Graehme Blair
Drew Hasse
EECS 355
Lab 4: Behavioral Divider
Simulation Results

Two simulations were run. Input files for 16 and 32 bit test cases were used to test the design. Numbers for the divider tests were read in from a text file in the pattern:
Dividend
Divisor
And then passed to the divider component.

16-bit divider simulation input:	Output:
12 3	12 / 3 = 4 -- 0
16 -3	16 / -3 = -5 -- 1
1000 0	1000 / 0 = 1000 -- 0 OVERFLOW
-12 7	-12 / 7 = -1 -- -5
25000 125	25000 / 125 = 200 -- 0
12 9	12 / 9 = 1 -- 3
3200 52	3200 / 52 = 61 -- 28
-7190 127	-7190 / 127 = -56 -- -78
12345 123	12345 / 123 = 100 -- 45
-9876 -102	-9876 / -102 = 96 -- -84
-128 -5	-128 / -5 = 25 -- -3

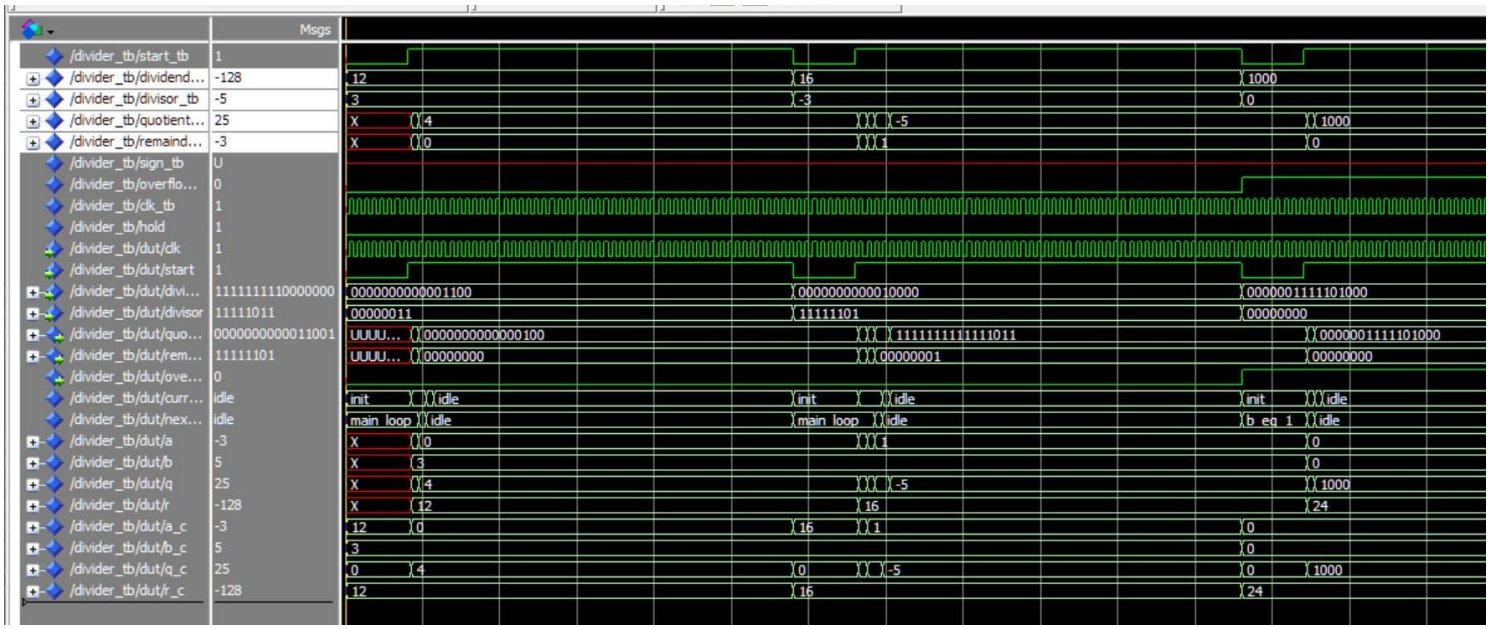

Figure 1: 16-bit test waveform

32-bit divider simulation input:

-1000000
2500
4598093
-12300
1000009
4598
-3200000
24501
3490000
0
123456
9876
-4543245
6475
63665742
567
-27538530
-9852
-6575097
45

Output:

-1000000 / 2500 = -400 -- 0

4598093 / -12300 = -373 -- 10193

1000009 / 4598 = 217 -- 2243

-3200000 / 24501 = -130 -- -14870

3490000 / 0 = 3490000 -- 0 OVERFLOW

123456 / 9876 = 12 -- 4944

-4543245 / 6475 = -701 -- -4270

63665742 / 567 = 112285 -- 147

-27538530 / -9852 = 2795 -- -2190

-6575097 / 45 = -146113 -- -12

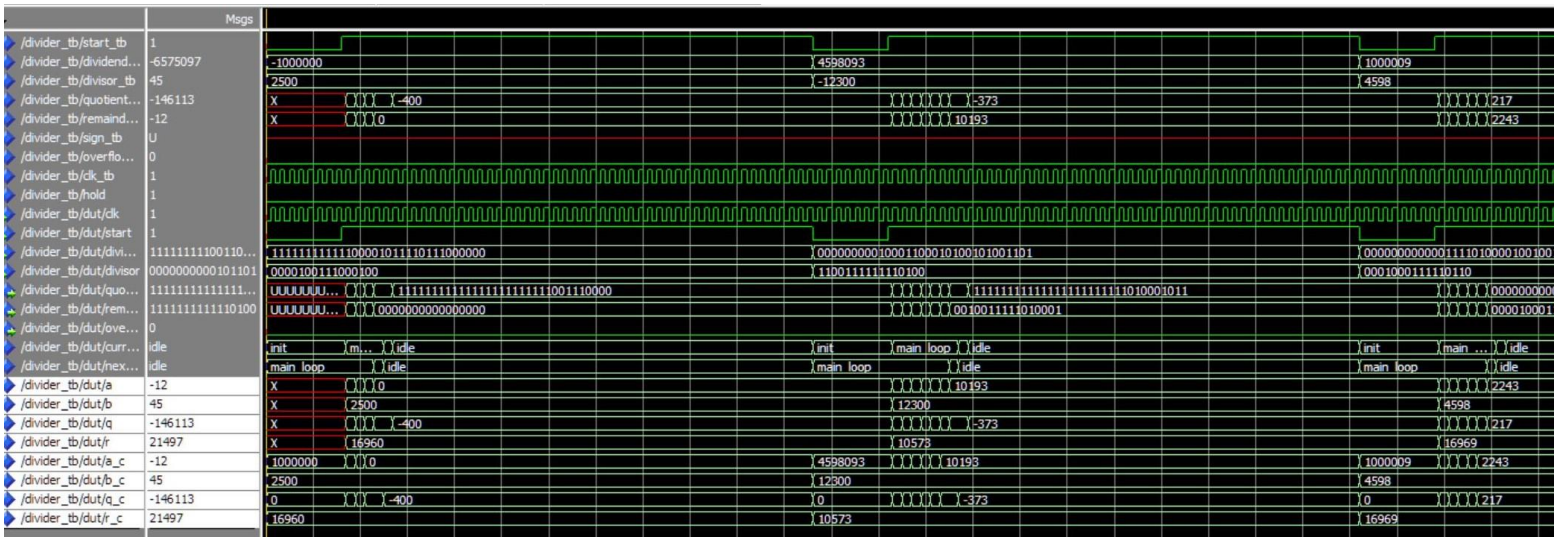

Figure 2: 32-bit waveform output

As shown by the simulations and output files, the divider calculates the correct answer with the appropriate signs for the quotient and remainder. The sign_tb which is shown as a red line above was removed from the divider because it was unnecssary.

Design:

The finite state machine used to implement this division consists of five states: idle, init, b_eq_1, main_loop, and epilogue. These are the same states shown in the finite state diagram from the lecture 9 slides. From the idle state, if the start button is pressed, the next state is the init state, in which the initial values are set so that the division can begin. The value of a is set to the dividend, b is set to the divisor, and the quotient to zero. From the init state, if b is 0 or 1, then the next state is the b_eq_1 state. This is because if b is 1, the quotient will just be a, and the remainder will be zero, so we can shortcut the process. If b is 0, then you are trying to divide by zero, and the overflow flag is set and the quotient is set to the value of a. If in the init state, b is not equal to zero or one, the FSM progresses to the main_loop state. This state implements the while loop in the pseudocode if the conditions are met, by calculating p using the get_msb_pos function, adding 2^p, to the current quotient, and updating the value of a. In this case when the conditions are met, the next state will be the main_loop. This state will continue to have itself as the next state until the conditions (b != 0 && a >= b) are not met, in which case, the FSM will advance to the epilogue. In the epilogue state, the sign of the quotient is found by XORing the MSB of dividend and divisor. If sign is 1, then the quotient is negated, since our loop was operating on the absolute values of the dividend and divisor. The remainder sign is determined by the sign of the dividend, so if MSB of dividend is 1, then the remainder is inverted. From the epilogue, the FSM will move to the idle state, since nothing else needs to be done, where it will wait for the start signal. Also, if at any point the start button is pressed, the current state will be set to the init state, and the calculation will start from there. Looking at the waveform in ModelSim, it appears that the average 32-bit division takes 8 or 9 cycles with this implementation. Some, are much faster as in the case when b = 1, and others may be slower, but most seem to be around 8 or 9. The 16-bit test cases appear to take around 4 or 5 cycles for the average case. The for loop implementation of get_msb_pos is slightly longer than the recursive version and looks more like a line. The recursive, version is much wider and more parallel, and resembles a tree. The two are functionally equivalent, but the recursive version, breaks up

the std_logic_vector into pieces and finds the MSB of each piece, finally returning the MSB position of the entire vector. In terms of area, the recursive implementation is more square shaped, while the long chain in the for loop implementation forms a rectangle, so it appears that the recursive version occupies a slightly larger area, but at the same time, it is also faster.

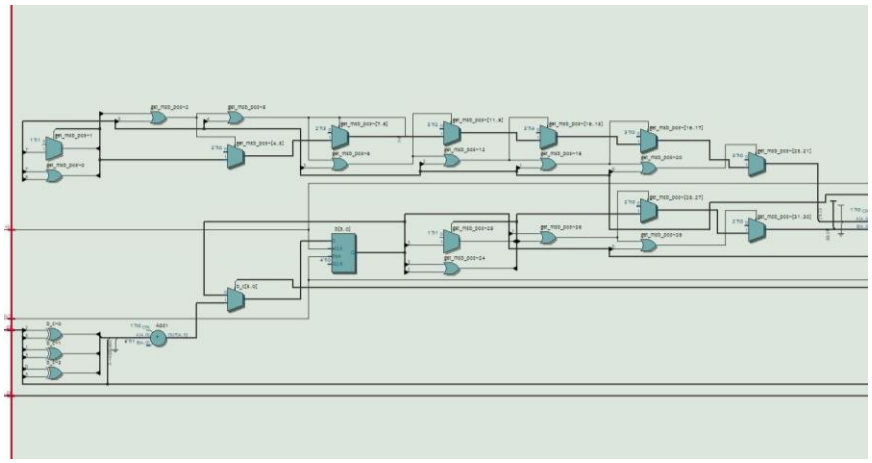

Figure 3: For loop implementation logic of get_msb_pos shown by RTL viewer

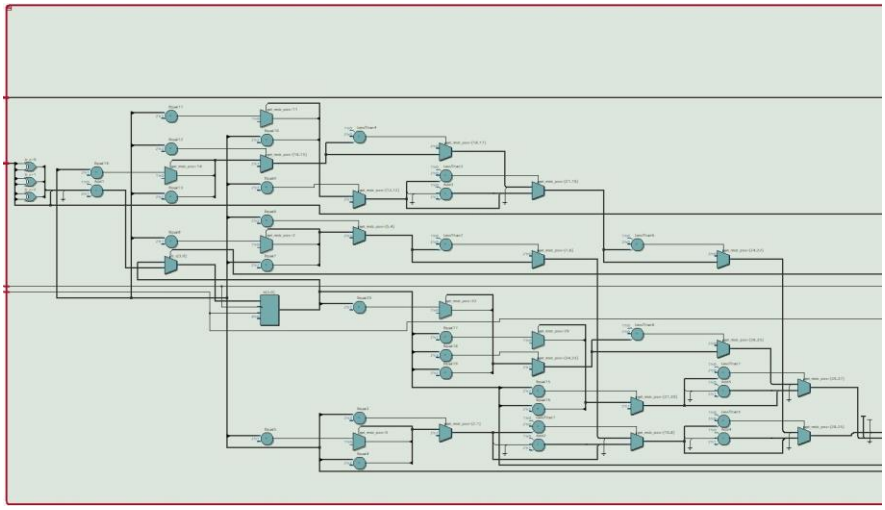

Figure 4: Recursive implementation logic of get_msb_pos shown by RTL viewer