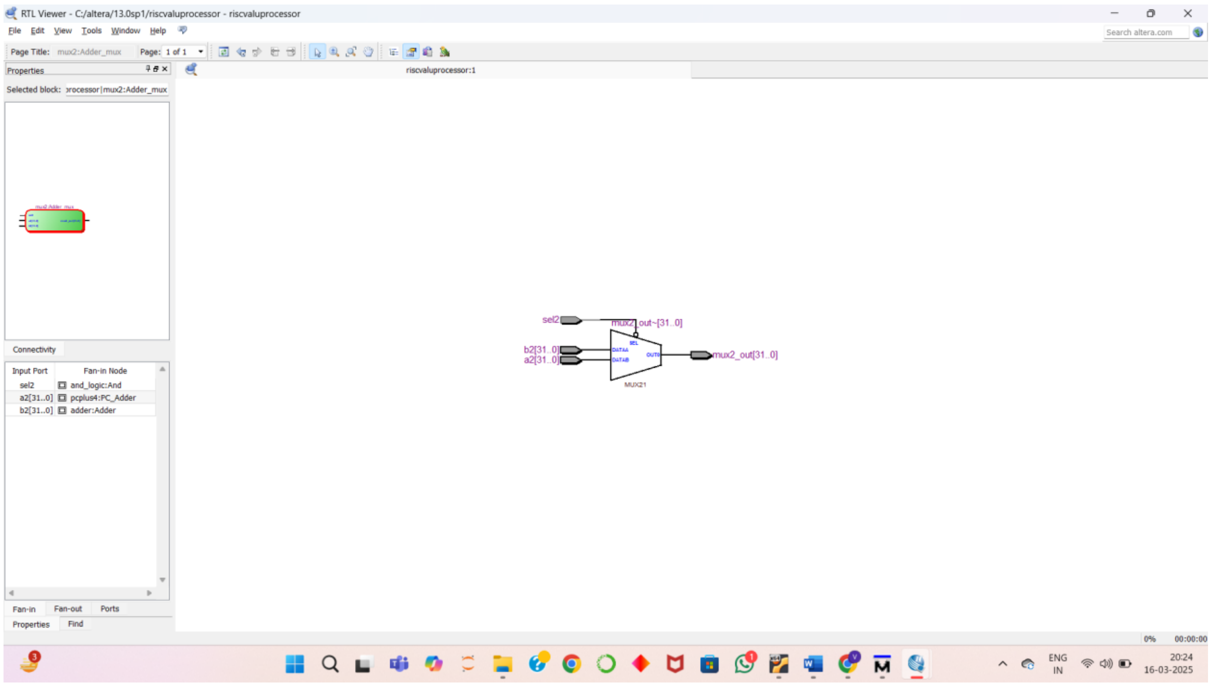
Input Port	Fan-in Node
a[31..0]	reg_file:Reg_File
b[31..0]	mux1:ALU_Mux
control_in[3..0]	alu_control:ALU_(

Fig. 13 RTL of ALU Unit.


RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor
File Edit View Tools Window Help
Page Title: and_logic:And
Page: 1 of 1
Properties
Selected block: aluprocessor|and_logic:And
riscvaluprocessor:1
Connectivity
branch
zero
and_out
and_out
Fan-in Fan-out Ports
Properties Find

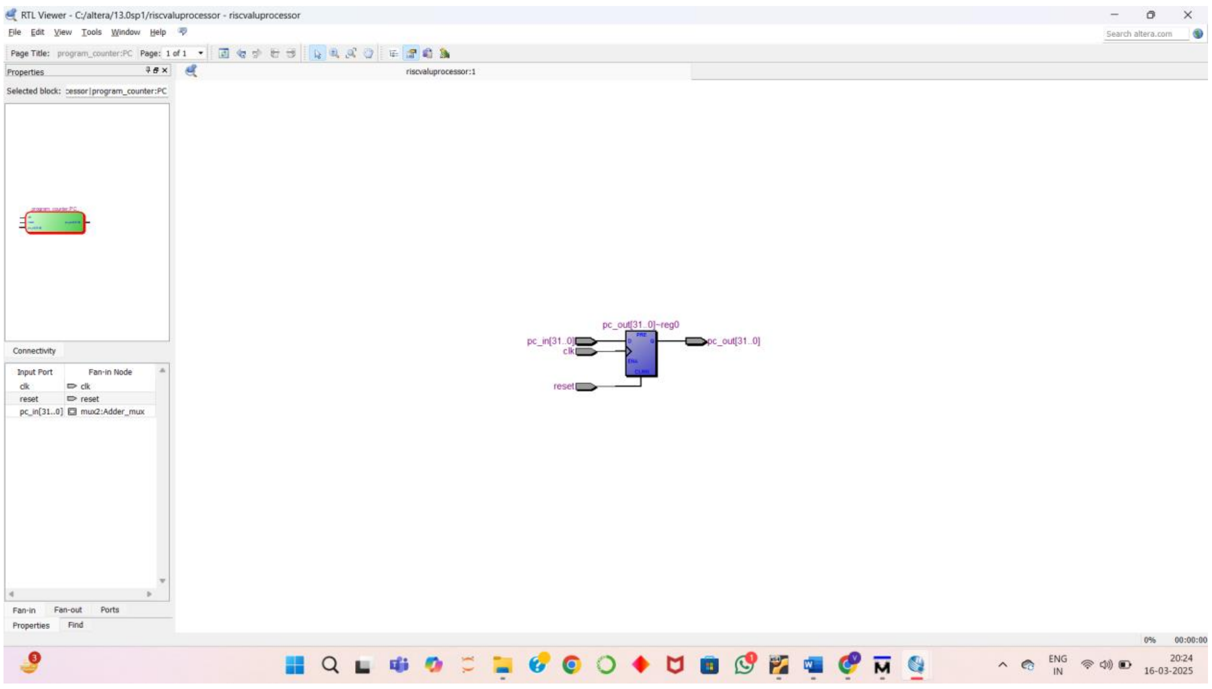
Input Port	Fan-in Node
branch	control_unit:Control_Uni
zero	alu_unit:ALU

Fig. 14 RTL Unit of AND out.


RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor
File Edit View Tools Window Help
Page Title: mux2:Adder_mux
Page: 1 of 1
Properties
Selected block: rocessor|mux2:Adder_mux
riscvaluprocessor:1
Connectivity
sel2
b2[31..0]
a2[31..0]
mux2_out~[31..0]
SEL
DATAA
DATAB
OUT0
MUX21
mux2_out[31..0]
Fan-in Fan-out Ports
Properties Find

Input Port	Fan-in Node
sel2	and_logic:And
a2[31..0]	pcplus4:PC_Adder
b2[31..0]	adder:Adder

Fig. 15 RTL Unit of ADDER Mux.

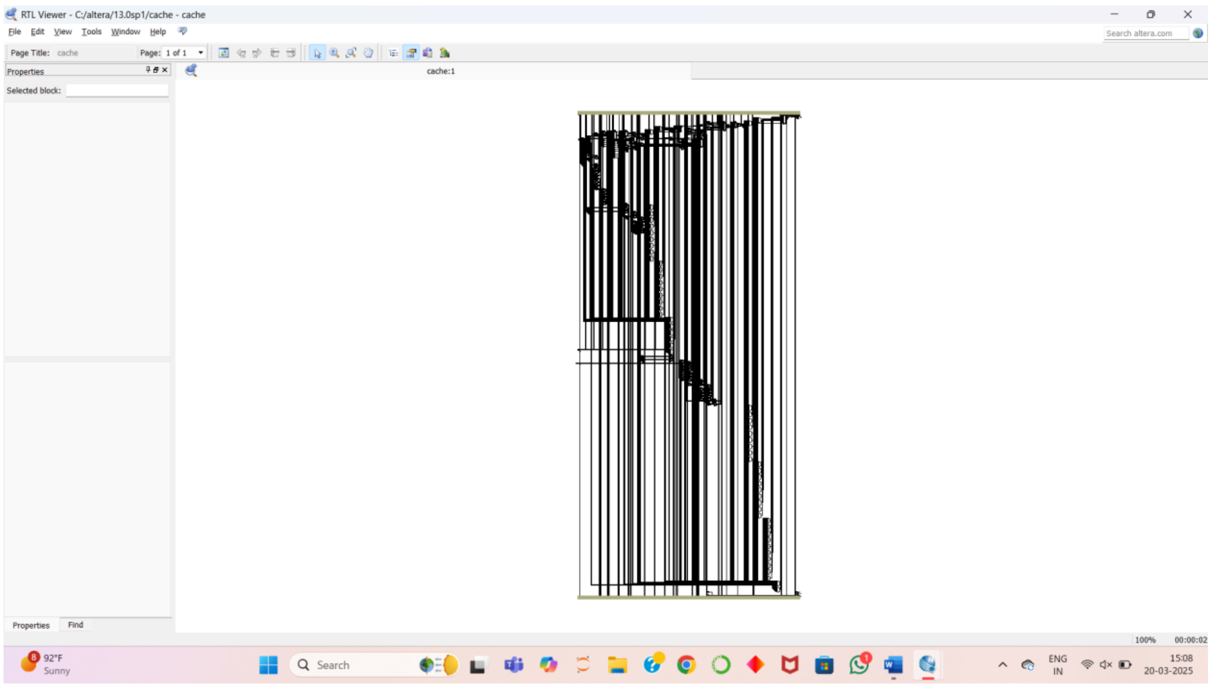


RTL Viewer - C:/altera/13.0sp1/riscvaluprocessor - riscvaluprocessor
File Edit View Tools Window Help
Page Title: program_counter:PC Page: 1 of 1
Properties
riscvaluprocessor:1
pc_out[31..0]~reg0
pc_in[31..0]
clk
reset
pc_out[31..0]
Connectivity
Fan-in Fan-out Ports
Properties Find

Selected block: :essor|program_counter:PC

Input Port	Fan-in Node
clk	clk
reset	reset
pc_in[31..0]	mux2:Adder_mux

Fig. 16 RTL Unit of Program Counter.



RTL Viewer - C:/altera/13.0sp1/cache - cache
File Edit View Tools Window Help
Page Title: cache Page: 1 of 1
Properties
Selected block:
cache:1
Properties Find

Fig.17 RTL of Cache

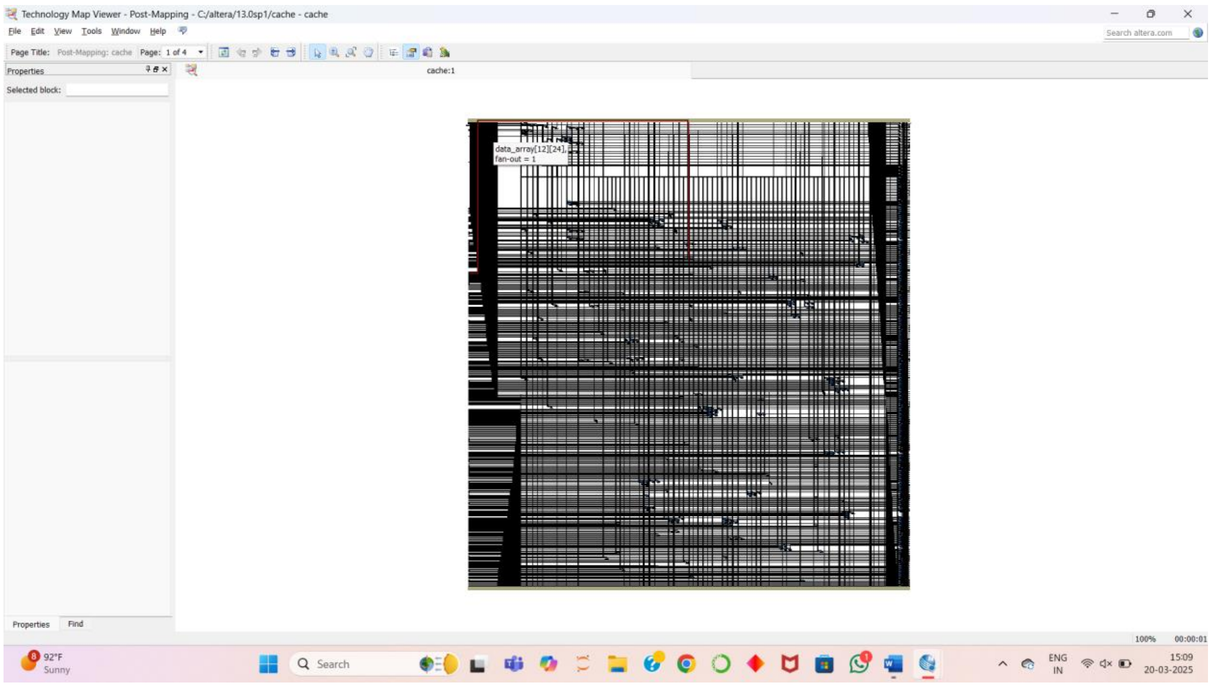

Technology Map Viewer - Post-Mapping - C:/altera/13.0sp1/cache - cache
File Edit View Tools Window Help
Page Title: Post-Mapping: cache Page: 1 of 4
Properties
Selected block:
cache:1
data_array[12][24]
fan-out = 1
Properties Find
100% 00:00:01
92°F Sunny
Search
ENG IN
15:09 20-03-2025

Fig.18 TTL of Cache

Conclusion – Here, after doing this project we got to know that how to implement, simulate RISCV Processor along with Cache Management, how to integrate RISCV Processor components with Cache (Instruction Cache) learnt also regarding Advantages and Disadvantages of RISC Processor with Cache, and how to improvise them with several techniques present.

References - 1. https://www.youtube.com/watch?v=dh88oe6O0QU&t=8823s [1],

2. (The Morgan Kaufmann Series in Computer Architecture and Design) David A. Patterson, John L. Hennessy - Computer Organization and Design RISC-V Edition_ The Hardware Software Interface-Morgan Kaufmann.pdf [2]

3. 8. Cache Memory.pdf [3]

Code in the Appendix –

```
module riscvpro1();

reg clk,reset;

riscvp_code uut(.clk(clk),.reset(reset));

initial begin

clk=0;
```

```
reset=1;
#5;
reset=0;
#400;
end

always begin
#5 clk=~clk;
end
endmodule

module riscvp_code(clk,reset);
input clk,reset;
wire
[31:0]pc_top,instruction_top,rd1_top,rd2_top,immext_top,mux1_top,sum_out_top,ne
xtopc_top,pcin_top,address_top,cache_data_out,writeback_top;
wire
regwrite_top,alusrc_top,zero_top,branch_top,sel2_top,memtoreg_top,memread_top,m
emwrite_top,hit_top,miss_top;
wire [1:0]aluop_top;
wire [3:0]control_top;

// Program Counter
program_counter PC(.clk(clk),.reset(reset),.pc_in(pcin_top),.pc_out(pc_top));

// Next to pc
pcplus4 PC_Adder(.frompc(pc_top),.nextopc(nextopc_top));

// Instruction Memory
instruction_memory
Inst_Memory(.clk(clk),.reset(reset),.read_address(pc_top),.instruction_out(instruction
_top));

// Reg file
```

```
reg_file
Reg_File(.clk(clk),.reset(reset),.regwrite(regwrite_top),.rs1(instruction_top[19:15]),.rs
2(instruction_top[24:20]),

.rd(instruction_top[11:7]),.write_data(writeback_top),.read_data1(rd1_top),.read_data
2(rd2_top));


// Imm. generator

immgen
ImmGen(.opcode(instruction_top[6:0]),.instruction(instruction_top),.immext(immext
_top));


// Control Unit

control_unit
Control_Unit(.instruction(instruction_top[6:0]),.branch(branch_top),.memread(memre
ad_top),.memtoreg(memtoreg_top),

.aluop(aluop_top),.memwrite(memwrite_top),.alusrc(alusrc_top),.regwrite(regwrite_t
op));


// Alu Control

alu_control
ALU_Control(.aluop(aluop_top),.fun7(instruction_top[30]),.fun3(instruction_top[14:
12]),.control_out(control_top));


// ALU

alu_unit
ALU(.a(rd1_top),.b(mux1_top),.control_in(control_top),.alu_result(address_top),.zero
(zero_top));


// Mux

mux1
ALU_Mux(.sel1(alusrc_top),.a1(rd2_top),.b1(immext_top),.mux1_out(mux1_top));


// Adder

adder Adder(.in_1(pc_top),.in_2(immext_top),.sum_out(sum_out_top));


// And_logic

and_logic And(.branch(branch_top),.zero(zero_top),.and_out(sel2_top));
```

```
// Mux2

mux2
Adder_mux(.sel2(sel2_top),.a2(nextopc_top),.b2(sum_out_top),.mux2_out(pcin_top))
;


// Data memory

data_memory        Data_mem(.clk(clk),.reset(reset),.memwrite(memwrite_top        &
miss_top),.memread(memread_top                                                    &
miss_top),.read_address(address_top),.write_data(rd2_top),.memdata_out(cache_data
_out));


// Mux3

mux3
Memory_mux(.sel3(memtoreg_top),.a3(address_top),.b3(cache_data_out),.mux3_out(
writeback_top));


// Cache

cache
CACHE(.clk(clk),.reset(reset),.addr(address_top),.wdata(rd2_top),.mem_read(memre
ad_top),.mem_write(memwrite_top),.hit(hit_top),.miss(miss_top));


endmodule


module program_counter(clk,reset,pc_in,pc_out);

input clk,reset;

input [31:0]pc_in;

output reg [31:0]pc_out;

always@(posedge clk or posedge reset)

begin

if(reset)

  pc_out<=32'b00;

else

  pc_out<=pc_in;

end
```

```
endmodule

module pcplus4(frompc,nextopc);
input [31:0]frompc;
output [31:0]nextopc;
assign nextopc=4+frompc;
endmodule

module instruction_memory(clk,reset,read_address,instruction_out);
input clk,reset;
input [31:0]read_address;
output [31:0]instruction_out;
reg [31:0]imemory[63:0];
integer k;

assign instruction_out=imemory[read_address];

always@(posedge clk or posedge reset)
begin
if(reset)
begin
  for(k=0;k<64;k=k+1)
  begin
  imemory[k]=32'b00;
  end
end
else
  // R type
  imemory[0]=32'h00000000;
  imemory[4]=32'b0000000_11001_10000_000_01101_0110011;
  imemory[8]=32'b0100000_00011_01000_000_00101_0110011;
  imemory[12]=32'b0000000_00011_00010_111_00001_0110011;
  imemory[16]=32'b0000000_00101_00011_110_00100_0110011;

  // I Type
  imemory[20]=32'b000000000011_10101_000_10110_0010011;
  imemory[24]=32'b000000000001_01000_110_01001_0010011;

  // L type
  imemory[28]=32'b000000001111_00101_010_01000_0000011;
  imemory[32]=32'b000000000011_00011_010_01001_0000011;

  // S type
  imemory[36]=32'b0000000_01111_00101_010_01100_0100011;
  imemory[40]=32'b0000000_01110_00110_010_01010_0100011;

  // SB type
  imemory[44]=32'h00948663;

end
endmodule

// Cache Management
module cache(clk,reset,addr,wdata,mem_read,mem_write,rdata,hit,miss);
input clk,reset,mem_read,mem_write;
input [31:0]addr,wdata;
output reg [31:0]rdata;
output reg hit, miss; // 'hit' and 'miss' should be declared as 'reg' because they are assigned inside 'always' block

parameter cache_size=16;
parameter index_bits=4;
parameter tag_bits=32-index_bits-2;
```

```
reg [tag_bits-1:0]tag_array[0:cache_size-1];
reg [31:0]data_array[0:cache_size-1];
reg valid[0:cache_size-1];

wire [tag_bits-1:0]tag=addr[31:index_bits+2];
wire [index_bits-1:0]index=addr[index_bits+1:2];

integer i;
always@(posedge clk)
begin
  if(reset)
  begin
    for(i=0;i<cache_size;i=i+1)
    begin
      valid[i]<=0;
      tag_array[i]<=0;
      data_array[i]<=0;
    end
  end

  else if(mem_read || mem_write)
  begin
    if(valid[index] && tag_array[index] == tag)
    begin
      hit<=1'b1;
      miss<=1'b0;
    end
    else
    begin
      hit<=1'b0;
      miss<=1'b1;
      tag_array[index]<=tag; // Fetch from memory in case of miss "DEADBEEF"
      data_array[index]<=32'hDEADBEEF;
      valid[index]<=1;
    end

    if(mem_read)
    begin
      rdata<=data_array[index];
    end

    if(mem_write)
    begin
      data_array[index]<=wdata;
    end
  end

  else
  begin
    hit<=1'b0;
    miss<=1'b0;
  end
end
endmodule

// Register File
module reg_file(clk,reset,regwrite,rs1,rs2,rd,write_data,read_data1,read_data2);
input clk,reset,regwrite;
input [4:0]rs1,rs2,rd;
input [31:0]write_data;
integer k;
reg [31:0]registers[31:0];
output [31:0]read_data1,read_data2;
```

```
initial begin
  registers[0]=0;
  registers[1]=4;
  registers[2]=2;
  registers[3]=24;
  registers[4]=4;
  registers[5]=1;
  registers[6]=44;
  registers[7]=4;
  registers[8]=2;
  registers[9]=1;
  registers[10]=23;
  registers[11]=4;
  registers[12]=90;
  registers[13]=10;
  registers[14]=20;
  registers[15]=30;
  registers[16]=40;
  registers[17]=50;
  registers[18]=60;
  registers[19]=70;
  registers[20]=80;
  registers[21]=80;
  registers[22]=90;
  registers[23]=70;
  registers[24]=60;
  registers[25]=65;
  registers[26]=4;
  registers[27]=32;
  registers[28]=12;
  registers[29]=34;
  registers[30]=5;
  registers[31]=10;
end



always@(posedge clk)
begin
if(reset)
begin
  for(k=0;k<32;k=k+1)
  begin
  registers[k]<=32'b00;
  end
end
else if(regwrite)
  begin
  registers[rd]<=write_data;
  end
end

assign read_data1=registers[rs1];
assign read_data2=registers[rs2];
endmodule

module immgen(opcode,instruction,immext);
input [6:0]opcode;
input [31:0]instruction;  // Fixed spelling mistake from "instuction" to "instruction"
output reg [31:0]immext;
always@(*)
begin
  case(opcode)
  7'b0000011:immext<={{20{instruction[31]}},instruction[31:20]}; // lw
  7'b0100011:immext<={{20{instruction[31]}},instruction[31:25],instruction[11:7]};
//sw

7'b1100011:immext<={{19{instruction[31]}},instruction[31],instruction[30:25],instru
ction[11:8],1'b0}; // sb
  endcase
end
endmodule


module
control_unit(instruction,branch,memread,memtoreg,aluop,memwrite,alusrc,regwrite);
input [6:0]instruction;
output reg branch,memread,memtoreg,memwrite,alusrc,regwrite;
output reg [1:0]aluop;
always@(*)
begin
  case(instruction)
  7'b0110011:     {alusrc,memtoreg,regwrite,memread,memwrite,branch,aluop}     <=
8'b001000_10;
  7'b0000011:     {alusrc,memtoreg,regwrite,memread,memwrite,branch,aluop}     <=
8'b111100_00;
  7'b0100011:     {alusrc,memtoreg,regwrite,memread,memwrite,branch,aluop}     <=
8'b100010_00;
  7'b1100011:     {alusrc,memtoreg,regwrite,memread,memwrite,branch,aluop}     <=
8'b000001_01;
  endcase
end
endmodule


module alu_unit(a,b,control_in,alu_result,zero);
input [31:0]a,b;
input [3:0]control_in;
output reg zero;
output reg [31:0]alu_result;
always@(control_in or a or b)
```

```
begin
  case(control_in)
  4'b0000: begin zero<=0;alu_result<=a&b;end
  4'b0001: begin zero<=0;alu_result<=a|b;end
  4'b0010: begin zero<=0;alu_result<=a+b;end
  4'b0110: begin if(a==b) zero<=1; else zero<=0; alu_result<=a-b;end
  default: begin zero<=1'b0; alu_result<=32'b00;end
  endcase
end
endmodule

module alu_control(aluop,fun7,fun3,control_out);
input fun7;
input [2:0]fun3;
input [1:0]aluop;
output reg [3:0]control_out;
always@(*)
begin
  case({aluop,fun7,fun3})
  6'b00_0_000: control_out<=4'b0010;
  6'b01_0_000: control_out<=4'b0110;
  6'b10_0_000: control_out<=4'b0010;
  6'b10_1_000: control_out<=4'b0110;
  6'b10_0_111: control_out<=4'b0000;
  6'b10_0_110: control_out<=4'b0001;
  endcase
end
endmodule

module
data_memory(clk,reset,memwrite,memread,read_address,write_data,memdata_out);
input clk,reset,memwrite,memread;
input [31:0]read_address,write_data;
output [31:0]memdata_out;
integer k;
reg [31:0]d_memory[63:0];
always@(posedge clk or posedge reset)
begin
if(reset)
begin
  for(k=0;k<64;k=k+1)
  begin
  d_memory[k]<=32'b00;
  end
end
  else if(memwrite)
  begin
  d_memory[read_address]<=write_data;
  end
end
assign memdata_out=(memread)?d_memory[read_address]:32'b00;
endmodule

module mux1(sel1,a1,b1,mux1_out);
input sel1;
input [31:0]a1,b1;
output [31:0]mux1_out;
assign mux1_out=(sel1==1'b0)?a1:b1;
endmodule

module mux2(sel2,a2,b2,mux2_out);
input sel2;
input [31:0]a2,b2;
output [31:0]mux2_out;
```

```
assign mux2_out=(sel2==1'b0)?a2:b2;
endmodule

module mux3(sel3,a3,b3,mux3_out);
input sel3;
input [31:0]a3,b3;
output [31:0]mux3_out;
assign mux3_out=(sel3==1'b0)?a3:b3;
endmodule


module and_logic(branch,zero,and_out);
input branch,zero;
output and_out;
assign and_out=branch&zero;
endmodule

module adder(in_1,in_2,sum_out);
input [31:0]in_1,in_2;
output [31:0]sum_out;
assign sum_out=in_1+in_2;
endmodule
```