1. Introduction

1.1 Background

With growing advancements in technology, the central processing unit (CPU) continues to play a critical role in computing systems. Designing efficient CPUs remains a key industrial demand. Understanding the difference between the architectures like Harvard or Von-Neumann architectures, Read Only Memory (ROM) and Random Access Memory (RAM), Pipelined and non-pipelined processor, hexadecimal, and decimal number system, and CISC and RISC processors are crucial for the engineering students. Understanding the working of it at the hardware level and how higher-level programming languages are converted to low-level programming language and further into machine code, all this understanding, a computer engineer possess.

The processor we have reused and understood is add-jump processor (short for design and implementation of the minimalist processor) sourced from [1], is a design having Reduced Instruction Set (RISC) with the Harvard architecture. The add and jump instructions are the basic building block for any processor as one is used for arithmetic operations while second is used for skipping the instruction (can be in either direction i.e. forward or backward). These instructions are used for implementation of high level or low-level program.

Understanding how such instructions are designed using provides invaluable insights about the processor system timing, memory design, and digital logic design. The processor is of 16 bits whereas logic blocks bits differ as per the designer requirement. So, this processor design can do jump and add, and the reason for choosing this project is because it can stronger the skills like digital electronics, microprocessors, and computer architecture.

1.2 Motivation

With the growing complexity of the processors in the fields of academic and industries, the add-jump processor is a way to gain hands on experience to test our digital design, microprocessors, and computer architecture knowledge, as it is a demanding field for the

industry for research sector and for higher education. Being simple is its strength and as the CPU is a complex component to design so, this will be a perfect start and the project also incorporates industry standard as well as hobbyist standard tools through which can deeper the knowledge of it by performing the following:

- Design using Hardware Description Language.
- Simulation and debugging.
- Waveform analysis.
- Timing Analysis.
- Implementation on the hardware (e.g. Field Programmable Array).
- To explore the tools like Xilinx Vivado, ModelSim, and Cadence's Incisive and NCSim.

Furthermore, this project joins the gap between understanding to experimentation, hence for deep research. These are helpful for more advanced topics in computer architecture.

1.3 Objectives

The primary objectives of this project are:

1. **Memory Design**

   To design a memory which is power efficient and is devoid of the contention.

2. **Simulation and Debugging**

   To design and simulate the pre-existing theory of the add jump functionality from the video display processor, employing tools like ModelSim to verify functionality and debug errors.

3. **Visualization of Processor Behavior**

   To use the GtkWave for viewing signal transitions, keeping track of how instruction

given by the user is implemented inside it to get the output.

4. **Proficient in Verilog HDL**

   To use Verilog to create smaller modules like full adder, priority encoder, counter, and decoder and for larger projects like top module and for testbenches.

5. **Documentation and Analysis**

   To use excel to capture different inputs and their outputs for checking the correctness of our design.

1.4 Scope of the Project

It is a great project for the academia and can be used for educating the upcoming engineers. This project will give a good overview on the Von Neumann Architecture. It is a single-cycle processor so, to make students remember RISC with an example like this, as it shows how encoding is done. It has a capacity to perform 16-bit addition, and will not include interrupts, pipelining and various other units present in it.

1.5 Contributions of the project

This project contributes in several key areas:

- **Educational Value**

  It serves as an introduction to processor design for students and beginners, providing exposure to the tools for HDLs and simulation purposes.

- **Industry Recognizable**

The project is a foundation for the future work as by the processor can facilitate in understanding the complex projects and processors we see today (from the companies like Intel, AMD, or Apple) can be seen in a different way after going through this simple project.

- **Practical Insights for the memory design**

  The electronics engineering students will get to know how processor can be implemented by making it themselves.

1.6 Expected Outcomes

Upon completion of this project, the following outcomes are expected:

- Add and jump instructions are working as intended to be.
- All types of inputs are covered so, no corner cases exist without debugging it.
- Expected and Actual waveform
- Successful output in the various tools which verifies it functionality.
- Implementation ready project for the FPGA.

2. Literature Review

2.1 What is CPU? What are its features?

Before we dive straight to the CPU which is a vast topic, we have to gain knowledge of the few terms like what are bits, how humans can communicate with the machine and why we use hexadecimal notation in our design, which can be explained as given below:

- **Digital**

  When we measure anything that is continuous like temperature, water, etc. it can be

done in two ways- digital way (specific value which is an integer) and analog way (long values which are rational numbers). If we take Digital method which is simple to understand similarly is the case with the circuits around us. For example: Clock is a digital circuit which employs embedded circuits like counters and decoder.

- **Bit**

It is the smallest unit of representation, and is short for digit in the binary form, when more than one bit exists say 8 bits it is termed as 1 Byte. This 1 Byte can follow any representation like octal, decimal, and hexadecimal each having different bit-patterns.

- **Hexadecimal Notation**

It is the most popular notation used for representing binary forms, in a simple manner just like decimal. It's range if we compare is slightly different which means that the first ten numbers count is equivalent to decimal 10 and the remaining six count is by the first six letters of the English. A, B, C, D, E, F which makes it to total fifteen numbers.

- **Instructions**

It specifies the task to be performed by the processor. It is found in the documentation of the processor you are using, and normally it is written in the low-level assembly programming language which is further converted to the binary bits, to perform the function as per the operation code. These are encoded normally in a following way:

Opcode	**Operand A**	**Operand B**

Figure 2.1: Instruction Encoding

- **Program**

  It is the set of instructions that are written by the user to perform certain computation or task. It is written mostly in the languages like Embedded C.

- **Logic Gates**

  It is defined as an integrated circuit which can perform logical operations like And, Or, Nor, etc. Nor and Nand gates are the universal gates, meaning any system can be realized using only or any of these two gates.

Definition and Features:

A microprocessor is a computer processor in which data processing logic and control is include in a single integrated circuit. [1] Its features are given below:

- Processor works with bits at the lowest level and it can be an n- bit processor where n is the number of the bits it can support.
- It operates on a single power supply usually named as VCC. Like it clocks oscillators are also present in it which is powered from a single clock generator circuit.
- It operates on a clock cycle normally with 50% duty cycle for the sequential circuits and can be for combinational inputs as well.
- It has address and data lines which is of the form 2^n which gives total memory locations possible where n is the no of the address lines or bits of the address line and data line is similar to the operands whose size is determined from its specification.
- The addressing mode specifies how the data is specified in the instruction, and it is of many types like Immediate addressing mode, Register addressing mode, Direct addressing mode etc.
- It has ALU which performs binary addition with or without carry, and can be extended further like logical operations etc. as per the designer's choice.
- It provides I/O interfacing and control signals for further processing of the peripherals like keyboard and decision making between where to read our instruction from for the

next cycle.

2.2 Existing Processor Architectures

The architecture is important to be understood as it provides how our data when given as input to it, leads to a certain output. There are several architectures before discussing that we need to be aware of how processor works in general. There exists various architecture specifying this instruction sets and each have it own advantage like RISC is a simple, fast and efficient architecture whose implementation can be seen by the examples of it like Arm Cortex-A9 and PIC32MX. There is another aspect of the architecture which is Von Neumann and Harvard Architectures. Both architectures are based on how the instruction and data is fetched from the memory. Memory can of many types like Primary memory or Data memory. Few architectures employ both and few only one. Why we have used architecture word here is the fact we will get to know after we have designed this project.

2.2.1 Von Neumann:

It was the architecture on which 8085 was based on. It was John von Neumann in the 1940s, who proposed this idea. When processor's instruction and data are stored in the same memory location it is said to have Von Neumann Architecture. It has several advantages like lesser components, simpler hardware, more cost effective.

2.2.2 Add and Jump Instructions

RISC & CISC based processor have a add and jump instructions as given in the following table 2.2.2.1.

Table 2.1: Example showing differences in between the instructions for RISC and CISC machine.

Instruction	RISC		CISC
Add	li x2, 3	# x2←−3	MVI C, 08H
	li x3, 4	# x3←−4	MOV A, C

	add x1, x2, x3 # x1←−x2+x3	ADI 20H MOV D, A
Jump	JAL x0, offset	JMP addr

2.3 Minimal Instruction Processors in Academic Research

Minimal Instruction Processors are the reduced instruction set computer which are made to study efficiency of the instruction set, hardware resource optimization. It is sometimes referred to be as single instruction computer as well. It has vey niche applications.

The Minimal Instruction Processor has roots in early RISC research in the 1980s, where the architects it has been observed that out of the processor's instructions, it involves few essential instructions by which the processor can be changed to stay within that restriction and thus reducing the hardware complexity.

The processor is ideal for educational purposes if it is devoid of the real-world complex systems processors. It is therefore to be used understanding the architecture with simple essential instructions. The data path knowledge it can give which tells about wiring inside the processor and its unit's which is useful for learning about how analysis of the blackbox is done, here blackbox refers to a block which has several components and for simplicity it is visualized as a one big block having two simple signal that are externally connected with inside to get the desired output.

The classic 8085 microprocessor is also a complex architecture if we see and it has a more complex control unit, whereas small processors have less instruction set which can help to compare the efficiency with the reference microprocessor 8085. We can implement it onto the FPGA to record the results and these results can disclose timing behavior of our design, through which it's performance can be predicted for the comparison. It is encouraged that the institutions to give the students an exposure to hands-on- experience of it which will further stronger the theory for the coursework.

2.4 What is FPGA?

It stands for Field Programmable Gate Array. These are reconfigurable silicon chips featuring programmable logic blocks and interconnects, facilitated by CAD tools for efficient design, offering flexibility and lower development costs with potentially higher unit costs and lower performance compared to ASICs. It does not involve coding. From market perspective two major companies manufactures FPGA that is Altera and Xilinx. FPGAs have logic blocks, I/O blocks, Interconnect and Memory blocks. Logic blocks have logic cells which further have LUTs, and other circuits like full adder and d-type flip-flop. LUTs stands for look up tables which like a truth table which is customizable as per our application. The interconnects have horizontal and vertical signal and intersection of these are switch matrices, which is also called as FPGA fabric. FPGA is a big world, and if we talk about the development process in it, it has two stages: Implementation and Verification.

Figure 2.2: Implementation Process Cycle

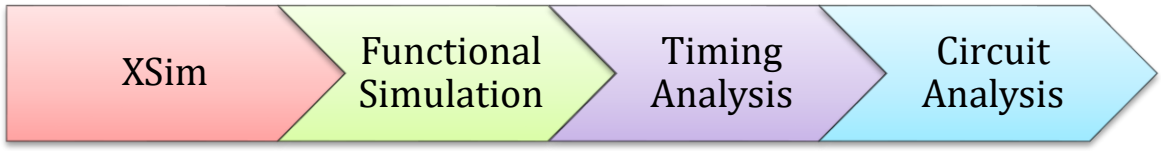

Figure 2.3: Verification Process Cycle

We have used ZCU104 [2] FPGA from the UltraScale+ family having called XCZU7EV. If we see there exists part number when we choose boards in the Xilinx Vivado Simulator, which provides more details about the FPGA (e.g.: xczu7ev). Below are the references of it along with the image of the ZCU104 (see Figure: 2.4).

Table 2.2: FPGA Abbreviation

Abbreviation	Full-Form
ZU	Zynq Ultra Scale+
7	Device size
EV	Video Codec
XC	Xilinx FPGA
ZU7EV	FPGA Model
-2	Speed Grade (Medium)
FFVC1156	Package type & pin count (1156 pins, flip-chip)
E	Temperature Range (Extended, Industrial, etc.)

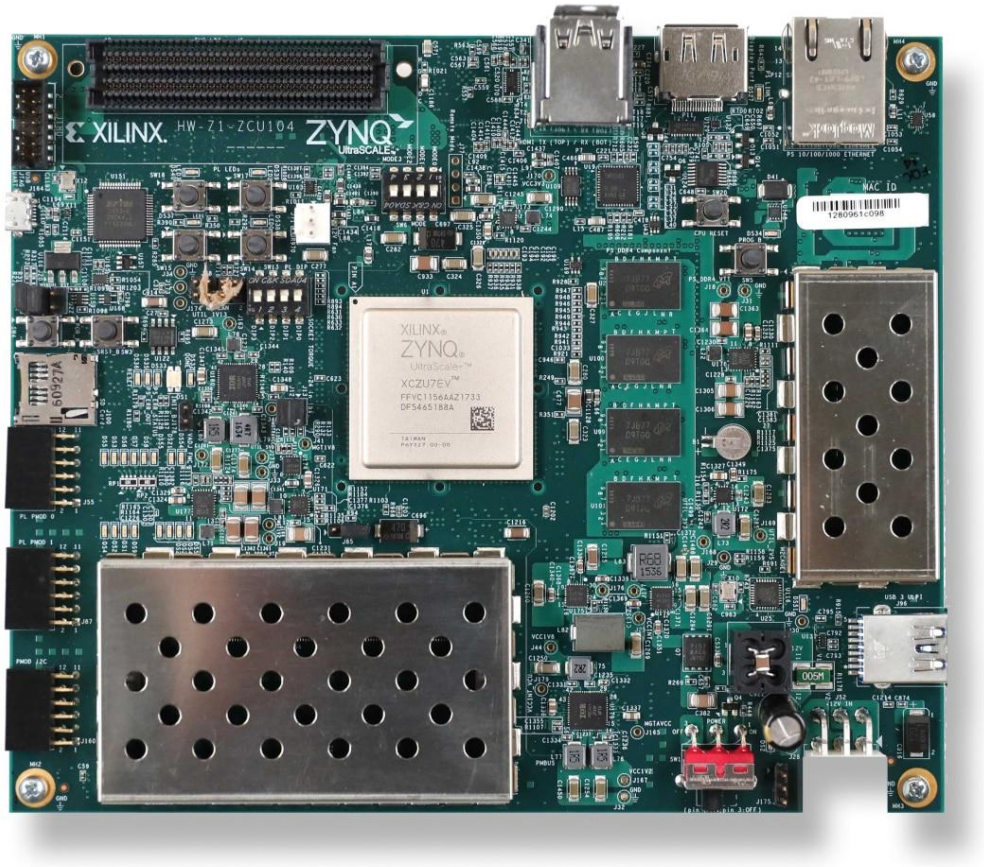

Figure 2.4: ZC104 [6] having several dip switches from C & K several pushbuttons and PMOD ports

2.5 Simulation tools

- **Modelsim Tool**

It is a tool used for compilation of the Verilog code for running the simulation. It does not include lint specifically as it is used to check functional correctness rather than syntax check (i.e. linting). Its operational structure involves vlib and its mapping vmap [2]. Vlog is the name of the compiler (same for TCl command for running Verilog files) is used for the design files and .mpf extension files for the analysis and compilation. Mpf is the project file which keeps the Verilog files in an organized manner and the vsim is the name of the simulator ModelSim supports. To sum it up, we can create a project and add Verilog files into it. By default, our designs are saved in the work library. It can be a different library as well called as resource

library which is provided by the external companies.

Figure 2.5: Logo of the ModelSim Simulator

- **NCLaunch – Cadence** [3]

NCLaunch is a tool which is launched in June 2000, by the Cadence suite to organize its toolchains. It is a interface that runs on Linux operating systems and provides interface to the different compilers. We use it for checking and compiling HDL or VHDL code and run for compilation. It is much more powerful due to it advanced tools and checking. It uses incisive in the older version for HDL analysis and in latest version it is changed into Xcleleum. It can be run in two modes either single run or multi-mode run. Single run is preferred for the small designs while later is for big designs. Below diagram tells us about the sequence for the overall flow for the RTL analysis and simulation.

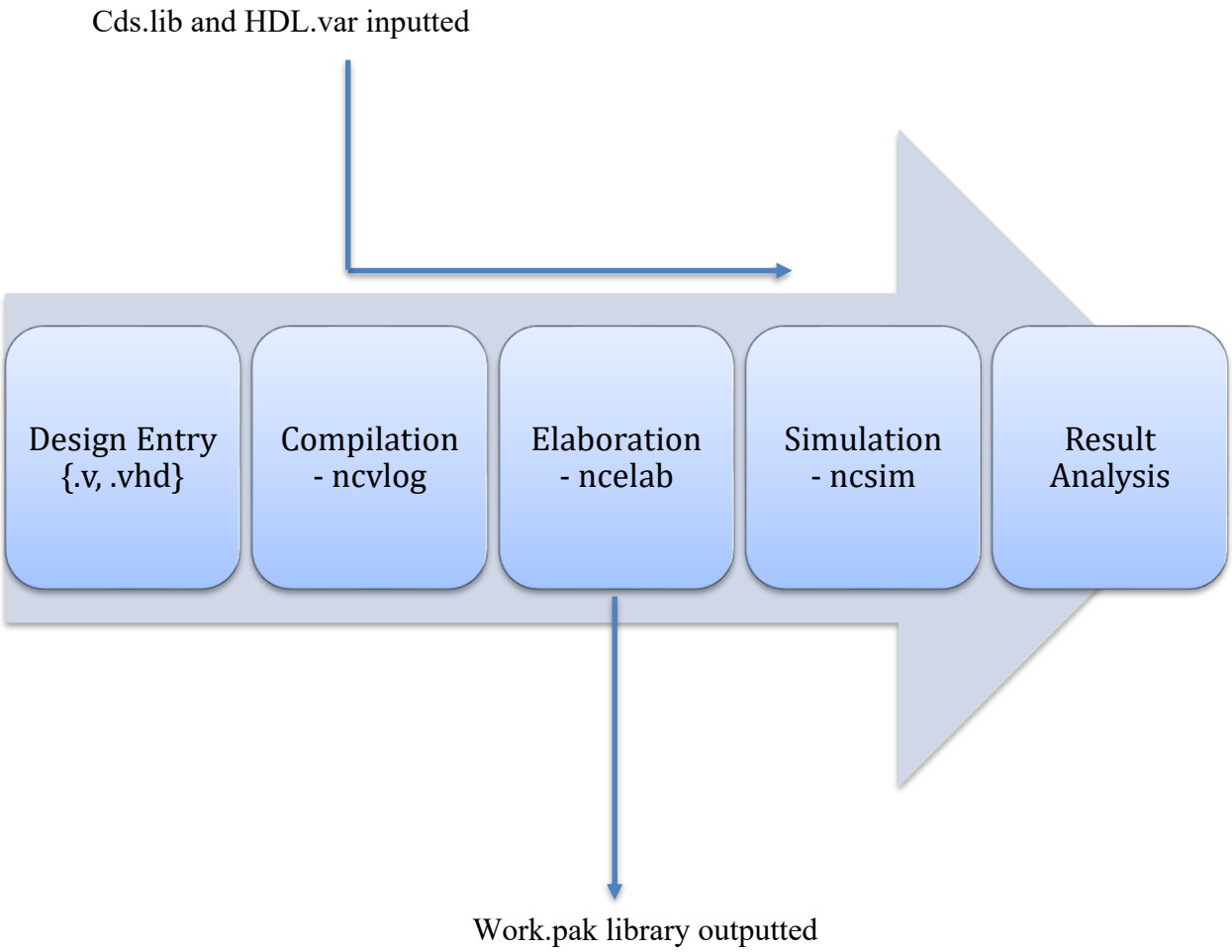

Figure 2.6: Design flow for the functional Simulation- nclaunch

- **Xilinx Vivado Tool**

It is a powerful tool for the Hardware Implementation in the AMD FPGAs. Before Vivado, Xilinx ISE was the main tool for FPGA/CPLD design. Due to less capability for handling the complex or high-end FPGA, Vivado is introduced in April 2012. It is tool we use for FPGA synthesis and SoC design. It is design flow [4] (as seen in the below Figure 2.7) includes RTL analysis which is same as linting, Synthesis which is done to convert our code into gate level netlist, Implementation is done to map our logic with the FPGA blocks and finally the Bitstream generation which is the binary file for configuring the FPGA.

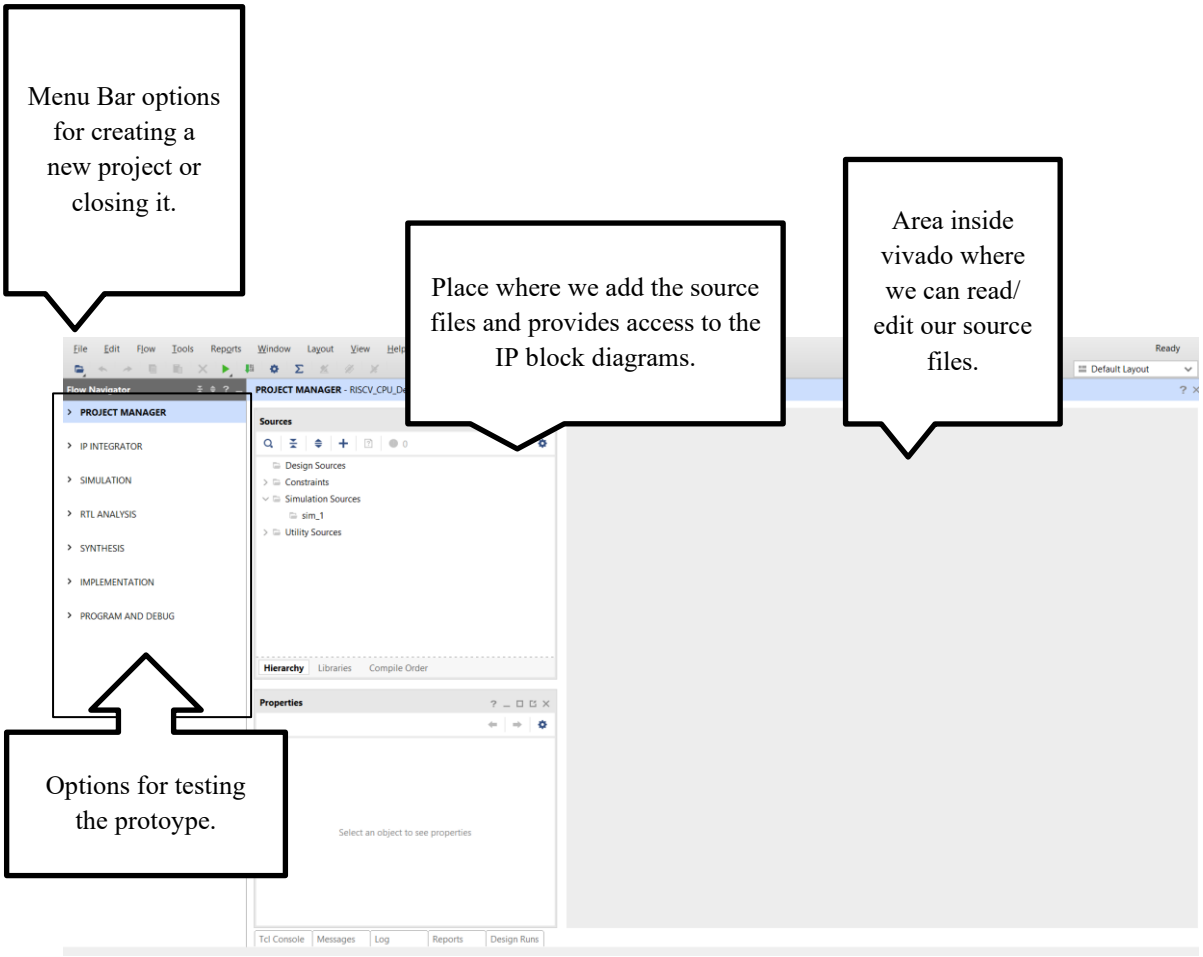

Figure 2.7: Vivado Tool Environment

- **Customasm Tool**

Customasm is a highly flexible, open-source assembler [5] designed for user-defined instruction sets. Developed by hlorenzi, its primary purpose is to translate assembly language code into machine code for processors that do not have existing toolchains. It can also be used for ASIC and FPGA engineers for creating an assembler for their new processors.

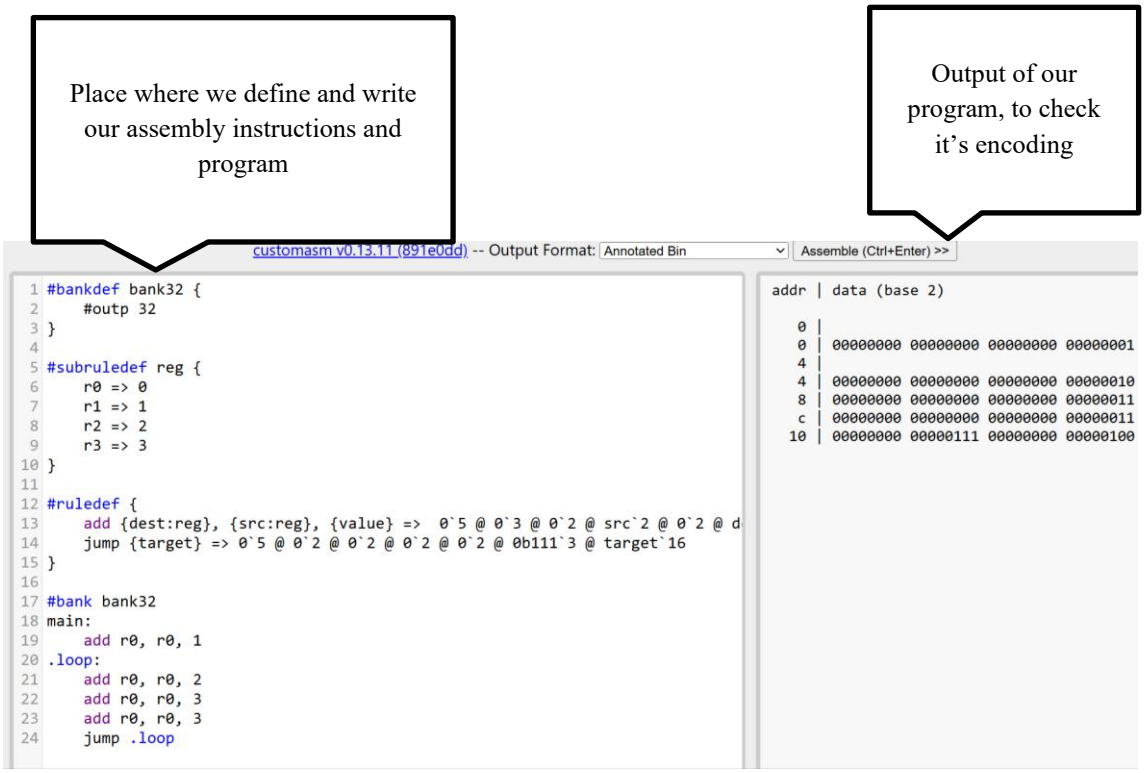

Figure 2.8: Customasm – Online Tool Interface.

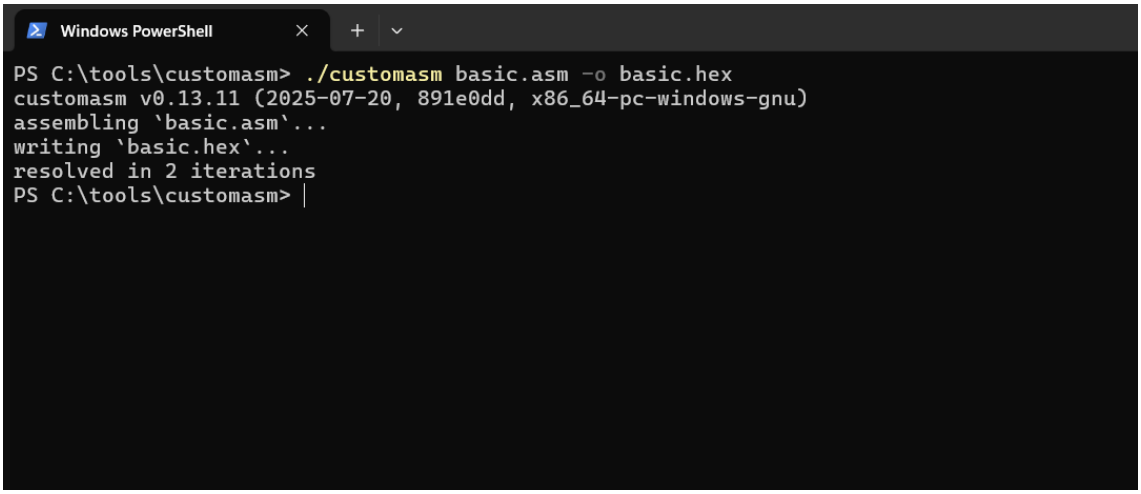

Figure 2.9: Command Line Interface to run the Customasm.

- **Digital Tool**:

It is an open-source tool [6] used for creating logic blocks involving digital electronics components like priority encoder, counter, adder, and logic gates. Embedded Circuits functionality is used to create the custom blocks. It have drag and drop functionality which is used to add the components or remove it etc. From the image given there is a ribbon located at the top of the used to add the components as per the designer. The components properties can be changed by right clicking on it. Editing and simulation both occurs separately meaning once the play button is clicked you cannot modify the circuit. For panning use right click and hover over. The tool is not robust for hardware design because it neglects hardware integration. The output functionality of the complex circuit can be understood by its waveform plotting functionality, which can be saved in .csv format. Its interface is given in Figure 2.10.

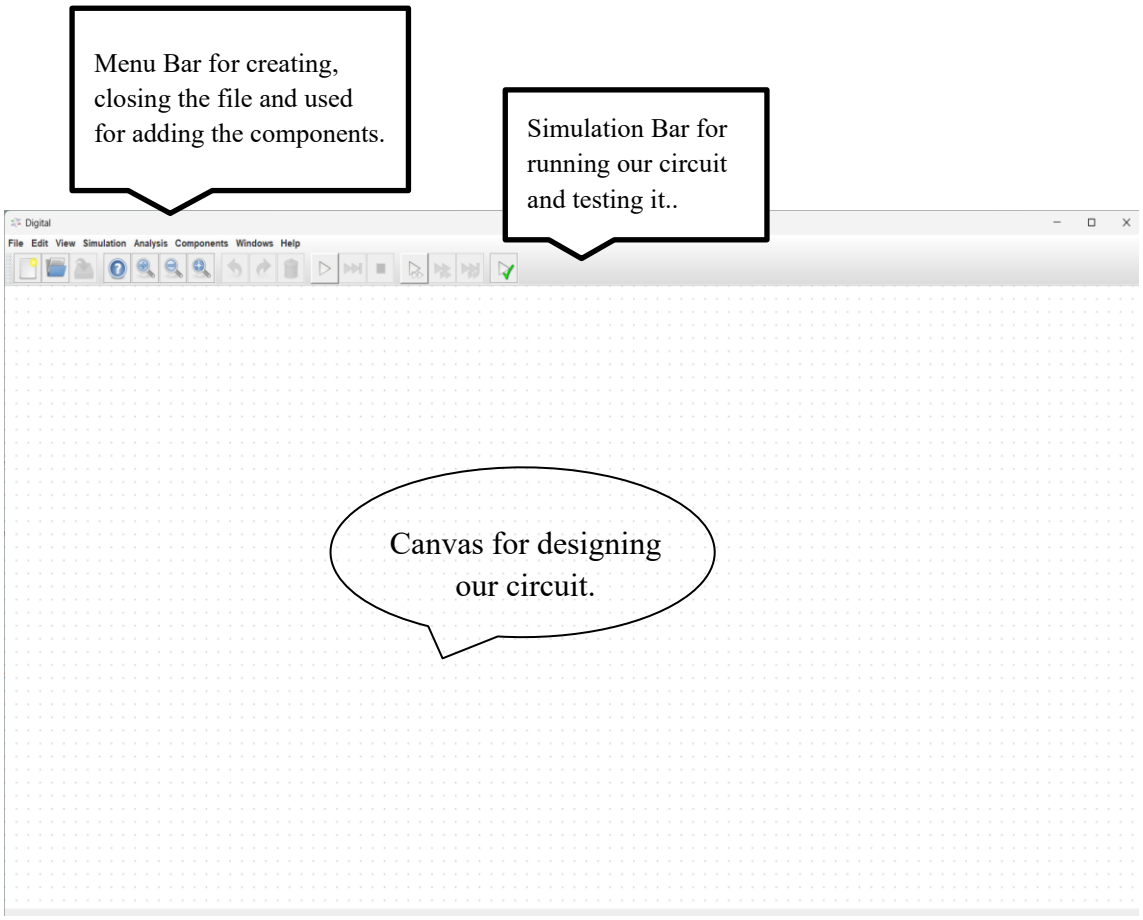

Figure 2.10: Digital Tool Environment

2.6 List of the Components

There were many components which are used in our design. In this list tools-based components are described.

- **Encoder**

  It is a combinational circuit. It is used in compression. The bit inputted at which line determines the output. It has a form of 2^n x n, where n is the number of outputs. It is not efficient when inputs are more than one or when all inputs are zero. This ambiguity is solved by its enhanced version called as priority encoder.

- **Priority Encoder**

  It is a combinational circuit which is used mostly instead of the traditional encoders. It gives priority to the highest input. It has any line which omits all 0s drawbacks of the conventional encoder.

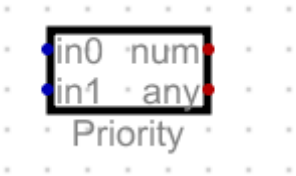

Figure 2.11: Digital Component – Priority Encoder

- **Decoder**

  It is used to decode the input signal into matching binary output line high. It is of the form n cross 2^n where n is the number of input lines. Its circuitry involves basic logic gates.

Figure 2.12: Digital Component – Decoder

- **Multiplexer (MUX)**

  It is widely used and can be as complex as 32 cross 1 or as simple as 2 cross 1. It has a form of 2^nx1 where n is the select lines. The input bits have no effect on the output in comparison to the encoders/decoders. The select line is used to select the input line.

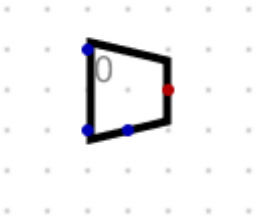

Figure 2.13: Digital Component – Multiplexer

- **Demultiplexer (DeMux)**

  It is opposite of multiplexer. It has 1 cross 2^n form, where n stand for select line, and here form refers to how input output is related.

Figure 2.14: Digital Component - Demultiplexer

- **D Flip-Flop**

  It is a sequential circuit which is formed from latch. It is an edge triggered circuit indicated by the triangle symbol on it clock pin. It is a common mode of storage. Mostly used in registers present inside the central processing unit (Unit). It has four signals, out of which D is the data signal C is for clock and Q is the output which is data when C is high else it remains same.

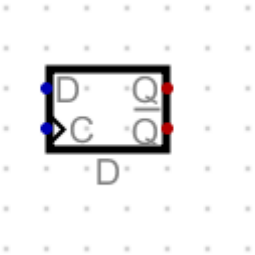

Figure. 2.15: Digital Component – D type Flip-Flip

- **Counter**

  It is the most popular application of the sequential circuits [7]. It is composed of JK flip flop or T flip flop as it has toggling. It can count from 0, or from n or can be mix i.e. hybrid. It is of two types synchronous and asynchronous.

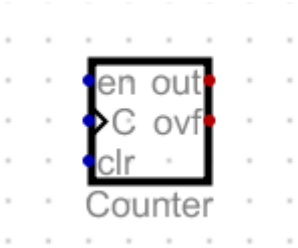

Figure 2.16: Digital Component – Counter

- **Counter with preset**

This counter is different from the normal countar as it can have preloaded value. It has various signals like enable, clock, dir, in, ld, clr, out and ovf signals. The pin-out is further explained in the Table 2.6.1

Table 2.3: Pin-Out Specification for the Counter with preset module.

S.No	Pin Name	Pin Label	Description
1.	Enable	**en**	Can be high or low.
2.	Clock	**C**	Positive edge triggered synchronous circuit.
3.	Direction	**dir**	It specifies the direction i.e from 0 else from n.
4.	Input	**in**	The data that which is stored only if ld pin is high.
5.	Load	**ld**	It can be logic one or logic 0.
6.	Clear	**clr**	It resets the counter if this pin is made to be high.
7.	Output	**out**	Returns the counted value
8.	Overflow	**Ovf**	When Out is maximum, then this pin becomes logic 1.

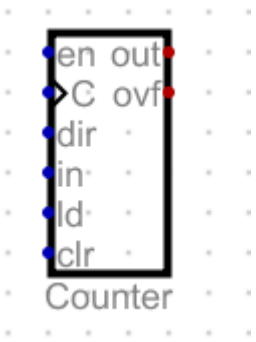

Figure 2.17: Digital Component – Counter with Preset

- **EEPROM (Electronically Erasable and Programmable Read Only Memory)**

It is a storage block which can store multiple instructions. It can store results independent of power supply, called as non-volatile memory [8]. The pins are:

Table 2.4: Pin-Out Specification for the EEPROM Module.

S.No	Pin Name	Pin Label	Description
1.	Address	a	Address of the data.
2.	Data input	din	The data to be store in the EEPROM.
3.	Store	str	This signal can be high or low. If it is high the inputted data is stored. It depends on clock signal
4.	Clock	C	It is the clock signal with positive triggering.
5.	Load	ld	It can be high or low. If it is high the contents EEPROM read cycle begins irrespective of the clock.
6.	Data Out	D	Loading of the stored data

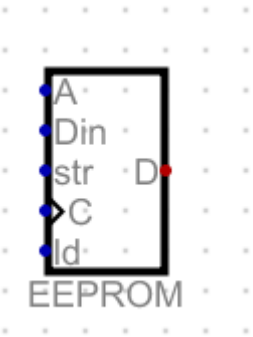

Figure 2.18: Digital Component - EEPROM

- **Adder**

It is a simple circuit which does addition of two numbers. From the image given below it is clearly seen it is a simple arithmetic component of full adder.

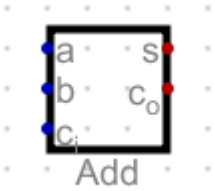

Figure 2.19: Digital Component – Adder Block

- **DIP Switch**

  It stands for Dual Inline Package. It is a simple on/off switch. It is more commonly used in printed circuit boards.

Figure 2.20 Digital Component – DIP Switch

- **Buttons**

 A simple push button which goes back to its original state when it is released.

 Can be used to interactively manipulate an input signal in a circuit with the mouse.

 Can be used to display an output signal in a circuit.

Note:

A symbol which has a triangular shape is called the Tunnel and is used to connect the components with same name without using wire.

- **AND Gate:**

  It is the logic gate used to show case effect of input zero on the output. The gate show logic high only when both inputs are logic one else logic zero.

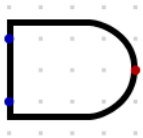

Figure 2.21: Digital Component – 2 Input AND gate

3. Methodology

3.1 Design Methodology and Architecture

The processor design was executed in two distinct levels to incrementally build complexity.

Level 1

Arithmetic Core: The initial phase focused on creating a simple arithmetic unit. Taking architectural cues from the 8085 microprocessors, a basic ADD instruction was implemented. This foundational circuit included not just an adder but also a rudimentary instruction decoder and memory block to fetch and interpret the 32-bit instructions.

Level 2

Control Flow Implementation: The second level introduced control flow logic. The core enhancement was the implementation of a JUMP instruction. To facilitate this, the Program Counter (PC) was redesigned from a simple up-counter to a counter with a parallel load capability, enabling it to be updated with a target address from the jump instruction.

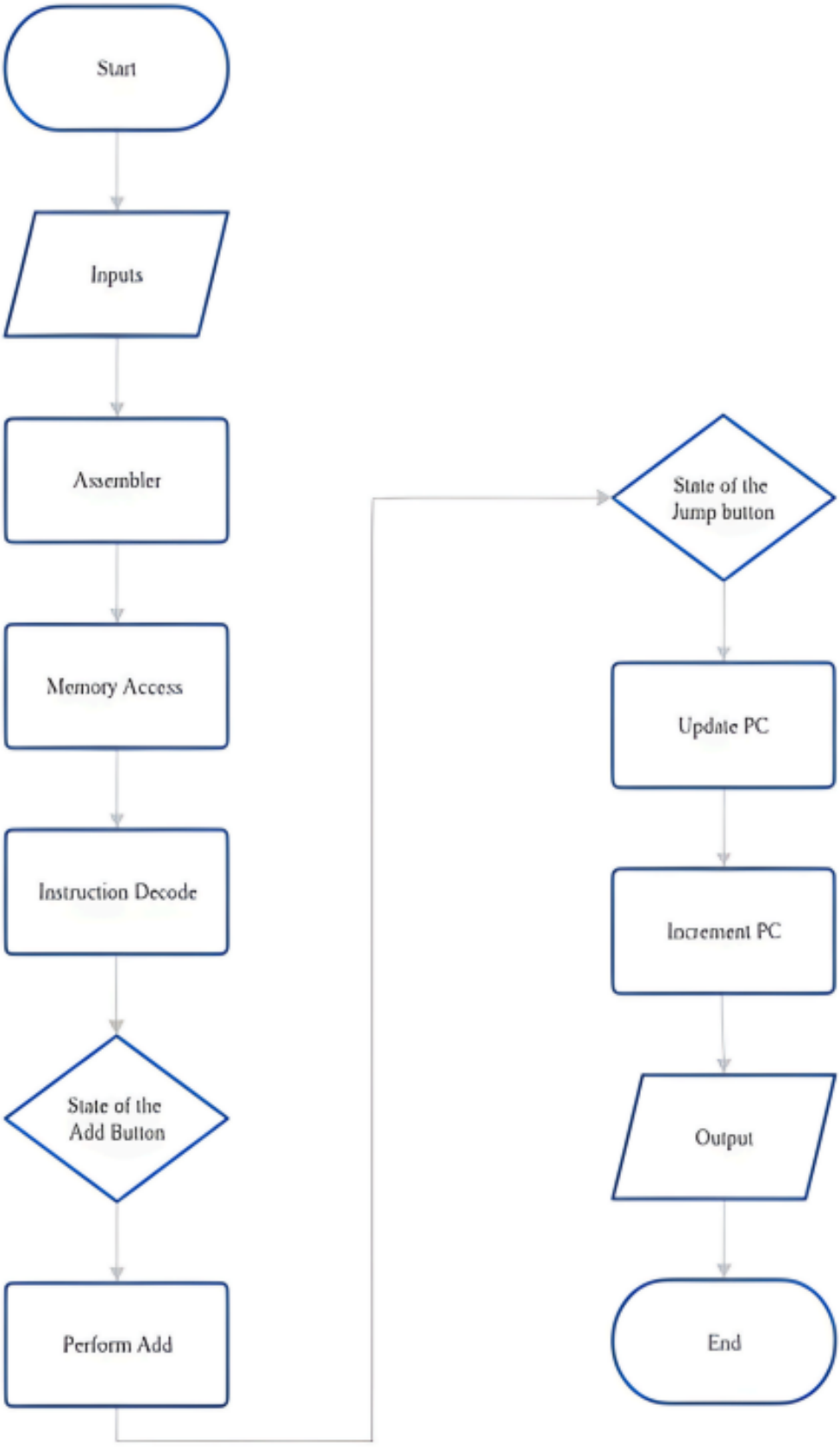

Figure 3.1: Flow Chart showing the working principle of the design

3.2 Tools and Implementation Flow

A specific toolchain was selected to support the project from design to hardware verification. **Schematic Design:** The Digital logic simulator was used for block-level design. Its lightweight framework, ease of use, and ability to generate Verilog code made it an ideal choice for rapid prototyping.

Simulation: The Verilog model was rigorously tested using ModelSim. A set of hexadecimal test cases was used to perform functional simulation and verify the logical integrity of the design before synthesis.

Hardware Platform: The Xilinx ZC104 FPGA was chosen as the target hardware. Its high density of I/O ports was necessary to accommodate the numerous inputs required by the test cases.

3.3 Hardware Verification

A top-level Verilog module was created to instantiate the processor and map its I/O to the FPGA's physical pins, with inputs connected to on-board switches. To overcome the limitation of I/O pin availability for output monitoring, the Xilinx Integrated Logic Analyzer (ILA) was integrated into the design. The ILA provided a virtual oscilloscope, enabling the capture and analysis of internal signals directly on the hardware, thereby confirming that the design operated as simulated. The overall workflow is initiated by loading the program into memory, after which the processor sequentially fetches and executes instructions, with the counter incrementing until the program halts (as shown in Figure 3.1).

3.4 Schematic:

- **Arithmetic Core - Level 1**

  The block diagram is show below having signal a which is the value or operand we have given as input to our design and signal sd is the source destination, which is used for choosing the output register and for showing add operation result.

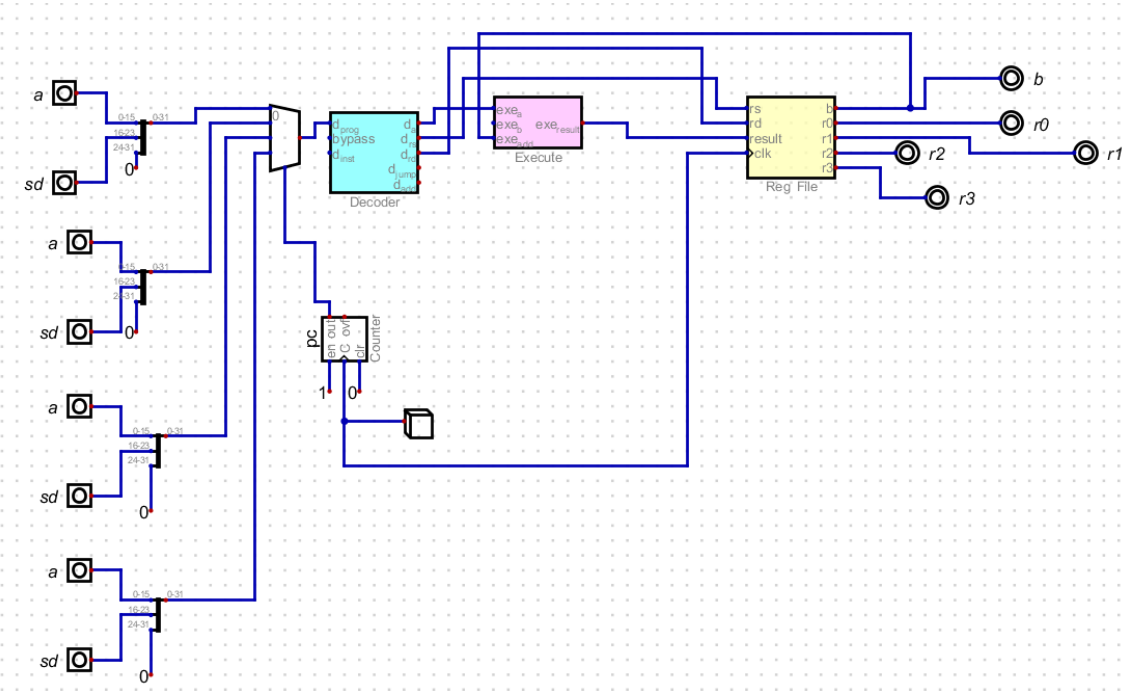

Figure 3.2: Block diagram of initial making of the CPU for the add instruction.

- **Control Flow Implementation – Level 2**

  The schematic of adder is modified in greater extent and can be divided into left hand and right-hand side indicated which has 6 blocks (see Figure 3.4).

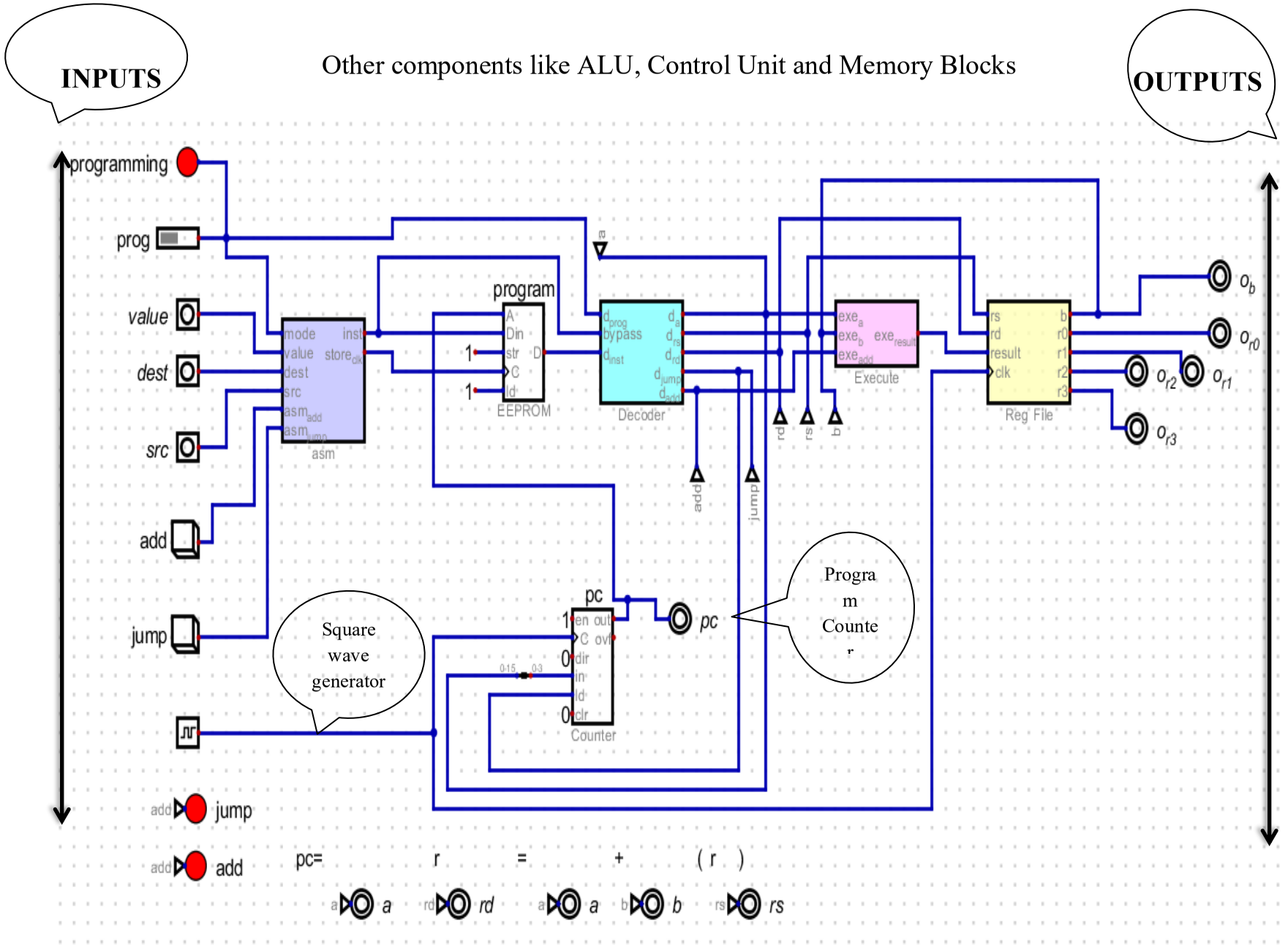

Figure 3.3: Block Diagram Made by using Digital Tool for Add-Jump Processor in which inputs are assembled in block 1 which are decoded further in the block 3 and further executed and stored in the register file block. Below most symbols (called as tunnels) is used for debugging.

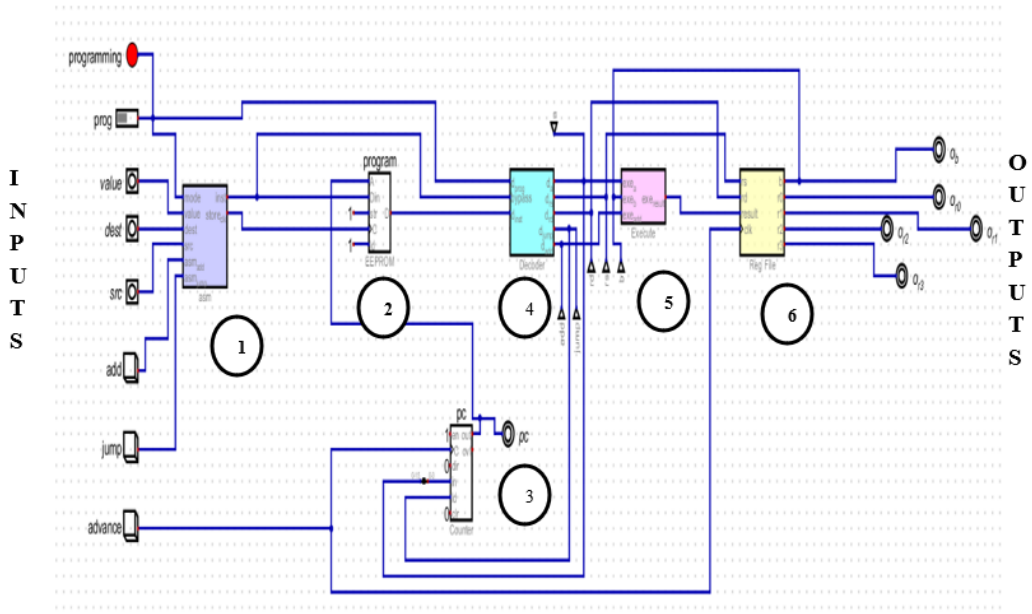

Figure 3.4: Block Diagram of the Design- Numbering of the modules

Value signal for both Add and Jump address so there is no externally specified address bus. Given below is the pin-diagram of our add-jump, in which input data bus is the first fifteen bits of the value (Check block 1 which will be discussed in later chapters), and output data bus is the result of the program. The address-bus bits can have 2^4 which is equals to 16 memory locations, hence is the program counter's output (which is a block three in the Figure 3.4).

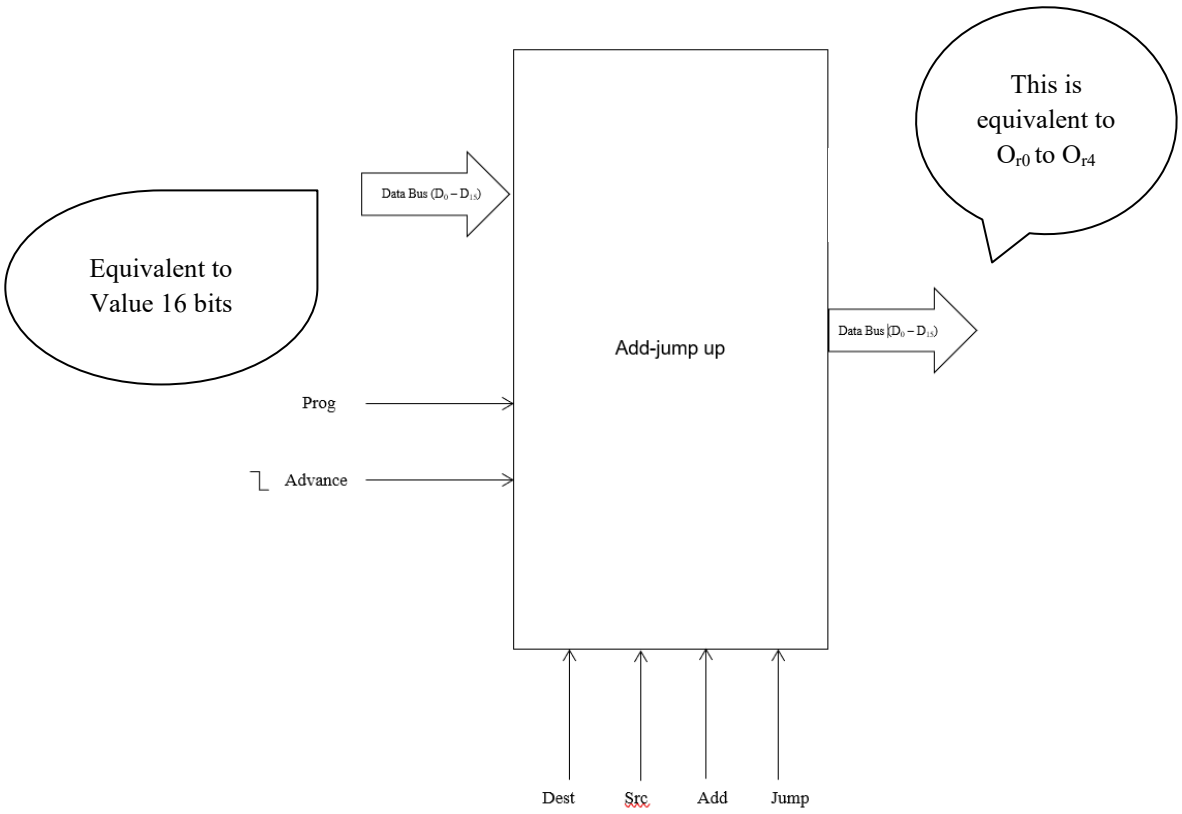

Figure 3.5: Design Block Diagram

On the input side (as given in the Figure 3.5) we have following definitions [1]:

- **Programming LED**

  Red Led indicator which shows programming mode status.

- **Mode Signal**

  It is a on or off DIP switch.

- **Value Bus**

  It is a 16-bit value called as operand.

- **Dest Signal**

  It is the destination register address of 2 bits which tells the register number for storing of the value inputted.

- **Src Signal**

  It is the source of 2 bits which tells which register to read from.

- **Add Signal**

  It is the push button which turns on the add line for the purpose of adding.

- **Jump Signal**

  It is the push button used to enable the jump instruction.

- **Advance Signal**

  This button is for clocking effect.

- **Asm Block**

  An embedded block responsible for processing of the inputs and creation of the inst (short for instruction) and $store_{clock}$ signals.

- **Program Block**

  This block is of EEPROM and is used to store the instruction and the data in a non-volatile manner. It receives inst and store clk signal at inputs and gives output as per its logic.

- **PC Signal**

It stands for program counter due to similar behavior of it. It increments the address value for the EEPROM and helps to store value at a particular address. The circuitry is mainly composed of any toggle flip-flop with preset and clear functionality, which is exploited for the execution of the jump instruction.

How the components join all together? To understand this we need to understand, the full block diagram divided in Left- and Right-Hand Side. For Simplicity we can break the block diagram into two sections which are discussed as below:

At the left-hand side (See Figure 3.6) there exists two main blocks i.e. assembler and decoder and a memory block for seeing its action when the power turns off or prog mode is off. It is like an ASIC design i.e. once programmed for specific addition or jump; the processor will work like that.

How does the data flows or what is its working principle?
Working is as per encoding of the instruction currently assembly is not defined but through binary you can assume that first few bits are like an opcode i.e. if it is 111 processor will jump and if it is 000, it will add. As all other inputs are left floating, or for simplicity made equals to zero, the processor will add in most of the input values. Now, question will arise then what is the purpose of add button if it will automatically add it. If we think deeply and go through the embedded circuits given below, the answer will be clearer.

On the output side or right-hand side (see Figure 3.7) we have following definitions:

- **b_{read}:** it shows contents of the flipflop as per rs select signal combination
- **r0:** it is the first flipflop inside the reg file block
- **r1:** it is the second flipflop inside the reg file block
- **r2:** it is the third flipflop inside the reg file block
- **r3:** it is the fourth flipflop inside the reg file block

Note:

Tunnels are utilized for debugging purpose (right-side below). It is used to indicate which instruction is currently active. The addition part is also added for clarity.

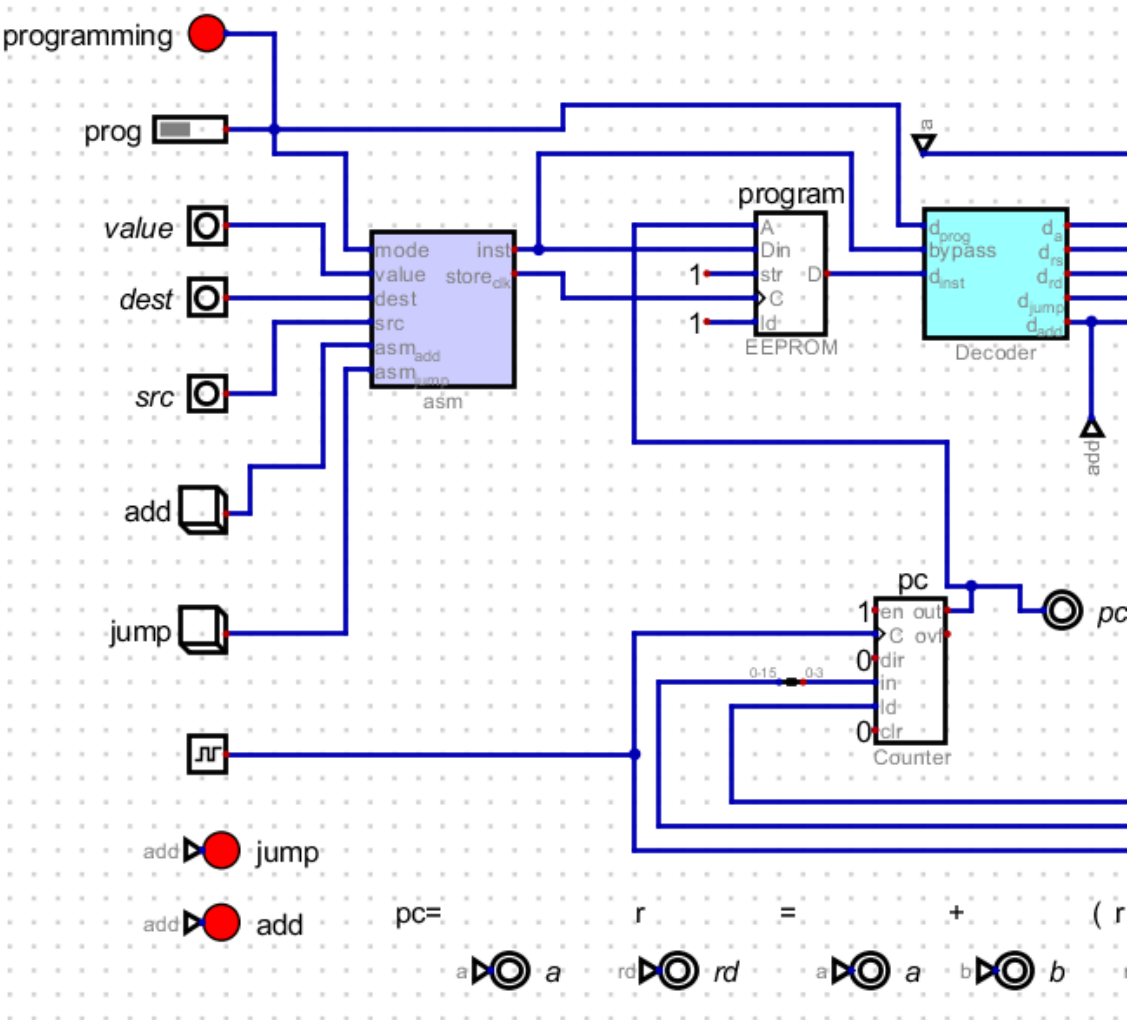

Figure 3.6: Left Hand Side of the Add-Jump Processor

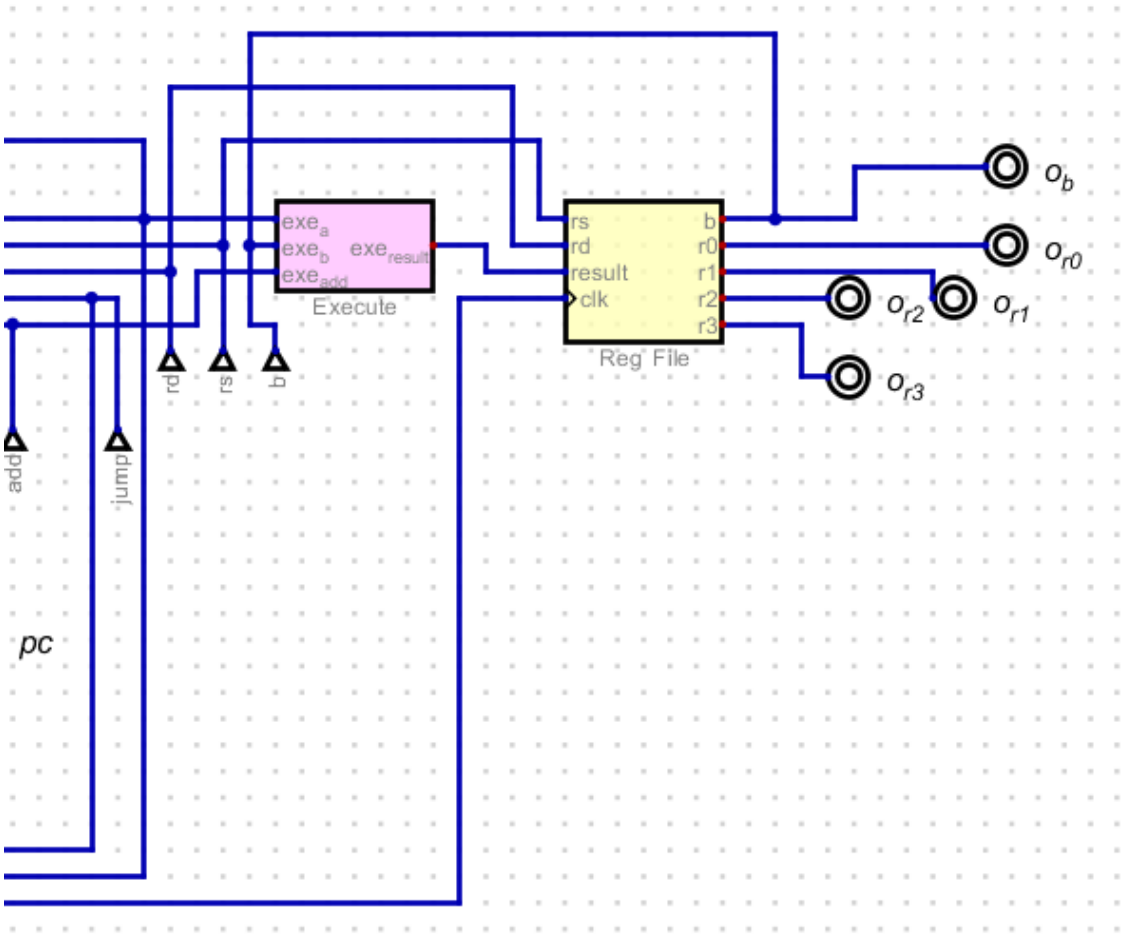

Figure 3.7: Right Hand Side of the Add-Jump Processor

3.5 Subcircuits Explanation

1. Assembler

Its embedded circuit (given below) involves components like 8 cross 3 priority encoder (e.g. 74HC148) whose only first and last input is utilized while others are left unconnected. It takes 3 bits in a 32-bit instruction for specification of the instruction (i.e. opcode). Mode (controlled by DIP switch) is the important signal responsible for overall functionality of the schematic. The bits which are assigned with zero constant is reserved for future application. The AND gate (like 74HC08) is utilized for creating EEPROM clock when we want to store.

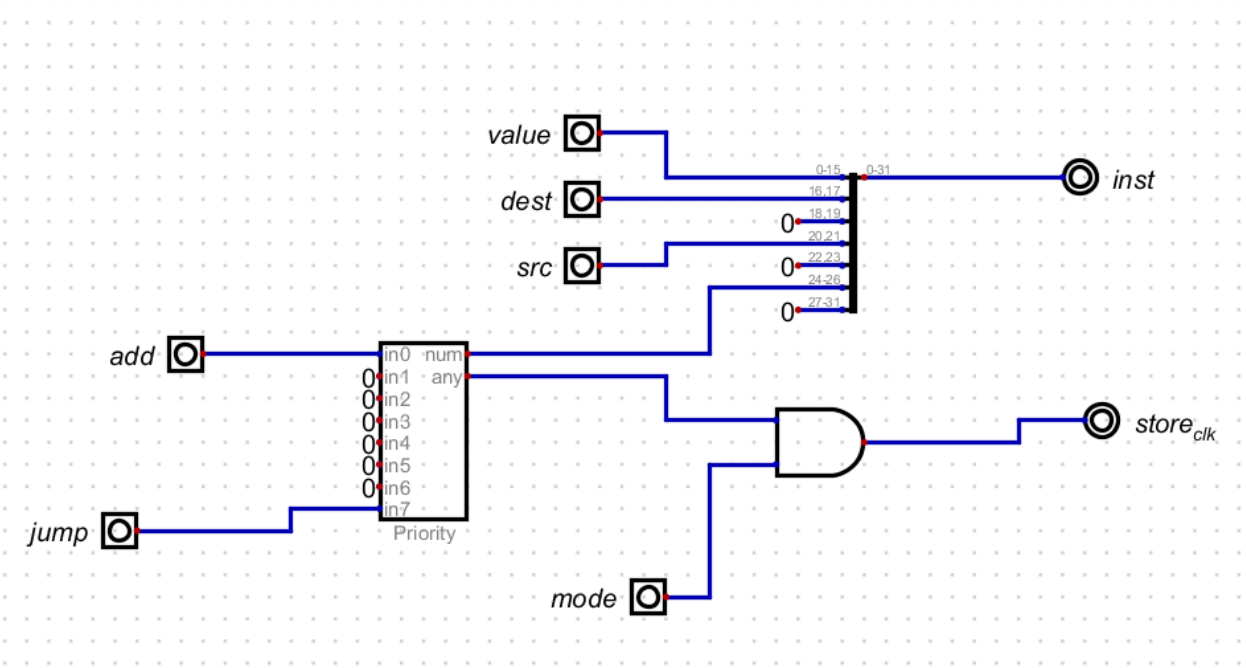

Figure 3.8: Assemble Block Subcircuit having Priority Encoder, And gate and wire manipulator.

2. Decoder

Decoder embedded block circuitry is like of image shown. It consists of 2 cross 1 mux (like 74HC157) and 3 cross 8 decoder (like 74HC138). This circuitry is responsible for knowing the operation to be performed. Bypass signal is utilized when the mode pin is high and we want to see the addition by avoiding the Von Neumann bottleneck. The inst signal is decoded for jump and add decision making. (example.111 bits in the 24^{th} to 26^{th} bit positions). Other signals are defined and is derived from the 32-bit instruction. This portion tells the executer block about what to do.

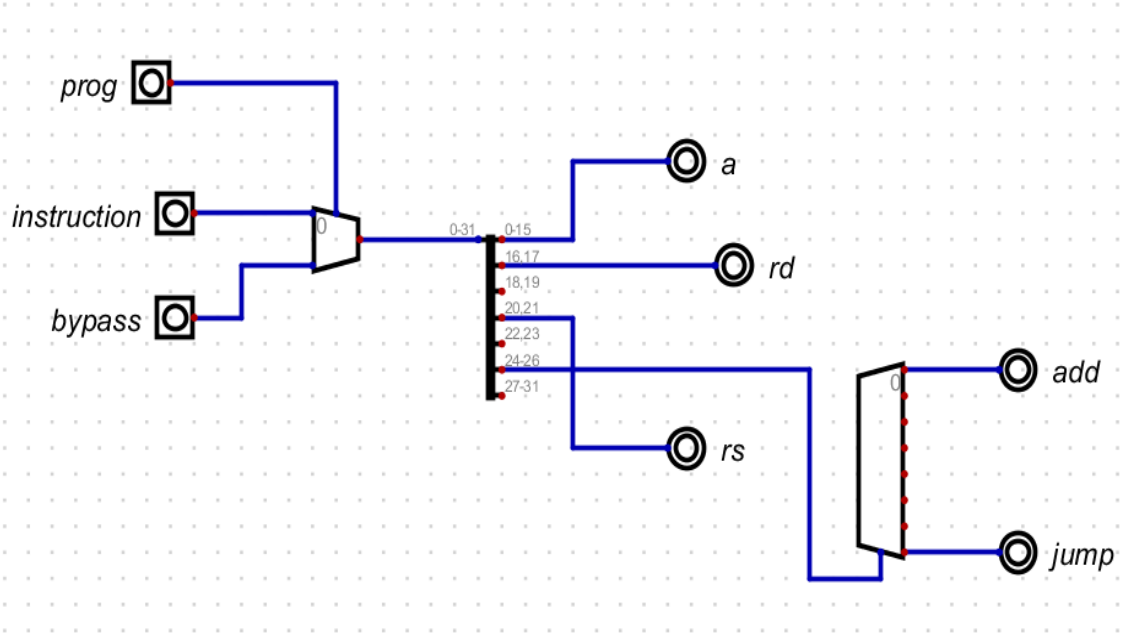

Figure 3.9: Decoder Block Subcircuit

3. Execute

The circuitry for execute embedded block (is shown in the image below) takes stored operand b and operand a from the instruction and performs addition. The multiplexer is used to avoid the addition result at the output if add select line is zero, (might be the case when jump instruction is used). Finally, the output signal is generated as per the instruction.

Note:

If we take a test case when a = b = 65535 then, with carry flag being grounded, we will get FFFF but without 1 in the MSB.

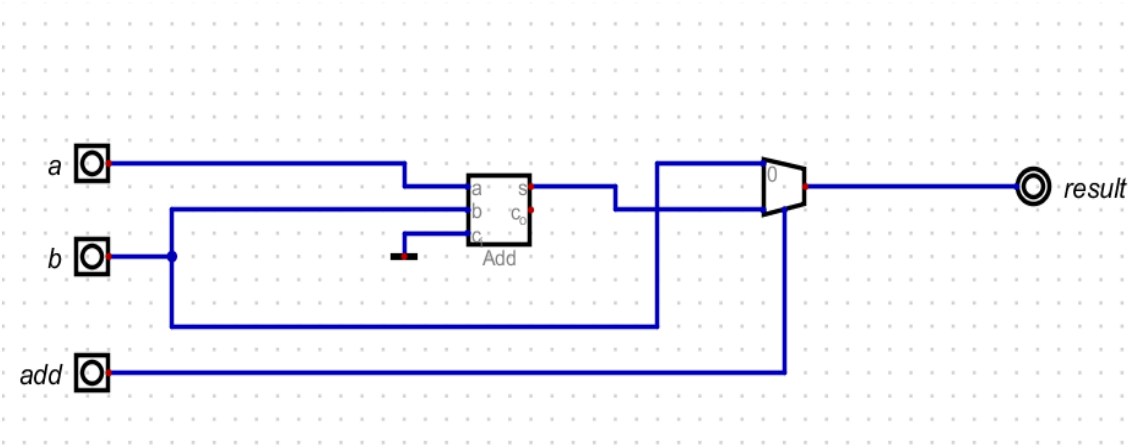

Figure 3.10: Execute Block Subcircuit

4. Reg File

It is an embedded block which have a circuitry given below. The result is stored in the series of flip-flops (like 74HC574), together termed as register file. The specified read destination (rd) signal, which is also used as input to the decoder and multiplexer and is used to for storing the result and to enable the flip-flops. The clock comes from the pushbutton advance, manually. The second 4x1 multiplexer utilized to read the stored data from the flip-flop specified by the read source (rs) signal.

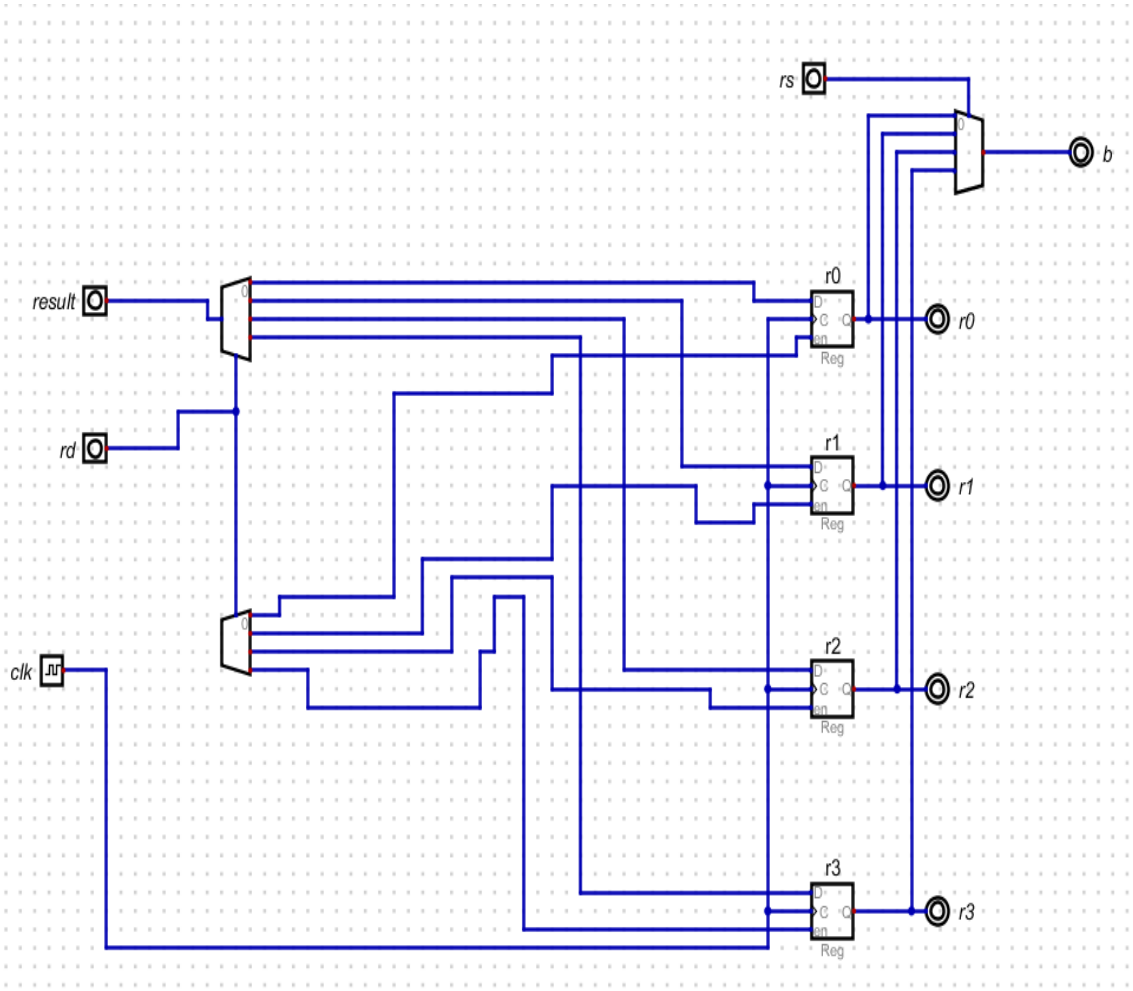

Figure 3.11: Register File Subcircuit.

4. HDL Simulation and Implementation

4.1 RTL Design

Specification and block diagram is the basis for HDL coding. For add-jump processor we have used behavioral mostly and along with data flow modelling, together popularly termed as RTL language. Each block coverage along with its functional coverage is achieved to be 100 percent. It can be verified through the waveforms provided. Concepts like parametrized module, Instantiations, a thorough on digital design topics like priority encoder, and gate, counter, decoder etc. is utilized for designing the HDL code along with debugging it through its testbench respectively. It has eighteen internal wires for connecting 1, 2, 3, 4, 5 and 6^{th} blocks.

The idea for designing is simple if the naming conventions for each block is followed correctly. For the testbench the inputs are made reg and outputs as wire, followed with the Dut instance and test cases.

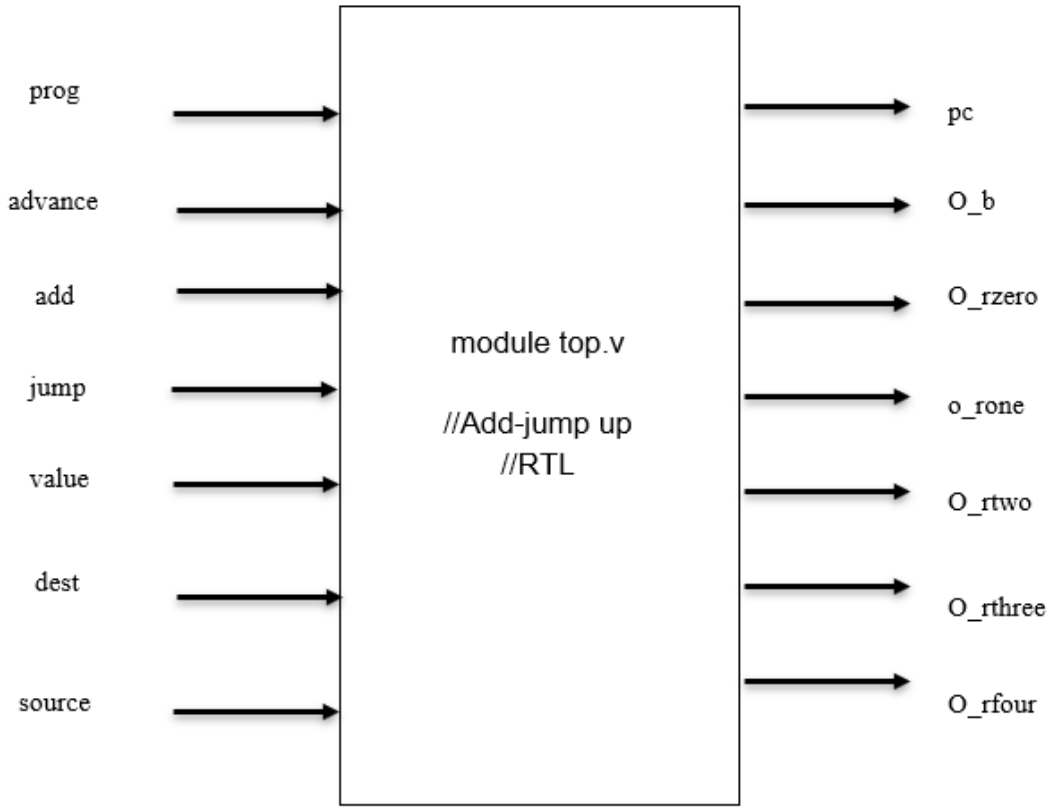

Figure 4.1: Block Diagram for the RTL Design

The test bench is having the following test cases:

Table 3.1: Test Cases for the add-jump processor functional verification

Prog	Value	Src	Dest	Add	Jump
Test Case 1					
1	0	0	0	1	0
-	5	0	0	-	-
-	10	1	1	-	-
-	20	2	2	-	-

-	30	3	3	-	-
-	2	0	0	0	1
Test Case 2					
0	-	-	-	1	0
-	15	0	0	1	0
-	0	2	1	-	-

The top module encapsulates the RTL logic (as seen in Figure 4.2) implementing an Add-Jump processor. External control signals such as prog, add, and jump allow switching between instruction loading and execution phases. The program counter (pc) ensures sequential execution but can be altered using the jump control. Data can be transferred between registers using dest and source, allowing flexible operations. The outputs are exposed to make the processor observable, aiding in testing and functional verification.

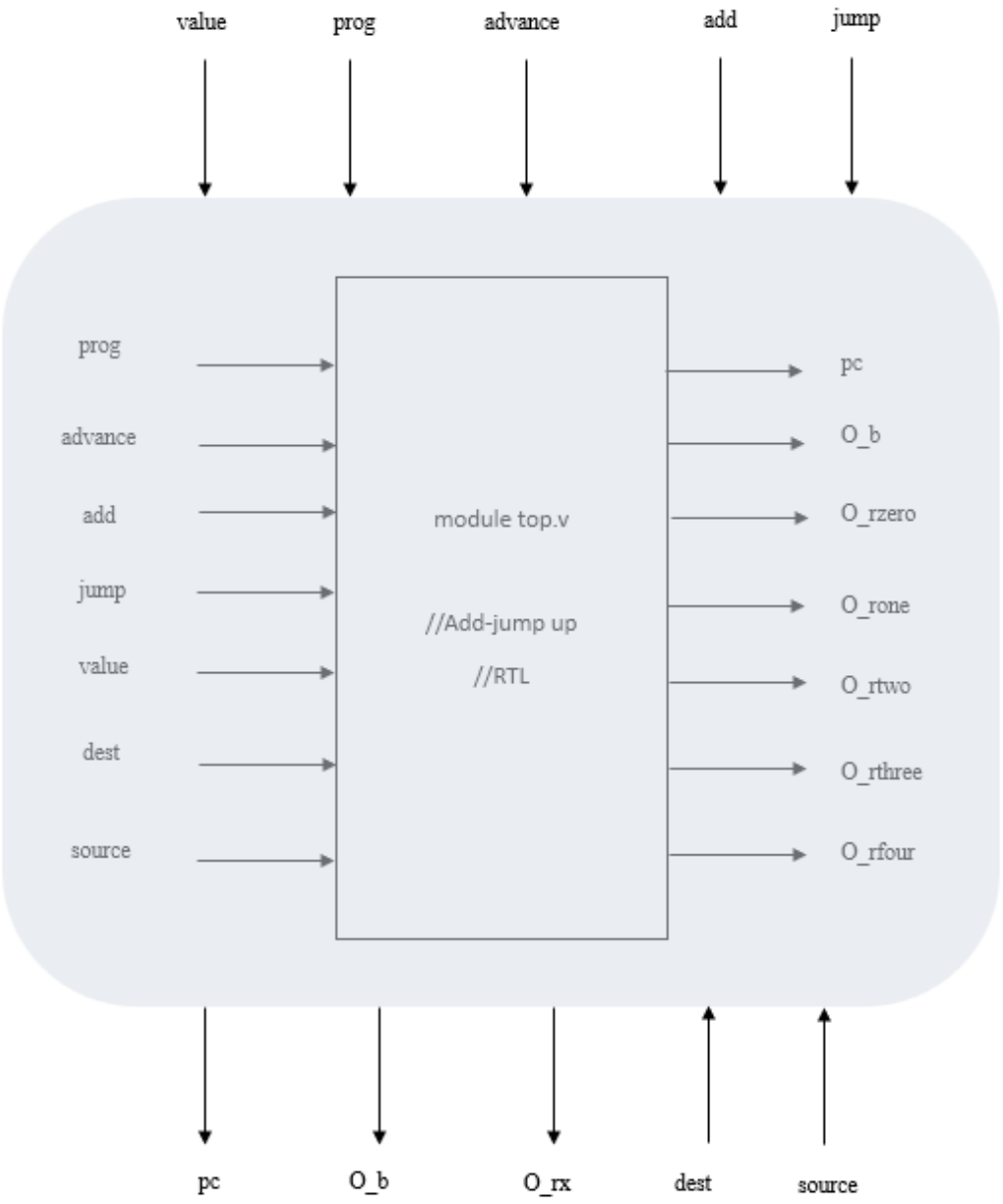

Figure 4.2: Testbench Block interpretation.

Verilog code for other functional units can be accessed by the following GitHub link. https://github.com/KapoorAkshit18/video_display_processor

The repository contains all source files, testbenches, simulation scripts, and documentation.

Table 3.2: Schematic Components

Component	**Input**	**Bits**	**Addr. Bits**	**Number**
Add		16		1
And	2	1		1
CounterPreset				1
Decoder		1	2	1
Decoder		1	3	1
Demultiplexer		16	2	1
EEPROM Dual Port		32	4	1
Multiplexer	2	16	1	1
Multiplexer	2	32	1	1
Multiplexer	4	16	2	1
PriorityEncoder	8	1		1
Register		16		4

4.2 Code Snippet's:

- Below (Figure 4.3) is the snippet of Multiplexer (where blue marked words are the keywords in Verilog HDL) using parameterized module which is seen inside the decoder embedded blocks, as we have discussed it consists of mux for decision making between bypass and stored instruction, based on prog signal.

```
mux_2x1_nbits #(
   .bits(32)
)
mux_2x1_nbits_i0 (
   .sel( d_prog ),
   .xin_0( d_inst ),
   .xin_1( bypass ),
   .xout( iw1 )
);

assign d_a = iw1[15:0];
assign d_rd = iw1[17:16];
assign d_rs = iw1[21:20];
assign iw2 = iw1[26:24];
decoder3 decoder3_i1 (
   .sel( iw2 ),
   .dout_0( d_add ),          // add signal
   .dout_7( d_jump )
);
```

Figure 4.3: Snippet for Decoder

- Below Snippet (Figure 4.4) is of Memory Block which is the EEPROM (like AT28C256) being used in our design. In real hardware we might require 4 of it as mostly the eeprom chips are of 8 bits wide. The below snippet shows the functionality of it. Rom[a] is a part of the memory declaraion which is 32 bits wide and is 16 bit's in the depth [9].

```
always@(posedge c)
begin
    if(str)    //check enable signal and if write enable is ON
        rom[a] <= d_in;
end
// To avoid Bus Contention
always@(posedge c)
begin
 if (str && ld)
        $display("Error: Both read and write active!");
    $stop;
end

//always reading from the ram, irrespective of clock.
assign d = ld ? rom[a] : 32'bz;    // why hi z instead of 0
```

Figure 4.4: Snippet of the EEPROM block

- Below snippet (Figure 4.5) shows how register file and executer blocks are instantiated inside the add_jump_cpu. It is seen that instantiation signal naming is done carefully for clarity and debugging purposes.

```
//=============================================================
// Execute
//=============================================================
    execute u_execute (
        .exe_a      (dec_imm),
        .exe_b      (exe_b),
        .exe_add    (dec_add),
        .exe_result (exe_result)
    );

//=============================================================
// Register File
//=============================================================
    regfile u_regfile (
        .rs     (dec_rs),
        .rd     (dec_rd),
        .result (exe_result),
        .clk    (clk_regfile),
        .o_b    (o_b),
        .o_r0   (o_r0),
        .o_r1   (o_r1),
        .o_r2   (o_r2),
        .o_r3   (o_r3)
    );
```

Figure 4.5: Snippet showing instances inside the top module

4.3 Implementation

Implementation is done by various tools, if we take ModelSim, following TCL commands are used to compile and simulate the .v extension files. Make sure we are in our project directory. We can make .do file (having full script for automation) for automation and .f extension file (having names of the design file) for convenience. Below are the steps we have followed:

```
#tcl
# Create work library
vlib work
# Map as explained earlier
vmap work work
# Create separate directories for Design and Testbenches and vlog
it
vlog design/*.v
vlog testbench/*.v
# Automate to save time for large designs
do simulate.do
```

Figure 4.6: Snippet of the TCL Script for ModelSim

For cadence and gtk wave, we have used `nclaunch -new` command and `gtkwave [name of tb].VCD` command (assuming that it has been created simulation in advance). For Vivado, the process becomes lengthy because we are required to add several new modules. As discussed, we have followed the flow of it and does not created block design because it was not needed, and created output products of the ILA Ip. And created the XDC file as per the FPGA pin mapping, shown below:

XCZU7EV (U1) Pin	Net Name	I/O Standard	Device
GPIO LEDs (Active High)(1)			
D5	GPIO_LED_0	LVCMOS33	DS38.2
D6	GPIO_LED_1	LVCMOS33	DS37.2
A5	GPIO_LED_2	LVCMOS33	DS39.2
B5	GPIO_LED_3	LVCMOS33	DS40.2
Directional Pushbuttons (Active High)			
B4	GPIO_PB_SW0	LVCMOS33	SW14.3
C4	GPIO_PB_SW1	LVCMOS33	SW15.3
B3	GPIO_PB_SW2	LVCMOS33	SW17.3
C3	GPIO_PB_SW3	LVCMOS33	SW18.3
CPU Reset Pushbutton (Active High)			
M11	CPU_RESET	LVCMOS33	SW20.3
GPIO DIP SW (Active High)			
E4	GPIO_DIP_SW0	LVCMOS33	SW13.8
D4	GPIO_DIP_SW1	LVCMOS33	SW13.7
F5	GPIO_DIP_SW2	LVCMOS33	SW13.6
F4	GPIO_DIP_SW3	LVCMOS33	SW13.5

Figure 4.7: Package Pins for Switches. Source adapted from [2]

It was a challenge for us to map all of the input and output sources to the FPGA; therefore, we have used DIP for 16-bit value connected to the pins as given below from the sourced from AMD. We have separated destination bits to the on-board push button and dip switch. Rest mapping can be understood by the results of the Vivado.

XCZU7EV (U1) Pin	Net Name	I/O Standard	PMOD Pin
G8	PMOD0_0	LVCMOS33	J55.1
H8	PMOD0_1	LVCMOS33	J55.3
G7	PMOD0_2	LVCMOS33	J55.5
H7	PMOD0_3	LVCMOS33	J55.7
G6	PMOD0_4	LVCMOS33	J55.2
H6	PMOD0_5	LVCMOS33	J55.4
J6	PMOD0_6	LVCMOS33	J55.6
J7	PMOD0_7	LVCMOS33	J55.8
J9	PMOD1_0	LVCMOS33	J87.1
K9	PMOD1_1	LVCMOS33	J87.3
K8	PMOD1_2	LVCMOS33	J87.5
L8	PMOD1_3	LVCMOS33	J87.7
L10	PMOD1_4	LVCMOS33	J87.2
M10	PMOD1_5	LVCMOS33	J87.4
M8	PMOD1_6	LVCMOS33	J87.6
M9	PMOD1_7	LVCMOS33	J77.8

Figure 4.8: GPIO PMOD Pins for external peripherals. Source adapted from [2]

4.4 Best Hardware Coding Practices:

1. Coding Style and Organization

☐ Use consistent naming conventions for signals, ports, and modules.

☐ Maintain one module/entity per file, and match file names with module/entity names.

☐ Separate design files (RTL) from testbench files.

☐ Use parameters (Verilog/SystemVerilog) or generics (VHDL) instead of hard-coding constants.

2. Clock and Reset

☐ Use synchronous design with a single clock domain where possible.

☐ Prefer asynchronous reset with synchronous release.

☐ Use active-low resets (rst_n) as they are common in industry IP cores.

☐ Avoid gated clocks; instead, use clock enables.

3. Sequential and Combinational Logic

☐ Use non-blocking assignments (<=) for sequential logic in Verilog.

☐ Use blocking assignments (=) for combinational logic in Verilog.

☐ In VHDL, always include proper sensitivity lists (or use process(all) in VHDL-2008).

☐ Ensure all branches in if and case statements are covered to avoid unintended latches.

Reusability and Parameterization

☐ Write modular, reusable code.

☐ Use parameters/generics to make modules configurable.

☐ Avoid magic numbers by defining constants in packages or headers.

4. Testbenches and Verification

☐ Keep testbenches separate from design code.

☐ Use self-checking testbenches with assertions.

☐ Generate clock and reset signals in the testbench, not inside the DUT.

☐ Use tasks/functions/procedures for stimulus generation.

5. Simulation and Debugging

☐ Use waveform viewers to observe signal activity during simulation.

☐ Apply $display/$monitor (Verilog) or report (VHDL) for logging.

☐ Insert assertions to catch errors early in simulation.

☐ Verify modules individually before system-level simulation.

6. Synthesis-Friendly Coding

☐ Avoid non-synthesizable constructs (e.g., delays, file I/O in RTL).

☐ Write code that infers hardware efficiently (e.g., BRAMs, DSPs).

☐ Pipeline long combinational paths to meet timing.

☐ Ensure proper reset logic for sequential elements.

7. Documentation and Maintainability

☐ Write meaningful comments explaining the purpose of code.

☐ Document design assumptions and interface specifications.

☐ Organize projects cleanly (RTL, simulation, constraints, scripts).

☐ Use version control (e.g., Git) for tracking changes.

8. General Guidelines

☐ Use linting tools to catch errors and enforce coding standards.

☐ Perform static timing analysis after synthesis/implementation.

☐ Follow coding guidelines for consistency.

☐ Write clear, readable, and maintainable code.

5. EXPERIMENTATION

Experiment 5.1 - To check add instruction

a. Case when programming signal is high

We have taken 3 inputs like 1, 2, and 3 which are added with the contents of different Registers. From the given table below. we can see that after every positive edge the result is obtained and written back to the destination register.

Table 5.1: Input-Output capture one from the simulation using Digital Tool

value	src	dest	o_b	o_r0	o_r1	o_r2	o_r3	prog	pc
0x0001	0x0	0	1	0x0001	0x0000	0x0000	0x0000	1	0x1
0x0002	0x0	1	1	0x0001	0x0003	0x0000	0x0000	1	0x2
0x0003	0x0	2	1	0x0001	0x0003	0x0004	0x0000	1	0x3
0x0003	0x0	3	1	0x0001	0x0003	0x0004	0x0004	1	0x4

If we perform the calculations, 1+0= 1, 2+1=3,3+1=4,3+1=4 since o_b is 1 here which is equivalent to the second operand for addition for changing values, (first operand). The same can be visualized by the assembly code and its encoding given below:

```
000000     000      00     00    00     00
00000000000000
[31:27] [26:24] [23:22] [21:20] [19:18] [17:16] [15:0]
 pad   opcode    pad     rs      pad      rd    value
```

Example:

```
address | data (base 2)
0       | 00000000 00000000 00000000 00000001; add r0, r0,
1       | 00000000 00000001 00000000 00000010; add r1, r0,
2       | 00000000 00000010 00000000 00000011; add r2, r0,
3       | 00000000 00000011 00000000 00000011; add r3, r0,
```

Figure 5.1: Add instruction Encoding using customasm

Note:

The counting of the binary digits is from Most Significant Bit to the Least Significand Bit.

b. Case when programming signal is low

Table 5.2: Input-Output capture two from the simulation using Digital Tool.

value	src	o_b	o_r0	o_r1	o_r2	o_r3	prog	pc
0x0001	0x0	1	0x0001	0x0000	0x0000	0x0000	0	0x1
0x0002	0x0	1	0x0001	0x0003	0x0000	0x0000	0	0x2
0x0003	0x0	1	0x0001	0x0003	0x0004	0x0000	0	0x3
0x0004	0x0	1	0x0001	0x0003	0x0004	0x0004	0	0x4

Experiment 5.2 - To check Jump Instruction

```
addr | data (base 2)

 0 | 00000000 00000000 00000000 00000001 ; add r0, r0, 1
 4 | 00000000 00000000 00000000 00000010 ; add r0, r0, 2
 8 | 00000000 00000000 00000000 00000011 ; add r0, r0, 3
 c | 00000000 00000000 00000000 00000011 ; add r0, r0, 3
10 | 00000000 00000111 00000000 00000100 ; jump .loop
```

Figure 5.2: Jump instruction Encoding using customasm

The assembly program snippet of the same is given below. The code is generated using an opensource customasm tool, where we can define our custom rules for the assembly code. The jump instruction is executed at the 4th clock cycle in the program counter after which it goes to the value or label, and continues the executing our program.

```
main:
    add r0, r0, 1
.loop:
    add r0, r0, 2
    add r0, r0, 3
    add r0, r0, 3
    jump .loop
```

Figure 5.3: Equivalent Jump Program

6. RESULT AND DISCUSSION

ModelSim Simulation Results interpretation is show below for the major six block and further functional logic units' waveforms can be referred from the appendix section.

1. If we see the below waveform, the waveform shows the how when programming mode is switched on, it affects the output. Our expected waveform was when prog is high the store_clk signal will also become high which is the case we have got from the actual waveform. Furthermore, the input values are encoded as expected for the add and jump signals respectively.

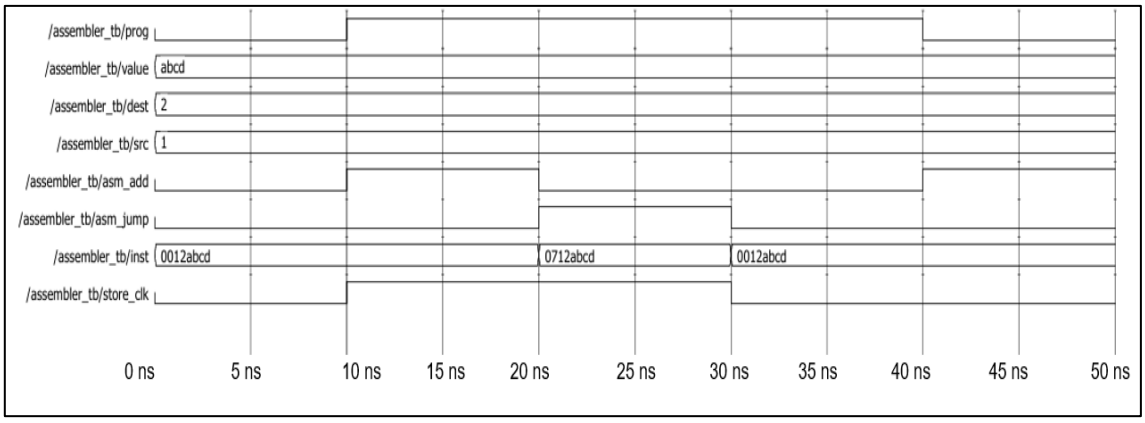

Figure 6.1 Simulation Waveform for the Assembler Block

2. If we see the below waveform of the write operation of the memory block inside our processor (see Figure 6.2), it is expected that the logic of the EEPROM block as per in the digital tool is followed properly and the actual results confirms it or not cannot be cleared from the given waveform as the output is don't care or high impedance value. So, we need one more waveform which can read these values from the memory location it is being written to. (see Figure 6.3). We can see that in the output the values being written earlier was stored and the reading operation is occurring step by step.

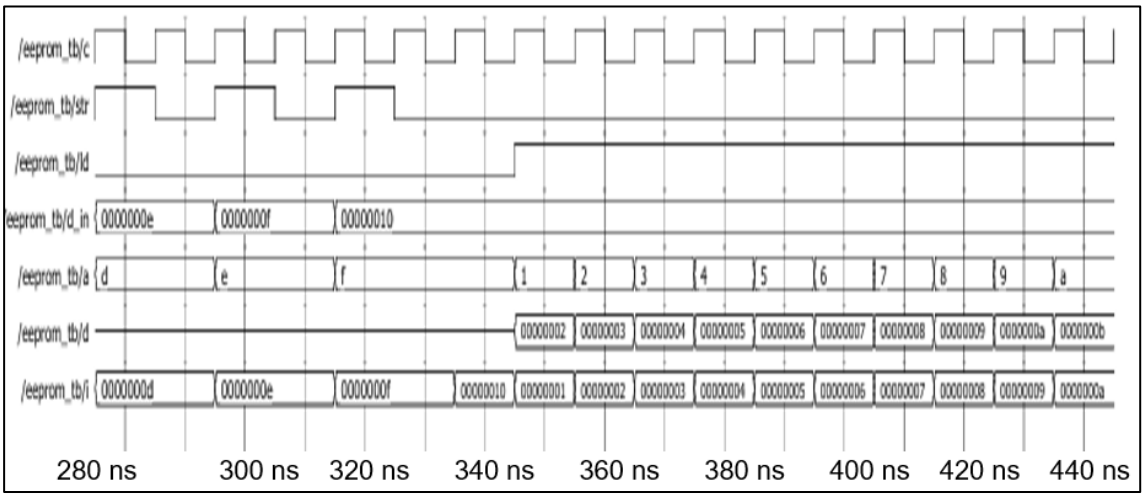

Figure 6.2: Waveform showing write logic of EEPROM memory inside the processor.

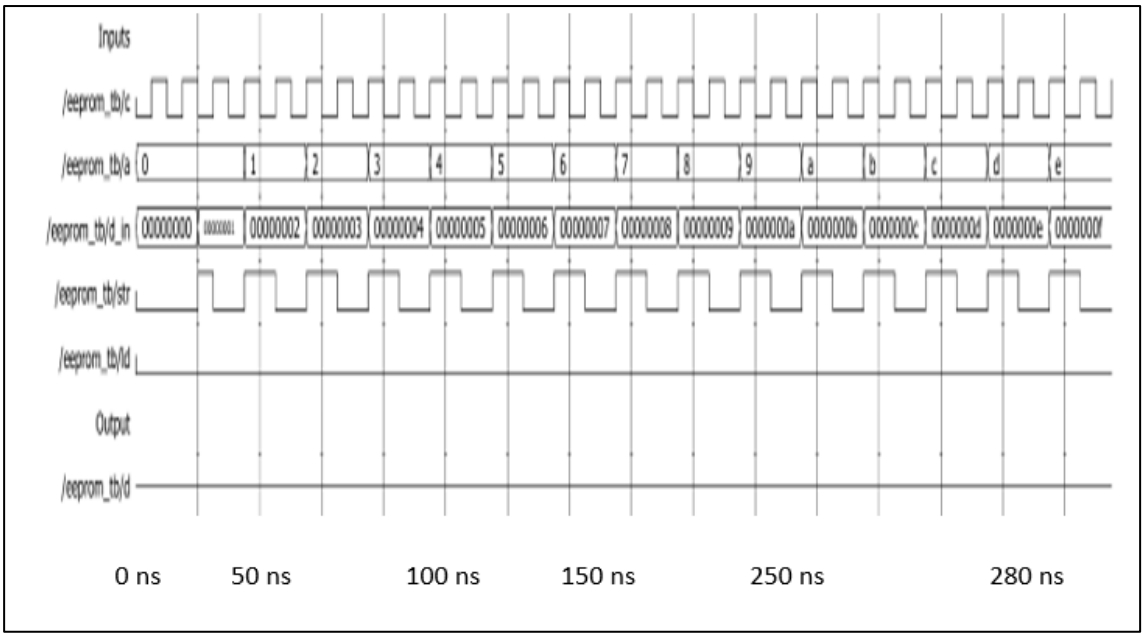

Figure 6.3: Waveform showing Read logic of EEPROM memory block inside the processor.

3. If we see the waveform of our counter block (Figure 6.4), we expect that it counts in both directions as per specification and it can also count if we wanted it to be from a specific number. For our processor we are interested in its up-counting preset functionality of it. It is seen that the actual waveform matches our expectation.

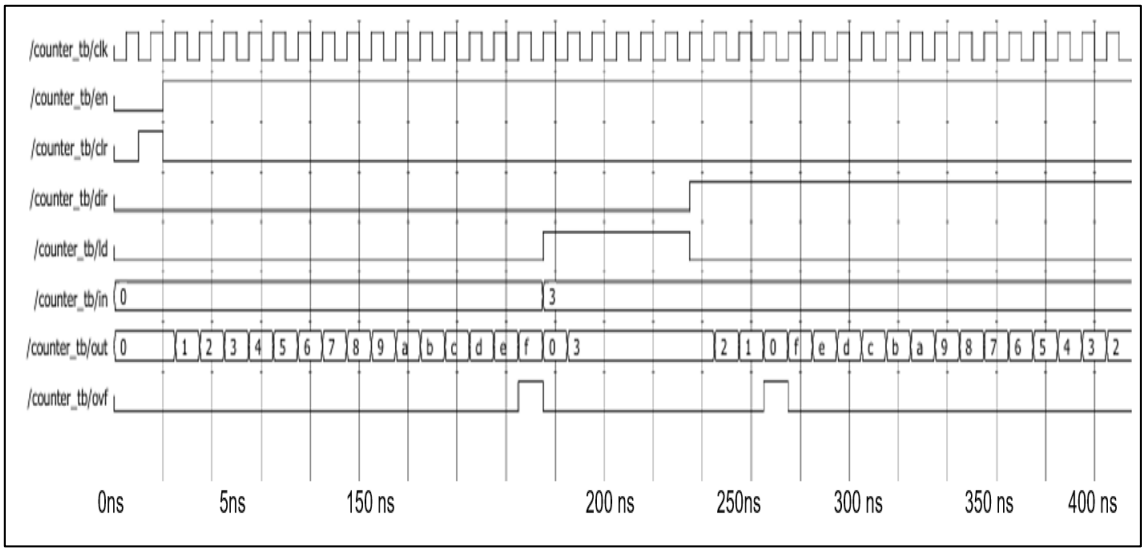

Figure 6.4: Simulation Waveform for the Counter with Preset (4 Bits)

4. If we see the waveform of the decoder block, we expect that the 32-bit instruction on its input side divide further where the division of the bits depends upon the type of the signal. The instruction signal is bypass and from the memory block, out of two we need to select one based on the prog mode, and the actual output showcases this verified in the waveform given below (see Figure 6.5).

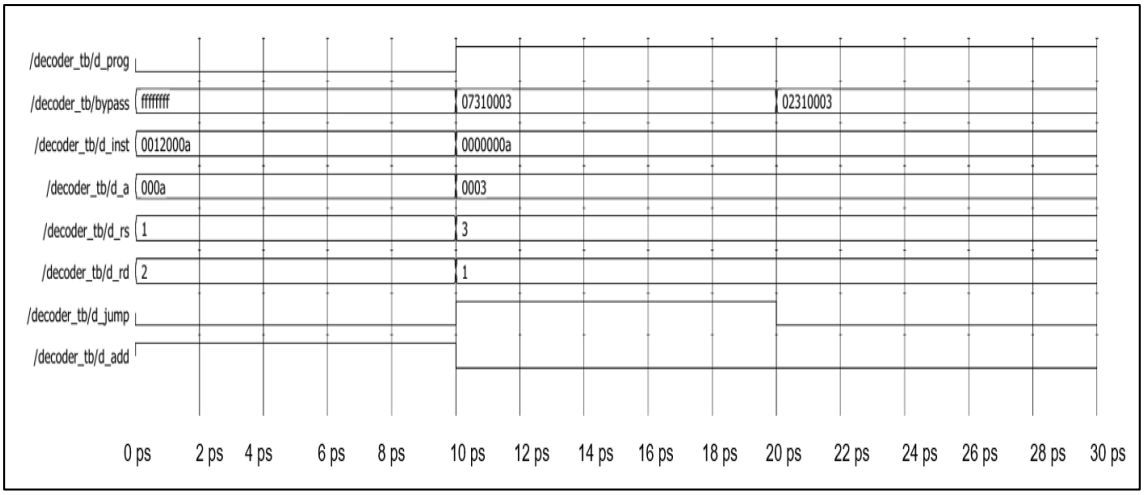

Figure 6.5: Simulation Waveform for the Decoder Block

5. If we see the waveform given below (Figure 6.6), we expect when we give value input along with add signal, it gets added with the contents of the register specified and the result is stored in the specified register. The actual waveform shows the similar behavior as 0x000a + 0x0005 = 0x000f. This confirms the reliability of the Execute Block of the add-jump processor.

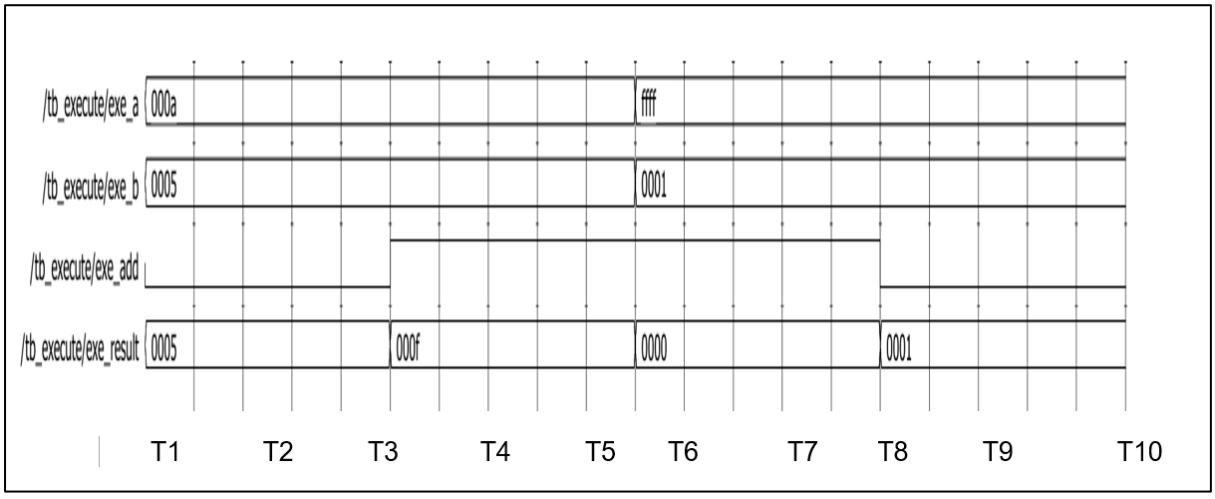

Figure 6.6: Simulation Waveform for the Executer Block

6. If we see the below waveform, we expect after next clock cycle new value is seen in the specified register decided by the rd signal.

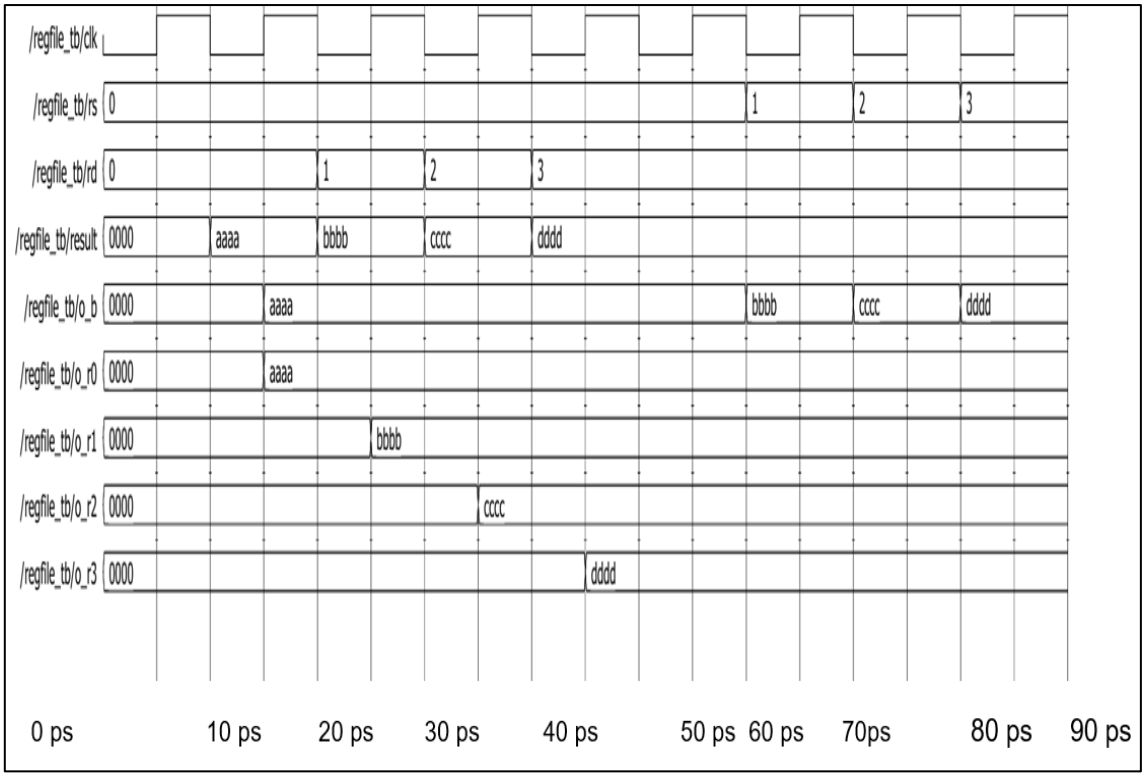

Figure 6.7: Simulation Waveform for the Register File Block.

7. Finally, we expect addition when prog is high initially and when the prog signal is made logic low (i.e. (by lowering the dip switch lever) we want it to operate from the memory block's load operation. We expect the jump instruction to be executed when the prog signal is logic low, the pc starts from the address as specified by the stored jump instruction's value field. From Figure 6.9 the actual waveform is like what we expect and from Figure 6.8 the jump instruction's working is confirmed.

Table 3.1: Test Cases for the Add-Jump Processor functional verification

Prog	Value	Src	Dest	Add	Jump
Test Case 1					
1	0	0	0	1	0
-	5	0	0	-	-
-	10	1	1	-	-
-	20	2	2	-	-
-	30	3	3	-	-

-	2	0	0	0	1
Test Case 2					
0	-	-	-	1	0
-	15	0	0	1	0
-	0	2	1	-	-

Observation:

- **Test Case 1**:

  - Values stored correctly at addresses 0–3;
  - jump executed as expected to address 2.
  - Actual results matched expected results.

- **Test Case 2**:

  - Add and store operations performed correctly
  - no jump occurred.
  - Actual results matched expected results.

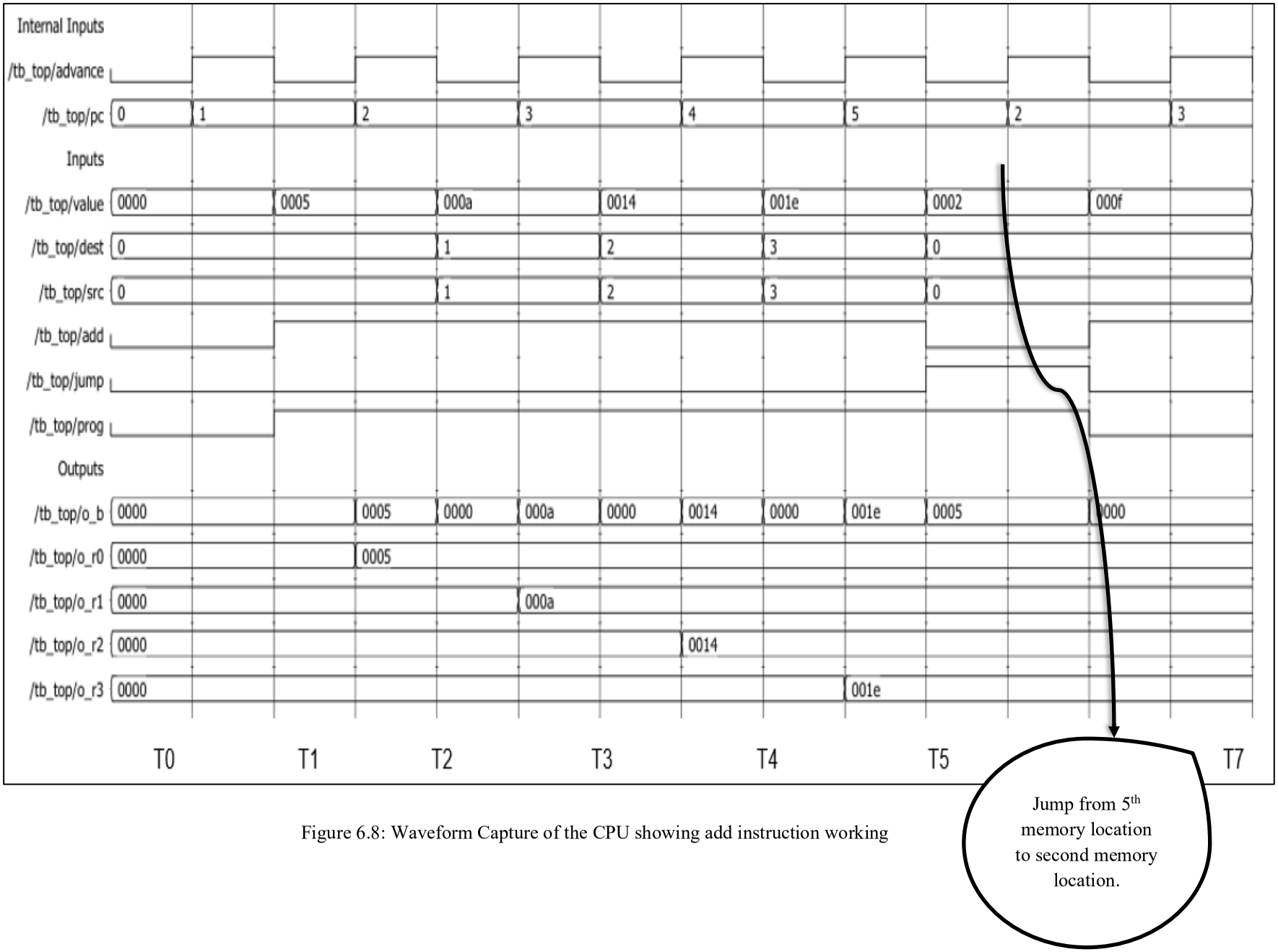

Figure 6.8: Waveform Capture of the CPU showing add instruction working

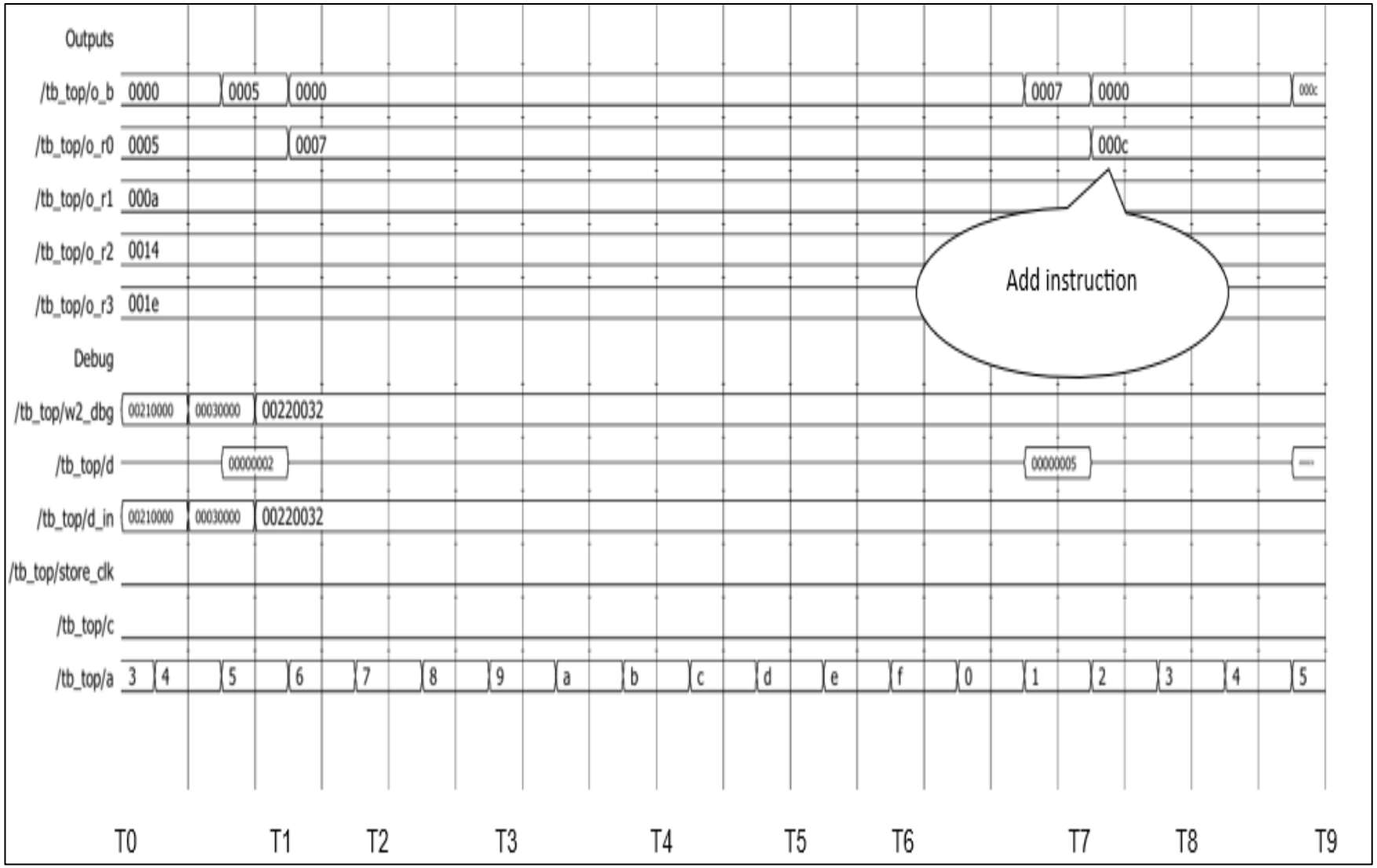

Figure 6.9: Waveform Capture of the CPU showing add instruction working

6.1 Vivado's Results:

The RTL analysis to Bit Stream Generation results [4] is discussed below:

1) **RTL Analysis:** It is seen that the behavioral simulation results for the CPU are identical with ModelSim and hence the functionality check is passed here by Vivado. The results for the FPGA wrapper (Figure 6.10) are expected to be same as of add-jump CPU results, it is seen there are several issues like opcodes are not getting activated, to fix this we have tried making modifications in the priority encoder HDL code.

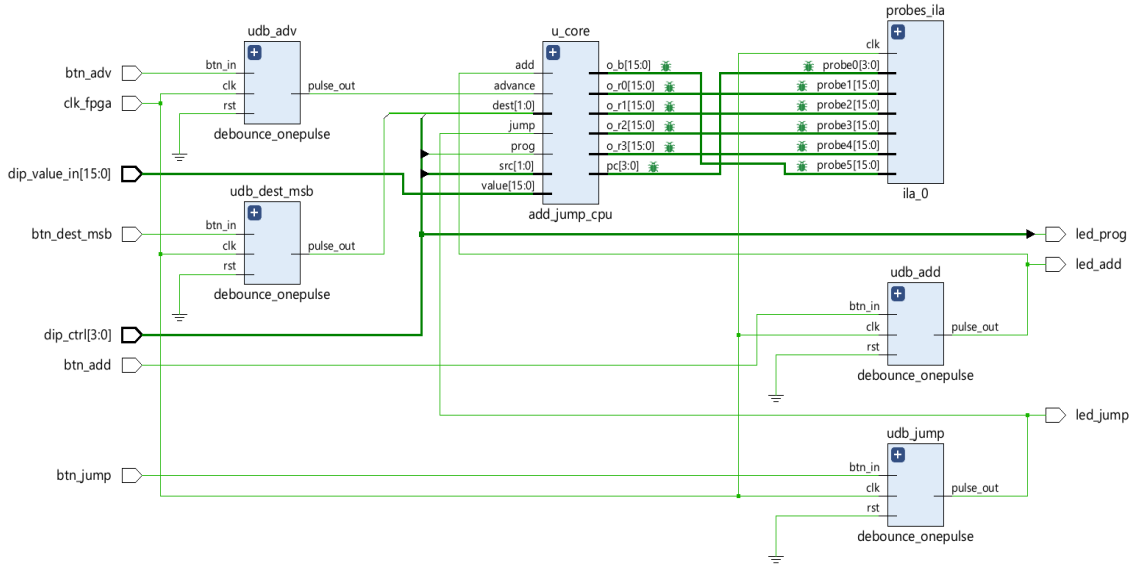

Figure 6.10: RTL Elaborated Schematic in Vivado

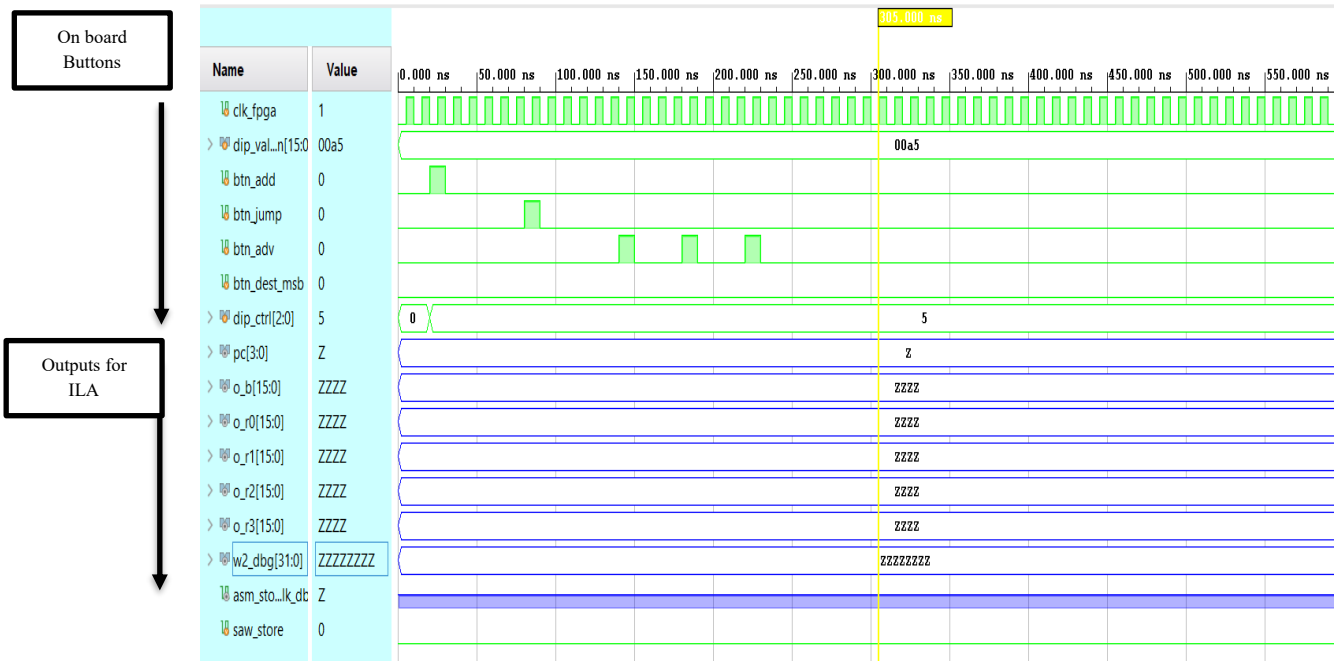

Figure 6.11: Waveform from the XSim simulator, showing the behaviors of the input push buttons and dip switches for driving the internal core.

2.) **Synthesis:** From the timing report, the maximum frequency is calculated and it comes out to be 57.14 Mhz. This frequency is theoretical because it may change as per practical implementation of our design. Our design needs to run below this frequency. If we see Figure 6.12: IO Package, the horizontal rails represent column part of our pin and vertical represents row part, e.g. B4 will be in the rightmost part of our package. It is used to locate our pins easily.

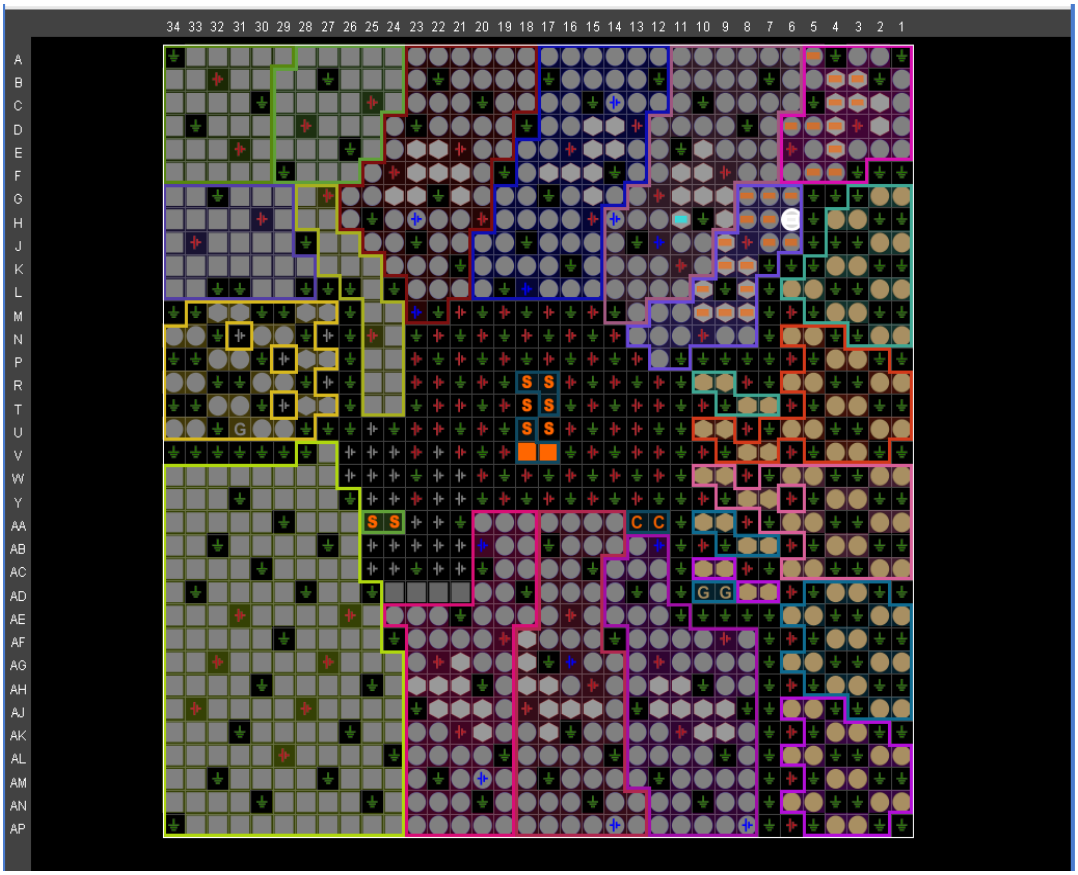

Figure 6.12: Implementation Result – Package Manager

From the utilization report following conclusions are made:

- The resource utilization results demonstrate that the design comfortably fits into the ZCU104 (XCZU7EV) device.
- The logic consumption is very low: only 0.75% of LUTs and 0.54% of flip-flops are used.
- Memory usage is negligible, with only 2.5 BRAM blocks utilized (0.8% of total available).
- IO utilization is higher in percentage (7.78%) since the design maps 28 ports to the FPGA board's DIP switches, buttons, and LEDs.
- Overall, the design leaves abundant unused resources, providing significant headroom for future extensions (e.g., adding more functionality, debugging, or interfacing with peripherals). Timing closure should not be difficult due to the low utilization.

Table 4.1.3 Utilization Report- Vivado Simulator

Resource	Utilization	Available	Utilization %
LUT	1738	230400	0.75
FF	2489	260800	0.54
BRAM	2.50	312	0.80
IO	28	360	7.78

3.) **Implementation:** From the below schematics we can see the mapping of the HDL code into the block representing LUTs and FDRE. The clocking definitions in the XDC constraint file requires improvement, overall, the design is ready to prototype.

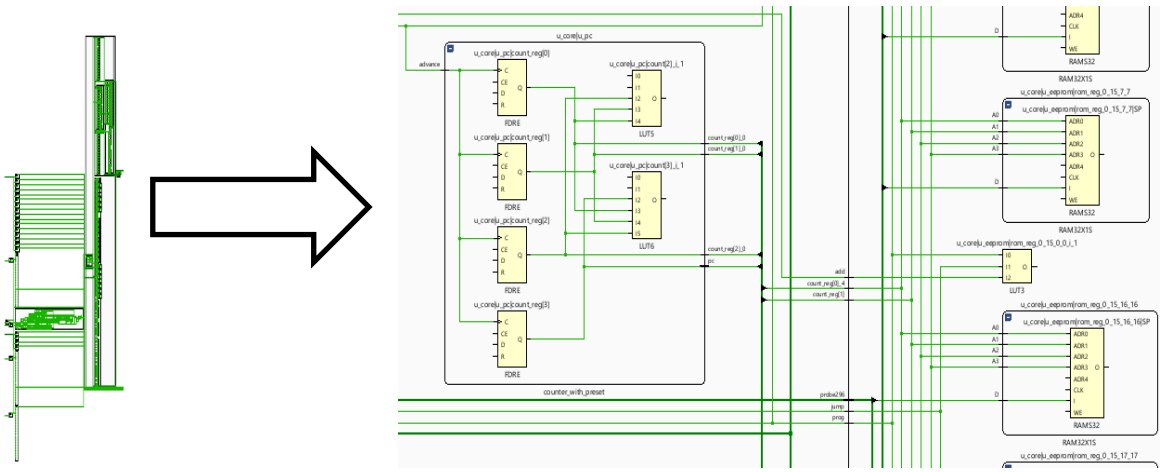

Figure 6.13: Schematic Result from the Vivado Simulator Implementation showing zoomed-out and zoomed-in snapshot of the mapped logic cells.

6.2 NCSim Results:

The functional simulation of the processor was initiated in the Cadence tool, and preliminary waveforms were obtained (Figure 6.14). The signals for the control inputs (prog, advance, jump, dest) and the corresponding outputs (pc, o_r0, o_r3, o_b) can be observed. However, the simulation results remain incomplete. At this stage, many of the output signals remain undefined (X or Z) because the full set of testbench stimuli and verification scenarios were not developed within the available time. Consequently, the simulation does not yet demonstrate the complete functional behavior of the processor. Completing this work would require further refinement of the testbench, integration of additional stimulus patterns, and debugging of the internal Datapath. These tasks were outside the current scope and timeline of this phase of the project. In summary, while initial simulation activity confirmed partial signal activity and clock-driven transitions, the full verification of the design remains pending and is identified as future work.

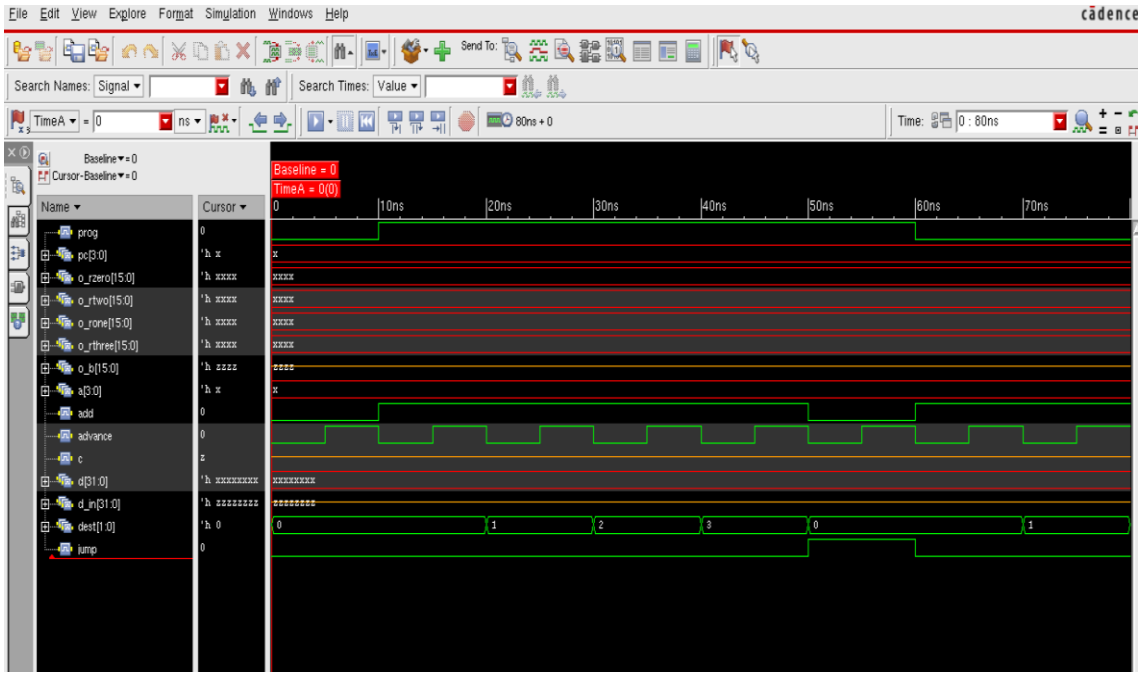

Figure 6.14: Waveform captured for the verification of the Add-Jump CPU using NCSim simulator [3]

Other Results- **Gtkwave**: If we see the results, there are so many waveforms which can be complex if we are not thorough with the project. This platform is lightweight and is used for

inspecting the waveforms on Linux platform. It has a simple drag and drop interface by which you can see all of the signals e.g. From top as well as from its submodules at once, which is helpful for debugging as it provides useful information about how the data flows. In it also the results are like Vivado and ModelSim for the CPU core, and the FPGA top.

6.3 Digital Tool Results:

The add-jump processor is simulated and verified (as seen in Figure 6.15). The waveform generated is given below from the digital tool. It is expected that the overflow condition is satisfied when we choose register zero as both source and destination. It is seen the addition do occurs but it might seem it is subtraction as the carry is ignored, it is the addition. So, our expected waveform matches with the actual waveform, hence our project is working as intended to be functionally.

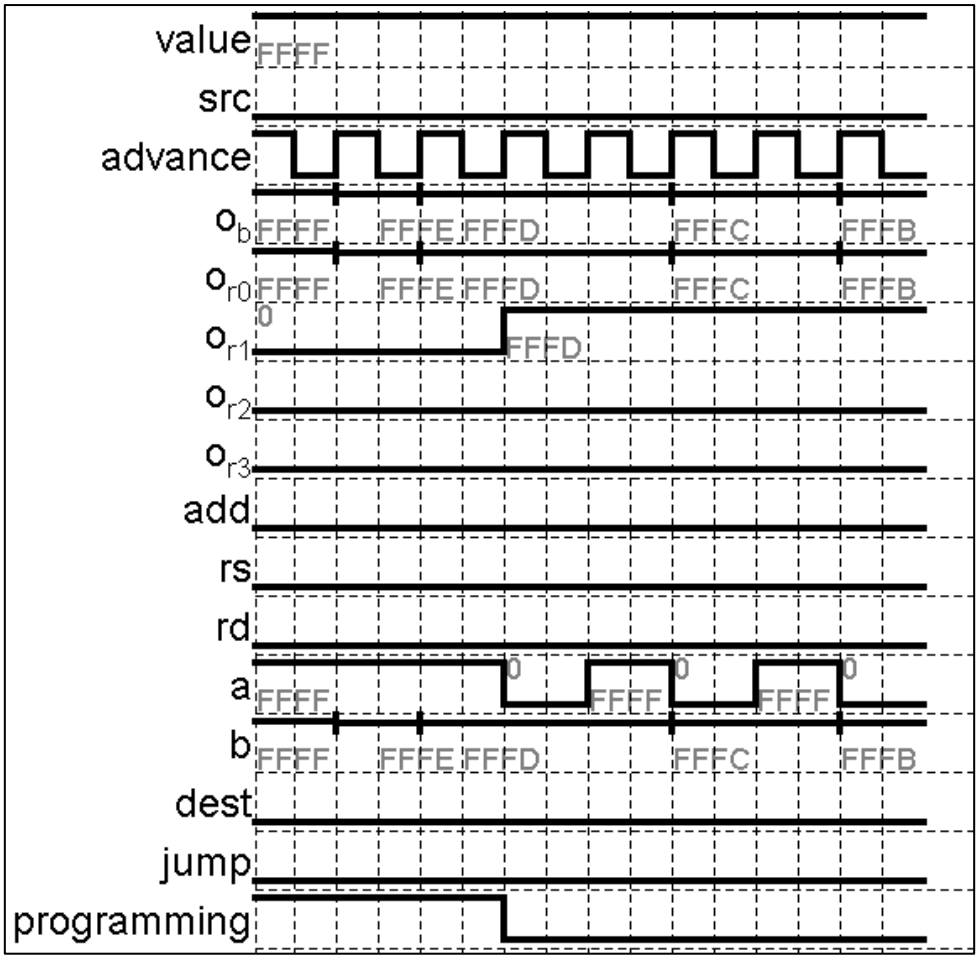

Figure 6.15: Waveform captured using Digital tool

7. CONCLUSION

While simulation results achieved 100% functionality, FPGA implementation highlighted areas for improvement, offering valuable insights for future enhancement.

The project aimed to design and test the 16-bit processor is done. and it is a great start in academic domain to learn the fundamentals of CPU design. The test cases and results altogether confirm its functionality. The project is implemented and it have provided valuable insights at hardware level by using DIP switches, pushbuttons, and PMOD interfaces. It is a great project in embedded systems involving all basics required to grasp higher concepts like pipelined computer processors. Future work should focus on correcting and redoing the FPGA implementation for checking its efficiency.

8. FUTURE SCOPE

- **Addition of more instructions**: Addition of more instructions like multiplication, logical instructions, and other memory instructions.
- **FPGA Implementation**: FPGA implementation will provide real time environment related information.
- **New Architecture:** Pipelining, converting it into Harvard Architecture and microcode capable control unit just like Intel's 8085.
- **Assembler and Compiler Support:** Develop software tools to convert high-level programs into machine code for the processor, and progressing towards the Advance level System Design.

References

[1] R. J. Sanche, "rj32," Rj45, [Online]. Available: https://github.com/rj45/rj32.

[2] ModelSim, "ModelSim User Manual," [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/swdocs/modelsim/modelsim_user_2024_2.pdf.

[3] Cadence, "Cadence Design Systems: NCLaunch User Guide," [Online]. Available: https://picture.iczhiku.com/resource/eetop/shiGDPQuGkDUjcmB.pdf.

[4] AMD, "Vivado Design Flows Overview," [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ug892-vivado-design-flows-overview.pdf.

[5] Lorenzi, "customasm," hlorenzi, [Online]. Available: https://github.com/hlorenzi/customasm.

[6] hneemann, "v0.31," [Online]. Available: http://github.com/hneemann/Digital/releases/tag/v0.31.

[7] P. A. P. Shah, "Digital Electronics," IIT Jammu, [Online]. Available: https://www.youtube.com/watch?v=Rj6Ti2nXfIE&t=739s.

[8] S. Vidhyadharan, "ROM, PROM, EPOM, EEPROM and Programming the ROM," Birla Institute of Technology & Science, Pilani, [Online]. Available: https://www.youtube.com/watch?v=YElaOIqCLp8.

[9] P. I. S. Gupta, "Modelling Memory," NPTEL, [Online]. Available: https://www.youtube.com/watch?v=yN_1LAKmR0g&t=1411s.

[10] AMD, "AMD Technical Information Portal," [Online]. Available: https://docs.amd.com/v/u/en-US/ug1267-zcu104-eval-bd.

[11] ModelSim, "ModelSim Tutorial," [Online]. Available: https://profile.iiita.ac.in/bibhas.ghoshal/COA_2020/Lab/modelsim_tut.pdf.

Appendix I- Supporting Waveforms

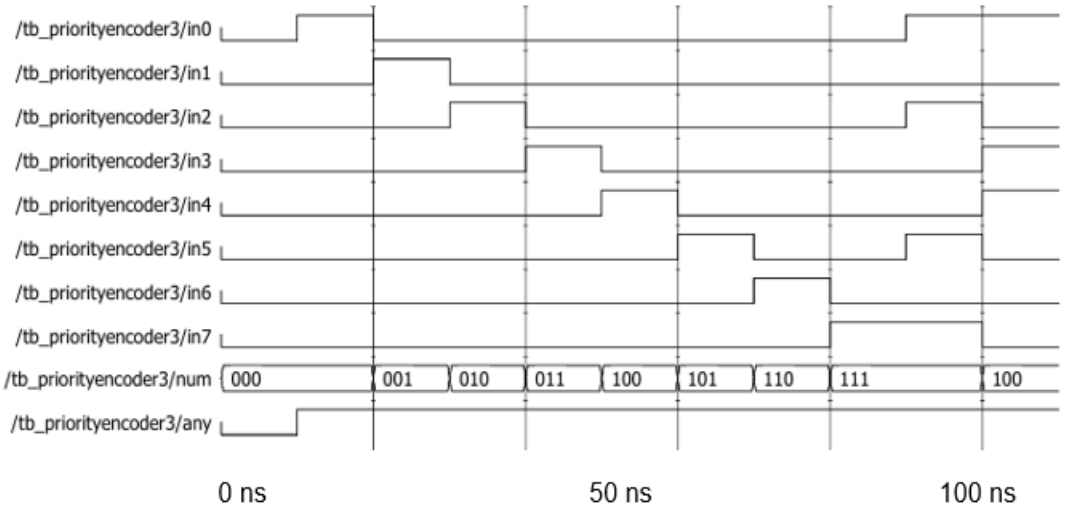

Figure I.1: Priority Encoder Simulation Waveform.

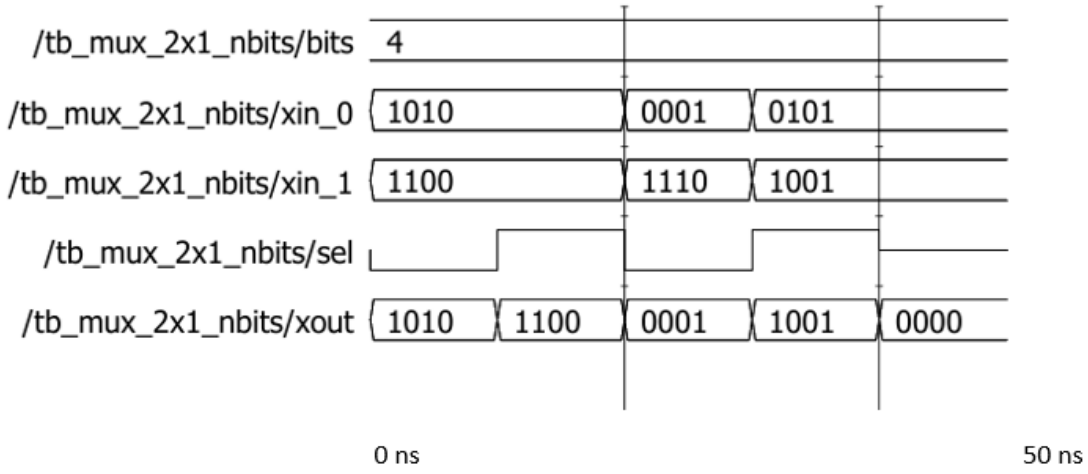

Figure I.2: 2x1 Multiplexer Simulation Waveform (4 bits)

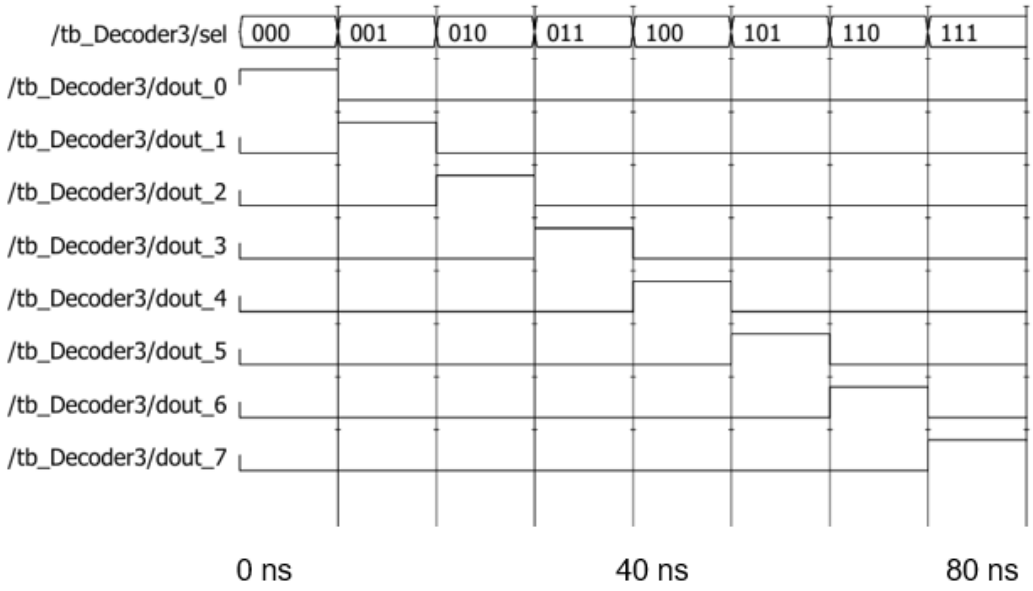

Figure I.3: 3x8 Decoder Simulation Waveform

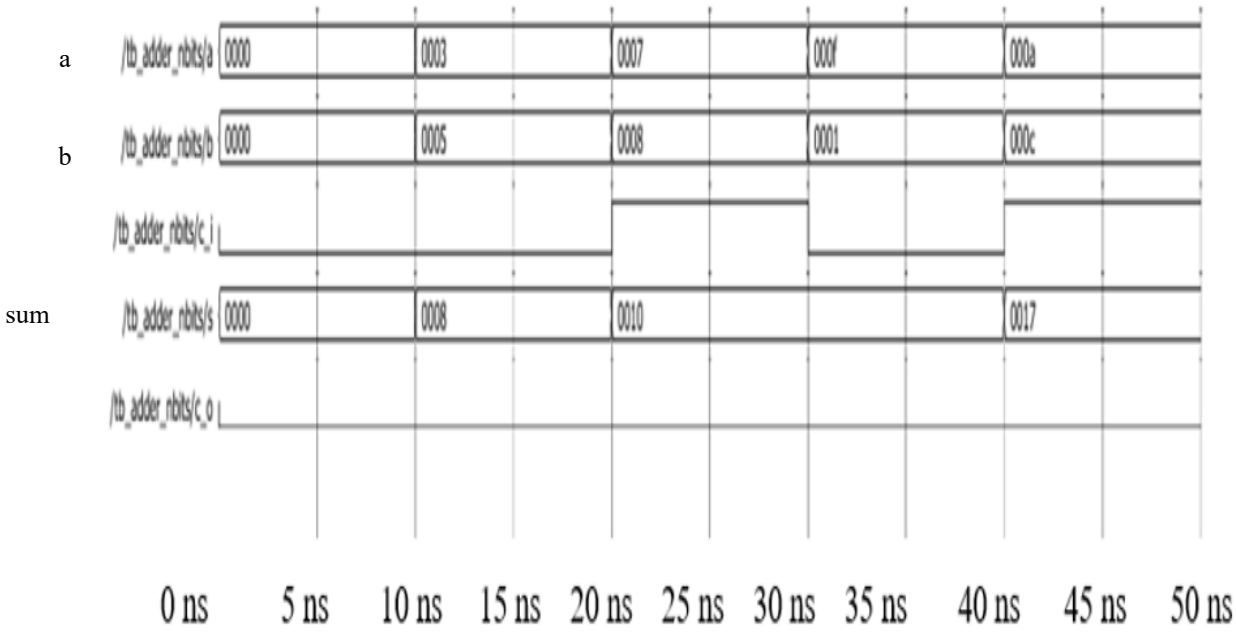

Figure I.4: 16-bit full adder Simulation Waveform

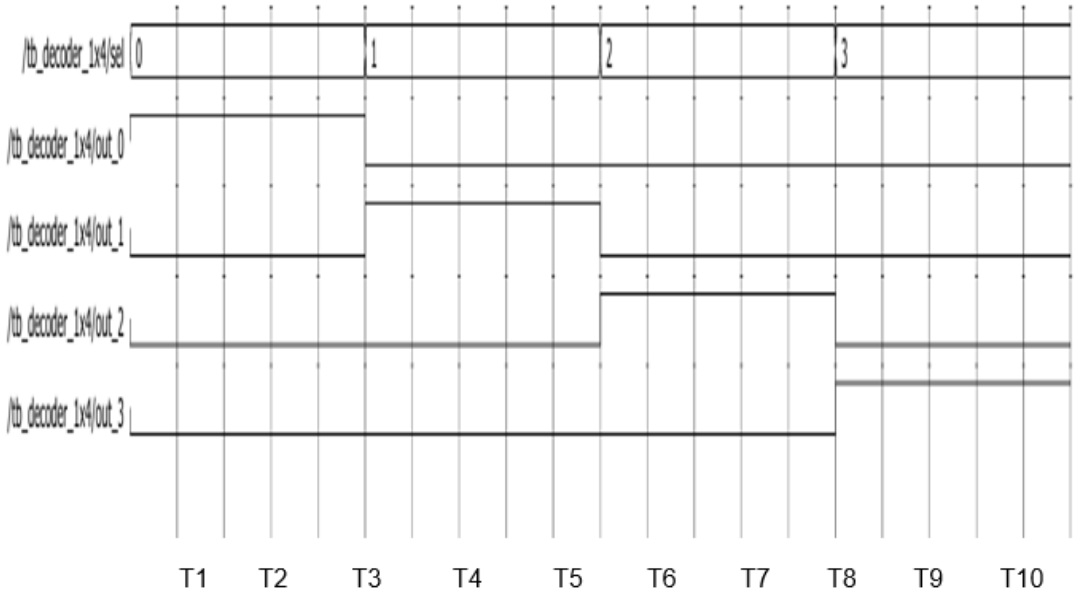

Figure I.5: 1x4 Decoder Simulation Waveform

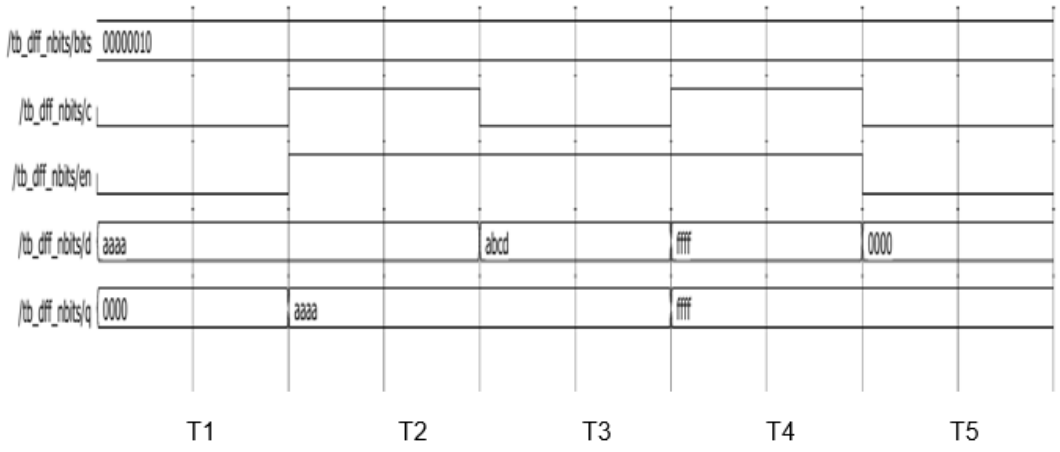

Figure I.6: 16-bit D-Flip Flop Simulation Waveform.

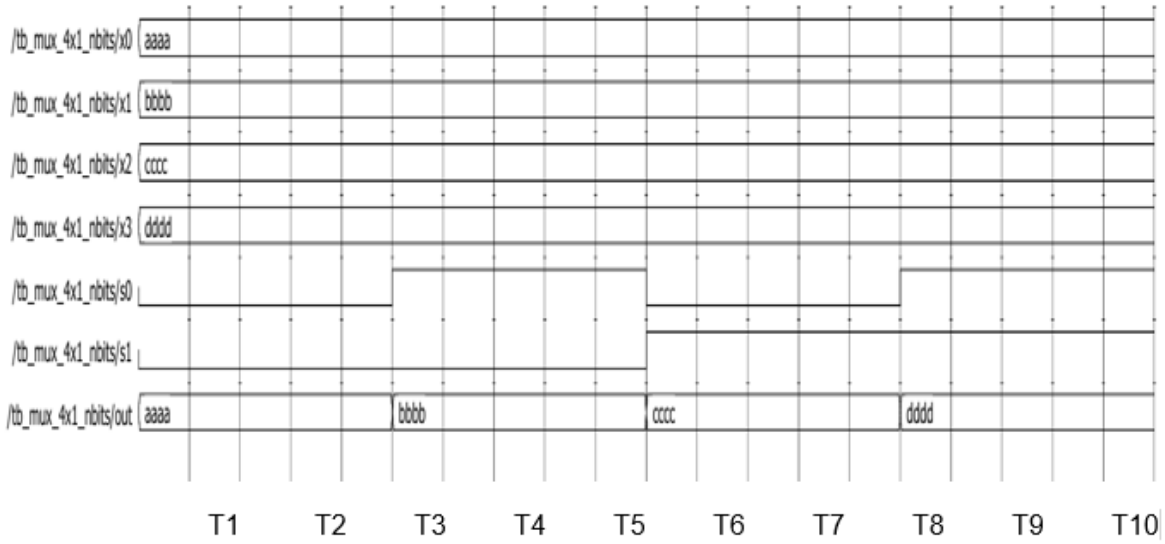

Figure I.7: 4x1 Multiplexer Simulation Waveform

Appendix II – Relevant Equations

1. Critical Frequency Formula for Static Timing Analysis

$$F_{max} = \frac{1}{t_{clock\_constr} - WNS_{setup}}$$

Appendix III – Code Listings

1. FPGA top wrapper is used to connect the main module to prototype it at hardware level. Below is the Verilog code of it.

```
//============================================================
// FPGA Top Wrapper for Minimalist Processor
// Debounce + One-Pulse for push buttons using combined module
// ILA instantiated for internal signals
//============================================================

module fpga_top (
    // Board Inputs
    input         clk_fpga,
    input  [15:0] dip_value_in,   // DIP switches for value
    input         btn_add,        // Push button for ADD
    input         btn_jump,       // Push button for JUMP
    input         btn_adv,        // Push button for advance/clock
    input         btn_dest_msb,   // Push button for dest[1]
    input  [2:0]  dip_ctrl,       // DIP[0]=dest[0], DIP[1]=src[0], DIP[2]=prog

    // Board Outputs
    output [3:0]  pc,
    output [15:0] o_b,
    output [15:0] o_r0,
    output [15:0] o_r1,
    output [15:0] o_r2,
    output [15:0] o_r3,
    output [31:0] w2_dbg,
    output        asm_store_clk_dbg
);

    //============================================================
    // Debounce + One-Pulse for push buttons
    //============================================================
    wire add, jump, dest_msb, advance_pulse;
`ifdef SIM
        // Simple edge detector for simulation
```

```
        assign add         = btn_add;
        assign jump        = btn_jump;
        assign dest_msb    = btn_dest_msb;
        assign advance_pulse = btn_adv;
    `else
        // Real debounce + one-pulse for synthesis
        debounce_onepulse   udb_add               (.clk(clk_fpga),   .rst(1'b0),
.btn_in(btn_add),      .pulse_out(add));
        debounce_onepulse   udb_jump              (.clk(clk_fpga),   .rst(1'b0),
.btn_in(btn_jump),     .pulse_out(jump));
        debounce_onepulse     udb_dest_msb     (.clk(clk_fpga),     .rst(1'b0),
.btn_in(btn_dest_msb), .pulse_out(dest_msb));
        debounce_onepulse   udb_adv               (.clk(clk_fpga),   .rst(1'b0),
.btn_in(btn_adv),      .pulse_out(advance_pulse));
    `endif

  /*     debounce_onepulse   udb_add               (.clk(clk_fpga),   .rst(1'b0),
.btn_in(btn_add),      .pulse_out(add));
    debounce_onepulse udb_jump    (.clk(clk_fpga), .rst(1'b0), .btn_in(btn_jump),
.pulse_out(jump));
    debounce_onepulse       udb_dest_msb       (.clk(clk_fpga),       .rst(1'b0),
.btn_in(btn_dest_msb), .pulse_out(dest_msb));
    debounce_onepulse udb_adv     (.clk(clk_fpga), .rst(1'b0), .btn_in(btn_adv),
.pulse_out(advance_pulse));

    assign add = btn_add;   // direct pass-through for sim
    assign jump = btn_jump;
    assign dest_msb = btn_dest_msb;
    assign advance_pulse = btn_adv;*/

    //=========================================================
    // Map DIP switches and buttons to processor signals
    //=========================================================
    wire [15:0] value   = dip_value_in;
    wire [1:0]  dest    = {dest_msb, dip_ctrl[0]};
    wire [1:0]  src     = {1'b0, dip_ctrl[1]};
    wire        prog    = dip_ctrl[2];
    wire       advance = advance_pulse; // processor clocked by one-pulse advance
```

```
    //=========================================================
    // Instantiate Processor Core
    //=========================================================
    add_jump_cpu u_core (
        .value   (value),
        .dest    (dest),
        .src     (src),
        .add     (add),
        .jump    (jump),
        .prog    (prog),
        .advance (advance),
        .pc      (pc),
        .o_b     (o_b),
        .o_r0    (o_r0),
        .o_r1    (o_r1),
        .o_r2    (o_r2),
        .o_r3    (o_r3),
        .w2_dbg  (w2_dbg),
        .asm_store_clk_dbg(asm_store_clk_dbg)
    );

    //=========================================================
    // Instantiate ILA (optional debug)
    //=========================================================
     u_ila probe_regs(
        .clk(clk_fpga),   // <<< FIXED: use system clock
        .probe_pc(pc),
        .probe_r0(o_r0),
        .probe_r1(o_r1),
        .probe_r2(o_r2),
        .probe_r3(o_r3),
        .probe_b(o_b)
    );

endmodule
```

Listing II.1 Verilog Module for the FPGA

2. Add-Jump processor is the main module whose Verilog code is given below and the results are accurate.

```
//============================================================
// Minimalist Instruction Set Processor (ADD + JUMP)
// Top-Level Module
//============================================================

module add_jump_cpu (
    // Inputs
    input        [15:0] value,
    input        [1:0]  dest,
    input        [1:0]  src,
    input               add,
    input               jump,
    input               prog,
    input               advance,

    // Outputs (must be reg if driven by always blocks or initialized)
    output   [3:0]  pc,
    output   [15:0] o_b,
    output   [15:0] o_r0,
    output   [15:0] o_r1,
    output   [15:0] o_r2,
    output   [15:0] o_r3,
    output   [31:0] w2_dbg,
    output    asm_store_clk_dbg
);

//============================================================
// Internal Signals
//============================================================

    // Assembler <-> Memory
    wire [31:0] asm_inst;
    wire        asm_store_clk;

    // Memory <-> Decoder
    wire [31:0] mem_data;
```

```
    // Program Counter
    wire [3:0]  pc_internal;

    // Decoder Outputs
    wire [15:0] dec_imm;
    wire [1:0]  dec_rs;
    wire [1:0]  dec_rd;
    wire        dec_jump;
    wire        dec_add;

    // Execute
    wire [15:0] exe_result;
    wire [15:0] exe_b;

    // Control Signals
    wire clk_regfile = advance;


//=============================================================
// Assembler
//=============================================================
    assembler u_assembler (
        .prog     (prog),
        .value    (value),
        .dest     (dest),
        .src      (src),
        .asm_add  (add),
        .asm_jump (jump),
        .inst     (asm_inst),
        .store_clk(asm_store_clk)
    );

    assign w2_dbg = asm_inst;  // Debug output
    assign asm_store_clk_dbg = asm_store_clk;

//=============================================================
// EEPROM / Instruction Memory
//=============================================================
```

```
    eeprom u_eeprom (
        .c    (asm_store_clk),
        .str  (1'b1),
        .ld   (1'b1),
        .d_in (asm_inst),        // Instruction from assembler
        .a    (pc_internal),     // Program Counter Address
        .d    (mem_data)         // Instruction to decoder
    );

//=============================================================
// Program Counter
//=============================================================
    counter_with_preset #(.bits(4)) u_pc (
        .c    (advance),
        .en   (1'b1),
        .dir  (1'b0),
        .in   (dec_imm[3:0]),
        .ld   (dec_jump),
        .clr  (1'b0),
        .out  (pc_internal)
    );

    assign pc = pc_internal;

//=============================================================
// Decoder
//=============================================================
    decoder u_decoder (
        .d_prog (prog),
        .bypass (asm_inst),    // Direct assembler bypass
        .d_inst (mem_data),    // Instruction from memory
        .d_a    (dec_imm),
        .d_rs   (dec_rs),
        .d_rd   (dec_rd),
        .d_jump (dec_jump),
        .d_add  (dec_add)
    );

//=============================================================
```

```
// Execute
//============================================================
    execute u_execute (
        .exe_a      (dec_imm),
        .exe_b      (exe_b),
        .exe_add    (dec_add),
        .exe_result (exe_result)
    );

//============================================================
// Register File
//============================================================
    regfile u_regfile (
        .rs     (dec_rs),
        .rd     (dec_rd),
        .result (exe_result),
        .clk    (clk_regfile),
        .o_b    (o_b),
        .o_r0   (o_r0),
        .o_r1   (o_r1),
        .o_r2   (o_r2),
        .o_r3   (o_r3)
    );

    assign exe_b = o_b;

endmodule
```

Listing II.2 Verilog Module for the Add-Jump CPU (Main Module)

3. The testbench codes of the fpga_top and add_jump_cpu module is given below.

```
module tb_fpga_top;

    reg         clk_fpga;
    reg  [15:0] dip_value_in;
    reg         btn_add;
    reg         btn_jump;
    reg         btn_adv;
    reg         btn_dest_msb;
    reg  [2:0]  dip_ctrl;

    wire [3:0]  pc;
    wire [15:0] o_b, o_r0, o_r1, o_r2, o_r3;
    wire [31:0] w2_dbg;
    wire        asm_store_clk_dbg;

    // Instantiate DUT
    fpga_top uut (
        .clk_fpga    (clk_fpga),
        .dip_value_in(dip_value_in),
        .btn_add     (btn_add),
        .btn_jump    (btn_jump),
        .btn_adv     (btn_adv),
        .btn_dest_msb(btn_dest_msb),
        .dip_ctrl    (dip_ctrl)
      /*  .pc          (pc),
        .o_b         (o_b),
        .o_r0        (o_r0),
        .o_r1        (o_r1),
        .o_r2        (o_r2),
        .o_r3        (o_r3),
       // .w2_dbg      (w2_dbg),
       // .asm_store_clk_dbg(asm_store_clk_dbg)*/
    );

    // Clock
    initial clk_fpga = 0;
    always #5 clk_fpga = ~clk_fpga; // 100MHz
```

```
    // Stimulus
    initial begin
        btn_add = 0;
        btn_jump = 0;
        btn_adv = 0;
        btn_dest_msb = 0;
        dip_value_in = 16'h00A5;
        dip_ctrl = 3'b000;

        #20;
        dip_ctrl = 3'b101; // prog=1, dest[0]=1, src=0

        // ADD
        $display("[%0t] Press ADD", $time);
        btn_add = 1; #10; btn_add = 0;
        #50;

        // JUMP
        $display("[%0t] Press JUMP", $time);
        btn_jump = 1; #10; btn_jump = 0;
        #50;

        // ADV
        $display("[%0t] Advance 3 times", $time);
        repeat(3) begin
            btn_adv = 1; #10; btn_adv = 0;
            #30;
        end

        #100;
        $display("[%0t] Simulation finished", $time);
        $stop;
    end

    // Live Monitor
    initial begin
        $monitor("[%0t] pc=%0d add=%b jump=%b prog=%b store_clk=%b inst=%h",
                 $time, pc, btn_add, btn_jump, dip_ctrl[2],
```

```
                  asm_store_clk_dbg, w2_dbg);
    end

    // VCD dump
    initial begin
        $dumpfile("fpga_top_tb.vcd");
        $dumpvars(0, tb_fpga_top);
    end

    // Self-check: detect store_clk
    reg saw_store = 0;
    always @(posedge clk_fpga) begin
        if (asm_store_clk_dbg) saw_store <= 1;
    end

    initial begin
        #500;
        if (saw_store)
            $display("PASS: asm_store_clk_dbg pulsed");
        else
            $display("FAIL: asm_store_clk_dbg never pulsed");
    end
endmodule
```

```
//============================================================
// Testbench for Minimalist Instruction Set Processor (ADD + JUMP)
//============================================================
module tb_add_jump_cpu;

    //============================================================
    // Inputs to DUT
    //============================================================
    reg [15:0] value;
    reg [1:0]  dest;
    reg [1:0]  src;
    reg        add;
    reg        jump;
    reg        prog;
    reg        advance;

    //============================================================
    // Outputs from DUT
    //============================================================
    wire [3:0]  pc;
    wire [15:0] o_b;
    wire [15:0] o_r0, o_r1, o_r2, o_r3;
    wire [31:0] w2_dbg;   // assembler instruction (debug)

    //============================================================
    // DUT Instantiation
    //============================================================
    add_jump_cpu uut (
        .value   (value),
        .dest    (dest),
        .src     (src),
        .add     (add),
        .jump    (jump),
        .prog    (prog),
        .advance (advance),
        .pc      (pc),
        .o_b     (o_b),
        .o_r0    (o_r0),
        .o_r1    (o_r1),
```

```
        .o_r2    (o_r2),
        .o_r3    (o_r3),
        .w2_dbg  (w2_dbg)
    );

    //=========================================================
    // Clock Generation (using advance as clock)
    //=========================================================
    initial advance = 0;
    always #5 advance = ~advance; // 10 ns period

    //=========================================================
    // Simulation
    //=========================================================
    initial begin
        // Setup waveform dump
        $dumpfile("tb_top.vcd");
        $dumpvars(0, tb_top);

        // Initialize Inputs
        value = 0;
        dest  = 0;
        src   = 0;
        add   = 0;
        jump  = 0;
        prog  = 0;

        #10; // Wait for global reset

        //--------------------------
        // Programming Phase
        //--------------------------
        prog = 1;
        add  = 1;
        jump = 0;

        // Add 5 to R0
        value = 16'd5;  dest = 2'b00; src = 2'b00; #10;
        // Add 10 to R1
```

```
        value = 16'd10; dest = 2'b01; src = 2'b01; #10;
        // Add 20 to R2
        value = 16'd20; dest = 2'b10; src = 2'b10; #10;
        // Add 30 to R3
        value = 16'd30; dest = 2'b11; src = 2'b11; #10;

        // Jump to address 2
        add = 0; jump = 1;
        value = 16'd2; dest = 2'b00; src = 2'b00; #10;
        jump = 0;

        // Disable programming mode for execution
        prog = 0; add = 1; jump = 0;

        //--------------------------
        // Execution Phase
        //--------------------------
        // Step 1: Add 15 to R0
        value = 16'd15; dest = 2'b00; src = 2'b00; #10;
        // Step 2: R1 = R1 + R2
        value = 16'd0;  dest = 2'b01; src = 2'b10; #10;
        // Step 3: R3 = R3 + R0
        value = 16'd0;  dest = 2'b11; src = 2'b00; #10;
        // Step 4: R2 = R2 + 50
        value = 16'd50; dest = 2'b10; src = 2'b10; #10;

        // Observe PC & Registers
        repeat (10) begin
            #10;
            $display("T=%0t | PC=%0d | o_b=%0d | R0=%0d | R1=%0d | R2=%0d |
R3=%0d",
                     $time, pc, o_b, o_r0, o_r1, o_r2, o_r3);
        end

        $stop;
    end

endmodule
```

Listing III.3 Testbench Modules for the FPGA Wrapper and Add-Jump CPU Core