Booth Multiplier

Syomantak Chaudhuri
170070004

April 2019

1 Algorithm

This algorithm focuses on trying to multiply two bits of multiplier at a time rather than the conventional 1 bit at a time approach. Consider multiplication AxB. If we encounter 00 in multiplier, then the result is 0. If we encounter 01 then results is A. If we encounter 10 then results is 2A. If we encounter 11 then results is 3A.0, A and 2A are easily obtained but for 3A we use a smart approach. Since, 3A = 4A - A, subtract A at this stage and tell the next stage to add 4A. Since the next stage is already 2 position ahead of this, we only need to add an extra A. To make the logic simpler, we write 2A as 4A - 2A and subtract 2A whenever we encounter 10 in multiplier.
So what I described above did not consider the 'carry' of A from previous sub-part of the multiplier. Taking that into account, if we encounter say 10 in B and previous bit is 1 then we know we need to add A at this position (previous bit 1 means either 10 or 11 previously - in both cases we add 4A which is equivalent to adding A at this shifted position). 10 also means 4A - 2A, therefore, at this position I need to do A-2A=-A and the next part would take care of the 4A. In this fashion, the following table is constructed:

Current Bits (X)	**Previous Bit (Y)**	**Increment due to X**	**Increment due to Y**	**Net change**
00	0	0	0	0
00	1	0	A	A
01	0	A	0	A
01	1	A	A	2A
10	0	-2A	0	-2A
10	1	-2A	A	-A
11	0	-A	0	-A
11	1	-A	A	0

Table 1: Increment rule

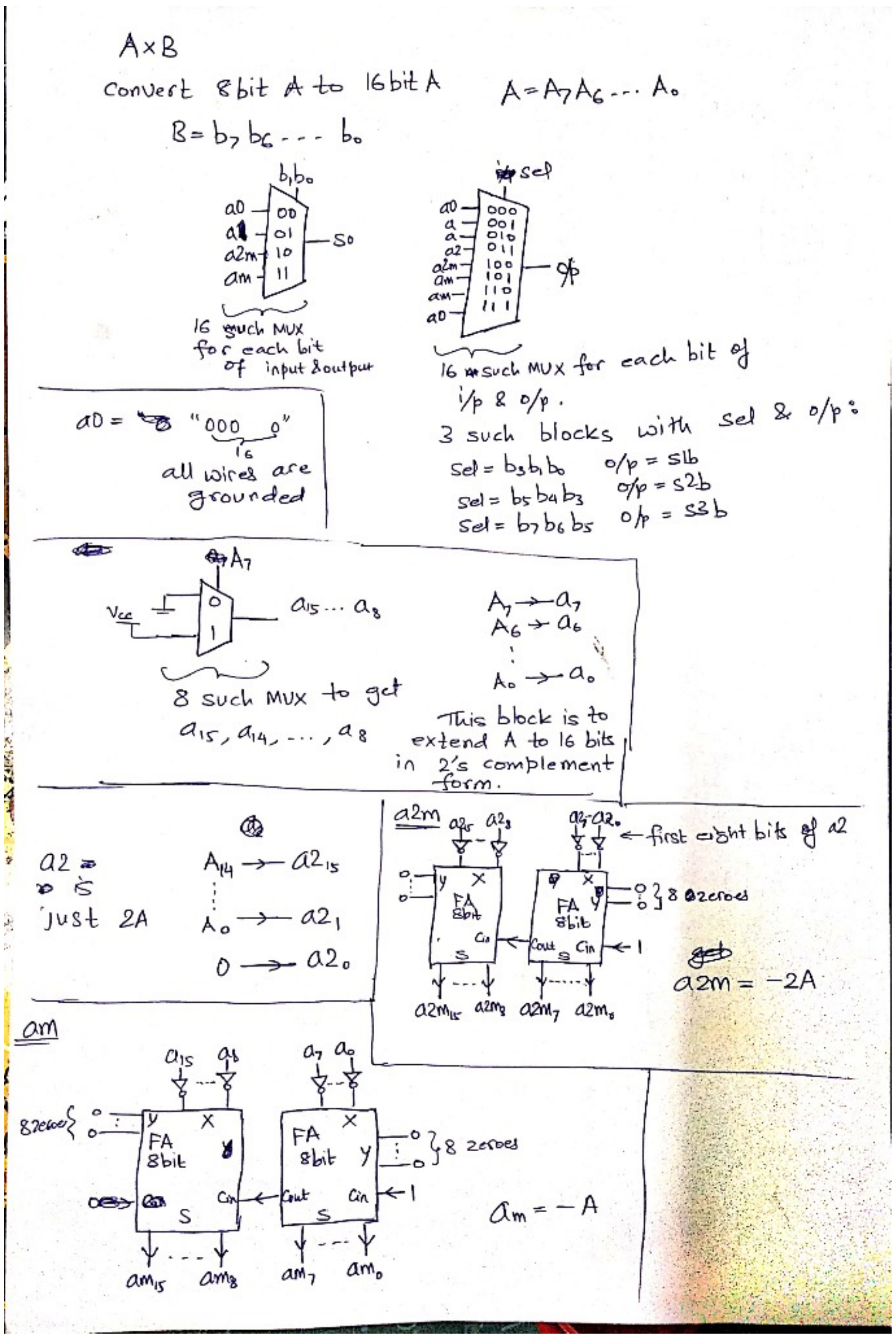
2 Components

- 8 MUX-2 input
- 32 NOT gates
- 10 8bit Full Adders
- 16 MUX-4 input
- 48 MUX-8 input

Each MUX-2 input has 2 two input AND gate, 1 two input OR gate and 1 NOT gate. Each MUX-4 input has 4 three input AND gate, 1 four input OR gate and 4 NOT gate. Each MUX-8 input has 24 two input AND gate, 7 two input OR gate and 12 NOT gate.

There were several cases where a bus-input MUX (8 inputs of 16 wires, each wire of an input having same hardware to get the output) would have saved the number of MUX.

3 Hardware



A×B

Convert 8bit A to 16bit A   A = A7A6 … A0

B = b7b6 … b0

16 such MUX for each bit of input & output

16 such MUX for each bit of i/p & o/p.

3 such blocks with sel & o/p:

sel = b3b1b0   o/p = s1b

sel = b5b4b3   o/p = s2b

sel = b7b6b5   o/p = s3b

a0 = "000…0" (16), all wires are grounded

8 such MUX to get a15, a14, …, a8

A7 → a7, A6 → a6, …, A0 → a0

This block is to extend A to 16 bits in 2's complement form.

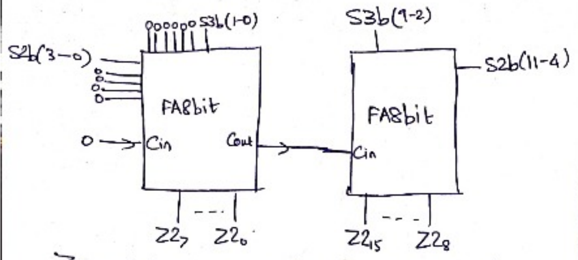
a2 is just 2A

A14 → a2 15, …, A0 → a2 1, 0 → a2 0

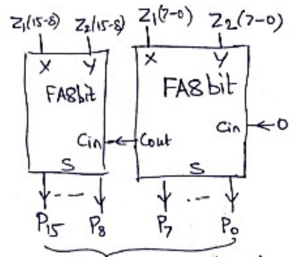
a2m = −2A

am = −A

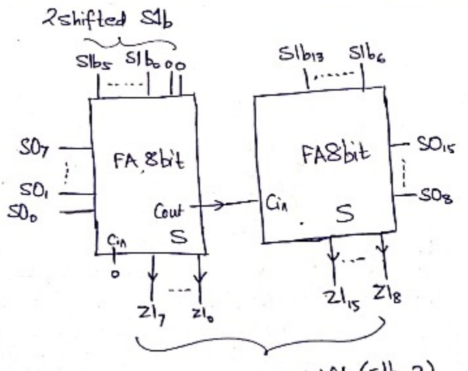
3



$Z_2 = leftshift(s2b, 4) + leftshift(s3b, 6)$



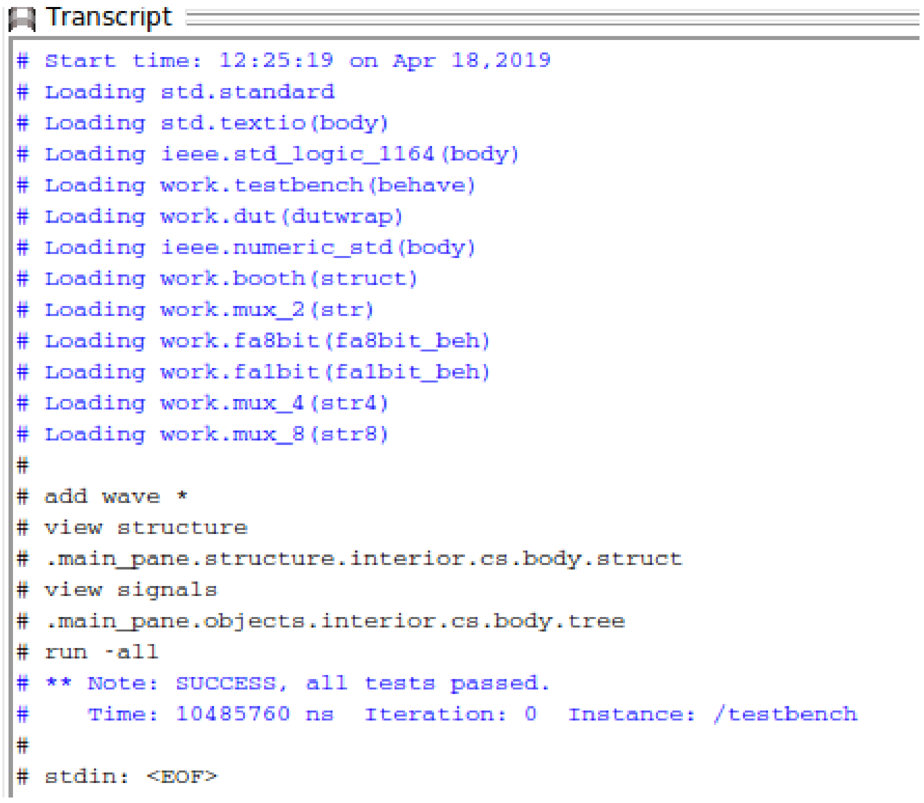
$P = Z_1 + Z_2$ final output



$Z_1 = S0 + leftshift(s1b, 2)$

4 Results

I generated a tracefile that covers all possible inputs and did RTL simulation. It passed all cases and output is shown. It indeed multiplies inputs very fast as seen in Gate level simulation.



Transcript

```
# Start time: 12:25:19 on Apr 18,2019
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.testbench(behave)
# Loading work.dut(dutwrap)
# Loading ieee.numeric_std(body)
# Loading work.booth(struct)
# Loading work.mux_2(str)
# Loading work.fa8bit(fa8bit_beh)
# Loading work.fa1bit(fa1bit_beh)
# Loading work.mux_4(str4)
# Loading work.mux_8(str8)
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ** Note: SUCCESS, all tests passed.
#    Time: 10485760 ns  Iteration: 0  Instance: /testbench
#
# stdin: <EOF>
```

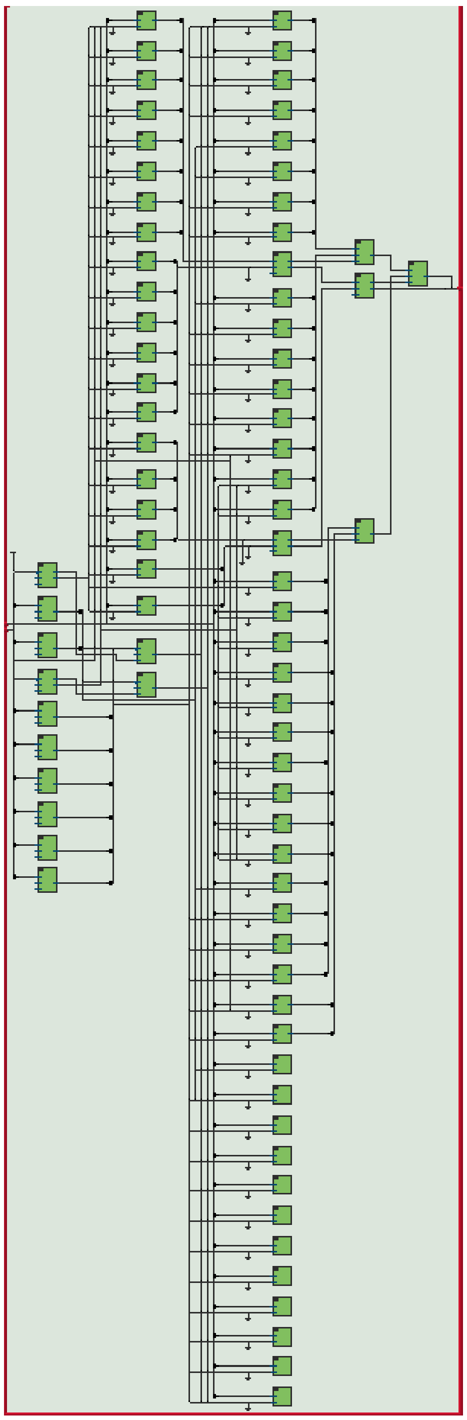


Figure 1: Netlist View