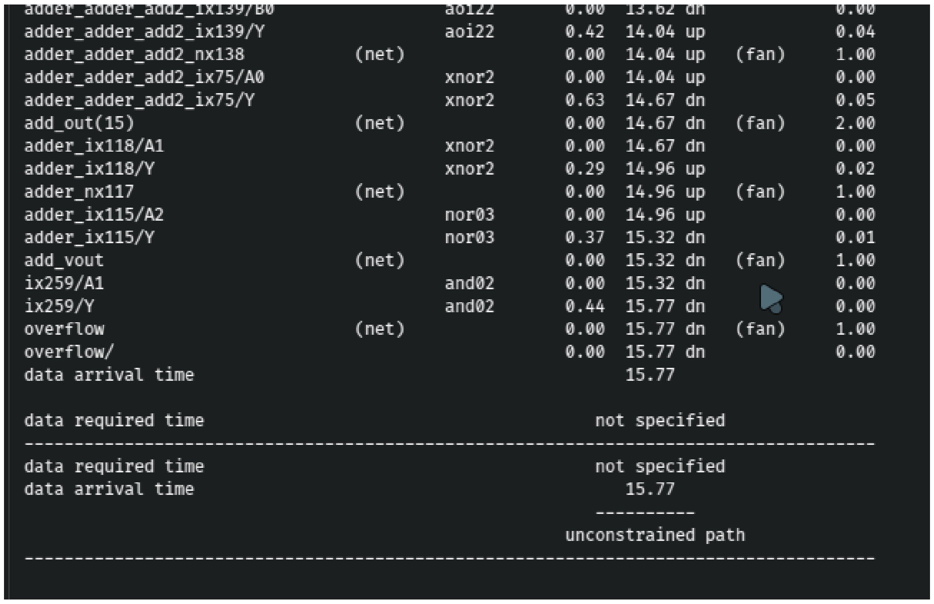
Problem 1:

Files (in p1 subdirectory):

- Alu.v - contains the alu module
- Alu_tb.v - testbench for full alu
- Adder.v - wrapper around the raw adder modules (prop_adder and conditional_sum_adder), adds required lines for subtraction, overflow.
- Prop_adder.v - simply ripple carry adder modules
- Conditional_sum_adder16.v - contains conditional sum adder modules
- Full_adder.v - contains a full adder module
- Adder_tb.v - testbench for adder module, derived from output in HW assignment
- Logic.v - contains module that does logical bitwise operations
- Comparator.v - contains module that compares inputs
- Shift.v - contains a 16 bit barrel shifter module
- Syn directory - contains input/output of synthesis
- syn/fast_adder_out - data from fast adder synthesis
- syn/slow_adder_out - data from slow adder synthesis

For this part of the homework, we were required to design a 16 bit ALU to be synthesized. For my heirarcale design, I chose to have the ALU consist of a few “submodules” that process different types of information differently. The submodules consist of an adder/subtractor, logical unit (bitwise operations), a comparator, and a barrel shifter. Each submodule is given the A and B inputs as well as the lowest 3 bits of the operation. The outputs of each submodule are then selected using a 4 input multiplexer with the select lines tied to the 2 most upper bits of the operation input of the ALU. I could have probably used parts of the logical unit to simplify the comparator (use the XOR function to check for equality) but I found that this will increase complexity of the instruction decode. For the purpose of this lab I decided to keep the design simple.

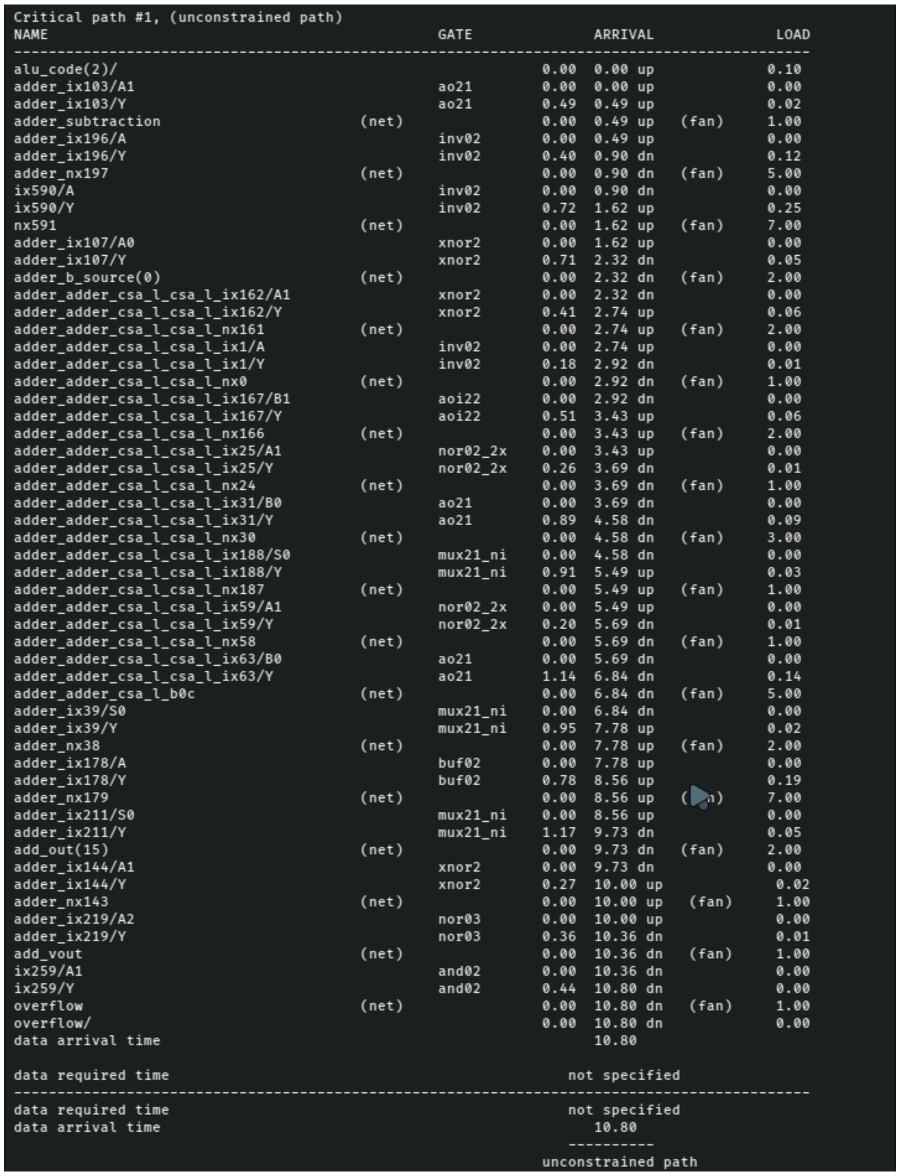
At first, I used a standard ripple carry adder as the add/subtract module. Later on a switch to a conditional sum adder in order to optimize the critical path of the ALU. In both cases, the adder was the slowest part of the design but the conditional sum adder made a significant improvement in terms of propagation delay in the design. The critical path of the ALU with the ripple carry adder was found to be 15.77 time units (which I assume are nanoseconds but it does not specify anywhere that I can see). This can be seen below in the output of the synthesis.



```
adder_adder_add2_ix139/B0                 aoi22       0.00  13.62 dn              0.00
adder_adder_add2_ix139/Y                  aoi22       0.42  14.04 up              0.04
adder_adder_add2_nx138           (net)                0.00  14.04 up    (fan)     1.00
adder_adder_add2_ix75/A0                  xnor2       0.00  14.04 up              0.00
adder_adder_add2_ix75/Y                   xnor2       0.63  14.67 dn              0.05
add_out(15)                      (net)                0.00  14.67 dn    (fan)     2.00
adder_ix118/A1                            xnor2       0.00  14.67 dn              0.00
adder_ix118/Y                             xnor2       0.29  14.96 up              0.02
adder_nx117                      (net)                0.00  14.96 up    (fan)     1.00
adder_ix115/A2                            nor03       0.00  14.96 up              0.00
adder_ix115/Y                             nor03       0.37  15.32 dn              0.01
add_vout                         (net)                0.00  15.32 dn    (fan)     1.00
ix259/A1                                  and02       0.00  15.32 dn              0.00
ix259/Y                                   and02       0.44  15.77 dn              0.00
overflow                         (net)                0.00  15.77 dn    (fan)     1.00
overflow/                                             0.00  15.77 dn              0.00
data arrival time                                           15.77

data required time                                       not specified
-------------------------------------------------------------------------------------
data required time                                       not specified
data arrival time                                           15.77
                                                         ----------
                                                      unconstrained path
-------------------------------------------------------------------------------------
```

The critical path of the ALU with the conditional sum adder was found to be 10.80 time units, which can be seen below.



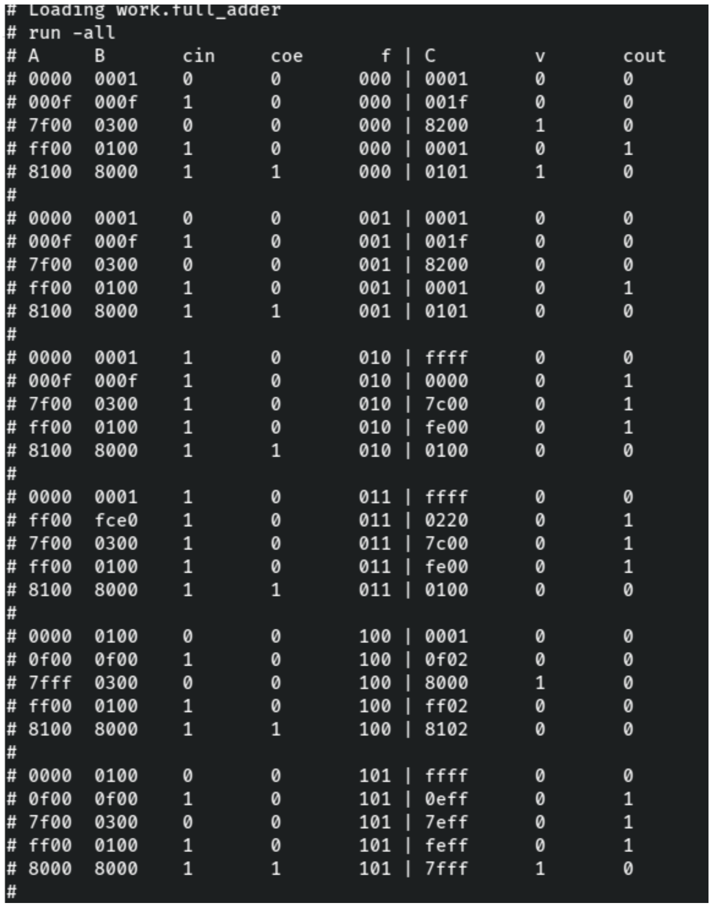
```
Critical path #1, (unconstrained path)
NAME                                            GATE             ARRIVAL                LOAD
---------------------------------------------------------------------------------------------
alu_code(2)/                                                 0.00  0.00 up               0.10
adder_ix103/A1                                  ao21         0.00  0.00 up               0.00
adder_ix103/Y                                   ao21         0.49  0.49 up               0.02
adder_subtraction                       (net)                0.00  0.49 up   (fan)       1.00
adder_ix196/A                                   inv02        0.00  0.49 up               0.00
adder_ix196/Y                                   inv02        0.40  0.90 dn               0.12
adder_nx197                             (net)                0.00  0.90 dn   (fan)       5.00
ix590/A                                         inv02        0.00  0.90 dn               0.00
ix590/Y                                         inv02        0.72  1.62 up               0.25
nx591                                   (net)                0.00  1.62 up   (fan)       7.00
adder_ix107/A0                                  xnor2        0.00  1.62 up               0.00
adder_ix107/Y                                   xnor2        0.71  2.32 dn               0.05
adder_b_source(0)                       (net)                0.00  2.32 dn   (fan)       2.00
adder_adder_csa_l_csa_l_ix162/A1                xnor2        0.00  2.32 dn               0.00
adder_adder_csa_l_csa_l_ix162/Y                 xnor2        0.41  2.74 up               0.06
adder_adder_csa_l_csa_l_nx161           (net)                0.00  2.74 up   (fan)       2.00
adder_adder_csa_l_csa_l_ix1/A                   inv02        0.00  2.74 up               0.00
adder_adder_csa_l_csa_l_ix1/Y                   inv02        0.18  2.92 dn               0.01
adder_adder_csa_l_csa_l_nx0             (net)                0.00  2.92 dn   (fan)       1.00
adder_adder_csa_l_csa_l_ix167/B1                aoi22        0.00  2.92 dn               0.00
adder_adder_csa_l_csa_l_ix167/Y                 aoi22        0.51  3.43 up               0.06
adder_adder_csa_l_csa_l_nx166           (net)                0.00  3.43 up   (fan)       2.00
adder_adder_csa_l_csa_l_ix25/A1                 nor02_2x     0.00  3.43 up               0.00
adder_adder_csa_l_csa_l_ix25/Y                  nor02_2x     0.26  3.69 dn               0.01
adder_adder_csa_l_csa_l_nx24            (net)                0.00  3.69 dn   (fan)       1.00
adder_adder_csa_l_csa_l_ix31/B0                 ao21         0.00  3.69 dn               0.00
adder_adder_csa_l_csa_l_ix31/Y                  ao21         0.89  4.58 dn               0.09
adder_adder_csa_l_csa_l_nx30            (net)                0.00  4.58 dn   (fan)       3.00
adder_adder_csa_l_csa_l_ix188/S0                mux21_ni     0.00  4.58 dn               0.00
adder_adder_csa_l_csa_l_ix188/Y                 mux21_ni     0.91  5.49 up               0.03
adder_adder_csa_l_csa_l_nx187           (net)                0.00  5.49 up   (fan)       1.00
adder_adder_csa_l_csa_l_ix59/A1                 nor02_2x     0.00  5.49 up               0.00
adder_adder_csa_l_csa_l_ix59/Y                  nor02_2x     0.20  5.69 dn               0.01
adder_adder_csa_l_csa_l_nx58            (net)                0.00  5.69 dn   (fan)       1.00
adder_adder_csa_l_csa_l_ix63/B0                 ao21         0.00  5.69 dn               0.00
adder_adder_csa_l_csa_l_ix63/Y                  ao21         1.14  6.84 dn               0.14
adder_adder_csa_l_b0c                   (net)                0.00  6.84 dn   (fan)       5.00
adder_ix39/S0                                   mux21_ni     0.00  6.84 dn               0.00
adder_ix39/Y                                    mux21_ni     0.95  7.78 up               0.02
adder_nx38                              (net)                0.00  7.78 up   (fan)       2.00
adder_ix178/A                                   buf02        0.00  7.78 up               0.00
adder_ix178/Y                                   buf02        0.78  8.56 up               0.19
adder_nx179                             (net)                0.00  8.56 up   (fan)       7.00
adder_ix211/S0                                  mux21_ni     0.00  8.56 up               0.00
adder_ix211/Y                                   mux21_ni     1.17  9.73 dn               0.05
add_out(15)                             (net)                0.00  9.73 dn   (fan)       2.00
adder_ix144/A1                                  xnor2        0.00  9.73 dn               0.00
adder_ix144/Y                                   xnor2        0.27  10.00 up               0.02
adder_nx143                             (net)                0.00  10.00 up   (fan)       1.00
adder_ix219/A2                                  nor03        0.00  10.00 up               0.00
adder_ix219/Y                                   nor03        0.36  10.36 dn               0.01
add_vout                                (net)                0.00  10.36 dn   (fan)       1.00
ix259/A1                                        and02        0.00  10.36 dn               0.00
ix259/Y                                         and02        0.44  10.80 dn               0.00
overflow                                (net)                0.00  10.80 dn   (fan)       1.00
overflow/                                                    0.00  10.80 dn               0.00
data arrival time                                                  10.80

data required time                                              not specified
---------------------------------------------------------------------------------------------
data required time                                              not specified
data arrival time                                                  10.80
                                                                ----------
                                                             unconstrained path
```

From 15 to 10 time units is a massive speed up at around 33% reduction in delay. In both cases, the critical path of the ALU was from the alu_code input, through the adder module, and out to the overflow output of the adder module. It can be seen in the two outputs of the synthesis (check the files under the syn directory as noted before, too long to screenshot into this report) that there are far fewer components for the signal to go through on this path when using the conditional sum adder, as the conditional sum adder reduces the propagation delay to O(Log(N)) where N is the number of bits in the input data (16 in our case) whereas a ripple carry adder has a delay of O(N) due to the carry going through each full adder. I did not find that the critical path could be optimized much beyond improving the adder.

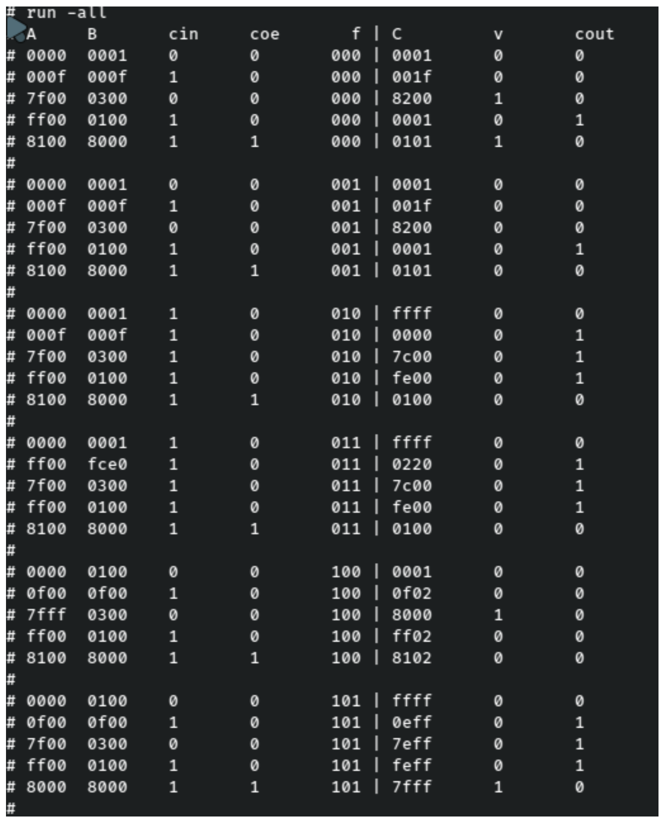
The conditional sum adder was first done via recursive modules, but was then basically rewritten by hand because the synthesis tool does not support parameters. I am, however, quite proud of my work on the recursive version which can be seen in the test.v file if you want to take a look.

The output of the adder testbench (adder_tb) using the conditional sum adder can be seen as follows:



```
# Loading work.full_adder
# run -all
# A     B       cin     coe       f | C         v       cout
# 0000  0001    0       0       000 | 0001      0       0
# 000f  000f    1       0       000 | 001f      0       0
# 7f00  0300    0       0       000 | 8200      1       0
# ff00  0100    1       0       000 | 0001      0       1
# 8100  8000    1       1       000 | 0101      1       0
#
# 0000  0001    0       0       001 | 0001      0       0
# 000f  000f    1       0       001 | 001f      0       0
# 7f00  0300    0       0       001 | 8200      0       0
# ff00  0100    1       0       001 | 0001      0       1
# 8100  8000    1       1       001 | 0101      0       0
#
# 0000  0001    1       0       010 | ffff      0       0
# 000f  000f    1       0       010 | 0000      0       1
# 7f00  0300    1       0       010 | 7c00      0       1
# ff00  0100    1       0       010 | fe00      0       1
# 8100  8000    1       1       010 | 0100      0       0
#
# 0000  0001    1       0       011 | ffff      0       0
# ff00  fce0    1       0       011 | 0220      0       1
# 7f00  0300    1       0       011 | 7c00      0       1
# ff00  0100    1       0       011 | fe00      0       1
# 8100  8000    1       1       011 | 0100      0       0
#
# 0000  0100    0       0       100 | 0001      0       0
# 0f00  0f00    1       0       100 | 0f02      0       0
# 7fff  0300    0       0       100 | 8000      1       0
# ff00  0100    1       0       100 | ff02      0       0
# 8100  8000    1       1       100 | 8102      0       0
#
# 0000  0100    0       0       101 | ffff      0       0
# 0f00  0f00    1       0       101 | 0eff      0       1
# 7f00  0300    0       0       101 | 7eff      0       1
# ff00  0100    1       0       101 | feff      0       1
# 8000  8000    1       1       101 | 7fff      1       0
#
```

Here is the output of the same test bench with the ripple carry:

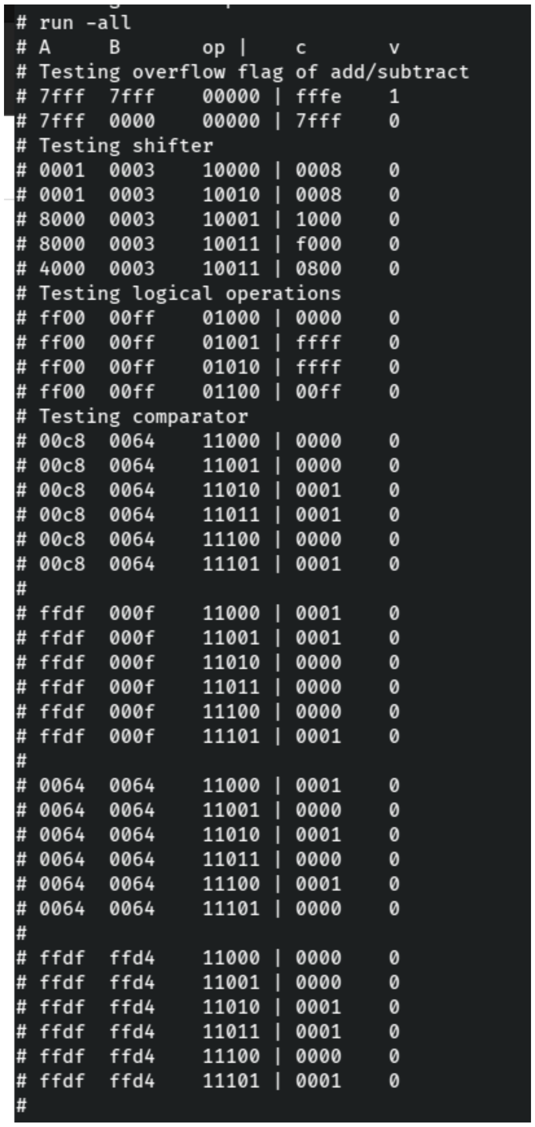


```
# run -all
 A      B       cin      coe        f | C          v        cout
# 0000  0001    0        0        000 | 0001       0        0
# 000f  000f    1        0        000 | 001f       0        0
# 7f00  0300    0        0        000 | 8200       1        0
# ff00  0100    1        0        000 | 0001       0        1
# 8100  8000    1        1        000 | 0101       1        0
#
# 0000  0001    0        0        001 | 0001       0        0
# 000f  000f    1        0        001 | 001f       0        0
# 7f00  0300    0        0        001 | 8200       0        0
# ff00  0100    1        0        001 | 0001       0        1
# 8100  8000    1        1        001 | 0101       0        0
#
# 0000  0001    1        0        010 | ffff       0        0
# 000f  000f    1        0        010 | 0000       0        1
# 7f00  0300    1        0        010 | 7c00       0        1
# ff00  0100    1        0        010 | fe00       0        1
# 8100  8000    1        1        010 | 0100       0        0
#
# 0000  0001    1        0        011 | ffff       0        0
# ff00  fce0    1        0        011 | 0220       0        1
# 7f00  0300    1        0        011 | 7c00       0        1
# ff00  0100    1        0        011 | fe00       0        1
# 8100  8000    1        1        011 | 0100       0        0
#
# 0000  0100    0        0        100 | 0001       0        0
# 0f00  0f00    1        0        100 | 0f02       0        0
# 7fff  0300    0        0        100 | 8000       1        0
# ff00  0100    1        0        100 | ff02       0        0
# 8100  8000    1        1        100 | 8102       0        0
#
# 0000  0100    0        0        101 | ffff       0        0
# 0f00  0f00    1        0        101 | 0eff       0        1
# 7f00  0300    0        0        101 | 7eff       0        1
# ff00  0100    1        0        101 | feff       0        1
# 8000  8000    1        1        101 | 7fff       1        0
#
```

It can be seen that the output of each matches up, and both are equal to the table presented in the homework document. Therefore I believe that the functionality of both of the adders is correct.

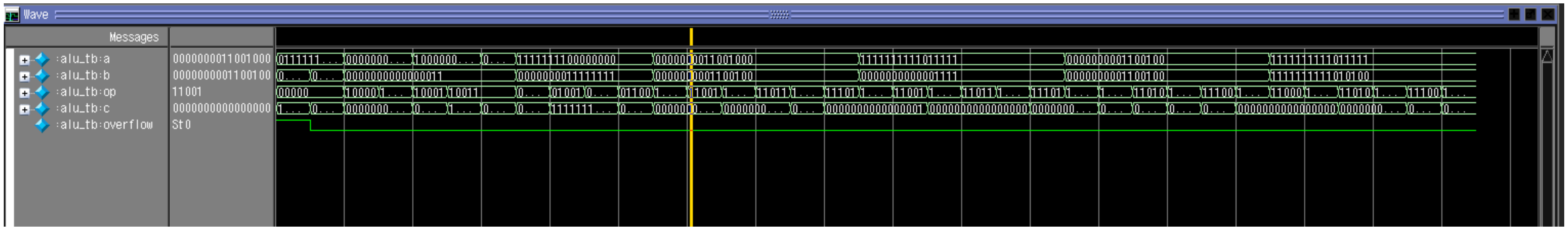
The logic unit simply completes all 4 operations (and, or , xor, not A) in parallel and then muxes between them. Each bit will only need to pass through 1 gate plus a mux, thus the delay in this module is quite minimal.

The barrel shifter simply uses 4 layers of muxes to shift by a power of 2 in each stage corresponding to the lowest 4 bits of the B input. In the worst case each bit will have to travel through 6 muxes (4 shift layers plus two inversion muxes (these allow shifting in either direction with minimal hardware)). Thus the delay of this module is quite minimal as well. The comparator unit checks for equality and for a greater than b and calculates the rest of the values based on that, then muxes between the different comparisons based on the op input. The output of the alu testbench (alu_tb.v) can be seen below as well as the waveform.



```
# run -all
# A     B       op |    c       v
# Testing overflow flag of add/subtract
# 7fff  7fff    00000 | fffe    1
# 7fff  0000    00000 | 7fff    0
# Testing shifter
# 0001  0003    10000 | 0008    0
# 0001  0003    10010 | 0008    0
# 8000  0003    10001 | 1000    0
# 8000  0003    10011 | f000    0
# 4000  0003    10011 | 0800    0
# Testing logical operations
# ff00  00ff    01000 | 0000    0
# ff00  00ff    01001 | ffff    0
# ff00  00ff    01010 | ffff    0
# ff00  00ff    01100 | 00ff    0
# Testing comparator
# 00c8  0064    11000 | 0000    0
# 00c8  0064    11001 | 0000    0
# 00c8  0064    11010 | 0001    0
# 00c8  0064    11011 | 0001    0
# 00c8  0064    11100 | 0000    0
# 00c8  0064    11101 | 0001    0
#
# ffdf  000f    11000 | 0001    0
# ffdf  000f    11001 | 0001    0
# ffdf  000f    11010 | 0000    0
# ffdf  000f    11011 | 0000    0
# ffdf  000f    11100 | 0000    0
# ffdf  000f    11101 | 0001    0
#
# 0064  0064    11000 | 0001    0
# 0064  0064    11001 | 0000    0
# 0064  0064    11010 | 0001    0
# 0064  0064    11011 | 0000    0
# 0064  0064    11100 | 0001    0
# 0064  0064    11101 | 0000    0
#
# ffdf  ffd4    11000 | 0000    0
# ffdf  ffd4    11001 | 0000    0
# ffdf  ffd4    11010 | 0001    0
# ffdf  ffd4    11011 | 0001    0
# ffdf  ffd4    11100 | 0000    0
# ffdf  ffd4    11101 | 0001    0
#
```

All the outputs appear to produce the correct values.



```
:alu_tb:a          0000000011001000   0111111... 0000000... 1000000... 0... 1111111100000000 0000000011001000 1111111111011111 0000000001100100 1111111111011111
:alu_tb:b          0000000001100100   0... 0... 0000000000000011 0000000011111111 0000000001100100 0000000000001111 0000000001100100 1111111111010100
:alu_tb:op         11001              00000 10000 1... 10001 10011 0... 01001 0... 01100 1... 11001 1... 11011 1... 11101 1... 11001 1... 11011 1... 11101 1... 1... 11010 1... 11100 1... 11000 1... 11010 1... 11100 1...
:alu_tb:c          0000000000000000   1... 0... 0000000... 0... 1... 0... 0... 1111111... 0... 0000000... 0000000... 0... 0000000000000001 0000000000000000 0000000... 0... 0... 0... 0... 0000000000000000 0000000... 0... 0...
:alu_tb:overflow   St0
```

Problem 2:

Files:

- freecellPlayer.v - The main file of the design, contains all game state
- testFreecell.v - The provided test bench with minimal modifications (naming)
- Freecell_notation.v - Contains macro defines that define the value of each card
- Card_stack.v - Contains a module that represents a stack of cards. Essentially a LIFO queue
- Decoder.v - contains 2-to-4 and 3-to-8 decoder modules.
- Freecell_logic.v - contains the freecell_logic module, handles move validation
- Order_validator.v - contains the *order_validator* module. Checks whether the source card can be moved on top of the destination stack, submodule of freecell_logic

Design:

I decided to represent each card as a set of 6 bits, the 2 most significant bits represent the 4 suites and 4 bits represent the 13 possible values (1-10, J-A). The game of freecell is made up of a number of stacks of cards. There are 8 stacks in the tableau with theoretically unlimited capacity (although only should go up to 22 in practice), 4 free cells with a limit of 1 card, and 4 home cells that can have up to 13 cards (all cards in the same suite). Given that all of these stacks are equivalent in functionality besides starting state and maximum size, it made sense to me to create a module that represents a single stack of cards. This can be seen in the *card_stack* module in the card_stack.v file. On the most basic level, this module is essentially a set of N 6-bit registers representing the cards contained in the stack with a counter keeping track of the number of cards currently in the stack. The top card in the stack is left as an output of the module, since the main game can only access that card at any given time. The push and pop inputs decide whether a card will be added or removed from the stack on any given clock edge. When the push input is high, the card on the *card_in* input will be loaded into the next available register and the counter will be incremented. When pop is high, the counter is simply decremented, automatically updating the top card. Cards cannot be pushed when the stack is full. Cards cannot be popped when the stack is empty.

On each positive edge of the clock, one card move will be attempted (could be invalid). If the operation is valid, the selected source card will be popped from its stack and pushed onto the destination stack. Because everything is done on the positive edge of the clock, there should be no latches within the design, only flip-flops. All functionality besides the registers/counters within the card stacks will be combinational logic.

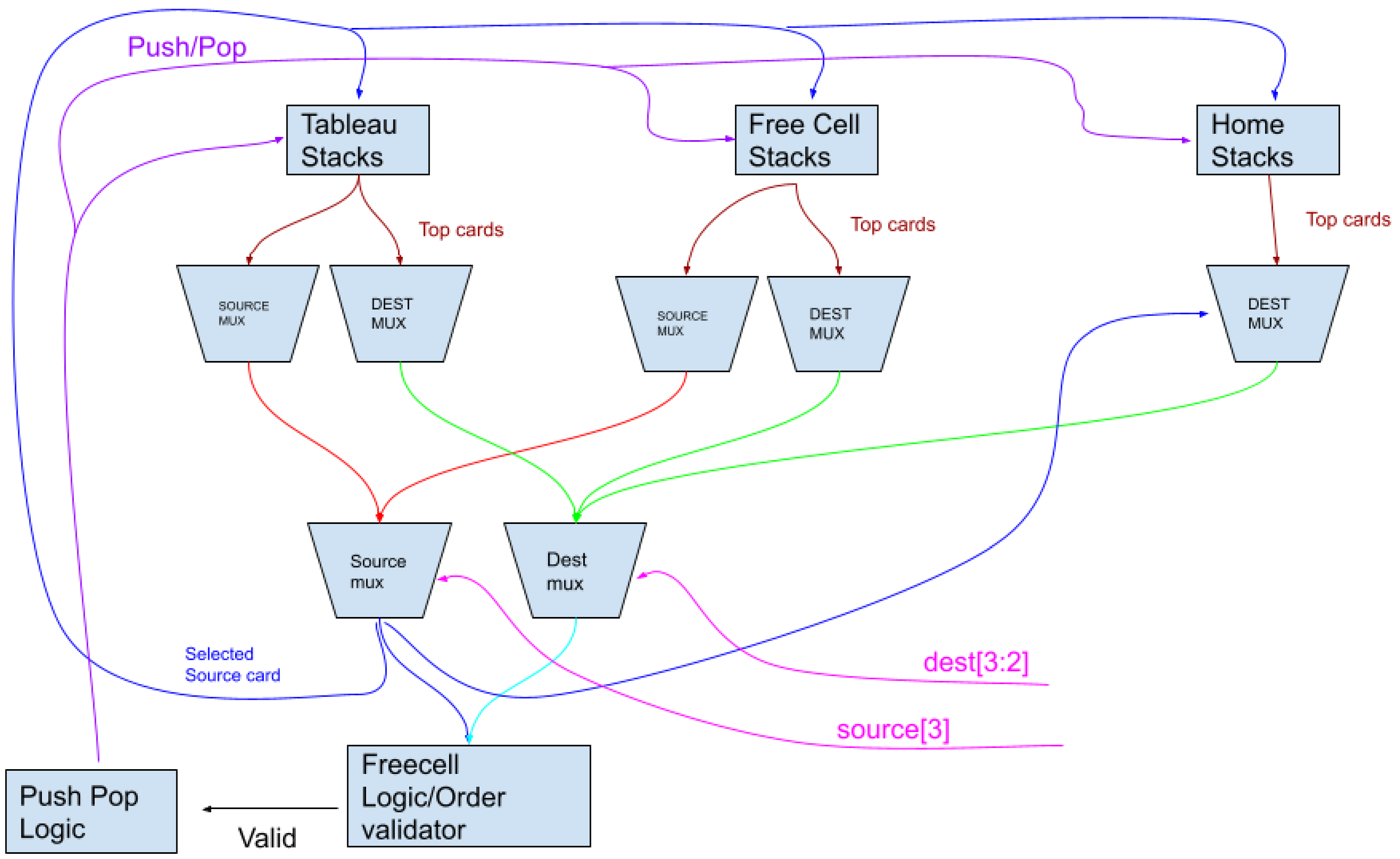
The selected source card and the top card of the destination stack (needed to validate a move) need to be selected for every set of inputs. Same goes for various other lines needed for the validator logic. The way I achieved this was by mixing between all of the top cards of all of the categories of stacks (tableau, home, free) and then muxing between the chosen cards of each of those categories. A diagram of this can be seen in the diagram on the next page. Decoders were used to decode the lowest 2-3 bits of dest and source in order to make operations for push and pop easier. This allowed me to do bitwise operations on a full packed array instead of on individual bits. This is easier to explain by looking at the code than it is for me to explain it.

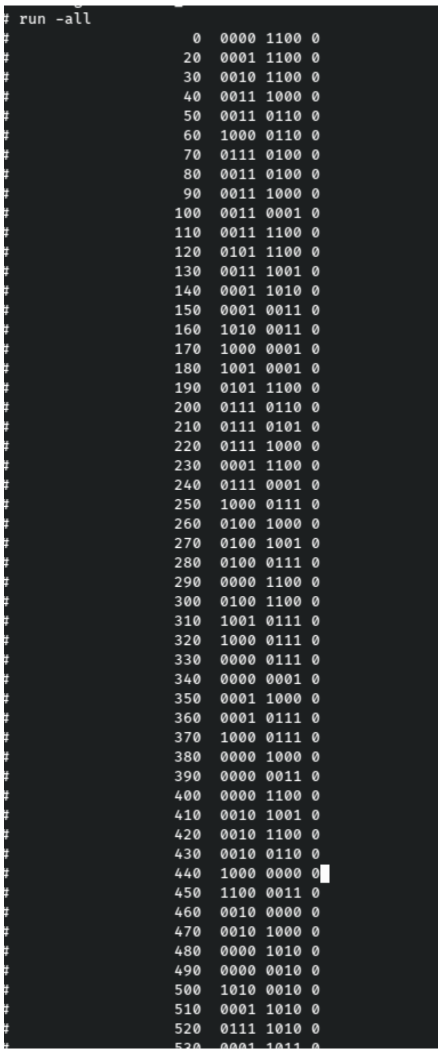
The freecell_logic and order_validator modules validate the move currently trying to be made. They take the src and dest inputs, the selected source card, the selected destination card, and whether or not the source/dest stacks are empty/full in order to determine whether a move is valid or not. A move can be invalid if any of the following conditions are true:

1. Source address = destination address (not really invalid, but it's the simplest way to prevent the move.
2. Source address is from home stack
3. Source card cannot be placed onto destination stack (by rules of game)
4. Source stack is empty
5. Destination stack is full

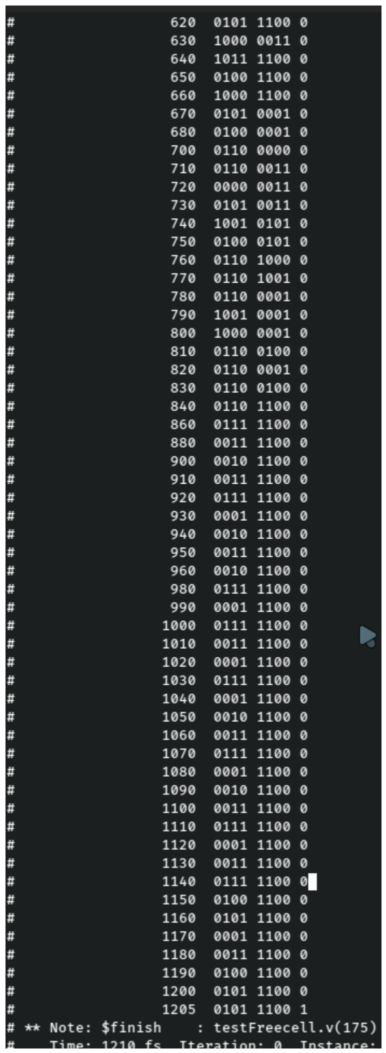
If any of those conditions are true, then the move is invalid. The valid line will be set low, which will then prevent any of the stacks from pushing or popping values.

The output of the testbench provided by Dr. Saab can be seen below the figure on the next page. This shows that the design wins on the final move of the game as expected. When run in the simulator it can be seen that the home cell card stacks have all the cards of one particular suite in the correct order. Once again, there should not be any latches in this design (all combinational and flip-flop).

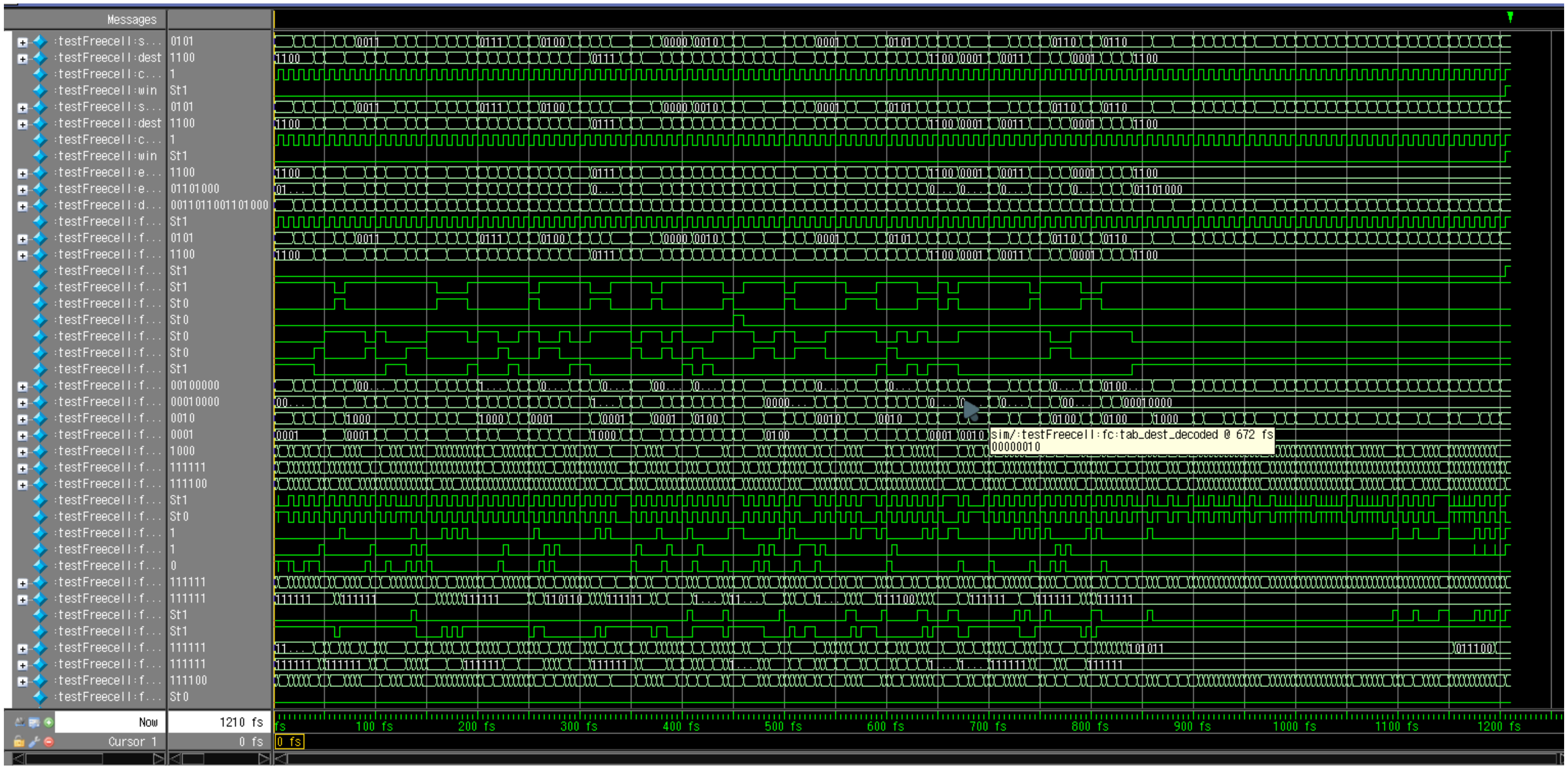

Push/Pop
Tableau Stacks
Free Cell Stacks
Home Stacks
Top cards
Top cards
Top cards
SOURCE MUX
DEST MUX
SOURCE MUX
DEST MUX
DEST MUX
Source mux
Dest mux
Selected Source card
dest[3:2]
source[3]
Push Pop Logic
Valid
Freecell Logic/Order validator



```
# run -all
#                      0  0000 1100 0
#                     20  0001 1100 0
#                     30  0010 1100 0
#                     40  0011 1000 0
#                     50  0011 0110 0
#                     60  1000 0110 0
#                     70  0111 0100 0
#                     80  0011 0100 0
#                     90  0011 1000 0
#                    100  0011 0001 0
#                    110  0011 1100 0
#                    120  0101 1100 0
#                    130  0011 1001 0
#                    140  0001 1010 0
#                    150  0001 0011 0
#                    160  1010 0011 0
#                    170  1000 0001 0
#                    180  1001 0001 0
#                    190  0101 1100 0
#                    200  0111 0110 0
#                    210  0111 0101 0
#                    220  0111 1000 0
#                    230  0001 1100 0
#                    240  0111 0001 0
#                    250  1000 0111 0
#                    260  0100 1000 0
#                    270  0100 1001 0
#                    280  0100 0111 0
#                    290  0000 1100 0
#                    300  0100 1100 0
#                    310  1001 0111 0
#                    320  1000 0111 0
#                    330  0000 0111 0
#                    340  0000 0001 0
#                    350  0001 1000 0
#                    360  0001 0111 0
#                    370  1000 0111 0
#                    380  0000 1000 0
#                    390  0000 0011 0
#                    400  0000 1100 0
#                    410  0010 1001 0
#                    420  0010 1100 0
#                    430  0010 0110 0
#                    440  1000 0000 0
#                    450  1100 0011 0
#                    460  0010 0000 0
#                    470  0010 1000 0
#                    480  0000 1010 0
#                    490  0000 0010 0
#                    500  1010 0010 0
#                    510  0001 1010 0
#                    520  0111 1010 0
#                    530  0001 1011 0
```



```
#                    620  0101 1100 0
#                    630  1000 0011 0
#                    640  1011 1100 0
#                    650  0100 1100 0
#                    660  1000 1100 0
#                    670  0101 0001 0
#                    680  0100 0001 0
#                    700  0110 0000 0
#                    710  0110 0011 0
#                    720  0000 0011 0
#                    730  0101 0011 0
#                    740  1001 0101 0
#                    750  0100 0101 0
#                    760  0110 1000 0
#                    770  0110 1001 0
#                    780  0110 0001 0
#                    790  1001 0001 0
#                    800  1000 0001 0
#                    810  0110 0100 0
#                    820  0110 0001 0
#                    830  0110 0100 0
#                    840  0110 1100 0
#                    860  0111 1100 0
#                    880  0011 1100 0
#                    900  0010 1100 0
#                    910  0011 1100 0
#                    920  0111 1100 0
#                    930  0001 1100 0
#                    940  0010 1100 0
#                    950  0011 1100 0
#                    960  0010 1100 0
#                    980  0111 1100 0
#                    990  0001 1100 0
#                   1000  0111 1100 0
#                   1010  0011 1100 0
#                   1020  0001 1100 0
#                   1030  0111 1100 0
#                   1040  0001 1100 0
#                   1050  0010 1100 0
#                   1060  0011 1100 0
#                   1070  0111 1100 0
#                   1080  0001 1100 0
#                   1090  0010 1100 0
#                   1100  0011 1100 0
#                   1110  0111 1100 0
#                   1120  0001 1100 0
#                   1130  0011 1100 0
#                   1140  0111 1100 0
#                   1150  0100 1100 0
#                   1160  0101 1100 0
#                   1170  0001 1100 0
#                   1180  0011 1100 0
#                   1190  0100 1100 0
#                   1200  0101 1100 0
#                   1205  0101 1100 1
# ** Note: $finish    : testFreecell.v(175)
#    Time: 1210 fs  Iteration: 0  Instance:
```



```
Messages
:testFreecell:s...   0101
:testFreecell:dest   1100
:testFreecell:c...   1
:testFreecell:win    St1
:testFreecell:s...   0101
:testFreecell:dest   1100
:testFreecell:c...   1
:testFreecell:win    St1
:testFreecell:e...   1100
:testFreecell:e...   01101000
:testFreecell:d...   0011011001101000
:testFreecell:f...   St1
:testFreecell:f...   0101
:testFreecell:f...   1100
:testFreecell:f...   St1
:testFreecell:f...   St1
:testFreecell:f...   St0
:testFreecell:f...   St0
:testFreecell:f...   St0
:testFreecell:f...   St0
:testFreecell:f...   St1
:testFreecell:f...   00100000
:testFreecell:f...   00010000
:testFreecell:f...   0010
:testFreecell:f...   0001
:testFreecell:f...   1000
:testFreecell:f...   111111
:testFreecell:f...   111100
:testFreecell:f...   St1
:testFreecell:f...   St0
:testFreecell:f...   1
:testFreecell:f...   1
:testFreecell:f...   0
:testFreecell:f...   111111
:testFreecell:f...   111111
:testFreecell:f...   St1
:testFreecell:f...   St1
:testFreecell:f...   111111
:testFreecell:f...   111111
:testFreecell:f...   111100
:testFreecell:f...   St0

Now        1210 fs
Cursor 1      0 fs

sim/:testFreecell:fc:tab_dest_decoded @ 672 fs
00000010
```