References on Web

PDF Resources: Designing the Hardware

[1] Laboratory Exercise 2 Numbers and Displays

[2] Translated manual for Master 21EDA board

[3] Altera DE1 Board

[4] VHDL Tutorial

Tools

[4] Digital Electronics Education Design Suite (DEEDS)

[5] LogicFriday

Contents

Introduction

The **Headings in red** in this document will mirror the headings in the Altera® tutorial Ref [1] so you can easily map between documents. You **WILL NEED** Ref [1] at least as this document is only providing the gotchas when walking through Ref [1]. Additional **Headings in blue** are internal to this document – used to break things up as you would expect headings to do.

Read the previous paragraph again. You are reading this document along with the Altera® tutorial [1].

Remember also from the blog, we are now using Quartus® II version 11.1 – driven by the chip on the board, the 144-pin EP2C5T144C8 Cyclone II. The predominant difference is the transitioning from SPOC to Qsys as system on chip designer. Both are available in 11.1, which suits us because there is a lot of free info on web for SOPC based design.

Legend:

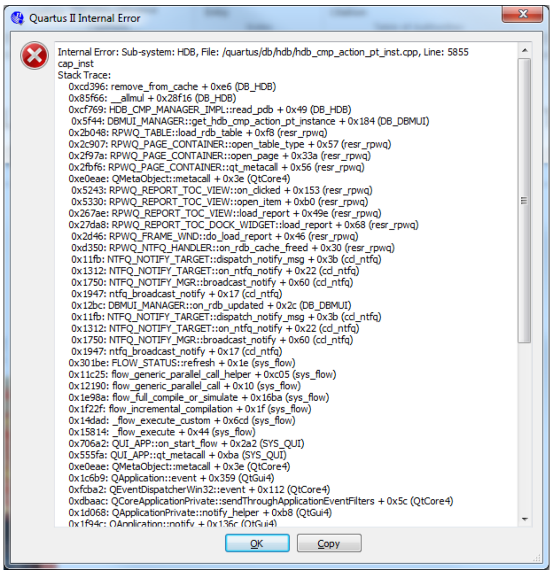
If I have been stumped by something I will use the image to the left to let you know a little investigation was in order.

If an important "Ah Ha!" moment occurred, I will also let you know.

If you're to go to the web I will give the hint.

STOP, we are swapping tutorials

Now don't forget something very important. Quartus ® II is clunky. Recall from the blog the crashing. What you will find is you may need to delete project and start again a couple of times so be prepared both spiritually and emotionally. You will find the Altera® tutorial leaves things out (which we will try to catch). You will also find, as I did, the tool may not even crash, but will not react to menu selections etc. Just take a deep breath and SCREAM, get over it and try again. Of course, that was while we were using 10.1, the switch to 11.1 may have changed that – we'll see … whoops, yes there we are (Figure 1).



```
Internal Error: Sub-system: HDB, File: /quartus/db/hdb/hdb_cmp_action_pt_inst.cpp, Line: 5855
cap_inst
Stack Trace:
   0xcd396: remove_from_cache + 0xe6 (DB_HDB)
   0x85f66: __allmul + 0x28f16 (DB_HDB)
   0xcf769: HDB_CMP_MANAGER_IMPL::read_pdb + 0x49 (DB_HDB)
    0x5f44: DBMUI_MANAGER::get_hdb_cmp_action_pt_instance + 0x184 (DB_DBMUI)
   0x2b048: RPWQ_TABLE::load_rdb_table + 0xf8 (resr_rpwq)
   0x2c907: RPWQ_PAGE_CONTAINER::open_table_type + 0x57 (resr_rpwq)
   0x2f97a: RPWQ_PAGE_CONTAINER::open_page + 0x33a (resr_rpwq)
   0x2fbf6: RPWQ_PAGE_CONTAINER::qt_metacall + 0x56 (resr_rpwq)
   0xe0eae: QMetaObject::metacall + 0x3e (QtCore4)
    0x5243: RPWQ_REPORT_TOC_VIEW::on_clicked + 0x153 (resr_rpwq)
    0x5330: RPWQ_REPORT_TOC_VIEW::open_item + 0xb0 (resr_rpwq)
   0x267ae: RPWQ_REPORT_TOC_VIEW::load_report + 0x49e (resr_rpwq)
   0x27da8: RPWQ_REPORT_TOC_DOCK_WIDGET::load_report + 0x68 (resr_rpwq)
    0x2d46: RPWQ_FRAME_WND::do_load_report + 0x46 (resr_rpwq)
    0xd350: RPWQ_NTFQ_HANDLER::on_rdb_cache_freed + 0x30 (resr_rpwq)
    0x11fb: NTFQ_NOTIFY_TARGET::dispatch_notify_msg + 0x3b (ccl_ntfq)
    0x1312: NTFQ_NOTIFY_TARGET::on_ntfq_notify + 0x22 (ccl_ntfq)
    0x1750: NTFQ_NOTIFY_MGR::broadcast_notify + 0x60 (ccl_ntfq)
    0x1947: ntfq_broadcast_notify + 0x17 (ccl_ntfq)
    0x12bc: DBMUI_MANAGER::on_rdb_updated + 0x2c (DB_DBMUI)
    0x11fb: NTFQ_NOTIFY_TARGET::dispatch_notify_msg + 0x3b (ccl_ntfq)
    0x1312: NTFQ_NOTIFY_TARGET::on_ntfq_notify + 0x22 (ccl_ntfq)
    0x1750: NTFQ_NOTIFY_MGR::broadcast_notify + 0x60 (ccl_ntfq)
    0x1947: ntfq_broadcast_notify + 0x17 (ccl_ntfq)
   0x301be: FLOW_STATUS::refresh + 0x1e (sys_flow)
   0x11c25: flow_generic_parallel_call_helper + 0xc05 (sys_flow)
   0x12190: flow_generic_parallel_call + 0x10 (sys_flow)
   0x1e98a: flow_full_compile_or_simulate + 0x16ba (sys_flow)
   0x1f22f: flow_incremental_compilation + 0x1f (sys_flow)
   0x14dad: _flow_execute_custom + 0x6cd (sys_flow)
   0x15814: _flow_execute + 0x44 (sys_flow)
   0x706a2: QUI_APP::on_start_flow + 0x2a2 (SYS_QUI)
   0x555fa: QUI_APP::qt_metacall + 0xba (SYS_QUI)
   0xe0eae: QMetaObject::metacall + 0x3e (QtCore4)
   0x1c6b9: QApplication::event + 0x359 (QtGui4)
   0xfcba2: QEventDispatcherWin32::event + 0x112 (QtCore4)
   0xdbaac: QCoreApplicationPrivate::sendThroughApplicationEventFilters + 0x5c (QtCore4)
   0x1d068: QApplicationPrivate::notify_helper + 0xb8 (QtGui4)
   0x1f94c: QApplication::notify + 0x136c (QtGui4)
```

Figure 1: Same old problem

Don't forget; as we build projects for each part of the lab remember to set unused pins.

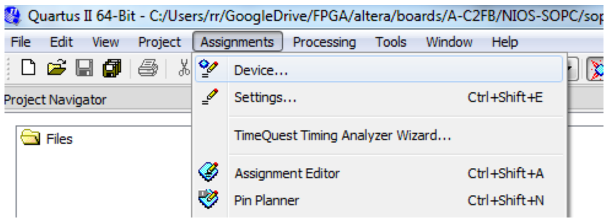
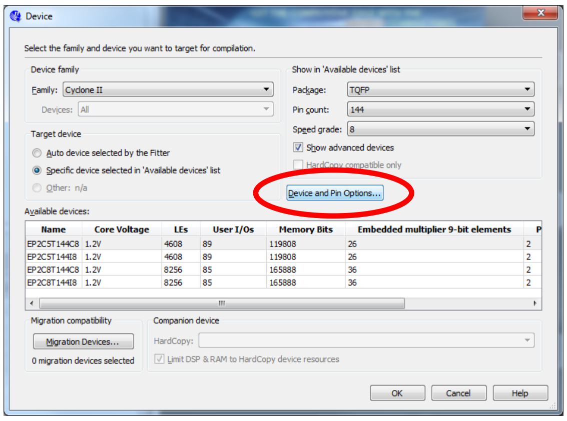

Quartus II 64-Bit - C:/Users/rr/GoogleDrive/FPGA/altera/boards/A-C2FB/NIOS-SOPC/sop
File
Edit
View
Project
Assignments
Processing
Tools
Window
Help
Project Navigator
Files
Device...
Settings... Ctrl+Shift+E
TimeQuest Timing Analyzer Wizard...
Assignment Editor Ctrl+Shift+A
Pin Planner Ctrl+Shift+N

Figure 2. An important missing step.

Open "Device" dialog (Figure 2) and you will see a button "Device and Pin Options ...", select that (Figure 3). This button doesn't exist on the dialog when the project is created so you will need to do this as separate step – right now.



Select the family and device you want to target for compilation.

Device family
Family: Cyclone II
Devices: All

Target device
( ) Auto device selected by the Fitter
(●) Specific device selected in 'Available devices' list
( ) Other: n/a

Show in 'Available devices' list
Package: TQFP
Pin count: 144
Speed grade: 8
☑ Show advanced devices
☐ HardCopy compatible only

Available devices:

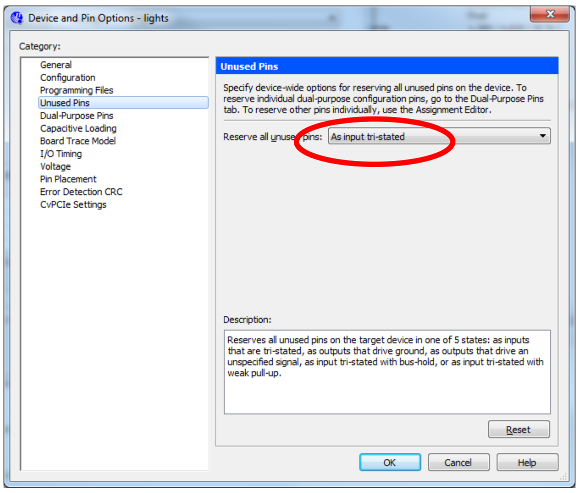
Name	Core Voltage	LEs	User I/Os	Memory Bits	Embedded multiplier 9-bit elements	P
EP2C5T144C8	1.2V	4608	89	119808	26	2
EP2C5T144I8	1.2V	4608	89	119808	26	2
EP2C8T144C8	1.2V	8256	85	165888	36	2
EP2C8T144I8	1.2V	8256	85	165888	36	2

Migration compatibility
0 migration devices selected

Companion device
HardCopy:
☑ Limit DSP & RAM to HardCopy device resources

Figure 3. We need to do something with our unused pins!

Change unused pins to tri-stated inputs (Figure 4).



Specify device-wide options for reserving all unused pins on the device. To reserve individual dual-purpose configuration pins, go to the Dual-Purpose Pins tab. To reserve other pins individually, use the Assignment Editor.

Reserve all unused pins: As input tri-stated

Description:

Reserves all unused pins on the target device in one of 5 states: as inputs that are tri-stated, as outputs that drive ground, as outputs that drive an unspecified signal, as input tri-stated with bus-hold, or as input tri-stated with weak pull-up.

Figure 4. Tell the pins to be "quiet"

Note also, I will be calling out figures, occasionally, from the Altera® tutorial so I will use Figure x for figures internal to this document and ***Figure y*** when referring to figures in the Altera® tutorial. Similarly, I will use ***Step x***. ***Table x.*** etc. to help remind you to go to the Altera® tutorial.

Ready, set, let's go.

Part I

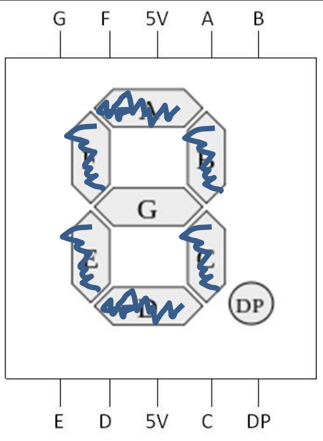
We run straight into a problem with this part of Laboratory 2 as it requires 10 switches so we know we need another way of doing this. As we did in Part V of Digital Labs Part 1, we can use constants to get around the absence of switches. Or, we can opt to drop one of the LED circuits – we have four switches yes and the problem uses four switches per segment. Yes, we'll do that.

Other than that, we just sketch out our LED segment lighting and then use LogicFriday to build a truth table. **WARNING:** Notice the inputs in Table 1 are **inverted** to take into account that the switches have pull-up resistors tying them to HIGH.

Table 1

LED_A	0	PIN_93	0
LED_B	1	PIN_92	0
LED_C	2	PIN_87	0
LED_D	3	PIN_86	0
LED_E	4	PIN_55	0
LED_F	5	PIN_58	0
LED_G	6	PIN_79	1

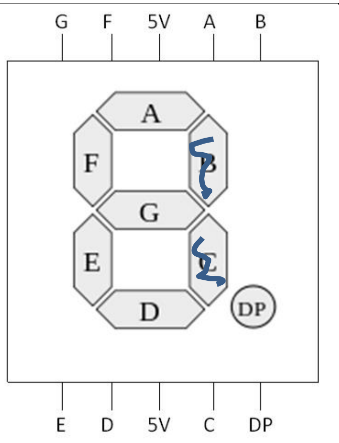
"1000000"



1111=>0

LED_A	0	PIN_93	1
LED_B	1	PIN_92	0
LED_C	2	PIN_87	0
LED_D	3	PIN_86	1
LED_E	4	PIN_55	1
LED_F	5	PIN_58	1
LED_G	6	PIN_79	1

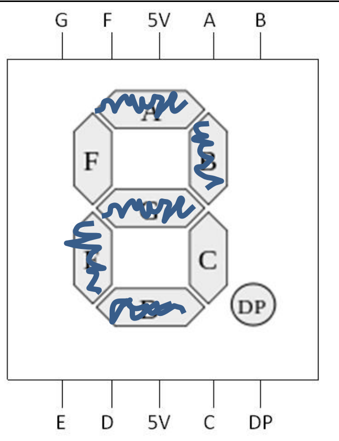
"1111001"



1110=>1

LED_A	0	PIN_93	0
LED_B	1	PIN_92	0
LED_C	2	PIN_87	1
LED_D	3	PIN_86	0
LED_E	4	PIN_55	0
LED_F	5	PIN_58	1
LED_G	6	PIN_79	0

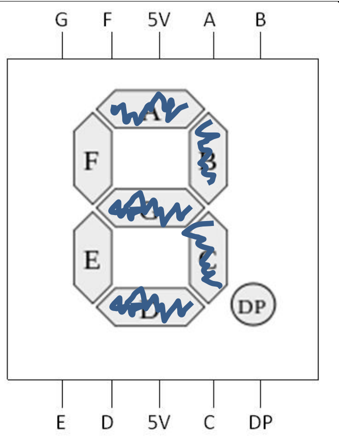
"0100100"



1101=>2

LED_A	0	PIN_93	0
LED_B	1	PIN_92	0
LED_C	2	PIN_87	0
LED_D	3	PIN_86	0
LED_E	4	PIN_55	1
LED_F	5	PIN_58	1
LED_G	6	PIN_79	0

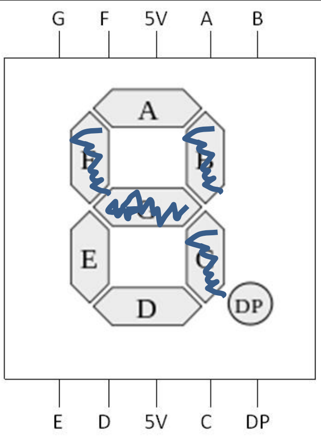
"0110000"



1100=>3

LED_A	0	PIN_93	1
LED_B	1	PIN_92	0
LED_C	2	PIN_87	0
LED_D	3	PIN_86	1
LED_E	4	PIN_55	1
LED_F	5	PIN_58	0
LED_G	6	PIN_79	0

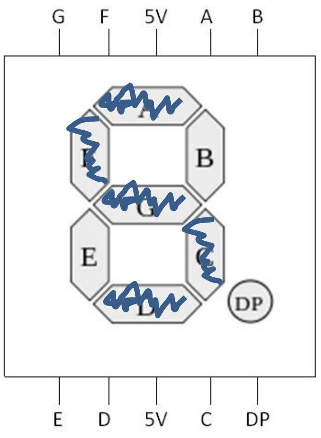
"0011001"



1011=>4

LED_A	0	PIN_93	0
LED_B	1	PIN_92	1
LED_C	2	PIN_87	0
LED_D	3	PIN_86	0
LED_E	4	PIN_55	1
LED_F	5	PIN_58	0
LED_G	6	PIN_79	0

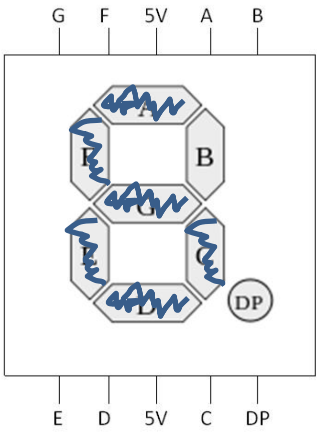
"0010010"



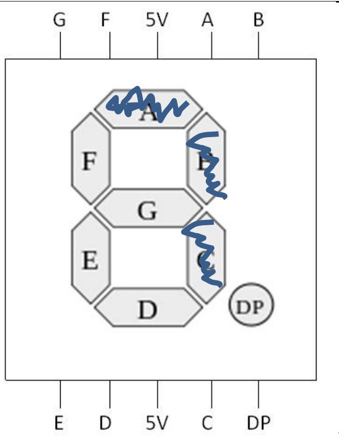
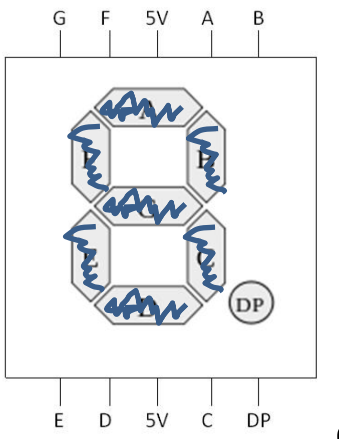
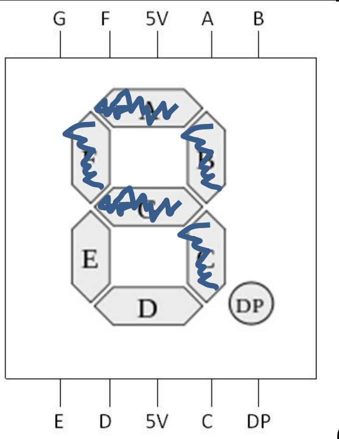
1010=>5

LED_A	0	PIN_93	0
LED_B	1	PIN_92	1
LED_C	2	PIN_87	0
LED_D	3	PIN_86	0
LED_E	4	PIN_55	0
LED_F	5	PIN_58	0
LED_G	6	PIN_79	0

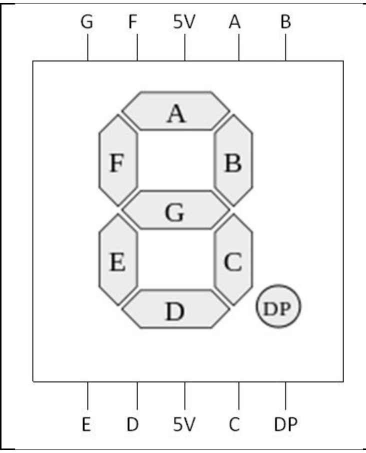
"0000010"



1001=>6

LED_A 0 PIN_93 0 LED_B 1 PIN_92 0 LED_C 2 PIN_87 0 LED_D 3 PIN_86 1 LED_E 4 PIN_55 1 LED_F 5 PIN_58 1 LED_G 6 PIN_79 1 "1111000"	 G F 5V A B A F B G E C D DP E D 5V C DP 1000=>7
LED_A 0 PIN_93 0 LED_B 1 PIN_92 0 LED_C 2 PIN_87 0 LED_D 3 PIN_86 0 LED_E 4 PIN_55 0 LED_F 5 PIN_58 0 LED_G 6 PIN_79 0 "0000000"	 0111=>8
LED_A 0 PIN_93 0 LED_B 1 PIN_92 0 LED_C 2 PIN_87 0 LED_D 3 PIN_86 1 LED_E 4 PIN_55 1 LED_F 5 PIN_58 0 LED_G 6 PIN_79 0 "0011000"	 G F 5V A B E D DP E D 5V C DP 0110=9

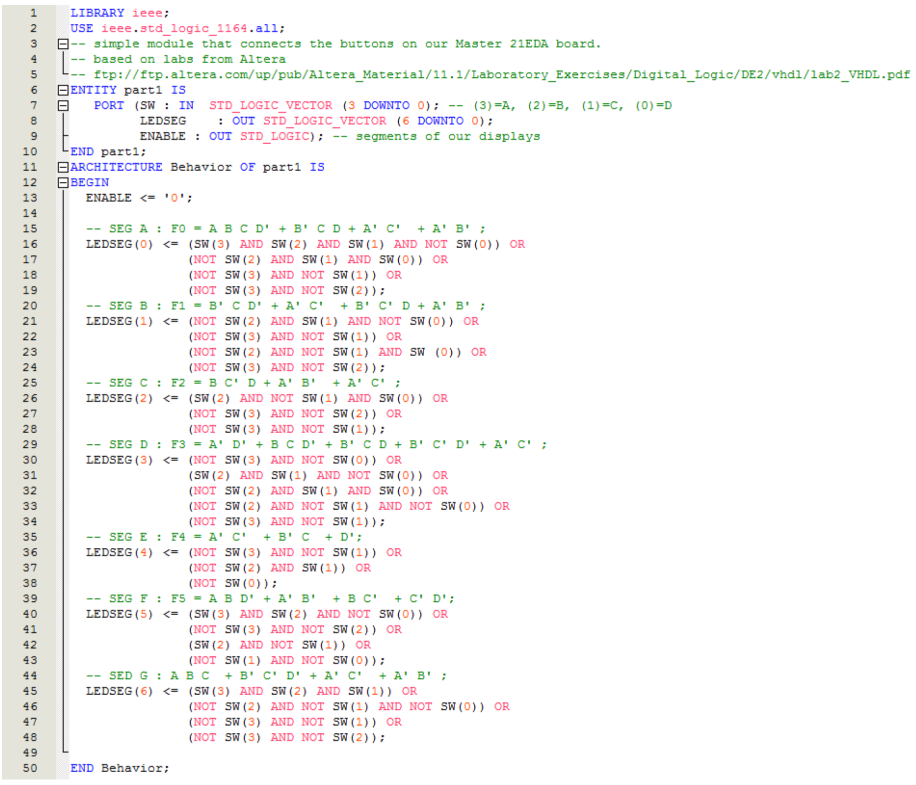
LED_A	0	PIN_93	1
LED_B	1	PIN_92	1
LED_C	2	PIN_87	1
LED_D	3	PIN_86	1
LED_E	4	PIN_55	1
LED_F	5	PIN_58	1
LED_G	6	PIN_79	1
otherwise			



The truth table becomes:

```
Minimized:
        F0 = A B C D' + B' C D + A' C' + A' B';
        F1 = B' C D' + A' C' + B' C' D + A' B';
        F2 = B C' D + A' B' + A' C';
        F3 = A' D' + B C D' + B' C D + B' C' D' + A' C';
        F4 = A' C' + B' C + D';
        F5 = A B D' + A' B' + B C' + C' D';
        F6 = A B C + B' C' D' + A' C' + A' B';
```

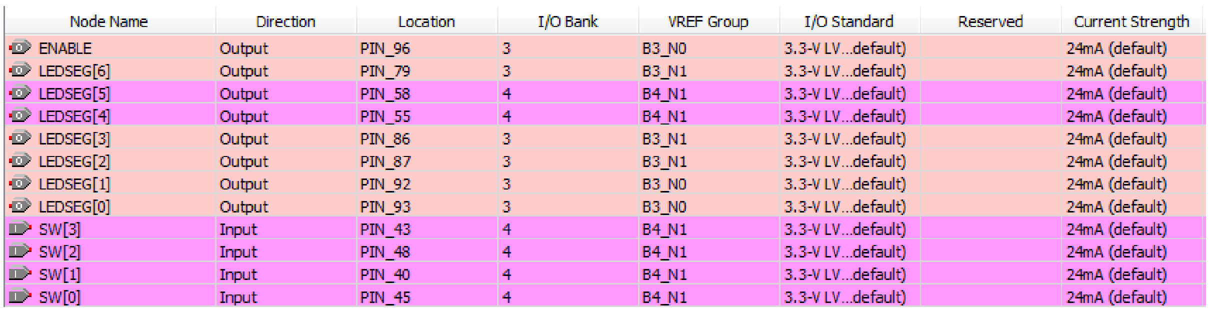
So coding that up we get Figure 5.



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- simple module that connects the buttons on our Master 21EDA board.
-- based on labs from Altera
-- ftp://ftp.altera.com/up/pub/Altera_Material/11.1/Laboratory_Exercises/Digital_Logic/DE2/vhdl/lab2_VHDL.pdf
ENTITY part1 IS
   PORT (SW : IN  STD_LOGIC_VECTOR (3 DOWNTO 0); -- (3)=A, (2)=B, (1)=C, (0)=D
         LEDSEG    : OUT STD_LOGIC_VECTOR (6 DOWNTO 0);
         ENABLE : OUT STD_LOGIC); -- segments of our displays
END part1;
ARCHITECTURE Behavior OF part1 IS
BEGIN
  ENABLE <= '0';

  -- SEG A : F0 = A B C D' + B' C D + A' C'  + A' B' ;
  LEDSEG(0) <= (SW(3) AND SW(2) AND SW(1) AND NOT SW(0)) OR
               (NOT SW(2) AND SW(1) AND SW(0)) OR
               (NOT SW(3) AND NOT SW(1)) OR
               (NOT SW(3) AND NOT SW(2));
  -- SEG B : F1 = B' C D' + A' C'  + B' C' D + A' B' ;
  LEDSEG(1) <= (NOT SW(2) AND SW(1) AND NOT SW(0)) OR
               (NOT SW(3) AND NOT SW(1)) OR
               (NOT SW(2) AND NOT SW(1) AND SW (0)) OR
               (NOT SW(3) AND NOT SW(2));
  -- SEG C : F2 = B C' D + A' B'  + A' C' ;
  LEDSEG(2) <= (SW(2) AND NOT SW(1) AND SW(0)) OR
               (NOT SW(3) AND NOT SW(2)) OR
               (NOT SW(3) AND NOT SW(1));
  -- SEG D : F3 = A' D' + B C D' + B' C D + B' C' D' + A' C' ;
  LEDSEG(3) <= (NOT SW(3) AND NOT SW(0)) OR
               (SW(2) AND SW(1) AND NOT SW(0)) OR
               (NOT SW(2) AND SW(1) AND SW(0)) OR
               (NOT SW(2) AND NOT SW(1) AND NOT SW(0)) OR
               (NOT SW(3) AND NOT SW(1));
  -- SEG E : F4 = A' C'  + B' C  + D';
  LEDSEG(4) <= (NOT SW(3) AND NOT SW(1)) OR
               (NOT SW(2) AND SW(1)) OR
               (NOT SW(0));
  -- SEG F : F5 = A B D' + A' B'  + B C'  + C' D';
  LEDSEG(5) <= (SW(3) AND SW(2) AND NOT SW(0)) OR
               (NOT SW(3) AND NOT SW(2)) OR
               (SW(2) AND NOT SW(1)) OR
               (NOT SW(1) AND NOT SW(0));
  -- SED G : A B C  + B' C' D' + A' C'  + A' B' ;
  LEDSEG(6) <= (SW(3) AND SW(2) AND SW(1)) OR
               (NOT SW(2) AND NOT SW(1) AND NOT SW(0)) OR
               (NOT SW(3) AND NOT SW(1)) OR
               (NOT SW(3) AND NOT SW(2));

END Behavior;
```

Figure 5: Voila!

Don't forget the pin assignments at Figure 6.



Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strength
ENABLE	Output	PIN_96	3	B3_N0	3.3-V LV...default)		24mA (default)
LEDSEG[6]	Output	PIN_79	3	B3_N1	3.3-V LV...default)		24mA (default)
LEDSEG[5]	Output	PIN_58	4	B4_N1	3.3-V LV...default)		24mA (default)
LEDSEG[4]	Output	PIN_55	4	B4_N1	3.3-V LV...default)		24mA (default)
LEDSEG[3]	Output	PIN_86	3	B3_N1	3.3-V LV...default)		24mA (default)
LEDSEG[2]	Output	PIN_87	3	B3_N1	3.3-V LV...default)		24mA (default)
LEDSEG[1]	Output	PIN_92	3	B3_N0	3.3-V LV...default)		24mA (default)
LEDSEG[0]	Output	PIN_93	3	B3_N0	3.3-V LV...default)		24mA (default)
SW[3]	Input	PIN_43	4	B4_N1	3.3-V LV...default)		24mA (default)
SW[2]	Input	PIN_48	4	B4_N1	3.3-V LV...default)		24mA (default)
SW[1]	Input	PIN_40	4	B4_N1	3.3-V LV...default)		24mA (default)
SW[0]	Input	PIN_45	4	B4_N1	3.3-V LV...default)		24mA (default)

Figure 6

You should get a RTL somewhat like that at Figure 7.

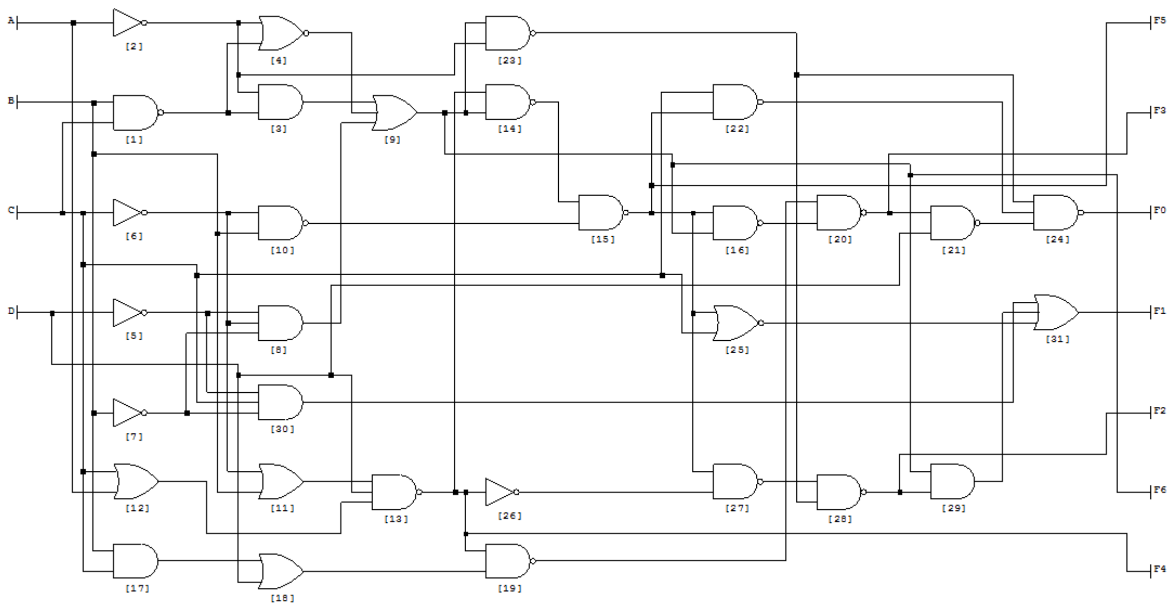


Figure 7

If you bother to code it up in LogicFriday I just used the 8 LED output widget as in Figure 8. If your wire up F0..F6 from Figure 7 then the LED match up with the "0011000" etc. segment enable maps of the logic table so you can interpret them.

So, with everything working you should have the 7-segment display that is enabled off PIN_96 light up as per design.

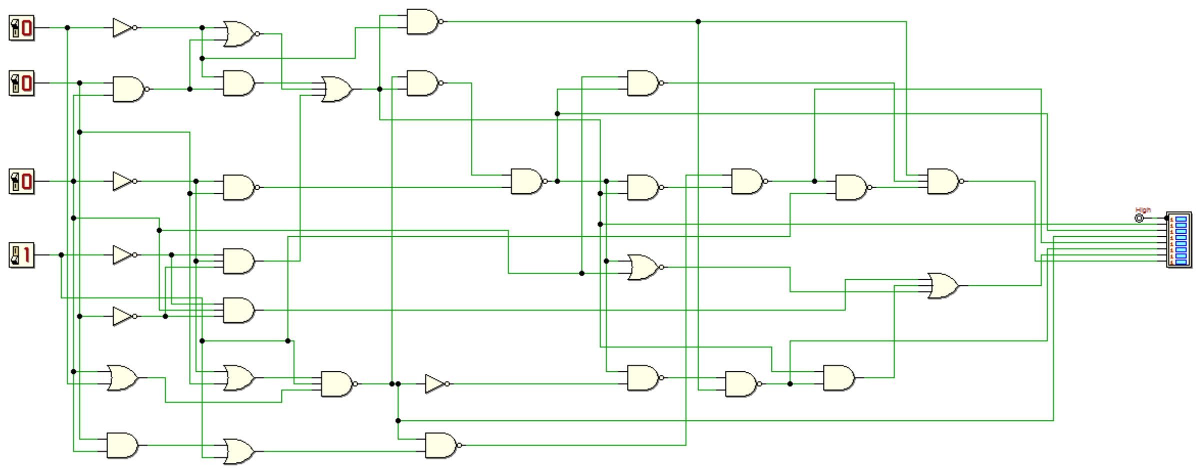


Figure 8

Part II

We really simply need to note something for "Circuit A" from **Figure 1**.

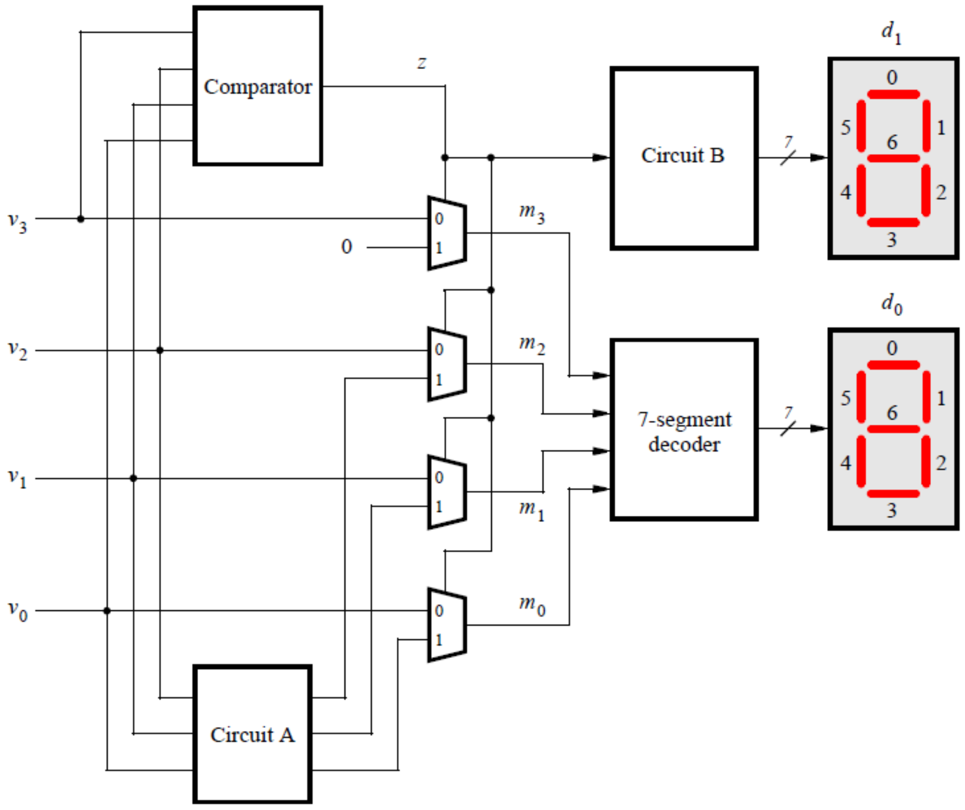


Figure 9: **Figure 1**

"Circuit A" looks like it needs to morph $v_2..v_0$ somehow when z is high. z being high when z>9.

Let's paint this out.

We've four KEYS to give us 00 through 15 on the two displays.

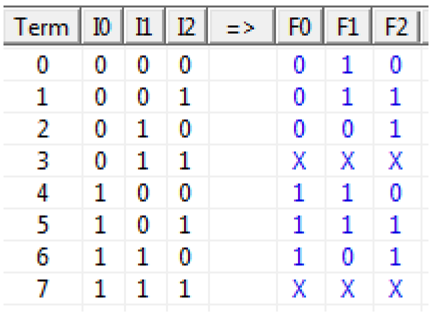
WARNING: The KEYS' column is to take into account that the keys (or buttons) on our board are pulled HIGH (so we want the table **inverted'** to drive our design). Looking at Table 2 below we see that for "Circuit A", once past VAL "09", we want to map I[210] (101 down to 000) to the values under F[210] (111 down to 010).

Table 2

KEYS	KEYS'	VAL	COMP			
0000	1111	00	0	This is the actual values we want at the multiplexor output >9		
0001	1110	01	0			
0010	1101	02	0			
0011	1100	03	0			
0100	1011	04	0			
0101	1010	05	0			
0110	1001	06	0			
0111	1000	07	0			
1000	0111	08	0			
1001	0110	09	0	I[210]		F[210]
1010	0101	10	1	101	0	111
1011	0100	11	1	100	1	110
1100	0011	12	1	011	2	101
1101	0010	13	1	010	3	100
1110	0001	14	1	001	4	011
1111	0000	15	1	000	5	010

To wit, we need to power up Logic Friday (note the instruction in the Altera LAB is to avoid using IF/THEN and CASE etc.).

Thus Figure 10:



Term	I0	I1	I2	=>	F0	F1	F2
0	0	0	0		0	1	0
1	0	0	1		0	1	1
2	0	1	0		0	0	1
3	0	1	1		X	X	X
4	1	0	0		1	1	0
5	1	0	1		1	1	1
6	1	1	0		1	0	1
7	1	1	1		X	X	X

Figure 10

Fun bit is that it makes more sense to have reverse the column labels for I0..I2 because I2 is actually the 2^0 column and I0 is the 2^3 column. No matter just reverse things in your head (of course there is no stopping one from setting the columns in LogicFriday). Note we are not using all the values so we can set 'X' or "Don't Care".

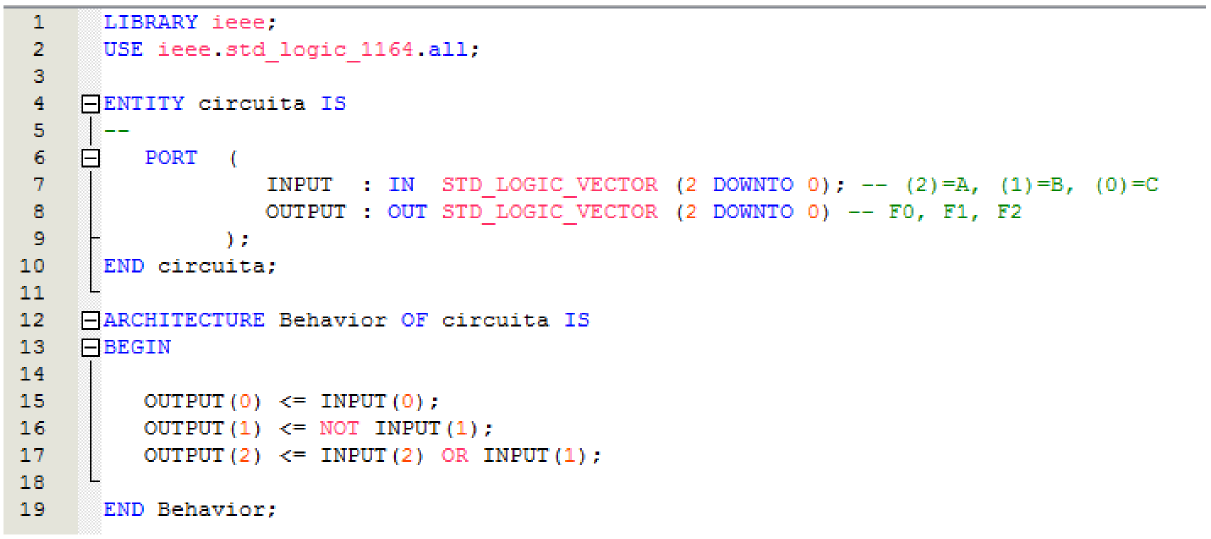
Entered by truth table:

F0 = I0 I1' I2' + I0 I1' I2 + I0 I1 I2';
F1 = I0' I1' I2' + I0' I1' I2 + I0 I1' I2' + I0 I1' I2;
F2 = I0' I1' I2 + I0' I1 I2' + I0 I1' I2 + I0 I1 I2';

Minimized:

F0 = I0;
F1 = I1';
F2 = I2 + I1;

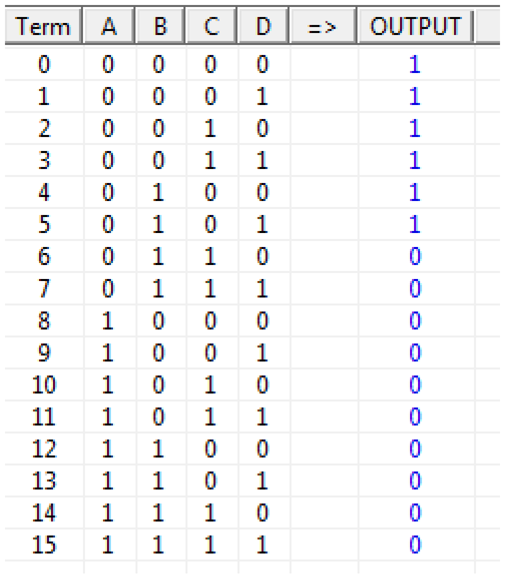
Coded up as Figure 11 below.



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY circuita IS
--
    PORT   (
                INPUT  : IN  STD_LOGIC_VECTOR (2 DOWNTO 0); -- (2)=A, (1)=B, (0)=C
                OUTPUT : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)  -- F0, F1, F2
            );
END circuita;

ARCHITECTURE Behavior OF circuita IS
BEGIN

    OUTPUT(0) <= INPUT(0);
    OUTPUT(1) <= NOT INPUT(1);
    OUTPUT(2) <= INPUT(2) OR INPUT(1);

END Behavior;
```

Figure 11

The "Comparator" is straight forward (again taking into account the input buttons being pulled HIGH) and thus Figure 12:



Term	A	B	C	D	=>	OUTPUT
0	0	0	0	0		1
1	0	0	0	1		1
2	0	0	1	0		1
3	0	0	1	1		1
4	0	1	0	0		1
5	0	1	0	1		1
6	0	1	1	0		0
7	0	1	1	1		0
8	1	0	0	0		0
9	1	0	0	1		0
10	1	0	1	0		0
11	1	0	1	1		0
12	1	1	0	0		0
13	1	1	0	1		0
14	1	1	1	0		0
15	1	1	1	1		0

Figure 12

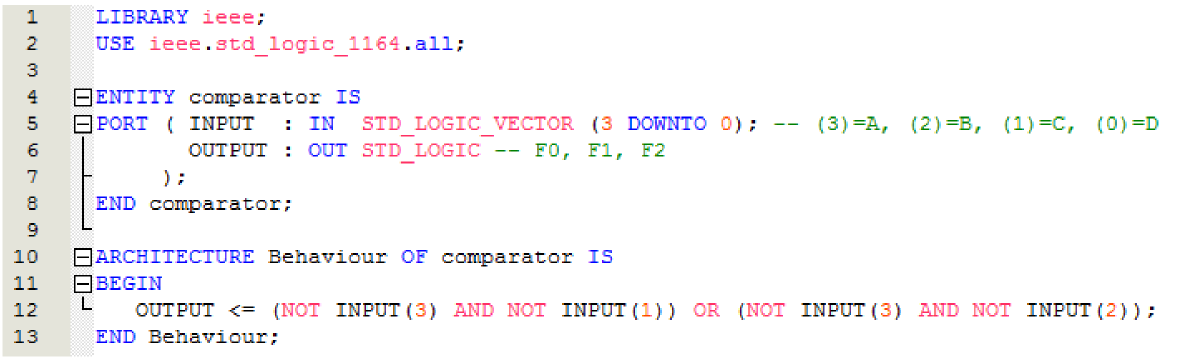
Entered by truthtable:

```
OUTPUT = A' B' C' D' + A' B' C' D + A' B' C D' + A' B' C D + A' B C'
D' + A' B C' D;
```

Minimized:

```
OUTPUT = A' C' + A' B';
```

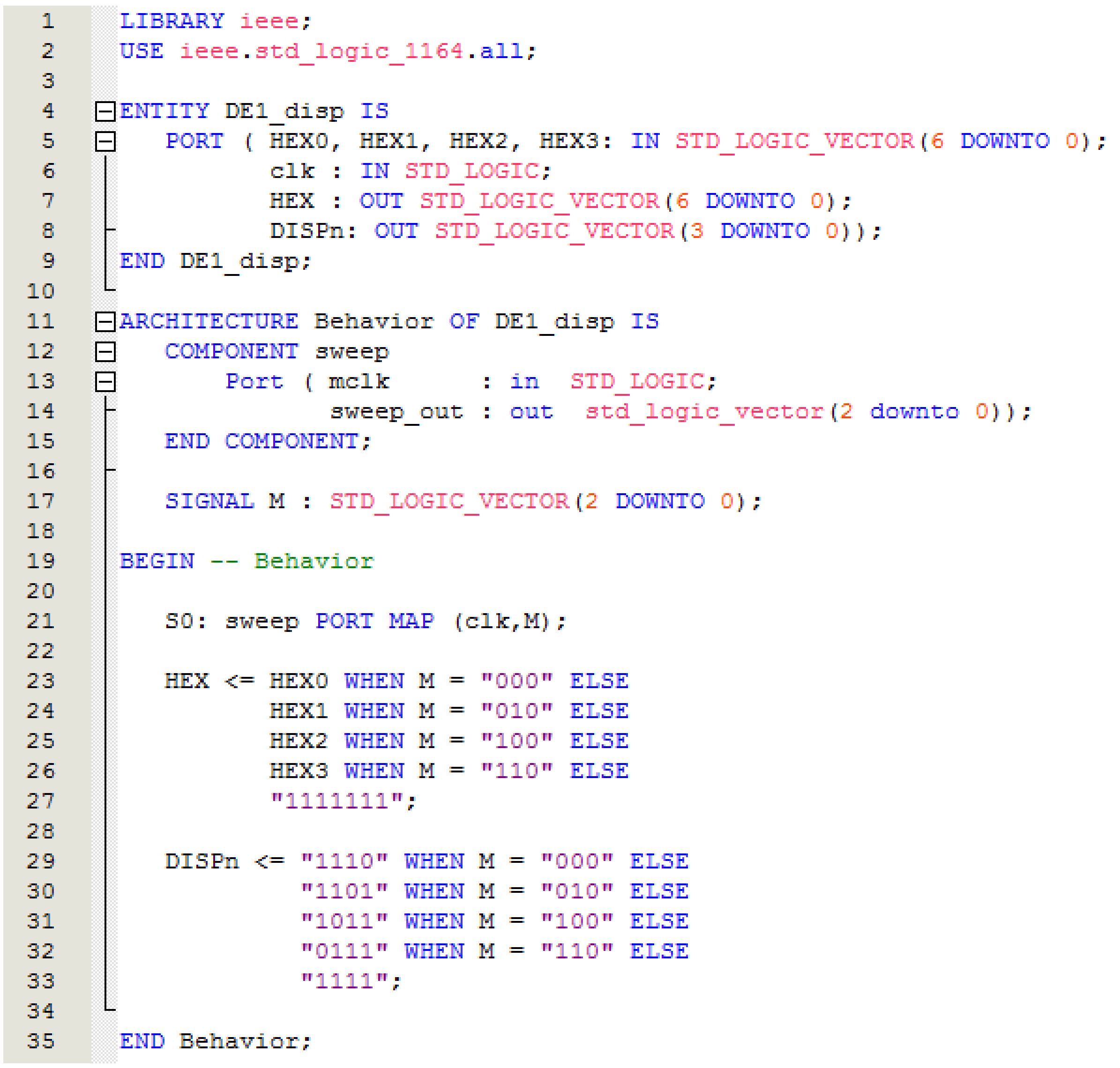
All that and all it becomes is Figure 13 below.



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY comparator IS
PORT ( INPUT   : IN  STD_LOGIC_VECTOR (3 DOWNTO 0); -- (3)=A, (2)=B, (1)=C, (0)=D
       OUTPUT  : OUT STD_LOGIC -- F0, F1, F2
     );
END comparator;

ARCHITECTURE Behaviour OF comparator IS
BEGIN
   OUTPUT <= (NOT INPUT(3) AND NOT INPUT(1)) OR (NOT INPUT(3) AND NOT INPUT(2));
END Behaviour;
```

Figure 13

The 7 segment code is from the previous Part I however, we drop the inbuilt enable as we are using the DE1_disp module from LAB 1 Part VI solution. We have fixed the "ghost" character we incurred with the original code by using the following code over the page. We'll assign d_0 and d_1 (from Figure 9: Figure 1) to HEX0 and HEX1 inputs respectively (Figure 14). We'll drive the other two inputs to blank out the 3rd and 4th displays.

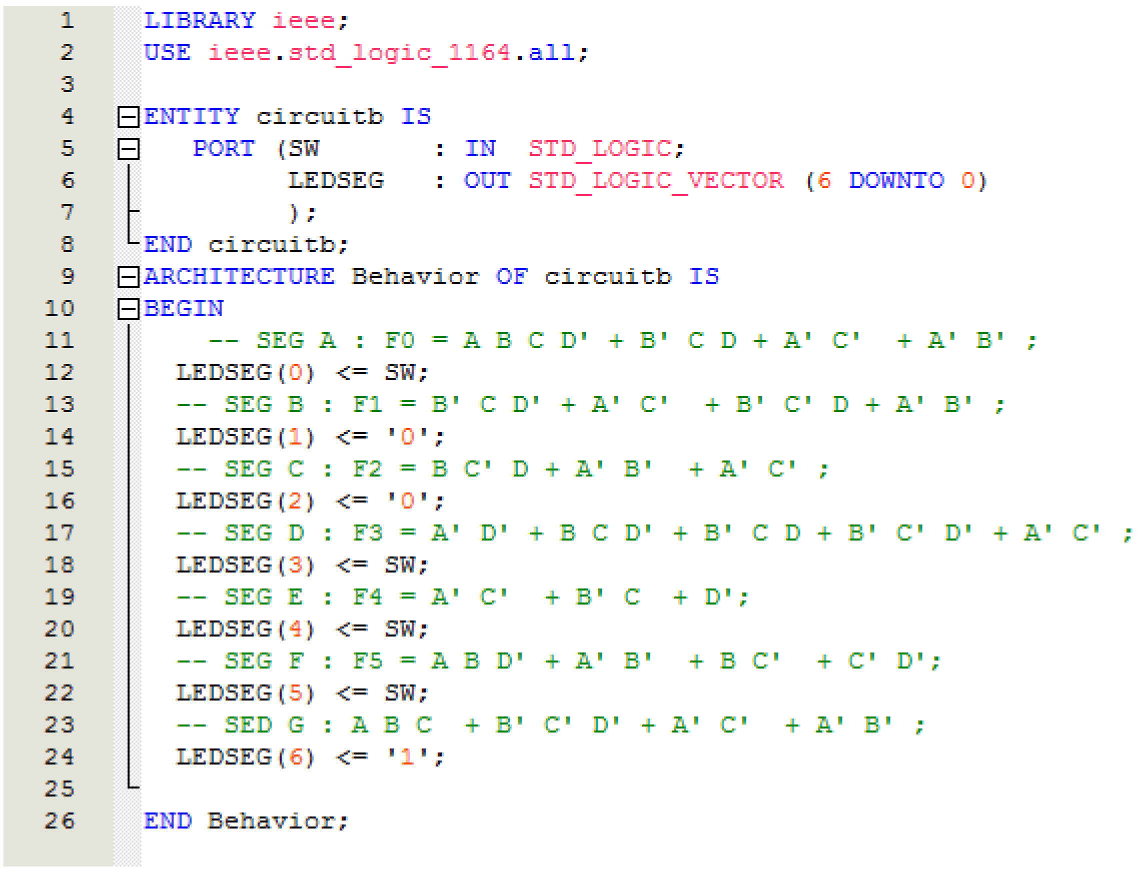


```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY DE1_disp IS
   PORT ( HEX0, HEX1, HEX2, HEX3: IN STD_LOGIC_VECTOR(6 DOWNTO 0);
          clk : IN STD_LOGIC;
          HEX : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
          DISPn: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END DE1_disp;

ARCHITECTURE Behavior OF DE1_disp IS
   COMPONENT sweep
       Port ( mclk      : in  STD_LOGIC;
              sweep_out : out  std_logic_vector(2 downto 0));
   END COMPONENT;

   SIGNAL M : STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN -- Behavior

   S0: sweep PORT MAP (clk,M);

   HEX <= HEX0 WHEN M = "000" ELSE
          HEX1 WHEN M = "010" ELSE
          HEX2 WHEN M = "100" ELSE
          HEX3 WHEN M = "110" ELSE
          "1111111";

   DISPn <= "1110" WHEN M = "000" ELSE
            "1101" WHEN M = "010" ELSE
            "1011" WHEN M = "100" ELSE
            "0111" WHEN M = "110" ELSE
            "1111";

END Behavior;
```

Figure 14

Circuit B is simply either 7-segment '0' WHEN z='0' ELSE 7-segment '1' WHEN z='1' (using VHDL lingo). The upshot is that Brian (Brian Drummond on Stack Overflow) pointed out the actual problem was the saturation of the bipolar transistor on the board – meaning once it was charged up (enabled) it then took time to drain and the "ghost" was the transistor still driven open (driving circuit to ground) into the next digit time slot.

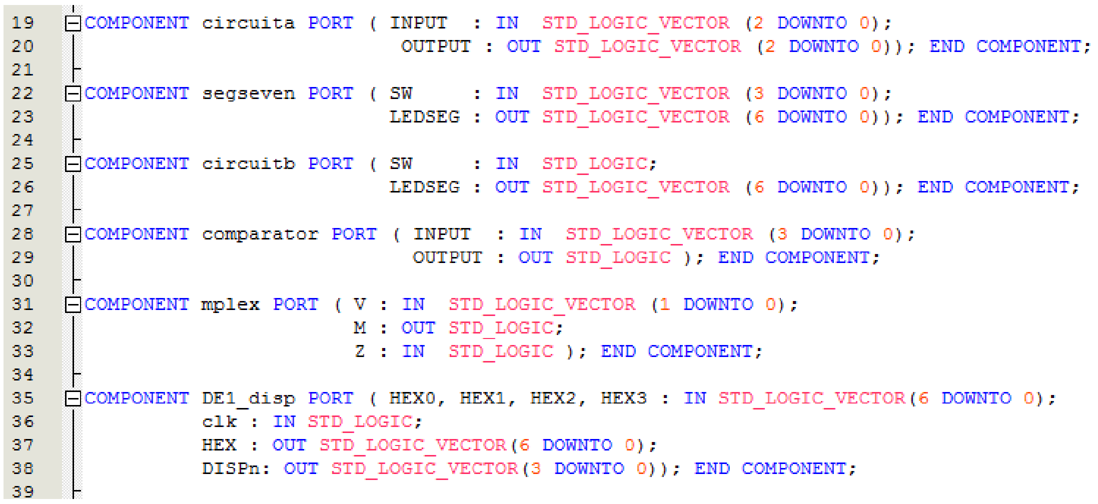
Without ceremony, "Circuit B" at Figure 15 over the page.



```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY circuitb IS
    PORT (SW        : IN  STD_LOGIC;
          LEDSEG    : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
          );
END circuitb;
ARCHITECTURE Behavior OF circuitb IS
BEGIN
    -- SEG A : F0 = A B C D' + B' C D + A' C'  + A' B' ;
  LEDSEG(0) <= SW;
  -- SEG B : F1 = B' C D' + A' C'  + B' C' D + A' B' ;
  LEDSEG(1) <= '0';
  -- SEG C : F2 = B C' D + A' B'  + A' C' ;
  LEDSEG(2) <= '0';
  -- SEG D : F3 = A' D' + B C D' + B' C D + B' C' D' + A' C' ;
  LEDSEG(3) <= SW;
  -- SEG E : F4 = A' C'  + B' C  + D';
  LEDSEG(4) <= SW;
  -- SEG F : F5 = A B D' + A' B'  + B C'  + C' D';
  LEDSEG(5) <= SW;
  -- SED G : A B C  + B' C' D' + A' C'  + A' B' ;
  LEDSEG(6) <= '1';

END Behavior;
```

Figure 15

To sum up then, we need to assemble a number of components. The definitions being below at Figure 16.



```vhdl
COMPONENT circuita PORT ( INPUT  : IN  STD_LOGIC_VECTOR (2 DOWNTO 0);
                          OUTPUT : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)); END COMPONENT;

COMPONENT segseven PORT ( SW     : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
                          LEDSEG : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)); END COMPONENT;

COMPONENT circuitb PORT ( SW     : IN  STD_LOGIC;
                          LEDSEG : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)); END COMPONENT;

COMPONENT comparator PORT ( INPUT  : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
                            OUTPUT : OUT STD_LOGIC ); END COMPONENT;

COMPONENT mplex PORT ( V : IN  STD_LOGIC_VECTOR (1 DOWNTO 0);
                       M : OUT STD_LOGIC;
                       Z : IN  STD_LOGIC ); END COMPONENT;

COMPONENT DE1_disp PORT ( HEX0, HEX1, HEX2, HEX3 : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
          clk : IN STD_LOGIC;
          HEX : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
          DISPn: OUT STD_LOGIC_VECTOR(3 DOWNTO 0)); END COMPONENT;

```

Figure 16

NOTE: the code for "segseven" is actually the code from Figure 5 of Part I. Just rename it from "part1" to "segseven" after moving the file to your Part II project etc.

The first 5 relating to the design in Figure 9: Figure 1 and the sixth being our tailoring of the Master 21EDA so that the 7-segment LED displays act somewhat like the ones on the DE1.

Wired up (in code) the result looks fine. More or less mimicking the design at Figure 9: Figure 1 as one would hope.

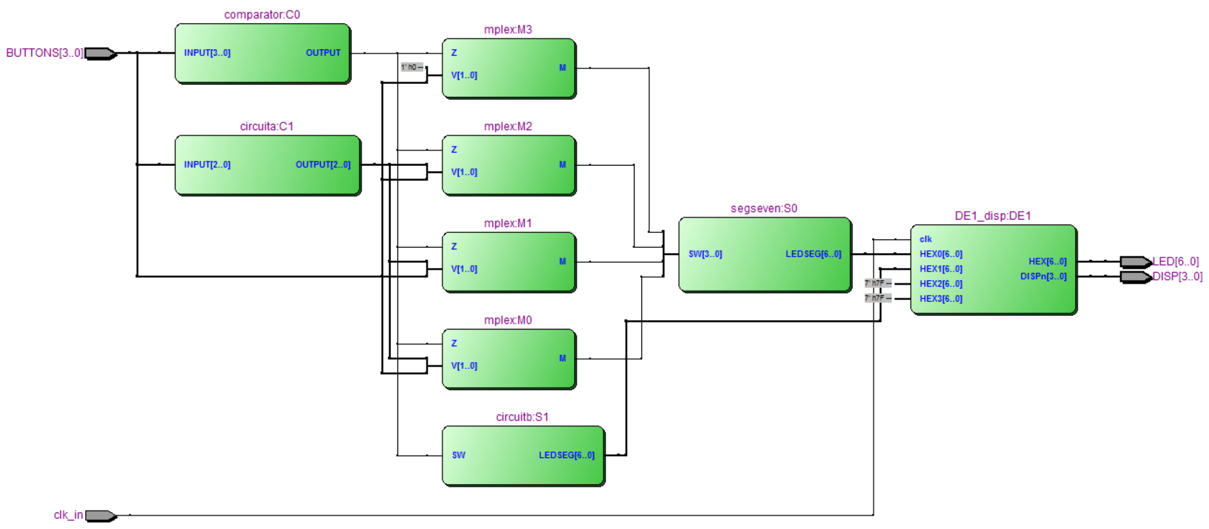
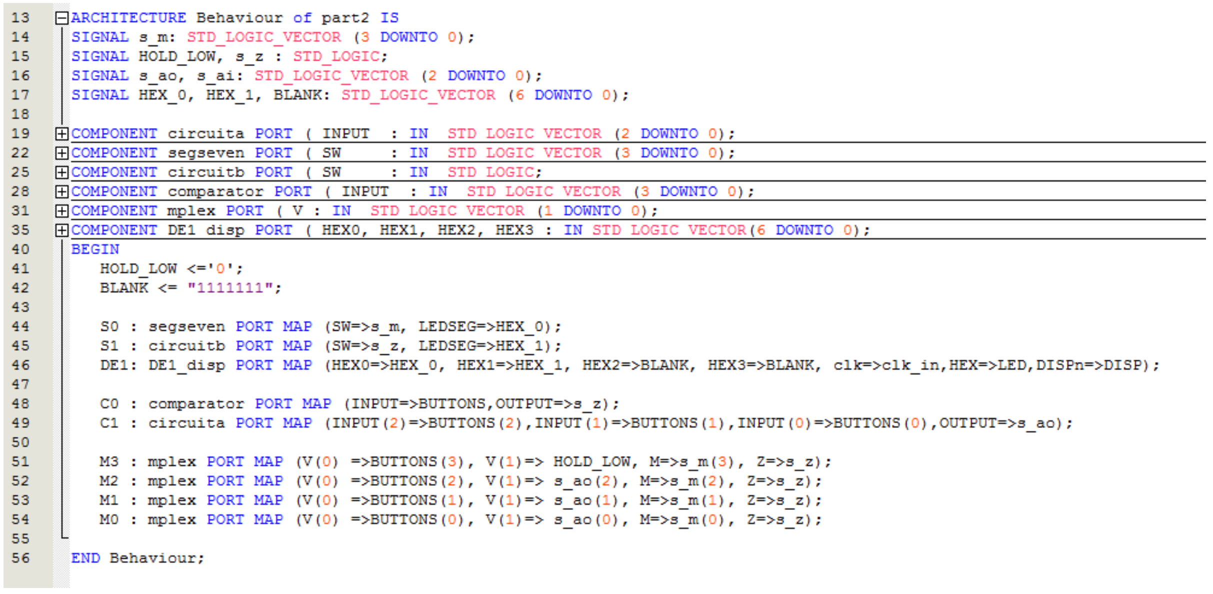


Figure 17

To wire it up we literally need some (signal) wires and so: we define:

```
s_m(3..0), HOLD_LOW, s_z, s_ao(2..0), s_ai(2..0), HEX_0(6..0), HEX1(6..0), BLANK(6..0)
```



```
ARCHITECTURE Behaviour of part2 IS
SIGNAL s_m: STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL HOLD_LOW, s_z : STD_LOGIC;
SIGNAL s_ao, s_ai: STD_LOGIC_VECTOR (2 DOWNTO 0);
SIGNAL HEX_0, HEX_1, BLANK: STD_LOGIC_VECTOR (6 DOWNTO 0);

COMPONENT circuita PORT ( INPUT   : IN  STD LOGIC VECTOR (2 DOWNTO 0);
COMPONENT segseven PORT ( SW      : IN  STD LOGIC VECTOR (3 DOWNTO 0);
COMPONENT circuitb PORT ( SW      : IN  STD LOGIC;
COMPONENT comparator PORT ( INPUT   : IN  STD LOGIC VECTOR (3 DOWNTO 0);
COMPONENT mplex PORT ( V : IN  STD LOGIC VECTOR (1 DOWNTO 0);
COMPONENT DE1 disp PORT ( HEX0, HEX1, HEX2, HEX3 : IN STD LOGIC VECTOR(6 DOWNTO 0);
BEGIN
   HOLD_LOW <='0';
   BLANK <= "1111111";

   S0 : segseven PORT MAP (SW=>s_m, LEDSEG=>HEX_0);
   S1 : circuitb PORT MAP (SW=>s_z, LEDSEG=>HEX_1);
   DE1: DE1_disp PORT MAP (HEX0=>HEX_0, HEX1=>HEX_1, HEX2=>BLANK, HEX3=>BLANK, clk=>clk_in,HEX=>LED,DISPn=>DISP);

   C0 : comparator PORT MAP (INPUT=>BUTTONS,OUTPUT=>s_z);
   C1 : circuita PORT MAP (INPUT(2)=>BUTTONS(2),INPUT(1)=>BUTTONS(1),INPUT(0)=>BUTTONS(0),OUTPUT=>s_ao);

   M3 : mplex PORT MAP (V(0) =>BUTTONS(3), V(1)=> HOLD_LOW, M=>s_m(3), Z=>s_z);
   M2 : mplex PORT MAP (V(0) =>BUTTONS(2), V(1)=> s_ao(2), M=>s_m(2), Z=>s_z);
   M1 : mplex PORT MAP (V(0) =>BUTTONS(1), V(1)=> s_ao(1), M=>s_m(1), Z=>s_z);
   M0 : mplex PORT MAP (V(0) =>BUTTONS(0), V(1)=> s_ao(0), M=>s_m(0), Z=>s_z);

END Behaviour;
```

Figure 18

To help decipher the wiring have a look at Figure 19 over the page.

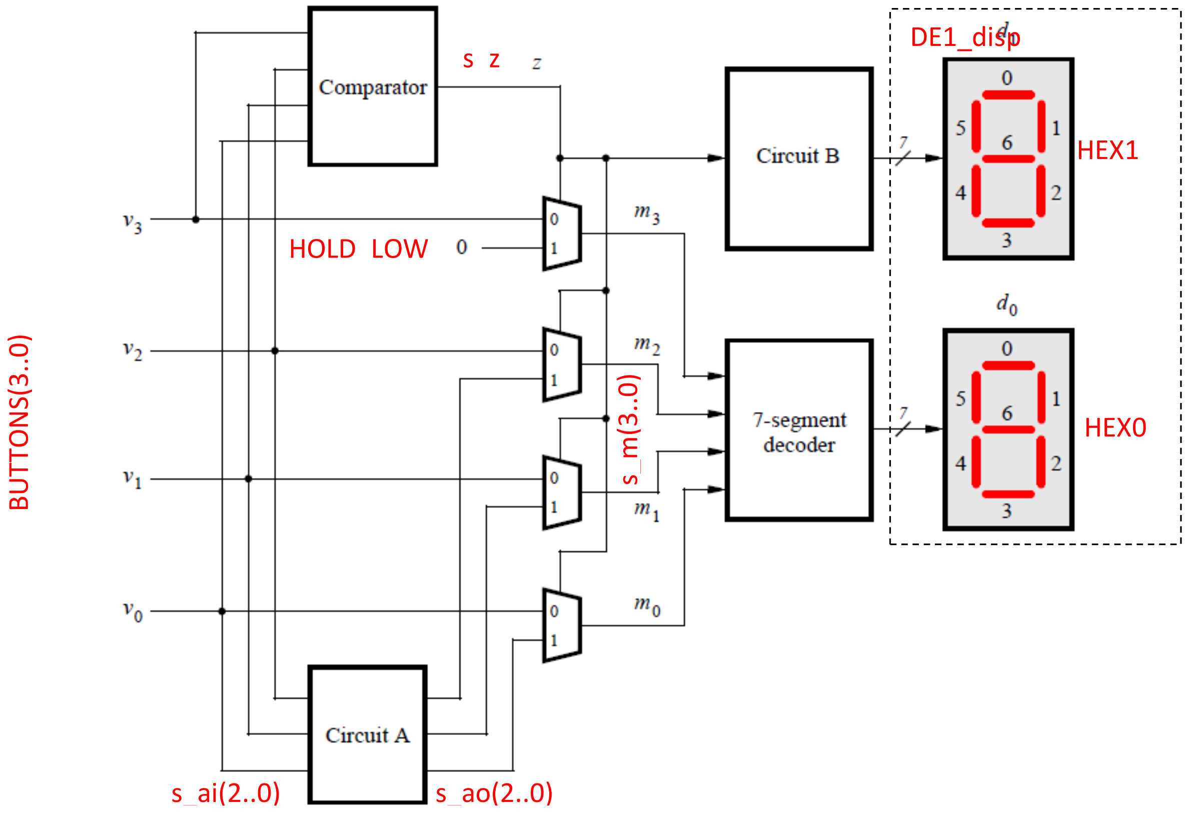
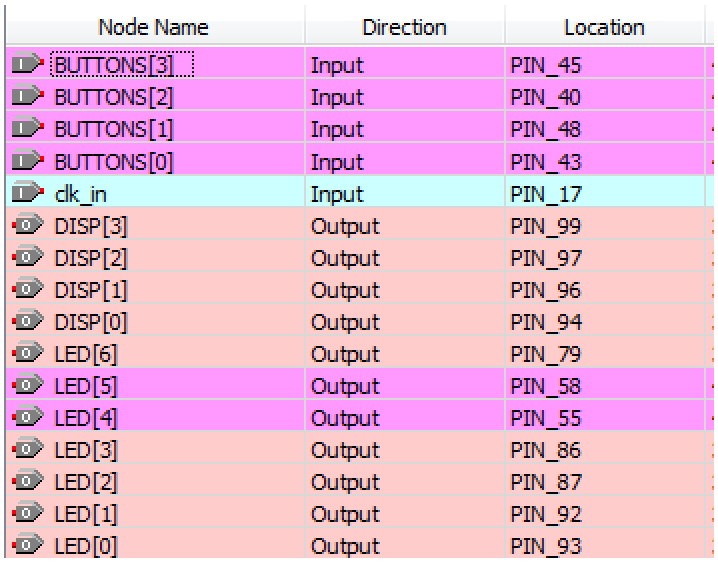


Figure 19

The final step, once compiled, is to wire up the design to pins as in Figure 20 below.



Node Name	Direction	Location
BUTTONS[3]	Input	PIN_45
BUTTONS[2]	Input	PIN_40
BUTTONS[1]	Input	PIN_48
BUTTONS[0]	Input	PIN_43
clk_in	Input	PIN_17
DISP[3]	Output	PIN_99
DISP[2]	Output	PIN_97
DISP[1]	Output	PIN_96
DISP[0]	Output	PIN_94
LED[6]	Output	PIN_79
LED[5]	Output	PIN_58
LED[4]	Output	PIN_55
LED[3]	Output	PIN_86
LED[2]	Output	PIN_87
LED[1]	Output	PIN_92
LED[0]	Output	PIN_93

Figure 20

Voila!

No ghost!

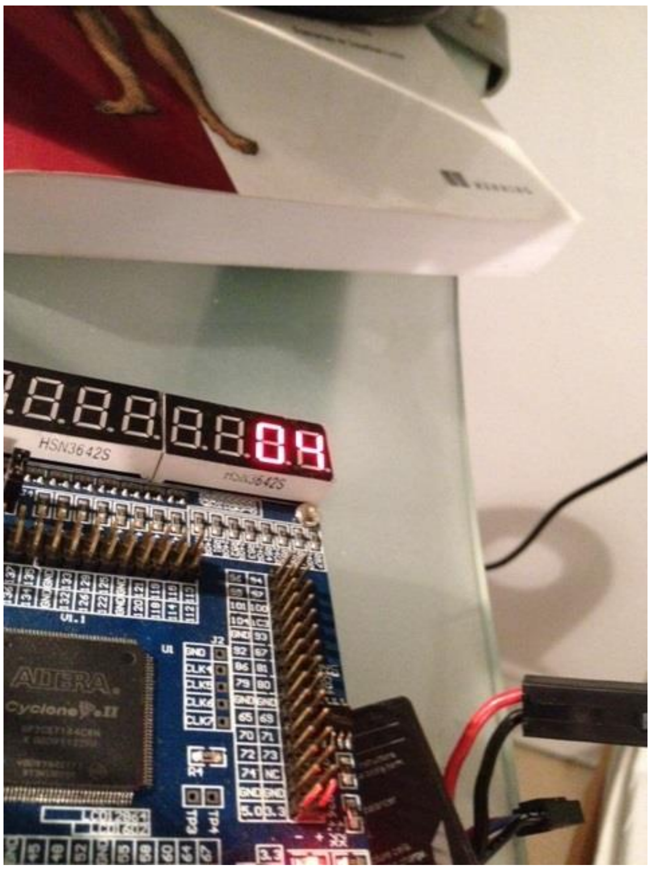


Figure 21

Part III

A ripple carry OR full adder in Figure 22:

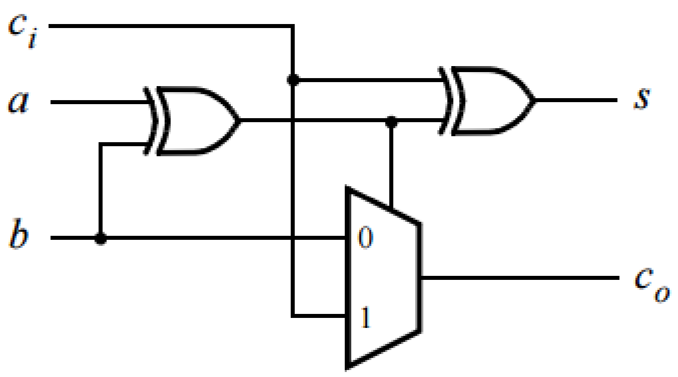


Figure 22

Begets Figure 23 in Deeds' Digital Circuit Simulator:

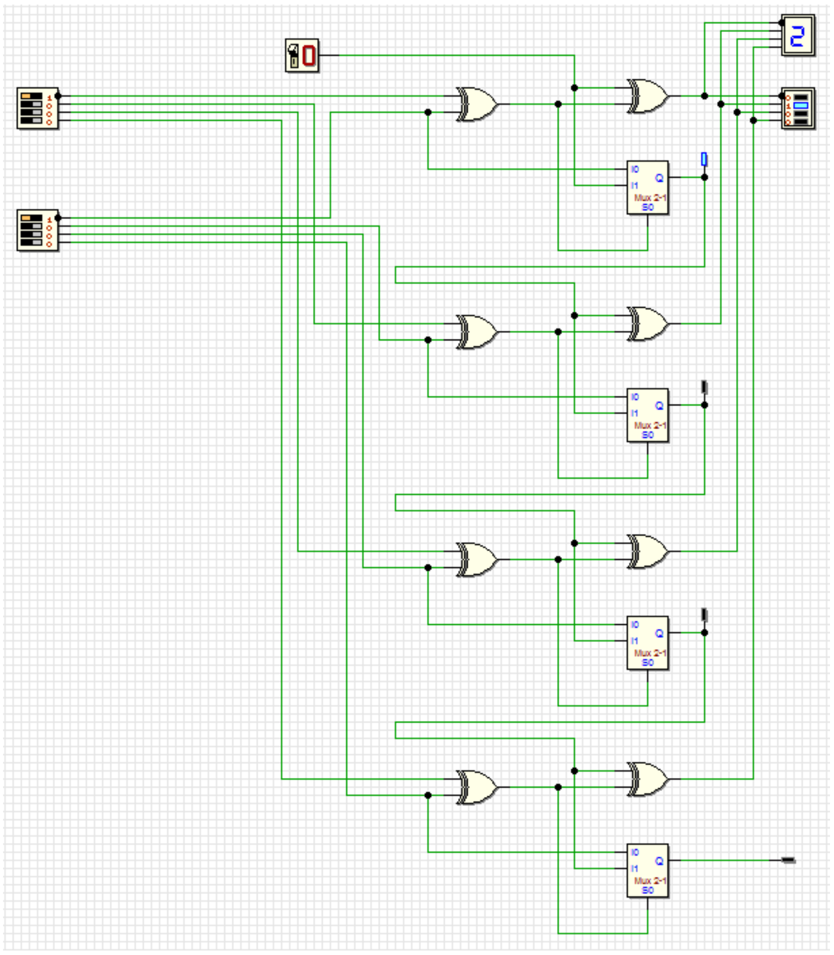
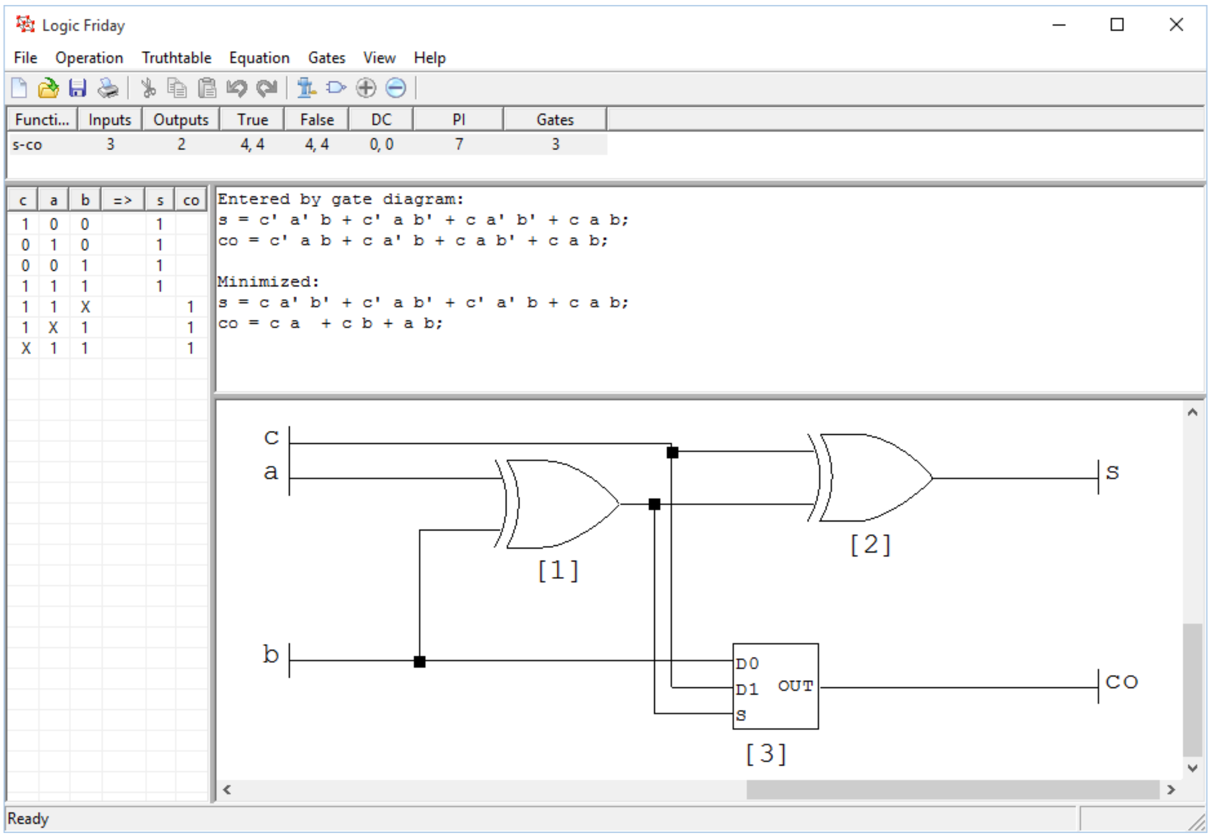


Figure 23

You can see above where there are test points on the carry flags and 1+1=2 with a carry flagged as one would expect. Now have a play. Is there a carry on 1+2? On 2+2? On 1+3? On 3+1?

This will be our test generator to check our LED outputs against inputs.

Now of course this is cheating! But LogicFriday allows you to draw your circuit as in Figure 24. When you submit you get your equations – minimized of course.



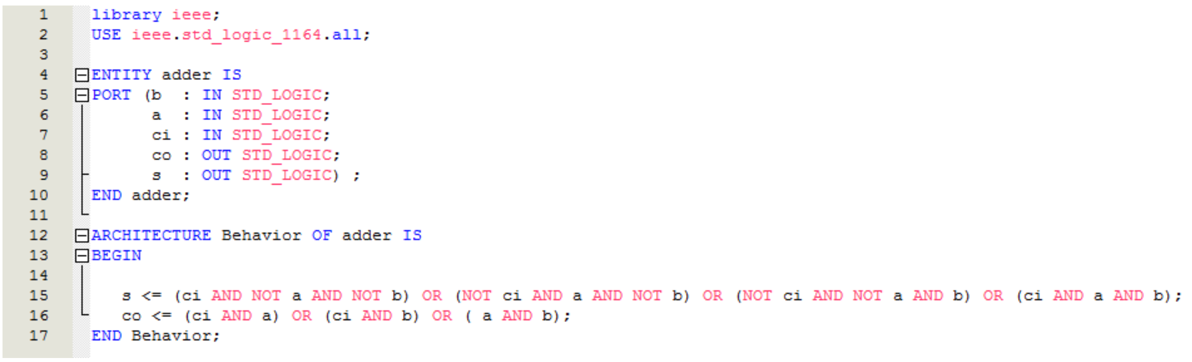
Functi...	Inputs	Outputs	True	False	DC	PI	Gates
s-co	3	2	4, 4	4, 4	0, 0	7	3

c	a	b	=>	s	co
1	0	0		1	
0	1	0		1	
0	0	1		1	
1	1	1		1	
1	1	X			1
1	X	1			1
X	1	1			1

```
Entered by gate diagram:
s = c' a' b + c' a b' + c a' b' + c a b;
co = c' a b + c a' b + c a b' + c a b;

Minimized:
s = c a' b' + c' a b' + c' a' b + c a b;
co = c a  + c b + a b;
```

Figure 24

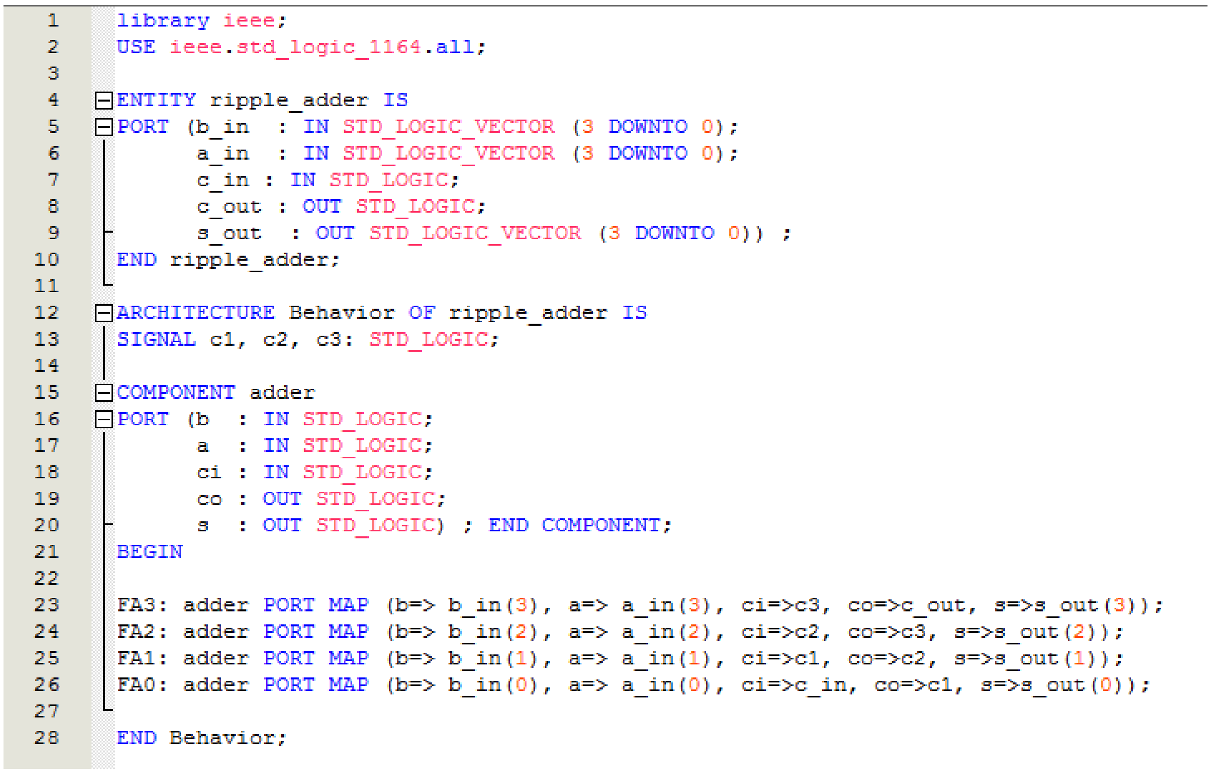
It is then straight forward to code up as in Figure 25.



```
library ieee;
USE ieee.std_logic_1164.all;

ENTITY adder IS
PORT (b   : IN STD_LOGIC;
      a   : IN STD_LOGIC;
      ci  : IN STD_LOGIC;
      co  : OUT STD_LOGIC;
      s   : OUT STD_LOGIC) ;
END adder;

ARCHITECTURE Behavior OF adder IS
BEGIN

   s <= (ci AND NOT a AND NOT b) OR (NOT ci AND a AND NOT b) OR (NOT ci AND NOT a AND b) OR (ci AND a AND b);
   co <= (ci AND a) OR (ci AND b) OR ( a AND b);
END Behavior;
```

Figure 25

Now we weave Figure 23 into code at Figure 26.

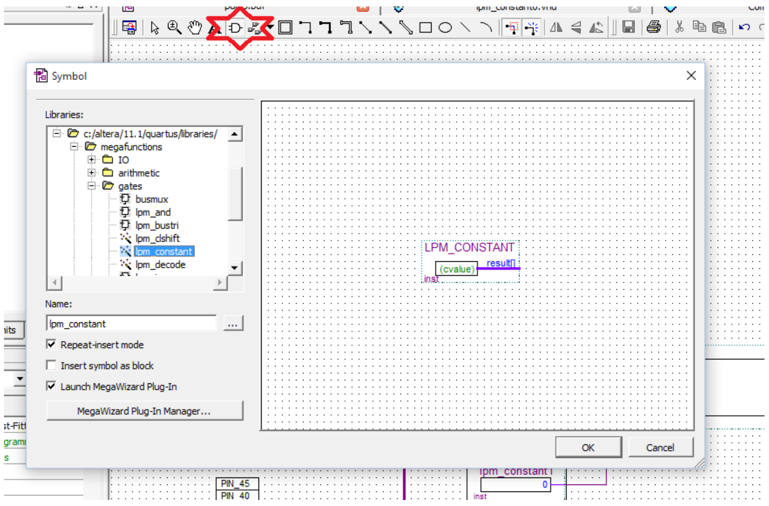


```
library ieee;
USE ieee.std_logic_1164.all;

ENTITY ripple_adder IS
PORT (b_in   : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      a_in   : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      c_in : IN STD_LOGIC;
      c_out : OUT STD_LOGIC;
      s_out   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END ripple_adder;

ARCHITECTURE Behavior OF ripple_adder IS
SIGNAL c1, c2, c3: STD_LOGIC;

COMPONENT adder
PORT (b   : IN STD_LOGIC;
      a   : IN STD_LOGIC;
      ci  : IN STD_LOGIC;
      co  : OUT STD_LOGIC;
      s   : OUT STD_LOGIC) ; END COMPONENT;
BEGIN

FA3: adder PORT MAP (b=> b_in(3), a=> a_in(3), ci=>c3, co=>c_out, s=>s_out(3));
FA2: adder PORT MAP (b=> b_in(2), a=> a_in(2), ci=>c2, co=>c3, s=>s_out(2));
FA1: adder PORT MAP (b=> b_in(1), a=> a_in(1), ci=>c1, co=>c2, s=>s_out(1));
FA0: adder PORT MAP (b=> b_in(0), a=> a_in(0), ci=>c_in, co=>c1, s=>s_out(0));

END Behavior;
```

Figure 26

Now, remember we are short switches so we need to cheat and use the schematic capture. We need 4 times 1 bit lpm_constant (Figure 27) for our 'b_in[3..0]' input. Well feed input 'a_in[3..0]' from the switches. May as well have an adjustable carry input as well. The schematic looks like Figure 28 over the page.

LPM_Constants explanation is at: http://quartushelp.altera.com/current/master.htm



Symbol

Libraries:
c:/altera/11.1/quartus/libraries/
megafunctions
IO
arithmetic
gates
busmux
lpm_and
lpm_bustri
lpm_clshift
lpm_constant
lpm_decode

Name: lpm_constant

[✓] Repeat-insert mode
[ ] Insert symbol as block
[✓] Launch MegaWizard Plug-In

Figure 27

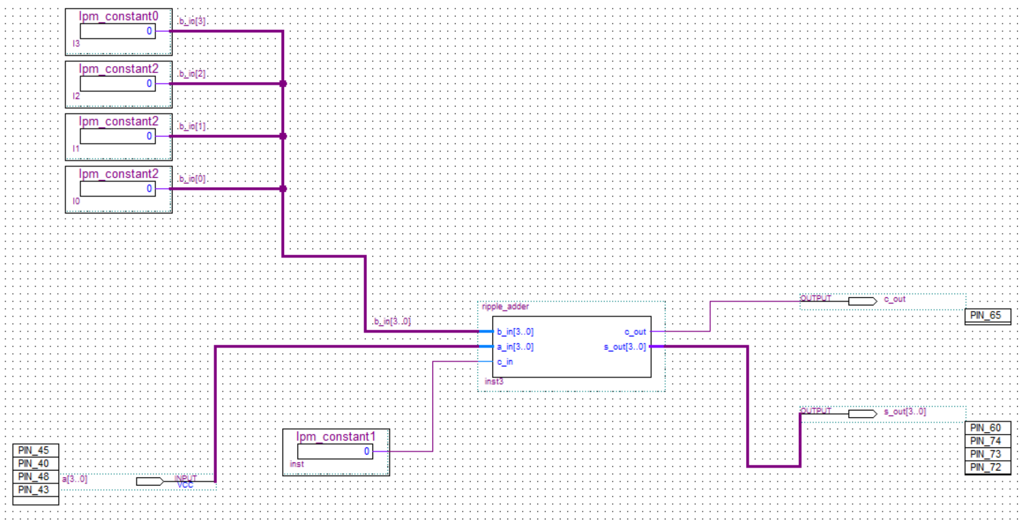


Figure 28

Note the naming on the "b_in[3..0]" buss in Figure 28. At the buss end you still need to name the buss as it didn't do what I thought might be the intuitive thing to do, which is to pick up the name of the inputs. Likely makes sense though since, being a buss, it might have a raft of input/output points and thus likely its own name.

The trick is at the other end, on the 1-bit constants is using the 'wire' within the buss we are interested in at each constant – which represent I3=8, I2=4, I1=2 and I0=1.

To make this clearer read Table 3 below.

Table 3

Binary Value	Pin	Ripple a_in	Constant ID	Ripple b_in	Ripple Out
8	PIN_45	a_in[3]	I3	b_in[3]	s_out[3]
4	PIN_40	a_in[2]	I2	b_in[2]	s_out[2]
2	PIN_48	a_in[1]	I1	b_in[1]	s_out[1]
1	PIN_43	a_in[0]	I0	b_in[0]	s_out[0]

In any event now we are cooking. Open the "In-System Memory Content Editor" shown at Figure 29.

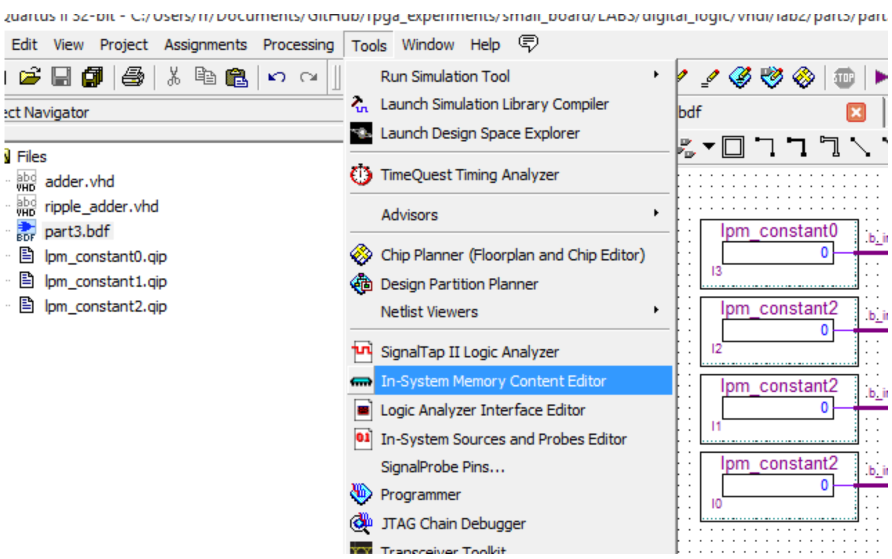
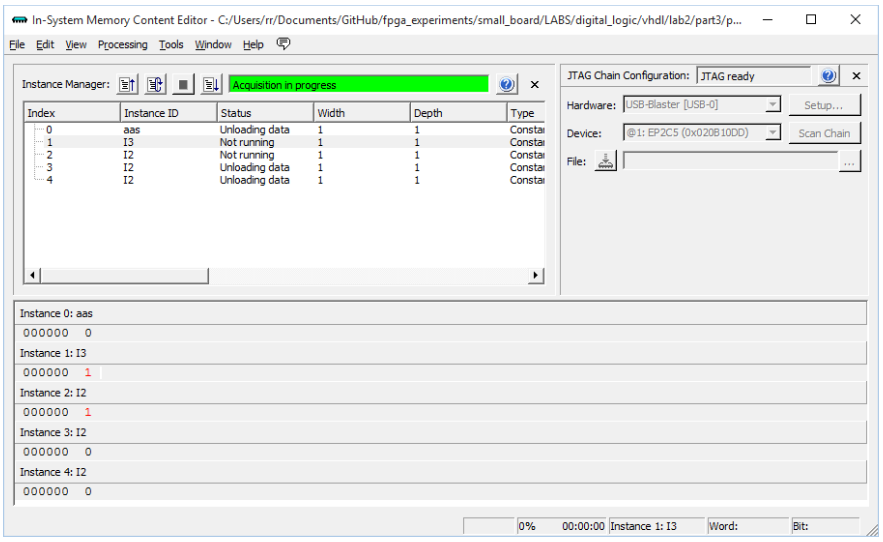

Edit View Project Assignments Processing Tools Window Help
ct Navigator
Files
adder.vhd
ripple_adder.vhd
part3.bdf
lpm_constant0.qip
lpm_constant1.qip
lpm_constant2.qip
Run Simulation Tool
Launch Simulation Library Compiler
Launch Design Space Explorer
TimeQuest Timing Analyzer
Advisors
Chip Planner (Floorplan and Chip Editor)
Design Partition Planner
Netlist Viewers
SignalTap II Logic Analyzer
In-System Memory Content Editor
Logic Analyzer Interface Editor
In-System Sources and Probes Editor
SignalProbe Pins...
Programmer
JTAG Chain Debugger
Transceiver Toolkit
lpm_constant0
lpm_constant2
lpm_constant2
lpm_constant2

Figure 29

Refer to Quartus II Handbook Version 11.1 Volume 3: Verification Section IV. System Debugging Tools for information on how to use the "In-System Memory Content Editor". Especially "15. In-System Modification of Memory and Constants".

Once you are familiar with the tool, or at least while you have the manual open, you can modify the b_in[] values as displayed in Figure 30.


In-System Memory Content Editor - C:/Users/rr/Documents/GitHub/fpga_experiments/small_board/LABS/digital_logic/vhdl/lab2/part3/p...

Instance Manager: Acquisition in progress

Index	Instance ID	Status	Width	Depth	Type
0	aas	Unloading data	1	1	Constai
1	I3	Not running	1	1	Constai
2	I2	Not running	1	1	Constai
3	I2	Unloading data	1	1	Constai
4	I2	Unloading data	1	1	Constai

JTAG Chain Configuration: JTAG ready
Hardware: USB-Blaster [USB-0]
Device: @1: EP2C5 (0x020B10DD)
File:

```
Instance 0: aas
000000   0
Instance 1: I3
000000   1
Instance 2: I2
000000   1
Instance 3: I2
000000   0
Instance 4: I2
000000   0
```

0%   00:00:00   Instance 1: I3   Word: Bit:

Figure 30

Now if your kindergarten math is up to scratch you can now play with adding binary. Again, you can cross check with simulation of the circuit in Deeds at Figure 23.

Part IV

You are to design a circuit that adds two BCD digits. The inputs to the circuit are BCD numbers *A* and *B*, plus a carry-in, cin. The output should be a two-digit BCD sum S_1S_0. Note that the largest sum that needs to be handled by this circuit is $S_1S_0 = 9 + 9 + 1 = 19$. Perform the steps given below.

Wow! Design a means to add two Binary Coded Decimal (BCD) numbers!

https://en.wikipedia.org/wiki/Binary-coded_decimal

So if your digital math is not up to speed, at little help is at hand.

http://www2.elo.utfsm.cl/~lsb/elo211/aplicaciones/katz/chapter5/chapter05.doc4.html

Short answer is, apparently, one will add 6 if previous sum > 9. When you read the explanation from the link above the table below might help. Remember 'X' means "don't care" (see also Figure 31).

#	B_3	B_2	B_1	B_0	Combinations	Representation	GATE
9	1	0	0	1			
10	1	0	1	0	B_3& B_1	1X1X	A2
11	1	0	1	1	B_3& B_1	1X1X	A2
12	1	1	0	0	B_3& B_2	11XX	A1
13	1	1	0	1	B_3& B_2	11XX	A1
14	1	1	1	0	(B_3& B_2)||(B_3& B_1)	11XX|1X1X	A1|A2
15	1	1	1	1	(B_3& B_2)||(B_3& B_1)	11XX|1X1X	A1|A2

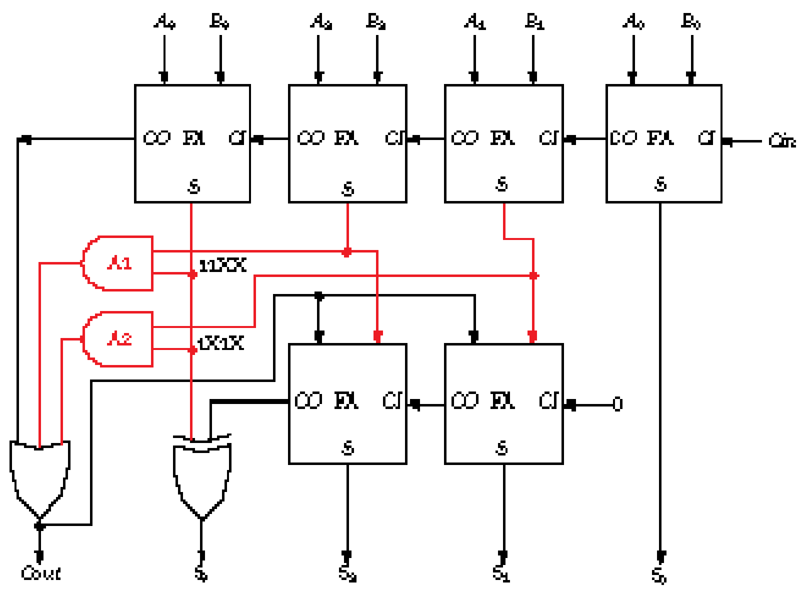


Figure 5.27 BCD adder block diagram

Figure 31

Note also the constraint that maximum number to output need only be 19. This helps as it simply means S_0 is 4 bit and Carry Out of the circuit can along with a display driver we have used previously, paint a "1" or a "0" in the 7-segment display.

Now all we have to do is simply rewire the 'adder' from Part III with wires and 'glue'.

WHAT! There is an XOR gate in there! Well, turns out, while we have been relying on binary AND and OR operators previously, there is also a raft of other binary operators in VHDL, namely: AND, OR, NAND, NOR, XOR, XNOR.

For a brush up read:

http://whatis.techtarget.com/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR

Any old how, sketching out the wiring on the diagram we get Figure 32.

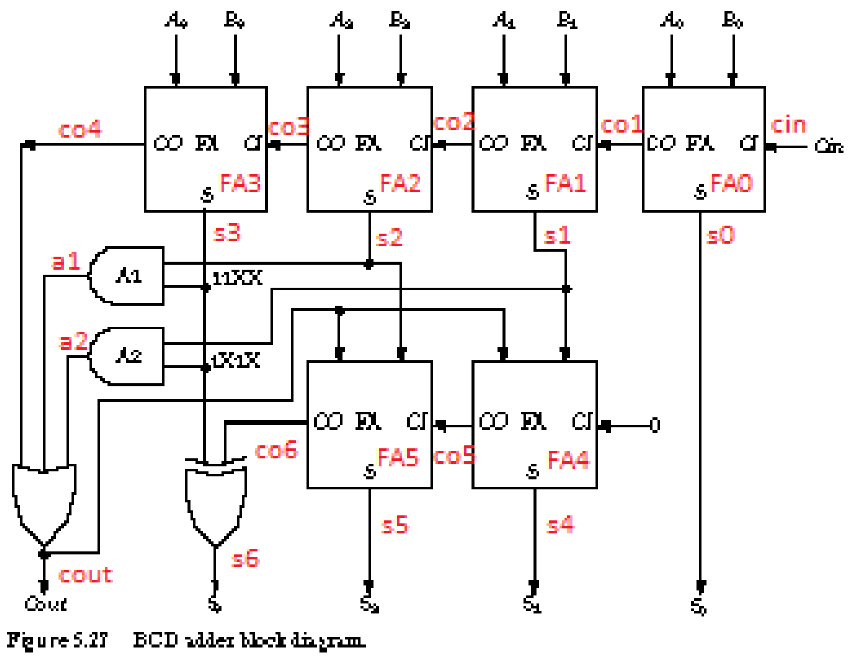
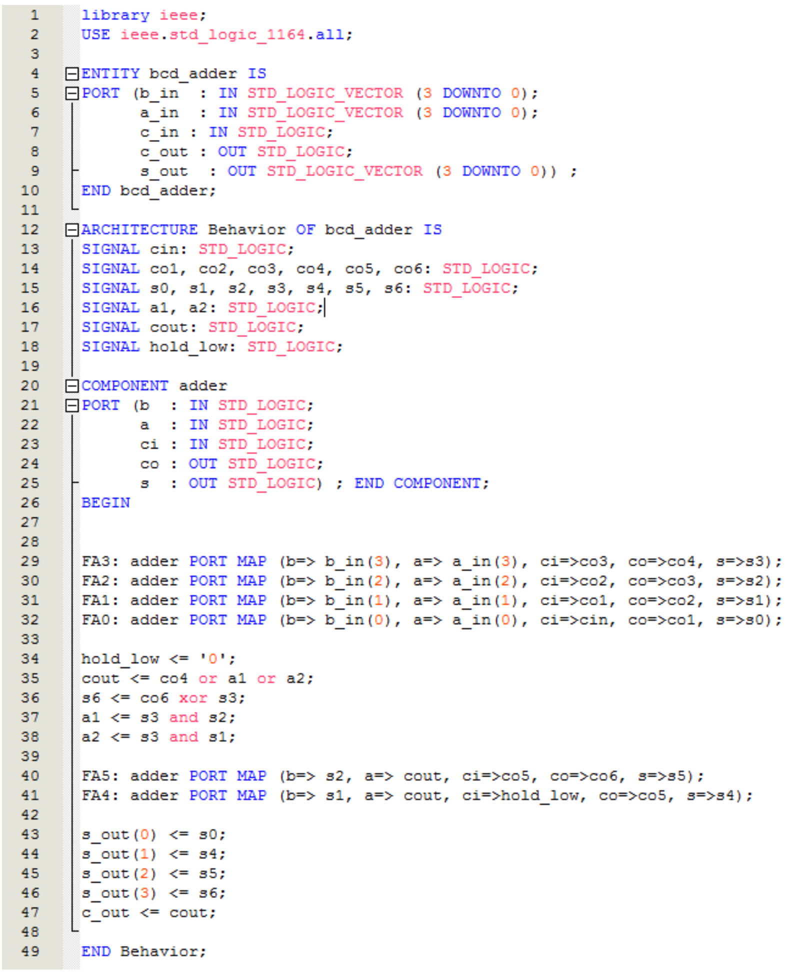


Figure 32

Wired up in code this looks somewhat like Figure 33.

Now all we have to really do is wire things up to give us firstly the bcd_adder. We will have six adders and some "glue". The glue is around a1, a2 cout and s6 of Figure 32.



```
library ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd_adder IS
PORT (b_in   : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      a_in   : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      c_in : IN STD_LOGIC;
      c_out : OUT STD_LOGIC;
      s_out  : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END bcd_adder;

ARCHITECTURE Behavior OF bcd_adder IS
SIGNAL cin: STD_LOGIC;
SIGNAL co1, co2, co3, co4, co5, co6: STD_LOGIC;
SIGNAL s0, s1, s2, s3, s4, s5, s6: STD_LOGIC;
SIGNAL a1, a2: STD_LOGIC;
SIGNAL cout: STD_LOGIC;
SIGNAL hold_low: STD_LOGIC;

COMPONENT adder
PORT (b  : IN STD_LOGIC;
      a  : IN STD_LOGIC;
      ci : IN STD_LOGIC;
      co : OUT STD_LOGIC;
      s  : OUT STD_LOGIC) ; END COMPONENT;
BEGIN


FA3: adder PORT MAP (b=> b_in(3), a=> a_in(3), ci=>co3, co=>co4, s=>s3);
FA2: adder PORT MAP (b=> b_in(2), a=> a_in(2), ci=>co2, co=>co3, s=>s2);
FA1: adder PORT MAP (b=> b_in(1), a=> a_in(1), ci=>co1, co=>co2, s=>s1);
FA0: adder PORT MAP (b=> b_in(0), a=> a_in(0), ci=>cin, co=>co1, s=>s0);

hold_low <= '0';
cout <= co4 or a1 or a2;
s6 <= co6 xor s3;
a1 <= s3 and s2;
a2 <= s3 and s1;

FA5: adder PORT MAP (b=> s2, a=> cout, ci=>co5, co=>co6, s=>s5);
FA4: adder PORT MAP (b=> s1, a=> cout, ci=>hold_low, co=>co5, s=>s4);

s_out(0) <= s0;
s_out(1) <= s4;
s_out(2) <= s5;
s_out(3) <= s6;
c_out <= cout;

END Behavior;
```

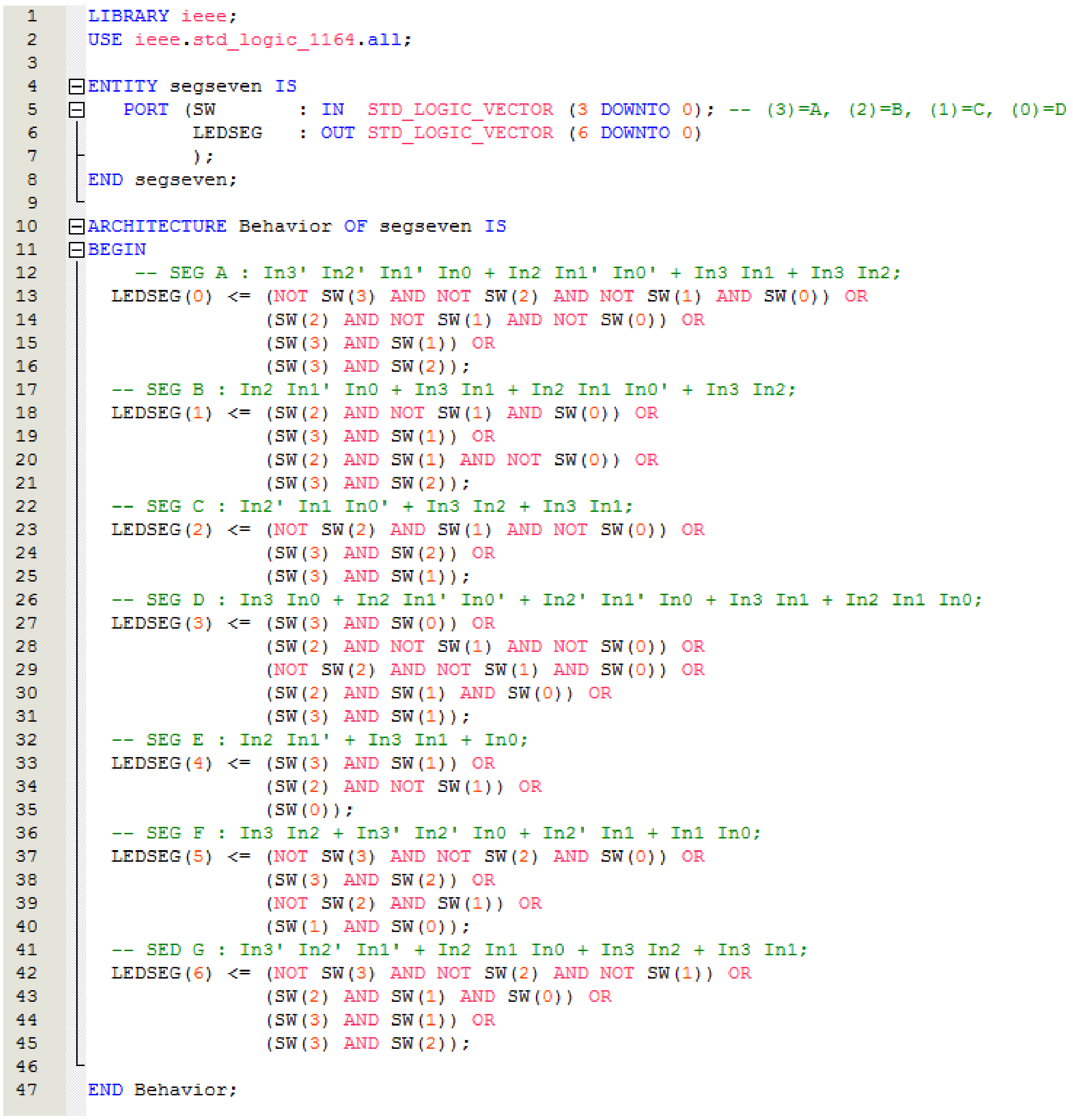
Figure 33

Other code we will need we can steal from Part II and use "circuitb" and "DE1_disp" (including "sweep"). However, we cannot use the segseven code as is as it assumed inputting from the switches which are pulled high by a resistor. We need to re-write that code so that the inputs range from the un-inverted 0000..1001 and not the **inverted** 1111..0110 of Table 1. Suffice to say minimised we get:

Minimized:

```
A = In3' In2' In1' In0 + In2 In1' In0' + In3 In1 + In3 In2;
B = In2 In1' In0 + In3 In1 + In2 In1 In0' + In3 In2;
C = In2' In1 In0' + In3 In2 + In3 In1;
D = In3 In0 + In2 In1' In0' + In2' In1' In0 + In3 In1 + In2 In1 In0;
E = In2 In1' + In3 In1 + In0;
F = In3 In2 + In3' In2' In0 + In2' In1 + In1 In0;
G = In3' In2' In1' + In2 In1 In0 + In3 In2 + In3 In1;
```

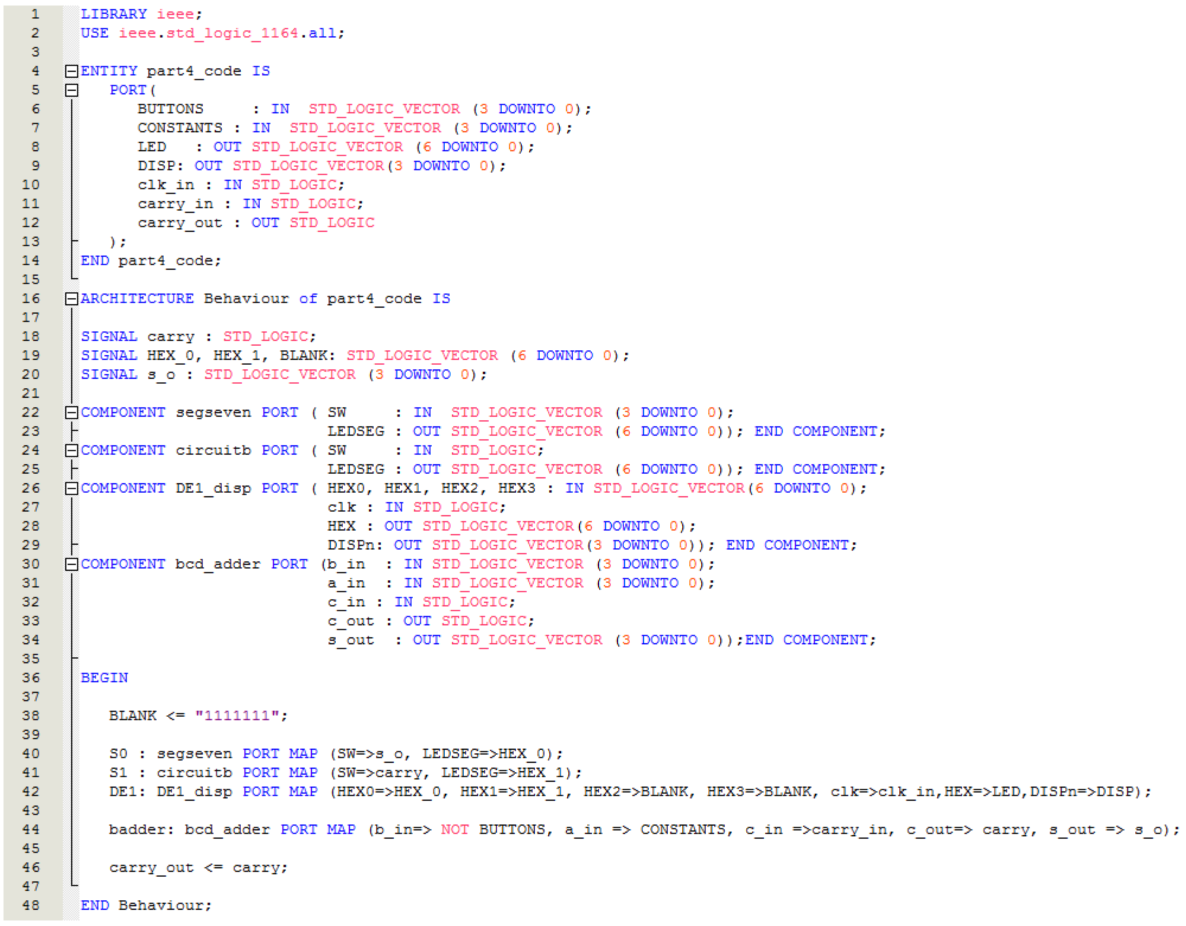
New code for segseven is at Figure 34 (using un-inverted inputs).



```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY segseven IS
   PORT (SW        : IN  STD_LOGIC_VECTOR (3 DOWNTO 0); -- (3)=A, (2)=B, (1)=C, (0)=D
         LEDSEG    : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
         );
END segseven;

ARCHITECTURE Behavior OF segseven IS
BEGIN
    -- SEG A : In3' In2' In1' In0 + In2 In1' In0' + In3 In1 + In3 In2;
  LEDSEG(0) <= (NOT SW(3) AND NOT SW(2) AND NOT SW(1) AND SW(0)) OR
               (SW(2) AND NOT SW(1) AND NOT SW(0)) OR
               (SW(3) AND SW(1)) OR
               (SW(3) AND SW(2));
  -- SEG B : In2 In1' In0 + In3 In1 + In2 In1 In0' + In3 In2;
  LEDSEG(1) <= (SW(2) AND NOT SW(1) AND SW(0)) OR
               (SW(3) AND SW(1)) OR
               (SW(2) AND SW(1) AND NOT SW(0)) OR
               (SW(3) AND SW(2));
  -- SEG C : In2' In1 In0' + In3 In2 + In3 In1;
  LEDSEG(2) <= (NOT SW(2) AND SW(1) AND NOT SW(0)) OR
               (SW(3) AND SW(2)) OR
               (SW(3) AND SW(1));
  -- SEG D : In3 In0 + In2 In1' In0' + In2' In1' In0 + In3 In1 + In2 In1 In0;
  LEDSEG(3) <= (SW(3) AND SW(0)) OR
               (SW(2) AND NOT SW(1) AND NOT SW(0)) OR
               (NOT SW(2) AND NOT SW(1) AND SW(0)) OR
               (SW(2) AND SW(1) AND SW(0)) OR
               (SW(3) AND SW(1));
  -- SEG E : In2 In1' + In3 In1 + In0;
  LEDSEG(4) <= (SW(3) AND SW(1)) OR
               (SW(2) AND NOT SW(1)) OR
               (SW(0));
  -- SEG F : In3 In2 + In3' In2' In0 + In2' In1 + In1 In0;
  LEDSEG(5) <= (NOT SW(3) AND NOT SW(2) AND SW(0)) OR
               (SW(3) AND SW(2)) OR
               (NOT SW(2) AND SW(1)) OR
               (SW(1) AND SW(0));
  -- SED G : In3' In2' In1' + In2 In1 In0 + In3 In2 + In3 In1;
  LEDSEG(6) <= (NOT SW(3) AND NOT SW(2) AND NOT SW(1)) OR
               (SW(2) AND SW(1) AND SW(0)) OR
               (SW(3) AND SW(1)) OR
               (SW(3) AND SW(2));

END Behavior;
```

Figure 34

Now, remember our board does not have enough switches to provide all inputs so we'll need a schematic as our design top element. Before that though we will build most of the circuit up in VHDL. This looks rather like the code for "part4_code" in Figure 35.



```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part4_code IS
   PORT(
      BUTTONS     : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
      CONSTANTS : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
      LED    : OUT STD_LOGIC_VECTOR (6 DOWNTO 0);
      DISP: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
      clk_in : IN STD_LOGIC;
      carry_in : IN STD_LOGIC;
      carry_out : OUT STD_LOGIC
   );
END part4_code;

ARCHITECTURE Behaviour of part4_code IS

SIGNAL carry : STD_LOGIC;
SIGNAL HEX_0, HEX_1, BLANK: STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL s_o : STD_LOGIC_VECTOR (3 DOWNTO 0);

COMPONENT segseven PORT ( SW     : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
                          LEDSEG : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)); END COMPONENT;
COMPONENT circuitb PORT ( SW     : IN  STD_LOGIC;
                          LEDSEG : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)); END COMPONENT;
COMPONENT DE1_disp PORT ( HEX0, HEX1, HEX2, HEX3 : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
                          clk : IN STD_LOGIC;
                          HEX : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
                          DISPn: OUT STD_LOGIC_VECTOR(3 DOWNTO 0)); END COMPONENT;
COMPONENT bcd_adder PORT (b_in  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                          a_in  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                          c_in : IN STD_LOGIC;
                          c_out : OUT STD_LOGIC;
                          s_out  : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));END COMPONENT;

BEGIN

   BLANK <= "1111111";

   S0 : segseven PORT MAP (SW=>s_o, LEDSEG=>HEX_0);
   S1 : circuitb PORT MAP (SW=>carry, LEDSEG=>HEX_1);
   DE1: DE1_disp PORT MAP (HEX0=>HEX_0, HEX1=>HEX_1, HEX2=>BLANK, HEX3=>BLANK, clk=>clk_in,HEX=>LED,DISPn=>DISP);

   badder: bcd_adder PORT MAP (b_in=> NOT BUTTONS, a_in => CONSTANTS, c_in =>carry_in, c_out=> carry, s_out => s_o);

   carry_out <= carry;

END Behaviour;
```

Figure 35

We then create a symbol for "part4_code" as per Figure 36

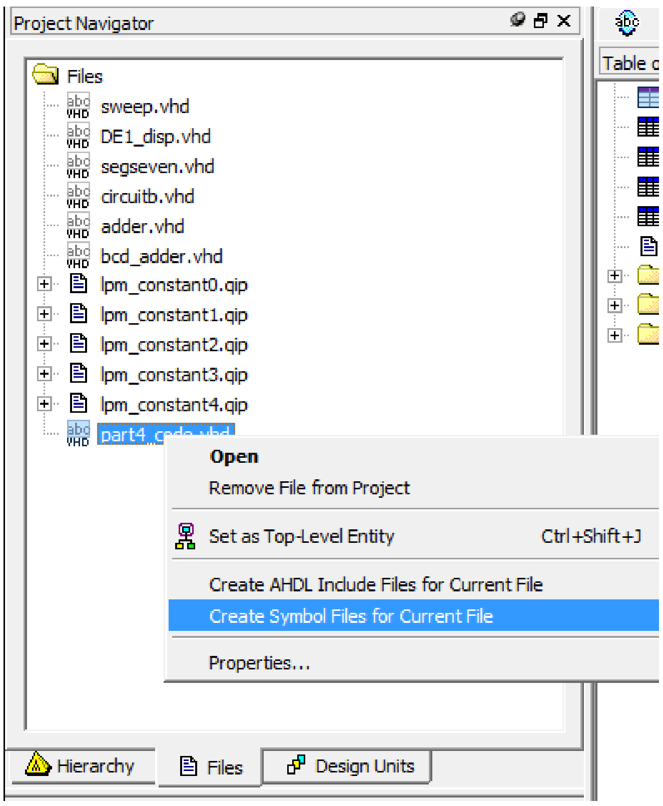


Figure 36

Now we can create the top element "part4" (Figure 37) and to be sure make it the top element (Figure 38).

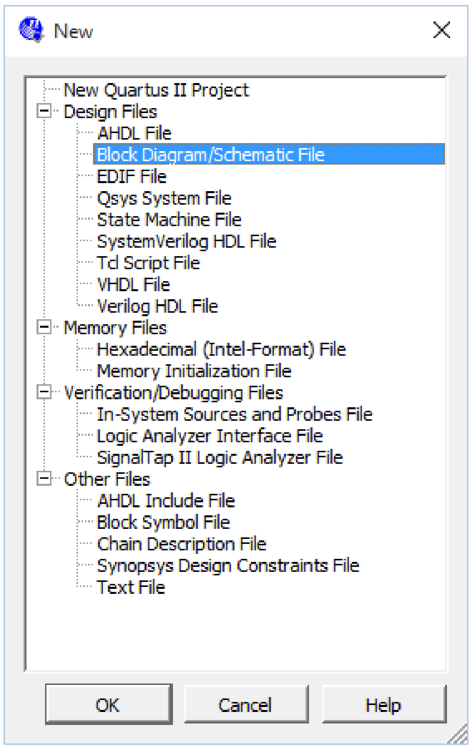
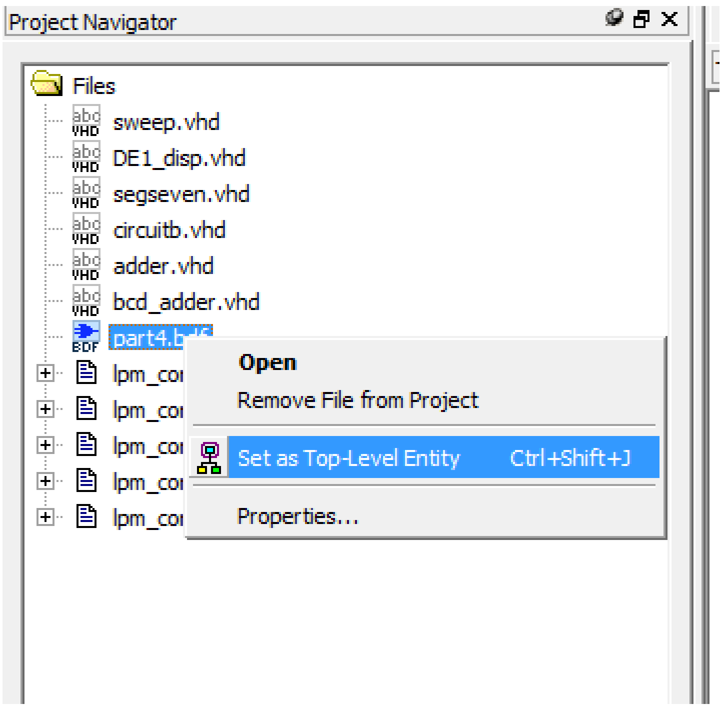


Figure 37

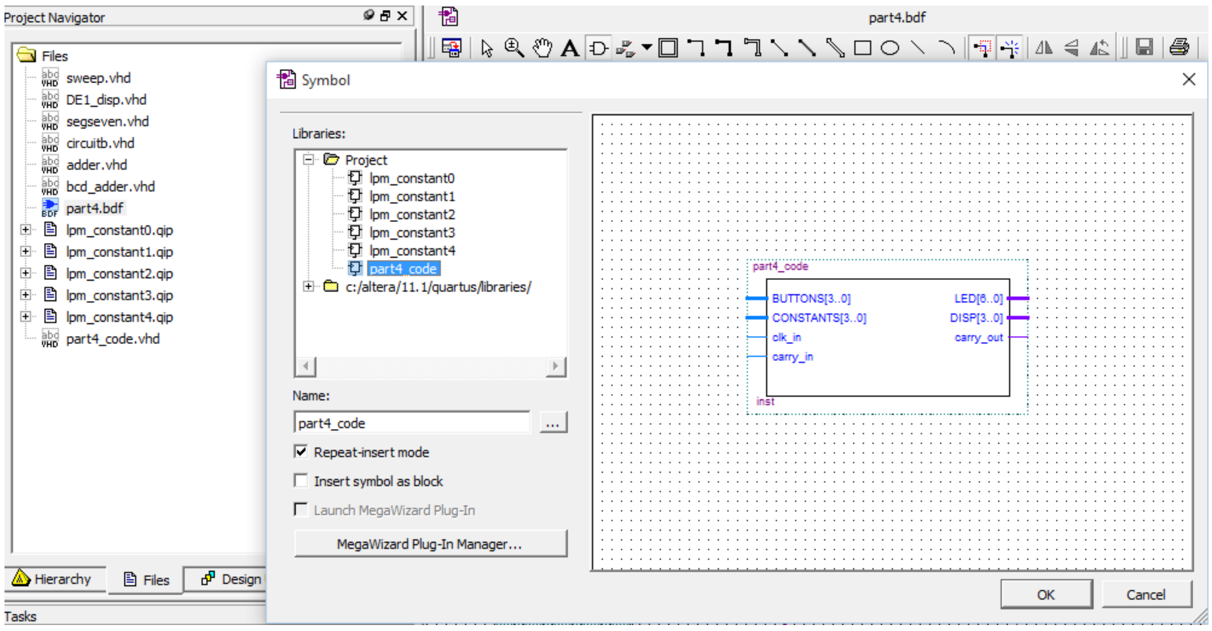

Project Navigator

Files
- sweep.vhd
- DE1_disp.vhd
- segseven.vhd
- circuitb.vhd
- adder.vhd
- bcd_adder.vhd
- part4.bdf
- lpm_co
- lpm_co
- lpm_co
- lpm_co
- lpm_co

Open
Remove File from Project
Set as Top-Level Entity Ctrl+Shift+J
Properties...

Figure 38

Now, we take our symbol for part4_code (Figure 39) and build a circuit around it (Figure 40).


Project Navigator

Files
- sweep.vhd
- DE1_disp.vhd
- segseven.vhd
- circuitb.vhd
- adder.vhd
- bcd_adder.vhd
- part4.bdf
- lpm_constant0.qip
- lpm_constant1.qip
- lpm_constant2.qip
- lpm_constant3.qip
- lpm_constant4.qip
- part4_code.vhd

Hierarchy | Files | Design

Tasks

part4.bdf

Symbol

Libraries:
- Project
  - lpm_constant0
  - lpm_constant1
  - lpm_constant2
  - lpm_constant3
  - lpm_constant4
  - part4_code
- c:/altera/11.1/quartus/libraries/

Name: part4_code

[✓] Repeat-insert mode
[ ] Insert symbol as block
[ ] Launch MegaWizard Plug-In

MegaWizard Plug-In Manager...

OK Cancel

Figure 39

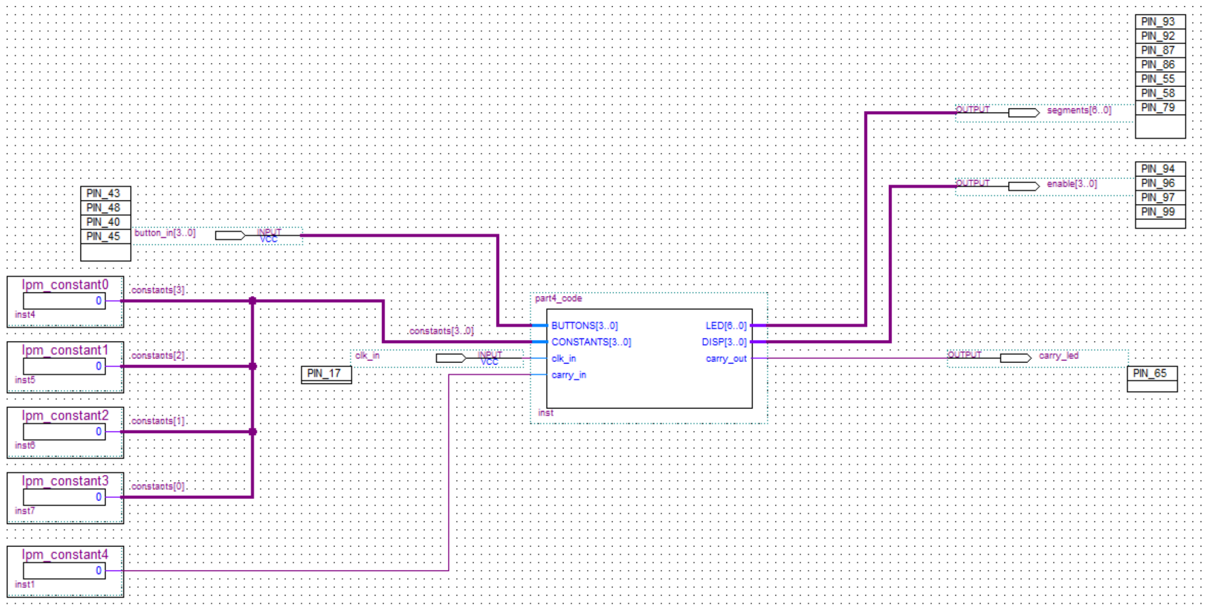
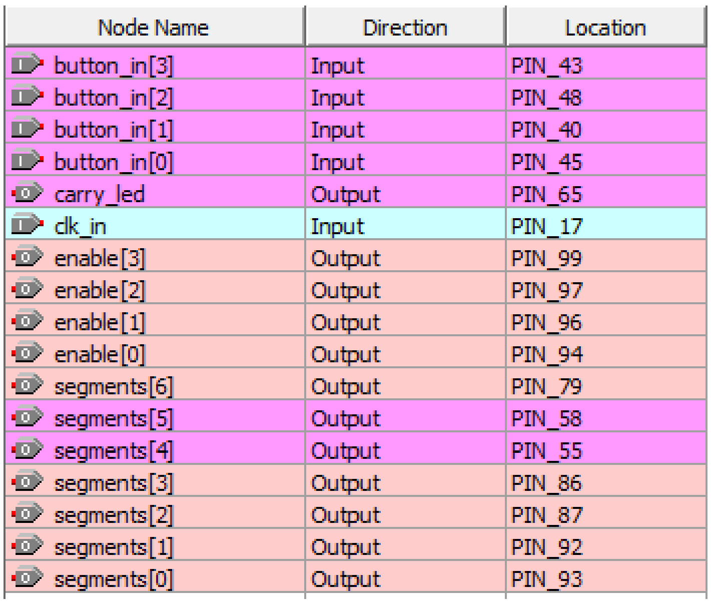


Figure 40

Now after compiling the suggested pins to use are in Figure 41.



Node Name	Direction	Location
button_in[3]	Input	PIN_43
button_in[2]	Input	PIN_48
button_in[1]	Input	PIN_40
button_in[0]	Input	PIN_45
carry_led	Output	PIN_65
clk_in	Input	PIN_17
enable[3]	Output	PIN_99
enable[2]	Output	PIN_97
enable[1]	Output	PIN_96
enable[0]	Output	PIN_94
segments[6]	Output	PIN_79
segments[5]	Output	PIN_58
segments[4]	Output	PIN_55
segments[3]	Output	PIN_86
segments[2]	Output	PIN_87
segments[1]	Output	PIN_92
segments[0]	Output	PIN_93

Figure 41

WOW! Look at the RTL for the bcd_adder at Figure 42! One-4-one of the sketching at Figure 32!

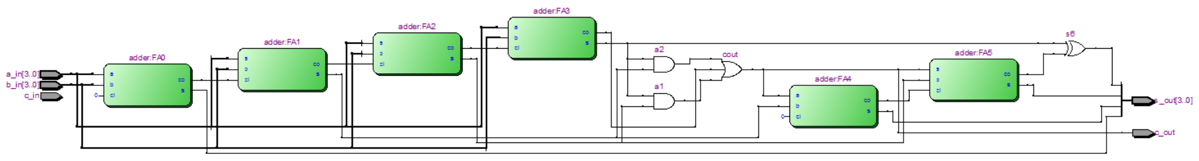


Figure 42

Phew!

Part V

Aim this lab is to use an algorithm for the BCD adder at Figure 43.

```
1   $T_0 = A + B + c_0$
2   if ($T_0 > 9$) then
3       $Z_0 = 10;$
4       $c_1 = 1;$
5   else
6       $Z_0 = 0;$
7       $c_1 = 0;$
8   end if
9   $S_0 = T_0 - Z_0$
10  $S_1 = c_1$
```

Figure 43

So, FPGA and VHDL math! DOH!

Have a read of bitweenie and the discussion on VHDL type conversion. We will likely do something, well, not illegal but clumsy or not preferred. Mostly the argument is we are wanting to fit the BCD adder into a current design.

Otherwise we need convert to "unsigned" and then cast again back to STD_LOGIC_VECTOR:

```
t <= STD_LOGIC_VECTOR(unsigned(a_in) + unsigned(b_in));
```

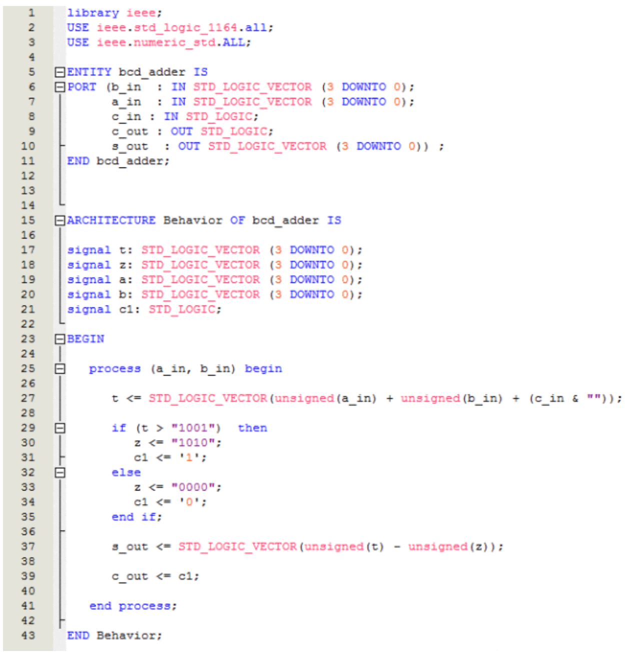
Although there is a "trick" when adding single wire STD_LOGIC signals:

```
t <= STD_LOGIC_VECTOR(unsigned(a_in) + unsigned(b_in) + (c_in & ""));
```

The gem, of course, was the '(c_in & "")'. The ampersand or & is a concatenation operator in VHDL. So, you are building a vector of one or, say {c_in} as opposed to c_in. That is the "" is an empty vector so an empty vector plus an entry is a non-empty vector of one entry. Think of vector then as array.

A good explanation, with examples, is on page 41 of a tutorial on VHDL by Peter Ashenden [4].

So, the long and short of it was the code in Figure 44 over the page.



```vhdl
library ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.ALL;

ENTITY bcd_adder IS
PORT (b_in  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      a_in  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      c_in : IN STD_LOGIC;
      c_out : OUT STD_LOGIC;
      s_out  : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END bcd_adder;



ARCHITECTURE Behavior OF bcd_adder IS

signal t: STD_LOGIC_VECTOR (3 DOWNTO 0);
signal z: STD_LOGIC_VECTOR (3 DOWNTO 0);
signal a: STD_LOGIC_VECTOR (3 DOWNTO 0);
signal b: STD_LOGIC_VECTOR (3 DOWNTO 0);
signal c1: STD_LOGIC;

BEGIN

   process (a_in, b_in) begin

      t <= STD_LOGIC_VECTOR(unsigned(a_in) + unsigned(b_in) + (c_in & ""));

      if (t > "1001")  then
         z <= "1010";
         c1 <= '1';
      else
         z <= "0000";
         c1 <= '0';
      end if;

      s_out <= STD_LOGIC_VECTOR(unsigned(t) - unsigned(z));

      c_out <= c1;

   end process;

END Behavior;
```

Figure 44

So, starting with the project for Part IV we simply replace the code for the "bcd_adder" with that at Figure 44 above. You can drop the code for the "adder" as all the functionality is in "bcd_adder" without having to use the "adder" component used in Part IV.

Part VI

You may now SCREAM!!