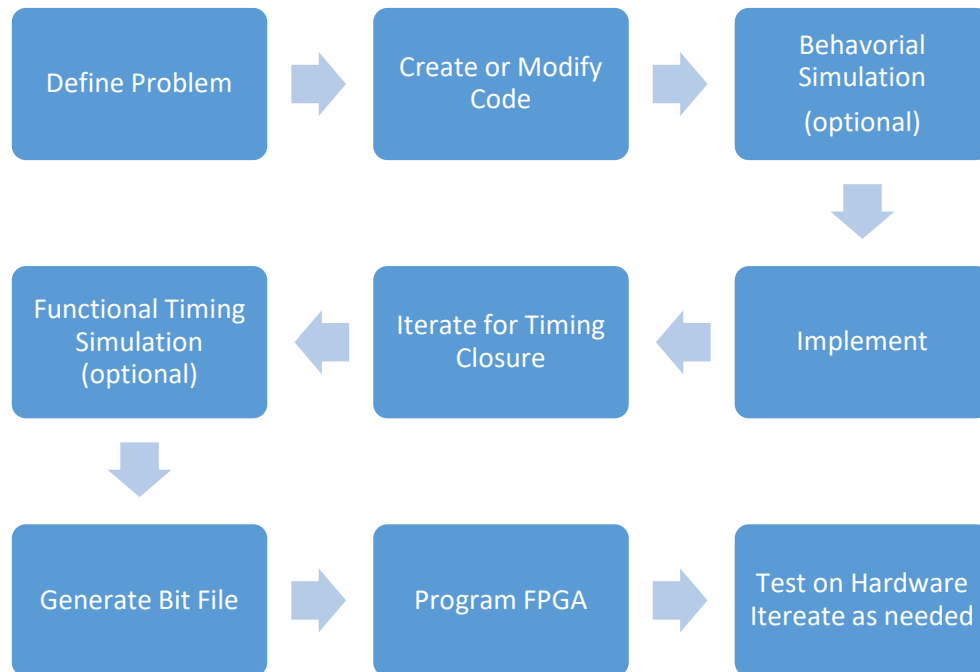


## Work Flow Introduction



This is a simplified workflow for creating a design using an FPGA or FPGA evaluation board and a language like VHDL. Tools such as Vivado from Xilinx allow rapid access to each step starting with the creation of code. Along the way the following will happen:

- Designs are checked for correct language syntax.
- Can the logic requested be physically implemented?
- Simulation at a functional or behavioral level to prove the design matches the description.
- Translation, mapping and place & route steps transform the design into logic hardware found within the FPGA.
- Further simulation is possible to check timing requirements.
- A “BIT file” is created that can be passed to the FPGA or FLASH memory that will finally configure the FPGA to implement the design.
- If the design also contains a processor such as the ZYNC or MicroBlaze, an ELF (Extensible Linking Format) file containing the executable code for the processor is also created.

Often times these steps must be repeated. A lot. Sometimes taking a few steps backwards to correct a problem found along the way. The Vivado design suite allows the designer to jump back quickly to revise portions of a design and then continue again forward through the cycle.

Install the latest edition of Vivado (2020.2 at the time of writing this tutorial). For this tutorial, the only FPGA that need to be supported is the ZYNC family. The free WebPACK license is sufficient for the tutorial designs. Consult the various Xilinx web pages for instructions on downloading, installing and licensing the Vivado software. Include Vitis when installing so that microprocessor code can be written, tested and run alongside logic implemented in the FPGA.

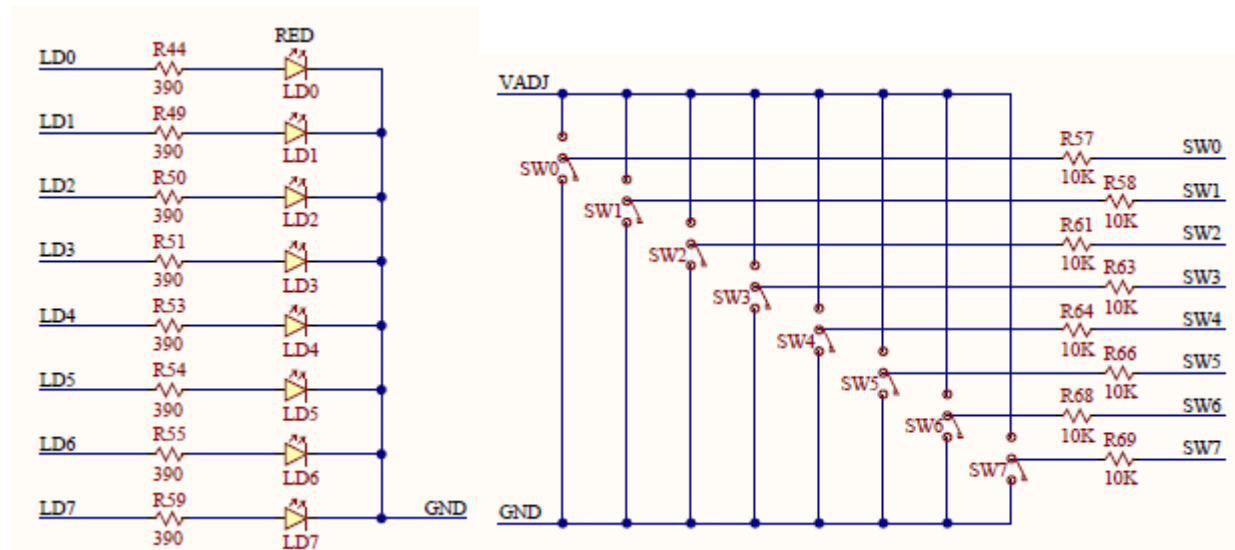
## Define the Problem

Defining the problem can be as simple as just making out a logic table or scribbling some quick logic designs. But often times it requires first writing a high-level description of the problem and gradually breaking each part down into smaller and smaller pieces until one can implement them with the resources found inside an FPGA.

How granular should this description be? It will always be a function of how well the problem is understood and how confident the FPGA designer is in their solution. For the purpose of this module, we will create two designs using the LEDs and switches found on a ZedBoard. The first will be a very simple one in which each switch directly controls the LED above it. In the second design, we will use four switches to implement a simple 2-bit by 2-bit adder. The result and carry will be displayed on the LEDs above the switches. These designs will also be examples of “combinatorial logic”, that is logic whose output is a function only of the current inputs. Combinatorial logic does not require clocks or take into account previous input values (i.e. memory). Designs are frequently a combination of sequential logic and combinatorial logic. And those logic blocks are running simultaneously and interacting.

## First Design : Direct Control of LEDs Using Switches on the ZedBoard

Consulting the schematics for the ZedBoard we see that each LED (LD0 to LD7) has their cathode connected to digital ground (GND), the anode connected to a series resistor of 390Ω which is in turn connected to an I/O pin of the FPGA. For each of the switches (SW0 to SW7), they are single pole-double throw (SPDT) where the common terminal of the switch moves between GND and VADJ. VADJ is the overall logic level for the board and should be set to 2.5V using the J18 jumper found on the ZedBoard. This value of VADJ=2.5V will become important later. The common terminal of the switch is then connected through a 10kΩ series resistor to an I/O pin on the FPGA.

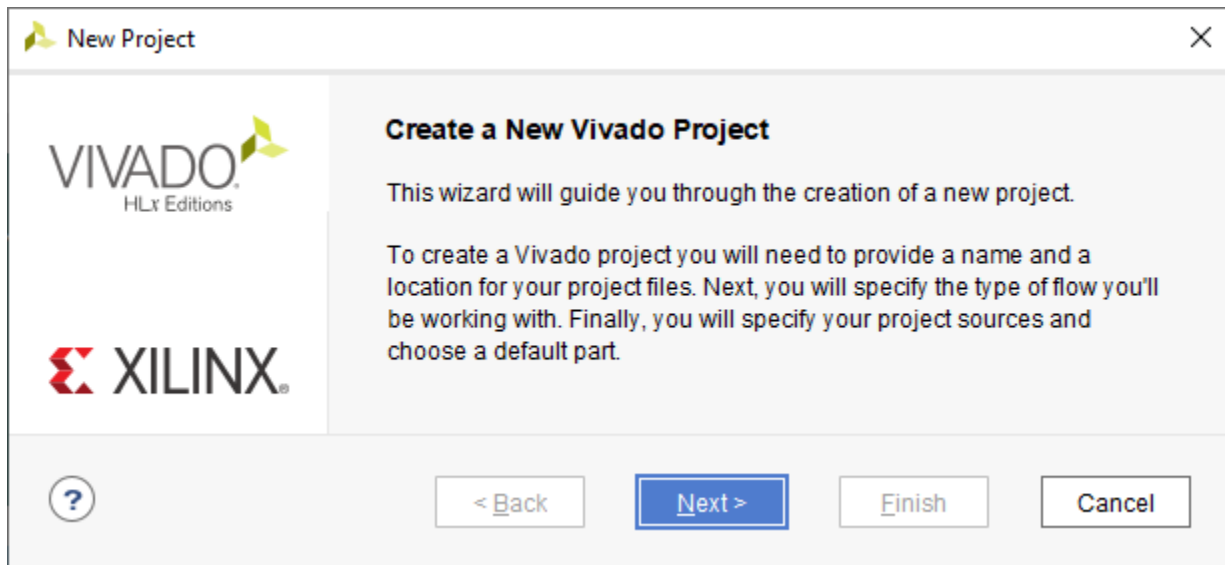


There are tables available in the ZedBoard User's Guide as well as pre-build XDC constraint files that have named all the signals associated with each pin of the ZedBoard. The XDC file, User's Guide, schematic and all other up-to-date files associated with the ZedBoard can be found at [Zedboard.org](http://Zedboard.org).

In this case, our simple design will have the FPGA read a value from SW0 to SW7 and translate that to signals driving the pins associated with the LEDs. Thus, if SW0 is connected to VADJ, the FPGA would put 3.3V on the pin associated with LDO. Note the LED I/O pins are permanently associated with the FPGA I/O banks that operate at 3.3V. This is a consequence of the way the ZedBoard schematic was created and not necessarily a restriction of the FPGA itself. Different FPGAs may have 1.8V or other I/O driver voltages available and could still read switches or drive LEDs with the appropriate external circuitry.

### Creating the Project and Code

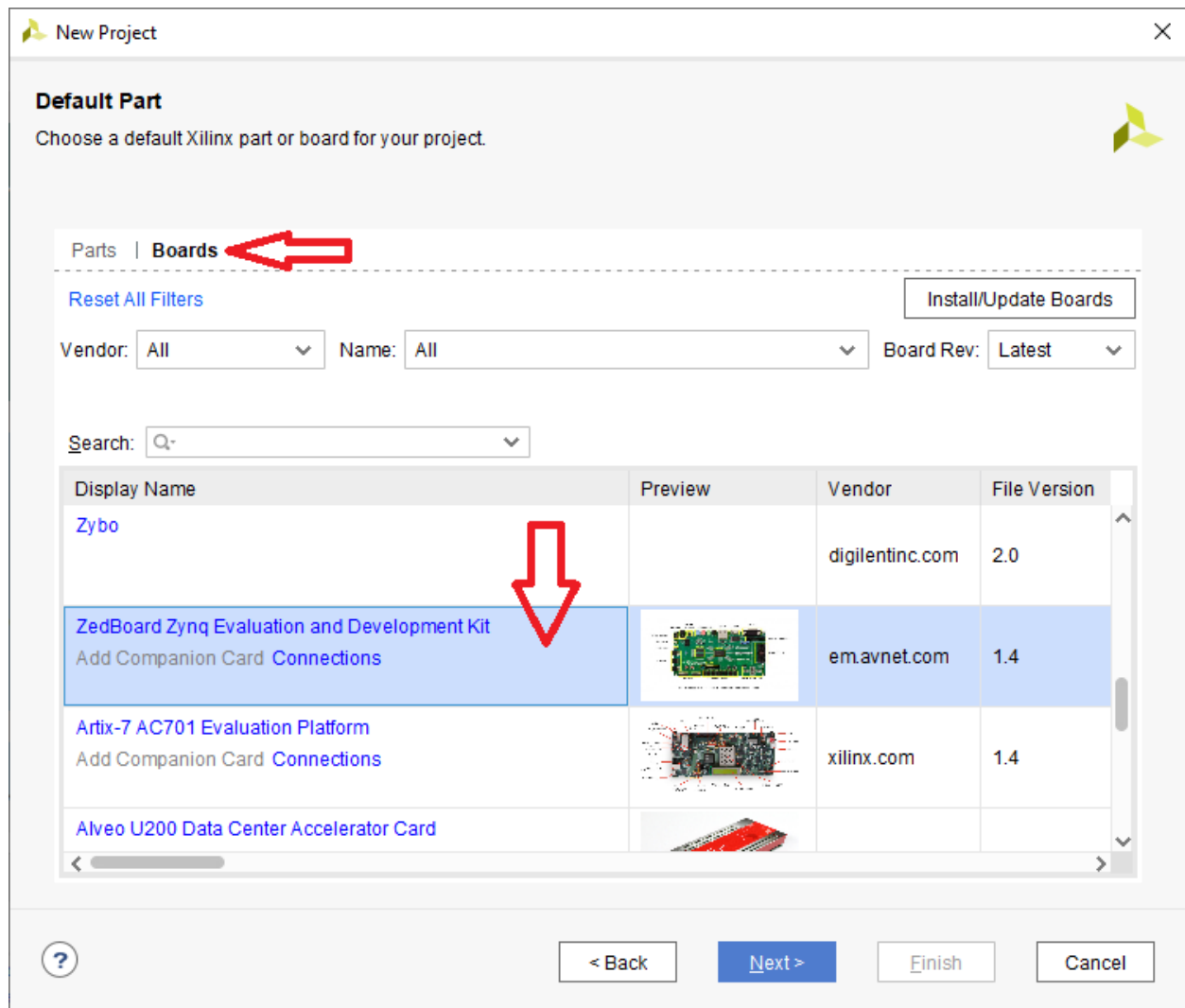
Start Vivado and under Quick Start select Create Project. We are going to use the Vivado project Wizard to facilitate project creation.



- Pick a good name for the project. The default is “project\_1” which is mostly useless. Also select a reasonable project location path. Avoid paths that live on remote drives as this will slow down compiling. Also consider that paths on SSD drives, while faster, will reduce the life of the drive and may not have sufficient room.
- We are going to create an RTL (Register Transfer Logic) project. In this context RTL generally represents projects that can be fully realized in hardware.
  - Tic the RTL project radio button in the Project Type Dialog.
  - Un-tic the “Do not specify sources at this time” option button. We are going to define our source file at the beginning of the design.
- In the Add Sources dialog, confirm that the Target language and Simulator language are both set to VHDL. You may use Verilog as the design language but this tutorial will be written using VHDL.
- Click “Create File” to define a design file.
  - Our design will have a single file and typically the first file to be created is the topmost file of the hierarchy. The top file is the one that defines how the FPGA interacts with the rest of the circuit. A logical file name would be “top”.
  - Unless you have a very good reason to change the file location, leave it as local to project.
- As there is only one file in this design, after clicking Next you are prompted to include constraints. Constraints are things like pin assignments, voltage levels, timing restrictions, etc.

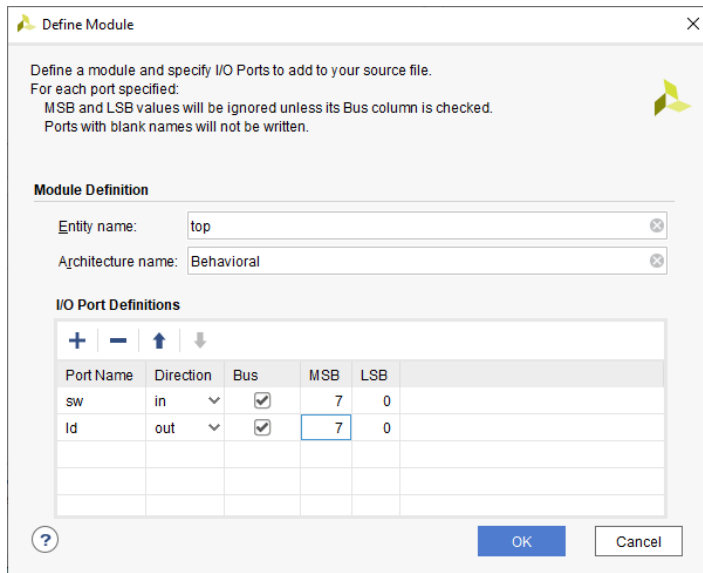
Because we are using a pre-defined ZedBoard project environment we do not need to add any constraints at this time.

- Finally select the part or board that is going to contain the design. We are using a ZedBoard. If you cannot find the ZedBoard Zynq Evaluation and Development Kit listed, locate the instructions at [Zedboard.org](http://Zedboard.org) for downloading and installing the necessary design files. Then return to this project.



After a bit of time a new dialog requesting that module definitions be added. We have only one module, "top.vhd". Leave the entity name the same as the root name of the module. And the architecture choice is Behavioral because we will be writing our VHDL code to describe the behavior of the module. Other choices such as Simulation, RTL or even Memory exist. All the choices are more to do with self-documenting the code than the specifics of compiling but it is best to stick to the common choices for VHDL. These are documented in the VHDL standard and various VHDL reference books.

We will need two I/O ports, one for the LEDs (output) and one for the switches (input). Add them using the lower portion of the dialog.



Define Module

Define a module and specify I/O Ports to add to your source file.  
For each port specified:  
MSB and LSB values will be ignored unless its Bus column is checked.  
Ports with blank names will not be written.

**Module Definition**

Entity name:

Architecture name:

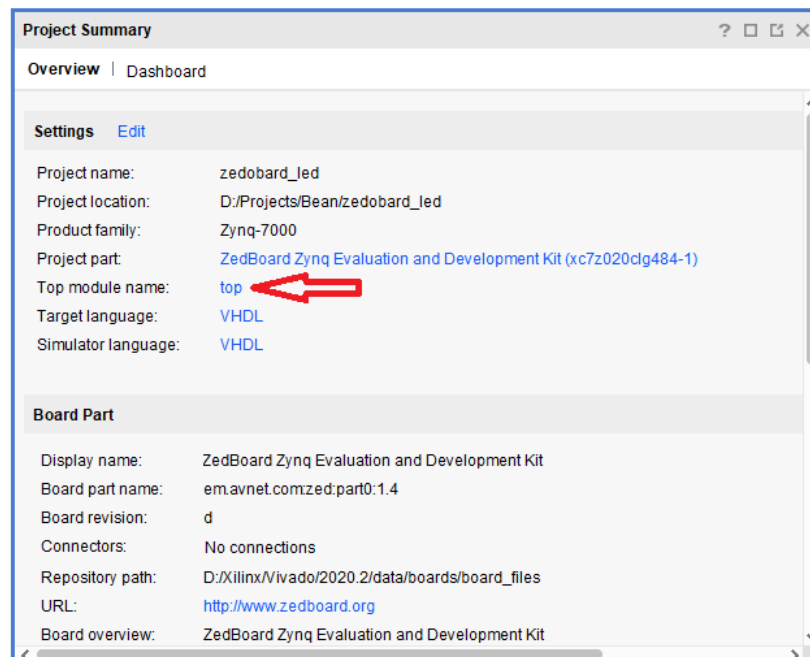
**I/O Port Definitions**

Port Name	Direction	Bus	MSB	LSB
sw	in	<input checked="" type="checkbox"/>	7	0
ld	out	<input checked="" type="checkbox"/>	7	0

OK Cancel

The port names selected should logically match the design. In this case, “sw” and “ld” are pre-defined values in the ZedBoard design and will save us time later in our design process. One could use “LED” or “SWITCH” or “Laurel” or “Hardy” but generally the names should match their purpose. Case is ignored in VHDL.

All VHDL projects need a “top module” and in our case it is the only module. Sometimes Vivado picks up on the naming convention and makes the assignment automatically. Other times it doesn’t. Look at the Project Summary Overview and confirm the top module name is “top”. This is another chance to change the target and simulator languages to VHDL if necessary.



Project Summary

Overview | Dashboard


**Settings** Edit

Project name: zedobard\_led

Project location: D:/Projects/Bean/zedobard\_led

Product family: Zynq-7000

Project part: [ZedBoard Zynq Evaluation and Development Kit \(xc7z020clg484-1\)](#)

Top module name: **top** 

Target language: VHDL

Simulator language: VHDL

**Board Part**

Display name: ZedBoard Zynq Evaluation and Development Kit

Board part name: em.avnet.com:zed:part0:1.4

Board revision: d

Connectors: No connections

Repository path: D:/Xilinx/Vivado/2020.2/data/boards/board\_files

URL: <http://www.zedboard.org>

Board overview: ZedBoard Zynq Evaluation and Development Kit

If we had skipped the creation of the module(s) when creating the project, modules can be added at any time using the “+” symbol in the sources dialog or Alt+A.

Double clicking on the module name will open the file and allow editing. By default, Xilinx add a header of comments for describing the file (USE THEM – GOOD DOCUMENTATION SAVES TIME) as well as shows the general structure of a VHDL file.

```
-----
-- Company:
-- Engineer:
--
-- Create Date: 03/02/2021 10:02:08 AM
-- Design Name:
-- Module Name: top - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity top is
    Port ( sw : in STD_LOGIC_VECTOR (7 downto 0);
          ld : out STD_LOGIC_VECTOR (7 downto 0));
end top;

architecture Behavioral of top is

begin

end Behavioral;
```

Consult the many VHDL tutorials and textbooks available for details on syntax. Please note that older tutorials (those written before 2007) may advocate the use of the “bit” data type and other older deprecated aspects of VHDL. Don’t use them as it will make code maintenance more difficult in the long run.

Our code for describing the behavior of the FPGA appears between the “begin” and “end Behavioral” lines. Because all we are doing is directly connecting the state of the switches to the state of the LEDs, we have just one line of code:

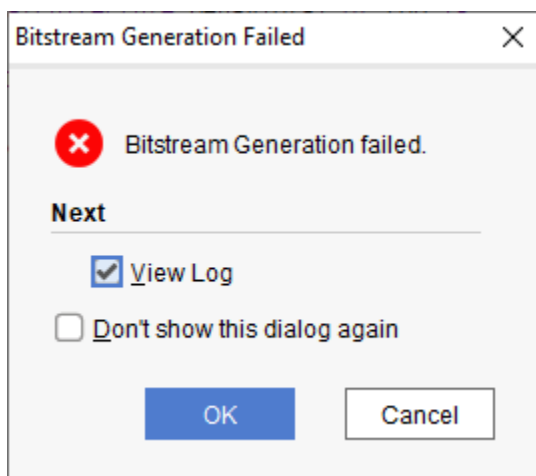
```
ld <= sw;
```

## Compiling Project and Fixing Errors

As this is a small, simple project we might be tempted to go straight to “Generate Bitstream”, the last step of the flow navigator. Then expect to drop right into flipping switches and seeing LEDs. But there is a small error in our project. To find it we will go ahead and attempt to generate the bitstream by clicking the option. Alternately, it can be selected from the Flow menu. This takes a little bit of time depending on the PC’s speed and memory. The first time this is done you may be given the option to assign how many cores of the PC are to be used. Unless you have a good reason, always select the maximum number or cores or jobs from the dropdown selected.

The status of the project compile / built appears in the upper right corner of the Vivado IDE.

After a short time you will be presented with an error dialog indicating the bitstream failed to generate. View the log tab as well as the messages tab at the bottom of the IDE.



The error is in the pin assignments or “pin planning” choices we made. At the top level, there should be a direct correlation between signal names and pins of the FPGA (not really pins any more as they are frequently BGA – Ball Grid Array – components but the name has stuck around). While we have picked good names such as SW and LD and they do match the names on the schematic we haven’t finished defining what or where they are on the FPGA. This is done in a constraints file.

When we picked the ZedBoard at the beginning of our project, there was an implied set of constraints imported and they do contain names like LD and SW but not quite in the right format. Go locate the “zedboard\_master\_XDC\_RevC\_D\_v3.xdc” at [zedboard.org](http://zedboard.org). Once downloaded and unpacked you can look at the file with a text editor. Search for SW or LD.

You will find text that looks like this:

```
# -----
# User DIP Switches - Bank 35
# -----
set_property PACKAGE_PIN F22 [get_ports {SW0}]; # "SW0"
set_property PACKAGE_PIN G22 [get_ports {SW1}]; # "SW1"
set_property PACKAGE_PIN H22 [get_ports {SW2}]; # "SW2"
set_property PACKAGE_PIN F21 [get_ports {SW3}]; # "SW3"
set_property PACKAGE_PIN H19 [get_ports {SW4}]; # "SW4"
set_property PACKAGE_PIN H18 [get_ports {SW5}]; # "SW5"
set_property PACKAGE_PIN H17 [get_ports {SW6}]; # "SW6"
set_property PACKAGE_PIN M15 [get_ports {SW7}]; # "SW7"
```

The problem isn't in choosing "SW" or "LD" for the root signal name, the problem is in how to deal with multiples of the signal, i.e. an array.

The master XDC file makes the assumption that all the signals are individually named, SW0, SW1, etc. However we created a VECTOR or array of signals so our signals have individual names like SW[0] or LD[3]. We made a vector assignment "ld <= sw;". We could have assigned them individually like "ld[0] <= sw[0]" which is useful if we wanted to scramble the assignments or use only a portion of the vector.

We can either fix the top.vhd file to list individual signal names (considerable typing and clutter) or we can provide an alternate set of definitions in our own XDC file which is processed at the end of the project compile.

Return to the Project Manager and add a source, this time selecting the "add or create constraints" radio button. Again create a file and give it a logical name such as "my\_constraints". Then locate the new file in the Sources dialog and open by double clicking the name. Into the blank file we are going to cut-and-paste from the master XDC and then edit lines.

Old :

```
# -----
# User DIP Switches - Bank 35
# -----
set_property PACKAGE_PIN F22 [get_ports {SW0}]; # "SW0"
set_property PACKAGE_PIN G22 [get_ports {SW1}]; # "SW1"
set_property PACKAGE_PIN H22 [get_ports {SW2}]; # "SW2"
set_property PACKAGE_PIN F21 [get_ports {SW3}]; # "SW3"
set_property PACKAGE_PIN H19 [get_ports {SW4}]; # "SW4"
set_property PACKAGE_PIN H18 [get_ports {SW5}]; # "SW5"
set_property PACKAGE_PIN H17 [get_ports {SW6}]; # "SW6"
set_property PACKAGE_PIN M15 [get_ports {SW7}]; # "SW7"

# -----
# User LEDs - Bank 33
# -----
set_property PACKAGE_PIN T22 [get_ports {LD0}]; # "LD0"
set_property PACKAGE_PIN T21 [get_ports {LD1}]; # "LD1"
set_property PACKAGE_PIN U22 [get_ports {LD2}]; # "LD2"
set_property PACKAGE_PIN U21 [get_ports {LD3}]; # "LD3"
set_property PACKAGE_PIN V22 [get_ports {LD4}]; # "LD4"
set_property PACKAGE_PIN W22 [get_ports {LD5}]; # "LD5"
set_property PACKAGE_PIN U19 [get_ports {LD6}]; # "LD6"
set_property PACKAGE_PIN U14 [get_ports {LD7}]; # "LD7"
```

New:

```
# -----
# User DIP Switches - Bank 35
# -----
set_property PACKAGE_PIN F22 [get_ports {sw[0] }]; # "SW0"
set_property PACKAGE_PIN G22 [get_ports {sw[1] }]; # "SW1"
set_property PACKAGE_PIN H22 [get_ports {sw[2] }]; # "SW2"
set_property PACKAGE_PIN F21 [get_ports {sw[3] }]; # "SW3"
set_property PACKAGE_PIN H19 [get_ports {sw[4] }]; # "SW4"
set_property PACKAGE_PIN H18 [get_ports {sw[5] }]; # "SW5"
set_property PACKAGE_PIN H17 [get_ports {sw[6] }]; # "SW6"
set_property PACKAGE_PIN M15 [get_ports {sw[7] }]; # "SW7"

# -----
# User LEDs - Bank 33
# -----
set_property PACKAGE_PIN T22 [get_ports {ld[0] }]; # "LD0"
```



```

set_property PACKAGE_PIN T21 [get_ports {ld[1]}}]; # "LD1"
set_property PACKAGE_PIN U22 [get_ports {ld[2]}}]; # "LD2"
set_property PACKAGE_PIN U21 [get_ports {ld[3]}}]; # "LD3"
set_property PACKAGE_PIN V22 [get_ports {ld[4]}}]; # "LD4"
set_property PACKAGE_PIN W22 [get_ports {ld[5]}}]; # "LD5"
set_property PACKAGE_PIN U19 [get_ports {ld[6]}}]; # "LD6"
set_property PACKAGE_PIN U14 [get_ports {ld[7]}}]; # "LD7"

```

Note that the signal name IS case-sensitive in an XDC file! Because we defined our signals to be “sw” and “ld” instead of “SW” and “LD”, the entries in the XDC file must match.

But keywords like “PACKAGE\_PIN” could be entered as “package\_pin” and still work.

It is possible to define arrays of signals in the XDC file format, sometimes it is easier and more clear to define individual signals. We will do this to define the voltage signaling standard for the pins by setting the entire bank.

Locate the following lines in the master XDC and copy them to our constraints file:

```

set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 33]];
set_property IOSTANDARD LVCMOS25 [get_ports -of_objects [get_iobanks 34]];
set_property IOSTANDARD LVCMOS25 [get_ports -of_objects [get_iobanks 35]];

```

Particularly for IO Banks 34 & 35 (where the switches connect), these lines may be commented out (“#”). Remove the comment and make sure the standard says “LVCMOS25” and that the J18 jumper on the Zedboard is set for 2.5V.

Save the file and try again to create a bitstream. Because we made a rather fundamental change to the file structure by adding constraints, Vivado will need to start-over in the compile sequence and resynthesize, route and implement the design. Ultimately it should finish with no errors but perhaps some warnings.

## Simulating and Implementing

This is a simple design and there isn’t a tremendous amount to learn from a simulation. We could jump straight into implementation and look at the results on the ZedBoard. By generating a bitstream, we are essentially ready to do this.

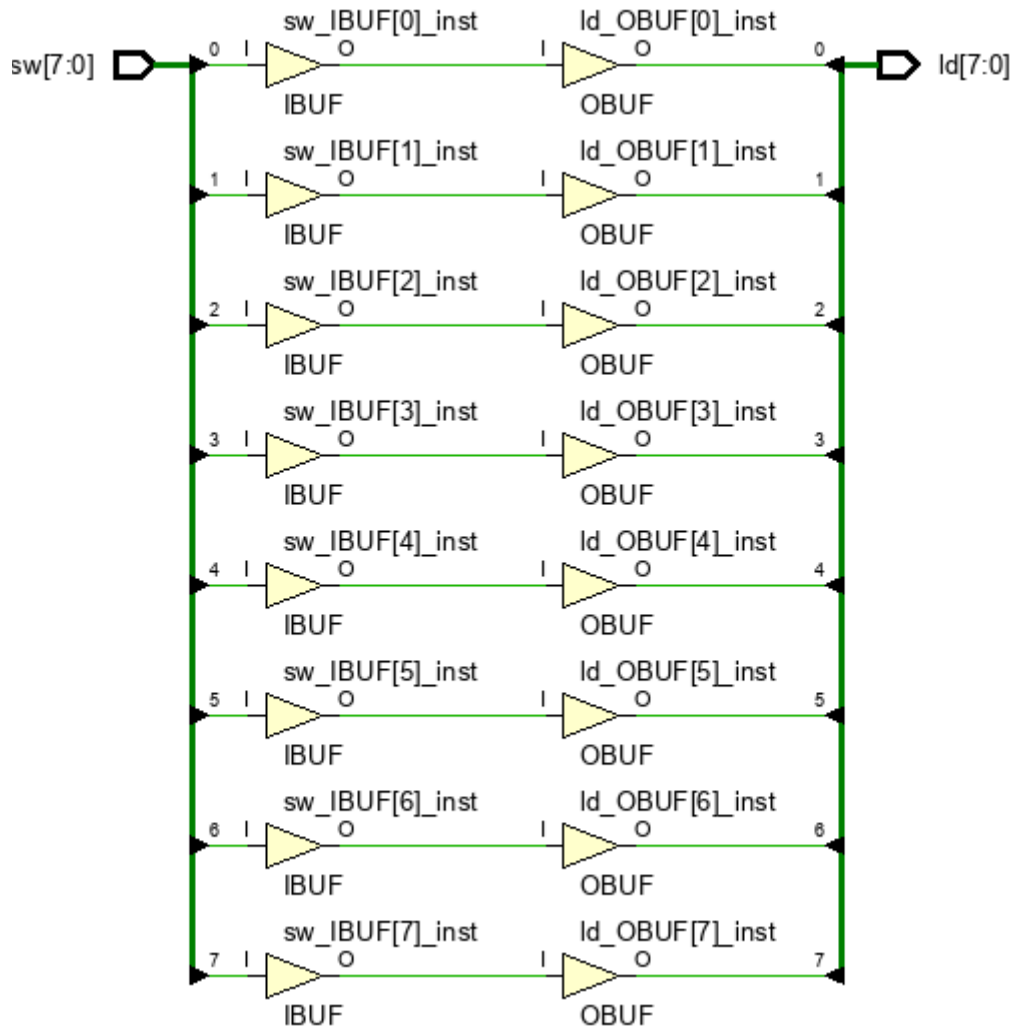
But because this is a simple design, it is also simple to explore and learn what Vivado can tell us about how the hardware was configured. In the short term, we will skip the simulation and explore the implementation.

Implementation is a multi-step process in Vivado that includes processes called Synthesis and Elaboration. These appear in the Project Manager dialog and in the Flow menu. By examining an elaborated design we can learn what sort of translation from VHDL to logic elements are planned by Vivado. Looking at the proposed logic schematic is sometimes helpful. In this case the schematic is ridiculously simple and can be represented by a single bus connecting inputs to outputs.



But in a more complicated design, one might see various logic symbols, flip-flops, memory elements, selectors, etc. By highlighting various items and browsing properties some design flaws can be located early or various properties pre-assigned.

Moving on to the Synthesis step, there is the option to “Open Synthesized Design”. There various options exist for reporting and further constraining a design. Again, the Schematic may prove insightful when debugging a design.



Here it is a bit more detailed what is happening inside the FPGA. Each signal is first buffered (IBUF) then routed to the corresponding OBUF for the LED pin and finally appears on the `Id[7:0]` output bus.

The “Constraints Wizard” can be used to define critical clock paths (our design has none) or other aspects related to timing. Further, near the bottom of the IDE a port called “I/O Ports” appears and there one can assign the pins, I/O standard, drive and termination of signals. These assignments will be collated into an XDC file if one has not already been assigned. Frequently this is an easier way to create the project specific XDC file instead of trying to remember all the syntax of XDC files.

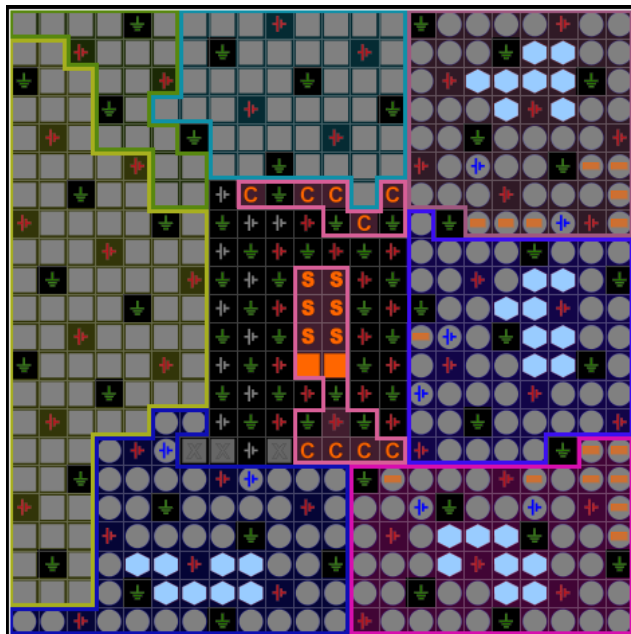
In the upper right corner a drop down menu allows you to switch context views of the project. In the Synthesis context, one can change to a view of the project “floorplan”, even drilling down to the individual gate level inside the silicon to see how signals move from one block to another within the FPGA. In a simple design this isn’t necessary but in more complicated designs

The step after synthesis, assuming it completes with no errors is “Implementation”. This is when the generic logic created from synthesis is actually turned a design that can be mapped into the FPGA’s physical resources. Once again, we can look at the resultant schematic as well as other aspects of the implementation.



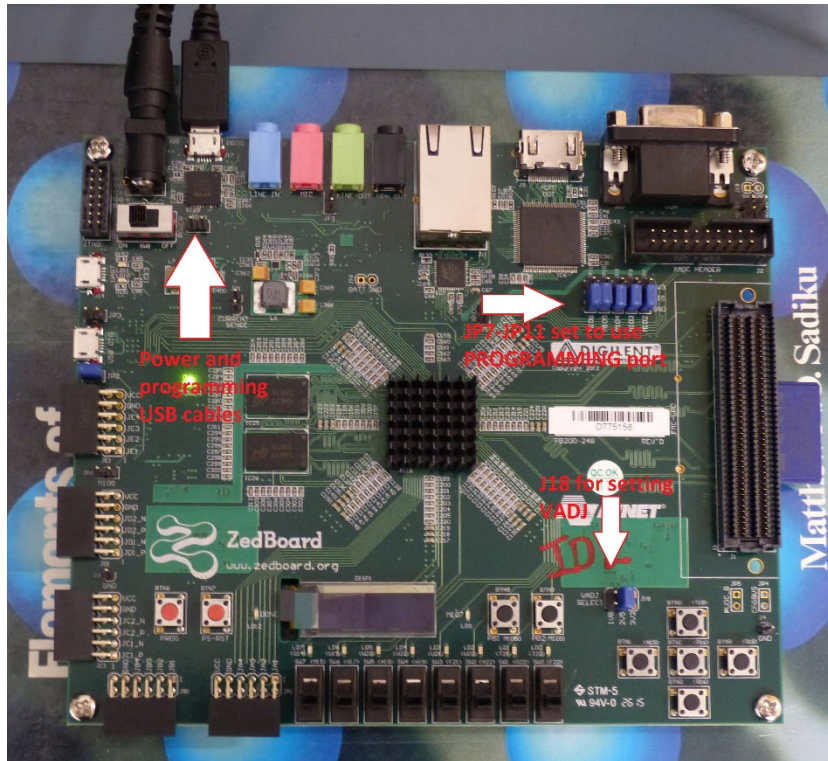
This is not substantially different than the other two schematics we’ve viewed. And it should be no surprise since there isn’t anything except I/O buffers present in the design.

Once again we can get views into how things are mapped into the FPGA by changing the aspect view. For example, the I/O Planning view in the Implementation shows how the various signals appearing on the bottom of the FPGA package (balls of the Ball-Grid-Array, BGA) appear.



## Bitfiles and Programming

Because our simple design doesn't need simulation we will move directly to programming a ZedBoard with our freshly generated bitfile. With the power supply attached and a USB cable attached to the Micro-USB connection next to the power switch (J17 PROG), turn on the ZedBoard. The PC may beep as the USB port is detected.

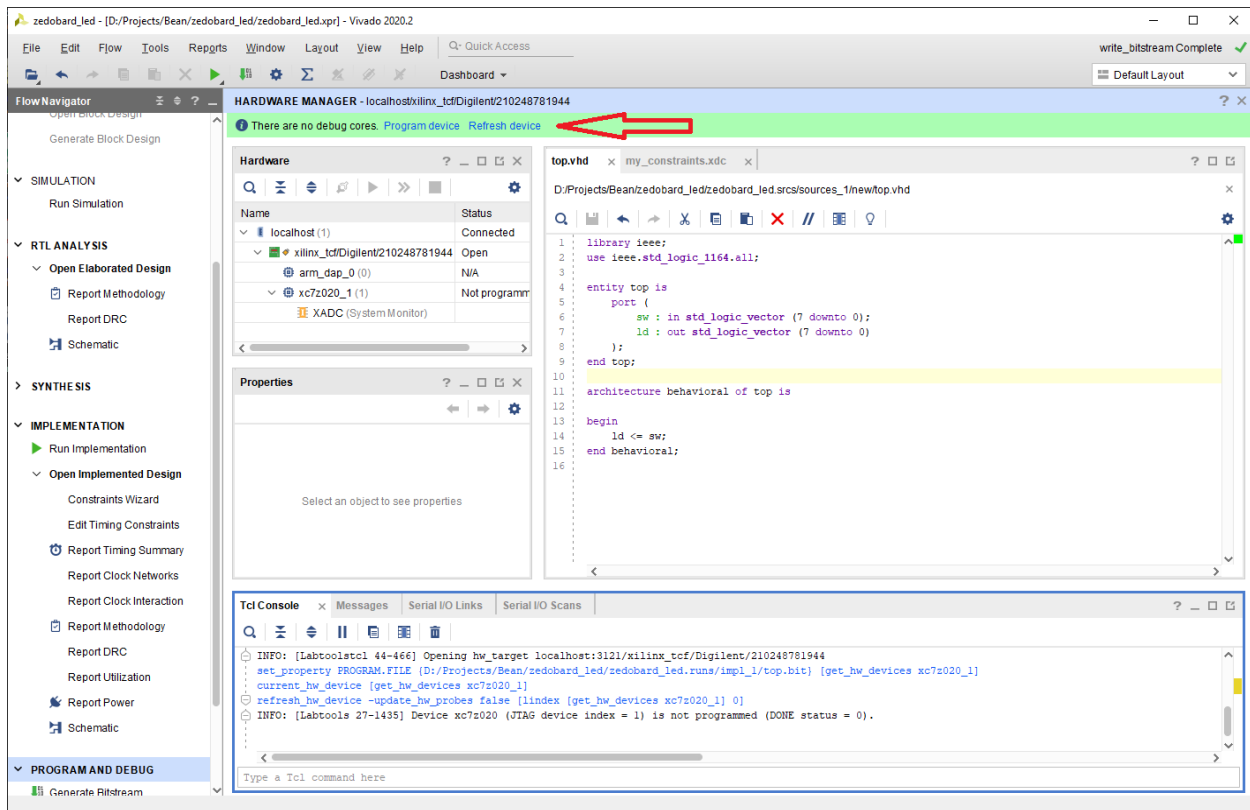


Before beginning to program the board, confirm the following jumper settings:

- JP18 should be set to VADJ
- JP7, JP8, JP9, JP10 and JP11 should all be set to their “lower” position shorting SIG and GND. This forces the FPGA to accept programming ONLY from the USB programming port.

In Vivado, pick “Open Hardware Manage” under Generate Bitstream or from the Flow. There should be a green bar appear near the top of the IDE with “Open Target”, click there to open the target (Zedboard), selecting “Auto Connect” if given the option. The software should automatically find the correct USB port.

If all goes well the green bar should now say “Program Device” and “Refresh device” and one can program the device (FPGA). Once the program has been loaded in the FPGA, start toggling the switches. As each switch is moved to the “UP” position, the corresponding LED should turn on.



Alternative to using the links on the green bar, right click over the FPGA part number (xc7Z020\_1) and from there reprogram the FPGA.

## Enjoy LEDs

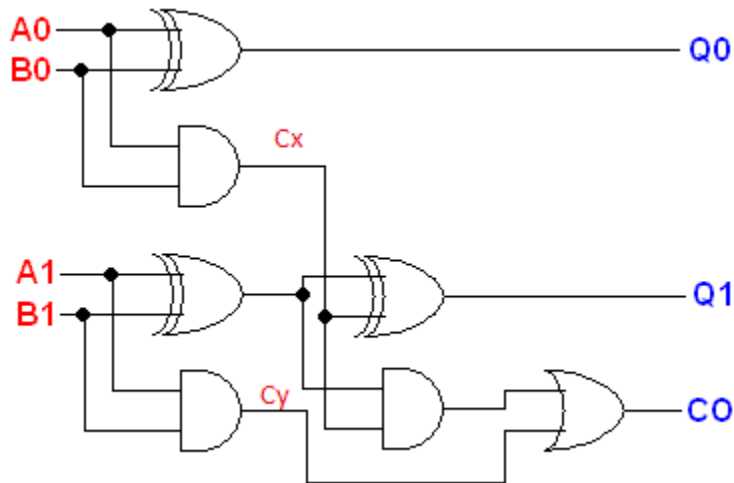
Once programmed and test the switch to LED connection consider changes that can be made.

Modify the VHDL code to turn on an LED when the switch is DOWN (i.e. open). Or reverse the order of switch to LED connection. There are multiple ways to do both tasks with respect to the VHDL syntax but the resultant synthesis schematics and implementation schematics should be the same.

## Second Design : Simple 2-bit Adder

Using SW[3..0] and LD[2..0] implement a full 2-bit adder with carry using logic statements in VHDL.

This is what a 2-bit adder with carry looks like as a schematic:



SW[0] will drive A0, SW[1] to B0 and so on. LD[0] will indicate the value of Q0, LD[1] for Q1, etc.

One possible implementation in VHDL would be:

```

architecture Behavioral of led_1 is
    signal Q0, Q1          : std_logic;
    signal Co              : std_logic;
    signal Cx, Cy          : std_logic;
    signal A0, A1, B0, B1 : std_logic;

begin
    -- 2 bit x 2 bit adder using gate descriptions
    A0 <= sw(0);
    A1 <= sw(1); -- temporary assignments to make equations match our drawing
    B0 <= sw(2);
    B1 <= sw(3);

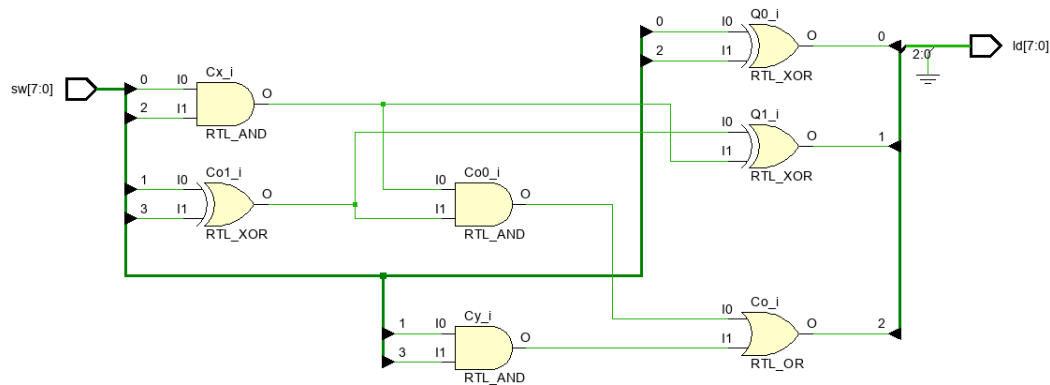
    Q0 <= A0 xor B0;
    Cx <= A0 and B0;
    Q1 <= (A1 xor B1) xor Cx;
    Cy <= A1 and B1;
    Co <= (Cx and (A1 xor B1)) or Cy;

    -- rest of the switches are ignored.
    -- set the output LEDs
    ld(0) <= Q0;
    ld(1) <= Q1;
    ld(2) <= Co;
    ld(7 downto 3) <= "00000"; -- unused, turn them off

end Behavioral;
  
```

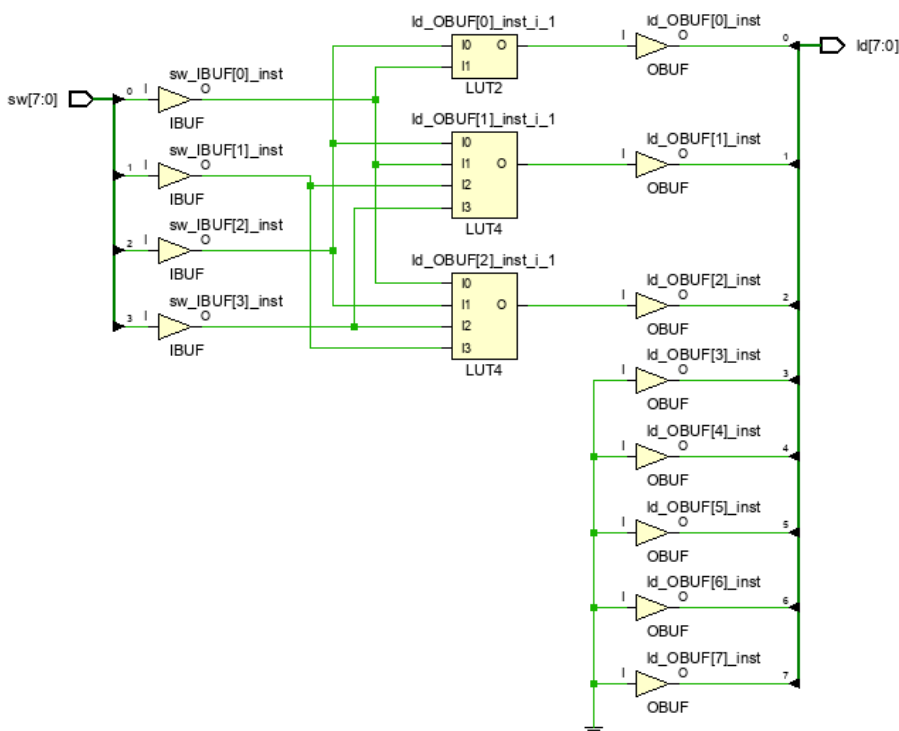
To make the code easier to read, the ports are mapped to temporary signals such as A0, B1, etc. There is no penalty to making temporary or internal use only signal assignments as the VHDL compiler will ultimately make the correct connections. Generally, one should focus on readability of the code and allow the compiler to handle the connections.

The elaborated schematic should very closely resemble our planned schematic:



And it does if you carefully trace out the paths and re-arrange the location of some of the symbols.

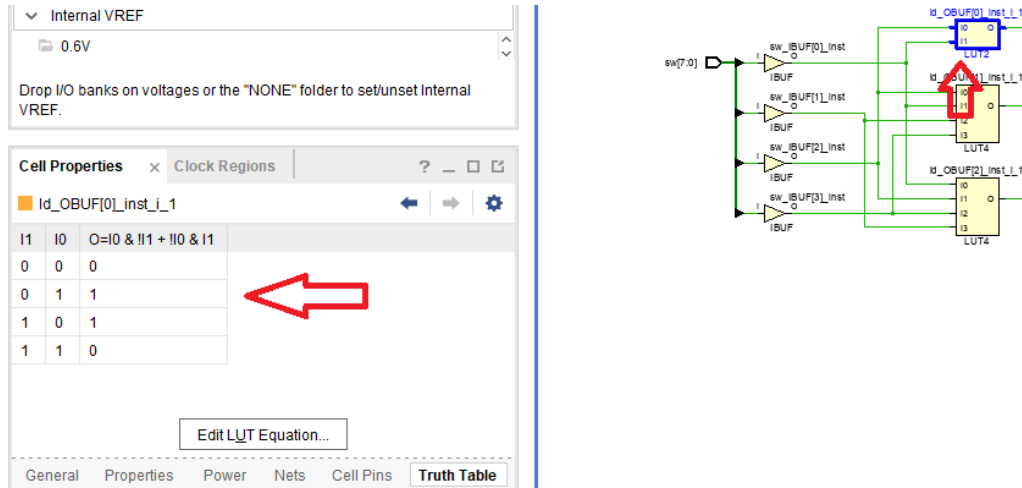
But now if you synthesize and re-examine the schematic it starts to shed light on what actually happens inside the FPGA.



The 4 un-used LEDs are being driven by ground (zero) as intended by of VHDL. But gone are the individual logic gates, replaced by symbols labeled LUT2 and LUT4. These stand for Look-Up-Table 2-input and 4-input. FPGAs are generally well appointed with RAM or flip-flops and not individual logic

gates. And because any logic that can be implemented as gate level equations as we did can be translated to one or more interconnected look-up tables which are simply RAM.

The “truth table” for any particular LUT is available for viewing and editing by clicking once on the desired LUT and then examining the “Truth Table” option in the Cell Properties window.



## Moving Beyond Logic Gates

The power of VHDL is that you don't need to write every operation out as a series of combinatorial logic gates. The compiler can deduce your intentions from the VHDL syntax. For example, the addition operation can be written as “sum <= A + B” if we properly define the data types for sum, A and B.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity led_1 is
    port (
        sw : in std_logic_vector (7 downto 0);
        ld : out std_logic_vector (7 downto 0)
    );
end led_1;

architecture archi of led_1 is

    signal a : unsigned(2 downto 0);
    signal b : unsigned(2 downto 0);
    signal sum : unsigned(2 downto 0);

begin
    -- map std_logic_vector to unsigned
    a <= unsigned('0' & sw(1 downto 0));
    b <= unsigned('0' & sw(3 downto 2));

    -- do the sum!
    sum <= a + b;

    -- now put answer back onto leds
    ld(2 downto 0) <= std_logic_vector(sum);
    ld(7 downto 3) <= "00000";

end archi;
```



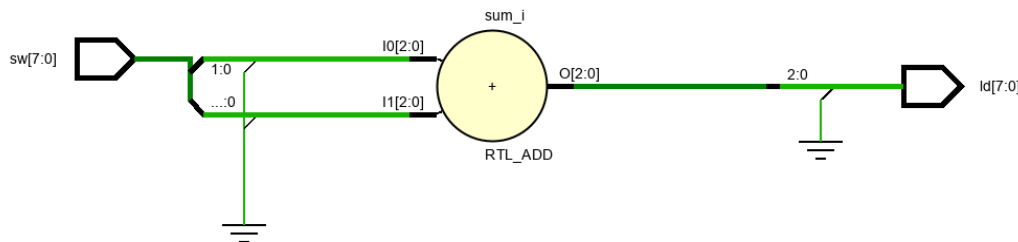
The ports of the module remain “sw[7..0]” and “ld[7..0]” with their std\_logic\_vector data type but inside the module we convert them to the “unsigned” data type so that we may use every-day arithmetic operations like “+”. It is possible to have module ports be “unsigned” or “signed” or “integer” too.

The lines

```
-- map std_logic_vector to unsigned
a <= unsigned('0' & sw(1 downto 0));
b <= unsigned('0' & sw(3 downto 2));
```

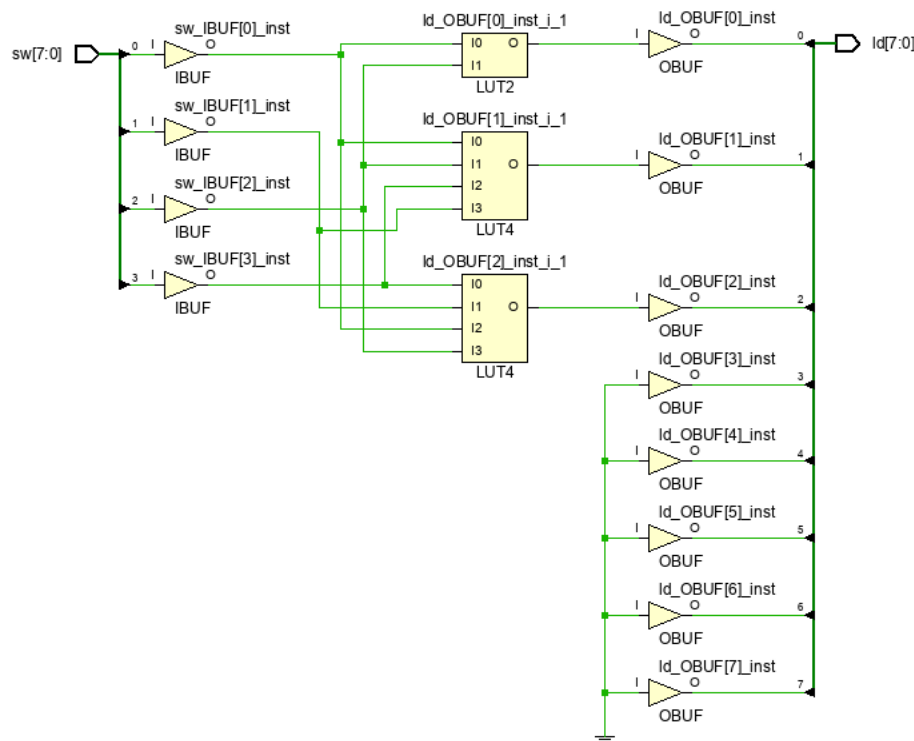
Handle the type conversion and the result from sum is converted back to a std\_logic\_vector for display on the leds.

Once again examining the elaborated schematic we see it has changed again:



This time reflecting our desire to add two numbers and present the result on output ports.

The Synthesized schematic though looks very familiar:



Because while we made a significant change to the content of the code we didn't change the way the code functions when viewed only from its input and output ports.

### Confirm Code Operation

Return to the first implementation using the logic gates and complete the compile all the way to a bit stream. Fill out the truth table confirming the operation. Then change the code to the addition form and compile and test again. Both sets of truth tables should match.

B1	B0	A1	A0	arith	Co	Q1	Q0
0	0	0	0	0+0			
0	0	0	1	0+1			
0	0	1	0	0+2			
0	0	1	1	0+3			
0	1	0	0	1+0			
0	1	0	1	1+1			
0	1	1	0	1+2			
0	1	1	1	1+3			
1	0	0	0	2+0			
1	0	0	1	2+1			
1	0	1	0	2+2			
1	0	1	1	2+3			
1	1	0	0	3+0			
1	1	0	1	3+1			
1	1	1	0	3+2			
1	1	1	1	3+3			

B1	B0	A1	A0	arith	Co	Q1	Q0
0	0	0	0	0+0			
0	0	0	1	0+1			
0	0	1	0	0+2			
0	0	1	1	0+3			
0	1	0	0	1+0			
0	1	0	1	1+1			
0	1	1	0	1+2			
0	1	1	1	1+3			
1	0	0	0	2+0			
1	0	0	1	2+1			
1	0	1	0	2+2			
1	0	1	1	2+3			
1	1	0	0	3+0			
1	1	0	1	3+1			
1	1	1	0	3+2			
1	1	1	1	3+3			

### Conclusion

The Vivado IDE provides tremendous flexibility in defining how a project will be implemented with respect to signal-to-pin relationships, input and output voltage levels and drive characteristics, timing and even exactly where inside the FPGA a circuit is implemented.

The VHDL compiler on the other hand will generally produce the same output when given clear and unambiguous instructions even when there are multiple ways of producing the same result.

Remember that it may be crucial to look at various intermediate steps when implementing more complex designs to confirm that the compiler understands what you have asked it to create.

Finally, the Vivado IDE can program the FPGA. Not shown were other features such as software simulation (the next module) or hardware debugging with the design running in hardware.