IITB - Proc CPU Design

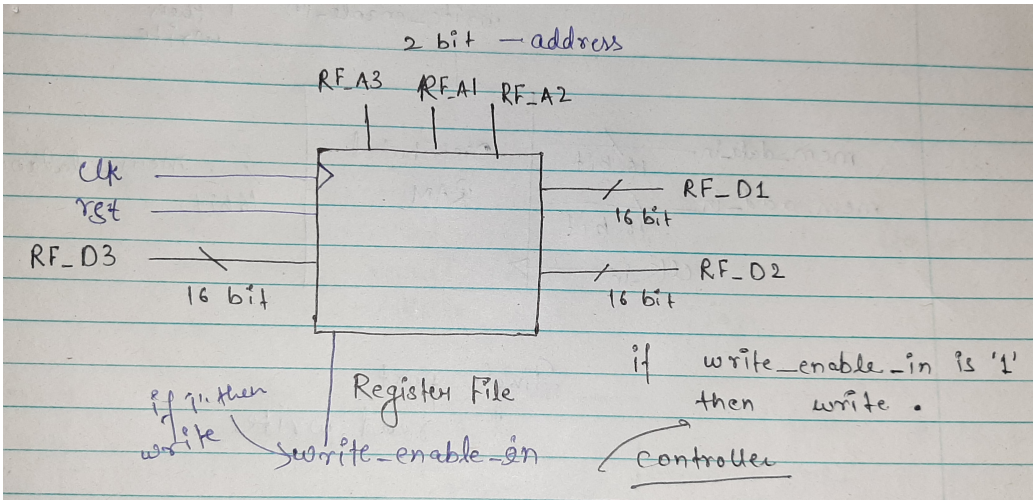
190050096 - 190050101 - 190050103 - 190050116

Sunday 23^{rd} May, 2021

IITB-Proc: a multi-cycle processor

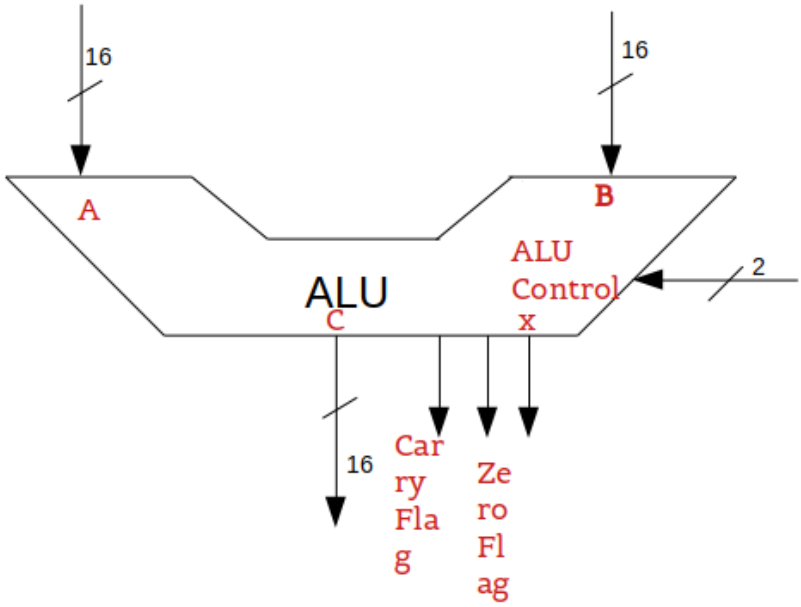
Note: We faced difficulties for $2^{16} * 16$ because devices installed in quartus doesn't have dedicated ram. So, TA suggested me to reduce ram size because of few instructions. So I use 2047 downto 0 of 15 downto 0 in my project.

§1. Register file



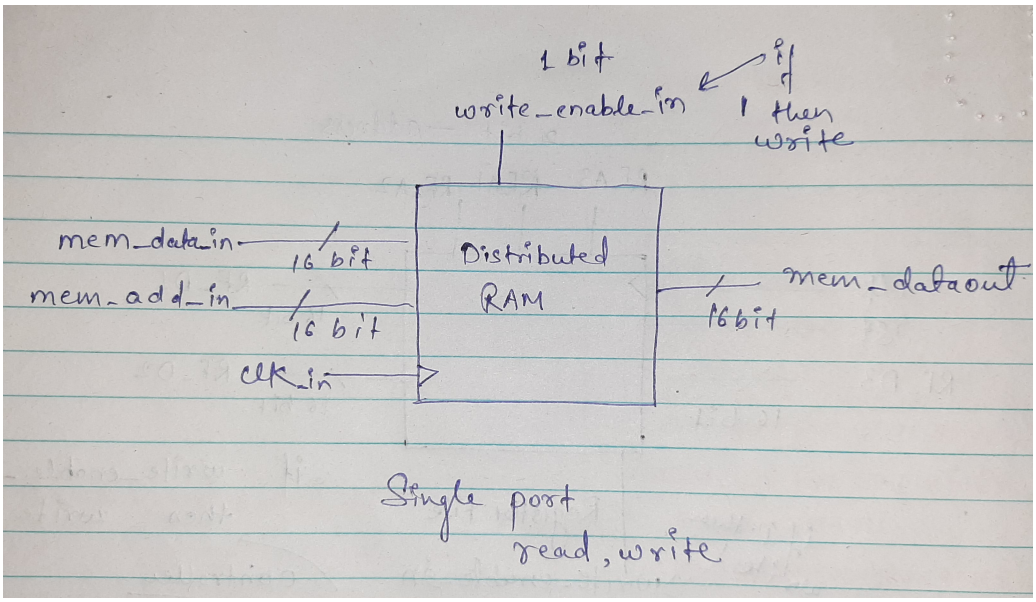
The register file consists of eight 16-bit registers. The RF stores the data provided at the port RF_D3 into location provided at port RF_A3, at every rising edge of clock cycle.
The read operation has been implemented as a combinational logic and write operation is implemented as behavioral logic using process statements.

§2. Arithmetic Logic Unit (ALU)



- 16 bit inputs ALU_A , ALU_B and a two bit alu_control input which determines what operation to perform and give ALU_C (16 bit output).
  alu_control = "00" (addition operation), "01" (NAND operation), "10" (XOR operation)
- carry_flag_c (store carry of addition), zero_flag_z (set to 1 if ALU_C = x"0000"), x (1 if ALU_C = x"0008") for LA/SA condition.

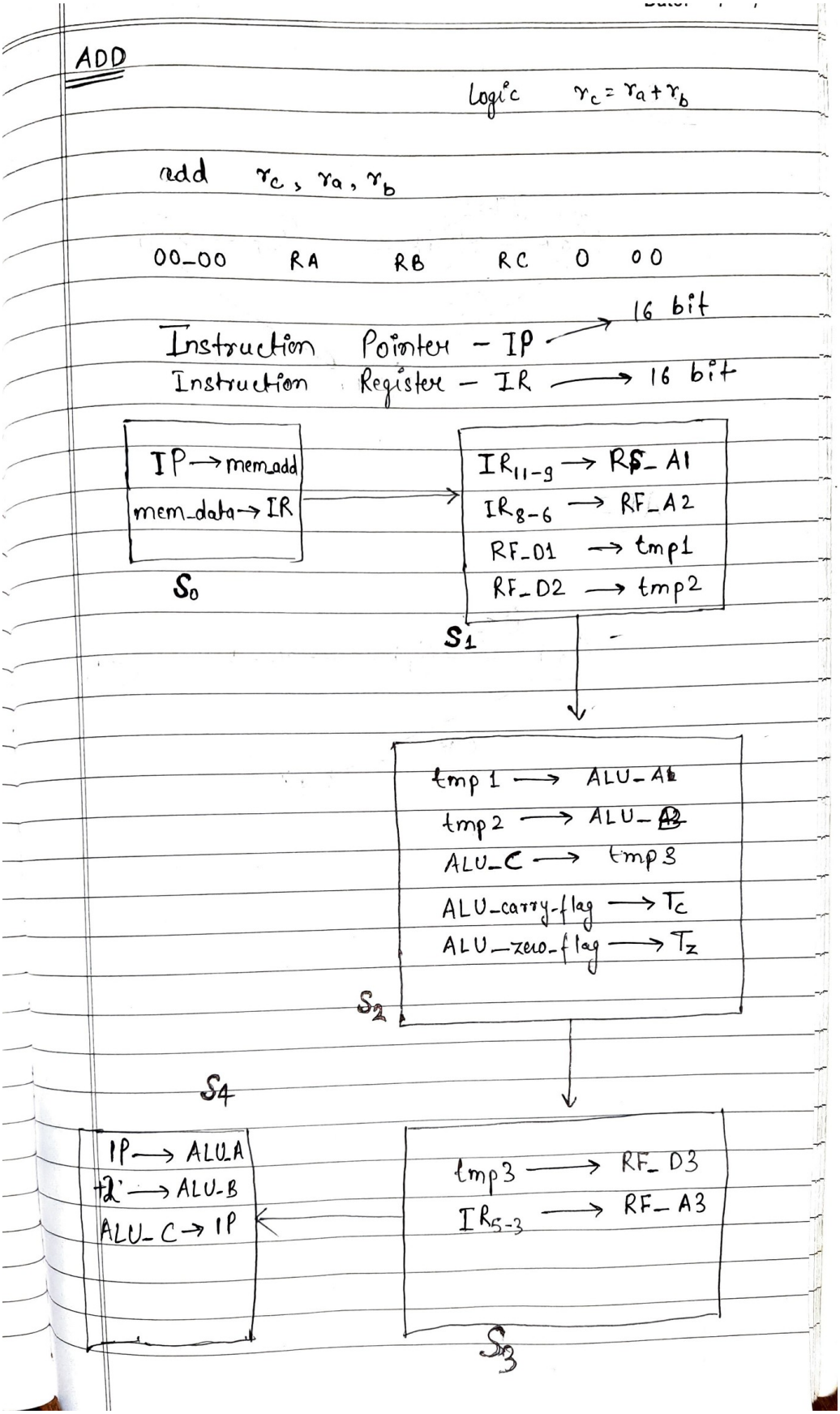
§3. Memory Unit



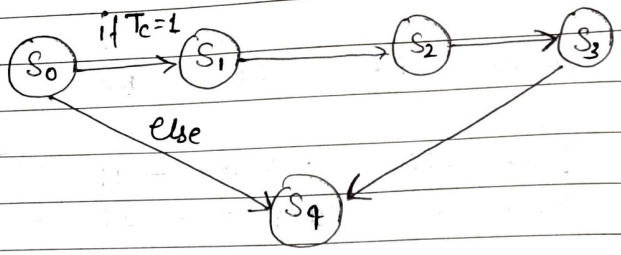
The read operation has been implemented as a combinational logic and write operation is implemented as behavioral logic using process statements.

§4. Main Unit & FSM

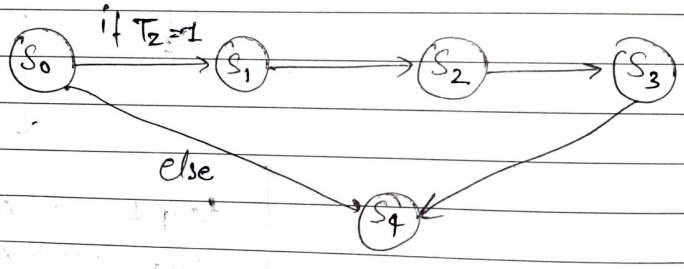
We have implemented the finite statemachine in this part using behavioural VHDL logic.
FSM



ADC



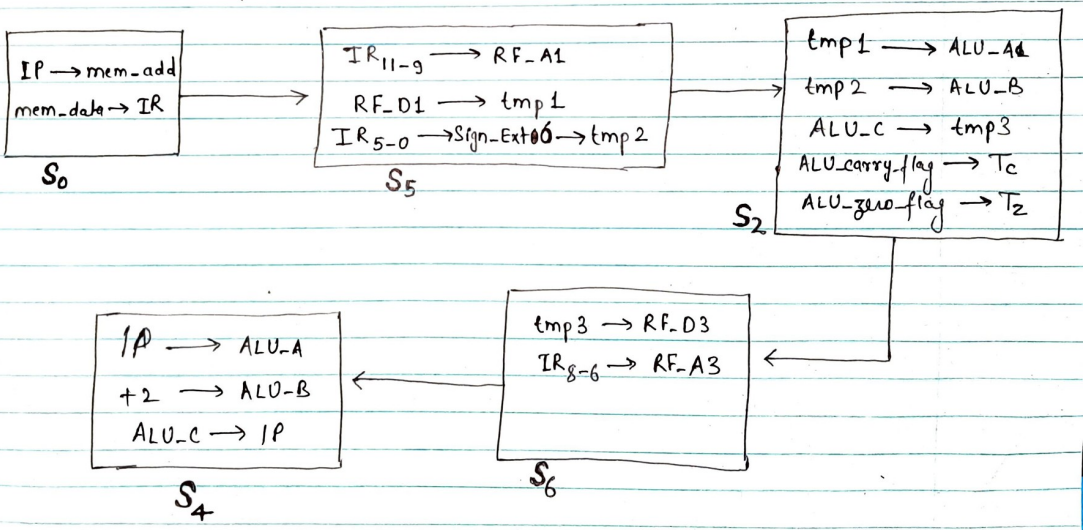
ADZ

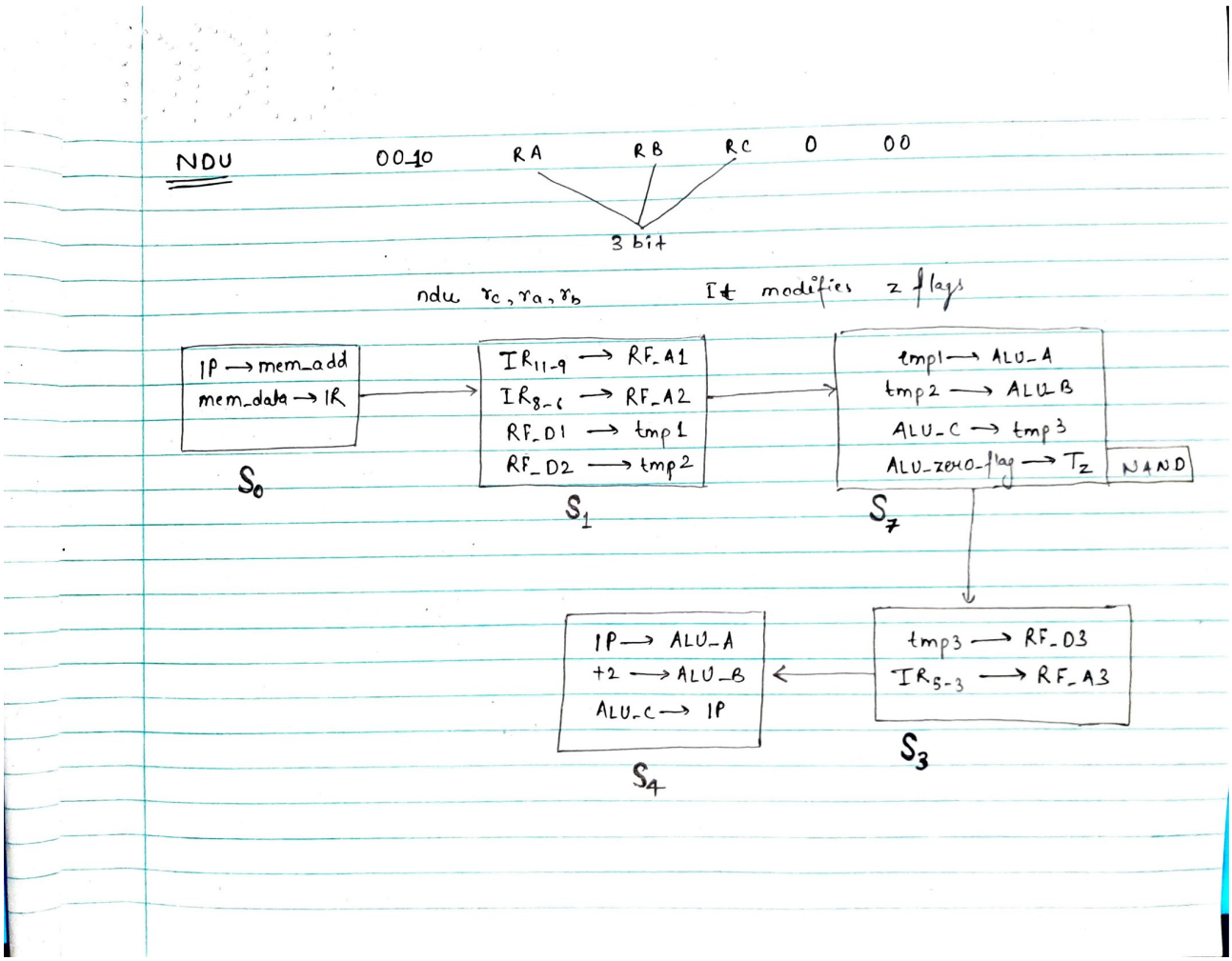


ADI Add Immediate (1)

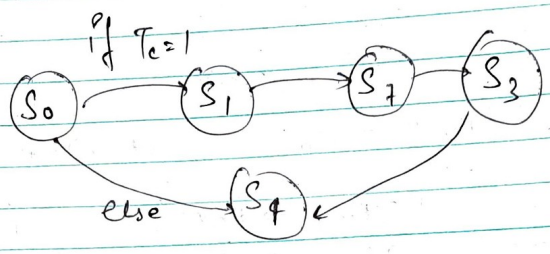
00_01 RA RB 6 bit imm - edate | add rb, ra, imm 6

Add content of regA with imm (sign extended) and store result in regB.
It modifies C and Z flags

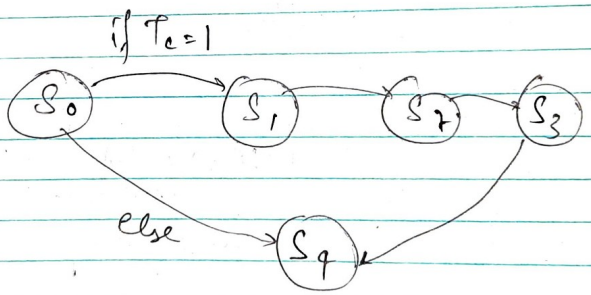



NDU
0010
RA
RB
RC
0
00
3 bit
ndu rc, ra, rb
It modifies z flags
IP → mem_add
mem_data → IR
S0
IR11-9 → RF_A1
IR8-6 → RF_A2
RF_D1 → tmp1
RF_D2 → tmp2
S1
tmp1 → ALU_A
tmp2 → ALU_B
ALU_C → tmp3
ALU_zero_flag → Tz
NAND
S7
tmp3 → RF_D3
IR5-3 → RF_A3
S3
IP → ALU_A
+2 → ALU_B
ALU_C → IP
S4

NDC



NDZ

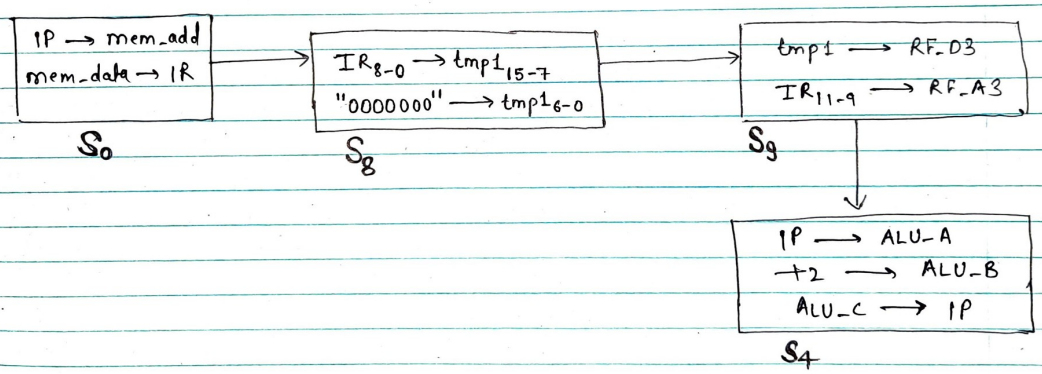


LHI 00_11 RA 9bit Immediate
4bit 3bit

Load Higher Immediate

lhi, ra, Imm

Place 9 bits immediate into msb 9 bits of register A and lower 7 bits are assigned to zero.



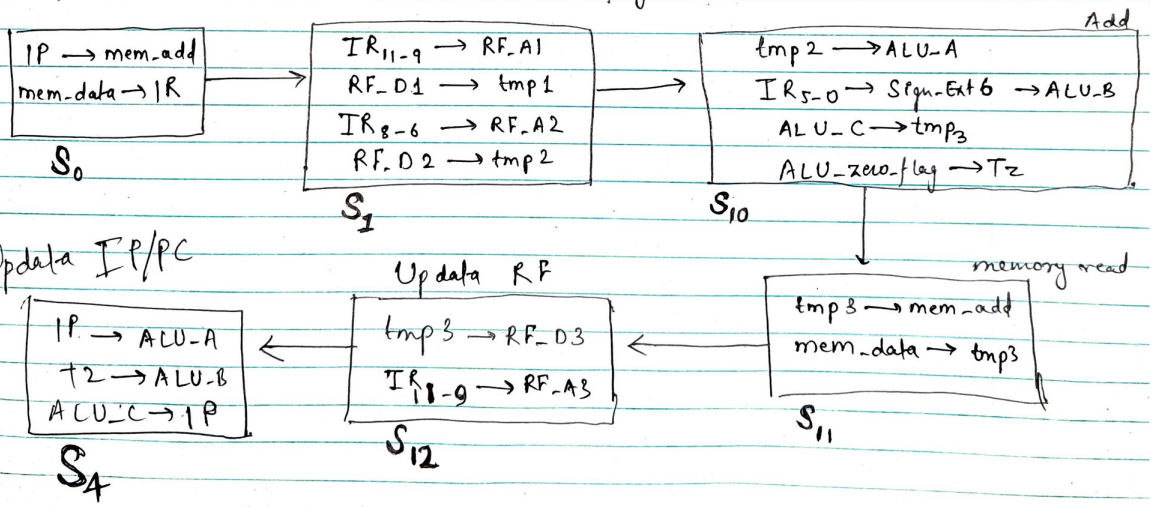
LW lw ra, rb, Imm

Load 01_00 RA RB 6 bit Immediate

3 bit

Load value from memory into regA. Memory address is computed by adding immediate 6 bits with content of regB.

* It modifies z flag.



SW

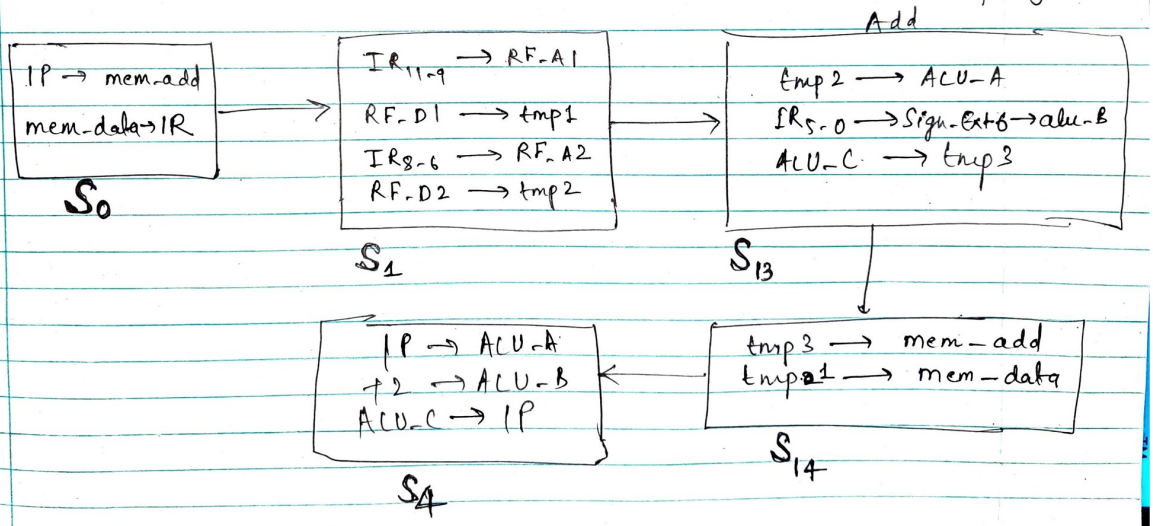
Store

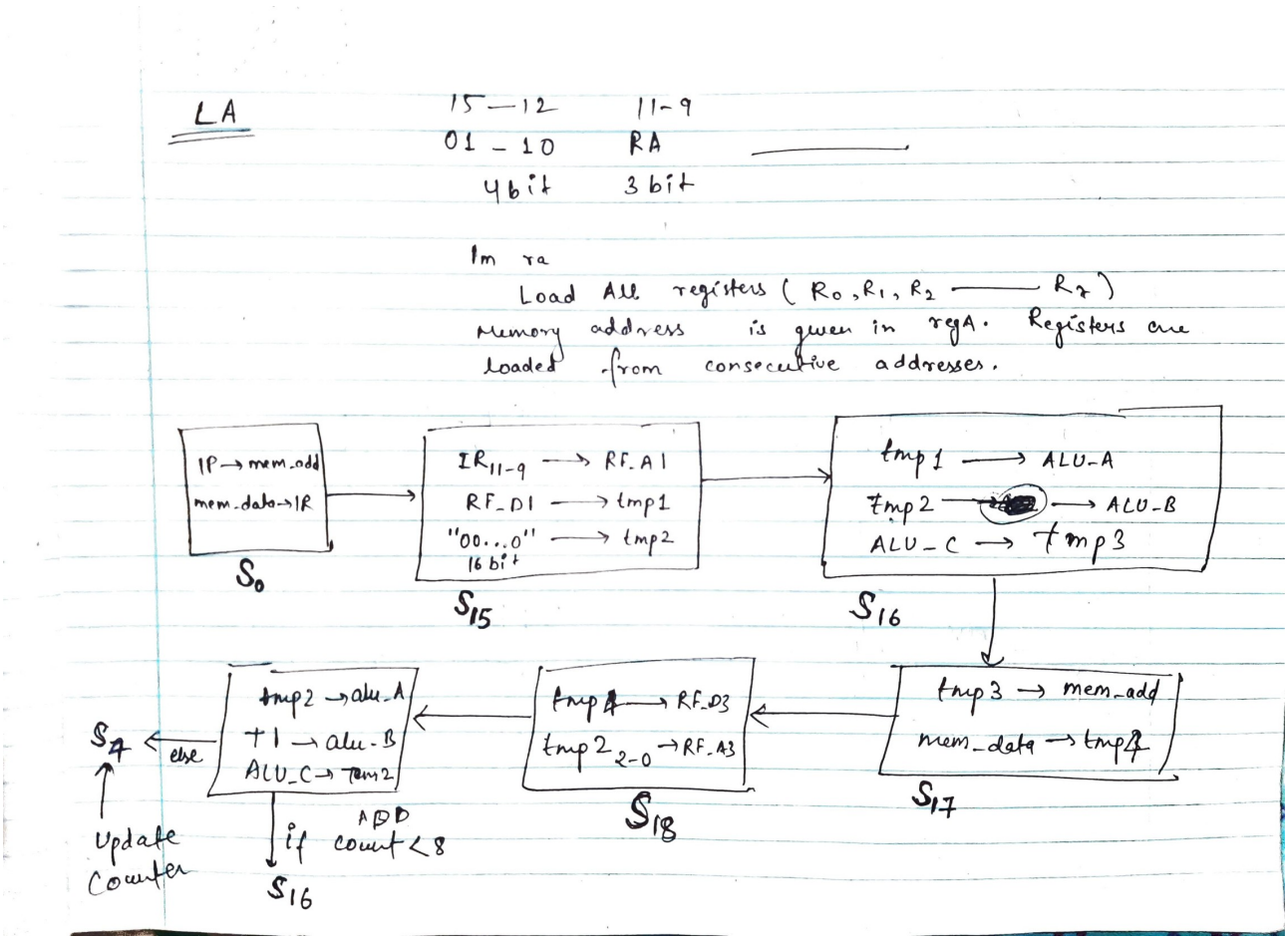
01_01 RA RB 6bit Immediate

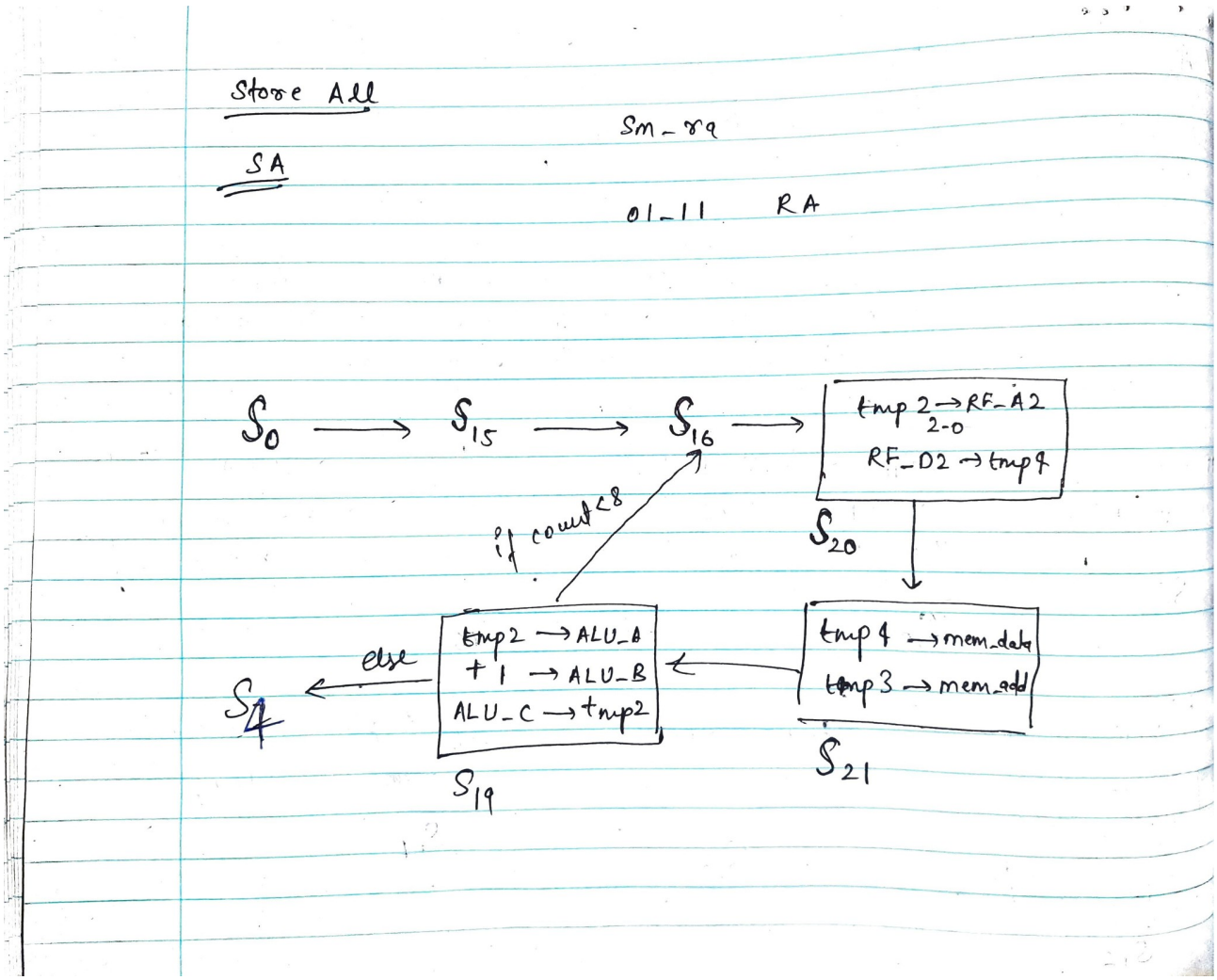
3 bit

SW r_a, r_b, Imm

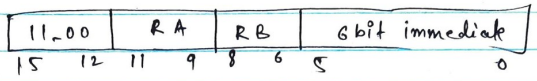
Store value from reg A into memory.
Memory address is formed by adding immediate 6 bits with content of reg B.




LA
15—12 11-9
01 - 10 RA
4bit 3bit
lm ra
Load All registers (R0, R1, R2 —— R7)
Memory address is given in regA. Registers are
loaded from consecutive addresses.
IP → mem_add
mem_data → IR
S0
IR11-9 → RF_A1
RF_D1 → tmp1
"00...0" → tmp2
16 bit
S15
tmp1 → ALU-A
tmp2 → ALU-B
ALU_C → tmp3
S16
tmp3 → mem_add
mem_data → tmp4
S17
tmp4 → RF_D3
tmp2 2-0 → RF_A3
S18
tmp2 → alu_A
+1 → alu_B
ALU_C → tmp2
else
S4
Update Counter
ADD
if count < 8
S16


Store All
SA
Sm_rq
01_11
RA
S_0
S_{15}
S_{16}
tmp 2 → RF_A2 2-0
RF_D2 → tmp 4
S_{20}
if count<8
tmp 4 → mem_data
tmp 3 → mem_add
S_{21}
tmp2 → ALU_A
+1 → ALU_B
ALU_C → tmp2
S_{19}
else
S_4

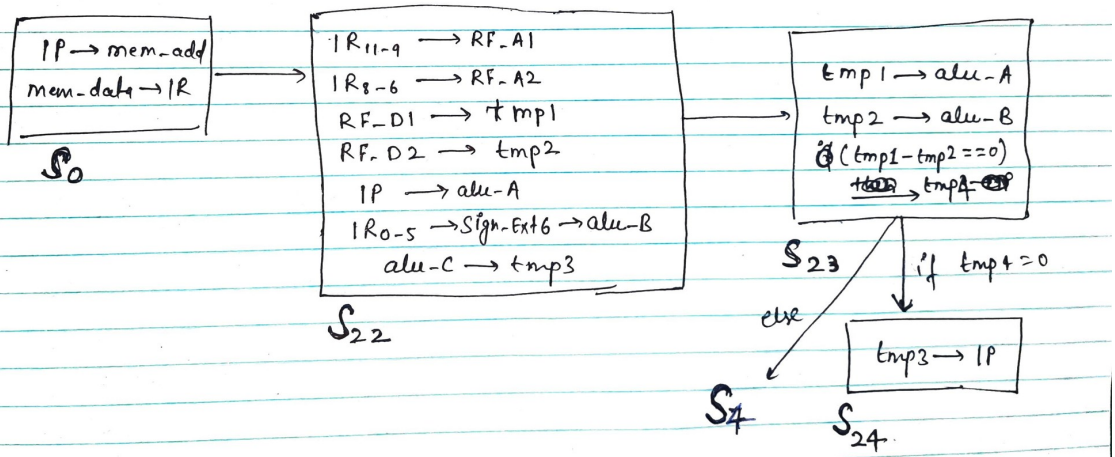
BEQ instruction



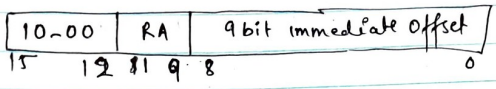
Branch on equality

beq ra, rb, imm

If content of regA and regB are same, branch to PC+imm, where PC is the address of beq
(IP+imm) (IP)



JAL :

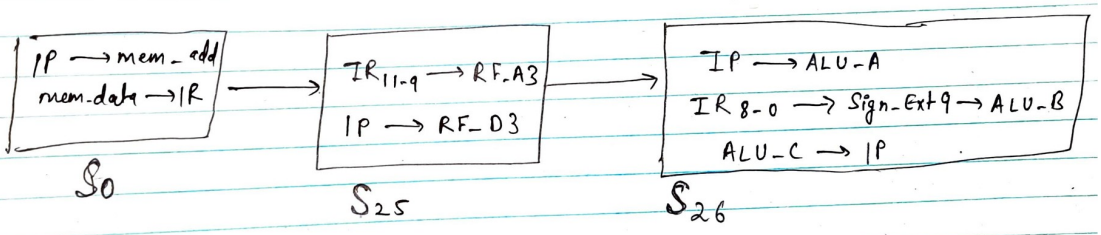


Jump and Link

jalr , ra , Imm

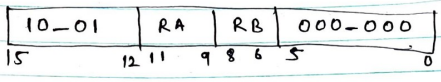
Branch to address IP + Imm (PC + Imm)

Store PC (IP) into reg A , where PC (IP) is the address of jalr instruction



S_0 S_{25} S_{26}

JLR :

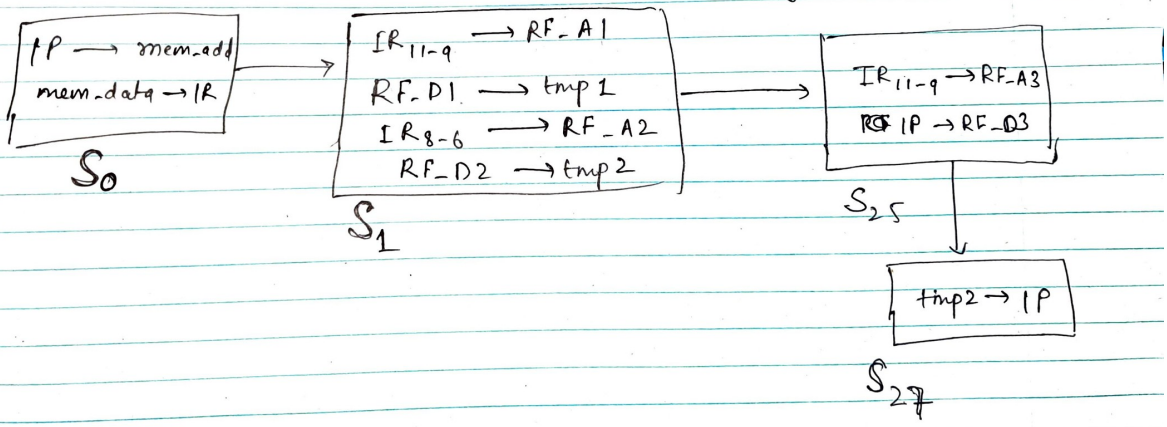


Jump and Link Register

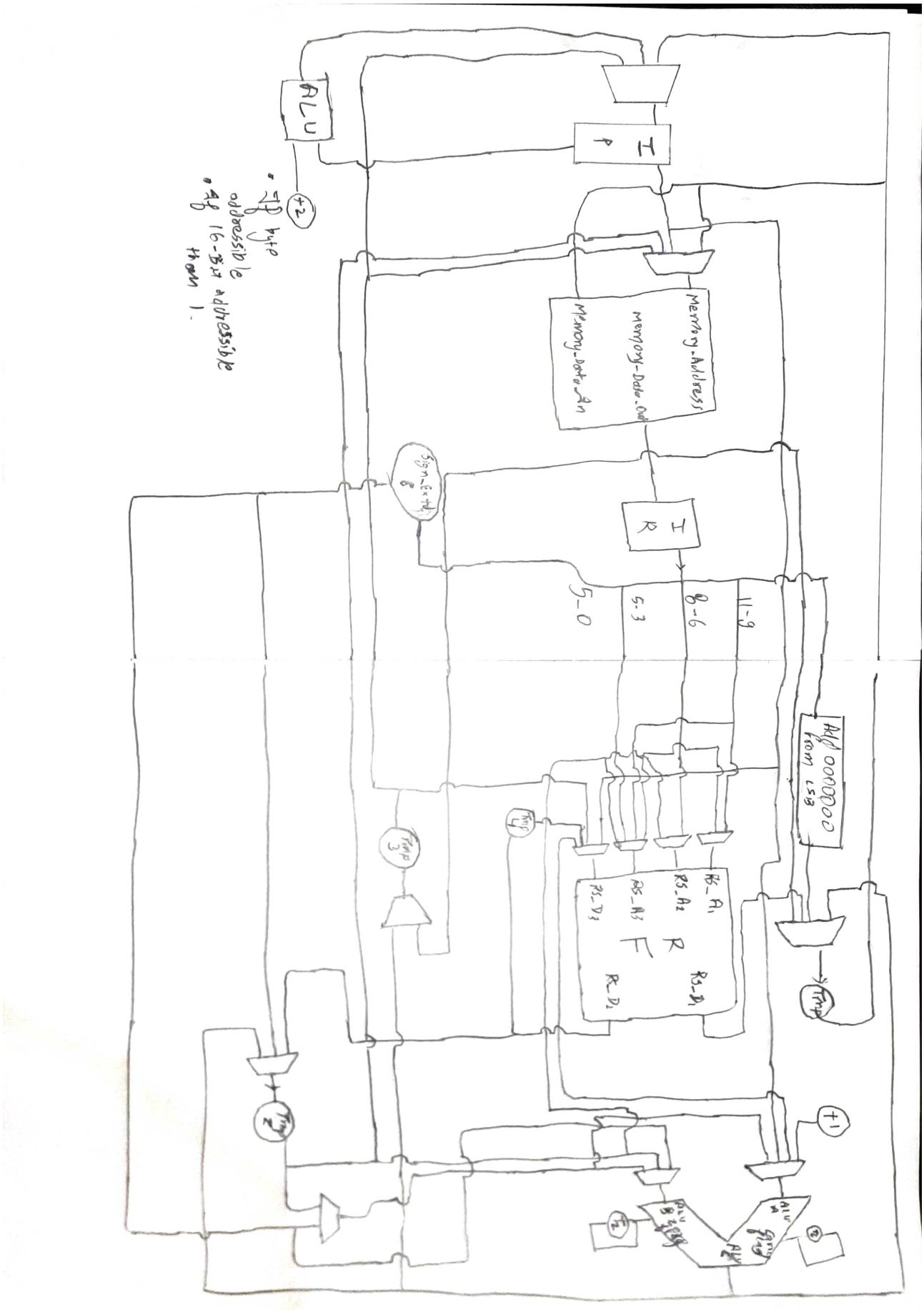
jalr r_a, r_b

Branch to the address in reg B

Store PC (IP) into reg A, where PC (IP) is the address of the jalr instruction.



Circuit Diagram

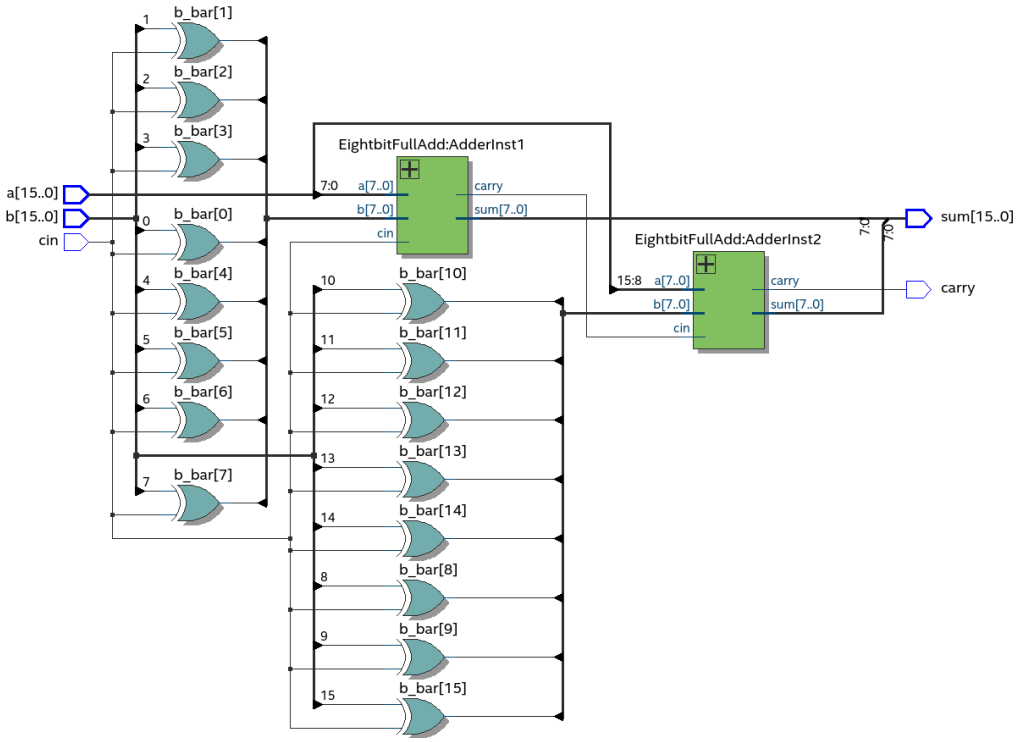


Testing of VHDL code in the Project

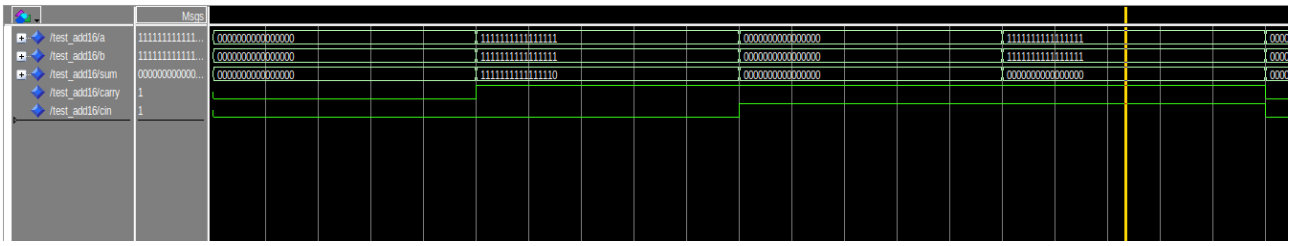
Testing of Component

1. `SixteenbitFullAdd`

   a. RTL Viewer



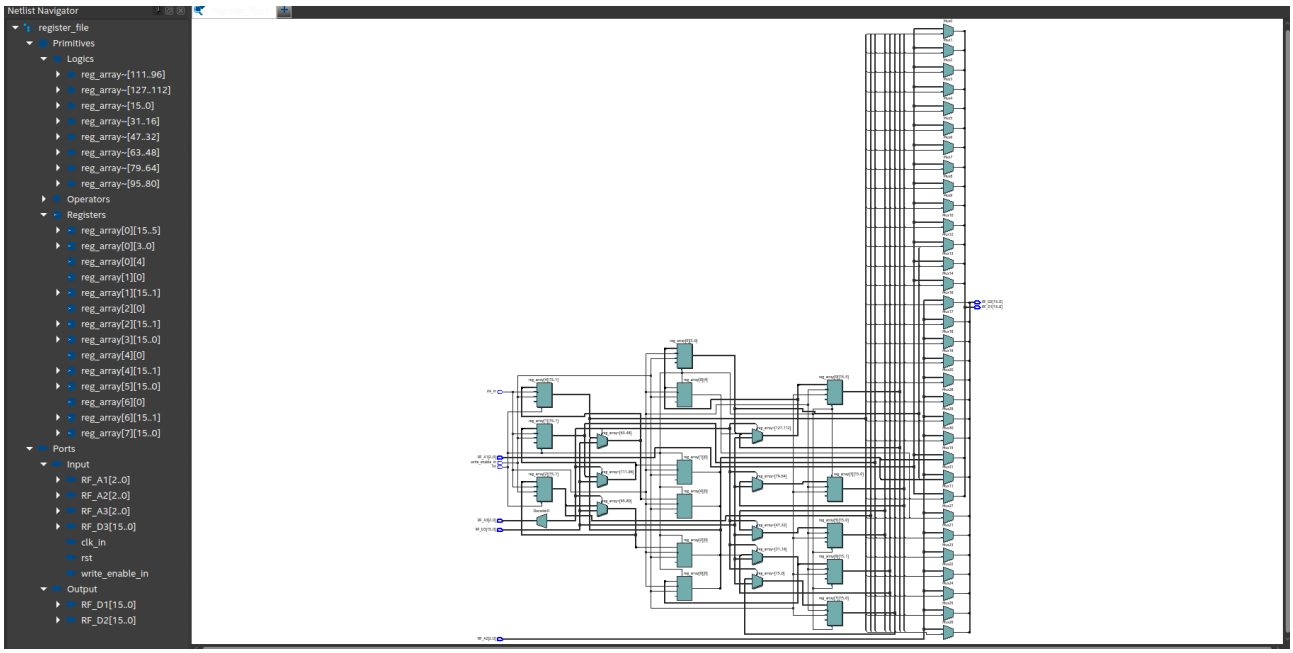
   b. Waveform



   c. Testbench for SixteenbitFullAdd
   TEST_ADD16.vhd

2. `Register File`

   a. RTL Viewer

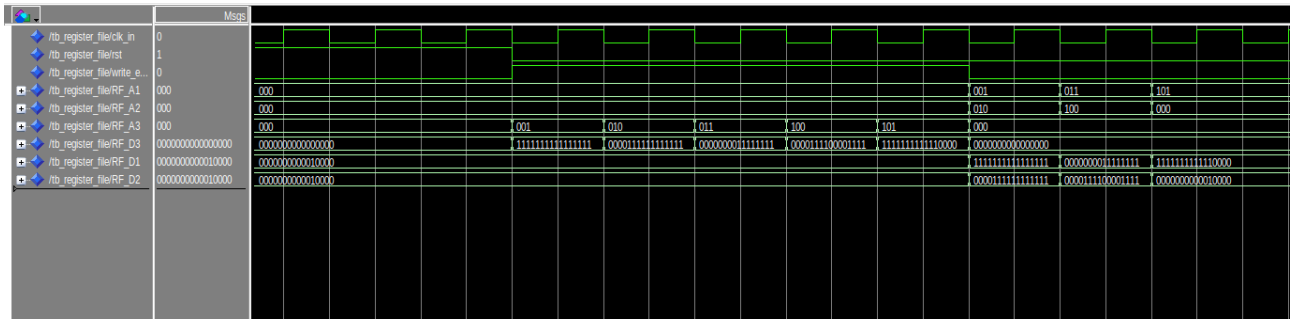


b. Waveform

First rst hold to 100 ns. When rst is high, we are storing some fixed random value in the register. Attached below.

```
if(rst = '1') then
  reg_array(0) <= x"0010";
  reg_array(1) <= x"0001";
  reg_array(2) <= x"0001";
  reg_array(3) <= x"ffff";
  reg_array(4) <= x"0001";
  reg_array(5) <= x"0000";
  reg_array(6) <= x"0001";
  reg_array(7) <= x"0000";
```

After that, we write five 16bit data into register(shown in waveform) then read that to show that writing writing take place at appropriate location.

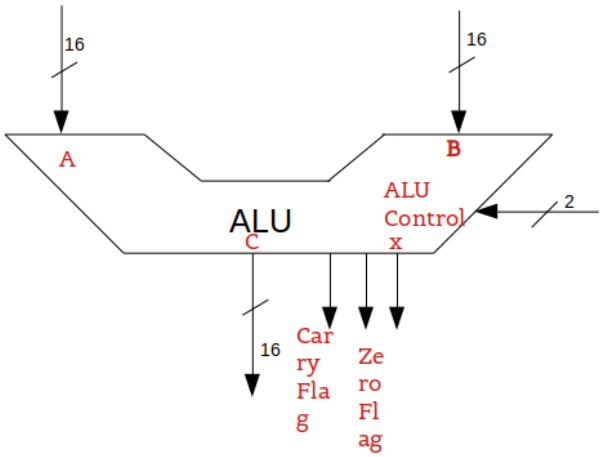


c. Testbench for SixteenbitFullAdd

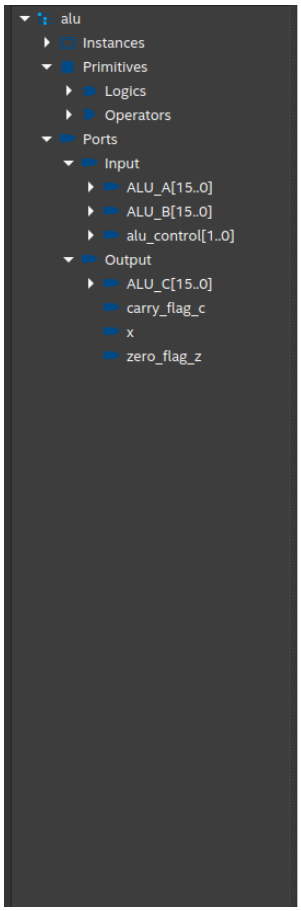
tb_register_file.vhd

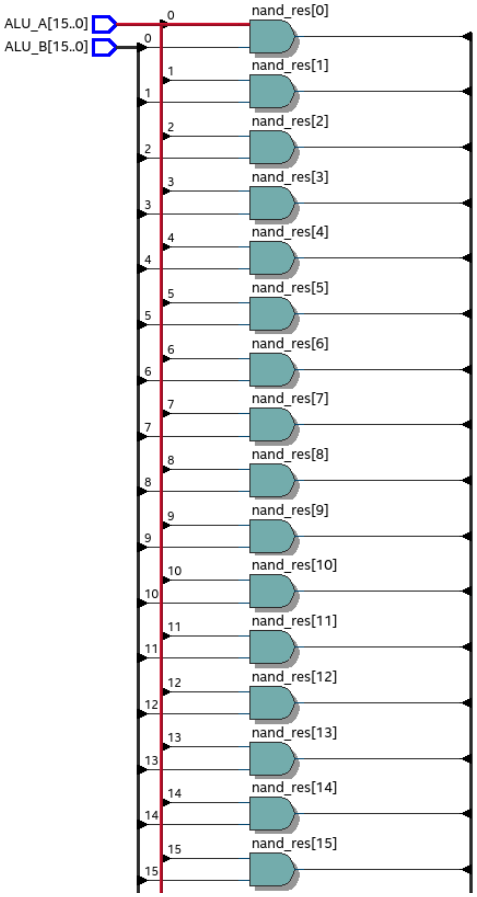
3. ALU

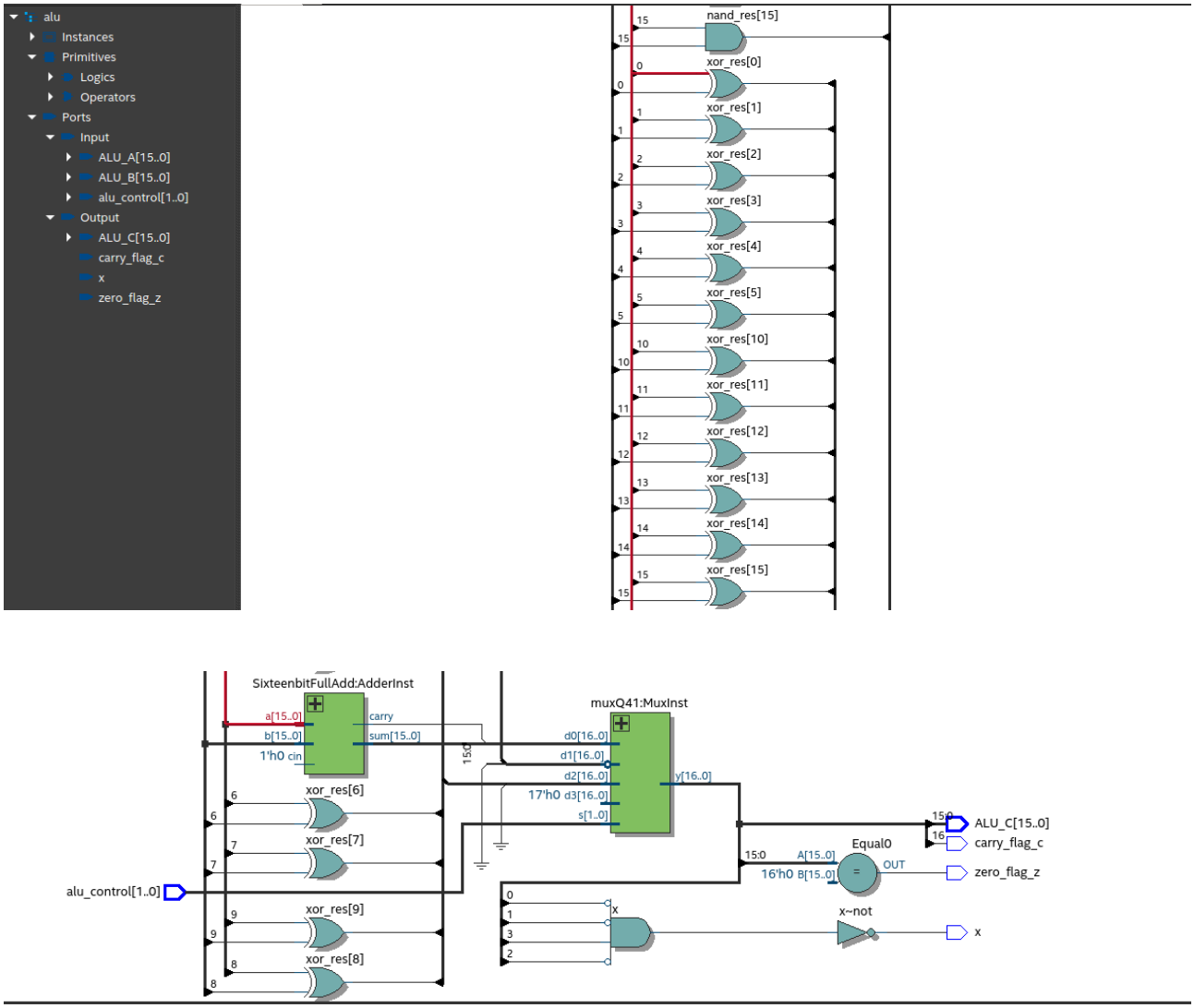
a. Diagram



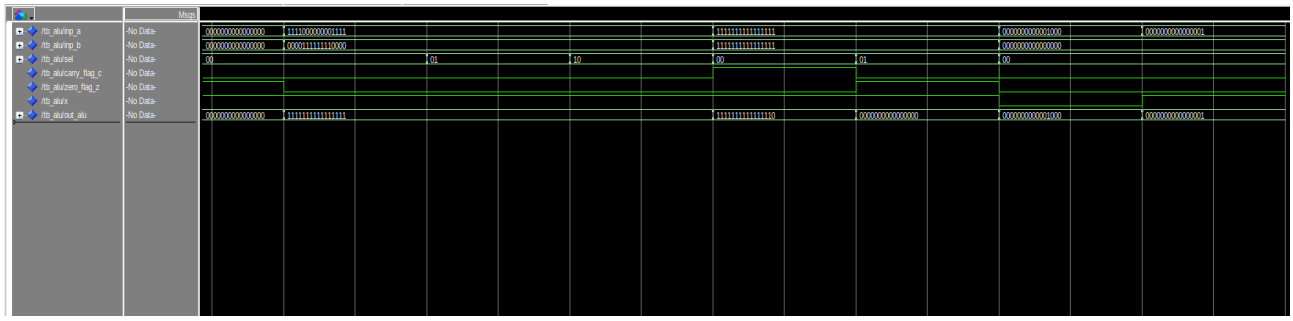
b. RTL Viewer







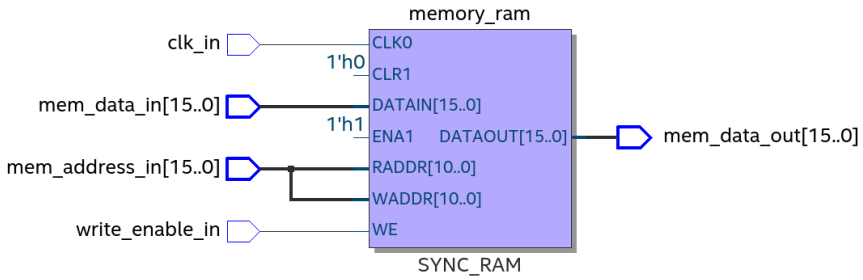
c. Waveform



d. Testbench for ALU
tb_alu.vhd

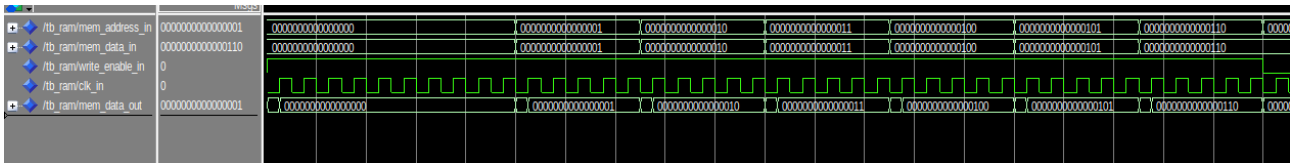
4. `Memory Async read sync write(single port` We faced difficulties for $2^{16} * 16$ because devices installed in quartus doesn't have dedicated ram. So, TA suggested me to reduce ram size because of few instructions. So I use 2047 downto 0 of 15 downto 0 in my project.

a. RTL Viewer

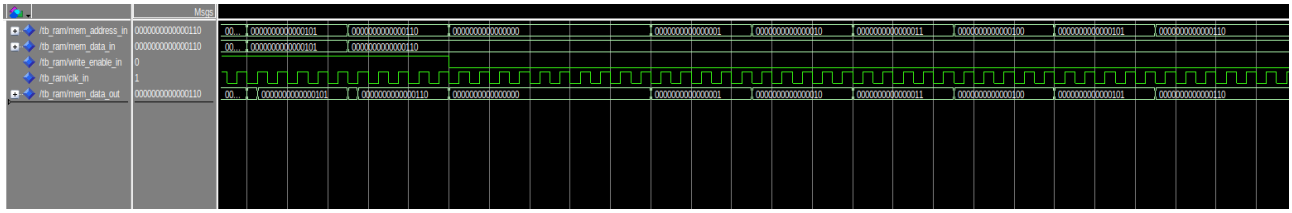


b. Waveform

– write



– read



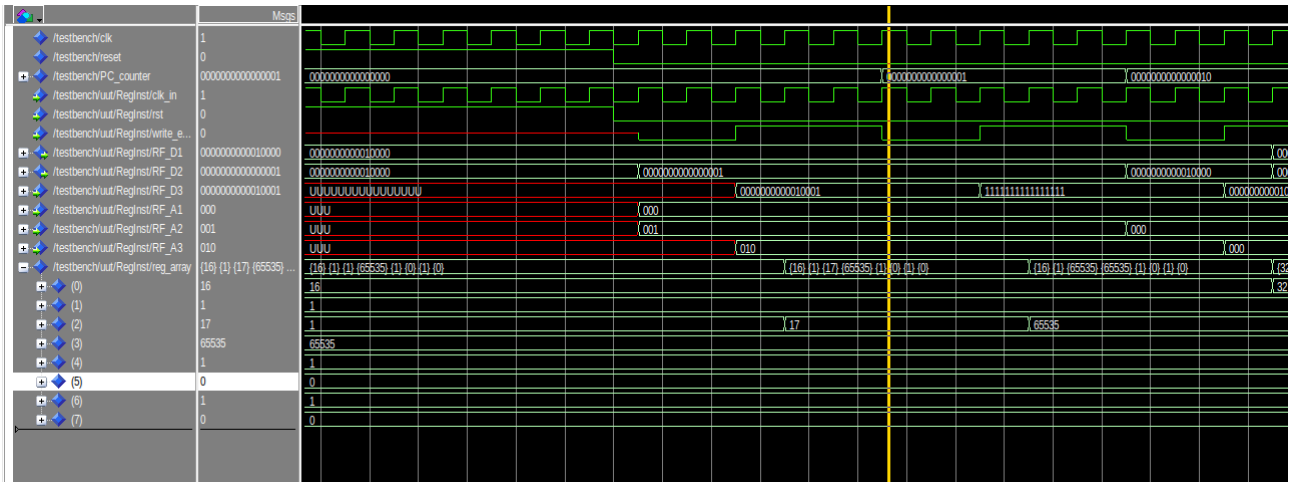
c. Testbench for memory_asyncread

tb_ram.vhd

TESTING OF INSTRUCTION

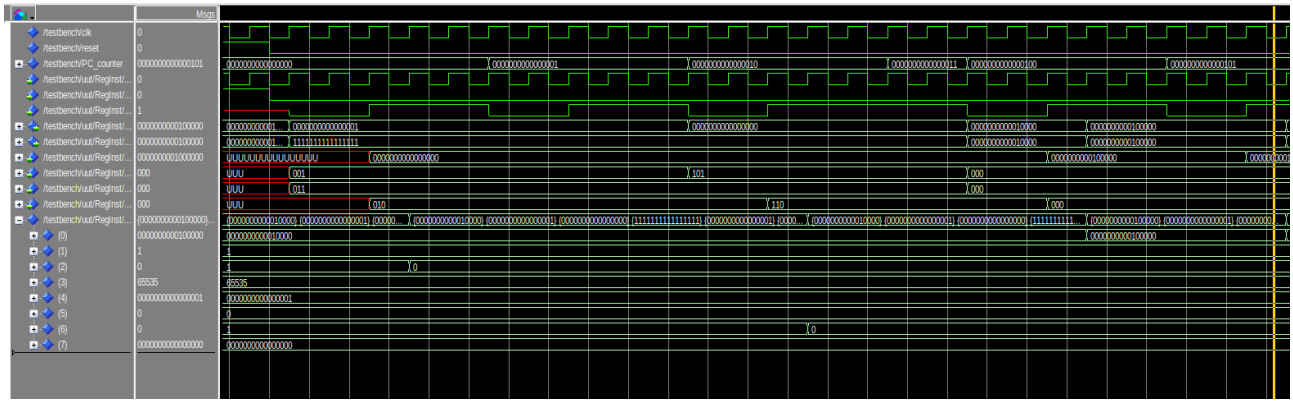
Decide what are all the sequences of instructions that you want to test and keep them as initial values in the RAM. Then supply clock and reset through the test bench and test it.

1. ADD & NDU
Let $RA \leftarrow$ "000", $RB \leftarrow$ "001" and $RC \leftarrow$ "010". So, we store 0 => 0000_000_001_010_0_00 & 1 => 0010_000_001_010_0_00.



Okay, as you can see initially 16 is stored in RA, 1 is stored in RB. After add operation you can see that 17 is stored in RC and after ndu operation 65535 is stored in RC.

2. ADD & ADC & ADZ & ADI
Let $RA \leftarrow$ "001", $RB \leftarrow$ "011" and $RC \leftarrow$ "010". So, we store 0 => 0000_001_011_010_0_00 & 1 => 0000_001_011_010_0_01.
for adi & adz, 2 => 0001_101_110_000000 (RA="101" and RB = "110"),
3 => 0000_001_011_010_0_10(RA, RB, RC defined above).



As you can see initially 1 is stored in RA, 65535 is stored in RB. After add operation you can see that 0 is stored in RC and after adc operation then still 0 is stored in RC because tc = 1 from previous add operation, After adi operation take RA index 5 and RB index 6 which make tz = 1 and then adz operation take place .

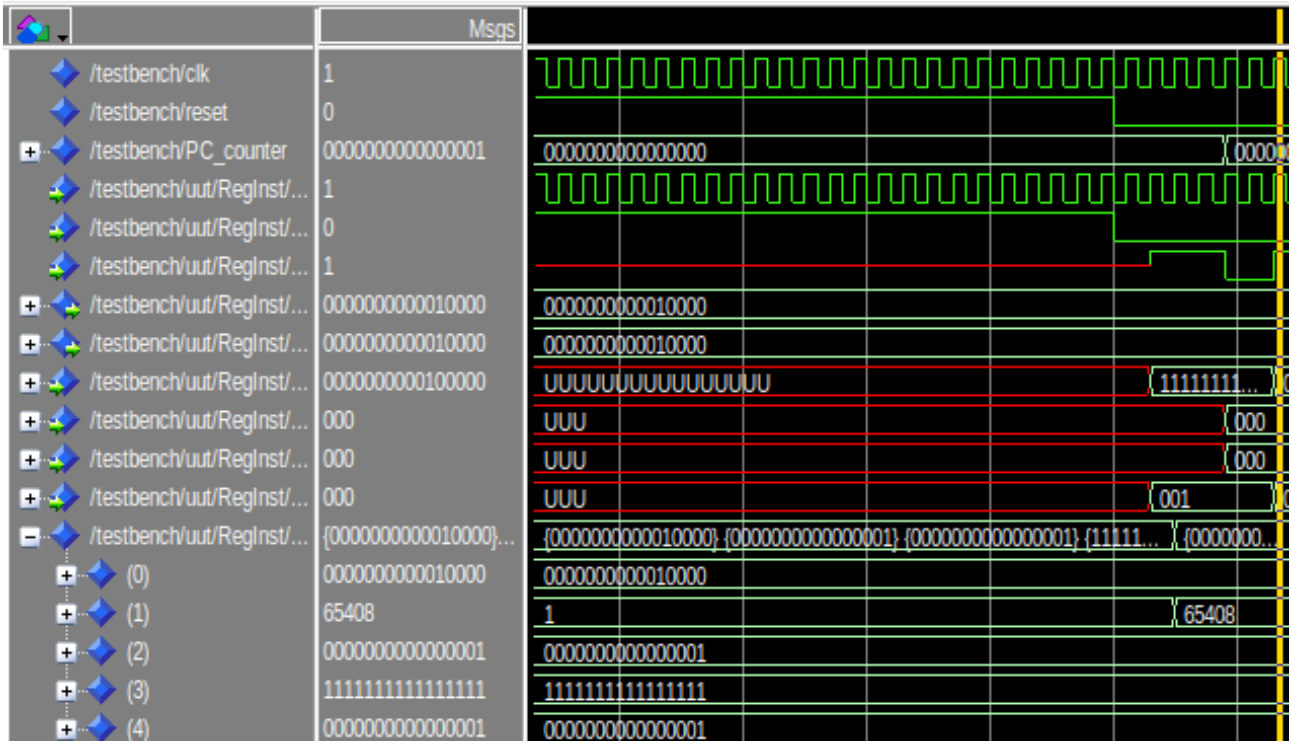
Similarly, ndc and ndz verified.

3. LHI & LW & SW

LHI

00_11 001 111111111

After reset, "001" register RA contains value = 1or "0000000000000001" (because we're initialising in reset operation for simplicity). After LHI operation, "001" register RA should contain 111111111_0000000 (in decimal 65,408). Let's see in the waveform.

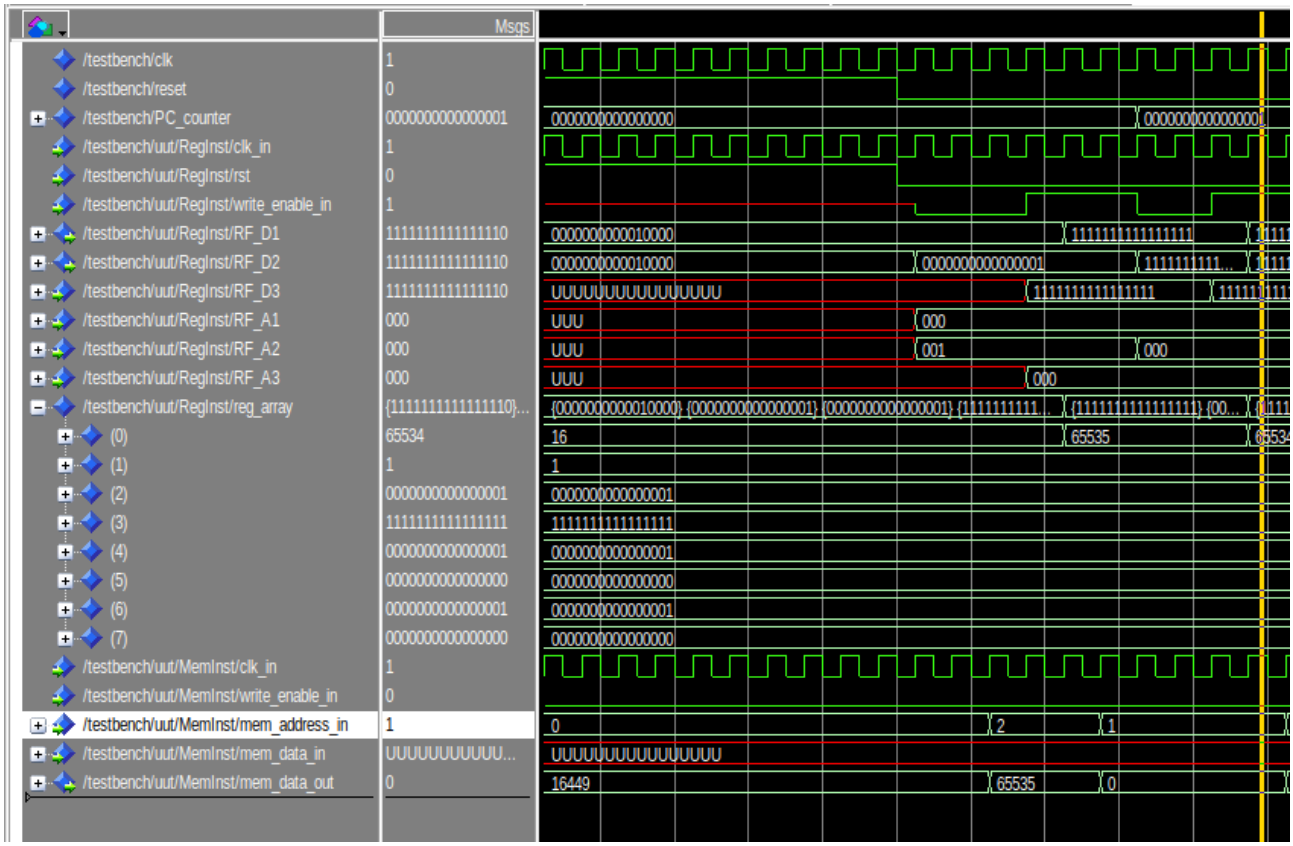


See in the waveform value changes from 1 to 65408. I modify radix to see in decimal.

LW

01_00 000 001 000001

RA = "000", RB = "001" , because of reset content of RB is 1 (in decimal unsigned). So, memory address from this come out to be 2 (in decimal unsigned). Let's store 1111_1111_1111_1111 at memory location 2. and see indeed our register RA contain 1111_1111_1111_1111 in waveform.

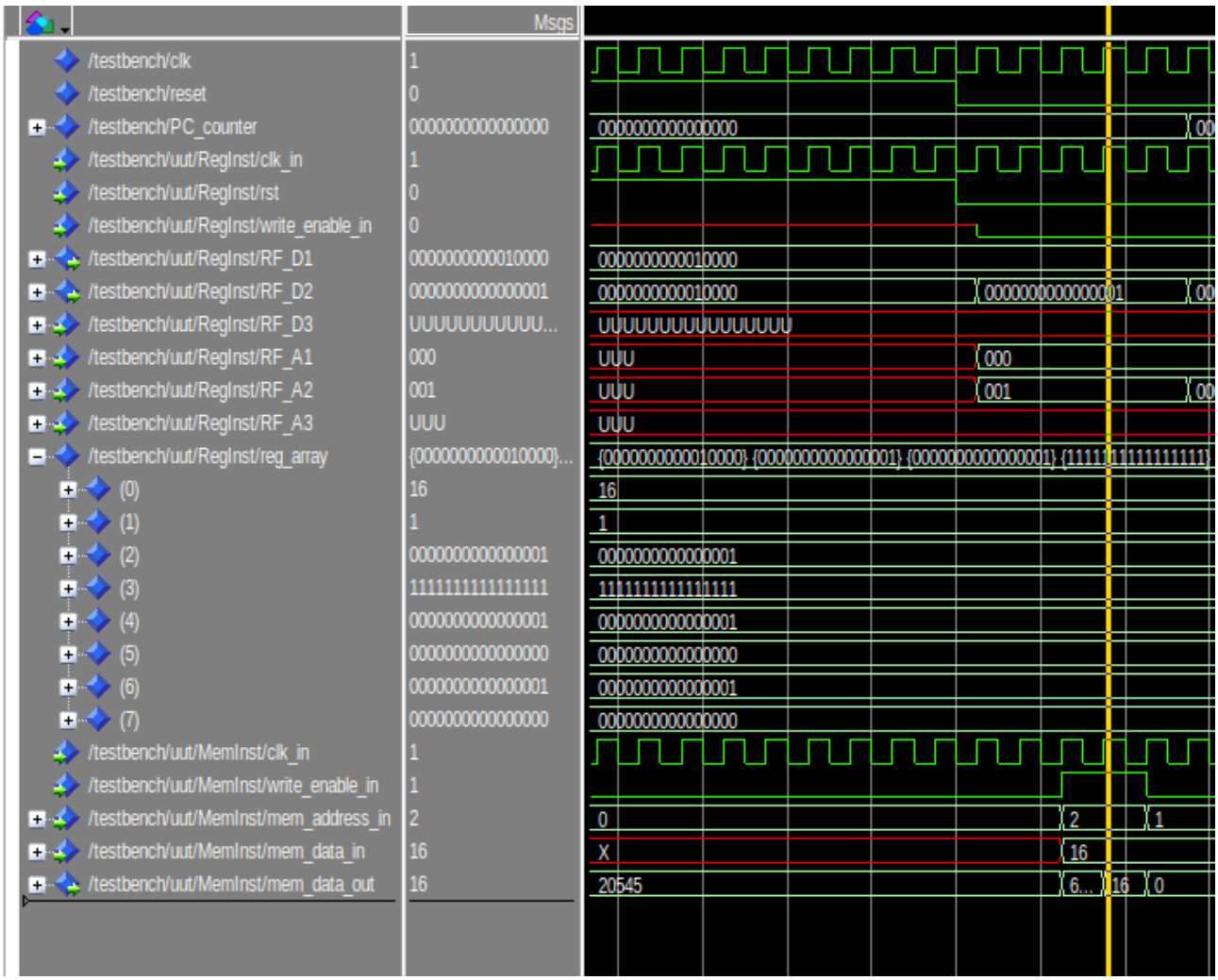


See in the waveform. I modify radix to see in decimal.

SW

01_01 000 001 000001

RA = "000", RB = "001" , because of reset content of RB is 1 and RA is 16(in decimal unsigned). So, memory address from this come out to be 2 (in decimal unsigned). Let's store see that 16 from register RA store in memory location 2 in the waveform.



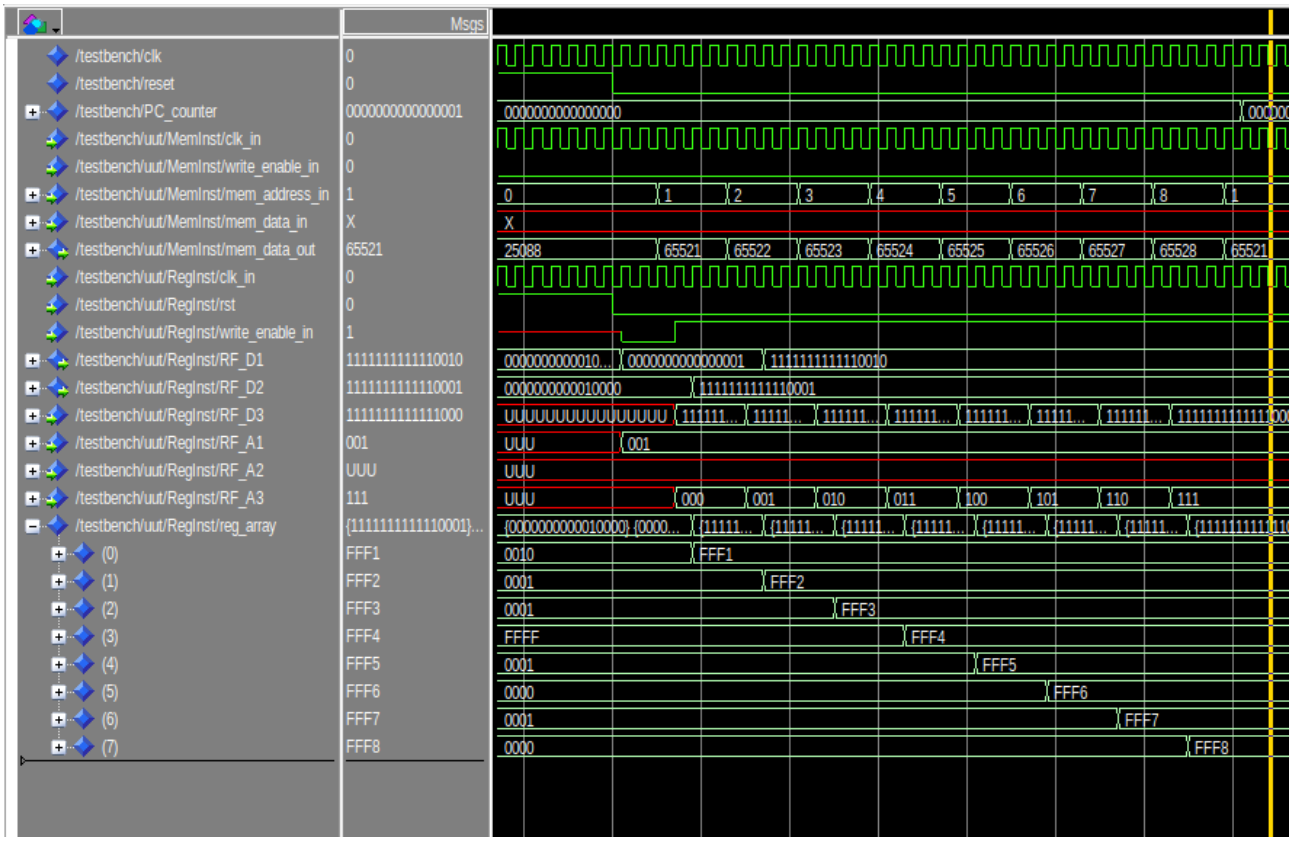
As you can see in last three line of waveform, 16 is stored in memory store at location 2. I modify radix to see in decimal.

4. LA & SA

LA

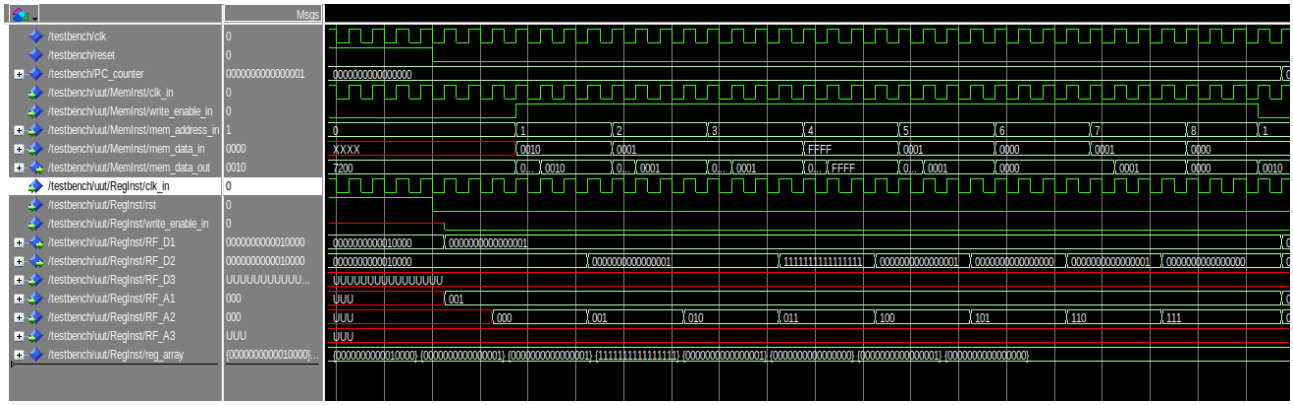
0110_001_000000000

RA = "001" (mem address store in RA). So, by reset 1 is store in RA. So, let's put 0 => 0110_001_000000000, 1 => x"fff1", 2 => x"fff2", 3 => x"fff3", 4 => x"fff4", 5 => x"fff5", 6 => x"fff6", 7 => x"fff7", 8 => x"fff8" in memory. Let's see our waveform what's stored in our register from R0 to R7.



Okay, so our waveform show correct result.

SA
0110_001_000000000
RA = ”001” (mem address store in RA). So, by reset 1 is store in RA. So, let’s put 0 => 0110_001_000000000 in memory. Due to reset, our register from R0 to R7 have R0 <= x”0010”, R1 <= x”0001”, R2 <= x”0001”, R3 <= x”ffff”, R4 <= x”0001”, R5 <= x”0000”, R6 <= x”0001”, R7 <= x”0000”. Let’s see waveform where and what data going to store in memory

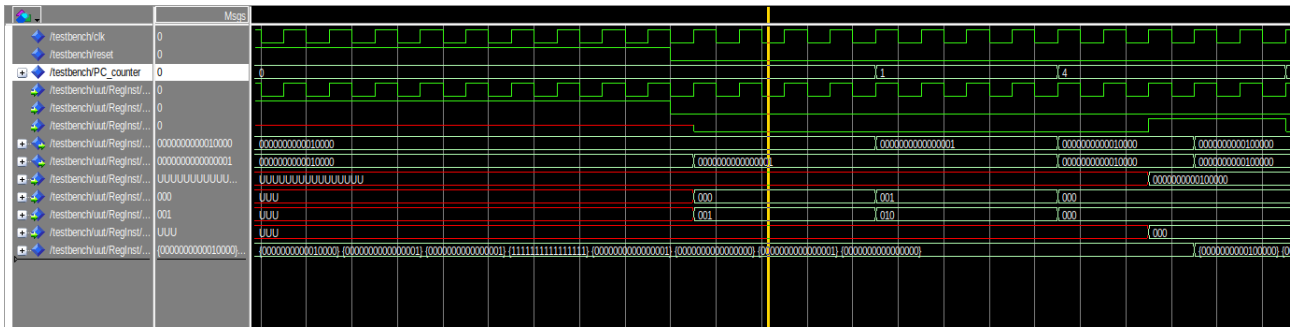


Our waveform show what value is going to store in memory and which position. This clearly matches with our assumption given above.

5. BEQ & JAL & JLR

BEQ
1100 000 001 000011

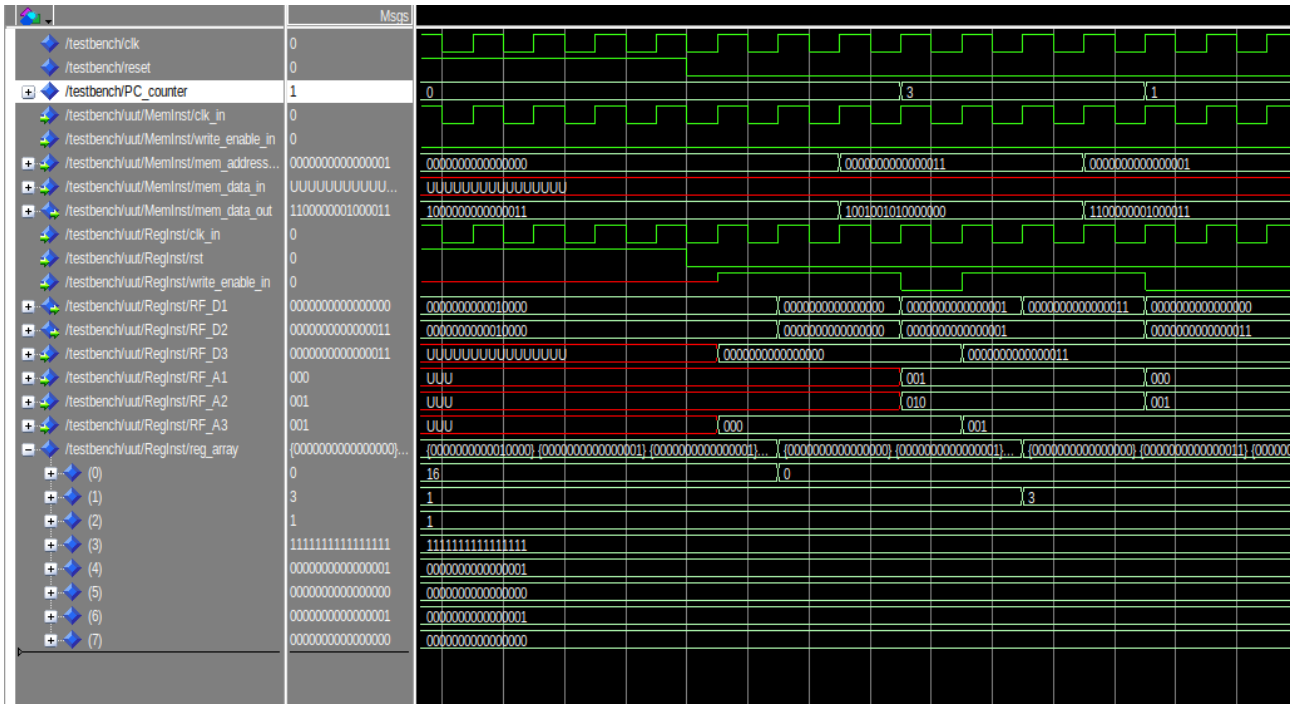
Due to reset, RA = 0 (contains 8) and RB = 1 (contains 1). So, there are not equal thats why IP move to next position.
1100 001 010 000011
Due to reset, RA = 1 (contains 1) and RB = 2 (contains 1). So, there are equal thats why IP move to IP + Imm. Let's see the waveform.



As you can see in the waveform that PC_counter move to next after first instruction then move to IP_Imm after second operation.

JAL & JLR

1000 000 000000011
RA = 0 (contains 8 initially) and IP = IP + Imm after instruction completion.
And let's store JLR instruction in IP+Imm location
1001 001 010 000000
RA = 1 (contains 1 initially) and RB = 2 (contains 1 initially) Lets see waveform.



As you can see in the waveform it clearly satisfies what we assume to be.

Testbench that I use is given here and all instruction is initialised in memory before use.

END
