Go Board and Lattice iCECube2 Installation Notes Peter Halverson Last update 3/15/2020
http://halverscience.net/index.html

NanLand Go Board Notes, including how to setup Lattice iCECube2 on Linux Ubuntu 16

These are Halverson's personal notes. They are unedited and there is no guarantee of accuracy. Use at your own risk. I am making them available to you because they might actually be helpful. If they are helpful, I would really appreciate a "thank you".

Link to the live version of these notes:
https://docs.google.com/document/d/17WlW-UdGRF7cPRGuRcYEWLN411LGdGJSxkyKU7m-21c/edit?usp=sharing

Go Board Index Page https://www.nandland.com/goboard/index.html **** Start here

NandLand home page: https://www.nandland.com/

Go Board home page: https://www.nandland.com/goboard/

GoBoard basic specs:
----Clock = 25 MHz
----Voltage = 3.3 V
----Memory: 64kBits Number of RAM4k Blocks: 16
----FPGA Device Family = iCE40, Device = HX1K, Device Package = VQ100
----The "Home page" for the iCE40 FPGA is here: http://www.latticesemi.com/Products/FPGAandCPLD/iCE40.aspx
The "Application Notes" link at the bottom of the page is useful.

Power consumption, running a DDS sine wave generator
GoBoard + DA3 25 mA
GoBoard alone 20 mA

---------- Communication with the Go Board via USB. ----------
I got this info from the comments section of the You Tube Video "Nandland Go Board - Watch This When You Receive Your Board" at https://www.youtube.com/watch?time_continue=123&v=wWMIY9kjlJ0

FOR LINUX USERS. For recent linux (over Ubuntu 11.10, kernel 3.0.0-19) (mine is kernel 4.4.0-140-generic based on ubuntu 16.04) you just need:
1. sudo apt-get install gtkterm
2. connect your go-board
3. sudo gtkterm -p /dev/ttyUSB1 -s 115200 -b 8 -t 1 <---------- YES ********

You DO NOT NEED to install any drivers; they already come with the kernel and they recognize the go-board (FTDI serial port).

BOTH ports /dev/ttyUSB0 and dev/ttyUSB1 are seen by:
dmesg | grep FTDI
that answers:
[5997.028880] usb 2-3: Manufacturer: FTDI
[5997.029520] ftdi_sio 2-3:1.0: FTDI USB Serial Device converter detected
[5997.030774] usb 2-3: FTDI USB Serial Device converter now attached to ttyUSB0
[5997.031048] ftdi_sio 2-3:1.1: FTDI USB Serial Device converter detected
[5997.031462] usb 2-3: FTDI USB Serial Device converter now attached to ttyUSB1

BUT only /dev/ttyUSB1 worked for me.

---------- Continuing with setting up the programming environment ----------
I am now using the info here: https://www.nandland.com/goboard/setting-up-environment-icecube.html

Lattice Semiconductor's web page for the IceCube2 FPGA programming tool:
http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/FPGAandLDS/iCEcube2.aspx
(Works on Windows and on Linux)

Downloading for Linux, 361 Mb, iCEcube2setup_Sep_12_2017_1708.tgz

Unpack and then right-click > run the iCEcube2setup_Sep_12_2017_1708 file.

It asks for the IceCube2 license file. (7/20/2021: I had to go through this process again.)
Get it from the Lattice web site > products> Software Tools > IceCube2 > Licensing > IceCube2 Free License.
SHORTCUT to the licensing:
https://www.latticesemi.com/en/Products/DesignSoftwareAndIP/FPGAandLDS/iCEcube2#_12092ABF818047B59CC430396492212C

It asks for my computer's mac address. In a terminal, I type
ifconfig
The first line of output says
eth0 Link encap:Ethernet HWaddr f4:4d:30:66:91:b9
and the mac address is the "HWaddr" above. (It wants the colons replaced with dashes so f4-4d-30-66-91-b9
It now emails the license to me. (I decide to install the license later)

When I start IceCube2, it looks but fails to find the license file here: /usr/local/flexlm/licenses/license.dat

So I create the directory and put the license.dat file in there. (I had to use "sudo")

The IceCube2 program is in ~/lscc/iCEcube2.2017.08/ICEcube2 It works!

(Note that there is a "LicenseSetup" program there that I guess would move the license to where it belongs automatically.)

I now get the Lattice Programmer tool from here:
http://www.latticesemi.com/Products/DesignSoftwareAndIP/ProgrammingAndConfigurationSw/Programmer.aspx

I click "Programmer Standalone 3.11.1 64-bit for Linux" and start the 62 Mb download.
programmer_3_11_x64-441-0-x86_64-linux.rpm

I extract the file and it seems to have unpacked a file intended to go into /usr/local
Wait and see for now.

I am now here: https://www.nandland.com/goboard/your-first-go-board-project.html

Create a file Switches_To_LEDs.v using Text Editor and save to ~/Documents

Create a project. My project directory is ~/Documents/GoBoard_work.

Project settings "In the New Project" window that need changing are
Device > Device Family > iCE40
> Device > HX1K

> Device Package > VQ100
IOBank Voltage(V) > topBank > 3.3 left, bottom, right also 3.3

Get constraints file. Go_Board_Constraints.pcf Put it in my project by clicking P&R Flow > Add P&R Files > right-click Constraints Files

ADDED NOTE: You eventually need Go_Board_Clock_Constraint.sdc, which you put into the project by opening Synthesis Tool > right-clicking Constraints Files

Try running Synthesis, and I get error: Child process exit with 2.

I tried to fix by reinstalling license (using the license installer) and running the setup and selecting "repair" but that didn't help. Finally I found in the discussion at the bottom of the page this solution:

> I tried a test project but failed to pass synthesis (I always got
> 'error 2' on both Windows and Linux).
> The solution to this problem is as follows: In Lattice iCEcube2: right-click
> on "Synthesis Tool".
> A pop-up appears, showing "Select Synthesis Tools...". Click on the pop-up
> and select "**Lattice LSE**". This changes the command-tree to
> "Run Lattice LSE Synthesis" making it finally possible to create a bitstream.

******** YES *******
Now Synthesis works.

---------- Getting the bit file into the Go Board ----------

I couldn't get the Diamond programmer to work. Use the open-source "IceStorm" programmer instead.

(The Diamond programmer download is a .rpm file, which tells me it is a RedHat or Centos type file. "Alien" is supposed to be able to install such files, but the installed software is not working.)

Instead, I will try to use IceStorm. (Suggestion in the discussion)
http://www.clifford.at/icestorm/

Installing prerequisites (this command is for Ubuntu 14.04):

```
sudo apt-get install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev \
                     qt5-default python3-dev libboost-all-dev cmake
```

Installing the IceStorm Tools (icepack, icebox, iceprog, icetime, chip databases):

```
git clone https://github.com/cliffordwolf/icestorm.git icestorm
cd icestorm
make -j$(nproc)
sudo make install
```

I want to use iceprog invoking it with the bitfile which was previously created
iceprog Switches_To_LEDs_bitmap.bin

but I get an error:

init..
Can't find iCE FTDI USB device (vendor_id 0x0403, device_id 0x6010 or 0x6014).
ABORT.

But THIS Works: <-------********** YES **********
sudo iceprog Switches_To_LEDs_bitmap.bin And the code programs the FPGA OK!!

---------- How to program the Go Board without having to use "sudo" ----------
I found advice here https://stackoverflow.com/questions/36633819/iceprog-cant-find-ice-ftdi-usb-device-linux-permission-issue

sudo pico /etc/udev/rules.d/53-lattice-ftdi.rules
I put this in the .rules file:
ACTION=="add", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", MODE:="666"
After fixing that file, and disconnecting and reconnecting the device, you should be able to program the FPGA as normal user without sudo. <-----********** YES, it works

---------- Try installing the rest of the IceStorm system. ----------

Note: As far as I can tell, the rest of IceStorm is not needed for using the Go Board. OK to skip.

I ran into a problem where cmake was looking for "Eigen3"
Solved with this command:
sudo apt-get install libeigen3-dev

Making the rest of IceStorm:

git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
cd arachne-pnr
make -j$(nproc)
sudo make install

git clone https://github.com/YosysHQ/nextpnr nextpnr
cd nextpnr
cmake -DARCH=ice40 -DCMAKE_INSTALL_PREFIX=/usr/local .
make -j$(nproc)
sudo make install

Simple user manual for IceStorm:
http://hedmen.org/icestorm-doc/icestorm.html#Configuration-process

---------- Simulation using EDA Playground ----------
https://www.edaplayground.com/
Login: Use my work address, PW="the usual K"
1/19/2021: A year ago it was ok. Now I'm having trouble to get it to work. Additional licensing requirements have been added and the one free simulation has errors connecting.

---------- Go Board's PMOD connectors ----------
Looking at the connector, into the holes you see two rows of 6 holes:
oooooo 1st row

oooooo 2nd row

The first row, which is for PMOD module 1, going left-to-right is
Pin 6, VCC (+3.3 V)
Pin 5, Ground
Pin 4, io_PMOD_4 (usually a clock signal to the module)
Pin 3, io_PMOD_3 (can be data or other signal)
Pin 2, io_PMOD_2 (usually data to or from the module)
Pin 1, io_PMOD_1 (usually a select or "activate" signal, active low, to the module)

The second row, which is for PMOD module 2, going left-to-right is
Pin 12, VCC (+3.3 V)
Pin 11, Ground
Pin 10, io_PMOD_4 (usually a clock signal to the module)
Pin 9, io_PMOD_3 (can be data or other signal)
Pin 8, io_PMOD_2 (usually data to or from the module)
Pin 7, io_PMOD_1 (usually a select or "activate" signal, active low, to the module)

---------- Go Board's VGA connector ----------
Looking at the connector, into the holes you see two rows of holes:
ooooo 1st row (Five holes)
 ooooo 2nd row (Five holes)
ooooo 3rd row (Five holes)

The first row, going left-to-right is
Pin 5, Ground
Pin 4, nc (no connection)
Pin 3, Blue signal (connects to o_VGA_Blu_0, _1, _2 via resistors)
Pin 2, Green signal (connects to o_VGA_Grn_0, _1, _2 via resistors)
Pin 1, Red signal (connects to o_VGA_Red_0, _1, _2 via resistors)

The second row, going left-to-right is
Pin 10, Ground
Pin 9, nc
Pin 8, Ground
Pin 7, Ground
Pin 6, Ground

The 3rd row, going left-to-right is
Pin 15, nc
Pin 14, Vertical sync signal, o_VGA_VSync (I sometimes use this to trigger the oscilloscope)
Pin 13, Horizontal sync signal, o_VGA_HSync
Pin 12, nc
Pin 11, nc

Note that the sync signals, pins 14 and 13 are a convenient place output
an oscilloscope trigger signal, assuming you're not connect a VGA display.

---------- pmod "AD1" ADC from Digilent ----------
"Home" page for PmodAD1:
https://reference.digilentinc.com/reference/pmod/pmodad1/start

"Sales" page:
https://store.digilentinc.com/pmod-ad1-two-12-bit-a-d-inputs/
Two channels, 1 Msample/s each channel. 500 kHz low pass anti-alias filters.

I got the AD1 module to work on an Arduino. Info here:
https://www.hackster.io/56479/using-the-pmod-ad1-with-arduino-uno-38dd4a
 In the process I learned that the optimum clock frequency is 20 MHz, and the 25 MHz clock on the GoBoard is inconvenient. Unfortunately, the Lattice ICE-40 HX1 on the GoBoard DOES NOT have an internal PLL to derive 20 MHz from the 25 MHz.

Background info on the SPI serial data format used by Pmod devices, including the AD1:
https://reference.digilentinc.com/learn/fundamentals/communication-protocols/spi/start
More info here: https://reference.digilentinc.com/learn/programmable-logic/tutorials/pmod-ips/start

J1 Pin connections to the controller or FPGA:
Pin 1 CS Chip Select
Pin 2 D0 Data out for 1st ADC
Pin 3 D1 Data out for 2nd ADC
Pin 4 CLK Serial Clock
Pin 5 GND
Pin 6 VCC

J2 pin connections for the analog output:
Pin 1 A0 Analog in for 1st ADC
Pin 2 GND
Pin 3 A1 Analog in for 2nd ADC
Pin 4 GND
Pin 5 GND
Pin 6 Vcc

---------- pmod "DA2" DAC from Digilent ----------
"Home" page for PmodDA2:
https://reference.digilentinc.com/reference/pmod/pmodda2/start

12-bit digital-to-analog converter
Two simultaneous conversion channels

Texas Instruments Data Sheet:
http://www.ti.com/lit/ds/symlink/dac121s101.pdf

Max clock = 30 MHz Since each output sample requires 16 (or 17?) clock cycles, this implies an output speed of 1.8e6 samples/second. But its nowhere near this fast.

It has trouble handling more than 250000 samples/s.
According to data sheet the "settling time" is 8 microseconds, which implies
a maximum of 125000 samples/s

J1 Pin connections to the controller or FPGA:
Pin 1 ~SYNC (active low)
Pin 2 DINA
Pin 3 DINB
Pin 4 SCLK

Pin 5 GND
Pin 6 VCC

J2 pin connections for the analog output , logging into the connector, left-to-right
Pin 6 Vcc (Furthest to the left)
Pin 5 GND
Pin 4
Pin 3 Vout 2
Pin 2
Pin 1 Vout 1 (Furthest to the right)

---------- pmod "DA3" DAC from Digilent ----------
"Home" page for PmodDA3:
https://reference.digilentinc.com/reference/pmod/pmodda3/start

High resolution, 16-bit Digital-to-Analog converter
Low noise analog output
SMA connector
2.5V reference voltage (Hence, the output range is 0 to 2.5 Volts)
Uses AD5541A which has a 1 microsecond settling time.
https://www.analog.com/media/en/technical-documentation/data-sheets/AD5541A.pdf
Max clock rate: 50 MHz

J1 Pin connections to the controller or FPGA:
Pin 1 ~CS (chip select, active low)
Pin 2 DIN
Pin 3 ~LDAC
Pin 4 SCLK
Pin 5 GND
Pin 6 VCC

---------- The Pmod Y Cable ----------
https://store.digilentinc.com/2x6-pin-to-dual-6-pin-pmod-splitter-cable/

---------- How to get different clock frequencies: the PLL ----------
Bad news: The ICE40 in the VQ100 package has NO plls. This info is from here:
http://www.latticesemi.com/view_document?document_id=47778

I haven't yet gotten it to work, but here is some info:

Application Note for the PLL: http://www.latticesemi.com/view_document?document_id=47778

The tool to create Verilog Code to activate (instantiate) the PLL seems to be Lattice Radiant, downloadable for Windows and Linux:
http://www.latticesemi.com/Products/DesignSoftwareAndIP/FPGAandLDS/Radiant

The Lattice Radiant User Manual is here:
https://www.latticesemi.com/-/media/LatticeSemi/Documents/UserManuals/EI2/FPGA-TN-02052-1-0-iCE40UP-sysCLOCK-PLL-Radiant-SW.ashx?document_id=52236

A good starting point for ICE40 info in general is:
http://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40#_21E33C7EC0BD48AA80FE384ED73CC895

---------- I need to implement Memory! How? Some info might be here ----------
Link to Application Note on Memory:
http://www.latticesemi.com/view_document?document_id=47775

NICE! The application note's Appendix A has example code on how to infor (automatically use) the memory. Single and dual port.

Memory capacity = 64 kBits = 8 kBytes = 4 kWords (16 bit) = 5.333 kWords (12 bit)

========== READ ADC - Verilog Code ==========

Separate this Verilog code into three files to compile

```
// Filename:  Read_ADC_Top.v
// Written by me, Peter Halverson, and posted on my web page http://halverscience.net/   Nov 4, 2019
// This Verilog code is intended to run on a GoBoard, available from nandland.com,
// and it uses a PmodAD1 analog-to-digital converter, available from Digilent
// This is the top module.  NOTE:  I had  trouble with the Lattice LSE Synthesis tool
// not "autoguessing" that this is the top module.  I had to explicitly tell the tool.
// To do that, go into Tool > Tool Options > LSE (tab) > Top Level Unit and type in "Read_ADC_Top"

module Read_ADC_Top
  (input  i_Clk,            // Wire from the Main Clock, 25 MHz
   output o_Segment1_A,  // Wiring to the 1st 7-segment display.
   output o_Segment1_B,
   output o_Segment1_C,
   output o_Segment1_D,
   output o_Segment1_E,
   output o_Segment1_F,
   output o_Segment1_G,
   output o_Segment2_A,  // Wiring to the 2nd 7-segment display.
   output o_Segment2_B,
   output o_Segment2_C,
   output o_Segment2_D,
   output o_Segment2_E,
   output o_Segment2_F,
   output o_Segment2_G,
   output o_LED_1,     // Wiring to the LEDs
   output o_LED_2,
   output o_LED_3,
   output o_LED_4,
   output io_PMOD_1,           // Wiring to the ADC board ~ chip select (active low)
   input  io_PMOD_2,  // data from ADC 0
   input  io_PMOD_3,  // data from ADC 1  (not used)
   output io_PMOD_4,           // clock to read out data
   input i_Switch_1,     // Wires to the switches.  Not used now, but useful for debugging.
   input i_Switch_2,
   input i_Switch_3,
   input i_Switch_4);

  wire w_Segment1_A, w_Segment2_A;
```

```
wire w_Segment1_B, w_Segment2_B;
wire w_Segment1_C, w_Segment2_C;
wire w_Segment1_D, w_Segment2_D;
wire w_Segment1_E, w_Segment2_E;
wire w_Segment1_F, w_Segment2_F;
wire w_Segment1_G, w_Segment2_G;
wire w_LED_1, w_LED_2, w_LED_3, w_LED_4;

wire [11:0] w_ADC_Data;
wire w_ADC_Data_Valid;
reg [31:0] r_Readout_Count  = 0;
parameter Readout_Period = 125000;  // 125000 gives 200 per second (25 MHz clock / 125000 )
reg [11:0] r_ADC_Data;   // Only low order 12 bits are used.   Four high order bits always zero
reg r_ADC_Data_Requested = 1'b0;

// Code to read the ADC and put the results on the 7-segment display (upper 8 bits)
// and the LEDS (lower 4 bits)
always @(posedge i_Clk) begin
        if (r_Readout_Count >= Readout_Period) begin
        r_Readout_Count <= 0;
        r_ADC_Data_Requested <= 1'b1;     //  Tell Read_ADC we want data
        end else begin
        r_Readout_Count <= r_Readout_Count + 1;
        r_ADC_Data_Requested <= 1'b0;
        end
        if (w_ADC_Data_Valid == 1'b1) begin
        //if (1'b1 == 1'b1) begin
        r_ADC_Data <= w_ADC_Data;         // Get the data from Read_ADC
        end
end

// Interface (instantiation) to the code that triggers the ADC and reads its data.
// parameter ADC_CLKS_PER_BIT determines the readout speed of the bits from the ADC.
// Its the GoBoard's clock frequency divided by the parameter.
// Example: 25,000,000 / 25 would give 1 mega-baud.
// Maximum, according to Analog Devices AD7476 data sheet is 20 mega-baud
Read_ADC #(.ADC_CLKS_PER_BIT(4)) Read_ADC_Inst   // 4 is gets me 6.25 Mbaud, 160 ns clock
(.i_Clock(i_Clk),
 .i_ADC_Data_Serial(io_PMOD_2),
 .i_ADC_Data_Requested(r_ADC_Data_Requested),
 .o_ADC_Data_Valid(w_ADC_Data_Valid),
 .o_ADC_Data(w_ADC_Data),
 .o_ADC_Chip_Select_Not(io_PMOD_1),
 .o_ADC_Clock(io_PMOD_4));

// Binary to 7-Segment Converter for Upper Digit, highest nibble of ADC data
Binary_To_7Segment SevenSeg1_Inst
(.i_Clk(i_Clk),
 .i_Binary_Num(r_ADC_Data[11:8]),
 .o_Segment_A(w_Segment1_A),
 .o_Segment_B(w_Segment1_B),
 .o_Segment_C(w_Segment1_C),
```

```
  .o_Segment_D(w_Segment1_D),
  .o_Segment_E(w_Segment1_E),
  .o_Segment_F(w_Segment1_F),
  .o_Segment_G(w_Segment1_G));

 assign o_Segment1_A = ~w_Segment1_A;
 assign o_Segment1_B = ~w_Segment1_B;
 assign o_Segment1_C = ~w_Segment1_C;
 assign o_Segment1_D = ~w_Segment1_D;
 assign o_Segment1_E = ~w_Segment1_E;
 assign o_Segment1_F = ~w_Segment1_F;
 assign o_Segment1_G = ~w_Segment1_G;

 // Binary to 7-Segment Converter for Lower Digit, middle nibble of ADC data
 Binary_To_7Segment SevenSeg2_Inst
 (.i_Clk(i_Clk),
  .i_Binary_Num(r_ADC_Data[7:4]),
  .o_Segment_A(w_Segment2_A),
  .o_Segment_B(w_Segment2_B),
  .o_Segment_C(w_Segment2_C),
  .o_Segment_D(w_Segment2_D),
  .o_Segment_E(w_Segment2_E),
  .o_Segment_F(w_Segment2_F),
  .o_Segment_G(w_Segment2_G));

 assign o_Segment2_A = ~w_Segment2_A;
 assign o_Segment2_B = ~w_Segment2_B;
 assign o_Segment2_C = ~w_Segment2_C;
 assign o_Segment2_D = ~w_Segment2_D;
 assign o_Segment2_E = ~w_Segment2_E;
 assign o_Segment2_F = ~w_Segment2_F;
 assign o_Segment2_G = ~w_Segment2_G;

 assign o_LED_1 = r_ADC_Data[3];   // Lowest nibble of ADC data will be displayed on the four LEDs
 assign o_LED_2 = r_ADC_Data[2];
 assign o_LED_3 = r_ADC_Data[1];
 //assign o_LED_3 = i_Switch_3;
 //assign o_LED_4 = i_Switch_4;
 //assign o_LED_4 = r_Readout_Count[22];
 assign o_LED_4 = r_ADC_Data[0];

endmodule // Read_ADC_Top

=============================================================

// Filename:  Read_ADC.v
// ADC Readout for Digilent PMOD AD1, which has two Analog Devices AD7476A 12-bit ADCs
// Written by Peter Halverson and posted on http://halverscience.net/  Nov. 4, 2019
// Based on UART RX from http://www.nandland.com

module Read_ADC
 #(parameter ADC_CLKS_PER_BIT = 25)
```

```
(
 input          i_Clock,
 input          i_ADC_Data_Serial,
 input          i_ADC_Data_Requested,// Set this to True when you want an ADC readout
 output         o_ADC_Data_Valid,   // When this is true, it means the adc data is ready
 output [11:0] o_ADC_Data,          //Its a 12 bit ADC
 output         o_ADC_Chip_Select_Not, // The ~CS line is active when False
 output         o_ADC_Clock         // Falling edge requests next bit, bit is read on rising edge
 );

parameter IDLE                 = 2'b00;   // State Machine states
parameter ADC_CONVERSION_DELAY = 2'b01;   // Not needed?  Datasheet says 10 ns minimum is needed.
parameter READ_DATA_BITS       = 2'b10;
parameter CLEANUP              = 2'b11;

reg [7:0]       r_Clock_Count          = 0;
reg [3:0]       r_Bit_Index    = 15; //4 zero bits + 12 data bits = 16 bits total
reg [15:0]      r_ADC_Data   = 0;  // First 4 bits always zero
reg             r_ADC_Data_Valid = 0;
reg [2:0]       r_SM_Main    = 0;
reg             r_ADC_Chip_Select_Not = 1;
reg             r_ADC_Clock            = 1;
reg [7:0]       r_Delay_Clock_Count = 0;
reg             r_Got_The_Bit= 1'b0;

// Read ADC state machine
always @(posedge i_Clock)
begin
        case (r_SM_Main)
        IDLE : begin
        r_ADC_Data_Valid    <= 1'b0;
        r_Clock_Count          <= 0;
        r_Delay_Clock_Count <= 0;
        r_Bit_Index    <= 14;  // ADC send 15 bits (not 16)   High order bits first
        r_ADC_Clock            <= 1'b1;
        if (i_ADC_Data_Requested == 1'b1) begin
        r_SM_Main <= ADC_CONVERSION_DELAY;
        r_ADC_Chip_Select_Not <= 1'b0;     // Start the conversion
        end else begin
        r_SM_Main <= IDLE;
        r_ADC_Chip_Select_Not <= 1'b1;     // Don't start the conversion
        end
        end // case: IDLE
        ADC_CONVERSION_DELAY : begin
        if (r_Delay_Clock_Count < 0) begin   // Adjust to give time for ADC to start a conversion
        // 0 gives 80 ns (Three cycles of 25 MHz clock), 1 gives 120 ns, 2 gives 160 ns, etc
        r_Delay_Clock_Count <= r_Delay_Clock_Count + 1;
        r_SM_Main <= ADC_CONVERSION_DELAY;
        end else begin
        r_Delay_Clock_Count <= 0;
        r_SM_Main <= READ_DATA_BITS;
        end
```

```
        end // case: ADC_CONVERSION_DELAY
        READ_DATA_BITS : begin
        if (r_Clock_Count < ADC_CLKS_PER_BIT-1) begin
        if (r_Clock_Count < (ADC_CLKS_PER_BIT/2)) begin
        r_ADC_Clock <= 1'b0;          // Falling edge tells ADC to output a bit
        r_Got_The_Bit <= 1'b0;
        end else begin
        r_ADC_Clock <= 1'b1;          // Rising edge is when we get the bit.  (It is now stable)
        if (r_Got_The_Bit == 1'b0) begin
        r_ADC_Data[r_Bit_Index] <= i_ADC_Data_Serial;  // GET THE DATA BIT!!
        r_Got_The_Bit <= 1'b1;   //We want to latch the bit only once
        // Check if we have received all bits
        if (r_Bit_Index > 0) begin       // We need to get more bits (We get 16 bits, 1st four are zero)
                r_Bit_Index <= r_Bit_Index - 1;  //ADC send bit 15 first, then bit 14, ...
                r_SM_Main   <= READ_DATA_BITS;
        end else begin
                r_ADC_Data_Valid <= 1'b1;             // All 15 bits are now valid. Tell caller to get the data.
                r_ADC_Chip_Select_Not <= 1'b1;     // Stop the conversion
                r_SM_Main     <= CLEANUP;
        end
        end
        end
        r_Clock_Count <= r_Clock_Count + 1;
        end else begin
        r_Clock_Count <= 0;
        end
        end // case: READ_DATA_BITS
        CLEANUP : begin      // Stay here 1 clock
        // r_Delay_Clock_Count <= 0;
        r_SM_Main <= IDLE;
        r_ADC_Data_Valid   <= 1'b0;
        end // case: CLEANUP
        default :
        r_SM_Main <= IDLE;
        endcase
 end

 assign o_ADC_Data_Valid   = r_ADC_Data_Valid;
 assign o_ADC_Data = r_ADC_Data[11:0];
 assign o_ADC_Chip_Select_Not = r_ADC_Chip_Select_Not;
 assign o_ADC_Clock = r_ADC_Clock;
endmodule // Read_ADC

============================================================

// Filename:  Binary_To_7Segment.v
////////////////////////////////////////////////////////////////////////////////
// File downloaded from http://www.nandland.com
////////////////////////////////////////////////////////////////////////////////
// This file converts an input binary number into an output which can get sent
// to a 7-Segment LED.  7-Segment LEDs have the ability to display all decimal
// numbers 0-9 as well as hex digits A, B, C, D, E and F.  The input to this
```

```
// module is a 4-bit binary number.  This module will properly drive the
// individual segments of a 7-Segment LED in order to display the digit.
// Hex encoding table can be viewed at:
// http://en.wikipedia.org/wiki/Seven-segment_display
//////////////////////////////////////////////////////////////////////////////////

module Binary_To_7Segment
 (
  input           i_Clk,
  input [3:0] i_Binary_Num,
  output          o_Segment_A,
  output          o_Segment_B,
  output          o_Segment_C,
  output          o_Segment_D,
  output          o_Segment_E,
  output          o_Segment_F,
  output          o_Segment_G
  );

 reg [6:0]        r_Hex_Encoding = 7'h00;

 // Purpose: Creates a case statement for all possible input binary numbers.
 // Drives r_Hex_Encoding appropriately for each input combination.
 always @(posedge i_Clk)
         begin
         case (i_Binary_Num)
         4'b0000 : r_Hex_Encoding <= 7'h7E;
         4'b0001 : r_Hex_Encoding <= 7'h30;
         4'b0010 : r_Hex_Encoding <= 7'h6D;
         4'b0011 : r_Hex_Encoding <= 7'h79;
         4'b0100 : r_Hex_Encoding <= 7'h33;
         4'b0101 : r_Hex_Encoding <= 7'h5B;
         4'b0110 : r_Hex_Encoding <= 7'h5F;
         4'b0111 : r_Hex_Encoding <= 7'h70;
         4'b1000 : r_Hex_Encoding <= 7'h7F;
         4'b1001 : r_Hex_Encoding <= 7'h7B;
         4'b1010 : r_Hex_Encoding <= 7'h77;
         4'b1011 : r_Hex_Encoding <= 7'h1F;
         4'b1100 : r_Hex_Encoding <= 7'h4E;
         4'b1101 : r_Hex_Encoding <= 7'h3D;
         4'b1110 : r_Hex_Encoding <= 7'h4F;
         4'b1111 : r_Hex_Encoding <= 7'h47;
         endcase
         end // always @ (posedge i_Clk)

 // r_Hex_Encoding[7] is unused
 assign o_Segment_A = r_Hex_Encoding[6];
 assign o_Segment_B = r_Hex_Encoding[5];
 assign o_Segment_C = r_Hex_Encoding[4];
 assign o_Segment_D = r_Hex_Encoding[3];
 assign o_Segment_E = r_Hex_Encoding[2];
 assign o_Segment_F = r_Hex_Encoding[1];
```

```
  assign o_Segment_G = r_Hex_Encoding[0];

endmodule // Binary_To_7Segment
```