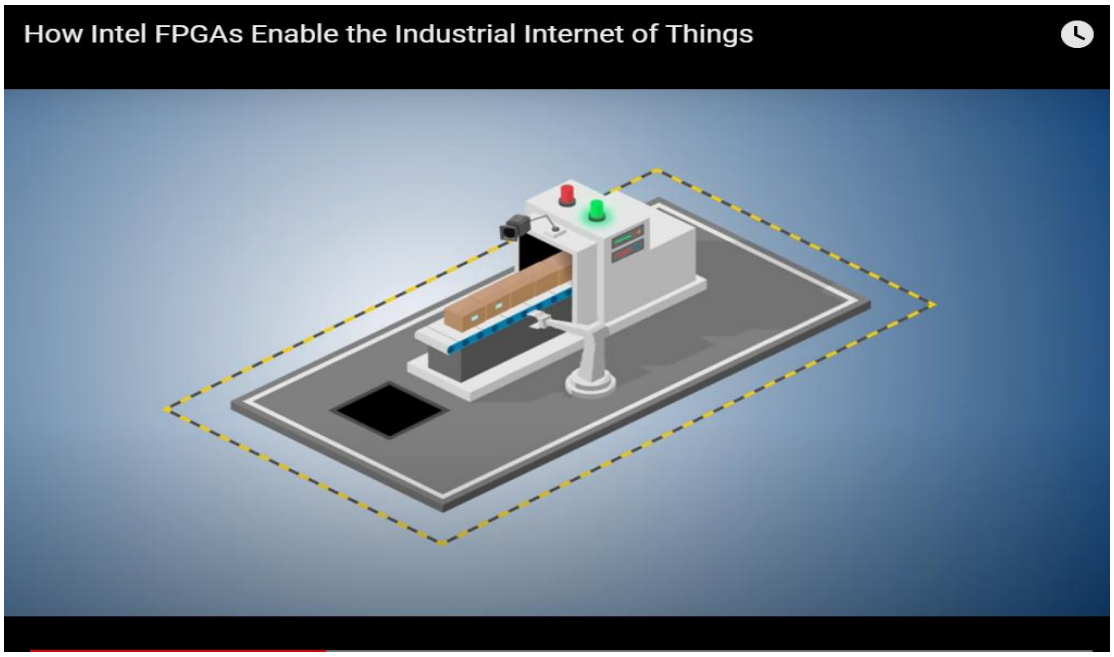
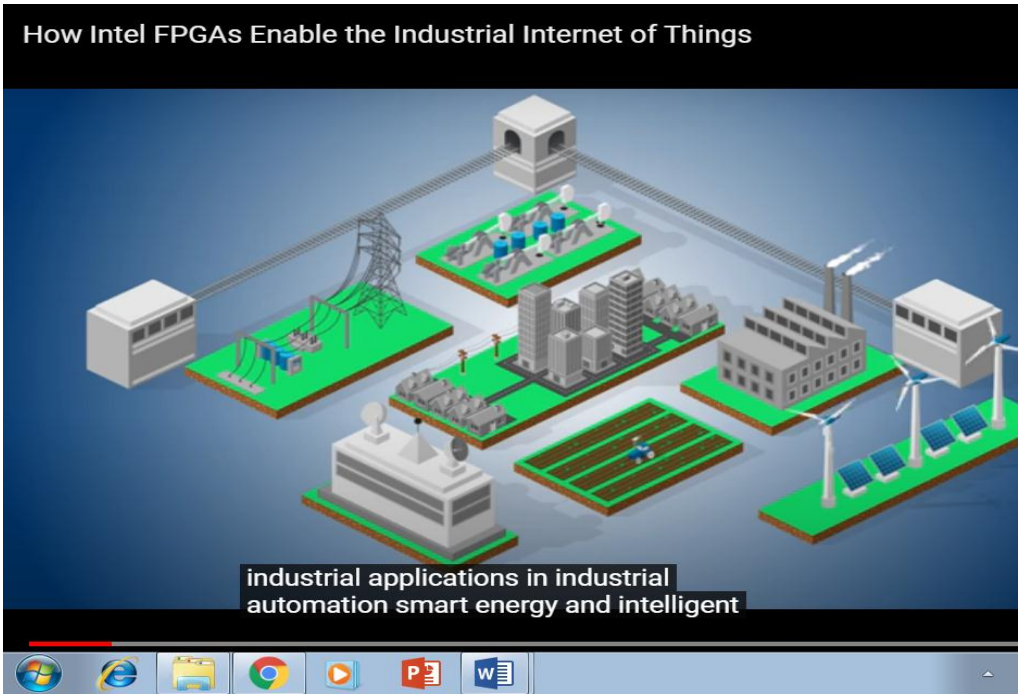
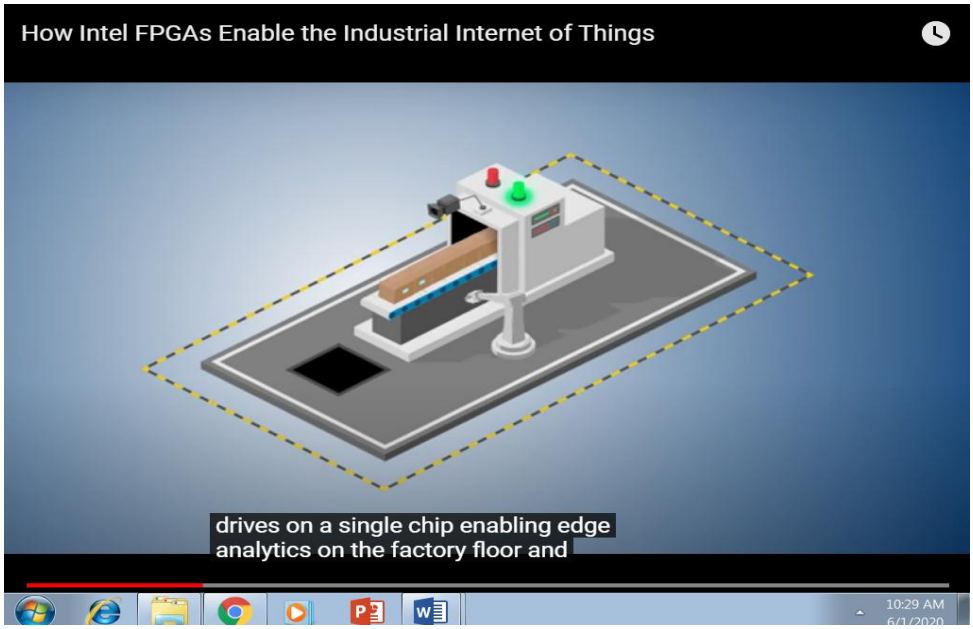
Date:	**01/6/2020**	**Name:**	**SAFIYA BANU**
Course:	**DIGITAL DESIGN USING HDL**	**USN:**	**4AL16EC061**
Topic:	Industry Applications of FPGA. FPGA Business Fundamentals. FPGA vs ASIC Design Flow. FPGA Basics – A Look Under the Hood	**Semester & Section:**	**8th , B**
Github Repository:	**Safiya-Courses**		

REPORT

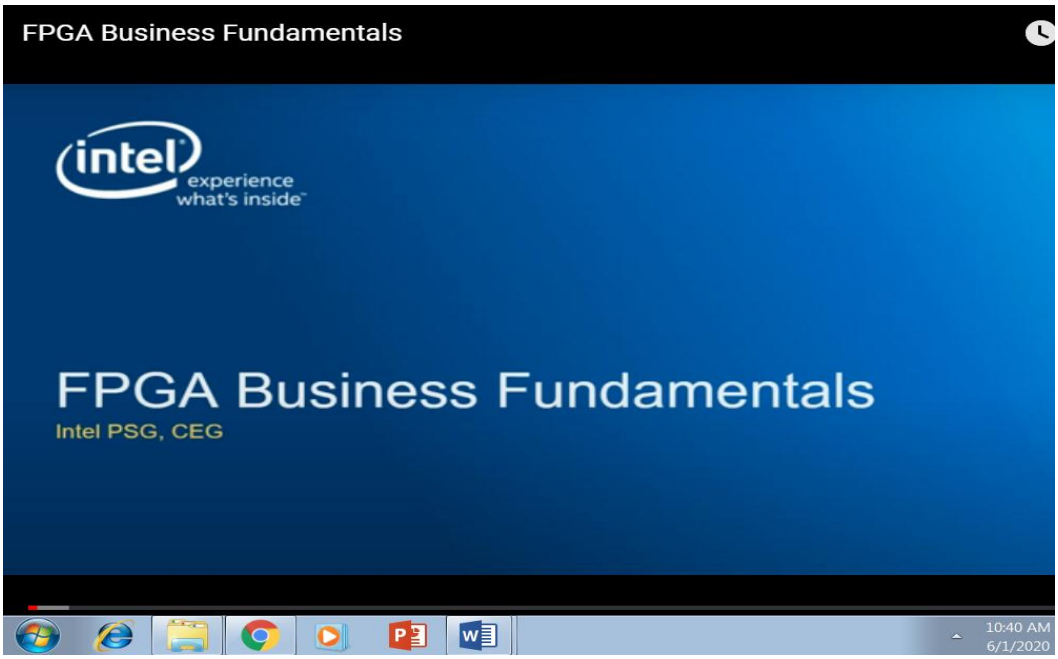
Industry Applications of FPGA.



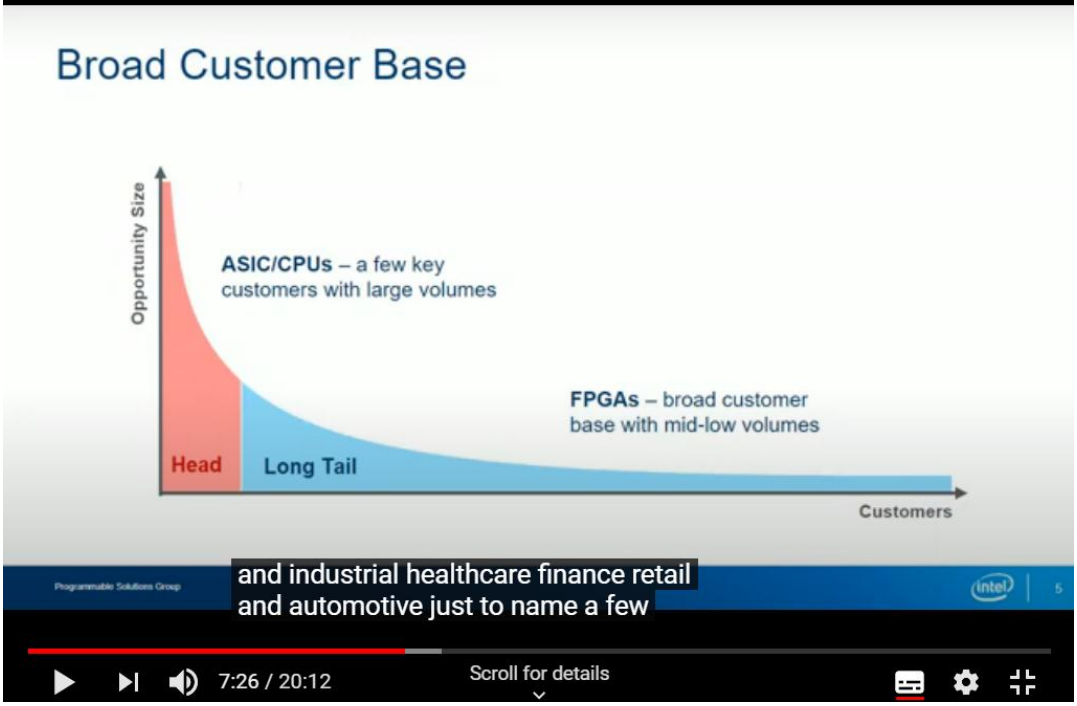

How Intel FPGAs Enable the Industrial Internet of Things
industrial applications in industrial
automation smart energy and intelligent


How Intel FPGAs Enable the Industrial Internet of Things
drives on a single chip enabling edge
analytics on the factory floor and
10:29 AM

FPGA Business Fundamentals.



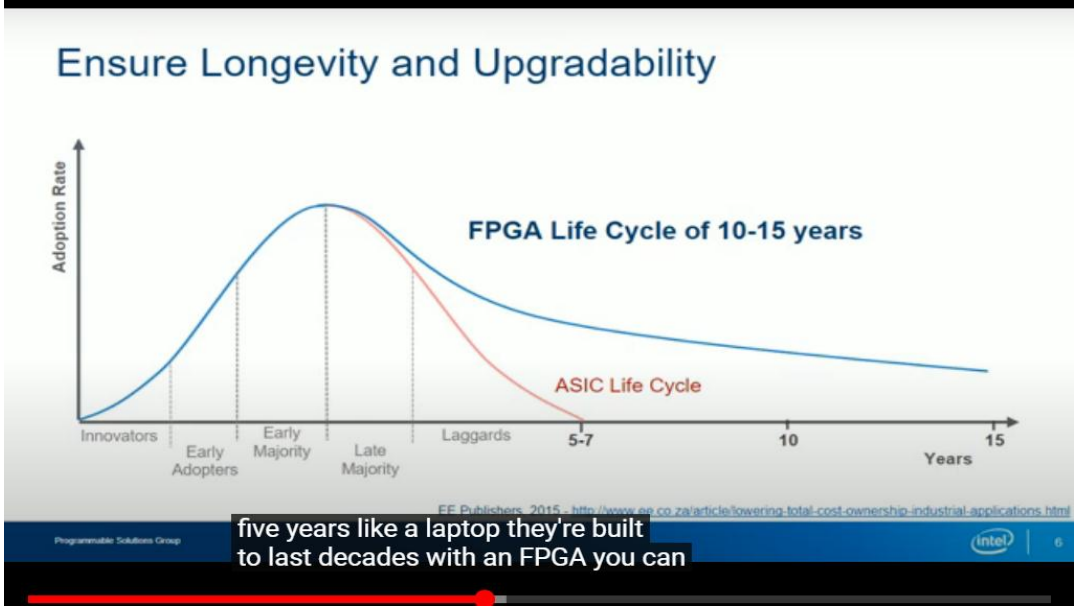
FPGA Business Fundamentals



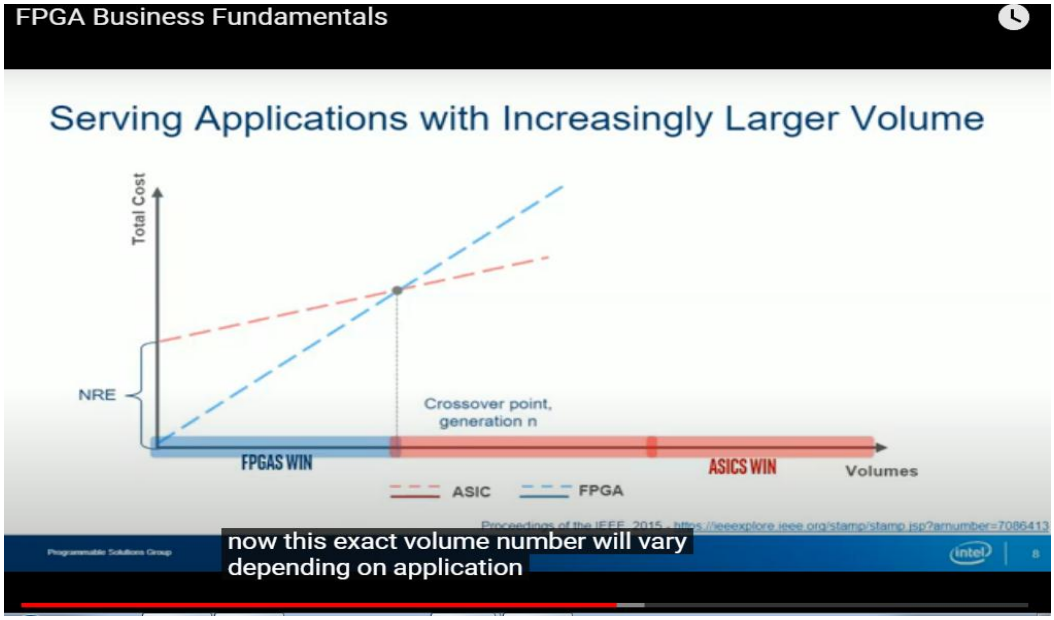
and industrial healthcare finance retail
and automotive just to name a few

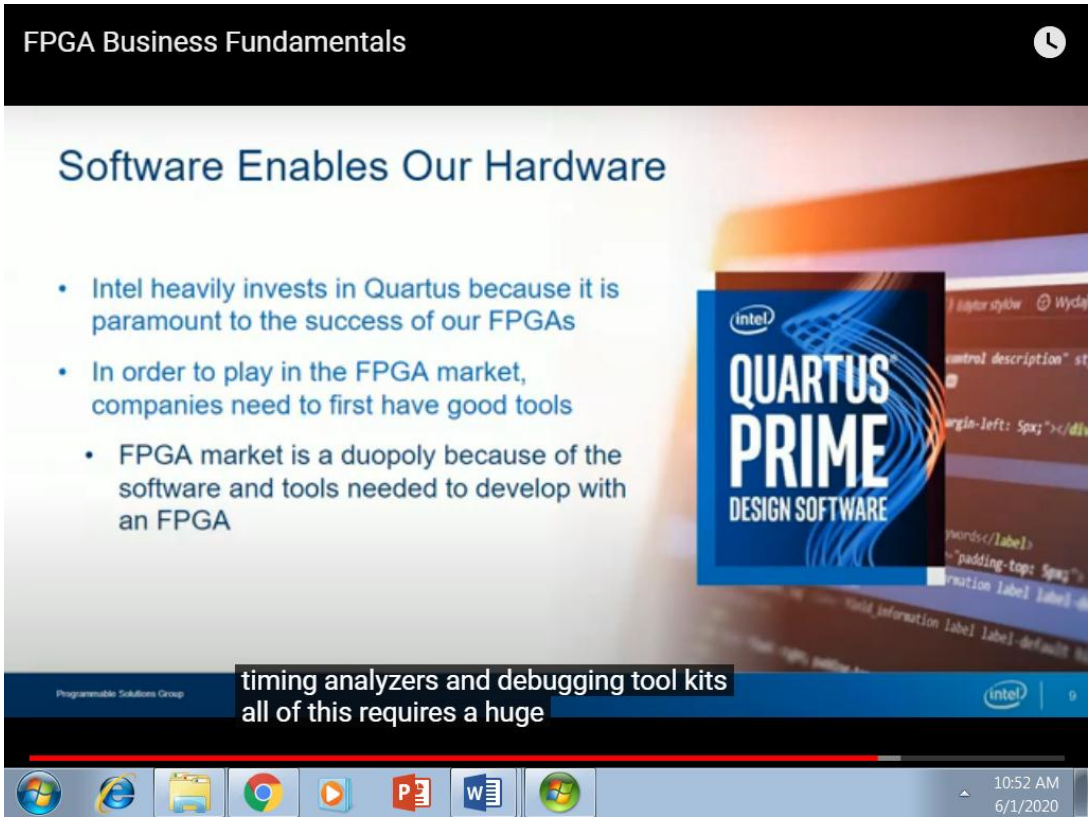
7:26 / 20:12 Scroll for details

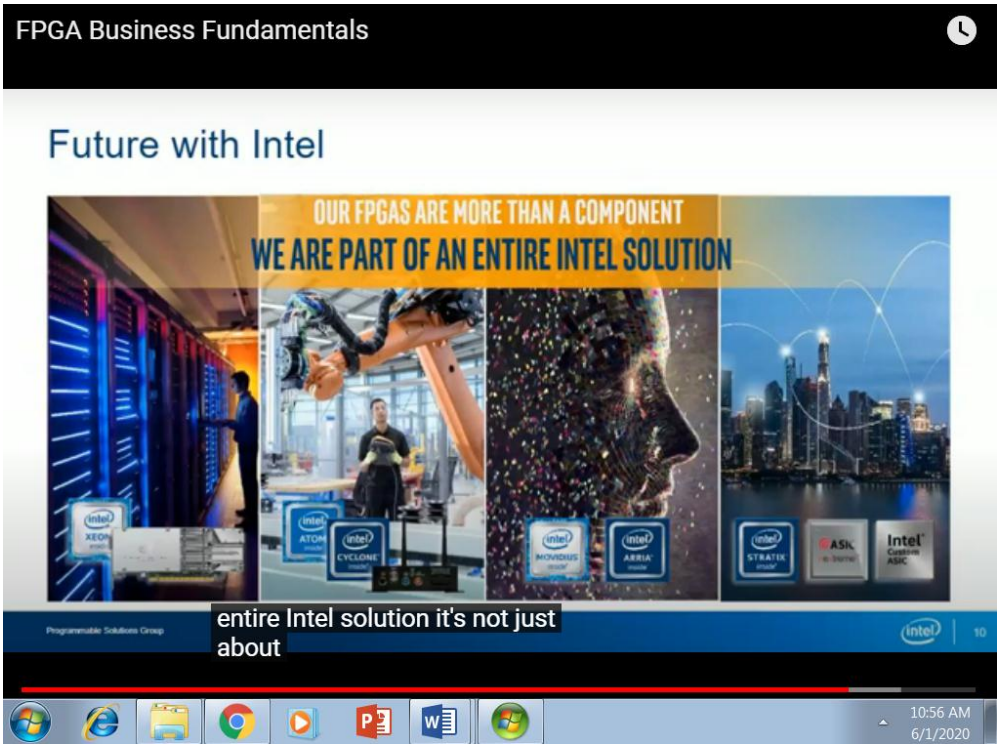
FPGA Business Fundamentals

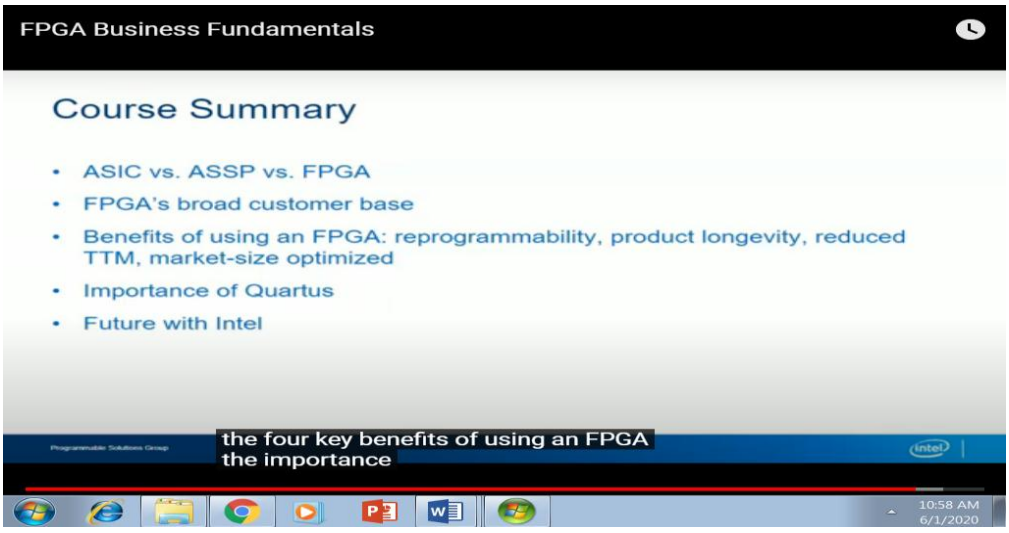


five years like a laptop they're built
to last decades with an FPGA you can

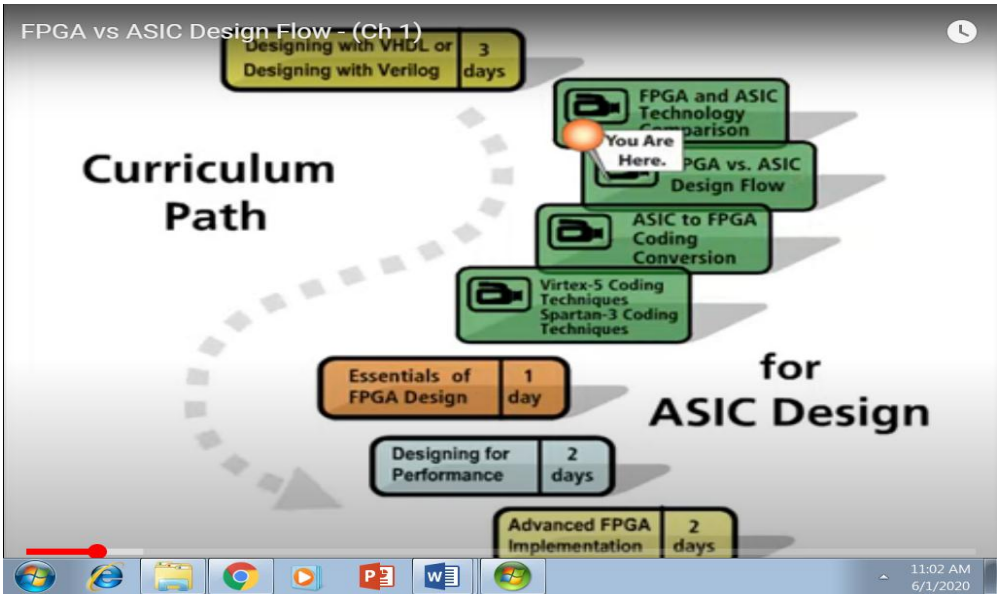

FPGA Business Fundamentals
Serving Applications with Increasingly Larger Volume
Total Cost
NRE
Crossover point, generation n
FPGAS WIN
ASICS WIN
Volumes
ASIC
FPGA
now this exact volume number will vary depending on application

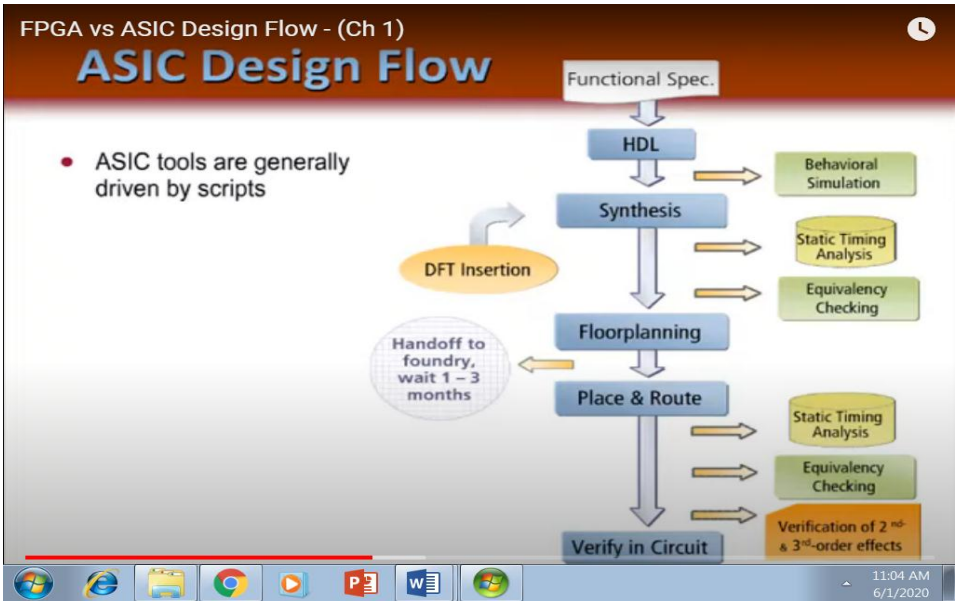

FPGA Business Fundamentals
Software Enables Our Hardware
• Intel heavily invests in Quartus because it is paramount to the success of our FPGAs
• In order to play in the FPGA market, companies need to first have good tools
• FPGA market is a duopoly because of the software and tools needed to develop with an FPGA
QUARTUS PRIME DESIGN SOFTWARE
timing analyzers and debugging tool kits all of this requires a huge
10:52 AM
6/1/2020

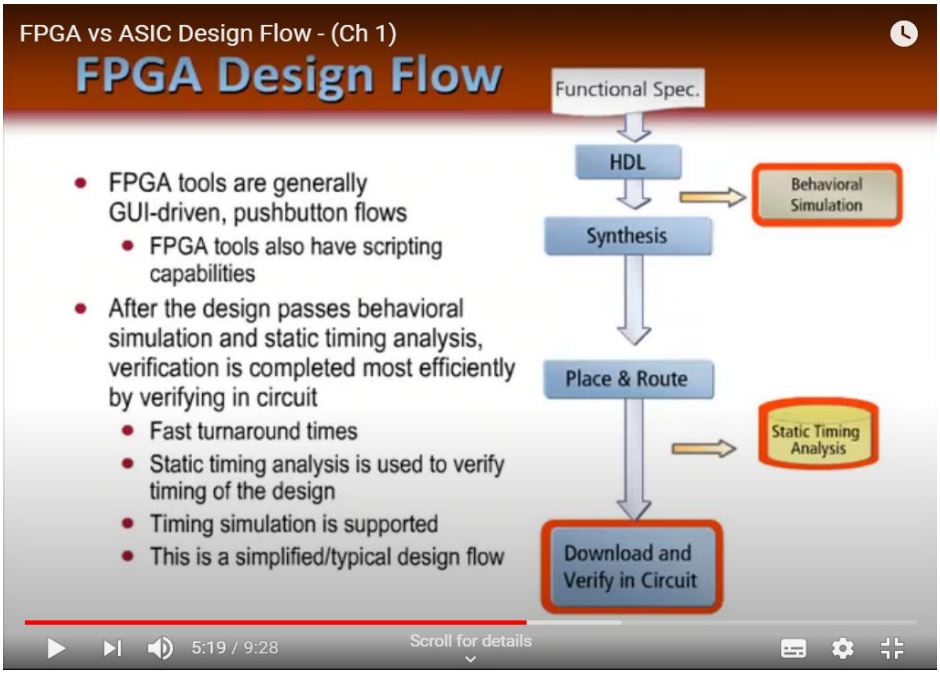


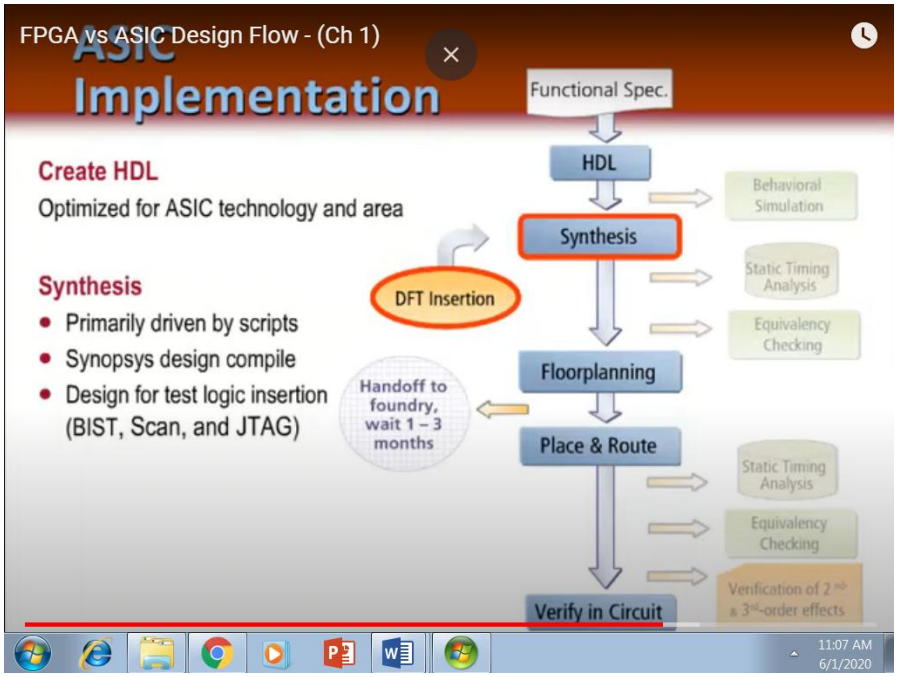


FPGA vs ASIC Design Flow.


FPGA vs ASIC Design Flow - (Ch 1)
Designing with VHDL or Designing with Verilog
3 days
FPGA and ASIC Technology Comparison
You Are Here.
FPGA vs. ASIC Design Flow
ASIC to FPGA Coding Conversion
Virtex-5 Coding Techniques
Spartan-3 Coding Techniques
Curriculum Path
Essentials of FPGA Design
1 day
Designing for Performance
2 days
Advanced FPGA Implementation
2 days
for ASIC Design
11:02 AM
6/1/2020


FPGA vs ASIC Design Flow - (Ch 1)
ASIC Design Flow
ASIC tools are generally driven by scripts
Functional Spec.
HDL
Behavioral Simulation
Synthesis
Static Timing Analysis
DFT Insertion
Equivalency Checking
Floorplanning
Handoff to foundry, wait 1 – 3 months
Place & Route
Static Timing Analysis
Equivalency Checking
Verification of 2nd & 3rd-order effects
Verify in Circuit
11:04 AM
6/1/2020


FPGA vs ASIC Design Flow - (Ch 1)
FPGA Design Flow
Functional Spec.
HDL
Behavioral Simulation
Synthesis
Place & Route
Static Timing Analysis
Download and Verify in Circuit
FPGA tools are generally GUI-driven, pushbutton flows
FPGA tools also have scripting capabilities
After the design passes behavioral simulation and static timing analysis, verification is completed most efficiently by verifying in circuit
Fast turnaround times
Static timing analysis is used to verify timing of the design
Timing simulation is supported
This is a simplified/typical design flow
5:19 / 9:28
Scroll for details

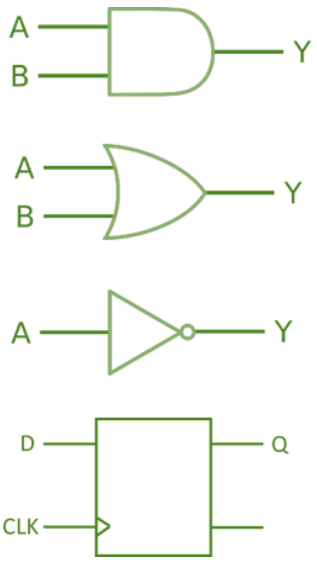

FPGA vs ASIC Design Flow - (Ch 1)
ASIC Implementation
Functional Spec.
HDL
Behavioral Simulation
Synthesis
Static Timing Analysis
Equivalency Checking
DFT Insertion
Floorplanning
Handoff to foundry, wait 1 – 3 months
Place & Route
Static Timing Analysis
Equivalency Checking
Verification of 2nd & 3rd-order effects
Verify in Circuit
Create HDL
Optimized for ASIC technology and area
Synthesis
Primarily driven by scripts
Synopsys design compile
Design for test logic insertion (BIST, Scan, and JTAG)
11:07 AM
6/1/2020

FPGA Basics – A Look Under the Hood

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.

How Does an FPGA work?



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).

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, 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 –

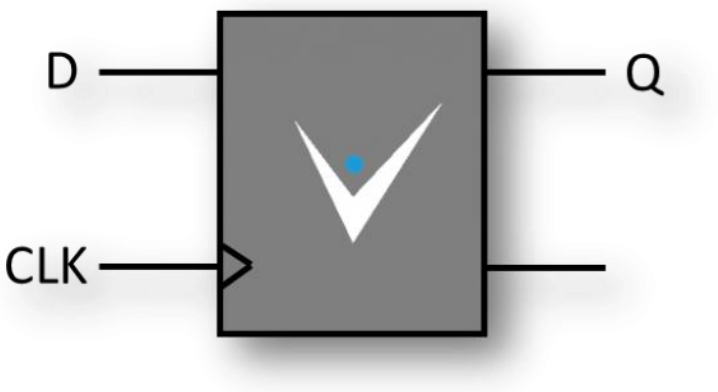
FPGA vendors solve this problem with dedicated silicon to lay down something on the order of 18-bit multiplier blocks. Some architectures have recognized the utility of digital signal processing taking place, and have taken it a step further with dedicated DSP (Digital Signal Processing) blocks, which 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).

Y = f(A,B)

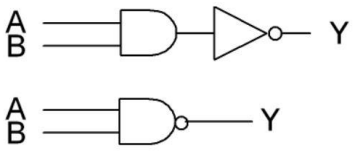
A	B	Y
0	0	Y_1
0	1	Y_2
1	0	Y_3
1	1	Y_4



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.

TASK

Verilog code for NAND gate – All modeling styles



GATE LEVEL MODELLING

```
module NAND_2_gate_level (output Y, input A, B);
   wire Yd;
    and (Yd,A,B);
```

```
not(Y,Yd);
endmodule
```

DATA FLOW MODELLING

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

BEHAVIOURAL MODELLING

```
module NAND_2_behavioural(output reg 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
```

Testbench of the NAND gate using Verilog

```
`include "NAND_2_behavioral.v"
module NAND_2_behavioral_tb;
reg A, B;
wire Y;
NAND_2_behavioral Indtance0 (Y, A, B);
initial begin
    A = 0; B = 0;
 #1 A = 0; B = 1;
 #1 A = 1; B = 0;
 #1 A = 1; B = 1;
end
initial begin
```

```
    $monitor ("%t | A = %d| B = %d| Y = %d", $time, A, B, Y);
    $dumpfile("dump.vcd");
    $dumpvars();
end
endmodule
```