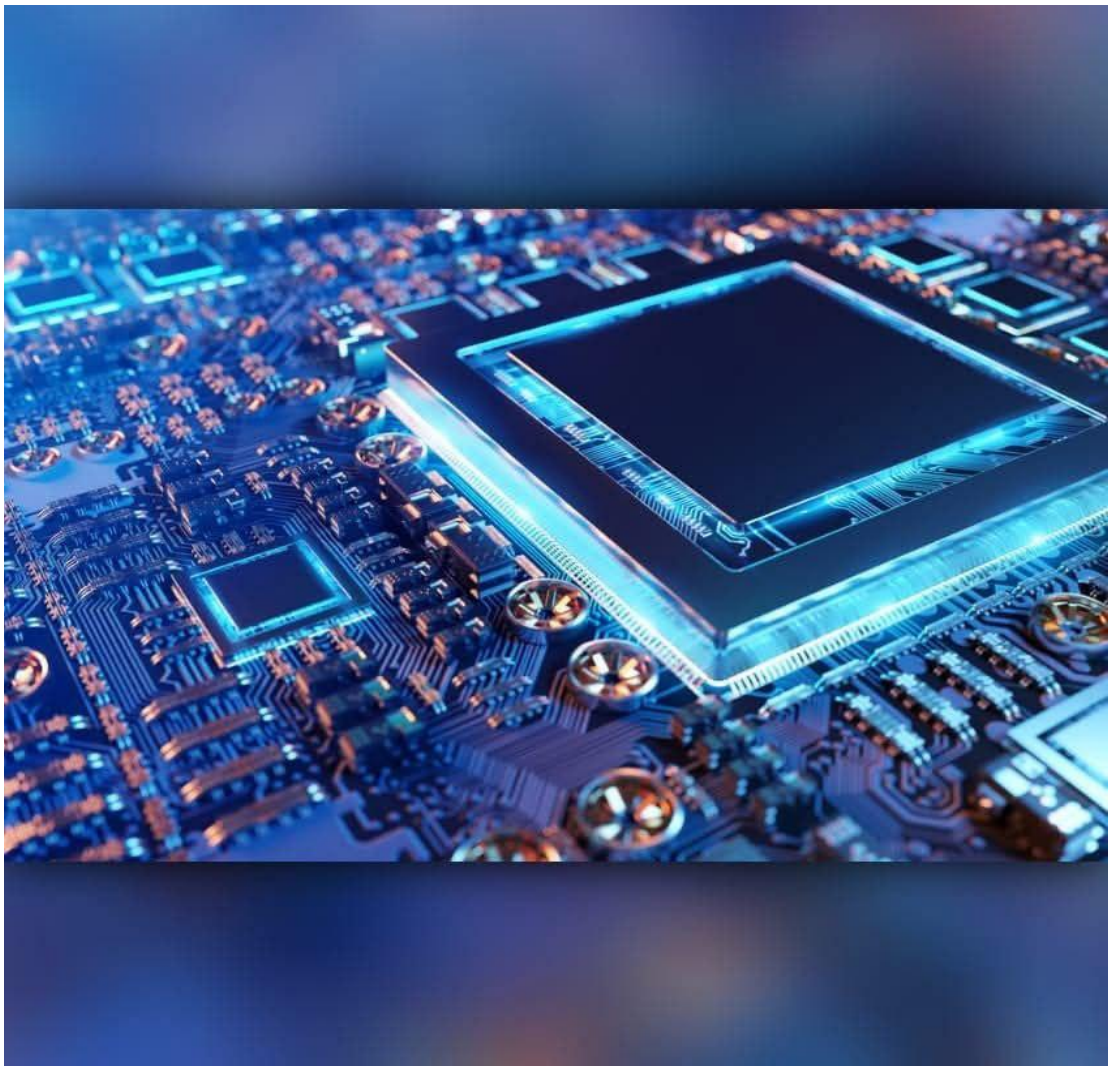
Digital IC Design

Single Cycle RISC-V Processor

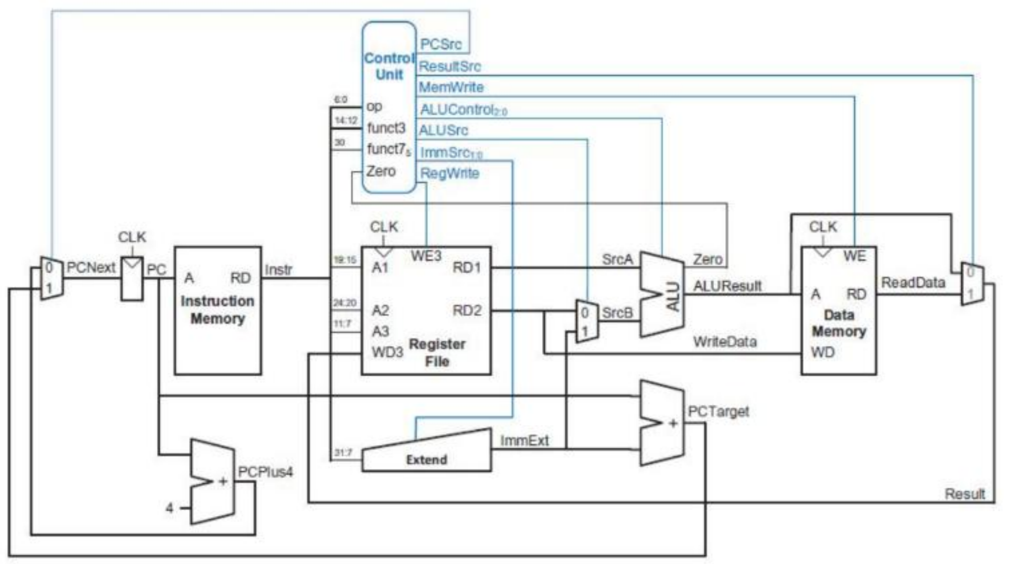


Prepared by: Omar Ahmed Abelaty

Introduction

This Project is an implementation to a 32-bit single-cycle micro architecture RISC-V processor based on Harvard Architecture. The single-cycle microarchitecture executes an entire instruction in one cycle. In other words, instruction fetch, instruction decode, execute, write back, and program counter update occurs within a single clock cycle.

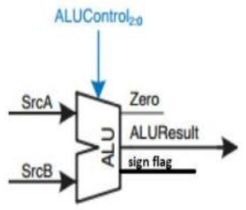
Design Architecture Overview



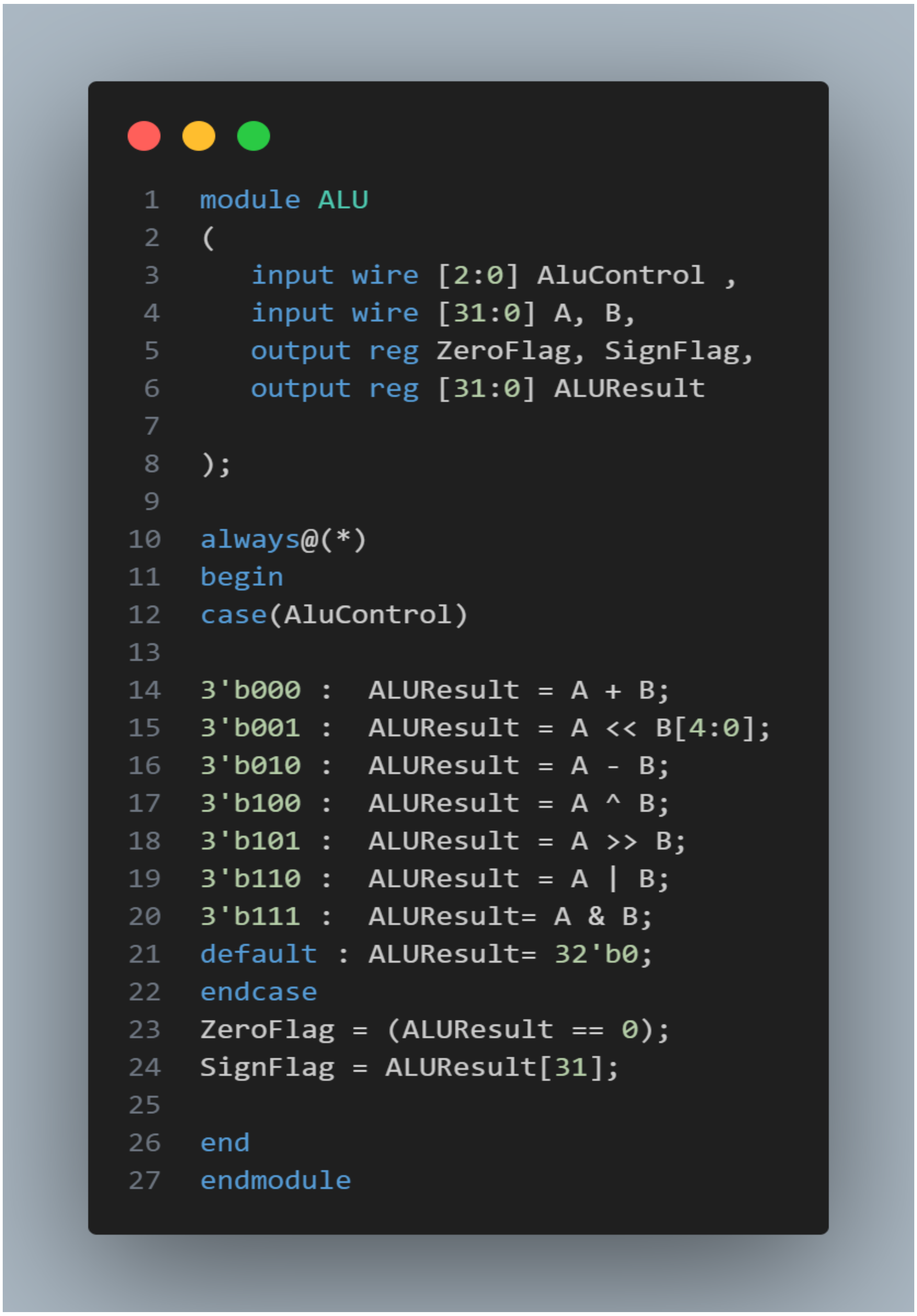
Main Modules

1-ALU

An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. The ALU forms the heart of most computer systems. The 3-bit ALUControl signal specifies the operation. The ALU generates a **32-bit ALUResult, a** Zero flag that indicates whether ALUResult == 0, and a sign flag that indicates ALU result sign (ALUResult [31]). Thefollowing table lists the specified functions that our ALU can perform.

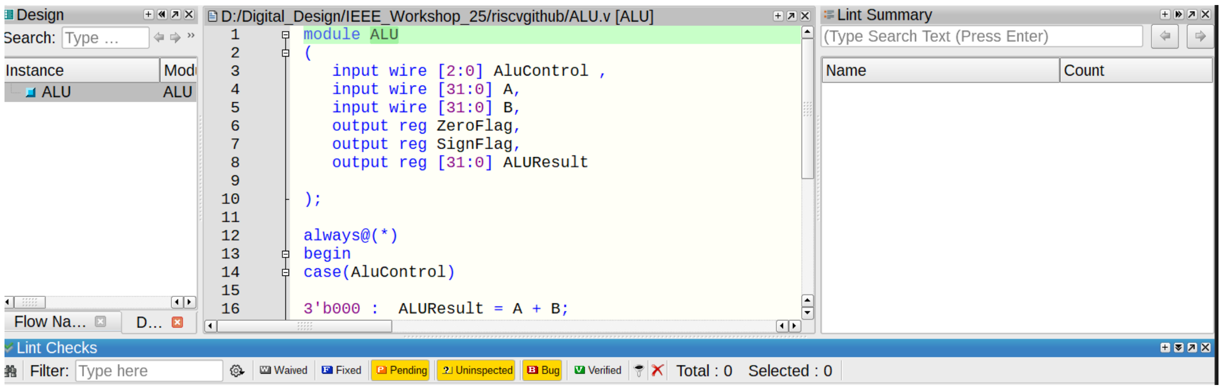


ALUControl	Function
A + B	000
A SHL B	001
A - B	010
A XOR B	100
A SHR B	101
A OR B	110
A AND B	111



```verilog
module ALU
(
    input wire [2:0] AluControl ,
    input wire [31:0] A, B,
    output reg ZeroFlag, SignFlag,
    output reg [31:0] ALUResult

);

always@(*)
begin
case(AluControl)

3'b000 :  ALUResult = A + B;
3'b001 :  ALUResult = A << B[4:0];
3'b010 :  ALUResult = A - B;
3'b100 :  ALUResult = A ^ B;
3'b101 :  ALUResult = A >> B;
3'b110 :  ALUResult = A | B;
3'b111 :  ALUResult= A & B;
default : ALUResult= 32'b0;
endcase
ZeroFlag = (ALUResult == 0);
SignFlag = ALUResult[31];

end
endmodule
```

No errors or warnings using QuestaLint



Design
Search: Type …

Instance	Mod
ALU	ALU

Flow Na… D…

D:/Digital_Design/IEEE_Workshop_25/riscvgithub/ALU.v [ALU]

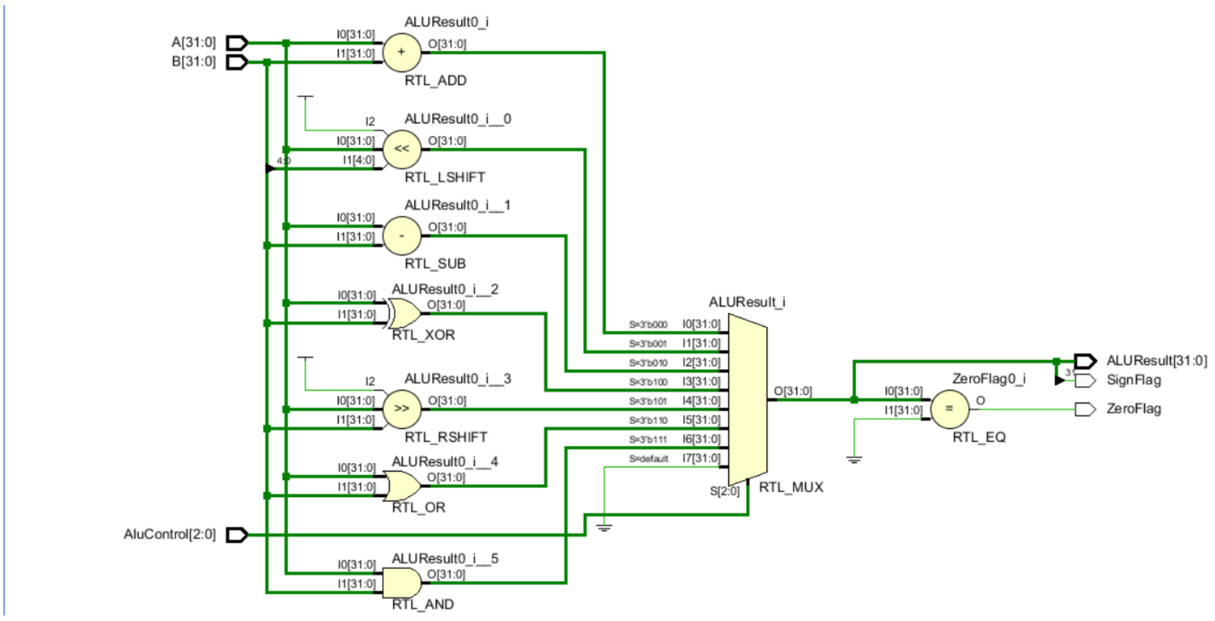
```
module ALU
(
   input wire [2:0] AluControl ,
   input wire [31:0] A,
   input wire [31:0] B,
   output reg ZeroFlag,
   output reg SignFlag,
   output reg [31:0] ALUResult

);

always@(*)
begin
case(AluControl)

3'b000 :  ALUResult = A + B;
```

Lint Summary
(Type Search Text (Press Enter)

Name	Count

Lint Checks
Filter: Type here   Waived   Fixed   Pending   Uninspected   Bug   Verified   Total : 0   Selected : 0

RTL Analysis Schematic using Vivado

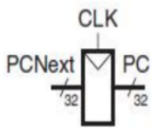


2-Program Counter

2.1. Program Counter Register

To fetch the instructions from the instruction memory, we need a pointer to keep track of the address of the current instruction for this task we use the program counter. The program counter is simply a 32-bit register that has the address of the current instruction at its output and the address of the next instruction at its input. Firstly, you need to implement this register and then the logic that calculates the address of the next instruction. The program counter has four inputs a 32-bit word which is the next address, the clock signal, asynchronous reset, and a load signal (always high except for the HLT instruction). And have one 32-bit output PC.

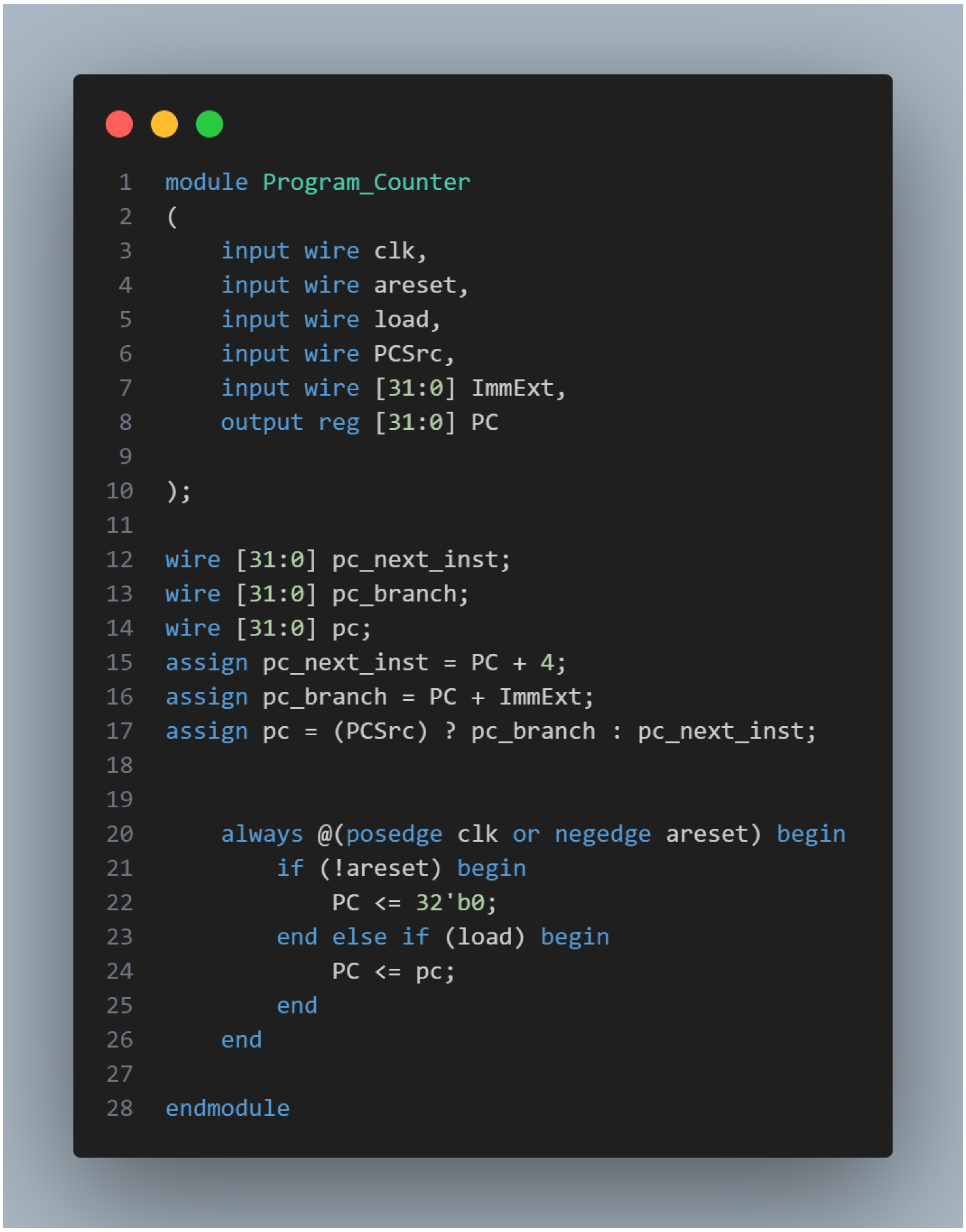
The following truth table describes the behavior of this register.



areset	load	clk	PC
0	x	x	0
1	0	Posedge	PC
1	1	Posedge	PCNext
1	x	Not posedge	PC

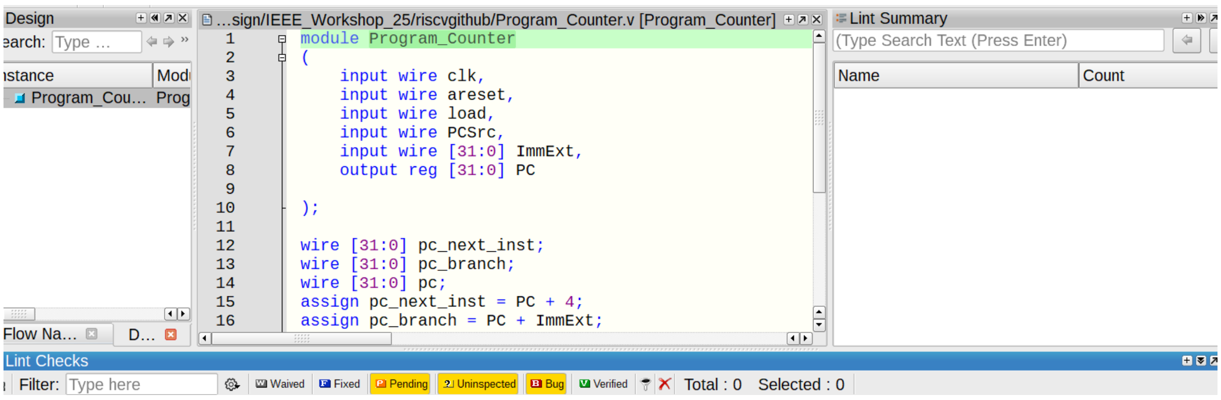
2.2. PC next logic

PCSrc	PCNext
0	PC + 4
1	PC + ImmExt



```verilog
module Program_Counter
(
    input wire clk,
    input wire areset,
    input wire load,
    input wire PCSrc,
    input wire [31:0] ImmExt,
    output reg [31:0] PC

);

wire [31:0] pc_next_inst;
wire [31:0] pc_branch;
wire [31:0] pc;
assign pc_next_inst = PC + 4;
assign pc_branch = PC + ImmExt;
assign pc = (PCSrc) ? pc_branch : pc_next_inst;


    always @(posedge clk or negedge areset) begin
        if (!areset) begin
            PC <= 32'b0;
        end else if (load) begin
            PC <= pc;
        end
    end

endmodule
```

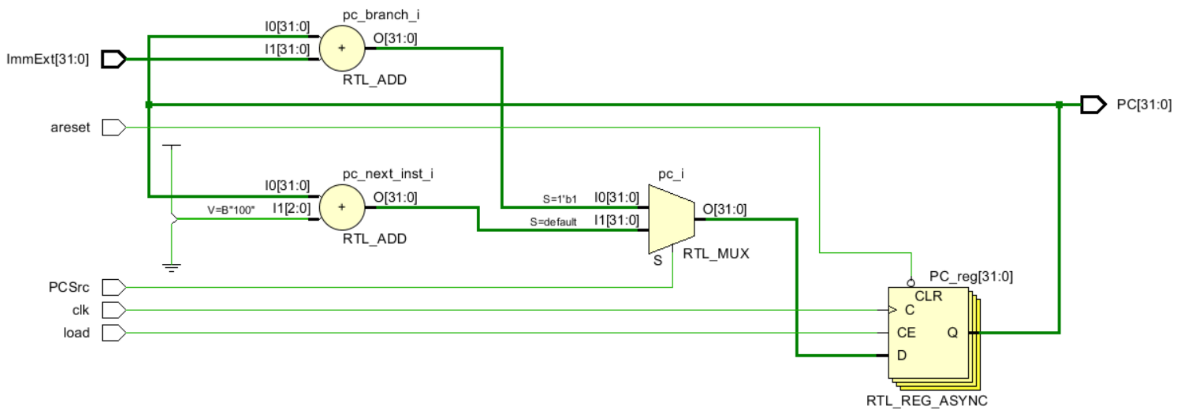
No errors or warnings using QuestaLint



```
module Program_Counter
(
    input wire clk,
    input wire areset,
    input wire load,
    input wire PCSrc,
    input wire [31:0] ImmExt,
    output reg [31:0] PC

);

wire [31:0] pc_next_inst;
wire [31:0] pc_branch;
wire [31:0] pc;
assign pc_next_inst = PC + 4;
assign pc_branch = PC + ImmExt;
```

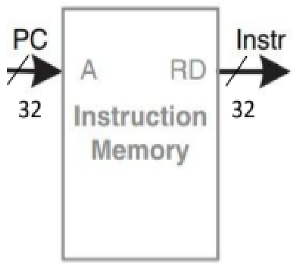
Total : 0   Selected : 0

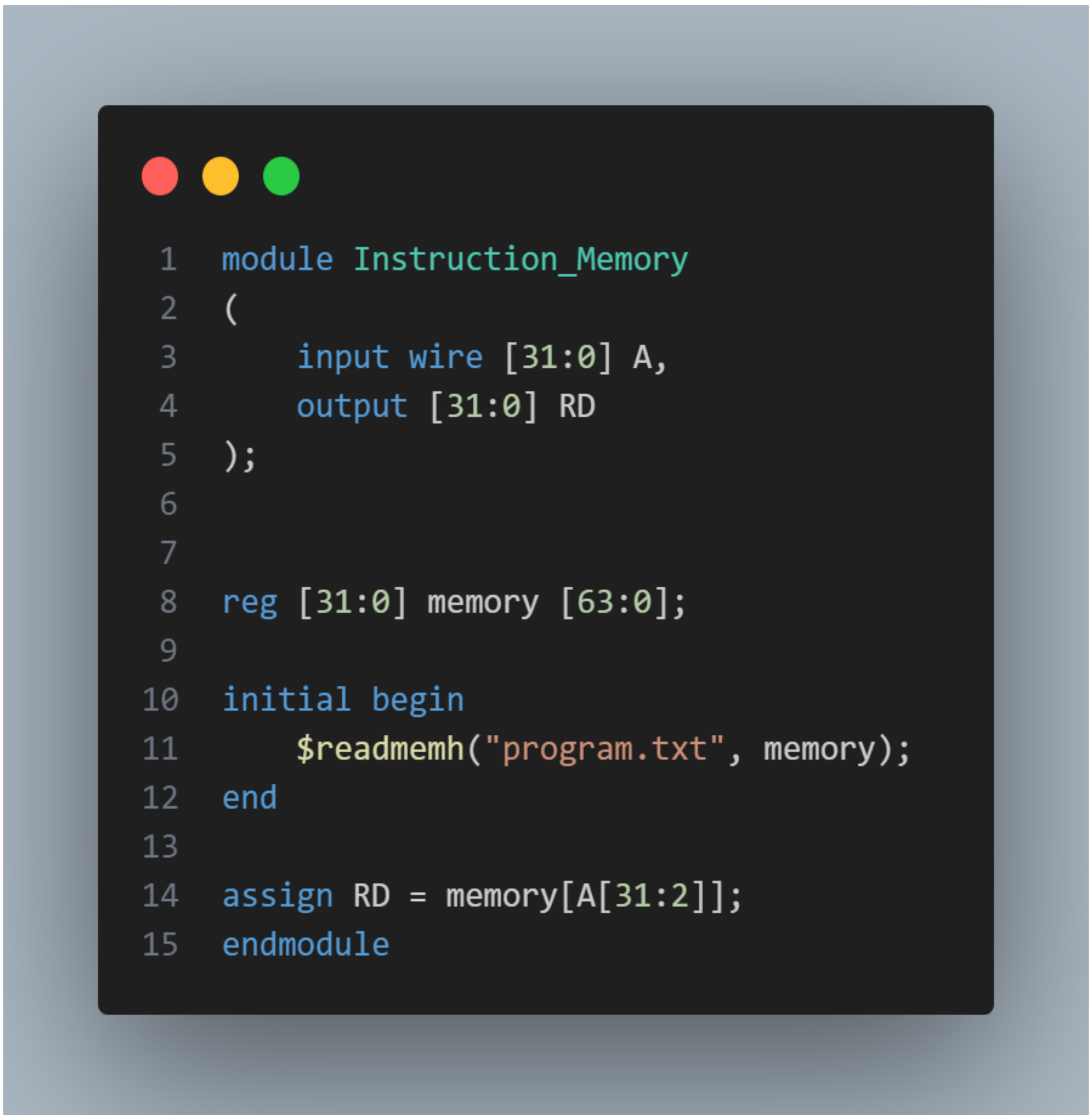
RTL Analysis Schematic using Vivado



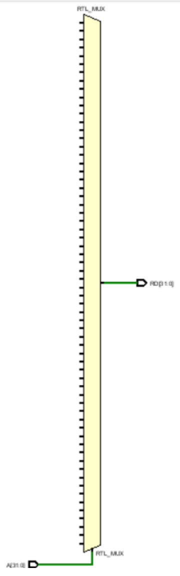
3-Instruction Memory

- The instruction memory has a single read port.
- It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.
- The PC is simply connected to the address input of the instruction memory.
- The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr.
- Our instruction memory is a Read Only Memory (ROM) that holds the program that your CPU will execute.
- The ROM Memory has width = 32 bits and depth = 64 entries.
- Instructions is read asynchronously.



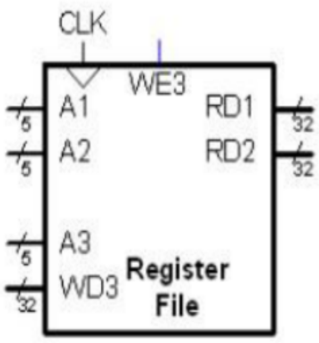


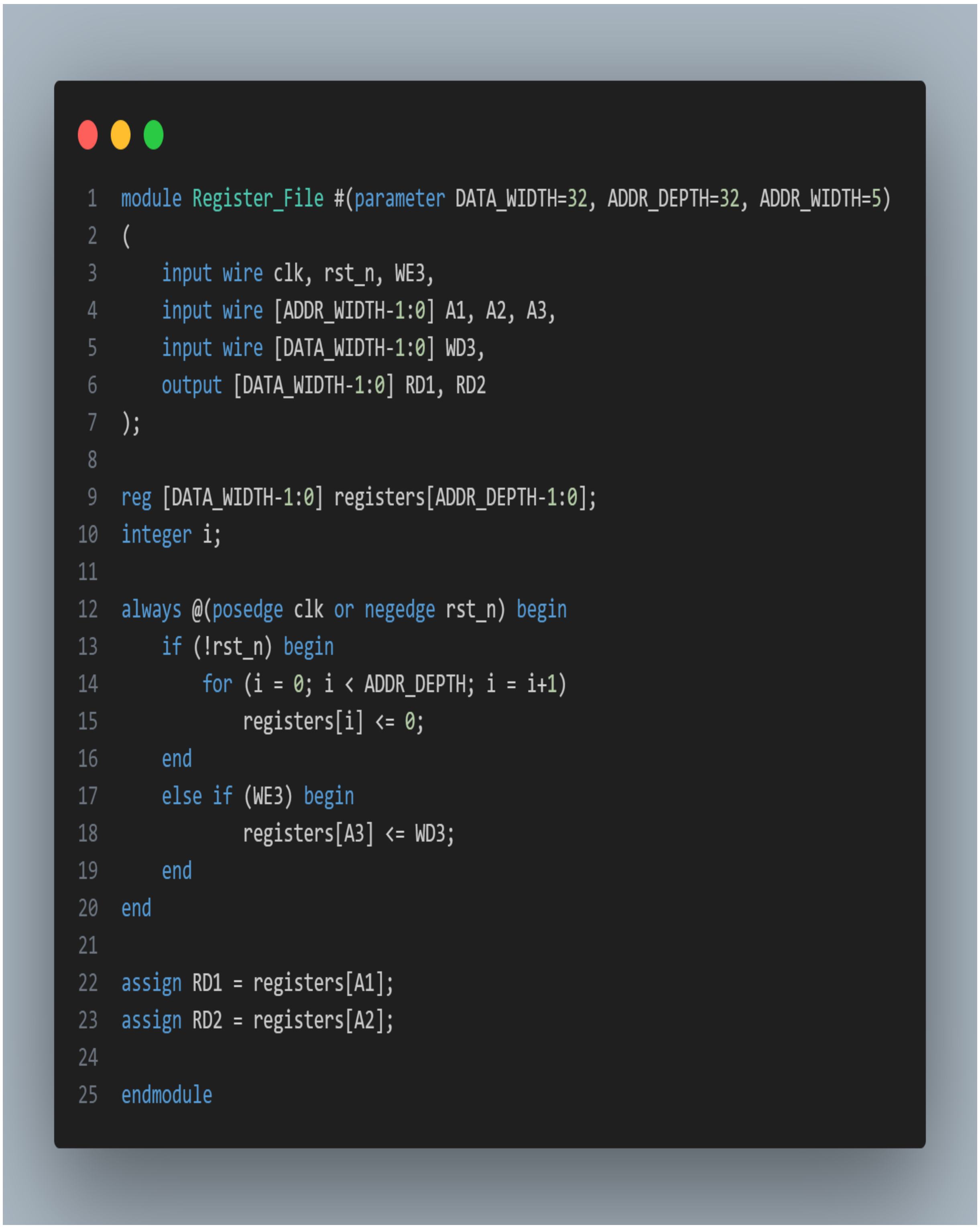
```verilog
module Instruction_Memory
(
    input wire [31:0] A,
    output [31:0] RD
);


reg [31:0] memory [63:0];

initial begin
    $readmemh("program.txt", memory);
end

assign RD = memory[A[31:2]];
endmodule
```


RTL_MUX
RD[31:0]
RTL_MUX
A[31:0]

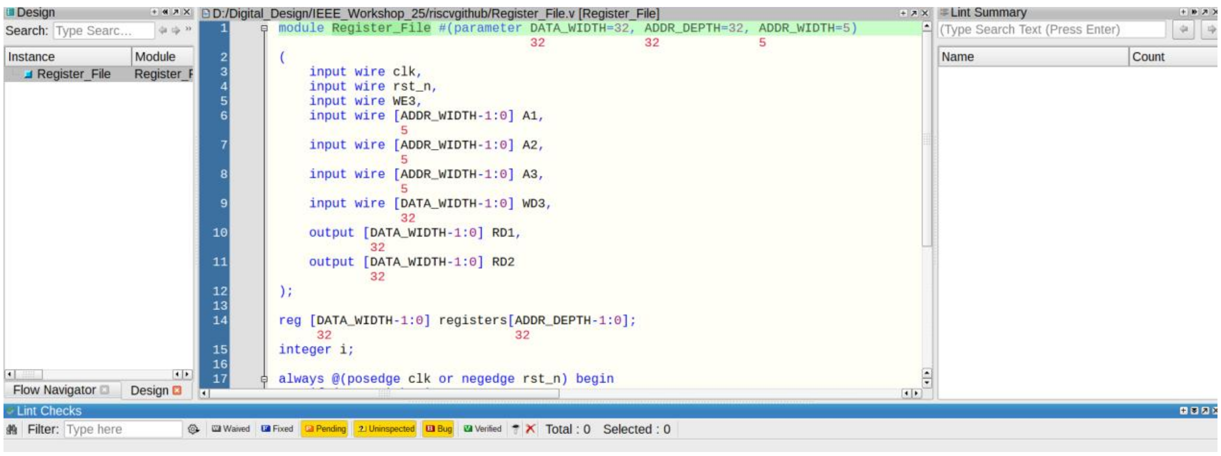
4-Register File

- The Register File contains the 32-bit registers.
- The register file has two read output ports (RD1 and RD2) and a single input write port (WD3), RD1 and RD2 are read with no respect to the clock edge.
- The register file is read asynchronously and written synchronously at the rising edge of the clock.
- The register file supports simultaneous read and writes. The register file has width = 32 bits and depth = 32 entries supports simultaneous read and writes.
- The register file has active low asynchronous reset signal.
- A1 is the register address from which the data are read through the output port RD1. Whereas A2 is corresponding to the register address of output port RD2.



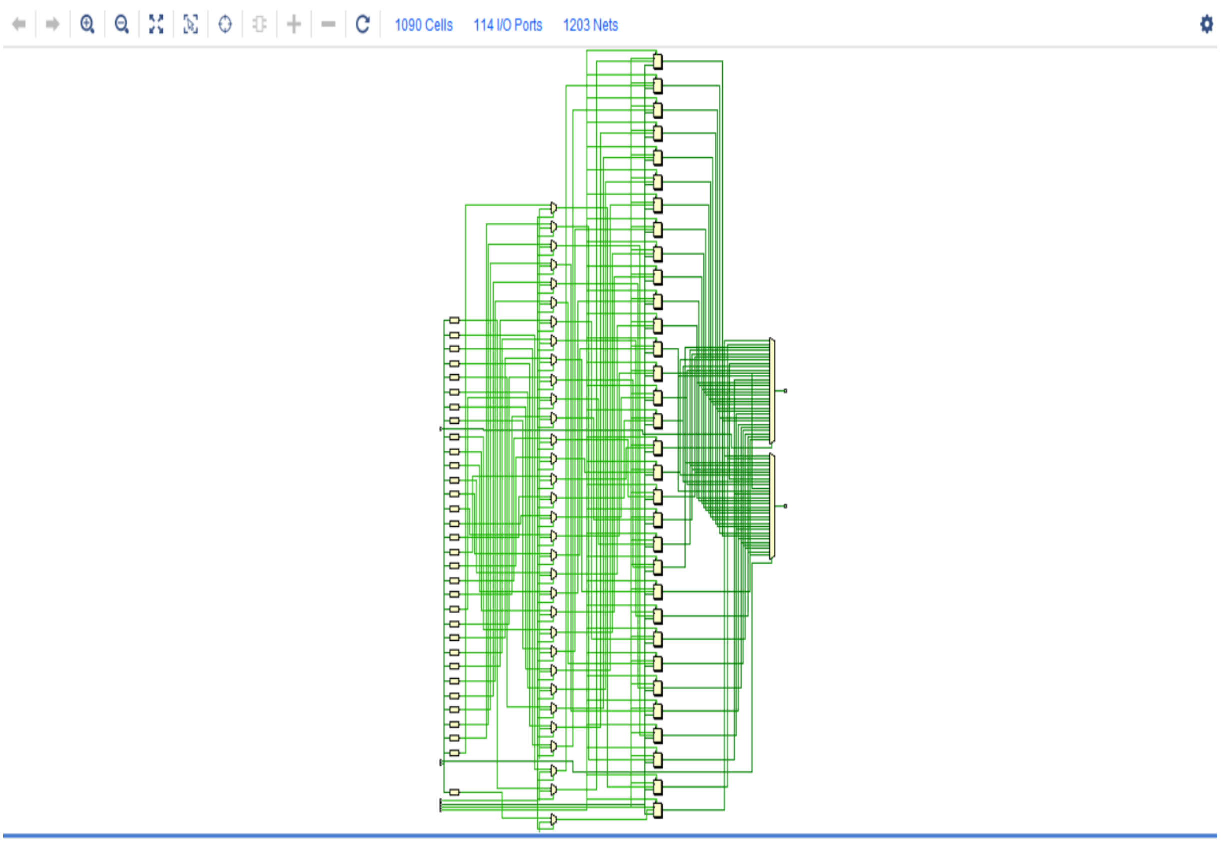


```verilog
module Register_File #(parameter DATA_WIDTH=32, ADDR_DEPTH=32, ADDR_WIDTH=5)
(
    input wire clk, rst_n, WE3,
    input wire [ADDR_WIDTH-1:0] A1, A2, A3,
    input wire [DATA_WIDTH-1:0] WD3,
    output [DATA_WIDTH-1:0] RD1, RD2
);

reg [DATA_WIDTH-1:0] registers[ADDR_DEPTH-1:0];
integer i;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        for (i = 0; i < ADDR_DEPTH; i = i+1)
            registers[i] <= 0;
    end
    else if (WE3) begin
            registers[A3] <= WD3;
    end
end

assign RD1 = registers[A1];
assign RD2 = registers[A2];

endmodule
```



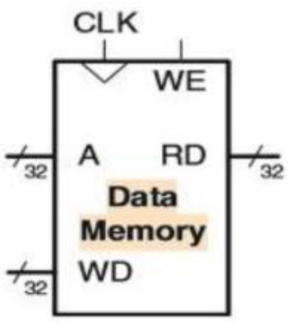
D:/Digital_Design/IEEE_Workshop_25/riscvgithub/Register_File.v [Register_File]

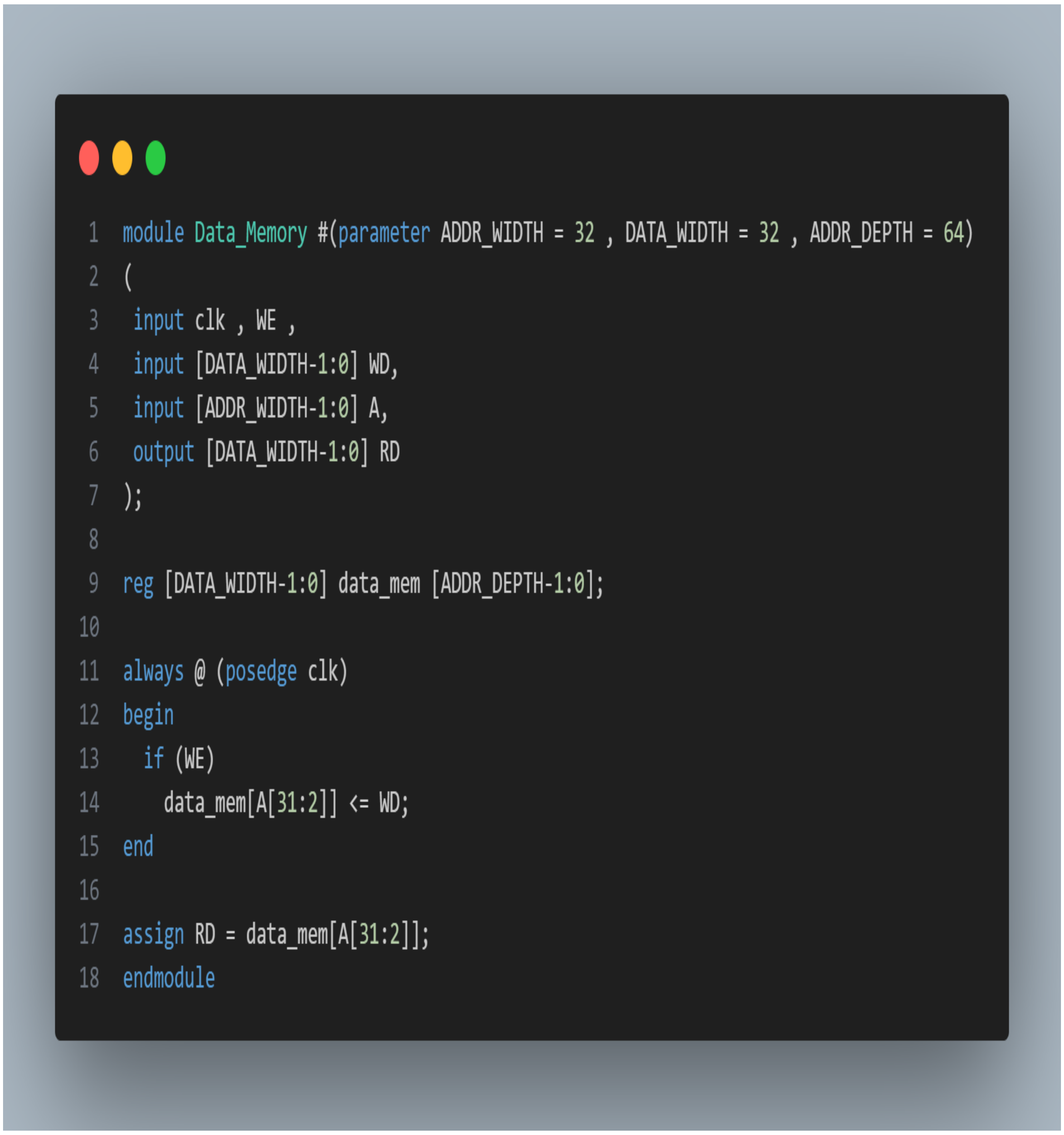
```
module Register_File #(parameter DATA_WIDTH=32, ADDR_DEPTH=32, ADDR_WIDTH=5)
                                           32              32              5
(
    input wire clk,
    input wire rst_n,
    input wire WE3,
    input wire [ADDR_WIDTH-1:0] A1,
                 5
    input wire [ADDR_WIDTH-1:0] A2,
                 5
    input wire [ADDR_WIDTH-1:0] A3,
                 5
    input wire [DATA_WIDTH-1:0] WD3,
                 32
    output [DATA_WIDTH-1:0] RD1,
             32
    output [DATA_WIDTH-1:0] RD2
             32
);

reg [DATA_WIDTH-1:0] registers[ADDR_DEPTH-1:0];
       32                          32
integer i;

always @(posedge clk or negedge rst_n) begin
```



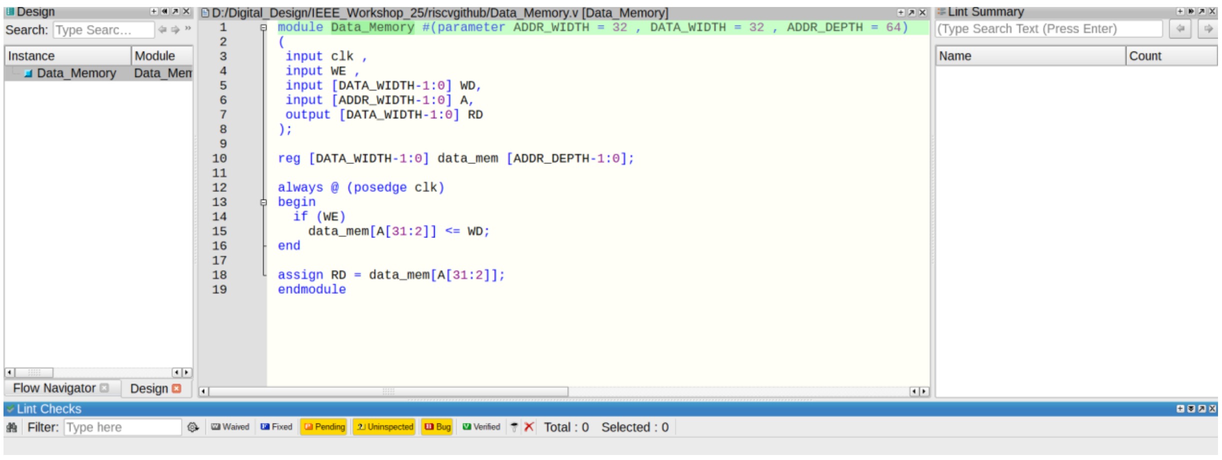
5-Data Memory

- It has a single read/write port.
- If its write enable, WE, is asserted, then it writes data WD into address A on the rising edge of the clock.
- It reads are asynchronous while writes are synchronous to the rising edge of the “clk” signal.
- The Word width of the data memory is 32-bits to match the datapath width. The data memory contains 64 entries.
- RD is read with no respect to the clock edge.
- A is the memory address from which the data are read through the output port RD.



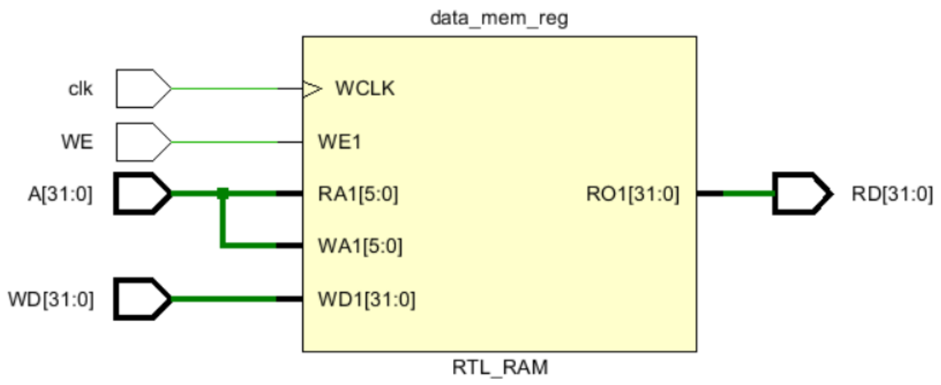


```verilog
module Data_Memory #(parameter ADDR_WIDTH = 32 , DATA_WIDTH = 32 , ADDR_DEPTH = 64)
(
 input clk , WE ,
 input [DATA_WIDTH-1:0] WD,
 input [ADDR_WIDTH-1:0] A,
 output [DATA_WIDTH-1:0] RD
);

reg [DATA_WIDTH-1:0] data_mem [ADDR_DEPTH-1:0];

always @ (posedge clk)
begin
  if (WE)
    data_mem[A[31:2]] <= WD;
end

assign RD = data_mem[A[31:2]];
endmodule
```



D:/Digital_Design/IEEE_Workshop_25/riscvgithub/Data_Memory.v [Data_Memory]

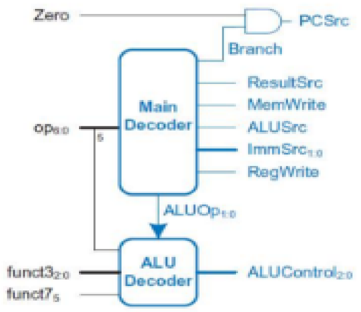
```verilog
module Data_Memory #(parameter ADDR_WIDTH = 32 , DATA_WIDTH = 32 , ADDR_DEPTH = 64)
(
 input clk ,
 input WE ,
 input [DATA_WIDTH-1:0] WD,
 input [ADDR_WIDTH-1:0] A,
 output [DATA_WIDTH-1:0] RD
);

reg [DATA_WIDTH-1:0] data_mem [ADDR_DEPTH-1:0];

always @ (posedge clk)
begin
  if (WE)
    data_mem[A[31:2]] <= WD;
end

assign RD = data_mem[A[31:2]];
endmodule
```



6-Control Unit

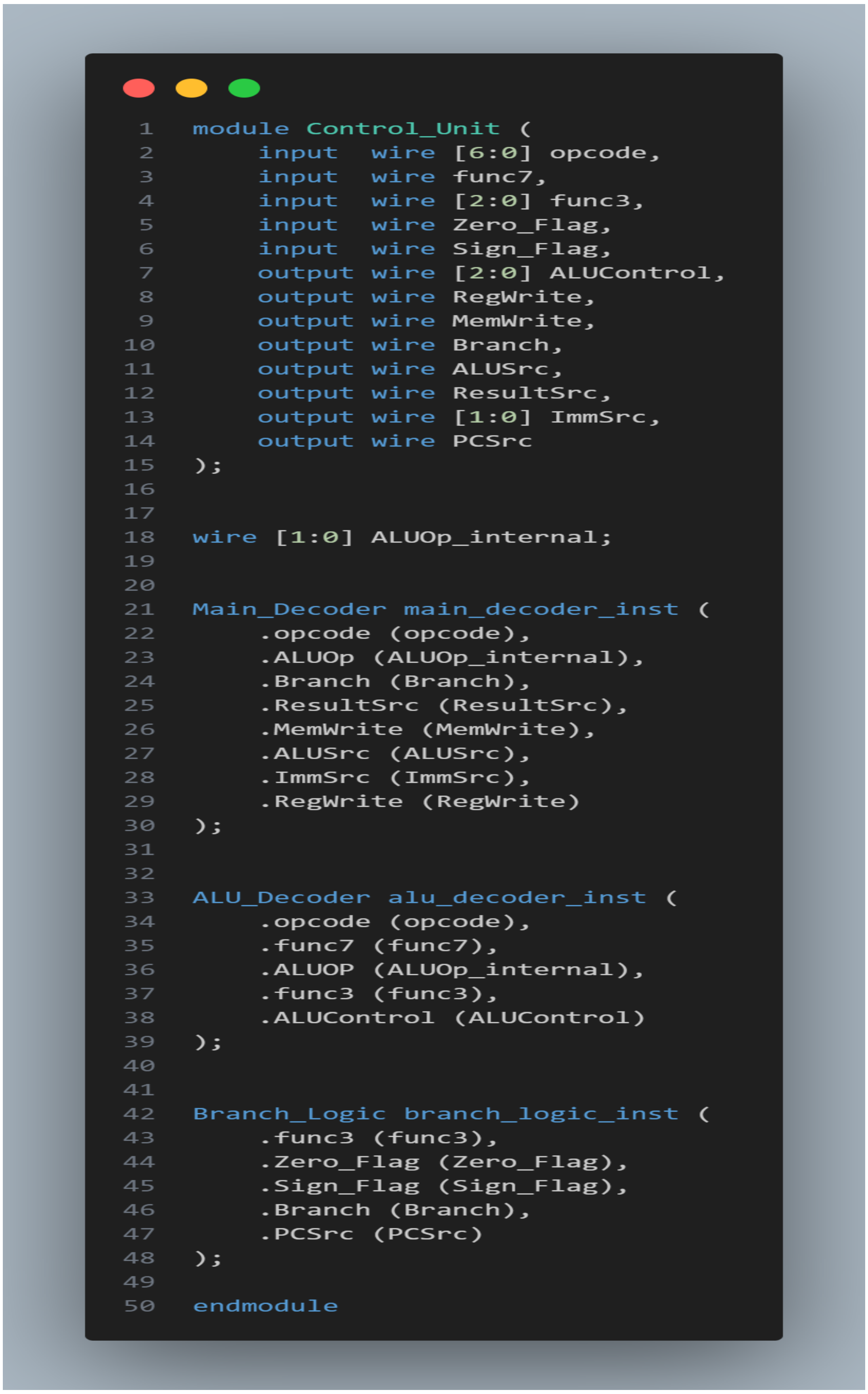
6.Control Unit

The control unit computes the control signals based on the opcode and funct3, funct7 fields ofthe instruction, Instr14:12 and Instr30 respectively. Most of the control information comes from the opcode, but R-type instructions and I-type instructions also use the funct3 and funct7 fields to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in the figure below.

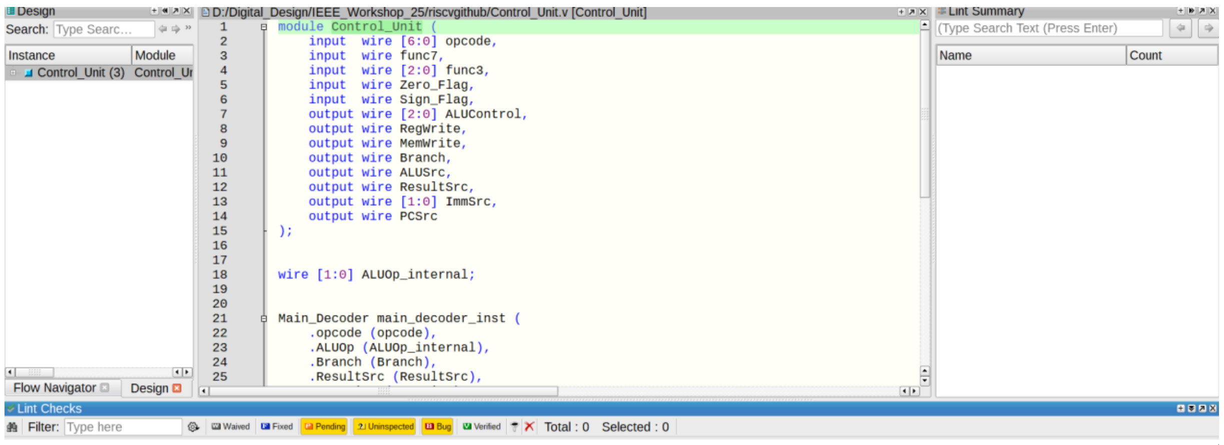


ALU Decoder truth table

ALUOP	$funct_3$	$\{op_5, funct_7\}$	ALUcontrol	Instruction
00	XXX	XX	000(add)	lw,sw
01	000	XX	010(sub)	beq
	001	XX	010(sub)	bnq
	100	XX	010(sub)	blt
10	000	00,01,10	000(add)	add
	000	11	010(sub)	subtract
	001	XX	001(shift left)	SHL
	100	XX	100(XOR)	XOR
	101	XX	101(shift right)	SHR
	110	XX	110(OR)	OR
	111	XX	111(AND)	AND
Default	XXX	XX	000	

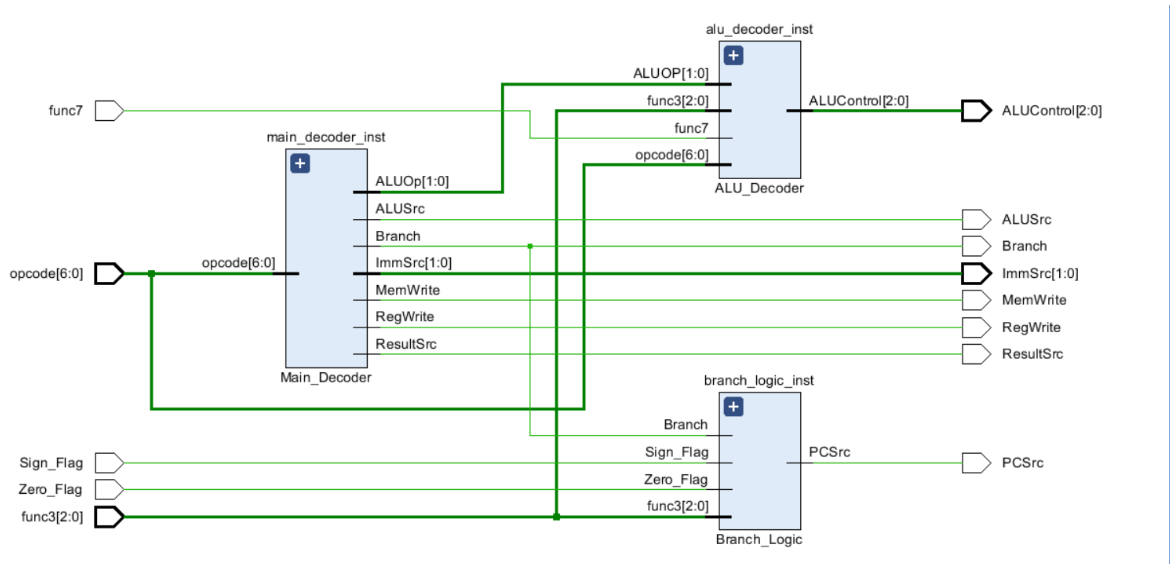


```verilog
module Control_Unit (
     input  wire [6:0] opcode,
     input  wire func7,
     input  wire [2:0] func3,
     input  wire Zero_Flag,
     input  wire Sign_Flag,
     output wire [2:0] ALUControl,
     output wire RegWrite,
     output wire MemWrite,
     output wire Branch,
     output wire ALUSrc,
     output wire ResultSrc,
     output wire [1:0] ImmSrc,
     output wire PCSrc
);


wire [1:0] ALUOp_internal;


Main_Decoder main_decoder_inst (
     .opcode (opcode),
     .ALUOp (ALUOp_internal),
     .Branch (Branch),
     .ResultSrc (ResultSrc),
     .MemWrite (MemWrite),
     .ALUSrc (ALUSrc),
     .ImmSrc (ImmSrc),
     .RegWrite (RegWrite)
);


ALU_Decoder alu_decoder_inst (
     .opcode (opcode),
     .func7 (func7),
     .ALUOP (ALUOp_internal),
     .func3 (func3),
     .ALUControl (ALUControl)
);


Branch_Logic branch_logic_inst (
     .func3 (func3),
     .Zero_Flag (Zero_Flag),
     .Sign_Flag (Sign_Flag),
     .Branch (Branch),
     .PCSrc (PCSrc)
);

endmodule
```

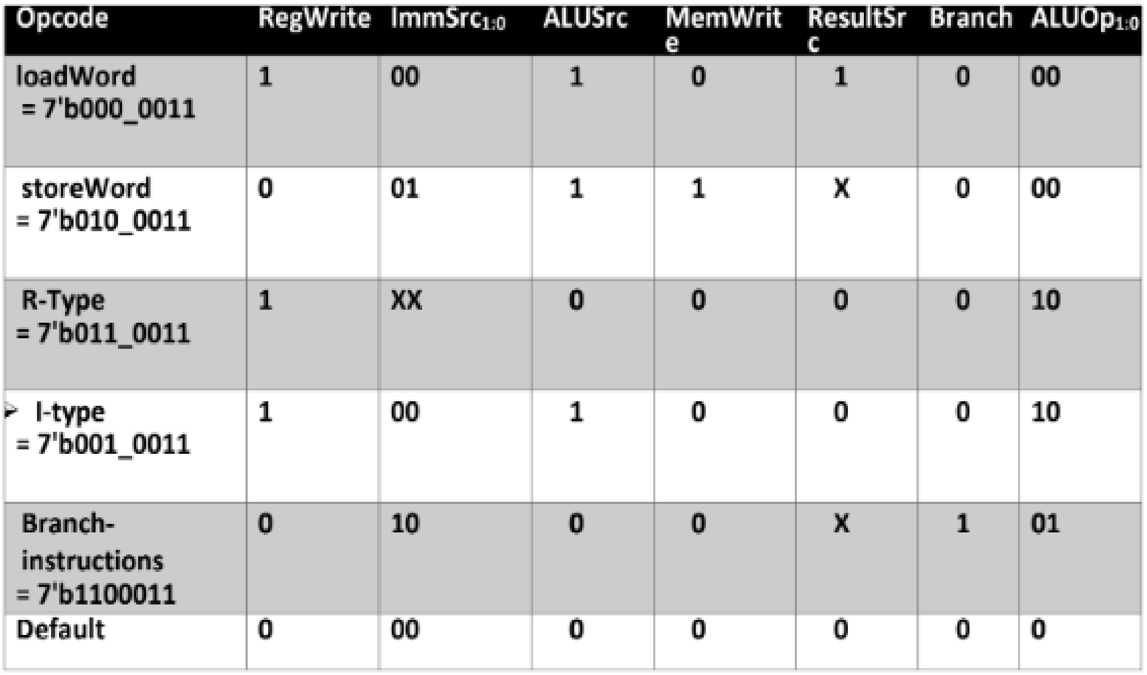


D:/Digital_Design/IEEE_Workshop_25/riscvgithub/Control_Unit.v [Control_Unit]

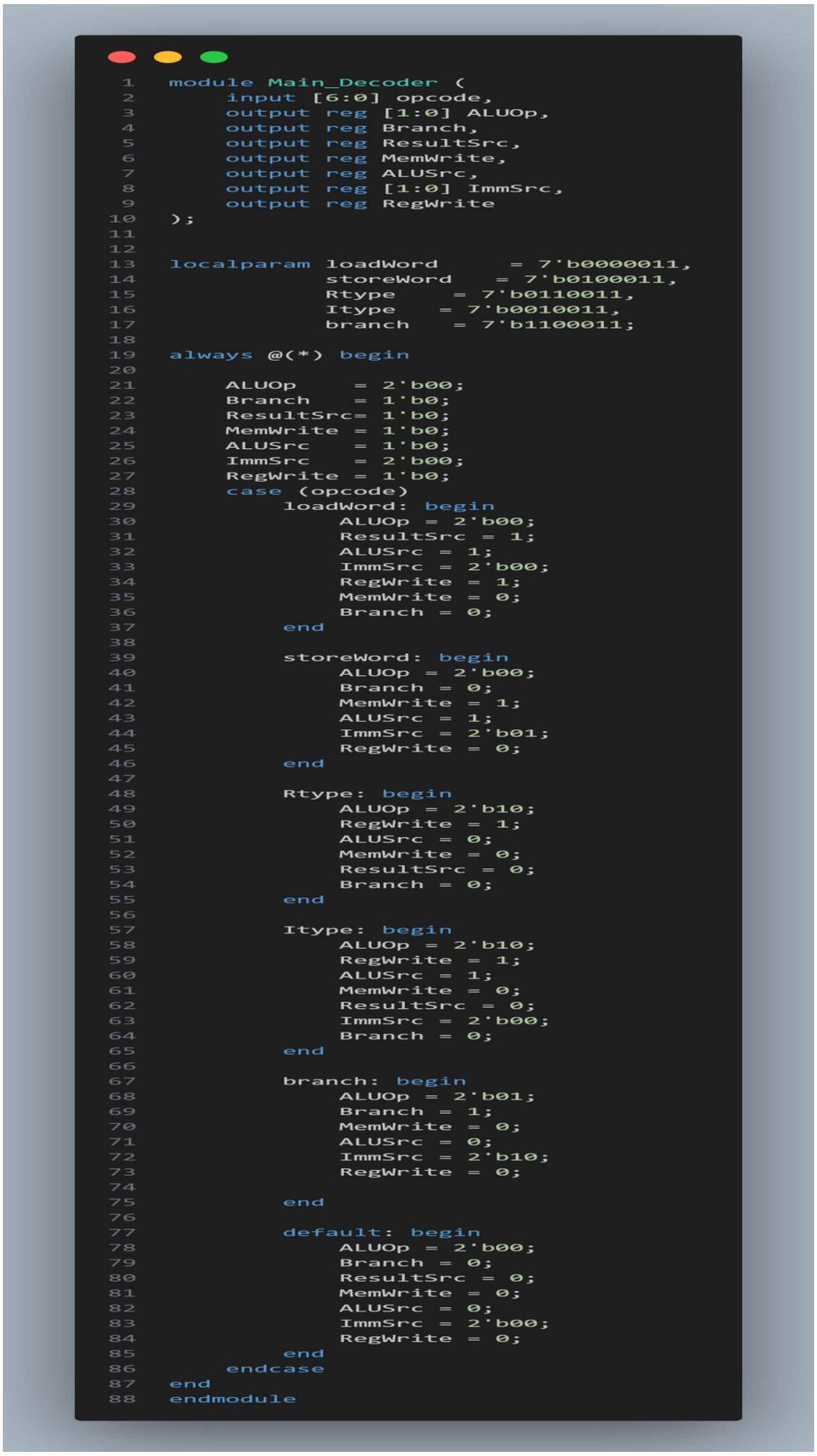
```
module Control_Unit (
    input  wire [6:0] opcode,
    input  wire func7,
    input  wire [2:0] func3,
    input  wire Zero_Flag,
    input  wire Sign_Flag,
    output wire [2:0] ALUControl,
    output wire RegWrite,
    output wire MemWrite,
    output wire Branch,
    output wire ALUSrc,
    output wire ResultSrc,
    output wire [1:0] ImmSrc,
    output wire PCSrc
);


wire [1:0] ALUOp_internal;


Main_Decoder main_decoder_inst (
    .opcode (opcode),
    .ALUop (ALUOp_internal),
    .Branch (Branch),
    .ResultSrc (ResultSrc),
```



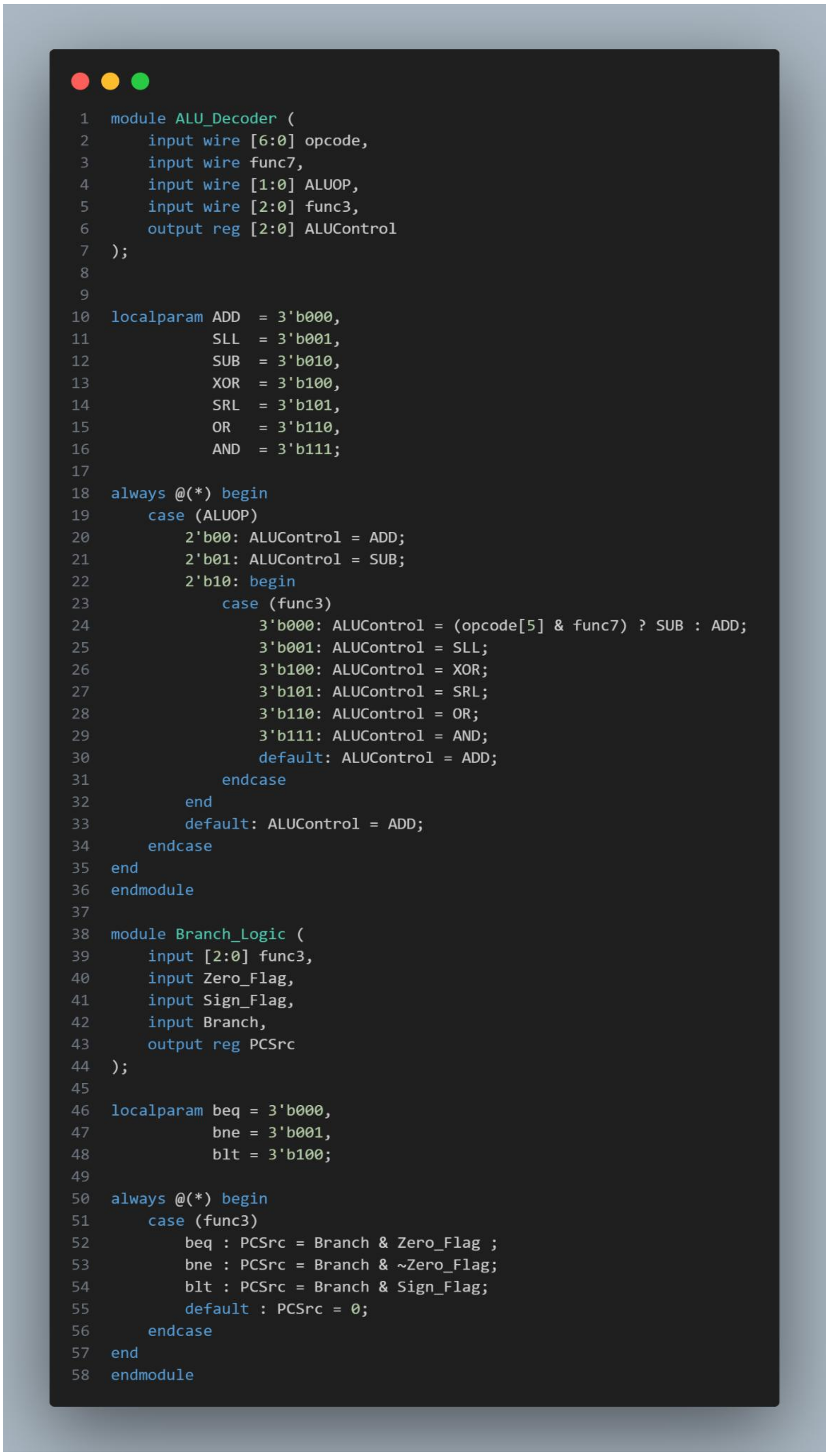
7- Main Decoder

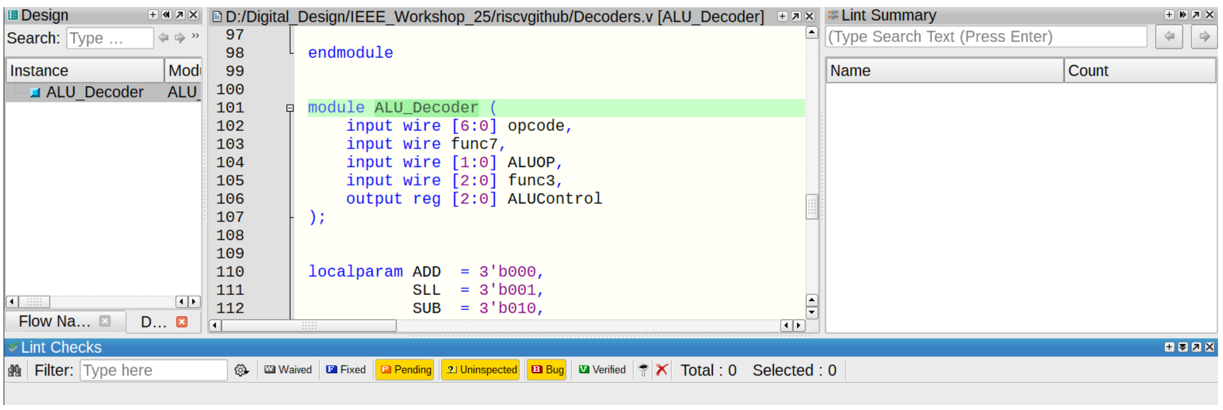


Opcode	RegWrite	ImmSrc$_{1:0}$	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp$_{1:0}$
loadWord = 7'b000_0011	1	00	1	0	1	0	00
storeWord = 7'b010_0011	0	01	1	1	X	0	00
R-Type = 7'b011_0011	1	XX	0	0	0	0	10
➢ I-type = 7'b001_0011	1	00	1	0	0	0	10
Branch-instructions = 7'b1100011	0	10	0	0	X	1	01
Default	0	00	0	0	0	0	0



```verilog
module Main_Decoder (
    input [6:0] opcode,
    output reg [1:0] ALUOp,
    output reg Branch,
    output reg ResultSrc,
    output reg MemWrite,
    output reg ALUSrc,
    output reg [1:0] ImmSrc,
    output reg RegWrite
);


localparam loadWord     = 7'b0000011,
           storeWord   = 7'b0100011,
           Rtype    = 7'b0110011,
           Itype   = 7'b0010011,
           branch   = 7'b1100011;

always @(*) begin

    ALUOp    = 2'b00;
    Branch   = 1'b0;
    ResultSrc= 1'b0;
    MemWrite = 1'b0;
    ALUSrc   = 1'b0;
    ImmSrc   = 2'b00;
    RegWrite = 1'b0;
    case (opcode)
        loadWord: begin
            ALUOp = 2'b00;
            ResultSrc = 1;
            ALUSrc = 1;
            ImmSrc = 2'b00;
            RegWrite = 1;
            MemWrite = 0;
            Branch = 0;
        end

        storeWord: begin
            ALUOp = 2'b00;
            Branch = 0;
            MemWrite = 1;
            ALUSrc = 1;
            ImmSrc = 2'b01;
            RegWrite = 0;
        end

        Rtype: begin
            ALUOp = 2'b10;
            RegWrite = 1;
            ALUSrc = 0;
            MemWrite = 0;
            ResultSrc = 0;
            Branch = 0;
        end

        Itype: begin
            ALUOp = 2'b10;
            RegWrite = 1;
            ALUSrc = 1;
            MemWrite = 0;
            ResultSrc = 0;
            ImmSrc = 2'b00;
            Branch = 0;
        end

        branch: begin
            ALUOp = 2'b01;
            Branch = 1;
            MemWrite = 0;
            ALUSrc = 0;
            ImmSrc = 2'b10;
            RegWrite = 0;

        end

        default: begin
            ALUOp = 2'b00;
            Branch = 0;
            ResultSrc = 0;
            MemWrite = 0;
            ALUSrc = 0;
            ImmSrc = 2'b00;
            RegWrite = 0;
        end
    endcase
end
endmodule
```


```verilog
module ALU_Decoder (
    input wire [6:0] opcode,
    input wire func7,
    input wire [1:0] ALUOP,
    input wire [2:0] func3,
    output reg [2:0] ALUControl
);


localparam ADD  = 3'b000,
           SLL  = 3'b001,
           SUB  = 3'b010,
           XOR  = 3'b100,
           SRL  = 3'b101,
           OR   = 3'b110,
           AND  = 3'b111;

always @(*) begin
    case (ALUOP)
        2'b00: ALUControl = ADD;
        2'b01: ALUControl = SUB;
        2'b10: begin
            case (func3)
                3'b000: ALUControl = (opcode[5] & func7) ? SUB : ADD;
                3'b001: ALUControl = SLL;
                3'b100: ALUControl = XOR;
                3'b101: ALUControl = SRL;
                3'b110: ALUControl = OR;
                3'b111: ALUControl = AND;
                default: ALUControl = ADD;
            endcase
        end
        default: ALUControl = ADD;
    endcase
end
endmodule

module Branch_Logic (
    input [2:0] func3,
    input Zero_Flag,
    input Sign_Flag,
    input Branch,
    output reg PCSrc
);

localparam beq = 3'b000,
           bne = 3'b001,
           blt = 3'b100;

always @(*) begin
    case (func3)
        beq : PCSrc = Branch & Zero_Flag ;
        bne : PCSrc = Branch & ~Zero_Flag;
        blt : PCSrc = Branch & Sign_Flag;
        default : PCSrc = 0;
    endcase
end
endmodule
```



Design
Search: Type ...
Instance | Modu
ALU_Decoder | ALU_
Flow Na... D...

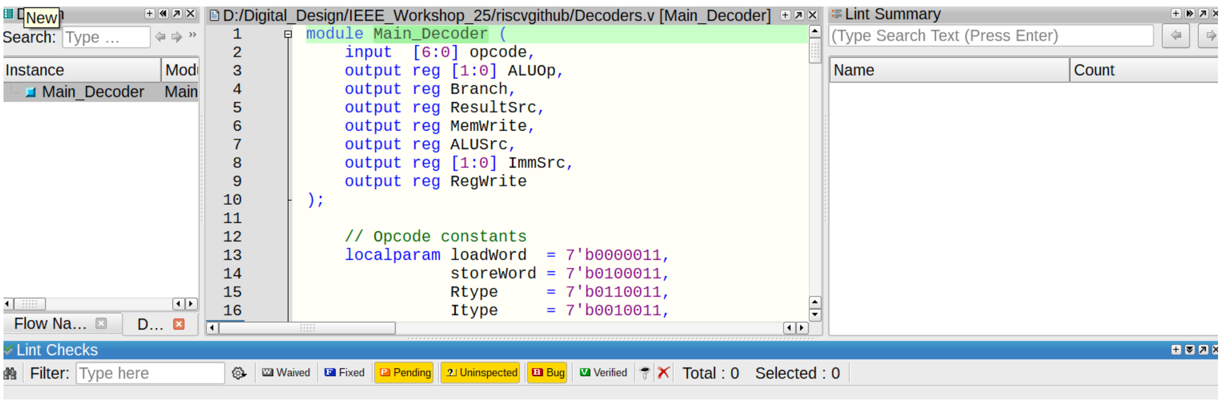
D:/Digital_Design/IEEE_Workshop_25/riscvgithub/Decoders.v [ALU_Decoder]

```
endmodule


module ALU_Decoder (
    input wire [6:0] opcode,
    input wire func7,
    input wire [1:0] ALUOP,
    input wire [2:0] func3,
    output reg [2:0] ALUControl
);


localparam ADD  = 3'b000,
           SLL  = 3'b001,
           SUB  = 3'b010,
```

Lint Summary
(Type Search Text (Press Enter)

Name	Count

Lint Checks
Filter: Type here Waived Fixed Pending Uninspected Bug Verified Total : 0 Selected : 0



New
Search: Type ...
Instance | Mod
Main_Decoder | Main
Flow Na... D...

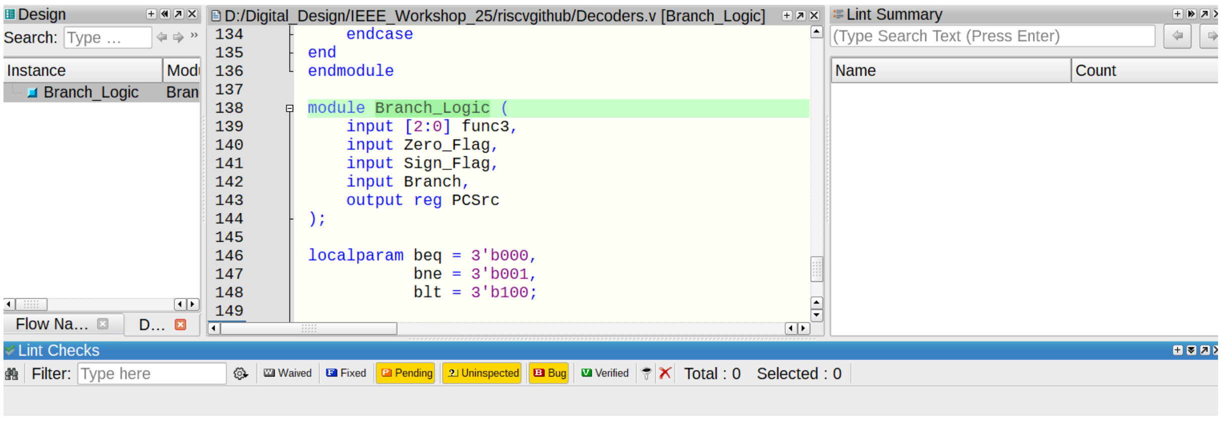
D:/Digital_Design/IEEE_Workshop_25/riscvgithub/Decoders.v [Main_Decoder]

```
module Main_Decoder (
    input   [6:0] opcode,
    output reg [1:0] ALUOp,
    output reg Branch,
    output reg ResultSrc,
    output reg MemWrite,
    output reg ALUSrc,
    output reg [1:0] ImmSrc,
    output reg RegWrite
);

    // Opcode constants
    localparam loadWord  = 7'b0000011,
               storeWord = 7'b0100011,
               Rtype     = 7'b0110011,
               Itype     = 7'b0010011,
```

Lint Summary
(Type Search Text (Press Enter)

Name	Count

Lint Checks
Filter: Type here Waived Fixed Pending Uninspected Bug Verified Total : 0 Selected : 0



Design
Search: Type ...
Instance | Mod
Branch_Logic | Bran
Flow Na... D...

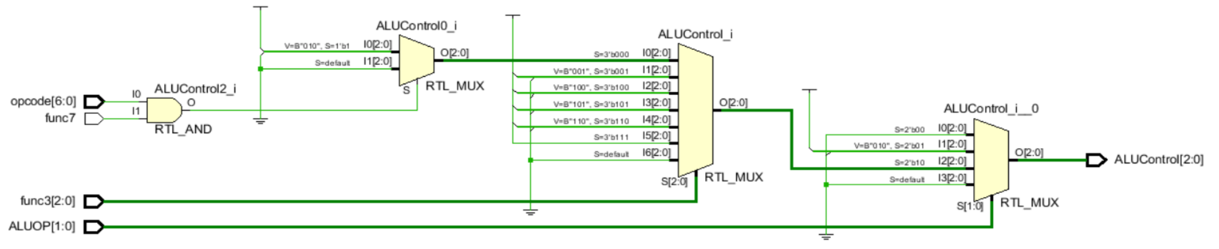
D:/Digital_Design/IEEE_Workshop_25/riscvgithub/Decoders.v [Branch_Logic]

```
    endcase
end
endmodule

module Branch_Logic (
    input [2:0] func3,
    input Zero_Flag,
    input Sign_Flag,
    input Branch,
    output reg PCSrc
);

localparam beq = 3'b000,
           bne = 3'b001,
           blt = 3'b100;

```

Lint Summary
(Type Search Text (Press Enter)

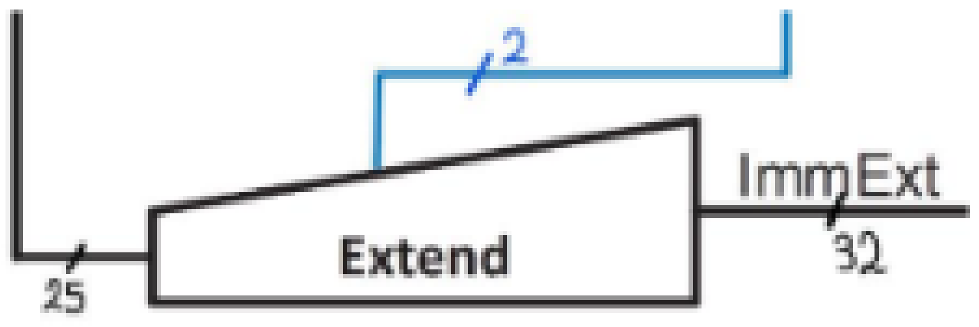
Name	Count

Lint Checks
Filter: Type here Waived Fixed Pending Uninspected Bug Verified Total : 0 Selected : 0

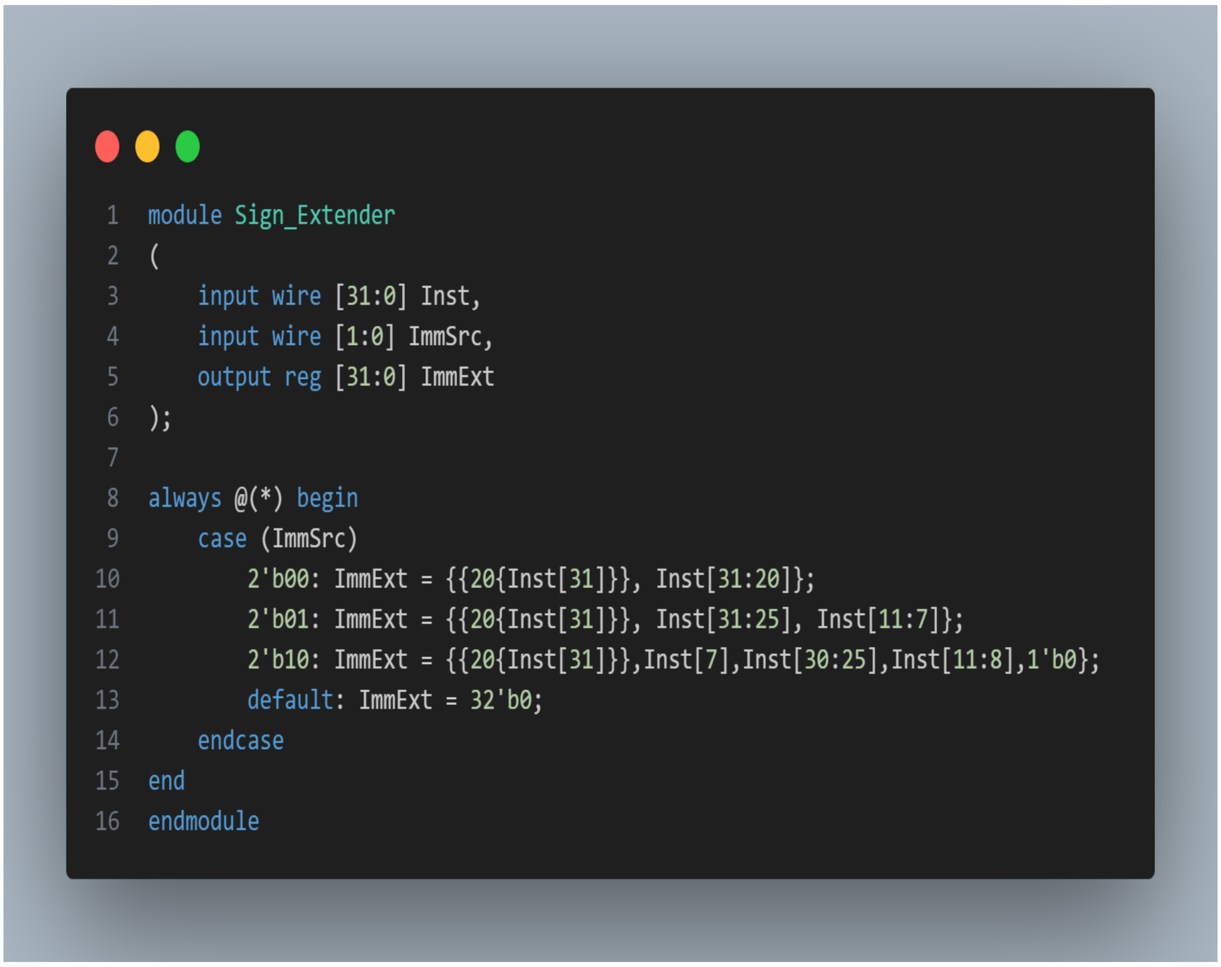


Small Modules

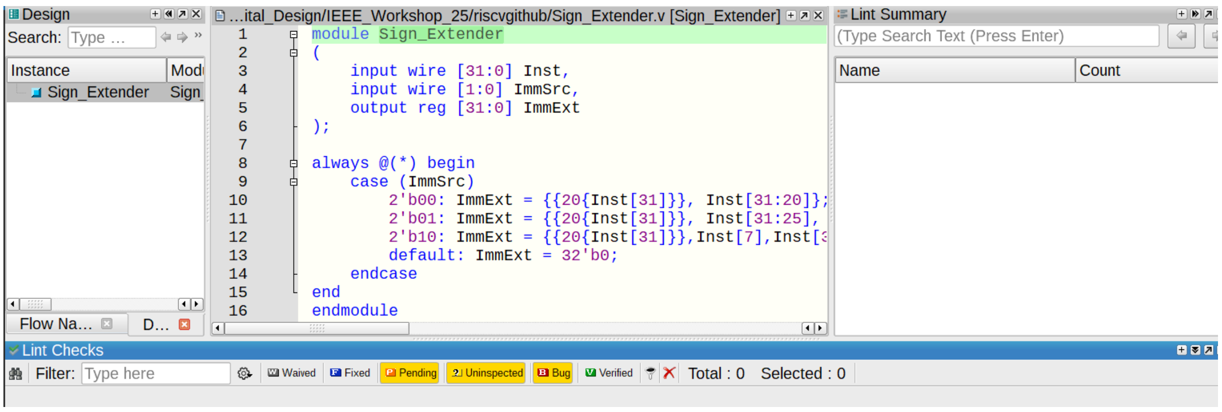
1-Sign Extender



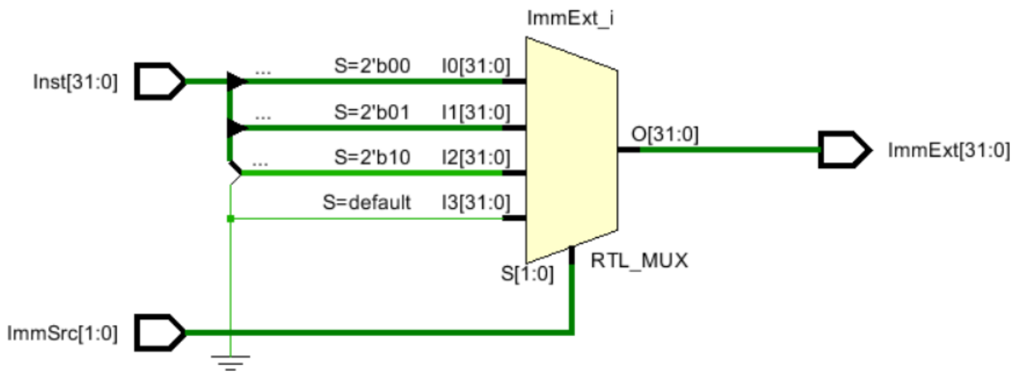
Sign extension simply copies the sign bit (most significant bit) of a short input (16 bits) into all the upper bits of the longer output (32 bits).



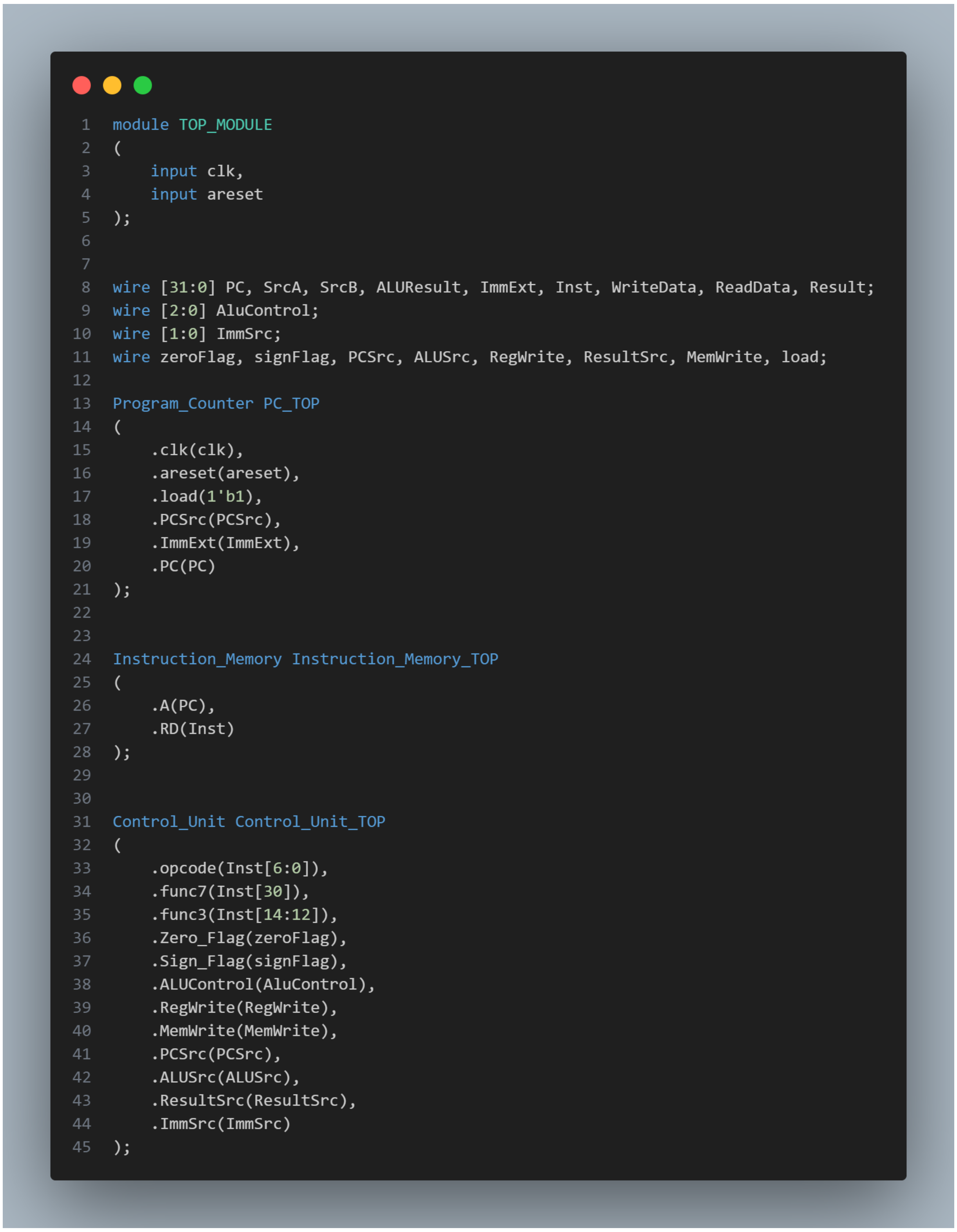
```verilog
module Sign_Extender
(
    input wire [31:0] Inst,
    input wire [1:0] ImmSrc,
    output reg [31:0] ImmExt
);

always @(*) begin
    case (ImmSrc)
        2'b00: ImmExt = {{20{Inst[31]}}, Inst[31:20]};
        2'b01: ImmExt = {{20{Inst[31]}}, Inst[31:25], Inst[11:7]};
        2'b10: ImmExt = {{20{Inst[31]}},Inst[7],Inst[30:25],Inst[11:8],1'b0};
        default: ImmExt = 32'b0;
    endcase
end
endmodule
```



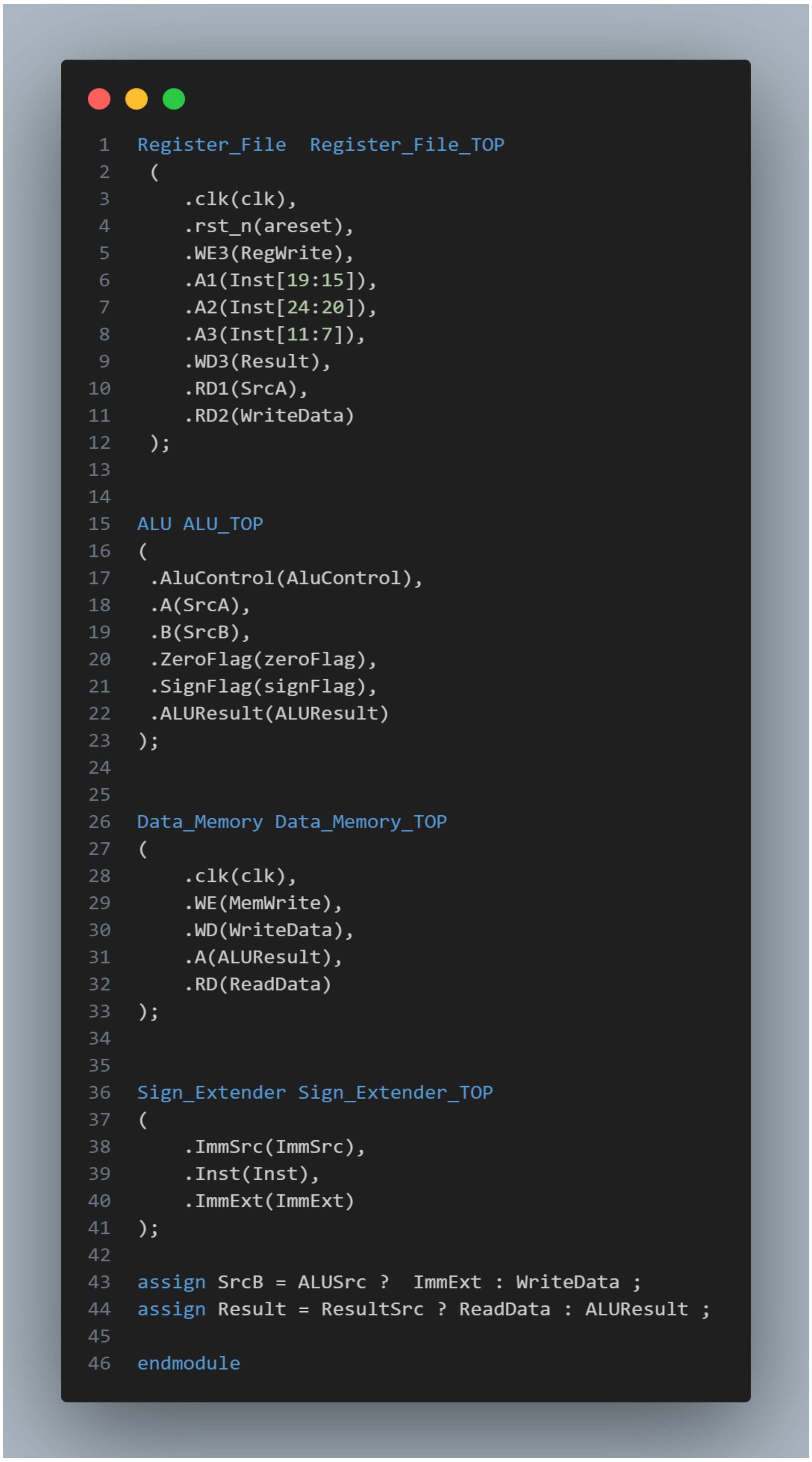
```verilog
module Sign_Extender
(
    input wire [31:0] Inst,
    input wire [1:0] ImmSrc,
    output reg [31:0] ImmExt
);

always @(*) begin
    case (ImmSrc)
        2'b00: ImmExt = {{20{Inst[31]}}, Inst[31:20]};
        2'b01: ImmExt = {{20{Inst[31]}}, Inst[31:25],
        2'b10: ImmExt = {{20{Inst[31]}},Inst[7],Inst[
        default: ImmExt = 32'b0;
    endcase
end
endmodule
```



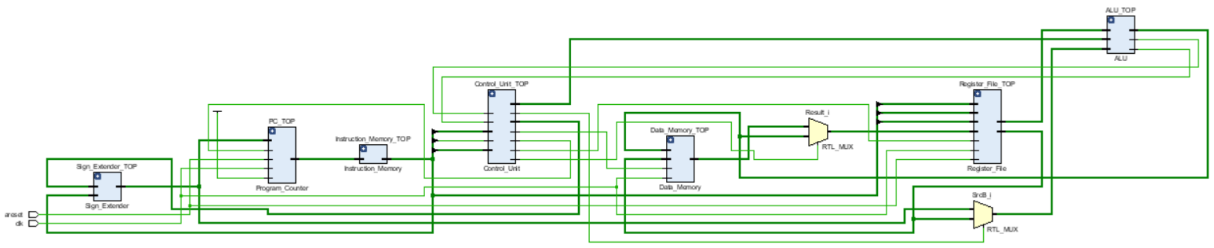
Top Module



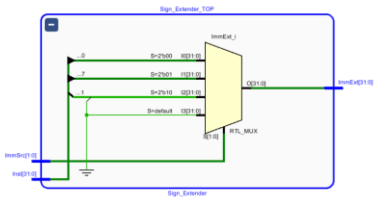
```verilog
module TOP_MODULE
(
    input clk,
    input areset
);


wire [31:0] PC, SrcA, SrcB, ALUResult, ImmExt, Inst, WriteData, ReadData, Result;
wire [2:0] AluControl;
wire [1:0] ImmSrc;
wire zeroFlag, signFlag, PCSrc, ALUSrc, RegWrite, ResultSrc, MemWrite, load;

Program_Counter PC_TOP
(
    .clk(clk),
    .areset(areset),
    .load(1'b1),
    .PCSrc(PCSrc),
    .ImmExt(ImmExt),
    .PC(PC)
);


Instruction_Memory Instruction_Memory_TOP
(
    .A(PC),
    .RD(Inst)
);


Control_Unit Control_Unit_TOP
(
    .opcode(Inst[6:0]),
    .func7(Inst[30]),
    .func3(Inst[14:12]),
    .Zero_Flag(zeroFlag),
    .Sign_Flag(signFlag),
    .ALUControl(AluControl),
    .RegWrite(RegWrite),
    .MemWrite(MemWrite),
    .PCSrc(PCSrc),
    .ALUSrc(ALUSrc),
    .ResultSrc(ResultSrc),
    .ImmSrc(ImmSrc)
);
```



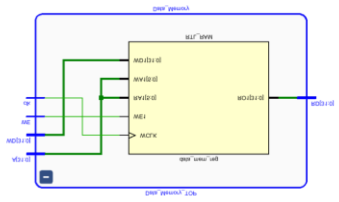
```
Register_File  Register_File_TOP
 (
     .clk(clk),
     .rst_n(areset),
     .WE3(RegWrite),
     .A1(Inst[19:15]),
     .A2(Inst[24:20]),
     .A3(Inst[11:7]),
     .WD3(Result),
     .RD1(SrcA),
     .RD2(WriteData)
 );


ALU ALU_TOP
(
 .AluControl(AluControl),
 .A(SrcA),
 .B(SrcB),
 .ZeroFlag(zeroFlag),
 .SignFlag(signFlag),
 .ALUResult(ALUResult)
);


Data_Memory Data_Memory_TOP
(
    .clk(clk),
    .WE(MemWrite),
    .WD(WriteData),
    .A(ALUResult),
    .RD(ReadData)
);


Sign_Extender Sign_Extender_TOP
(
    .ImmSrc(ImmSrc),
    .Inst(Inst),
    .ImmExt(ImmExt)
);

assign SrcB = ALUSrc ?  ImmExt : WriteData ;
assign Result = ResultSrc ? ReadData : ALUResult ;

endmodule
```

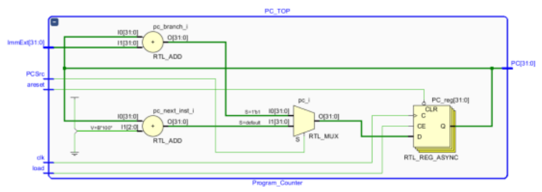


ALU_TOP
ALU
Control_Unit_TOP
Control_Unit
Register_File_TOP
Register_File
Result_i
RTL_MUX
Data_Memory_TOP
Data_Memory
PC_TOP
Program_Counter
Instruction_Memory_TOP
Instruction_Memory
Sign_Extender_TOP
Sign_Extender
SrcB_i
RTL_MUX
areset
clk

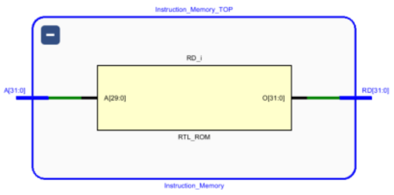


Sign_Extender_TOP
ImmExt_i
S=2'b00
S=2'b01
S=2'b10
S=default
I0[31:0]
I1[31:0]
I2[31:0]
I3[31:0]
O[31:0]
S[1:0]
RTL_MUX
ImmExt[31:0]
ImmSrc[1:0]
Inst[31:0]
Sign_Extender

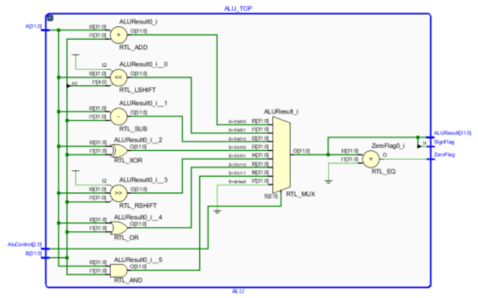




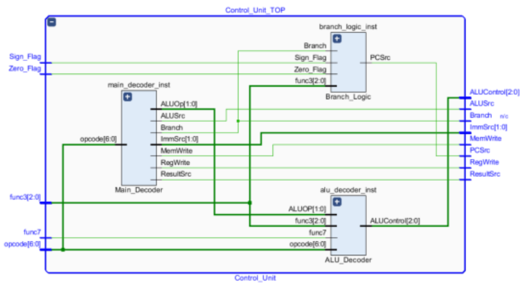
PC_TOP
pc_branch_i
I0[31:0]
I1[31:0]
O[31:0]
RTL_ADD
ImmExt[31:0]
PCSrc
areset
pc_next_inst_i
RTL_ADD
pc_i
RTL_MUX
PC_reg[31:0]
CLR
C
CE
D
Q
RTL_REG_ASYNC
PC[31:0]
clk
load
Program_Counter



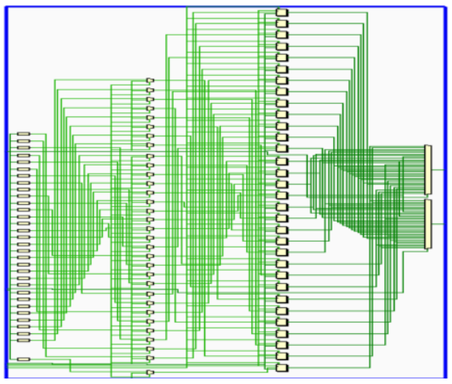
Instruction_Memory_TOP
RD_i
A[29:0]
O[31:0]
RTL_ROM
A[31:0]
RD[31:0]
Instruction_Memory



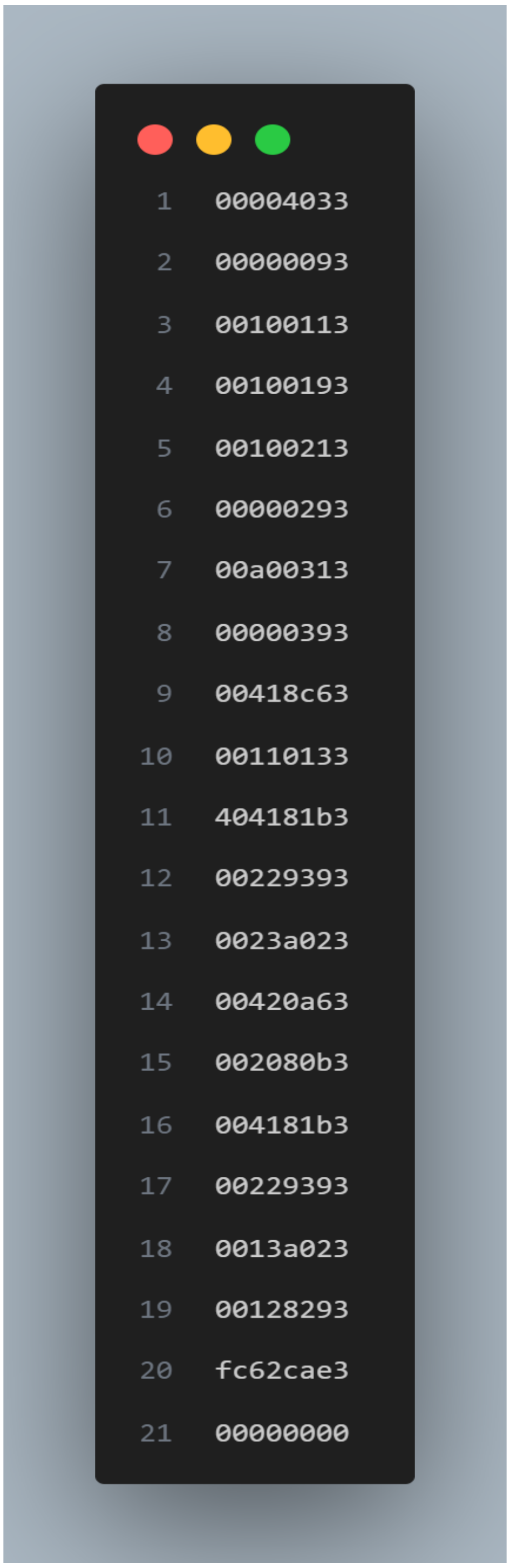
ALU_TOP
A[31:0]
ALUResult0_i
RTL_ADD
ALUResult0_i__0
RTL_LSHIFT
ALUResult0_i__1
RTL_SUB
ALUResult0_i__2
RTL_XOR
ALUResult0_i__3
RTL_RSHIFT
ALUResult0_i__4
RTL_OR
ALUResult0_i__5
RTL_AND
ALUResult_i
RTL_MUX
ZeroFlag0_i
RTL_EQ
ALUResult[31:0]
SignFlag
ZeroFlag
AluControl[2:0]
B[31:0]
ALU



Control_Unit_TOP
branch_logic_inst
Branch
Sign_Flag
Zero_Flag
func3[2:0]
PCSrc
Branch_Logic
main_decoder_inst
opcode[6:0]
ALUOp[1:0]
ALUSrc
Branch
ImmSrc[1:0]
MemWrite
RegWrite
ResultSrc
Main_Decoder
alu_decoder_inst
ALUOP[1:0]
func3[2:0]
func7
opcode[6:0]
ALUControl[2:0]
ALU_Decoder
ALUControl[2:0]
ALUSrc
Branch
ImmSrc[1:0]
MemWrite
PCSrc
RegWrite
ResultSrc
Sign_Flag
Zero_Flag
func3[2:0]
func7
opcode[6:0]
Control_Unit

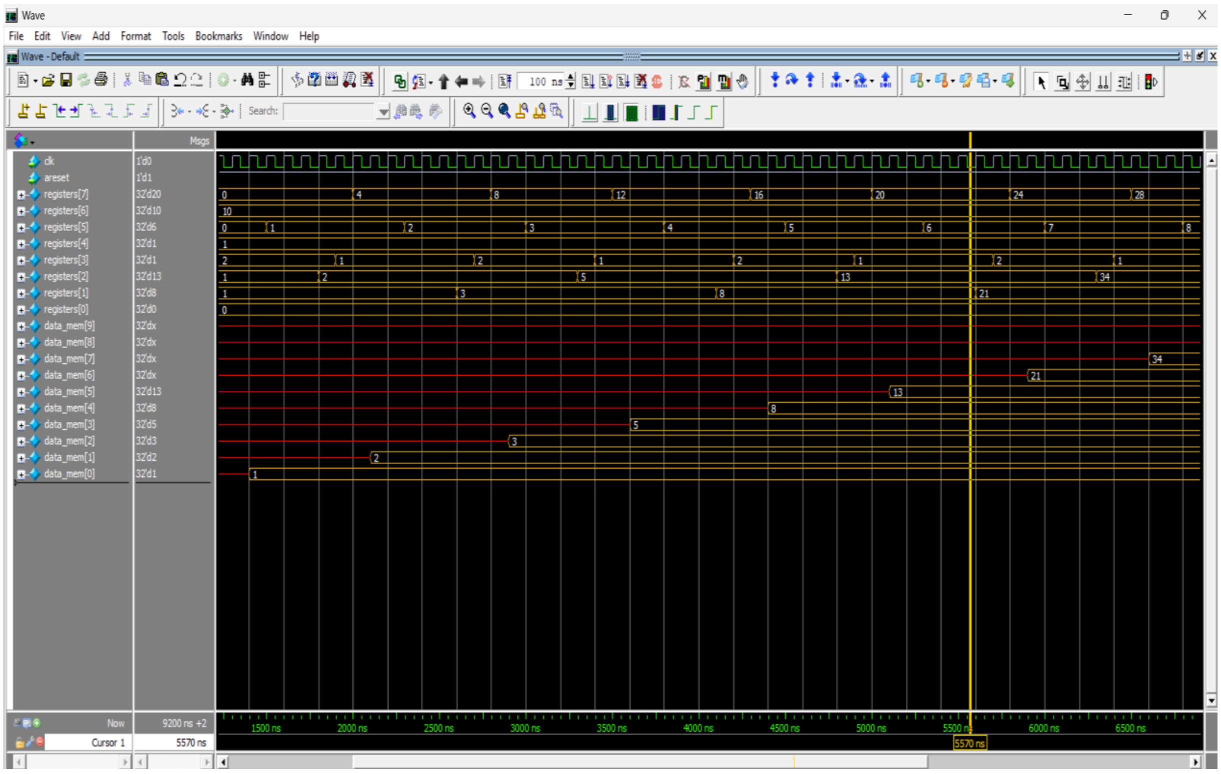


Program.text for verification to my design



```
00004033
00000093
00100113
00100193
00100213
00000293
00a00313
00000393
00418c63
00110133
404181b3
00229393
0023a023
00420a63
002080b3
004181b3
00229393
0013a023
00128293
fc62cae3
00000000
```

Simulation Results from QuestaSim



```
clk              1'd0
areset           1'd1
registers[7]     32'd20
registers[6]     32'd10
registers[5]     32'd6
registers[4]     32'd1
registers[3]     32'd1
registers[2]     32'd13
registers[1]     32'd8
registers[0]     32'd0
data_mem[9]      32'dx
data_mem[8]      32'dx
data_mem[7]      32'dx
data_mem[6]      32'dx
data_mem[5]      32'd13
data_mem[4]      32'd8
data_mem[3]      32'd5
data_mem[2]      32'd3
data_mem[1]      32'd2
data_mem[0]      32'd1

Now       9200 ns +2
Cursor 1  5570 ns
```

These results are expected as Fibomichi series