DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING
UNIVERSITY OF MORATUWA

EN3021 - Digital System Design

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

S.H.D.Himeka 200222K

October 15 , 2023

Table of Contents

Abstract

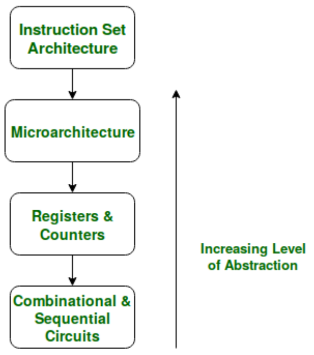
The performance and capabilities of a computer are largely dependent on the processor design. Everything from smartphones to supercomputers benefit from efficient design in terms of increased speed, reduced power consumption, and general functionality. Innovations in design enable advancements, makes it vital in this ever-evolving field.

Introduction

This report offers a detailed explanation of the design and hardware implementation of a custom processor in RISC V architecture using an Alterra DE-2 board Field Programmable Gate Arrays. All the program files, codes testbenches are attached to the appendices.

Processor design

Processor designing is creating fundamental hardware components of the computers central processing unit(CPU), which includes its architecture, Instruction set, microarchitectures and circuitry deciding data paths, controls and memory. Designers aim to improve its performance, power efficiency, and software compatibility. Producing CPUs that satisfy the ever-increasing demands of modern computing is a hard and needs deep knowledge of digital logic, circuit design, and computer architecture.


Instruction Set Architecture
Microarchitecture
Registers & Counters
Combinational & Sequential Circuits
Increasing Level of Abstraction

levels of abstraction in computer architecture form a hierarchical framework. The ISA defines the interface for software, the microarchitecture defines the implementation details, registers and counters manage state and control, and combinational logic circuits perform the actual operations.

Instruction Set Architectures

The set of instructions, which the CPU is built on able to understand and execute is simply the instruction Set Architecture. The programs running on a processor compiled into a set of low level instructions called assembly language, is a part of the instruction set architecture. The ISA specifies the sorts of instructions that the processor will support.

Instructions for Arithmetic/Logic: These Instructions carry out a variety of Arithmetic and Logical operations on one or more operands.

Instructions for Data Transfer: These instructions are in charge of moving instructions from memory to processor registers and vice versa.

Instructions for Branches and Jumps: These instructions are in charge of interrupting the sequential flow of instructions and jumping to instructions in different locations, which is required for the implementation of functions and conditional statements.

They may have fixed lengths or variable lengths.

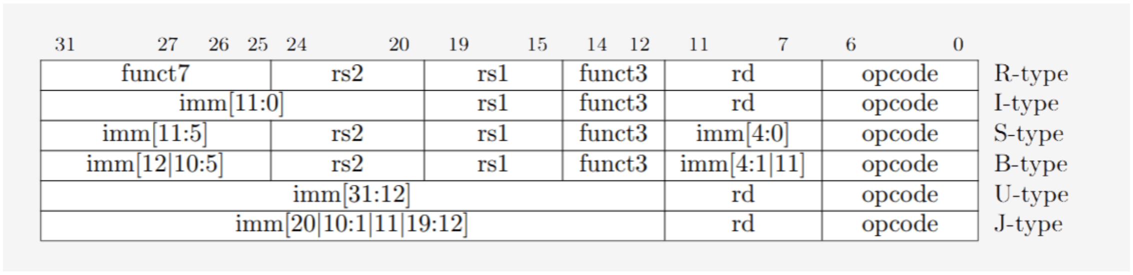
Some of the most common ISA s are x86, MIPS, ARM, RISC-V, and PowerPC. The ISA chosen for a specific computer system is determined by its intended usage, performance requirements, and other design factors.

RISC V ISA

RISC-V is an open-source Instruction Set Architecture (ISA) that is renowned for its ease of use, flexibility, and adaptability. Because it is open source, anybody may use and adapt it without paying license costs, encouraging cooperation and innovation. The RISC-V architecture adheres to the reduced instruction set (RISC) principle, with a small and orthogonal set of instructions that simplifies both compiler and hardware design. Scalability with multiple data register widths is provided by standardized base versions such as the RV32I and RV64I. RISC Vs modular framework allows users to add custom extensions, allowing them to modify the architecture to suit application requirements.

The RISC-V ISA uses 32 registers, 31 of which are general purpose registers (registers x1 to x31), and one register (register x0) is hardware connected to a constant 0. In addition to the 32 registers, a unique register known as the PC register is utilized to hold the address of the current instruction. Depending on the implementation, the register width might be 32 bits, 64 bits, or 128 bits.

The R-type (register-register instructions), I-type (loads and jump-and-link instructions), S-type (store instructions), B-type (branch instructions), U-type (load upper-immediate instructions), and J-type (jump instructions) instruction types compose the RISC-V ISA.



31 27 26 25	24 20	19 15	14 12	11 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|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

General Flow of RISC V ISA- Instruction Life Cycle

The five steps that make up a RISC-V processor's typical operation are,

- Instruction fetch
- Decode
- Execute
- Memory access
- Writeback.

The PC outputs the address to the instruction memory, which makes the 32-bit instruction or machine code available, during the instruction fetch stage. The instruction code is gathered during the decoding stage and sent as control signals to various modules. The various modules, including the ALU, register file, and Data Modification Module, are used to carry out the instruction during the execute stage. Depending on the type of instruction, data is either read from or stored in the data memory during the memory access stage. Depending on the instructions, the writeback stage stores a value in the register file.

- R-type instructions:

This instruction type has an opcode of 0110011.These mainly involve register operations. The relevant values in rs1 and rs2 registers in register files are fetched to the alu, relevant alu operation is carried out there, and the result stored back to the rd register in register files.

- I type instructions:

This instruction type has an opcode of 0100011.The register value in rs1 and the given immediate value sign extended to 32 bits are fetched to the alu, relevant alu operation is carried out there, and the result is stored back to the rd register in register files.

- Load instructions (I type) :

This instruction type has an opcode of 0000011.The register value in rs1 and the given immediate value sign extended to 32 bits (offset) are fetched to the alu, add operation is carried out to find the memory location of target. According to the func3; the data in that memory location is stored back to the rd register in register files to the relevant lengths.

- Store instructions (SW type):

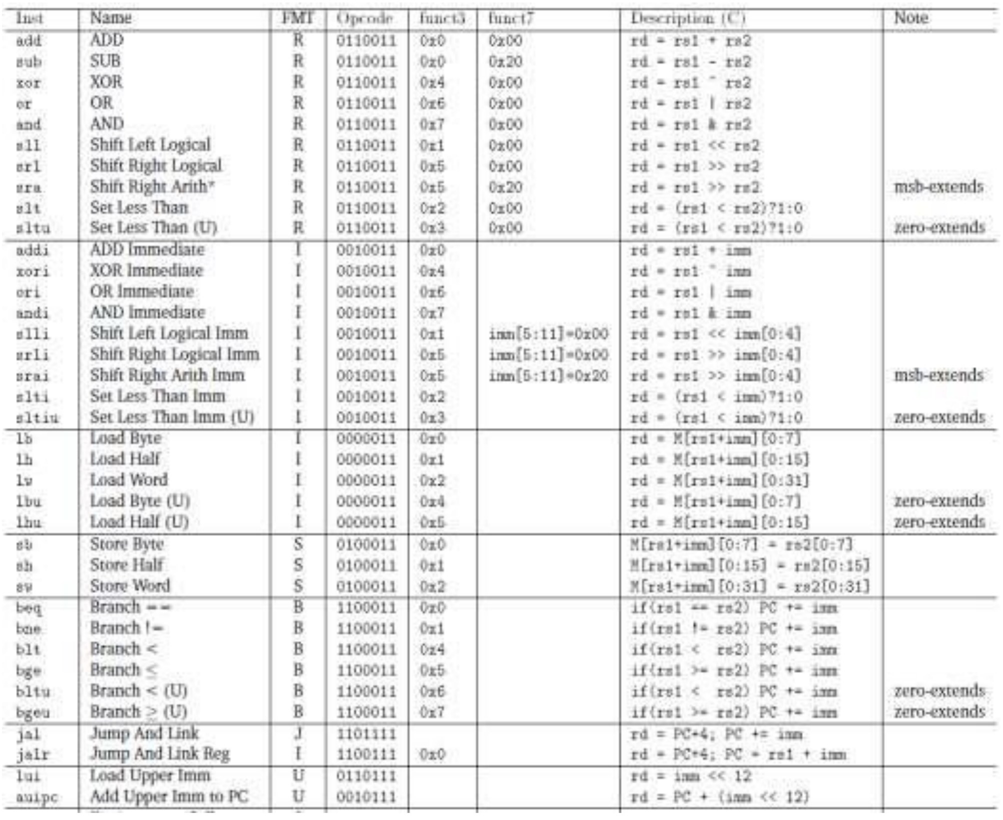
This instruction type has an opcode of 0100011.The register value in rs1 and the given immediate value sign extended to 32 bits (offset) are fetched to the alu, add operation is carried out to find the memory location of target. The data in that rs2 register in register files is then stored to that memory location up to relevant lengths given by func3.

- Branch instructions (B type):

This instruction type has an opcode of 1100011.The register value in rs1 and rs2 are fetched to the alu and compared according to given instruction by func3. If the comparisons satisfied given immediate value sign extended to 32 bits (offset) is added to the program counter value which determines the instruction to be implemented next.

- JALR (I type) instruction:

This instruction type has an opcode of 1100111. The register value in rs1 and the given immediate value sign extended to 32 bits (offset) are fetched to the alu, add operation is carried. The result is added to the program counter value which determines the instruction to be implemented next. In the same time the next instruction which was to be implemented (PC+4 value) is stored in rd register of register files.



Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≤	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

Microarchitectures

It is the layer that sits between the Instruction Set Architecture (ISA) and the physical hardware, controlling how the ISA's instructions are performed and how data flows through the CPU.

Pipelines, data pathways, control units, caches, and memory hierarchies are all components of microarchitectures. Microarchitecture design decisions have a direct influence on a processor's performance, power efficiency, and capabilities. Different microarchitectures can run the same ISA, but their speed, energy consumption, and overall efficiency may differ. Pipelining, which divides instruction execution into phases, is a key feature of microarchitectures.

About Controls

Controls in a computer's brain, known as the central processing unit (CPU), play a crucial role in managing how tasks are carried out. There are two main ways to control these tasks: hardwired control and microprogramming.

Hardwired Control

The control signals here are generated by combinational logics or finite state machines. It uses hardware components like decoders, multiplexers, gates to produce control signals based on the instructions that are being executed.

These are fast but can be complex and challenging when designing processors with complex ISAs.

Microprogramming

This is a more flexible approach which uses microinstructions in the controls. The control unit here simply acts as a memory which stores and gives out the relevant signals for the microinstructions. This approach is easily modifiable and well suited for complex ISA implementation.

Task Given

Introduce the task

The main task of the given project is to design a 32 bit non-pipelined RISC-V processor using Microprogramming with 3 bus structure. This should be RV32I implementation.

The following types of instructions are to be specifically implemented.

1. All computational instructions covered by instruction types R and I.

2. All memory access instructions (load and store) - I and S type instructions

3. All Control Flow instructions : SB type

And these 2 new instructions also needs to be implemented

(a) MEMCOPY - Copies an array of size N from one location to another. N > 1

- Determine the max N that you can use for this instruction)

(b) MUL - Unsigned Multiplication (Note that RV32I does not include multiplication instructions)

- Identify the limits for operands of this instruction

Task Breakdown

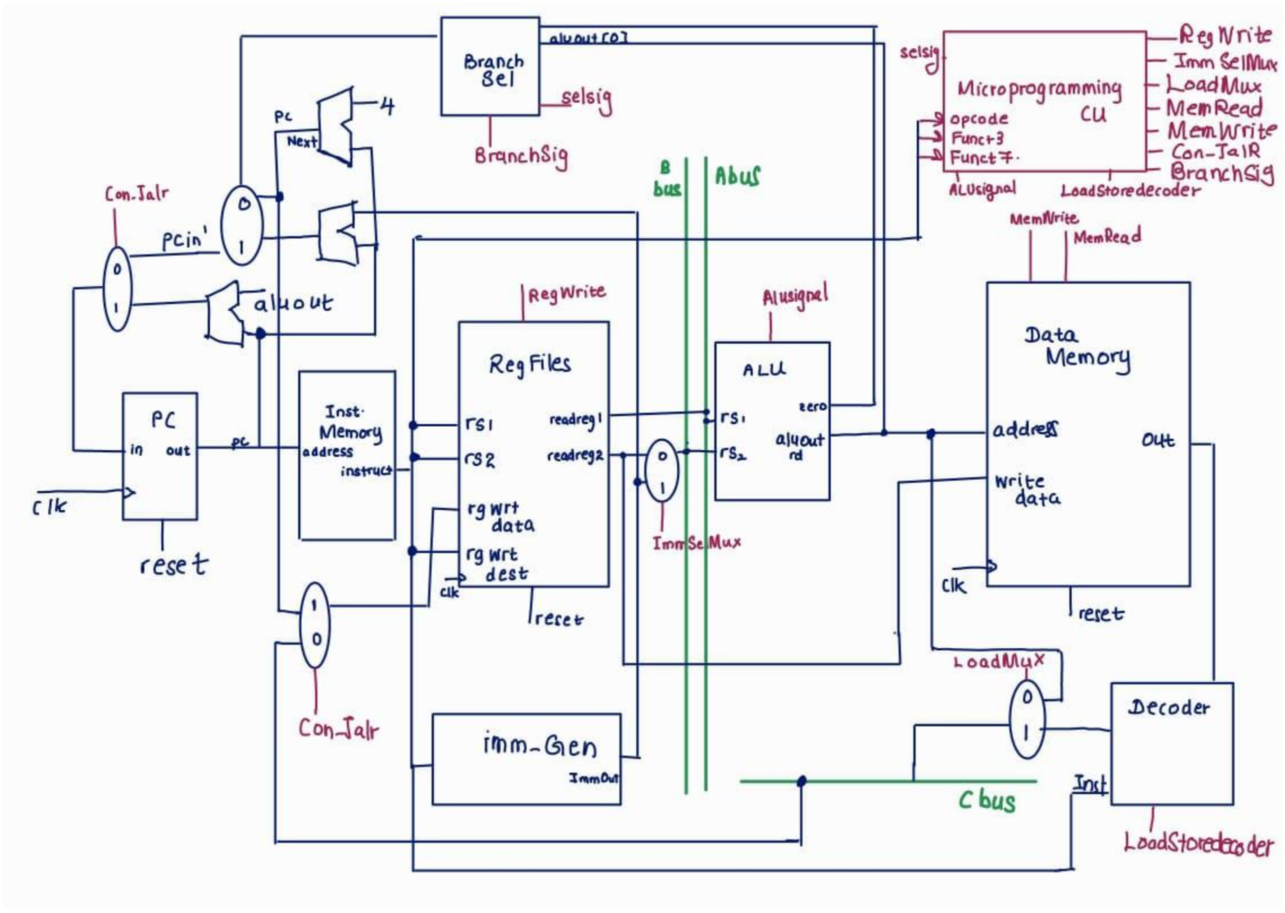
1. Design of the Instruction Set Architecture (ISA):
   - Describe the R-type, I-type, S-type, and SB-type instructions that are part of the RV32I instruction set.
   - Specify the format, operation, and encoding for the new MEMCOPY and MUL instructions.
2. Design for Microprogramming:
   - To specify the control signals for each stage of execution, creating microinstructions for each instruction.
   - To interpret microinstructions and manage the processor, implement a microinstruction control unit.
3. Data Path Planning:
   - The memory interface, buses, registers, and ALU (Arithmetic Logic Unit) to create the processor's data path.
   - Verify that all instruction types, including the new MEMCOPY and MUL instructions, can be accommodated on the data path.
4. Coding, testing and simulation results.
   - Code each module and Datapath
   - For each type of instruction, should create a thorough set of test cases that includes boundary conditions and corner cases.
   - Simulate the execution of instructions to ensure that the processor is functioning correctly.

My Approach

Here I used System Verilog Hardware Description Language (HDL), and implementations are performed using 'Quartus' II online edition version 20.1 along with an Altera DE2-115 Education and Development board equipped with a Cyclone IV FPGA.

The architecture – The Design of ISA

Since we were asked to use RISC V ISA with 3 bus architecture; I first drew the modules which I'm going to use, and connected them according to the architecture.

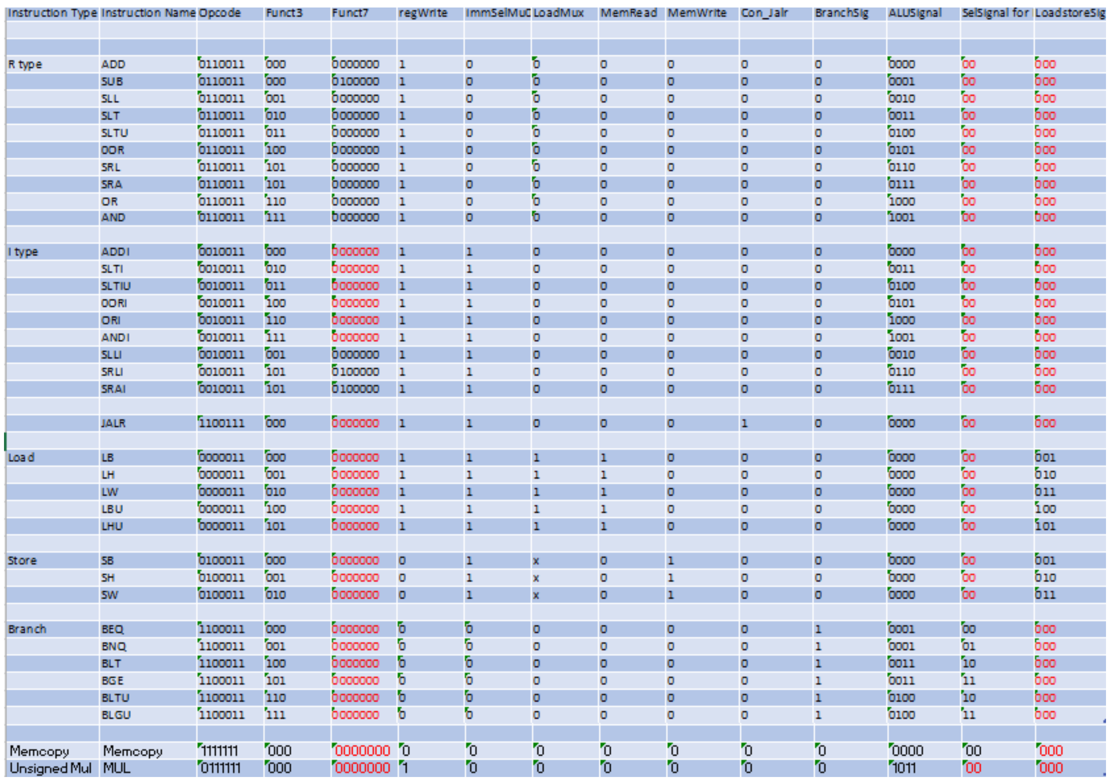


Briefing about this; I've used main modules namely pc, Instruction memory, Reg Files, ALU, Data Memory, Decoder for Load Store instructions, Branch signal selector and Microprogramming Control Unit. I have used 3 adders and 5 Multiplexers as well. The purpose of all these modules and components will be discussed later.

Since we needed to include a 3-bus architecture, I've added the register1 value and register2 values from the register Files into the A and B buses respectively. The output of the LoadMux Multiplexer was then sent to the C bus.

Design of Micro programming

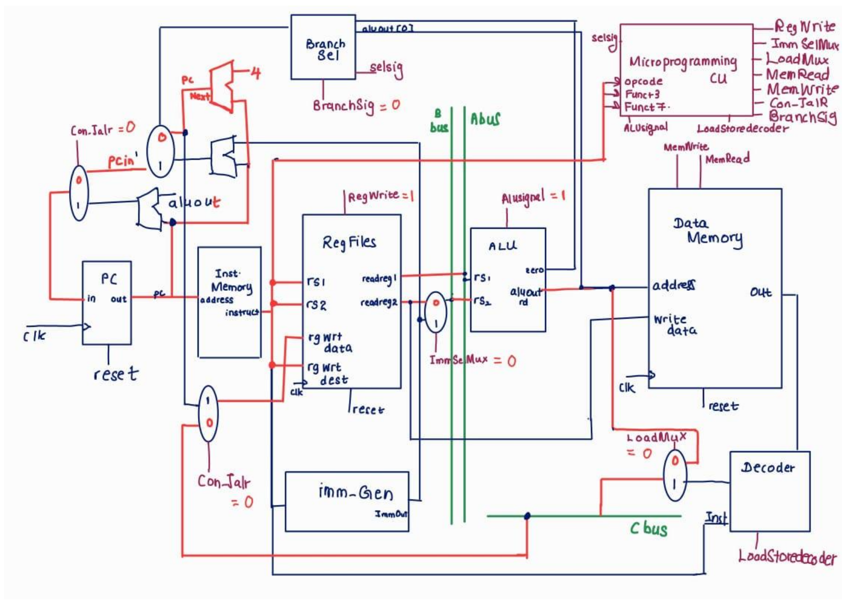
In the Control unit I've used the microprogrammed control approach. For that first I tabulated all the relevant signals for all the instruction types.



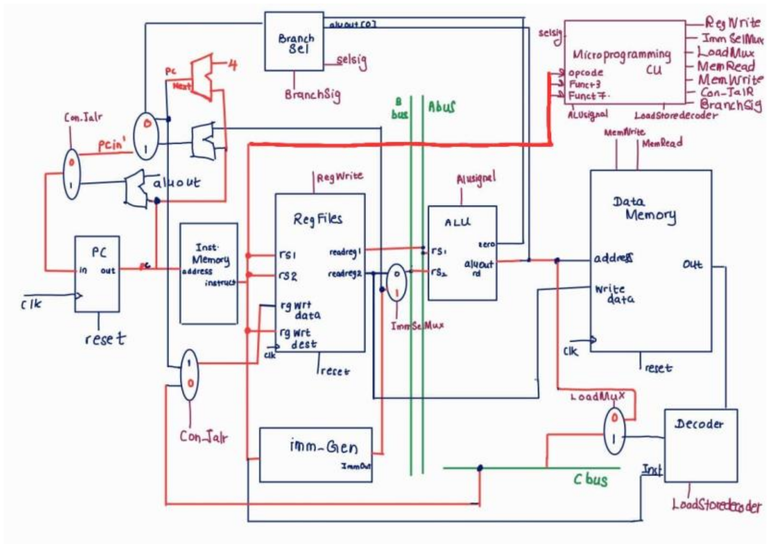
Instruction Type	Instruction Name	Opcode	Funct3	Funct7	regWrite	ImmSelMu(	LoadMux	MemRead	MemWrite	Con_Jalr	BranchSig	ALUSignal	SelSignal for	LoadstoreSig
R type	ADD	0110011	000	0000000	1	0	0	0	0	0	0	0000	00	000
	SUB	0110011	000	0100000	1	0	0	0	0	0	0	0001	00	000
	SLL	0110011	001	0000000	1	0	0	0	0	0	0	0010	00	000
	SLT	0110011	010	0000000	1	0	0	0	0	0	0	0011	00	000
	SLTU	0110011	011	0000000	1	0	0	0	0	0	0	0100	00	000
	OOR	0110011	100	0000000	1	0	0	0	0	0	0	0101	00	000
	SRL	0110011	101	0000000	1	0	0	0	0	0	0	0110	00	000
	SRA	0110011	101	0000000	1	0	0	0	0	0	0	0111	00	000
	OR	0110011	110	0000000	1	0	0	0	0	0	0	1000	00	000
	AND	0110011	111	0000000	1	0	0	0	0	0	0	1001	00	000
I type	ADDI	0010011	000	0000000	1	1	0	0	0	0	0	0000	00	000
	SLTI	0010011	010	0000000	1	1	0	0	0	0	0	0011	00	000
	SLTIU	0010011	011	0000000	1	1	0	0	0	0	0	0100	00	000
	OORI	0010011	100	0000000	1	1	0	0	0	0	0	0101	00	000
	ORI	0010011	110	0000000	1	1	0	0	0	0	0	1000	00	000
	ANDI	0010011	111	0000000	1	1	0	0	0	0	0	1001	00	000
	SLLI	0010011	001	0000000	1	1	0	0	0	0	0	0010	00	000
	SRLI	0010011	101	0100000	1	1	0	0	0	0	0	0110	00	000
	SRAI	0010011	101	0100000	1	1	0	0	0	0	0	0111	00	000
	JALR	1100111	000	0000000	1	1	0	0	0	1	0	0000	00	000
Load	LB	0000011	000	0000000	1	1	1	1	0	0	0	0000	00	001
	LH	0000011	001	0000000	1	1	1	1	0	0	0	0000	00	010
	LW	0000011	010	0000000	1	1	1	1	0	0	0	0000	00	011
	LBU	0000011	100	0000000	1	1	1	1	0	0	0	0000	00	100
	LHU	0000011	101	0000000	1	1	1	1	0	0	0	0000	00	101
Store	SB	0100011	000	0000000	0	1	x	0	1	0	0	0000	00	001
	SH	0100011	001	0000000	0	1	x	0	1	0	0	0000	00	010
	SW	0100011	010	0000000	0	1	x	0	1	0	0	0000	00	011
Branch	BEQ	1100011	000	0000000	0	0	0	0	0	0	1	0001	00	000
	BNQ	1100011	001	0000000	0	0	0	0	0	0	1	0001	01	000
	BLT	1100011	100	0000000	0	0	0	0	0	0	1	0011	10	000
	BGE	1100011	101	0000000	0	0	0	0	0	0	1	0011	11	000
	BLTU	1100011	110	0000000	0	0	0	0	0	0	1	0100	10	000
	BLGU	1100011	111	0000000	0	0	0	0	0	0	1	0100	11	000
Memcopy	Memcopy	1111111	000	0000000	0	0	0	0	0	0	0	0000	00	000
Unsigned Mul	MUL	0111111	000	0000000	1	0	0	0	0	0	0	1011	00	000

Datapath Design

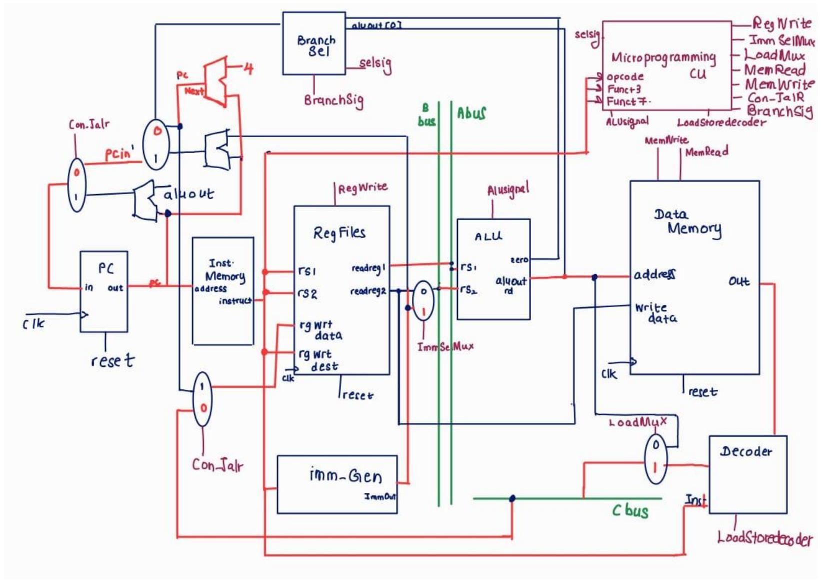
Starting from the control signals, I have developed a Datapath for each instruction type according to the architecture I've drawn previously.



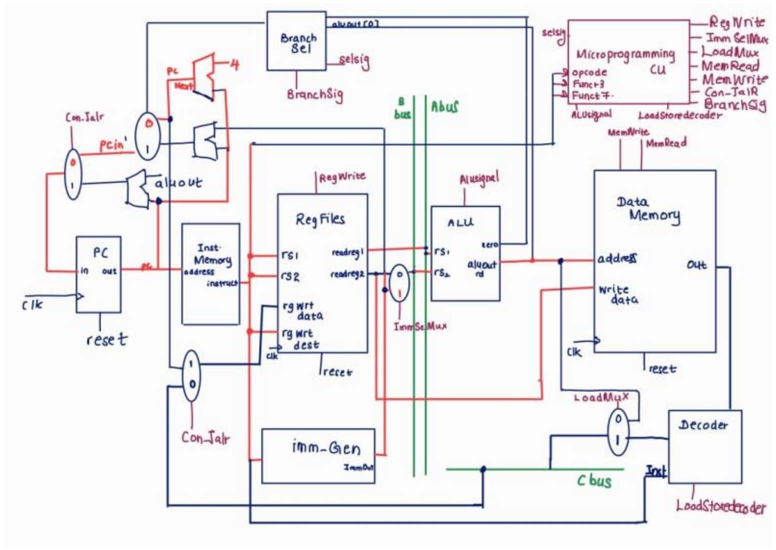
R type and MUL Datapath



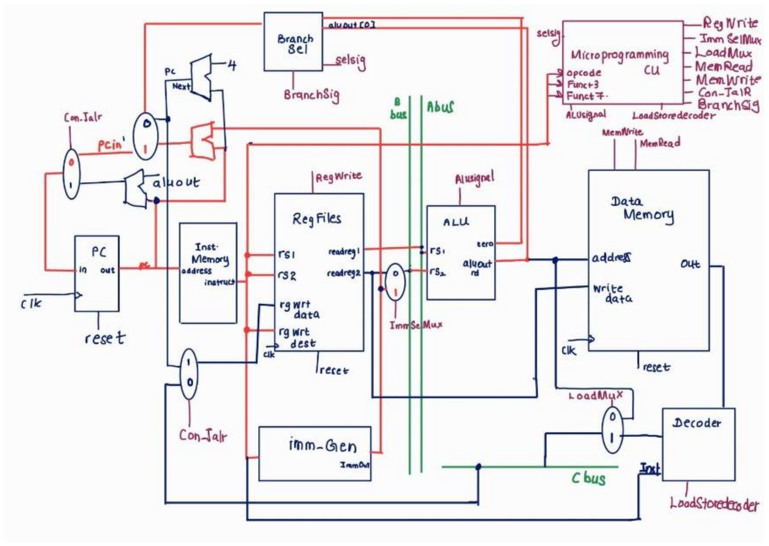
Immediate type Datapath.



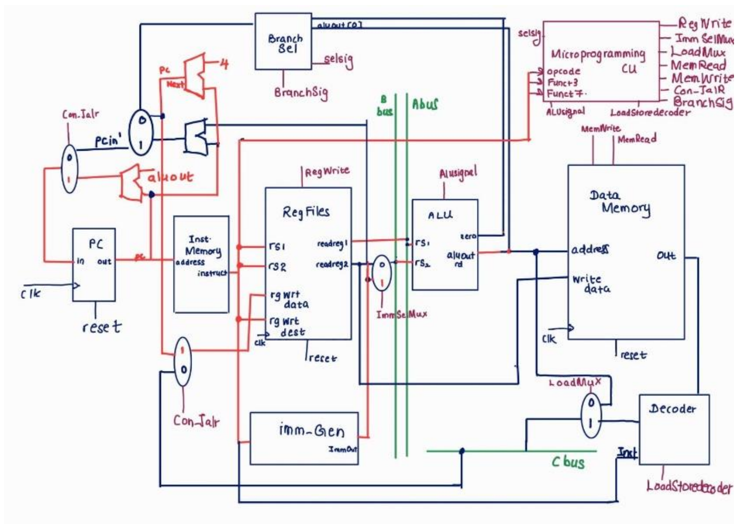
Load instruction Datapath



Store Instruction Datapath

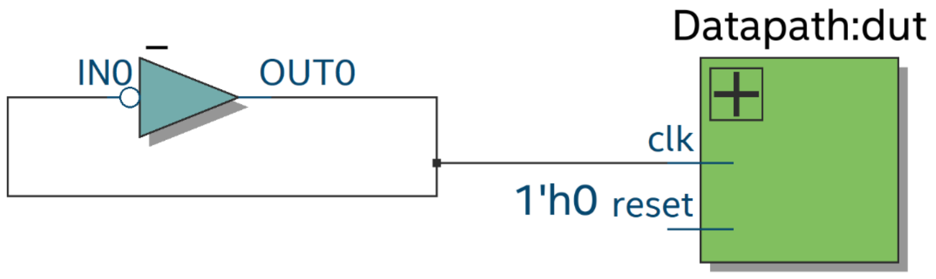


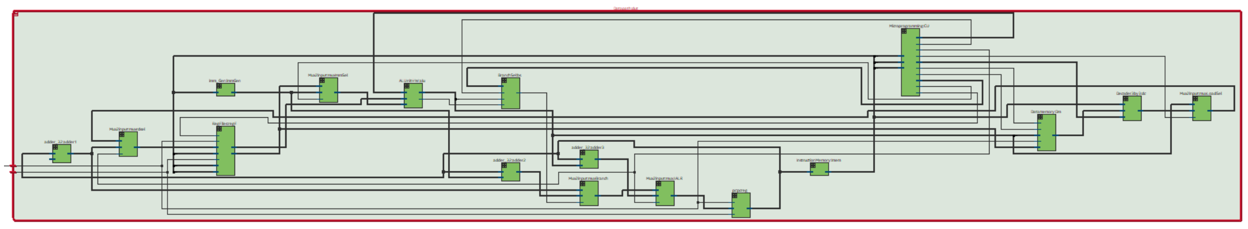
Branch type Datapath



JALR Datapath

Netlist viewer





Coding, Testing and Simulation results.

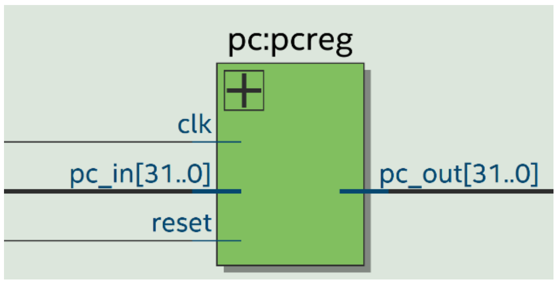
While coding the different entities, I developed testbenches for each module and tested them individually.

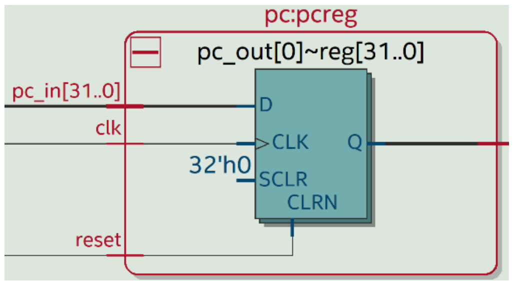
Modules and Components

Modules:

PC register:

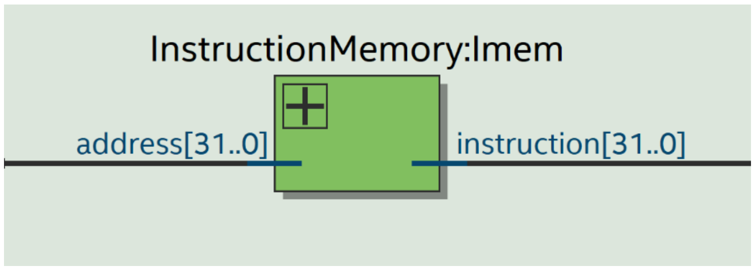
The PC (Program Counter) Register is a storage component that holds the memory address of the next instruction to be fetched and executed in a CPU. It has a data width of 32-bits. The PC is usually increased by 4 during the normal instruction execution. This increased by a factor 4 is because the riscV processor we are designing is byte addressable.
The PC register points to the calculated value(pc_out +aluout) of the jump addresses when J-type (jump), (pc_out+4) when U-type (load upper-immediate), and (pc_out+immediate value) B-type (branch) instructions are executed.

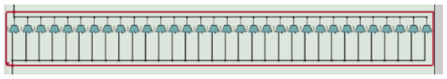




This is sequential block is programmed to get updated on the clock or reset signal's rising edge. The program counter is set to zero when the "reset" signal is active; otherwise, out value is left with the value specified by input.

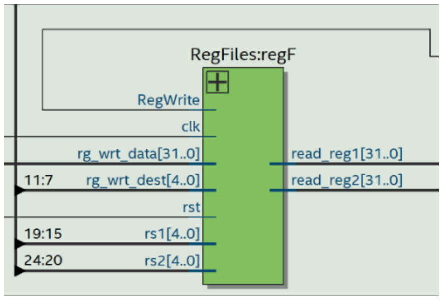
Instruction Memory:



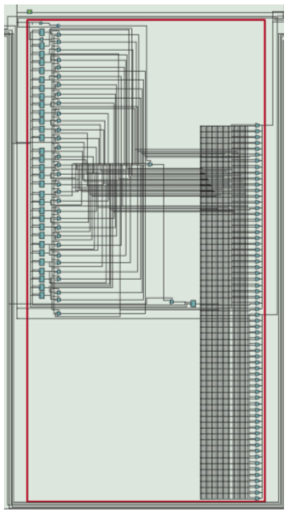


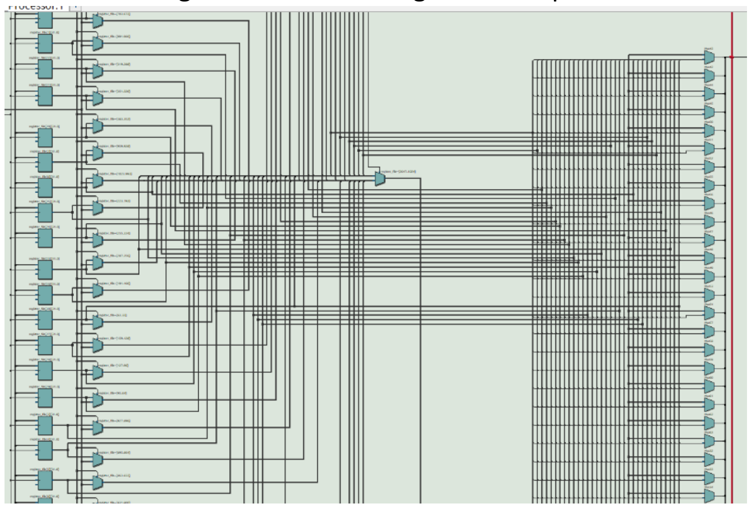
Responsible for storing and providing program instructions to the CPU. It fetches instructions based on the address provided by the PC Register and sends them to other parts of the CPU for decoding and execution.This models the RISC-V processor's instruction memory component. It accepts an address as an input, fetches the corresponding 32-bit instruction from a memory array inside the processor, and outputs the information. In order to simulate instruction fetching behavior, which is crucial for a processor's execution of instructions. The module includes initialization for various RISC-V instructions at specific memory addresses. This is where the actual program instructions would be implemented.

Register memory:



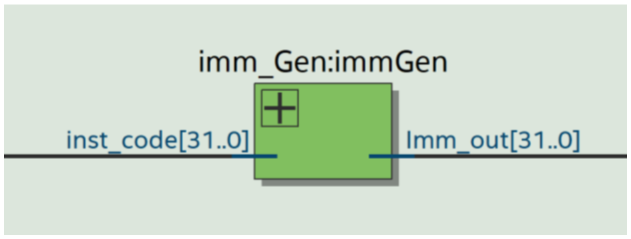
Register File is a collection of registers used for data storage and manipulation within the CPU. It holds operands and results for arithmetic and logical operations and is accessible by the CPU's instruction set.The module updates the register file when the clock signal rises by checking for reset or a request to write to a register (regWrite signal). Synchronous reset signal, when asserted, resets all registers to zero. The values of the specified registers are then assigned as outputs.





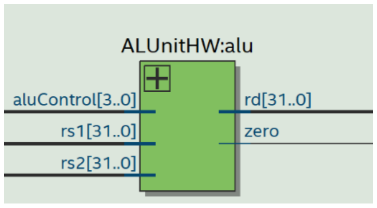
Immediate Generator:

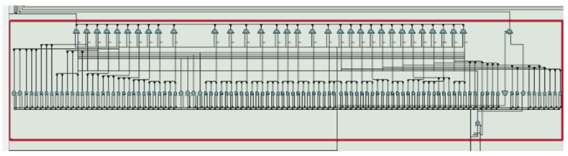
Sign extension is applied to the immediate value to handle various offset scenarios.



ALU:

Execution of arithmetic and logical operations in the CPU takes place here. It performs operations like addition, subtraction, AND, OR, etc., based on control signals from the Control Unit.

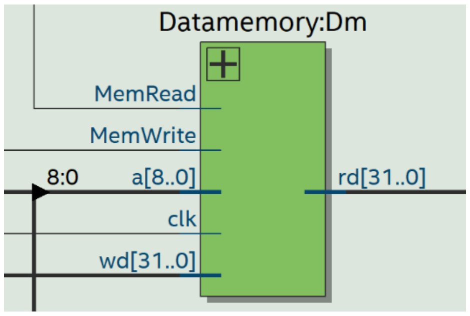


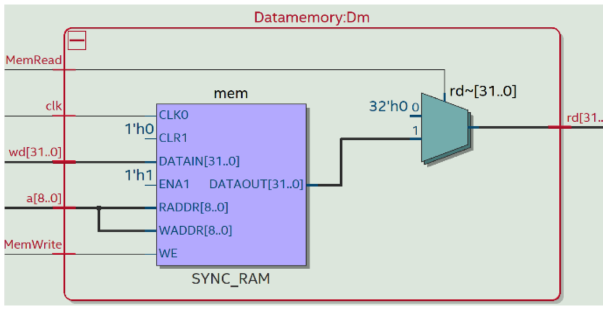


It takes two input values, rs1 and rs2, along with a 4-bit aluControl signal (provided by the microprogramming control unit) to determine the operation to perform. Various arithmetic and logical operations, including addition, subtraction, left and right shifts, comparisons, bitwise XOR, OR, and AND, are carried out by the module. Both an output result rd and a "zero" signal indicating whether the result is zero are provided.

Data Memory:

Used for temporary data storage in a computer system. It stores data that can be read from or written to by the CPU, facilitating operations like load and store.

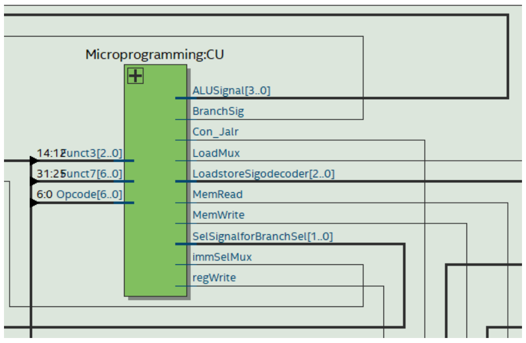




The memory read signal (MemRead), memory write signal (MemWrite), read/write address (a), and write data (wd) are all inputs that the module accepts. The module outputs the data at the specified address a when MemRead is active, and loaded to the write destination register in register files. Data memory writes the data (reg 2 value wd) to the specified address when MemWrite is active.

Control Unit:

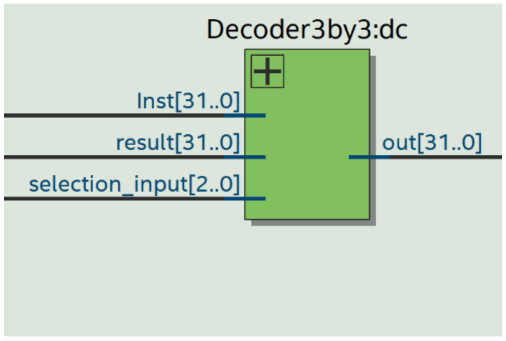
Responsible for sequencing instructions, generating control signals, and coordinating data flow between CPU components. It interprets instruction codes, generates microoperations, and controls the execution of program instructions.

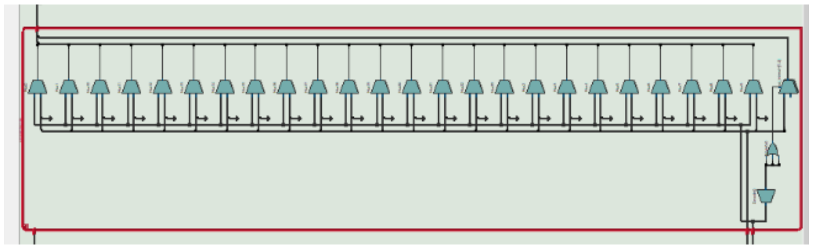


This serves as the processor's microcode-based control unit. To determine the processor's control signals, it decodes the inputs from the given opcode, funct3, and funct7. A ROM (Read-Only Memory) built into the module stores microinstructions corresponding to various instructions and operation types. In order to choose the appropriate microinstruction, which configures various control signals depending on the executed instruction, it performs extensive decoding of the input values. These control signals include register write (regWrite), immediate selector (immSelMux), memory load selector (LoadMux), memory read (MemRead), memory write (MemWrite), conditional branch (BranchSig), ALU operation selector (ALUSignal), and others. This microprogramming module plays a key role in coordinating the control signals required for a processor to carry out a variety of instructions.

Load Store Decode:

Responsible for decoding load and store instructions in computer architecture.

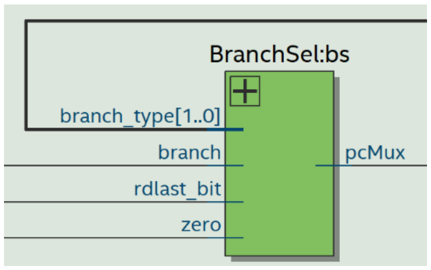


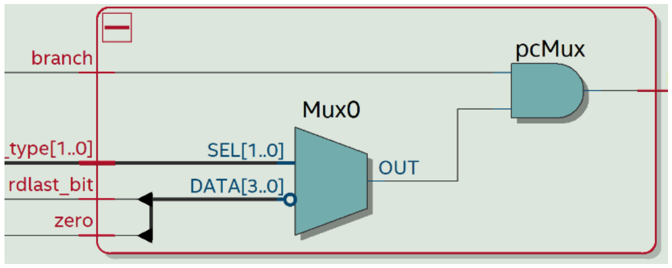


This module is designed to decode the type of load or store instructions that need to be activated. If it needs to be load word, load byte, load half byte or unsigned versions or store word, store byte, store half byte are determined in this block. The relevant instructions are chosen according to a selection input signal by the microprogrammed control unit. The output is fed to the C bus through the LoadSel Mux.

Branch Select:

Used to determine if the branch should be taken based on the specific branch type and the provided condition flags. It uses the branch type input to differentiate between various branch instructions , (e.g., equal, not equal, less than) and according to the branch signal which are signals given by the microprogrammed control unit, computes the pcMux output, indicating if the branch should be taken by evaluating the specified condition based on the branch type.



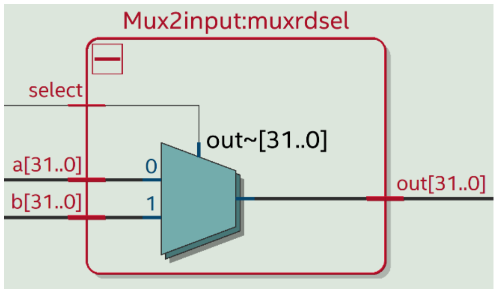


Components:

2: 1 multiplexer:

Selects one of its two input signals based on a control signal. In this I have used 6 multiplexers namely;

- MuxImmSel:
  To select between register2Value output from ALU and Immediate Output from the Immediate Generator. When the ImmSelMux signal is high it puts Immediate Output to the B bus.
- LoadSelMux:
  To select between load store decoder output and alu output value from ALU which is to be sent to C bus. When the LoadSelMux signal is high, it puts LoadStore decoder output into the C bus.
- MuxBranch:
  To select between pc output + 4 value or pc output + Immediate output value. When the BranchSig is high, pc output+immediate output is fed to the mux output.
- JumpSel Mux:
  To select between output from branchsel Mux and pc+alu output value. When the Con_Jalr signal from control unit is high, pc+ alu output is fed to the input of the pc register.
- MuxrdSel:
  To select the data value that is going to be saved to the write register destination of the RegFiles. When the Con_Jalr signal is high we save the pc+4 value to the write destination register in the register files.
- Muxmemcopyl:
  To select immout or rs2 value as the no.of copying elements



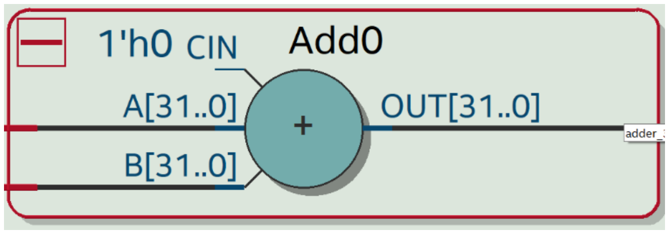
Hardware architecture of a multiplexer

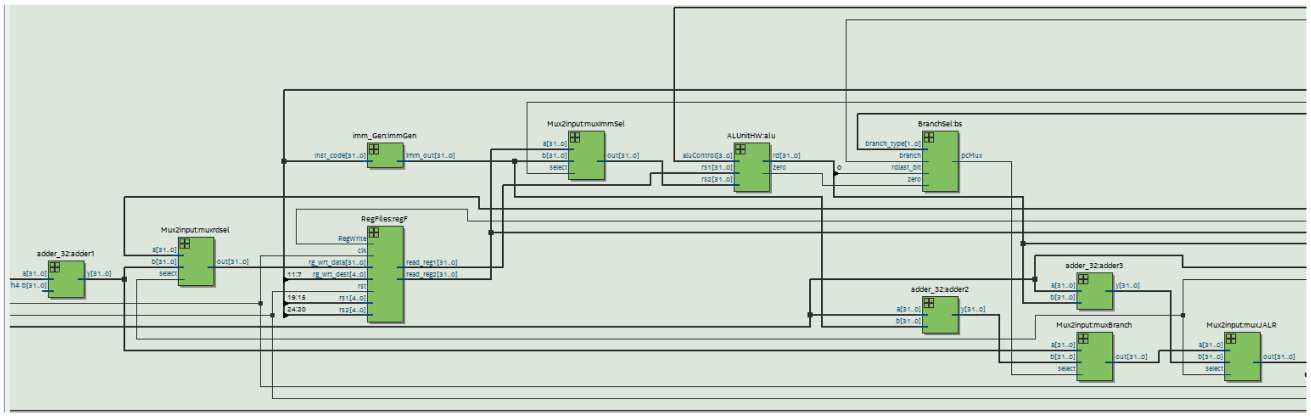
Adders:

Units designed to perform addition operations on binary numbers.

I have used 3 adders in my design, namely adder1, adder2 and adder3, adding pc+4, pc+ immediate output and pc+ alu Output respectively.

Hardware architecture of an adder





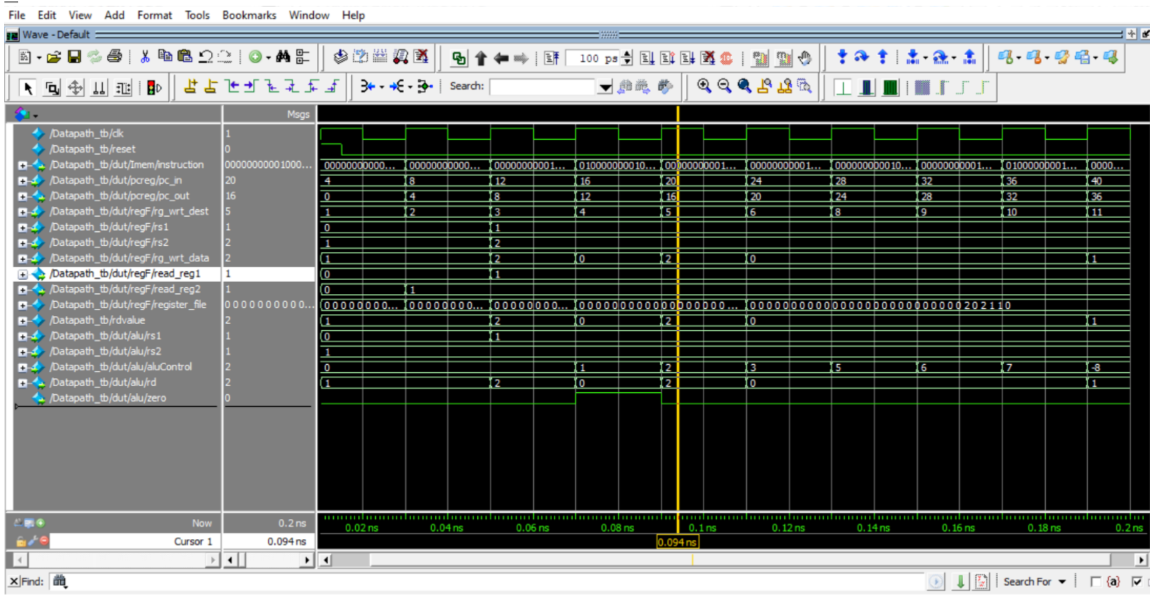
Results of Simulations

I have got some sample instructions and simulated the resulting waveforms using Altera Modelsim simulation tool.

I type and R type

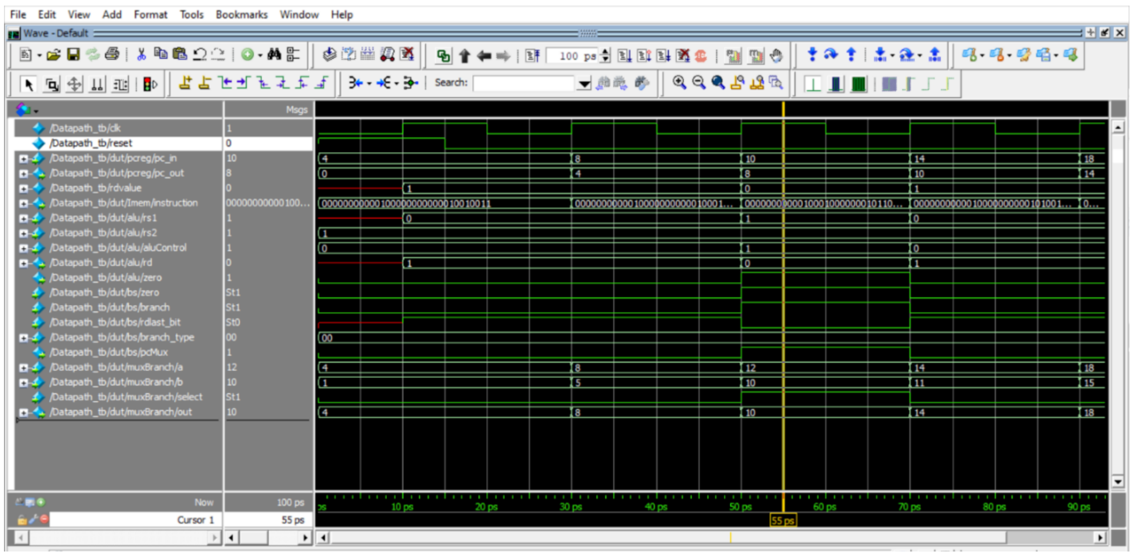
```
// initialize memory with RISC-V instructions:
assign Inst_memory[0]=32'b0000000_00001_00000_000_00001_0010011; //addi r1 r0 1
assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011; //addi r2 r0 1

  //R TYPE instruction

assign Inst_memory[8]=32'b0000000_00010_00001_000_00011_0110011; //add r3 r1 r2
assign Inst_memory[12]=32'b0100000_00010_00001_000_00100_0110011; //sub r4 r1 r2
assign Inst_memory[16]=32'b0000000_00010_00001_001_00101_0110011; //sll r5 r1 r2
assign Inst_memory[20]=32'b0000000_00010_00001_010_00110_0110011; //slt r6 r1 r2
assign Inst_memory[22]=32'b0000000_00010_00001_011_00111_0110011; //sltu r7 r1 r2
assign Inst_memory[24]=32'b0000000_00010_00001_100_01000_0110011; //xor  r8 r1 r2
assign Inst_memory[28]=32'b0000000_00010_00001_101_01001_0110011; //srl  r9 r1 r2
assign Inst_memory[32]=32'b0100000_00010_00001_101_01010_0110011; //sra r10 r1 r2
assign Inst_memory[36]=32'b0000000_00010_00001_110_01011_0110011; //or r11 r1 r2
assign Inst_memory[40]=32'b0000000_00010_00001_111_01100_0110011; //and r12 r1 r2
```

First 2 I type instructions are carried out. The two registers r1 and r2 get the immediate values 1. Next every R type instructions gets executed except for instruction at inst_memory[22]. You can check the output from either alu/rd or rdvalue in register files. Thus the proper implementation of both types of instructions are guranteed.


File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
/Datapath_tb/clk
/Datapath_tb/reset
/Datapath_tb/dut/Imem/instruction
/Datapath_tb/dut/pcreg/pc_in
/Datapath_tb/dut/pcreg/pc_out
/Datapath_tb/dut/regF/rg_wrt_dest
/Datapath_tb/dut/regF/rs1
/Datapath_tb/dut/regF/rs2
/Datapath_tb/dut/regF/rg_wrt_data
/Datapath_tb/dut/regF/read_reg1
/Datapath_tb/dut/regF/read_reg2
/Datapath_tb/dut/regF/register_file
/Datapath_tb/rdvalue
/Datapath_tb/dut/alu/rs1
/Datapath_tb/dut/alu/rs2
/Datapath_tb/dut/alu/aluControl
/Datapath_tb/dut/alu/rd
/Datapath_tb/dut/alu/zero
Now 0.2 ns
Cursor 1 0.094 ns

Branch Instructions

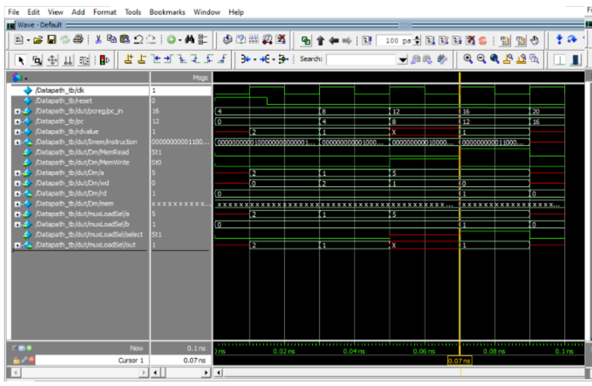
```
assign Inst_memory[0]=32'b0000000_00001_00000_000_00001_0010011; //addi r1 r0 1
assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011; //addi r2 r0 1

//branch test
assign Inst_memory[8]=32'b0000000_00001_00010_000_00010_1100011;//beq rs1 rs2 (2)
assign Inst_memory[10]=32'b0000000_00001_00000_000_00101_0010011;//addi r5 r0 1
assign Inst_memory[14]=32'b0000000_00001_00000_000_00111_0010011;
```

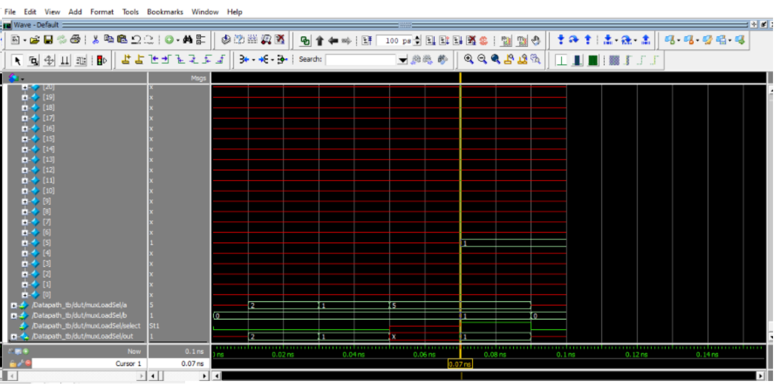

File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
/Datapath_tb/clk
/Datapath_tb/reset
/Datapath_tb/dut/pcreg/pc_in
/Datapath_tb/dut/pcreg/pc_out
/Datapath_tb/rdvalue
/Datapath_tb/dut/Imem/instruction
/Datapath_tb/dut/alu/rs1
/Datapath_tb/dut/alu/rs2
/Datapath_tb/dut/alu/aluControl
/Datapath_tb/dut/alu/rd
/Datapath_tb/dut/alu/zero
/Datapath_tb/dut/bs/zero
/Datapath_tb/dut/bs/branch
/Datapath_tb/dut/bs/rdlast_bit
/Datapath_tb/dut/bs/branch_type
/Datapath_tb/dut/bs/pcMux
/Datapath_tb/dut/muxBranch/a
/Datapath_tb/dut/muxBranch/b
/Datapath_tb/dut/muxBranch/select
/Datapath_tb/dut/muxBranch/out
Now 100 ps
Cursor 1 55 ps

First by addi instructions I add 1 and 1 data values to the registers 1 and 2 respectively. Third one is a branch if equal instruction. It compares values in register 1 and register 2 , since both are equal and the branch control signal given, the bs module outputs a high signal as pcMux to select values from pc+4 or pc+immediate. Since the signal is high it selects pc+immediate value (which is 8+2=10) here. And the inst_memory[10] gets executed.

Store Instructions and Load Instructions

```
//store test
assign Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;//addi r1 r0 2
assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;//addi r2 r0 1
assign Inst_memory[8]=32'b0000000_00010_00001_010_00011_0100011; //sw  r2 3(x1) store 1 in me

assign Inst_memory[12]=32'b0000000_00011_00001_010_00011_0000011; //lw  r3 3(x1)
```

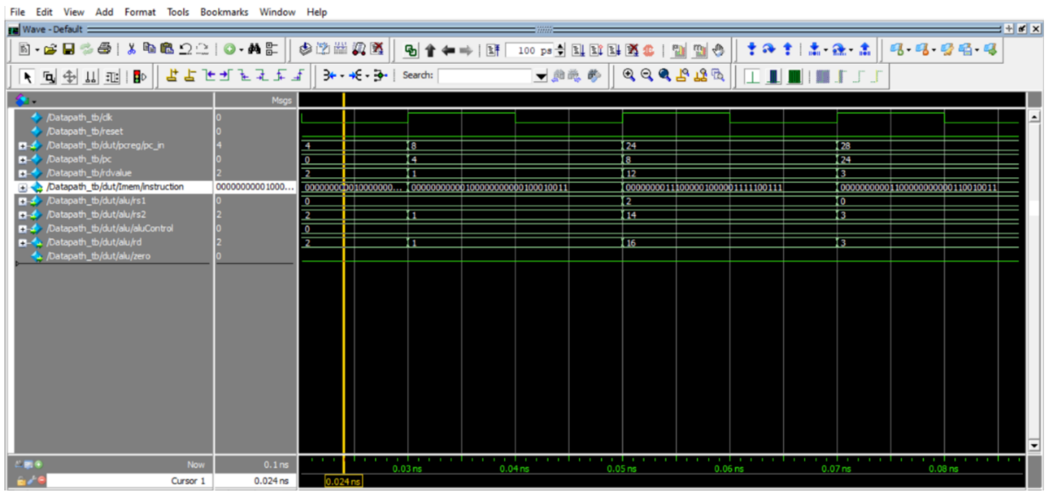

/Datapath_tb/clk
/Datapath_tb/reset
/Datapath_tb/dut/pcreg/pc_in
/Datapath_tb/pc
/Datapath_tb/rdvalue
/Datapath_tb/dut/Imem/instruction
/Datapath_tb/dut/Dm/MemRead
/Datapath_tb/dut/Dm/MemWrite
/Datapath_tb/dut/Dm/a
/Datapath_tb/dut/Dm/wd
/Datapath_tb/dut/Dm/rd
/Datapath_tb/dut/Dm/mem
/Datapath_tb/dut/muxLoadSel/a
/Datapath_tb/dut/muxLoadSel/b
/Datapath_tb/dut/muxLoadSel/select
/Datapath_tb/dut/muxLoadSel/out


/Datapath_tb/dut/muxLoadSel/a
/Datapath_tb/dut/muxLoadSel/b
/Datapath_tb/dut/muxLoadSel/select
/Datapath_tb/dut/muxLoadSel/out

First by addi instructions I add 2 and 1 data values to the registers 1 and 2 respectively. When the third store instruction execution happens, the value in register 2(1) is stored to the memory location of 5, as in the figure to right. [register 1 value(2) + immediate value(3) =5] then as the 4th instruction load instruction to access the data in that memory location(5) is carried out. We load the value in memory location 5 (1) [register 1 value(2) + immediate value(3) =5] to register 3 in register files. It is visible at rdvalue 4th instruction in figure to left.

JALR instruction (I type)

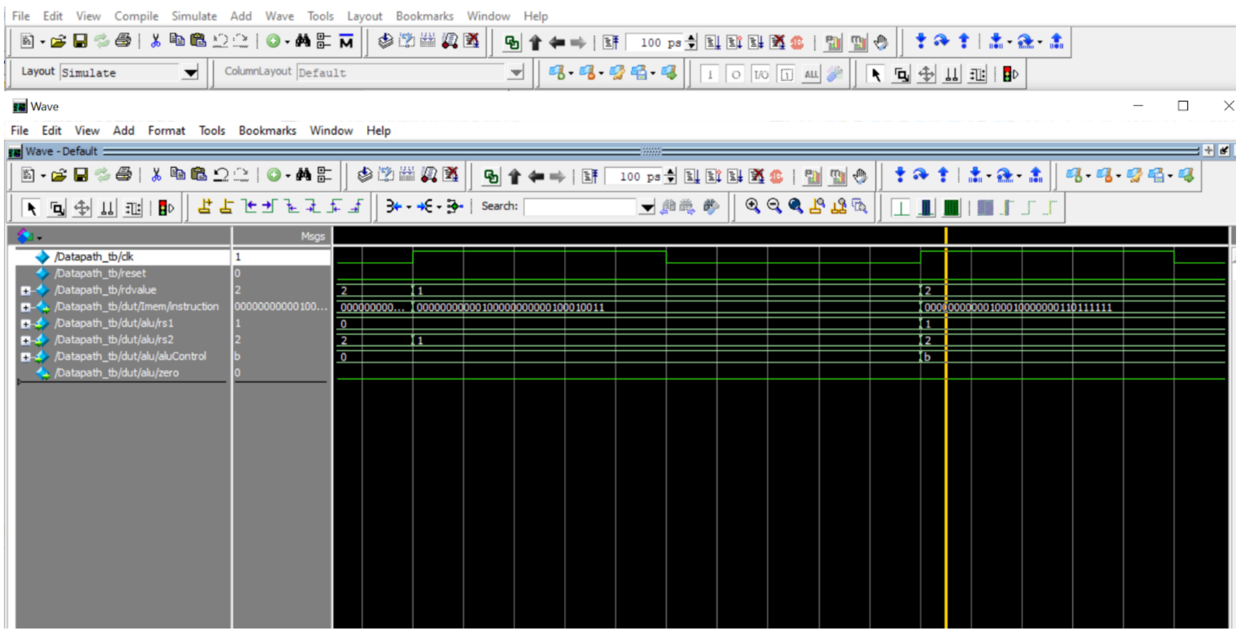
```
//jalr test
assign Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;//addi r1 r0 2
assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;//addi r2 r0 1

assign Inst_memory[8]=32'b0000000_01110_00001_000_00111_1100111;// jalr
assign Inst_memory[24]=32'b0000000_00011_00000_000_00011_0010011;//addi r3 r0 3
```



First by addi instructions I add 2 and 1 data values to the registers 1 and 2 respectively. In the 3rd instruction, value in register 1(2) gets added to the sign extended immediate value(14), and the result(16) is added to the PC (8+16)as the next instruction(24) to be implemented. The Inst_memory[24] instruction gets implemented as well. At the same time register file write register, which is rdvalue here stores PC+4 value(which is 8+4=12)in 3rd execution. Therefore, JALR instruction is properly executed.

Unsigned Multiplication

```
assign Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;//addi r1 r0 2
assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;//addi r2 r0 1
assign Inst_memory[8]=32'b0000000_00001_00010_000_00011_0111111;
```



```
/Datapath_tb/clk                    1
/Datapath_tb/reset                  0
/Datapath_tb/rdvalue                2                 2 | 1 | 2
/Datapath_tb/dut/Imem/instruction   00000000000100...  000000000... | 00000000000100000000000100010011 | 00000000000100010000000110111111
/Datapath_tb/dut/alu/rs1            1                 0 | 1
/Datapath_tb/dut/alu/rs2            2                 2 | 1 | 2
/Datapath_tb/dut/alu/aluControl     b                 0 | b
/Datapath_tb/dut/alu/zero           0
```

First by addi instructions I add 2 and 1 data values to the registers 1 and 2 respectively. In next instruction unsigned multiplication takes place. The values in r1 and r2 registers get multiplied and stored (2*1=2) to register 3.

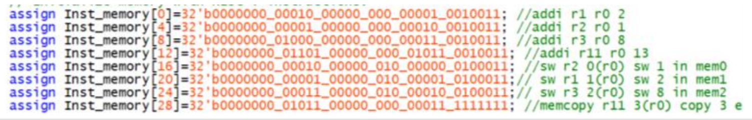
limits for operands of this instruction: When two numbers are multiplied, the total number of bits in the output number equals the sum of the bits in the two multiplied numbers. As a result, the total number of bits in the output should not exceed 32 since we are using 32-bit registers. Therefore, the restriction is that the sum of the bits for the two operands cannot be greater than 32.

Memcopy Instruction

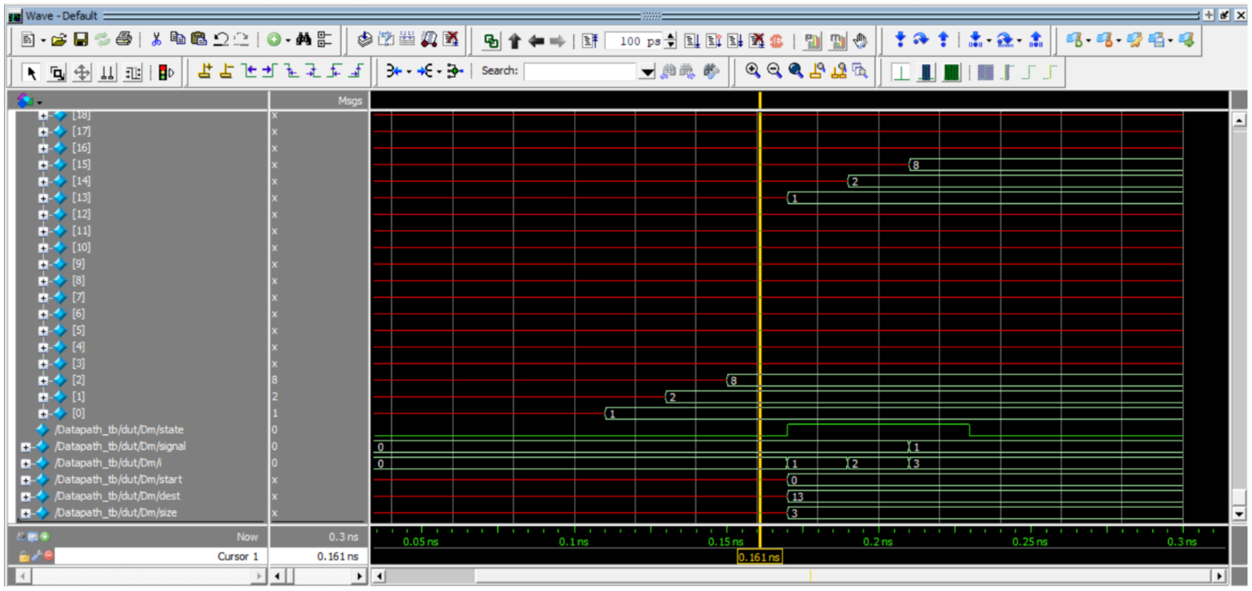
This instruction is to copy an array of values in memory files to another location. The plan was to implement a store type instruction; where immediate value specifies no of elements to be copied, value in rs1 and value in rs2 cites from where to start copying from (memory address of copy start value) and to where we should copy them to respectively. The opcode I've chosen for this case is 1111111 and func3 is 000.

Maximum Size of Copy Array per my plan can be [(2^12)-1]

Here first I get the registers r1,r21r3,r4 with values 2,1,8,13 respectively using addi. Then I store the first 3 values in 3 registers to mem0,mem1 and mem2. Next I perform the memcopy function, where 3 values(given immediate/ no.of values to copy) from mem[r0 value(0)] gets copied to mem[r11 value(13)] and 2 more consecutive memory location from that memory location.



```
assign Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011; //addi r1 r0 2
assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011; //addi r2 r0 1
assign Inst_memory[8]=32'b0000000_01000_00000_000_00011_0010011; //addi r3 r0 8
assign Inst_memory[12]=32'b0000000_01101_00000_000_01011_0010011; //addi r11 r0 13
assign Inst_memory[16]=32'b0000000_00010_00000_010_00000_0100011; //sw r2 0(r0) sw 1 in mem0
assign Inst_memory[20]=32'b0000000_00001_00000_010_00001_0100011;// sw r1 1(r0) sw 2 in mem1
assign Inst_memory[24]=32'b0000000_00011_00000_010_00010_0100011;// sw r3 2(r0) sw 8 in mem2
assign Inst_memory[28]=32'b0000000_01011_00000_000_00011_1111111; //memcopy r11 3(r0) copy 3 e
```

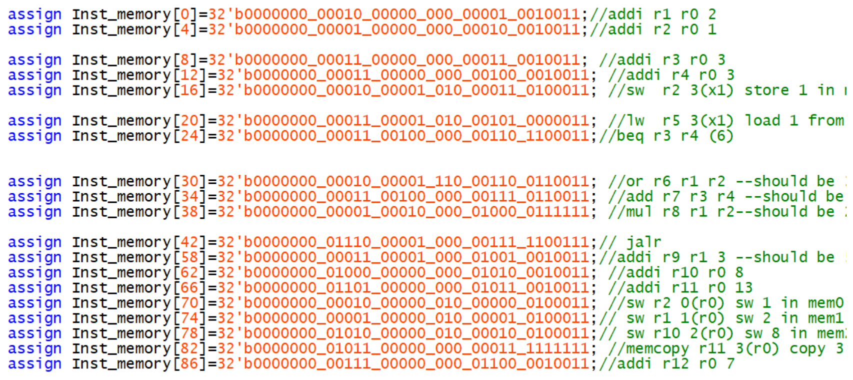


Hardware Verification

FPGA implementation

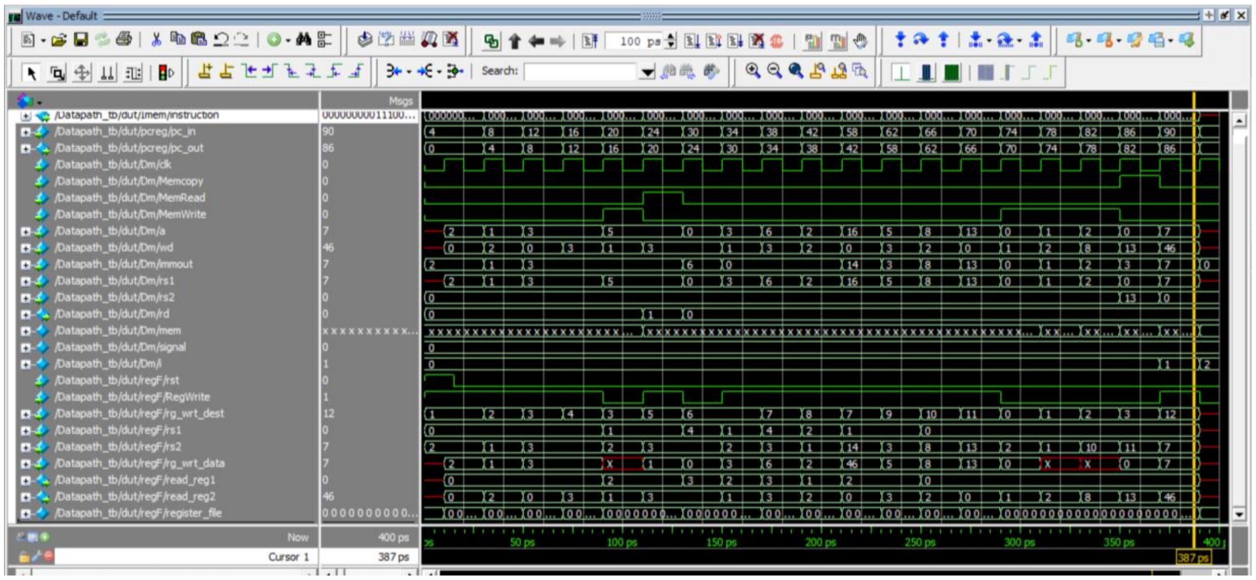
Here I used TerasIC DE2 115 FPGA development and education board for hardware verifications.

I implemented a small program which covers every necessary instruction and when switch inputs are given I'll display the write register value and memory register values on the seven segment displays.



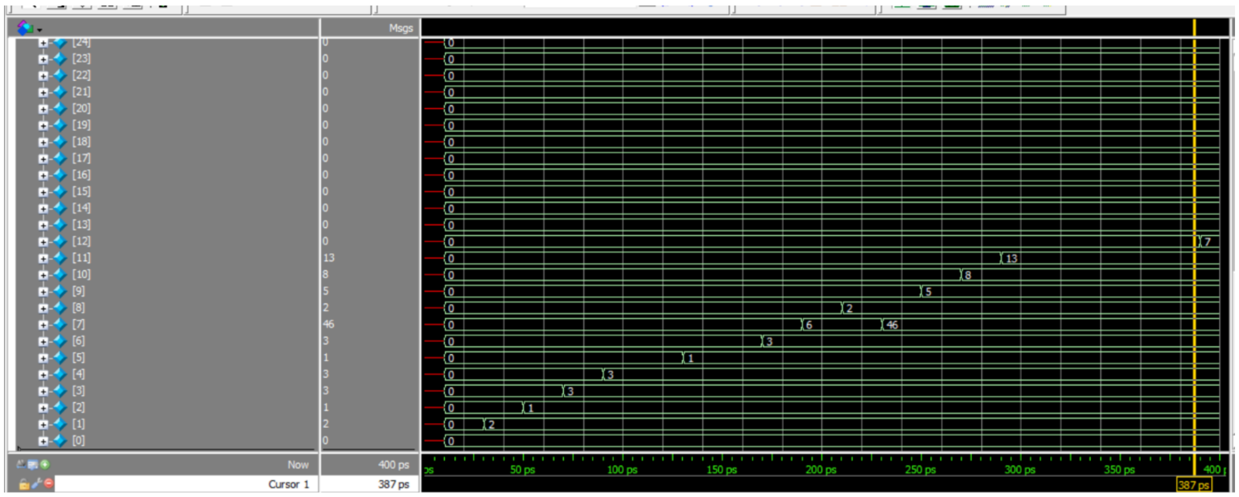
```
assign Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;//addi r1 r0 2
assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;//addi r2 r0 1

assign Inst_memory[8]=32'b0000000_00011_00000_000_00011_0010011; //addi r3 r0 3
assign Inst_memory[12]=32'b0000000_00011_00000_000_00100_0010011; //addi r4 r0 3
assign Inst_memory[16]=32'b0000000_00010_00001_010_00011_0100011; //sw  r2 3(x1) store 1 in

assign Inst_memory[20]=32'b0000000_00011_00001_010_00101_0000011; //lw  r5 3(x1) load 1 from
assign Inst_memory[24]=32'b0000000_00011_00100_000_00110_1100011;//beq r3 r4 (6)


assign Inst_memory[30]=32'b0000000_00010_00001_110_00110_0110011; //or r6 r1 r2 --should be
assign Inst_memory[34]=32'b0000000_00011_00100_000_00111_0110011; //add r7 r3 r4 --should be
assign Inst_memory[38]=32'b0000000_00001_00010_000_01000_0111111; //mul r8 r1 r2--should be

assign Inst_memory[42]=32'b0000000_01110_00001_000_00111_1100111;// jalr
assign Inst_memory[58]=32'b0000000_00011_00001_000_01001_0010011;//addi r9 r1 3 --should be
assign Inst_memory[62]=32'b0000000_01000_00000_000_01010_0010011; //addi r10 r0 8
assign Inst_memory[66]=32'b0000000_01101_00000_000_01011_0010011; //addi r11 r0 13
assign Inst_memory[70]=32'b0000000_00010_00000_010_00000_0100011; //sw r2 0(r0) sw 1 in mem0
assign Inst_memory[74]=32'b0000000_00001_00000_010_00001_0100011;// sw r1 1(r0) sw 2 in mem1
assign Inst_memory[78]=32'b0000000_01010_00000_010_00010_0100011;// sw r10 2(r0) sw 8 in mem
assign Inst_memory[82]=32'b0000000_01011_00000_000_00011_1111111; //memcopy r11 3(r0) copy 3
assign Inst_memory[86]=32'b0000000_00111_00000_000_01100_0010011;//addi r12 r0 7
```

Here's the simulation of the small program I implemented.

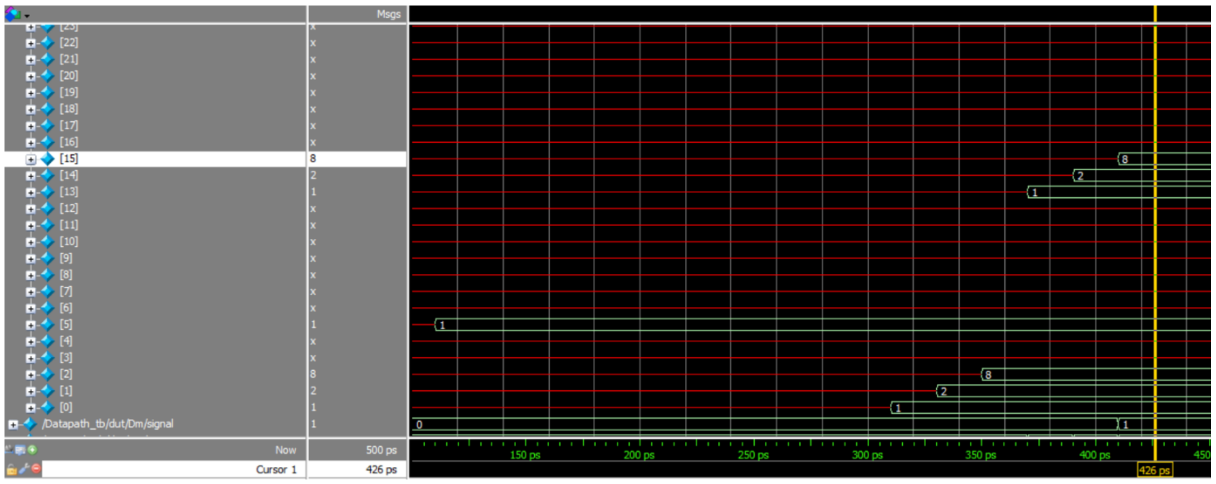


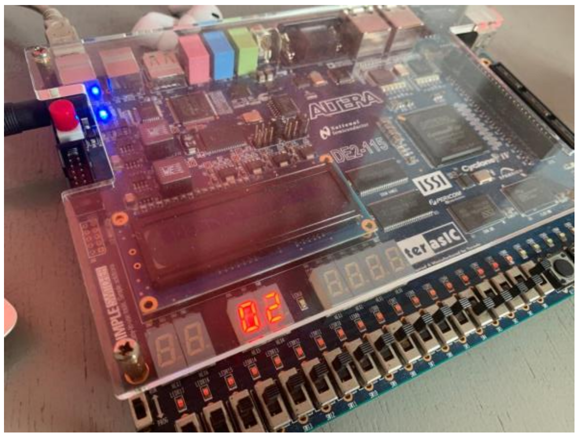
/Datapath_tb/dut/imem/instruction
/Datapath_tb/dut/pcreg/pc_in
/Datapath_tb/dut/pcreg/pc_out
/Datapath_tb/dut/Dm/clk
/Datapath_tb/dut/Dm/Memcopy
/Datapath_tb/dut/Dm/MemRead
/Datapath_tb/dut/Dm/MemWrite
/Datapath_tb/dut/Dm/a
/Datapath_tb/dut/Dm/wd
/Datapath_tb/dut/Dm/immout
/Datapath_tb/dut/Dm/rs1
/Datapath_tb/dut/Dm/rs2
/Datapath_tb/dut/Dm/rd
/Datapath_tb/dut/Dm/mem
/Datapath_tb/dut/Dm/signal
/Datapath_tb/dut/Dm/i
/Datapath_tb/dut/regF/rst
/Datapath_tb/dut/regF/RegWrite
/Datapath_tb/dut/regF/rg_wrt_dest
/Datapath_tb/dut/regF/rs1
/Datapath_tb/dut/regF/rs2
/Datapath_tb/dut/regF/rg_wrt_data
/Datapath_tb/dut/regF/read_reg1
/Datapath_tb/dut/regF/read_reg2
/Datapath_tb/dut/regF/register_file
Now 400 ps
Cursor 1 387 ps

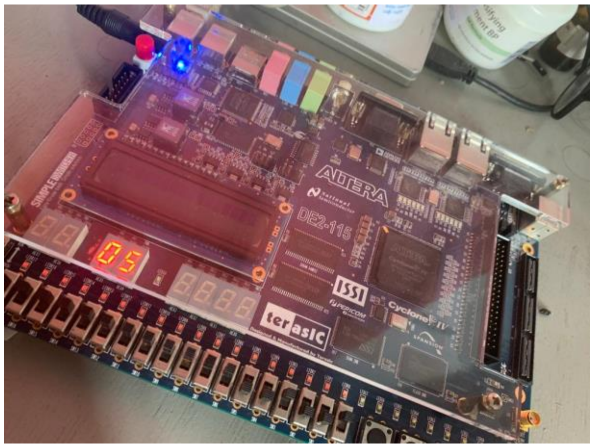
regFiles



memory registers







I made a separate hexdigit module, for seven segment display operations and called it on the top module. When selected the relevant switches that depicts write destination register, it reads the value in that register and displays the value on the seven segment.

Here I have selected first five switches as inputs(5 bit input), and as per my 1^{st} instruction, when I give 00001 as switch inputs, the value 2 gets displayed on seven segments.(Fig 1)

When I select 01001 the destination register address of the 58th instruction, the value 5 gets displayed on the seven segment which proves the accurate implementation of all the middle instructions.

Verified similarly for all the instructions, thus that assured the accurate implementation of the processor on FPGA for all types of given instructions.

This is the implementation without last 7 instructions.

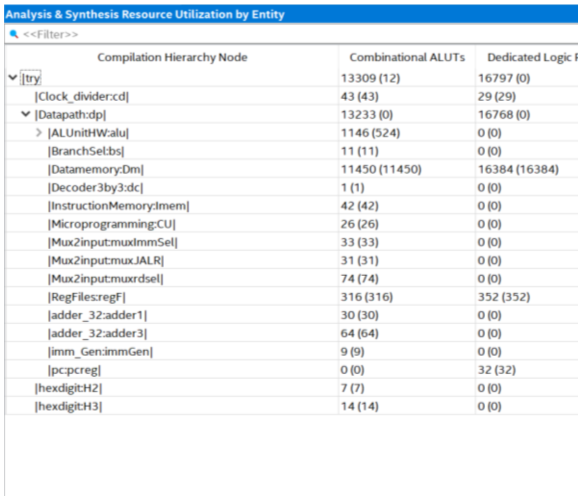
Resource Utilization Report

Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Memory Bits	DSP Elements	DSP 9x9	DSP 18x18	Pins	Virtual Pins	Full Hierarchy Name	Entity Name
|try	13302 (13)	16797 (0)	0	0	0	0	21	0	|try	try
|Clock_divider:cd|	43 (43)	29 (29)	0	0	0	0	0	0	|try|Clock_divider:cd	Clock_divider
|Datapath:dp|	13232 (0)	16768 (0)	0	0	0	0	0	0	|try|Datapath:dp	Datapath
|ALUnitHW:alu|	1144 (523)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	ALUnitHW
|lpm_mult:Mult0|	621 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_mult
|multcore:mult_core|	621 (289)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	multcore
|mpar_add:padder|	272 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	mpar_add
|lpm_add_sub:adder[0]|	32 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	32 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	32 (32)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[1]|	28 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	28 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	28 (28)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[2]|	24 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	24 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	24 (24)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[3]|	20 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	20 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	20 (20)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[4]|	16 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	16 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	16 (16)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[5]|	12 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	12 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	12 (12)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[6]|	8 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	8 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	8 (8)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[7]|	4 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	4 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	4 (4)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|mpar_add:sub_par_add|	128 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	mpar_add
|lpm_add_sub:adder[0]|	30 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	30 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	30 (30)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[1]|	22 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	22 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	22 (22)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[2]|	14 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	14 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	14 (14)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[3]|	6 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	6 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	6 (6)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|mpar_add:sub_par_add|	56 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	mpar_add
|lpm_add_sub:adder[0]|	26 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	26 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	26 (26)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|lpm_add_sub:adder[1]|	10 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	10 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	10 (10)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|mpar_add:sub_par_add|	20 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	mpar_add
|lpm_add_sub:adder[0]|	18 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	18 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|addcore:first_seg_adder|	18 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	18 (18)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|mpar_add:sub_par_add|	2 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	mpar_add
|lpm_add_sub:adder[0]|	2 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	lpm_add_sub
|addcore:adder|	2 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|addcore:first_seg_adder|	2 (0)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	addcore
|a_csnbuffer:result_node|	2 (2)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	a_csnbuffer
|mul_boothc:booth_enc|	60 (60)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|ALUnitHW:al	mul_boothc
|BranchSel:bs|	11 (11)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|BranchSel:bs	BranchSel
|Datamemory:Dm|	11450 (11450)	16384 (16384)	0	0	0	0	0	0	|try|Datapath:dp|Datamemory	Datamemory
|Decoder3by3:dc|	1 (1)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|Decoder3by3	Decoder3by3
|InstructionMemory:Imem|	42 (42)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|InstructionM	InstructionMemory
|Microprogramming:CU|	26 (26)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|Micrograi	Microprogramming
|Mux2input:muxImmSel|	33 (33)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|Mux2input:n	Mux2input
|Mux2input:muxJALR|	31 (31)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|Mux2input:n	Mux2input
|Mux2input:muxrdsel|	74 (74)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|Mux2input:n	Mux2input
|RegFiles:regF|	316 (316)	352 (352)	0	0	0	0	0	0	|try|Datapath:dp|RegFiles:regF	RegFiles
|adder_32:adder1|	30 (30)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|adder_32:ad	adder_32
|adder_32:adder3|	64 (64)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|adder_32:ad	adder_32
|imm_Gen:immGen|	10 (10)	0 (0)	0	0	0	0	0	0	|try|Datapath:dp|imm_Gen:im	imm_Gen
|pc:pcreg|	0 (0)	32 (32)	0	0	0	0	0	0	|try|Datapath:dp|pc:pcreg	pc
|hexdigit:H2|	7 (7)	0 (0)	0	0	0	0	0	0	|try|hexdigit:H2	hexdigit
|hexdigit:H3|	7 (7)	0 (0)	0	0	0	0	0	0	|try|hexdigit:H3	hexdigit

Resource Usage Report

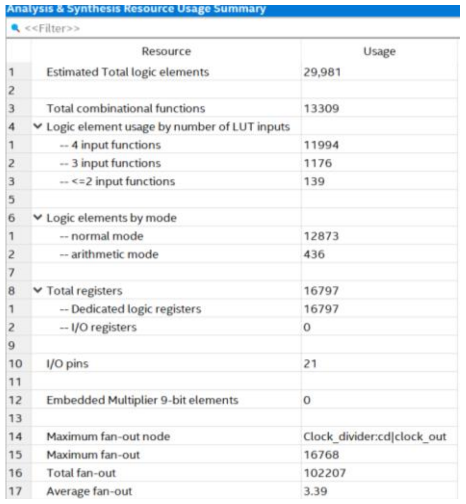
Resource	Usage
Estimated Total logic elements	29,974
Total combinational functions	13302
Logic element usage by number of LUT inputs	
-- 4 input functions	11982
-- 3 input functions	1178
-- <=2 input functions	142
Logic elements by mode	
-- normal mode	12866

-- arithmetic mode	436
Total registers	16797
-- Dedicated logic registers	16797
-- I/O registers	0
I/O pins	21
Embedded Multiplier 9-bit elements	0
Maximum fan-out node	Clock_divider:cd|clock_out
Maximum fan-out	16768
Total fan-out	102171
Average fan-out	3.39


Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic R
|try	13309 (12)	16797 (0)
|Clock_divider:cd|	43 (43)	29 (29)
|Datapath:dp|	13233 (0)	16768 (0)
|ALUnitHW:alu|	1146 (524)	0 (0)
|BranchSel:bs|	11 (11)	0 (0)
|Datamemory:Dm|	11450 (11450)	16384 (16384)
|Decoder3by3:dc|	1 (1)	0 (0)
|InstructionMemory:Imem|	42 (42)	0 (0)
|Microprogramming:CU|	26 (26)	0 (0)
|Mux2input:muxImmSel|	33 (33)	0 (0)
|Mux2input:muxJALR|	31 (31)	0 (0)
|Mux2input:muxrdsel|	74 (74)	0 (0)
|RegFiles:regF|	316 (316)	352 (352)
|adder_32:adder1|	30 (30)	0 (0)
|adder_32:adder3|	64 (64)	0 (0)
|imm_Gen:immGen|	9 (9)	0 (0)
|pc:pcreg|	0 (0)	32 (32)
|hexdigit:H2|	7 (7)	0 (0)
|hexdigit:H3|	14 (14)	0 (0)


Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated Total logic elements	29,981
2		
3	Total combinational functions	13309
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	11994
2	-- 3 input functions	1176
3	-- <=2 input functions	139
5		
6	Logic elements by mode	
1	-- normal mode	12873
2	-- arithmetic mode	436
7		
8	Total registers	16797
1	-- Dedicated logic registers	16797
2	-- I/O registers	0
9		
10	I/O pins	21
11		
12	Embedded Multiplier 9-bit elements	0
13		
14	Maximum fan-out node	Clock_divider:cd|clock_out
15	Maximum fan-out	16768
16	Total fan-out	102207
17	Average fan-out	3.39

Appendices : Synthesizable RTL Codes

Pc

```
module pc (
    input wire clk, reset,
    input wire [31:0] pc_in,
    output reg [31:0] pc_out
);

always_ff @(posedge clk, posedge reset) begin
    if (reset)
```

```
        pc_out <= 32'b0;
    else
        pc_out <= pc_in;
end

endmodule
```

Instruction Memory

```
module InstructionMemory #(parameter INS_ADDRESS=32,INS_W=32)(
    input logic [INS_ADDRESS - 1:0] address, // Read address of the
instruction memory, comes from PC
    output logic [INS_W - 1:0] instruction
);

    logic [INS_W - 1:0] Inst_memory [90:0];
    // Define the instruction memory as a simple array of 32-bit
instructions.

    // Initialize memory with RISC-V instructions.
       //memcopy test
//    assign Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;
//addi r1 r0 2
//    assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;
//addi r2 r0 1
//     assign Inst_memory[8]=32'b0000000_01000_00000_000_00011_0010011;
//addi r3 r0 8
//     assign Inst_memory[12]=32'b0000000_01101_00000_000_01011_0010011;
//addi r11 r0 13
//    assign Inst_memory[16]=32'b0000000_00010_00000_010_00000_0100011;
//sw r2 0(r0) sw 1 in mem0
//     assign Inst_memory[20]=32'b0000000_00001_00000_010_00001_0100011;//
sw r1 1(r0) sw 2 in mem1
//     assign Inst_memory[24]=32'b0000000_00011_00000_010_00010_0100011;//
sw r3 2(r0) sw 8 in mem2
//     assign Inst_memory[28]=32'b0000000_01011_00000_000_00011_1111111;
//memcopy r11 3(r0) copy 3 elements including mem[r0] to mem[r11](mem13)
and after
//


//    //R TYPE instruction
//
//    assign Inst_memory[8]=32'b0000000_00010_00001_000_00011_0110011;
//add r3 r1 r2
//    assign Inst_memory[12]=32'b0100000_00010_00001_000_00100_0110011;
//sub r4 r1 r2
//    assign Inst_memory[16]=32'b0000000_00010_00001_001_00101_0110011;
//sll r5 r1 r2
//    assign Inst_memory[20]=32'b0000000_00010_00001_010_00110_0110011;
```

```
//slt r6 r1 r2
//    assign Inst_memory[22]=32'b0000000_00010_00001_011_00111_0110011;
//sltu r7 r1 r2
//    assign Inst_memory[24]=32'b0000000_00010_00001_100_01000_0110011;
//xor  r8 r1 r2
//    assign Inst_memory[28]=32'b0000000_00010_00001_101_01001_0110011;
//srl  r9 r1 r2
//    assign Inst_memory[32]=32'b0100000_00010_00001_101_01010_0110011;
//sra r10 r1 r2
//    assign Inst_memory[36]=32'b0000000_00010_00001_110_01011_0110011;
//or r11 r1 r2
//    assign Inst_memory[40]=32'b0000000_00010_00001_111_01100_0110011;
//and r12 r1 r2
//


      //branch test
//    assign Inst_memory[8]=32'b0000000_00001_00010_000_00010_1100011;//beq
rs1 rs2 (2)
//   assign
Inst_memory[10]=32'b0000000_00001_00000_000_00101_0010011;//addi r5 r0 1
//   assign Inst_memory[14]=32'b0000000_00001_00000_000_00111_0010011;

       //load test
//     assign
Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;//addi r1 r0 2
//  assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;//addi
r2 r0 1
//     assign Inst_memory[8]=32'b0000000_00011_00001_010_00011_0000011;
//lw  r3 3(x1)
//
       //store test
       assign
Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;//addi r1 r0 2
    assign Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;//addi
r2 r0 1

       assign Inst_memory[8]=32'b0000000_00011_00000_000_00011_0010011;
//addi r3 r0 3
    assign Inst_memory[12]=32'b0000000_00011_00000_000_00100_0010011;
//addi r4 r0 3
       assign Inst_memory[16]=32'b0000000_00010_00001_010_00011_0100011;
//sw  r2 3(x1) store 1 in mem5

       assign Inst_memory[20]=32'b0000000_00011_00001_010_00101_0000011;
//lw  r5 3(x1) load 1 from mem5
       assign
Inst_memory[24]=32'b0000000_00011_00100_000_00110_1100011;//beq r3 r4 (6)
```

```
       assign Inst_memory[30]=32'b0000000_00010_00001_110_00110_0110011;
//or r6 r1 r2 --should be 3
       assign Inst_memory[34]=32'b0000000_00011_00100_000_00111_0110011;
//add r7 r3 r4 --should be 6
       assign Inst_memory[38]=32'b0000000_00001_00010_000_01000_0111111;
//mul r8 r1 r2--should be 2

       assign Inst_memory[42]=32'b0000000_01110_00001_000_00111_1100111;//
jalr
       assign
Inst_memory[58]=32'b0000000_00011_00001_000_01001_0010011;//addi r9 r1 3 --
should be 5
       assign Inst_memory[62]=32'b0000000_01000_00000_000_01010_0010011;
//addi r10 r0 8
       assign Inst_memory[66]=32'b0000000_01101_00000_000_01011_0010011;
//addi r11 r0 13
    assign Inst_memory[70]=32'b0000000_00010_00000_010_00000_0100011; //sw
r2 0(r0) sw 1 in mem0
       assign Inst_memory[74]=32'b0000000_00001_00000_010_00001_0100011;//
sw r1 1(r0) sw 2 in mem1
       assign Inst_memory[78]=32'b0000000_01010_00000_010_00010_0100011;//
sw r10 2(r0) sw 8 in mem2
       assign Inst_memory[82]=32'b0000000_01011_00000_000_00011_1111111;
//memcopy r11 3(r0) copy 3 elements including mem[r0] to mem[r11](mem13)
and after
       assign
Inst_memory[86]=32'b0000000_00111_00000_000_01100_0010011;//addi r12 r0 7


       //jalr test
//     assign
Inst_memory[0]=32'b0000000_00010_00000_000_00001_0010011;//addi r1 r0 2
//    assign
Inst_memory[4]=32'b0000000_00001_00000_000_00010_0010011;//addi r2 r0 1
//
//     assign Inst_memory[8]=32'b0000000_01110_00001_000_00111_1100111;//
jalr
//     assign
Inst_memory[24]=32'b0000000_00011_00000_000_00011_0010011;//addi r3 r0 3
//
    // Read operation: Output the instruction at the provided address.
    assign instruction = Inst_memory[address];

endmodule
```

RegFiles

```
module RegFiles #(
      // Parameters
      parameter DATA_WIDTH   = 32,  // number of bits in each register
      parameter ADDRESS_WIDTH = 5, //number of registers = 2^ADDRESS_WIDTH
      parameter NUM_REGS = 32
   )
   (
   input  clk, //clock
   input  rst,//synchronous reset; if it is asserted (rst=1), all registers
are reseted to 0
   input  RegWrite, //write signal
   input  [ADDRESS_WIDTH-1:0] rg_wrt_dest, //address of the register that
supposed to written into
   input  [ADDRESS_WIDTH-1:0] rs1, //first address to be read from
   input  [ADDRESS_WIDTH-1:0] rs2, //second address to be read from
   input  [DATA_WIDTH-1:0] rg_wrt_data, // data that supposed to be written
into the register file

   output logic [DATA_WIDTH-1:0] read_reg1, //content of
reg_file[rg_rd_addr1] is loaded into
   output logic [DATA_WIDTH-1:0] read_reg2 //content of
reg_file[rg_rd_addr2] is loaded into
   );

integer      i;

logic [DATA_WIDTH-1:0] register_file [NUM_REGS-1:0];

always @(posedge clk) begin
   if(rst==1'b1) //if reset iterates thru all registers and sets them to 0
      for (i = 0; i < NUM_REGS; i = i + 1)
          register_file[i] <= 0;
   else if(RegWrite==1'b1 && (rg_wrt_dest!=32'b0)) //if reset is not 1 and
regWrite signal enabled, write data value written to destination register
      register_file[rg_wrt_dest] <=rg_wrt_data;

end

assign read_reg1 = register_file[rs1];
assign read_reg2 = register_file[rs2];


endmodule
```

ALU

```
module ALUnitHW #(
    parameter DATA_WIDTH = 32
)(
    input logic [DATA_WIDTH-1:0] rs1,
    input logic [DATA_WIDTH-1:0] rs2,
    input logic [3:0] aluControl,

    output logic [DATA_WIDTH-1:0] rd,
    output logic zero
);


    always_comb begin
        rd = 'd0;
        zero = 'b0;

            case(aluControl)
                  4'ha: rd='b0;

            //R TYPE
                  4'h0: rd = rs1+rs2; // ADD
                  4'h1:
                              begin
                              rd = rs1-rs2; //SUB
                              zero =(rd==0)?1'b1:1'b0;
                              end
                  4'h2: rd = rs1<<(rs2[4:0] & 5'b11111); //SLL
                  4'h3: rd = ($signed(rs1) < $signed(rs2)) ? 1'b1 : 1'b0;
//SLT
                  4'h4: rd = (rs1 < rs2) ? 1'b1 : 1'b0; //SLTU
                  4'h5: rd = rs1 ^ rs2; //XOR
                  4'h6: rd = rs1 >> (rs2[4:0] & 5'b11111); //SRL
                  4'h7: rd = $signed(rs1) >>> (rs2[4:0] & 5'b11111); //SRA
                  4'h8: rd = rs1 | rs2; //OR
                  4'h9: rd = rs1 & rs2; //AND
                  4'hb: rd = rs1*rs2; //unsigned multiplication


                  default: rd = 'b0;
        endcase
       end


endmodule
```

Data Memory

```
module Datamemory #(
    parameter DM_ADDRESS = 9,
    parameter DATA_W = 32
)(
    input logic clk,
    input logic Memcopy,
    input logic MemRead,  // comes from the control unit
    input logic MemWrite, // Comes from the control unit
    input logic [DM_ADDRESS - 1:0] a,  // Read / Write address - 9 LSB bits
of the ALU output
    input logic [DATA_W - 1:0] wd,   // Write Data
    input logic [31:0] immout,      // Number of elements to copy
    input logic [31:0] rs1, // Memory address to start copying from
    input logic [31:0] rs2, // Memory address to start copying to
    output logic [DATA_W - 1:0] rd
);

    logic [DATA_W - 1:0] mem[(2**DM_ADDRESS) - 1:0];

       logic state=0;
       logic [1:0]signal=2'b00;
       int i;
       logic [31:0]start,dest,size;

    always_comb begin
        if (MemRead)
            rd <= mem[a];
        else
            rd <= '0;
    end

    always_ff @(posedge clk) begin
        if (MemWrite)
            mem[a] = wd;
        if (Memcopy) begin
            state=1;
                        start=rs1;
                        i=0;
                        dest=rs2;
                        size=immout;
        end
              if(state==1)begin
                  case(signal)
                  2'b00: begin
                        mem[dest+i]=mem[start+i];
                        i=i+1;
                        if(i<size)begin
                              signal=2'b00;
                        end
                        else begin
                              signal=2'b01;
                        end
                  end
```

```
                   2'b01:begin
                         state<=0;
                   end
                   endcase
               end
               end
endmodule
```

Branch Sel

```
module BranchSel(
input logic zero,branch,
input logic rdlast_bit,
input logic [1:0]branch_type,
output logic pcMux
);


always_comb begin
    unique case(branch_type)
        2'b00: pcMux<=(branch & zero);//beq
        2'b01: pcMux<=(branch & (!zero));//bnq
        2'b10: pcMux<=(branch & (rdlast_bit));//blt,bltu
        2'b11: pcMux<=(branch & !(rdlast_bit));//bge,bglu
    endcase
end


endmodule
```

Decoder for Load and store

```
module Decoder3by3 (
    input [31:0] result, // 32-bit input for selecting one of 3 outputs
    input [31:0] Inst, // Instruction input

       input logic [2:0] selection_input,
       output [31:0] out // Output

);
     // 3-bit selection input
    logic [31:0] out_internal; // Internal signal to hold the output value


    always_comb begin
        case (selection_input)
            3'b001: out_internal = {result[7]? {24{1'b1}} :24'b0,
result[7:0]}; //lb or sb
```

```
            3'b010: out_internal = {result[15]? {16{1'b1}} :16'b0,
result[15:0]}; // lh or sh
            3'b011: out_internal = result; // lw or sw
                        3'b100: out_internal = {24'b0,result[7:0]};//lbu
                        3'b101: out_internal = {16'b0,result[15:0]};//lhu
            default: out_internal = 32'b0; // Default value
        endcase
    end

    assign out = out_internal; // Assign the internal signal to the output
endmodule
```

Microprogramming Unit

```
module Microprogramming #(
    parameter OPCODE_LENGTH = 7,
    parameter FUNCT3_LENGTH = 3,
    parameter FUNCT7_LENGTH = 7

)(

      input logic [OPCODE_LENGTH-1:0] Opcode,
   input logic [FUNCT3_LENGTH-1:0] Funct3,
   input logic [FUNCT7_LENGTH-1:0] Funct7,


    output logic regWrite,
    output logic immSelMux,
    output logic LoadMux,
    output logic MemRead,
    output logic MemWrite,
    output logic Con_Jalr,
    output logic BranchSig,
    output logic [3:0]ALUSignal,
    output logic [1:0]SelSignalforBranchSel,
    output logic [2:0]LoadstoreSigodecoder,
       output logic memcopy

);
logic [16:0]Control_signal;
logic[16:0] ROM[40:0];
logic [15:0]sig;

always_comb begin

      ROM[0] <= 17'b01000000000000000; //ADD
      ROM[1] <= 17'b01000000000100000; //SUB
      ROM[2] <= 17'b01000000001000000; //SLL
      ROM[3] <= 17'b01000000001100000; //SLT
      ROM[4] <= 17'b01000000010000000; //SLTU
      ROM[5] <= 17'b01000000010100000; //XOR
      ROM[6] <= 17'b01000000011000000; //SRL
      ROM[7] <= 17'b01000000011100000; //SRA
      ROM[8] <= 17'b01000000100000000; //OR
      ROM[9] <= 17'b01000000100100000; //AND


      ROM[10] <= 17'b01100000000000000; //ADDI
      ROM[11] <= 17'b01100000001100000; //SLTI
      ROM[12] <= 17'b01100000010000000; //SLTIU
      ROM[13] <= 17'b01100000010100000; //XORI
      ROM[14] <= 17'b01100000100000000; //ORI
      ROM[15] <= 17'b01100000100100000; //ANDI
      ROM[16] <= 17'b01100000001000000; //SLLI
      ROM[17] <= 17'b01100000011000000; //SRLI
      ROM[18] <= 17'b01100000011100000; //SRAI

      ROM[19] <= 17'b01100010000000000; //JALR

      ROM[20] <= 17'b01111000000000001; //LB
      ROM[21] <= 17'b01111000000000010; //LH
      ROM[22] <= 17'b01111000000000011; //LW
      ROM[23] <= 17'b01111000000000100; //LBU
      ROM[24] <= 17'b01111000000000101; //LHU

      ROM[25] <= 17'b001x0100000000001; //SB
      ROM[26] <= 17'b001x0100000000010; //SH
      ROM[27] <= 17'b001x0100000000011; //SW

      ROM[28] <= 17'b000x0001000100000; //BEQ
      ROM[29] <= 17'b000x0001000101000; //BNQ
      ROM[30] <= 17'b000x0001001110000; //BLT
      ROM[31] <= 17'b000x0001001111000; //BGE
      ROM[32] <= 17'b000x0001010010000; //BLTU
      ROM[33] <= 17'b000x0001010011000; //BLGU

      ROM[34] <= 17'b00000000000000000;
      ROM[35] <= 17'b01000000101100000; //unsigned multiplication
      ROM[36] <= 17'b101x0000110000011; //memcopy

end


always_comb begin
      Control_signal = ROM[34]; // Default value
      //Selection_signal = 2'b00; // Default value

    case (Opcode)
```

```
        7'b0110011: begin
            case (Funct3)
                3'b000: begin
                    case (Funct7)
                        7'b0000000: Control_signal = ROM[0];  // ADD
                        7'b0100000: Control_signal = ROM[1]; // SUB
                                               default: Control_signal =
ROM[34];
                    endcase
                end

                3'b001: Control_signal = ROM[2]; // SLL
                3'b010: Control_signal = ROM[3]; // SLT
                3'b011: Control_signal = ROM[4]; // SLTU
                3'b100: Control_signal = ROM[5]; // XOR

                3'b101: begin
                    case (Funct7)
                        7'b0000000: Control_signal = ROM[6]; // SRL
                        7'b0100000: Control_signal = ROM[7]; // SRA
                                               default: Control_signal =
ROM[34];
                    endcase
                end

                3'b110: Control_signal = ROM[8]; // OR
                3'b111: Control_signal = ROM[9]; // AND
                                 default: Control_signal = ROM[10];
            endcase
        end
                   7'b0010011: begin
                case(Funct3)
                    3'b000: Control_signal = ROM[10];//ADDI
                    3'b010: Control_signal = ROM[11];//SLTI
                    3'b011: Control_signal = ROM[12];//SLTIU
                    3'b100: Control_signal = ROM[13];//XORI
                    3'b110: Control_signal = ROM[14];//ORI
                    3'b111: Control_signal = ROM[15];//ANDI
                    3'b001: begin
                        case(Funct7)
                            7'b0000000: Control_signal = ROM[16];//SLLI
                                                   default:
Control_signal = ROM[34];
                                               endcase
                                           end
                                  3'b101: begin
                                               case(Funct7)
                            7'b0100000: Control_signal = ROM[17];//SRLI
                            7'b0100000: Control_signal = ROM[18];//SRAI
                            default: Control_signal = ROM[34];
                        endcase
                    end

                    default: Control_signal = ROM[34];
                endcase
            end
                        //7'b0000011 : Control_signal =ROM[22]; //For load
and store functions
                        //7'b0100011 : Control_signal =ROM[27];

                        7'b1100011 : begin //for branch
                              case(Funct3)
                                    3'b000 : begin
                                                      Control_signal =
ROM[28]; //beq
                                                      //Selection_signal =
ROM[28];
                                                      end
                                    3'b001 : begin
                                                      Control_signal =
ROM[29]; //bnq
                                                      //Selection_signal =
ROM[29];
                                                      end
                                    3'b100 : begin
                                                      Control_signal =
ROM[30]; //blt
                                                      //Selection_signal =
ROM[30];
                                                      end
                                    3'b101 : begin
                                                      Control_signal =
ROM[31]; //bge
                                                      //Selection_signal =
ROM[31];
                                                      end
                                    3'b110 : begin
                                                      Control_signal =
ROM[32]; //bltu
                                                      //Selection_signal =
ROM[32];
                                                      end
                                    3'b111 : begin
                                                      Control_signal =
ROM[33]; //blgu
                                                      //Selection_signal
=ROM[33];
                                                      end
```

```
                                  default: Control_signal = ROM[34];
                               endcase
                         end
                         7'b1100111 : begin
                                           case(Funct3)
                                                 3'b000 : Control_signal =
ROM[19];//JALR
                                           endcase
                                     end
                         7'b0000011 : begin
             case (Funct3)
                 3'b000 : Control_signal = ROM[20];//[7:0] lb
                                3'b001 : Control_signal = ROM[21];// lh
                                3'b010 : Control_signal = ROM[22]; //lw
                                3'b100 : Control_signal = ROM[23];
//lbu[7:0]
                                3'b101 : Control_signal = ROM[24];
//lhu[15:0]


                         endcase
                   end
                   7'b0100011 : begin
                               case (Funct3)
                 3'b000 : Control_signal = ROM[25];//[7:0] sb
                                3'b001 : Control_signal = ROM[26];// sh
                                3'b010 : Control_signal = ROM[27]; //sw
                               endcase
                         end
                   7'b0111111 : Control_signal = ROM[35]; //unsigned
multiplication
                   7'b1111111 : Control_signal = ROM[36]; //memcopy
                   endcase
             end


assign sig=Control_signal[15:0];
assign memcopy =Control_signal[16];
assign regWrite=Control_signal[15];
assign immSelMux=Control_signal[14];
assign LoadMux=Control_signal[13];
assign MemRead=Control_signal[12];
assign MemWrite = Control_signal[11];
assign Con_Jalr=Control_signal[10];
assign BranchSig = Control_signal[9];
assign ALUSignal = Control_signal[8:5];
assign SelSignalforBranchSel = Control_signal[4:3];
assign LoadstoreSigodecoder = Control_signal[2:0];
```

```
endmodule
```

Immediate Generator

```
module imm_Gen(
    input logic [31:0] inst_code,
    output logic [31:0] Imm_out
);

    logic [4:0] srai;
    assign srai = inst_code[24:20];

always_comb
    case(inst_code[6:0])
        7'b0000011 /*I-type load*/     :
            Imm_out = {inst_code[31]? {20{1'b1}}:20'b0 , inst_code[31:20]};
        7'b0010011 /*I-type */     :
            begin
            if((inst_code[31:25]==7'b0100000&&inst_code[14:12]==3'b101)||(i
nst_code[14:12]==3'b001)||inst_code[14:12]==3'b101)
                Imm_out = {srai[4]? {27{1'b1}}:27'b0,srai};
            else
                Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:20]};
            end
        7'b0100011 /*S-type*/    :
            Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:25],
inst_code[11:7]};
        7'b1100011 /*B-type*/    :
            Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[7],
inst_code[30:25],inst_code[11:8],1'b0};
        7'b1100111 /*JALR*/    :
            Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[30:25],
inst_code[24:21], inst_code[20]};
        default                  :
            Imm_out = {32'b0};
    endcase

endmodule
```

Multiplexer

```
module Mux2input (
    input logic [31:0] a,
    input logic [31:0] b,
    input logic select,
    output logic [31:0] out
);

  assign out = (select) ? b : a;

endmodule
```

Adder

```
module adder_32
    #(parameter WIDTH = 32)
    (input logic [WIDTH-1:0] a, b,
     output logic [WIDTH-1:0] y);


assign y = a + b;

endmodule
```

Datapath

```
module Datapath(
       input logic clk, reset,
       output logic[31:0] rdvalue,
       output logic[5:0] swval


       //input logic[31:0] pc1
    //output logic [31:0] result,

);
logic [1:0] sel_sig;
logic [31:0]pc;
logic[31:0]Inst;
logic
[31:0]reg1_val,reg2_val,result,result_Aftrl,A_bus,B_bus,B_inbus,C_bus,immOu
t,aluOut,aluOut1,dmemOut,pcIn,pcplusImm,valueplusImm;
logic [3:0] aluControl_signal;
logic [31:0] pcNext,pcIn1;
//logic
regWrite,alucontrol,immSelMux,LoadMux,MemRead,MemWrite,pcBranch,Con_Jalr;
logic isZero, pcBranch1;
logic
```

```
regWrite,immSelMux,LoadMux,MemRead,MemWrite,Con_Jalr,BranchSig,memcopy;
logic [2:0]LoadstoreSigodecoder;



adder_32 adder1(
      .a(pc),
      .b(32'b100),
      .y(pcNext)
);

adder_32 adder2(
      .a(pc),
      .b(immOut),
      .y(pcplusImm)
);

adder_32 adder3(
      .a(pc),
      .b(aluOut),
      .y(valueplusImm)
);

Mux2input muxrdsel(
      .a(C_bus),
      .b(pcNext),
      .select(Con_Jalr),
      .out(rdvalue)
);



Mux2input muxJALR(
      .a(pcIn1),
      .b(valueplusImm),
      .select(Con_Jalr),
      .out(pcIn)
);



Mux2input muxBranch(
      .a(pcNext),
      .b(pcplusImm),
      .select(pcBranch1),
      .out(pcIn1)
);

BranchSel bs(
        .zero(isZero),
        .branch(BranchSig),
        .rdlast_bit(aluOut[0]),
        .branch_type(sel_sig),
        .pcMux(pcBranch1)
);

//pc counter
pc pcreg(
        .clk(clk),
        .reset(reset),
        .pc_in(pcIn),
        .pc_out(pc)
);


//instruction mem
InstructionMemory Imem(
        .address(pc),
        .instruction(Inst)
);

//registers

RegFiles regF(
        .clk(clk),
        .rst(reset),
        .RegWrite(regWrite),
        .rg_wrt_dest(Inst[11:7]),
        .rs1(Inst[19:15]),
        .rs2(Inst[24:20]),
        .rg_wrt_data(rdvalue),
        .read_reg1(reg1_val),
        .read_reg2(reg2_val),
        .extaddress(swval)

);
//sign_extender
imm_Gen immGen(
        .inst_code(Inst),
        .Imm_out(immOut)
);


Mux2input muxImmSel(
        .a(reg2_val),
        .b(immOut),
        .select(immSelMux),
        .out(B_inbus)
```

```
);

assign A_bus = reg1_val;


Microprogramming CU(
      .Opcode(Inst[6:0]),
      .Funct3(Inst[14:12]),
      .Funct7(Inst[31:25]),
      .regWrite(regWrite),
      .immSelMux(immSelMux),
      .LoadMux(LoadMux),
      .MemRead(MemRead),
      .MemWrite(MemWrite),
      .BranchSig(BranchSig),
      .Con_Jalr(Con_Jalr),
      .ALUSignal(aluControl_signal),
      .SelSignalforBranchSel(sel_sig),
      .LoadstoreSigodecoder(LoadstoreSigodecoder),
      .memcopy(memcopy)


);


Mux2input muxmemcopy(
      .a(B_inbus),
      .b(reg2_val),
      .select(memcopy),
      .out(B_bus)
);

ALUnitHW alu(
      .rs1(A_bus),
      .rs2(B_bus),
      .rd(aluOut),
      .rd1(aluOut1),
      .aluControl(aluControl_signal),
      .zero(isZero)
);


Mux2input muxLoadSel(
      .a(aluOut),
      .b(result),
      .select(LoadMux),
      .out(result_Aftrl)
);
```

```
Datamemory Dm(
      .clk(clk),
      .immout(immOut),
      .MemRead(MemRead),
      .MemWrite(MemWrite),
      .Memcopy(memcopy),
      .a(aluOut),
      .rs1(aluOut),
      .rs2(aluOut1),
      .wd(reg2_val),
      .rd(dmemOut)

);

assign C_bus = result_Aftrl;

Decoder3by3 dc(
      .result(dmemOut),
      .Inst(Inst),
      .out(result),
      .selection_input(LoadstoreSigodecoder)
);
endmodule
```

For Hardware Verifications

```
module hexdigit(
      input logic [3:0] in,
      output reg [6:0] out
);

always @*
begin
      out= 7'b1111111;

      case(in)

            4'h0: begin

            out[6]=1'b1;
            out[5]=1'b0;
            out[4]=1'b0;
            out[3]=1'b0;
            out[2]=1'b0;
            out[1]=1'b0;
            out[0]=1'b0;
            end
```

```
4'h1:begin

out[6]=1'b1;
out[5]=1'b1;
out[4]=1'b1;
out[3]=1'b1;
out[2]=1'b0;
out[1]=1'b0;
out[0]=1'b1;
end

4'h2: begin

out[6]=1'b0;
out[5]=1'b1;
out[4]=1'b0;
out[3]=1'b0;
out[2]=1'b1;
out[1]=1'b0;
out[0]=1'b0;
end

4'h3: begin

out[6]=1'b0;
out[5]=1'b1;
out[4]=1'b1;
out[3]=1'b0;
out[2]=1'b0;
out[1]=1'b0;
out[0]=1'b0;
end

4'h4: begin
out[6]=1'b0;
out[5]=1'b0;
out[4]=1'b1;
out[3]=1'b1;
out[2]=1'b0;
out[1]=1'b0;
out[0]=1'b1;

end

4'h5: begin
out[6]=1'b0;
out[5]=1'b0;
out[4]=1'b1;
```

```
            out[3]=1'b0;
            out[2]=1'b0;
            out[1]=1'b1;
            out[0]=1'b0;
            end

            4'h6: begin
            out[6]=1'b0;
            out[5]=1'b0;
            out[4]=1'b0;
            out[3]=1'b0;
            out[2]=1'b0;
            out[1]=1'b1;
            out[0]=1'b0;
            end

            4'h7: begin
            out[6]=1'b1;
            out[5]=1'b1;
            out[4]=1'b1;
            out[3]=1'b1;
            out[2]=1'b0;
            out[1]=1'b0;
            out[0]=1'b0;
            end

            4'h8: begin
            out[6]=1'b0;
            out[5]=1'b0;
            out[4]=1'b0;
            out[3]=1'b0;
            out[2]=1'b0;
            out[1]=1'b0;
            out[0]=1'b0;
            end

            4'h9:begin
            out[6]=1'b0;
            out[5]=1'b0;
            out[4]=1'b1;
            out[3]=1'b0;
            out[2]=1'b0;
            out[1]=1'b0;
            out[0]=1'b0;
            end

            default:
            out= 7'b1111111;
      endcase
```

```
end
endmodule
```

Clock divider

```
module Clock_divider (
    input clock_in, // input clock on FPGA
    output logic clock_out // output clock after dividing the input clock
by divisor
);

    reg [27:0] counter = 28'd0;
    parameter DIVISOR = 28'd500000; // Updated for 100Hz output

    always @(posedge clock_in) begin
        counter <= counter + 28'd1;
        if (counter >= (DIVISOR - 1))
            counter <= 28'd0;
        clock_out <= (counter < DIVISOR / 2) ? 1'b1 : 1'b0;
    end

endmodule
```

Top Module

```
module try(
    input logic clk, reset,
    input logic [4:0] sw,
    output logic [6:0] HEX4,
    output logic [6:0] HEX5
);

logic [31:0] rdvalue;
logic [4:0] swval;

Datapath dp (
    .clk(clk1),
    .reset(reset),
    .swval(swval),
    .rdvalue(rdvalue)
);

Clock_divider cd(
      .clock_in(clk),
      .clock_out(clk1)
);
```

```
hexdigit H2 ({sw[3:0]==swval[3:0]}?rdvalue[3:0]:4'b1111, HEX4);
hexdigit H3 ({sw[4]==swval[4]}?rdvalue[7:4]:4'b1111, HEX5);


endmodule
```