32bit RISCV Pipeline 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 (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

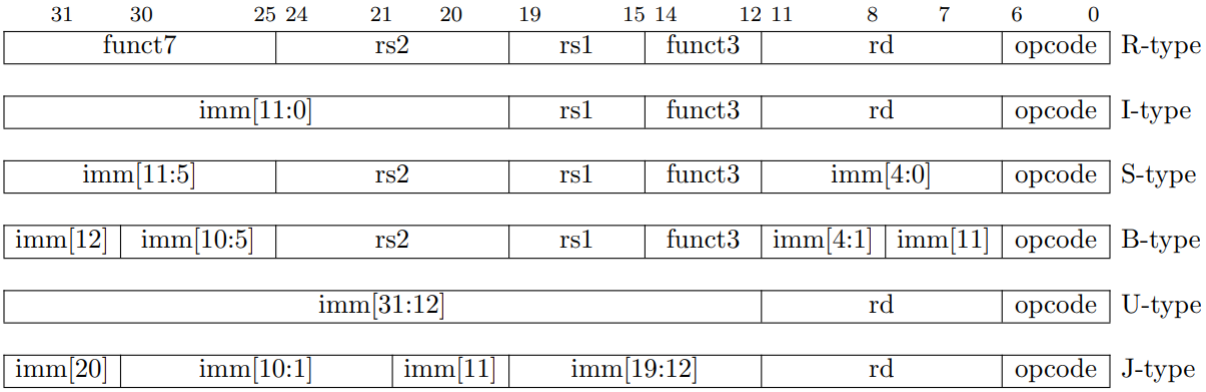


Figure 1: Base Instruction Set

Instruction	Opcodes
LUI	0110111
AUIPC	0010111
JAL	1101111
JALR	1100111
Branch	1100011
Load	0000011
Store	0100011
I-type	0010011
R-Type	0110011

Table 1: Opcodes

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.

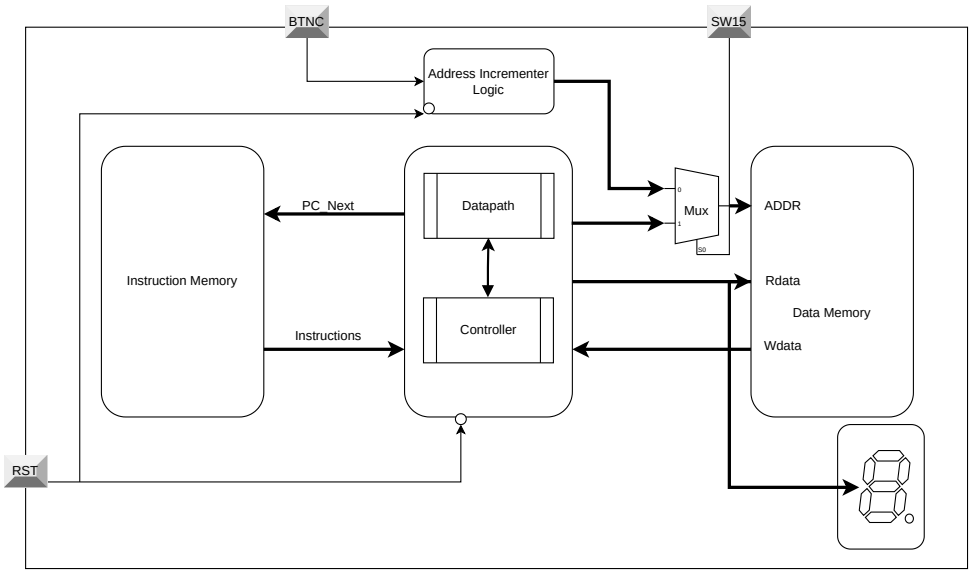


Figure 2: FPGA Wrapper

For Push button switches, we have used the debounce circuitry to pretect 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.

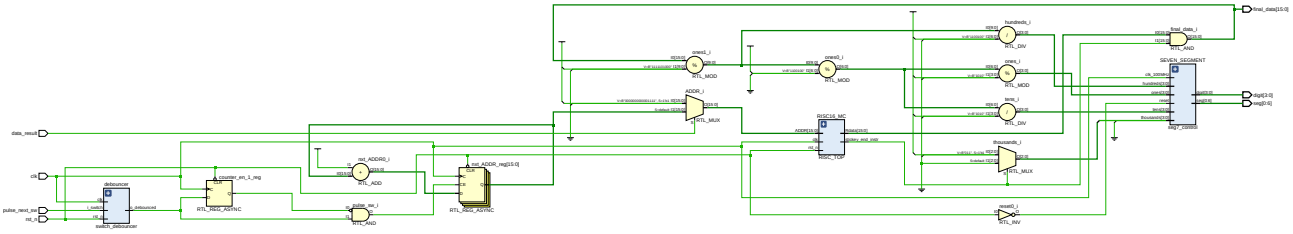


Figure 3: Schematic

:

IV MicroArchitecture

The 5-stage pipelined in-order processor is a classic CPU design approach used to improve processing efficiency by dividing instruction execution into multiple stages. This approach allows multiple instructions to be processed simultaneously, with each instruction at a different stage in the pipeline. The 5-stage in-order pipeline, also known as the RISC pipeline, is frequently used in simplified or educational CPU designs, making it an ideal model for explaining pipelining concepts.

Instruction Fetch: The instruction fetch stage retrieves the next instruction from memory, typically using a program counter (PC) to keep track of the instruction address. The program counter is incremented to point to the next instruction, preparing it for the next cycle.

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 compuet 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.

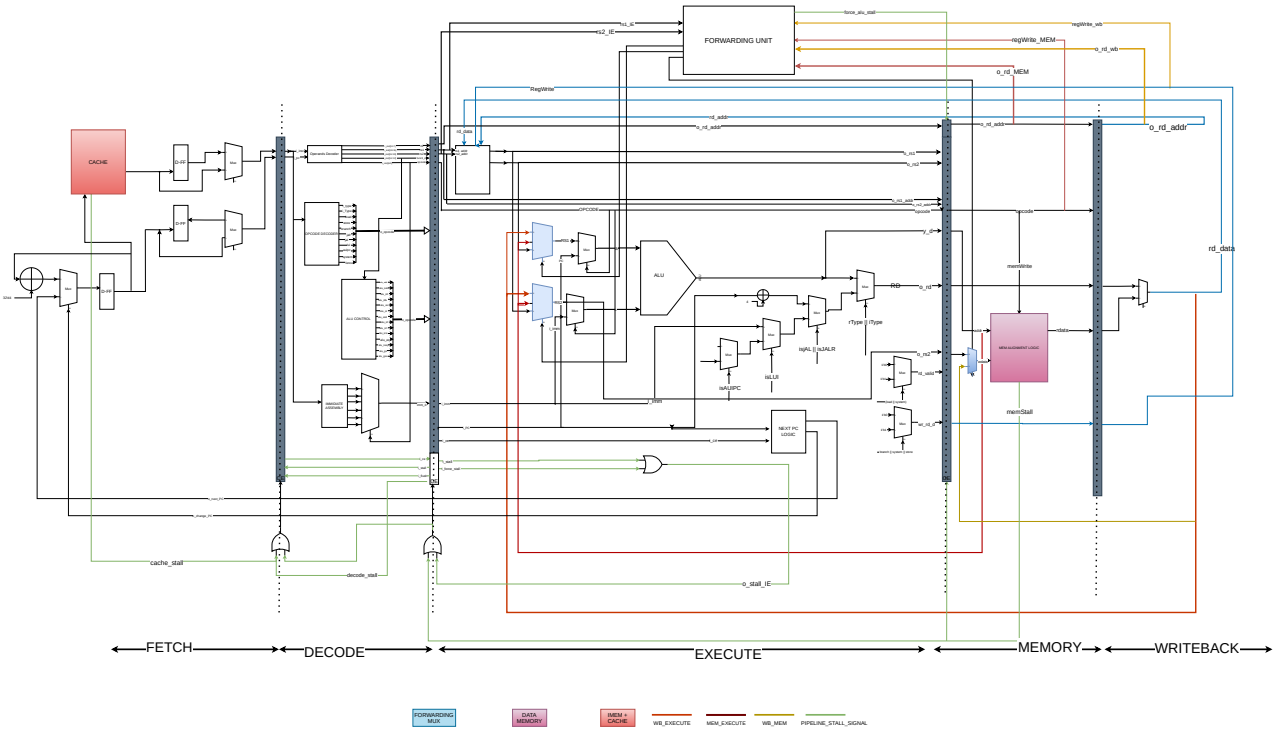


Figure 4: Microarchitecture of Processor , Reference: Haris and Haris

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.

Stall Logic: Whenever the next stage is stalled, it stalls it previous stages. For Ex. If MEM Stage is stalled due to some memory operation, it will stall all the previous stages i.e FETCH, DEOCDE and EXECUTE. In case of

Branch and Jump instruction we have to flush the previous 2 stages of the pipeline. When we see a LOAD instruction, we need to stall the IE Stage since the result of load is available at the WB stage and hence IE stage has to wait.

But if LOAD is followed by STORE, then we forward the value of load from WB stage to the MEM Stage. This is illustrated below in the timing diagram.

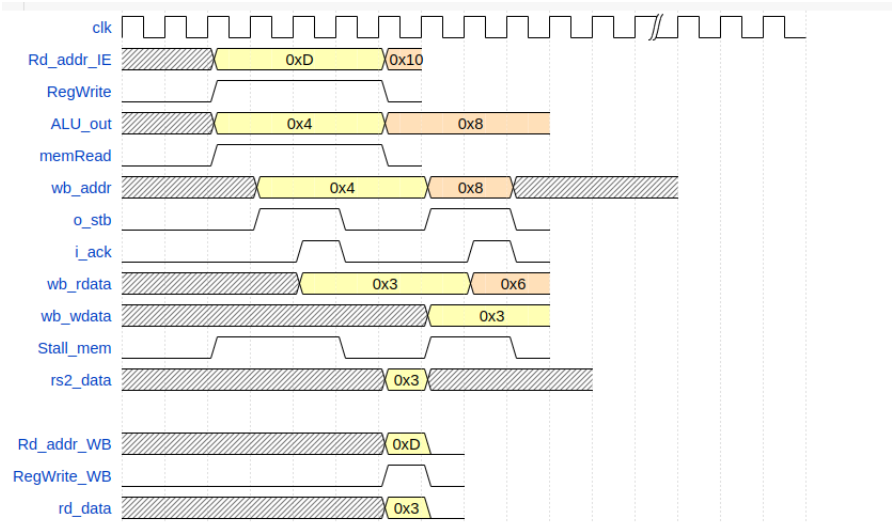


Figure 5: Jump immediate formation

Following is the program order:

lw x13,4(x0)
sw x13,8(x0)

Hence the value of ***Rd_ addr*** is 0xD and ***regWrite*** is high in the mem stage. ALU_out gives us the destination addr (0x4 in our case) along with the control signal memRead. The Wishbone controller sends the read request(***wb_ addr***) to the memory along with the strobe signal ***(i_ stb***). Until we get the i_ack signal, Stall_mem signal is high indicating that instruction is still in the memory stage, and hence all the previous stages will be stalled. When we get the ***wb_ Rdata*** from the memory (0x3), the load moves on to the WB stage. Since the value of ***rs2_ addr*** in MEM stage is same as the ***Rd_ addr*** in WB stage, this data(0x3) is forwarded from WB stage to the mem stage in ***rs2_ data.***

4.1 Forwarding Paths

1. IE/MEM.memory –> ID/IE.Execute Stage: if((rs1_execute == rd_mem) && regWrite_mem && !memRead) then we forward the *Rd_ data* from *Memory stage* to the *Execute stage* as shown in figure 4.
2. MEM/WB.WriteBack –> ID/IE.Execute Stage: if((rs1_execute == rd_wb) && regWrite_wb) then we forward the *Rd_ data* from *Writeback stage* to the *Execute stage* as shown in figure 4
3. MEM/WB.WriteBack –> IE/MEM.memory. if((rs2_mem == rd_wb) && memWrite) then we forward the *Rd_ data* from *Writeback stage* to *Memory Stage* as shown in figure 4

4.2 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 ±1MiB 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.

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

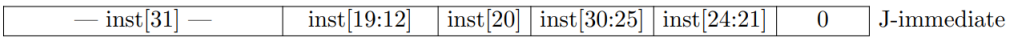


Figure 6: Jump immediate formation

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

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

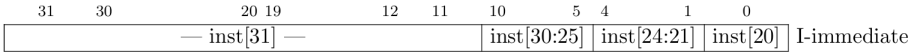


Figure 7: Branch Immediate Formation

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.

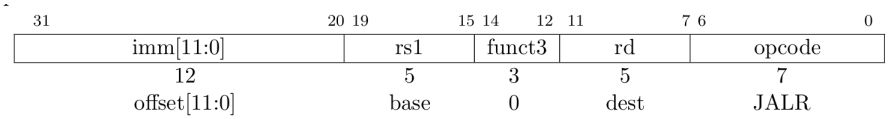


Figure 8: JALR Instruction

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 parallely 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.

V Level 1 Schematic

5.1 RISC Top RTL

Figure shows the top level RTL schematic of the design. Pipelined processor core is integrated with ICACHE, Data Memory and Instruction Memory. FPGA wrapper is also designed to test the processor on the board.

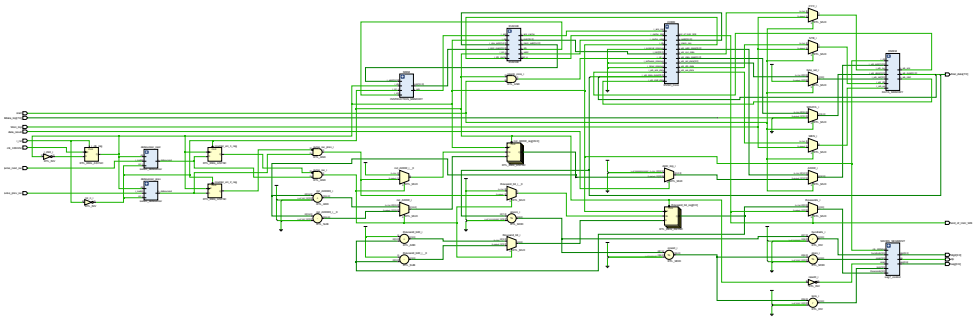


Figure 9: RISC Top

5.2 RISC Top Synthesized

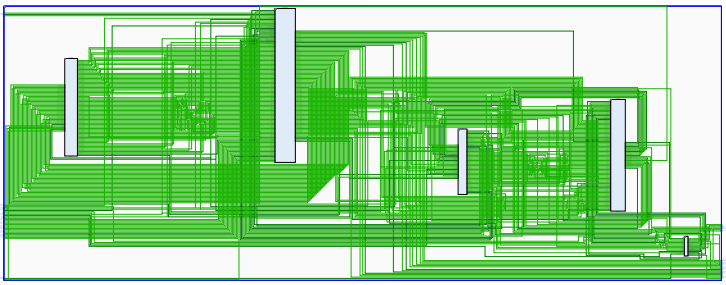


Figure 10: RISC Top Synthesized

5.3 ICACHE

For instructions we have the used the Direct Map cache organisation. Each instruction address maps to a single line in the cache. This structure is simple but prone to conflict misses, as only one memory address can map to each cache line. The Instruction Cache (I-Cache) is a dedicated cache memory that stores instructions for the CPU to execute, separate from the data cache, which stores data operands. The primary purpose of an instruction cache is to improve the instruction fetch rate by reducing the time needed to access frequently used instructions. Specifications are as follows:

Cache Parameter	Value	Bits Required
Cache Size	1KB	10
Main Memory Size	32KB	15
Block Size	16B (4 Words)	4
# of Sets	64	6
Tag bits + valid		5+1

Table 2: Cache Specification

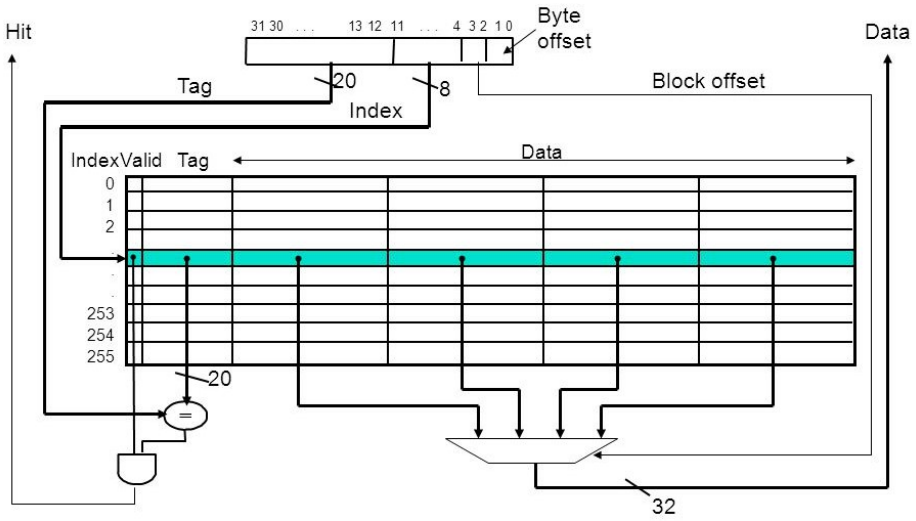


Figure 11: Direct Mapped Cache organisation

If we have a cache hit, then the requested instruction will be given back to the processor with a delay of 1 cycle. If the block is not present in the cache then the block has to be fetched from the main memory. In that case the processor FETCH stage will be stalled unitl the whole line (4 Instructions) are fetched from the memory.

Direct Mapped cache has a very low HIT time which is good for L1 cache, but Miss rates are high.

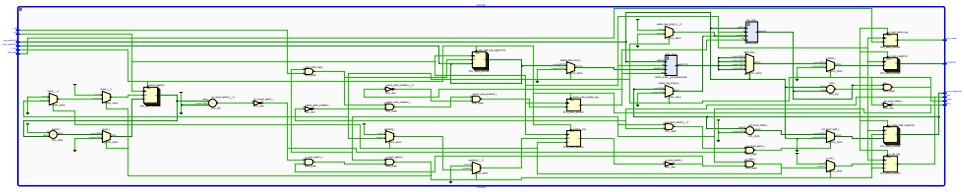


Figure 12: ICACHE Controller

5.4 RF Controller

This Register File has 32, 32 bit register since we have 5 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.

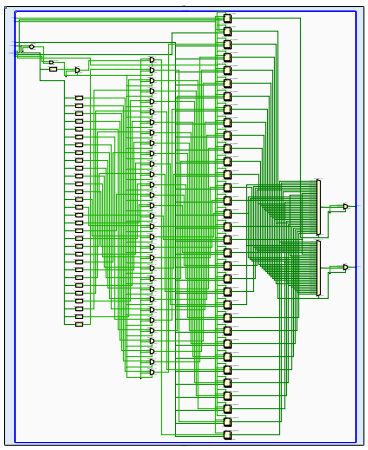


Figure 13: Register File Controller

5.5 Fetch Stage

FETCH stage fetches instruction from the memory and deliver it to the next stage. If memory/Cache is stalled, then the whole pipelined is stalled. The value of next PC in case of Branch/Jump comes from the IE stage. In our case we'll not stall the pipeline, but will flush the 2 instruction after the Branch instruction if the Branch is not taken. Since jump and call are always taken, we have to stall the pipeline.

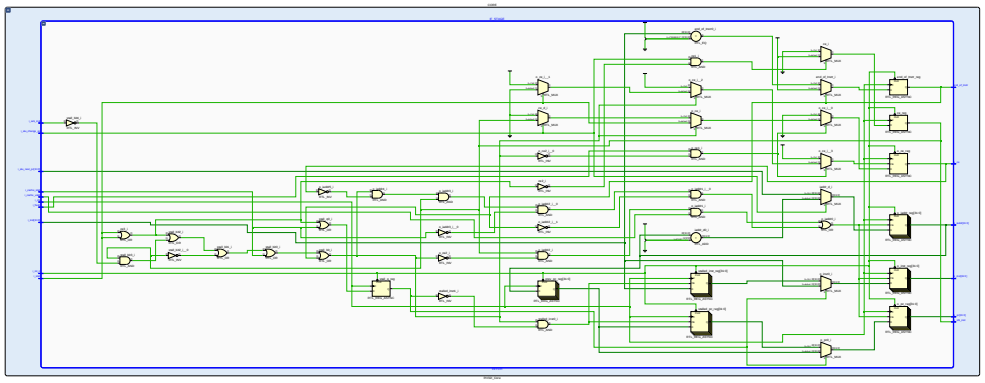


Figure 14: IF Stage

5.6 Decode Stage

DECODE Stage has mainly 2 blocks, One for Opcode decoding and generating control signals and another one for immidiate decoding. Immidiate decoding is given in below figure.

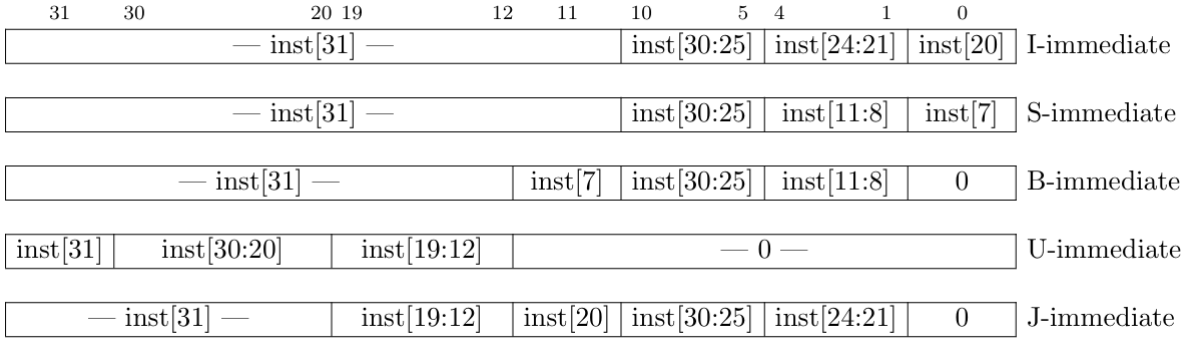


Figure 15: Immediate Decoding

ALU Function decoding logic is also there to inform ALU which operation needs to be performed.

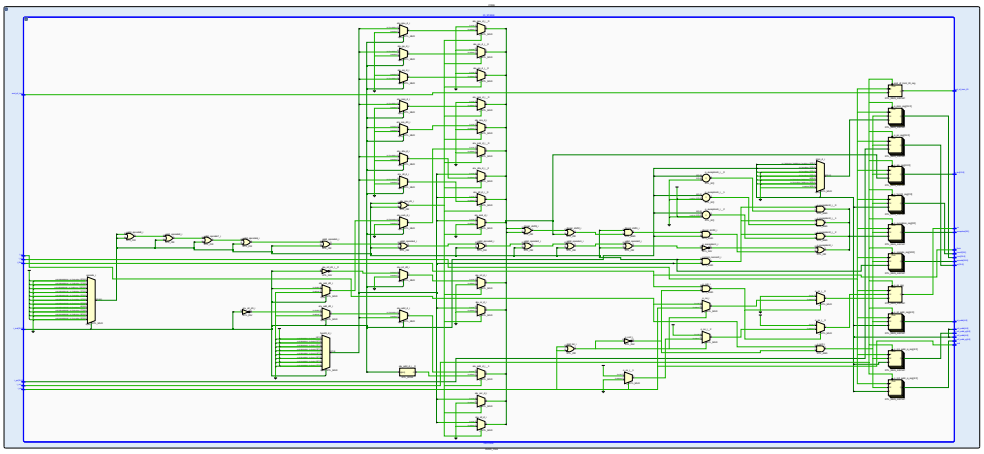


Figure 16: ID Stage

5.7 Execute Stage

This is the main functional stage of the processor pipeline, where we have all the integer Arithmetic and Logical operations.

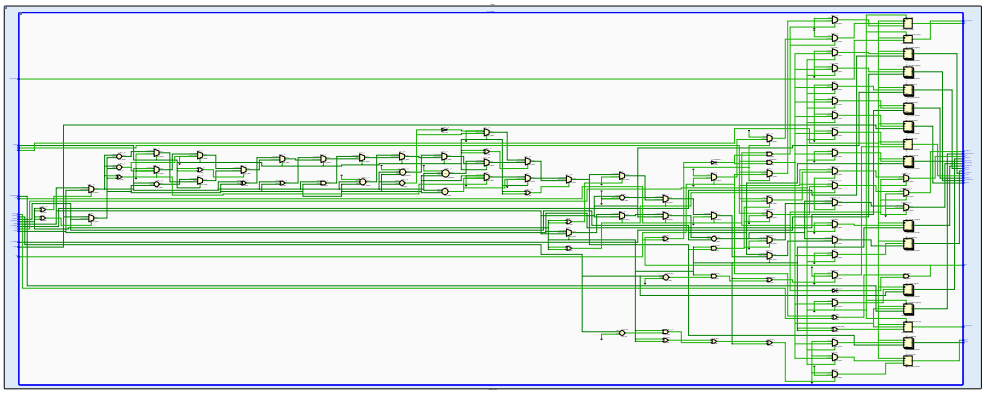


Figure 17: IE Stage

The Next PC generation also takes place here. Figure below shows the design of IE stage. For Branch/Jump, the imm is added to the PC coming from the ID/IE pipeline. For JALR the value of PC comes from the register file, hence the imm has to be added to the value of RS1.

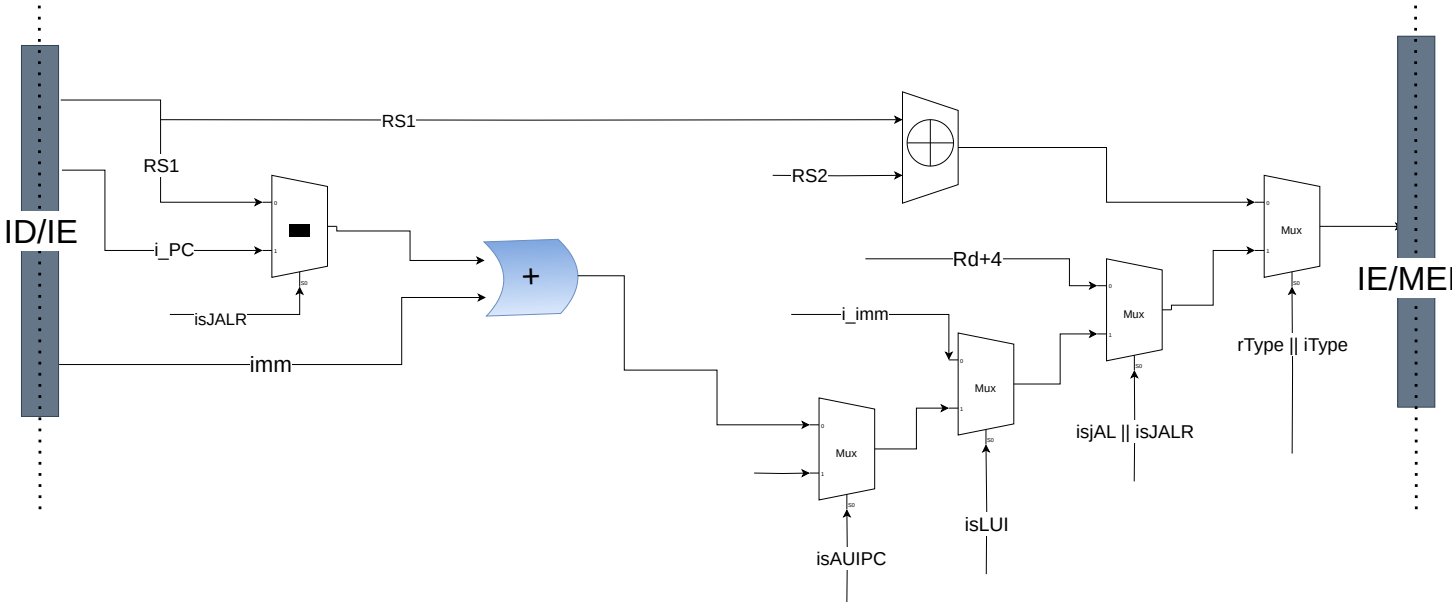


Figure 18: IE Stage

Rd is the data that needs to be stored in the destination register in Register File. For different instructions, the value of Rd changes. For R-type and I-type instructions, Rd is the result of ALU, for JAL/JALR we need to store PC+4 in the register file as a context/Return address, for LUI we have to store Uimm in the register fille. For AUIPC, we have to store PC + imm (Upped Immidiate) to the register file.

5.8 MEM Stage

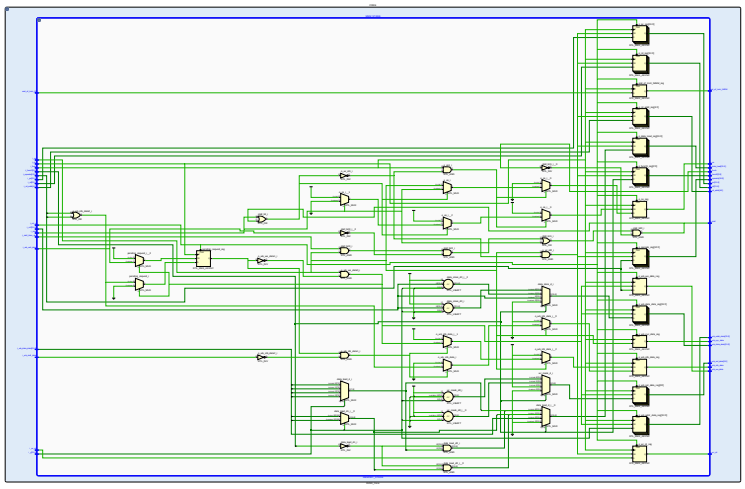


Figure 19: Mem Stage

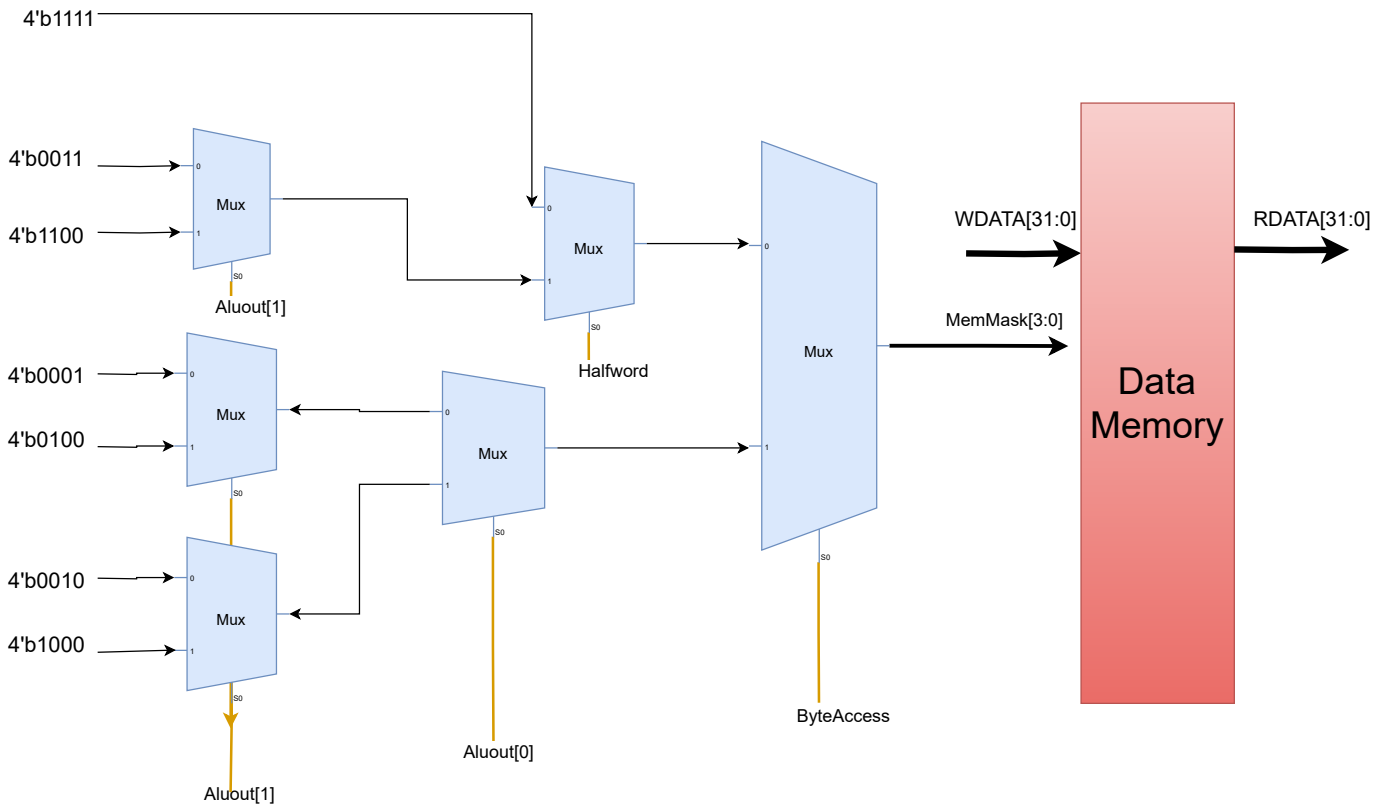


Figure 20: Memory Write Alignment

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.

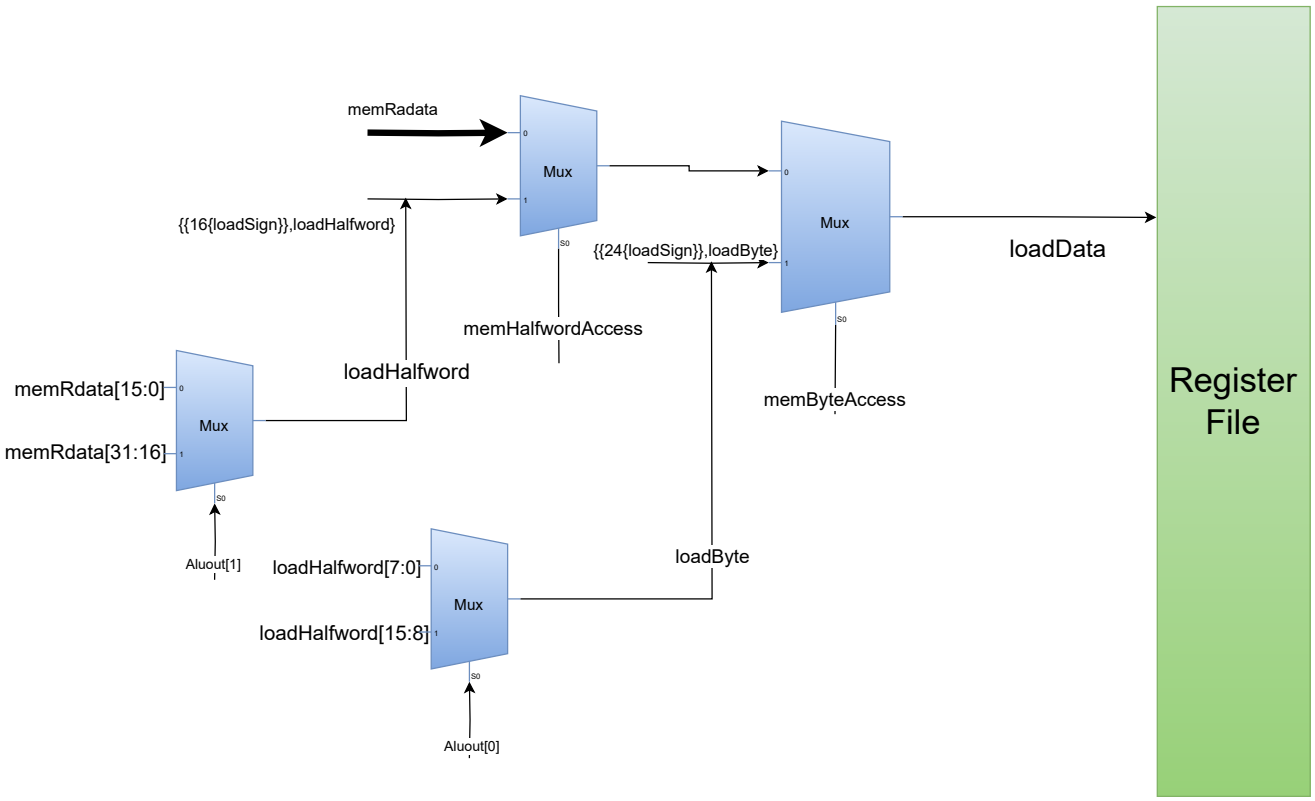


Figure 21: Memory Read Alignment

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

5.9 WB Stage

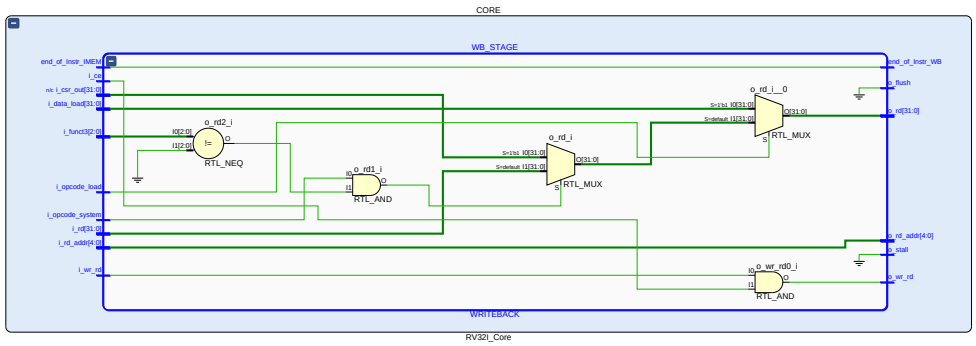


Figure 22: WB Stage

5.10 Instruction and Data memory

RISC-V follows a load-store architecture, which means it uses separate instructions to load data from memory into registers and to store data back to memory. This setup, along with the typical separation of instruction and data memory in modern RISC-V cores, is designed to enhance the processor's efficiency and simplify control logic.

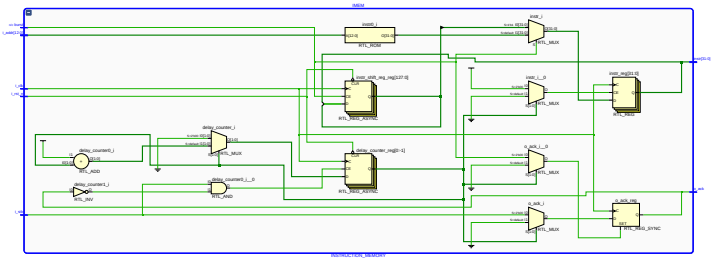


Figure 23: Instruction Memory

RISC-V instruction memory may support physical or virtual addressing (using a memory management unit or MMU) for larger systems. In simpler RISC-V implementations, however, instruction memory may operate without an MMU, accessing physical memory directly.

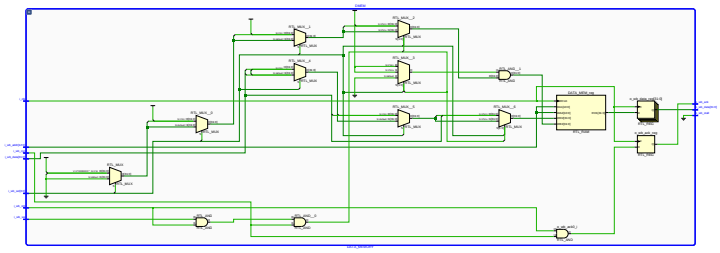


Figure 24: Data Memory

Data memory is accessed by load and store instructions, which transfer data between memory and CPU registers. Unlike instruction memory, data memory needs to support both read and write operations since data may be modified and updated during program execution. In RISC-V, data memory accesses must be aligned to the size of the data being accessed. For instance, a 32-bit (4-byte) load/store should be aligned to a 4-byte boundary.

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 procesor. The final output of registers is stored in the memory and is then compared with the golden values.

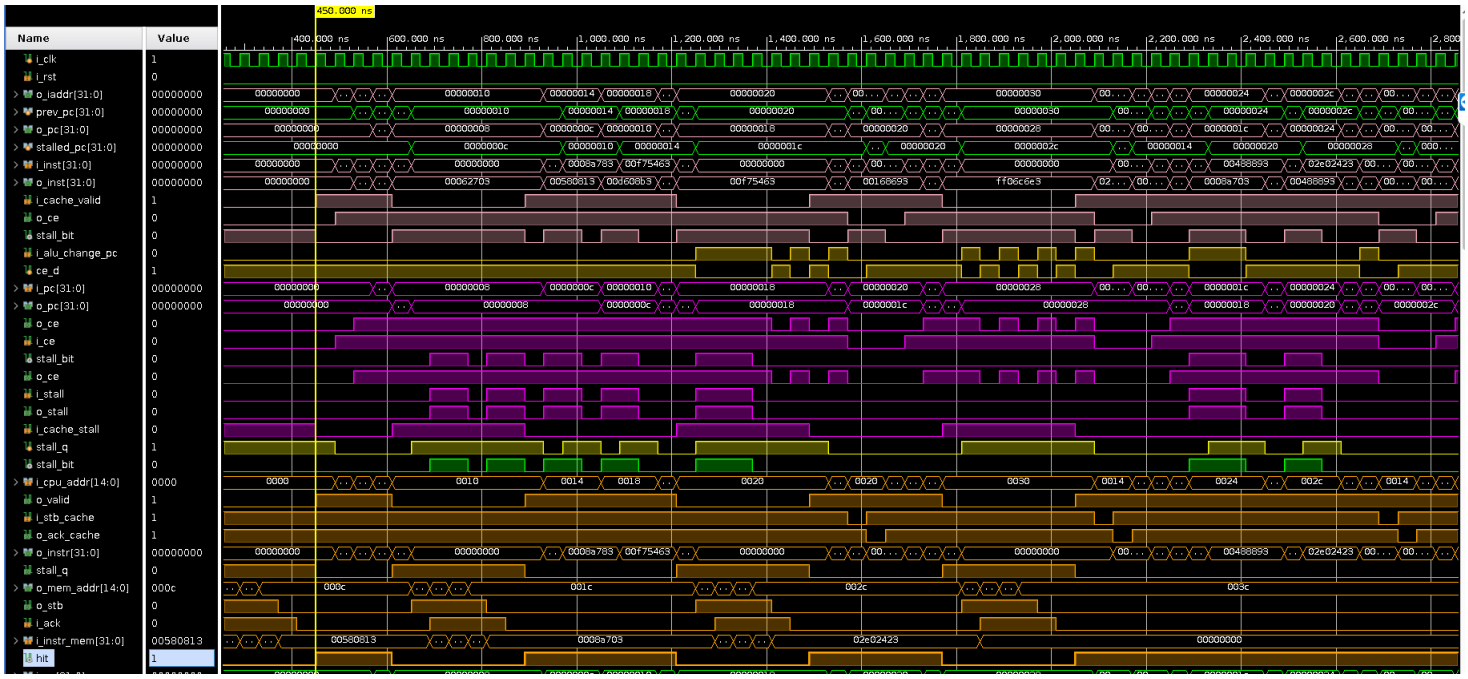


Figure 25: Memory Write Instructions

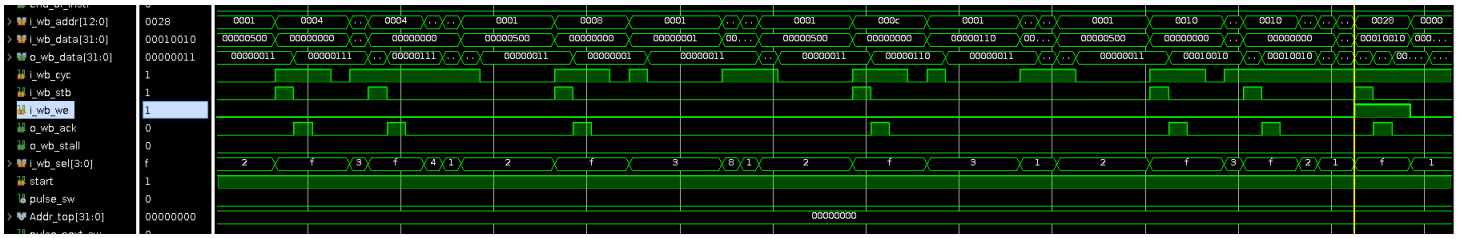


Figure 26: Memory Write Content after Execution

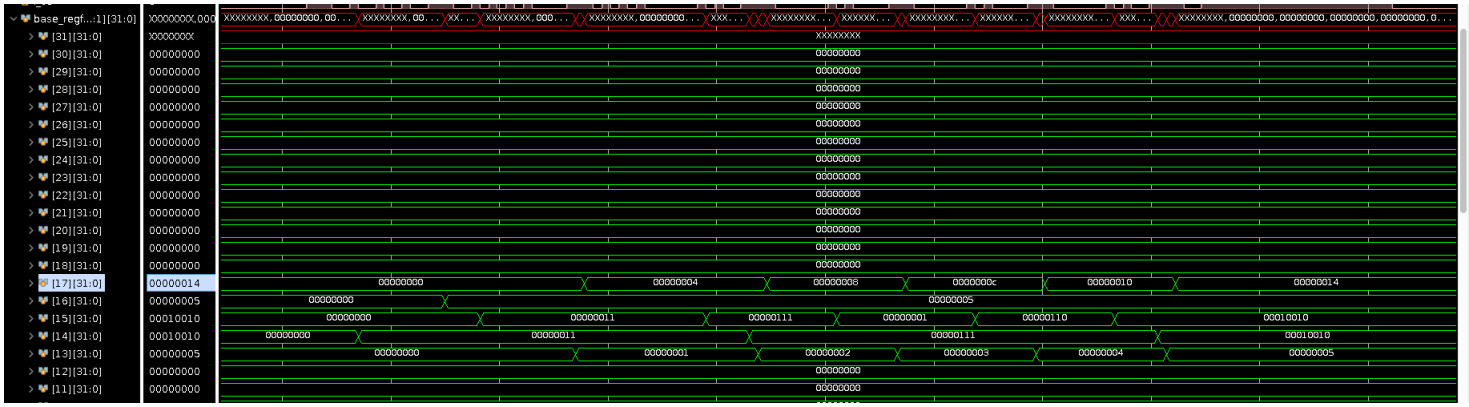


Figure 27: Register Operations

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 procesor. 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:

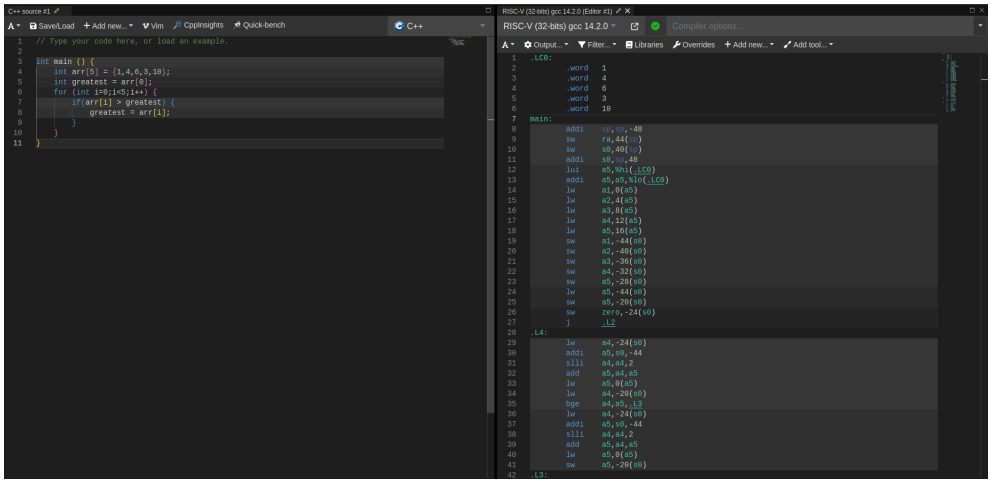


Figure 28: Greatest Number RISCV assembly code

Algorithm 2 Assembly code

```
main:
        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 = 95 Mhz

The timing is met with a Positive setup slack of 5.228ns for 15ns clock period.

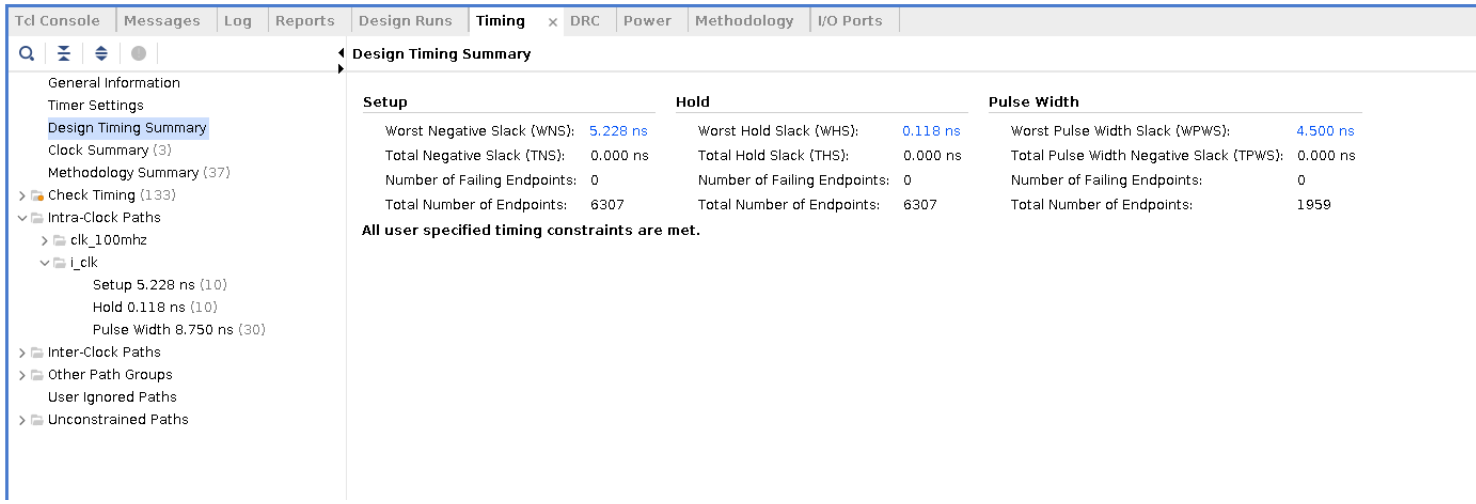
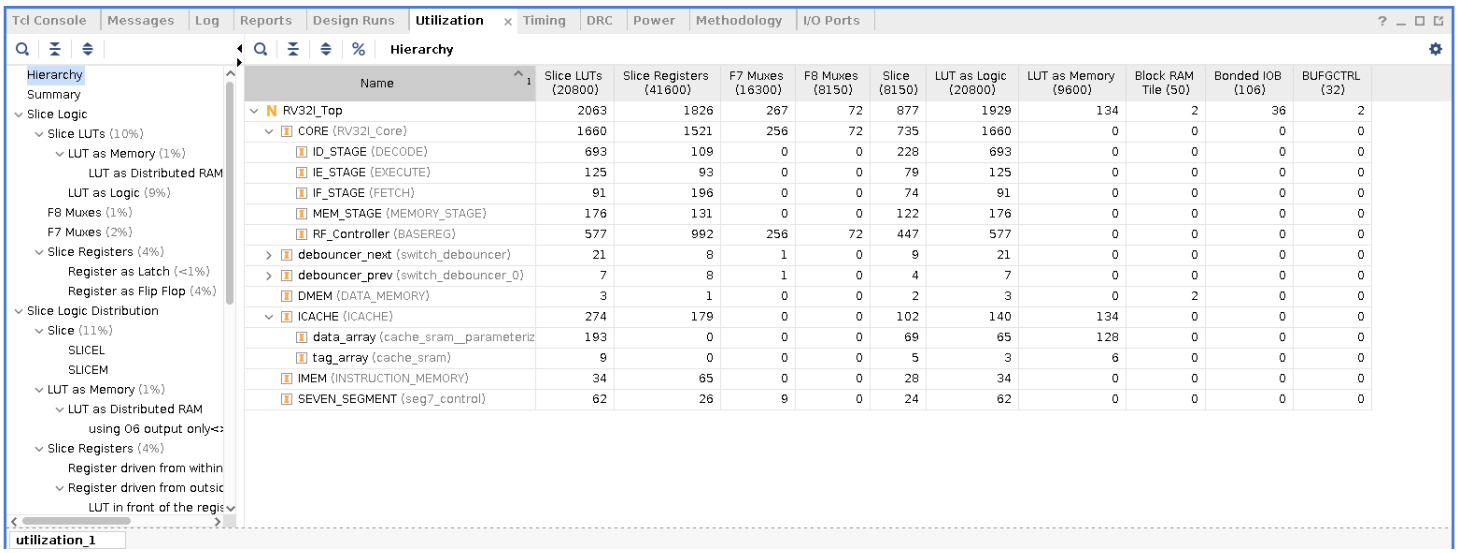


Figure 29: Timing Summary

VIII Resource Utilisation



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
RV32I_Top	2063	1826	267	72	877	1929	134	2	36	2
CORE (RV32I_Core)	1660	1521	256	72	735	1660	0	0	0	0
ID_STAGE (DECODE)	693	109	0	0	228	693	0	0	0	0
IE_STAGE (EXECUTE)	125	93	0	0	79	125	0	0	0	0
IF_STAGE (FETCH)	91	196	0	0	74	91	0	0	0	0
MEM_STAGE (MEMORY_STAGE)	176	131	0	0	122	176	0	0	0	0
RF_Controller (BASEREG)	577	992	256	72	447	577	0	0	0	0
debouncer_next (switch_debouncer)	21	8	1	0	9	21	0	0	0	0
debouncer_prev (switch_debouncer_0)	7	8	1	0	4	7	0	0	0	0
DMEM (DATA_MEMORY)	3	1	0	0	2	3	0	2	0	0
ICACHE (ICACHE)	274	179	0	0	102	140	134	0	0	0
data_array (cache_sram__parameteriz	193	0	0	0	69	65	128	0	0	0
tag_array (cache_sram)	9	0	0	0	5	3	6	0	0	0
IMEM (INSTRUCTION_MEMORY)	34	65	0	0	28	34	0	0	0	0
SEVEN_SEGMENT (seg7_control)	62	26	9	0	24	62	0	0	0	0

Figure 30: Resource Utilisation

IX Power Consumption

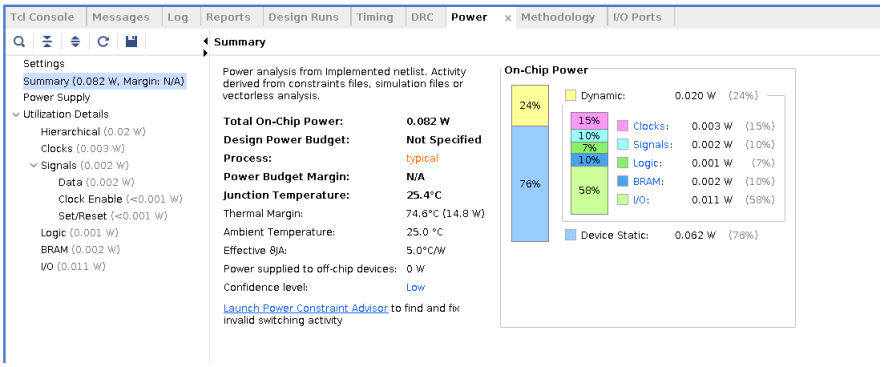


Figure 31: Power Consumption

X Single Cycle and Pipelined design comparision

	Single Cycle Design	Pipelined Design
Stages		
Clock Period	18.18ns	10.52ns
Frequency	55Mhz	95Mhz
Latency	18.18ns	52.63ns
Throughput	55M instr/clk	95M Instr/clk
CPI	1	>1 (Due to stall)

Table 3: Comparision between single cycle and pipelined design

XI DEMO on Basys3

We have used Xilinx Basys3 FPGA for our demo.

The program is to find the maximum number in an array of 5. Figure below shows the 5 numbers that we have taken as an input from the switches. The Seven Segment display order are as follows: <Mem_Location. Data >

Hence we have

0.015, 1.002, 2.099, 3.098, 4.042

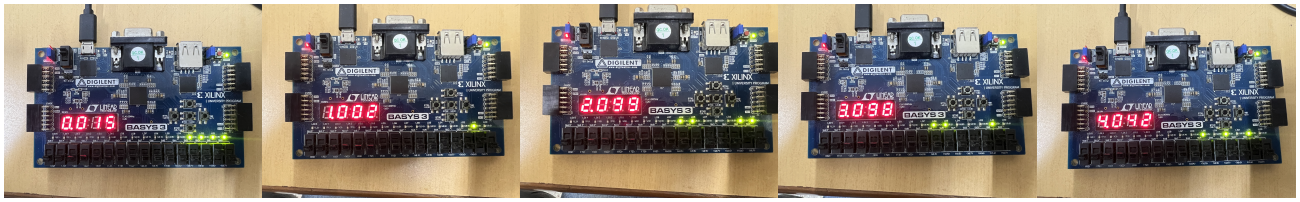


Figure 32: Demo on Basys3

We will get the result in the 8th location of memory.

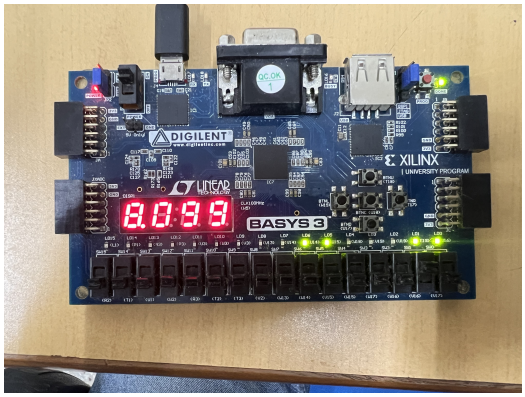


Figure 33: Demo on Basys3

XII Further Design Improvements

1. Trap and Interrupt address needs to be added to the WB stage.
2. CSR Support needs to added
3. Currently direction of Branch instructions are available at the IE stage, which causes a pipeline flush of next 2 instruction after the branch instruction. Early branch resolution can be added to the design, which will reduce the branch penalty from 2 to 1. For further improvements we need to add Branch Prediction modules, BTB, RAS and Direction predictor.

XIII Advantages of Pipeline Design

1. Increased throughput due to instruction overlap.
2. Relatively simple control logic compared to out-of-order processors.
3. Lower complexity and power consumption, making it suitable for embedded systems.

XIV Challenges and Considerations

1. Performance is limited by hazards that require stalling or flushing the pipeline.
2. In-order execution restricts parallelism, as instructions are executed strictly in the order they appear.
3. Not as efficient for handling complex dependencies or branches compared to out-of-order processors.

XV Conclusion

A 5-stage pipelined in-order processor is a foundational design in CPU architecture. By overlapping instruction execution, it improves processing speed and throughput compared to a single-cycle processor. Despite being simple and effective for basic workloads, it faces challenges with hazards and dependencies, which can reduce its efficiency in more complex applications. This architecture forms the basis of more advanced pipelined and out-of-order processors used in modern CPUs.

References

[1] Digital Design and Computer Architecture by David Money Haris and Sarah L. Haris

[2] Onur Mutlu Lectures

[3] Computer Organisation and Design RISCV Edition by DAVID.A.PATTERSON, JOHN.L.HENNESY