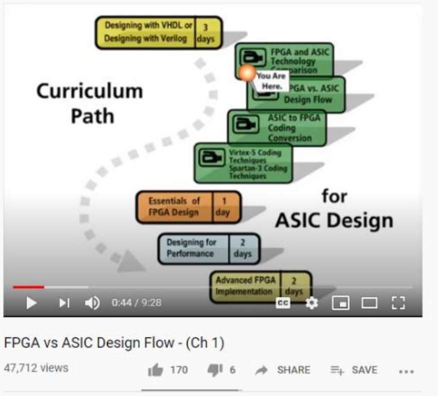
DAILY ASSESSMENT

Date:	01 JUN 2020	Name:	PAVITHRAN S
Course:	DIGITAL DESIGN USING HDL	USN:	4AL17EC068
Topic:	• Industry Applications of FPGA • FPGA Business Fundamentals • FPGA vs ASIC Design Flow • FPGA Basics – A Look Under the Hood	Semester & Section:	6^{TH} B
Github Repository:	Pavithran		

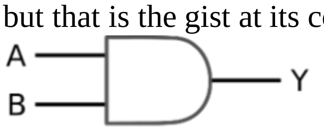
FORENOON SESSION DETAILS

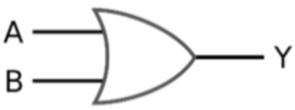


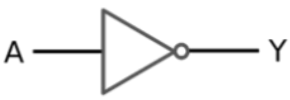
Report Hand written or typed format:

What is an FPGA?

An FPGA is a (mostly) digital, (re-)configurable ASIC. I say mostly because there are analog and mixed-signal aspects to modern FPGAs. For example, some have A/D converters and PLLs. I put *re-* in parenthesis because there are actually one-time-programmable FPGAs, where once you configure them, that's it, never again. However, most FPGAs you'll come across are going to be re-configurable. So what do I mean by digitally configurable ASIC? I mean that at the core of it, you're designing a digital logic circuit, as in AND, OR, NOT, flip-flops, etc. Of course that's not entirely accurate and there's much more to it than that, but that is the gist at its core.







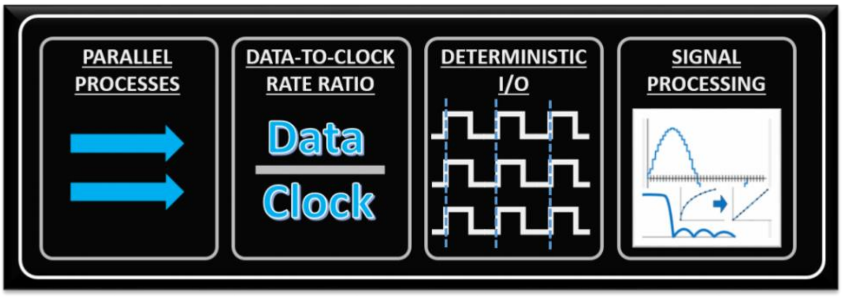
The players –

There are currently two big boys: Altera (part of Intel) and Xilinx, and some supporting players (e.g. Actel (owned by Microsemi)).

The main underlying technology options are SRAM-based (this is the most common technology), flash, and anti-fuse. As you might imagine, each option has its own pros and cons. Check this out for some more details.

Strengths / best suited for:

Much of what will make it worthwhile to utilize an FPGA comes down to the low-level functions being performed within the device. There are four processing/algorithm attributes defined below that FPGAs are generally well-suited for. While just one of these needs may drive you toward an FPGA, the more of these your application has, the more an FPGA-based solution will appeal.



Parallel processes – if you need to process several input channels of information (e.g. many simultaneous A/D channels) or control several channels at once (e.g. several PID loops).

High data-to-clock-rate-ratio – if you've got lots of calculations that need to be executed over and over and over again, essentially continuously. The advantage is that you're not tying up a centralized processor. Each function can operate on its own.

Large quantities of deterministic I/O – the amount of determinism that you can achieve with an FPGA will usually far surpass that of a typical sequential processor. If there are too many operations within your required loop rate on a sequential processor, you may not even have enough time to close the loop to update all of the I/O within the allotted time.

Signal processing – includes algorithms such as digital filtering, demodulation, detection algorithms, frequency domain processing, image processing, or control algorithms.

Weaknesses / not optimal for:

With any significant benefit, there's often times a corresponding cost. In the case of FPGAs, the following are generally the main disadvantages of FPGA-based solutions.

Complex calculations infrequently – If the majority of your algorithms only need to make a computation less than 1% of the time, you've generally still allocated those logic resources for a particular function (there are exceptions to this), so they're still sitting there on your FPGA, not doing anything useful for a significant amount of time.

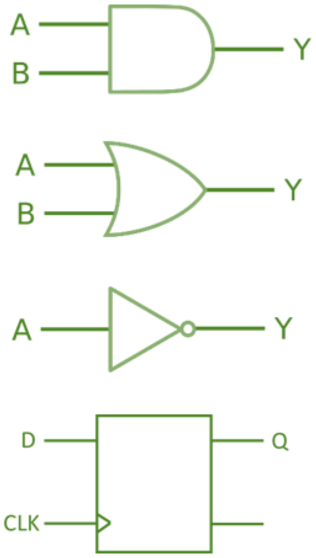
Sorting/searching – this really falls into the category of a sequential process. There are algorithms that attempt to reduce the number of computations involved, but in general, this is a sequential process that doesn't easily lend itself to efficient use of parallel logical resources. Check out the sorting section here and check out this article here for some more info.

Floating point arithmetic – historically, the basic arithmetic elements within an FPGA have been fixed-point binary elements at their core. In some cases, floating point math can be achieved (see Xilinx FP Operator and Altera FP White Paper), but it will chew up a lot of logical resources. Be mindful of single-precision vs double-precision, as well as deviations from standards. However, this FPGA weakness appears to be starting to fade, as hardened floating-point DSP blocks are starting to be embedded within some FPGAs (see Altera Arria 10 Hard Floating Point DSP Block).

Very low power – Some FPGAs have low power modes (hibernate and/or suspend) to help reduce current consumption, and some may require external mode control ICs to get the most out of this. Check out an example low power mode FPGA here. There are both static and dynamic aspects to power consumption. Check out these power estimation spreadsheets to start to get a sense of power utilization under various conditions. However, if low power is critical, you can generally do better power-wise with low-power architected microprocessors or microcontrollers.

Very low cost – while FPGA costs have come down drastically over the last decade or so, they are still generally more expensive than sequential processors.

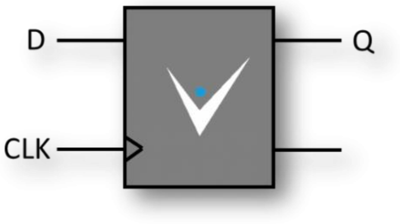
How Does an FPGA work?



You're designing a digital circuit more than anything else, basically at one layer of abstraction above the logic gate (AND, OR, NOT) level. At the most basic level, you need to think about how you're specifying the layout and equations at the level of LUTs (Look-Up Tables) and FFs (Flip-Flops).

$Y = f(A,B)$

A	B	Y
0	0	Y_1
0	1	Y_2
1	0	Y_3
1	1	Y_4



Otherwise you're circuit can get very large and slow very quickly. You've got a very detailed level of control at your fingertips, which is very powerful, but can be overwhelming, so start slow. You'll be determining the # of bits, and exact math / structure of each function.

An FPGA is a synchronous device, meaning that logical operations are performed on a clock cycle-by-cycle basis. Flip-flops are the core element to enabling this structure.

In general, you're going to put digital data into an FPGA and get digital data out of it through various low-voltage digital I/O lines, sometimes many bits in parallel (maybe through one or more A/D converter outputs or an external DRAM chip), sometimes through high-speed serial I/O (maybe connecting to an Ethernet PHY or USB chip).

What's inside – Core components (or at least what everyone likes to think about):

LUT (Look-Up Table) –

The name LUT in the context of FPGAs is actually misleading, as it doesn't convey the full power of this logical resource. The obvious use of a LUT is as a logic lookup table (see examples here and here), generally with 4 to 6 inputs and 1 to 2 outputs to specify any logical operation that fits within those bounds. There are however two other common uses for a LUT:

LUT as a shift register – shift registers are very useful for things like delaying the timing of an operation to align the outputs of one algorithm with another. Size varies based on FPGA.

LUT as a small memory – you can configure the LUT logic as a VERY small volatile random-access memory block. Size varies based on FPGA

FF (Flip-flop) –

Flip-flops store the output of a combinational logic calculation. This is a critical element in FPGA design because you can only allow so much asynchronous logic and routing to occur before it is registered by a synchronous resource (the flip-flop), otherwise the FPGA won't make timing. It's the core of how an FPGA works.

Flip-flops can be used to register data every clock cycle, latch data, gate off data, or enable signals.

Block Memory –

It's important to note that there are generally several types of memory in an FPGA. We mentioned the configuration of a LUT resource. Another is essentially program memory, which is intended to store the compiled version of the FPGA program itself (this may be part of the FPGA chip or as a separate non-volatile memory chip). What we're referring to here though, is neither of those types of memory. Here we're referring to dedicated blocks of volatile user memory within the FPGA. This memory block is generally on the order of thousands of bits of memory, is configurable in width and depth, and multiple blocks of memory can be chained together to create larger memory elements. They can generally be configured as either single-port or dual-port random access, or as a FIFO. There will generally be many block memory elements within an FPGA.

Multipliers or DSP blocks –

Have you ever seen the number of digital logic resources that it takes to create a 16-bit by 16-bit multiplier? It's pretty crazy, and would chew through your logical and routing resources pretty quickly. Check out the 2-bit by 2-bit example here: https://en.wikipedia.org/wiki/Binary_multiplier. FPGA vendors solve this problem with dedicated silicon to lay down something on the order of 18-bit multiplier blocks. Some architectures have recognized the utility of digital signal processing taking place, and have taken it a step further with dedicated DSP (Digital Signal Processing) blocks, which can not only multiply, but add and accumulate as well.

I/O (Input/Output) –

If you're going to do something useful with an FPGA, you generally have to get data from and/or provide data outside the FPGA. To facilitate this, FPGAs will include I/O blocks that allow for various voltage standards (e.g. LVCMOS, LVDS) as well as timing delay elements to help align multiple signals with one another (e.g. for a parallel bus to an external RAM chip).

Clocking and routing –

This is really a more advanced topic, but critical enough to at least introduce. You'll likely use an external oscillator and feed it into clocking resources that can multiply, divide, and provide phase-shifted versions of your clock to various parts of the FPGA

Task for Day-1

Write a verilog code to implement NAND gate in all different styles.

Gate level modeling style

```
module NAND_2_gate_level(output Y, input A, B);
   wire Yd;
   and (Yd, A, B);
   not (Y, Yd);
endmodule
```

Dataflow modeling style

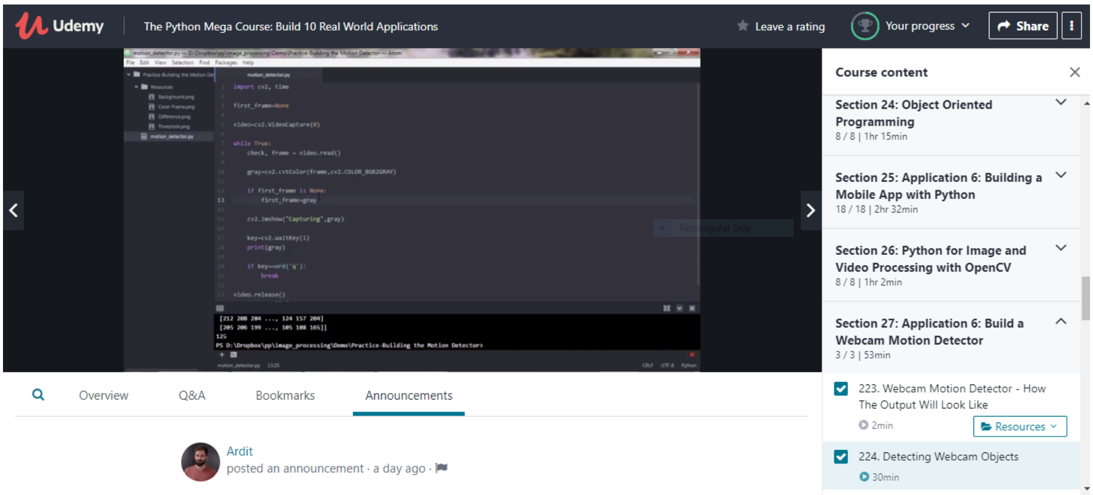
```
module NAND_2_data_flow(output Y, input A, B);
    assign Y = ~(A & B);
endmodule
```

Behavioral modeling style

```
module NAND_2_data_flow(output Y, input A, B);
always @ (A or B) begin
     if (A == 1'b1 &  B== 1'b1) begin
           Y = 1'b0;
     end
     else
          Y = 1'b1;
end
endmodule
```

DAILY ASSESSMENT

Date:	**01 JUN 2020**	**Name:**	**PAVITHRAN S**
Course:	**PYTHON**	**USN:**	**4AL17EC068**
Topic:	• **Application 6: Build a Webcam Motion Detector**	**Semester & Section:**	**6^{TH} B**
Github Repository:	**Pavithran**		

AFTERNOON SESSION DETAILS


Report can be hand witten or typed format:
Motion detection is the detection of the change in the position of an object with respect to its surroundings and vice-versa. Buckle up your seat belts to drive through this motion detector application along with me and your lovable Python.You may be able to perform the following tasks using this application, though the list is non-exhaustive: 1) Find in front of screen time during working from home. 2)Monitor your child's in front of screen time. 3) Find trespassing in your backyard. 4)Locate unwanted public/animal movements around yoroom/house/alley and what not.......Photo by

Williapm Thomas on Unsplash Hardware Requirements: A computer with a webcam or any type of camera installed.

Software Requirements: Python 3 or above.Additional Requirements: 30 mins of your time, Enthusiasm about the topic I will guide you step by step into building the application. Firstly, you will capture the first frame via webcam. This frame will be treated as the baseline frame. Motion will be detected by calculating the phase difference between this baseline frame and the new frame with some object. The new frames will be called Delta frame.Then you will refine your delta frame using pixel intensity. The refined frame will be called the Threshold frame. Then you will apply some intricate image processing techniques like Shadow Removal, Dilation, Contouring, etc. on the Threshold frame to capture substantial objects. Detected Object You will be able to capture the time stamp when an object entered the frame and exited the frame. Thus, you will be able to find the screen-on time.I won't embed my code here as I would like you to improve the blood circulation on your fingertips.To start with basic installations, please install python 3 or above, pandas, and opencv via pip. Once done, you are ready to begin:

STEP 1: Import required libraries:

STEP 2: Initialize variables, lists, data frames:You will get to know when each one of the above will be required in the below code.

STEP 3: Capture the video frames using webcam:OpenCV has in-built functions to open the camera and capture video frames. “0” denotes the camera at the hardware port number 0 in your computer. If you have multiple cameras or external cameras or a CCTV setup installed, you may provide the port number accordingly.

STEP 4: Converting the captured frame to gray-scale and applying Gaussian Blur to remove noise:We convert the color frame to gray frame as an extra layer of color is not required. GaussianBlur is used for image smoothing and it will, in turn, enhance the detection accuracy. In the GaussianBlur function, for the 2nd parameter, we define the width and height of the Gaussian Kernel and for the 3rd parameter, we provide standard deviation value. These are set of higher order differential calculus theorems, so you may use standard values of the kernel size as (21,21) and standard-deviation as 0.