RISC -V Architecture Based Processor Design Report

Made By:
Akashay Singla

Table of Contents

1. Tools Used:

a. Synopsys VCS
b. GTK Wave
c. Design Vision

2. Different Modules of Designed Processor:

1. Program counter (64 bit)

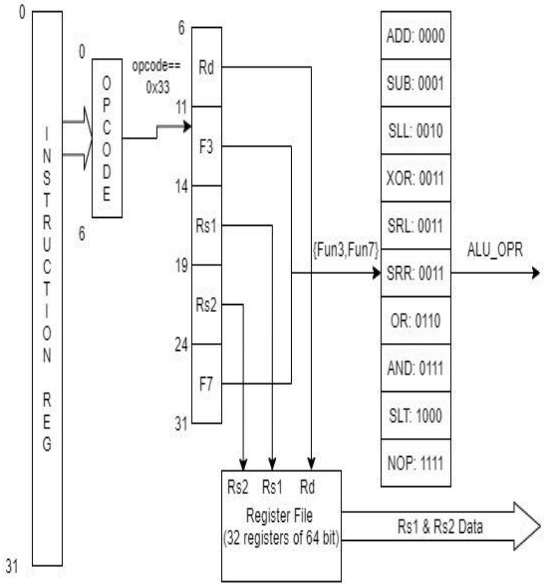
2. Instruction memory:

It is a byte addressable memory which is used to store the program. RISC-V instructions are of 32-bit so 4 bytes are required to store each instruction. This is the reason of incrementing the counter by decimal value 4.

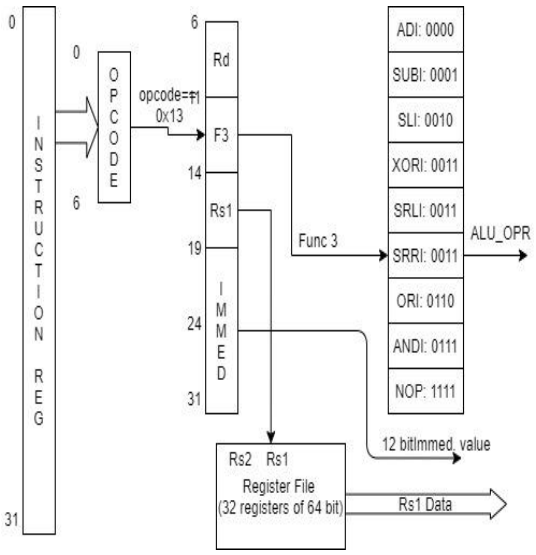
3. Decoder:

It provides 5-bit addresses of two inputs for fetching the data from register file and destination address for writing the results into register file. It also provides the below signals for different operations:

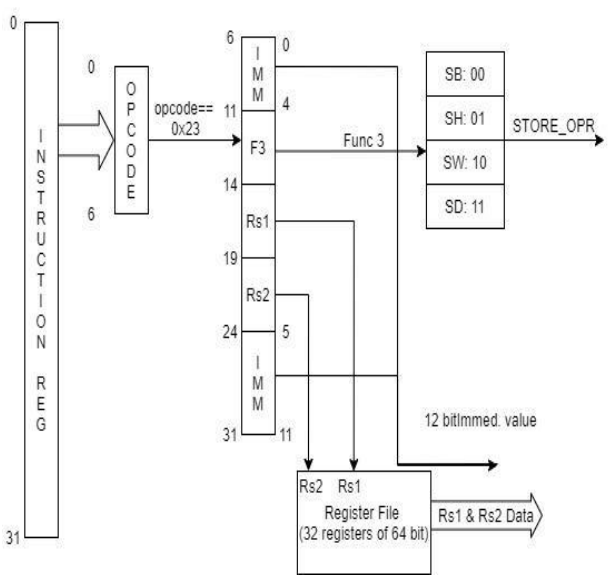
a. **Load Operation:** It is a 3-bit signal for memory operations. It tells the size of data which is going to be loaded from the data memory.
    i. 000: Load byte instruction
    ii. 001: Load halfword instruction
    iii. 010: Load word instruction
    iv. 011: Load unsigned byte
    v. 100: load unsigned halfword
    vi. 101: Load doubleword instruction
b. **Store Operation:** It is a 2-bit signal for memory operations. It tells the size of data which is going to be stored in the data memory.
    i. 00 store byte instruction
    ii. 01 store half word instruction
    iii. 10 store word instruction
    iv. 11 store double word instruction
c. **Memory Read Enable:** It tells that data is going to be loaded from the data memory.
d. **Memory Write Enable:** It tells that data is going to be stored in the data memory.
e. **Branch Enable:** This signal is set to 1 when the decoder receives branch instruction.
f. **Register Write Enable:** This signal is set to 1 when the result of ALU or output of memory is going to be written on the given destination address of the register file.



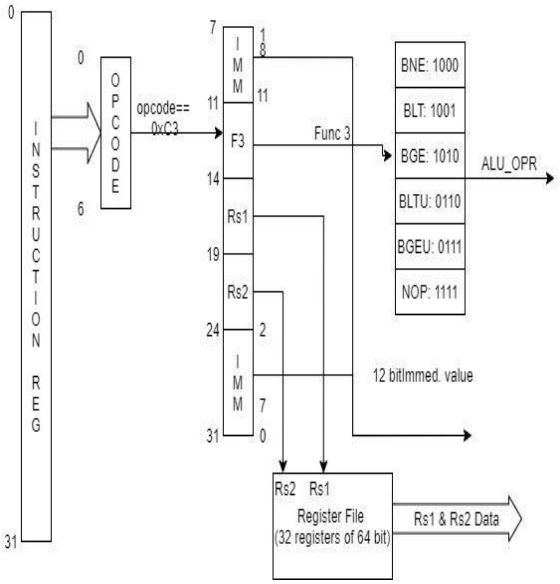
Arithmetic operation



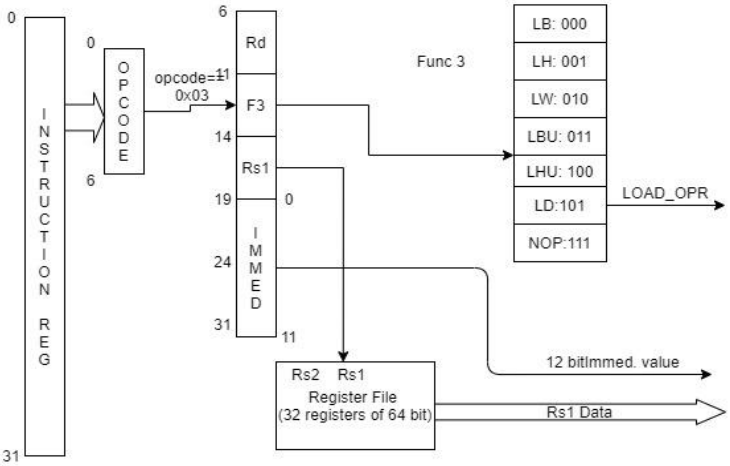
Immediate operation



Store operation



Branch operation

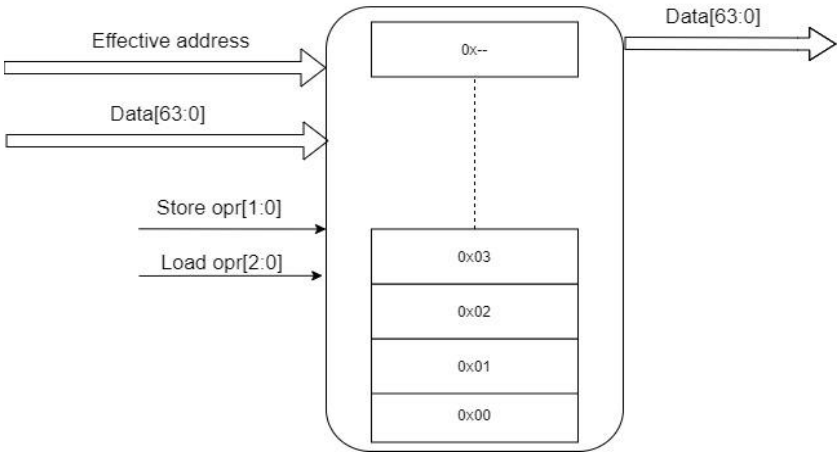


Load operation

4. **Register File:** It is an array of registers in CPU. In RISC-V architecture, sources and destination addresses are of 5-bit so that's it is an array of 32 registers which are 64-bit wide in the design.
5. **ALU (Arithmetic Logic Unit):** It performs all mathematical operations in the designed processor. Below are the operations performed in the design:

Sr. No.	Arithmetic operations	Sr. No	Branch Operations
1	Add	8	Branch equal
2	Sub	9	Branch not equal
3	Right shift	10	Branch less than
4	Left shift	11	Branch greater than
5	XOR	12	Branch JAL
6	AND		
7	OR		

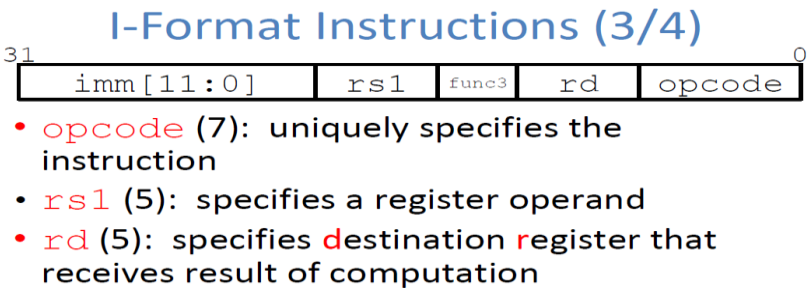
6. **Data Memory:** In RISC-V architecture, data memory is a byte addressable so it can store or load the data by byte, halfword, word & double word. In the design, 4KB memory has been implemented with their load/store control signals.



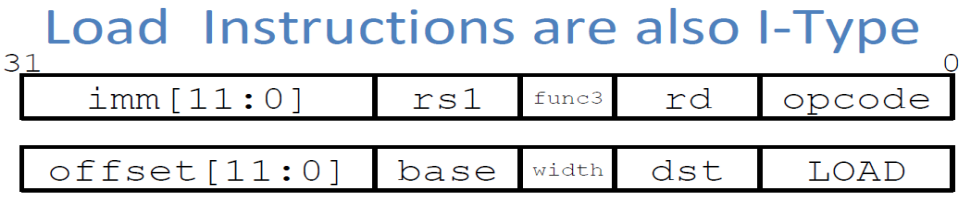
Data Memory

3. **Four types of instructions provide the immediate value:**

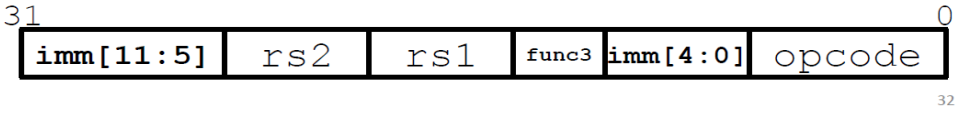
1. **I-Format Instructions:**



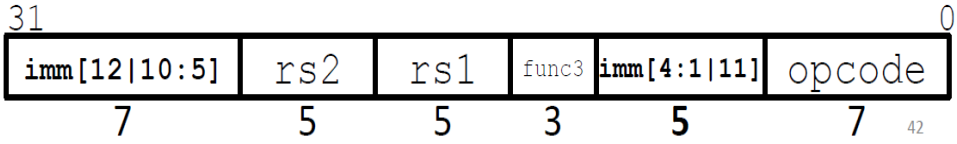
2. **Load Instruction:**



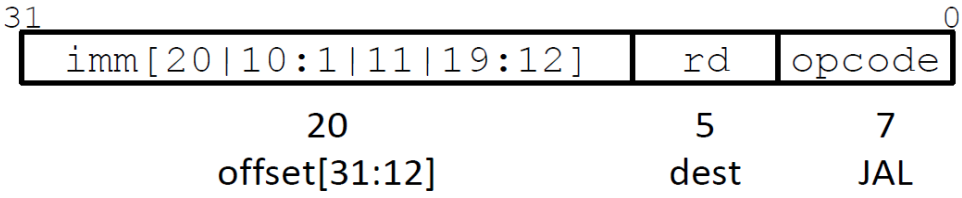
3. **Store Instruction:**



4. **Branch Instruction:**



5. **JAL instruction:** For general jumps, JAL instruction is used to jump anywhere in the code memory

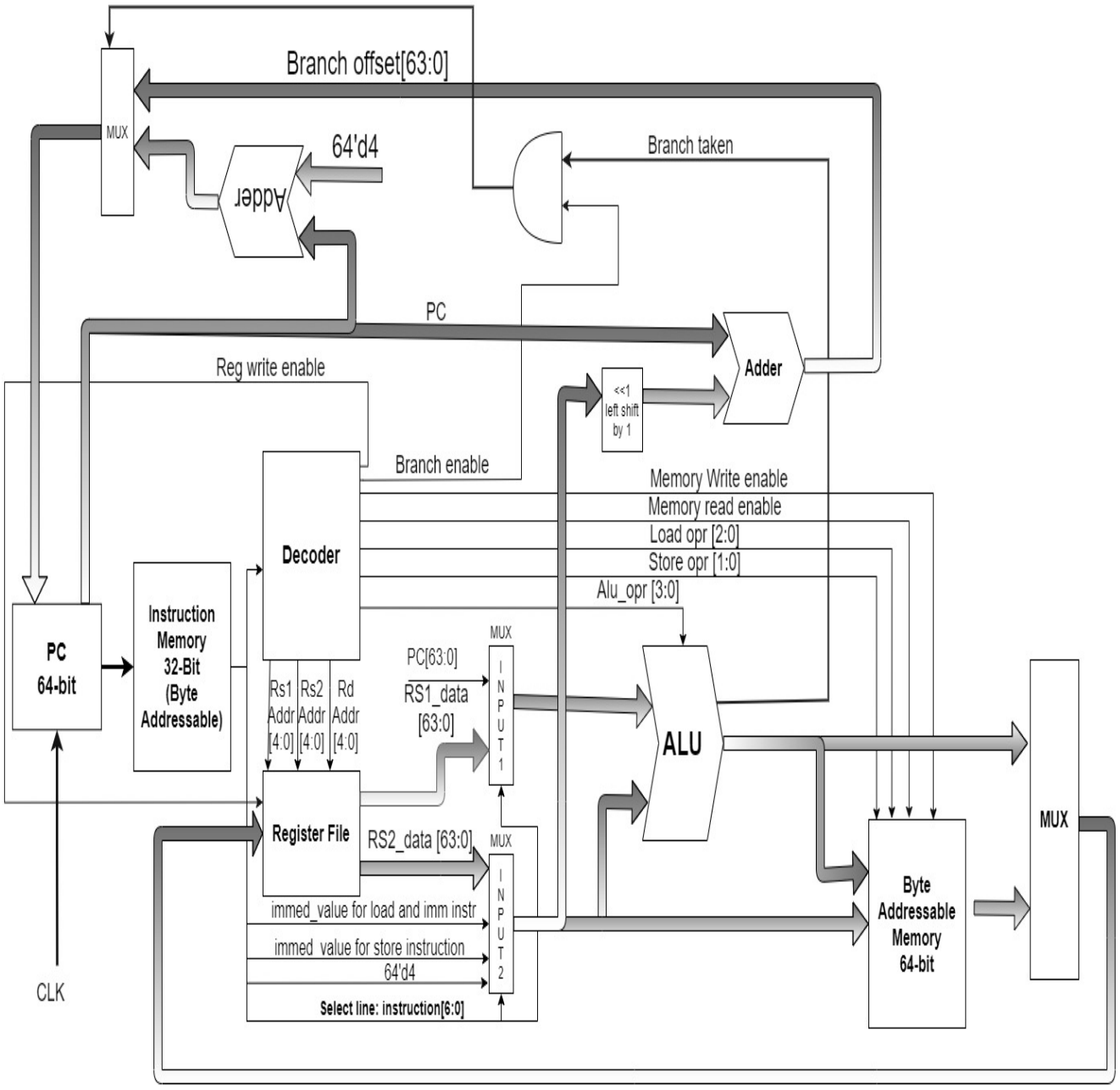


Below logics are implemented for them:

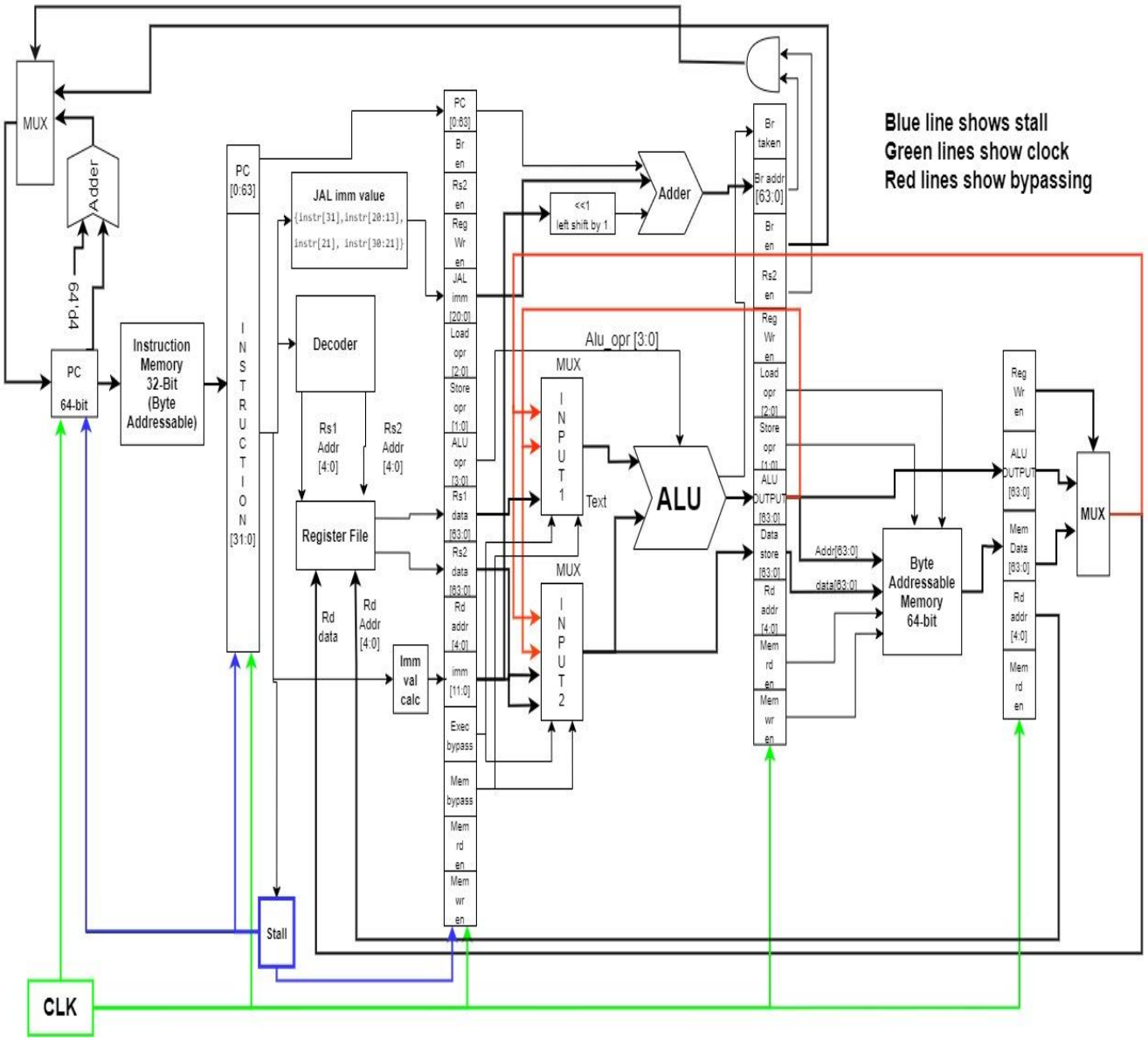
```
/*I-type & L-Load type instruction's immediate value*/
imm_val = (instr_buff[6:0] == 7'b0010011 || instr_buff[6:0] == 7'b0000011 )? instr_buff[31:20]:
/*Store instruction's immediate value*/
(instr_buff[6:0] == 7'b0100011) ?{instr_buff[31:25],instr_buff[11:7]}:
/*Branch instruction's immediate value*/
(instr_buff[6:0] == 7'b1100011)? {instr_buff[31],instr_buff[7],instr_buff[30:25],instr_buff[11:8]}:
 12'bz;
```

jal_imm_val = (instr_buff[6:0] === 7'b1101111)? ((instr_buff[31]==1'b1)?
{44'hFFFFFFFFFFF,instr_buff[31],instr_buff[20:13], instr_buff[21], instr_buff[30:21]} :
{44'h00000000000,instr_buff[31],instr_buff[20:13], instr_buff[21], instr_buff[30:21]}): 5'bzzzzz;

4. Single Cycle Datapath:

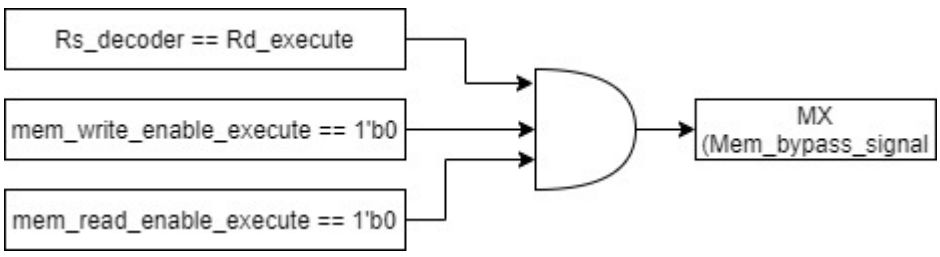


5. 5-Stage Pipeline Datapath:



Two types of bypassing are implemented in the design:

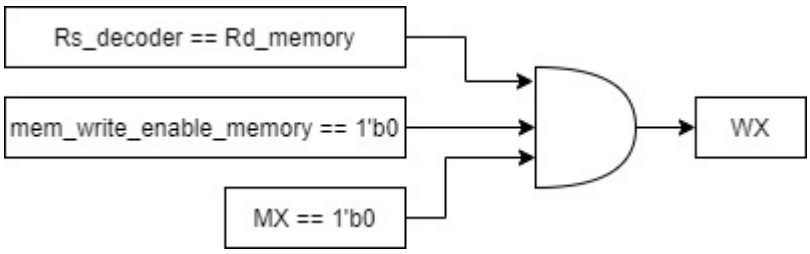
1. **MX:** If the current decoder stage's instruction is dependent upon the result of the execute stage's instruction which is not memory instruction, the result of execute stage's instruction is bypassed to the execute stage in the next clock cycle.



Below logics are implemented in the design:
exec_bypass_sig_Rs1 = ((Rs1_addr === Rd_addr_buff) && (mem_wr_en_buff === 1'b0) && (mem_rd_en_buff === 1'b0))? 1'b1: 1'b0;

exec_bypass_sig_Rs2 = ((Rs2_addr === Rd_addr_buff) && (mem_wr_en_buff === 1'b0) && (mem_rd_en_buff === 1'b0))? 1'b1: 1'b0;

2. **WX:** If the current decoder stage's instruction is dependent upon the result of the memory stage's instruction, result will be bypassed to execute stage on the next clock cycle. If memory stage's instruction is a memory instruction, then memory data is bypassed otherwise ALU output is bypassed to the execution stage. There should be no MX when WX is set to 1'b1.



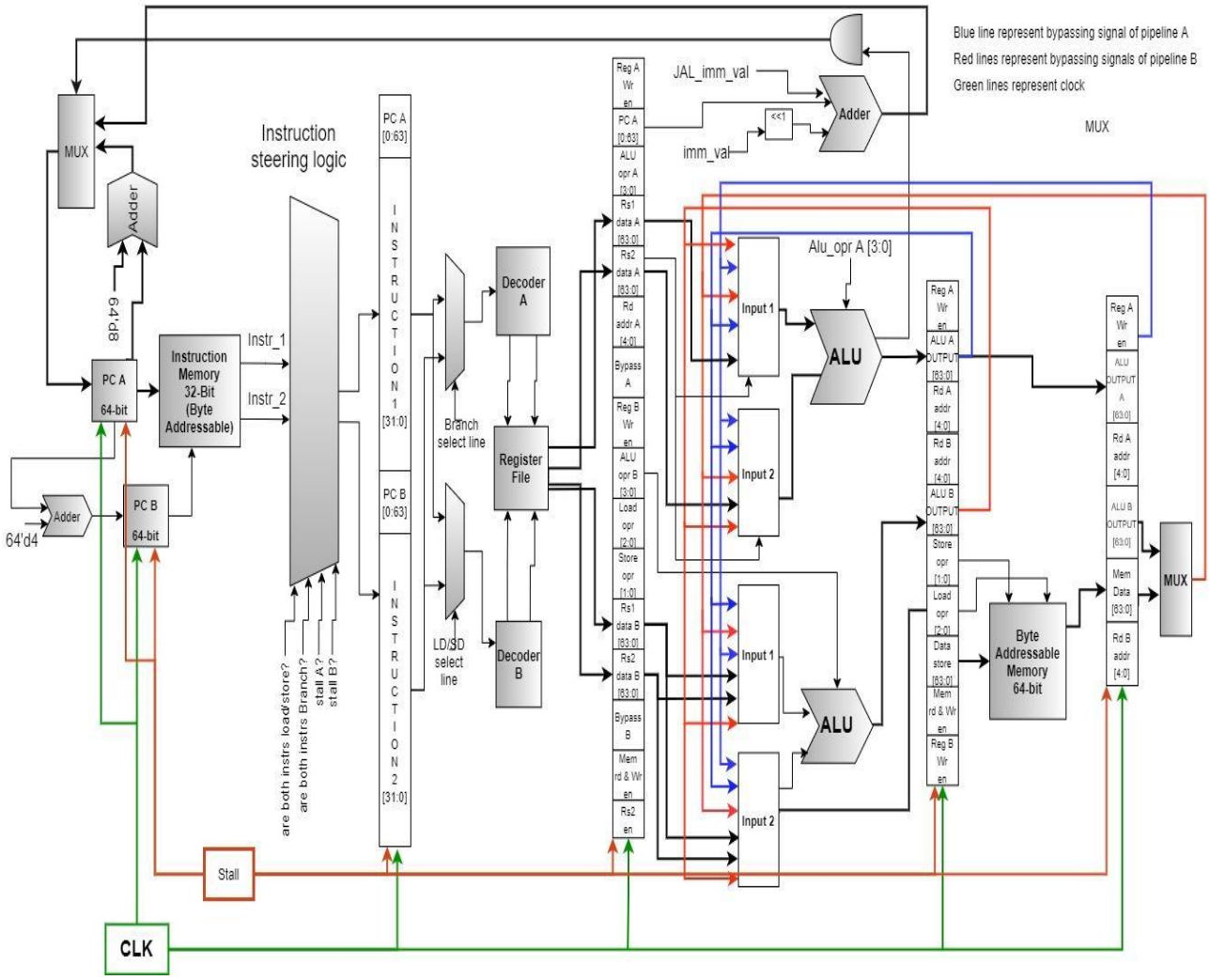
Below logic is implemented in the design:
mem_bypass_sig_Rs1 = ((Rs1_addr === Rd_addr_buff2) && (mem_wr_en_buff2 === 1'b0) && (exec_bypass_sig_Rs1 == 1'b0))? 1'b1: 1'b0;

mem_bypass_sig_Rs2 = ((Rs2_addr === Rd_addr_buff2) && (mem_wr_en_buff2 === 1'b0) && (exec_bypass_sig_Rs2 == 1'b0))? 1'b1: 1'b0;

3. **Stall Logic:** If one of the source address of current decoder stage's instruction is equal to the address of execute stage's instruction which is memory instruction then stall is required to get the data from memory and then it is bypassed to the execute stage. Below logic is implemented in the design:

stall = (((Rs1_addr == Rd_addr_buff) || (Rs2_addr == Rd_addr_buff)) && (mem_rd_en_buff == 1'b1 && mem_wr_en_buff == 1'b0))? 1'b1 : 1'b0;

6. 2-Way In-Order Superscalar Datapath:



1. Pipeline A deals with branch and arithmetic instructions only
2. Pipeline B deals with memory and arithmetic instructions only

Instruction issue steering logic:

1. **Are both instructions load/store?**
   If the decoder stage finds that both fetched instructions are load, then it allows to pass only a single instruction having pc value less than other instruction through pipeline B and pipeline A gets NOP instruction.
   Also, during the next cycle, a new instruction goes into pipeline B's decoder and the load/store instruction which was not executed in the previous cycle goes into pipeline A's decoder. Below logic was implemented to find both instructions are load/store:

   are_instrs_ld_sd = ((instr1_buff[6:0] == 7'b0000011 || instr1_buff[6:0] == 7'b0100011) && (instr2_buff[6:0] == 7'b0000011 || instr2_buff[6:0] == 7'b0100011))? 1'b1: 1'b0;

2. **Are both instructions branch?**

If the decoder stage finds that both fetched instructions are branch instructions, then it allows to pass only a single instruction having pc value less than other instruction through pipeline A and pipeline B gets NOP instruction.
Also, during the next cycle, a new instruction goes into pipeline B's decoder stage and the branch instruction which was not executed in the previous cycle goes into pipeline A's decoder. Below logic was implemented to find both instructions are branches:

```
are_instrs_br = ((instr1_buff[6:0]  ==  7'b1100011) && (instr2_buff[6:0]  == 7'b1100011))? 1'b1 : 1'b0;
```

3. **Stall A:**
    a. First condition:
    If decoder A's instruction is dependent upon the result of decoder B's instruction and pc of A's instruction is greater than decoder B's instruction, pipeline A is stalled for one cycle.
    b. Second condition:
    If decoder A's instruction is dependent upon the load/store instruction of pipeline B which is in the execute stage, pipeline A is stalled for one cycle so that data from memory will be available and then bypassed it to the execute stage.

Below logic was implemented:

```
stall_A = (((Rs1_addr1 == Rd_addr2 || Rs2_addr1 == Rd_addr2)&&(exch_pc > exch_pc4))|| (((Rs2_addr1 == Rd_addr2_buff)||(Rs1_addr1 == Rd_addr2_buff))&&(mem_rd_en2_buff == 1'b1)))? 1'b1:1'b0;
```

4. **Stall B:**
    a. First condition:
    If decoder B's instruction is dependent upon the result of decoder A's instruction and pc of B's instruction is greater than decoder A's instruction, pipeline B is stalled for one cycle.
    b. Second condition:
    If decoder B's instruction is dependent upon the load/store instruction of execute stage of its pipeline, pipeline B is stalled for one cycle so that data from memory will be available and then bypassed it to the execute stage.

```
stall_B = ((((Rs1_addr2 == Rd_addr2_buff) || (Rs2_addr2 == Rd_addr2_buff)) && (mem_rd_en2_buff == 1'b1))|| (((Rs1_addr2 == Rd_addr1) || (Rs2_addr2 == Rd_addr1))&&(exch_pc4 > exch_pc)))? 1'b1: 1'b0;
```

Note: In both Stall A & Stall B cases, Pipeline A's decoder stage gets the prior instruction which was not executed and pipeline B's decoder stage gets the new instruction.

5. **Two filters before entering into the decoder ensures that:**
    1. *Pipeline A's decoder does not get the load/store instruction.*
    2. *Pipeline B's decoder doesn't get branch instruction.*

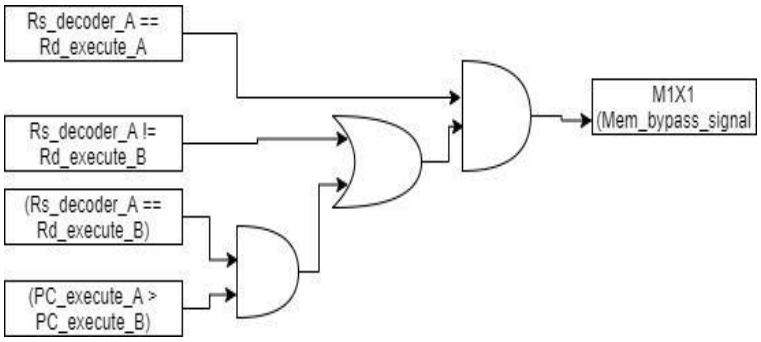
```
instr_inputA =(((instr1_buff[6:0] !== 7'b0000011 && instr1_buff[6:0] !== 7'b0100011)&& (instr2_buff[6:0] !== 7'b1100011) && (instr2_buff[6:0] !== 7'b1101111) ) ||(((instr1_buff[6:0] ===
```

7'b1100011 && instr2_buff[6:0] === 7'b1100011) || (instr1_buff[6:0] === 7'b1101111 && instr2_buff[6:0] === 7'b1101111)) && (pc4_buff > pc_buff)))? instr1_buff: (instr2_buff[6:0] !== 7'b0000011 && instr2_buff[6:0] !== 7'b0100011) ? instr2_buff : 32'bz;

instr_inputB = (((instr2_buff[6:0] !== 7'b1100011) && (instr2_buff[6:0] !== 7'b1101111) && (instr1_buff[6:0] !== 7'b0000011 && instr1_buff[6:0] !== 7'b0100011)) || ((instr1_buff[6:0] === 7'b0000011 || instr1_buff[6:0] === 7'b0100011) && (instr2_buff[6:0] === 7'b0000011 || instr2_buff[6:0] === 7'b0100011) &&(pc_buff > pc4_buff))) ? instr2_buff : ((instr1_buff[6:0] !== 7'b1100011) && (instr1_buff[6:0] !== 7'b1101111)) ? instr1_buff : 32'bz;

Intra and inter bypassing:

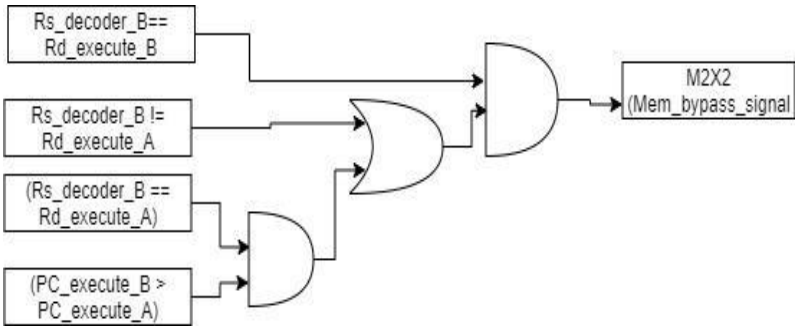
1. **M1X1:** If pipeline A's decoder stage's instruction is dependent upon the result of pipeline A's execute stage's instruction with two below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



exec_bypass1_Rs1 = ((Rs1_addr1 === Rd_addr1_buff) && (((Rs1_addr1 === Rd_addr2_buff)&&(pc_buff2 > pc4_buff2)) || (Rs1_addr1 !== Rd_addr2_buff))) ? 1'b1: 1'b0;

exec_bypass1_Rs2 = ((Rs2_addr1 === Rd_addr1_buff) && (((Rs2_addr1 === Rd_addr2_buff)&&(pc_buff2 > pc4_buff2))||(Rs2_addr1 !== Rd_addr2_buff)))? 1'b1: 1'b0;

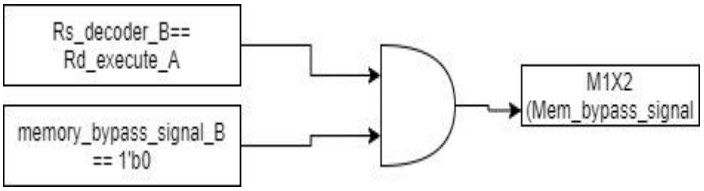
2. **M2X2:** If pipeline B's decoder stage's instruction is dependent upon pipeline B's execute stage's instruction with two below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



exec_bypass2_Rs1 = ((Rs1_addr2 === Rd_addr2_buff) && (mem_wr_en2_buff === 1'b0) && (mem_rd_en2_buff === 1'b0) && (((Rs1_addr2 === Rd_addr1_buff)&&(pc_buff2 < pc4_buff2))||(Rs1_addr2 !== Rd_addr1_buff))) ? 1'b1: 1'b0;

exec_bypass2_Rs2 = ((Rs2_addr2 === Rd_addr2_buff) && (mem_wr_en2_buff === 1'b0) && (mem_rd_en2_buff === 1'b0) && (((Rs2_addr2 === Rd_addr1_buff)&&(pc_buff2 < pc4_buff2))||(Rs2_addr2 !== Rd_addr1_buff))) ? 1'b1: 1'b0;

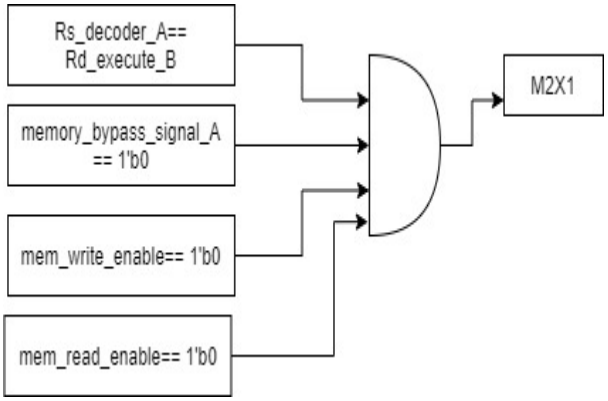
3. **M1X2:** If pipeline B's decoder stage's instruction is dependent upon the pipeline A's execute stage's instruction with below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



exec_intr_bp2_Rs1 = ((Rs1_addr2 === Rd_addr1_buff)&& (exec_bypass2_Rs1 == 1'b0)) ? 1'b1 : 1'b0;

exec_intr_bp2_Rs2 = ((Rs2_addr2 === Rd_addr1_buff)&& (exec_bypass2_Rs1 == 1'b0)) ? 1'b1 : 1'b0;

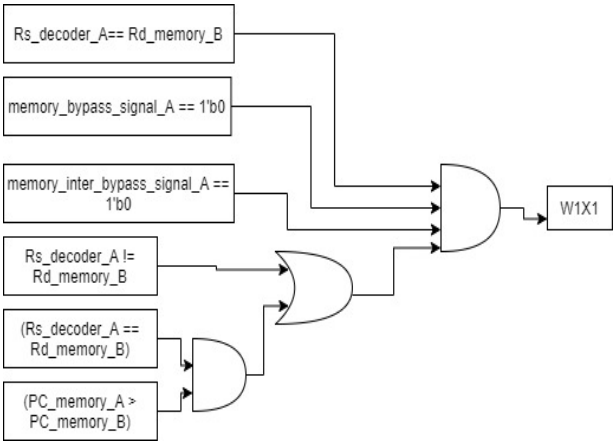
4. **M2X1:** If pipeline A's decoder stage's instruction is dependent upon the result of pipeline B's execute stage's instruction with below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



exec_intr_bp1_Rs1 =((Rs1_addr1 === Rd_addr2_buff) && (mem_wr_en2_buff === 1'b0) && (mem_rd_en2_buff === 1'b0) && (exec_bypass1_Rs1 == 1'b0)) ? 1'b1: 1'b0;

exec_intr_bp1_Rs2 =((Rs2_addr1 === Rd_addr2_buff) && (mem_wr_en2_buff === 1'b0) && (mem_rd_en2_buff === 1'b0) && (exec_bypass1_Rs2 == 1'b0)) ? 1'b1: 1'b0;

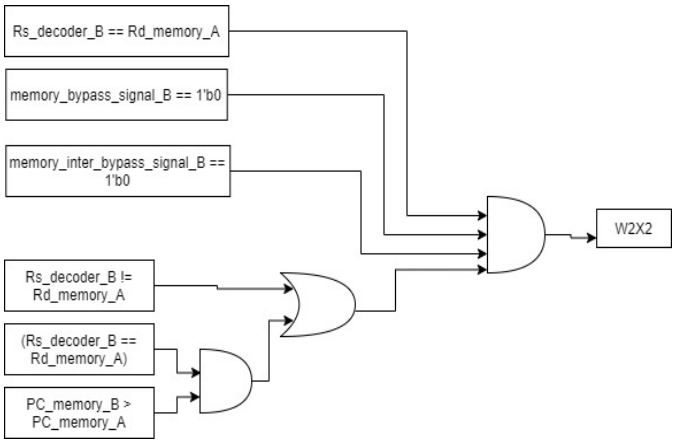
5. **W1X1:** If pipeline A's decoder stage's instruction is dependent upon the result of pipeline A's memory stage's instruction with below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



mem_bypass1_Rs1 = ((Rs1_addr1 === Rd_addr1_buff2) && (exec_bypass1_Rs1 == 1'b0) && (exec_intr_bp1_Rs1 == 1'b0) && (((Rs1_addr1 === Rd_addr2_buff2)&&(pc_buff3 > pc4_buff3))||(Rs1_addr1 !== Rd_addr2_buff2)))? 1'b1: 1'b0;

mem_bypass1_Rs2 = ((Rs2_addr1 === Rd_addr1_buff2) && (exec_intr_bp1_Rs2 == 1'b0) && (exec_bypass1_Rs2 == 1'b0) && (((Rs2_addr1 === Rd_addr2_buff2)&&(pc_buff3 > pc4_buff3))||(Rs2_addr1 !== Rd_addr2_buff2)))? 1'b1: 1'b0;

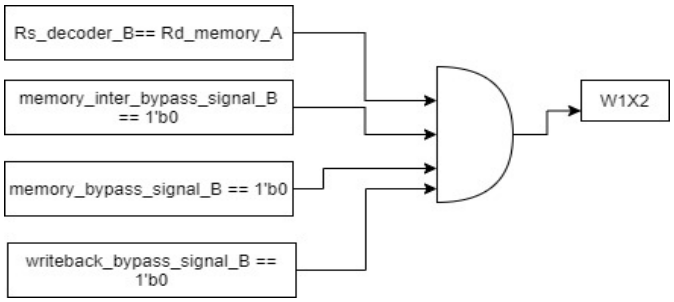
6. **W2X2:** If pipeline B's decoder stage's instruction is dependent upon the result of pipeline B's memory stage's instruction with below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



wrb_bypass1_Rs1 = ((Rs1_addr1 === Rd_addr1_buff3) && (exec_bypass1_Rs1 == 1'b0)&&(mem_bypass1_Rs1==1'b0)&& (exec_intr_bp1_Rs1 == 1'b0) && (((Rs2_addr1 === Rd_addr2_buff3)&&(pc_buff4 > pc4_buff4))||(Rs2_addr1 !== Rd_addr2_buff3)))? 1'b1 : 1'b0;

wrb_bypass1_Rs2 = ((Rs2_addr1 === Rd_addr1_buff3 && reg_wr_en1_buff3 == 1'b1) && (exec_bypass1_Rs2 == 1'b0)&&(mem_bypass1_Rs2==1'b0)&& (exec_intr_bp1_Rs2 == 1'b0) && (((Rs2_addr1 === Rd_addr2_buff3)&&(pc_buff4 > pc4_buff4))||(Rs2_addr1 !== Rd_addr2_buff3))) ? 1'b1 : 1'b0;

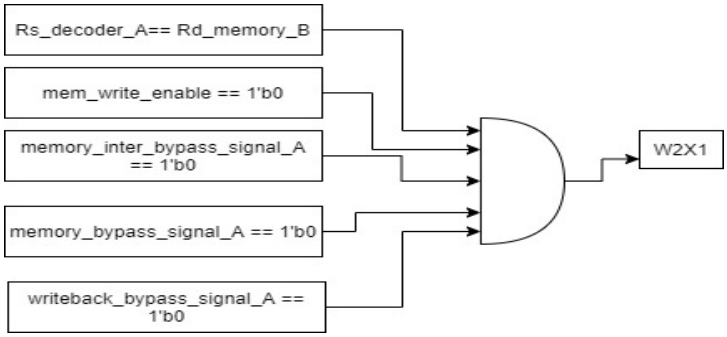
7. **W1X2:** If pipeline B's decoder stage's instruction is dependent upon the pipeline A's memory stage's instruction with below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



mem_intr_bp2_Rs1 = ((Rs1_addr2 === Rd_addr1_buff2)&&(exec_intr_bp2_Rs1 == 1'b0) && (exec_bypass2_Rs1 == 1'b0)&& (mem_bypass1_Rs2 == 1'b0)) ? 1'b1: 1'b0;

mem_intr_bp2_Rs2 =((Rs2_addr2 === Rd_addr1_buff2)&& (exec_intr_bp2_Rs2 == 1'b0) && (exec_bypass2_Rs2 == 1'b0)&& (mem_bypass2_Rs2 == 1'b0)) ? 1'b1: 1'b0;

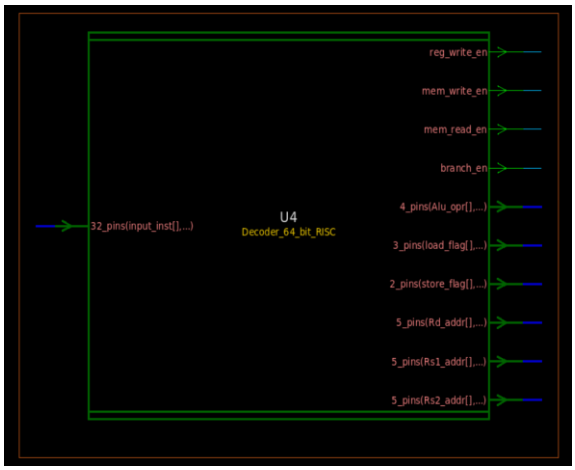
8. **W2X1:** If pipeline A's decoder stage's instruction is dependent upon the pipeline B's memory stage's instruction with below conditions represented by diagram, the result of instruction is bypassed to the execute stage in the next clock cycle.



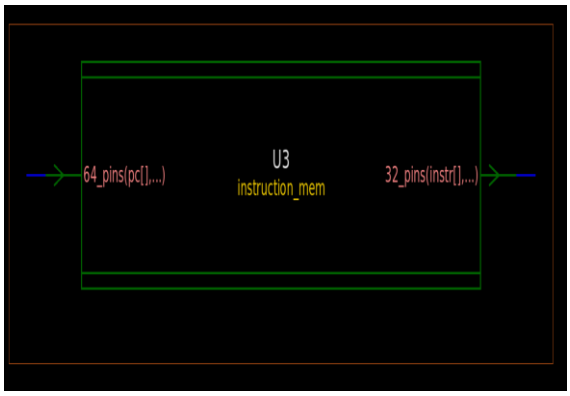
mem_intr_bp1_Rs1 = ((Rs1_addr1 === Rd_addr2_buff2) && (mem_wr_en2_buff2 === 1'b0) && (exec_intr_bp1_Rs1 == 1'b0) && (exec_bypass1_Rs1 == 1'b0)&& (mem_bypass1_Rs1 == 1'b0)) ? 1'b1: 1'b0;
mem_intr_bp1_Rs2 = ((Rs2_addr1 === Rd_addr2_buff2) && (mem_wr_en2_buff === 1'b0) && (exec_intr_bp1_Rs2 == 1'b0) && (exec_bypass1_Rs2 == 1'b0)&& (mem_bypass1_Rs2 == 1'b0))? 1'b1: 1'b0;

7. Synthesis Results of designs:

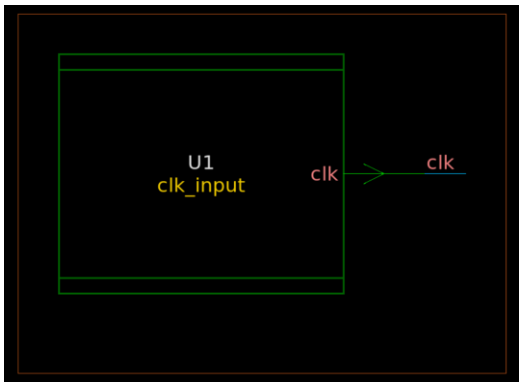
1. Decoder:



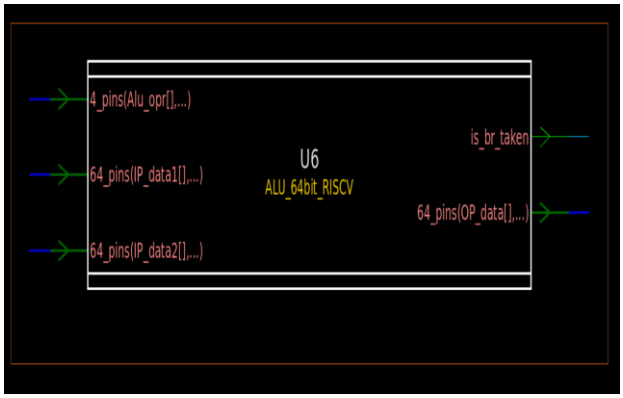
2. **Instruction Memory:**



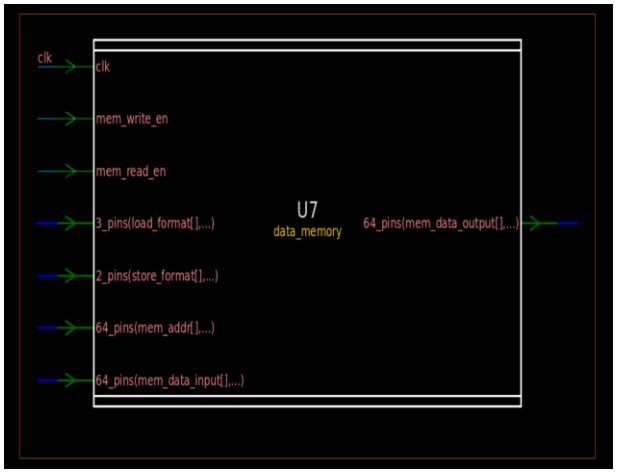
3. **Clock:**



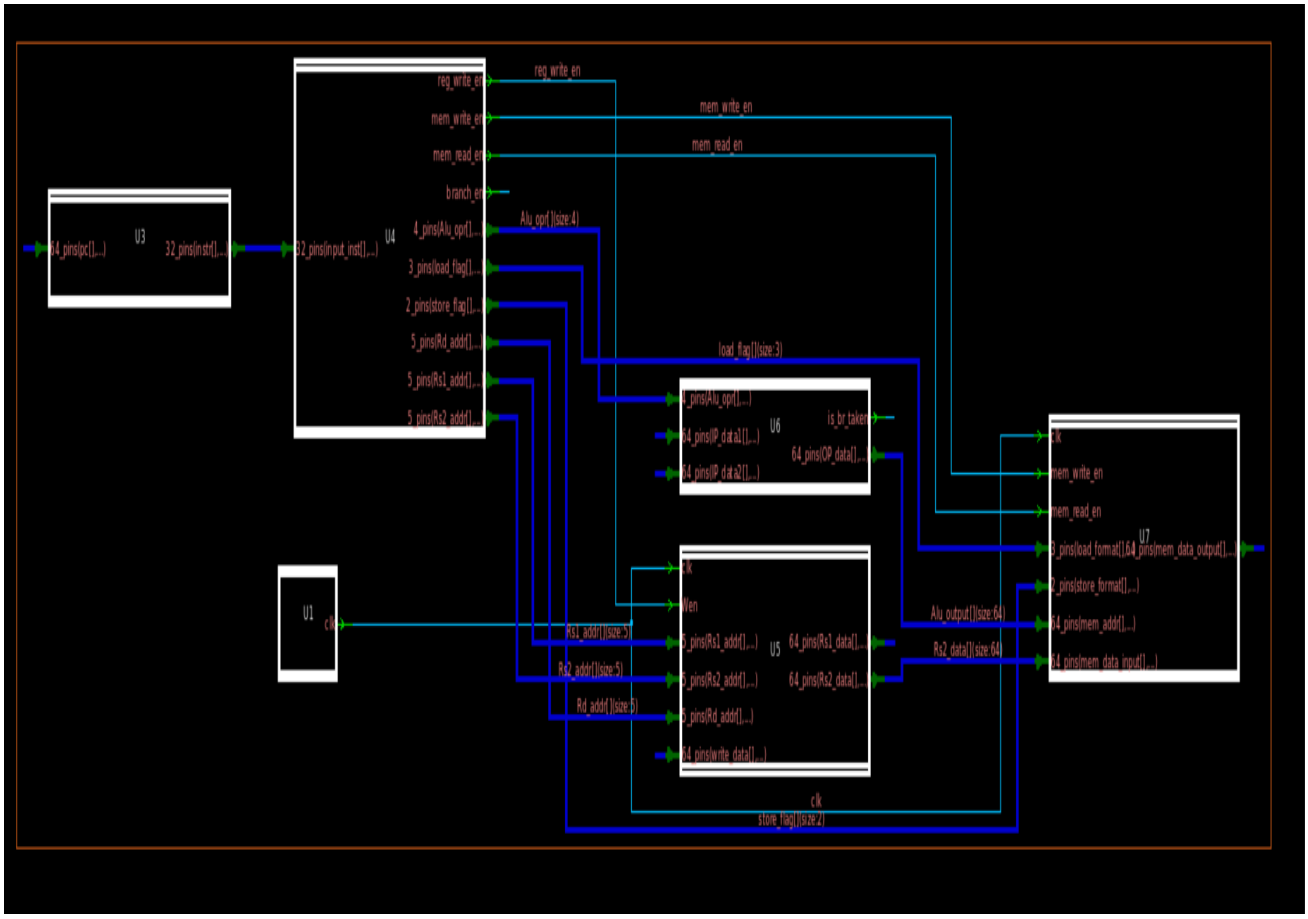
4. **ALU:**



5. **Data Memory:**

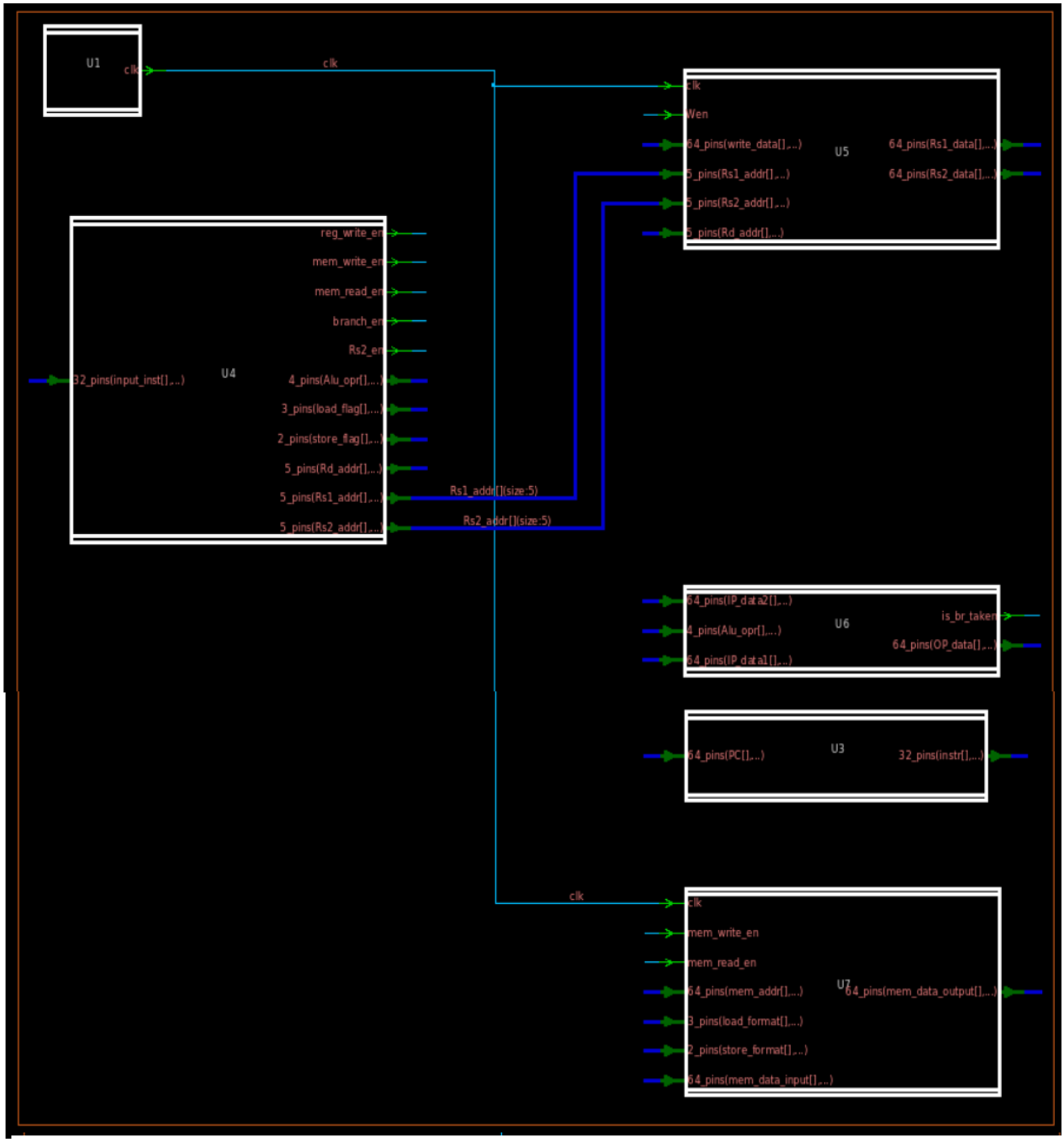


6. Single Cycle Datapath:



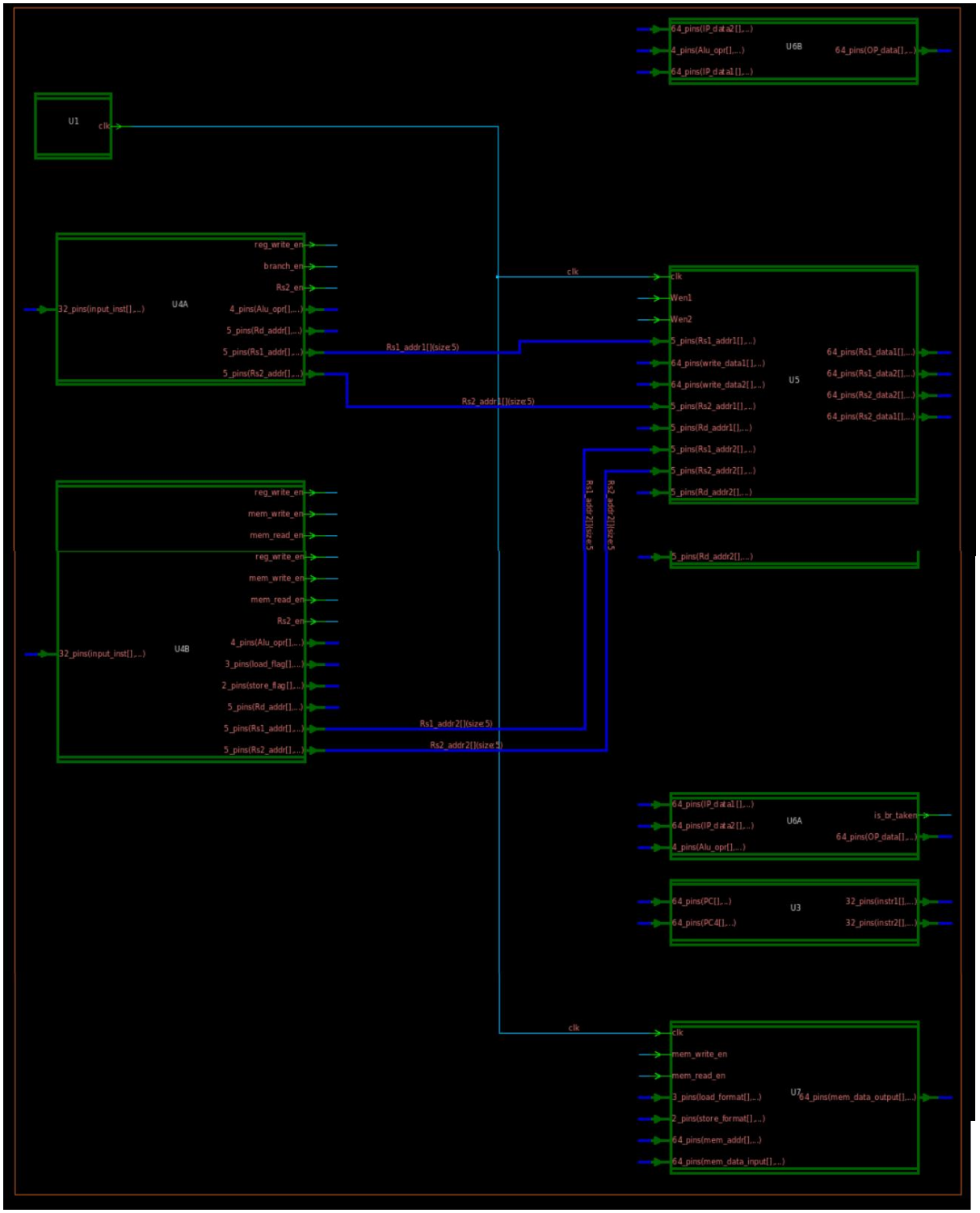
Note: In the above diagram, design vision tool generated the different modules but it didn't make their connections. While generating schematic, it provides register level schematic view not top-level view

7. 5-Stage Pipeline Datapath:



Note: In the above diagram, design vision tool generated the different modules but it didn't make their connections. While generating schematic, it provides register level schematic view not top-level view

8. In-order 2- Way Superscalar:



Note: In the above diagram, design vision tool generated the different modules but it didn't make their connections. While generating schematic, it provides register level schematic view not top-level view