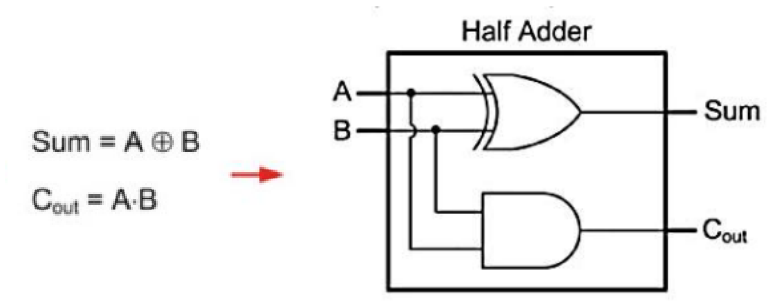
Professor Bhavnagarwala　　　　　　　　　　　　　　　　　　　　　　Anik Barua

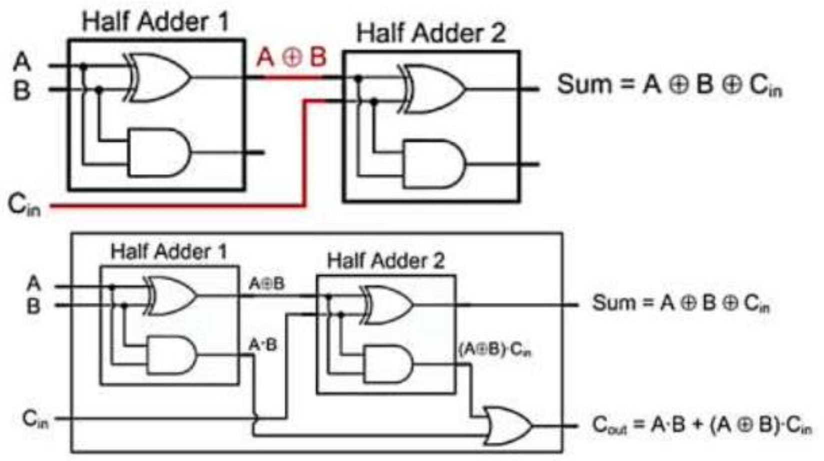
CS 2204 - Section C　　　　　　　　　　　　　　　　　　　　　　　　11-19-2021

Lab 9 [Extra Credit]: 4b adder using a simple half-adder

Today's lab is about developing Verilog modules for a 4b adder using a simple half-adder cell as building block. First we will write a Verilog module for a half adder and verify its functionality with test bench simulations. Then using that half-adder module, we will write the full-adder cell and verify its functionality with test bench simulations. Finally using that full adder module, we write the Verilog module for a 4b adder and verify its functionality. Before we write the modules, we need to understand how half adder and full adder works. When creating an adder, we have to design the incremental sub-systems in way that it can be re-used like using half adder to create full adder. This technique reduces the design effort, cost and minimizes the difficulty of troubleshooting. Half Adder computes the sum, S and carry out, C from its two input A and B. It is different from a Full adder because it doesn't take the carry-in bit into account. So for half adder the possible inputs are - 0+0 = 0 (0 is sum), 0+1 = 1, 1+0 = 1, and 1+1 = 10 (1 is Carry). As we can see, it only takes 2 inputs A, B so 2^2 = 4 possible combinations.

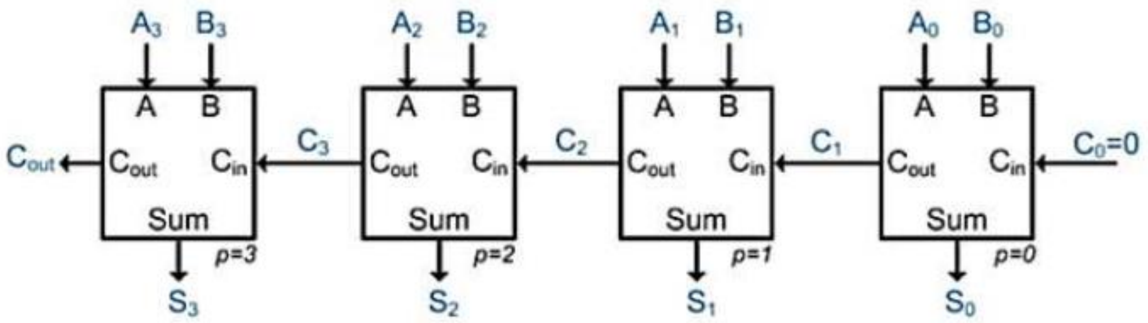


Next is Full Adder, it is a circuit that produces a sum and carry out like half adder but also considers the carry-in as input. So it has three inputs in total - A, B and Cin, so 2^3 = 8 possible combinations. We can create a full adder by combining two half adder, where the first adder will take the A & B input, and the second half adder will take the sum from first HA and Cin as the input. So we have a sum and Cout (added two Cout from each HA) in the end for output.


Half Adder 1
Half Adder 2
A
B
C_{in}
$A \oplus B$
$\text{Sum} = A \oplus B \oplus C_{in}$
Half Adder 1
Half Adder 2
A
B
C_{in}
$A \oplus B$
$A \cdot B$
$(A \oplus B) \cdot C_{in}$
$\text{Sum} = A \oplus B \oplus C_{in}$
$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$

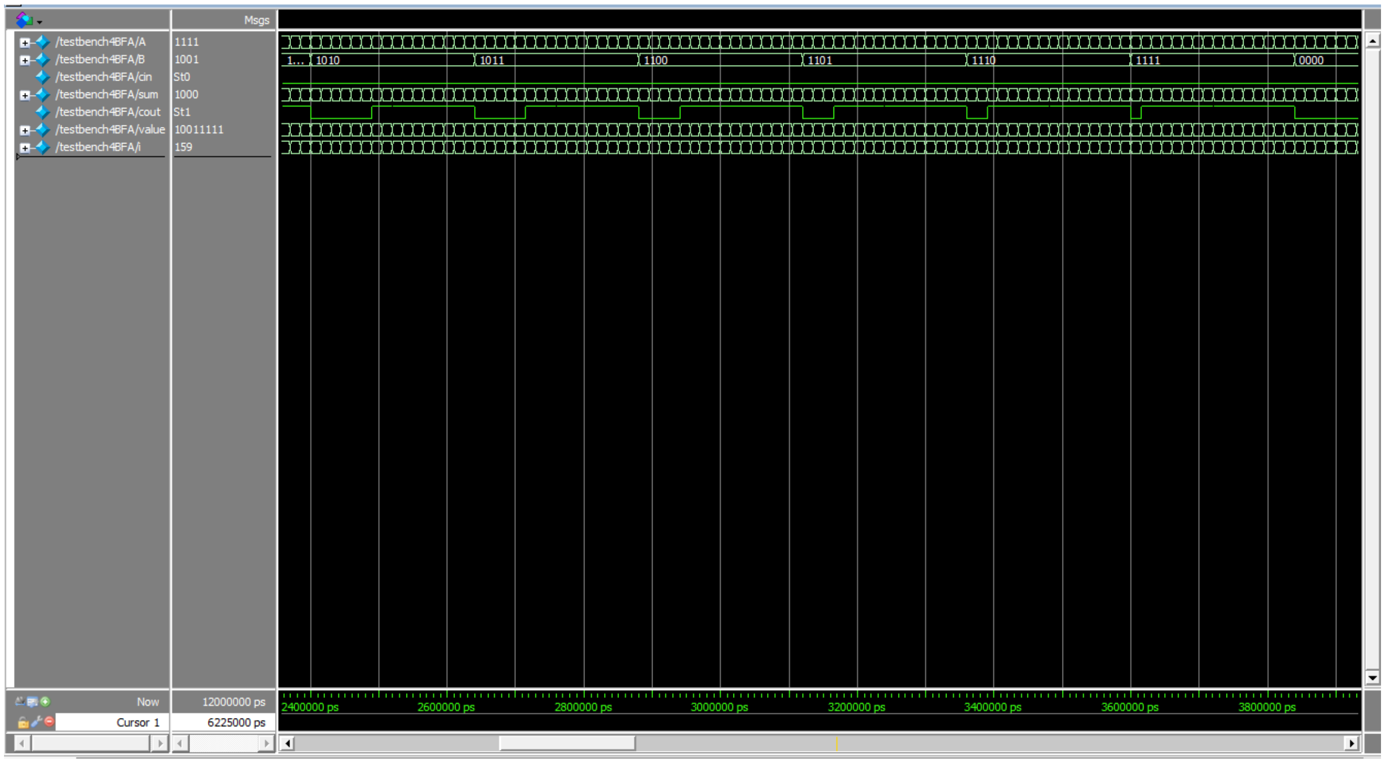
C_{in}	A	B	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Now we can use the full adder to create multibit adder, in this case we will create a 4b adder that will uses 4 full adder. In a full adder, we have a sum and a carry out for each bit position. So in multi-bit adder, the carry out of each full adder is used as the carry in for the next significant bit. That’s why each subsequent full adder needs to wait for the carry to be produced by the previous stage. So, the carry is said to ‘ripple through’ the circuit and 4b adder is using this Ripple Carry Adder architecture.



Now we can write the modules for each - half adder, full adder and a 4b adder. For half adder, we have inputs A, B and output sum and Cout. From the circuit we can see it has a XOR gate for sum and an AND gate for Cout with A, B input. So we can write it as - assign sum = A ^ B (^ is XOR) and assign Cout = A & B. That's our module for HA and for testbench we have to try out all possible combinations. In testbench we will have reg A, B and wire as SUM and Cout. We need a reg 2 bit value that using which we can assign values to A and B. Next, we will create a half adder dut and map all the inputs and outputs. After that we will declare a integer i (32 bit). Now using for loop, we have to try all 4 combinations and we can do it by assigning {value} = i which will take the two MSB (since value is 2 bit) and make A <= value[0] and B <= value[1]. This will try out all combinations and we have #15 nanoseconds delay for each cycle. After verifying this using simulation waveforms, we can start with Full adder module. For Full adder, wire S1, C1, C2 to connect the two half adder outputs inside the module. Then we will create two half adder, where we will have output S1 and C1 from the first HA, and the second HA will take S1 as input and will output sum and C2. Next we have to add Cout's from each HA, and we can do that by assigning assign Cout = C1 + C2. Then we will create a testbench for this FA similar to HA but we have to make some changes considering the circuit and new Cin as input. For Cin, we will have a reg Cin and the reg value will now be 3 bit because we have 3 inputs. Next, same as before we will create a half adder and for the loop, now i < 8 because we have 2^3 = 8 total combinations. Inside the for loop we have to assign the value for Cin <= value[2] along with A and B. Everything will stay the same and this concludes the testbench for Full adder. After running the simulation and verifying it, we can start with 4b adder. For the 4b module, we will have input A (4 bits), B(4 Bits), Cin and for output we will have sum (4 bit) and a Cout.

Then based on the circuit, we need 3 wires - C1, C2, C3 that will be connected to each full adder. Next we will create 4 full adder and assign the input and output sequence based on the Fig. 3 circuit. For the testbench, we will have 4 bit reg A, 4 bit reg B, Cin assigned to 0 initially, wire 4 bit sum, wire cout and reg of 8 bit value. We need 8 bits because each full adder takes 2 bits so 4*2=8 bits in total. Then we will create a fourbitadder and for the loop, now i < 256 because we have 2^8 = 256 total combinations. Inside the for loop we will assign the A value to first 4 MSB and B to next 4 MSB. This will try out all possible combinations for the testbench and if we run the simulation we get the following waveforms -

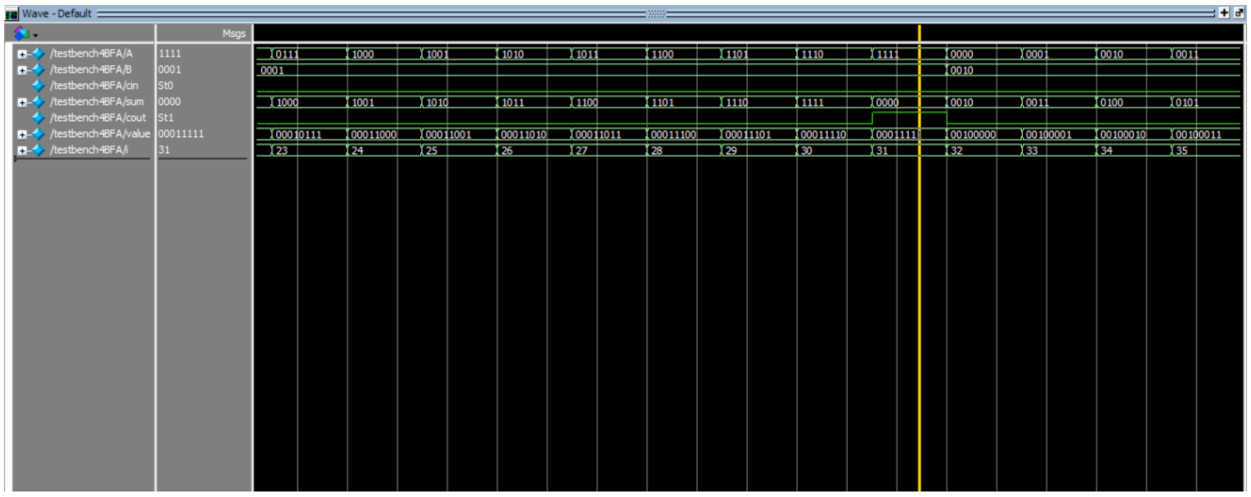


```
                         Msgs
/testbench4BFA/A         1111
/testbench4BFA/B         1001
/testbench4BFA/cin       St0
/testbench4BFA/sum       1000
/testbench4BFA/cout      St1
/testbench4BFA/value     10011111
/testbench4BFA/i         159

Now                      12000000 ps
Cursor 1                 6225000 ps
```

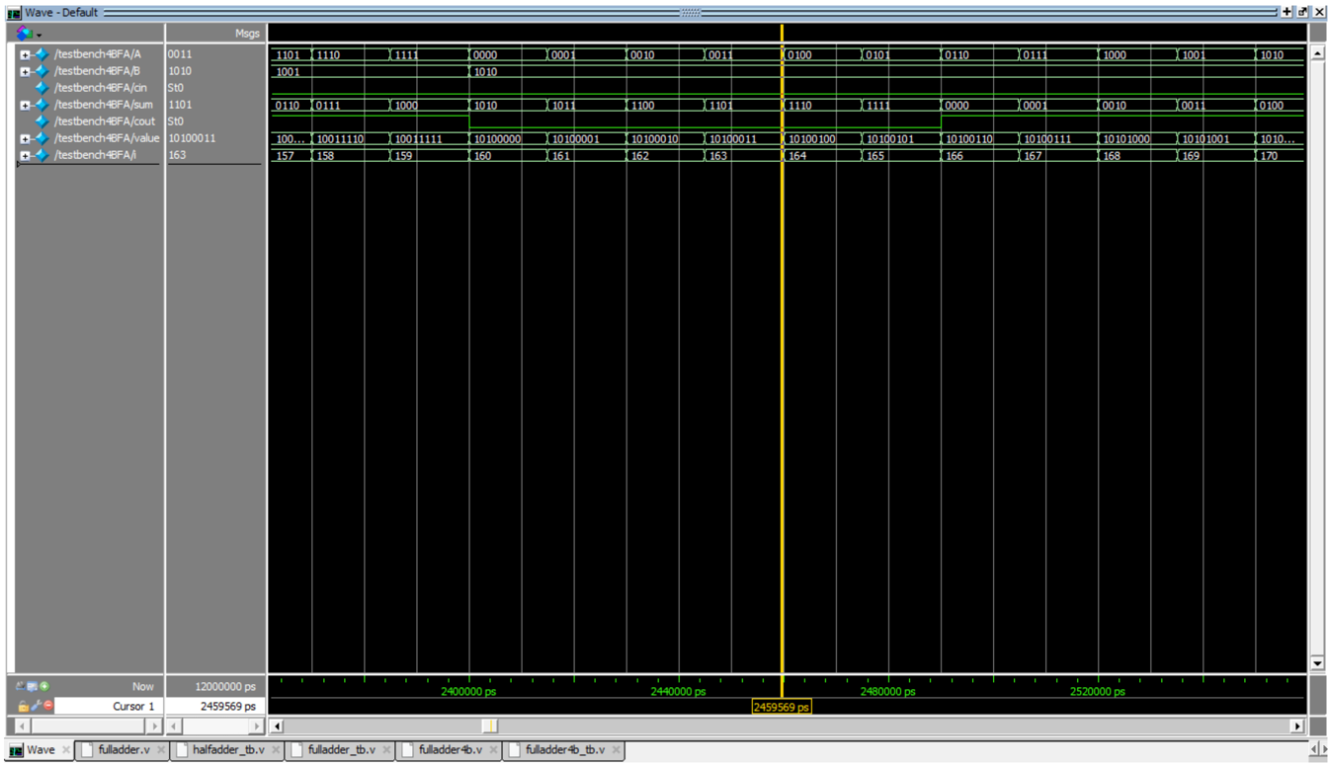
Waveform Verification -

To verify 4b adder works, we can try adding the values from the waveforms and see if we get the same sum and Cout. In the following waveform:



```
Wave - Default
                          Msgs
/testbench4BFA/A          1111
/testbench4BFA/B          0001
/testbench4BFA/cin        St0
/testbench4BFA/sum        0000
/testbench4BFA/cout       St1
/testbench4BFA/value      00011111
/testbench4BFA/i          31
```

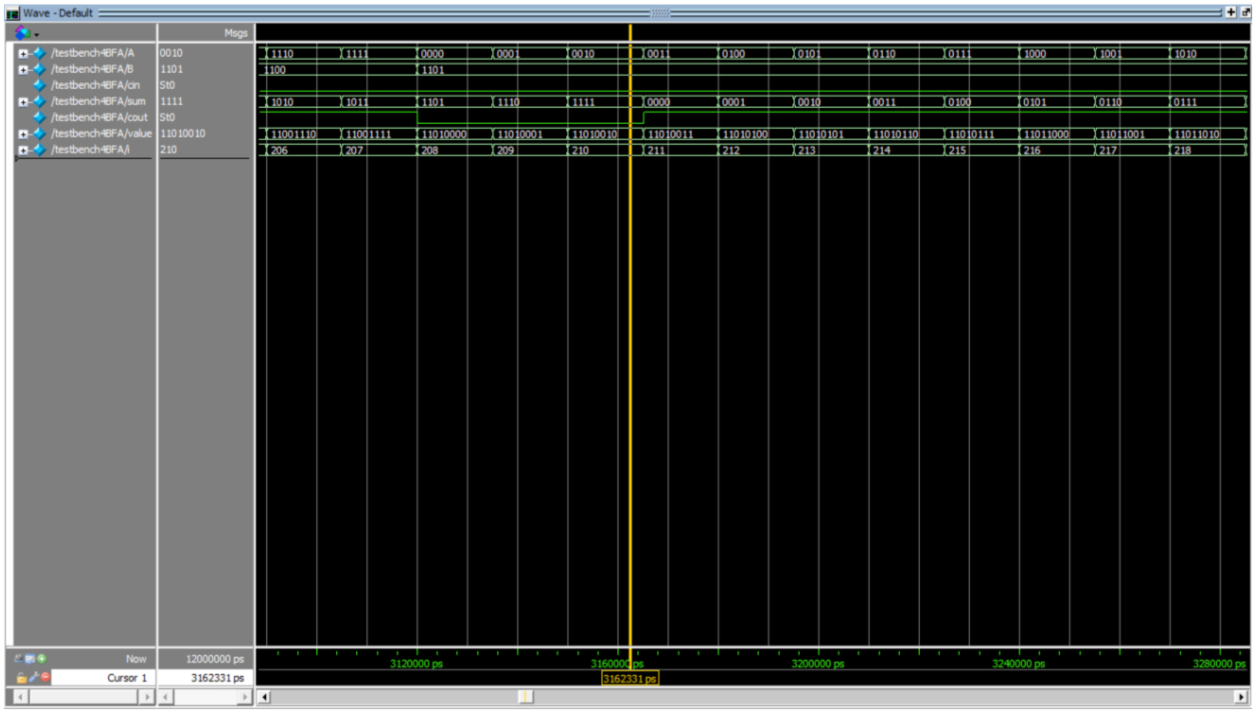
For the yellow line, we have A = 1111 and B = 0001. If we add (A + B) we should get sum = 0000 and Cout as 1. We can see we have the exact value for sum = 0000 and Cout goes to 1 in the waveform. Another one we can verify from the following waveform, where A = 0011 and B = 1010. If we add A + B, we get 1101 for sum and Cout as 0. From the waveform, we can see the same value for sum and cout.



```
Wave - Default
                          Msgs
/testbench4BFA/A          0011
/testbench4BFA/B          1010
/testbench4BFA/cin        St0
/testbench4BFA/sum        1101
/testbench4BFA/cout       St0
/testbench4BFA/value      10100011
/testbench4BFA/i          163

Now       12000000 ps
Cursor 1  2459569 ps
```

For the next waveform, we have A = 0010 and B = 1101 (yellow line). If we add A + B we get sum = 1111 and cout as 0. We get the same values from the waveforms for sum = 1111 and cout = 0. Trying out all this addition examples verifies that 4b adder functionality works and the code for



```
/testbench4BFA/A      0010      1110 1111 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010
/testbench4BFA/B      1101      1100 1101
/testbench4BFA/cin    St0
/testbench4BFA/sum    1111      1010 1011 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110 0111
/testbench4BFA/cout   St0
/testbench4BFA/value  11010010  11001110 11001111 11010000 11010001 11010010 11010011 11010100 11010101 11010110 11010111 11011000 11011001 11011010
/testbench4BFA/i      210       206 207 208 209 210 211 212 213 214 215 216 217 218
Now       12000000 ps
Cursor 1  3162331 ps
```

each half-adder, full-adder, 4b adder including the testbenches are attached.