UNIVERSITY OF MORATUWA

Faculty of Engineering

Non-pipelined Single Stage (Cycle) CPU Design - Individual Project

H.D.K.G.WIJESIRI – 200728R

Github -
https://github.com/kaveeshwaragayanath/RISCV_Single_Cycle_Processor

Department of Electronic and Telecommunication Engineering

- ## INTRODUCTION

RISC-V 32I single cycle CPU is a type of processor that implements the Reduced Instruction Set Computing (RISC) architecture with a 32-bit instruction set and a single cycle implementation. Each instruction is executed in a single clock cycle, making it a fast and efficient way to process instructions. This type of processor is based on the open-source RISC-V ISA, which is becoming popular in embedded systems, microcontrollers, and other applications. The 32I in the name refers to the 32-bit instruction width, providing a balance between code density and processing performance.

- ## PROBLEM

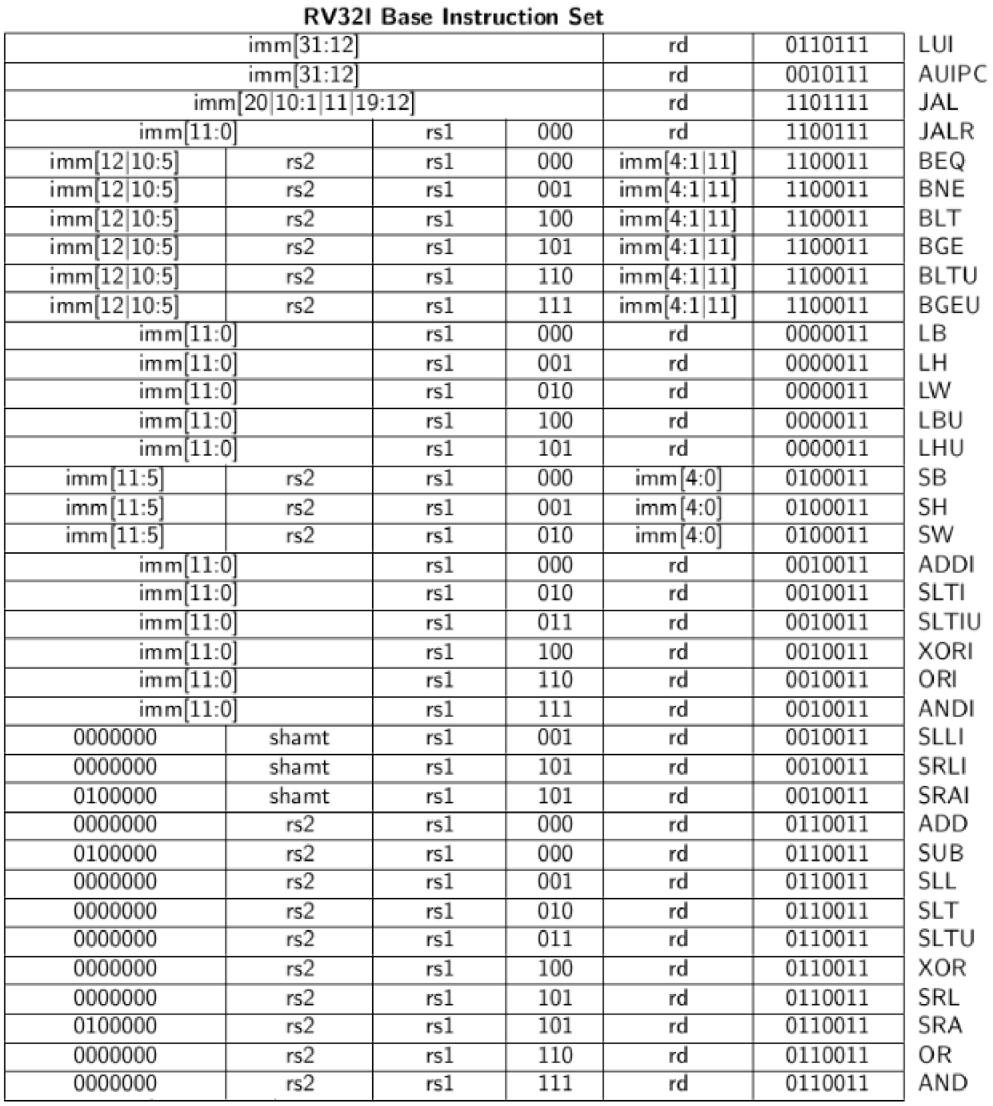
This individual project focuses on the development of a 32-bit non-pipelined single-cycle processor within an FPGA environment, specifically adhering to the RV32I architecture. The project comprises three stages, with stage 1 dedicated to the creation of the single-cycle processor. It utilizes Microprogramming and features a 3-bus structure. Additionally, the project entails the introduction of two new instructions: MEMCOPY, designed for copying arrays of varying sizes, and MUL (Unsigned Multiplication), addressing the absence of native multiplication instructions in RV32I.

- ## INSTRUCTION SET ARCHITECTURE

General Architecture

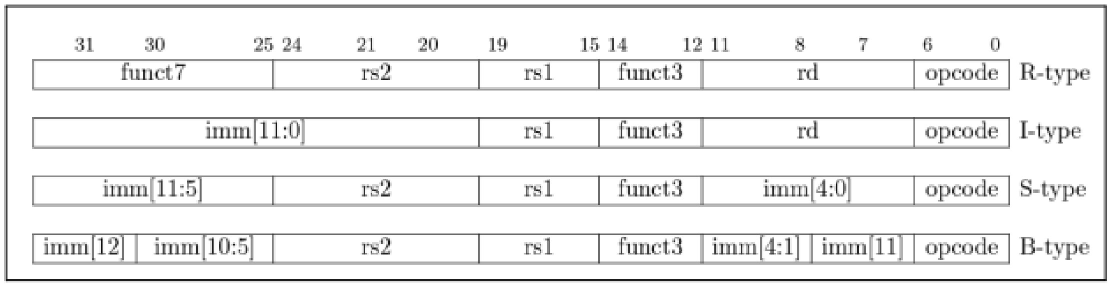
RISC-V implementations require a base integer ISA and optional extensions. Instructions are 32-bit and naturally aligned, and little-endian memory is used. In this case we implemented following instructions.

Instruction Set

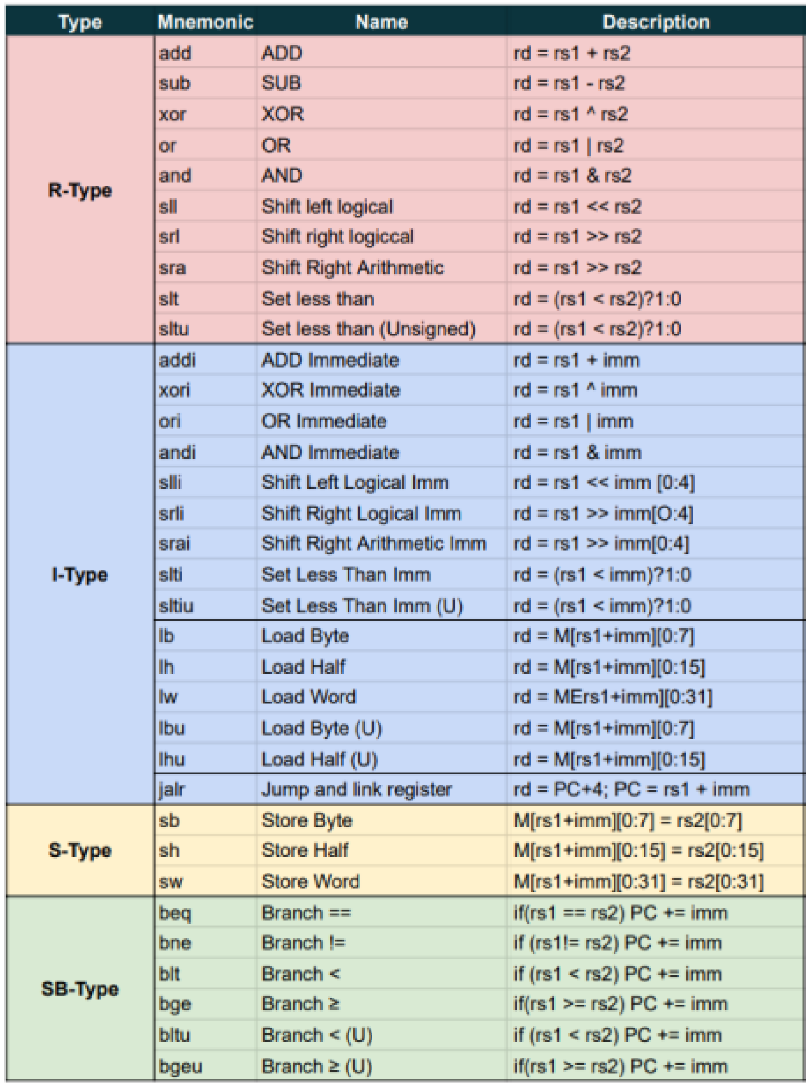


RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND



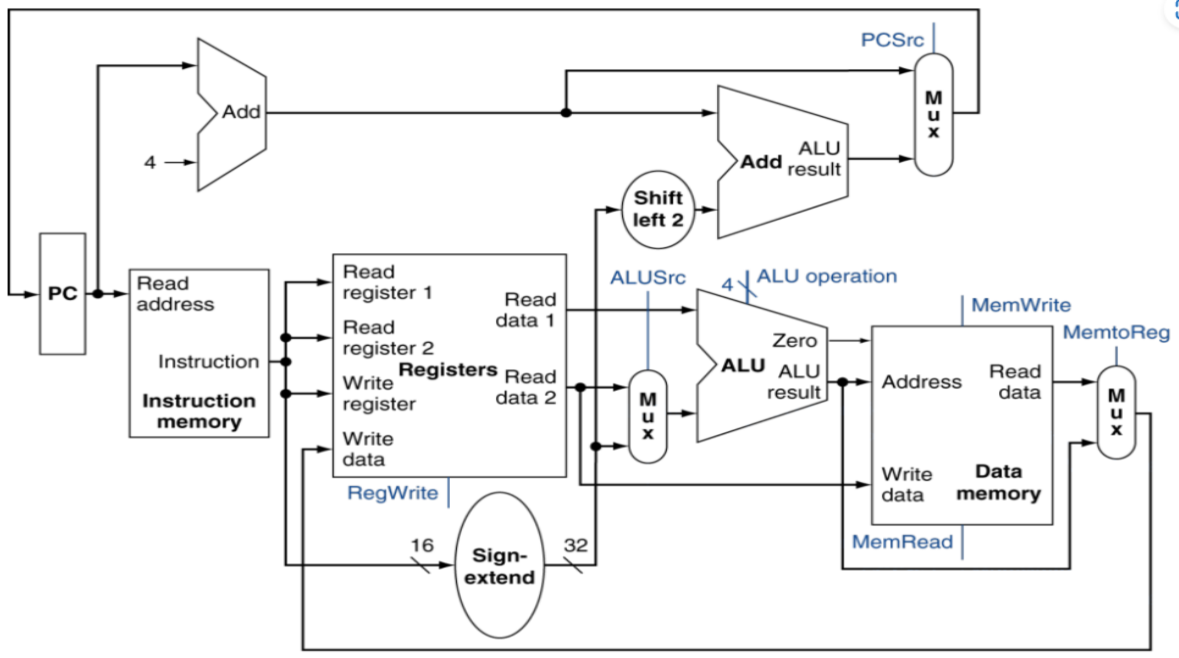
31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type



Type	Mnemonic	Name	Description
R-Type	add	ADD	rd = rs1 + rs2
	sub	SUB	rd = rs1 - rs2
	xor	XOR	rd = rs1 ^ rs2
	or	OR	rd = rs1 | rs2
	and	AND	rd = rs1 & rs2
	sll	Shift left logical	rd = rs1 << rs2
	srl	Shift right logiccal	rd = rs1 >> rs2
	sra	Shift Right Arithmetic	rd = rs1 >> rs2
	slt	Set less than	rd = (rs1 < rs2)?1:0
	sltu	Set less than (Unsigned)	rd = (rs1 < rs2)?1:0
I-Type	addi	ADD Immediate	rd = rs1 + imm
	xori	XOR Immediate	rd = rs1 ^ imm
	ori	OR Immediate	rd = rs1 | imm
	andi	AND Immediate	rd = rs1 & imm
	slli	Shift Left Logical Imm	rd = rs1 << imm [0:4]
	srli	Shift Right Logical Imm	rd = rs1 >> imm[O:4]
	srai	Shift Right Arithmetic Imm	rd = rs1 >> imm[0:4]
	slti	Set Less Than Imm	rd = (rs1 < imm)?1:0
	sltiu	Set Less Than Imm (U)	rd = (rs1 < imm)?1:0
	lb	Load Byte	rd = M[rs1+imm][0:7]
	lh	Load Half	rd = M[rs1+imm][0:15]
	lw	Load Word	rd = MErs1+imm][0:31]
	lbu	Load Byte (U)	rd = M[rs1+imm][0:7]
	lhu	Load Half (U)	rd = M[rs1+imm][0:15]
	jalr	Jump and link register	rd = PC+4; PC = rs1 + imm
S-Type	sb	Store Byte	M[rs1+imm][0:7] = rs2[0:7]
	sh	Store Half	M[rs1+imm][0:15] = rs2[0:15]
	sw	Store Word	M[rs1+imm][0:31] = rs2[0:31]
SB-Type	beq	Branch ==	if(rs1 == rs2) PC += imm
	bne	Branch !=	if (rs1!= rs2) PC += imm
	blt	Branch <	if (rs1 < rs2) PC += imm
	bge	Branch ≥	if(rs1 >= rs2) PC += imm
	bltu	Branch < (U)	if (rs1 < rs2) PC += imm
	bgeu	Branch ≥ (U)	if(rs1 >= rs2) PC += imm

Data Path

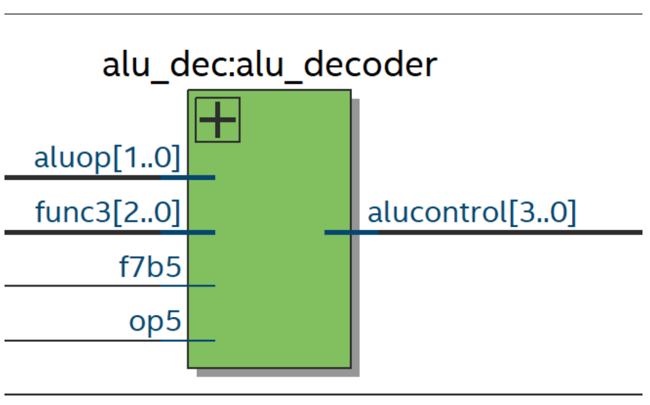
(In the data memory module there are extractor modules to execute various load and store instructions.

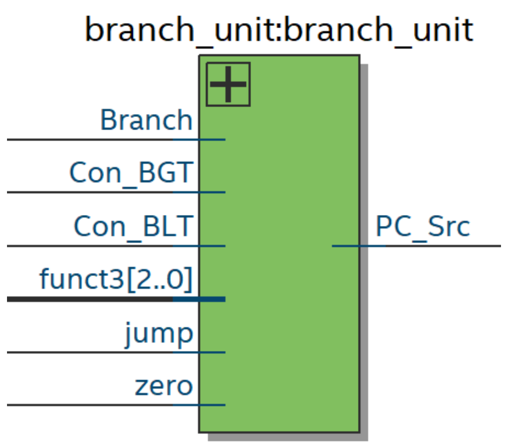


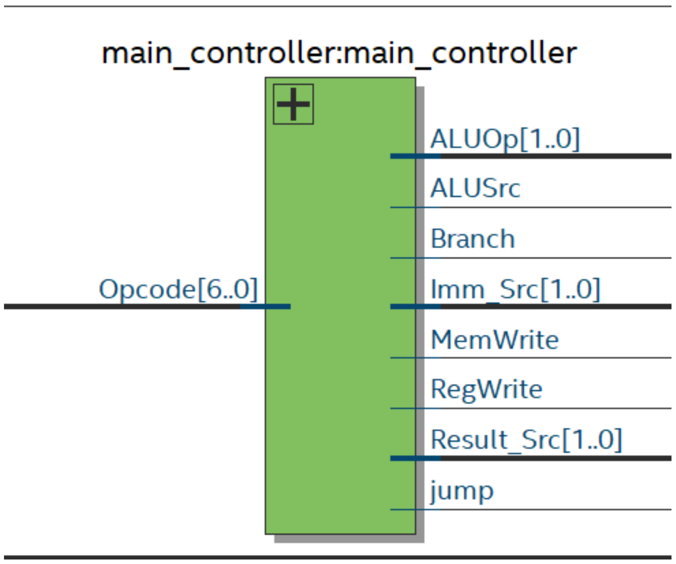
Control Unit

All control signals were generated using three blocks.

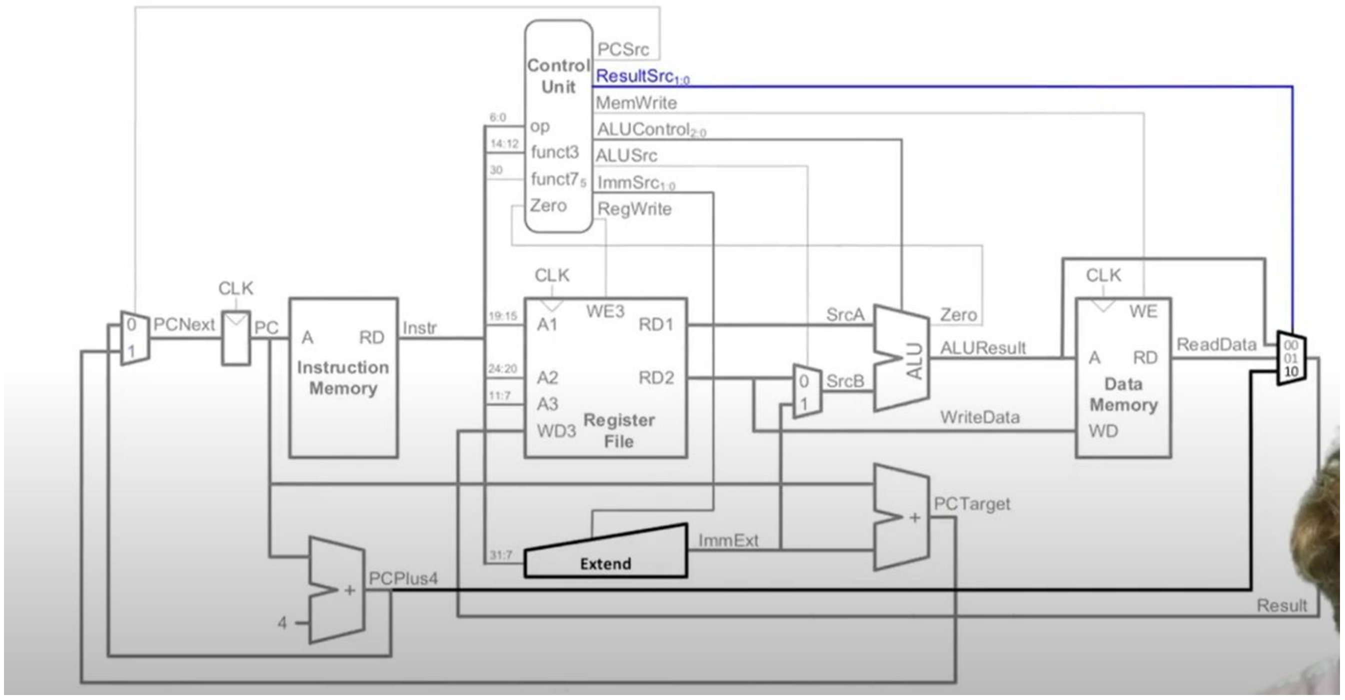
1. Main Controller
2. Alu Decoder
3. Branch unit







Full Path (Data path + Control Path)



Modules and Components

1. Program Counter(PC)

The program counter (PC) is a register in a computer's central processing unit (CPU) that contains the address of the instruction being executed at the current time. The program counter is automatically incremented after each instruction is executed, so that the next instruction to be executed is located at the next memory address.

2. Control Unit

The control unit in a RISC-V 32I processor decodes the instructions fetched from instruction memory and generates control signals to direct the operations within the processor. It determines the instruction type and handles all register, immediate, branching, load and store instructions. Instructions in RISC-V 32I architecture are fixed-length and binary encoded. The control unit decodes each instruction and sends control signals to control the flow of data and instructions within the processor. These signals are used to coordinate the operations of the various components of the processor, such as the ALU, the register file, and the memory unit. The control unit generates the following control signals based solely on the instruction’s opcode.

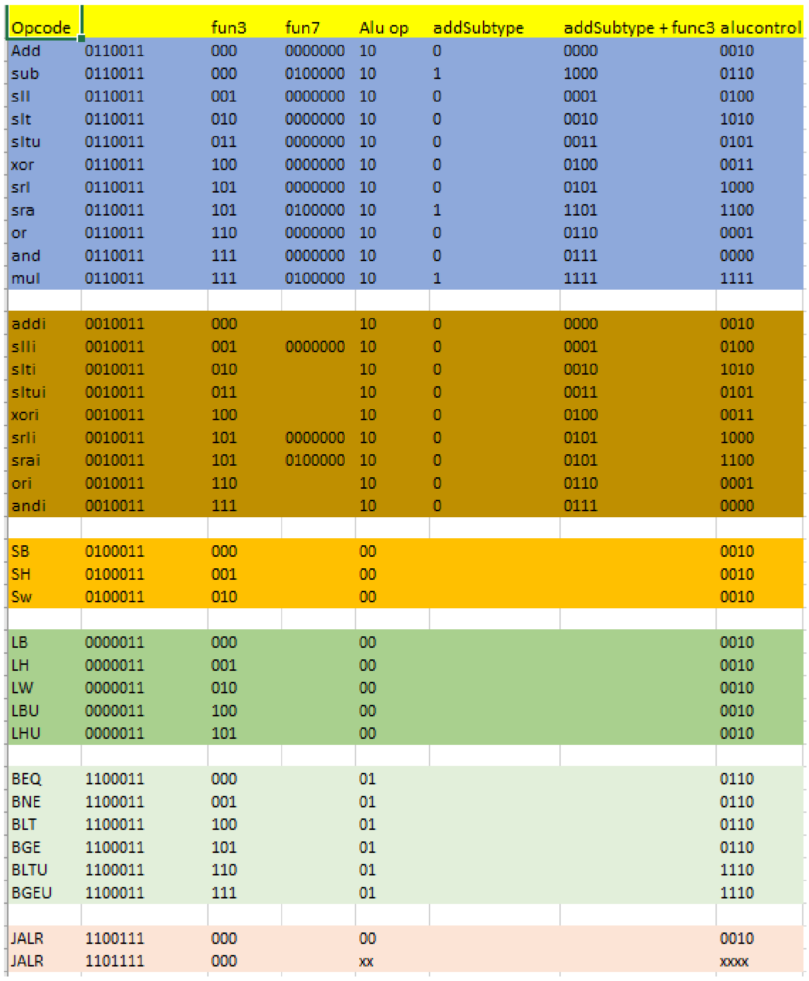
From Main Decoder :

- Jump - specifically for the I-Type JALR instruction and JAL instruction.
- Branch - specifies whether the processor should take a branch instruction, i.e., change the program flow to a different location in the code as specified in the branch instruction.
- Result Src – 2-bit signal that is used to select PC+4 or read data from memory or ALU Result for storing in the register during register write.
- MemWrite – signal that is enables writing data to the data memory.
- RegWrite – signal that is enables writing to the register.
- Imm src –2-bit signal to select the type of immediate.
- Alu op – 2 bit signal allows for subsequent ALU decoder module to determine the LU operation

From Alu   Decoder :

This takes alu op , func3 and func7 as inputs and generate correct alu control signal.

- Alu controller – generate correct alu control signal



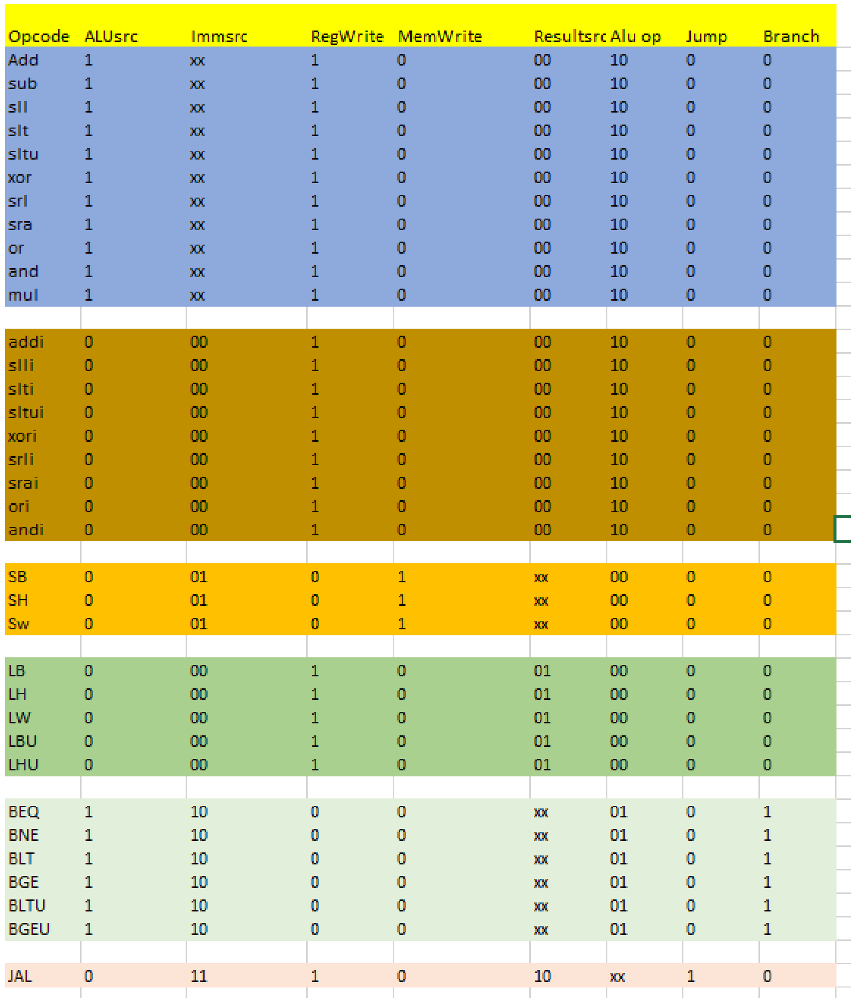
Opcode		fun3	fun7	Alu op	addSubtype	addSubtype + func3	alucontrol
Add	0110011	000	0000000	10	0	0000	0010
sub	0110011	000	0100000	10	1	1000	0110
sll	0110011	001	0000000	10	0	0001	0100
slt	0110011	010	0000000	10	0	0010	1010
sltu	0110011	011	0000000	10	0	0011	0101
xor	0110011	100	0000000	10	0	0100	0011
srl	0110011	101	0000000	10	0	0101	1000
sra	0110011	101	0100000	10	1	1101	1100
or	0110011	110	0000000	10	0	0110	0001
and	0110011	111	0000000	10	0	0111	0000
mul	0110011	111	0100000	10	1	1111	1111
addi	0010011	000		10	0	0000	0010
slli	0010011	001	0000000	10	0	0001	0100
slti	0010011	010		10	0	0010	1010
sltui	0010011	011		10	0	0011	0101
xori	0010011	100		10	0	0100	0011
srli	0010011	101	0000000	10	0	0101	1000
srai	0010011	101	0100000	10	0	0101	1100
ori	0010011	110		10	0	0110	0001
andi	0010011	111		10	0	0111	0000
SB	0100011	000		00			0010
SH	0100011	001		00			0010
Sw	0100011	010		00			0010
LB	0000011	000		00			0010
LH	0000011	001		00			0010
LW	0000011	010		00			0010
LBU	0000011	100		00			0010
LHU	0000011	101		00			0010
BEQ	1100011	000		01			0110
BNE	1100011	001		01			0110
BLT	1100011	100		01			0110
BGE	1100011	101		01			0110
BLTU	1100011	110		01			1110
BGEU	1100011	111		01			1110
JALR	1100111	000		00			0010
JALR	1101111	000		xx			xxxx

From Branch   Decoder :

This module takes branch , jump , BLT , BGT, func3, zero and generate PC src.

- PC src – generate branch or not

Control Signals



Opcode	ALUsrc	Immsrc	RegWrite	MemWrite	Resultsrc	Alu op	Jump	Branch
Add	1	xx	1	0	00	10	0	0
sub	1	xx	1	0	00	10	0	0
sll	1	xx	1	0	00	10	0	0
slt	1	xx	1	0	00	10	0	0
sltu	1	xx	1	0	00	10	0	0
xor	1	xx	1	0	00	10	0	0
srl	1	xx	1	0	00	10	0	0
sra	1	xx	1	0	00	10	0	0
or	1	xx	1	0	00	10	0	0
and	1	xx	1	0	00	10	0	0
mul	1	xx	1	0	00	10	0	0
addi	0	00	1	0	00	10	0	0
slli	0	00	1	0	00	10	0	0
slti	0	00	1	0	00	10	0	0
sltui	0	00	1	0	00	10	0	0
xori	0	00	1	0	00	10	0	0
srli	0	00	1	0	00	10	0	0
srai	0	00	1	0	00	10	0	0
ori	0	00	1	0	00	10	0	0
andi	0	00	1	0	00	10	0	0
SB	0	01	0	1	xx	00	0	0
SH	0	01	0	1	xx	00	0	0
Sw	0	01	0	1	xx	00	0	0
LB	0	00	1	0	01	00	0	0
LH	0	00	1	0	01	00	0	0
LW	0	00	1	0	01	00	0	0
LBU	0	00	1	0	01	00	0	0
LHU	0	00	1	0	01	00	0	0
BEQ	1	10	0	0	xx	01	0	1
BNE	1	10	0	0	xx	01	0	1
BLT	1	10	0	0	xx	01	0	1
BGE	1	10	0	0	xx	01	0	1
BLTU	1	10	0	0	xx	01	0	1
BGEU	1	10	0	0	xx	01	0	1
JAL	0	11	1	0	10	xx	1	0

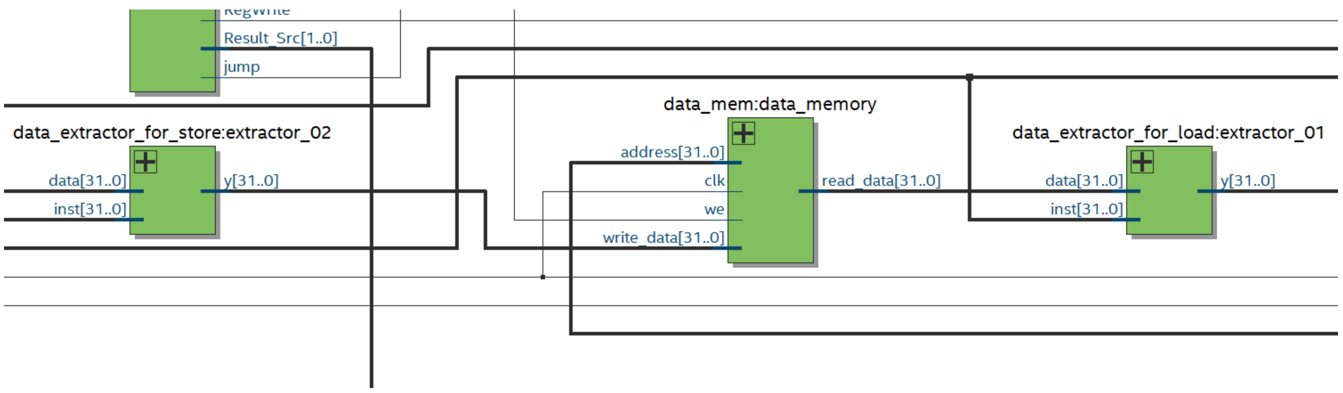
3. Instruction Memory

Instruction memory stores the program instructions that the processor executes. It is separate from the data memory, which stores the data being processed. The processor fetches instructions from the instruction memory, decodes them in the control unit, and generates control signals to execute them. Instructions are fixed-length and binary encoded, making them easy to decode and execute. Instruction memory is typically implemented as read-only memory (ROM) or flash memory, which cannot be modified during execution. The data memory is typically organized as a large array of memory cells. In our design, the size of the instruction memory is 32x256.

4. Data Memory

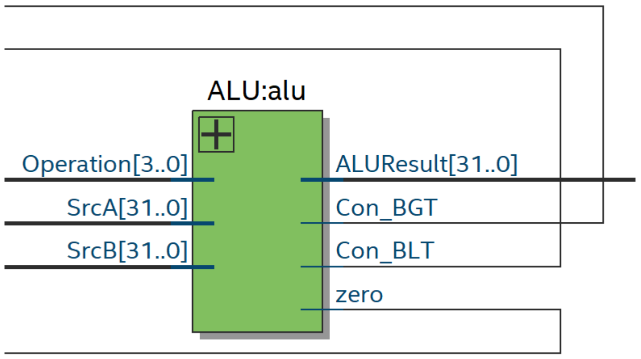
In the context of a non-pipeline single-cycle RISC-V processor, the data memory plays a crucial role as it comprises 1024 memory cells, with each cell capable of storing 32 bits of data. These memory cells serve as the primary data storage location within the processor, enabling the storage and retrieval of important information during program execution.

For the correct execute of the all load instructions(lh,lw,lb,lbu,lhu) and all store instruction(sw,sb,sh) I added two extractor modules.



5. ALU

The ALU is responsible for performing a wide range of operations, including addition, subtraction, multiplication, division, bit shifting, and bitwise logical operations like AND, OR, and XOR. In a RISC-V 32I processor, the ALU performs the operations specified by the ALU Controller, which receives its input from the control unit and the instruction decoder. The ALU performs the selected operation on the inputs received from the register file, and the result is stored back in the register file or in a memory location.

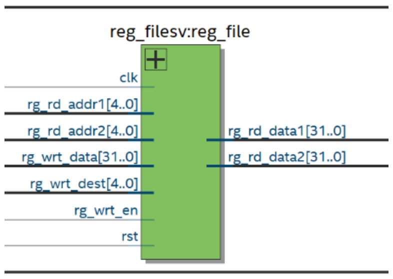


6. Immediate Generator

It checks instruction type and choose correct immediate and extend the immediate to 32 bits.

7. Register File

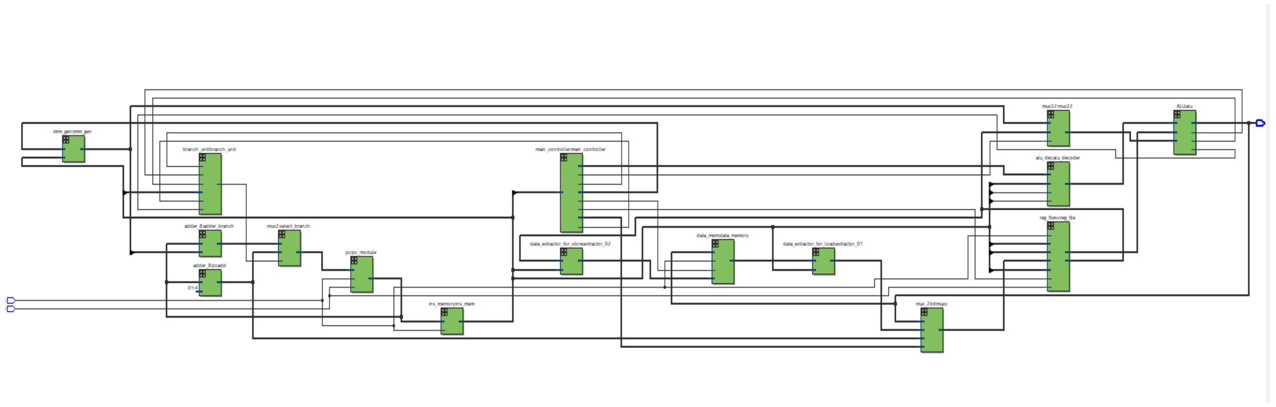
There are 32 registers, x0-x32 in the register file. Integer values are stored in those general-purpose registers x1-x31. The constant 0 is hardwired into register x0. However, the typical software calling convention uses register x1 to hold the return address on a call since there isn't a hardwired subroutine return address link register. The x registers for RV32 are 32 bits wide.



Reg	ABI/Alias	Description	Saved
`x0`	`zero`	Hard-wired zero	
`x1`	`ra`	Return address	
`x2`	`sp`	Stack pointer	yes
`x3`	`gp`	Global pointer	
`x4`	`tp`	Thread pointer	
`x5`	`t0`	Temporary/alternate link register	
`x6-7`	`t1-2`	Temporaries	
`x8`	`s0/fp`	Saved register/frame pointer	yes
`x9`	`s1`	Saved register	yes
`x10-11`	`a0-1`	Function arguments/return value	
`x12-17`	`a2-7`	Function arguments	
`x18-27`	`s2-11`	Saved registers	yes
`x28-31`	`t3-6`	Temporaries	

Implementation :

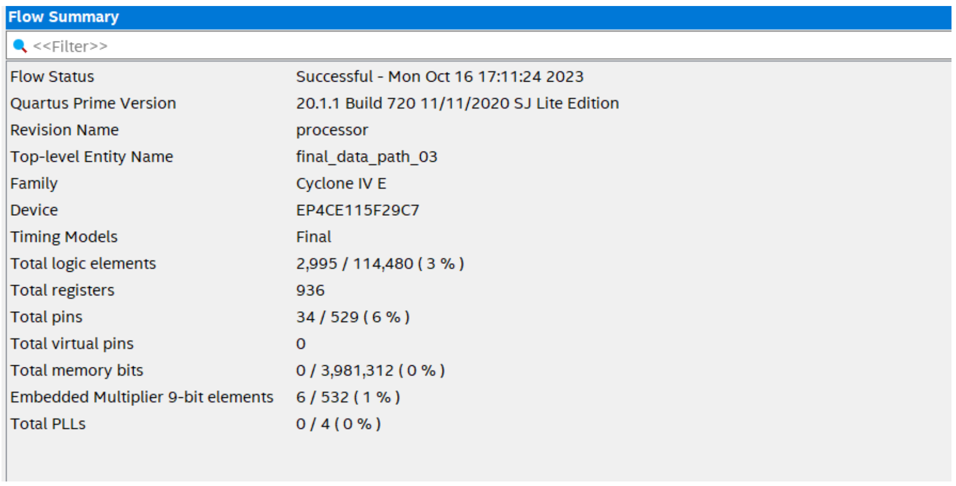
Netlist View of the complete processor



Implementation in Quartus prime

 The development of this processor utilized Quartus Prime software and the System Verilog HDL language. Upon completion of the design, each module was compiled and tested. The top-level processor design was then compiled and subjected to simulation using the integrated ModelSim software within Quartus Prime and each instruction was tested.The data paths were analyzed and validated through the Netlist Viewer (RTL Viewer) option.

The flow summary as folllows :

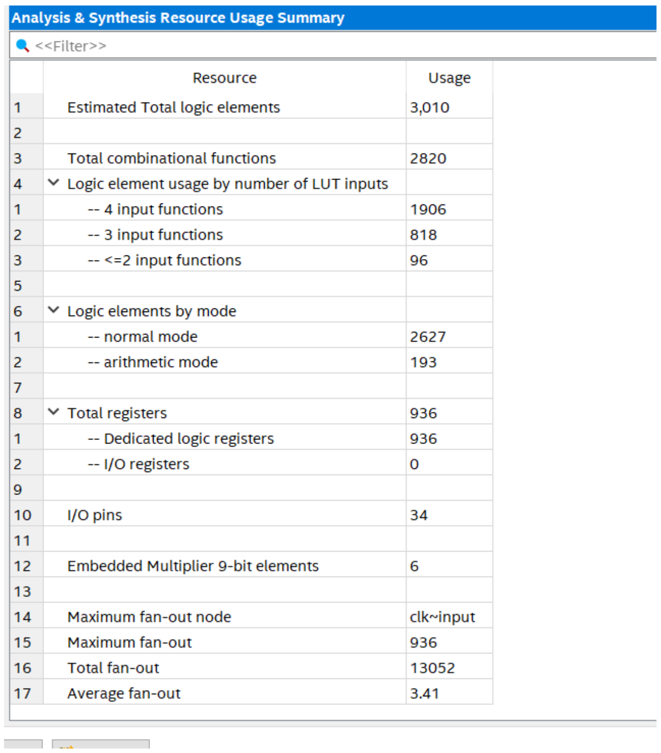


Flow Summary

<<Filter>>

Flow Status	Successful - Mon Oct 16 17:11:24 2023
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	processor
Top-level Entity Name	final_data_path_03
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2,995 / 114,480 (3 %)
Total registers	936
Total pins	34 / 529 (6 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	6 / 532 (1 %)
Total PLLs	0 / 4 (0 %)

The resource usage as follows:

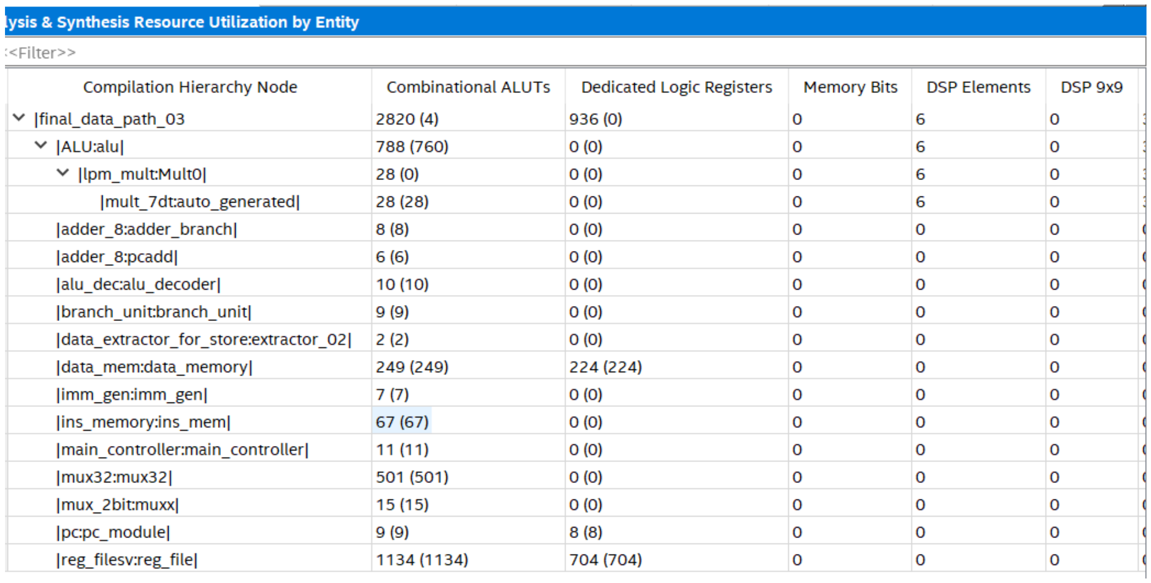


Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated Total logic elements	3,010
2		
3	Total combinational functions	2820
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	1906
2	-- 3 input functions	818
3	-- <=2 input functions	96
5		
6	Logic elements by mode	
1	-- normal mode	2627
2	-- arithmetic mode	193
7		
8	Total registers	936
1	-- Dedicated logic registers	936
2	-- I/O registers	0
9		
10	I/O pins	34
11		
12	Embedded Multiplier 9-bit elements	6
13		
14	Maximum fan-out node	clk~input
15	Maximum fan-out	936
16	Total fan-out	13052
17	Average fan-out	3.41

Resource utilization summary



lysis & Synthesis Resource Utilization by Entity

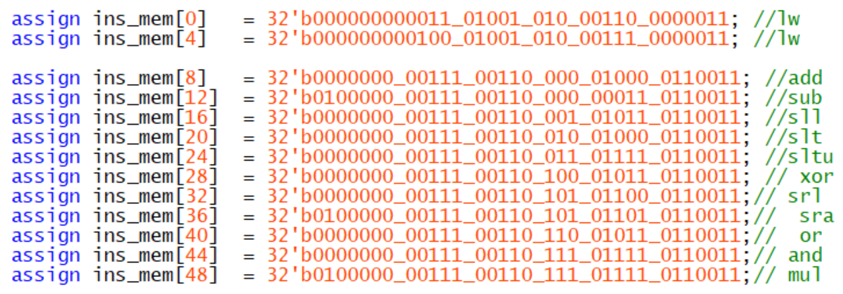
<<Filter>>

Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Memory Bits	DSP Elements	DSP 9x9
|final_data_path_03	2820 (4)	936 (0)	0	6	0
|ALU:alu|	788 (760)	0 (0)	0	6	0
|lpm_mult:Mult0|	28 (0)	0 (0)	0	6	0
|mult_7dt:auto_generated|	28 (28)	0 (0)	0	6	0
|adder_8:adder_branch|	8 (8)	0 (0)	0	0	0
|adder_8:pcadd|	6 (6)	0 (0)	0	0	0
|alu_dec:alu_decoder|	10 (10)	0 (0)	0	0	0
|branch_unit:branch_unit|	9 (9)	0 (0)	0	0	0
|data_extractor_for_store:extractor_02|	2 (2)	0 (0)	0	0	0
|data_mem:data_memory|	249 (249)	224 (224)	0	0	0
|imm_gen:imm_gen|	7 (7)	0 (0)	0	0	0
|ins_memory:ins_mem|	67 (67)	0 (0)	0	0	0
|main_controller:main_controller|	11 (11)	0 (0)	0	0	0
|mux32:mux32|	501 (501)	0 (0)	0	0	0
|mux_2bit:muxx|	15 (15)	0 (0)	0	0	0
|pc:pc_module|	9 (9)	8 (8)	0	0	0
|reg_filesv:reg_file|	1134 (1134)	704 (704)	0	0	0

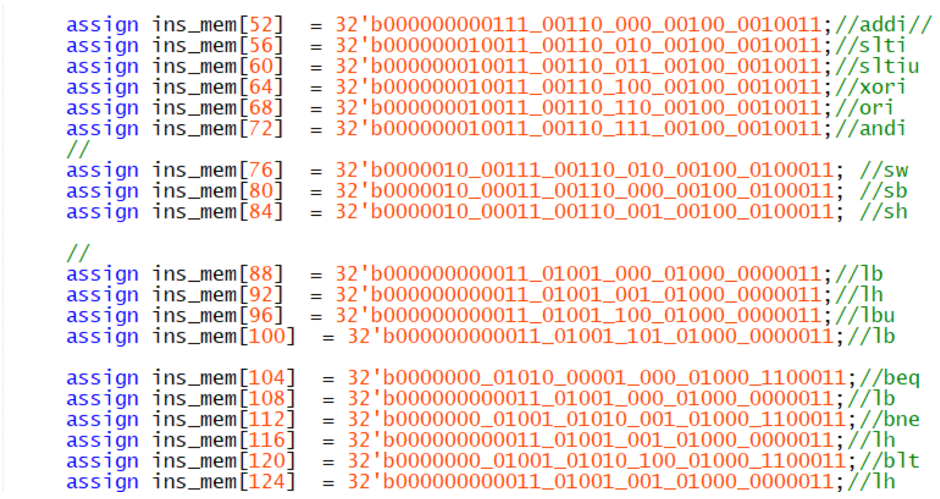
Testing and Simulation

The following instruction set run with modelsim. The results listed below.

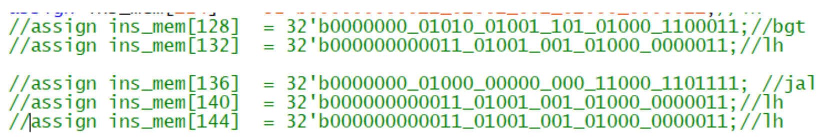
Instruction set:



```
assign ins_mem[0]   = 32'b000000000011_01001_010_00110_0000011; //lw
assign ins_mem[4]   = 32'b000000000100_01001_010_00111_0000011; //lw

assign ins_mem[8]   = 32'b0000000_00111_00110_000_01000_0110011; //add
assign ins_mem[12]  = 32'b0100000_00111_00110_000_00011_0110011; //sub
assign ins_mem[16]  = 32'b0000000_00111_00110_001_01011_0110011; //sll
assign ins_mem[20]  = 32'b0000000_00111_00110_010_01000_0110011; //slt
assign ins_mem[24]  = 32'b0000000_00111_00110_011_01111_0110011; //sltu
assign ins_mem[28]  = 32'b0000000_00111_00110_100_01011_0110011; // xor
assign ins_mem[32]  = 32'b0000000_00111_00110_101_01100_0110011;// srl
assign ins_mem[36]  = 32'b0100000_00111_00110_101_01101_0110011;//  sra
assign ins_mem[40]  = 32'b0000000_00111_00110_110_01011_0110011;//  or
assign ins_mem[44]  = 32'b0000000_00111_00110_111_01111_0110011;// and
assign ins_mem[48]  = 32'b0100000_00111_00110_111_01111_0110011;// mul
```

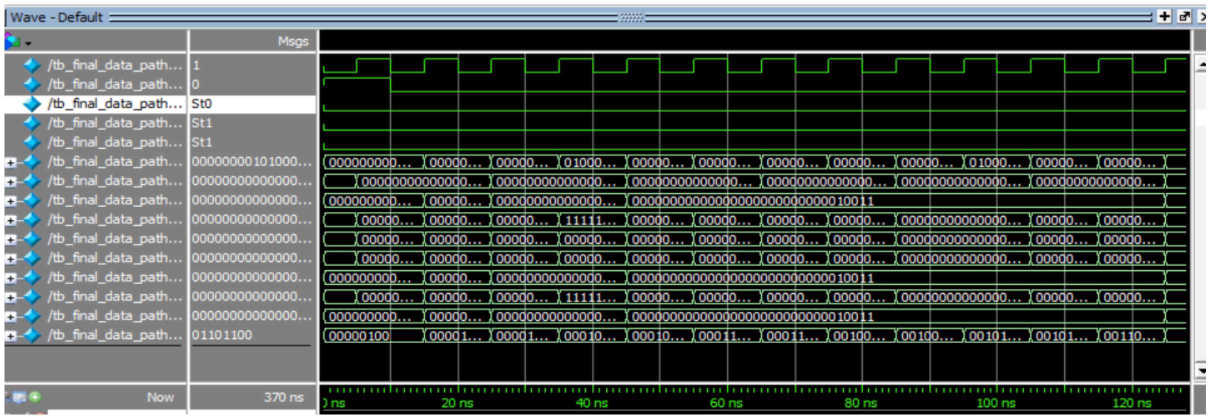


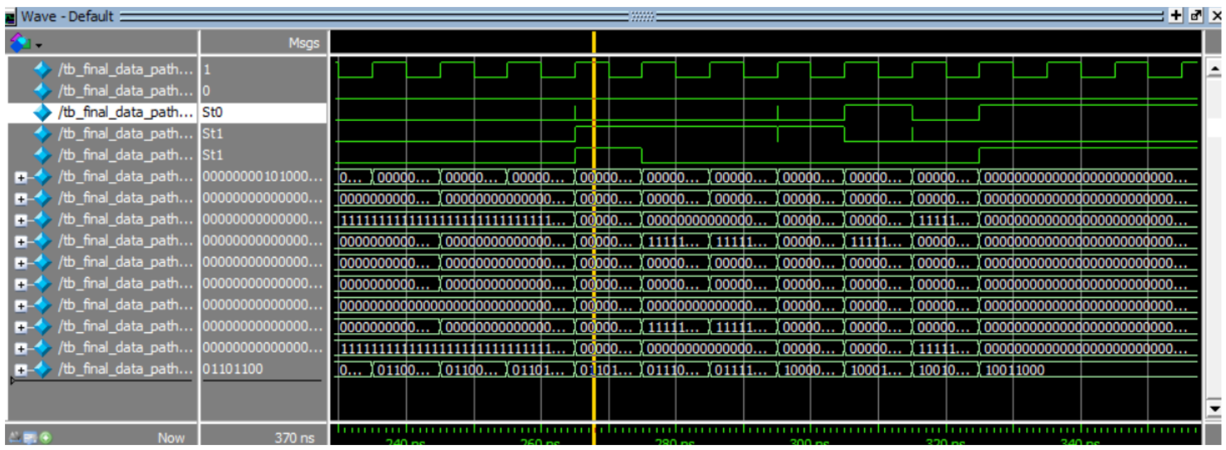
```
assign ins_mem[52]  = 32'b000000000111_00110_000_00100_0010011;//addi//
assign ins_mem[56]  = 32'b000000010011_00110_010_00100_0010011;//slti
assign ins_mem[60]  = 32'b000000010011_00110_011_00100_0010011;//sltiu
assign ins_mem[64]  = 32'b000000010011_00110_100_00100_0010011;//xori
assign ins_mem[68]  = 32'b000000010011_00110_110_00100_0010011;//ori
assign ins_mem[72]  = 32'b000000010011_00110_111_00100_0010011;//andi
//
assign ins_mem[76]  = 32'b0000010_00111_00110_010_00100_0100011; //sw
assign ins_mem[80]  = 32'b0000010_00011_00110_000_00100_0100011; //sb
assign ins_mem[84]  = 32'b0000010_00011_00110_001_00100_0100011; //sh

//
assign ins_mem[88]  = 32'b000000000011_01001_000_01000_0000011;//lb
assign ins_mem[92]  = 32'b000000000011_01001_001_01000_0000011;//lh
assign ins_mem[96]  = 32'b000000000011_01001_100_01000_0000011;//lbu
assign ins_mem[100]  = 32'b000000000011_01001_101_01000_0000011;//lb

assign ins_mem[104]  = 32'b0000000_01010_00001_000_01000_1100011;//beq
assign ins_mem[108]  = 32'b000000000011_01001_000_01000_0000011;//lb
assign ins_mem[112]  = 32'b0000000_01001_01010_001_01000_1100011;//bne
assign ins_mem[116]  = 32'b000000000011_01001_001_01000_0000011;//lh
assign ins_mem[120]  = 32'b0000000_01001_01010_100_01000_1100011;//blt
assign ins_mem[124]  = 32'b000000000011_01001_001_01000_0000011;//lh
```



```
//assign ins_mem[128]  = 32'b0000000_01010_01001_101_01000_1100011;//bgt
//assign ins_mem[132]  = 32'b000000000011_01001_001_01000_0000011;//lh

//assign ins_mem[136]  = 32'b0000000_01000_00000_000_11000_1101111; //jal
//assign ins_mem[140]  = 32'b000000000011_01001_001_01000_0000011;//lh
//assign ins_mem[144]  = 32'b000000000011_01001_001_01000_0000011;//lh
```

Modelsim Results:

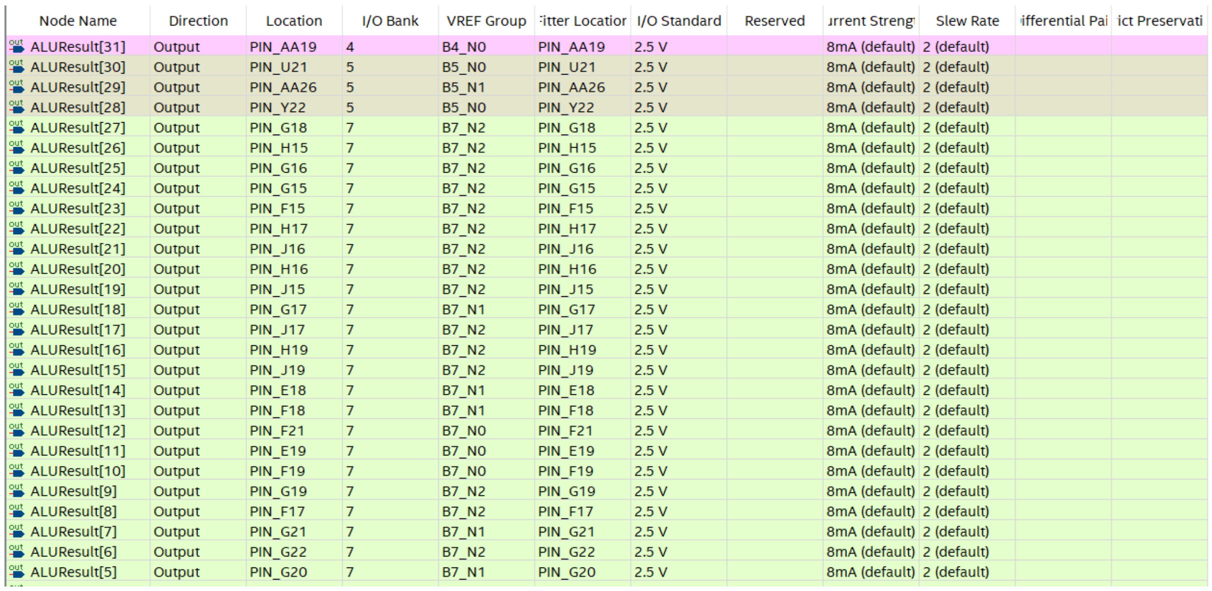

```
Wave - Default
                              Msgs
/tb_final_data_path...  1
/tb_final_data_path...  0
/tb_final_data_path...  St0
/tb_final_data_path...  St1
/tb_final_data_path...  St1
/tb_final_data_path...  00000000101000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  01101100
Now                     370 ns
```


```
Wave - Default
                              Msgs
/tb_final_data_path...  1
/tb_final_data_path...  0
/tb_final_data_path...  St0
/tb_final_data_path...  St1
/tb_final_data_path...  St1
/tb_final_data_path...  00000000101000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  00000000000000...
/tb_final_data_path...  01101100
Now                     370 ns
```

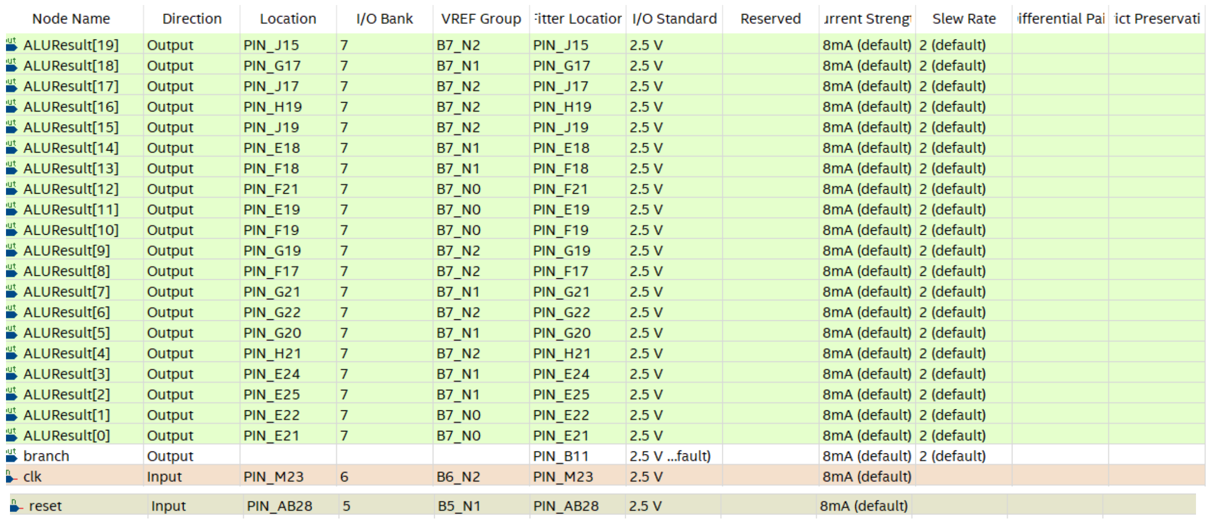
Demonstration :

For the demonstration of our processor, we have implemented processor on the Cyclone IV EP4CE115F29C7 FPGA board. For this we have used the board’s inbuilt features, push buttons, seven segment display and green, red LED indicators. I set clock as a push button and reset as a switch and AluResult as output LEDS.

Pin planner for the FPGA board:



Node Name	Direction	Location	I/O Bank	VREF Group	itter Locatior	I/O Standard	Reserved	urrent Streng	Slew Rate	ifferential Pai	ict Preservati
ALUResult[31]	Output	PIN_AA19	4	B4_N0	PIN_AA19	2.5 V		8mA (default)	2 (default)		
ALUResult[30]	Output	PIN_U21	5	B5_N0	PIN_U21	2.5 V		8mA (default)	2 (default)		
ALUResult[29]	Output	PIN_AA26	5	B5_N1	PIN_AA26	2.5 V		8mA (default)	2 (default)		
ALUResult[28]	Output	PIN_Y22	5	B5_N0	PIN_Y22	2.5 V		8mA (default)	2 (default)		
ALUResult[27]	Output	PIN_G18	7	B7_N2	PIN_G18	2.5 V		8mA (default)	2 (default)		
ALUResult[26]	Output	PIN_H15	7	B7_N2	PIN_H15	2.5 V		8mA (default)	2 (default)		
ALUResult[25]	Output	PIN_G16	7	B7_N2	PIN_G16	2.5 V		8mA (default)	2 (default)		
ALUResult[24]	Output	PIN_G15	7	B7_N2	PIN_G15	2.5 V		8mA (default)	2 (default)		
ALUResult[23]	Output	PIN_F15	7	B7_N2	PIN_F15	2.5 V		8mA (default)	2 (default)		
ALUResult[22]	Output	PIN_H17	7	B7_N2	PIN_H17	2.5 V		8mA (default)	2 (default)		
ALUResult[21]	Output	PIN_J16	7	B7_N2	PIN_J16	2.5 V		8mA (default)	2 (default)		
ALUResult[20]	Output	PIN_H16	7	B7_N2	PIN_H16	2.5 V		8mA (default)	2 (default)		
ALUResult[19]	Output	PIN_J15	7	B7_N2	PIN_J15	2.5 V		8mA (default)	2 (default)		
ALUResult[18]	Output	PIN_G17	7	B7_N1	PIN_G17	2.5 V		8mA (default)	2 (default)		
ALUResult[17]	Output	PIN_J17	7	B7_N2	PIN_J17	2.5 V		8mA (default)	2 (default)		
ALUResult[16]	Output	PIN_H19	7	B7_N2	PIN_H19	2.5 V		8mA (default)	2 (default)		
ALUResult[15]	Output	PIN_J19	7	B7_N2	PIN_J19	2.5 V		8mA (default)	2 (default)		
ALUResult[14]	Output	PIN_E18	7	B7_N1	PIN_E18	2.5 V		8mA (default)	2 (default)		
ALUResult[13]	Output	PIN_F18	7	B7_N1	PIN_F18	2.5 V		8mA (default)	2 (default)		
ALUResult[12]	Output	PIN_F21	7	B7_N0	PIN_F21	2.5 V		8mA (default)	2 (default)		
ALUResult[11]	Output	PIN_E19	7	B7_N0	PIN_E19	2.5 V		8mA (default)	2 (default)		
ALUResult[10]	Output	PIN_F19	7	B7_N0	PIN_F19	2.5 V		8mA (default)	2 (default)		
ALUResult[9]	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V		8mA (default)	2 (default)		
ALUResult[8]	Output	PIN_F17	7	B7_N2	PIN_F17	2.5 V		8mA (default)	2 (default)		
ALUResult[7]	Output	PIN_G21	7	B7_N1	PIN_G21	2.5 V		8mA (default)	2 (default)		
ALUResult[6]	Output	PIN_G22	7	B7_N2	PIN_G22	2.5 V		8mA (default)	2 (default)		
ALUResult[5]	Output	PIN_G20	7	B7_N1	PIN_G20	2.5 V		8mA (default)	2 (default)		



Node Name	Direction	Location	I/O Bank	VREF Group	itter Locatior	I/O Standard	Reserved	urrent Streng	Slew Rate	ifferential Pai	ict Preservati
ALUResult[19]	Output	PIN_J15	7	B7_N2	PIN_J15	2.5 V		8mA (default)	2 (default)		
ALUResult[18]	Output	PIN_G17	7	B7_N1	PIN_G17	2.5 V		8mA (default)	2 (default)		
ALUResult[17]	Output	PIN_J17	7	B7_N2	PIN_J17	2.5 V		8mA (default)	2 (default)		
ALUResult[16]	Output	PIN_H19	7	B7_N2	PIN_H19	2.5 V		8mA (default)	2 (default)		
ALUResult[15]	Output	PIN_J19	7	B7_N2	PIN_J19	2.5 V		8mA (default)	2 (default)		
ALUResult[14]	Output	PIN_E18	7	B7_N1	PIN_E18	2.5 V		8mA (default)	2 (default)		
ALUResult[13]	Output	PIN_F18	7	B7_N1	PIN_F18	2.5 V		8mA (default)	2 (default)		
ALUResult[12]	Output	PIN_F21	7	B7_N0	PIN_F21	2.5 V		8mA (default)	2 (default)		
ALUResult[11]	Output	PIN_E19	7	B7_N0	PIN_E19	2.5 V		8mA (default)	2 (default)		
ALUResult[10]	Output	PIN_F19	7	B7_N0	PIN_F19	2.5 V		8mA (default)	2 (default)		
ALUResult[9]	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V		8mA (default)	2 (default)		
ALUResult[8]	Output	PIN_F17	7	B7_N2	PIN_F17	2.5 V		8mA (default)	2 (default)		
ALUResult[7]	Output	PIN_G21	7	B7_N1	PIN_G21	2.5 V		8mA (default)	2 (default)		
ALUResult[6]	Output	PIN_G22	7	B7_N2	PIN_G22	2.5 V		8mA (default)	2 (default)		
ALUResult[5]	Output	PIN_G20	7	B7_N1	PIN_G20	2.5 V		8mA (default)	2 (default)		
ALUResult[4]	Output	PIN_H21	7	B7_N2	PIN_H21	2.5 V		8mA (default)	2 (default)		
ALUResult[3]	Output	PIN_E24	7	B7_N1	PIN_E24	2.5 V		8mA (default)	2 (default)		
ALUResult[2]	Output	PIN_E25	7	B7_N1	PIN_E25	2.5 V		8mA (default)	2 (default)		
ALUResult[1]	Output	PIN_E22	7	B7_N0	PIN_E22	2.5 V		8mA (default)	2 (default)		
ALUResult[0]	Output	PIN_E21	7	B7_N0	PIN_E21	2.5 V		8mA (default)	2 (default)		
branch	Output				PIN_B11	2.5 V ...fault)		8mA (default)	2 (default)		
clk	Input	PIN_M23	6	B6_N2	PIN_M23	2.5 V		8mA (default)			
reset	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V		8mA (default)			

Explannation:

First I stored data in the following locations and following values :

……………………..,10010 – 10010 , 10011 – 10011, ……………………………..

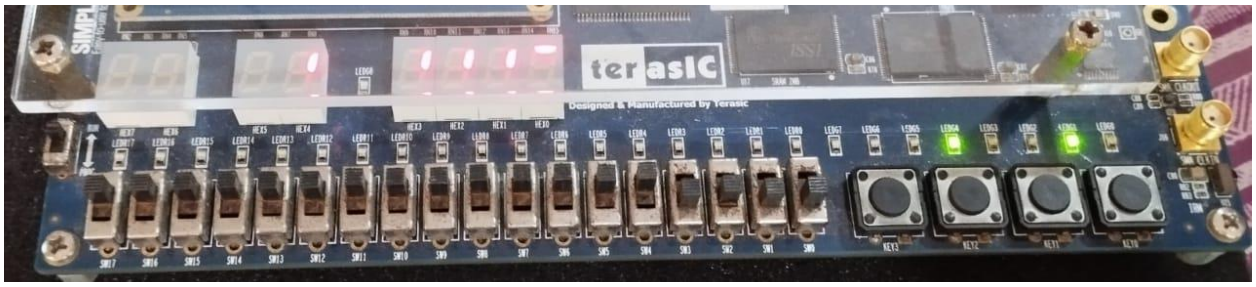
1. Lw instruction : 000000000011_01001_010_00110_0000011

   Rs1 – 1001 rd = 110

   Imm Value – 11

   ALUResult = 11 + 1111 = 10010 (This shown in below)

   Read data = 10010 (store this in 110 register)



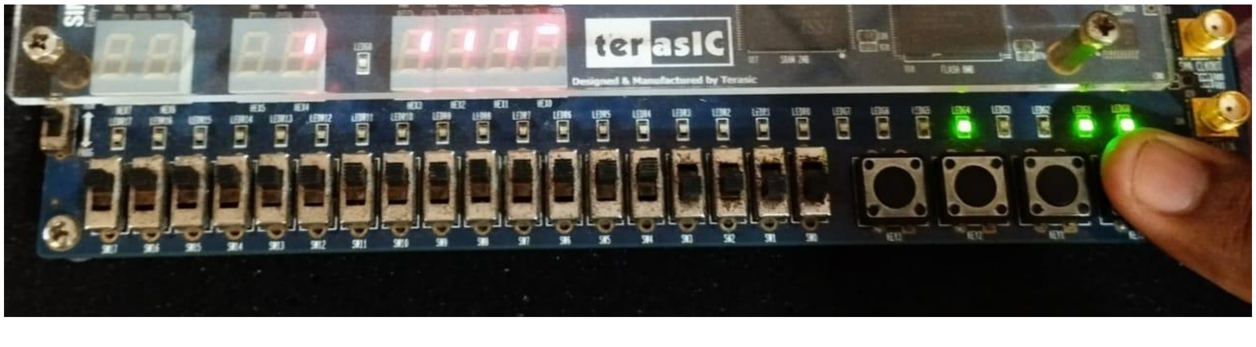
2. Lw instruction : 000000000100_01001_010_00111_0000011

   Rs1 – 1001 rd = 111

   Imm Value – 100

   ALUResult = 100 + 1111 = 10011 (This shown in below)

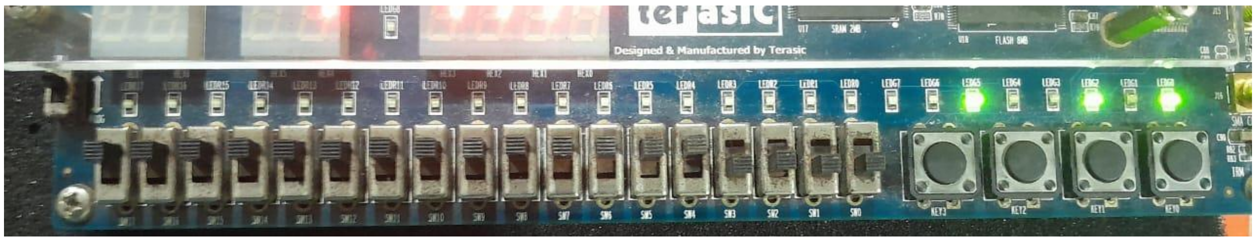
   Read data = 10011 (store this in 111 register)



3. Add instruction : 0000000_00111_00110_000_01000_0110011
   Rs1 – 110 (store value 10010)
   Rs2 - 111 (store value 10011)

AluResult – 10010 + 10011 = 100101 (This shown in below)

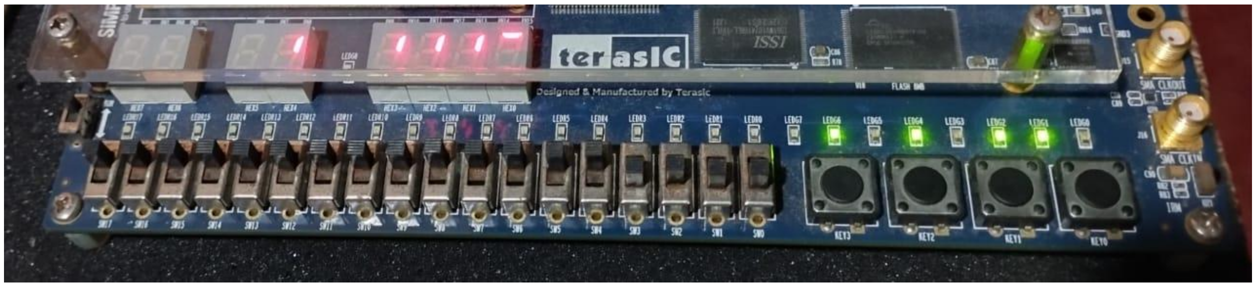
Rd – 1000 (store alu result in this register)



4. Mul Instruction : 0100000_00111_00110_111_01111_0110011
   Rs1 – 110 (store value 10010)
   Rs2 - 111 (store value 10011)

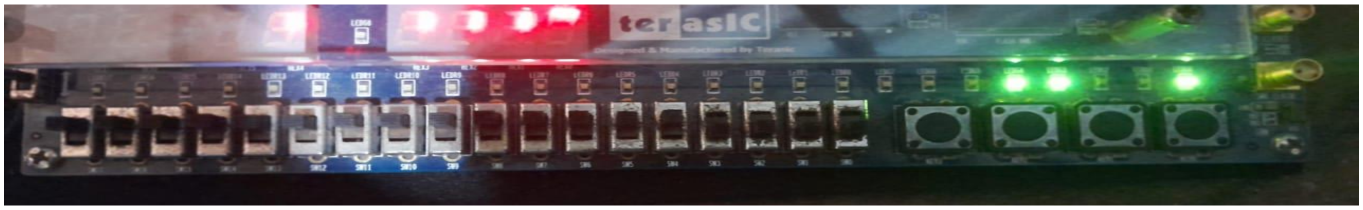
   AluResult – 10010 * 10011 = 101010110 (This shown in below)

   Rd – 1111 (store alu result in this register)



Same as above all R type instruction execute correctly give correct LED output like above.

5. ADDi instruction : 000000000111_00110_000_00100_0010011
   Rs1 – 110 (store value 10010)
   Rd – 100
   Immediate – 111

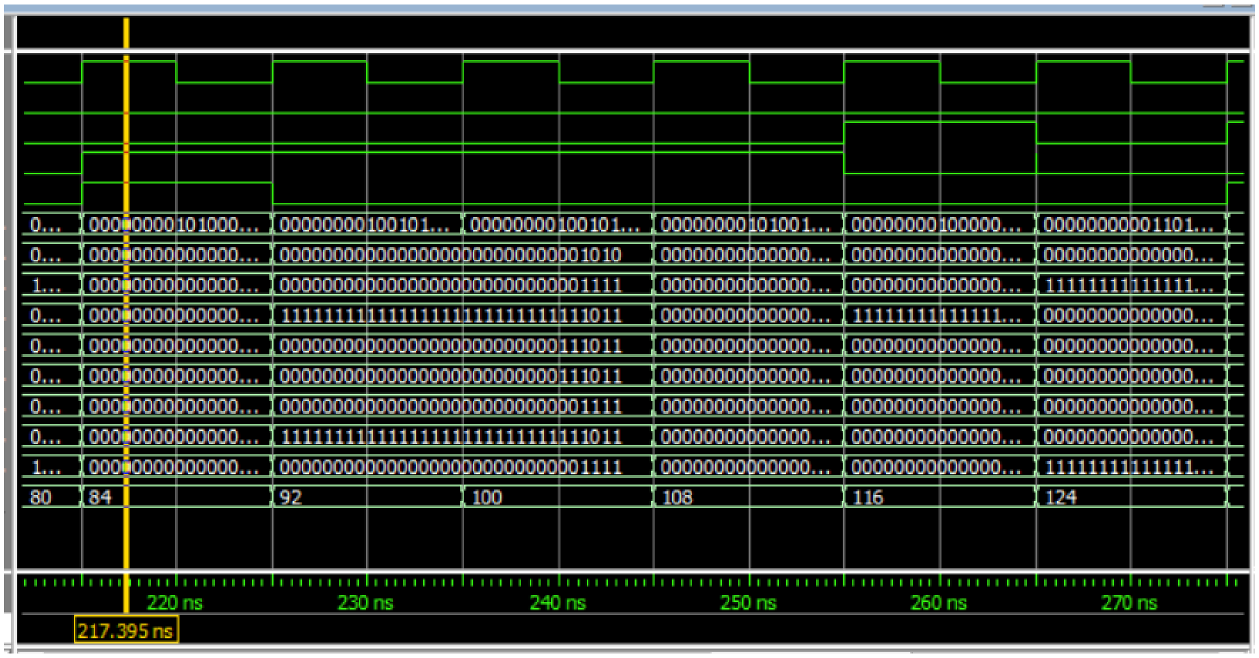


ALUResult – 11001

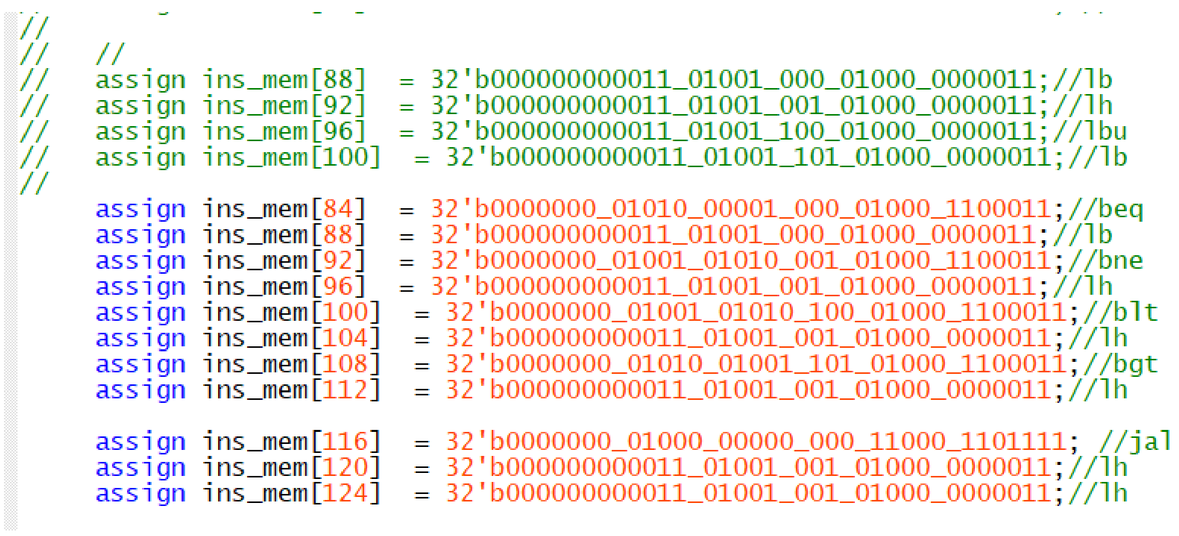
Same as above all I Type instructions execute correctly and give correct LED outputs .

6. BEQ instruction :

I set all the branch instructions (BEQ , BNE , BLT , BGT , BGTU , BLTU) to branch by 8(immediate).



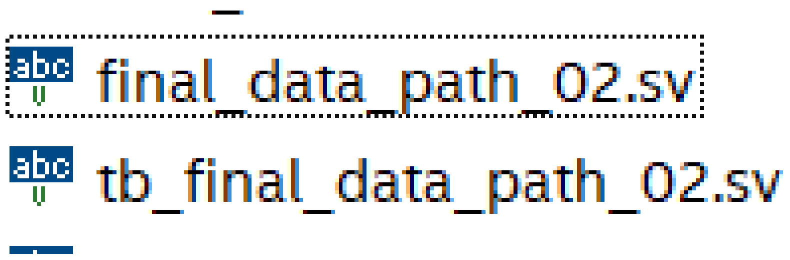
These are the branch instruction you can see pc increment by 8 and I listed instructions below.



```
//
//    //
//    assign ins_mem[88]  = 32'b000000000011_01001_000_01000_0000011;//lb
//    assign ins_mem[92]  = 32'b000000000011_01001_001_01000_0000011;//lh
//    assign ins_mem[96]  = 32'b000000000011_01001_100_01000_0000011;//lbu
//    assign ins_mem[100]  = 32'b000000000011_01001_101_01000_0000011;//lb
//
      assign ins_mem[84]  = 32'b0000000_01010_00001_000_01000_1100011;//beq
      assign ins_mem[88]  = 32'b000000000011_01001_000_01000_0000011;//lb
      assign ins_mem[92]  = 32'b0000000_01001_01010_001_01000_1100011;//bne
      assign ins_mem[96]  = 32'b000000000011_01001_001_01000_0000011;//lh
      assign ins_mem[100]  = 32'b0000000_01001_01010_100_01000_1100011;//blt
      assign ins_mem[104]  = 32'b000000000011_01001_001_01000_0000011;//lh
      assign ins_mem[108]  = 32'b0000000_01010_01001_101_01000_1100011;//bgt
      assign ins_mem[112]  = 32'b000000000011_01001_001_01000_0000011;//lh

      assign ins_mem[116]  = 32'b0000000_01000_00000_000_11000_1101111; //jal
      assign ins_mem[120]  = 32'b000000000011_01001_001_01000_0000011;//lh
      assign ins_mem[124]  = 32'b000000000011_01001_001_01000_0000011;//lh
```

So branch instruction also working correctly.

- **For see simulation results run final_data_path_02 file with its testbench file named tb_final_dta_path_02.**



final_data_path_02.sv
tb_final_data_path_02.sv