

# 32bit RISC-V Single Cycle Processor

Sanidhya Saxena, DESE, Indian Institute of Science, Bangalore-560012

## I INTRODUCTION

The RISC-V (Reduced Instruction Set Computer - V) architecture is an open-source instruction set architecture (ISA) that is designed to be simple, scalable, and modular. The architecture is particularly known for its flexibility, allowing implementations ranging from small microcontrollers to high-performance processors. In this report, we will focus on the design and implementation of a 16-bit multi-cycle RISC-V processor, which balances the complexity of hardware design with the performance of the processor.

## II INSTRUCTION SET ARCHITECTURE

There are a further two variants of the instruction formats (B/J) based on the handling of immediates. The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits ( $\text{imm}[10:1]$ ) and sign bit stay in fixed positions, while the lowest bit in S format ( $\text{inst}[7]$ ) encodes a high-order bit in B format.

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

Figure 1: Base Instruction Set

### III FPGA WRAPPER ARCHITECTURE

We have designed a FPGA Wrapper to test the design on the Fpga Basys-3 board. The logic on top contains the increment switch which is used to increment the ADDRESS and we get the content of DataMemory on the board. To get the result (Greatest of them), we have muxed a switch on to that. Hence if the switch is 1, we'll get the maximum number and if we make switch as 0, we can increment the address and check what is there in our data memory.

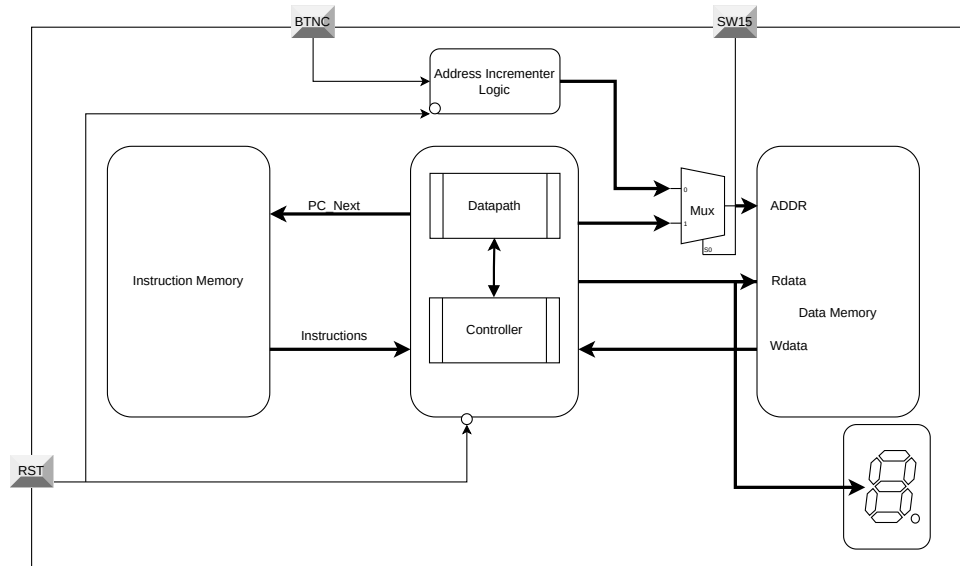


Figure 2: FPGA Wrapper

For Push button switches, we have used the debounce circuitry to protect it from unwanted switching. We have used a 2-DFF synchronisation for the bTnC Switch on FPGA.

For controlling the seven segment display we have used a BCD decoder that will display the result on Seven segment display.

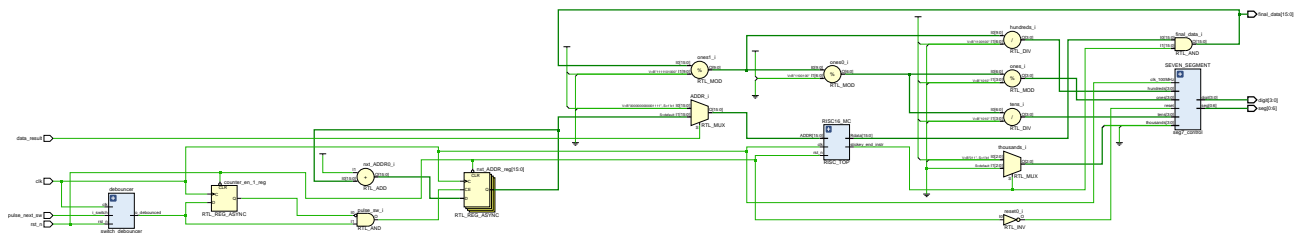


Figure 3: Schematic

:

## IV MICROARCHITECTURE

A 32-bit single-cycle RISC-V datapath executes one instruction per clock cycle. In a single-cycle design, every stage of instruction execution—fetch, decode, execute, memory access, and write-back—happens in one clock cycle.

**Instruction Fetch:** Instruction is fetched from instruction memory by the current program counter and is used for decoding the control signals, register file read and immediate generation.

**Instruction Decode:** Decoder is used to generate various control signals for each type of instruction like isLUI, isAUIPC, isJAL, isJALR, isLOAD, isSTORE, isBranch etc. on the basis of 7 bit opcode and is given as a select line of various multiplexers. The ALU Decoder is used to generate the AluControl[3:0] for ALU which is then used to perform various functions inside the ALU. The immediate assembly also happens in this stage.

**Instruction Execute:** Current instruction is executed in the ALU based on AluControl and the output of ALU is either used as an address to the data memory during load or store instructions. For R-type instructions ALU computes the result and stores it back to the register file based on the destination RF address. It is also used by the conditional branch instructions to calculate various flags like isZero, SLT (Set less than), SGT(Set Greater than) etc.

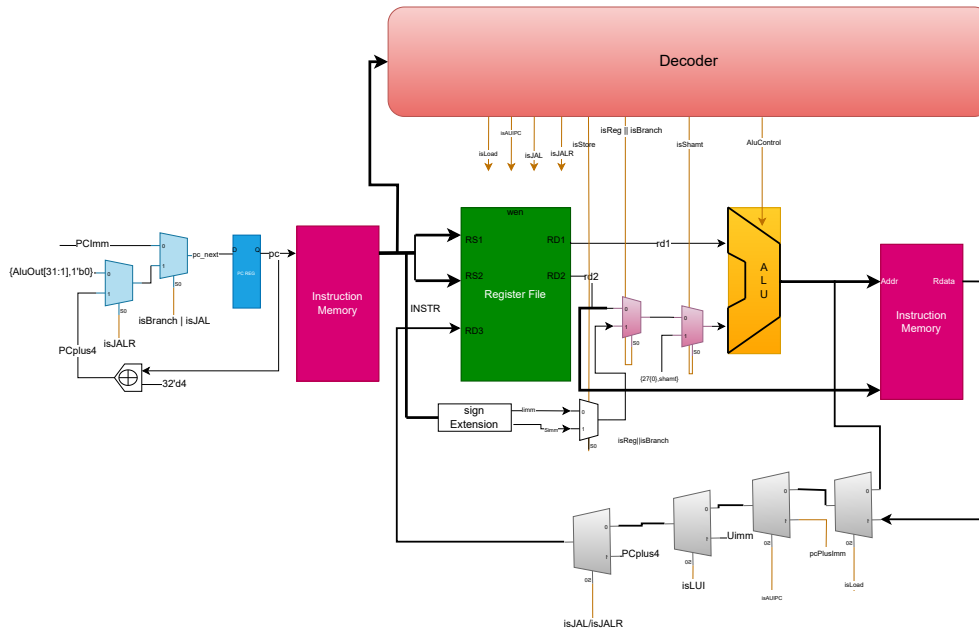


Figure 4: Microarchitecture of Processor , Reference: Haris and Haris

**Memory Stage:** Aluout from the ALU is used to access the data memory during load and store instructions. The memory is byte accessible and all the accesses are aligned.

**Write Back Stage:** The result of data memory is written back into Register file during load operation and mux selects between data from Aluout (ALU) or Rdata (Memory) and writes it into register file with the help of regWrite control signal generated by the decoder.

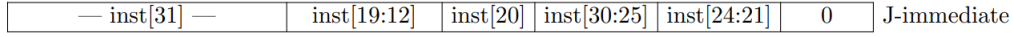


Figure 5: Jump immediate formation

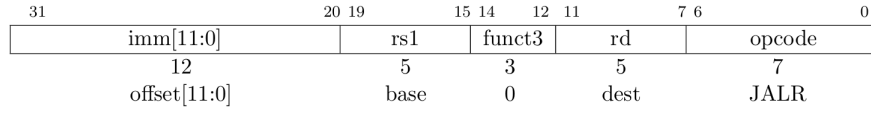


Figure 7: JALR Instruction

#### 4.1 PC formation

Since we have multiple instructions that operates on PC, the datapath for each instruction is designed.

**Unconditional Jumps:** The **jump and link (JAL)** instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a  $\pm 1\text{MiB}$  range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

$$PC_{target} \leftarrow PC + Jimm[19 : 0]$$

**Conditional Branch Instruction:** Immediate generation for branch is of I type. All branch instructions used the B-type instruction format. 12-bit B-immediate encodes the signed offset in multiple of 2bytes. The offset is sign-extended and added to the current PC to give the target address.

$$PC_{target} \leftarrow PC + Bimm[11 : 0]$$

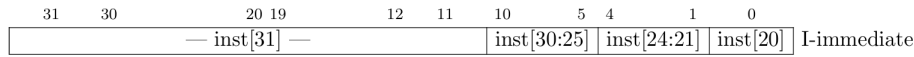


Figure 6: Branch Immediate Formation

Since both the JAL and BRANCH used the relative PC addressing, we have to add the PCImm to the current PC to get the target PC. Hence a MUX is used as shown in figure 4. if JAL/BRANCH occurs, then PC is added to the PCImm formed for Branch(Bimm) or Jump(Jimm) instruction and gives the target PC.

The indirect jump instruction **JALR (jump and link register)** uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd.

**JAL/JALR:** Hence in case of both JAL and JALR, we need to save (PC+4) in the destination register, hence the WriteData of the Register file has a MUX which intersects the path coming from the data memory and parallelly writes the value of PCplus4 in the register file.

The JALR instruction was defined to enable a two-instruction sequence to jump anywhere in a 32-bit absolute address range. A LUI instruction can first load rs1 with the upper 20 bits of a target address, then JALR can add in the lower bits. Similarly, AUIPC then JALR can jump anywhere in a 32-bit pc-relative address range.

**LUI/AUIPC:** Hence for LUI we need to store the Uimm to the destination register and in the case of AUipc we need to store the PCplusimm value into the destination register.

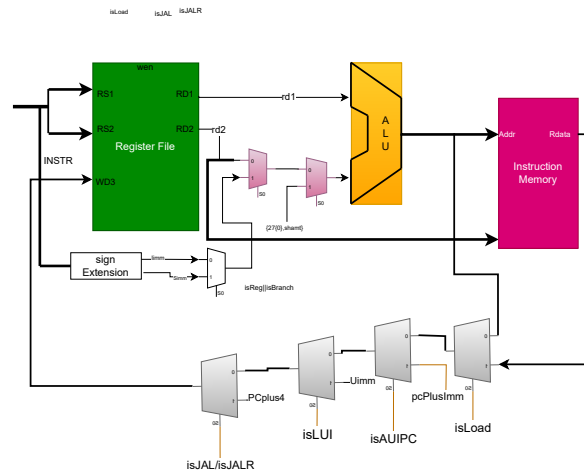


Figure 8: Write Data path for Register file

The above figure shows all the path used for storing PC for linking and jumping to a subroutine.

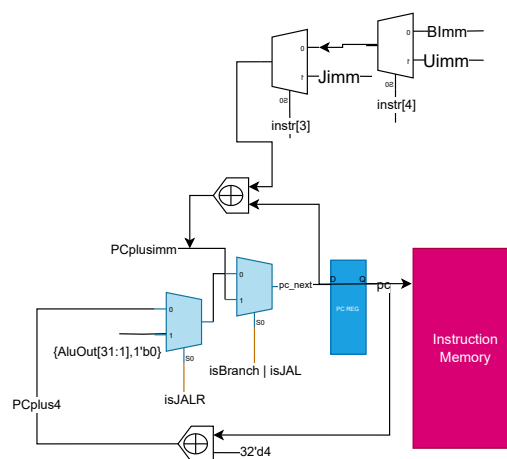


Figure 9: PC Immediate generation

## 4.2 Memory Alignment

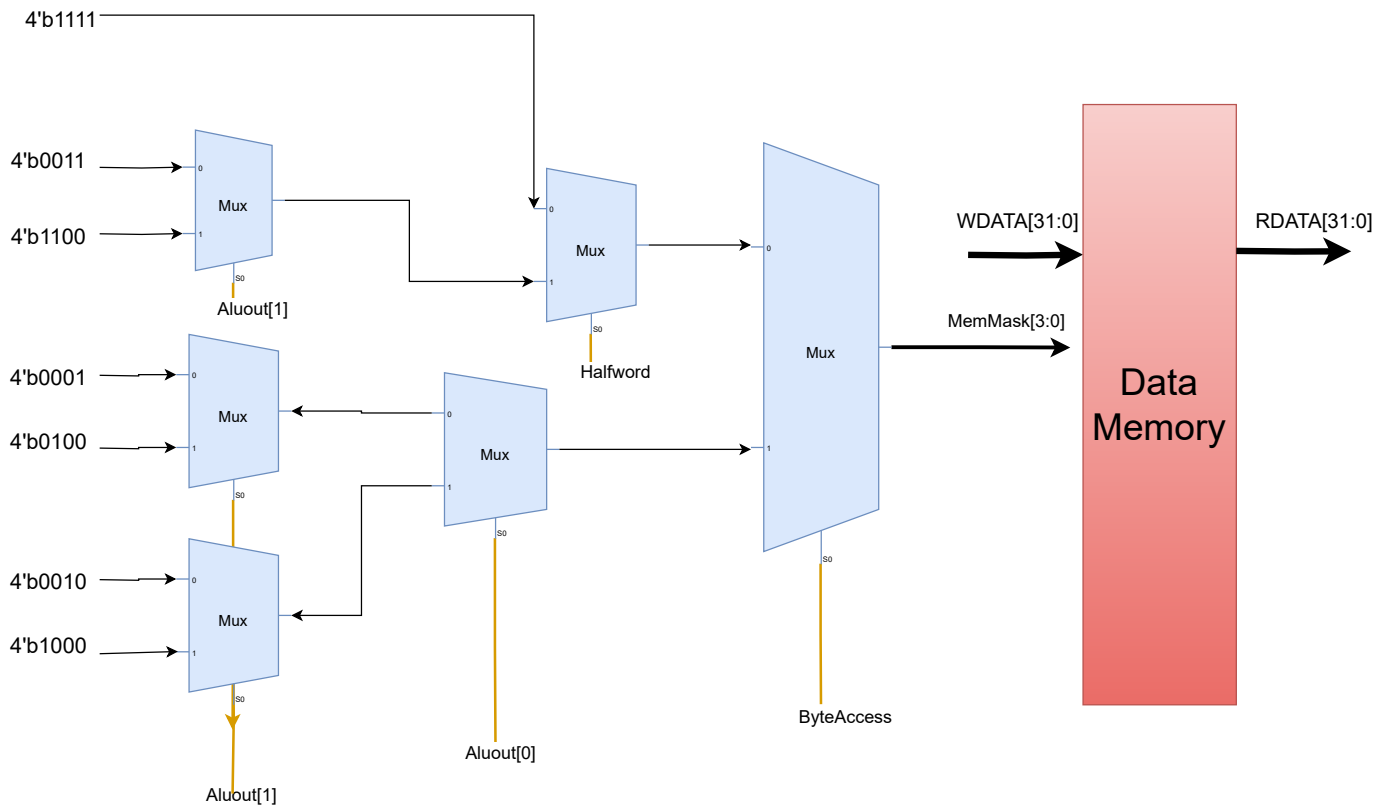


Figure 10: Memory Write Alignment

All the data writes to the memory during store operations are aligned. Word is aligned to the 32bit boundary, half-word is aligned to the 16-bit boundary and byte can be placed anywhere in the memory. Hence a MemMask[3:0] logic is designed to unmask only those bits in the memory who's value has to be changed and rest will remain unchanged, this goes into the byte enable of data memory.

- Word (32-bit): Aligned on a 4-byte boundary (addresses that are multiples of 4).
- Half-word (16-bit): Aligned on a 2-byte boundary (addresses that are multiples of 2).
- Byte (8-bit): Can be stored at any address (no alignment requirement).

The Wdata[31:0] is coming from the RS2 and is assembled based on the lower 2 lsb bits of the Aluout.

- SW(store word): The memory address must be a multiple of 4 (4-byte aligned). Valid Address: 0x0, 0x4, 0x8, 0xC
- sh(store half-word): The memory address must be a multiple of 2 (2-byte aligned). Valid Address: 0x00, 0x02, 0x04, 0x06, 0x08
- sb (store byte): Stores an 8-bit byte from a register to memory.

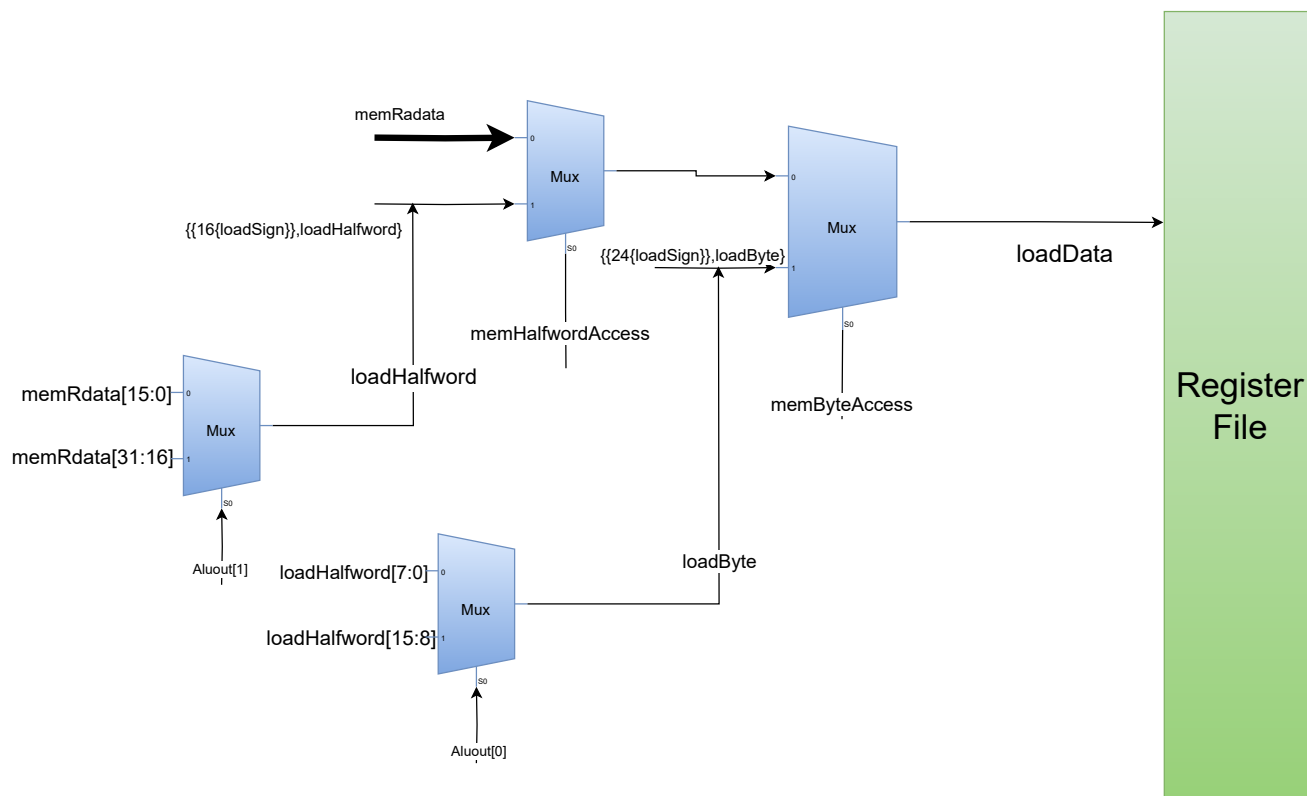


Figure 11: Memory Read Alignment

When loading data from memory, proper alignment allows the memory controller to fetch data efficiently, typically in one clock cycle for aligned accesses. Misaligned accesses may cause significant performance degradation or exceptions, depending on the system's memory handling.

Based on instruction

## V LEVEL 1 SCHEMATIC

### 5.1 RISC Top

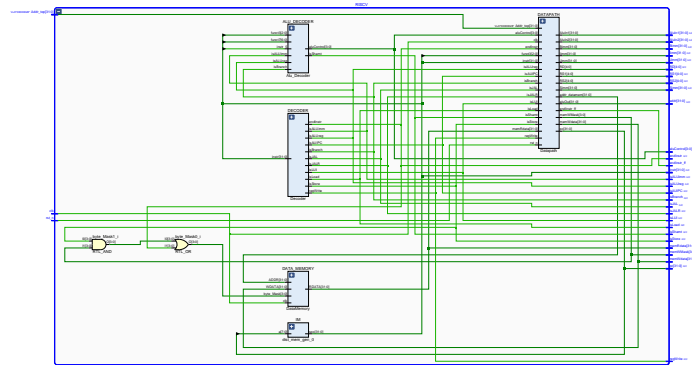


Figure 12: RISC Top

### 5.2 RISC Datapath Single Cycle

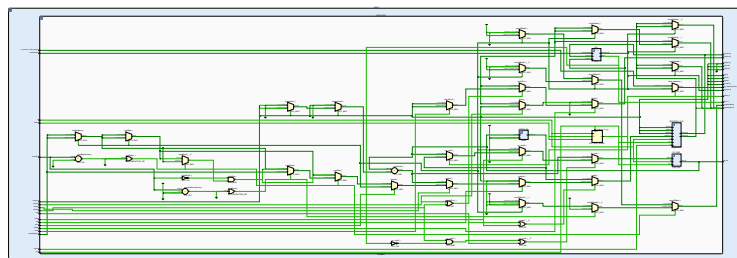


Figure 13: RISC Datapath

### 5.3 REGISTER FILE

This Register File has 8, 16bit register since we have 3 bits to address these registers. We have 2 read port and one write port. Reads are combinational is cotrolled using a mux at the output which is selected based on two address RS1 and RS2. Write is sequential and the register is enabled using a decoder based on the input write address coming from the controller.



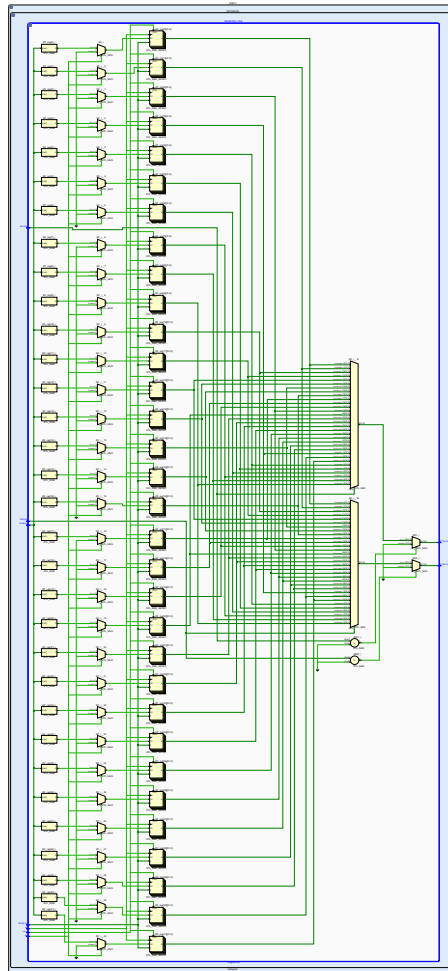


Figure 14: Register File

#### 5.4 ALU

We have a ALU opcode of 2 bits, hence we can perform 4 different type of operations inside the ALU, but since we only require arithmetic ADD and SUB, we have mapped them to ALuop=00 and ALuop=01 respectively.

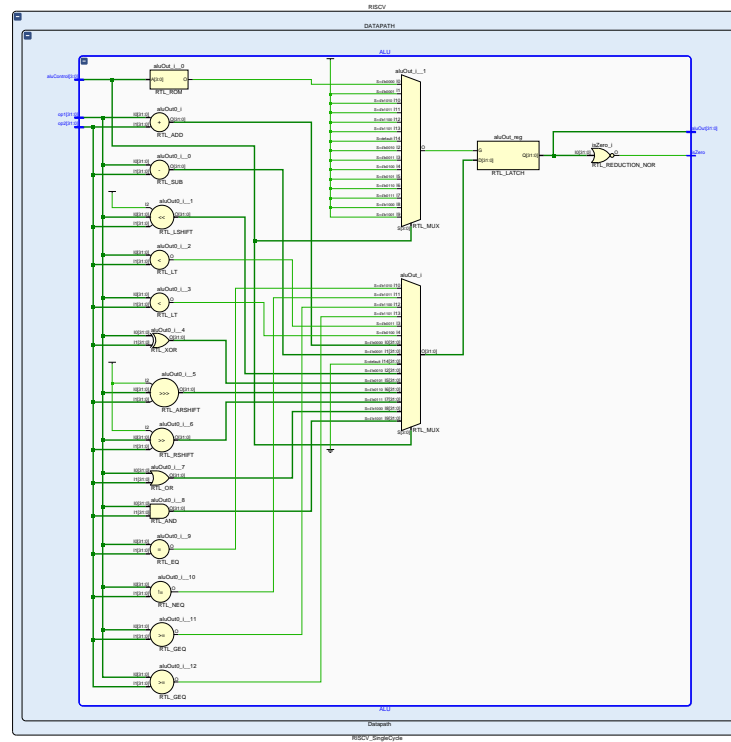


Figure 15: ALU Design

## 5.5 ALU Decoder

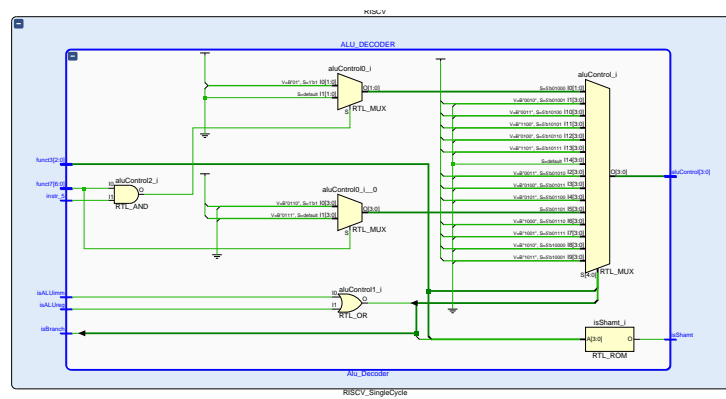


Figure 16: ALU Decoder Design

## 5.6 Decoder

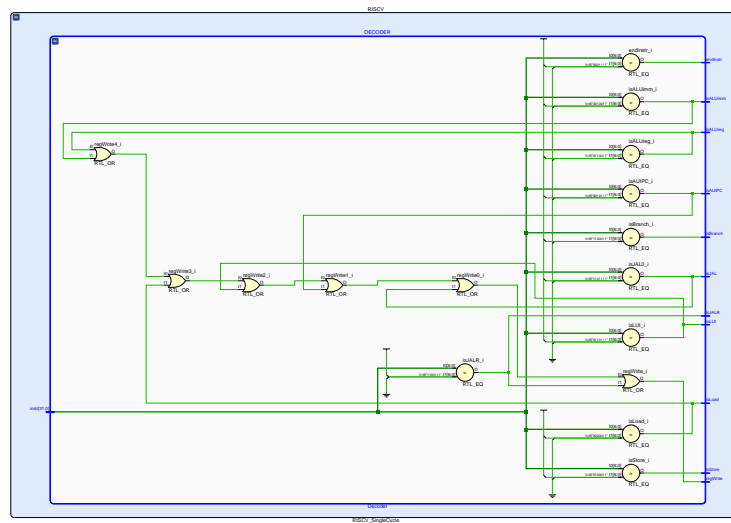


Figure 17: Decoder

## 5.7 Memories

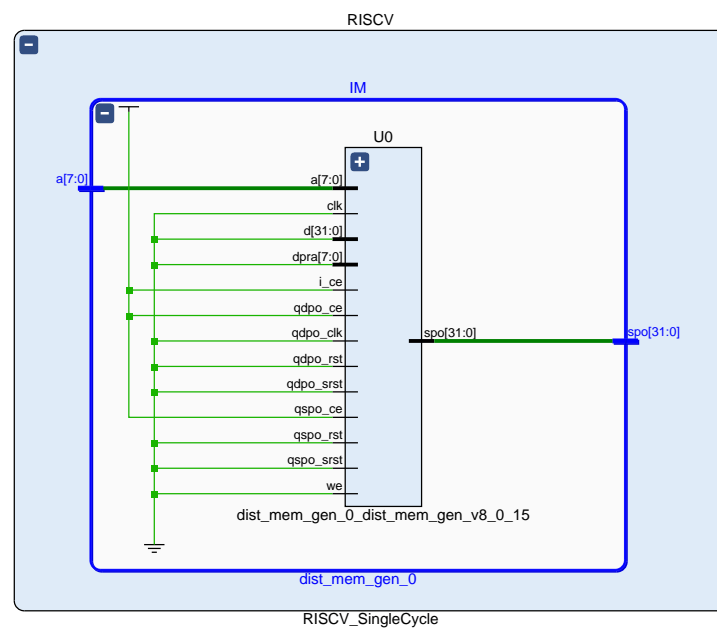


Figure 18: Instruction Memory

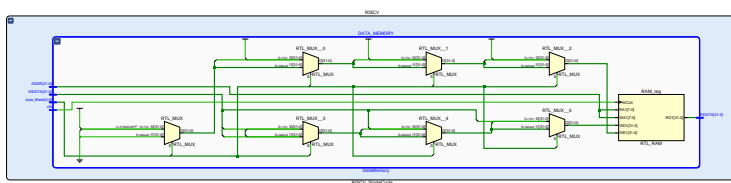


Figure 19: Data Memory

## VI FUNCTIONAL SIMULATION

### 6.1 Directed Simulation

A Self-Checking testbench is designed to test our processor. We have stored instruction in the instruction memory using a coe file and is then run sequentially on the processor. The final output of registers is stored in the memory and is then compared with the golden values.

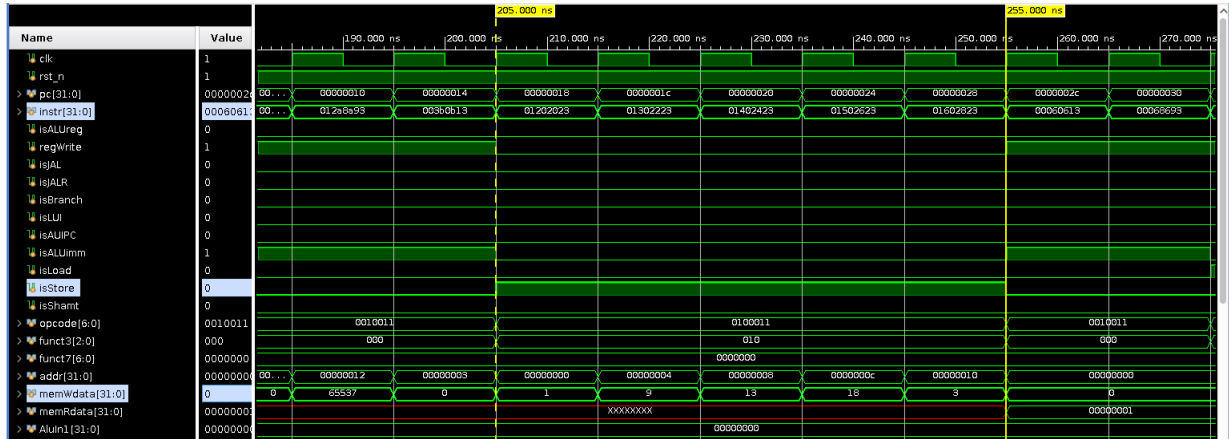


Figure 20: Memory Write Instructions

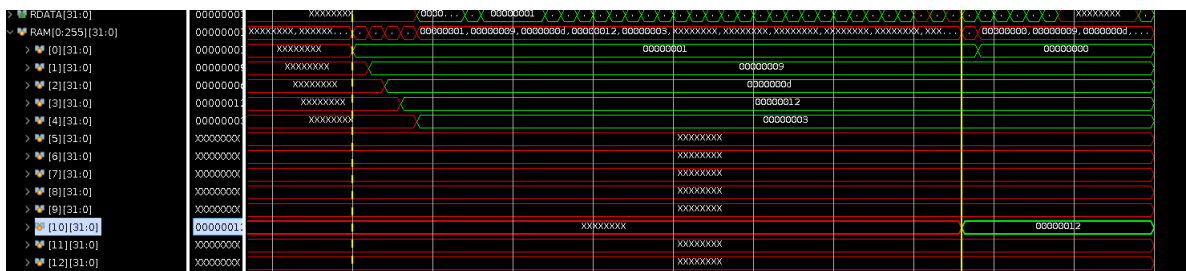


Figure 21: Memory Write Content after Execution

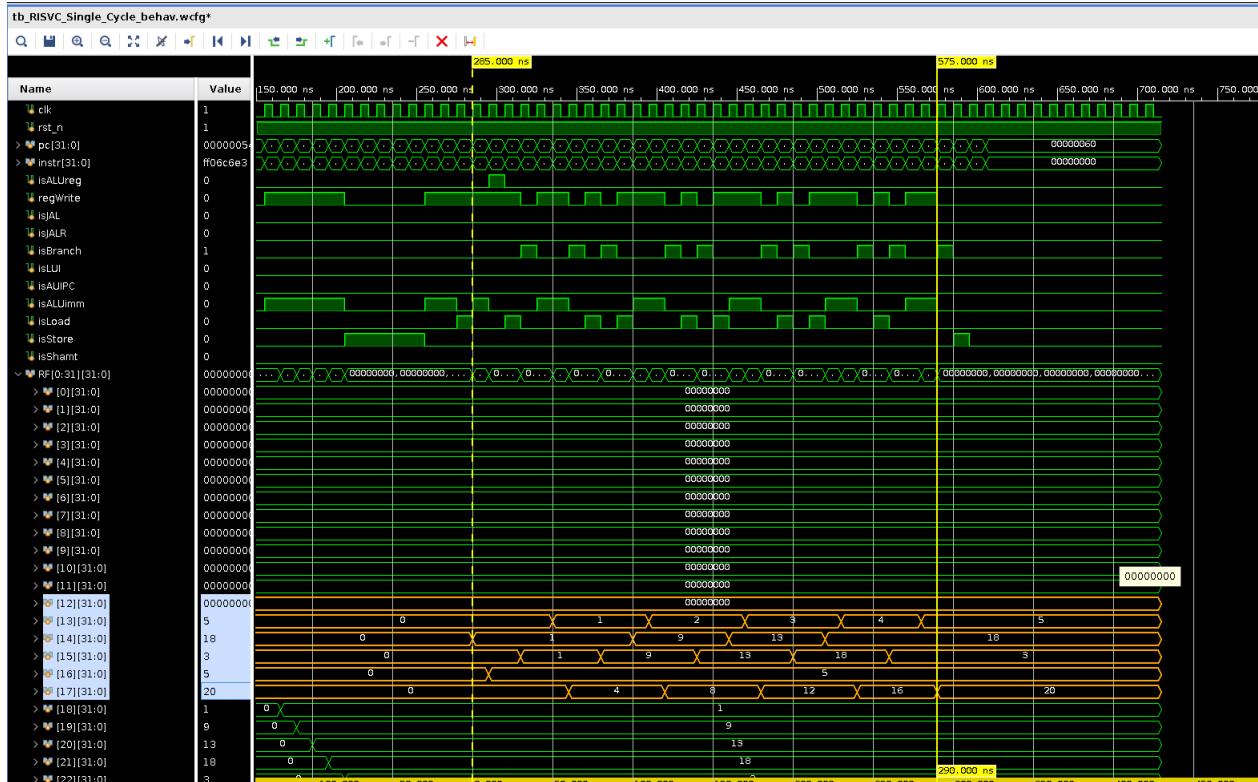


Figure 22: Register Operations

A Self-Checking testbench is designed to test our processor. We have stored instruction in the instruction memory using a coe file and is then run sequentially on the processor. The final output of registers is stored in the memory and is then compared with the golden values.

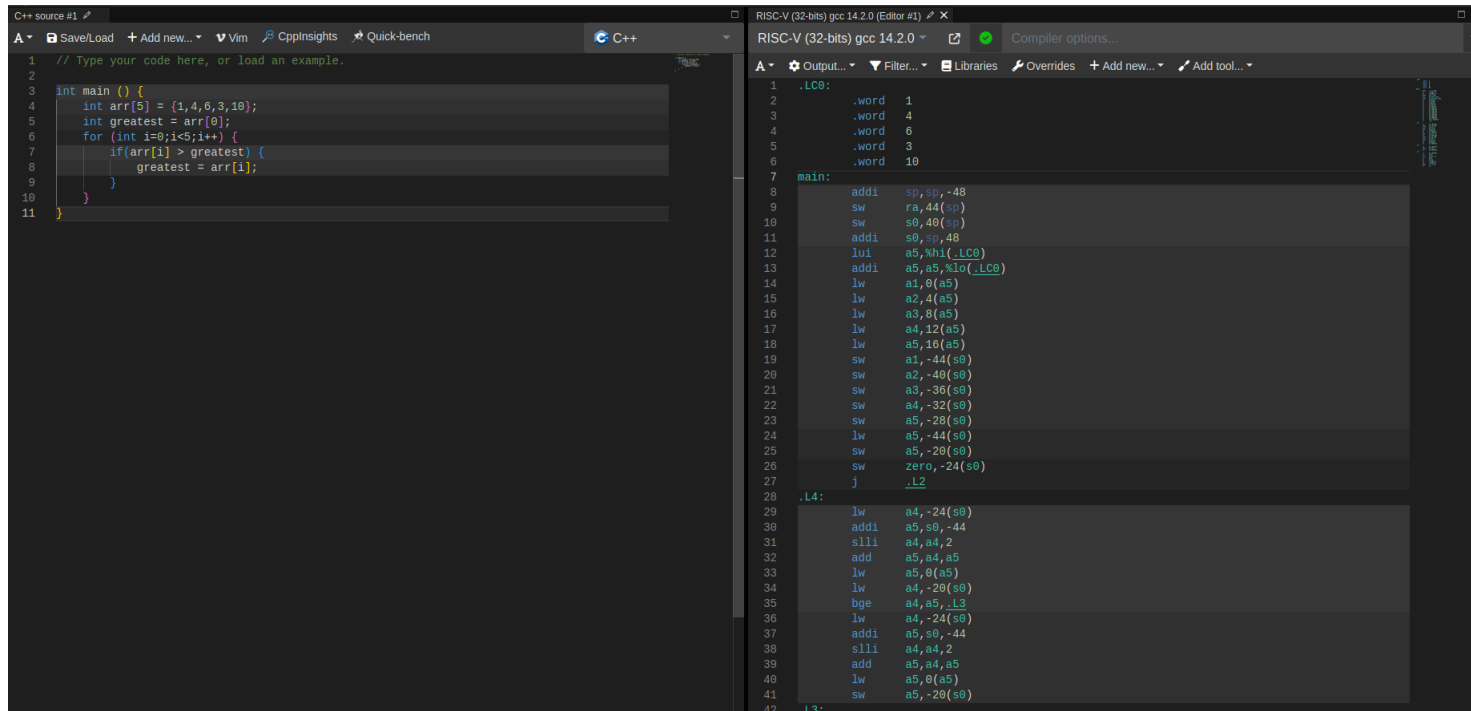
In this we ran a code to find out the greatest element in an array of 10 random integers. The .C code was compiled into assembly and then the machine instruction(binary) were given to the Instruction memory and 10 random integers are on the data memory.

### Algorithm 1 C code

```
/* C Code to find out the Greatest */

int main() {
    int arr[10] = {0,1,8,15,4,16,23,9,10,25};
    int greatest = arr[0];
    for(int i =0;i<10;i++) {
        if(arr[i]>greatest) greatest = arr[i];
    }
}
```

Assembly generated for above C code is as follows:



```

1 // Type your code here, or load an example.
2
3 int main () {
4     int arr[5] = {1,4,6,3,10};
5     int greatest = arr[0];
6     for (int i=0;i<5;i++) {
7         if(arr[i] > greatest) {
8             greatest = arr[i];
9         }
10    }
11 }

```

```

1 .LC0:
2     .word 1
3     .word 4
4     .word 6
5     .word 3
6     .word 10
7 main:
8     addi sp,sp,-48
9     sw ra,44(sp)
10    sw s0,40(sp)
11    addi s0,sp,48
12    lui a5,%hi(.LC0)
13    addi a5,a5,%lo(.LC0)
14    lw a1,0(a5)
15    lw a2,4(a5)
16    lw a3,8(a5)
17    lw a4,12(a5)
18    lw a5,16(a5)
19    sw a1,-44(s0)
20    sw a2,-40(s0)
21    sw a3,-36(s0)
22    sw a4,-32(s0)
23    sw a5,-28(s0)
24    lw a5,-44(s0)
25    sw a5,-20(s0)
26    sw zero,-24(s0)
27    j .L2
28 .L4:
29    lw a4,-24(s0)
30    addi a5,s0,-44
31    slli a4,a4,2
32    add a5,a4,a5
33    lw a5,0(a5)
34    lw a4,-20(s0)
35    bge a4,a5,.L3
36    lw a4,-24(s0)
37    addi a5,s0,-44
38    slli a4,a4,2
39    add a5,a4,a5
40    lw a5,0(a5)
41    sw a5,-20(s0)
42 .L3:

```

Figure 23: Greatest Number RISC-V assembly code

## Algorithm 2 Assembly code

```

main:
    addi s2,s2,0x1
    addi s3,s3,0x9
    addi s4,s4,0xd
    addi s5,s5,0x12
    addi s6,s6,0x3
    sw s2,0(zero)
    sw s3,4(zero)
    sw s4,8(zero)
    sw s5,12(zero)
    sw s6,16(zero)
    addi a2,a2,0
    addi a3,a3,0
    lw a4,0(a2)
    addi a6,a6,0x5
    add a7,a2,a3
.loop:
    lw a5,0(a7)
    bge a4,a5,.addr1
    lw a4,0(a7)
.addr1:
    addi a3,a3,1
    addi a7,a7,0x4
    blt a3,a6,.loop
    sw a4,40(zero)

```

## VII TIMING SUMMARY

Clock Frequency = 50 Mhz

The timing is met with a Positive setup slack of 0.399ns

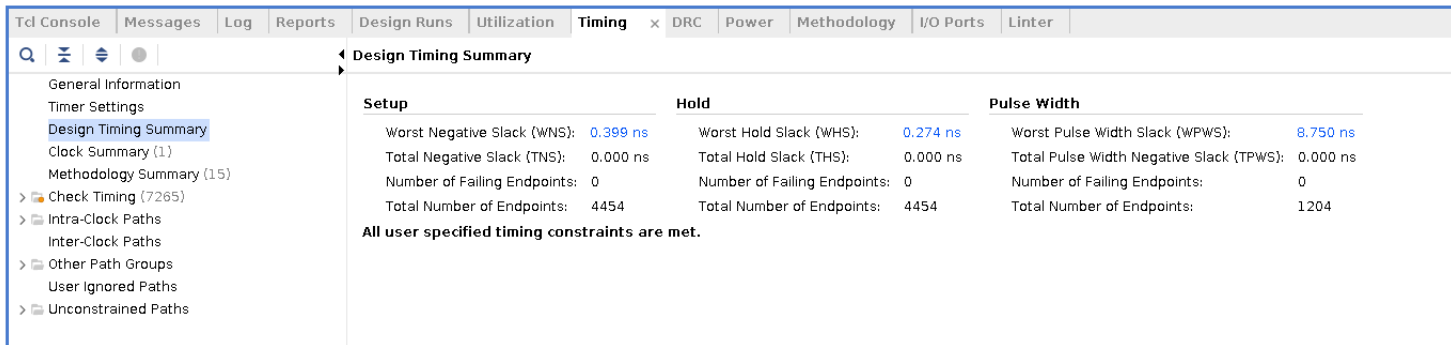


Figure 24: Timing

## VIII RESOURCE UTILISATION

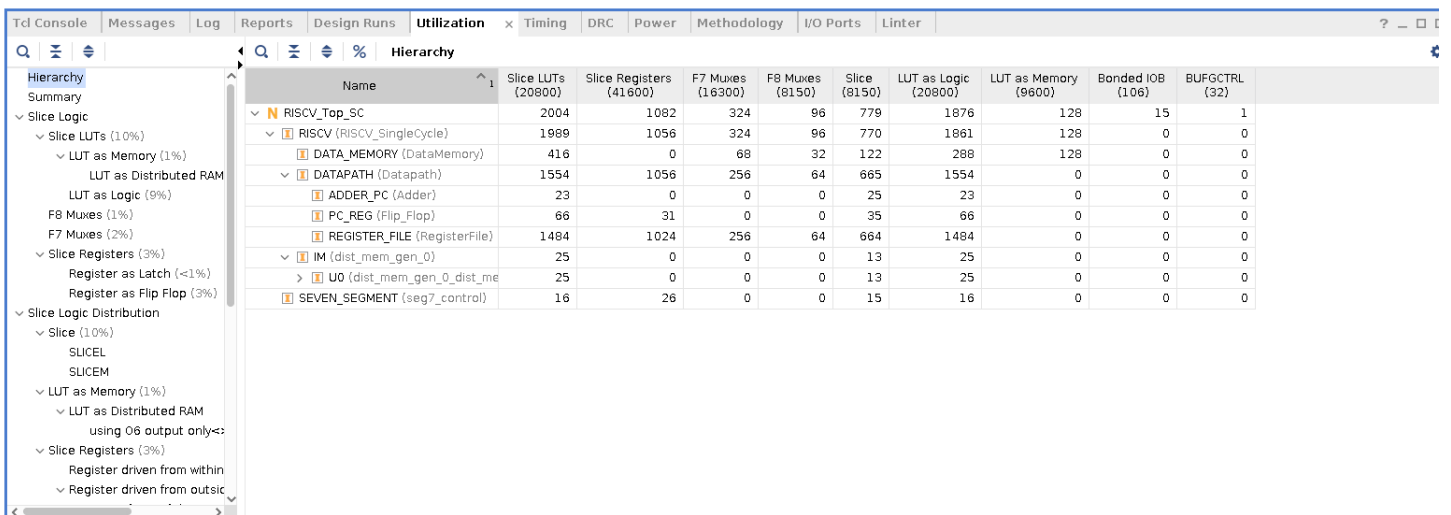


Figure 25: Resource

## IX POWER CONSUMPTION

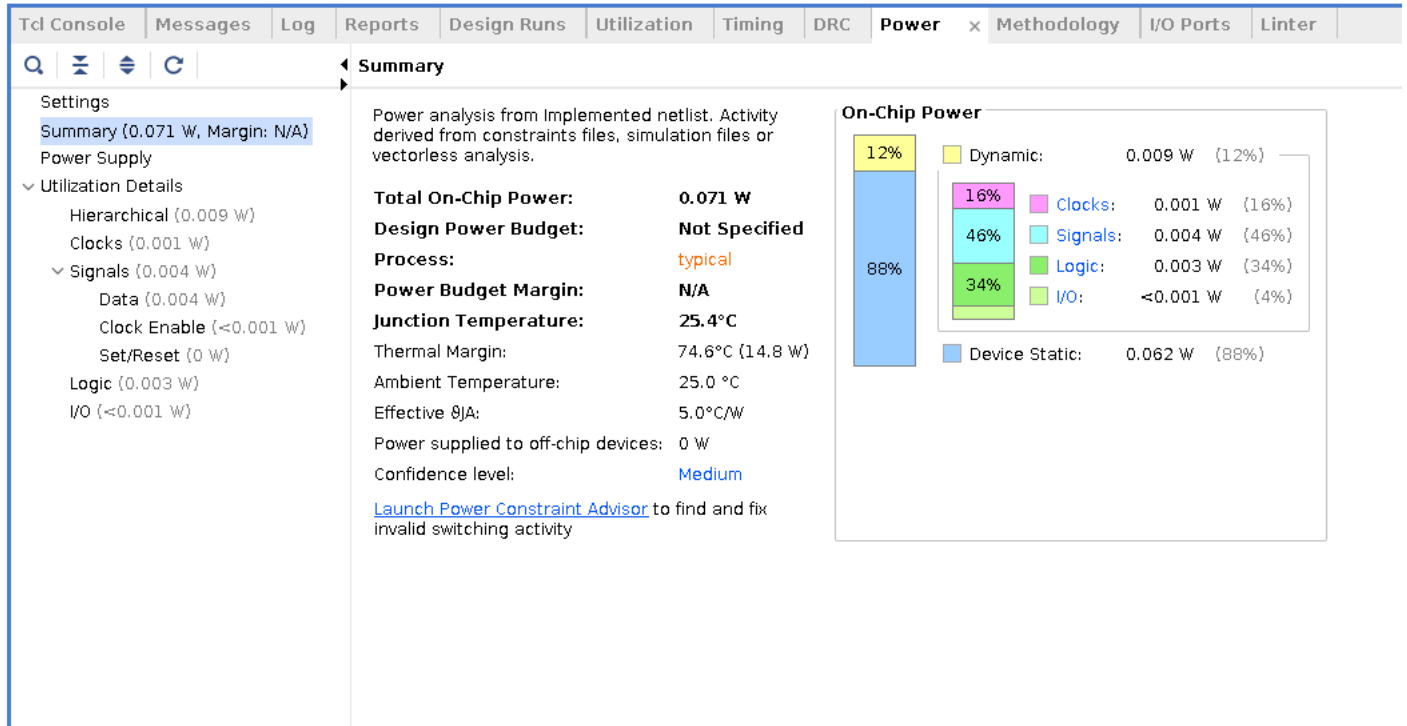


Figure 26: Power Consumption

## X ADVANTAGES OF MULTICYCLE DESIGN

1. *Simplicity in Design and Implementation:* Since each instruction is completed in one clock cycle, the control unit is simpler to design. There's no need to handle hazards or stalls as in pipelined architectures.
2. *Predictable Timing:* In a single-cycle design, the processor does not face data hazards, control hazards, or the need for complex techniques like forwarding or branch prediction. Every instruction takes the same time to execute.
3. *Reduced Power Consumption:* Single-cycle designs often operate at lower clock frequencies compared to pipelined designs. This can lead to reduced power consumption, especially in systems where power efficiency is critical.

## XI CHALLENGES AND CONSIDERATIONS

1. The clock cycle must be long enough to accommodate the slowest instruction, such as load/store instructions that require memory access. This means that faster instructions (e.g., addition or logic operations) also take as long as the slowest ones, wasting time and lowering efficiency.
2. As the complexity of instructions increases, the overall clock cycle time must be further extended, leading to significantly degraded performance as more complex instructions are added.
3. *Low Overall Clock Speed:* The need to execute all instructions in a single clock cycle requires the clock cycle to be long enough to handle the most time-consuming instruction. This leads to a slower clock speed, which limits the performance of the CPU.
4. *Waste of Hardware Resources:* Since only one instruction is executed at a time, many functional units (e.g., arithmetic logic unit, memory unit) may be idle for a large portion of the cycle. In more complex designs like pipelining, these units could be utilized simultaneously, increasing hardware efficiency.

## XII CONCLUSION

While a single-cycle design is not the most efficient for high-performance computing, its simplicity, predictability, and ease of implementation make it a good fit for specific applications where simplicity and low power consumption are



more important than maximizing throughput.

## REFERENCES

- [1] Digital Design and Computer Architecture by David Money Harris and Sarah L. Harris
- [2] Onur Mutlu Lectures
- [3] Computer Organisation and Design RISC-V Edition by DAVID.A.PATTERSON, JOHN.L.HENNESY