VHDL 101

Patrick Mintram

December 2, 2019

1 Introduction

This session is intended to wet your appetite to the world of FPGAs, so that if you choose to you can start having a play in your own time. This session is:

1. A brief overview of VHDL
2. A chance to get hands on with some Hardware
3. A chance to make a `hello world` in Hardware

Things this session is *not*:

1. An introduction to Digital Design
2. A comprehensive deep dive into VHDL
3. Likely to finish on time

By the end of this session you should have configured an FPGA to react to a switch input, and to see that output on an LED.

If you want to get straight to writing of VHDL using the reference project skip to section 6: "OK, now lets actually do some VHDL".

2 How to use this guide

It is not expected that session participants follow this document step by step. It expected that participants reference it to work at own pace, or to be able to recreate the work in their own time. Participants are encouraged to make notes are they require on this document, once printed.
The hand icon will appear when something is purposely had some handwaving applied to it to keep it simple.

Anything in a box is an instruction to complete, like 'click this' or 'type this'

2.1 Terminology

This document contains a number of terms which may be new to people so a brief overview of these follows:

- **HDL** stands for Hardware Description Language, of which VHDL is one. Others include Verilog, System Verilog and System C. There are of course many more, but these 4 are the ones that I have come across most. Note that these are specifically used to describe digital circuits, if you want to describe analog circuits there are flavours such as Verilog-AMS(Analog and Mixed-Signal) or VHDL-AMS(Analog/Mixed-Signal Extension), I have never come across these in the wild though.
- **Entity** is a keyword in VHDL. Each item you describe in VHDL is in an *entity* and is split into two parts; the entity *declaration* and the entity *architecture*.
- **Module** is not a keyword in VHDL, but at a high level the term may be banded around as a way of describing a distinct design unit, e.g. an entity is a module of an overall design.
- **Architecture** is another keyword. The architecture of an entity is it's implementation details. An entity may have more than one architecture, where the one instantiated can be selected via a configuration.
- **Behavioural** is a type of architecture implementation in which the source code describes the behaviour of the module. This is most familiar to us as software engineers as it is a high level description of the module using abstractions such as `if else` etc. This may not be the most efficient way of doing the job, but it's relatively quick and easier to understand, even if it's not synthesisable. *Writing the flowchart*
- **RTL** is a type of architecture implementation in which the source code is fully synthesisable may go into detail about the gates in the module. *Writing the circuit diagram*
- **Structural** is a type of architecture implementation in which the source code is a collection of instantiations of other components.
- **Synthesis** is the process of turning your description into blocks of hardware. There are a number of steps to turning VHDL source code into a format actually usable by an FPGA, and often people refer to the process as compilation or synthesis.
- **PMOD**s are a type of plug in which are developed to fit onto Digilent dev boards such as the CAN Transciever shown in fig. 1. These are really easy ways to add functionality to your dev board without spending loads of money.

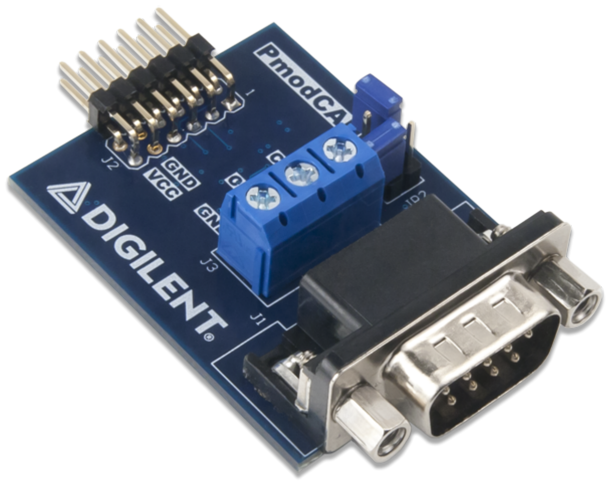

Figure 1: A CAN PMOD

Contents

List of Figures

List of Tables

3 Why Should I Care?

FPGAs enable Low Latency processing[1], so performing a transform on data coming in and getting the result output an be much faster than in a traditional CPU based approach. They also provide far more IO configurability than the traditional approach; the IO logic, and the pin it's connected to are totally configurable in code and constraints files👋. Say a requirement changes from an 8 bit UART bus to a proprietary 11 bit UART bus - this would require a whole new microcontroller in a traditional approach however with an FPGA this might only require a change to a `generic` and a recompile.

For the reasons stated above, typical uses of FPGAs include signal processing such as filtering[2], and high speed IO such as devices produced by SpeedGoat[3].

3.1 Toolchain

One takeaway from this should be that toolchains are important. Each device vendor will have their own proprietary toolchain. This means that you can approach FPGA development in one of two ways:

1. Choose a toolchain you're familiar with, then a device from the manufacturer
2. Choose a device which suits your requirements, then potentially suffer with an unfamiliar toolchain

Fortunately there are few realistic choices when it comes to this decision; use the Quartus toolchain with Intel[4] devices, or using the Vivado toolchain[5] with Xilinx devices. I am most familiar with Vivado, so this workshop is based around that. The reason for this choice is that the *Zynq* range of devices from Xilinx are a System on Chip (SOC) which allows me to use either one/two ARM cores or some Programmable Logic or any combination of these in any projects I'm undertaking. Similar devices may exist from the Intel range, but at the time of buying my dev board they didn't. For an overview of some devices and toolchains you might come across see table 1.

There is also a third option when it comes to toolchains; if you don't care about synthesis a well known simulation tool is `ghdl`. This allows for your VHDL code to be written, analysed, elaborated, and and testbenches run very quickly and without any synthesis. One of the obvious limitations with this is that it doesn't allow you to put the hardware onto a board. There are plenty of docs available online to reference when it comes to using this and this projects `build.sh` in the `scripts` directory might help as a starting point.

I have seen on twitter lots of talk[7] of the ULX3S[8], this is a low cost Lattice based dev board but unlike most other ones mentioned it's open source, including it's toolchain consisting of `yosys, nextpnr, icestrom, iverilog, symbiflow`. NB: I haven't done much looking into this so couldn't tell you what the tools do or if they are any good, that's an *exercise left for the reader*.

[1] `https://blog.esciencecenter.nl/why-use-an-fpga-instead-of-a-cpu-or-gpu-b234cd4f309c`
[2] `https://digital-library.theiet.org/content/journals/10.1049/iet-cdt.2016.0067`
[3] `https://www.speedgoat.com/products/simulink-programmable-fpgas-fpga-i-o-modules-io334`
[4] Altera
[5] Older Xilinx devices can use the ISE suit from Xilinx
[7] `https://twitter.com/ico_TC`
[8] `https://radiona.org/ulx3s/`

Manufacturer	Toolchain	Device
Intel	Quartus	Stratix Cyclone Arria ...
Xilinx	Vivado	Ultrascale Ultrascale+ Spartan-7 Virtex-7 Kintex-7 Artix-7 Zynq-7000
Xilinx	ISE	Virtex-6 Spartan-6 Coolrunner CPLD
ghdl	ghdl[6]	Simulation only

Table 1: An overview of devices and toolchains you might come across

4 Important things to remember

There is one main thing to remember through all of this: **It's not software it's hardware**. Everything you do should be done with the hardware you're creating in the back of your mind. You should make sure you are familiar with the design guidance for your device of choice. This is because different devices are made of different things - the Xilinx guidance, for example, states that *for multiplexers greater than 64:1, the tradeoffs need to be considered*[9] anything below this the device is super quick.

[9] `https://wiki.electroniciens.cnrs.fr/images/Xilinx_HDL_Coding_style.pdf`

5 Finally lets get to learning about some VHDL

5.1 How a Module design turns into VHDL

In this section is an example turning from block diagram designs, into VHDL source code itself, along with an over of the data types and keywords seen in Table 2 and Table 3.

5.1.1 Black Box

Using a counter for a module, where the output `Q` increments when the `clk` ticks, and the module is enabled as seen in fig. 2. The width of `Q` is determined by the value of `data_width`. If, for example the aim is to count up to a `max` of 10, then `data_width` would have to be *at least* 4 wide ($\lfloor log_2(10) \rfloor + 1 = 4$). This can be seen as VHDL in fig. 3.

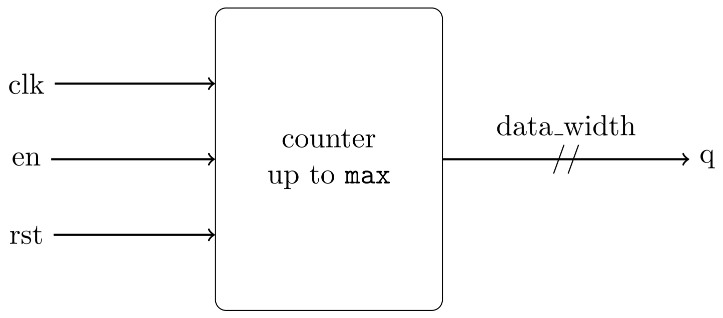

Figure 2: A Black Box Entity

```vhdl
entity counter is
    generic(
        data_width : positive;
        max        : positive
    );
    port(
        clk        : in std_ulogic;
        en         : in std_ulogic;
        rst        : in std_ulogic;
        q          : out std_ulogic_vector(data_width-1 downto 0)
    );
end entity;
```

Figure 3: A Black Box Entity in VHDL

5.1.2 White Box

Now that the inputs and outputs are designed and implemented, we can look at the insides of the module. I have chosen to use a Moore Machine[10] like in fig. 4[11] to do this. My implementation in fig. 6 should describe the logic diagram shown in fig. 5. Shown in fig. 5 is a system whereby the output `q` is based on the current state output only, and not the current state plus some

[10]`https://www.tutorialspoint.com/automata_theory/moore_and_mealy_machines.htm`
[11]`http://www-inst.eecs.berkeley.edu/~cs150/fa05/Lectures/07-SeqLogicIIIx2.pdf`

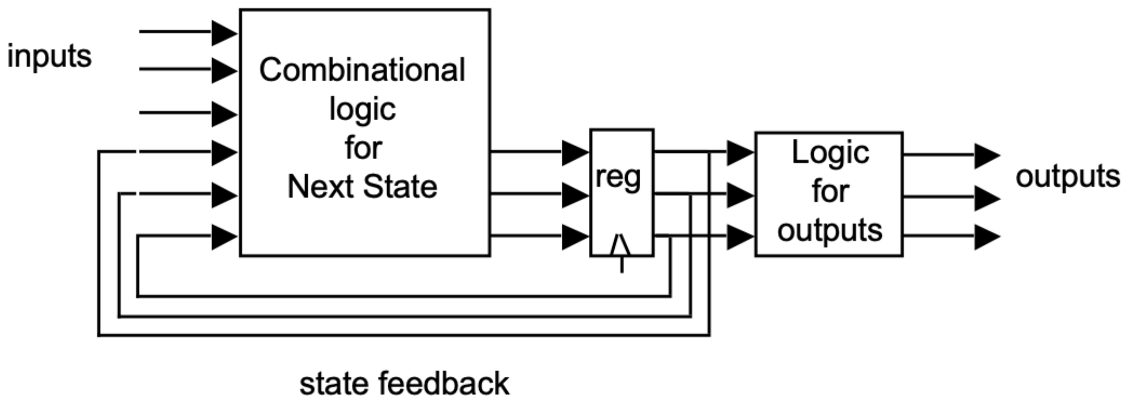

Figure 4: Generic Moore Machine

Keyword	Overview
process	The start of a process, in a process each instructions happens sequentially
signal	👋Think of it as a wire that exists between processes. They are updated at the end of the process
variable	👋Variables can only exists in a process, they update immediately unlike signals
architecture	Signifies the start of the internal details of an entity
rising_edge *	Syntactic sugar equivalent to 'if clk = '1' and clk'event'

* Not really a keyword, but important to know.

Table 2: An overview of keywords we have seen

combination of the inputs, hence it's a *Moore Machine*. There is a *combinatorial logic* section which works out the next thing to get clocked into the register, and a *synchronous* part which clocks that through and handles the advancement of the state.

When examining the source code in fig. 6 it important to remember that each process happens concurrently. Similarly any thing outside of a `process` block such as that on line 38 of the snippet also happens concurrently. If it helps, each line of code can be thought of as it's own process. To clarify that statement, and with reference to fig. 6, there are 3 concurrent things happening:

1. The `process` between lines 12-25, which is the *combinatorial logic*
2. The `process` between lines 27-36, which is the *synchronous logic*
3. The assignment of the output `q` on line 38

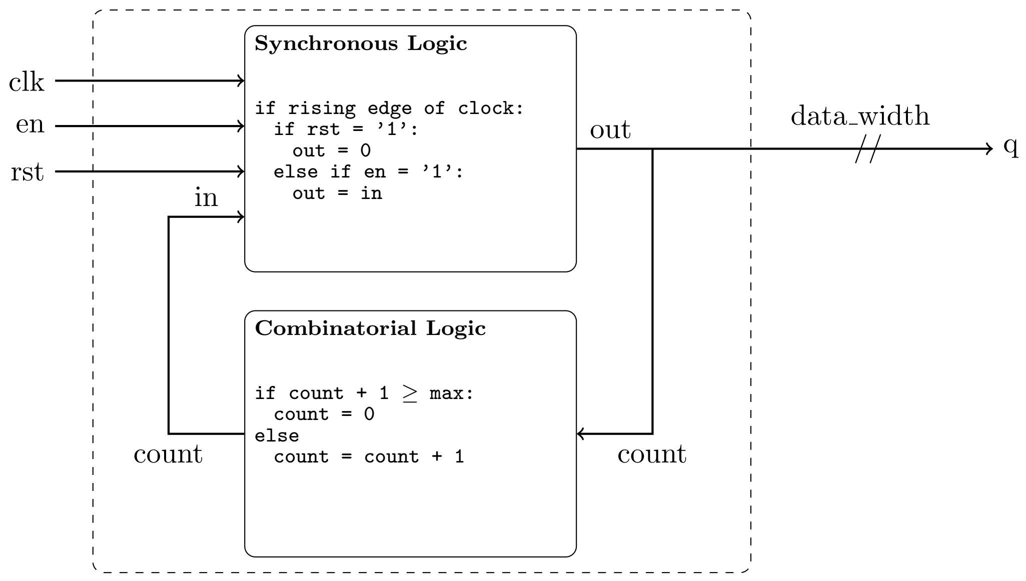

Figure 5: A White Box Entity with pseudocode Moore Machine

Data Type	Overview
std_ulogic [*]	This is defined in the ieee.std_logic_1164 package and is an enumeration with the values 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H' and '-'
std_ulogic_vector	Again defined in the ieee.std_logic_1164 package this is an array of std_ulogics
positive	This is a VHDL standard type and it's an integer with a range of 1 to at least $2^{31} - 1$
record	This is a VHDL standard type, similar to a struct

[*] Many online examples use std_logic, this is the 'resolved' version of std_ulogic. They are effectively the same except a signal of type std_logic can be controlled from different places and errors will only show themselves at runtime.

Table 3: An overview of data types we have seen

5.2 Synthesiseable vs. Non-synthesiseable code

VHDL has lots of things that can't be translated into hardware, such as the `wait` statement and files. It's important to be aware that not everything you describe can turn into hardware. If in doubt consult resources such as the *Doulus Guide* in section 7: Further Activities.

5.3 How to see output from our VHDL

There are two ways to do this

1. **Simulation** - this is often the quickest method as it allows us to delve into every signal and process of our design. This also happens on our development machine so we can make great progress and do 95% of our coding without a device. Another bonus is that it doesn't require fully synthesiseable code, so you can use `wait` statements and stuff.

```vhdl
architecture beh of counter is

    type private_register_t is record
        count: natural range 0 to max + 1;
    end record;

    signal reg_in : private_register_t := (count => 0);
    signal reg_out: private_register_t := (count => 0);

begin

    process(reg_out)
        variable v : private_register_t := (count => 0);
    begin
        v := reg_out;

        if v.count + 1 > max then
            v.count := 0;
        else
            v.count := v.count + 1;
        end if;

        reg_in <= v;

    end process;

    process(clk, rst, en)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                reg_out <= (count => 0);
            elsif en = '1' then
                reg_out <= reg_in;
            end if;
        end if;
    end process;

    q <= std_ulogic_vector(to_unsigned(reg_out.count, q'length));

end architecture;
```

Figure 6: The White Box Entity in VHDL

2. **Putting on the hardware** - this is where stuff gets tough. In this step the design will run at full speed on a device. You should only be doing this when you feel sufficiently confident that it'll work. If anything goes wrong here it can be extremely tough to debug it, or even detect it! It's very worthwhile having an oscilloscope[12] at this point to help.

5.3.1 Simulation

Simulations can be stimulated via a test bench (my personal favourite) or manually in some simulators. For this tutorial I have created a testbench to do this for both the `counter` module and the `top_model` of the design. Note that the testbenches provided don't actually do any testing, they just instantiate the modules and wait for a certain amount of time. In real life these would contain a number of `assert` and `report` statements to allow a test to automatically fail. These test benches, and the `top_model` also give an example of structural implementation, so you can see how a module is instantiated and connected up. Simulations will pop out a waveform, which can be quite complicated to interpret if the design is big and has loads of signal, but in the case of our counter it's pretty easy. Figure 7 shows the output of `gtkwave` where the `clk` input and the output `q` can be seen plotted, note how *after* the `clk` has a rising edge the value of `q` changes.

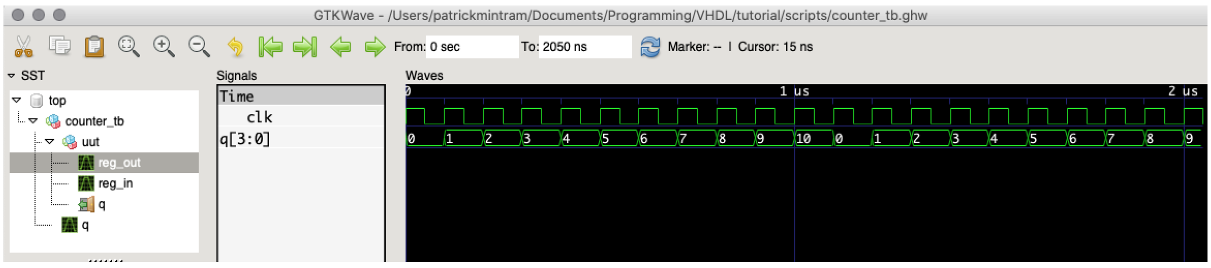

Figure 7: gtkwave output of our counter

Although the example in fig. 7 was made using `gtkwave` most waveforms generated will look the same. My advice here is to just have a play - download a simulation tool such as `ghdl` and `gtkwave`, or use the ones provided with you device's toolchain and go wild!

[12]aka, DSO, silly scopes, or just scopes

6 OK, now lets actually do some VHDL

This section is where the majority of the instruction boxes are, as such it's very specific the to Vivado, so if you're the using a different tool at home it might not be too helpful.

6.1 Reference Project Overview

The reference project provided enables you to flash some LEDs at different speeds using the counter previously discussed - this demonstrates the ability to create designs with different things happening at different times and at different rates. It also contains some stubs for you to put in your own functionality; turning off and on an LED from a switch.

6.1.1 File Types

A VHDL project will typically contain the following files:

1. Design Source Files `*.vhd`: These contain the actual source code for the hardware you're describing. These should contain only synthesisable code.
2. Design Test Benches `*_tb.vhd`: These contain test benches for your design entities, usually suffixed with `_tb` so you know it's a test bench, and will likely contain non synthesisable code.
3. Constraints Files `*.(ucf|xcd)`: These files describe any constraints of the platform or design, such as pins used, timing requirements etc. Note that these extensions are Xilinx specific, Quartus uses `.qsd`.
4. Scripts `*.(sh|tcl)`: These are provided to help with setting up and managing the project. `tcl` files seem to be what most tools prefer, and tools will often provide a number of utility functions to help with this.

6.2 How to get started with a new design

There are heaps of tutorials online for how to do this in all sorts of toolchains, such as `https://reference.digilentinc.com/vivado/getting_started/start` or `https://www.intel.com/content/www/us/en/programmable/documentation/yoq1529444104707.html` so I won't cover that here. If you're reading this and it's printed then good luck typing those in! Ask me for an electronic copy and I'll make sure you get one, alternatively google *Quartus getting started* or similar.

6.3 Open the reference project

6.4 Build the reference project

6.5 Run the simulation

6.6 Load onto the Device

6.7 Implement the architecture of the stubs

6.8 Device Overview

The Zedboard is a development board based around a Xilinx Zynq-7000 device. As previously mentioned it's got 2 ARM cores on board along with a bunch of Programmable Logic (seen in fig. 8). It's important to be aware of this because any bitstreams and designs we produce need to include an interface to one more more of these cores. Infact for the reference project they will serve as our clock source. There are heaps of things you can use which are already built in as peripherals to the PS such as GPIO, Serial Comms and PS-PL interuppt control. The fun part about FPGAs though is learning how to do this in the Programmable Logic!

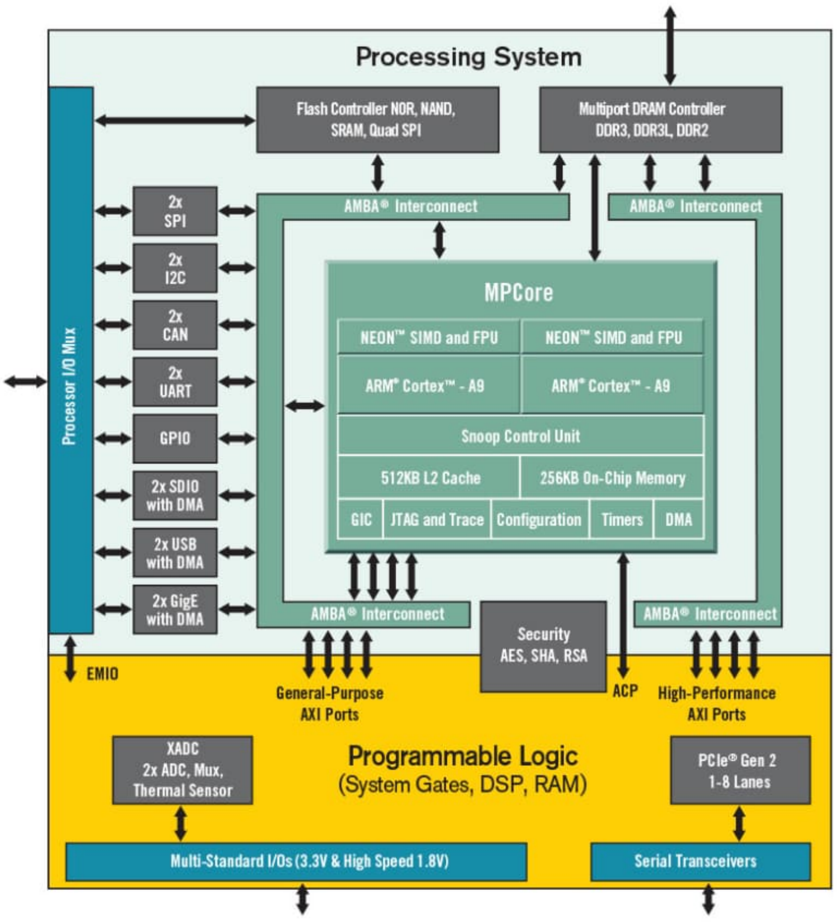

Figure 8: The Zynq device architecture

6.8.1 How to find what connects to what

This is an important thing to be able to do, and if you love datasheets like me you're in for a treat! First we need to identify which switches and LEDs on the physical hardware we want to use, for this I want to use slide switches 0 and 1, and LED 0. Although not marked in fig. 9 they are just above the slide switches.

Find the switches and LEDs on the actual Zedboard

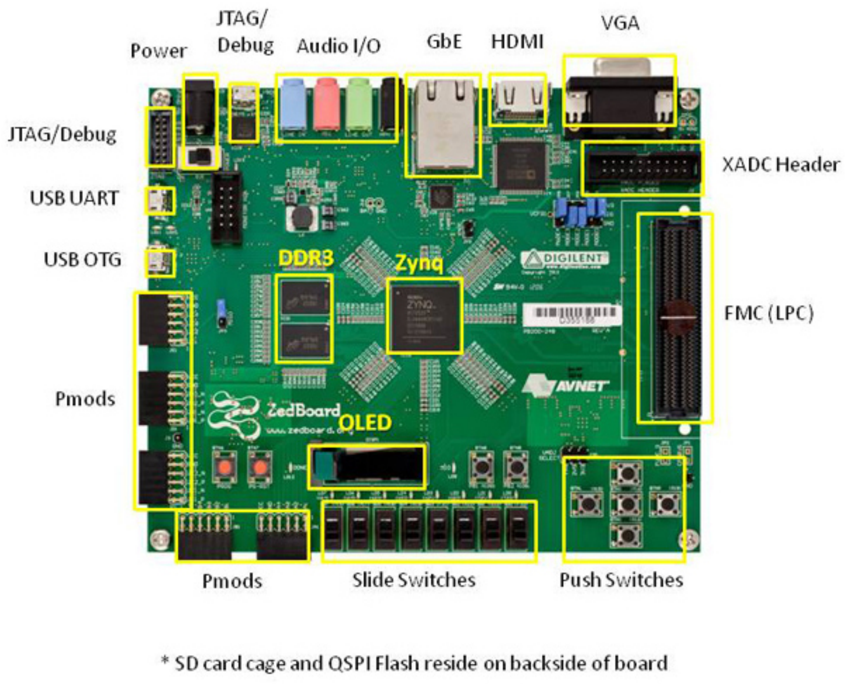

Figure 9: The Zedbaord Layout

The next step is to reference the schematic[13].

Find `SW0`, `SW1` and `LD0`. This can be done with a `ctrl-f` usually. These can both be found on sheet 3 of the schematic, and are shown in fig. 10 and fig. 11.

[13]Available on `https://www.xilinx.com`

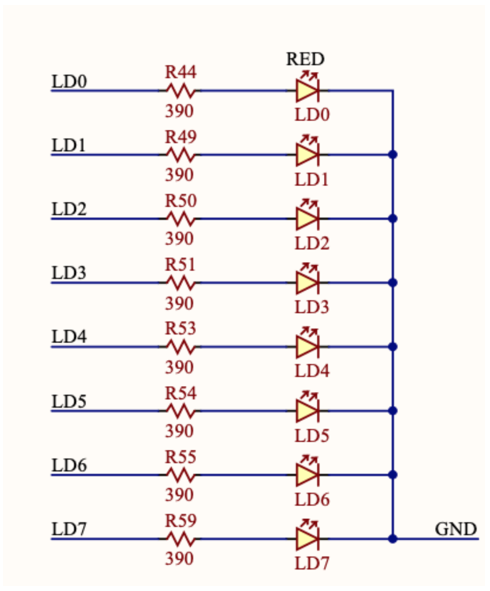

Figure 10: The Zedboard LEDs on the schematic

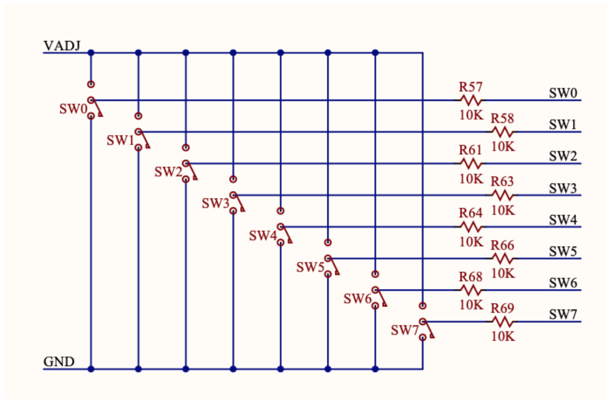

Figure 11: The Zedboard switches on the schematic

On the schematic, how we have found the physical components we need to find where they connect to the FPGA BGA[14]. This can be done by finding the other end of the `SW0` net shown on the right hand side of fig. 11. It should take you to a sheet 9, and in the bottom right hand corner are the switches. Shown in fig. 12.

Identify the BGA grid location of the connection to the FPGA

[14]Ball Grid Array - the pads attached to the FPGA IC

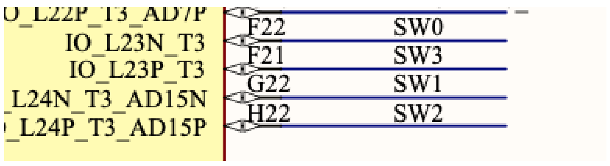

Figure 12: The Zedboard switches connect to the FPGA at BGA location F22 and G22

7 Further Activities

If this has been fun, and you want to learn more here are some useful resources I have had successes with:

- Effective Coding with VHDL[15]
- Awesome VHDL[16]
- Digital Fundamentals[17]
- Doulos Guide[18]

[15]`https://www.amazon.co.uk/Effective-Coding-VHDL-Principles-Practice/dp/0262034220`
[16]`https://github.com/VHDL/awesome-vhdl`
[17]`https://www.amazon.co.uk/Digital-Fundamentals-Thomas-L-Floyd/dp/0132737965`
[18]`https://www.ics.uci.edu/~jmoorkan/vhdlref/vhdl_golden_reference_guide.pdf`