IMPLEMENTATION OF RISC-V RV32I BASE INTEGER INSTRUCTION SET ARCHITECTURE

RISC-V is an open standard instruction set architecture based on established reduced instruction set computer principles. Unlike most other ISA designs, RISC-V is provided under royalty-free open-source licenses.

An Instruction set architecture(ISA) defines the necessary instruction set that a microcontroller must be able to execute, so that it defines a clear standard software developer can follow without worrying about the underlying hardware assembly that executes these instructions. It basically act as the abstraction layer between the software developer and the hardware designers so that the software developers can focus on writing software for a specific architecture without worrying about how things are working under the hood in microarchitecture level, meanwhile the hardware designers can design and implement the microarchitecture in any way possible without. RISC-V is open source ISA meaning that any company can produce its own microcontroller, microprocessors and microarchitecture without any licensing fees unlike other popular ISA like x86.

In this project I have implemented the RV32I base integer instruction set of RISC-V. However, there are many versions depending upon the need and you can even add or remove extensions on the current version depending upon what is needed for the current work.

The goal of the project is to integrate the RV32I microarchitecture to create a simple microcontroller with few GPIOS and special function ports.

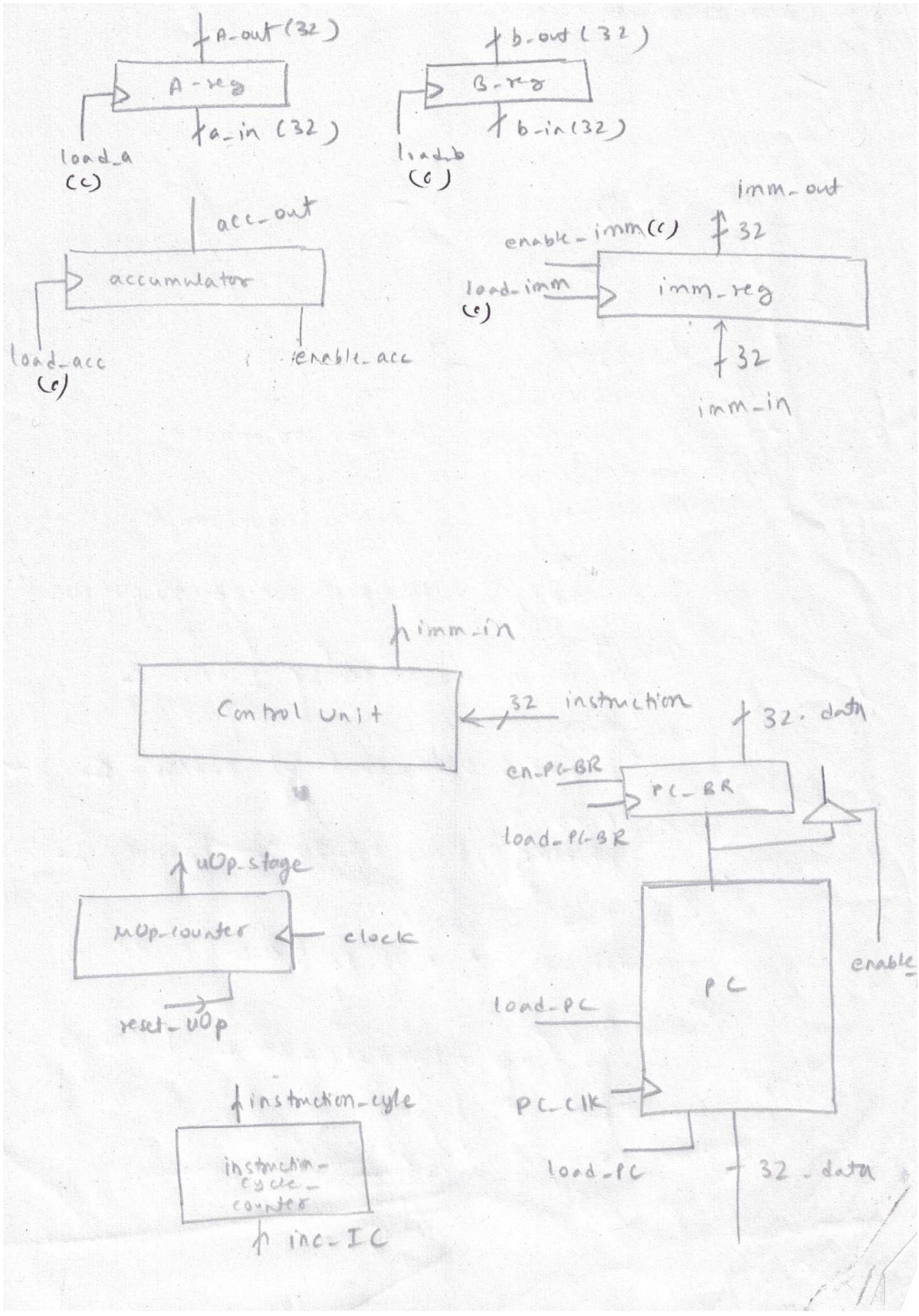
The progress up until now on the FPGA has been as follows.

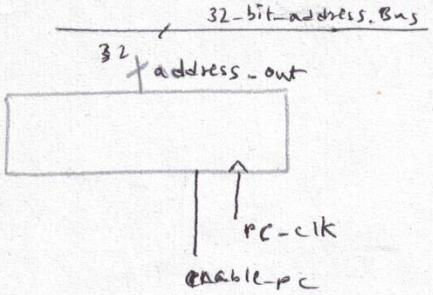
RV32I Base Instruction Set Encoding [1, p. 104]

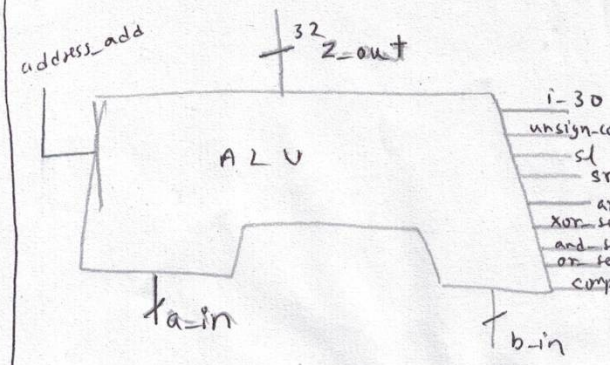
31 25	24 20	19 15	14 12	11 7	6 0			
imm[31:12]				rd	0 1 1 0 1 1 1	U-type	lui	rd,imm
imm[31:12]				rd	0 0 1 0 1 1 1	U-type	auipc	rd,imm
imm[20|10:1|11|19:12]				rd	1 1 0 1 1 1 1	J-type	jal	rd,pcrel_21
imm[11:0]		rs1	0 0 0	rd	1 1 0 0 1 1 1	I-type	jalr	rd,imm(rs1)
imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	beq	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bne	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	blt	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bge	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bltu	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bgeu	rs1,rs2,pcrel_13
imm[11:0]		rs1	0 0 0	rd	0 0 0 0 0 1 1	I-type	lb	rd,imm(rs1)
imm[11:0]		rs1	0 0 1	rd	0 0 0 0 0 1 1	I-type	lh	rd,imm(rs1)
imm[11:0]		rs1	0 1 0	rd	0 0 0 0 0 1 1	I-type	lw	rd,imm(rs1)
imm[11:0]		rs1	1 0 0	rd	0 0 0 0 0 1 1	I-type	lbu	rd,imm(rs1)
imm[11:0]		rs1	1 0 1	rd	0 0 0 0 0 1 1	I-type	lhu	rd,imm(rs1)
imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sb	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1	S-type	sh	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sw	rs2,imm(rs1)
imm[11:0]		rs1	0 0 0	rd	0 0 1 0 0 1 1	I-type	addi	rd,rs1,imm
imm[11:0]		rs1	0 1 0	rd	0 0 1 0 0 1 1	I-type	slti	rd,rs1,imm
imm[11:0]		rs1	0 1 1	rd	0 0 1 0 0 1 1	I-type	sltiu	rd,rs1,imm
imm[11:0]		rs1	1 0 0	rd	0 0 1 0 0 1 1	I-type	xori	rd,rs1,imm
imm[11:0]		rs1	1 1 0	rd	0 0 1 0 0 1 1	I-type	ori	rd,rs1,imm
imm[11:0]		rs1	1 1 1	rd	0 0 1 0 0 1 1	I-type	andi	rd,rs1,imm
0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1	I-type	slli	rd,rs1,shamt
0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srli	rd,rs1,shamt
0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srai	rd,rs1,shamt
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	add	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	sub	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	R-type	sll	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1	R-type	slt	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1	R-type	sltu	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	R-type	xor	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	srl	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	sra	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	R-type	or	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	R-type	and	rd,rs1,rs2

The table above shows the instructions available on the RV32I. The fence instruction is not implemented as its operation involves more than one cpu core. I have not considered implementing more than one RV32I core for this project hence it is not implemented at all.

PROCESSOR CORE ORGANIZATION







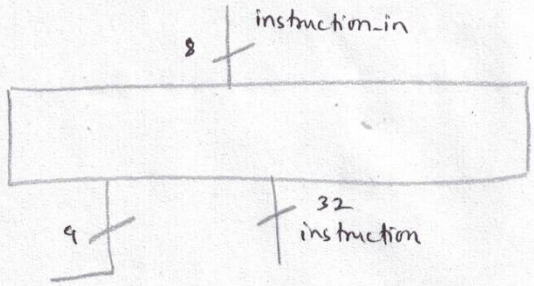
arith = add or sub, i_30 dictates

sl = shift left logical

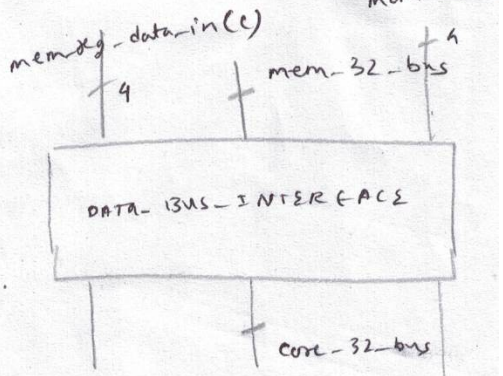
sr = shift right logical/arith depends on i_30

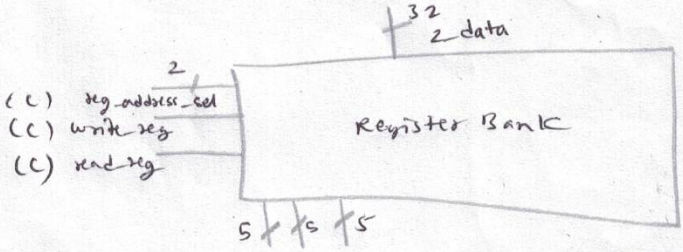
comp_sel = select comparator result

unsign_comp = 1, means the comparision is on unsigned number



load_instruction (C)
= 0 = none / default
= 1 = Load LSB
= 2
= 4
= 8 = Load MSB





reg_address[2:0]	mnemonic
0 0	R0
0 1	R1
1 0	RS1
1 1	RS2

The figure shows different sub modules used within the organization to realize the operation.

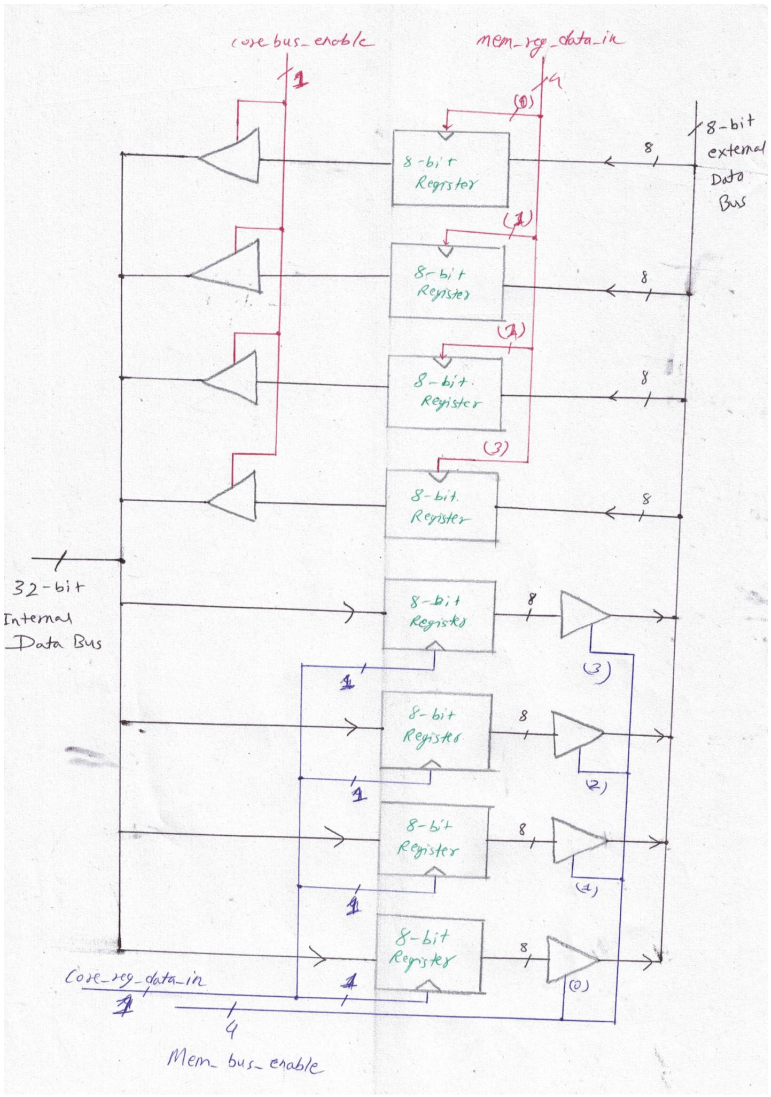
RISC-V follows the *little-endian* system, where the least significant byte is stored at the lower memory address and the last most significant byte is stored at the upper location.

For example the instruction bgeu x1,x2,-8 hex equivalent is FE20FCE3.

Such that the 32 bit instruction is stored as follows.

0x00000003=>FE
0x00000002=>20
0x00000001=>FC
0x00000000=>E3

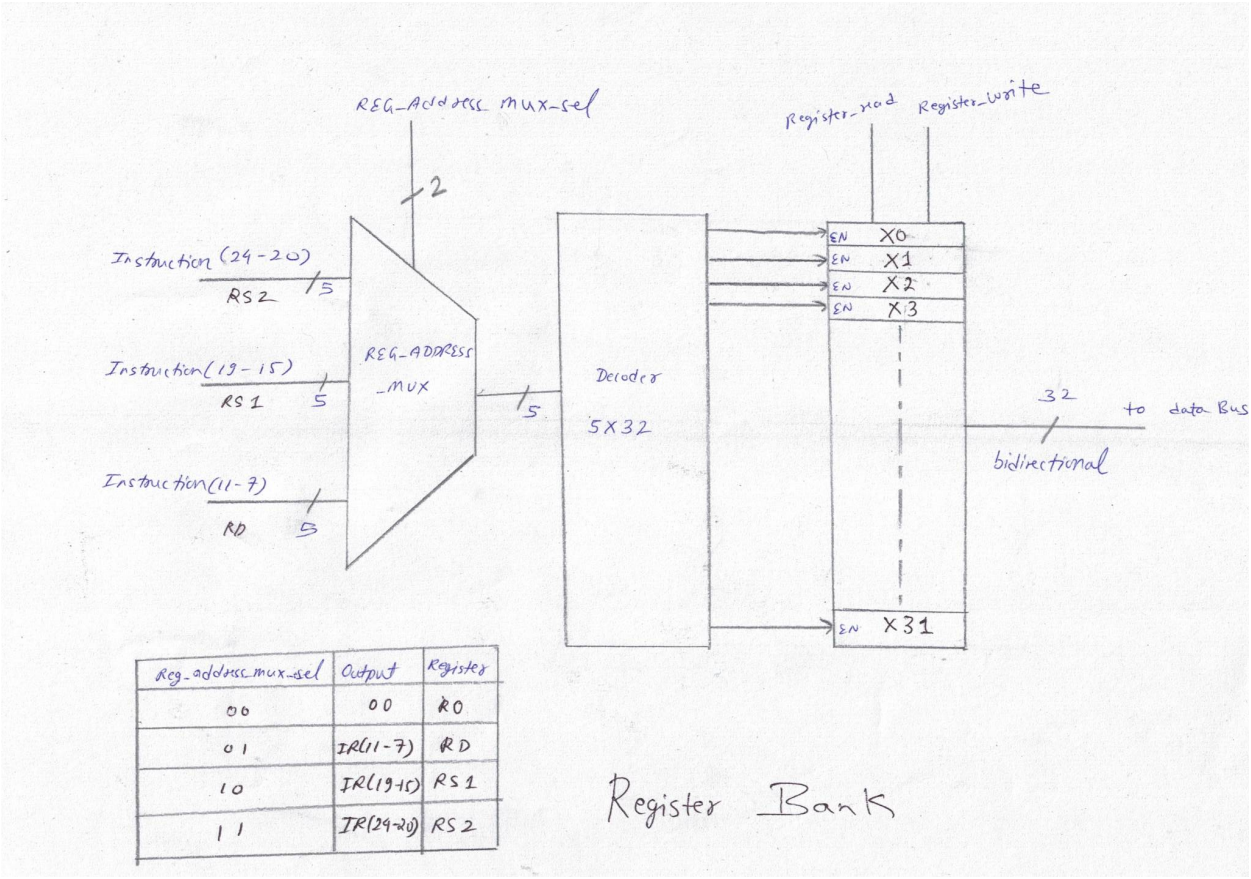
Since 1 byte of data is transferred between memory and the processor core, whilst the internal operation is 32 bit. There must be an interface that converts the 8 bit to 32 bit and vice versa.



The diagram shows the implementation of a module, its primitives and associated control signal which achieves the function of translation the data to required width.

This is necessary since there are instructions where we just need to load and store 1 byte and half word of data. For these, this interface provides the function for ordering the data into the correct length.

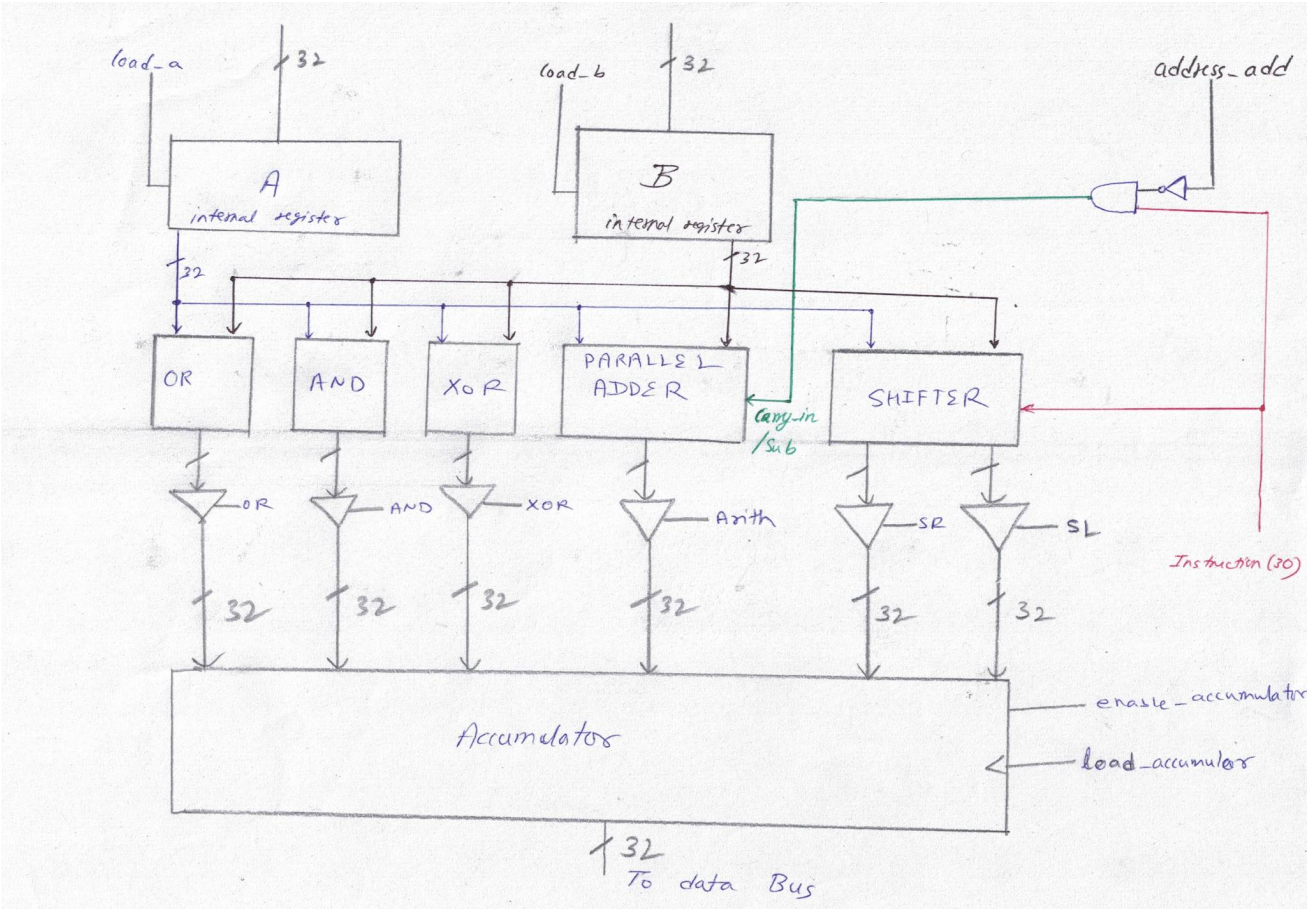
There are many registers within the RISC-V ISA, 32 to be exact. It is designed such that the register 0 permanent holds the value 0 and can not be further changed. The digital circuit configuration below provides the organization used to implement the 32 registers, it control signals as well as its operations.



Reg_address_mux_sel	Output	Register
00	00	R0
01	IR(11-7)	RD
10	IR(19-15)	RS1
11	IR(24-20)	RS2

The instructions are developed such that the address of the source and destination register are encoded within the instruction itself. Hence, the bits of instruction are fed to the decoder itself. The reg_addresss_nux_sel selection line is used to select the sources and destination register address. These locations are selected depending upon the instruction being executed.

The RISC-V instructions are encoded such that bits within the instruction specify whether the operation is addition or subtraction, logical shift or arithmetic shift. This is specified by the 30th bit position(counting from 0) of the instruction field. The ALU itself is built from simple logic gates and modules.The figure below shows the organization of the ALU.



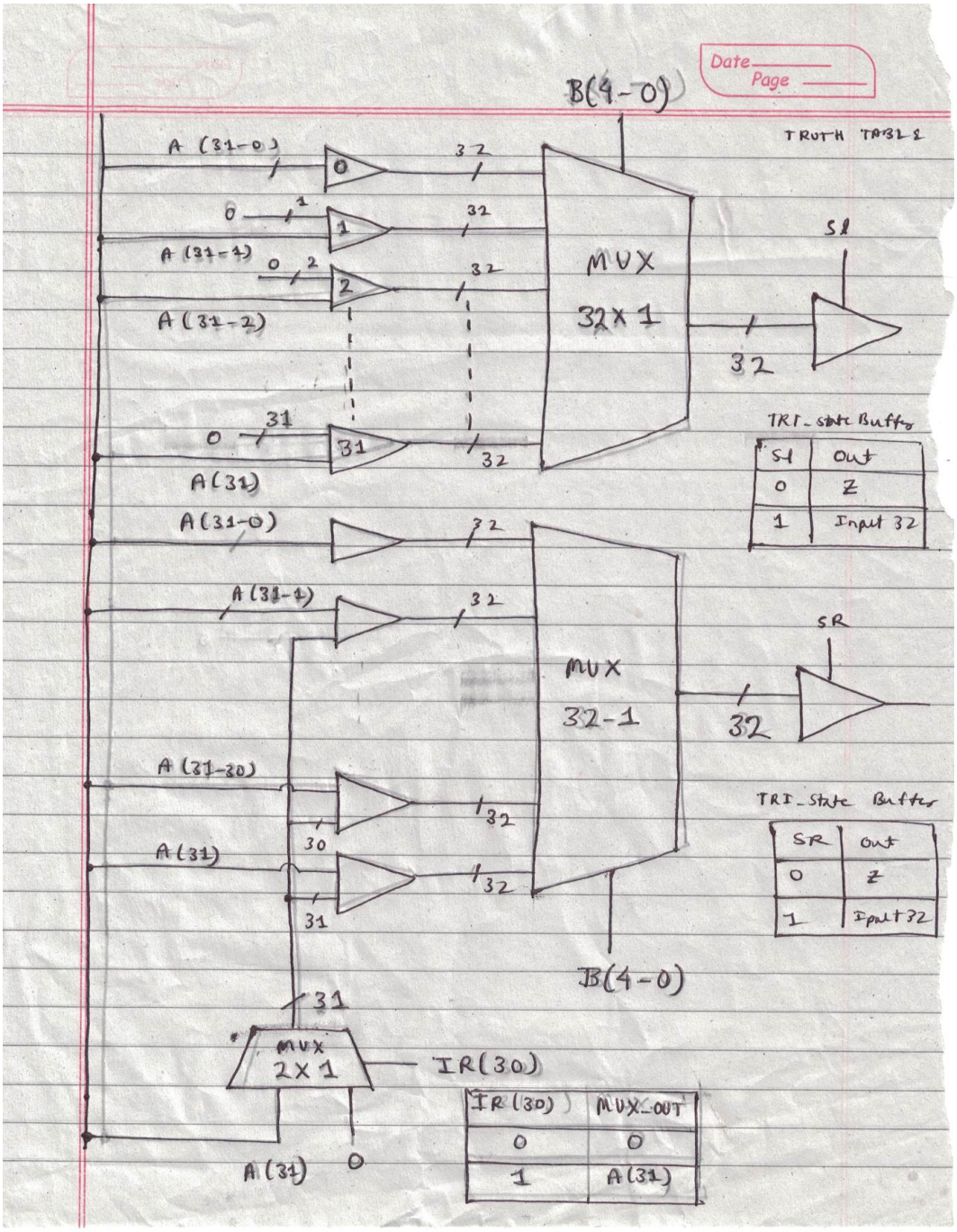
The instruction(30) is the 30th bit of the institution which encoded the arithmetic operation as either addition or subtraction, arithmetic or logical shift etc.

0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srli rd,rs1,shamt
0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srai rd,rs1,shamt
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	add rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	sub rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	R-type	sll rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1	R-type	slt rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1	R-type	sltu rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	R-type	xor rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	srl rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	sra rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	R-type	or rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	R-type	and rd,rs1,rs2

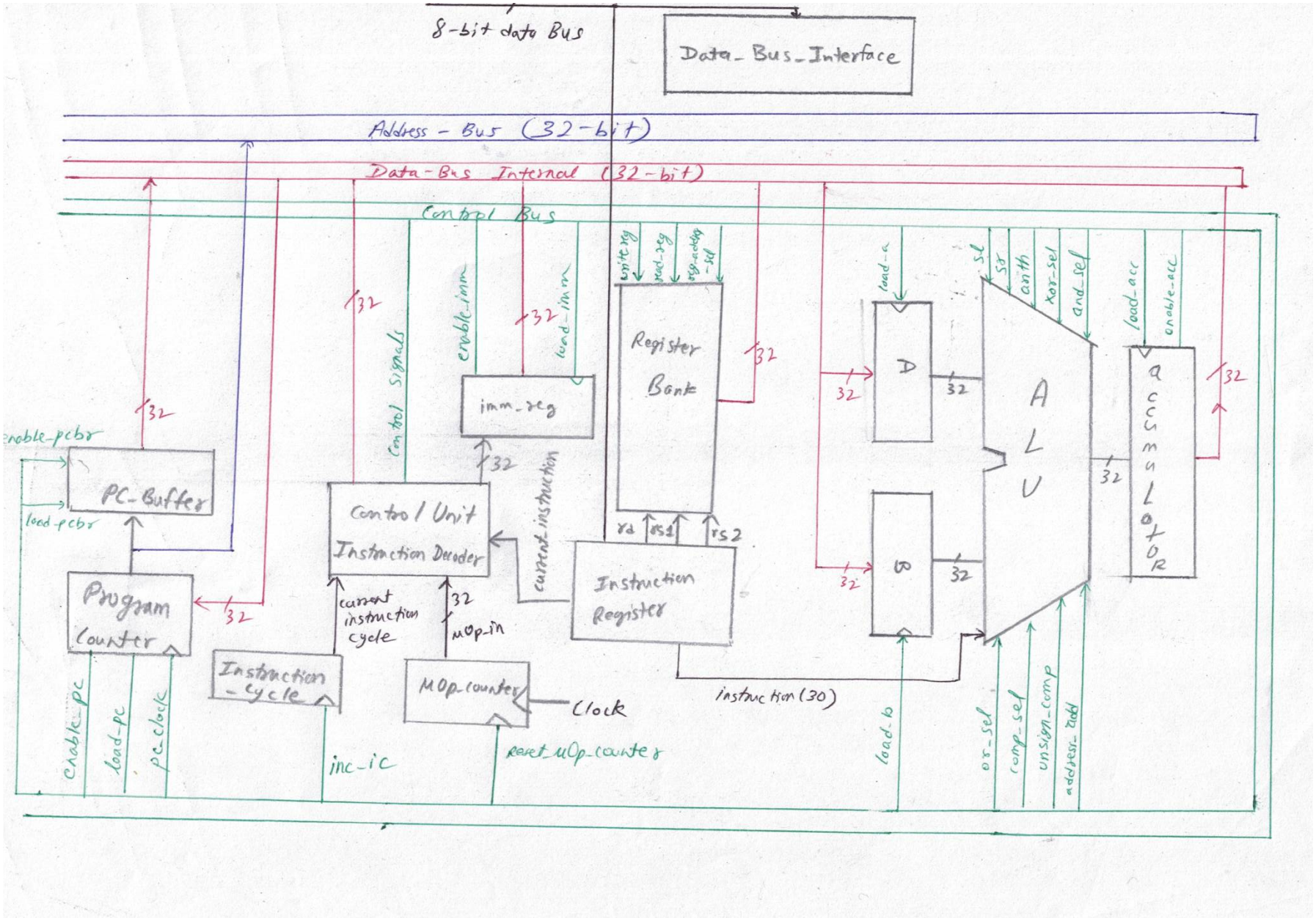
The highlighted part depicts the difference.

The address_add signal is used to ignore the instruction(30) and is used for addition only. This is used for jump type instructions where the relative address itself is stored as a negative value. The sifter is implemented as an array of tri-state buffers instead of the

flop flop as shifting a word by 32 times requires lots of cycles. This approach requires more resources on the chip but it saves lots of shifting time.



From the diagram it is clear that the shifting operation does not depend on the flip flop but rather on the buffer array. Where instruction(30) is 1 the arithmetic shift operation is performed.


8-bit data Bus
Data_Bus_Interface
Address - Bus (32-bit)
Data-Bus Internal (32-bit)
Control Bus
32
control signals
enable_imm
load_imm
write_reg
read_reg
reg_address_sel
Register Bank
imm_reg
load_a
sl
sr
arith
xor_sel
and_sel
load_acc
enable_acc
D
W
ALU
accumulator
enable_pcbr
load_pcbr
PC_Buffer
Program Counter
Control Unit
Instruction Decoder
current_instruction
Instruction Register
rd
rs1
rs2
current instruction cycle
uOp_in
Instruction_cycle
uOp_counter
Clock
instruction (30)
enable_pc
load_pc
pc_clock
inc_ic
reset_uOp_counter
load_b
os_sel
comp_sel
unsign_comp
address_add

The figure above shows the entire organization of the RV32I microarchitecture. The Program counter performs the same operations as every other program counter. It points to the address location of the current instruction. The program counter buffer stores the address of the current instruction being executed. Since RISC-V is little-endian it has to increment the program counter 4 times to fetch all the 32 bits of complete instruction.

If the starting address of instruction addi x1, x0, 100 is 0x000000A0. The PC buffer stores 0x000000A0 while the program counter points to 0x000000A3 where the last Most Significant Byte(MSB) is stored and is necessary to fetch this byte to complete the 32 bit instruction.

The instruction counter keeps track of the current instruction cycles. The instruction cycles are instruction fetch and instruction decode and execute.

The instruction fetch cycle:

The instruction is fetched in this cycle according to little-endian.

The instruction decode and execute cycle:

The instruction is decoded and executed in this cycle.

The micro-operation(uOP) counter is used to counter the micro-operation that the current instruction is in. The execution of an instruction requires various little micro-operation such as enabling the accumulator, selecting the source register etc, hence, these atomic operations are executed for one clock pulse performing the smallest task that can be done to execute a complete instruction in a certain number of micro-operations. The maximum number of micro-operation for the decode and execute instruction cycle is 32. Only the load word(LW) instruction utilizes the maximum micro operations during the decode and execute instruction cycle. All the instructions use 13 micro-operation cycles for the instruction fetch cycle. Hence the maximum time to execute a instruction is (13+32)*clock period.

SIMULATION RESULTS

To simplify the evaluation process I'm using the online RISC-V assembler as well the RISC-V interpreter.

```
1 .global _boot
2 .text
3
4 _boot:                        /* x0  = 0     0x000 */
5     /* Test ADDI */
6     lui x1, 0x12345
7     auipc x1, 0x10001
8     jal x1, boot
```

Hex Dump

```
123450b7
10001097
ff9ff0ef
```

Objdump

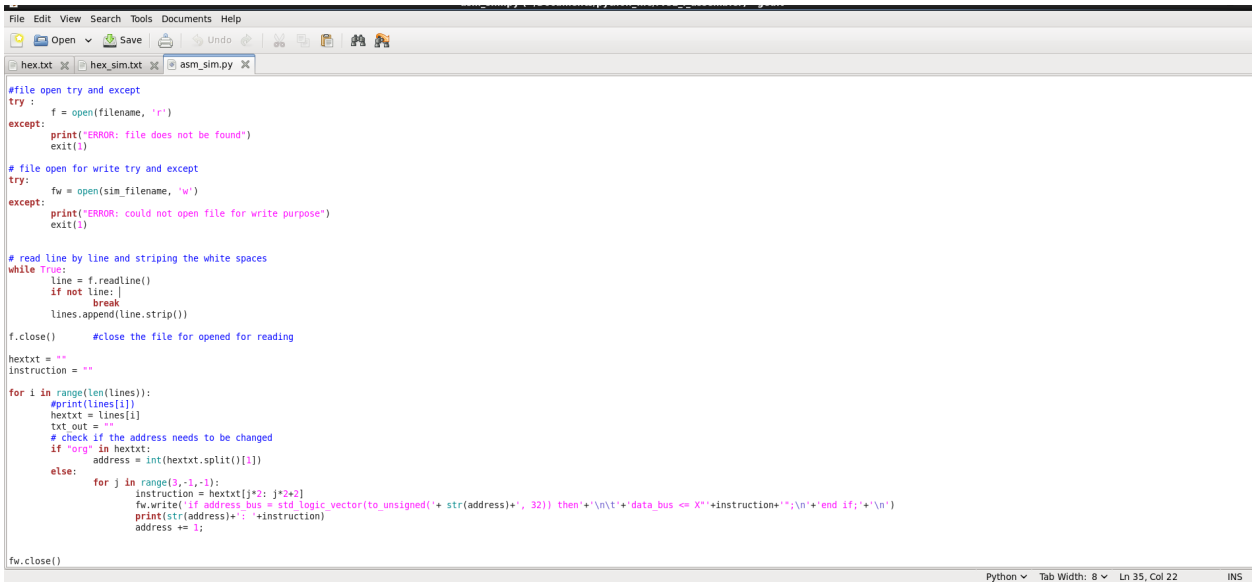
```
file.elf:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000000 <_boot>:
   0:   123450b7                lui     ra,0x12345
   4:   10001097                auipc   ra,0x10001
   8:   ff9ff0ef                jal     ra,0 <_boot>
```

The above figure shows the online RISC-V assembler. Although the jump instruction is not correctly assembled for RV32I every other code is converted to machine code correctly,

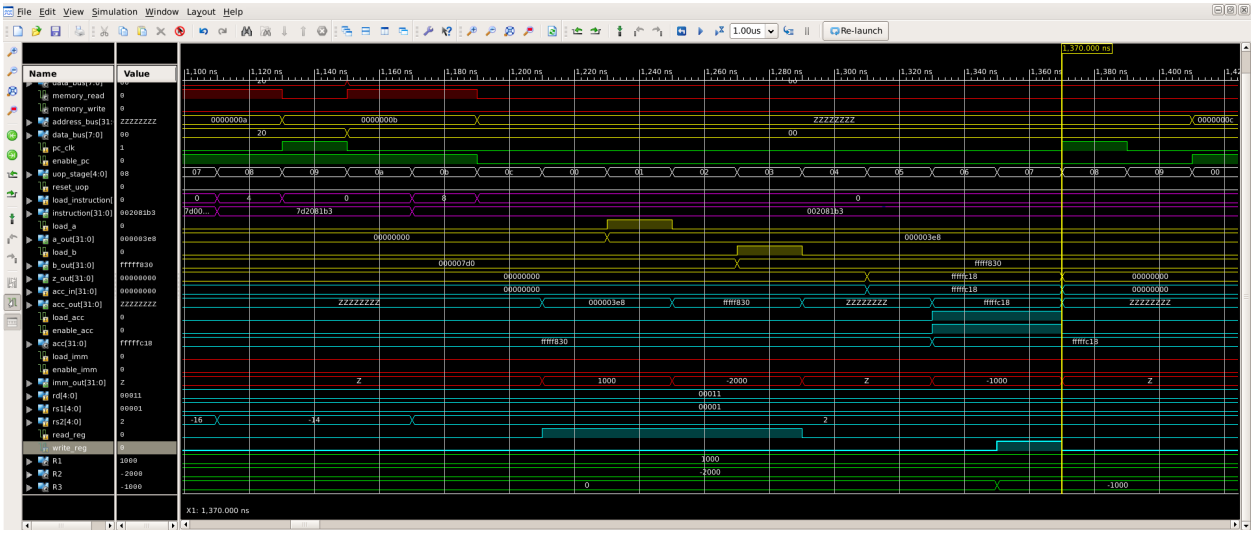
For simulation purposes I have also written a python program which converts the hex machine code into test bench code that simulates the memory operation.



R-type and I-type arithmetic instructions

The instruction in the R-TYPE formats are as follows:

- ADD
- SUB
- SLL
- SLT
- SLTU
- XOR
- SRL
- SRA
- OR
- AND

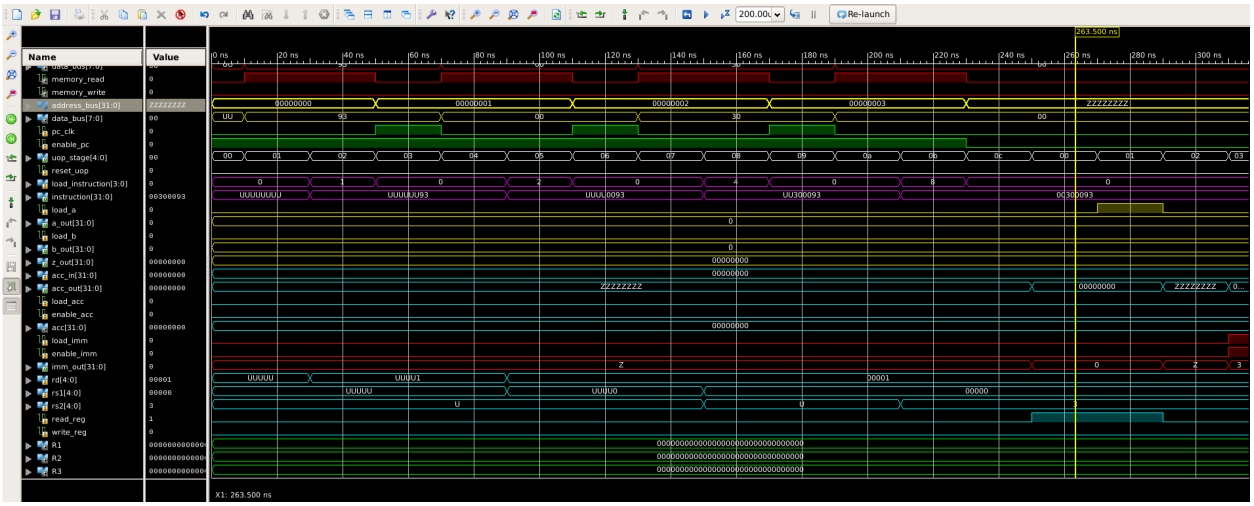


The above diagram shows the **add x3,x2,x1** operation where x2 and x1 are set to signed values of -200 and 100 respectively.

For complete R-type instruction simulation is available in the video: https://youtu.be/ynKmTSlXjaM

The arithmetic instructions in the I-type format are as follows:

- ADDI
- SLTI
- SLTIU
- XORI
- ORI
- ANDI
- SLLI
- SRLI
- SRAI



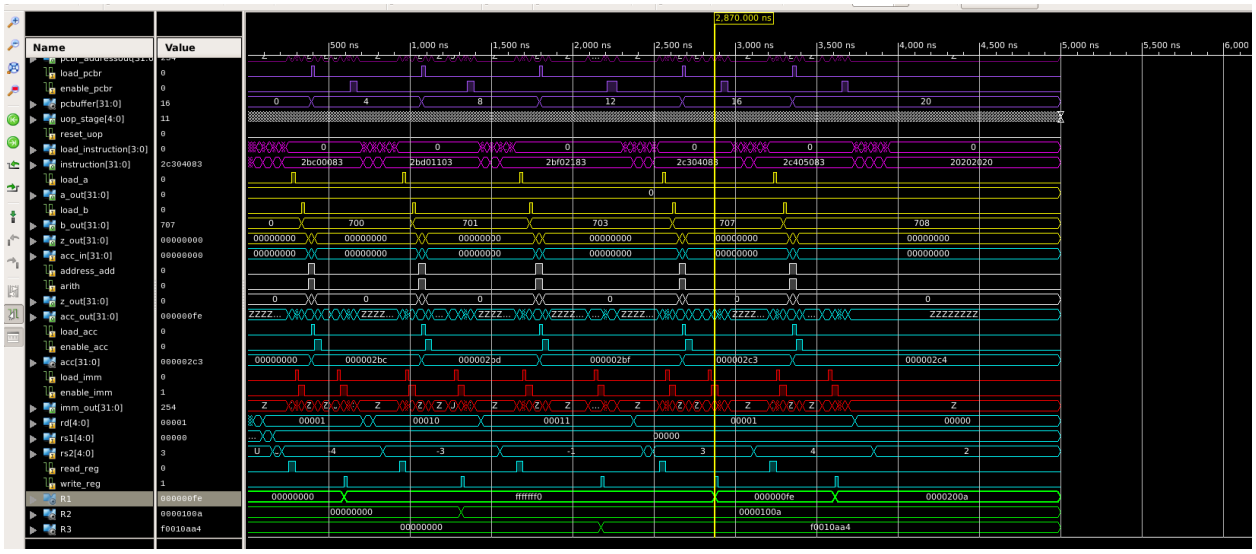
The figure shows the I-type instruction cycle.
[The complete i type instruction simulation is available in the video](https://youtu.be/fqTNZXLOyow): https://youtu.be/fqTNZXLOyow

I-tyle load instructions

The i type load instruction are:

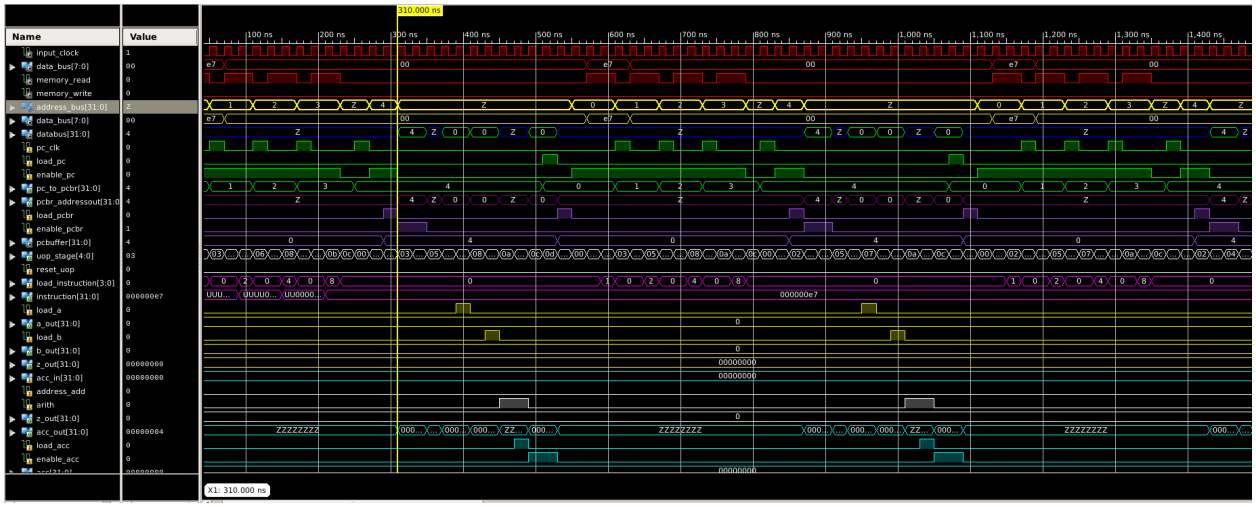
- LB
- LH
- LW
- LBU
- LHU

These instructions are used to load a sign extended byte, half word(2 bytes), zero extended byte, half word and word from the memory location given by the immediate field in little-endian.



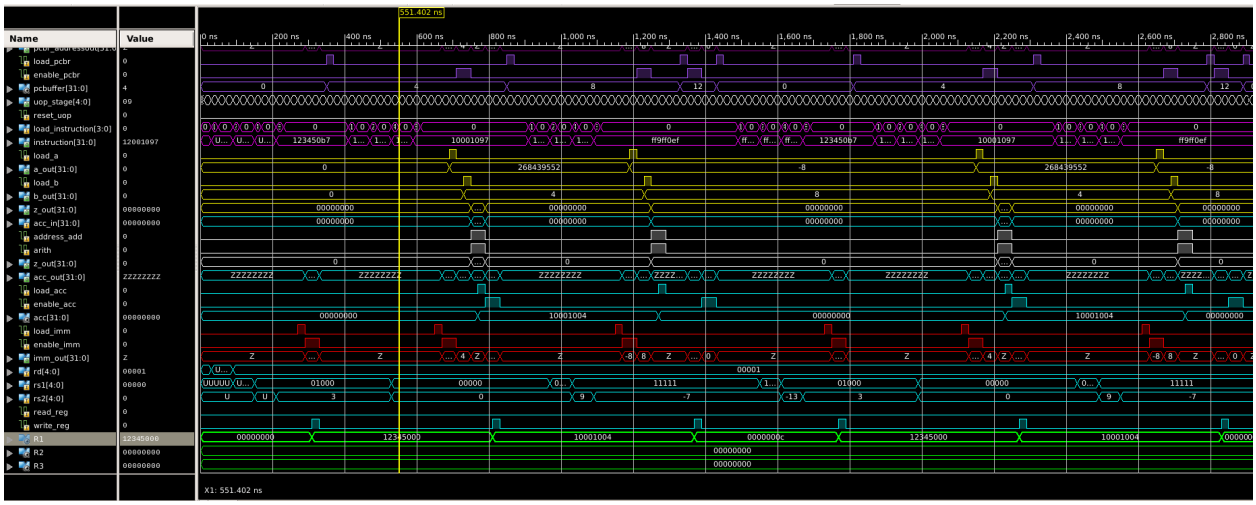
The figure shows the execution of **ld x1, 700** where the data from address 700 is signed extended to 0xffffffff0 and stored in x1 register, **lh x2, 701** where the data from address 701 is signed extended to 0x0000100a and stored in x2 register, **lw x3, 703** where the data from address 703 is 0xf0010aa4 stored to x3 register, **lbu x1, 707** where the data from address 707 is zero extended to 0x000000fe and stored in x1 register and **lhu x1, 708** where the data from address 708 is zero extended to 0x0000200a and stored in x1 register.

I-type jump instruction



The above figure shows the execution of the **JALR x1, -4(x0)**, where the program is jumping to its own address and looping.

U-TYPE AND J-TYPE INSTRUCTION



The figure shows the execution of **LUI x1, 0x12345** such that the upper bytes are zero extended and loaded to the destination register.

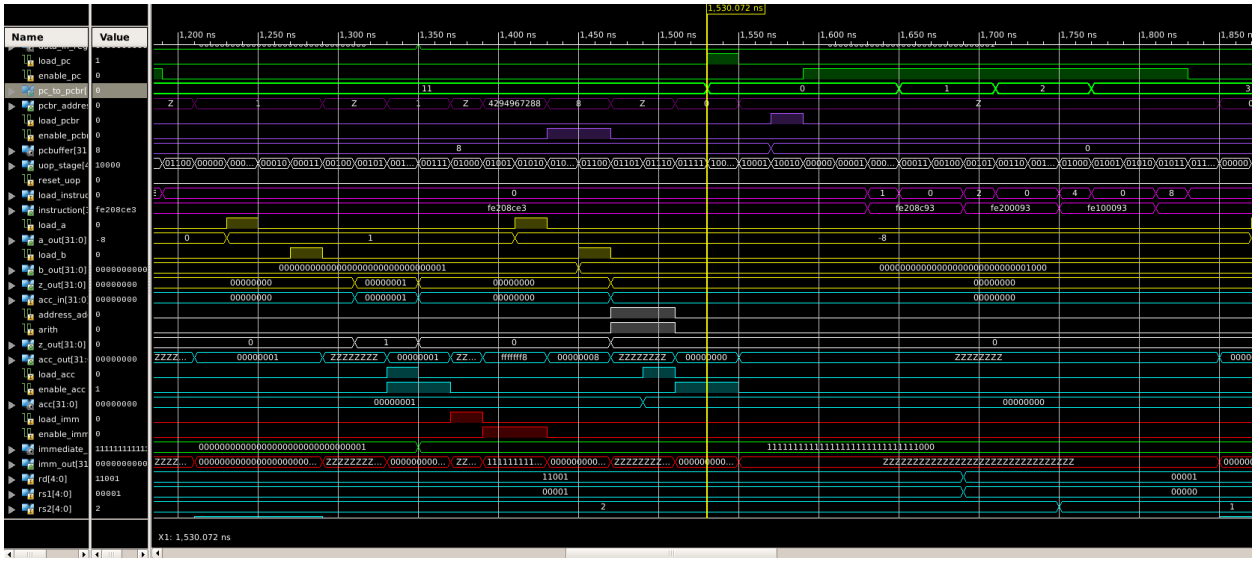
AUIPC x1x, 10001 such that the upper byte is zero extended and added with the address of current instruction which is 0x00000004 hence the result is stored to z1.

And **JAL x1, -8** where the current address is stored to the destination register, in this case it is x1. And the current instruction address is added with the offset address. The obtained result is the new address of current instruction and program jumps to that address.

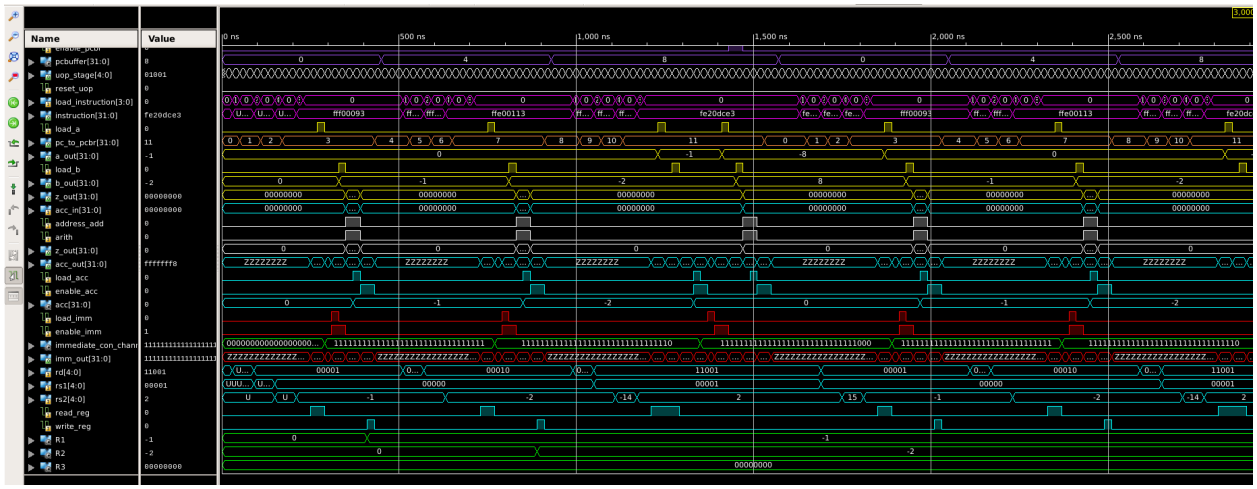
B-TYPE branch instruction

This type of instruction is used for conditional branches. The instruction are as follows:

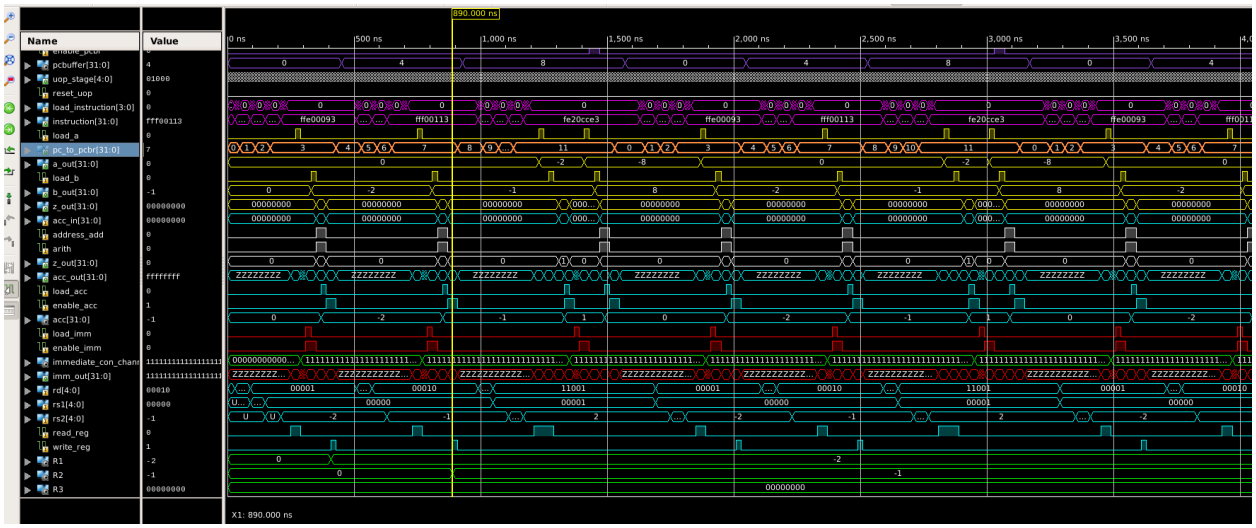
- BEQ
- BNE
- BLT
- BGE
- BLTU
- BGEU



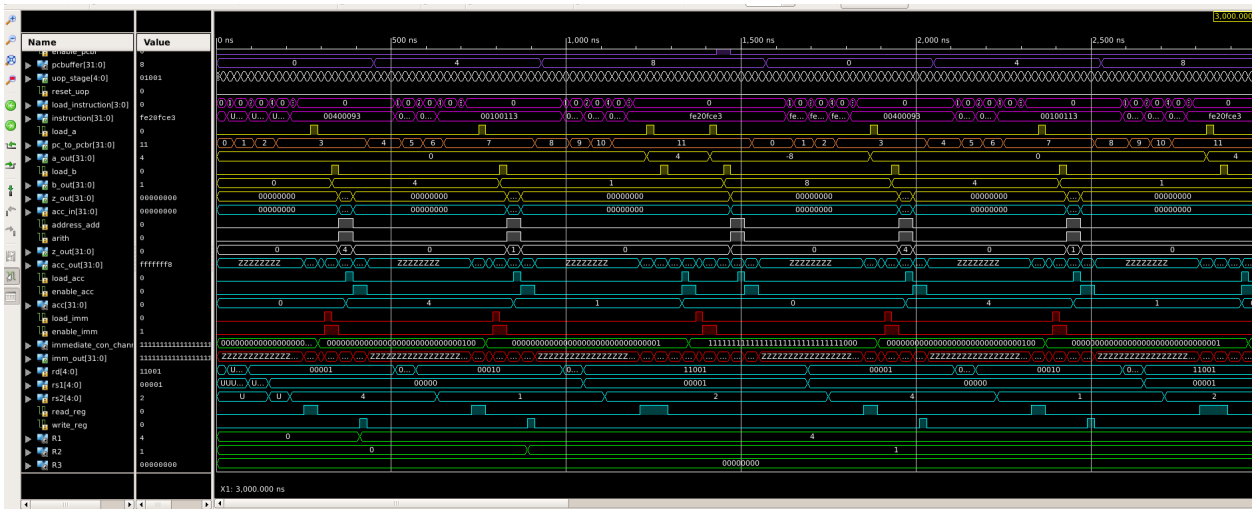
Execution of **BEQ X1,X2,-8.** The value of x1 and x2 are equal hence the program counter is set to 0 as the current instruction address is 8. 8 plus -8 gives a relative address of 0. So, the PC jumps to 0 and loops again.



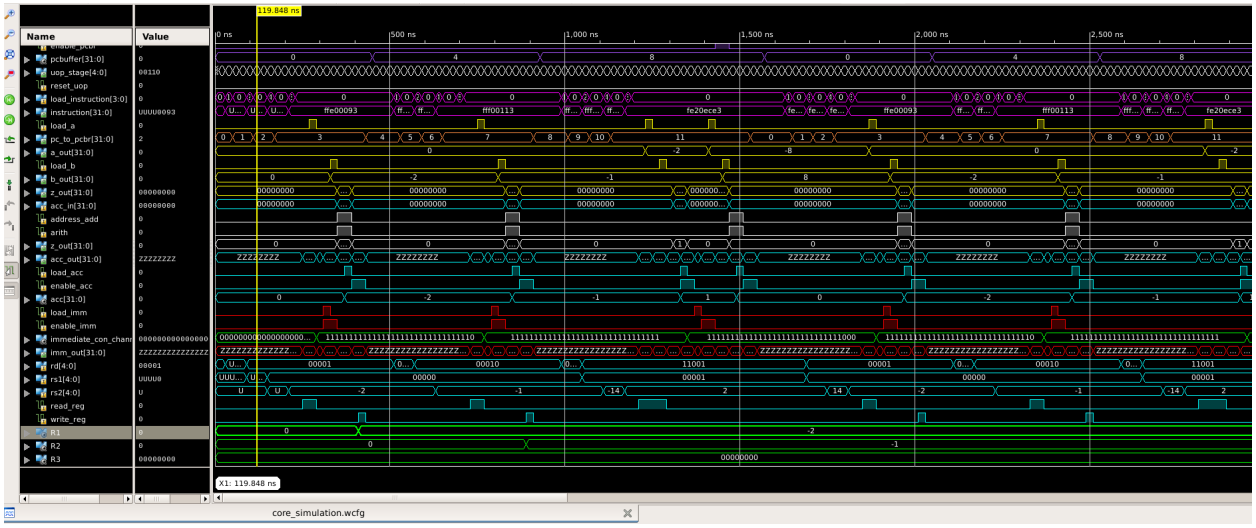
Execution of **BNE X1,X2, -8**. As the value of x1 is -1 and x2 is -2 the condition of jump is satisfied and the program jumps to the relative address of 0.



Execution of **BLT x1, x2,-8.** The value of x1 is -2 which is less than -1 of x2. Hence, the program jumps to the relative address of 0 and loops continuously.



Execution of **BGE x1, x2,-8.** The value of x1 is 4 which is less than 1 of x2. Hence, the program jumps to the relative address of 0 and loops continuously.



Execution of **BLTU x1, x2,-8.** The unsigned value of x1 is less than of x2. Hence, the program jumps to the relative address of 0 and loops continuously.

Other module simulations

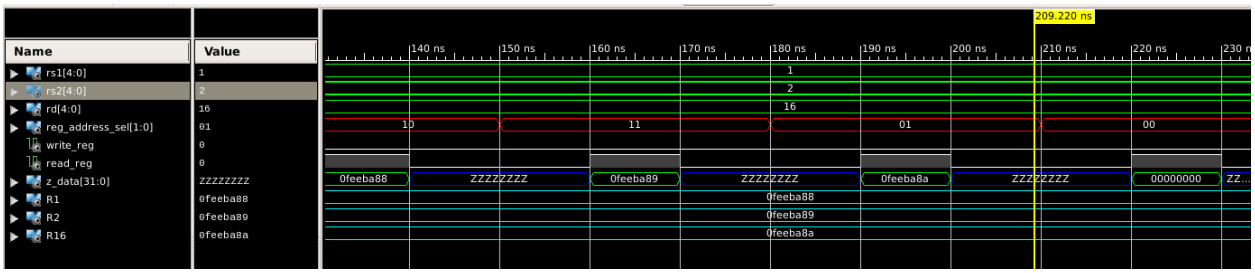


figure:Register bank read simulation

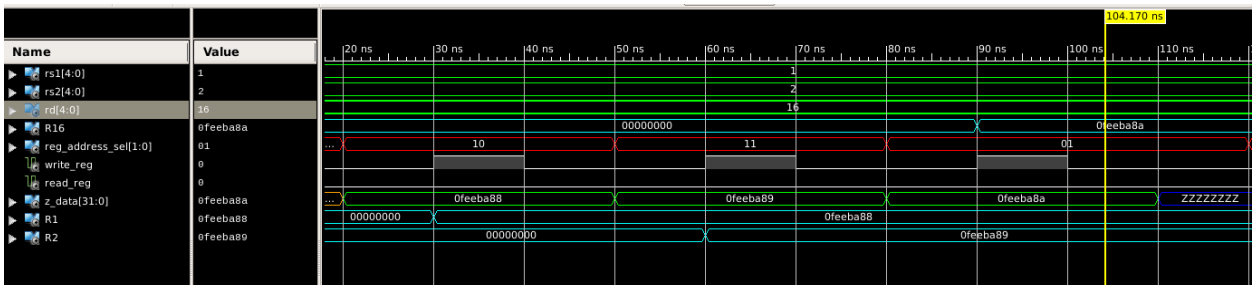


Figure: register bank write simulation

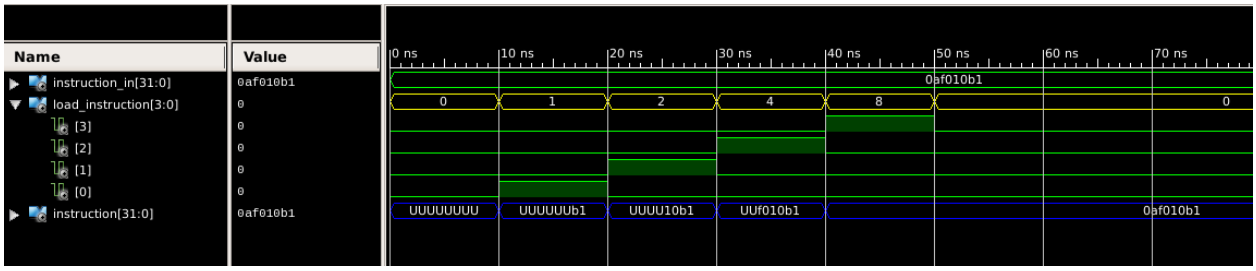


Figure: loading instruction on instruction register

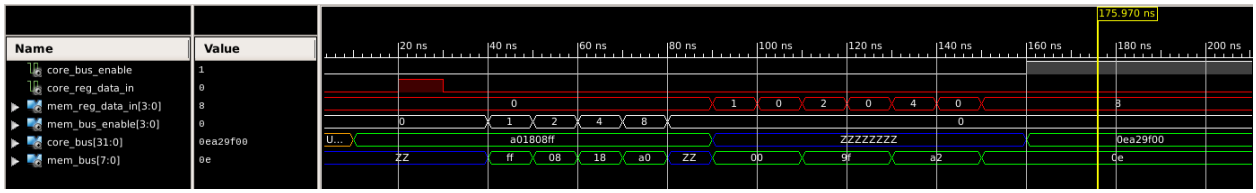


Figure: internal 32 bit to external 8 bit data bus interface

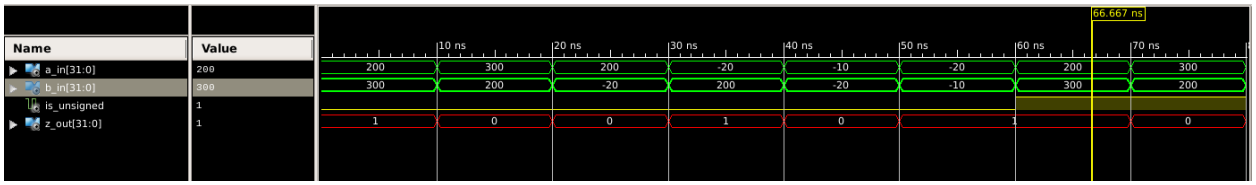


Figure: signed and unsigned comparator implementation

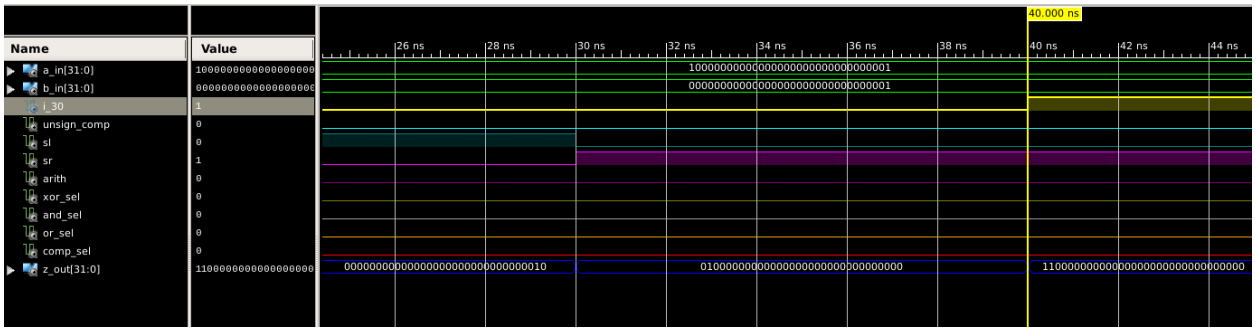


Figure: Shifting operation in ALU