A Project Report

on

Implementation of RISC Architecture

Submitted by

D. Vinuthna Sri(18BEC009)

M.S. Gayathri Sahithi(18BEC029)

N. Bhargav Kumar(18BEC030)

Y. Vaanisha(18BEC052)

Under the guidance of

Dr. Jagadish D N

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY
DHARWAD

ABSTRACT

A microprocessor is the controlling unit of a micro-computer, and it performs various arithmetic, logical (ALU) operations, and communicates with the devices connected to it(peripherals). The main components of a microprocessor are a register file, ALU unit, control unit, data and instruction memory. There are various architectures upon which a microprocessor can be designed according to. RISC-V is one of the most popular architectures to design a microprocessor. It features an instruction set containing of 47 instructions. A microprocessor can be designed upon any number of bits as per the user requirement (8,16,32,64 bit). This work mainly focuses on the backend design of a simple beginner friendly Microprocessor using RISC-V ISA. This work helps understanding the basic concepts of how a processor works and helps gain experience in designing a new microprocessor.

Contents

Introduction

Microprocessors or the microcontrollers are called the brains of SoC's, as they are the controlling unit of a micro-computer. They perform arithmetic, logical and control operations (ALU). The main components of a microprocessor are a register file, ALU unit, control unit, data and instruction memory. Microprocessors on an SoC or a board are called central processing unit (CPU) core, and is fabricated on a single chip called an IC. This IC design follows the VLSI design flow starting from gathering the requirements like whether the microprocessor is a general-purpose processor or application specific processor, the cost, the timing constraints, the power consumption etc. followed by the frontend and backend design flows. The frontend design flow consists of all the steps in the chip fabrication process from gathering 'specifications' until 'functional verification'. The backend design flow consists of logic synthesis (Gate level simulation), place and route, fabrication, post silicon validation.

There are various architectures upon which a microprocessor can be designed. Each architecture has various features and related instruction sets. Two major types of instruction set architectures are RISC (reduced instruction set) and CISC (complex instruction set). RISC architecture is based on a simple instruction set that has simple instructions which can be executed in a single clock cycle. RISC-V is an extension of RISC architecture which is provided under open-source licenses. This architecture contains an instruction set of 47 instructions, and the base names of this ISA are RVWMO, RV32I, RV32E, RV64I, RV128I.

This paper mainly focuses on the backend design flow of the microprocessor designed based on RISC-V ISA. This work helps understanding the basic concepts of how a processor works and helps gain experience in designing a new microprocessor using basic terminologies. The software dependencies used for this project are Vivado ISE Design Suite for HDL coding and RTL Analysis, and a linux-based tool called Yosys.

Literature Review

RISC-V ISA is a load-store architecture, and its floating-point instructions use IEEE 754 floating-point. Following are some of the features of RISC-V:

- RISC-V is an open-source Instruction Set Architecture (ISA) based on established reduced instruction set computer (RISC) principles.
- It will allow smaller device manufacturers to build hardware without paying royalties and allow developers and researchers to design and experiment with a proven and freely available instruction set architecture.
- Can be extended or customized for a variety of hardware or application requirements.
- RISC-V in was made specifically to be easy to teach while pragmatic enough to actually allow the implementation of high-performance microprocessors.

The RISC-V ISA has a base and extension set shown as follows (source: Wikipedia)

Name	Description	Version	Status[a]	Instruction count
Base				
RVWMO	Weak Memory Ordering	2.0	Ratified	
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified	40
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open	40
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified	15
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open	15
Extension				
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified	8 (RV32) / 13 (RV64)
A	Standard Extension for Atomic Instructions	2.1	Ratified	11 (RV32) / 22 (RV64)
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified	26 (RV32) / 30 (RV64)
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified	26 (RV32) / 32 (RV64)
Zicsr	Control and Status Register (CSR)	2.0	Ratified	6
Zifencei	Instruction-Fetch Fence	2.0	Ratified	1
G	Shorthand for the IMAFDZicsr Zifencei base and extensions, intended to represent a standard general-purpose ISA	N/A	N/A	
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified	28 (RV32) / 32 (RV64)
L	Standard Extension for Decimal Floating-Point	0.0	Open	
C	Standard Extension for Compressed Instructions	2.0	Ratified	40
B	Standard Extension for Bit Manipulation	1.0	Frozen	42
J	Standard Extension for Dynamically Translated Languages	0.0	Open	
T	Standard Extension for Transactional Memory	0.0	Open	
P	Standard Extension for Packed-SIMD Instructions	0.9.10	Open	
V	Standard Extension for Vector Operations	1.0	Frozen	186
K	Standard Extension for Scalar Cryptography	1.0.0	Ratified	49
N	Standard Extension for User-Level Interrupts	1.1	Open	3
H	Standard Extension for Hypervisor	1.0.0-rc	Frozen[29]	15
S	Standard Extension for Supervisor-level Instructions[30]	1.12	Frozen	7
Zam	Misaligned Atomics	0.1	Open	
Ztso	Total Store Ordering	0.1	Frozen	

For ASIC layout design a tool Coriolis/alliance is helpful for placing and routing a design.

Yosys:
A framework for Verilog RTL, synthesis.

Some of the features and typical applications:

- Process almost any synthesizable Verilog-2005 design
- Converting Verilog to BLIF / EDIF/ BTOR / SMT-LIB / simple RTL Verilog / etc.
- Built-in formal methods for checking properties and equivalence
- Mapping to ASIC standard cell libraries (in Liberty File Format)
- Foundation and/or front-end for custom flows

We get the ASIC mapping of the design using these tools.

Motivation

Although there are thousands of microprocessors available in the market, understanding all the functionalities of each different part of processor is a tedious task. This can be done more precisely once we start building a microprocessor from the scratch on our own. Designing a new microprocessor from the scratch includes gathering the requirements, designing each and every module using HDL, RTL analysis all the way till FPGA prototyping and developing ASIC design. Beginners like undergrad engineers, VLSI engineers, students who are seeking industrial exposure can consider building a new microprocessor on their own to understand the concepts better. The motivation of our work is to learn the backend work flow of a microprocessor designed by us to get the deeper insights of how a microprocessor is actually fabricated at the industrial level.

The first part of our project focuses on several building blocks of CPU that are designed using Verilog (Hardware description languages) and the implementation is done by integrating them to form the “Top module” of CPU. The generated bitstream is then dumped on to a FPGA for prototyping.

The second part of our project focuses on the ASIC design of our processor using Coriolis/Alliance tool which takes VHDL codes (Hardware description language) of our CPU as input and generates the ASIC design.

Processor Design

We developed a single cycle, non-pipelined RISC-V CPU from scratch. Our CPU will perform *11* basic operations defined in RISC-V instruction set and can be easily extended to include more advanced as well as custom instruction sets. Various CPU modules are designed as separate blocks and then integrated in the top module. The following are the blocks of our microprocessor:

❖ **Data path unit:**

- **Write Data Selection Mux** (wda-mux):
  if set, ALU produces memory address and contents of it are transferred back to register, else output of ALU is stored to register
- **ALU Second Input Selection Mux**(alu-mux):
  if set, information from instruction is passed to ALU, else register file contents are passed to ALU
- **Branch taken**(taken):
  Checks if branching is required
- **Register File or Register Bank**(reg-file):
  A single register bank containing of 32 registers, each 32-bit wide. Reads combinational logic and writing always happens on positive edge of clock cycle when register address is non-zero
- **ALU** (alu):
  32-bit instruction set performs various arithmetic and logical operations. Total of 47 instructions are available in RISC-V ISA.
- **ALU control** (alu-control):
  A separate control unit for ALU which specifies what type of operation is required for what type of instruction (ex: jump requires ADD, branch requires XOR etc)
- **Immediate generation** (imm-gen):
  Derives information from bits in instruction itself (for I, S, SB types)
- **Data memory** (data-mem):
  Reads data in binary format to thereafter send data to registers and perform operations.

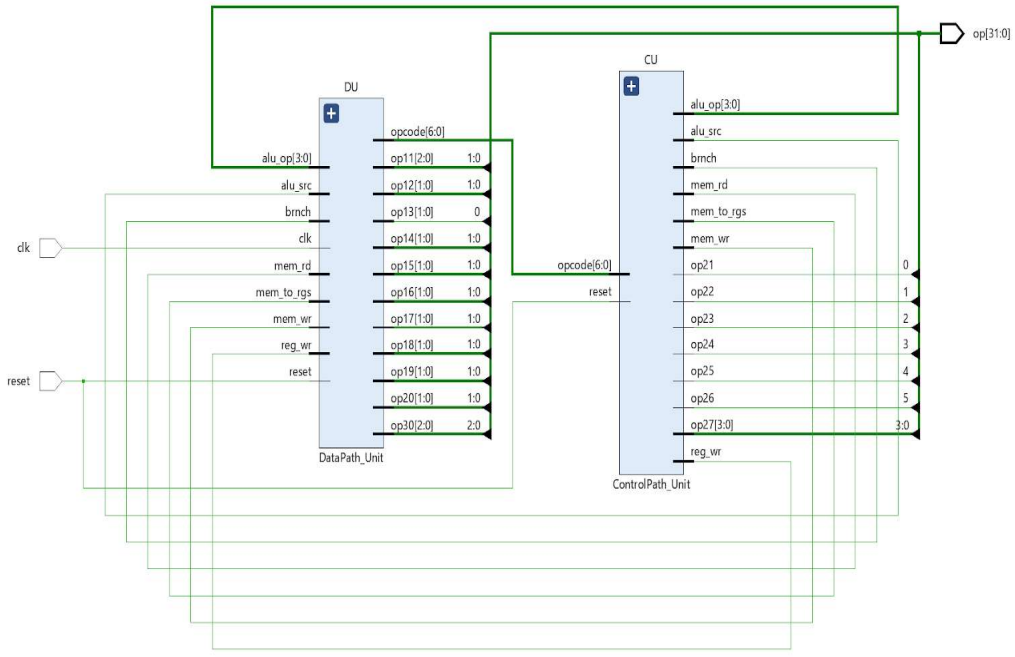
- **Instruction memory** (inst-mem):
  A module to read instructions in binary format.
- **Jump adder** (jmp-addr):
  Moves PC to required position

❖ **<u>Control path unit:</u>**

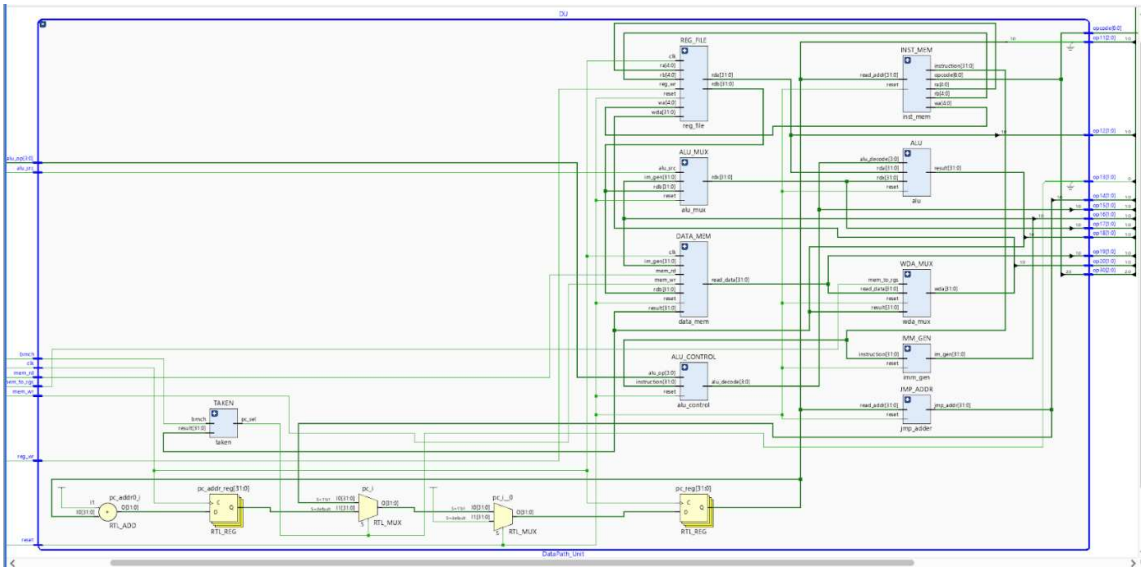
- **Control** (control):
  RISC-V defines bits [6:0] that needs to be used by control unit, which controls the type of instructions to be executed or being executed. The four types are R, I, S, SB types.

❖ **CPU-top** (top):
The top hierarchy module where all the above modules are instantiated and made into a single block.

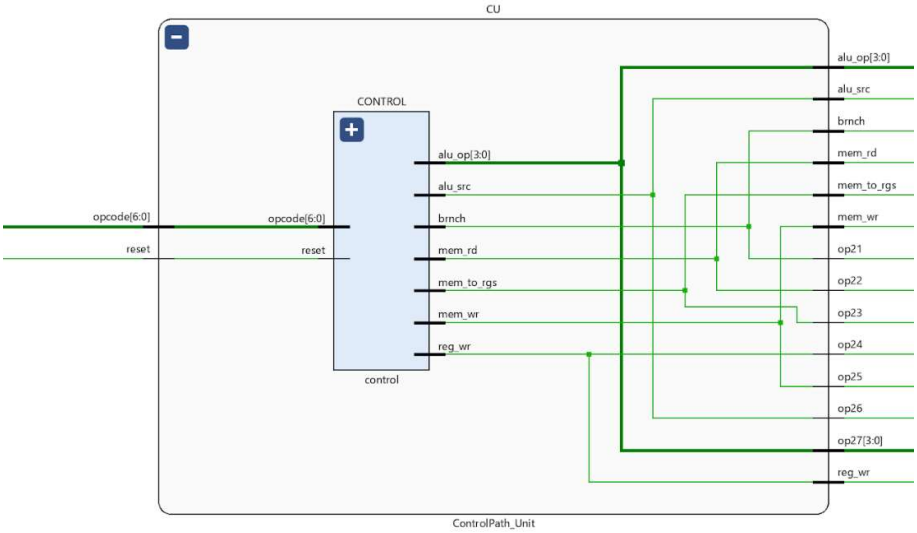
Elaborated design of our microprocessor generated after RTL analysis in Vivado too



Elaborated design of Data path unit:



Elaborated design of Control path unit:



Approach

Part 1:

In the process of processor design, FPGA Prototyping and Layout generation are the crucial steps. They give the information about power consumption, LUTs utilized, performance, Critical paths etc. So, we implemented our project in two parts. The first objective of our project is to synthesize, implement, generate bit stream and prototyping on Zynq 7000 FPGA by dumping the bit stream on FPGA. Synthesis and implementation provides information about power consumption, utilization, timing analysis.

For behavioral verification we dumped a small code snippet with 10 instructions:

	Machine code
LW r1, imm1	00000000001000000000000010000011
LW r2, imm2	00000000000100001000000100000011
SUB r3, r1, r2	01000000000100000000000110110011
ADD r4, r1, r2	00000000000100000000001000110011
AND r5, r1, r2	00000000000100000111001010110011
OR r6, r1, r2	00000000000100000110001100110011
XOR r7, r1, r2	00000000000100000100001110110011
SLL r8, r1, r2	00000000000100000001010000110011
SRL r9, r1, r2	00000000000100000101010010110011
SLT r10, r1, r2	00000000000100000010010100110011
SLTU r11, r1, r2	00000000000100000011010110110011

I-Type Instruction Decode:

Immediate and memory load operations based on the data from register and the instruction.

Imm [31:20]	rs1 [19:15]	Funct3 [14:12]	rd [11:7]	Opcode [6:0]	op	description
		000		0000011	LW rd, imm(rs1)	R[rd] = {M[R[rs1]+imm[31:20]]}

R-Type Instruction Decode:

Arithmetic and logical operations like and,or,xor,add,sub on registers.

Funct7 [31:25]	rs2 [24:20]	rs1 [19:15]	Funct3 [14:12]	rd [11:7]	Opcode [6:0]	op	description
			000		0110011	ADD rd ,rs1,rs2	R[rd] = R[rs1] + R[rs2]

S-Type Instruction Decode:

Store data in a register based on the data in instruction and register.

imm [31:25]	rs2 [24:20]	rs1 [19:15]	Funct3 [14:12]	imm [11:7]	Opcode [6:0]	op	description
			010		0100011	SW rs2,imm(rs1)	{M[R[rs1]+imm[31:25,11:7]}= R[rs2]

SB-Type Instruction Decode:

Increment pc with immediate value if there is a branch instruction like beq,bne.

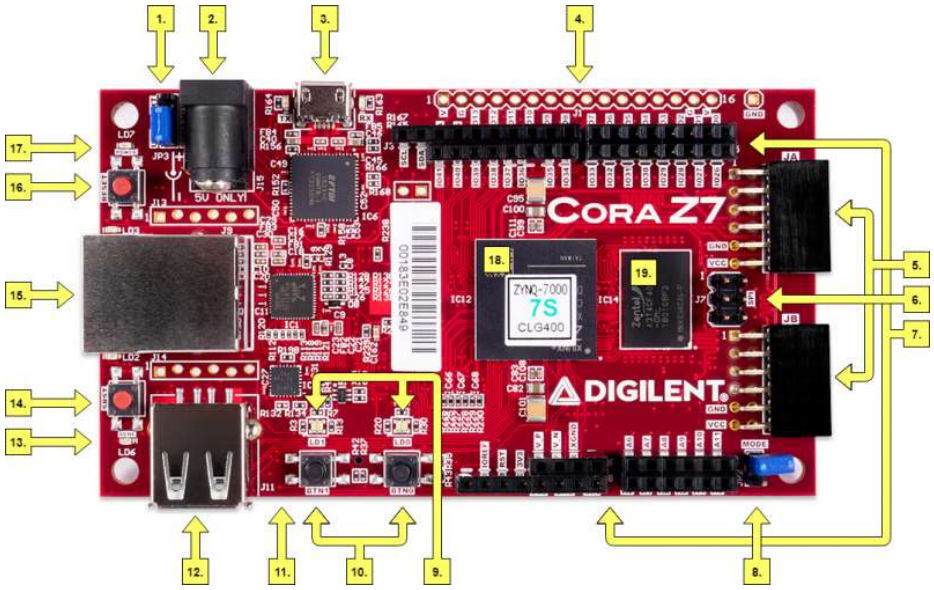
Funct7 [31:25]	r2 [24:20]	r1 [19:15]	Funct3 [14:12]	rd [11:7]	Opcode [6:0]	op	description
			000		1100011	BEQ rs1,rs2,imm	(if R[rs1] == R[rs2])

							PC = PC +(imm)

Cora Z7 Zynq7000s Board

The board we used to dump our code and generate a bitstream.

- Low cost
- Single core
- Frequency:667MHz
- ARM Cortex-A9 processor
- Memory: 512MB DDR3 USB and Ethernet connectivity
- General purpose input/output ports
- Arduino shield and Pmod connections for add-on hardware devices
- Programmable from microSD card



Part 2:

The second objective is to generate the ASIC design/ Layout of the microprocessor.

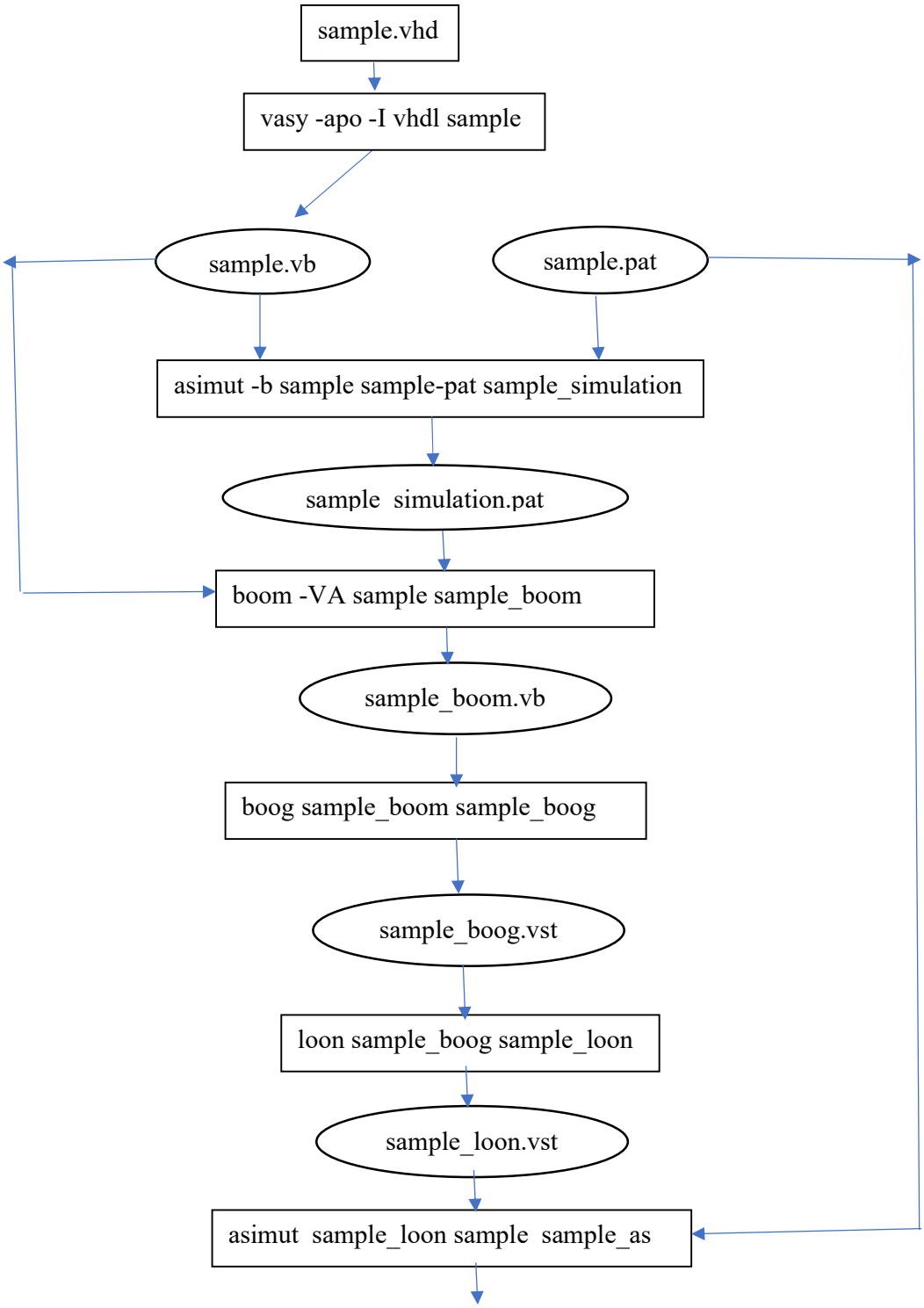
Our second part includes obtaining the layout design of our processor as an ASIC

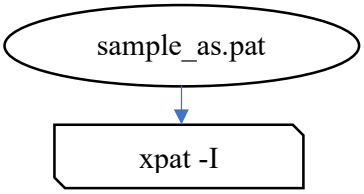
For which we are using a tool called alliance.

Alliance is an open source for cad tools and vlsi libraries. This tool is designed back in 1990s

So to convert our vhdl codes to layout, it only supports vhdl'87. So in order to go further we firstly removed all the if else blocks in our vhdl which are not supported in vhdl'87 and replaced them with case and when statements.There are many that need to be followed in

order to get our layout design. The below mentioned flow chart gives a walk through of what to do.





In our approach after mapping to library cells we run the below commands in command prompt:

- ocp sample_oo sampleoop
- nero -V -p sampleoop sample_oo sampleoor
- graal

vasy -- behavioral description from the vhdl file

asimut – this requires .pat file to simulate (behavioral functional verification)

xpat – displays the pattern

boom – optimization

boog – mapping to library cells

ocp – placing the design

nero – placing and routing the design

graal – to view our layout design

Results and Discussions

Part 1:

Expected result for a I type and R type instructions:

	I type	R type
opcode [6:0]	00011	0110011
brnch	0	0
mem_rd	0	0
mem_to_rgs	1	0
alu_op [3:0]	0	2
mem_wr	0	0
alu_src	1	0
reg_wr	1	1

Other outputs like alu decode,rda,rdb,rdx,ra,rb,wa,result vary for every instruction which can be seen in the below simulation results.

rda = R[ra];

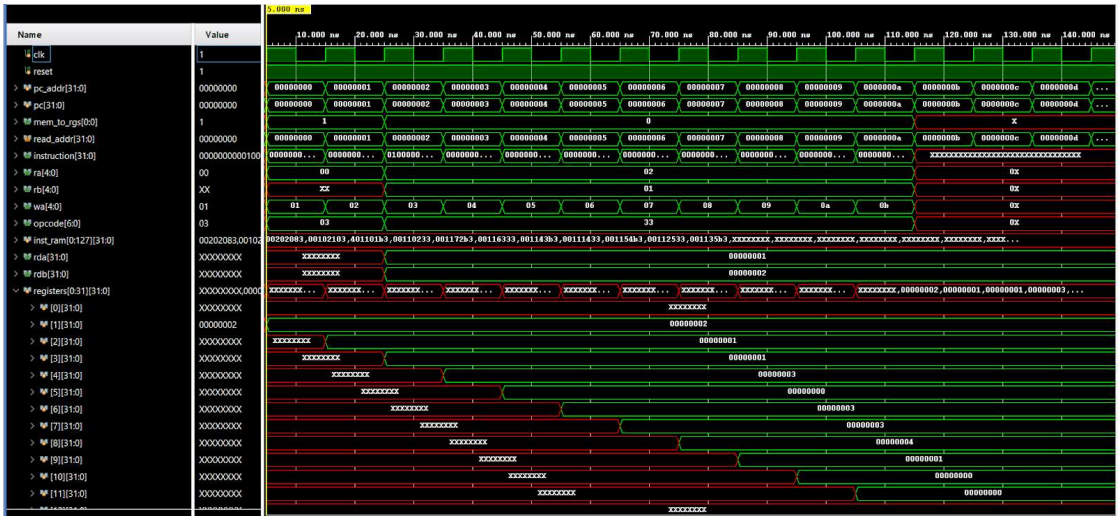
rdb = R[rb];

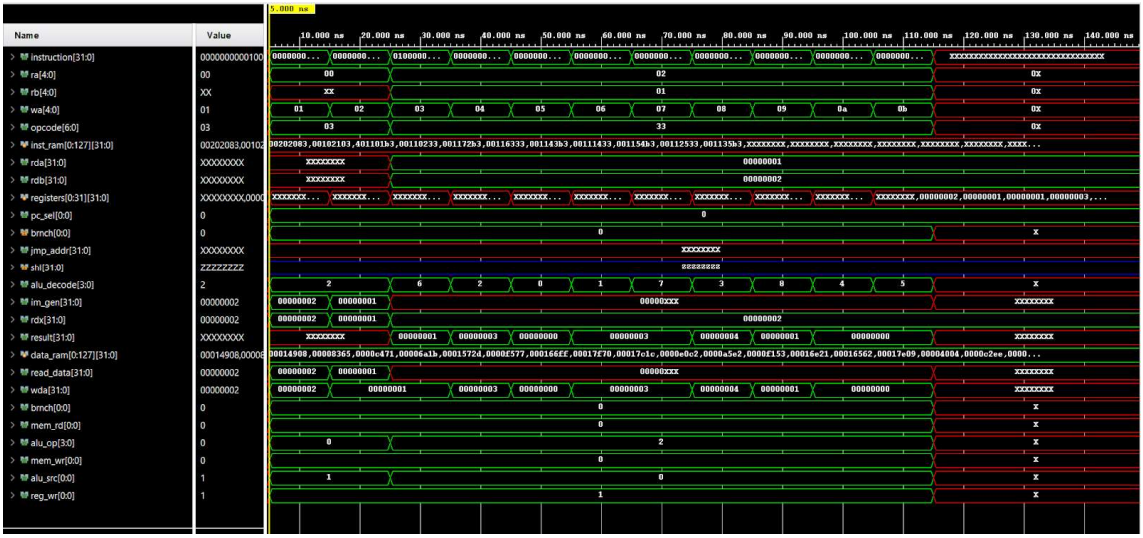
wda = R[wa];

The register transfer level schematic is generated post HDL synthesis . It represents the pre-optimized design in terms of generic symbols, such as multiplexers, adders, counters,buffers, AND,OR gates that are independent of the targeted Xilinx device.

For every clock cycle the instruction changes according to the instruction all other variables are also changed i.e., for every 10ns there is a change in the values till all the instructions execute.

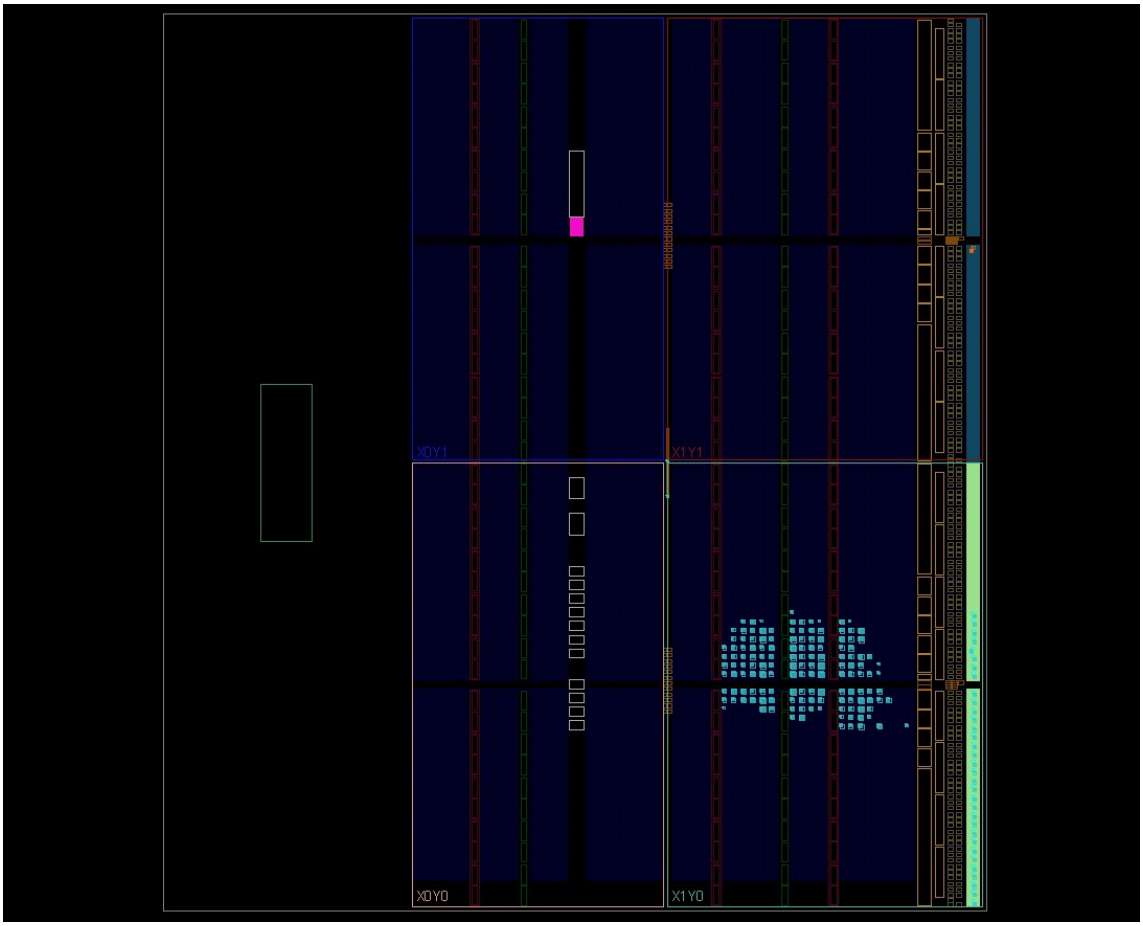
Simulation:





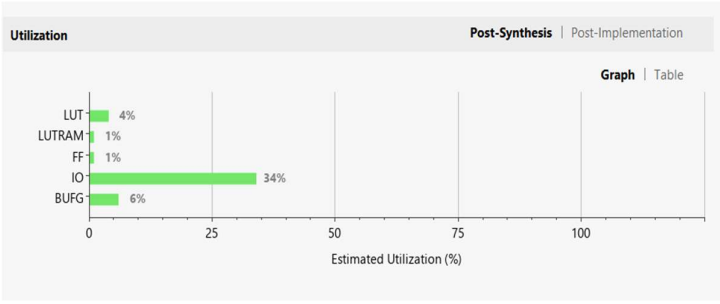
Implemented design:

The device shown below is our selected FPGA board in which some of the cells were selected which are configured in master file of the specified board.



Post synthesis:

- **Estimated Utilization**



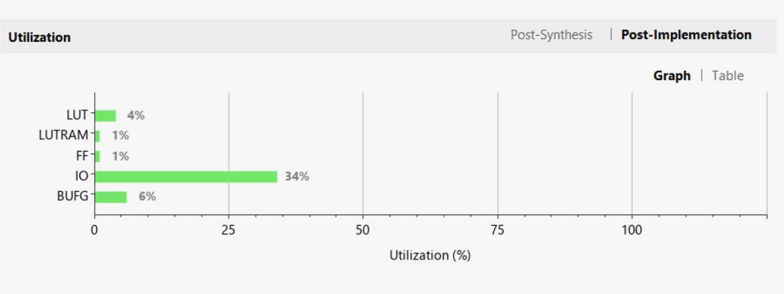
Utilization — **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	530	14400	3.68
LUTRAM	48	6000	0.80
FF	14	28800	0.05
IO	34	100	34.00
BUFG	2	32	6.25

Post Implememtation:

- **Actual Utilization**

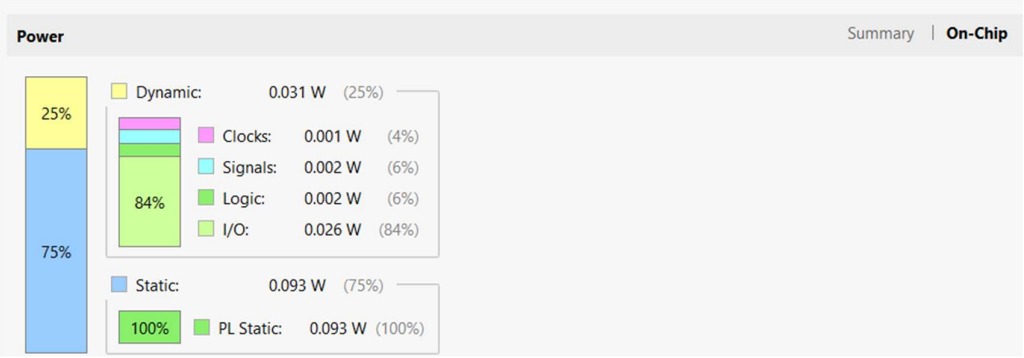


Utilization — Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	511	14400	3.55
LUTRAM	48	6000	0.80
FF	16	28800	0.06
IO	34	100	34.00
BUFG	2	32	6.25

Power report:



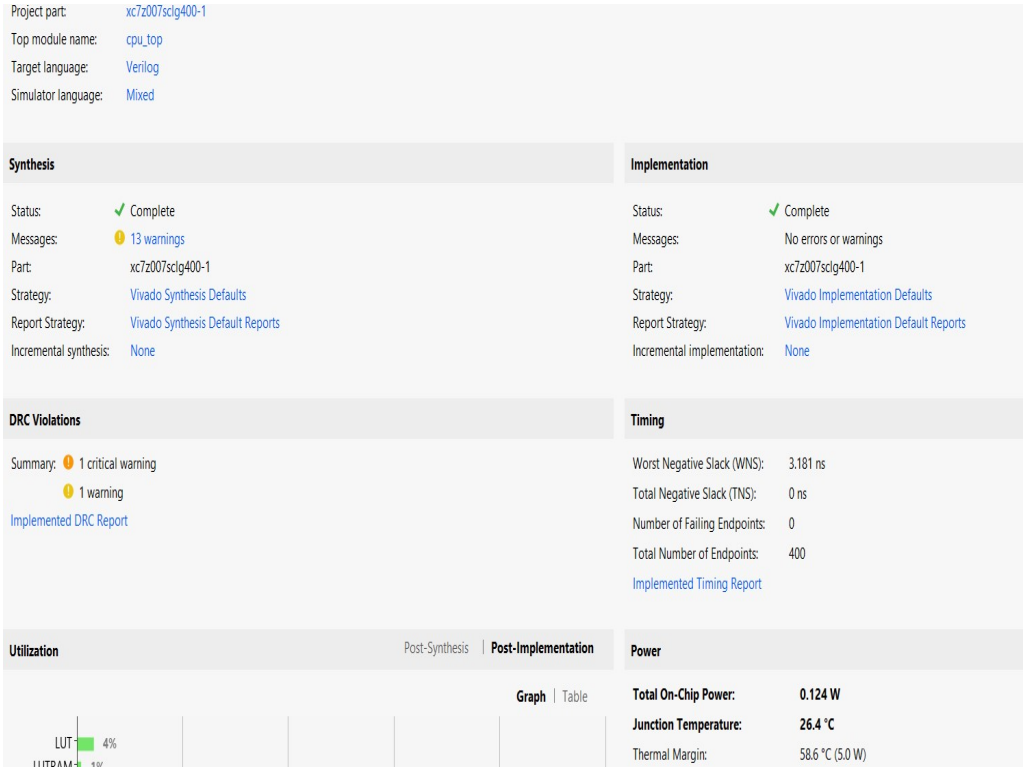
Power — **Summary** | On-Chip

Total On-Chip Power:	**0.124 W**
Junction Temperature:	**26.4 °C**
Thermal Margin:	58.6 °C (5.0 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

The total on-chip power is 0.124W and the confidence is medium which depicts that we can dump the code on to the FPGA and can expect a correct output. If the confidence level is low we cannot dump the code on to the board, it doesn't give expected output.

Project Summary:

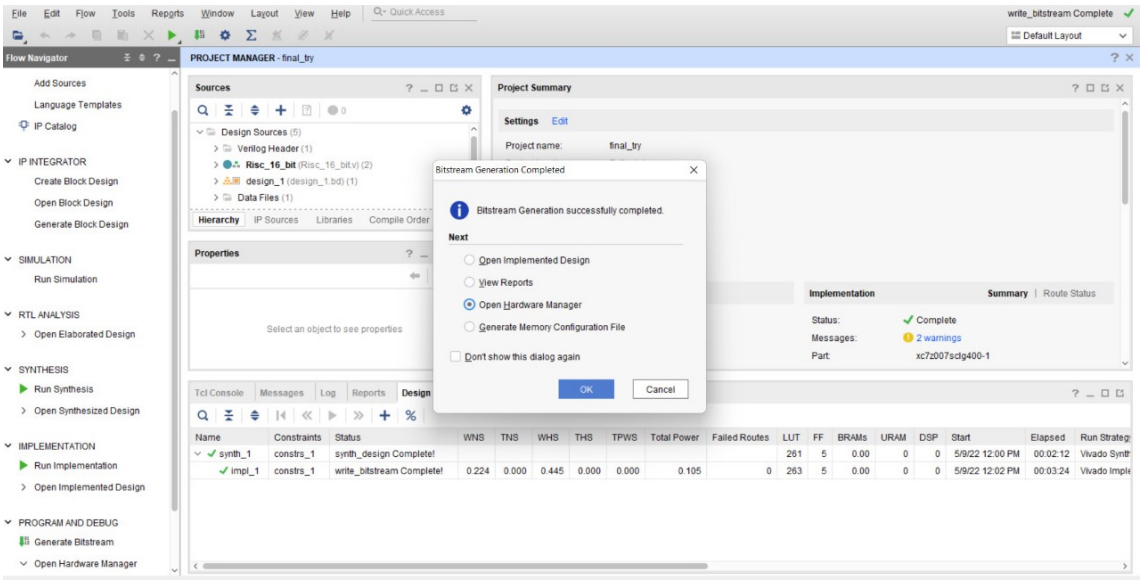


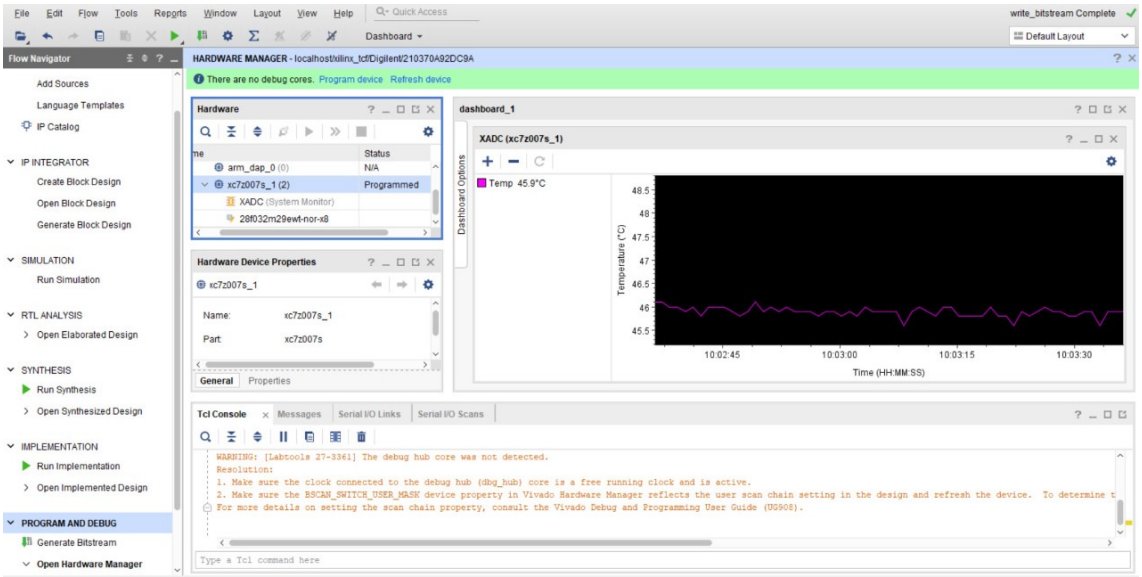
We can observe that the worst negative slack is 3.181ns, this implies the difference between the given delay in timing constraints and the actual delay after implementation. 3.181ns is the time exceeded by the processor that our expected delay.

Write to bitstream:

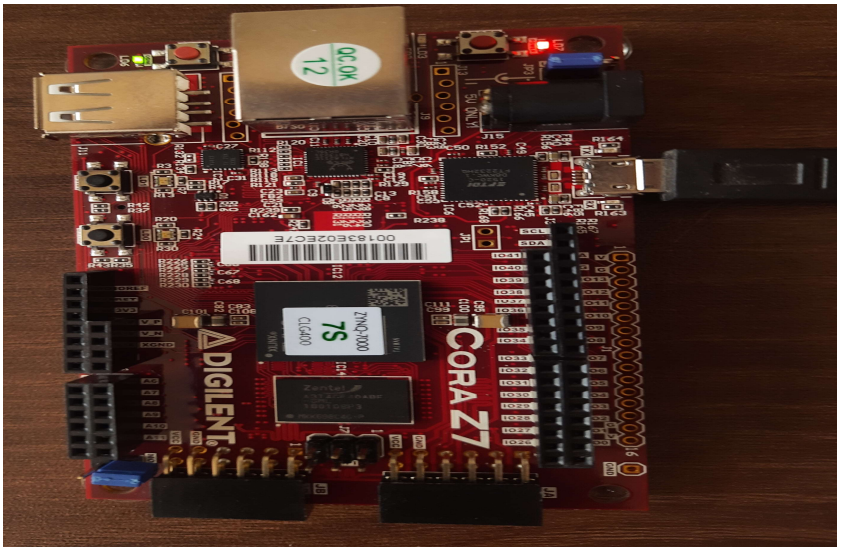
When we connect the FPGA board and start writing our bitstream to it we can observe a hardware manager in which we actually programme our board.

In hardware manager we first connect to the target device the we get an option to program device when we open the target where there are no debug errors. We can also see the temperature plot of the device in Vivado.





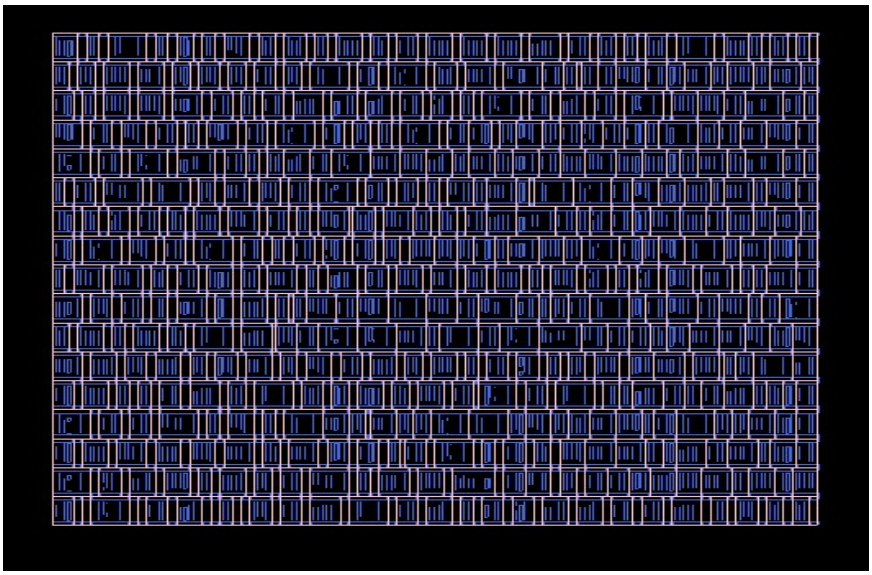
The device is programmed shown below. Here we can see a green light which indicates the device is programmed and the red light is the power supply.



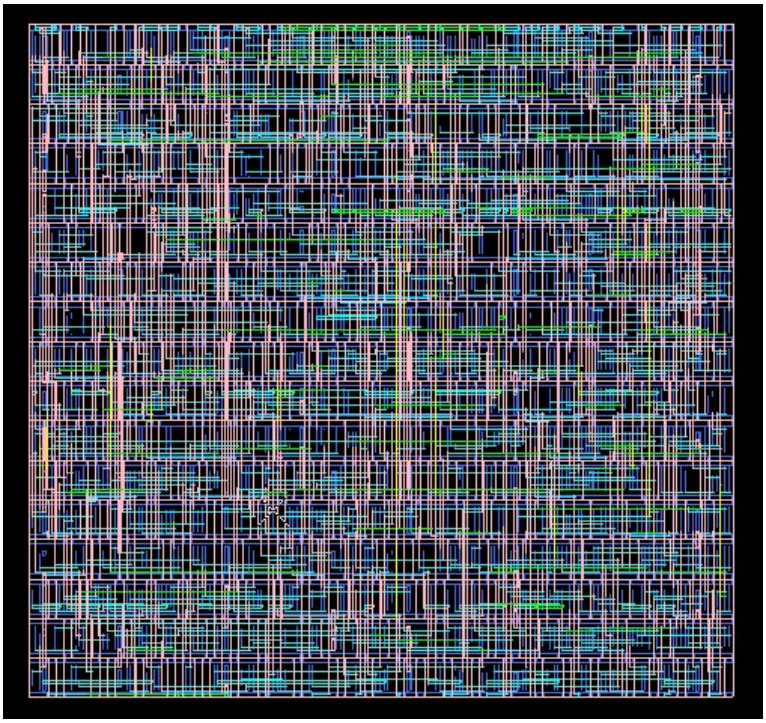
Part 2:

ALU Module:

- The placed layout design seen in Graal:

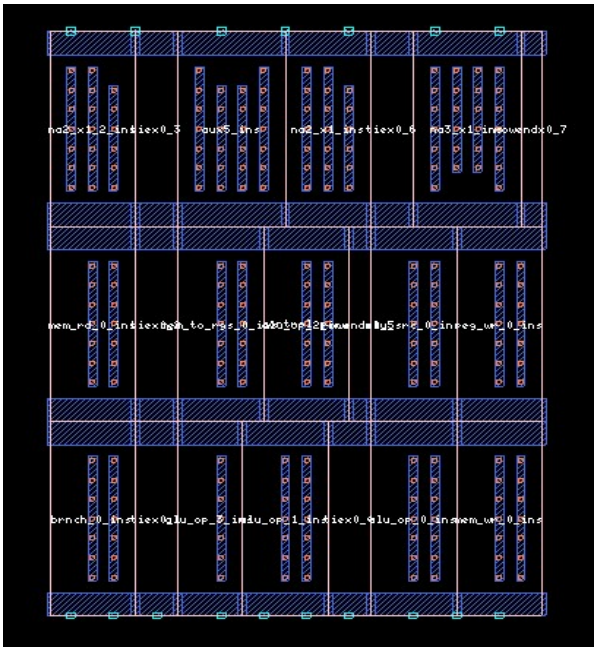


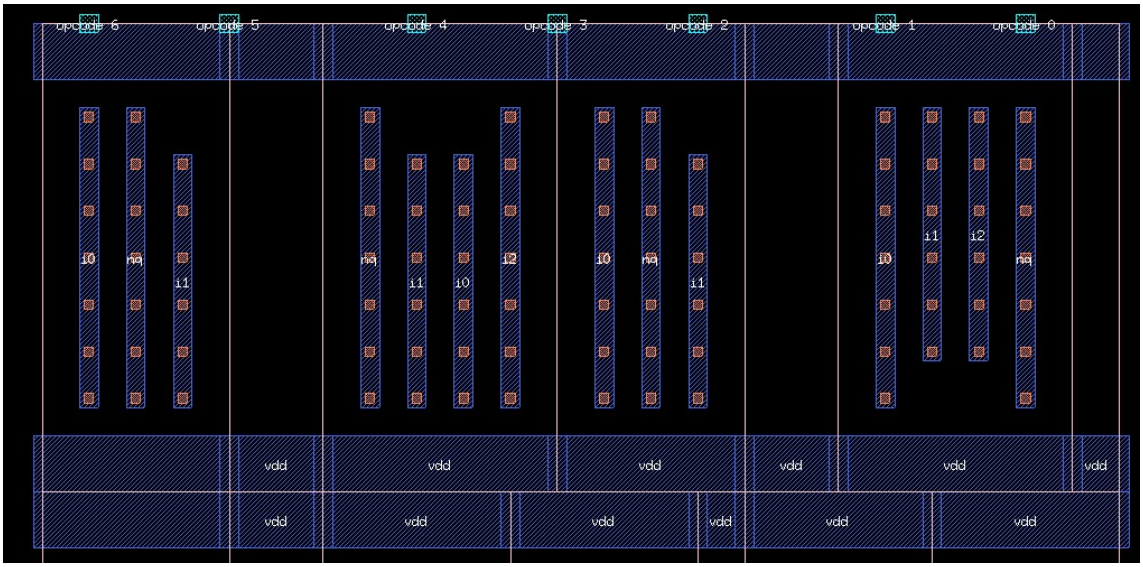
- Place and route layout design



CONTROL Module:

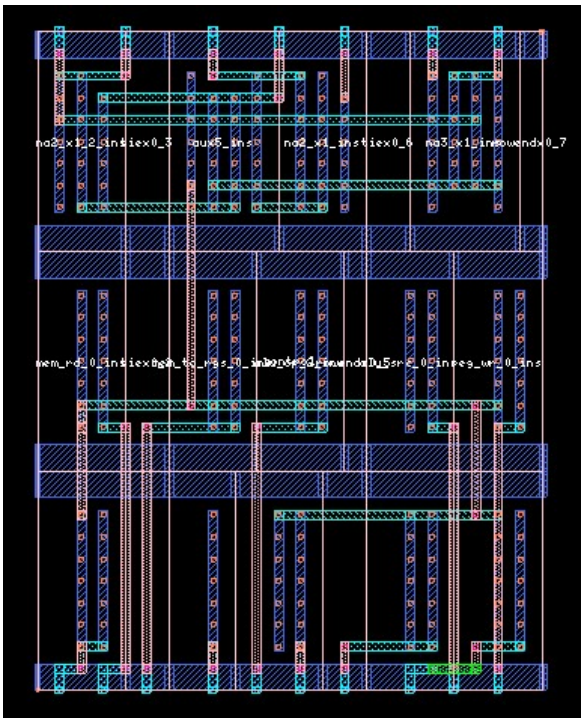
- The placed layout design seen in Graal:





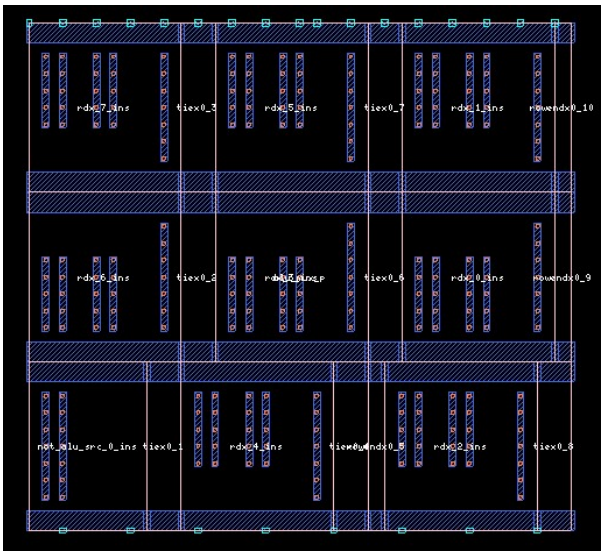
We can see the input ports on the top and output ports in the bottom of the layout.

- Place and route layout design



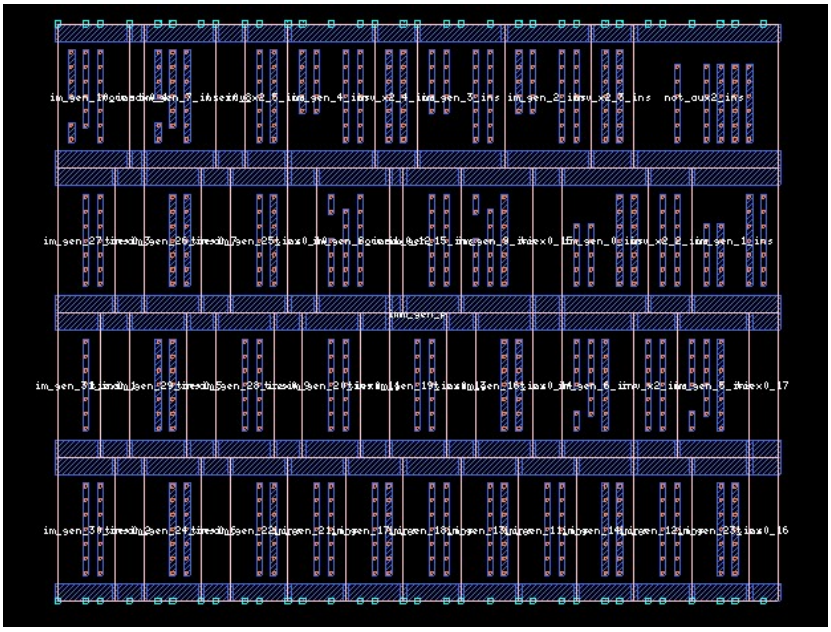
ALU MUX Module:

- The placed layout design seen in Graal:



IMMEDIATE GENERATION Module:

- The placed layout design seen in Graal:



Conclusion

We have designed and simulated a 32-bit RISC processor for a particular instruction set by using Xilinx 14.7 ISE design suite. In post implementation step we can see how many look up table, buffers, flipflops are actually being used among the available. The processor is consuming a total power of 0.124W targeting the Cora Z7 Zynq 7000s board.

And the other part of obtaining a layout design of ASIC is also done successfully using a CAD tool alliance which is an open source. We can see place and route layout design with specified number of metal layers that we require. This process further can be continued by fabricating the ASIC chip.

So, as a summary, towards the front-end design, we wrote the HDL codes for each module of our processor, and instantiated all of them using a cpu-top, then performed the behavioral simulation, logic synthesis, and implementation on an FPGA (CORA Z7). We went on to generate bitstream for the same, and tested the functionality on the board, which is indicated by a LED called “done”.
Towards back-end design, we generated the gate-level netlists for our design after converting them into another HDL, then we proceeded to Boolean minimization, binding on gates, optimization on networks and mapped the netlist design with ASIC cell libraries (sxlib in Alliance). The next step is placement of design, and routing of our design. All the simulation reports and schematics are presented in the report, and also the summary of post-synthesis and post-implementation utilization, power consumption, and timing analysis of our design.

References

- How-to-design-a-RISC-V-processor by Shirish Bahirat
- Alliance tool
- Coriolis
- YoSysHQ
- Single Cycle RISC-V Micro Architecture Processor and its FPGA Prototype by Don Kurian Dennis, Ayushi Priyam, Sukhpreet Singh Virk, Sajal Agrawal, Tanuj Sharma, Arijit Mondal and Kailash Chandra Ray Indian Institute of Technology Patna.
- Design and Application of RISC Processor by Mohammad Zaid, Prof. Pervez Mustajab
- A very simple 8-bit RISC processor for FPGA by S. de Pablo, J.A. Cebrián, L.C. Herrero, A.B. Rey