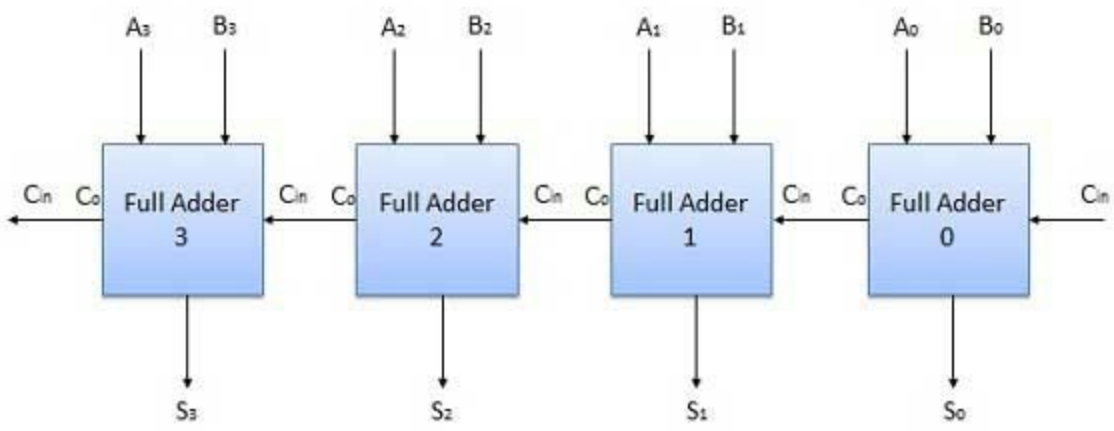
CS-505 Assignment 1 ALU Design

Ashish Jindal (AJ523)

Adder

The implementation is a 32 bit ripple carry adder created using 32 full adder units.



The implementation has been realised using verilog generate statement with the help of which we can generate 32 full adders at compile time using a simple for loop. The testbench corresponding to same is in module adder.test.v

Subtractor

The implementation of subtractor has been done using the adder module only. This is possible because : (A - B) = (A + (-B))
where “-B” represents the 2’s complement of B which can be calculated by first getting it’s 1’s compliment and then adding 1 (Carry input bit set to 1).
Testbench for same is in module subtractor.test.v

Shifter

I used a 32:1 multiplexer to implement the shifter. The multiplexer has been implemented in a generic way using “parameter” feature in verilog, giving it the flexibility in terms of width at compile time.

The idea behind shifter implementation is somewhat as explained in following example.
Following example illustrates its working for a 8-bit number.
Let input = 10011101
and we need to right shift it by 2 places, so the output should be 00100111
We use a 8:1 multiplexer to always select the 2nd bit of the number and then keep shifting it rightwards 1-bit at a time using following expression -
shifted result = {zeroes[i - 1 : 0], input[WIDTH - 1 : i]}
where “zeroes” is a 8-bit register containing all 0’s

Following will be the result on various steps of the loop -

when i = 1, output = 01001110
when i = 2, output = 00100111
when i = 3, output = 00010011
when i = 4, output = 00001001
when i = 5, output = 00000100
when i = 6, output = 00000010
when i = 7, output = 00000001

we directly select the bit at 2nd place (starting from 0) in the original input
when i = 0, output = 10011101

The output is the red bit marked in above example which is 00100111

The arithmetic shift counterpart of the above implementation differs only in the fact that we need to keep replicating the most significant bit while right shifting. Which can be implemented by keeping another register “Ones” which consists of all 1’s and using one of the two registers “Zeroes” and “Ones” depending on the what the most significant bit of the input is.

The same implementation has been used to implement left shift and its arithmetic counterpart. The only difference in the implementation is that in this case the output register is filled from the opposite direction and the arithmetic left shift would give same output at logical left shift.

The test cases for this module are in shifter.test.v
The shifter implementation works, but there seems to be some issue in my test bench code because of which the result from my shifter and the result calculated using in built operators appear at the output at different times (There is a offset of 1-2 cycles). The problem seems to be related to non-blocking update. but am not sure…

Multiplier
Following example illustrates my multiplier implementation -
Suppose we are multiplying 2 4-bit numbers
Multiplicand - 0101
Multiplier - 1101

```
      0101
      1101
------------
      0101       +
    0000         +
  0101           +
0101             = 0111100
```

The summary of algorithm is, always see the least significant bit of the multiplier, if it's 1 then we add the 1-bit left shifted multiplicand to the current result else we just left shift the multiplicand and use it again for further calculation.
This method can also be looked at as repetitive addition.

This current implementation works for simple test cases, but when I wrote a separate test bench for this module, I couldn't integrate it properly and same happened while integrating it with the ALU. The testbench gives indication of wrong results because the signals are not in sync.

Divider

The implementation is similar to long division technique we use for dividing numbers manually. Use the divisor to start dividing the dividend from most significant digits and keep track of difference as the quotient. The final value remaining from the dividend is considered the remainder.
It is illustrated in following example -

Dividend: 1011
Divisor: 0101

```
     1011
0101          ( - )  The result is negative so put a 0 bit in quotient

   1011
0101          ( - )  The result is negative so put a 0 bit in quotient

  1011
0101          ( - )  The result is positive, use the difference and put 1 in quotient

0001
0101          ( - ) The result is negative, so put 0 in quotient and use dividend as remainder.
```

Quotient = 0010
Remainder = 0001

I tested my divider input using some hard coded inputs and it works, but when I used control signals and clock to integrate it with test bench and ALU, it stopped working. The reason seems to be same as with multiplier, the control signals are not in sync properly.