EE 3420 Lab Guide 2: Keypad to DTMF w/ NIOS

Written by: Grant Seligman, Gabe Garves, and James Starks

Example Overview:

Design a circuit that will take in a keypad input and output to two tone frequencies to a pair of buzzers. We will do this by having the NIOS II softcore CPU take in data from the keypad and send it out to a selector module. Depending on the button pressed the selector module will output the correct tones to a set of buzzers. The tone duration will be controlled by the NIOS by enabling a pio connection to the selector module.

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*/E	0	#/F	D

Figure 1

Though these are the frequencies used in this lab, you could actually set up your own frequency values as well. Though you may need to find a speaker set that could handle frequencies under 1000Hz or above 2400Hz.

This guide will show you how to set up the tones for keypad row 0. It will be up to you to create the rest of the tone clock divider modules and connect them to the full selector module.

Verilog Code Breakdown:

To begin, you will need the keypad module from the last lab. But we have provided it for you here in **Figure 2.**

```
27  module keypad_decoder(key, row, column, clk);
28      output reg [3:0] key;
29      output reg [3:0] row;
30
31      input wire clk;
32      input wire [3:0] column;
33
34      always @ (posedge clk) begin
35          case (row)
36              4'b1110:
37                  case (column)
38                      4'b1110 : key = 4'b0001;
39                      4'b1101 : key = 4'b0010;
40                      4'b1011 : key = 4'b0011;
41                      4'b0111 : key = 4'b1010;
42                      default : row = 4'b1101;
43                  endcase
44              4'b1101:
45                  case (column)
46                      4'b1110 : key = 4'b0100;
47                      4'b1101 : key = 4'b0101;
48                      4'b1011 : key = 4'b0110;
49                      4'b0111 : key = 4'b1011;
50                      default : row = 4'b1011;
51                  endcase
52              4'b1011:
53                  case (column)
54                      4'b1110 : key = 4'b0111;
55                      4'b1101 : key = 4'b1000;
56                      4'b1011 : key = 4'b1001;
57                      4'b0111 : key = 4'b1100;
58                      default : row = 4'b0111;
59                  endcase
60              4'b0111:
61                  case (column)
62                      4'b1110 : key = 4'B1110;
63                      4'b1101 : key = 4'b0000;
64                      4'b1011 : key = 4'b1111;
65                      4'b0111 : key = 4'b1101;
66                      default : row = 4'b1110;
67                  endcase
68              default: row = 4'b1110;
69          endcase
70      end
71  endmodule
```

Figure 2

Next we will need to create the tone clock divider modules for row 0. We will take an input clock signal of 12Mhz, put it through a Phase Lock Loop (PLL) to step this signal down to 1Mhz, then write custom modules to step down to the frequencies we need.

Basically, we are creating a set of clock converters. We will cover the PLL in the **FPGA Implementation section**, lets first create the clock converters.

```
1   //Stepdown To 697Hz
2
3   /*
4   AUTHOR: JAMES STARKS
5   DATE: 4/17/2020
6   FROM: TXST SENIOR DESIGN FALL 2019-SPRING 2020
7   FOR: TEXAS STATE UNIVERITY STUDENT AND INSTRUCTOR USE
8   */
9   module stepdown_697Hz(out_clk,in_clk)
10      output reg out_clk;
11      input wire in_clk;
12
13      //Take in_clk frequency and divided it by the frequency
14      //you want. Then divide the quotient by 2 to get div value.
15      integer div = 716;
16
17      //Register size depends div value.
18      //Take the logbase 2 of div value to get count.
19      //Make sure to round the bit value up.
20      reg [9:0] count;
21
22      initial
23      begin
24          count = 9'b0;
25          outclk = 0;
26      end
27
28      //Count increases with every positive clock edge
29      //until the if condition is met the count will reset
30      always@(posedge inclk)
31      begin
32          count = count + 1;
33          if (count == div)
34          begin
35              outclk = ~outclk;
36              count = 0;
37          end
38      end
39  endmodule
40
```

Figure 4

You will need to create this step down code for each frequency specified for the lab. It's tedious but that is why copy and paste exists.

The first thing you need to know is how many divisions out of the 1MHz clock signal it takes to reduce it down to 697Hz. Your numbers may vary from the guides because we tried to get the tones as close as possible to real phone dial tones.

$$Divisions = \frac{Input\ Frequency}{(2 \times Desired\ Frequency)}$$

$$Divisions = \frac{Input\ Frequency}{(2 \times Desired\ Frequency)} \approx 716$$

The 2 in the divisor takes into account that for every clock cycle, there is a positive edge and a negative edge. Since the always block is running on just the positive edge, the code activates at half the clock cycle.

Next you will have to calculate the number of bits that the counter register has to store in order to reach the division value.

$$¿of\ Bits = \log_2(Divisions)$$
$$¿of\ Bits = \log_2(716) \approx 10\ Bits$$

Make sure to round up your answer for the bits because you can't store a decimal value of bits, only whole bits.

Now go ahead and make modules for the rest of the frequencies as shown in Figure 1. Once you are done creating those modules, you will need to make a 60Hz module. This will be for the keypad and it will be explained later why you need it.

Next we will need to create a selector module that will have all the clock divider modules, the keypad, and an enable as input and will output the two tones needed to create the tones we want. Then in the next section we will connect everything together with NIOS. **See Figures 5-7** for the selector code.

```
1  //Selector Module
2  /*
3  AUTHOR: JAMES STARKS
4  DATE: 4/17/2020
5  FROM: TXST SENIOR DESING FALL 2019-SPRING 2020
6  FOR: TEXAS STATE UNIVERSITY STUDENT AND INSTRUCTOR USE
7  */
8  module select(out_row, out_column, select, en, row_0, row_1, row_2, row_3, column_0, column_1, column_2, column_3);
9      output reg out_row;
10     output reg out_column;
11
12     wire [3:0] rows;
13     input wire row_0;
14     input wire row_1;
15     input wire row_2;
16     input wire row_3;
17
18     wire [3:0] columns;
19     input wire column_0;
20     input wire column_1;
21     input wire column_2;
22     input wire column_3;
23
24     //keypad button input from NIOS
25     input wire [3:0] select;
26     //enable input
27     input wire en;
28     //Setting up an array of inputs making them
29     //easier to call in the case statements
30     assign rows = {row_3, row_2, row_1, row_0};
31     assign columns = {column_3, column_2, column_1, column_0};
32
```

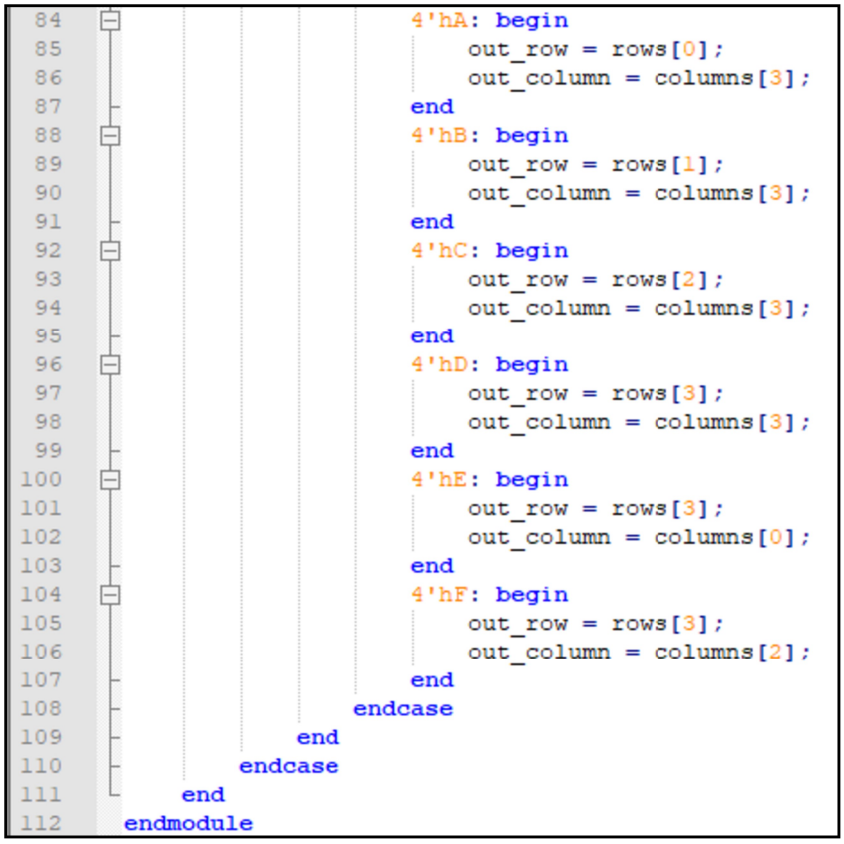
Figure 5

This module is quite large but using case statements is the most efficient in hardware instead of if/if-else/else statements.

The selector takes in all the clock converter inputs and waits for the enable and keypad value from NIOS. Once the enable is triggered, the keypad value is read and the row and column tones are sent out to the two buzzers. NIOS will flip off the enable after a set amount of time so buzzers don't run continuously. More on this later.

```
33      //Each keypad 4-bit input represents a case.
34      //Each case will output a set of frequencies based on the
35      //row and column frequency inputs.
36      always@(en) begin
37          case(en)
38              1'b0: begin
39                  out_row = 0;
40                  out_column = 0;
41              end
42              1'b1: begin
43                  case(select)
44                      4'h0: begin
45                          out_row = rows[3];
46                          out_column = columns[1];
47                      end
48                      4'h1: begin
49                          out_row = rows[0];
50                          out_column = columns[0];
51                      end
52                      4'h2: begin
53                          out_row = rows[0];
54                          out_column = columns[1];
55                      end
56                      4'h3: begin
57                          out_row = rows[0];
58                          out_column = columns[2];
59                      end
60                      4'h4: begin
61                          out_row = rows[1];
62                          out_column = columns[0];
63                      end
64                      4'h5: begin
65                          out_row = rows[1];
66                          out_column = columns[1];
67                      end
68                      4'h6: begin
69                          out_row = rows[1];
70                          out_column = columns[2];
71                      end
72                      4'h7: begin
73                          out_row = rows[2];
74                          out_column = columns[0];
75                      end
76                      4'h8: begin
77                          out_row = rows[2];
78                          out_column = columns[1];
79                      end
80                      4'h9: begin
81                          out_row = rows[2];
82                          out_column = columns[2];
83                      end
```

Figure 6



```
84                      4'hA: begin
85                          out_row = rows[0];
86                          out_column = columns[3];
87                      end
88                      4'hB: begin
89                          out_row = rows[1];
90                          out_column = columns[3];
91                      end
92                      4'hC: begin
93                          out_row = rows[2];
94                          out_column = columns[3];
95                      end
96                      4'hD: begin
97                          out_row = rows[3];
98                          out_column = columns[3];
99                      end
100                     4'hE: begin
101                         out_row = rows[3];
102                         out_column = columns[0];
103                     end
104                     4'hF: begin
105                         out_row = rows[3];
106                         out_column = columns[2];
107                     end
108                 endcase
109             end
110         endcase
111     end
112 endmodule
```

Figure 7

Once all your modules have been made or copied into Quartus. Make each one into a Symbol File and then continue onto the FPGA Implementation section.

FPGA Implementation:

For this project you will need to create a NIOS platform with 3 **PIO** modules, one for a **4-bit** input from the **keypad decoder** and another 4-bit output to the **DTMF tone selector**. The final **PIO** will be used to control the **enable** pin on the **tone selector**. See **Figure 5** to see how to configure the Platform Designer with the NIOS processor. Make sure to resolve all errors and warnings before generating.

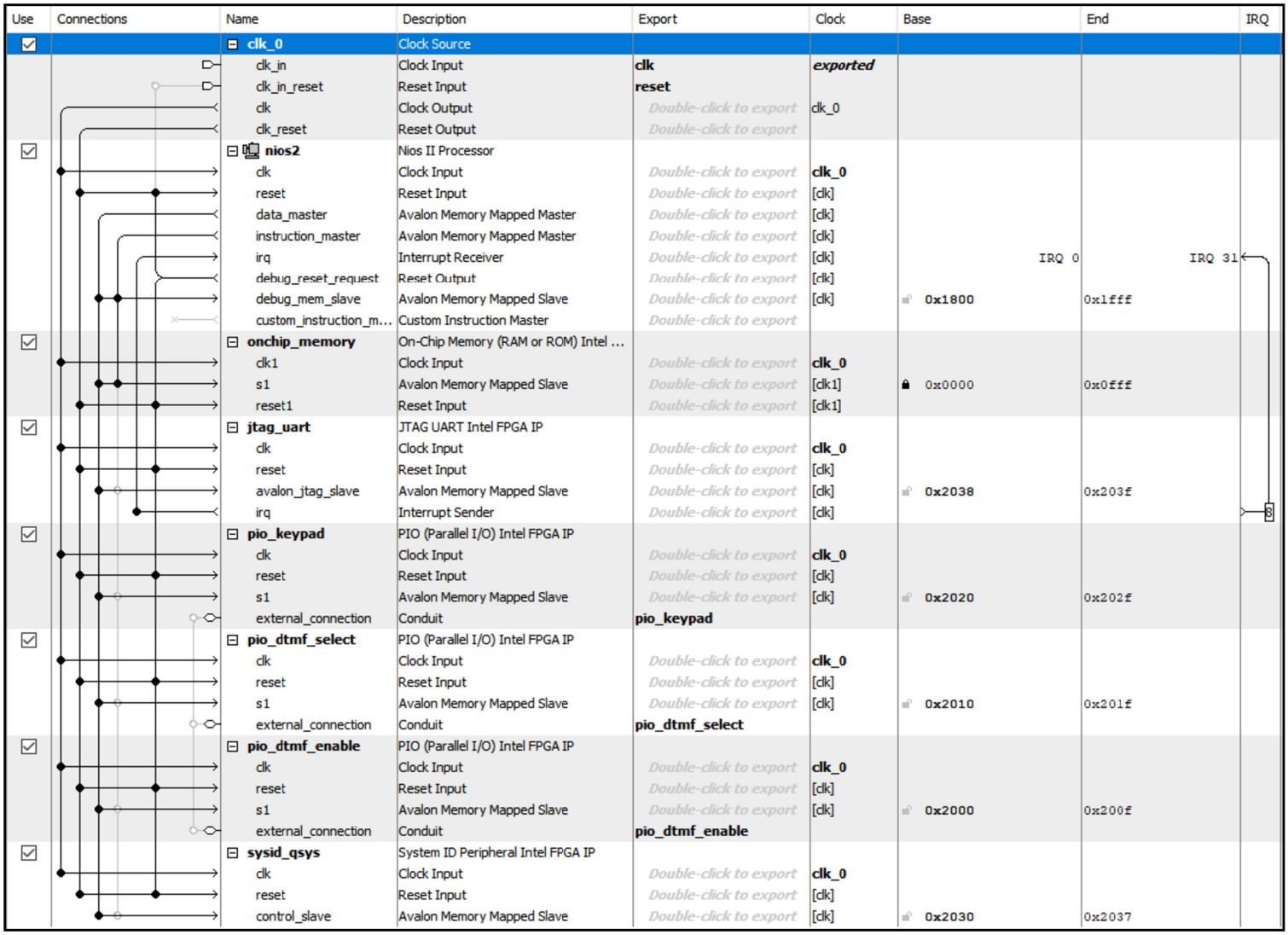


Figure 5

Now we must add a PLL to the block diagram to help with tone generation. We will use a built in PLL to drop the clock from 12MHz to 1MHz. This will help improve the reliability of the clock dividers that will be used to generate a tone.

Search in the IP catalog for “**ALTPLL**” and launch the wizard. Follow the images below to see which settings to apply.

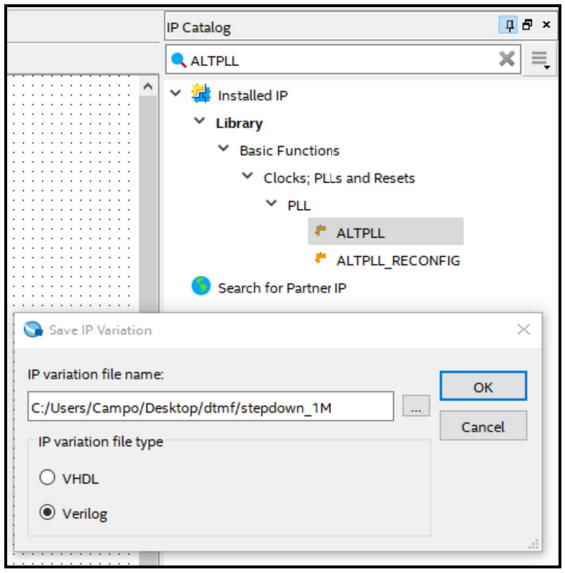


Figure 6

Figure 7 will be the first window you will see. For the PLL, you will need an input clock signal. This will be set to 12Mhz because we will pull from the Max1000’s on-board clock.

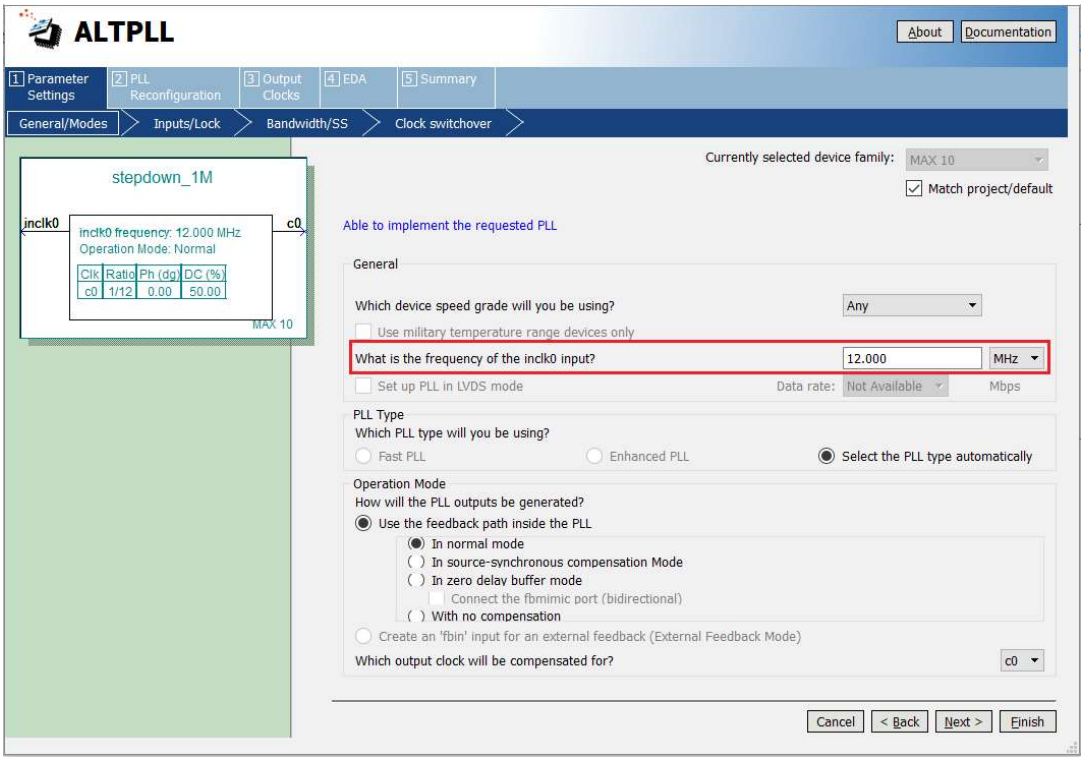


Figure 7

Next you will need to set the output clock. For this lab, it will be 1MHz. As a note, the lowest the PLL can step down to is 10kHz. You are welcome to drop it down to 10KHz but you will have to recalculate your values for your clock converter modules. To set the output, select **Output Clocks** tap at the top, select the **"Enter output clock frequency",** and type in the desired frequency as shown in **Figure 8.**

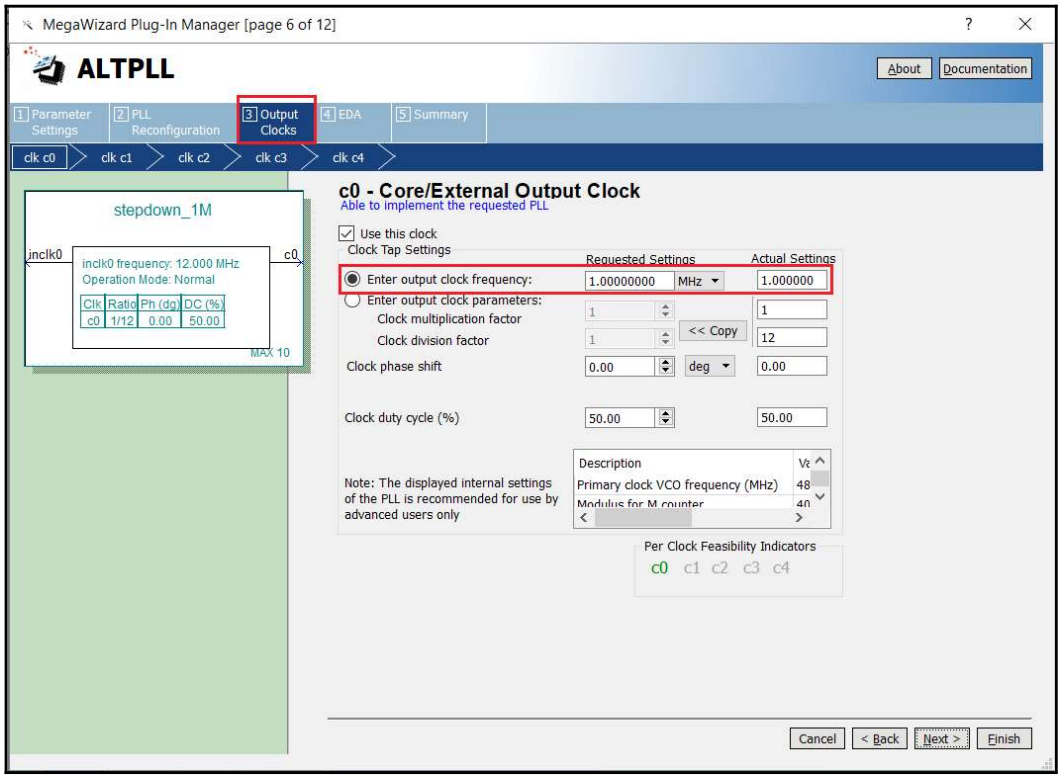


Figure 8

A single PLL module can have up to 5 different clock outputs. PLLs are very useful if you need a very stable clock frequency. They can also be used to step up the original on-board clock frequency but for this lab, this is unnecessary.

To finish, click on the **Summary** tab at the top of the PLL menu and make sure the three file options are selected as shown in **Figure 9.** Then select **Finish.**

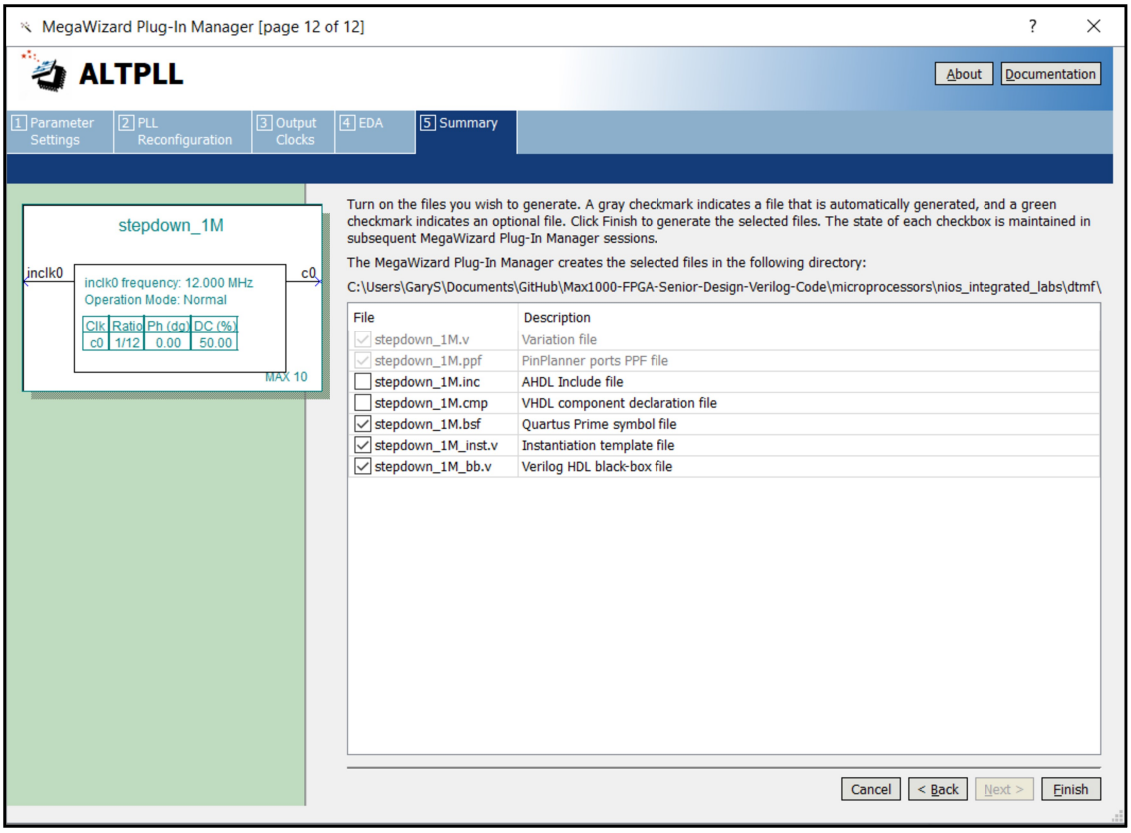


Figure 9

*The PLL manager will create all the necessary files including the symbol file needed to use the PLL in the **Block Diagram file (.bdf).***

Here is a list of files you will have to **include** in your project and **create symbol files** of...

- select.v

- keypad_decoder
- stepdown_60Hz.v
- stepdown_697Hz.v
- stepdown_770Hz.v
- stepdown_852Hz.v
- stepdown_941Hz.v
- stepdown_1209Hz.v
- stepdown_1336Hz.v
- stepdown_1477Hz.v
- stepdown_1633Hz.v

Now you will need to create a **Block Diagram** file so you can drop in all of your symbol files. You will first start off by connecting the output of the PLL into the input of each separate verilog clock divider. Then the output of each clock divider will go into the selector row and column inputs. Each of these stepped down frequencies will then be sent to the selector module which uses the output from the NIOS to select two frequencies to play.

The 60Hz clock divider module will be connected to the keypad module. This will allow the keypad to scan for any button inputs 15 times a second. The keypad module has some issues running at higher frequencies so it is best to run it under 100Hz.

Now go ahead and add **NIOS** into your **Block Diagram** file. Connect the keypad[3..0], select[3..0], the 12MHz clock and reset to NIOS. Then connect the **DTMF enable** from NIOS to **en** input on the selector. Use **Figure 10** as your connection guide.

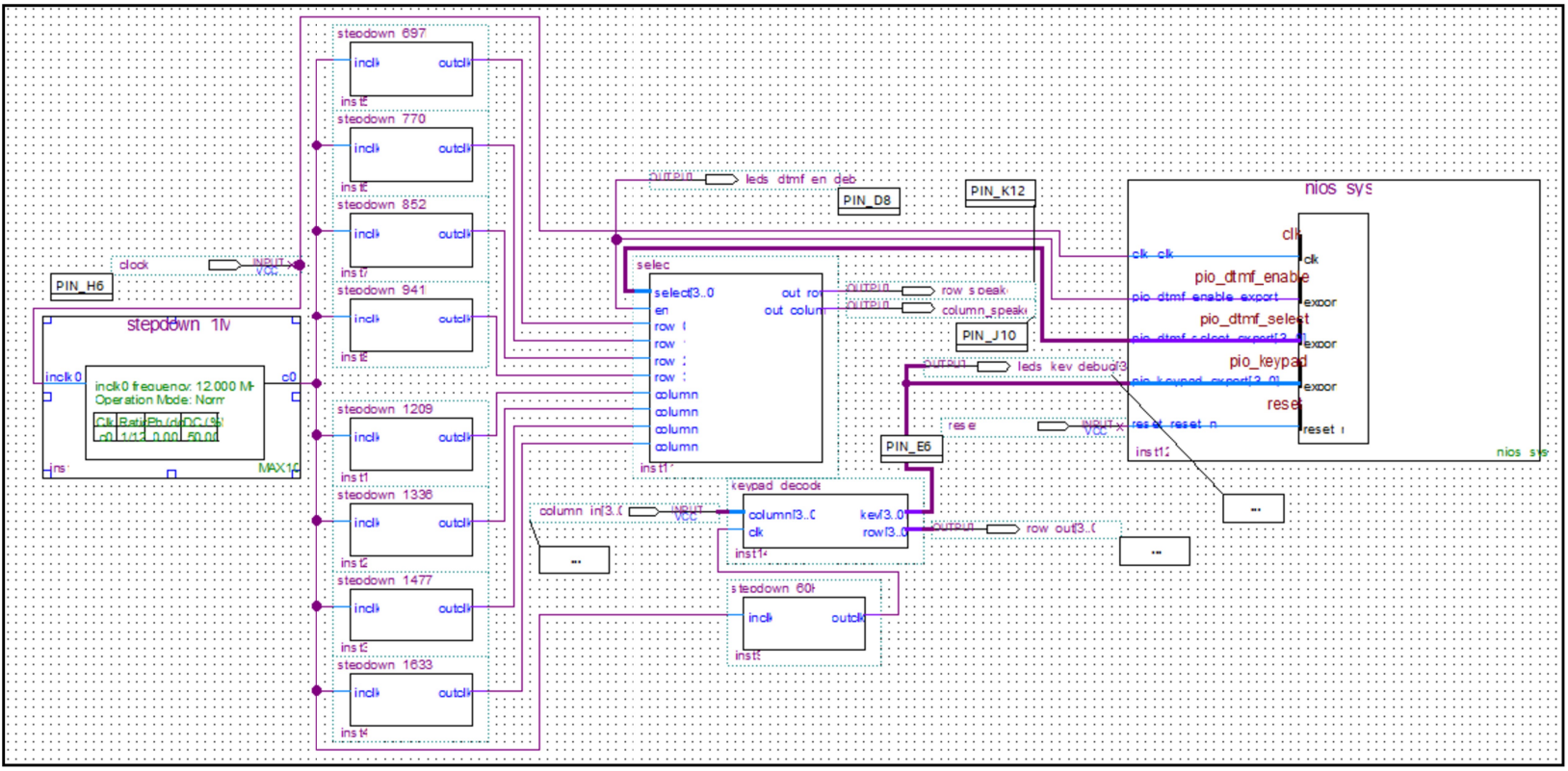


Figure 10

Before compiling make sure these additional Quartus settings are applied…

- **Assignments>>Device>>Device and Pin Options>>Configuration** set **Configuration mode** to "**Single Uncompressed Image w/ Memory Initialization**."
- Also in the **Device and Pin Options** window set **Unused Pins** to "**As input tri-stated**," and under **Voltage**, **Default I/O standard** to "**3.3-V LVTTL**."
- **Assignments>>Settings>>Operating Settings and Conditions** and set both **VCCA voltage** and **VCC_ONE voltage** to "**3.3V**."

Now, run the Quartus compiler to generate input/output nodes in the **Assignments>>Pin Planner** so you can now set the **pin locations**. Use **Figure 11 and 12** as a guide to set your pins. It may prove useful to look back at the Keypad to Seven Segment Lab and review how the keypad is connected. Note: the 12MHz Clock is located on PIN_H6. Below in **Figure 12** is an example of a completely planned out pin assignment.

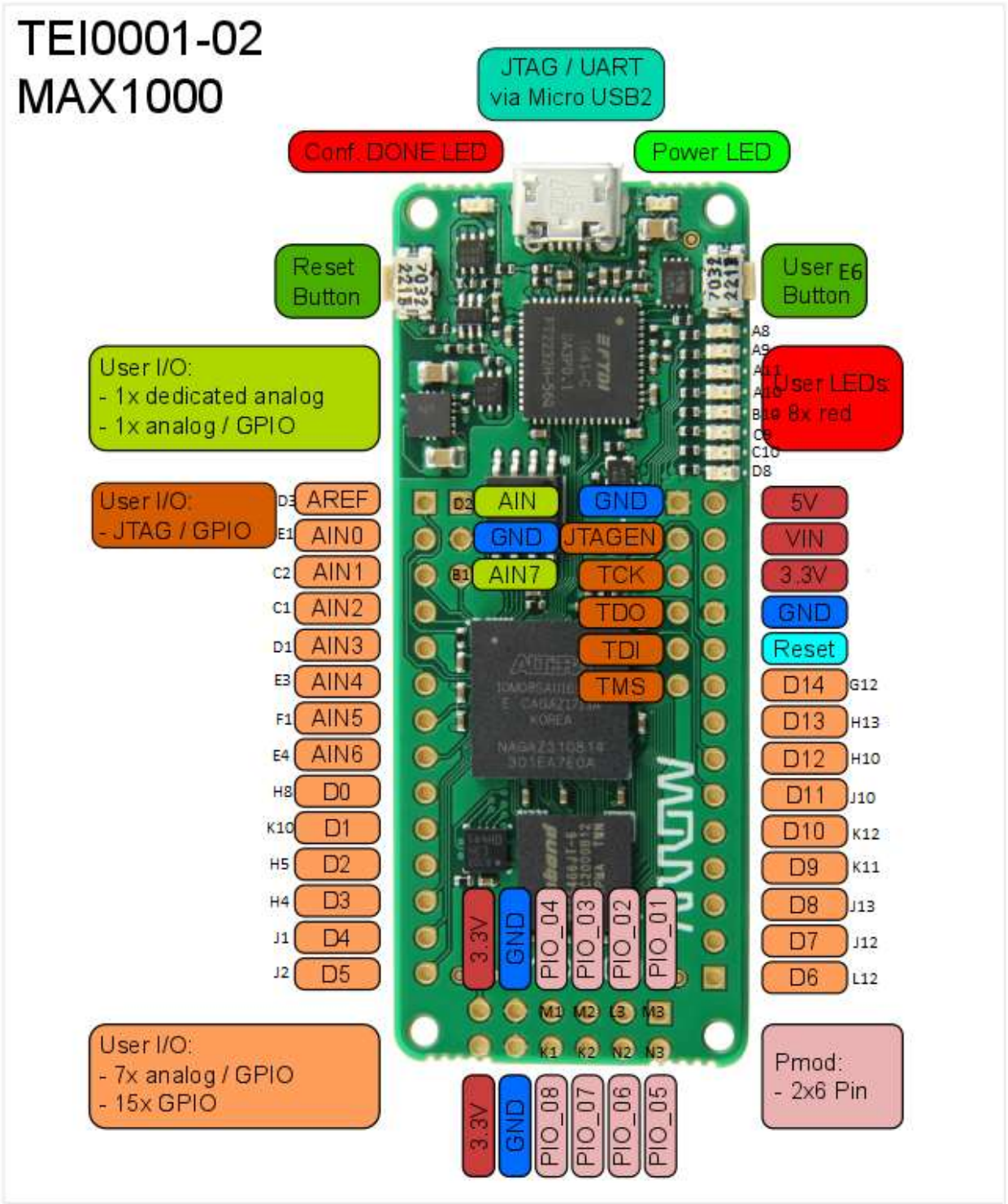

TEI0001-02
MAX1000
JTAG / UART
via Micro USB2
Conf. DONE LED
Power LED
Reset
Button
User E6
Button
User I/O:
- 1x dedicated analog
- 1x analog / GPIO
User LEDs:
8x red
User I/O:
- JTAG / GPIO
AREF
AIN0
AIN1
AIN2
AIN3
AIN4
AIN5
AIN6
D0
D1
D2
D3
D4
D5
AIN
GND
JTAGEN
AIN7
TCK
TDO
TDI
TMS
5V
VIN
3.3V
GND
Reset
D14
D13
D12
D11
D10
D9
D8
D7
D6
3.3V
GND
PIO_04
PIO_03
PIO_02
PIO_01
User I/O:
- 7x analog / GPIO
- 15x GPIO
3.3V
GND
PIO_08
PIO_07
PIO_06
PIO_05
Pmod:
- 2x6 Pin

Figure 11

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate
altera_reserved_tck	Input	PIN_G2	1B	B1_N0	PIN_G2	3.3 V Sc... Trigger		8mA (default)	
altera_reserved_tdi	Input	PIN_F5	1B	B1_N0	PIN_F5	3.3 V Sc... Trigger		8mA (default)	
altera_reserved_tdo	Output	PIN_F6	1B	B1_N0	PIN_F6	3.3-V LVTTL		8mA (default)	2 (default)
altera_reserved_tms	Input	PIN_G1	1B	B1_N0	PIN_G1	3.3 V Sc... Trigger		8mA (default)	
clock	Input	PIN_H6	2	B2_N0	PIN_H6	3.3-V LVTTL		8mA (default)	
column_in[3]	Input	PIN_K10	5	B5_N0	PIN_K10	3.3-V LVTTL		8mA (default)	
column_in[2]	Input	PIN_H5	2	B2_N0	PIN_H5	3.3-V LVTTL		8mA (default)	
column_in[1]	Input	PIN_J1	2	B2_N0	PIN_J1	3.3-V LVTTL		8mA (default)	
column_in[0]	Input	PIN_J2	2	B2_N0	PIN_J2	3.3-V LVTTL		8mA (default)	
column_speaker	Output	PIN_J10	5	B5_N0	PIN_J10	3.3-V LVTTL		8mA (default)	2 (default)
leds_dtmf_en_debug	Output	PIN_D8	8	B8_N0	PIN_D8	3.3-V LVTTL		8mA (default)	2 (default)
leds_key_debug[3]	Output	PIN_A8	8	B8_N0	PIN_A8	3.3-V LVTTL		8mA (default)	2 (default)
leds_key_debug[2]	Output	PIN_A9	8	B8_N0	PIN_A9	3.3-V LVTTL		8mA (default)	2 (default)
leds_key_debug[1]	Output	PIN_A11	8	B8_N0	PIN_A11	3.3-V LVTTL		8mA (default)	2 (default)
leds_key_debug[0]	Output	PIN_A10	8	B8_N0	PIN_A10	3.3-V LVTTL		8mA (default)	2 (default)
reset	Input	PIN_E6	8	B8_N0	PIN_E6	3.3-V LVTTL		8mA (default)	
row_out[3]	Output	PIN_K11	5	B5_N0	PIN_K11	3.3-V LVTTL		8mA (default)	2 (default)
row_out[2]	Output	PIN_J13	5	B5_N0	PIN_J13	3.3-V LVTTL		8mA (default)	2 (default)
row_out[1]	Output	PIN_J12	5	B5_N0	PIN_J12	3.3-V LVTTL		8mA (default)	2 (default)
row_out[0]	Output	PIN_L12	5	B5_N0	PIN_L12	3.3-V LVTTL		8mA (default)	2 (default)
row_speaker	Output	PIN_K12	5	B5_N0	PIN_K12	3.3-V LVTTL		8mA (default)	2 (default)

Figure 12

After setting pins, recompile the project and a **<project_name>_time_limited.sof** file will generate in the **/output_files** directory. This is because the current Quartus license requires the FPGA to remain connected to the computer when NIOS IPs are running on the FPGA.

NIOS II Setup with Eclipse:

You will find the software side of the project is very similar to the Keypad to Seven Segment Display Lab. Like before the program will read data from the keypad decoder module, store it, then pass it on to the selector module. But now you will need to switch the enable high for the tone to play. In the sample code below, **Figure 13,** the selector was enabled for 100ms and then turned off, so the tone isn't played continuously.

Like before, start to create a **New NIOS II Application and BSP from template** and select the **Small Hello World** template. Next generate the **BSP** project and you can start editing the application project. **Would recommend going back and following the steps in the Keypad to Seven Segment if you are having trouble with the Eclipse setup.**

main.c | system.h | alt_stdio.h | stdio.h

```
1  //NIO II C Program Simple Data Allocation
2  /*
3   * AUTHOR: JAMES STARKS
4   * DATE: 4/20/2020
5   * FROM: TXST SENIOR DESIGN PROJECT FALL 2019-SPRING 2020
6   * FOR: TEXAS STATE UNIVERSITY STUDENTS AND INSTRUCTOR USE
7   * DESCRIPTION: Keypad controlled DTMF Lab example.
8   */
9
10 #include <stdio.h>
11 #include "unistd.h"
12 #include "system.h"
13 #include "altera_avalon_pio_regs.h"
14
15 int main()
16 {
17     // Used for storing keypad value
18     unsigned char key = 0;
19     unsigned char old_key = 0;
20
21     // Program loop
22     while(1)
23     {
24         // Store value in PIO_KEYPAD_BASE address into key variable.
25         key = IORD_ALTERA_AVALON_PIO_DATA(PIO_KEYPAD_BASE);
26
27         // Check if a new key has been pressed.
28         if(old_key == key) continue;
29         else
30         {
31             // Update and print
32             old_key = key;
33             printf("%i\t", key);
34             // Write key data to DTMF selector module
35             IOWR_ALTERA_AVALON_PIO_DATA(PIO_DTMF_SELECT_BASE, key);
36             // Enable DTMF module for 100ms.
37             IOWR_ALTERA_AVALON_PIO_DATA(PIO_DTMF_ENABLE_BASE, 1);
38             usleep(100000);
39             IOWR_ALTERA_AVALON_PIO_DATA(PIO_DTMF_ENABLE_BASE, 0);
40         }
41
42     }
43
44     return 0;
45 }
46
```

Figure 13

Finally, **build** your application project and then run as **NIOS II Hardware**.