16-Bit RISC Processor

CPE 531: ISA Technical Report

Author:

Sanjoy Dev
(sd0396@uah.edu)

University of Alabama in Huntsville

Date: November 30, 2025

1 Introduction

This project is about designing an Instruction Set Architecture (ISA) for a new 16-bit microprocessor using RISC design principle. The primary goal of this design is to maintain low cost and minimal number of clock cycles per instruction (CPI). The ISA contains 16 unique instructions which are of fixed length. It reduces the size and complexity of the control unit. The data type used here is a 16 bit signed integer (2's complement) with no byte operations. And the ISA supports linear Addressing of 1K 16-bit word memory which is word addressable. Thus, these reduce the overall cost. The design uses a single cycle execution strategy where every instruction completes in one clock cycle. Moreover, the most important functionalities of a programming language like C is Arithmetic operations, storing and loading data from memory, conditions/loops and functions. All these functionalities are covered in this Instruction Set Architecture.

2 Instruction Set Architecture

2.1 User Programmable Regirster

The ISA has a total of 4 programmable registers. These 4 registers can be specified in the instruction formats using exactly 2 bits. The list of the programmable registers listed below:

Table 1: User Programmable Registers

Name	**Address**	**Use**	**Preserved across a call?**
$zero	00	The constant value 0	N/A
$a0	01	Arguments for function call	No
$s0	10	Saved temporary (callee-saved)	Yes
$ra	11	Return address	No

Here, $zero is hardwired to 0x0000 which can't be overwritten. $a0 is the Argument Register. This register is used to pass arguments to functions. $s0 is the Saved Temporary register which is used as a general purpose variable storage and arithmetic operations. Finally, $ra is the Return Address register which stores the next program counter address that is required for returning from functions.

The initial proposal specified 8 programmable registers. As suggested in the initial proposal the design was optimized to support 8 bit immediate values in order to implement **lui** (Load Upper Immediate) instruction. For that reason the registers address field was reduced to 2 bits each which resulted in using 4 programmable registers in the final design.

2.2 Instruction Formats

The ISA uses 3 types of Instruction format for all the operations. They are: Register Type, Immediate Type and Jump Type. The 16 bit command splits into smaller fields. The first one "opcode" is common for all and is of 4 bits [15-12]. The ISA uses in total 16 instructions. Using 4 bits opcode will help to specify all the 16 instructions by the CPU's control unit.

2.2.1 R-Type (Register Type) Format

This format is specified for register to register operations. The format is:

opcode [15:12]	rd [11:10]	rs [9:8]	rt [7:6]	Not used [5:0]
4 Bits	2 Bits	2 Bits	2 Bits	6 Bits

The R-type format is used to do arithmetic operations between two registers. The fields rd, rs and rt are the registers where rs and rt are the source registers and rd is the destination register. Here, I used 2 bits to uniquely identify the 4 programmable registers listed above. For example, register $zerois00, registers0$ is 10 etc. I didn't use shamt and func in this case. The operations can be easily identified by the CPU's control unit using opcode.

2.2.2 I-Type (Immediate Type) Format

This format is specified for operations involving registers and constants/offsets. The format is

opcode [15:12]	rt [11:10]	rs [9:8]	Immediate [7:0]
4 Bits	2 Bits	2 Bits	8 Bits

The I-type format is used for operations done using constant, memory access using load and store and conditional branches. In this case, rt is used as the destination register. And rs is used as the source register. For instructions like addi, lw, sw, lui and ori, rs works as a source or base. On the other hand, for instructions like beq, both rs and rt works as a source register. The immediate field holds the constant values. As it is of 8 bits, it can hold constant values ranging from -128 to +127.

2.2.3 J-Type (Jump Type) Format

This format is used for moving Program Counter to specific memory locations. The format is:

opcode [15:12]	Address [11:0]
4 Bits	12 Bits

The jump instructions are specified by the opcode. The j-type format is used for unconditional jumps across the memory which can't be done using I-type instructions. And the address field contains the jump address location. For a jump in a memory space of 1K words or 1024 words, a minimum of 10 bits is required. As we don't need any other fields for the jump instructions, we got 12 bits to specify the address.

2.3 Instruction Set List

The 16 instructions with their opcode, format and operation in Verilog is stated below:

Table 2: Instruction Set List

Name	**Mnemonic**	**opcode**	**Format**	**Operation in Verilog**
Add	add	0000	R	R[rd] = R[rs] + R[rt]
Subtract	sub	0001	R	R[rd] = R[rs] - R[rt]
AND	and	0010	R	R[rd] = R[rs] & R[rt]
OR	or	0011	R	R[rd] = R[rs] | R[rt]
Add Immediate	addi	0100	I	R[rt] = R[rs] + SignExtImm
Load Word	lw	0101	I	R[rt] = Memory[R[rs] + SignExtImm]
Store Word	sw	0110	I	Memory[R[rs] + SignExtImm] = R[rt]
Set Less Than	slt	0111	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0
Branch Equal	beq	1000	I	if (R[rs] == R[rt]) PC = PC + 1 + BranchAddr
Branch Not Equal	bne	1001	I	if (R[rs] != R[rt]) PC = PC + 1 + BranchAddr
Jump	j	1010	J	PC = JumpAddr
Jump and Link	jal	1011	J	R[3] = PC + 1; PC = JumpAddr
Jump Register	jr	1100	R	PC = R[rs]
Load Upper Imm.	lui	1101	I	R[rt] = {Imm, 8'b0}
Or Immediate	ori	1110	I	R[rt] = R[rs] | {8'b0, Imm}
Halt	halt	1111	-	Stops the clock

The below table shows the Assembly Language and corresponding Machine Language in Binary for each of the 16 instructions:

Table 3: Instruction Set Encoding

Mnemonic	Format	Assembly Language	Machine Language
add	R	add rd, rs, rt	`0000_rd_rs_rt_000000`
sub	R	sub rd, rs, rt	`0001_rd_rs_rt_000000`
and	R	and rd, rs, rt	`0010_rd_rs_rt_000000`
or	R	or rd, rs, rt	`0011_rd_rs_rt_000000`
addi	I	addi rt, rs, Immd	`0100_rt_rs_Immediate`
lw	I	lw rt, Immd(rs)	`0101_rt_rs_Immediate`
sw	I	sw rt, Immd(rs)	`0110_rt_rs_Immediate`
slt	R	slt rd, rs, rt	`0111_rd_rs_rt_000000`
beq	I	beq rs, rt, Label	`1000_rt_rs_Offset`
bne	I	bne rs, rt, Label	`1001_rt_rs_Offset`
j	J	j Target	`1010_TargetAddress`
jal	J	jal Target	`1011_TargetAddress`
jr	R	jr $ra	`1100_00_11_0000000000`
lui	I	lui rt, Immd	`1101_rt_00_Immediate`
ori	I	ori rt, rs, Immd	`1110_rt_rs_Immediate`
halt	-	halt	`1111_111111111111`

2.4 Instruction Justification

The most important functionalities of a programming language like C is Arithmetic operations, storing and loading data from memory, conditions/loops and functions. All these functionalities can be covered by the following instructions in the table.

2.4.1 Arithmetic Operations

For arithmetic operations we have both R-type (add, sub, and, or) and I-type (addi). The R-types are used for arithmetic operations between 2 registers. On the other hand, I-type can be used for arithmetic operations with a signed constant. Moreover, a key tradeoff was the inclusion of I-type instructions (lui and ori). These 2 instructions allow the processor to construct a 16 bit immediate in 2 cycles.

2.4.2 Conditional Statements

For conditional logic like (==, !=) can be implemented using the instructions bne and beq. Moreover, inclusion of slt instruction allows the other relational operations like (<,>, <=, >=) in C program.

2.4.3 Loop Operations

Loop operations can be done by combination of slt, bne and beq. Where slt can be used for condition checking and bne/beq for loop termination. Moreover, unconditional jump instruction j can be used to loop back to the start of the iteration.

2.4.4 Functions and Recursion

Function calls and recursion can be done with the support of J-Type(jal and jr) instructions. The jal instruction saves the (PC+1) address into the dedicated return address register. Moreover, the jr instruction allows the callee to return control to the caller.

2.4.5 Memory and System Control

Access to array elements or stack variables can be done using the I-type(lw and sw) instructions. Finally, the instruction halt is used to terminate the processor.

So, all of these instructions are enough to cover up all the functionalities of a programming language like C. And these instructions fulfill the goal of the design to be of low cost with minimal number of clock cycles per instruction.

3 C Language Construct for the Instructions

The last section described about the 16 different instructions and their justification to be appropriately used in C construct. This section will demonstrate the translation of standard C code in assembly language of the processor which will justify the instruction set.

3.1 Arithmetic and Logical Operations

Basic arithmetic and bit-wise operations map directly to R-Type instructions. Example:

C Code	Assembly Implementation
`a = b + c;`	`add $s0, $s0, $a0`
`a = b - c;`	`sub $s0, $s0, $a0`
`a = b & c;`	`and $s0, $s0, $a0`
`a = b | c;`	`or $s0, $s0, $a0`
`a = b + 5;`	`addi $s0, $s0, 5`
`a = b | 0x00FF;`	`ori $s0, $s0, 0x00FF`
`a = 0x12 «8;`	`lui $s0, 0x12`

3.2 Memory Access Instructions

Assuming `int arr[16];` where $s0 holds the base address of array.

Pseudocode	C Code	Assembly Implementation
`$a0 = Memory[$s0 + 4]`	`a[0] = arr[4];`	`lw $a0, 4($s0)`
`Memory[$s0 + 4] = $a0`	`arr[4] = a[0];`	`sw $a0, 4($s0`

3.3 Control Flow

Conditional structures can be implemented using beq (branch equal) to detect the condition and j (jump) to navigate.

C Language:

```
if (a == b) {
    x = 1;
}
else {
    x = 0;
}
```

Assembly Language:

```
  beq $a0, $s0, TRUE
  addi $s0, $zero, 0
  j FALSE
TRUE:
  addi $s0, $zero, 1
FALSE:
```

3.4 Loops

3.4.1 While Loop:

The `slt` instruction is used to check the condition of loop.

C Language:

```
int i = 0;
while (i < 10) {
    i++;
}
```

Assembly Language:

```
  addi $a0, $zero, 0
LOOP:
  addi $s0, $zero, 10
  slt $s0, $a0, $s0
  beq $s0, $zero, END
  addi $a0, $a0, 1
  j LOOP
END:
```

3.4.2 For Loop:

For C construct for loop we need initialization, condition and incrementing the iterator.

C Language:

```
for (i = 0; i < 5; i++) {
    s0 = s0 + a0;
}
```

Assembly Language:

```
  addi $a0, $zero, 0
LOOP:
  addi $s0, $zero, 5
  slt $s0, $a0, $s0
  beq $s0, $zero, END
  add $s0, $s0, $a0
  addi $a0, $a0, 1
  j LOOP
END:
```

3.5 Functions

Function calls and recursion can be done with the support of J-Type(jal and jr) instructions.

C Language:

```
void main() {
    func();
}
void func() {
    return;
}
```

Assembly Language:

```
main:
  jal func    //call func, save PC+1 to $ra
func:
  jr $ra      //return to caller
```

3.6 Other Relational Operators

The ISA also supports other relational operators like (>, >=, <=) using the slt instruction just by swaping the operands and using brnach equal operations.

Pseudocode	**C Code**	**Assembly Implementation**
`Check (b < a)`	`if (a > b);`	`slt $s0, $s0, $a0`
`Check !(a < b)`	`if (a >= b);`	`slt $s0, $a0, $s0` ; `beq $s0, $zero, L`
`Check !(b < a)`	`if (a <= b);`	`slt $s0, $s0, $a0` ; `beq $s0, $zero, L`

3.7 Program Termination

To stop execution we can use assembly language `halt` which corresponds to the C construct `return 0;`.

4 HDL Implementation and Design

This section describes about the hardware implementation of the 16 Bit RISC Processor. The design was implemented using SystemVerilog.

4.1 High Level Block Diagram

The processor utilizes a single cycle architecture. The entire fetch, decode, execute and write-back occurs in a single cycle. The high level block diagram below illustrates the data flow between the modules.

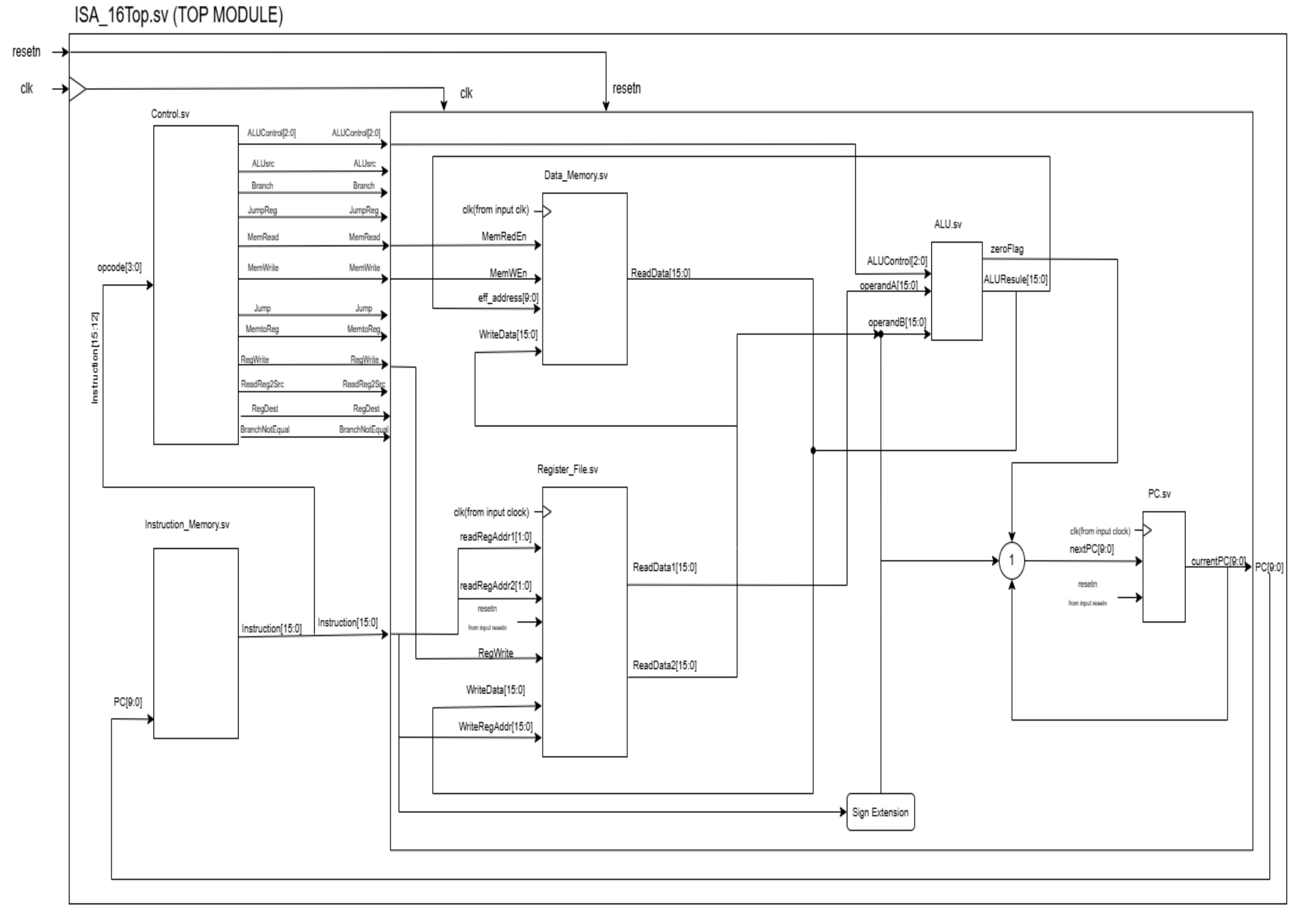


Figure 1: High-Level Block Diagram of 16-bit RISC Processor

In the above figure, the clock and reset was not directly connected to the modules as the design's connection got quite complicated. So, in bracket I wrote that the connection are gained from the clock and reset (active low) inputs. Moreover, the MUXs and adders are not displayed in the block diagram. The circular 1 in the diagram represents use of some specific MUXs. All the MUXs and Adders that are used in the design are described in Section 2, Subsection 2.9.

The design's top module is `ISA_16Top.sv`, which instantiates the modules `Datapath.sv`, `Control.sv` and `Instruction_Memory.sv`. The Datapath module then instantiates all the functional component modules like `Data_Memory.sv`, `Register_File.sv`, `ALU.sv` and `PC.sv`. It contains all the MUXs, adders and sign extentions that is needed to connect the modules.

On the other hand, the Control Module is a combinational logic that contains all the necessary control signals which it passes to the Datapath module. And the Instruction Memory module takes the current

Program Counter and outputs the instruction.

4.2 Module Justification and I/O Signals

4.2.1 PC.sv

This module is a 10-bit register which holds the memory address of the next instruction that is to be fetched. It updates the value of PC on every positive clock edge. As the ISA supports linear addressing of 1K 16-bit word memory. It is word addressable so 1024 words can be represented by the 10 bit width program counter.

4.2.2 Instruction_Memory.sv

This module stores the instruction of the processor which is stored in a hex file (instruction.hex). It is a 1024 word size and 16-bit width memory block that matches the ISA's word addressable specification. It takes the 10-bit address from the PC and outputs the 16-bit instruction stored at the specific location.

4.2.3 Register_File.sv

This module handles the registers of the processor. It holds the four programmable registers: `$zero`, `$a0`, `$s0`, and `$ra`. It reads the 2 source registers and writes to the destination register. For R-type and I-type instruction, the first source rs is read from `Instruction[9:8]`. The second source for R-type instruction rt is at `Instruction[7:6]`. And the destination for both type of instruction (rd for R-type and rt for I-type) is at `Instruction[11:10]`. Only for "jal" instruction the destination register is connected to the return address register (`$ra`) at binary value 11.

4.2.4 ALU.sv

This module contains a combinational block that performs all the arithmetic and logical operations according to the ALUControl control signal. It is used for the R-type instruction like add(addition), sub(subtract), and(bitwise and), or and slt(set less than). And for I-type instruction like addi(add immediate) and ori(or immediate). Moreover, it is also used for address calculation for I-Type instruction lw(load word) and sw(store word) where the ALU adds the base register to the sign extended immediate. Additionally, it is also used by the branch instructions beq (branch if equal) and bne(branch not equal) that checks the zero flag and decides the next PC.

4.2.5 Data_Memory.sv

This is the main memory of the processor. It is used by the lw(load word) and sw(store word) instruction of the ISA. For load word, it reads data from processor's main memory and for store word, it writes data to the main memory. The effective address to write and read data from the main memory is calculated using the ALU.

4.2.6 Datapath.sv

This is the most important module that assembles the processor. It instantiates all the modules explained above and connects them. It includes different MUXs and address to flow the data between the components. The MUXs are used according to the signal output from the Control module.

4.2.7 Control.sv

This module contains the control logic of the processor. It takes 4-bit opcode from the instruction module as an input. It has a combinational block that uses the opcode to provide the control signals for different types of instructions.

4.2.8 ISA_16Top.sv

It is the top level module of the entire project. It instantiates the modules Control, Datapath and Instruction_Memory and connect them to form the complete processor.

4.3 Module I/O, Control Logic and use of MUXs and Adders

Table 4: Module I/O Signals

Module Name	Inputs	Outputs
`PC.sv`	`clk`, `resetn`, `nextPC[9:0]`	`currentPC[9:0]`
`Instruction_Memory.sv`	`PC[9:0]`	`Instruction[15:0]`
`Register_File.sv`	`clk`, `resetn`, `RegWrite`, `readRegAddr1[1:0]`, `readRegAddr2[1:0]`, `writeRegAddr[1:0]`, `writeData[15:0]`	`readData1[15:0]`, `readData2[15:0]`
`ALU.sv`	`operandA[15:0]`, `operandB[15:0]`, `ALUControl[2:0]`	`AluResult[15:0]`, `zeroFlag`
`Data_Memory.sv`	`clk`, `MemWEn`, `MemRedEn`, `eff_address[9:0]`, `WriteData[15:0]`	`ReadData[15:0]`
`Datapath.sv`	`clk`, `resetn`, `Instruction[15:0]`, `ReadReg2Src`, `RegDest`, `RegWrite`, `ALUsrc`, `ALUControl[2:0]`, `MemRead`, `MemWrite`, `MemToReg`, `Branch`, `BranchNotEqual`, `Jump`, `JumpReg`	`PC[9:0]`, `zeroFlag`
`Control.sv`	`opcode[3:0]`	`ReadReg2Src`, `RegDest`, `RegWrite`, `ALUsrc`, `ALUControl[2:0]`, `MemRead`, `MemWrite`, `MemToReg`, `Branch`, `BranchNotEqual`, `Jump`, `JumpReg`
`ISA_16Top.sv`	`clk`, `resetn`	N/A

Table 5: Control Signal for the ISA

Instruction Type/Name	Example(s)	RegWrite	RegDest	ReadReg2Src	ALUsrc	MemRead	MemWrite	MemToReg	Branch	BranchNotEqual	Jump	JumpReg
R-Type	add, sub, and, or, slt	1	0	0	0	0	0	0	0	0	0	0
I-Type (Imm.)	addi, ori	1	0	0	1	0	0	0	0	0	0	0
Load Upper Immediate	lui	1	0	0	1	0	0	0	0	0	0	0
Load Word	lw	1	0	0	1	1	0	1	0	0	0	0
Store Word	sw	0	0	1	1	0	1	0	0	0	0	0
Branch Equal	beq	0	0	1	0	0	0	0	1	0	0	0
Branch Not Equal	bne	0	0	1	0	0	0	0	0	1	0	0
Jump	j	0	0	0	0	0	0	0	0	0	1	0
Jump & Link	jal	1	1	0	0	0	0	0	0	0	1	0
Jump Register	jr	0	0	0	0	0	0	0	0	0	1	1
Halt	halt	0	0	0	0	0	0	0	0	0	0	0

Table 6: Adders and MUXs

Logic Block	Purpose	Inputs	Control Signals
`PC+1 Adder (1)`	Calculates the address of the next instruction	`currentPC` (10-bit), `10'd1` (constant)	N/A
`Branch Target Adder`	Calculates the effective address for a branch instruction	`pcPlus1` (10-bit), `SignExtImm[9:0]` (10-bit offset)	N/A
`Sign Extender`	Converts the 8-bit immediate into a 16-bit signed value	`immediate[7:0]` (8-bit)	N/A
`Next PC MUX (1)`	Selects the next PC	`pcPlus1`, `branchTarget`, `jumpAddr[9:0]`, `readData1[9:0]`	`Jump`, `JumpReg`, `(Branch & zeroFlag)`, `(BranchNotEqual & !zeroFlag)`
`ALU Source MUX`	Selects the second operand for the ALU	`readData2[15:0]` , `SignExtImm` (16-bit)	`ALUsrc`
`ReadReg2Src MUX`	Selects the address for the Register File's second read port (`rt`)	`Instruction[7:6]` (R-type `rt`), `Instruction[11:10]` (I-type `rt`)	`ReadReg2Src`
`Register Dest. MUX`	Selects the address for the Register File's write port	`Instruction[11:10]` (for `rd/rt`), `2'b11` (for `$ra` on `jal`)	`RegDest`
`Writeback MUX`	Selects the data to be written back into the Register File	`AluResult` , `memReadData`, `pcPlus1`	`MemToReg`, `RegDest`

4.4 Design Improvements

Since the HDL Simulation Progress Report, some changes were done to solve the limitations of the earlier design. They are lister below:

- **Branch Logic:** In the initial design the bne (Branch Not Equal) instruction was incomplete. There was only single branch signal from control which couldn't distinguish between beq and bne. So, it was corrected by including a dedicated BranchNotEqual control signal. And the nextPC was updated according to the logic:

$$\text{BranchTaken} = (\text{Branch} \ \& \ \text{ZeroFlag}) \mid (\text{BranchNotEqual} \ \& \ !\text{ZeroFlag})$$

- **Jump and Link Logic:** In the initial design the jal (Jump and Link) instruction, the write-back MUX only allowed writing data from the ALU or Memory and didn't allow to save the (PC+1) value int the return address register $ra. It was solved by expanding the write-back MUX in the Datapath module. Now, a path was added to write the (PC+1) value directly to the writeData port of the Register File when RegDest signal is asserted during jal instruction. It then writes the (PC+1) value to the register index 11 ($ra).
- **Load Upper Immediate Logic:** The initial design was lacking the mechanism to shift 8 bit immediate data to the upper byte for lui (Load Upper Immediate) instruction. The ALU was updated to include another operation which takes the 8 bit immediate and shifts it's position to upper 8 bits [15:8] and concatenate with zeors in the lower 8 bits. Now, a 16 bit word can be loaded using the instructions lui and ori.

4.5 Design Tradeoff

The design tradeoffs are stated below:

- The design uses only 4 programmable registers and 1K 16-bit words of word addressable memory. It reduces the design complexity and use of memory. But it reduces the flexibility. It does not have temporary registers and the argument call function register is also limited to 1.
- The design uses the same bit indexes for destination register for both R-type and I-type instructions (11:10) which simplifies the write back process. But a MUX had to be implemented for jal (jump and link) instruction to store the return address at return address register.
- The design uses an opcode of 4 bits which reduces the number of instruction that restricts the use of some core instructions.

- The 16-bit processor reduces complexity and hardware cost but it limits the size of data values and addressable memory space.

5 Simulation Result

The design was verified using a SystemVerilog testbench (`tbISA_16Top.sv`) using ModelSim Altera Tool. The simulation verifies the processor's ability to execute arithmetic, memory access, branching and function calls in a single clock cycle per instruction.

5.1 Instructions in instruction.hex file

The given instructions in hexadecimal in the instruction.hex file are:

Table 7: Instruction Hex to Assembly Mapping

Instruction (hex)	Assembly Mnemonic
440a	addi $a0, $zero, 10
4814	addi $s0, $zero, 20
0980	add $s0, $a0, $s0
1a40	sub $s0, $s0, $a0
2580	and $a0, $a0, $s0
3580	or $a0, $a0, $s0
440f	addi $a0, $zero, 15
7580	slt $a0, $a0, $s0
d800	lui $s0, 0
ea32	ori $s0, $s0, 50
6804	sw $s0, 4($zero)
5404	lw $a0, 4($zero)
8901	beq $a0, $s0, 1
4801	addi $s0, $s0, 1
9101	bne $a0, $zero, 1
ffff	halt
a012	j 0x12
ffff	halt
b015	jal 0x15
ffff	halt
0000	nop
c300	jr $ra

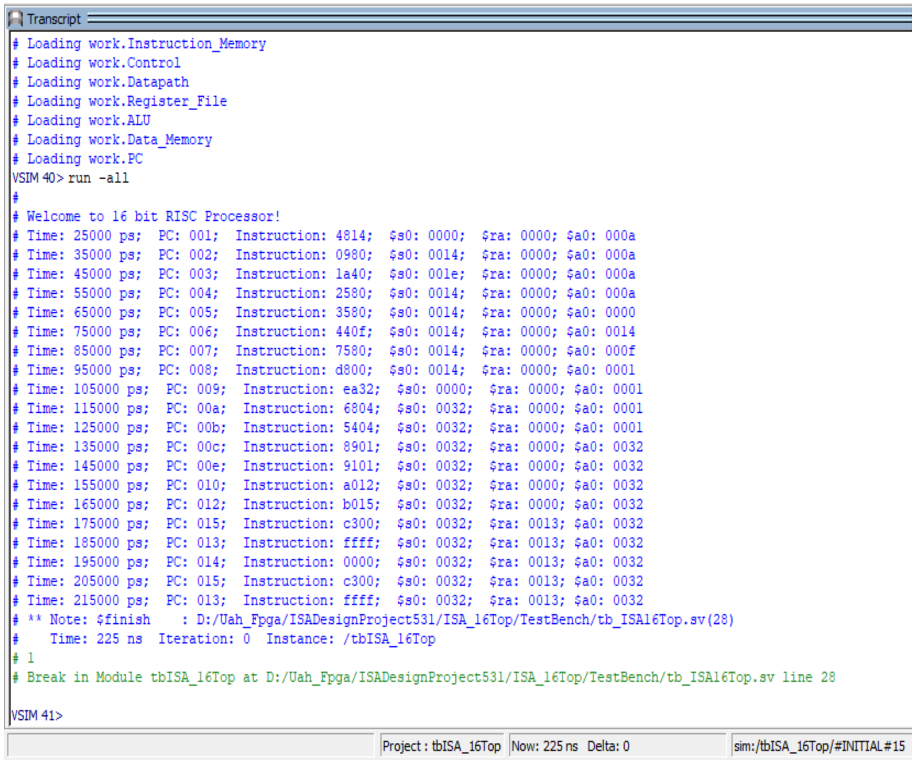
5.2 Simulation Trace Log

The table below shows the execution flow of the processor over 215 nanoseconds. It tracks the Program Counter, executed instruction and the values of the programmable registers except $zero register.

Table 8: Simulation Trace Log

Assembly Mnemonic	Time (ns)	PC (Hex)	Instruction (Hex)	$s0 (Value)	$ra (Value)	$a0 (Value)
addi $a0, $zero, 10	25	01	440a	0000	0000	000a
addi $s0, $zero, 20	25	01	4814	0000	0000	000a
add $s0, $a0, $s0	35	02	0980	0014	0000	000a
sub $s0, $s0, $a0	45	03	1a40	001e	0000	000a
and $a0, $a0, $s0	55	04	2580	0014	0000	000a
or $a0, $a0, $s0	65	05	3580	0014	0000	0000
addi $a0, $zero, 15	75	06	440f	0014	0000	000f
slt $s0, $a0, $s0	85	07	7580	0014	0000	000f
lui $s0, 0	95	08	d800	0014	0000	0001
ori $s0, $s0, 50	105	09	ea32	0000	0000	0001
sw $s0, 4($zero)	115	0a	6804	0032	0000	0001
lw $a0, 4($zero)	125	0b	5404	0032	0000	0001
beq $a0, $s0, 1	135	0c	8901	0032	0000	0032
bne $a0, $zero, 1	145	0e	9101	0032	0000	0032
j 0x12	155	10	a012	0032	0000	0032
jal 0x15	165	12	b015	0032	0013	0032
jr $ra	175	15	c300	0032	0013	0032
halt	185	13	ffff	0032	0013	0032
NOP	195	14	0000	0032	0013	0032
jr $ra	205	15	c300	0032	0013	0032
halt	215	13	ffff	0032	0013	0032

5.3 Simulation Output in ModelSim Altera



```
# Loading work.Instruction_Memory
# Loading work.Control
# Loading work.Datapath
# Loading work.Register_File
# Loading work.ALU
# Loading work.Data_Memory
# Loading work.PC
VSIM 40> run -all
#
# Welcome to 16 bit RISC Processor!
# Time: 25000 ps;  PC: 001;  Instruction: 4814;  $s0: 0000;  $ra: 0000; $a0: 000a
# Time: 35000 ps;  PC: 002;  Instruction: 0980;  $s0: 0014;  $ra: 0000; $a0: 000a
# Time: 45000 ps;  PC: 003;  Instruction: 1a40;  $s0: 001e;  $ra: 0000; $a0: 000a
# Time: 55000 ps;  PC: 004;  Instruction: 2580;  $s0: 0014;  $ra: 0000; $a0: 000a
# Time: 65000 ps;  PC: 005;  Instruction: 3580;  $s0: 0014;  $ra: 0000; $a0: 0000
# Time: 75000 ps;  PC: 006;  Instruction: 440f;  $s0: 0014;  $ra: 0000; $a0: 0014
# Time: 85000 ps;  PC: 007;  Instruction: 7580;  $s0: 0014;  $ra: 0000; $a0: 000f
# Time: 95000 ps;  PC: 008;  Instruction: d800;  $s0: 0014;  $ra: 0000; $a0: 0001
# Time: 105000 ps;  PC: 009;  Instruction: ea32;  $s0: 0000;  $ra: 0000; $a0: 0001
# Time: 115000 ps;  PC: 00a;  Instruction: 6804;  $s0: 0032;  $ra: 0000; $a0: 0001
# Time: 125000 ps;  PC: 00b;  Instruction: 5404;  $s0: 0032;  $ra: 0000; $a0: 0001
# Time: 135000 ps;  PC: 00c;  Instruction: 8901;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 145000 ps;  PC: 00e;  Instruction: 9101;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 155000 ps;  PC: 010;  Instruction: a012;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 165000 ps;  PC: 012;  Instruction: b015;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 175000 ps;  PC: 015;  Instruction: c300;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 185000 ps;  PC: 013;  Instruction: ffff;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 195000 ps;  PC: 014;  Instruction: 0000;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 205000 ps;  PC: 015;  Instruction: c300;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 215000 ps;  PC: 013;  Instruction: ffff;  $s0: 0032;  $ra: 0013; $a0: 0032
# ** Note: $finish    : D:/Uah_Fpga/ISADesignProject531/ISA_16Top/TestBench/tb_ISA16Top.sv(28)
#    Time: 225 ns  Iteration: 0  Instance: /tbISA_16Top
# 1
# Break in Module tbISA_16Top at D:/Uah_Fpga/ISADesignProject531/ISA_16Top/TestBench/tb_ISA16Top.sv line 28

VSIM 41>
```

Figure 2: ModelSim Altera Output Log

5.4 C construct of the Instructions

```
void main() {
    int a = 10;             // addi $a0, $zero, 10
    int b = 20;             // addi $s0, $zero, 20
    b = a + b;              // add (b becomes 30)
    b = b - a;              // sub (b returns to 20)
    a = a & b;              // and (a becomes 0)
    a = a | b;              // or  (a becomes 20)
    a = 15;                 // addi $a0, $zero, 15
    a = (a < b) ? 1 : 0;    // slt $a0, $a0, $s0
    b = 50;                 // lui + ori loading 16 bit operand
    memory[4] = b;          // sw $s0, 4($zero)
    a = memory[4];          // lw $a0, 4($zero)

    // beq $a0, $s0, 1
    if (a != b) {
        b = b + 1;          //addi $s0, $s0, 1 (skipped)
    }
    // bne $a0, $zero, 1
    if (a == 0) {
        exit();             //halt
    }

    goto jump_target;       // j 0x12


jump_target:
    function_call();        // jal 0x15
    return;                 // halt (PC = 0x13)
}

void function_call() {
    return;                 // jr $ra
}
```

5.5 Result Analysis

The simulation trace log and the output log confirms the successful operation of all major architectural features. The Values here are represented in Hexadecimal.

- **Arithmetic Operations:** At 35 ns the add instruction sums the values of $s0 (20, 14 in Hex) and $a (10, 000a in hex) to produce 30 (0x001e) in register $s0. After that, at 45 ns the sub instruction subtracts $a0(10) from $s0(30) and returns the value 20 in register $s0. This confirms that the ALU control signals and the Register File read/write ports are working correctly.
- **Memory Access:** The memory access are done by the lw(load word) and sw(store word) instructions. At 115 ns, the processor stores the value 50 (0x32) into the memory address 4 using the store instruction. After that, at 125 ns using the load word instruction the processor retreives the data from address 4 and loads it into the register $a0. It can be observed that the register $a0 updates to 0x0032 which confirms that the Data Memory is correctly mapped and the address calculation (Base + offset) is working correctly. Moreover, it also confirms that the MemWrite and MemRead signals are also correctly working.
- **Control Flow and Branching:** At 135 ns the beq instruction compares $a0(50) with $s0(50). As they are equal the ZeroFlag is asserted and the PC jumps from 00c to 00e skipping the next PC (00d). After that, at 145 ns, the bne instruction compares $a0 with $zero. As they are not equal it triggers another

branch. As a result it can be said that the Branch and BranchNotEqual control signals are also correcctly working here.

- **Function Call:** At time 165 ns the jal instruction performs 2 simultaneous operations. First, it updates the PC to the target function address (015). Second, it writes the next PC (PC+1 = 013) to the return address register $ra. Finally the jr instruction at 175 ns reads this value and returns the PC to 013. This verifies that the processor supports nested fuction calls and recursion.

6 FPGA Implementation and Demonstration

In order to validate the functionality processor the design was synthesized in Intel Quartus Prime Lite and it was implemented in DE2-115 FPGA board. Some modifications were done to the system verilog files that were provided to be simulated in ModelSim Altera. The modified codes are separately provided in the Appendix.

6.1 Modifications for FPGA Implementation

A hardware top module "`ISA_16Top_DE2115.sv`" was created. It has interface with the physical FPGA inputs and outputs like CLOCK_50, KEY, HEX0–HEX7, and LEDR. The original processor module (ISA_16Top) was instantiated in this top module.

On the FPGA we replaced the continuous clock that we used in simulation with a button triggered clock with the help of push button keys. This was done to verify the result of the FPGA and match it with the simulation result on each clock pulse.

6.2 Demonstration

The processor's execution is visualized in real-time using the FPGA's mapped peripherals. The pushbutton KEY1 as a manual clock input. Each press to the KEY will provide a clock pulse. The current Program Counter (PC) value is displayed on the seven segment HEX displays, HEX0-HEX2 in order to track the current PC on each clock pulse. At the same time, HEX4-HEX7 is used to track the instruction in hexadecimal on each clock pulse.

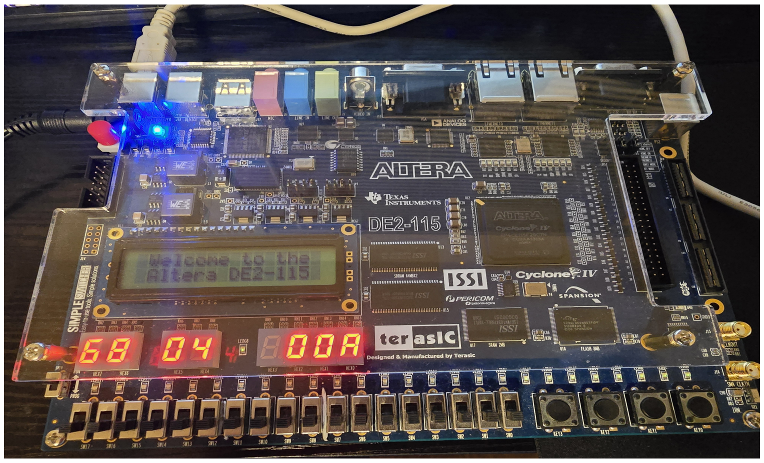
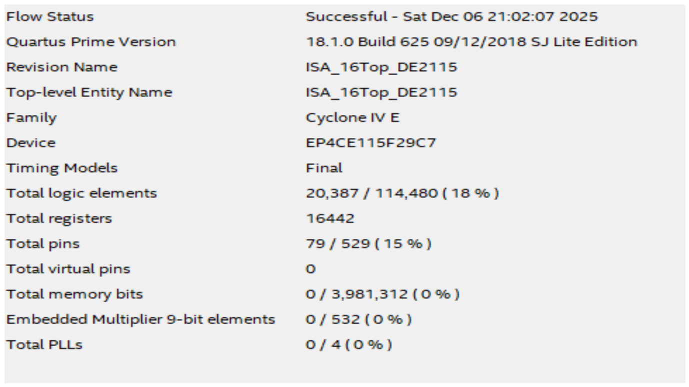


Figure 3: A demonstration picture, Instruction: 6804, PC: 00a

A demonstration video is provided with the ISA final report which verifies that the FPGA demonstration matches the result of the simulation that was done in ModelSim Altera.

6.3 Synthesis Flow Summary

The design was succesfully synthesized and compiled in Quartus Prime Lite using the Cyclone IVE EP4CE115F29C7 device. The flow summary is:



Flow Status	Successful - Sat Dec 06 21:02:07 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	ISA_16Top_DE2115
Top-level Entity Name	ISA_16Top_DE2115
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	20,387 / 114,480 (18 %)
Total registers	16442
Total pins	79 / 529 (15 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 4: Synthesis Flow Summary in Quartus Prime Lite

The logical elements utilized was 18% which is quite expected for a single cycle 16 bit processor having ALU, register file, program counter, control unit, datapath multiplexers, and extensive I/O mapping for displays. Moreover the amount of registers used was around 16,442. And, the memory bits utilization was 0%. This is because the project did not use any block RAM interface. The instruction memory and the data memory was synthesized as distributed logic.

6.4 Netlist: RTL View

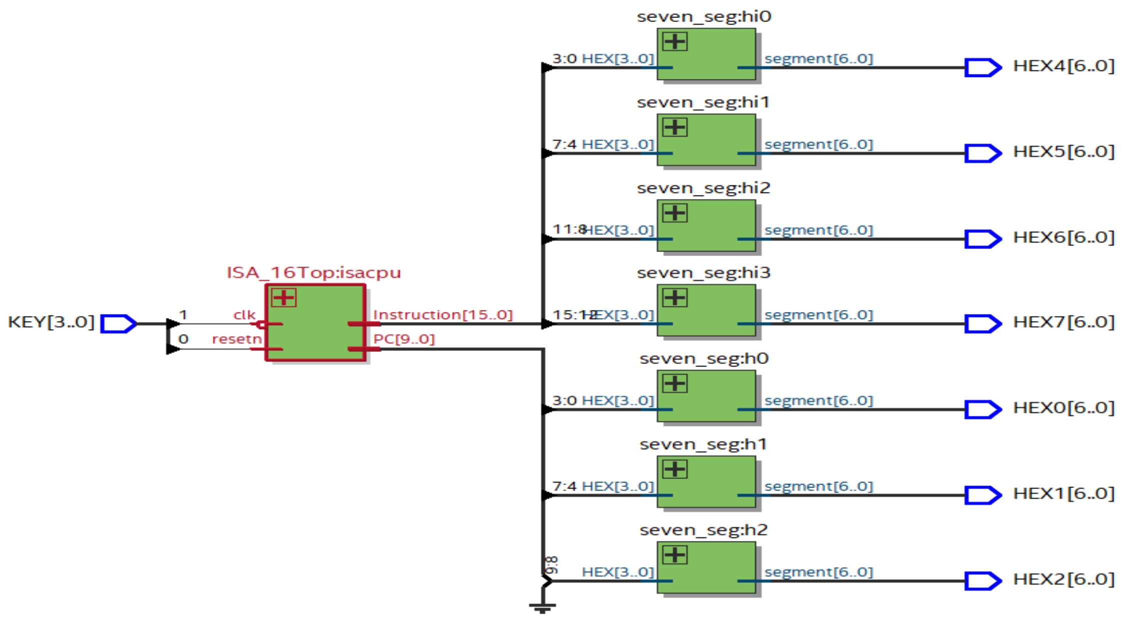


Figure 5: RTL View of the FPGA Implementation

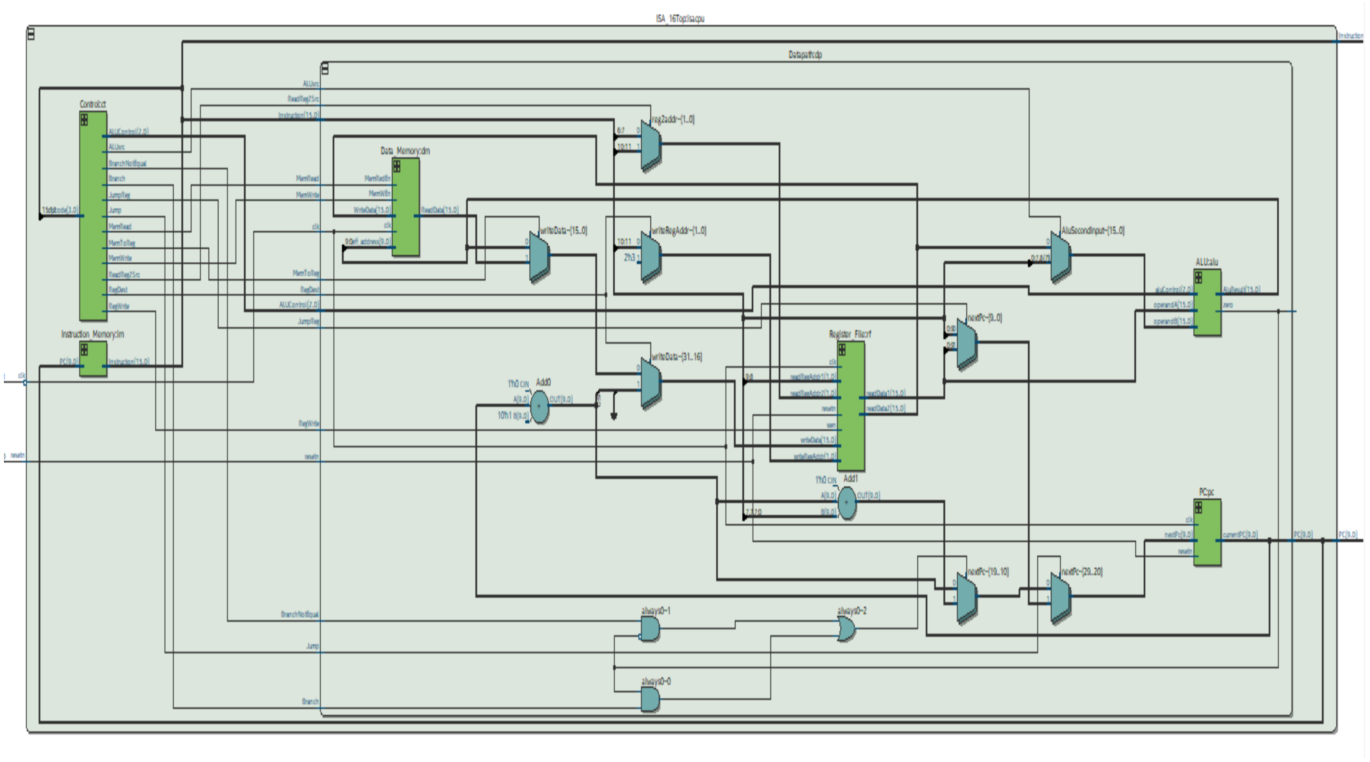


Figure 6: RTL View of Datapath and Top Module

7 Conclusion

This project does a complete design of a 16-bit single cycle RISC processor. The ISA has 4 programmable registers, 16-bit fixed-length instructions and a 1K word, word addressable memory. It has 16 instructions that fulfils the major C constructs including arithmetic, logical, memory access, branching, and function calls.

By optimizing the register file to four programmable registers, the design successfully balanced the trade-off between register availability by allowing 8 bit immediate field that simplifies constant loading. This project demonstrates a fully functional, low cost RISC processor which was verified with the help of testbench on ModelSim Altera. Moreover, the project was also demonstrated with the help of DE2-115 FPGA.

A Appendix: Code

A.1 PC.sv

```
module PC (
 input logic clk,      //clock
 input logic resetn,     //active low reset
 input logic [9:0] nextPc,   //next PC address
 output logic [9:0] currentPC //current PC address
);

 always_ff @(posedge clk, negedge resetn) begin
  if(!resetn) currentPC <= 10'h000; //PC reset to 0
  else currentPC <= nextPc;    // go to next PC on posedge
 end
```

```
endmodule
```

A.2 Instruction_Memory.sv

```
module Instruction_Memory (
 input logic [9:0] PC, //10 bit Program Counter
 output logic [15:0] Instruction //16 bits instruction output

);

 //instruction memory for 1024 words each of them of 16 bits
 logic [15:0] instruction_Mem [0:1023];

 //we will load the instruction bits through instruction.hex file and put it in instruction memory
 //D:\Uah_Fpga\ISADesignProject531\try2\ISA_16Top
 initial begin
  $readmemh("D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top/instruction.hex", instruction_Mem);
 end

 assign Instruction = instruction_Mem[PC]; //read 16 bits instruction

endmodule
```

A.3 Register_File.sv

```
module Register_File (
 input logic clk,      //clock
 input logic resetn,     //active low reset
 input logic wen,      //write enable
 input logic [1:0] readRegAddr1,  //rs, first source register address
 input logic [1:0] readRegAddr2, //rt, second source register address
 input logic [1:0] writeRegAddr,  //rt/rd, destination register
 input logic [15:0] writeData, //write data
 output logic [15:0] readData1, //read data 1
 output logic [15:0] readData2  //read data 2

);


 //internal 16 bit register 4*16
 logic [15:0] register [0:3];

 //if the read address is 0 ($zero) then output is 16bit 0
 assign readData1 = (readRegAddr1 == 2'b00)? 16'h0000: register[readRegAddr1];
 assign readData2 = (readRegAddr2 == 2'b00)? 16'h0000: register[readRegAddr2];

 //write operation with async reset
 always_ff @(posedge clk, negedge resetn) begin
  if(!resetn) begin
```

```
   register[0] <= 16'h0000; //$zero
   register[1] <= 16'h0000; //a0
   register[2] <= 16'h0000; //s0
   register[3] <= 16'h0000; //ra
  end

  //shouldn't write for zero. So writeRegAddr != 0
  else if (wen && writeRegAddr != 2'b00) begin
   register[writeRegAddr] <= writeData;
  end

 end

endmodule
```

A.4 ALU.sv

```
module ALU (
 input logic [15:0] operandA, //from readData1
 input logic [15:0] operandB, //from readData2
 input logic [2:0] aluControl, //defines what operation
 output logic [15:0] AluResult, //Alu Result
 output logic zero     //zero flag for branch operation


);


 always_comb begin
  case (aluControl)
   3'b000: AluResult = operandA + operandB; //add
   3'b001: AluResult = operandA - operandB; //sub
   3'b010: AluResult = operandA & operandB; //and
   3'b011: AluResult = operandA | operandB; //or
   //SLT will first check if A<B. If true resilt will be 1. Else result will be 0
   3'b100: AluResult = ($signed(operandA) < $signed(operandB)? 16'h0001:16'h0000); //set less than. Ins
   3'b101: AluResult = {operandB[7:0], 8'h00}; //LUI load upper 8 bits immediate
   default: AluResult = 16'h0000;
  endcase
 end

 assign zero = (AluResult == 16'h0000); //when aluResult is 0, we will get zero flag 1

endmodule
```

A.5 Data_Memory.sv

```
module Data_Memory(
 input logic clk,       //clock
 input logic MemWEn,     //write enable
 input logic MemRedEn,    //read enable
 input logic [9:0] eff_address,  //effictive address from alu
 input logic [15:0] WriteData,  //16 bit write data
 output logic [15:0] ReadData  //16 bit read data

);

 //data memory for 1024 words each of them of 16 bits
 logic [15:0] data_Mem [0:1023];


 //synchronous write at MemWriteEn enable
 always_ff @(posedge clk) begin
  if(MemWEn) data_Mem [eff_address] <= WriteData;
 end


 //prevents stale data from reading when not enable
 assign ReadData = (MemRedEn)? data_Mem[eff_address] : 16'h0000; //read 16 bits memory

endmodule
```

A.6 Datapath.sv

```
module Datapath (
 input logic clk,       //clock
 input logic resetn,       //active low reset
 input logic [15:0] Instruction,  //16 bit Instruction
 input logic ReadReg2Src,     //ReadReg2Src signal from control
 input logic RegDest,      //RegDest signal from control
 input logic RegWrite,     //RegWrite signal from control
 input logic ALUsrc,       //ALUsrc signal from control
 input logic [2:0] ALUControl,   //ALUControl signal from control
 input logic MemRead,      //MemRead signal from control
 input logic MemWrite,      //MemWrite signal from control
 input logic MemToReg,      //MemToReg signal from control
 input logic Branch,       //Branch signal from control
 input logic BranchNotEqual,     //Branch Not Equal signal from control
 input logic Jump,       //Jump signal from control
 input logic JumpReg,      //JumpReg signal from control
 output logic [9:0] PC,      //program counter output
 output logic zeroFlag     //zero flag from ALU

);


 //specify common instruction fields internal according to the format
 logic [3:0] opcode;
 logic [1:0] rs, rt, rd;
 logic [7:0] immidiate;
 logic [11:0] jumpAddr;

 //split and assign the 16 bit instruction according to format (all)
```

```
assign opcode = Instruction[15:12];
assign rd = Instruction[11:10];
assign rs = Instruction[9:8];
assign rt = Instruction[7:6];
assign immidiate = Instruction[7:0];
assign jumpAddr = Instruction[11:0];

//Register File internal
logic [1:0] writeRegAddr;  //rt/rd, destination register
logic [15:0] writeData;  //write data
logic [15:0] readData1;  //read data 1
logic [15:0] readData2;  //read data 2
logic [1:0] reg2addr;   //second reg address for the Register file


//ALU internal
logic [15:0] AluSecondInput; //first input will be readData1, but for some cases we decide the input fo
logic [15:0] AluResult; //Alu output result

//data memory internal
logic [15:0] memReadData; //output read data



//sign extension of the 8 bit immidiate value
logic [15:0] SignExtImm;
//we will extend according to the sign on the MSB
assign SignExtImm = {{8{immidiate[7]}}, immidiate}; //Bit swizling and concatening

//next PC logic
logic [9:0] currentPC, nextPc, pcPlus1, branchTarget;

assign pcPlus1 = currentPC + 10'd1; //word addresable, goes to immidiate next instruction
assign branchTarget = pcPlus1 + SignExtImm[9:0];  //branch target address for branch instructions

always_comb begin
 if(Jump) begin
  if(JumpReg) nextPc = readData1[9:0]; //for JR (Jump Register)
  else nextPc = jumpAddr[9:0]; //for J/JAL (Jump/ Jump and Link)
 end
 else if ((Branch & zeroFlag) || (BranchNotEqual & ~zeroFlag)) nextPc = branchTarget; //When we get Bra
 else nextPc = pcPlus1;  //else we just go to the next instruction
end




//Now we are goint to connect the datapath
//we will need couple of MUX for the flow according to signals

//The behaivior of destination register for jump and link will be different
//But for both R type destination register is rd(Instruction[11:10]), and for I type rt(Instruction[11
//(MUX:RegDest) for JAL, writeRegAddr = 2'b11 represents $ra, saves the return address
assign writeRegAddr = RegDest? 2'b11 : Instruction[11:10];
```

```
//(MUX:RegDest) and (MUX:MemToReg)
//If RegDest is 1(JAL) we write pcPlus1. We concatenate 7 bit 0 at front with pcPlus1
//Else if MemToReg is 1 we write Memory data
//Else we write Alu Result
assign writeData = RegDest?  {6'b0, pcPlus1} : (MemToReg? memReadData : AluResult);


//(MUX:ReadReg2Src) for selecting rt (R type/I type) rt, for R type is [7:6], for I type [11:10]
assign reg2addr = ReadReg2Src ? Instruction[11:10]:Instruction[7:6]; //When 0 it is R type, and For 1


//Instantiating the Register File
Register_File rf (
 .clk(clk),      //clock
 .resetn(resetn),     //active low reset
 .wen(RegWrite),      //write enable
 .readRegAddr1(rs),  //rs, first source register address always [9:8]
 .readRegAddr2(reg2addr), //rt, for R type is [7:6], for I type [11:10]
 .writeRegAddr(writeRegAddr),  //rt/rd, destination register
 .writeData(writeData), //write data
 .readData1(readData1), //read data 1
 .readData2(readData2)  //read data 2


);


//(MUX:ALUsrc)  to choose signExtImm or readData2
assign AluSecondInput = ALUsrc ? SignExtImm:readData2;

//Instantiating the ALU
ALU alu(
 .operandA(readData1), //from readData1
 .operandB(AluSecondInput), //from readData2
 .aluControl(ALUControl), //defines what operation
 .AluResult(AluResult), //Alu Result
 .zero(zeroFlag)  //zero flag for branch operation


);

//Instantiating the Data_Memory
Data_Memory dm (
 .clk(clk),       //clock
 .MemWEn(MemWrite),     //write enable
 .MemRedEn(MemRead),    //read enable
 .eff_address(AluResult[9:0]),  //effictive address from alu
 .WriteData(readData2),  //16 bit write data from Register file Read Data 2
 .ReadData(memReadData)  //16 bit read data from memory

);
```

```
 //Instantiating the PC (Program Counter)
 PC pc(
  .clk(clk),      //clock
  .resetn(resetn),     //active low reset
  .nextPc(nextPc),   //next PC address
  .currentPC(currentPC) //current PC address
 );

 assign PC = currentPC;

endmodule
```

A.7 Control.sv

```
module Control(
 input logic [3:0] opcode,     //4 bits opcode
 output logic ReadReg2Src,     //ReadReg2Src signal from control
 output logic RegDest,      //RegDest signal from control
 output logic RegWrite,     //RegWrite signal from control
 output logic ALUsrc,       //ALUsrc signal from control
 output logic [2:0] ALUControl,   //ALUControl signal from control
 output logic MemRead,      //MemRead signal from control
 output logic MemWrite,      //MemWrite signal from control
 output logic MemToReg,      //MemToReg signal from control
 output logic Branch,       //Branch signal from control
 output logic BranchNotEqual,  //Branch Not Equal signal from control
 output logic Jump,       //Jump signal from control
 output logic JumpReg      //JumpReg signal from control

);

 //Here, we will decide what control signals will be put for each of the operations

 always_comb begin
  //first we put the default values to 0
  ReadReg2Src = 1'b0;
  RegDest = 1'b0;
  RegWrite = 1'b0;
  ALUsrc = 1'b0;
  ALUControl = 3'b000;
  MemRead = 1'b0;
  MemWrite = 1'b0;
  MemToReg = 1'b0;
  Branch = 1'b0;
  BranchNotEqual = 1'b0;
  Jump = 1'b0;
  JumpReg = 1'b0;

  //Now we will put the signals according to the opcode using a case statement
  case(opcode)

   //R Type Instruction, for all of them data will be written in the Register File. So RegWrite = 1 for
```

```
4'b0000: begin
 RegWrite = 1'b1;
 ALUControl = 3'b000; //Add
end
4'b0001: begin
 RegWrite= 1'b1;
 ALUControl = 3'b001; //Sub
end
4'b0010: begin
 RegWrite = 1'b1;
 ALUControl = 3'b010; //And
end
4'b0011: begin
 RegWrite = 1'b1;
 ALUControl = 3'b011; //Or
end
4'b0111: begin
 RegWrite = 1'b1;
 ALUControl = 3'b100; //SLT
end


//I type Instruction
4'b0100: begin
 ALUsrc = 1'b1;   //select second input of ALu (signExtImm)
 RegWrite = 1'b1;
 ALUControl = 3'b000; //Addi
end
//For LW/Load Word, we don't need readData2 so ReadReg2Src not needed here.
4'b0101: begin
 ALUsrc = 1'b1;   //select second input of ALu (signExtImm)
 MemRead = 1'b1;   //for LW, we read data from memory
 RegWrite = 1'b1;   //Data written in Reg File
 MemToReg = 1'b1;   //the read data form memory is then Written back
 ALUControl = 3'b000; //we use add operation to find the effective address (base addr + signExtImm)
end

//For SW/Store Word
4'b0110: begin
 ALUsrc = 1'b1;   //select second input of ALu (signExtImm)
 MemWrite = 1'b1;   //Data Written to memory
 ALUControl = 3'b000; //we use add operation to find the effective address (base addr + signExtImm)
 ReadReg2Src = 1'b1;  //we select rt's instuction address

end
//For BEQ, Branch Instruction
4'b1000: begin
 Branch = 1'b1;
 ALUControl = 3'b001; //We will use Subtraction. If result 0, we get zeroFlag
 ReadReg2Src = 1'b1;
end
//For BNE, Branch Instruction
4'b1001: begin
 BranchNotEqual = 1'b1;
```

```
    ALUControl = 3'b001; //We will use Subtraction. If result 0, we get zeroFlag
    ReadReg2Src = 1'b1;
    //Need to think about separating BNE and BEQ (work left)
   end
   //For LUI, Load Upper Immidiate
   4'b1101: begin
    ALUsrc = 1'b1;   //select second input of ALu (signExtImm)
    RegWrite = 1'b1;
    ALUControl = 3'b101; //We will use add operation to perform this
    //Need to think about it (work done!)
   end
   4'b1110: begin
    ALUsrc = 1'b1;   //select second input of ALu (signExtImm)
    RegWrite = 1'b1;
    ALUControl = 3'b011; //Ori

   end

   //J type instruction
   4'b1010: begin
    Jump = 1'b1;  //for jump (j) instruction
   end
   // for jal (jump and link)
   4'b1011: begin
    Jump = 1'b1;
    RegWrite = 1'b1;
    RegDest = 1'b1;   //selects write to $ra in Datapath, saves return address PC+1 to ra
   end

   //for jr (jump register)
   4'b1100: begin
    Jump = 1'b1;
    JumpReg = 1'b1;

   end

   //HALT case
   4'b1111: begin
    //All signals stay 0
   end

  endcase
 end
endmodule
```

A.8 ISA_16Top.sv

```
module ISA_16Top(
 input logic clk,
 input logic resetn
 //no output logic here

);

 logic [9:0] PC;      //10 bit program counter
 logic [15:0] Instruction;  //16 bit instruction
 logic ReadReg2Src, RegDest, RegWrite, ALUsrc, MemRead, MemWrite, MemToReg, Branch, BranchNotEqual, Jum
 logic [2:0] ALUControl;   //Control for ALU
 logic zeroFlag;      //for zeroFlag from ALU

 //Instantiating the Instruction_Memory
 Instruction_Memory im (
  .PC(PC), //10 bit Program Counter
  .Instruction(Instruction)  //16 bits instruction output

 );

 //Instantiating the Control this will give us the control outputs for the datapath
 Control ct (
  .opcode(Instruction[15:12]),     // first 4 bits of instruction, opcode
  .ReadReg2Src(ReadReg2Src),      //ReadReg2Src signal from control
  .RegDest(RegDest),       //RegDest signal from control
  .RegWrite(RegWrite),      //RegWrite signal from control
  .ALUsrc(ALUsrc),        //ALUsrc signal from control
  .ALUControl(ALUControl),      //ALUControl signal from control
  .MemRead(MemRead),       //MemRead signal from control
  .MemWrite(MemWrite),       //MemWrite signal from control
  .MemToReg(MemToReg),       //MemToReg signal from control
  .Branch(Branch),        //Branch signal from control
  .BranchNotEqual(BranchNotEqual),
  .Jump(Jump),        //Jump signal from control
  .JumpReg(JumpReg)       //JumpReg signal from control
 );



 //Now we will pass the 16 bits instruction to the Datapath  along with clk and resetn
 //All the control signals will come from the output of Control Module
 //Instantiating the Datapath
 Datapath dp (
  .clk(clk),       //clock
  .resetn(resetn),       //active low reset
  .Instruction(Instruction),  //16 bit Instruction
  .ReadReg2Src(ReadReg2Src),     //ReadReg2Src signal from control
  .RegDest(RegDest),      //RegDest signal from control
  .RegWrite(RegWrite),     //RegWrite signal from control
  .ALUsrc(ALUsrc),       //ALUsrc signal from control
  .ALUControl(ALUControl),   //ALUControl signal from control
  .MemRead(MemRead),      //MemRead signal from control
  .MemWrite(MemWrite),      //MemWrite signal from control
  .MemToReg(MemToReg),      //MemToReg signal from control
  .Branch(Branch),       //Branch signal from control
  .BranchNotEqual(BranchNotEqual),
  .Jump(Jump),       //Jump signal from control
  .JumpReg(JumpReg),      //JumpReg signal from control
  .PC(PC),      //program counter output
  .zeroFlag(zeroFlag)     //zero flag from ALU

 );

endmodule
```

A.9 Instruction.hex

```
440A //ADDI $a0, $zero, 10
4814 //ADDI $s0, $zero, 20
0980 //ADD $s0, $a0, $s0
1A40 //SUB $s0, $s0, $a0
2580 //AND $a0, $a0, $s0
3580 //OR $a0, $a0, $s0
440F //ADDI $a0, $zero, 15
7580 //SLT $a0, $a0, $s0
D800 //LUI $s0, 0
EA32 //ORI $s0, $s0, 50
6804 //SW $s0, 4($zero)
5404 //LW $a0, 4($zero)
8901 //BEQ $a0, $s0, 1 ;skip next PC
4801 //ADDI $s0, $s0, 1
9101 //BNE $a0, $zero, 1
FFFF //HALT
A012 //J 0x12
FFFF //HALT
B015 //JAL 0x15
FFFF //HALT
0000 //NOP
C300 //JR $ra
```

A.10 Testbench Code (tbISA_16Top):

```
‘timescale 1ns/1ps

module tbISA_16Top;

 logic clk;
 logic resetn;

 //Creating the DUT(Device Under Test) instantiating the top module
 ISA_16Top dut (.clk(clk), .resetn(resetn));

 //clock generation
 always #5  clk = ~clk; //toggling the clock


 initial begin
  clk = 0;
  resetn = 0; //asserting active low resetn
  $display("\nWelcome to 16 bit RISC Processor!");
  //Holding resetn for 25 ns
  #25;

  resetn = 1; //Release active low resetn

  //Now the simulation will run for 200 ns
  #200;

  $finish;
 end


 //Display Outcomes
 //We will display the changes everytime there is a change in the PC

 always @(dut.PC) begin
  //Display the time, current PC, Instruction
  //We are also going to trace the register values here
  if(resetn) begin
   $display("Time: %0t;  PC: %h;  Instruction: %h;  $s0: %h;  $ra: %h; $a0: %h", $time, dut.PC, dut.Ins
  end

 end
endmodule
```

B FPGA Code Modifications

B.1 ISA_16Top_DE2115.sv

```
module ISA_16Top_DE2115 (
 input logic CLOCK_50,
 input logic [3:0] KEY,       //Press for clock pulse
 output logic [17:0] LEDR,       //to display current instuction
 output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 //to display the PC

);


 logic resetn;
 assign resetn = KEY[0]; //active low resetn

 //internal logic for cpu signals
 logic [9:0] fpga_pc;
 logic [15:0] fpga_instruction;

 // use KEY[1] as step clock (press → one instruction)
   logic cpu_clk;
   assign cpu_clk = ~KEY[1];   // KEY[1]: 1 (released), 0 (pressed) → invert, then posedge when pressed


 ISA_16Top isacpu (
  .clk(cpu_clk),
  .resetn(resetn),
  .PC(fpga_pc),      //10 bit program counter
  .Instruction(fpga_instruction)  //16 bit instruction


 );


 //output current PC in HEX display
 logic [3:0] fp_hex0, fp_hex1, fp_hex2, fp_hex3;

 assign fp_hex0 = fpga_pc[3:0];
 assign fp_hex1 = fpga_pc[7:4];
 assign fp_hex2 = {2'b00, fpga_pc[9:8]};

 //instantiating the 7 segment display module which will connect to the segment
 seven_seg h0 (.HEX(fp_hex0), .segment(HEX0));
 seven_seg h1 (.HEX(fp_hex1), .segment(HEX1));
 seven_seg h2 (.HEX(fp_hex2), .segment(HEX2));

 //HEX3 blank
 assign HEX3 = 7'b111_1111;

 logic [3:0] instruction_hex0, instruction_hex1, instruction_hex2, instruction_hex3;


 //Instruction output in HEX display
 assign instruction_hex0 = fpga_instruction[3:0];
   assign instruction_hex1 = fpga_instruction[7:4];
   assign instruction_hex2 = fpga_instruction[11:8];
   assign instruction_hex3 = fpga_instruction[15:12];

 seven_seg hi0 (.HEX(instruction_hex0), .segment(HEX4)); // HEX4 = instr[3:0]
   seven_seg hi1 (.HEX(instruction_hex1), .segment(HEX5)); // HEX5 = instr[7:4]
   seven_seg hi2 (.HEX(instruction_hex2), .segment(HEX6)); // HEX6 = instr[11:8]
   seven_seg hi3 (.HEX(instruction_hex3), .segment(HEX7)); // HEX7 = instr[15:12]


endmodule
```

B.2 seven_seg.sv

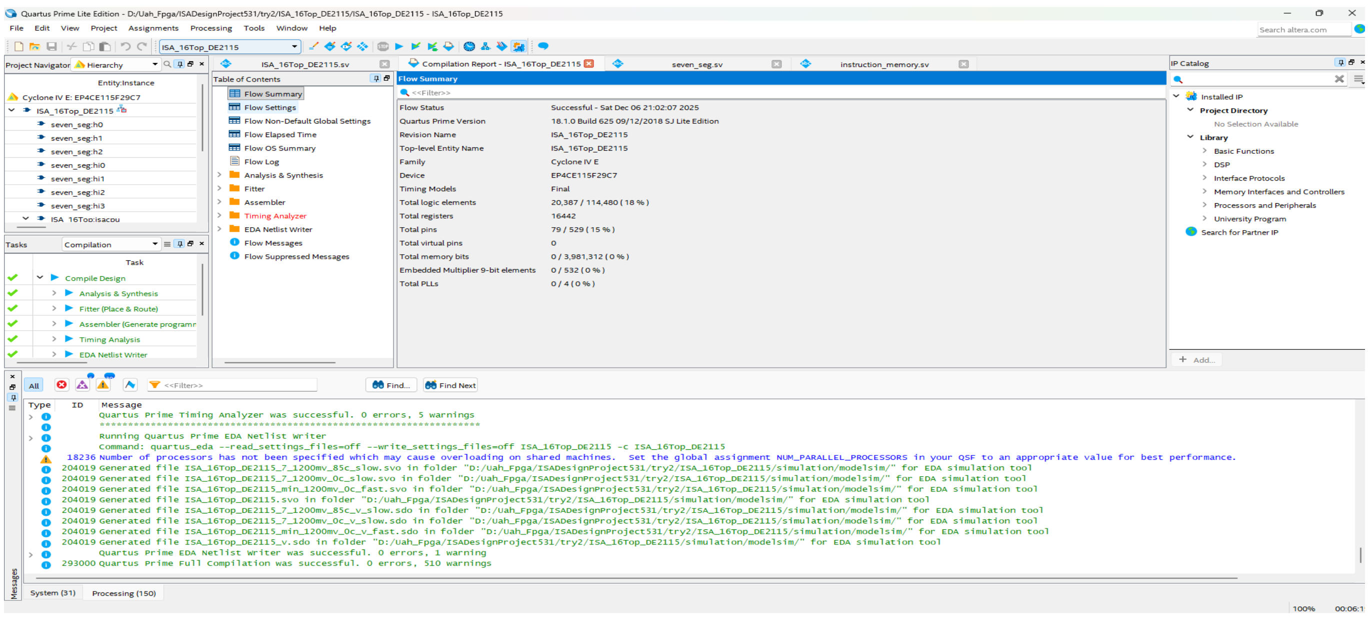
```
module seven_seg(

 input logic [3:0] HEX,
    output logic [6:0] segment
);

  always_comb
    case (HEX)
       0: segment = 7'b100_0000;
       1: segment = 7'b111_1001;
       2: segment = 7'b010_0100;
       3: segment = 7'b011_0000;
       4: segment = 7'b001_1001;
       5: segment = 7'b001_0010;
       6: segment = 7'b000_0010;
       7: segment = 7'b111_1000;
       8: segment = 7'b000_0000;
       9: segment = 7'b001_1000;
      10: segment = 7'b000_1000; // 'A'
      11: segment = 7'b000_0011; // 'b'
      12: segment = 7'b100_0110; // 'C'
      13: segment = 7'b010_0001; // 'd'
      14: segment = 7'b000_0110; // 'E'
      15: segment = 7'b000_1110; // 'F'
     default: segment = 7'b111_1111;
   endcase
endmodule
```

B.3 ISA_16Top.sv

```
module ISA_16Top(
 input logic clk,
 input logic resetn,
    //Modifications made here using them as output
 output logic [9:0] PC,     //10 bit program counter
 output logic [15:0] Instruction  //16 bit instruction

);


 logic ReadReg2Src, RegDest, RegWrite, ALUsrc, MemRead, MemWrite, MemToReg, Branch, BranchNotEqual, Jum
 logic [2:0] ALUControl;   //Control for ALU
 logic zeroFlag;      //for zeroFlag from ALU

 //Instantiating the Instruction_Memory
 Instruction_Memory im (
  .PC(PC), //10 bit Program Counter
  .Instruction(Instruction)  //16 bits instruction output

 );

 //Instantiating the Control this will give us the control outputs for the datapath
 Control ct (
  .opcode(Instruction[15:12]),     // first 4 bits of instruction, opcode
  .ReadReg2Src(ReadReg2Src),      //ReadReg2Src signal from control
  .RegDest(RegDest),       //RegDest signal from control
  .RegWrite(RegWrite),      //RegWrite signal from control
  .ALUsrc(ALUsrc),        //ALUsrc signal from control
  .ALUControl(ALUControl),      //ALUControl signal from control
  .MemRead(MemRead),       //MemRead signal from control
  .MemWrite(MemWrite),       //MemWrite signal from control
  .MemToReg(MemToReg),       //MemToReg signal from control
  .Branch(Branch),        //Branch signal from control
  .BranchNotEqual(BranchNotEqual),
  .Jump(Jump),        //Jump signal from control
  .JumpReg(JumpReg)       //JumpReg signal from control
 );



 //Now we will pass the 16 bits instruction to the Datapath  along with clk and resetn
 //All the control signals will come from the output of Control Module
 //Instantiating the Datapath
 Datapath dp (
  .clk(clk),       //clock
  .resetn(resetn),       //active low reset
  .Instruction(Instruction),  //16 bit Instruction
  .ReadReg2Src(ReadReg2Src),     //ReadReg2Src signal from control
  .RegDest(RegDest),      //RegDest signal from control
  .RegWrite(RegWrite),     //RegWrite signal from control
  .ALUsrc(ALUsrc),       //ALUsrc signal from control
  .ALUControl(ALUControl),   //ALUControl signal from control
  .MemRead(MemRead),      //MemRead signal from control
  .MemWrite(MemWrite),      //MemWrite signal from control
  .MemToReg(MemToReg),      //MemToReg signal from control
  .Branch(Branch),       //Branch signal from control
  .BranchNotEqual(BranchNotEqual),
  .Jump(Jump),       //Jump signal from control
  .JumpReg(JumpReg),      //JumpReg signal from control
  .PC(PC),      //program counter output
  .zeroFlag(zeroFlag)     //zero flag from ALU

 );

endmodule
```

C Compilation Reports

C.1 Quartus Prime for Hardware Synthesis



Quartus Prime Lite Edition - D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/ISA_16Top_DE2115 - ISA_16Top_DE2115

Compilation Report - ISA_16Top_DE2115

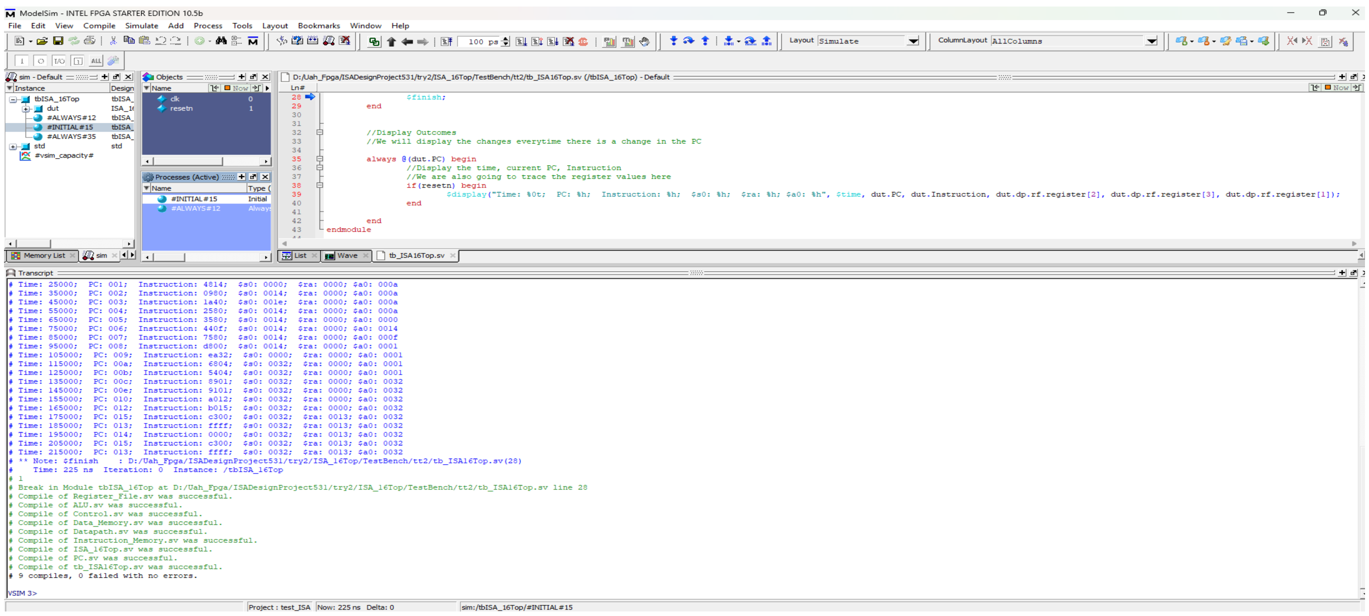
Flow Summary

Flow Status	Successful - Sat Dec 06 21:02:07 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	ISA_16Top_DE2115
Top-level Entity Name	ISA_16Top_DE2115
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	20,387 / 114,480 (18 %)
Total registers	16442
Total pins	79 / 529 (15 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

```
Type  ID     Message
             Quartus Prime Timing Analyzer was successful. 0 errors, 5 warnings
             ************************************************************
             Running Quartus Prime EDA Netlist Writer
             Command: quartus_eda --read_settings_files=off --write_settings_files=off ISA_16Top_DE2115 -c ISA_16Top_DE2115
       18236 Number of processors has not been specified which may cause overloading on shared machines.  Set the global assignment NUM_PARALLEL_PROCESSORS in your QSF to an appropriate value for best performance.
      204019 Generated file ISA_16Top_DE2115_7_1200mv_85c_slow.svo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
      204019 Generated file ISA_16Top_DE2115_7_1200mv_0c_slow.svo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
      204019 Generated file ISA_16Top_DE2115_min_1200mv_0c_fast.svo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
      204019 Generated file ISA_16Top_DE2115.svo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
      204019 Generated file ISA_16Top_DE2115_7_1200mv_85c_v_slow.sdo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
      204019 Generated file ISA_16Top_DE2115_7_1200mv_0c_v_slow.sdo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
      204019 Generated file ISA_16Top_DE2115_min_1200mv_0c_v_fast.sdo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
      204019 Generated file ISA_16Top_DE2115_v.sdo in folder "D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top_DE2115/simulation/modelsim/" for EDA simulation tool
             Quartus Prime EDA Netlist Writer was successful. 0 errors, 1 warning
      293000 Quartus Prime Full Compilation was successful. 0 errors, 510 warnings
```

Figure 7: Compilation Report in Quartus Prime Lite

C.2 ModelSim Altera for Software Simulation



ModelSim - INTEL FPGA STARTER EDITION 10.5b

D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top/TestBench/tt2/tb_ISA16Top.sv (/tbISA_16Top) - Default

```
            $finish;
        end


        //Display Outcomes
        //We will display the changes everytime there is a change in the PC

        always @(dut.PC) begin
            //Display the time, current PC, Instruction
            //We are also going to trace the register values here
            if(resetn) begin
                $display("Time: %0t;  PC: %h;  Instruction: %h;  $s0: %h;  $ra: %h; $a0: %h", $time, dut.PC, dut.Instruction, dut.dp.rf.register[2], dut.dp.rf.register[3], dut.dp.rf.register[1]);
            end

        end
    endmodule
```

Transcript

```
# Time: 25000;  PC: 001;  Instruction: 4814;  $s0: 0000;  $ra: 0000; $a0: 000a
# Time: 35000;  PC: 002;  Instruction: 0980;  $s0: 0014;  $ra: 0000; $a0: 000a
# Time: 45000;  PC: 003;  Instruction: 1a40;  $s0: 001e;  $ra: 0000; $a0: 000a
# Time: 55000;  PC: 004;  Instruction: 2580;  $s0: 0014;  $ra: 0000; $a0: 000a
# Time: 65000;  PC: 005;  Instruction: 3580;  $s0: 0014;  $ra: 0000; $a0: 0000
# Time: 75000;  PC: 006;  Instruction: 440f;  $s0: 0014;  $ra: 0000; $a0: 0014
# Time: 85000;  PC: 007;  Instruction: 7580;  $s0: 0014;  $ra: 0000; $a0: 000f
# Time: 95000;  PC: 008;  Instruction: d800;  $s0: 0014;  $ra: 0000; $a0: 0001
# Time: 105000;  PC: 009;  Instruction: ea32;  $s0: 0000;  $ra: 0000; $a0: 0001
# Time: 115000;  PC: 00a;  Instruction: 6804;  $s0: 0032;  $ra: 0000; $a0: 0001
# Time: 125000;  PC: 00b;  Instruction: 5404;  $s0: 0032;  $ra: 0000; $a0: 0001
# Time: 135000;  PC: 00c;  Instruction: 8901;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 145000;  PC: 00e;  Instruction: 9101;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 155000;  PC: 010;  Instruction: a012;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 165000;  PC: 012;  Instruction: b015;  $s0: 0032;  $ra: 0000; $a0: 0032
# Time: 175000;  PC: 015;  Instruction: c300;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 185000;  PC: 013;  Instruction: ffff;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 195000;  PC: 014;  Instruction: 0000;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 205000;  PC: 015;  Instruction: c300;  $s0: 0032;  $ra: 0013; $a0: 0032
# Time: 215000;  PC: 013;  Instruction: ffff;  $s0: 0032;  $ra: 0013; $a0: 0032
# ** Note: $finish    : D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top/TestBench/tt2/tb_ISA16Top.sv(28)
#    Time: 225 ns  Iteration: 0  Instance: /tbISA_16Top
# 1
# Break in Module tbISA_16Top at D:/Uah_Fpga/ISADesignProject531/try2/ISA_16Top/TestBench/tt2/tb_ISA16Top.sv line 28
# Compile of Register_File.sv was successful.
# Compile of ALU.sv was successful.
# Compile of Control.sv was successful.
# Compile of Data_Memory.sv was successful.
# Compile of Datapath.sv was successful.
# Compile of Instruction_Memory.sv was successful.
# Compile of ISA_16Top.sv was successful.
# Compile of PC.sv was successful.
# Compile of tb_ISA16Top.sv was successful.
# 9 compiles, 0 failed with no errors.

VSIM 3>
```

Project : test_ISA   Now: 225 ns Delta: 0   sim:/tbISA_16Top/#INITIAL#15

Figure 8: Compilation Report in ModelSim Altera