GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

2022 FALL CSE331 COMPUTER ORGANIZATION

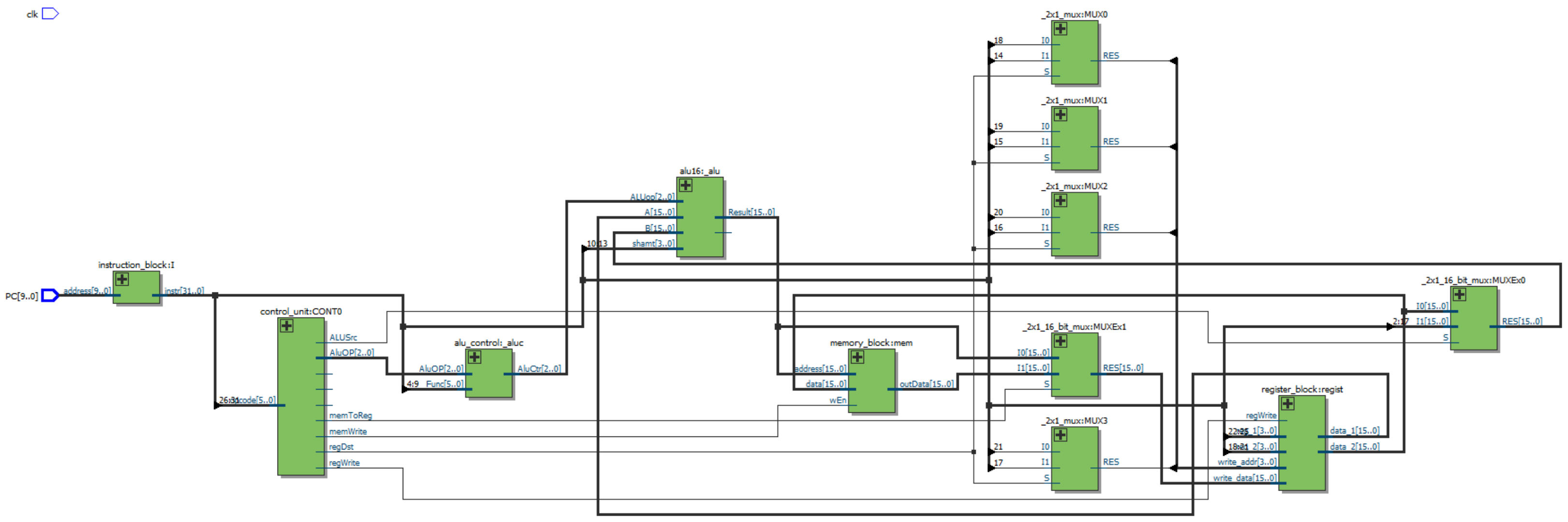
MERT GÜRŞİMŞİR

1901042646

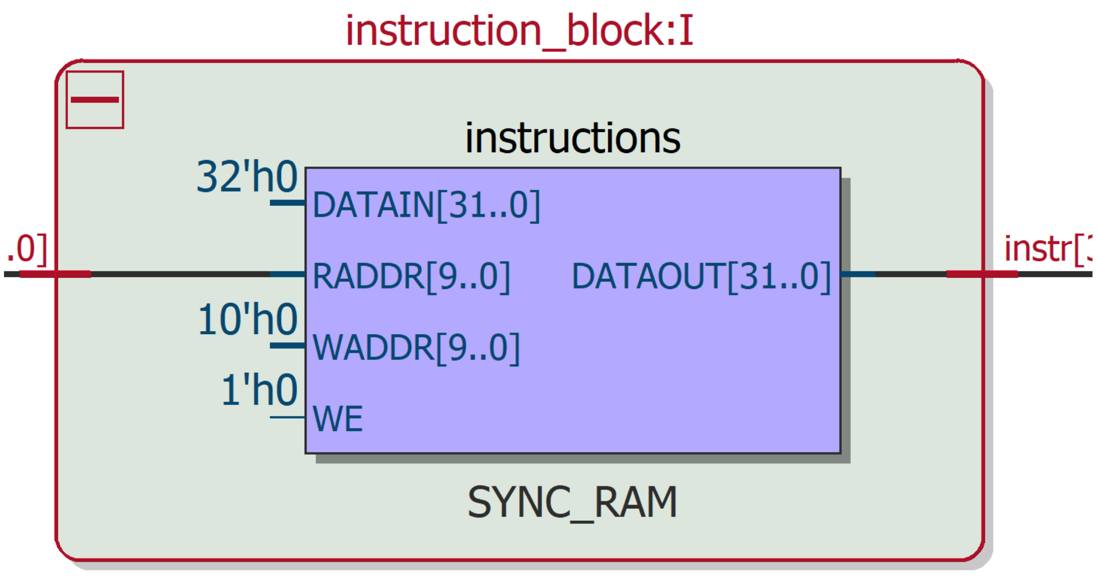
PROJECT REPORT

IMPORTANT NOTE: I have used my own computer file path for the txt files for instruction memory, data memory, and registers.

RTL VIEWER RESULTS WITH EXPLANATIONS

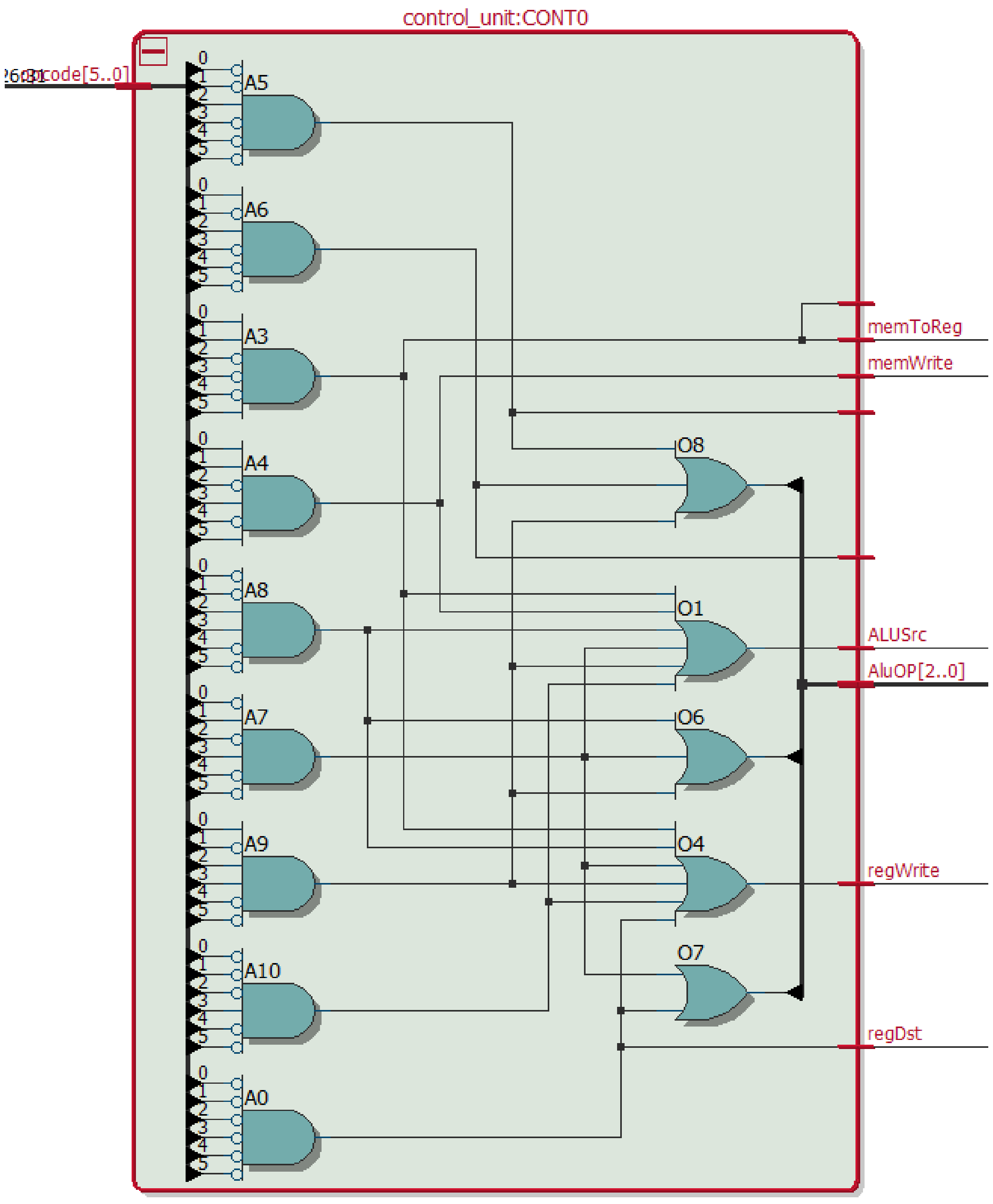


This is the general overview of the datapath. Here at first we look at the instruction memory to get the instruction. Instruction block:



Here we get the address of the instruction and give the instruction located at that address.

Then, to solve which instruction is this, we have to look at the opcode field. So control unit shows up.



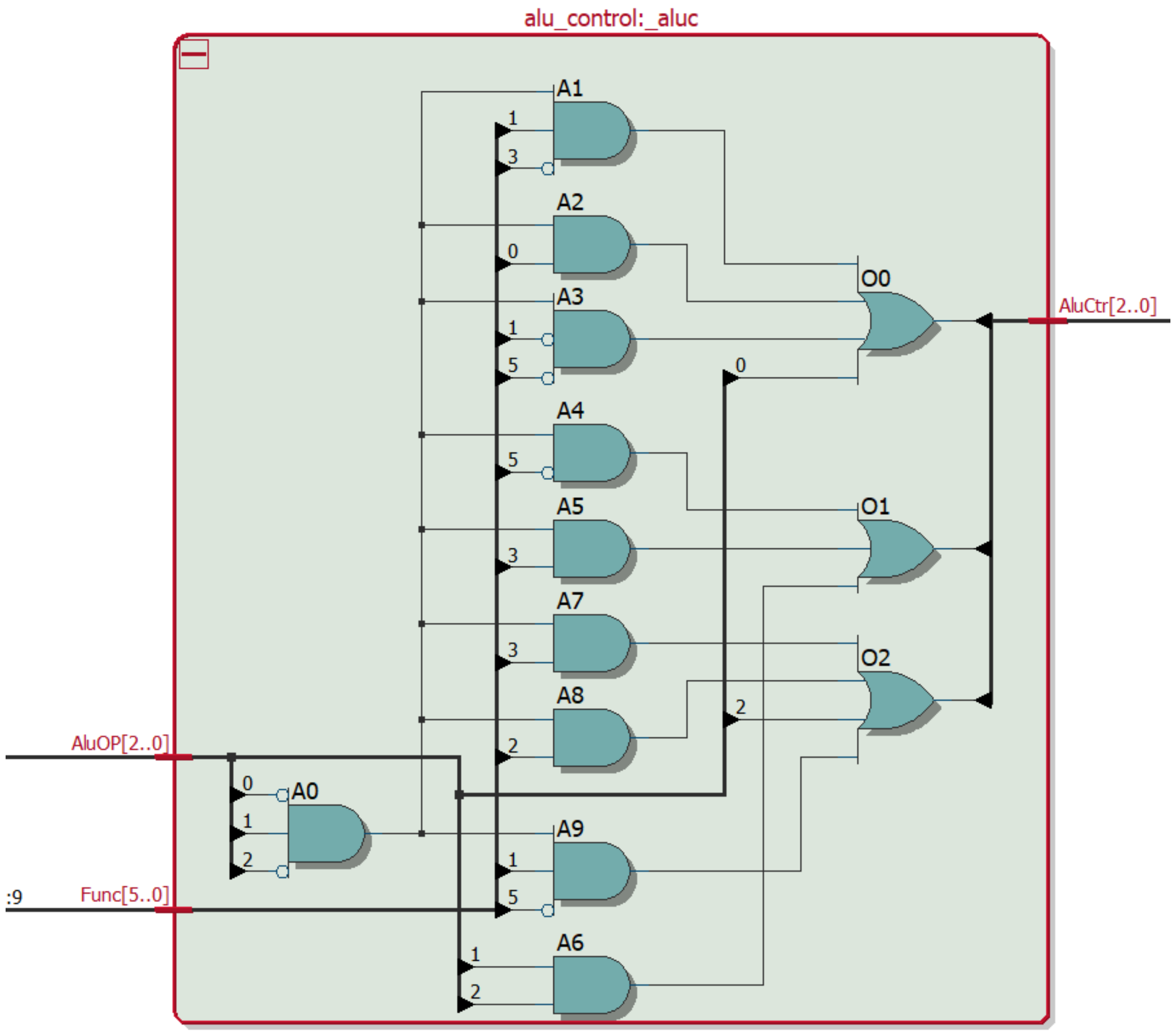
It is better to analyze why I implemented control unit in this way so here is the table:

INSTRUCTION	beq	bne	mem to Reg	alu src	reg dst	mem write	reg write	memread	ALUOP	OPCODE	FUNCTION FIELD	ALU ctr
add	0	0	0	0	1	0	1	0	010	000000	100000	000
sub	0	0	0	0	1	0	1	0	010	000000	100010	001
srl	0	0	0	0	1	0	1	0	010	000000	000010	111
sll	0	0	0	0	1	0	1	0	010	000000	000000	011
and	0	0	0	0	1	0	1	0	010	000000	100100	100
or	0	0	0	0	1	0	1	0	010	000000	100101	101
slt	0	0	0	0	1	0	1	0	010	000000	101010	110
jr	0	0	0	0	1	0	1	0		000000	001000	
j										000010	xxxxxx	
jal										000011	xxxxxx	
lw	0	0	1	1	0	0	1	1	000	100011	xxxxxx	000
sw	0	0	X	1	X	1	0	0	000	101011	xxxxxx	000
beq	1	0	X	0	X	0	0	0	001	000100	xxxxxx	001
bne	0	1	X	0	X	0	0	0	001	000101	xxxxxx	001
slti	0	0	0	1	0	0	1	0	110	001010	xxxxxx	110
andi	0	0	0	1	0	0	1	0	100	001100	xxxxxx	100
ori	0	0	0	1	0	0	1	0	101	001101	xxxxxx	101
addi	0	0	0	1	0	0	1	0	000	001000	xxxxxx	000

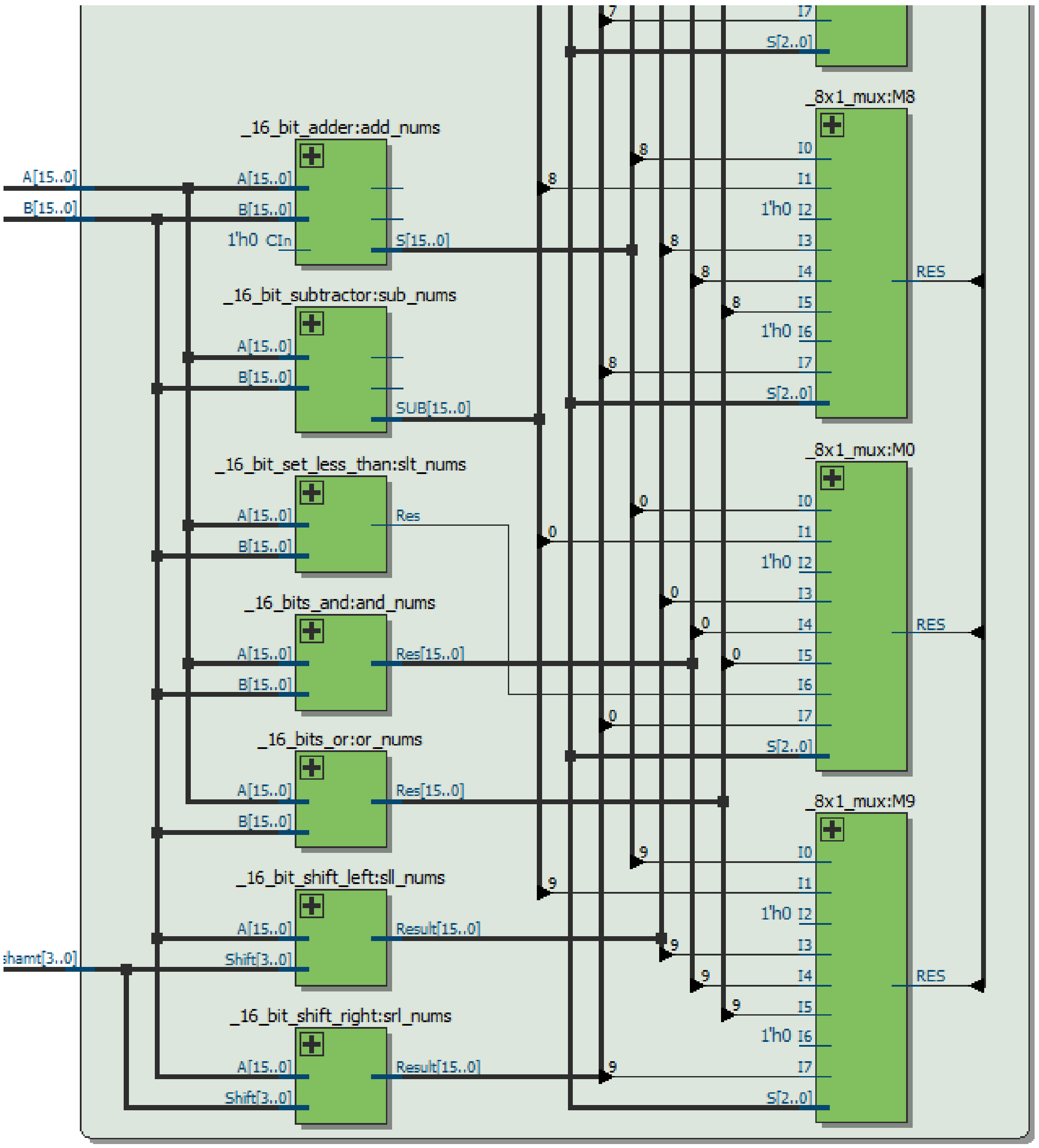
Table 1.0

I have used MIPS green sheet for the OPCODE and FUNCTION fields. I have chosen proper ALUOP values for the instructions. I have examined each control bits one by one and "AND" opcode bits to get the instruction's name. Each instruction needs different control signals. After I had written down the needs for each instruction, I have simply combined them in control unit.

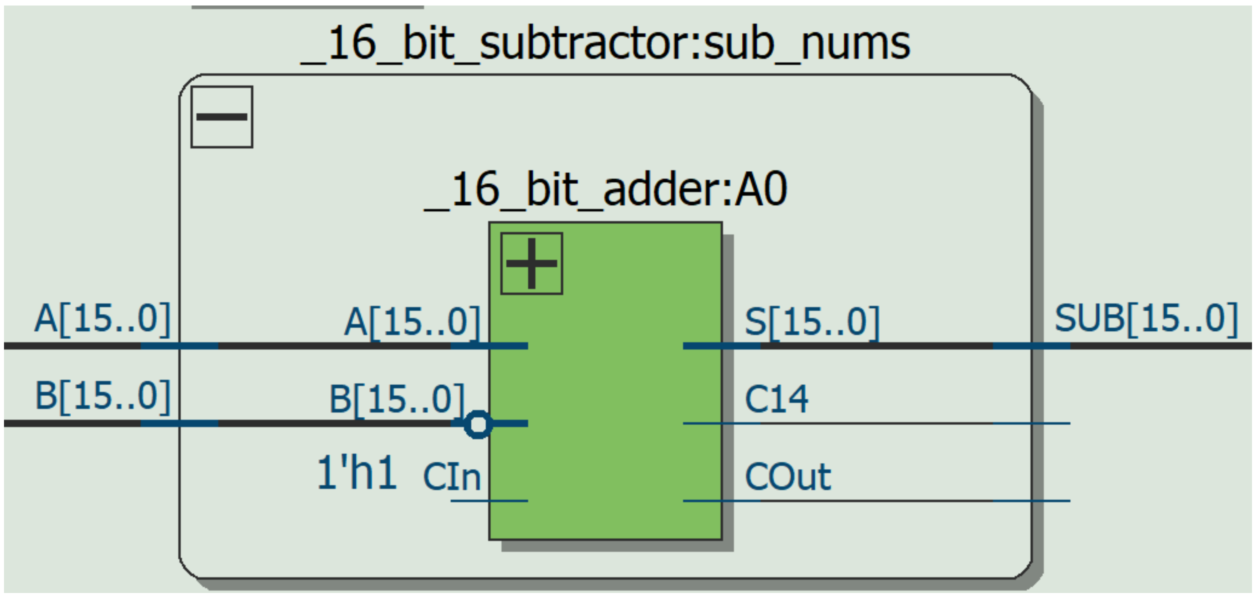
I have sent ALUOP values to ALU CONTROL with FUNCTION field to determine the ALU operation.



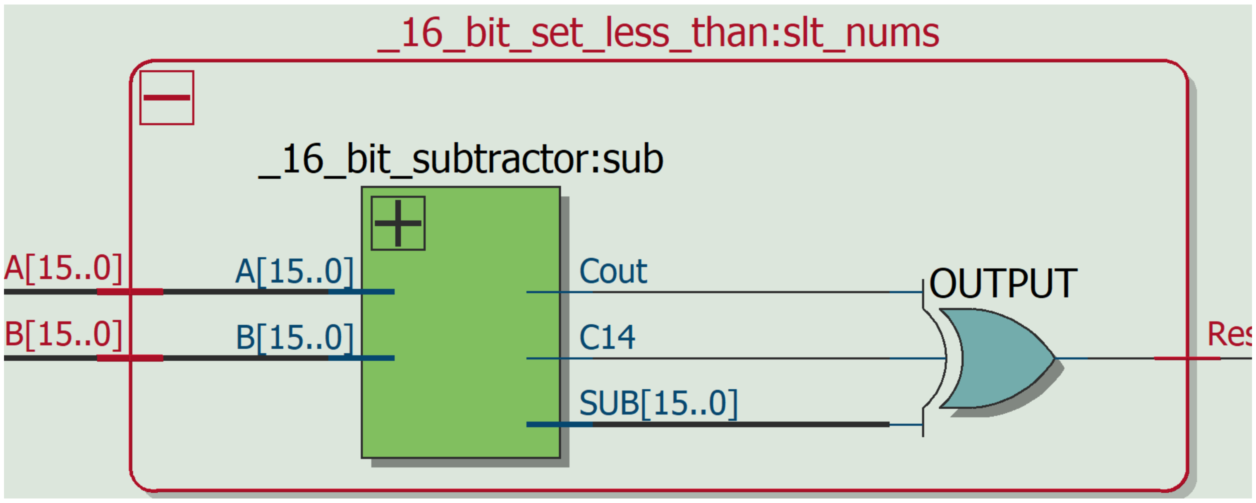
Again, I have examined the table I have done and determined the each 3 bits of ALUCTR (the output) one by one here in ALU CONTROL. Then ALU comes into play.



ALU is the place where we do operations. In adder module, I have used full adders with half adders to implement 16 bit adder for our words. In substractor module, I have used the full adder again with one operand reversed and gave it 1 as carry in.

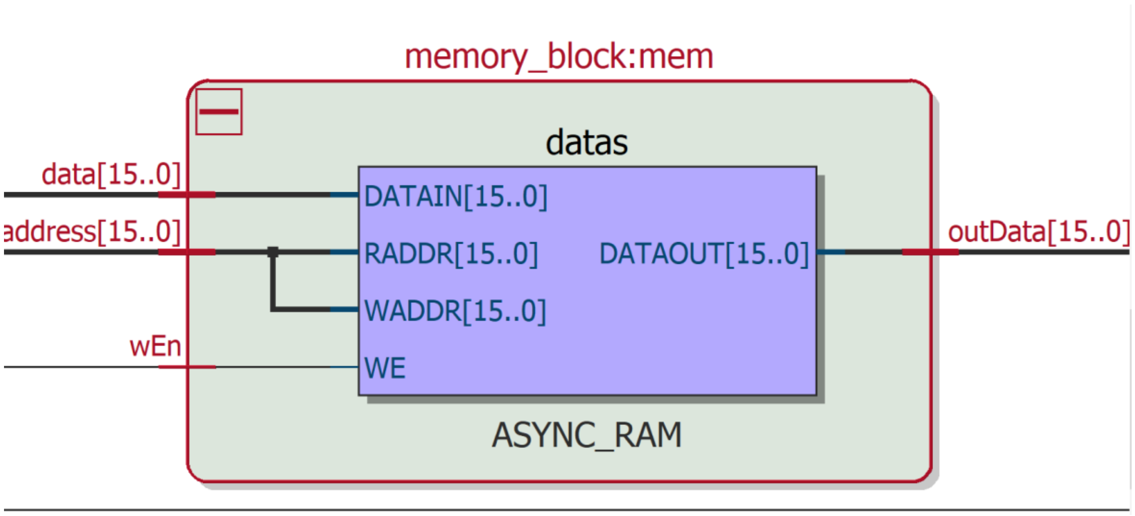


For the set less than operation, I have used the subtractor and decide if output is negative or not by using xor gate.

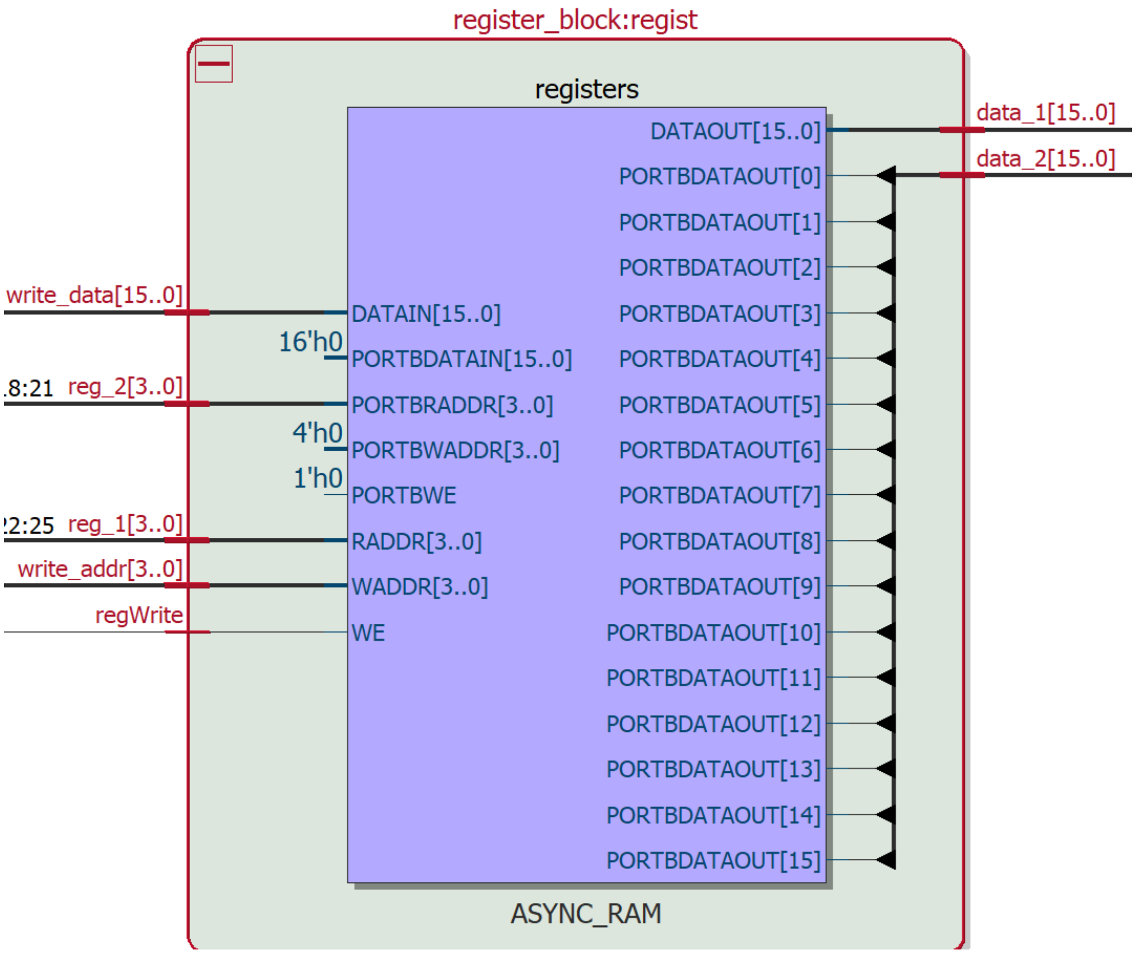


Other modules simply implement logical operations bit by bit.

After I got all the results for necessary operations, I have chosen among them with 8x1 multiplexers.



Memory block gets address, data, write enable and its output is read data.

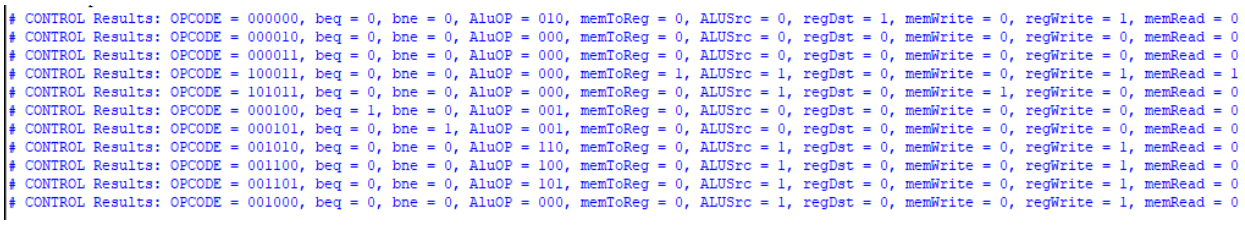


Register is very similar to data memory. It gets 3 addresses (rs, rt, rd) and data to be written and also regWrite signal that tells register to write. Its output is read registers.

I have tried to stick to the datapath that we have seen in lectures. I have used mux for write register address, ALU input, and data that is going to be written to register. Combining them properly was the way to achieve an operating processor.

TESTS

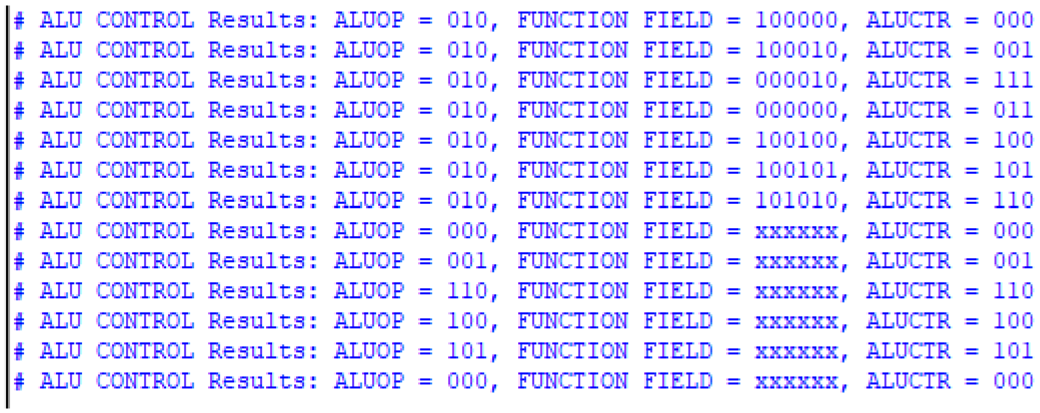
Control unit test results:



```
# CONTROL Results: OPCODE = 000000, beq = 0, bne = 0, AluOP = 010, memToReg = 0, ALUSrc = 0, regDst = 1, memWrite = 0, regWrite = 1, memRead = 0
# CONTROL Results: OPCODE = 000010, beq = 0, bne = 0, AluOP = 000, memToReg = 0, ALUSrc = 0, regDst = 0, memWrite = 0, regWrite = 0, memRead = 0
# CONTROL Results: OPCODE = 000011, beq = 0, bne = 0, AluOP = 000, memToReg = 0, ALUSrc = 0, regDst = 0, memWrite = 0, regWrite = 0, memRead = 0
# CONTROL Results: OPCODE = 100011, beq = 0, bne = 0, AluOP = 000, memToReg = 1, ALUSrc = 1, regDst = 0, memWrite = 0, regWrite = 1, memRead = 1
# CONTROL Results: OPCODE = 101011, beq = 0, bne = 0, AluOP = 000, memToReg = 0, ALUSrc = 1, regDst = 0, memWrite = 1, regWrite = 0, memRead = 0
# CONTROL Results: OPCODE = 000100, beq = 1, bne = 0, AluOP = 001, memToReg = 0, ALUSrc = 0, regDst = 0, memWrite = 0, regWrite = 0, memRead = 0
# CONTROL Results: OPCODE = 000101, beq = 0, bne = 1, AluOP = 001, memToReg = 0, ALUSrc = 0, regDst = 0, memWrite = 0, regWrite = 0, memRead = 0
# CONTROL Results: OPCODE = 001010, beq = 0, bne = 0, AluOP = 110, memToReg = 0, ALUSrc = 1, regDst = 0, memWrite = 0, regWrite = 1, memRead = 0
# CONTROL Results: OPCODE = 001100, beq = 0, bne = 0, AluOP = 100, memToReg = 0, ALUSrc = 1, regDst = 0, memWrite = 0, regWrite = 1, memRead = 0
# CONTROL Results: OPCODE = 001101, beq = 0, bne = 0, AluOP = 101, memToReg = 0, ALUSrc = 1, regDst = 0, memWrite = 0, regWrite = 1, memRead = 0
# CONTROL Results: OPCODE = 001000, beq = 0, bne = 0, AluOP = 000, memToReg = 0, ALUSrc = 1, regDst = 0, memWrite = 0, regWrite = 1, memRead = 0
```

These results can be checked with the Table 1.0. Here only input is opcode and all others are outputs. I have looked where a specific bit becomes 1 and created the logical equation accordingly. For example, memWrite becomes 1 only if the instruction is sw so I have checked if instruction is sw with opcode.

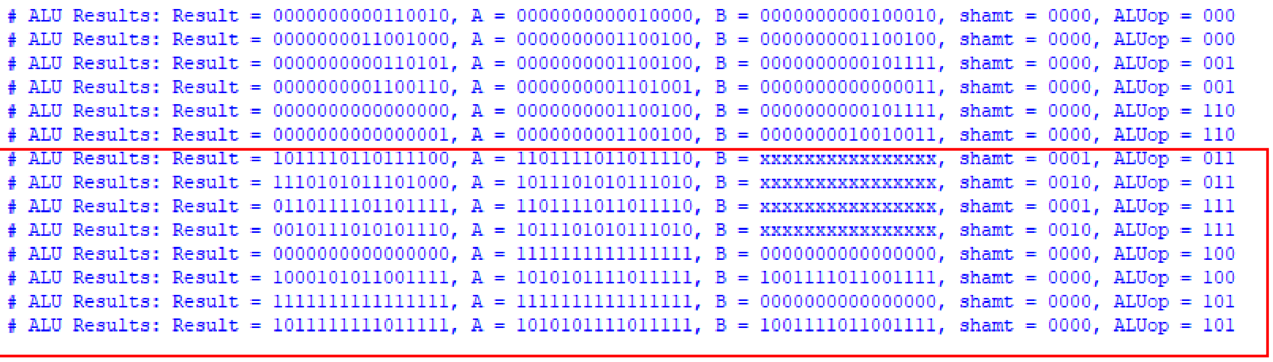
ALU Control test results:



```
# ALU CONTROL Results: ALUOP = 010, FUNCTION FIELD = 100000, ALUCTR = 000
# ALU CONTROL Results: ALUOP = 010, FUNCTION FIELD = 100010, ALUCTR = 001
# ALU CONTROL Results: ALUOP = 010, FUNCTION FIELD = 000010, ALUCTR = 111
# ALU CONTROL Results: ALUOP = 010, FUNCTION FIELD = 000000, ALUCTR = 011
# ALU CONTROL Results: ALUOP = 010, FUNCTION FIELD = 100100, ALUCTR = 100
# ALU CONTROL Results: ALUOP = 010, FUNCTION FIELD = 100101, ALUCTR = 101
# ALU CONTROL Results: ALUOP = 010, FUNCTION FIELD = 101010, ALUCTR = 110
# ALU CONTROL Results: ALUOP = 000, FUNCTION FIELD = xxxxxx, ALUCTR = 000
# ALU CONTROL Results: ALUOP = 001, FUNCTION FIELD = xxxxxx, ALUCTR = 001
# ALU CONTROL Results: ALUOP = 110, FUNCTION FIELD = xxxxxx, ALUCTR = 110
# ALU CONTROL Results: ALUOP = 100, FUNCTION FIELD = xxxxxx, ALUCTR = 100
# ALU CONTROL Results: ALUOP = 101, FUNCTION FIELD = xxxxxx, ALUCTR = 101
# ALU CONTROL Results: ALUOP = 000, FUNCTION FIELD = xxxxxx, ALUCTR = 000
```

These results can also be checked with the Table 1.0. Here ALUOP and FUNCTION FIELD are the inputs of ALU Control and ALUCTR is the output of it. I have determined each bit of the ALUCTR by looking when it becomes 1. I didn't write all the equations, just necessary ones just as we did at lectures.

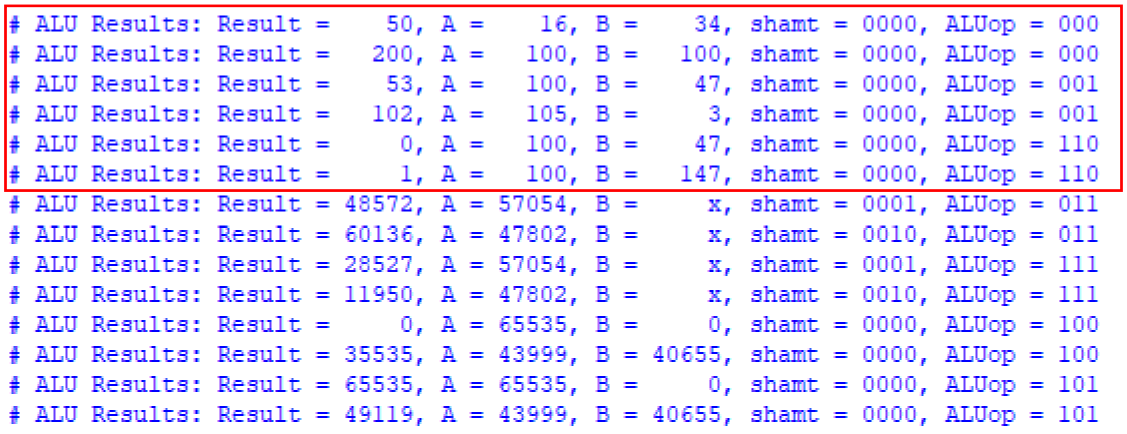
ALU test results for binary operations:



```
# ALU Results: Result = 0000000000110010, A = 0000000000010000, B = 0000000000100010, shamt = 0000, ALUop = 000
# ALU Results: Result = 0000000011001000, A = 0000000001100100, B = 0000000001100100, shamt = 0000, ALUop = 000
# ALU Results: Result = 0000000000110101, A = 0000000001100100, B = 0000000000101111, shamt = 0000, ALUop = 001
# ALU Results: Result = 0000000001100110, A = 0000000001101001, B = 0000000000000011, shamt = 0000, ALUop = 001
# ALU Results: Result = 0000000000000000, A = 0000000001100100, B = 0000000000101111, shamt = 0000, ALUop = 110
# ALU Results: Result = 0000000000000001, A = 0000000001100100, B = 0000000010010011, shamt = 0000, ALUop = 110
# ALU Results: Result = 1011110110111100, A = 1101111011011110, B = xxxxxxxxxxxxxxxx, shamt = 0001, ALUop = 011
# ALU Results: Result = 1110101011101000, A = 1011101010111010, B = xxxxxxxxxxxxxxxx, shamt = 0010, ALUop = 011
# ALU Results: Result = 0110111101101111, A = 1101111011011110, B = xxxxxxxxxxxxxxxx, shamt = 0001, ALUop = 111
# ALU Results: Result = 0010111010101110, A = 1011101010111010, B = xxxxxxxxxxxxxxxx, shamt = 0010, ALUop = 111
# ALU Results: Result = 0000000000000000, A = 1111111111111111, B = 0000000000000000, shamt = 0000, ALUop = 100
# ALU Results: Result = 1000101011001111, A = 1010101111011111, B = 1001111011001111, shamt = 0000, ALUop = 100
# ALU Results: Result = 1111111111111111, A = 1111111111111111, B = 0000000000000000, shamt = 0000, ALUop = 101
# ALU Results: Result = 1011111111011111, A = 1010101111011111, B = 1001111011001111, shamt = 0000, ALUop = 101
```

Here instructions are sll, sll, srl, srl, and, and, or, or in order. You can check the ALUop to understand which operation is done, shamt to shift number, and A and B to see the operands.

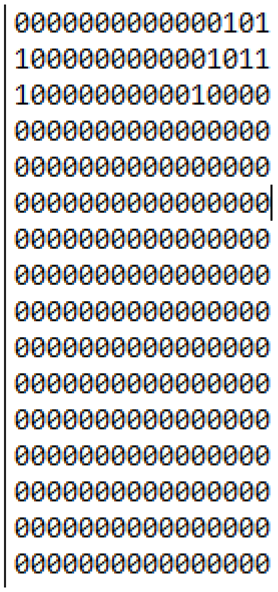
ALU test results for addition, subtraction and set less than in order:



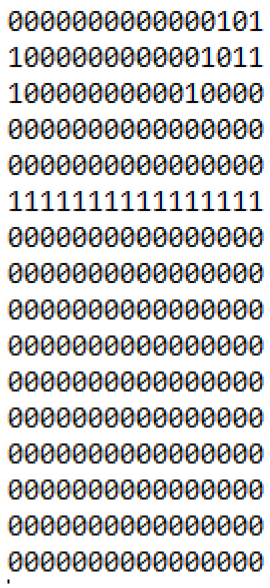
```
# ALU Results: Result =    50, A =    16, B =    34, shamt = 0000, ALUop = 000
# ALU Results: Result =   200, A =   100, B =   100, shamt = 0000, ALUop = 000
# ALU Results: Result =    53, A =   100, B =    47, shamt = 0000, ALUop = 001
# ALU Results: Result =   102, A =   105, B =     3, shamt = 0000, ALUop = 001
# ALU Results: Result =     0, A =   100, B =    47, shamt = 0000, ALUop = 110
# ALU Results: Result =     1, A =   100, B =   147, shamt = 0000, ALUop = 110
# ALU Results: Result = 48572, A = 57054, B =     x, shamt = 0001, ALUop = 011
# ALU Results: Result = 60136, A = 47802, B =     x, shamt = 0010, ALUop = 011
# ALU Results: Result = 28527, A = 57054, B =     x, shamt = 0001, ALUop = 111
# ALU Results: Result = 11950, A = 47802, B =     x, shamt = 0010, ALUop = 111
# ALU Results: Result =     0, A = 65535, B =     0, shamt = 0000, ALUop = 100
# ALU Results: Result = 35535, A = 43999, B = 40655, shamt = 0000, ALUop = 100
# ALU Results: Result = 65535, A = 65535, B =     0, shamt = 0000, ALUop = 101
# ALU Results: Result = 49119, A = 43999, B = 40655, shamt = 0000, ALUop = 101
```

Here instructions are add, add, sub, sub, slt, slt in order. For all of these instructions, adder is used.

Register test results:



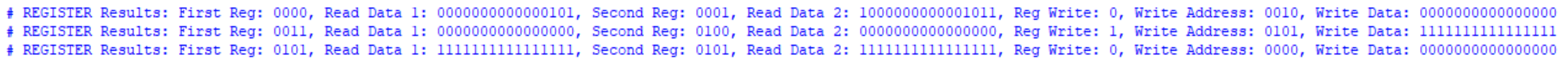
```
0000000000000101
1000000000001011
1000000000010000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```



```
0000000000000101
1000000000001011
1000000000010000
0000000000000000
0000000000000000
1111111111111111
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

before

after



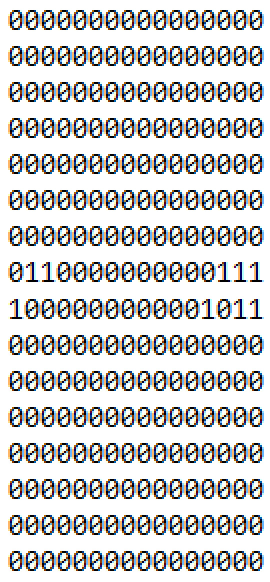
```
# REGISTER Results: First Reg: 0000, Read Data 1: 0000000000000101, Second Reg: 0001, Read Data 2: 1000000000001011, Reg Write: 0, Write Address: 0010, Write Data: 0000000000000000
# REGISTER Results: First Reg: 0011, Read Data 1: 0000000000000000, Second Reg: 0100, Read Data 2: 0000000000000000, Reg Write: 1, Write Address: 0101, Write Data: 1111111111111111
# REGISTER Results: First Reg: 0101, Read Data 1: 1111111111111111, Second Reg: 0101, Read Data 2: 1111111111111111, Reg Write: 0, Write Address: 0000, Write Data: 0000000000000000
```

Firstly we read data at index 0, then at index 1 and don’t write in first test.

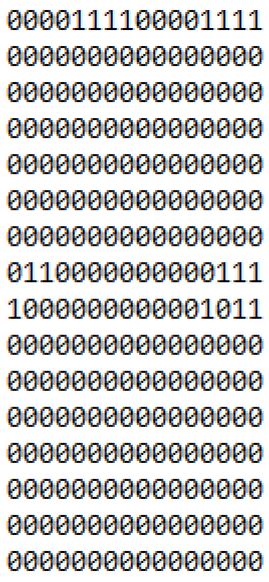
In second test, we read third register and fourth register and try to write to register at index 5 the data 1111111111111111.

In third test, we read the correct result at index 5.

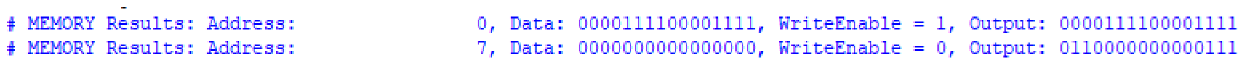
Memory test results (only first few contents are used out of 65536 words):



```
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0110000000000111
1000000000001011
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```



```
0000111100001111
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0110000000000111
1000000000001011
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

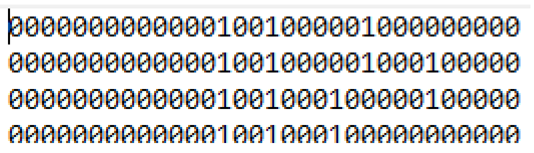


```
# MEMORY Results: Address:              0, Data: 0000111100001111, WriteEnable = 1, Output: 0000111100001111
# MEMORY Results: Address:              7, Data: 0000000000000000, WriteEnable = 0, Output: 0110000000000111
```

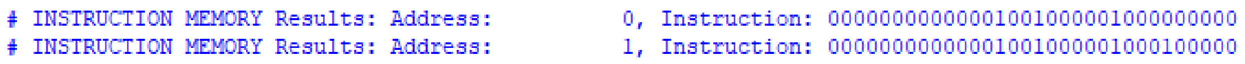
Firstly we try to write to adres 0 the data 0000111100001111.

In second test, we read 7th register content and see the output correctly.

Instruction memory test results with first few instructions:



```
00000000000001001000001000000000
00000000000001001000001000100000
00000000000001001000100000100000
00000000000001001000100000000000
```

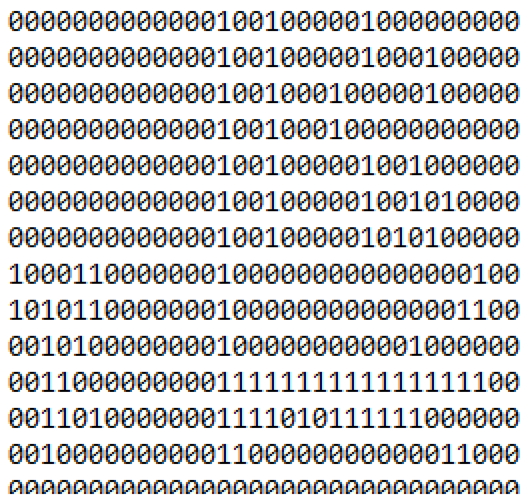


```
# INSTRUCTION MEMORY Results: Address:          0, Instruction: 00000000000001001000001000000000
# INSTRUCTION MEMORY Results: Address:          1, Instruction: 00000000000001001000001000100000
```

Firstly we read instruction at address 0.

In second test, we read instruction at address 1.

MIPS working instructions' tests:

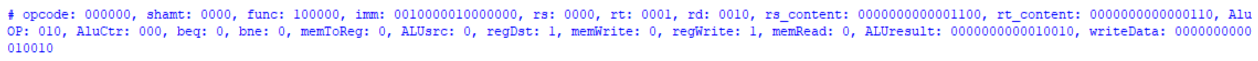


```
00000000000001001000001000000000
00000000000001001000001000100000
00000000000001001000100000100000
00000000000001001000100000000000
00000000000001001000001001000000
00000000000001001000001001010000
00000000000001001000001010100000
10001100000001000000000000000100
10101100000001000000000000001100
00101000000001000000000001000000
00110000000001111111111111111100
00110100000001111010111111000000
00100000000001100000000000011000
00000000000000000000000000000000
```

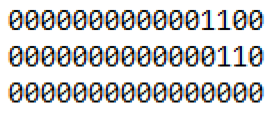
I have added 13 instructions for testing purposes to instruction memory which are:

- add
- sub
- srl
- sll
- and
- or
- slti
- lw
- sw
- slti
- andi
- ori
- addi

add instruction:

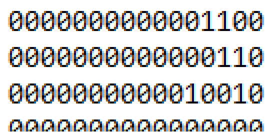


```
# opcode: 000000, shamt: 0000, func: 100000, imm: 0010000010000000, rs: 0000, rt: 0001, rd: 0010, rs_content: 0000000000001100, rt_content: 0000000000000110, Alu
OP: 010, AluCtr: 000, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000010010, writeData: 0000000000
010010
```



```
0000000000001100
0000000000000110
0000000000000000
```

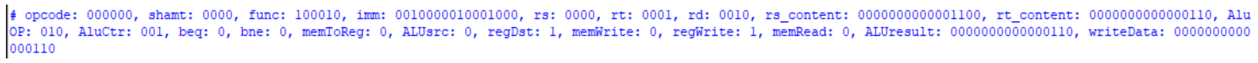
rs, rt, and rd contents at the beginning.



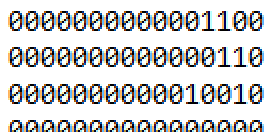
```
0000000000001100
0000000000000110
0000000000010010
0000000000000000
```

Same contents after running add instruction.

sub instruction:

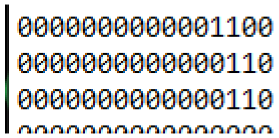


```
# opcode: 000000, shamt: 0000, func: 100010, imm: 0010000010001000, rs: 0000, rt: 0001, rd: 0010, rs_content: 0000000000001100, rt_content: 0000000000000110, Alu
OP: 010, AluCtr: 001, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000000110, writeData: 0000000000
000110
```



```
0000000000001100
0000000000000110
0000000000010010
0000000000000000
```

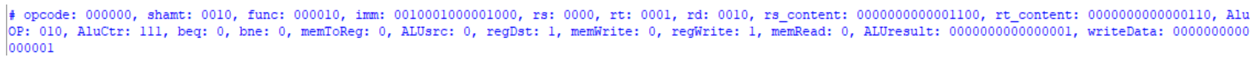
rs, rt, and rd contents at the beginning.



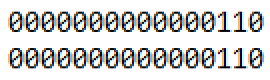
```
0000000000001100
0000000000000110
0000000000000110
0000000000000000
```

Same contents after running sub instruction.

srl (with shamt value 2) instruction:

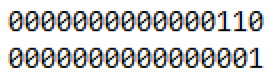


```
# opcode: 000000, shamt: 0010, func: 000010, imm: 0010001000001000, rs: 0000, rt: 0001, rd: 0010, rs_content: 0000000000001100, rt_content: 0000000000000110, Alu
OP: 010, AluCtr: 111, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000000001, writeData: 0000000000
000001
```



```
0000000000000110
0000000000000110
```

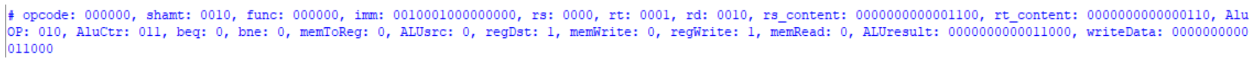
rt and rd contents at the beginning.



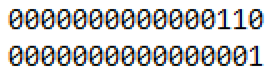
```
0000000000000110
0000000000000001
```

Same contents after running srl instruction.

sll (with shamt value 2) instruction:

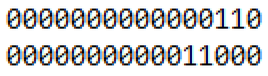


```
# opcode: 000000, shamt: 0010, func: 000000, imm: 0010001000000000, rs: 0000, rt: 0001, rd: 0010, rs_content: 0000000000001100, rt_content: 0000000000000110, Alu
OP: 010, AluCtr: 011, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000011000, writeData: 0000000000
011000
```



```
0000000000000110
0000000000000001
```

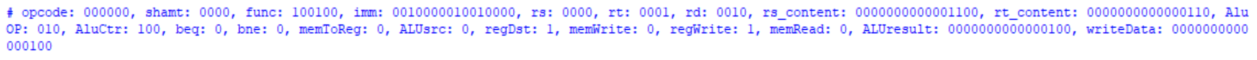
rt and rd contents at the beginning.



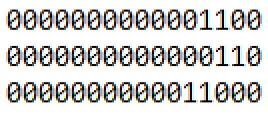
```
0000000000000110
0000000000011000
```

Same contents after running sll instruction.

and instruction:

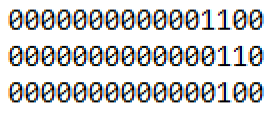


```
# opcode: 000000, shamt: 0000, func: 100100, imm: 0010000010010000, rs: 0000, rt: 0001, rd: 0010, rs_content: 0000000000001100, rt_content: 0000000000000110, Alu
OP: 010, AluCtr: 100, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000000100, writeData: 0000000000
000100
```



```
0000000000001100
0000000000000110
0000000000011000
```

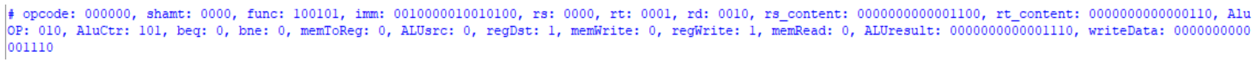
rs, rt, and rd contents at the beginning.



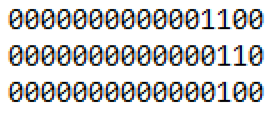
```
0000000000001100
0000000000000110
0000000000000100
```

Same contents after running and instruction.

or instruction:

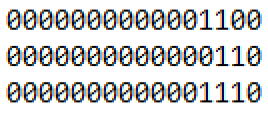


```
# opcode: 000000, shamt: 0000, func: 100101, imm: 0010000010010100, rs: 0000, rt: 0001, rd: 0010, rs_content: 0000000000001100, rt_content: 0000000000000110, Alu
OP: 010, AluCtr: 101, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000001110, writeData: 0000000000
001110
```



```
0000000000001100
0000000000000110
0000000000000100
```

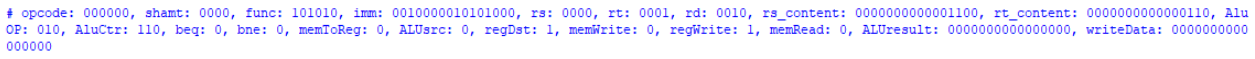
rs, rt, and rd contents at the beginning.



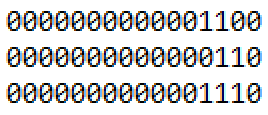
```
0000000000001100
0000000000000110
0000000000001110
```

Same contents after running and instruction.

slt instruction:

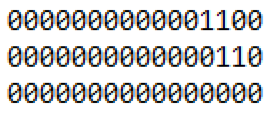


```
# opcode: 000000, shamt: 0000, func: 101010, imm: 0010000010101000, rs: 0000, rt: 0001, rd: 0010, rs_content: 0000000000001100, rt_content: 0000000000000110, Alu
OP: 010, AluCtr: 110, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000000000, writeData: 0000000000
000000
```



```
0000000000001100
0000000000000110
0000000000001110
```

rs, rt, and rd contents at the beginning.

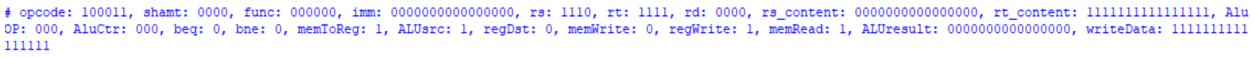


```
0000000000001100
0000000000000110
0000000000000000
```

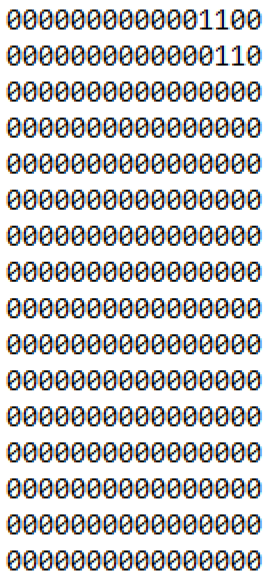
Same contents after running and instruction.

lw instruction:

Here, I am going to load Mem[$rs + 0] to rt register. I take rs register whose content is 0. So rt register will be equal to content of Mem[0]. rt register is going to be the last register.

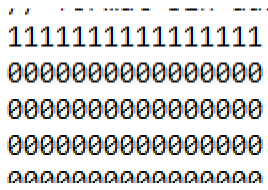


```
# opcode: 100011, shamt: 0000, func: 000000, imm: 0000000000000000, rs: 1110, rt: 1111, rd: 0000, rs_content: 0000000000000000, rt_content: 1111111111111111, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 1, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 1, ALUresult: 0000000000000000, writeData: 1111111111
111111
```



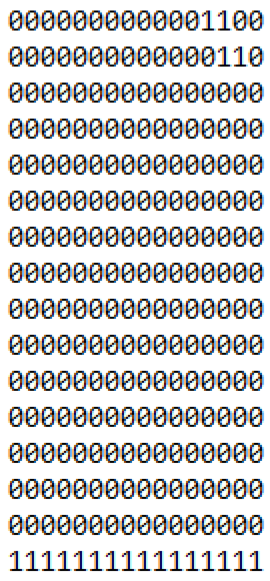
```
0000000000001100
0000000000000110
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

Register contents at the beginning.



```
1111111111111111
0000000000000000
0000000000000000
0000000000000000
```

First few contents of memory at the beginning.

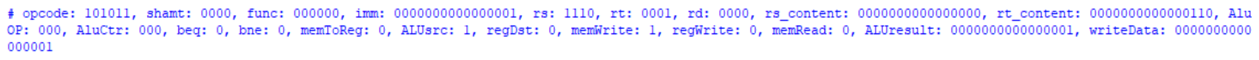


```
0000000000001100
0000000000000110
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
1111111111111111
```

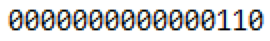
Register contents after running lw instruction.

sw instruction:

Here, I am going to store content of rt register to Mem[$rs + 1]. Content of rs is 0 so Mem[1] is going to be equal to content of rt register which is 110.

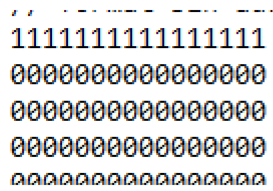


```
# opcode: 101011, shamt: 0000, func: 000000, imm: 0000000000000001, rs: 1110, rt: 0001, rd: 0000, rs_content: 0000000000000000, rt_content: 0000000000000110, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 0, ALUsrc: 1, regDst: 0, memWrite: 1, regWrite: 0, memRead: 0, ALUresult: 0000000000000001, writeData: 0000000000
000001
```



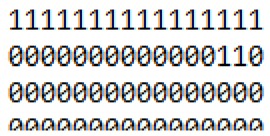
```
0000000000000110
```

rt register.



```
1111111111111111
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

First few contents of memory at the beginning.

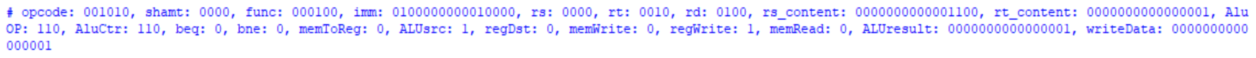


```
1111111111111111
0000000000000110
0000000000000000
0000000000000000
```

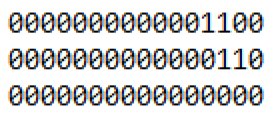
First few memory contents after running sw instruction.

slti instruction:

We are going to set rt to 1 if rs content (1100) < immediate (0100000000010000).



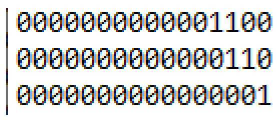
```
# opcode: 001010, shamt: 0000, func: 000100, imm: 0100000000010000, rs: 0000, rt: 0010, rd: 0100, rs_content: 0000000000001100, rt_content: 0000000000000001, Alu
OP: 110, AluCtr: 110, beq: 0, bne: 0, memToReg: 0, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000000001, writeData: 0000000000
000001
```



```
0000000000001100
0000000000000110
0000000000000000
```

At the beginning:

First row → rs content
Third row → rt content



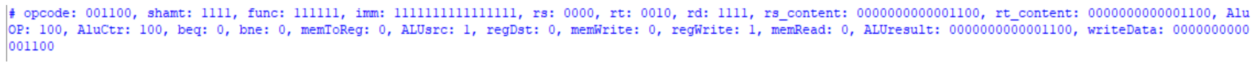
```
0000000000001100
0000000000000110
0000000000000001
```

At the end:

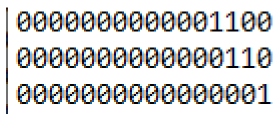
First row → rs content
Third row → rt content

andi instruction:

We are going to set rt to $rs (1100) & immediate (1111111111111111).



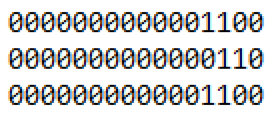
```
# opcode: 001100, shamt: 1111, func: 111111, imm: 1111111111111111, rs: 0000, rt: 0010, rd: 1111, rs_content: 0000000000001100, rt_content: 0000000000001100, Alu
OP: 100, AluCtr: 100, beq: 0, bne: 0, memToReg: 0, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000000000001100, writeData: 0000000000
001100
```



```
0000000000001100
0000000000000110
0000000000000001
```

At the beginning:

First row → rs content
Third row → rt content



```
0000000000001100
0000000000000110
0000000000001100
```

At the end:

First row → rs content
Third row → rt content

We get the value itself because we and it with all 1s.

ori instruction:

We are going to set rt to $rs (1100) | immediate (1110101111110000).

```
# opcode: 001101, shamt: 1011, func: 111100, imm: 1110101111110000, rs: 0000, rt: 0010, rd: 1110, rs_content: 0000000000001100, rt_content: 1110101111111100, Alu
OP: 101, AluCtr: 101, beq: 0, bne: 0, memToReg: 0, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 1110101111111100, writeData: 1110101111
111100
```

```
0000000000001100
0000000000000110
0000000000001100
```

At the beginning:

First row → rs content
Third row → rt content

```
0000000000001100
0000000000000110
1110101111111100
```

At the end:

First row → rs content
Third row → rt content

addi instruction:

We are going to set rt to $rs (1100) + immediate (1000000000000110).

```
# opcode: 001000, shamt: 0000, func: 000001, imm: 1000000000000110, rs: 0000, rt: 0010, rd: 1000, rs_content: 0000000000001100, rt_content: 1000000000010010, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 0, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 1000000000010010, writeData: 1000000000
010010
```

```
0000000000001100
0000000000000110
1110101111111100
```

At the beginning:

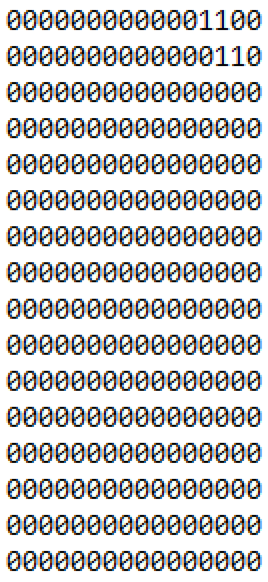
First row → rs content
Third row → rt content

```
0000000000001100
0000000000000110
1000000000010010
```

At the end:

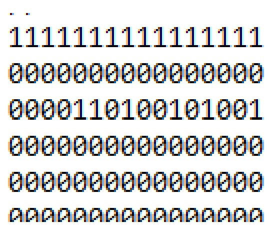
First row → rs content
Third row → rt content

Special Testbench – 1



```
0000000000001100
0000000000000110
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

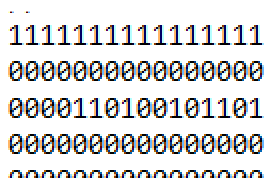
Register contents at the beginning of test.



```
1111111111111111
0000000000000000
0000110100101001
0000000000000000
0000000000000000
0000000000000000
```

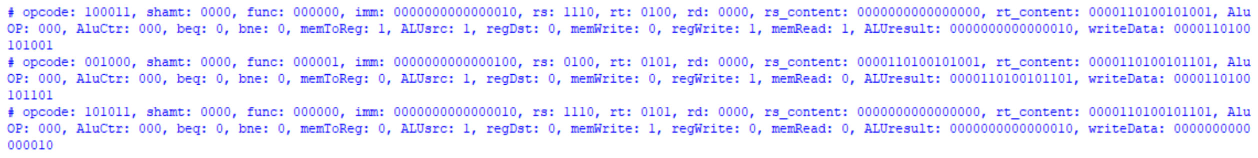
First few memory contents at the beginning of test.

I will load third register content to the 0100. register.
Then I will add 4 to it.
Then I will store it again to its first place.



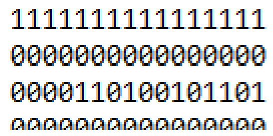
```
1111111111111111
0000000000000000
0000110100101101
0000000000000000
0000000000000000
```

First few memory contents at the end.



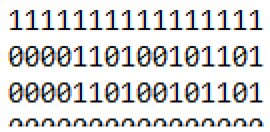
```
# opcode: 100011, shamt: 0000, func: 000000, imm: 0000000000000010, rs: 1110, rt: 0100, rd: 0000, rs_content: 0000000000000000, rt_content: 0000110100101001, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 1, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 1, ALUresult: 0000000000000010, writeData: 0000110100
101001
# opcode: 001000, shamt: 0000, func: 000001, imm: 0000000000000100, rs: 0100, rt: 0101, rd: 0000, rs_content: 0000110100101001, rt_content: 0000110100101101, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 0, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000110100101101, writeData: 0000110100
101101
# opcode: 101011, shamt: 0000, func: 000000, imm: 0000000000000010, rs: 1110, rt: 0101, rd: 0000, rs_content: 0000000000000000, rt_content: 0000110100101101, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 0, ALUsrc: 1, regDst: 0, memWrite: 1, regWrite: 0, memRead: 0, ALUresult: 0000000000000010, writeData: 0000000000
000010
```

Special Testbench – 2



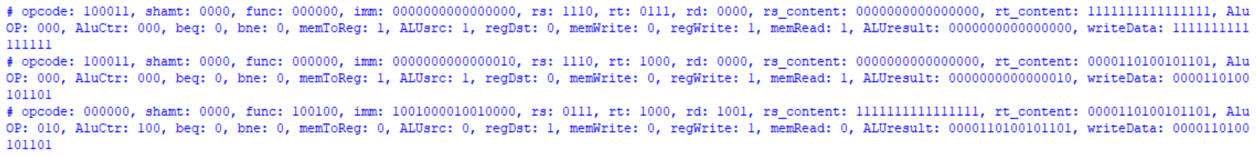
```
1111111111111111
0000000000000000
0000110100101101
0000000000000000
```

First few memory contents at the beginning. I will take first and third words, put them to registers, then add them and put the result in register.



```
1111111111111111
0000110100101101
0000110100101101
0000000000000000
```

Register contents that I put first and third memory addresses and the “and” value of these two.



```
# opcode: 100011, shamt: 0000, func: 000000, imm: 0000000000000000, rs: 1110, rt: 0111, rd: 0000, rs_content: 0000000000000000, rt_content: 1111111111111111, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 1, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 1, ALUresult: 0000000000000000, writeData: 1111111111
111111
# opcode: 100011, shamt: 0000, func: 000000, imm: 0000000000000010, rs: 1110, rt: 1000, rd: 0000, rs_content: 0000000000000000, rt_content: 0000110100101101, Alu
OP: 000, AluCtr: 000, beq: 0, bne: 0, memToReg: 1, ALUsrc: 1, regDst: 0, memWrite: 0, regWrite: 1, memRead: 1, ALUresult: 0000000000000010, writeData: 0000110100
101101
# opcode: 000000, shamt: 0000, func: 100100, imm: 1001000010010000, rs: 0111, rt: 1000, rd: 1001, rs_content: 1111111111111111, rt_content: 0000110100101101, Alu
OP: 010, AluCtr: 100, beq: 0, bne: 0, memToReg: 0, ALUsrc: 0, regDst: 1, memWrite: 0, regWrite: 1, memRead: 0, ALUresult: 0000110100101101, writeData: 0000110100
101101
```