ECSE318 Homework 1

Benjamin Scholar

CaseID: bbs27

Email: scholar@case.edu

Problem 1:

Files:

- P1.v - contains the unsigned_parallel_multiplier module
- P1_tb.v - testbench
- Csa_module.v - contains a module that is basically an array of full adders
- Prop_adder.v - contains a carry propagation adder

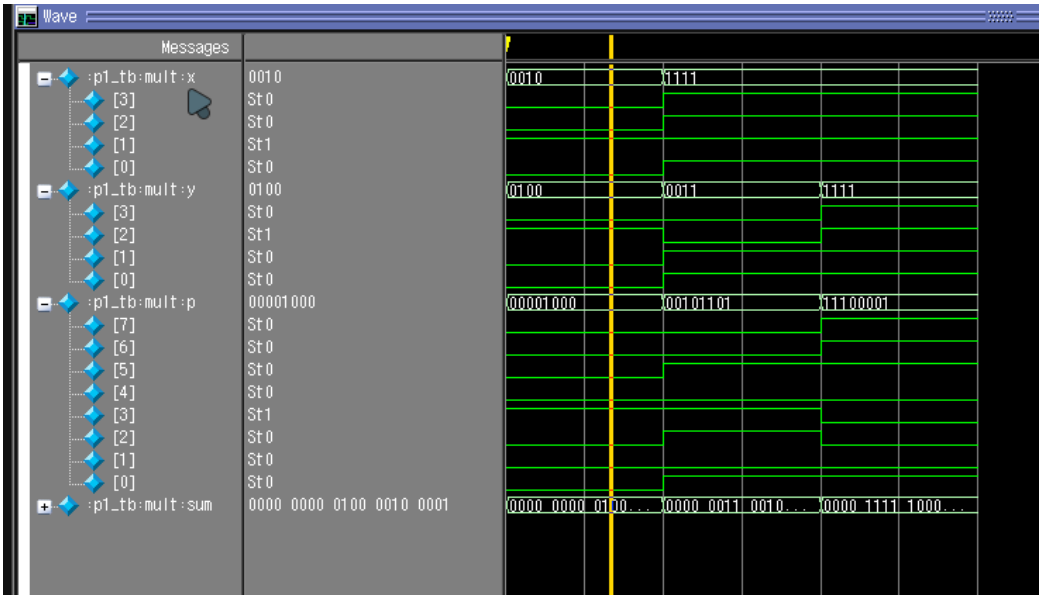
The objective of problem 1 is to create an unsigned parallel multiplier using the structural style of verilog. An array of carry save adders in combination with a final carry propagate adder stage is used to achieve this functionality. The final verilog structural verilog model can be found in the *p1.v* file within this submission. The file contains the *unsigned_parallel_multiplier* module, which contains the logic for this problem. Additionally, the testbench program for this problem can be found in *p1_tb.v*.

Regarding the design of this module, a generate loop was utilized to simplify the design of the file and allow for the module to support variable data width using a parameter. In short, a 2-dimensional array of wires is declared for both the carry and summation bits and each iteration of the generate loop declares a carry save adder module connecting (declared in the *csa_module.v* file) as well as an assign statement to the output. The *csa_module* module is simply an array of full adders that outputs the carry out and sum to two separate output arrays instead of completing the addition operation. An additional carry propagate adder (found in the *prop_adder.v* file) is used to generate the upper N bits (where N is the input data width). The *proper_adder* module contains a carry propagation adder, which is simply an array of full adders with an internal wire array to propagate the carry bits between the adders/input/output. The test bench file is fairly self explanatory, as it contains only the reg declarations, the module declaration, and a monitor statement which prints to the simulation console on every change to

the inputs/outputs. The simplicity of the test bench is due to the fact that this multiplier does not require any control signals.

The module was tested with the required inputs specified in the homework document (2*4, 15*3) as well as an additional case which multiplies 15 with itself. The additional test case is used to make sure that the carry functionality works correctly with large values, as I have had similar problems in the past with multipliers. I did not, however, run into an issue with the carry in this instance. A simulation output and the corresponding wave form can be seen below. It can be seen that the module generates the correct output for each set of inputs.

```
# vsim -do {run -all} -c work.p1_tb
# //  QuestaSim-64 6.5f Jun 16 2010 Linux 5.15.0-46-generic
# //
# //  Copyright 1991-2010 Mentor Graphics Corporation
# //             All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading work.p1_tb
# Loading work.unsigned_parallel_multiplier
# Loading work.prop_adder
# Loading work.csa_module
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim
#          File in use by: bscholar  Hostname: Scholar-Zenbo
#          Attempting to use alternate WLF file "./wlfteFoha
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.w
#          Using alternate file: ./wlfteFohaz
# ** Warning: (vsim-3009) [TSCALE] - Module 'csa_module' does
#        Region: :p1_tb:mult:row_loop[0]:csa
# Loading work.full_adder
# ** Warning: (vsim-3009) [TSCALE] - Module 'full_adder' does
#        Region: :p1_tb:mult:cpa:adders[3]
# run -all
# x=2, y=4 ⇒ p=8
# x=15, y=3 ⇒ p=45
# x=15, y=15 ⇒ p=225
```



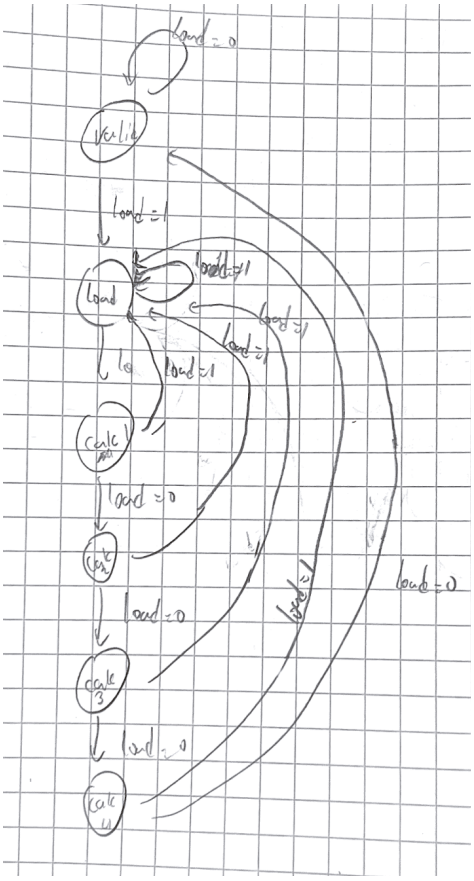
Problem 2:

Files:

- P2.v - contains the cyclical_multiplier module
- P2_tb.v - test bench
- Unidirectional_shift_reg.v - shift register module
- Register.v - normal register module
- Mux.v - structural mux

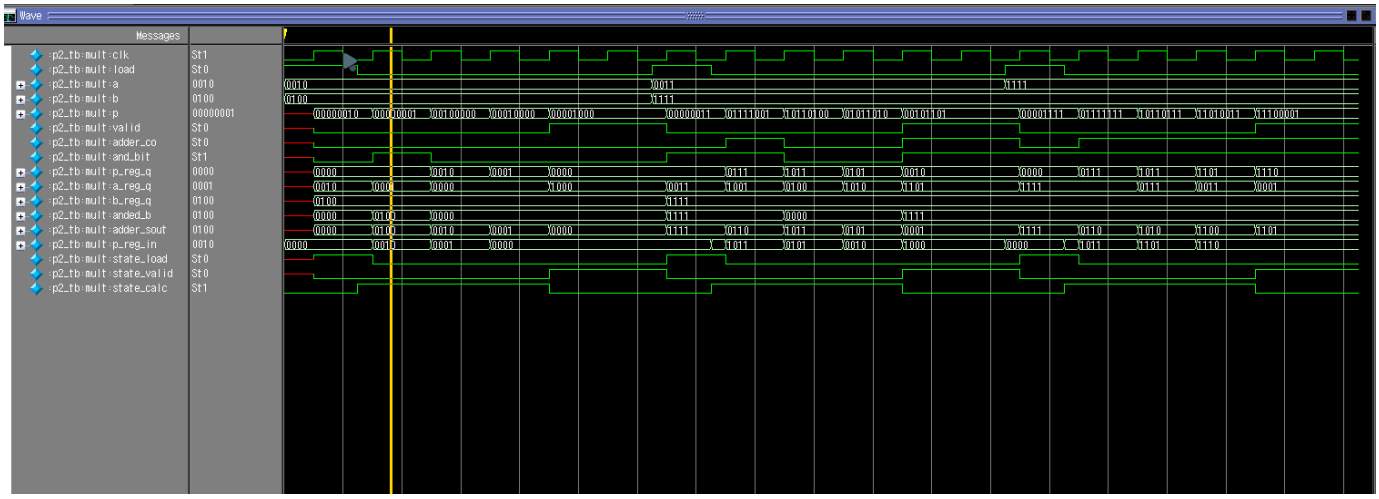
For problem 4, a cyclical multiplier was designed and written in structural verilog. This is done using two registers (see the *register* module) and one shift register (see the *unidirectional_shift_reg* module). Additionally, a state machine implemented via a counter is designed in order to control the multiplier. Because this design takes N cycles to complete a full operation, where N is the number of bits in the input data, the module requires a couple of extra input/output lines to determine which state it should be in. A *load* signal is used as an input so that the multiplier can be signaled that it's time to load new data into the registers. A *valid* output signal is used to specify when the multiplier's current operation is complete.

The *cyclic_control* module is the state machine that controls the multiplier. It was determined that N + 2 states are needed for N bits of input data, therefore only a $\log_2(N + 2)$ bit counter is needed to control the module. The two additional states are the valid state and the load state, the rest of the states represent the number of bit shifts that have occurred in the current multiplication operation. It can be seen that the state machine will always go into the load state when the load input line is high, therefore the state machine can be put into the correct state without a reset line. The state diagram is as follows:



The A register is a shift register and is loaded with the raw input A when entering the load state. The P and B registers are both non-shifting registers, and are loaded with 0 and the raw B input respectively when entering the load state. When in the valid/hold state, all 3 registers hold their contents and do not load/shift. When in any of the calculation states, the B register holds its data. On every cycle, the output of the entire B register is ANDed with the lowest bit of the A register. This value is then added together with the output of the P register.On the next clock edge, the P register loads the upper N bits of the output of the adder (including the adder's carry out, not including the lowest bit of the output of the adder) while the A register shifts 1 bit to the right, taking the lowest bit of the adder's output as the shift-in input. Doing this allows for the loading and shifting of the P and A registers to be done on one cycle. It was a possibility that I could have made the P register a shift register as well, but I ran into difficulties getting the data to be loaded into P and shifted between both registers on one cycle to be very difficult. My method allowed the design to complete each operation in 6 clock cycles.

The printed output and waveforms can be seen below.




```
VSIM 22> run -all
# a=2, b=4 => p=x, valid=x
# a=2, b=4 => p=2, valid=0
# a=2, b=4 => p=1, valid=0
# a=2, b=4 => p=32, valid=0
# a=2, b=4 => p=16, valid=0
# a=2, b=4 => p=8, valid=1
#
# a=3, b=15 => p=8, valid=1
# a=3, b=15 => p=3, valid=0
# a=3, b=15 => p=121, valid=0
# a=3, b=15 => p=180, valid=0
# a=3, b=15 => p=90, valid=0
# a=3, b=15 => p=45, valid=1
#
# a=15, b=15 => p=45, valid=1
# a=15, b=15 => p=15, valid=0
# a=15, b=15 => p=127, valid=0
# a=15, b=15 => p=183, valid=0
# a=15, b=15 => p=211, valid=0
# a=15, b=15 => p=225, valid=1
# 1
```

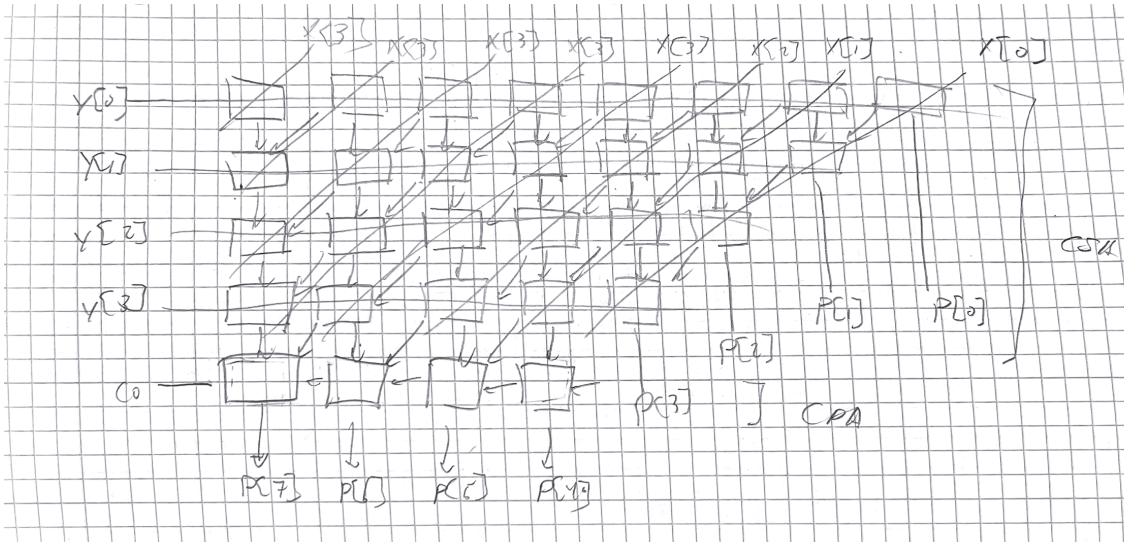
Problem 3:

Files:

- P3.v - contains the signed_multiplier module
- P3_tb.v - testbench
- Twos_complement.v - generates the complement of an input in two's complement representation.
- Conditional_complement.v - contains a module that selects between a number and its complement based on an input line (basically a mux)
- Csa_module.v - contains a module that is basically an array of full adders
- Prop_adder.v - contains a carry propagation adder
- Sign_extend.v - contains a sign extension module with variable input/output widths

This problem has a very similar solution to problem 1. The differences are that the inputs are signed numbers instead of unsigned numbers. As stated in the assignment document, if only the multiplicand is negative (MSB = 1), then we can continue multiplying without taking the complement. If the multiplier is negative, however, we must take the complement of both the multiplicand and multiplier. This was achieved using the *conditional_complement* and *twos_complement* modules.The two's complement module simply inverts the bits of the input data and then adds one. The conditional complement module muxes between the raw input data and the output of a twos_complement module. When the multiplier is negative, the twos_complement modules are set to choose the complement of both the multiplier and multiplicand. The output of the multiplicand's conditional_complement multiplier is sign extended (using the *sign_extend* module) to double the input data width.

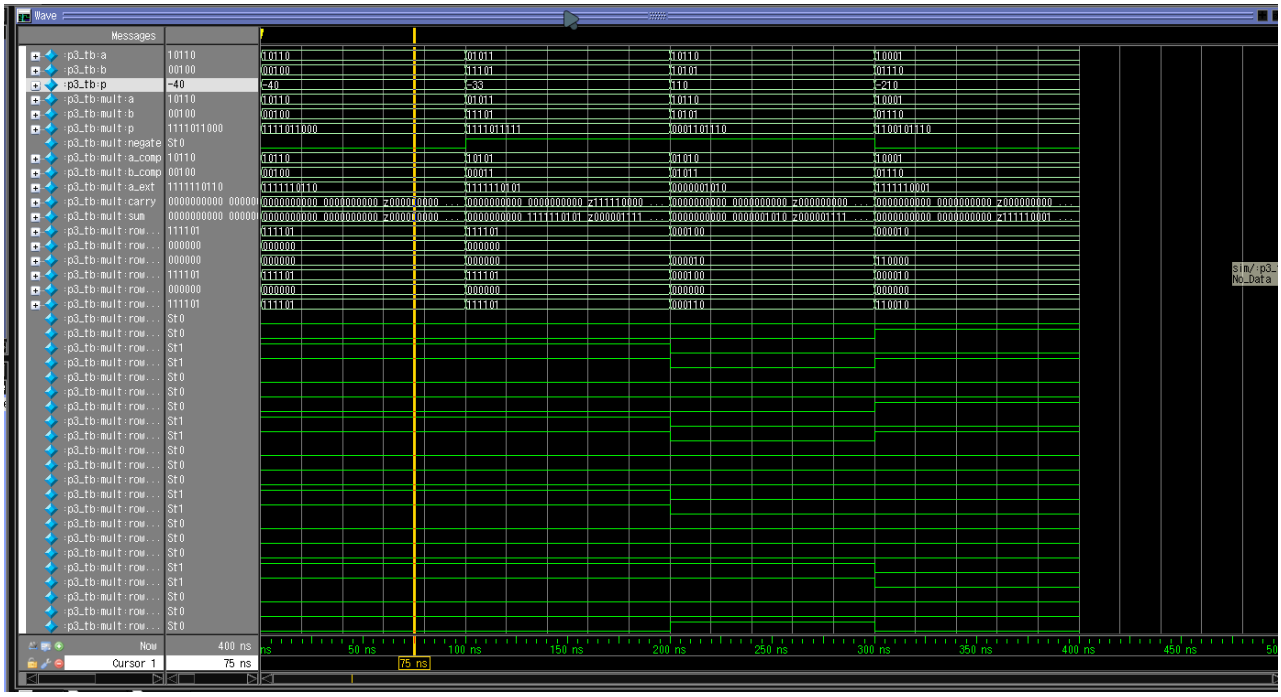
The multiplier was layed out in a similar way to problem 1. A (very rough) sketch of the structure can be seen below. Once again, an array of carry save adders in addition to a final stage containing a carry propagate adder. This was done via a generate loop, as it allowed the design to be more flexible in terms of data width and easier to design in the first place.



The design was tested on the required inputs (-10*4, 11*-3, -10*-11) as well as 15*-14 for additional assurance that the design still works for higher input values. The output and waveforms of the design can be seen below. The printed outputs are printed in hex because I could not figure out how to print the data in signed form. The signed outputs in decimal can be seen in the waveform.

```
VSIM 12> run -all
# a=   22, b=    4 => p=  3d8
# a=   11, b=   29 => p=  3df
# a=   22, b=   21 => p=  06e
# a=   17, b=   14 => p=  32e
# 1
# Break in Module p3_tb at p3_tb.v line 38

VSIM 13>
```


Wave
Messages
:p3_tb:a
:p3_tb:b
:p3_tb:p
:p3_tb:mult:a
:p3_tb:mult:b
:p3_tb:mult:p
:p3_tb:mult:negate
:p3_tb:mult:a_comp
:p3_tb:mult:b_comp
:p3_tb:mult:a_ext
:p3_tb:mult:carry
:p3_tb:mult:sum
:p3_tb:mult:row...
Now
400 ns
Cursor 1
75 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350 ns
400 ns
450 ns
Sim:/p3_
No_Data

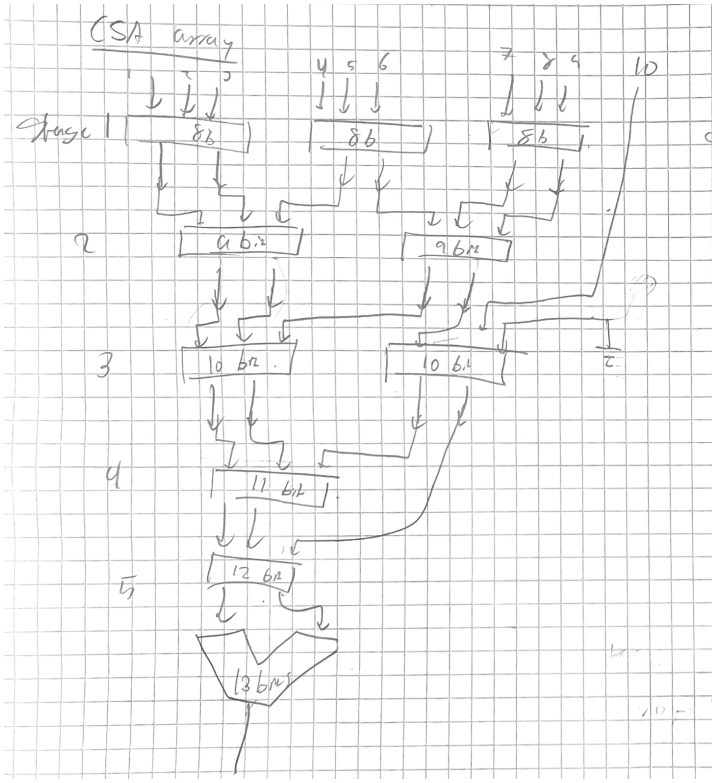
Problem 4:

Files:

- P4.v - contains the csa10x8 module, which contains the logic for this problem
- P4_tb.v
- Csa_module.v
- prop_adder.v

The end goal of problem 4 is to create a carry save adder with 10 8-bit inputs. The *csa8x10* module is within the *p4.v* file within this submission along the corresponding test bench in *p4_tb.v*.

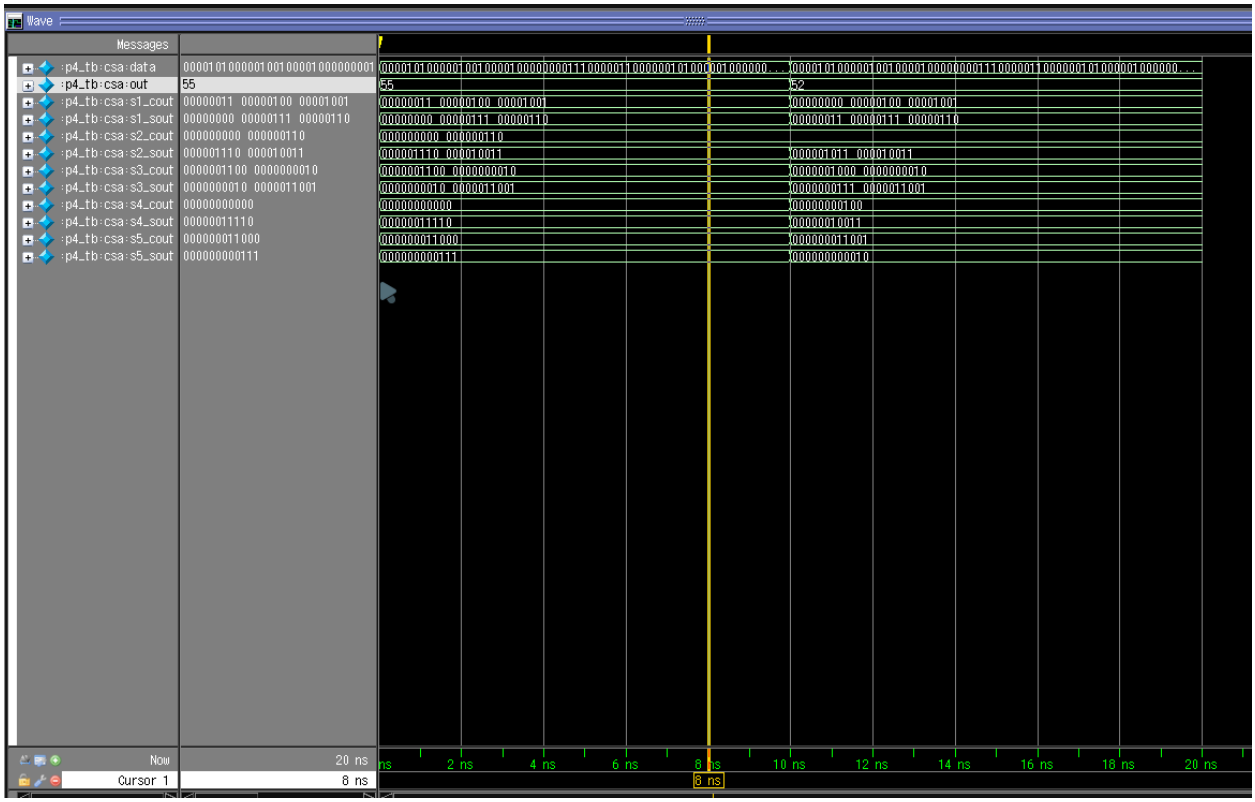
For my design, I found that 5 stages of carry save adders plus an additional propagation adder stage were necessary for 10 8-bit wide inputs. Nine carry save adder modules of various widths were used in total. Each carry save module can only take 3 inputs at a time due to the fact that having 4 inputs could result in there being 2 carry out bits for each bit in the input (1 + 1 + 1 + 1 = 100). The same carry save modules (*csa_module)* are being used as in problems 1 and 3. The layout that I chose can be seen below:



It can be seen that the carry save modules increase in width as the stage number increases. Between each stage, the carry outs of the previous stage are shifted 1 bit to the left and the sum outs have an extra 0 appended as the MSB. An additional choice that I made was to use a single 80 bit (8 bits * 10 inputs) wide data input instead of 10 individual input ports with 8 bits each in order to make my life easier. The first 8 bits, for example, represent the first integer and so on. At first, I attempted to use a 2-dimensional array as input, but I soon found out that that feature is not supported in base Verilog, only SystemVerilog.

The module was tested with the two input sequences specified in the assignment document (adding 1-10 and adding 3-10 which should equal 55 and 52 respectively). The simulation output and waveform can be seen below. It can be seen that the output of the module is correct in both test cases.

```
# vsim -do {run -all} -c work.p4_tb
# //  QuestaSim-64 6.5f Jun 16 2010 Linux 5.15.0-46-generic
# //
# //  Copyright 1991-2010 Mentor Graphics Corporation
# //              All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading work.p4_tb
# Loading work.csa8x10
# Loading work.csa_module
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim
#           File in use by: bscholar  Hostname: Scholar-Zenbo
#           Attempting to use alternate WLF file "./wlftD7J2X
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.w
#           Using alternate file: ./wlftD7J2Xu
# ** Warning: (vsim-3009) [TSCALE] - Module 'csa_module' does
#         Region: :p4_tb:csa:s1m1
# Loading work.full_adder
# ** Warning: (vsim-3009) [TSCALE] - Module 'full_adder' does
#         Region: :p4_tb:csa:s5m1:adders[11]
# run -all
# out=55
# out=52
```


Wave
Messages
:p4_tb:csa:data
:p4_tb:csa:out
:p4_tb:csa:s1_cout
:p4_tb:csa:s1_sout
:p4_tb:csa:s2_cout
:p4_tb:csa:s2_sout
:p4_tb:csa:s3_cout
:p4_tb:csa:s3_sout
:p4_tb:csa:s4_cout
:p4_tb:csa:s4_sout
:p4_tb:csa:s5_cout
:p4_tb:csa:s5_sout
55
00000011 00000100 00001001
00000000 00000111 00000110
000000000 000000110
000001110 000010011
000000001100 0000000010
000000010 0000011001
00000000000
000000011110
000000011000
000000000111
Now
20 ns
Cursor 1
8 ns
2 ns
4 ns
6 ns
8 ns
10 ns
12 ns
14 ns
16 ns
18 ns
20 ns

Problem 5:

Files:

- P5.v - contains the p5_struct and p5_behav modules
- P5_tb.v - testbench
- Dff.v - contains a d-flip-flop module

Both versions of the state machine are contained within the *p5.v* file, and the test bench is located in the *p5_tb.v*. Within the *p5.v* file, the structural version of the state machine is within the *p5_struct* module and the behavioral version is in the *p5_behav* module.

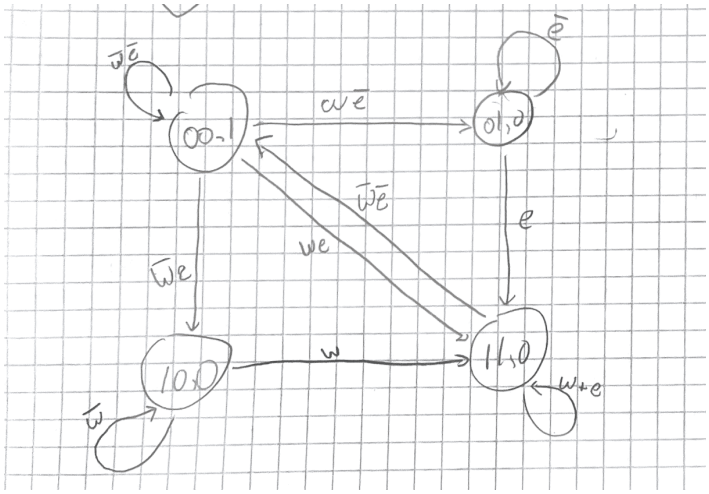
It was determined that the excitation equations for the given state machine are as follows:

$$A = a\bar{b} + aw + e$$
$$B = \bar{a}b + be + w$$

A K-map depicting the next state was then generated from the excitation equations, which can be seen below. Stable states are shown with blue highlights.

ab \ we	00	01	11	10
00	00	10	11	01
01	01	11	11	01
11	00	11	11	11
10	10	10	11	11

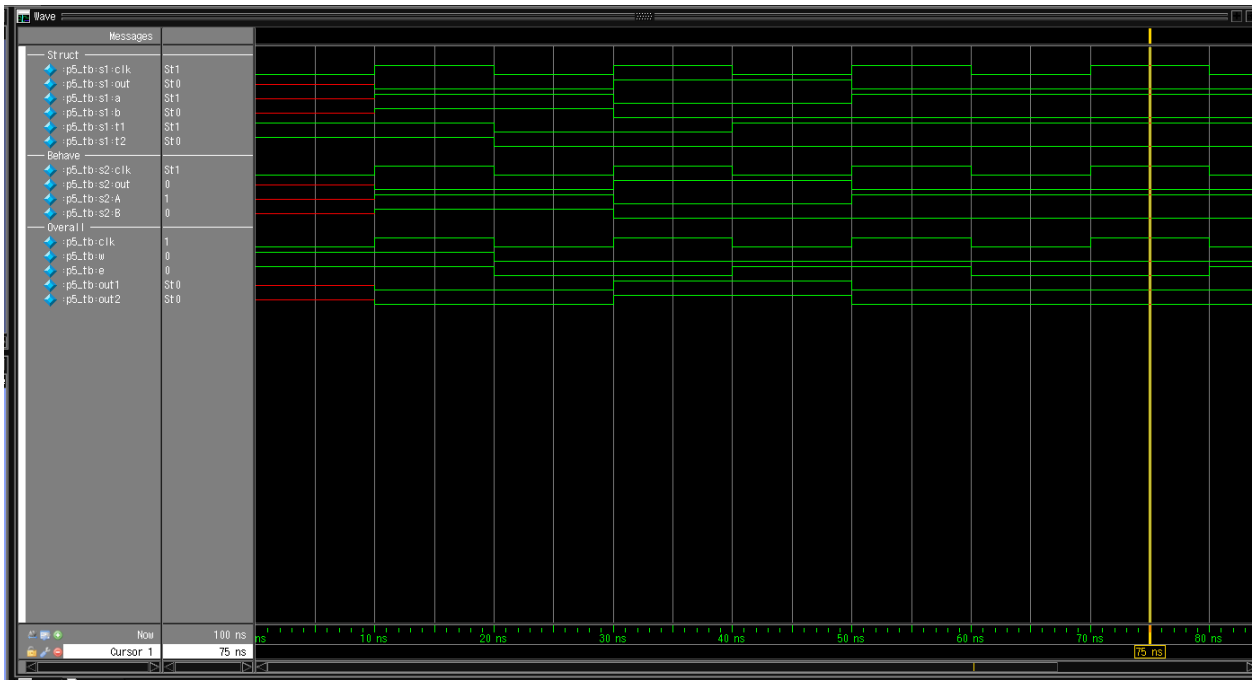
The state transition diagram can be seen as follows:



It can be seen that every node has an output value associated with it, thus making this a Moore machine. The output is simply the output of the flip-flop containing A nor the output of the flip-flop containing B.

The structural version of the state machine uses two d-flip-flop modules (see *dff.v*). The *dff* module is behavioral because I could not get the structural version of the dff to work. The behavioral version of the machine simply uses an always block triggered on the positive edge of the clock with two non-blocking assignments. The waveform of the state machines can be seen below.

It can be seen from the next-state K-map that when w and e are both equal to 1, the next state of {A, B} will be equal to 11. This allows for the state machine to not have a reset line. It can be seen in the test bench that {W, E} is set to 11 initially in order to make sure that the state within the flip-flop is defined.


Wave
Messages
Struct
/p5_tb/s1/clk
/p5_tb/s1/out
/p5_tb/s1/a
/p5_tb/s1/b
/p5_tb/s1/t1
/p5_tb/s1/t2
Behave
/p5_tb/s2/clk
/p5_tb/s2/out
/p5_tb/s2/A
/p5_tb/s2/B
Overall
/p5_tb/clk
/p5_tb/w
/p5_tb/e
/p5_tb/out1
/p5_tb/out2
St1
St0
St1
St0
St1
St0
St1
0
1
0
1
0
0
St0
St0
Now
100 ns
Cursor 1
75 ns
ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns
75 ns