Gebze Technical University

Computer Engineering

CSE 331- 2019 Fall

HOMEWORK 2 REPORT

MUHAMMED OKUMUŞ

151044017

Course Assistant: Fatma Nur Esirci

Modules

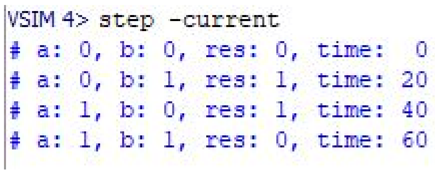
XOR Gate

This is not the xor gate provided by verilog, I implemented xor_gate module using AND, OR and NOT gates. It’s implemented according to this logical expression:

$Q = (A.B') + (A'.B)$

Gate cost: 2 AND + 1 OR + 2 NOT = 4 cost

Test bench results:



```
VSIM 4> step -current
# a: 0, b: 0, res: 0, time:  0
# a: 0, b: 1, res: 1, time: 20
# a: 1, b: 0, res: 1, time: 40
# a: 1, b: 1, res: 0, time: 60
```

MUX 4:1

Logical expression:

$x = A.S_1'.S_0'$

$y = B.S_1'.S_0$

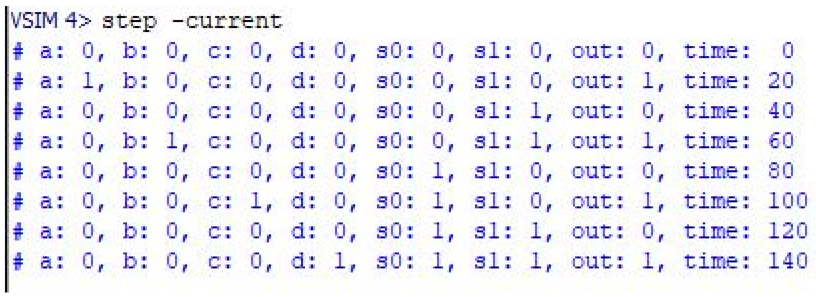
$z = C.S_1.S_0'$

$w = D.S_1.S_0$

$Q = x + y + z + w$

Gate cost: 4 AND(3 input) + 1 OR(4 input) + 2 NOT = 6 cost

Test bench results:



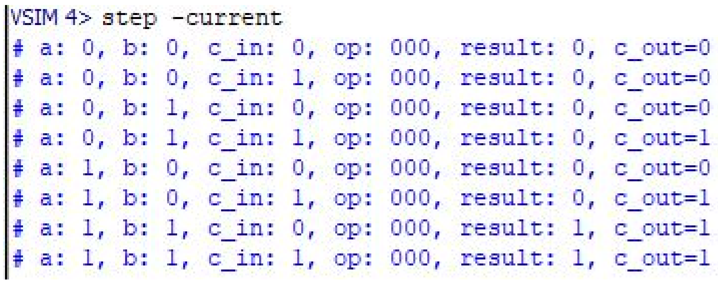
```
VSIM 4> step -current
# a: 0, b: 0, c: 0, d: 0, s0: 0, s1: 0, out: 0, time:   0
# a: 1, b: 0, c: 0, d: 0, s0: 0, s1: 0, out: 1, time:  20
# a: 0, b: 0, c: 0, d: 0, s0: 0, s1: 1, out: 0, time:  40
# a: 0, b: 1, c: 0, d: 0, s0: 0, s1: 1, out: 1, time:  60
# a: 0, b: 0, c: 0, d: 0, s0: 1, s1: 0, out: 0, time:  80
# a: 0, b: 0, c: 1, d: 0, s0: 1, s1: 0, out: 1, time: 100
# a: 0, b: 0, c: 0, d: 0, s0: 1, s1: 1, out: 0, time: 120
# a: 0, b: 0, c: 0, d: 1, s0: 1, s1: 1, out: 1, time: 140
```

1-Bit ALU

Design is exactly the same as the one provided in the assignment sheet.

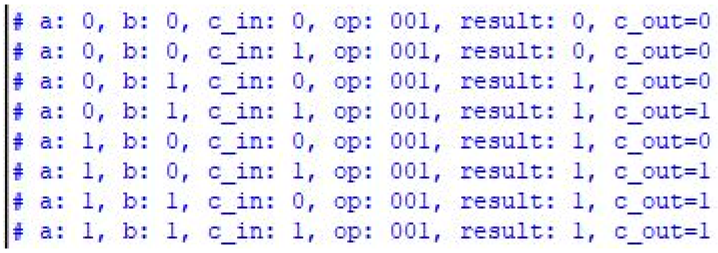
Gate cost: 1 XOR + 4 AND + 3 OR + 2 NOT + 1 MUX = 17 cost

Test bench results for **AND** operation:



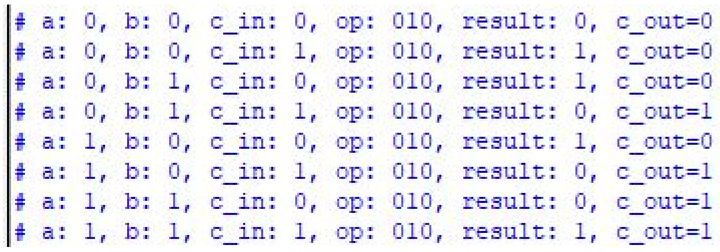
```
VSIM 4> step -current
# a: 0, b: 0, c_in: 0, op: 000, result: 0, c_out=0
# a: 0, b: 0, c_in: 1, op: 000, result: 0, c_out=0
# a: 0, b: 1, c_in: 0, op: 000, result: 0, c_out=0
# a: 0, b: 1, c_in: 1, op: 000, result: 0, c_out=1
# a: 1, b: 0, c_in: 0, op: 000, result: 0, c_out=0
# a: 1, b: 0, c_in: 1, op: 000, result: 0, c_out=1
# a: 1, b: 1, c_in: 0, op: 000, result: 1, c_out=1
# a: 1, b: 1, c_in: 1, op: 000, result: 1, c_out=1
```

Test bench results for **OR** operation:



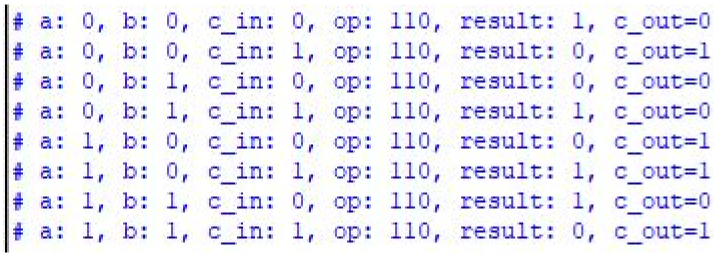
```
# a: 0, b: 0, c_in: 0, op: 001, result: 0, c_out=0
# a: 0, b: 0, c_in: 1, op: 001, result: 0, c_out=0
# a: 0, b: 1, c_in: 0, op: 001, result: 1, c_out=0
# a: 0, b: 1, c_in: 1, op: 001, result: 1, c_out=1
# a: 1, b: 0, c_in: 0, op: 001, result: 1, c_out=0
# a: 1, b: 0, c_in: 1, op: 001, result: 1, c_out=1
# a: 1, b: 1, c_in: 0, op: 001, result: 1, c_out=1
# a: 1, b: 1, c_in: 1, op: 001, result: 1, c_out=1
```

Test bench results for **ADD** operation:



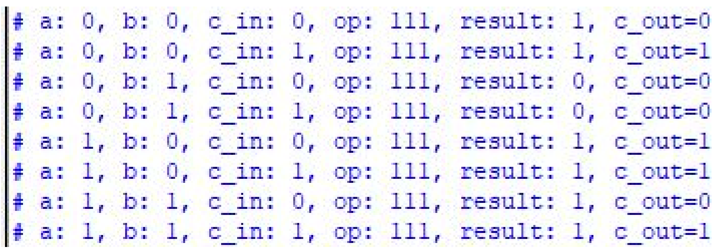
```
# a: 0, b: 0, c_in: 0, op: 010, result: 0, c_out=0
# a: 0, b: 0, c_in: 1, op: 010, result: 1, c_out=0
# a: 0, b: 1, c_in: 0, op: 010, result: 1, c_out=0
# a: 0, b: 1, c_in: 1, op: 010, result: 0, c_out=1
# a: 1, b: 0, c_in: 0, op: 010, result: 1, c_out=0
# a: 1, b: 0, c_in: 1, op: 010, result: 0, c_out=1
# a: 1, b: 1, c_in: 0, op: 010, result: 0, c_out=1
# a: 1, b: 1, c_in: 1, op: 010, result: 1, c_out=1
```

Test bench results for **SUB** operation(not working properly):



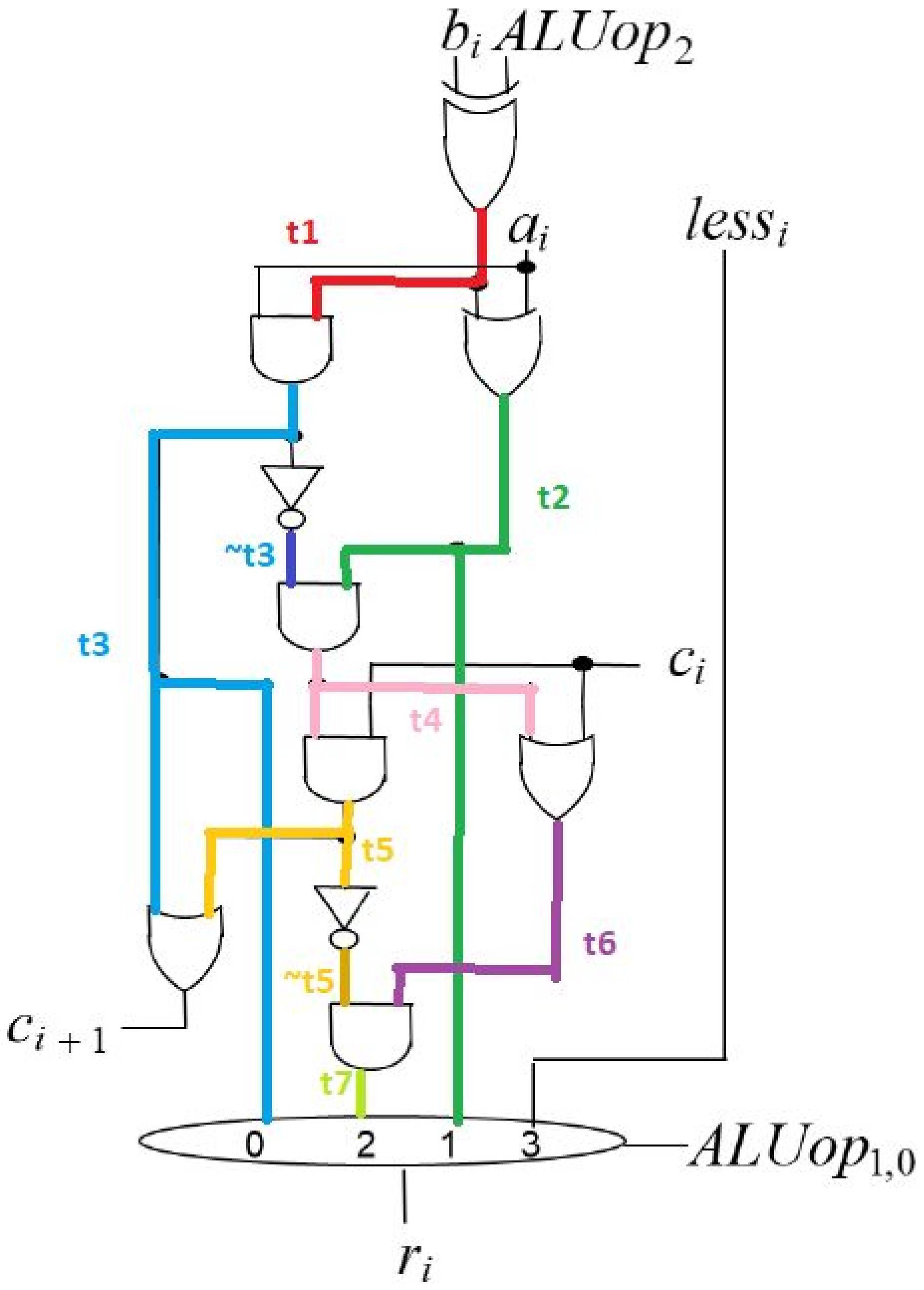
```
# a: 0, b: 0, c_in: 0, op: 110, result: 1, c_out=0
# a: 0, b: 0, c_in: 1, op: 110, result: 0, c_out=1
# a: 0, b: 1, c_in: 0, op: 110, result: 0, c_out=0
# a: 0, b: 1, c_in: 1, op: 110, result: 1, c_out=0
# a: 1, b: 0, c_in: 0, op: 110, result: 0, c_out=1
# a: 1, b: 0, c_in: 1, op: 110, result: 1, c_out=1
# a: 1, b: 1, c_in: 0, op: 110, result: 1, c_out=0
# a: 1, b: 1, c_in: 1, op: 110, result: 0, c_out=1
```

Test bench results for **SLT** operation(not working properly):



```
# a: 0, b: 0, c_in: 0, op: 111, result: 1, c_out=0
# a: 0, b: 0, c_in: 1, op: 111, result: 1, c_out=1
# a: 0, b: 1, c_in: 0, op: 111, result: 0, c_out=0
# a: 0, b: 1, c_in: 1, op: 111, result: 0, c_out=0
# a: 1, b: 0, c_in: 0, op: 111, result: 1, c_out=1
# a: 1, b: 0, c_in: 1, op: 111, result: 1, c_out=1
# a: 1, b: 1, c_in: 0, op: 111, result: 1, c_out=0
# a: 1, b: 1, c_in: 1, op: 111, result: 1, c_out=1
```

Wiring

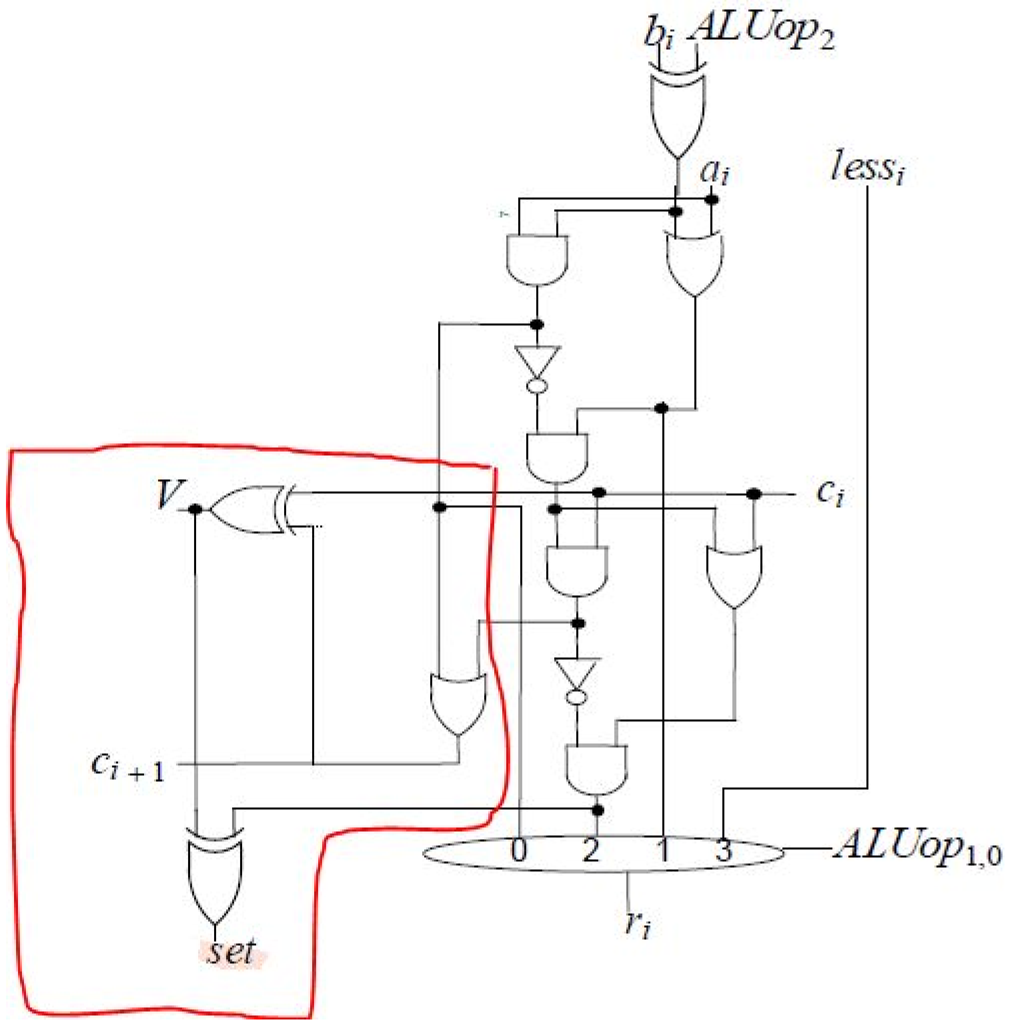


As implemented in ***alu.v***

ALU_MSB(Most significant bit)

The design of this module is exactly the same as 1-bit ALU but it has extra **set** and **V** outputs

Gate cost: 3 XOR + 4 AND + 3 OR + 2 NOT + 1 MUX = 26 cost

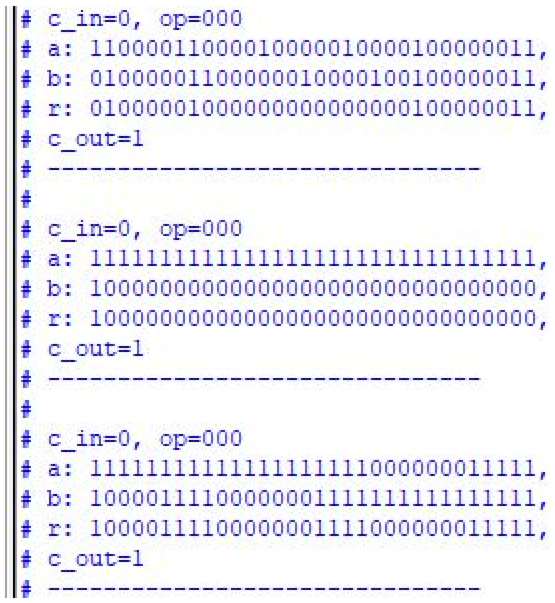


32-Bit ALU

32-Bit ALU consists of 31 1-Bit ALUs and 1 ALU_MSB.

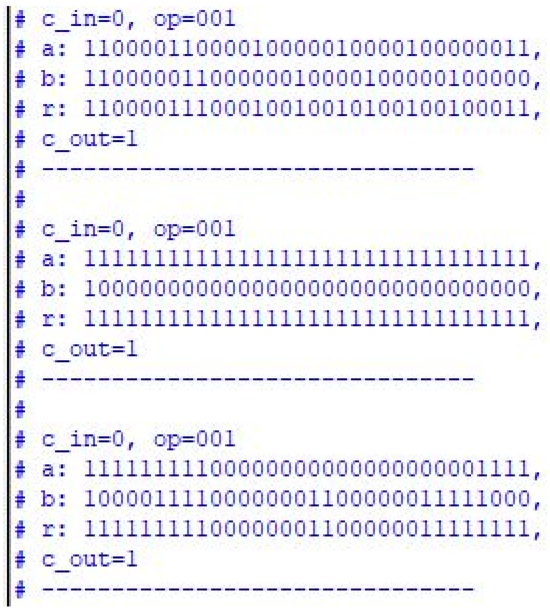
Gate cost: 31 ALU_1bit + 1 ALU_msb = 31x11 + 20 = 361 cost

Test bench results for **AND** operation:



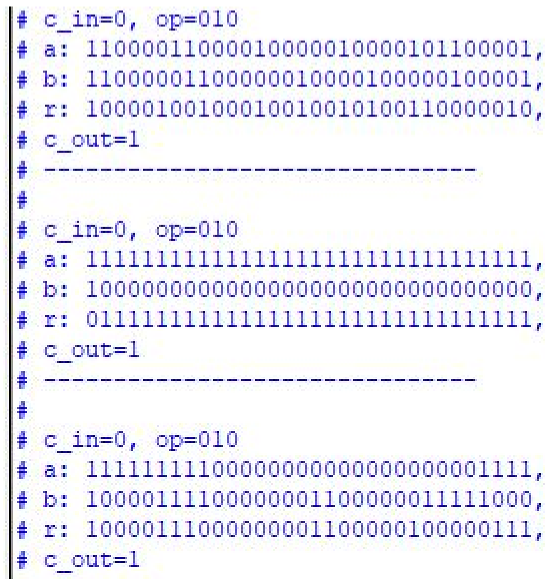
```
# c_in=0, op=000
# a: 11000011000010000001000010000011,
# b: 01000001100000010000100100000011,
# r: 01000001000000000000000100000011,
# c_out=1
# -------------------------------
#
# c_in=0, op=000
# a: 11111111111111111111111111111111,
# b: 10000000000000000000000000000000,
# r: 10000000000000000000000000000000,
# c_out=1
# -------------------------------
#
# c_in=0, op=000
# a: 11111111111111111111000000011111,
# b: 10000111100000001111111111111111,
# r: 10000111100000001111000000011111,
# c_out=1
# -------------------------------
```

Test bench results for **OR** operation:



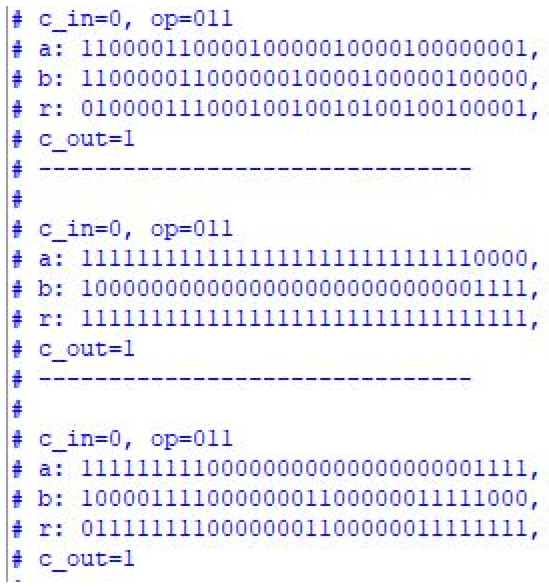
```
# c_in=0, op=001
# a: 11000011000010000001000010000011,
# b: 11000001100000001000010000100000,
# r: 11000011100010010010100100100011,
# c_out=1
# -------------------------------
#
# c_in=0, op=001
# a: 11111111111111111111111111111111,
# b: 10000000000000000000000000000000,
# r: 11111111111111111111111111111111,
# c_out=1
# -------------------------------
#
# c_in=0, op=001
# a: 11111111100000000000000000001111,
# b: 10000111100000001100000001111000,
# r: 11111111100000001100000001111111,
# c_out=1
# -------------------------------
```

Test bench results for **ADD** operation:



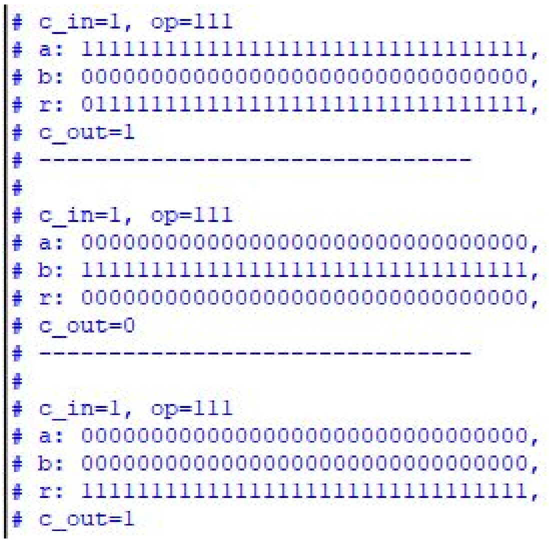
```
# c_in=0, op=010
# a: 11000011000010000010000101100001,
# b: 11000001100000010000100000100001,
# r: 10000100100010010010100110000010,
# c_out=1
# -------------------------------
#
# c_in=0, op=010
# a: 11111111111111111111111111111111,
# b: 10000000000000000000000000000000,
# r: 01111111111111111111111111111111,
# c_out=1
# -------------------------------
#
# c_in=0, op=010
# a: 11111111000000000000000000001111,
# b: 10000111100000001100000011111000,
# r: 10000111000000001100000100000111,
# c_out=1
```

Test bench results for **SUB** operation(not working properly):



```
# c_in=0, op=011
# a: 11000011000010000010000100000001,
# b: 11000001100000010000100000100000,
# r: 01000011100010010010100100100001,
# c_out=1
# -------------------------------
#
# c_in=0, op=011
# a: 11111111111111111111111111110000,
# b: 10000000000000000000000000001111,
# r: 11111111111111111111111111111111,
# c_out=1
# -------------------------------
#
# c_in=0, op=011
# a: 11111111000000000000000000001111,
# b: 10000111100000001100000011111000,
# r: 01111111100000001100000011111111,
# c_out=1
```

Test bench results for **SLT** operation(not working properly):



```
# c_in=1, op=111
# a: 11111111111111111111111111111111,
# b: 00000000000000000000000000000000,
# r: 01111111111111111111111111111111,
# c_out=1
# -------------------------------
#
# c_in=1, op=111
# a: 00000000000000000000000000000000,
# b: 11111111111111111111111111111111,
# r: 00000000000000000000000000000000,
# c_out=0
# -------------------------------
#
# c_in=1, op=111
# a: 00000000000000000000000000000000,
# b: 00000000000000000000000000000000,
# r: 11111111111111111111111111111111,
# c_out=1
```

Conclusion

Even though I implemented the exact same circuit provided for the 1-Bit ALU and alu_msb(provided in lecture notes), the subtraction and set less than operations didn’t work correctly, I managed to get correct results with a stand alone subtraction/slt modules but we were required to implement the exact same design so I scrapped those modules. In this design, there are 34 XOR gates, 256 AND gates 128 OR gates, 128 NOT gates are used. So total of 546 gates are used.