ECSE 222 Digital Logic
Lab Assignment I

Group 62

Antoine Wang 260766084

Yicheng Song 260763294

Section 1: Description of 7-segment Decoder

The purpose of the 7-segment decoder is to convert binary number (4-bit) into a 7-bit representation which can be used to trigger the LED light. Since the LED light pattern is defined as a different set of binary combinations, certain conversion must be applied, which is handled in the decoder circuit.

```
architecture Display of decoder is
begin
    process(segmentcode)
    begin
        case segmentcode is -- List all the possi
            when "0000" => segments <= "1000000";
            when "0001" => segments <= "1111001";
            when "0010" => segments <= "0100100";
            when "0011" => segments <= "0110000";
            when "0100" => segments <= "0011001";
            when "0101" => segments <= "0010010";
            when "0110" => segments <= "0000010";
            when "0111" => segments <= "1111000";
            when "1000" => segments <= "0000000";
            when "1001" => segments <= "0010000";
            when "1010" => segments <= "0001000";
            when "1011" => segments <= "0000011";
            when "1100" => segments <= "1000110";
            when "1101" => segments <= "0100001";
            when "1110" => segments <= "0000110";
            when "1111" => segments <= "0001110";
            when others => segments <= "1111111";
        end case;
    end process;
end Display;
```

Figure 1. The implementation of 7-segment decoder using selected assignment

As figure 1.1 illustrates, we implemented our decoder in a selecting manner since we believe that the choice of a selected signal assignment would minimize the cost of the circuit. It will also minimize the length of critical path. The selected assignment basically works as a multiplexer where 4-bit binary number are the control bit and a straightforward decision is made if any case matches with the input. On the other hand, for the conditional signal assignment, the input 4-bit binary representation needs to propagate a series of "when" clause until it finally finds a case that matches. Then an output can be generated. In other words, we anticipate that selection gives the decoder a O(1) complexity while conditional method defines O(n) complexity.

Section 2: Testing of 7-segment Decoder

We test the 7-segment decoder circuit by using test bench generated by Quartus from the VHDL code and simulate it through ModelSim.

After compiling the code of 7-segment decoder circuit on Quartus, we use the test bench to generate test inputs for simulation. A FOR LOOP is used to increment the value of input signals in the loop and generate totally 16 different inputs: 2 different condition for each of the 4 bits, which represent from 0 to 15 in decimal, or from 0 to F in hexadecimal.

```
BEGIN

        FOR i in 0 to 15 loop
        segmentcode <= std_logic_vector (to_unsigned(i,4));
        wait for 10 ns;
        end loop;-- code executes for every event on sensitivity list
WAIT;
```

Figure 1. FOR LOOP for generating inputs of 7-segement decoder test

Then we import the 7-segment decoder code and the test bench into ModelSim and compile. By dragging signals from the "Objects" window to the "Wave" window and running the simulation, we can check the simulated output and test the circuit. The simulation plot is illustrated by figure 2.1 below. Full diagram will be presented in Section Appendix.

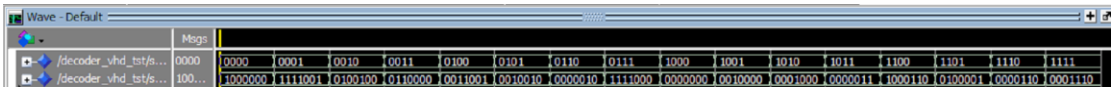


Figure 2. All 16 inputs with their relative outputs simulated by ModelSim

We can make sure that the circuit works correctly by checking the output simulated by ModelSim with our expected output. For instance, with the input set as '1000' which represent 8 in hexadecimal, 7 segments of the LED should be all on. As the LEDs on DE1 board are active-low, which means the segment is on with output '0' and is off with output '1', the output of input '1000' should be '0000000'.

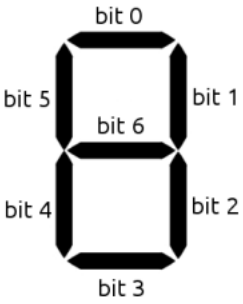
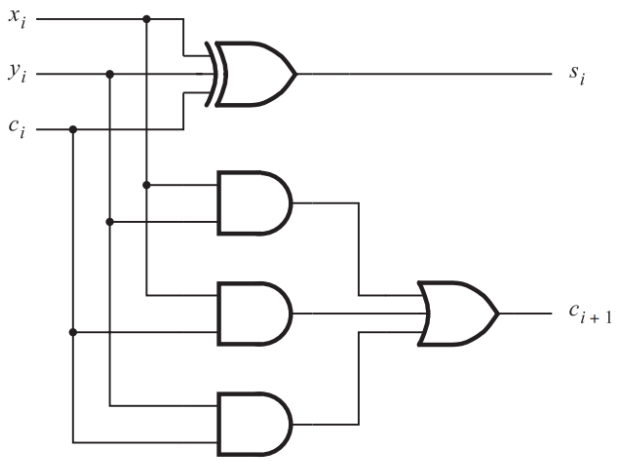


Figure 3. LED display of '8' in hexadecimal with input '1000'

If all the 16 pairs of input and output are the same with our expected ones, we can conclude that the code works correctly.

Section 3: Description of the Adder Circuit

The adder circuit includes a full adder and a 5-bit adder. Full adder simply calculates the sum as well as the carry-out with those two given inputs and the carry-in by using the mechanism we learned:




```
architecture Func of fulladd is
   begin
      S     <= A XOR B XOR Cin; -- logic function for sum bit S
      Cout  <= (A AND B) OR (Cin AND A) OR (Cin AND B); -- Logic function for carry out bit
   end Func;
```

Figure 1&2. Mechanism and algorithm of full adder circuit

Taking A, B, and Cin as inputs, S and Cout as outputs.

```
entity fulladd is -- Declare the entity title for a one bit full adder
   port (Cin, A, B   : in  std_logic;  -- Input: A,B are two operands for addition and Cin is the carry-in
         S, Cout     : out std_logic); -- Output: S is the sum bit and Cout is the carry out bit
end fulladd;
```

Figure 3. Declaration of inputs and outputs of full adder circuit.

The 5-bit adder is based on full adder. At first, we claim the inputs and outputs, with the two 5-bit binary input signals and the 5-bit binary output in terms of vector together with carry-in and carry-out in terms of single bit.

```
entity bit5adder is
   port (A, B  : in  std_logic_vector(4 downto 0); -- Input: Two 5-bit binary representation as operands
         Cin   : in  std_logic;                    -- Input: a 1-bit binary bit as carry in
         S     : out std_logic_vector(4 downto 0); -- Output: A 5-bit binary representation as sum
         Cout  : out std_logic);                   -- Output: A 1-bit binary representation as carry-out.
   end bit5adder;                                  --         Sum and carryout makes up a integrated 6 bit sum
```

Figure 4. Declaration of inputs and outputs of 5-bit adder.

Then 5 stages are claimed for calculating the 5-bit outputs and the carry-out, each stage calculate one bit of the output result and the last stage, stage 4, also give the value of carry-out.

```
begin
    stage0 : fulladd port map (Cin,          A(0), B(0), S(0), c_internal(0));
    stage1 : fulladd port map (c_internal(0), A(1), B(1), S(1), c_internal(1));
    stage2 : fulladd port map (c_internal(1), A(2), B(2), S(2), c_internal(2));
    stage3 : fulladd port map (c_internal(2), A(3), B(3), S(3), c_internal(3));
    stage4 : fulladd port map (Cin  => c_internal(3), -- Take c-internal(3) as t
                               A    => A(4),          -- Take A(4) as the oprand
                               B    => B(4),          -- Take B(4) as the oprand
                               S    => S(4),          -- Output is the MSB of th
                               Cout => Cout);         -- Output: carry-out.
end structure;
```

Figure 5. 5 stages for calculating the result.

Carry-out from previous stage is considered as the carry-in of next stage. For example, the carry-out of stage in the port map is c_internal(0), then the carry-in for stage 1 is c-internal(0). Carry-out of stage 4 is assigned to Cout as the carry-out of 5-bit adder. It is added to the front

of the 5-bit output to generate a 6-bit binary number, which is the sum of those two 5-bit binary numbers A and B.

There are 6 decoder instances used in the design. We expand those two 5-bit input and the 6-bit output to 8 bits with adding 0s from the left of the MSB. The 8-bit binary number can be represented as hexadecimal numbers by taking every 4 bits as a digit. So, we firstly take 4 bits from the LSB of two 5-bit operand inputs to get the digit on the LSB represented in hexadecimal.

```
AFront <= A(3 downto 0);
BFront <= B(3 downto 0);
```

Figure 6&7. Partition of the two 5-bit operands

Then we add '000' at the front of the remaining bit to represent the MSB of those two input operands.

```
ABack  <= "000" &  A(4);
BBack  <= "000" &  B(4);
```

Figure 8&9. Formation of the 4-bit binary number.

After calculating the result of the addition, we got a 6-bit binary number and we also need to expand it to 8 bits in order to implement the partition.

```
G5: bit5adder port map (A,B,'0',S_in,Cout_in);
result <= "00"& Cout_in & S_in;

resultFront <= result(3 downto 0);
resultBack  <= result(7 downto 4);
```

Figure 10. Expansion and partition of the 6-bit output.

Every 4-bit binary number represents a digit in hexadecimal so that every 4-bit binary number is linked to a decoder instance. As a result, there are totally 6 instances used: three 8-bit number and 2 instances for each.

```
G1: decoder port map (AFront ,decoded_A(6 downto 0));
G2: decoder port map (ABack ,decoded_A(13 downto 7));
G3: decoder port map (BFront ,decoded_B(6 downto 0));
G4: decoder port map (BBack ,decoded_B(13 downto 7));

G6: decoder port map (resultFront ,decoded_AplusB(6 downto 0));
G7: decoder port map (resultBack ,decoded_AplusB(13 downto 7));
```

Figure 11. Decode all 6 instances by using decoder we built earlier.

Section 4: Testing of the Adder Circuit

Two methods are used to test the adder circuit: testing on the hardware and modelsim simulation. The most direction way is to map the inputs and outputs on to the DEC board. By regulating the switches manually, the LED light segments should arrange into familiar patterns which corresponds to the hexa-decimal representation that fits the common sense. Figure 4.1 below illustrates one specific testing on the board.

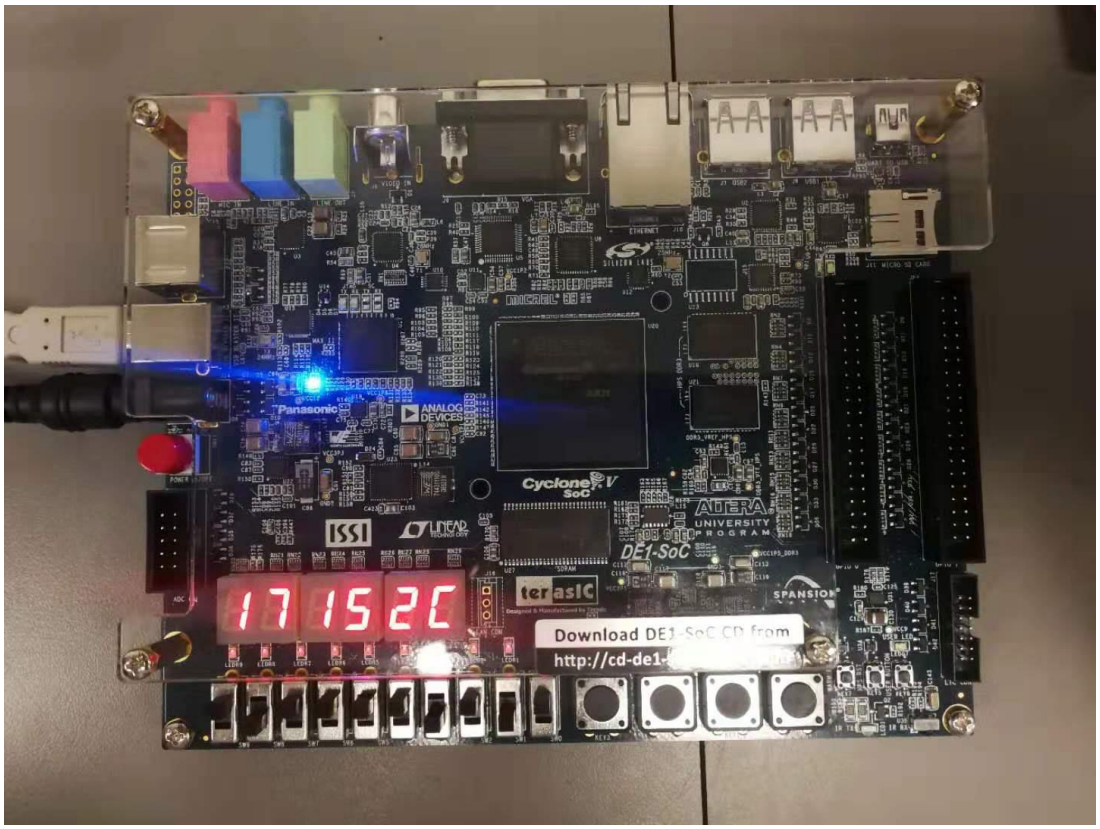


Figure 1. One of the tests that we performed on the board (succeeded)

The advantage of this test is that it is fast and repeatable for the testers to keep change the switch combinations to get different readings, However, the shortcoming is also significant. The test result is lack of generality since manually only a certain number of samples can be tested. In addition, in case that bugs exist, or the circuit performs unexpectedly because of hardware problem, there is no obvious reason which can be analyzed thought direct observation. An example is that our group encountered a technical issue when operating the board. The fifth switch (sw4) is constantly giving 1 value even it is positioned at 0. The problem will only be resolved when an external force is exerted onto the switch, pulling it down to correct the flawed connection. This simple issue initially is time-consuming to solve since by simply observing the LED patterns, it seems like an algorithmic or a mapping issue.

Later, to counter potential bugs we implemented the test bench for our adder circuit using Quartus Testing Bench Writer, like how the decoder is tested, a template is generated. Since the addition in total provides 1024 combinations (32*32) and it would be too extensive to present them all on the wave diagram, what we used instead is the sample test. In the loop we defined in the test bench, we fix operand B and iterate A from 0 to 31. By examining the 32 different patterns we can conclude that the adder circuit is free from errors if all the outputs matches the expected results. If inconsistency exists, we modify B to another constant and perform a new sample test.

Below (Figure 4.2) is a screenshot taken for one of the tests with extreme condition (B = 11111). The full screenshot is in Section Appendix. Note that the outputs, the representations of A, B and sum are all LED light segment format. Since each value uses 2 LED lights, a total of 14 bits are used, partitioned as bit 0-6 and 7-13 for each number on the LED light.

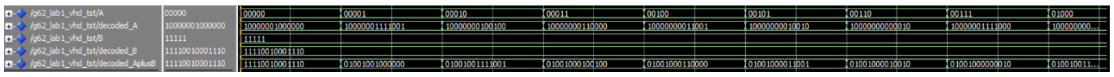
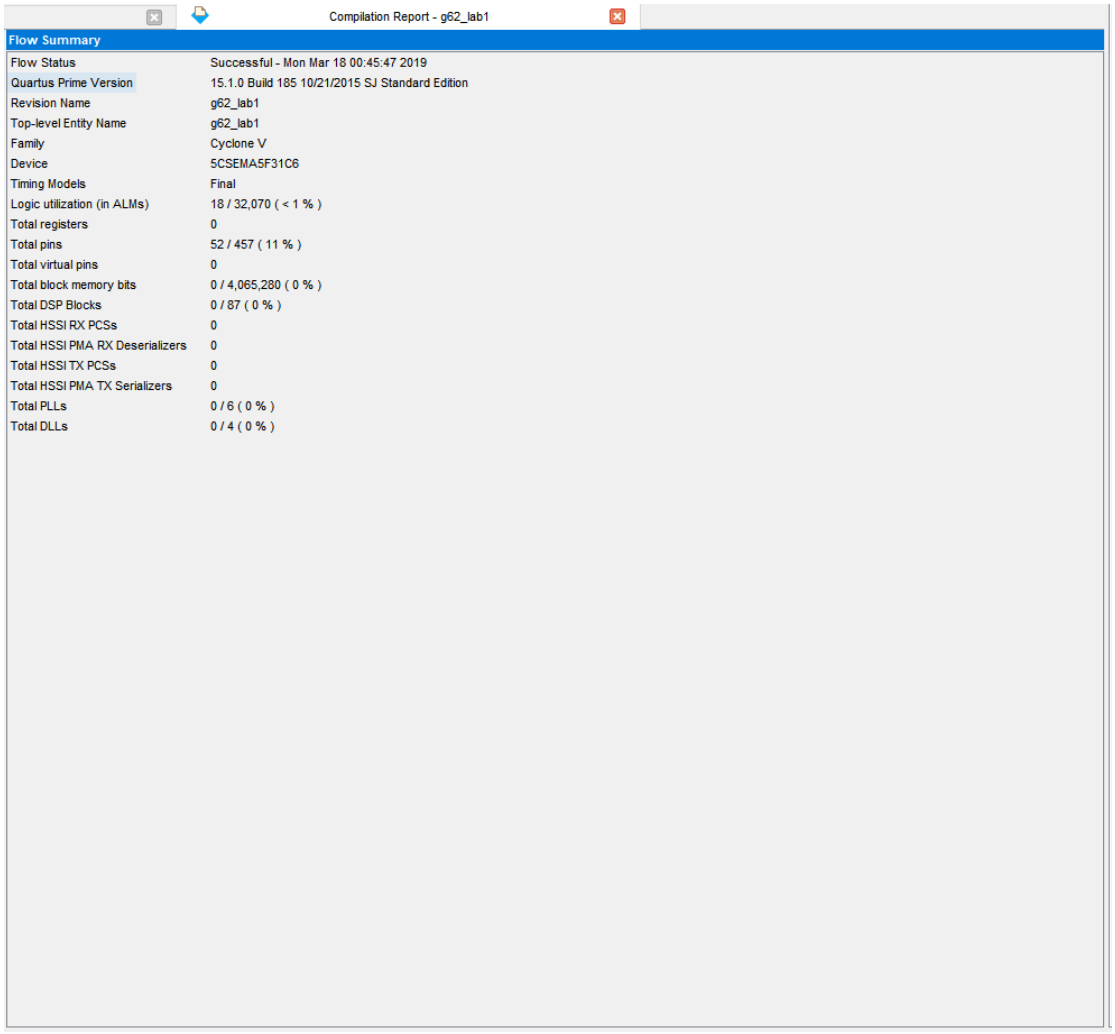


Figure 2. Part of input and output pairs of adder circuit simulated by ModelSim (B = 31)

Section 5: Resource Summary and RTL Schematic Diagram

Overall, since the algorithm is not complicated with no extensive bit processing, the resources utilized is relatively insignificant. Though over 10% pins are used, it is necessary and can not be further simplified since they guarantee the functioning of the board simulation.



Compilation Report - g62_lab1

Flow Summary

Flow Status	Successful - Mon Mar 18 00:45:47 2019
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Standard Edition
Revision Name	g62_lab1
Top-level Entity Name	g62_lab1
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	18 / 32,070 (< 1 %)
Total registers	0
Total pins	52 / 457 (11 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 1. A summary of the FPGA resource utilization (from the Compilation Report's Flow Summary)

However, for algorithms we can simplify the ripple adder to reduce complexity. Below is the RTL Schematic Diagram (Figure 5.2). Since our choice of 5-bit carry-ripple adder, the corresponding circuit appears to be complicated at the adding process. Note in the diagram only one multiplexer at the decoder phase is presented as an example.

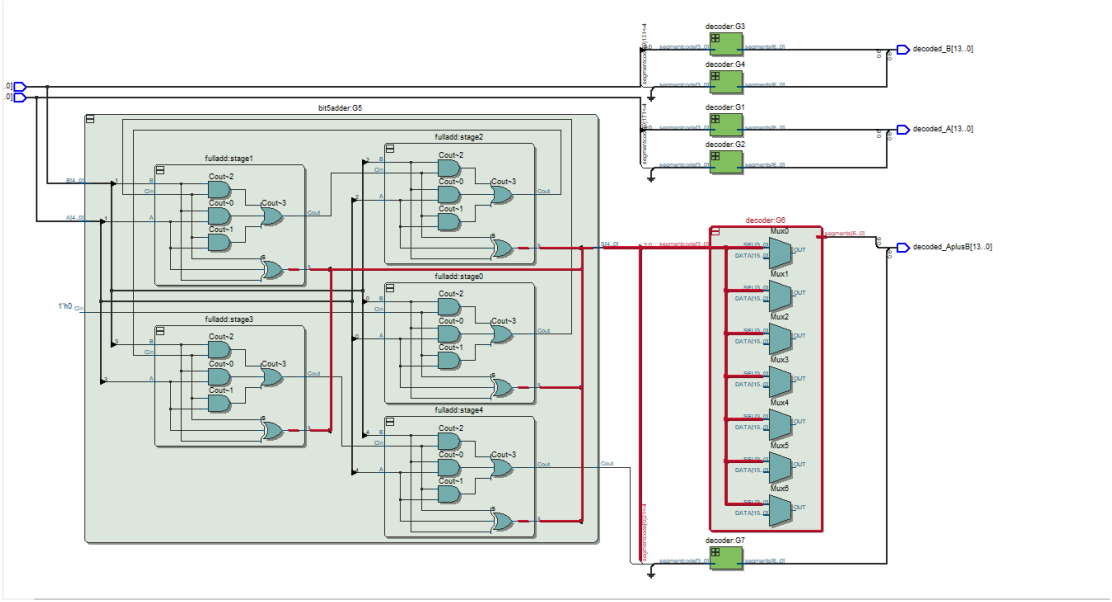


Figure 2. RTL Schematic Diagram for the Entire Adder Circuit

Code Mapping:

1. Each fulladd entity corresponds to one of the five full adder circuit.

```
architecture Func of fulladd is
   begin
      S     <= A XOR B XOR Cin; -- logic function for sum
      Cout  <= (A AND B) OR (Cin AND A) OR (Cin AND B); -
   end Func;
```

2. Stage 0-4 of fulladd entity is later defined in the bit5adder entity. Correspondence is defined in the code from stage 0 to 4.

```
stage0 : fulladd port map (Cin,          A(0), B(0), S(0), c_internal(0)); --
stage1 : fulladd port map (c_internal(0), A(1), B(1), S(1), c_internal(1)); --
stage2 : fulladd port map (c_internal(1), A(2), B(2), S(2), c_internal(2)); --
stage3 : fulladd port map (c_internal(2), A(3), B(3), S(3), c_internal(3)); --
stage4 : fulladd port map (Cin  => c_internal(3), -- Take c-internal(3) as the
                           A    => A(4),          -- Take A(4) as the oprand
                           B    => B(4),          -- Take B(4) as the oprand
                           S    => S(4),          -- Output is the MSB of the
                           Cout => Cout);         -- Output: carry-out.
```

3. In total three outputs are generated. Output decoded_A maps to decoder G1 and G2 in the diagram and decoded_B maps to decoder G3 and G4. They are presented in the upper right corner of the diagram and they do not pass the adder since they only experienced conversion from input to output.

```
G1: decoder port map (AFront ,decoded_A(6 downto 0)); -
G2: decoder port map (ABack ,decoded_A(13 downto 7));
G3: decoder port map (BFront ,decoded_B(6 downto 0)); -
G4: decoder port map (BBack ,decoded_B(13 downto 7));
```

Decoder G6 and G7 maps to the decoding port map of the sum of A and B. Case G5 is assigned to the ripple carry adder instance.

```
G5: bit5adder port map (A,B,'0',S_in,Cout_in); -- Perform the ri
result <= "00"& Cout_in & S_in;                -- Concatenate th

resultFront <= result(3 downto 0);             -- Partition the
resultBack  <= result(7 downto 4);
G6: decoder port map (resultFront ,decoded_AplusB(6 downto 0));
G7: decoder port map (resultBack ,decoded_AplusB(13 downto 7));
```

Appendix

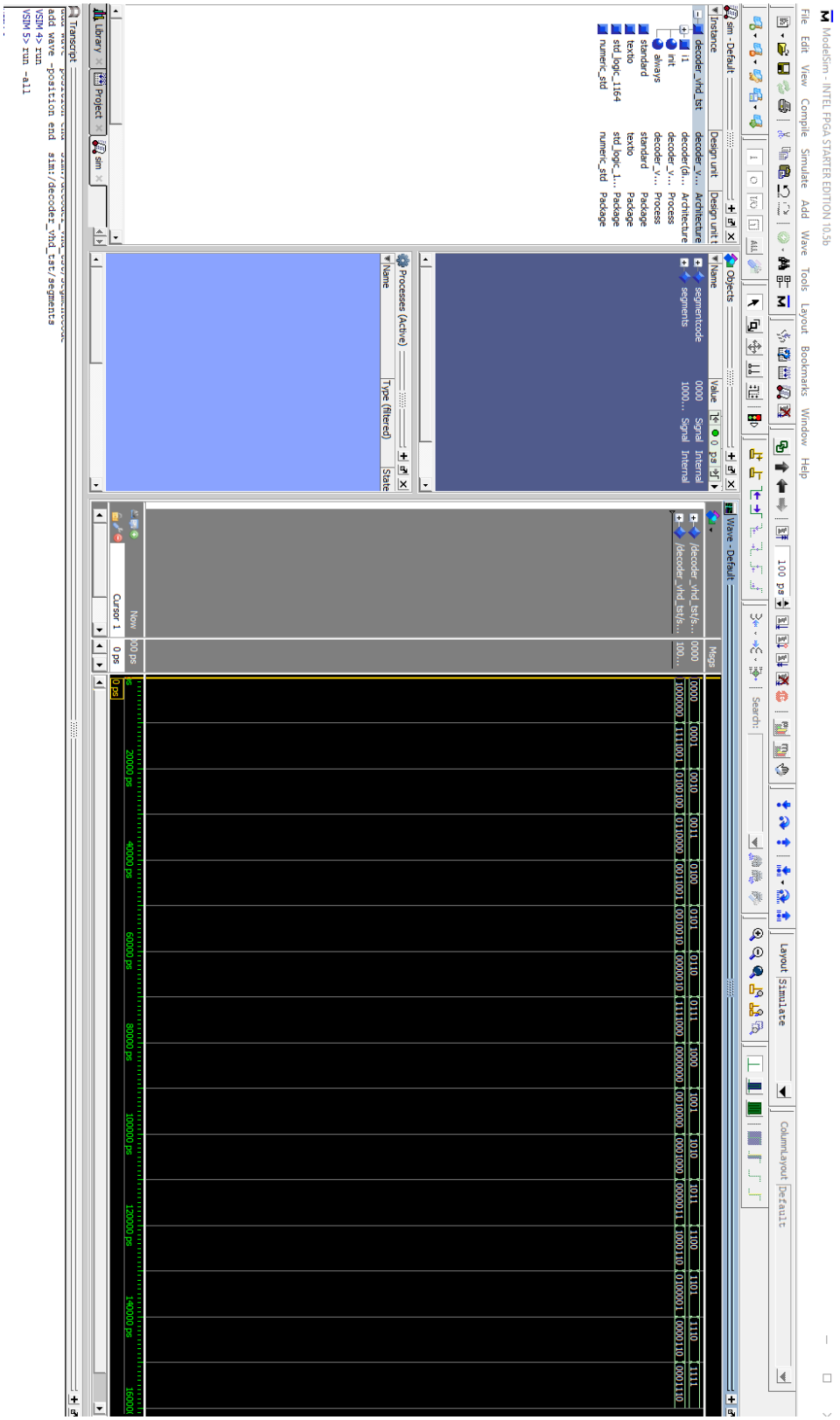


Figure 1. All 16 inputs with their relative outputs simulated by ModelSim (Full diagram of fig 2.2)

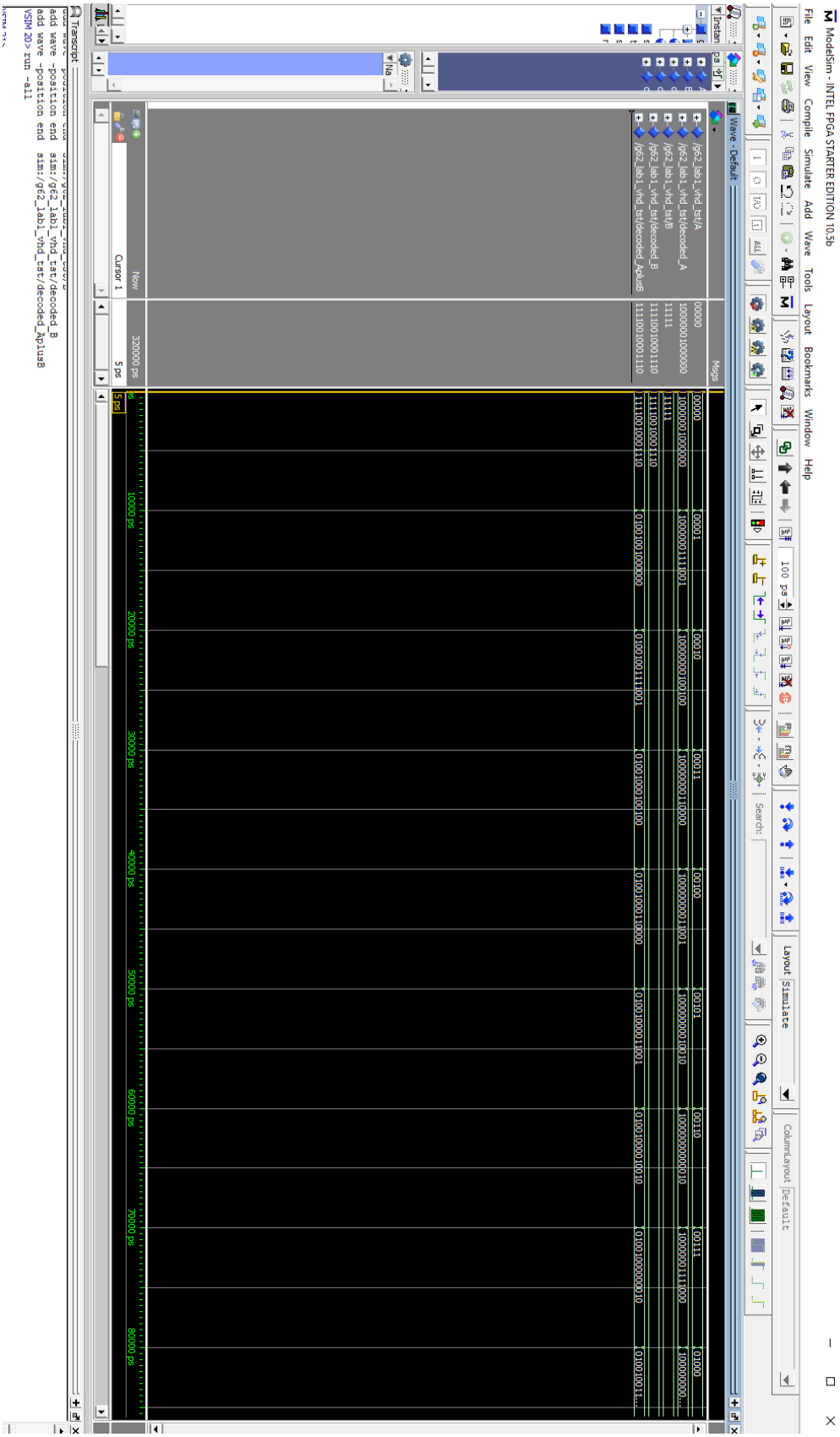


Figure 2. Part of input and output pairs of adder circuit simulated by ModelSim (B = 31) (full diagram of fig 4.2)