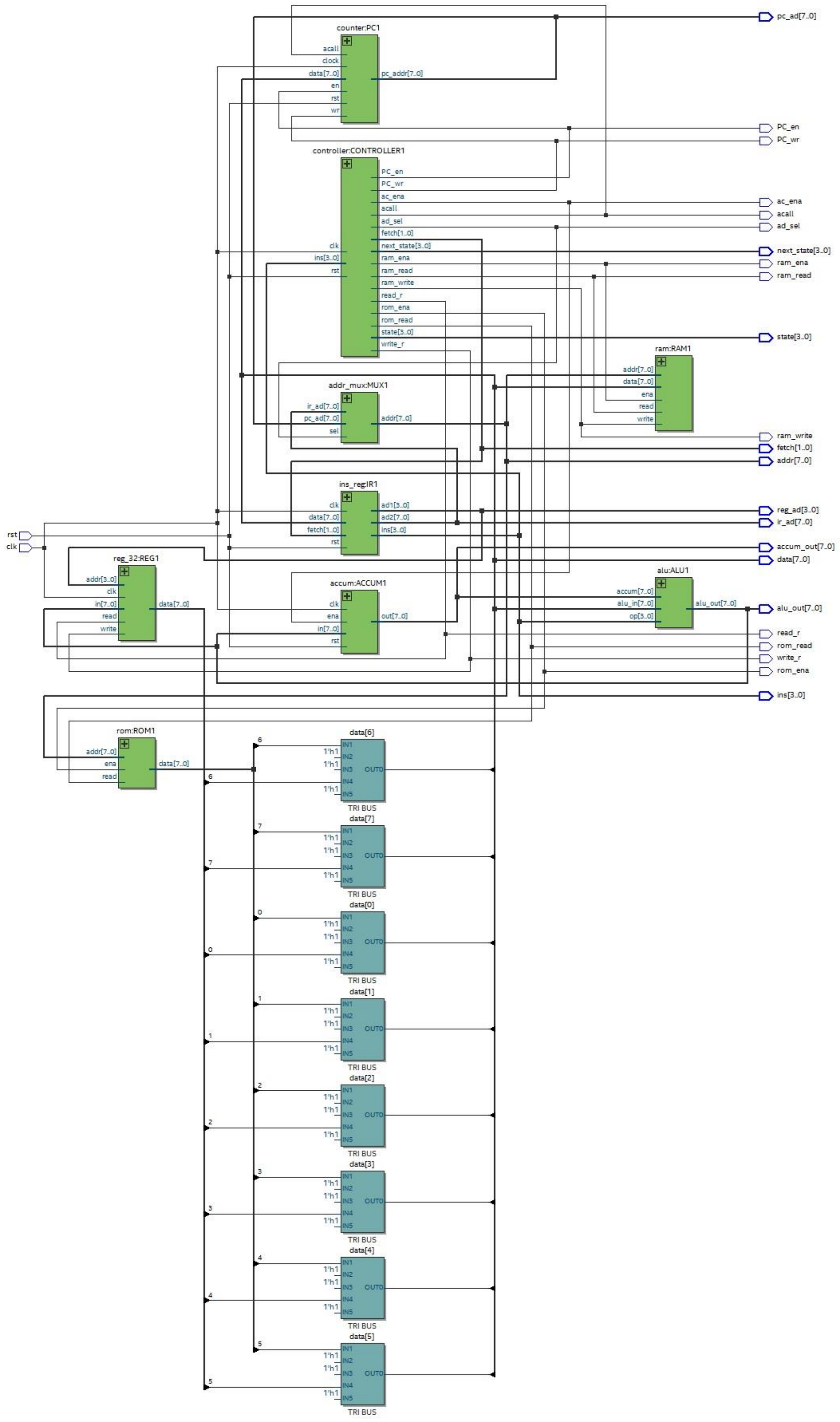
Jovanko Kenshian
00000032025

Specification

8-bit RISC CPU with 4-bit functioning as Operational Code/Instruction code, and 4 bit as Register Address.

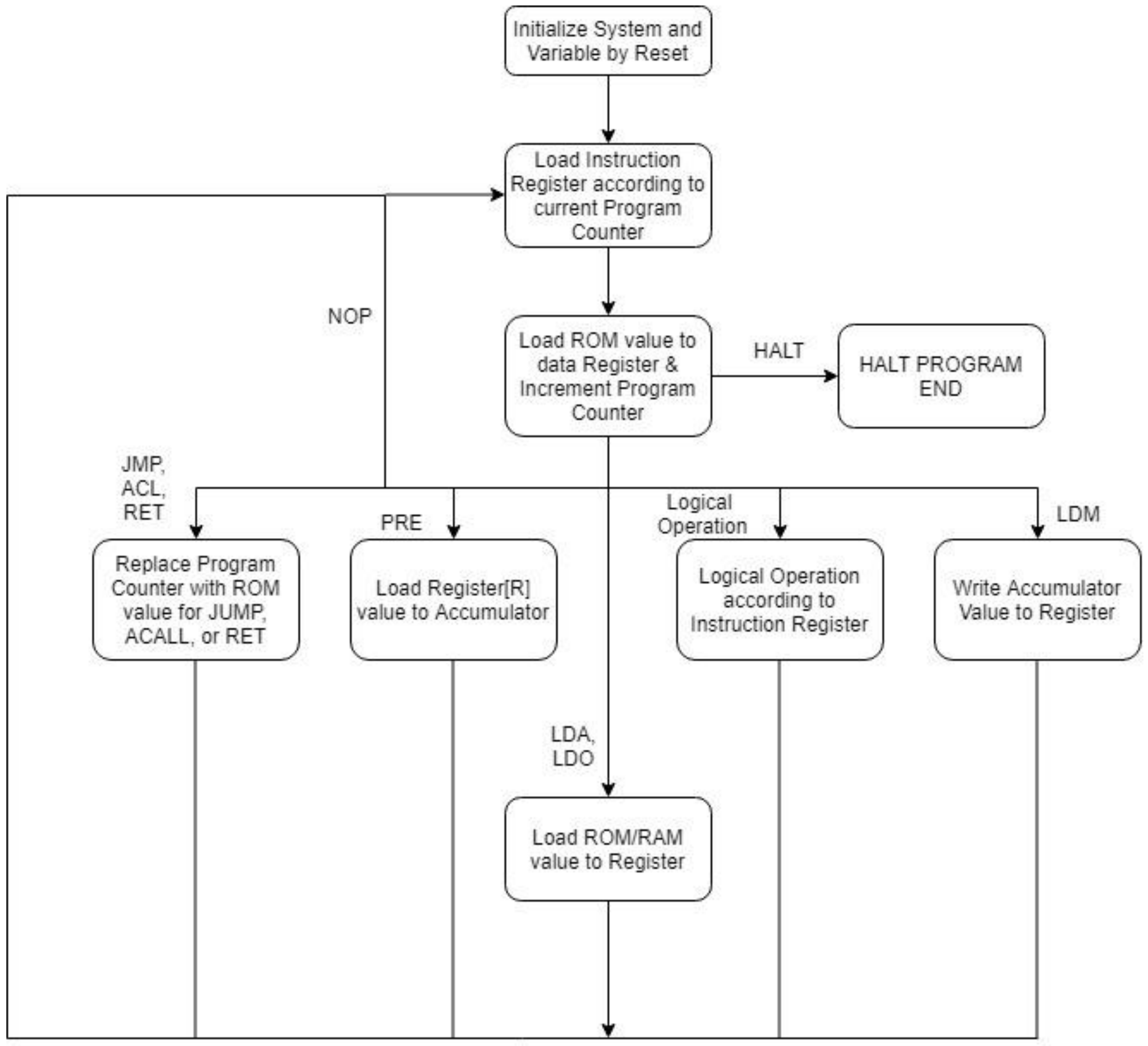
Jovanko Kenshian
00000032025

RTL of RISC Architecture



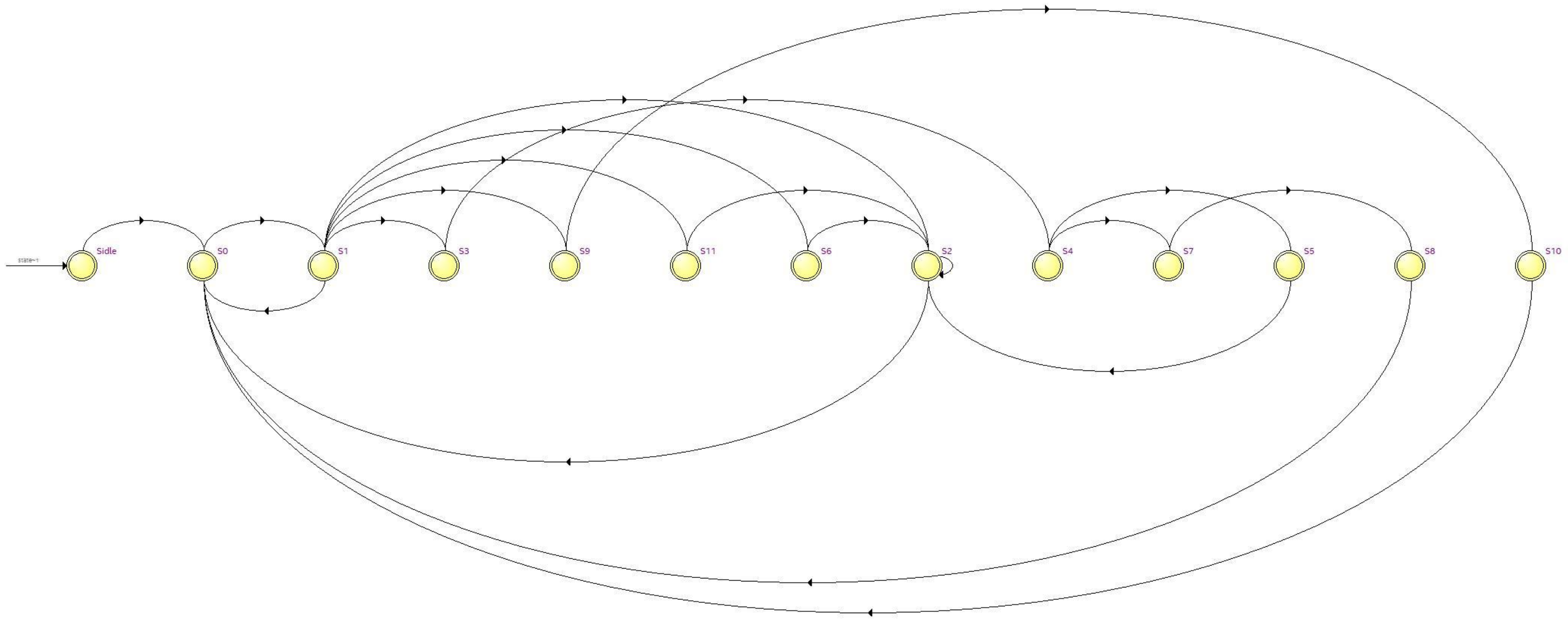
Jovanko Kenshian
00000032025

System Flowchart


Initialize System and Variable by Reset
Load Instruction Register according to current Program Counter
NOP
Load ROM value to data Register & Increment Program Counter
HALT
HALT PROGRAM END
JMP, ACL, RET
PRE
Logical Operation
LDM
Replace Program Counter with ROM value for JUMP, ACALL, or RET
Load Register[R] value to Accumulator
Logical Operation according to Instruction Register
Write Accumulator Value to Register
LDA, LDO
Load ROM/RAM value to Register

Jovanko Kenshian
00000032025

State Diagram



Jovanko Kenshian
00000032025

State Table Transition Table

No	Source State	Destination State	Condition
1	S0	S1	
2	S1	S0	ins==NOP
3		S3	ins==LDA || ins==LDO || ins==STO
4		S6	ins==JMP || ins==ACL || ins==RET
5		S9	ins==PRE | ins==ADD | ins==SUB | ins==LAND | ins==LOR | ins==LNOT | ins==INC
6		S11	ins==LDM
7		S2	ins==HLT
8	S2	S0	
9		S2	ins==HLT
10	S3	S4	
11	S4	S5	ins==LDA || ins==LDO
12		S7	
13	S5	S2	
14	S6	S2	
15	S7	S8	
16	S8	S0	
17	S9	S10	
18	S10	S0	
19	S11	S2	
20	Sidle	S0	

Jovanko Kenshian
00000032025

State Explanation and Description

No	States	Explanation of States
1	Sidle	Controller is idle until machine is reset
2	S0	Fetch ROM according to PC to know instruction and register value
3	S1	Increment PC Counter and read ROM value
4	S2	No process is done to stabilize data & to stop machine from operating (loop)
5	S3	Fetch address value from ROM which will be used to load ROM or RAM in that address value to register. This is also used to Store Results from Register to RAM.
6	S4	Increment PC Counter and read ROM value
7	S5	Write Register with data fetch from RAM or ROM
8	S6	Jump to specific address by replacing PC with data value. PC will also be stored in stack when ACALL ins is run. RET will fetch the stored PC value to return before to PC address before ACALL
9	S7	Read Register value according to address value (data low byte)
10	S8	Write RAM
11	S9	Fetch Register to ALU or process logical operation in ALU
12	S10	Read Register value
13	S11	Write Register with value in ALU

Jovanko Kenshian
00000032025

OpCode List

OpCode			
Hex	Binary	Mnemonic	Description
0	0000	NOP (1 byte)	No Operation is Done
1	0001	LDO (2 byte)	Load ROM value to Register[R]
2	0010	LDA (2 byte)	Load RAM value to Register[R]
3	0011	STO (2 byte)	Store Intermediate Results to RAM
4	0100	PRE (1 byte)	Prefetch Data from Register[R]
5	0101	JMP (2 byte)	Jump to specific ROM Address
6	0110	ADD (1 byte)	Add Acccumulator with value from Register[R]
7	0111	SUB (1 byte)	Substract Acccumulator with value from Register[R]
8	1000	LAND (1 byte)	Logical AND Acccumulator with value from Register[R]
9	1001	LOR (1 byte)	Logical OR Acccumulator with value from Register[R]
A	1010	LNOT (1 byte)	Logical NOT Acccumulator
B	1011	INC (1 byte)	Increment Accumulator
C	1100	ACL (2 byte)	Jump to specific ROM Address & save PC Address to Stack
D	1101	RET (1 byte)	Return to previous Address (ACALL)
E	1110	LDM (1 byte)	Store Intermediate Results to Register[R]
F	1111	HLT (1 byte)	Stop Process

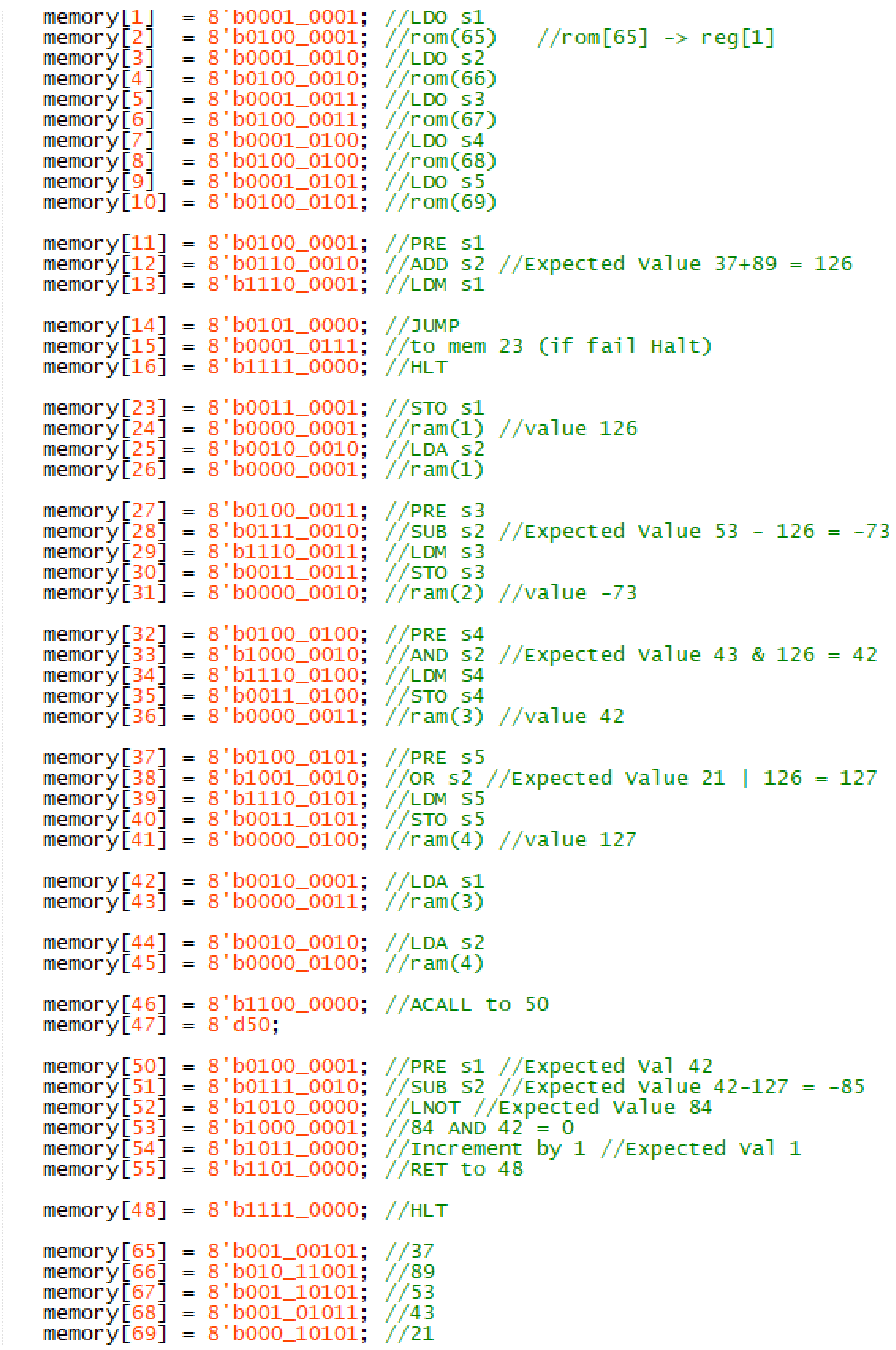
Instruction Bytes

1 Byte => [0000][0000]	[OpCode Bit] [Reg Address]
2 Byte => [0000][0000][0000_0000]	[OpCode Bit] [Reg Address] [Data]

Jovanko Kenshian
00000032025

Simulation

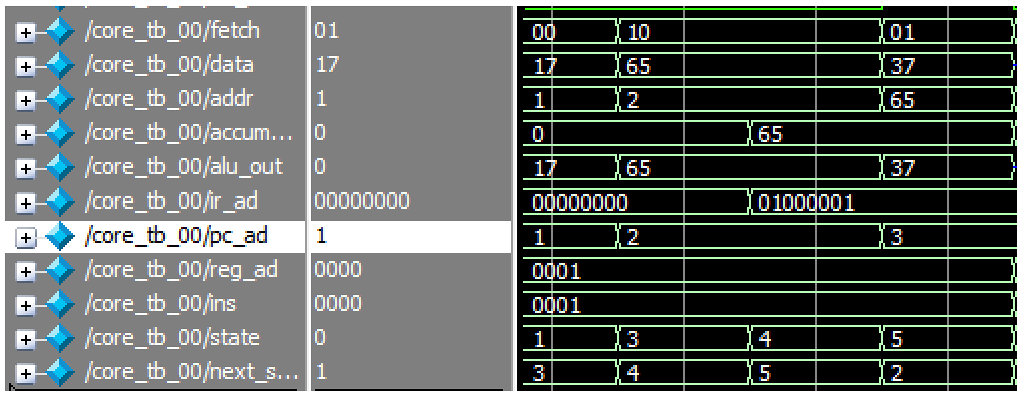
The design will be simulated by using ROM address value as the input



```
memory[1]  = 8'b0001_0001; //LDO s1
memory[2]  = 8'b0100_0001; //rom(65)   //rom[65] -> reg[1]
memory[3]  = 8'b0001_0010; //LDO s2
memory[4]  = 8'b0100_0010; //rom(66)
memory[5]  = 8'b0001_0011; //LDO s3
memory[6]  = 8'b0100_0011; //rom(67)
memory[7]  = 8'b0001_0100; //LDO s4
memory[8]  = 8'b0100_0100; //rom(68)
memory[9]  = 8'b0001_0101; //LDO s5
memory[10] = 8'b0100_0101; //rom(69)

memory[11] = 8'b0100_0001; //PRE s1
memory[12] = 8'b0110_0010; //ADD s2 //Expected Value 37+89 = 126
memory[13] = 8'b1110_0001; //LDM s1

memory[14] = 8'b0101_0000; //JUMP
memory[15] = 8'b0001_0111; //to mem 23 (if fail Halt)
memory[16] = 8'b1111_0000; //HLT

memory[23] = 8'b0011_0001; //STO s1
memory[24] = 8'b0000_0001; //ram(1) //value 126
memory[25] = 8'b0010_0010; //LDA s2
memory[26] = 8'b0000_0001; //ram(1)

memory[27] = 8'b0100_0011; //PRE s3
memory[28] = 8'b0111_0010; //SUB s2 //Expected Value 53 - 126 = -73
memory[29] = 8'b1110_0011; //LDM s3
memory[30] = 8'b0011_0011; //STO s3
memory[31] = 8'b0000_0010; //ram(2) //value -73

memory[32] = 8'b0100_0100; //PRE s4
memory[33] = 8'b1000_0010; //AND s2 //Expected Value 43 & 126 = 42
memory[34] = 8'b1110_0100; //LDM S4
memory[35] = 8'b0011_0100; //STO s4
memory[36] = 8'b0000_0011; //ram(3) //value 42

memory[37] = 8'b0100_0101; //PRE s5
memory[38] = 8'b1001_0010; //OR s2 //Expected Value 21 | 126 = 127
memory[39] = 8'b1110_0101; //LDM S5
memory[40] = 8'b0011_0101; //STO s5
memory[41] = 8'b0000_0100; //ram(4) //value 127

memory[42] = 8'b0010_0001; //LDA s1
memory[43] = 8'b0000_0011; //ram(3)

memory[44] = 8'b0010_0010; //LDA s2
memory[45] = 8'b0000_0100; //ram(4)

memory[46] = 8'b1100_0000; //ACALL to 50
memory[47] = 8'd50;

memory[50] = 8'b0100_0001; //PRE s1 //Expected Val 42
memory[51] = 8'b0111_0010; //SUB S2 //Expected Value 42-127 = -85
memory[52] = 8'b1010_0000; //LNOT //Expected Value 84
memory[53] = 8'b1000_0001; //84 AND 42 = 0
memory[54] = 8'b1011_0000; //Increment by 1 //Expected Val 1
memory[55] = 8'b1101_0000; //RET to 48

memory[48] = 8'b1111_0000; //HLT

memory[65] = 8'b001_00101; //37
memory[66] = 8'b010_11001; //89
memory[67] = 8'b001_10101; //53
memory[68] = 8'b001_01011; //43
memory[69] = 8'b000_10101; //21
```

Jovanko Kenshian
00000032025

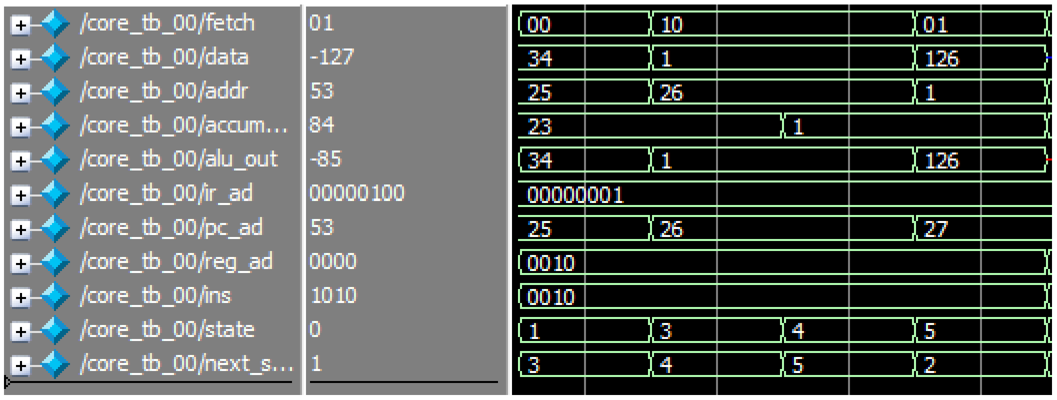
Testing ROM Fetching (LDO)



```
/core_tb_00/fetch      01         00 | 10 | 01
/core_tb_00/data       17         17 | 65 | 37
/core_tb_00/addr       1          1 | 2 | 65
/core_tb_00/accum...   0          0 | 65
/core_tb_00/alu_out    0          17 | 65 | 37
/core_tb_00/ir_ad      00000000   00000000 | 01000001
/core_tb_00/pc_ad      1          1 | 2 | 3
/core_tb_00/reg_ad     0000       0001
/core_tb_00/ins        0000       0001
/core_tb_00/state      0          1 | 3 | 4 | 5
/core_tb_00/next_s...  1          3 | 4 | 5 | 2
```

Instruction LDO was tested by fetching ROM value according to the 2nd byte address value and storing it at a register. The simulation shown in the picture above is taken using the input from ROM 1st and 2nd Address. From the data variable that is shown in the picture, it could be seen that the fetch process occurs in three different process. First the CPU reads the instruction address and register address from the ROM 1st Address to know what instruction needs to be run and where to store the fetched data. The CPU understands that it needs to fetch value indirectly from ROM and reads the next ROM address, which is ROM 2nd Address. The data value changes to 65, which is the address value where the CPU needs to fetch data from. The CPU then fetches the data from ROM 65th Address, with the value of 37, and stores it on the Register 1st address.

Testing RAM Fetching (LDA)

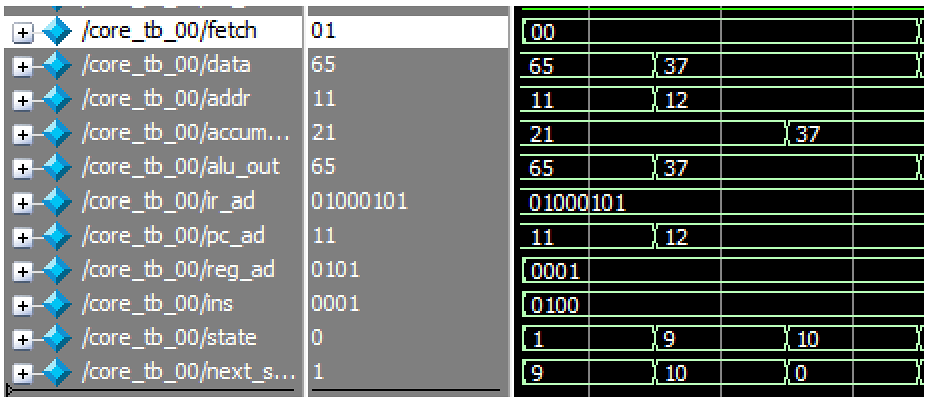


```
/core_tb_00/fetch      01         00 | 10 | 01
/core_tb_00/data       -127       34 | 1 | 126
/core_tb_00/addr       53         25 | 26 | 1
/core_tb_00/accum...   84         23 | 1
/core_tb_00/alu_out    -85        34 | 1 | 126
/core_tb_00/ir_ad      00000100   00000001
/core_tb_00/pc_ad      53         25 | 26 | 27
/core_tb_00/reg_ad     0000       0010
/core_tb_00/ins        1010       0010
/core_tb_00/state      0          1 | 3 | 4 | 5
/core_tb_00/next_s...  1          3 | 4 | 5 | 2
```

Instruction LDA was tested by fetching RAM value according to the 2nd byte address value and storing it at a register. The simulation shown in the picture above is taken using the input from ROM 25th and 26th Address from the data variable that is shown in the picture, it could be seen that the fetch process occurs in three different process. First the CPU reads the instruction address and register address from the ROM 25th Address to know what instruction needs to be run and where to store the fetched data. The CPU understands that it needs to fetch value indirectly from RAM and reads the next ROM address, which is ROM 26th Address. The data value changes to 1, which is the address value where the CPU needs to fetch data from. The CPU then fetches the data from RAM 1st Address, with the value of 126, and stores it on the Register 2nd address.

Jovanko Kenshian
00000032025

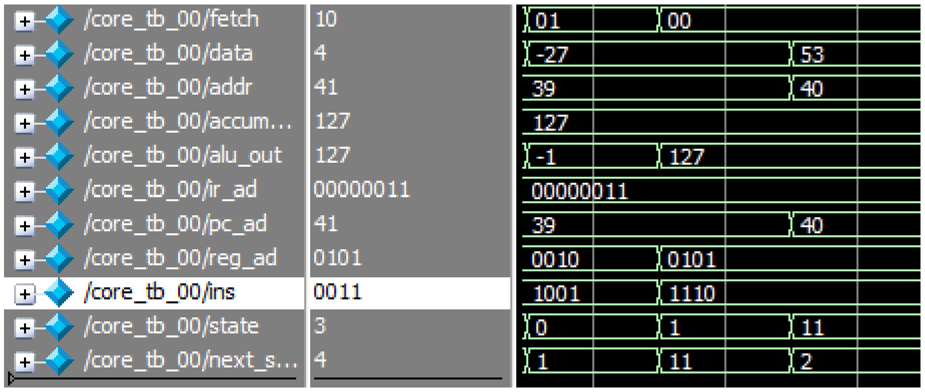
Testing Register Fetching (PRE)



```
/core_tb_00/fetch      01         00
/core_tb_00/data       65         65        37
/core_tb_00/addr       11         11        12
/core_tb_00/accum...   21         21                  37
/core_tb_00/alu_out    65         65        37
/core_tb_00/ir_ad      01000101   01000101
/core_tb_00/pc_ad      11         11        12
/core_tb_00/reg_ad     0101       0001
/core_tb_00/ins        0001       0100
/core_tb_00/state      0          1         9         10
/core_tb_00/next_s...  1          9         10        0
```

Instruction PRE was tested by fetching Register value according to the lower byte in the ROM value. The process works by reading the instruction, in the data upper bit, and the register address, in the data lower bit. The data was fetched from the Register according to the register address. The picture above is run using the ROM 11th address value. From the picture, it could be seen that the data value is 65 or 0100_0001. The CPU then fetches the 1st Register Address and moves it to ALU. The fetch works since the value of data and alu_out changes to 37.

Testing Storing Value to Register from Accumulator (LDM)

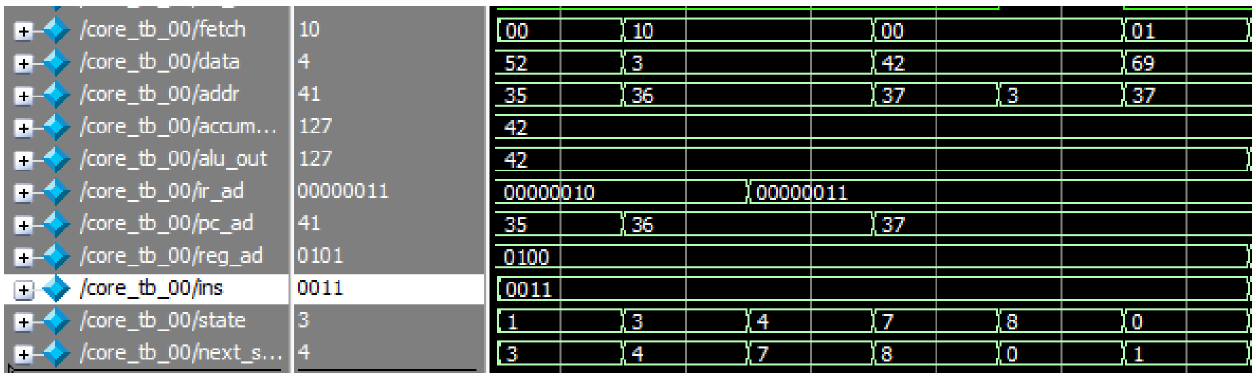


```
/core_tb_00/fetch      10         01        00
/core_tb_00/data       4          -27                 53
/core_tb_00/addr       41         39                  40
/core_tb_00/accum...   127        127
/core_tb_00/alu_out    127        -1        127
/core_tb_00/ir_ad      00000011   00000011
/core_tb_00/pc_ad      41         39                  40
/core_tb_00/reg_ad     0101       0010      0101
/core_tb_00/ins        0011       1001      1110
/core_tb_00/state      3          0         1         11
/core_tb_00/next_s...  4          1         11        2
```

Instruction LDM was tested by storing ALU value to Register according to the lower byte in the ROM value. The process works by reading the instruction, in the data upper bit, and the register address, in the data lower bit. The data was fetched from the Register according to the register address. The picture above is run using the ROM 39th address value. From the picture, it could be seen that the data value is -27 or 1110_0101. The CPU then stores the ALU value to the 5th register.

Jovanko Kenshian
00000032025

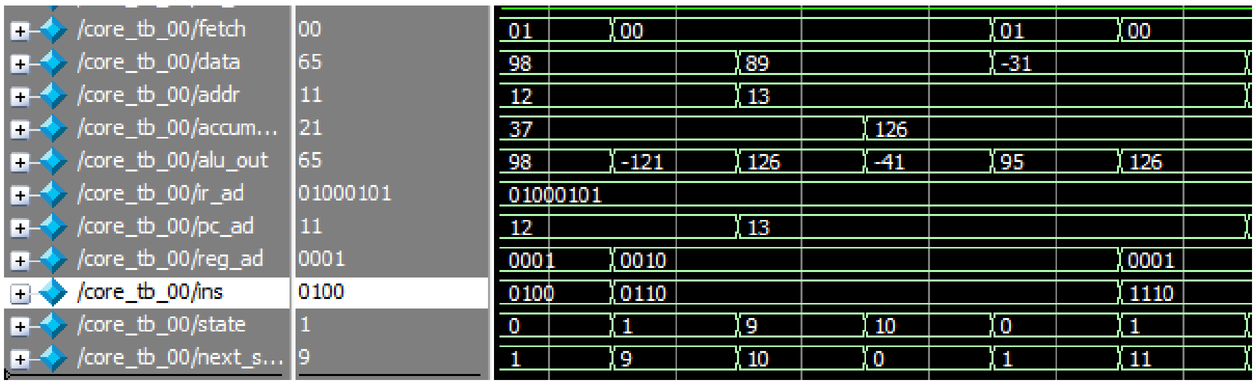
Testing Storing Value to RAM from Accumulator (STO)



/core_tb_00/fetch	10
/core_tb_00/data	4
/core_tb_00/addr	41
/core_tb_00/accum...	127
/core_tb_00/alu_out	127
/core_tb_00/ir_ad	00000011
/core_tb_00/pc_ad	41
/core_tb_00/reg_ad	0101
/core_tb_00/ins	0011
/core_tb_00/state	3
/core_tb_00/next_s...	4

Instruction STO was tested by storing ALU value to RAM address according to the 2nd byte address value. The simulation shown in the picture above is taken using the input from ROM 35th and 36th Address from the data variable that is shown in the picture, it could be seen that the fetch process occurs in three different process. First the CPU reads the instruction address and register address from the ROM 35th Address to know what instruction needs to be run and where to store the fetched data. The CPU understands that it needs to store value to RAM and reads the next ROM address, which is ROM 36th Address. The data value changes to 3, which is the RAM address value where the CPU needs to store data to. The CPU then fetches the data from ALU, with the value of 42, and stores it on the Register 3th address.

Testing Logical Operation (ADD)

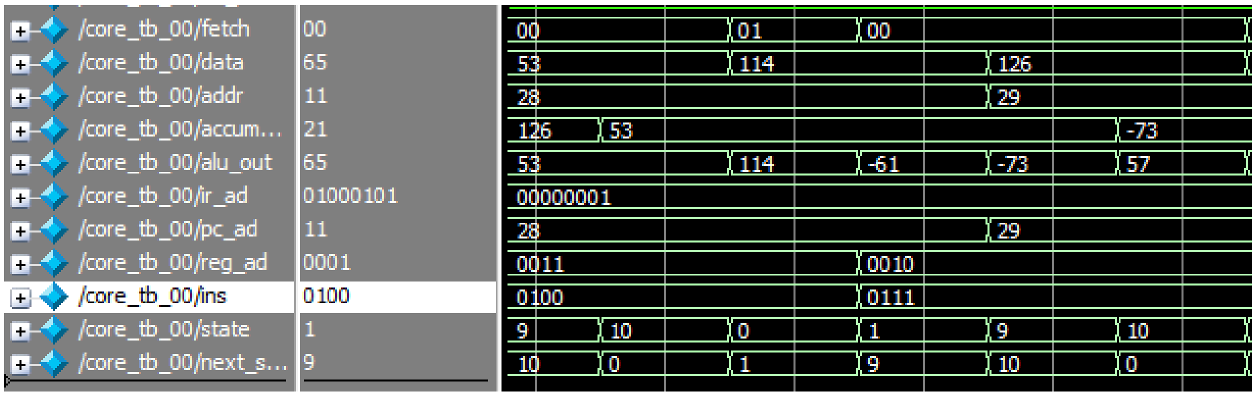


/core_tb_00/fetch	00
/core_tb_00/data	65
/core_tb_00/addr	11
/core_tb_00/accum...	21
/core_tb_00/alu_out	65
/core_tb_00/ir_ad	01000101
/core_tb_00/pc_ad	11
/core_tb_00/reg_ad	0001
/core_tb_00/ins	0100
/core_tb_00/state	1
/core_tb_00/next_s...	9

Instruction ADD is tested by adding the preloaded Register 1st Address with Register 2nd Address. For this simulation, the instruction address used are taken from ROM 12th Address. The process works in two steps. First, the CPU fetches instruction from ROM. The instruction address is 0110_0010, which means adding ALU value with Register 2nd Address value. The CPU then fetches Register 2nd Address value, which is 89. The ALU then adds the ALU with the loaded value. The Result could be seen in the alu_out which is 126.The value in the alu_out is sent to the accumulator.

Jovanko Kenshian
00000032025

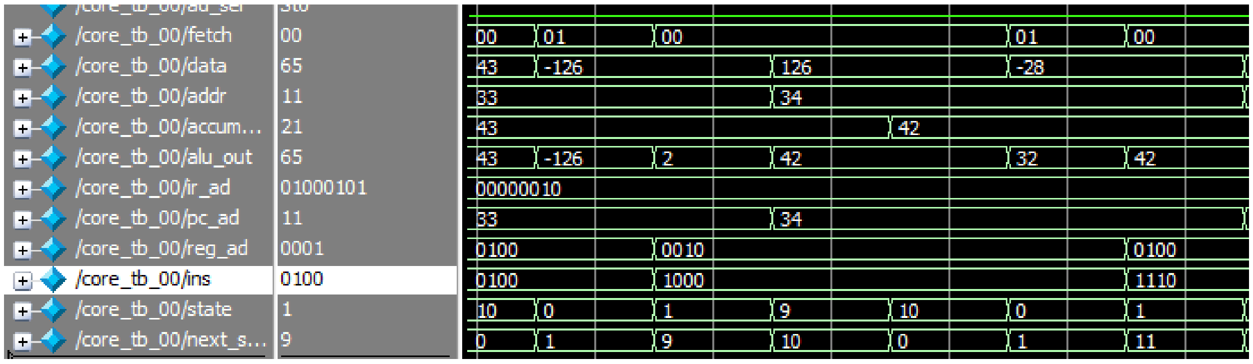
Testing Logical Operation (SUB)



```
/core_tb_00/fetch      00        00 | 01 | 00
/core_tb_00/data       65        53 | 114 | 126
/core_tb_00/addr       11        28 | 29
/core_tb_00/accum...   21        126 | 53 | -73
/core_tb_00/alu_out    65        53 | 114 | -61 | -73 | 57
/core_tb_00/ir_ad      01000101  00000001
/core_tb_00/pc_ad      11        28 | 29
/core_tb_00/reg_ad     0001      0011 | 0010
/core_tb_00/ins        0100      0100 | 0111
/core_tb_00/state      1         9 | 10 | 0 | 1 | 9 | 10
/core_tb_00/next_s...  9         10 | 0 | 1 | 9 | 10 | 0
```

Instruction ADD is tested by subtracting the preloaded Register 3rd Address, with the value of 53, with Register 2nd Address. For this simulation, the instruction address used are taken from ROM 28th Address. The process works in two steps. First, the CPU fetches instruction from ROM. The instruction address is 0111_0010, which means subtracting ALU value with Register 2nd Address value. The CPU then fetches Register 2nd Address value, which is 126. The ALU then adds the ALU with the loaded value. The Result could be seen in the alu_out which is -73. The value in the alu_out is sent to the accumulator.

Testing Logical Operation (AND)

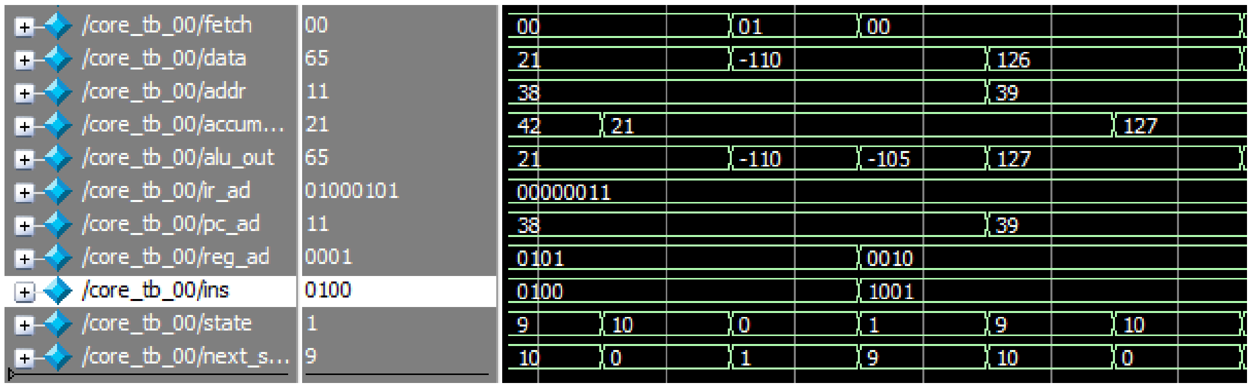


```
/core_tb_00/fetch      00        00 | 01 | 00 | 01 | 00
/core_tb_00/data       65        43 | -126 | 126 | -28
/core_tb_00/addr       11        33 | 34
/core_tb_00/accum...   21        43 | 42
/core_tb_00/alu_out    65        43 | -126 | 2 | 42 | 32 | 42
/core_tb_00/ir_ad      01000101  00000010
/core_tb_00/pc_ad      11        33 | 34
/core_tb_00/reg_ad     0001      0100 | 0010 | 0100
/core_tb_00/ins        0100      0100 | 1000 | 1110
/core_tb_00/state      1         10 | 0 | 1 | 9 | 10 | 0 | 1
/core_tb_00/next_s...  9         0 | 1 | 9 | 10 | 0 | 1 | 11
```

Instruction AND is tested by running the logical operation AND to the preloaded Register 4th Address, with the value of 43, with Register 2nd Address. For this simulation, the instruction address used are taken from ROM 33rd Address. The process works in two steps. First, the CPU fetches instruction from ROM. The instruction address is 1000_0010, which means running the logical operation AND to the ALU value with Register 2nd Address value. The CPU then fetches Register 2nd Address value, which is 126. The ALU then runs the logical operation AND to the ALU with the loaded value. The Result could be seen in the alu_out which is 42. The value in the alu_out is sent to the accumulator.

Jovanko Kenshian
00000032025

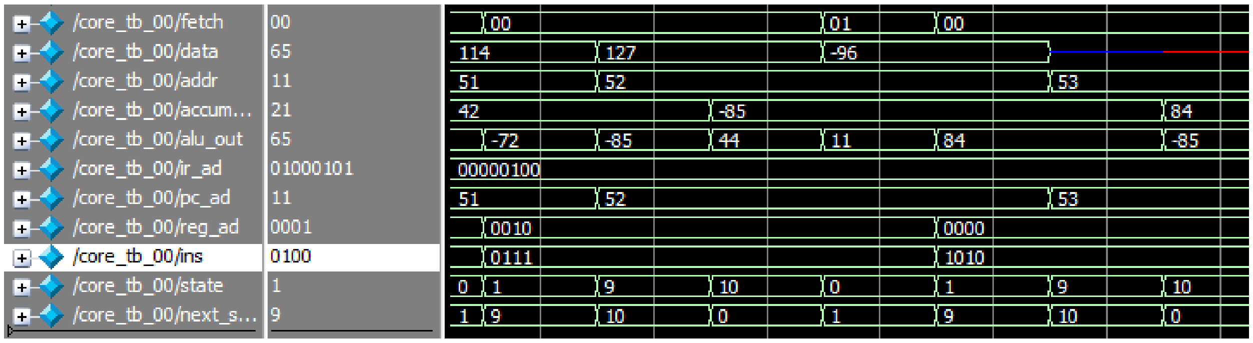
Testing Logical Operation (OR)



```
/core_tb_00/fetch      00
/core_tb_00/data       65
/core_tb_00/addr       11
/core_tb_00/accum...   21
/core_tb_00/alu_out    65
/core_tb_00/ir_ad      01000101
/core_tb_00/pc_ad      11
/core_tb_00/reg_ad     0001
/core_tb_00/ins        0100
/core_tb_00/state      1
/core_tb_00/next_s...  9
```

Instruction OR is tested by running the logical operation OR to the preloaded Register 5th Address, with the value of 21, with Register 2nd Address. For this simulation, the instruction address used are taken from ROM 38th Address. The process works in two steps. First, the CPU fetches instruction from ROM. The instruction address is 1001_0010, means running the logical operation OR to the ALU value with Register 2nd Address value. The CPU then fetches Register 2nd Address value, which is 126. The ALU then runs the logical operation OR to the ALU with the loaded value. The Result could be seen in the alu_out which is 127. The value in the alu_out is sent to the accumulator.

Testing Logical Operation (NOT)

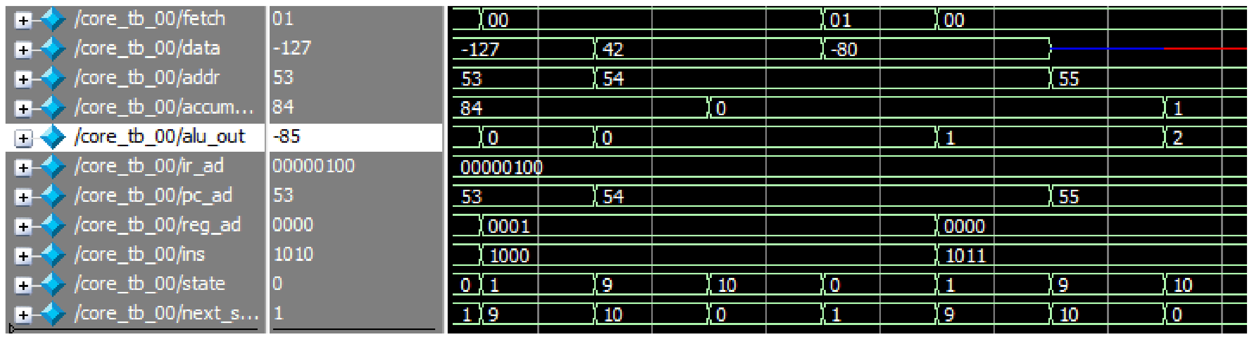


```
/core_tb_00/fetch      00
/core_tb_00/data       65
/core_tb_00/addr       11
/core_tb_00/accum...   21
/core_tb_00/alu_out    65
/core_tb_00/ir_ad      01000101
/core_tb_00/pc_ad      11
/core_tb_00/reg_ad     0001
/core_tb_00/ins        0100
/core_tb_00/state      1
/core_tb_00/next_s...  9
```

Instruction NOT is tested by running the logical operation NOT to the Accumulator value. For this simulation, the instruction address used are taken from ROM 38th Address. The process works by running the logical operation NOT in the ALU to the Accumulator value, which has the value of -85. The Result could be seen in the alu_out which is 84. The value in the alu_out is sent to the accumulator.

Jovanko Kenshian
00000032025

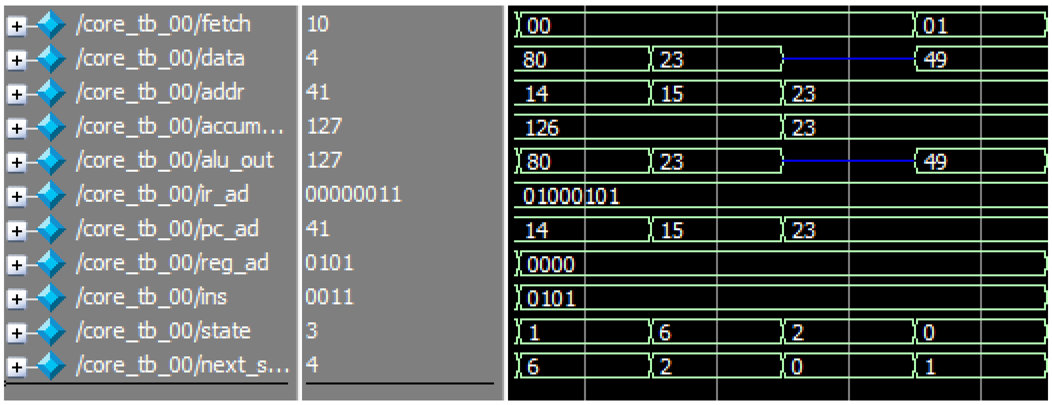
Testing Logical Operation (INC)



/core_tb_00/fetch	01
/core_tb_00/data	-127
/core_tb_00/addr	53
/core_tb_00/accum...	84
/core_tb_00/alu_out	-85
/core_tb_00/ir_ad	00000100
/core_tb_00/pc_ad	53
/core_tb_00/reg_ad	0000
/core_tb_00/ins	1010
/core_tb_00/state	0
/core_tb_00/next_s...	1

Instruction INC is tested incrementing the Accumulator value. For this simulation, the instruction address used are taken from ROM 38th Address. The process works by incrementing the Accumulator value, which has the value of 0. The Result could be seen in the alu_out which is 1. The value in the alu_out is sent to the accumulator.

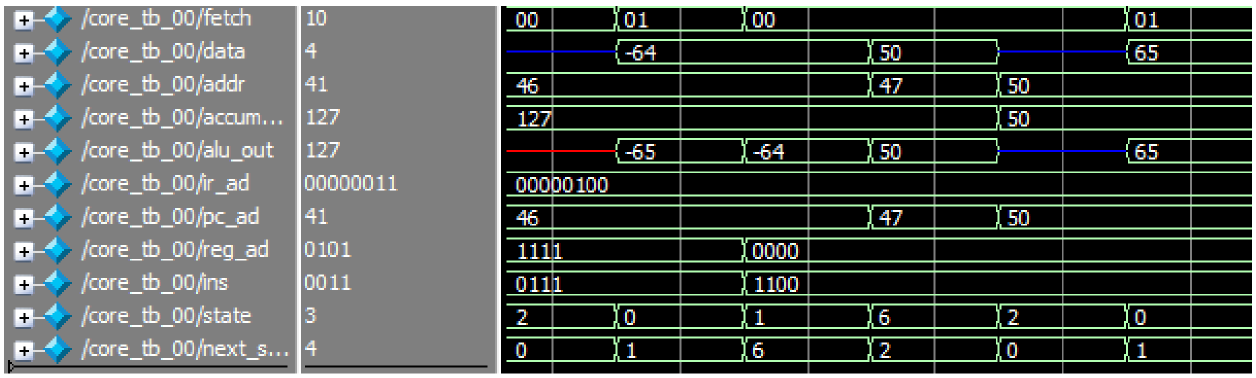
Testing Jump (JMP)



/core_tb_00/fetch	10
/core_tb_00/data	4
/core_tb_00/addr	41
/core_tb_00/accum...	127
/core_tb_00/alu_out	127
/core_tb_00/ir_ad	00000011
/core_tb_00/pc_ad	41
/core_tb_00/reg_ad	0101
/core_tb_00/ins	0011
/core_tb_00/state	3
/core_tb_00/next_s...	4

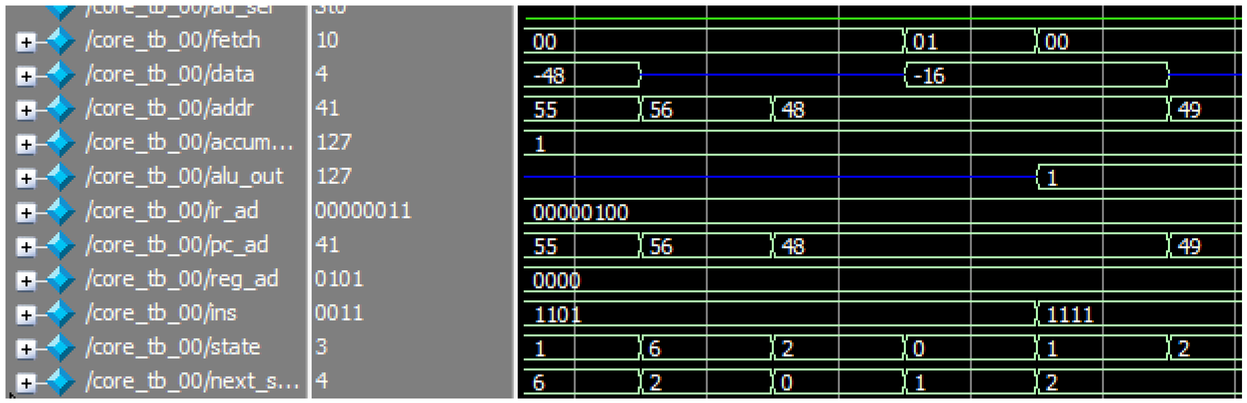
Instruction JMP is tested by jumping the PC to 23. For this simulation, the instruction address used are taken from ROM 14th and 15th Address. The process works in two steps. First, the CPU reads the instruction from the value fetched from the ROM 14th Address. It then understands that it needs to jump. The PC is then incremented to get the value from the ROM 15th Address. The value is then transferred to the PC and the PC replaced the PC value with the value from the ROM 15th Address, which is 23.

Testing ACALL (ACL)



Instruction ACL is tested by jumping the PC to 50. For this simulation, the instruction address used are taken from ROM 46th and 47th Address. The process works in three steps. First, the CPU reads the instruction from the value fetched from the ROM 46th Address. It then understands that it needs to jump. The PC is then incremented to get the value from the ROM 47th Address. The PC then stores its PC value in a stack so it could return after the ACALL function is done. The value is then transferred to the PC and the PC replaced the PC value with the value from the ROM 47th Address, which is 50.

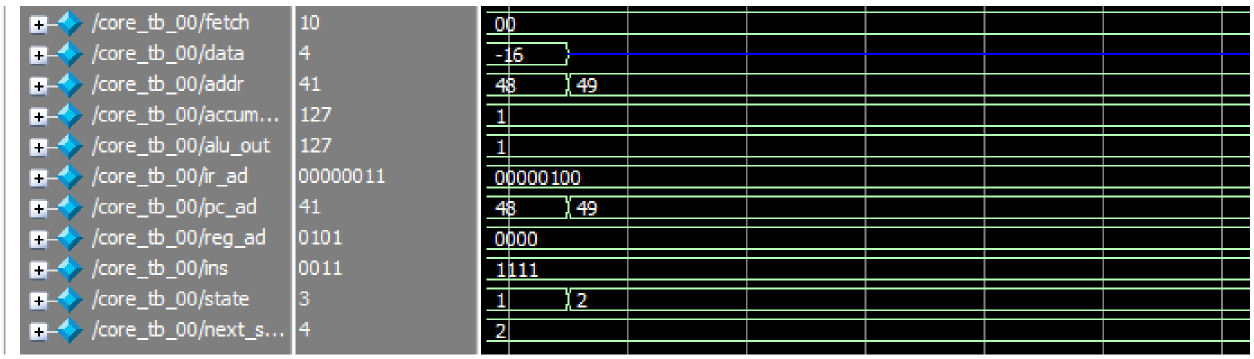
Testing RET (RET)



Instruction RET is tested by returning the PC back to the PC value before ACALL is called, which is 48. For this simulation, the instruction address used are taken from ROM 55th Address. The process works by reading the instruction from the value fetched from the ROM 55th Address. It then understands that it needs to return the PC value to before ACALL was called. The PC then pop the stack and changes the PC value to the popped value, which is 47. Since the PC doesn't needs to run the 47th counter, it directly increments the PC value. The result is 48.

Jovanko Kenshian
00000032025

Testing HALT (HLT)



```
/core_tb_00/fetch      10         00
/core_tb_00/data       4          -16
/core_tb_00/addr       41         48    49
/core_tb_00/accum...   127        1
/core_tb_00/alu_out    127        1
/core_tb_00/ir_ad      00000011   00000100
/core_tb_00/pc_ad      41         48    49
/core_tb_00/reg_ad     0101       0000
/core_tb_00/ins        0011       1111
/core_tb_00/state      3          1     2
/core_tb_00/next_s...  4          2
```

Instruction HALT is tested by giving the instruction input of 1111 to the CPU. When the CPU reads the input of 1111, the CPU stops processing data and freezes itself by looping in the same state.