

Project 1: Go Board – Your First Project

Video: <https://youtu.be/1eo21vHxw0>

Link: <https://nandland.com/project-1-your-first-go-board-project/>

Make an LED blink when you press a Button

At this point – my patient reader – you have got the Go Board, you installed all of the necessary programs, let's just quickly review what you should have complete at this point.

I also created a YouTube video for this project, should you prefer to follow along with that.

1. Installed [iCEcube2](#) software
2. Installed [Programmer](#) Standalone

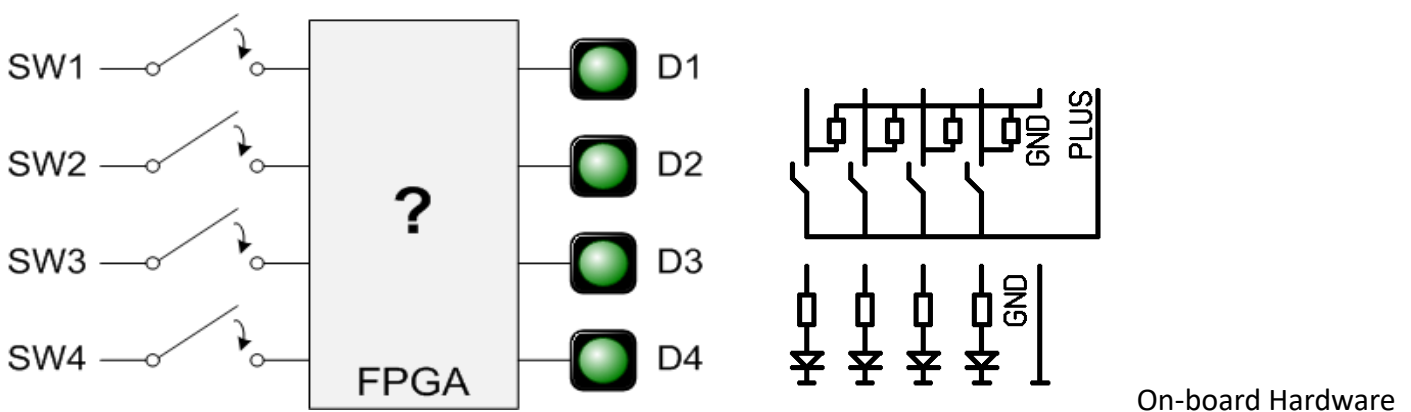
Now it's time to get cracking! The first project that we are going to do is going to be as simple as possible. All we are going to do is program the FPGA to perform one task:

When a button is pushed, one of the LEDs should light up.

This is a perfect project to do first, because it will take us through the entire build flow:

**from coding the VHDL or Verilog,
to building the FPGA bitstream,
to programming the Go Board.**

This tutorial will walk you through everything you need to know. Note that this project is 90% the same should you be learning VHDL or Verilog, so I lumped both into the same lesson. The Go Board tutorials build on each other, so it's important to do them in order and remember what you learned! Let's get started!



The first thing you are going to do is to create your VHDL or Verilog code that tells the FPGA what to do. Again, we are going to be creating code that tells the FPGA to light up one of the four LEDs when a particular switch is pushed.

The image to the right shows what this looks like. When Switch 1 (SW1) is pushed down on the Go Board, we want D1 to light up. When the button SW1 is released, D1 should turn off. This is a simple FPGA design, but going through the code and the build process will get you comfortable with everything before we get to more complicated designs.

Project Design

No matter which language you decide to use, you should give the project some thought before you begin. As shown in the image above, you have eight signals to deal with:

Four of the signals are inputs and four others are outputs.

The inputs are your switches, the outputs are your LEDs. How can you design code such that when a button

is pushed, an LED is illuminated? Give it some thought, try to write the code on your own. The answers for both Verilog and VHDL are shown below.

Verilog Code – Switches_To_LEDs.v:

```
1 module Switches_To_LEDs
2   (input i_Switch_1,
3     input i_Switch_2,
4     input i_Switch_3,
5     input i_Switch_4,
6     output o_LED_1,
7     output o_LED_2,
8     output o_LED_3,
9     output o_LED_4);
10
11 assign o_LED_1 = i_Switch_1;
12 assign o_LED_2 = i_Switch_2;
13 assign o_LED_3 = i_Switch_3;
14 assign o_LED_4 = i_Switch_4;
15
16 endmodule
```

```
1 module      Switches_To_LEDs      -- module and      name_of_module
2   (input i_Switch_1,              -- ( input ...      ,
3     input i_Switch_2,              --   input ...      ,
4     input i_Switch_3,              --   input ...      ,
5     input i_Switch_4,              --   input ...      ,
6
7     output o_LED_1,                --   output          ,
8     output o_LED_2,                --   output          ,
9     output o_LED_3,                --   output          ,
10    output o_LED_4);                --   output          ; )
11
12 assign o_LED_1 = i_Switch_1; --      connect signal  ;
13 assign o_LED_2 = i_Switch_2; --      connect signal  ;
14 assign o_LED_3 = i_Switch_3; --      connect signal  ;
15 assign o_LED_4 = i_Switch_4; --      connect signal  ;
```

16

```
endmodule                                --      endmodule
```

Let's talk about the Verilog code above.

The first keyword you encounter is **module**.

Modules are the blocks of code that perform some functionality. Modules can contain all of the code you need, such as the code above, or they can instantiate other modules.

More complicated designs will create levels of hierarchy by having many modules.

On the module, you must define the interfaces, which consist of **inputs** and **outputs**.

In this case, we are creating four inputs and four outputs. You can think of these inputs and outputs of being either a 0 or 1 in binary. The inputs and outputs are going to be connected to specific pins on the Go Board. This connection is done later (and is not part of the Verilog code).

The last keyword introduced in this first project is **assign**.

Assign is used to connect two signals together.

There are some rules about this keyword, but for now I'll just tell you that it must be used outside an **always block**.

Always blocks are for a later project, so don't worry about them for now. Once all of the output signals are assigned to the input signals, we have successfully connected all LED outputs to the Switch inputs.

Now when i_Switch_1 goes high (to a binary 1) o_LED_1 will also go high.

Conversely when i_Switch_1 goes low (to binary 0) o_LED_1 will also go low. You are literally tying the two of them together in your FPGA.

There will actually be a physical connection inside the FPGA between these two pins.

Neat huh?! You need to remember that the code you are writing is designing a physical circuit inside the FPGA. This particular circuit is pretty simple, but it's still a circuit!

VHDL Code – Switches_To_LEDs.vhd:

```
1library ieee;
2use ieee.std_logic_1164.all;
3
4entity Switches_To_LEDs is
5  port (
6    -- Push-Button Switches:
7    i_Switch_1 : in std_logic;
8    i_Switch_2 : in std_logic;
9    i_Switch_3 : in std_logic;
10   i_Switch_4 : in std_logic;
11
12   -- LED Pins:
13   o_LED_1 : out std_logic;
14   o_LED_2 : out std_logic;
15   o_LED_3 : out std_logic;
16   o_LED_4 : out std_logic
17  );
18end entity Switches_To_LEDs;
19
```

```

20architecture RTL of
21Switches_To_LEDs is
22begin
23  o_LED_1 <= i_Switch_1;
24  o_LED_2 <= i_Switch_2;
25  o_LED_3 <= i_Switch_3;
26  o_LED_4 <= i_Switch_4;
27
28end RTL;

```

```

1  library ieee;                -- libraries needed
2  use ieee.std_logic_1164.all; --
3
4  entity Switches_To_LEDs is    -- entity      xxxxxxxxx  is
5    port (                     -- port (    -- define the IO ports
6      -- Push-Button Switches: -- first the Push-Buttons
7      i_Switch_1 : in std_logic; --      name  :  in  std_logic;
8      i_Switch_2 : in std_logic; --      name  :  in  std_logic;
9      i_Switch_3 : in std_logic; --      name  :  in  std_logic;
10     i_Switch_4 : in std_logic; --      name  :  in  std_logic;
11
12     -- LED Pins:              -- then the LED Pins
13     o_LED_1 : out std_logic;  --      name  :  out  std_logic;
14     o_LED_2 : out std_logic;  --      name  :  out  std_logic;
15     o_LED_3 : out std_logic;  --      name  :  out  std_logic;
16     o_LED_4 : out std_logic   --      name  :  out  std_logic  ( no ; )
17   );                          -- to end definitions -- ) and ;
18end entity Switches_To_LEDs;  -- end   entity  xxxxxxxxx  ;
19
20    -- now the architecture, the connections are defined
21architecture RTL of Switches_To_LEDs is -- architecture xx  of yyyy  is
22begin                                -- begin the definitions
23  o_LED_1 <= i_Switch_1;            -- o_LED_1  <= (connected to) i_Switch_1
24  o_LED_2 <= i_Switch_2;            -- o_LED_2  <= (connected to) i_Switch_2

```

```

25  o_LED_3 <= i_Switch_3;      -- o_LED_3  <= (connected to) i_Switch_3
26  o_LED_4 <= i_Switch_4;      -- o_LED_4  <= (connected to) i_Switch_4
27
    end RTL;                    -- end    of the xx architecture

```

Let's talk about the VHDL code above.

Perhaps the first thing you might notice if you compare the Verilog code to the VHDL code is that the VHDL code is a bit longer. This is a common theme.

VHDL generally takes more typing to accomplish the same task.

The first two lines of this code tell the tools about which libraries and packages you are using in your code. Take it for granted that these two lines will likely be in every single piece of VHDL you ever write. For now, don't worry about what they're actually doing, just put them there.

The next keyword you encounter is **entity**.

The **entity** of a design can be thought of as the interface to the outside world.

Entity and **architecture** are the two VHDL keywords that define the functionality of some block of code. Together, an entity and architecture make up some piece of functionality in your FPGA design.

A single entity/architecture combination can contain all of the code you need, such as the code above, or they can instantiate other entities.

More complicated designs will create levels of hierarchy by having many entity/architectures.

Next, you need to specify any signals that are on this interface after using the **port** keyword.

Once you give the signals a name (e.g. `i_Switch_1`), you need to specify the direction: ***in, out, or inout*** (used rarely).

After that, give your signals a type. **std_logic** is the most prevalent type in VHDL and can be thought of as a bit containing 0 or 1 (though it can actually have more values than just that as we will see later).

These inputs and outputs are going to be connected to specific pins on the Go Board.

This connection is done later (and is not part of the VHDL code).

After the architecture, you need to use the **<= assignment symbol**.

An assignment is used to connect two signals together.

Note that if you try to just use a normal assignment like `=` as used in C, you'll get a tool error.

Once all of the output signals are assigned to the input signals, we have successfully connected all LED outputs to the Switch inputs.

Now when `i_Switch_1` goes high (to a binary 1) `o_LED_1` will also go high. Conversely when `i_Switch_1` goes low (to binary 0) `o_LED_1` will also go low. You are literally tying the two of them together in your FPGA. There will actually be a physical connection inside the FPGA between these two pins. Neat huh?! You need to remember that the code you are writing is designing a physical circuit inside the FPGA. This particular circuit is pretty simple, but it's still a circuit!

Building your FPGA design

Once you have the coding done, it's time to build the design!

Open iCEcube. Click **File** then **New Project**.

You will be greeted with an intimidating window asking for a bunch of specific information about your FPGA board.

I'll be nice and tell you how to set those values:

New Project

Project

Project Name:

Project Directory:

Device

Device Family:

Device:

Device Package:

Operating Condition

Junction Temperature (in degrees Celsius)

Range: Best: Typical: Worst:

Core Voltage(V)

Voltage Tolerance Range: Best: Typical: Worst:

IOBank Voltage(V)

topBank bottomBank

leftBank rightBank

Perform timing analysis based on

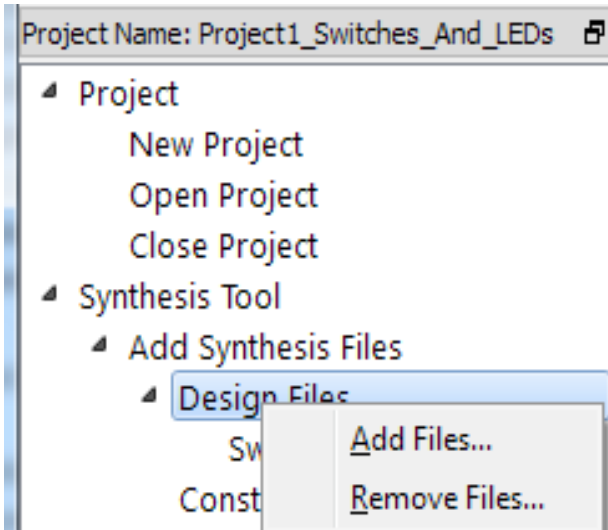
☐ Best ☐ Typical ☒ Worst

☒ Start From Synthesis

☐ Start From BackEnd

☐ IP Generation

Go Board Project Settings



These are the settings that you will be using for every single Go Board project, so keep that in mind. Click Next when you're done here.

This next dialog box prompts you to add the Verilog or VHDL source file that you created previously. You can do that now. Or alternatively you can skip this step by clicking Finish.

Instead you can expand **Synthesis Tool** and right click on **Design Files** to add your file to your project. The Verilog or VHDL that you add is used for **Synthesis**.

What is Synthesis?

It's probably worth taking a minute to discuss the first phase of your FPGA build process.

The iCEcube2 tool takes the code that you wrote and goes through a process called *synthesis*.

Synthesis converts the VHDL or Verilog code into physical elements that are available on your particular FPGA. An FPGA has dedicated pieces of logic on it.

It has **wires**, it has **Look-Up Tables**, it has **Registers**, and it has some **Memory**.

It can have other things, but these are by far the most targeted elements during synthesis. It's OK if you aren't comfortable with each of these right now, we'll talk about them each individually.

Since this design is very simple, it will use very little of your FPGA resources in the synthesis process, just a few wires to route signals.

After the design is done with Synthesis, it goes to **Place and Route (P&R)**.

What is Place and Route?

Place and Route is the second phase of the FPGA build process.

Synthesis converts the code to physical components available to your FPGA.

Place and Route takes those components and **places** them on your FPGA.

You can think of an FPGA as a large grid of these components. Place and Route will assign which component gets used where.

It is a particularly important step of the FPGA build process for designs that use clocks. This particular design does not require the use of a clock, so Place and Route is not as important.

We will see in future projects how clocks play a pivotal role in FPGAs, and you'll understand more about Place and Route then.

One very important thing that Place and Route does is it assigns the signals at the top level of your FPGA design to physical pins on the device.

You need to tell the tools how to map each signal in your design to a physical pin on the device!

This is a very important step and if you do not tell Place and Route how to assign the pins, it will just choose them on its own and it will be wrong. **Does i_Switch_1 go on Pin 1 or Pin 100?**

This gets set in a **Constraints File**. Again, I'll be nice and give you the constraints file needed for the Go Board. This provides the mapping for all of your pins on the projects that we will encounter. You'll notice in the constraint file that there's more signals than we created in the code above. That's OK, we'll use those extra signals eventually.

Ding Ding Ding! Come and get your Constraint File!

Here's a link to the [Go Board Constraints.pcf](#). Save this file in your project directory and add it to your project in iCEcube. This is done under **P&R Flow** then expand **Add P&R Files** then add it under **Constraint Files**. *(see the contents of the constraints file as well at the end of this project.)*

Building the FPGA Bitstream File

We're getting close, I promise.

Everything is set. You created your project. The code has been added to your project. The constraints have been added.

Now it's time to build the bitstream!

This is the file that is used by your FPGA to program itself.

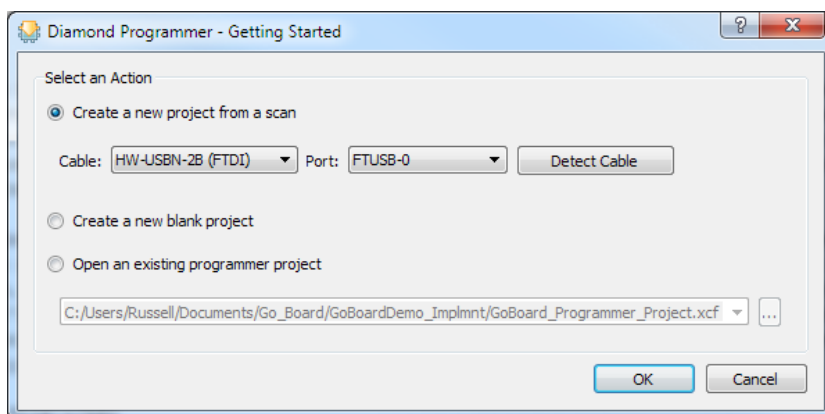
Click **Tool** then **Run All**. This should go through the entire build process, from **Synthesis**, through **Place & Route**, to generating your **bitstream file**.

(Just a note here, you'll see Bitmap in iCEcube2, Bitmap is the program that creates the bitstream).

You should see some nice green check marks appear. Take a minute to look at the Output from the build process, the tools give you some neat information about what's happening behind the scenes.

Programming the FPGA

Open the Programmer Standalone tool. You should be greeted with this screen.



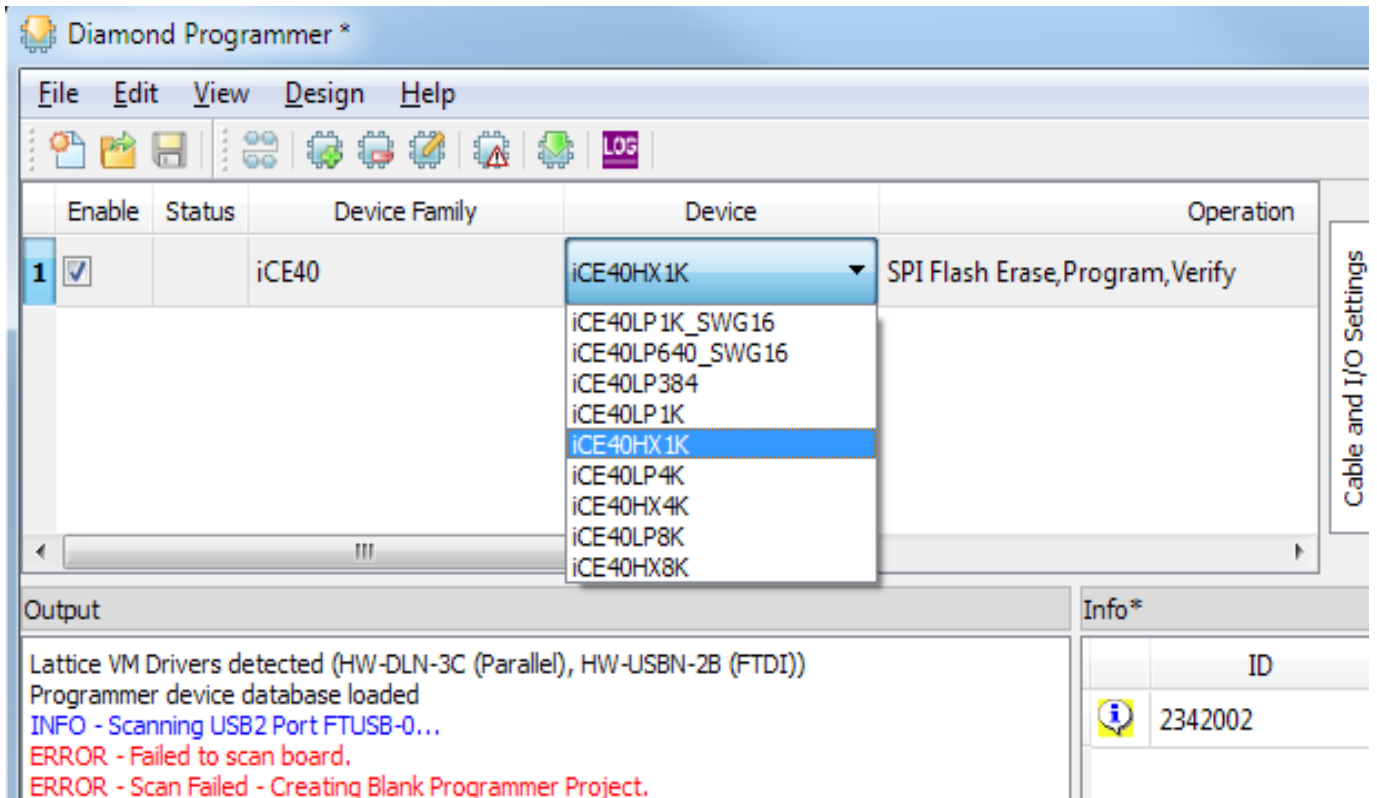
Programmer Standalone First Screen

Now, the way the Go Board works is that there is an integrated circuit that turns the USB connection to an SPI interface, which is used to program a Flash chip that's installed on the Go Board.

The purpose of using Programmer Standalone is to load this Flash over the USB.

Once the flash has been loaded, the FPGA will boot up from the flash and you'll see the fruits of your labor! Since you're going to a SPI Interface, the Programmer will be unable to detect the FPGA. That's OK, you will need to tell the Programmer about what FPGA you do have on the Go Board.

Select **Device Family** and **Device** as shown in the figure below.



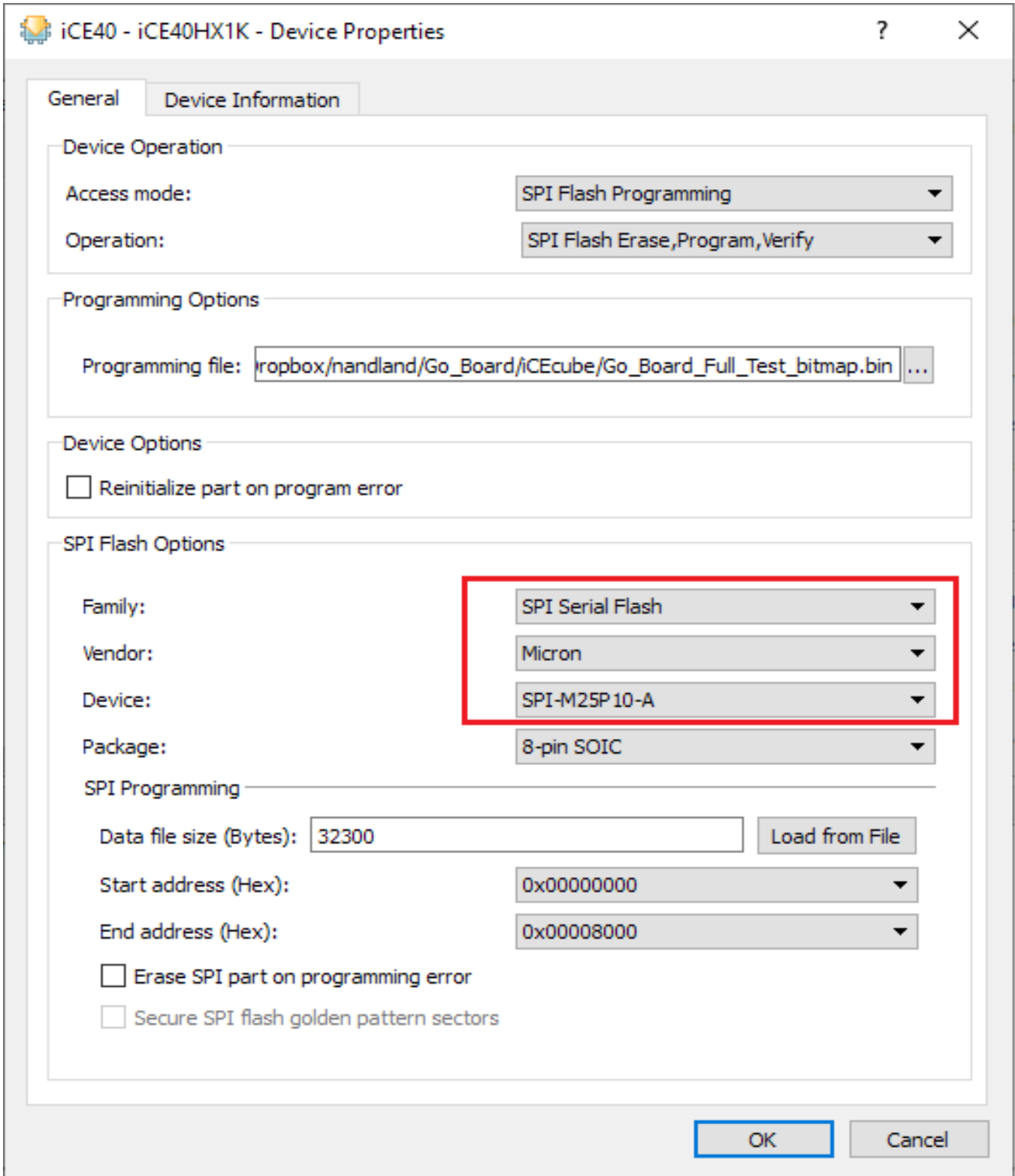
Programmer First Screen

Now double click under **Operation**
and next to **Access Mode**
select SPI Flash Programming.

That will bring up the picture as shown below. You will need to select your programming bitstream that you built.

This is located in your iCEcube2 project directory /sbt/outputs/bitmap/.

Make sure to select **Family** and **Vendor** as shown in the figure below.



iCE40 - iCE40HX1K - Device Properties

General | **Device Information**

Device Operation

Access mode: SPI Flash Programming

Operation: SPI Flash Erase,Program,Verify

Programming Options

Programming file: /ropbox/nandland/Go_Board/iCEcube/Go_Board_Full_Test_bitmap.bin ...

Device Options

☐ Reinitialize part on program error

SPI Flash Options

Family: SPI Serial Flash

Vendor: Micron

Device: SPI-M25P 10-A

Package: 8-pin SOIC

SPI Programming

Data file size (Bytes): 32300 Load from File

Start address (Hex): 0x00000000

End address (Hex): 0x00008000

☐ Erase SPI part on programming error

☐ Secure SPI flash golden pattern sectors

OK Cancel

Go Board SPI Flash Settings

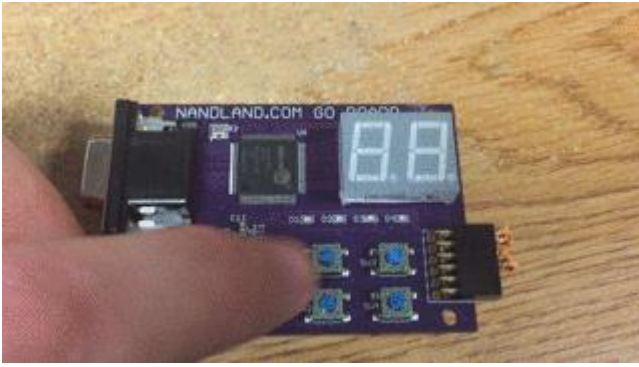
Now click **Design** and select **Program**.

If everything was done correctly, you should see **Operation Successful**.

This means that your SPI Flash has been programmed and your FPGA is running!

Try pushing a button! LEDs blink! **Congratulations, you've built your first FPGA project!**

Once you set this up, you should save your Programmer settings so that you only need to go through this once!



Board Project 1 Completion!

For the next project, we will learn about a fundamental FPGA component: Look-Up Tables

```

#####
# iCEcube PCF
# Version:          2014.12.27052
# File Generated:   Apr 27 2015 09:46:33
# Family & Device:  iCE40HX1K
# Package:          VQ100
#####

```

Main FPGA Clock

```
set_io i_Clk          15
```

LED Pins:

```
set_io o_LED_1        56
set_io o_LED_2        57
set_io o_LED_3        59
set_io o_LED_4        60
```

Push-Button Switches

```
set_io i_Switch_1     53
set_io i_Switch_2     51
set_io i_Switch_3     54
set_io i_Switch_4     52
```

7 Segment Outputs

```
set_io o_Segment1_A    3
set_io o_Segment1_B    4
set_io o_Segment1_C    93
set_io o_Segment1_D    91
set_io o_Segment1_E    90
set_io o_Segment1_F    1
set_io o_Segment1_G    2
set_io o_Segment2_A   100
set_io o_Segment2_B    99
set_io o_Segment2_C    97
set_io o_Segment2_D    95
set_io o_Segment2_E    94
set_io o_Segment2_F    8
set_io o_Segment2_G    96
```

UART Outputs

```
set_io i_UART_RX       73
set_io o_UART_TX       74
```

VGA Outputs

```
set_io o_VGA_HSync     26
set_io o_VGA_VSync     27
set_io o_VGA_Red_0     36
set_io o_VGA_Red_1     37
set_io o_VGA_Red_2     40
set_io o_VGA_Grn_0     29
set_io o_VGA_Grn_1     30
set_io o_VGA_Grn_2     33
set_io o_VGA_Blu_0     28
set_io o_VGA_Blu_1     41
set_io o_VGA_Blu_2     42
```

PMOD Signals

```
set_io io_PMOD_1       65
set_io io_PMOD_2       64
set_io io_PMOD_3       63
set_io io_PMOD_4       62
set_io io_PMOD_7       78
set_io io_PMOD_8       79
set_io io_PMOD_9       80
set_io io_PMOD_10      81
```

By just changing the constraints file and the VHDL , the 4 Push Buttons and the 4 LEDs can be used with external hardware, to test the PMOD Interface

```
# #####
# iCEcube PCF
# Version:          2014.12.27052
# File Generated:   Apr 27 2015 09:46:33
# Family & Device:  iCE40HX1K
# Package:          VQ100
# #####

### Main FPGA Clock
set_io i_Clk          15

### LED Pins:
set_io o_LED_1        56
set_io o_LED_2        57
set_io o_LED_3        59
set_io o_LED_4        60

## Push-Button Switches
set_io i_Switch_1     53
set_io i_Switch_2     51
set_io i_Switch_3     54
set_io i_Switch_4     52

### 7 Segment Outputs
set_io o_Segment1_A   3
set_io o_Segment1_B   4
set_io o_Segment1_C   93
set_io o_Segment1_D   91
set_io o_Segment1_E   90
set_io o_Segment1_F   1
set_io o_Segment1_G   2
set_io o_Segment2_A   100
set_io o_Segment2_B   99
set_io o_Segment2_C   97
set_io o_Segment2_D   95
set_io o_Segment2_E   94
set_io o_Segment2_F   8
set_io o_Segment2_G   96

## UART Outputs
set_io i_UART_RX      73
set_io o_UART_TX      74

## VGA Outputs
set_io o_VGA_HSync     26
set_io o_VGA_VSync     27
set_io o_VGA_Red_0     36
set_io o_VGA_Red_1     37
set_io o_VGA_Red_2     40
set_io o_VGA_Grn_0     29
set_io o_VGA_Grn_1     30
set_io o_VGA_Grn_2     33
set_io o_VGA_Blu_0     28
set_io o_VGA_Blu_1     41
set_io o_VGA_Blu_2     42

## PMOD Signals
set_io io_PMOD_1       65    ## i_Switch_1
set_io io_PMOD_2       64    ## i_Switch_2
set_io io_PMOD_3       63    ## i_Switch_3
set_io io_PMOD_4       62    ## i_Switch_4
set_io io_PMOD_7       78    ## o_LED_1
set_io io_PMOD_8       79    ## o_LED_2
set_io io_PMOD_9       80    ## o_LED_3
set_io io_PMOD_10      81    ## o_LED_4
```

```

--
library ieee;
use ieee.std_logic_1164.all;

entity Switches_To_LEDs is
  port (
    -- Push-Button Switches:
    i_Switch_1 : in std_logic;
    i_Switch_2 : in std_logic;
    i_Switch_3 : in std_logic;
    i_Switch_4 : in std_logic;

    -- LED Pins:
    o_LED_1 : out std_logic;
    o_LED_2 : out std_logic;
    o_LED_3 : out std_logic;
    o_LED_4 : out std_logic
  );
end entity Switches_To_LEDs;

architecture RTL of Switches_To_LEDs is
begin
  o_LED_1 <= i_Switch_1;
  o_LED_2 <= i_Switch_2;
  o_LED_3 <= i_Switch_3;
  o_LED_4 <= i_Switch_4;

end RTL;

--Changed to this file to use the PMOD Interface
library ieee;
use ieee.std_logic_1164.all;

entity Switches_To_LEDs is
  port (
    -- Push-Button Switches:
    io_PMOD_1 : in std_logic;  -- switch 1
    io_PMOD_2 : in std_logic;  -- switch 2
    io_PMOD_3 : in std_logic;  -- switch 3
    io_PMOD_4 : in std_logic;  -- switch 4

    -- LED Pins:
    io_PMOD_7 : out std_logic;  -- LED 1
    io_PMOD_8 : out std_logic;  -- LED 2
    io_PMOD_9 : out std_logic;  -- LED 3
    io_PMOD_10 : out std_logic  -- LED 4
  );
end entity Switches_To_LEDs;

architecture RTL of Switches_To_LEDs is
begin
  io_PMOD_7 <= io_PMOD_1;
  io_PMOD_8 <= io_PMOD_2;
  io_PMOD_9 <= io_PMOD_3;
  io_PMOD_10 <= io_PMOD_4;

end RTL;

```

