EEE3096S: Embedded Systems II

LECTURE 23:

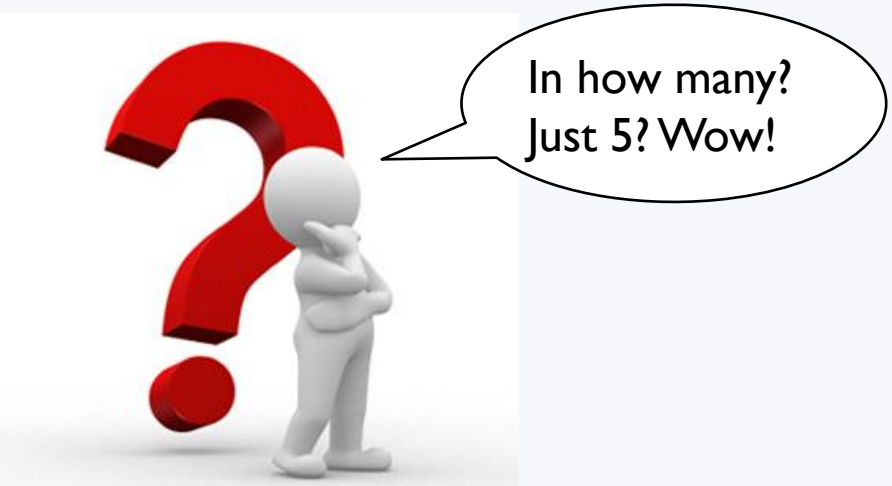
A TASTE OF VERILOG

Presented by:

STANLEY MBEWE

Electrical Engineering
University of Cape Town

Hardware Description Languages:
a taste in 5-lectures

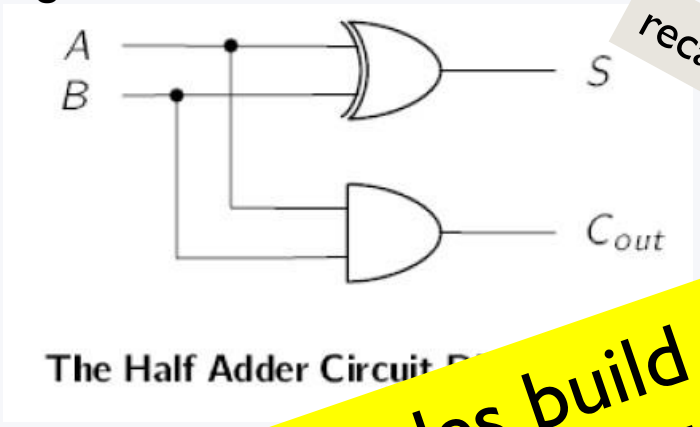


Yes, that is the plan! It is just a 'taste!'…

- To learn the essential syntax
- To understand simple HDL code
- To know what HDL is used for

In order to become more expert in HDL consider taking "EEE4120F High Performance Digital Embedded Systems" next year.

Digital Circuit



The Half Adder Circuit

recap

These slides build on knowledge of digital logic and circuits (prereq.s)

this week

Verilog


```
module half_adder (A,B,S,Cout);
  input  A, B;
  output S, Cout;

  assign o_sum  = A^B; /* bitwise xor */
  assign o_carry= A&B; /* bitwise and */
endmodule /* half_adder */
```

By assuming you have recapped and understood much of your **digital logic circuits** and the building blocks for these, then VHDL or Verilog HDL is *not* actually such a difficult thing, it is a language for describing pretty much the same circuits that you should be fairly used to by now. What's more, and why we are using Verilog, is that it is easier that VHDL / AHDL and because it has a syntax a lot like C / Python (Verilog95 syntax is probably *simpler* than C actually).

Handouts:

Please look on Amathuba
Module 6a/Notes

EEE3096F Verilog Cheat sheet
(for C programmers)

Numbers and constants
Example: 4-bit constant 10 in binary, hex and in decimal: 4'b1010 == 4'ha == 4'd10
(numbers are unsigned by default)
Concatenation of bits using {}:
4'b1011 == {2'b10 , 2'b11}
Constants are declared using parameter:
parameter myparam = 51

Operators
Arithmetic: and (+), subtract (-), multiply (*), divide (/) and modulus (%) all provided.
Shift: left (<<), shift right (>>)
Relational ops: equal (==), not-equal (!=), less-than (<), less-than or equal (<=), greater-than (>), greater-than or equal (>=).
Bitwise ops: and (&), or (|), xor (^), not (~)
Logical operators: and (&&) or (||) not (!) note that these work as in C, e.g. (2 && 1) == 1
Bit reduction operators: [n] n-bit to extract
Conditional operator: ? to multiplex result
Example: (a==1)? funcif1 : funcif0
The above (if *a* is a single bit) is equivalent to:
((a==1) && funcif1)
|| ((a!=1) && funcif0)

Registers and wires
Declaring a 4 bit wire with index starting at 0:
wire [3:0] w;
Declaring an 8 bit register:
reg [7:0] r;
Declaring a 32 element memory 8 bits wide:
reg [7:0] mem [0:31]
Bit extract example:
r[5:2] returns 4 bits between pos 2 to 5 inclusive

Assignment
Assignment to wires uses the assign primitive outside an always block, e.g.:
assign mywire = a & b

Registers are assigned to inside an always block which specifies where the clock comes from, e.g.:
always@(posedge myclock)
cnt = cnt + 1;

Blocking vs. unblocking assignment <= vs. =
The <= assignment operator is non-blocking (i.e. if use in an always@(posedge) it will be performed on every positive edge. If you have many non-blocking assignments they will all updated in parallel. The <= operator must be used inside an always block – you can't use it in an assign statement.
The blocking assignment operator = can be used in either an assign block or an always block. But it causes assignments to be performed in sequential order. This tends to result in slower circuits, so avoid using it (especially for synthesized circuits) unless you have to.

Case and if statements
Case and if statements are used inside an always block to conditionally update state, e.g.:
always @(posedge clock)
if (add1 && add2) r <= r+3;
else if (add2) r <= r+2;
else if(add1) r <= r+1;
Note that we don't need to specify what happens when add1 and add2 are both false since the default behavior is that r will not be updated.
Equivalent function using a case statement:
always @(posedge clock)
case({add2,add1})
2'b11 : r <= r+3;
2'b10 : r <= r+2;
2'b01 : r <= r+1;
default: r <= r;
endcase

Module declarations
Modules pass inputs, outputs as wires by default.
module ModName (
output reg [3:0] result, // register output
input [1:0] bitsin, input clk);
... code ...
endmodule

Verilog Cheats Verilog_CheatSheet.pdf
A concise reference manual focusing on what you need to know for this course.

Verilog Cheat sheet for C programmers.pdf
A useful quick reference sheet planned for use by C programmers.

Verilog_Reference.pdf
Detailed Verilog reference (used in EEE4120F) with more explanations and examples.

OUTLINE OF LECTURE

- Programmable chips
- **Impressiveness of FPGAs**
- FPGA interns
- PLD/FPGA Development Flow
- Learning Verilog HDL
- Building blocks
- Non-synthesizable data types

PROGRAMMABLE CHIPS

What you probably already know…

- **What are Programmable Chips?**
- In comparison to hard-wired chips, a programmable chip can be configured according to user needs, providing a means to use the same chip(s) for a variety of different applications.
- This facility makes programmable chips attractive for use in many products, including prototyping situations and in final systems.
- Further benefits of programmable chips:

  Low starting cost (eg. Web pack + dev kit), risk reduction, quick turnaround time

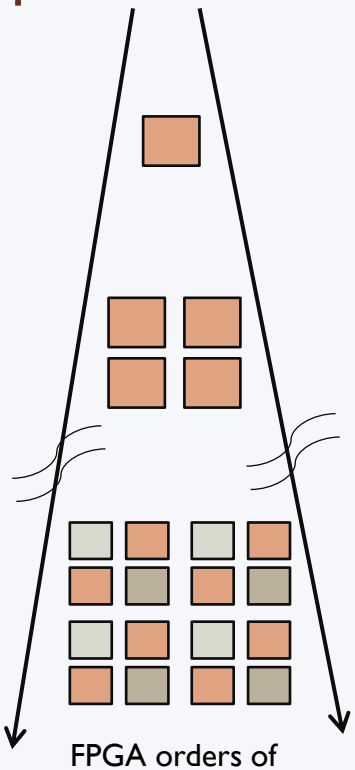
PROGRAMMABLE CHIPS

Types of programmable logic chips:

PLAs, CPLDs and FPGAs

These vary from: simple → complex cheap → expensive

- **PLA = Programmable Logic Array**
  - Simple: just AND and OR gates; but *Cheap*
- **CPLA = Complex PLA**
  - Midrange: compose interconnected PLAs
- **FPGAs = Field Programmable Gate Array**
  - Complex: programmable logic blocks and programmable interconnects; but *Expensive*



Impressiveness of FPGAs

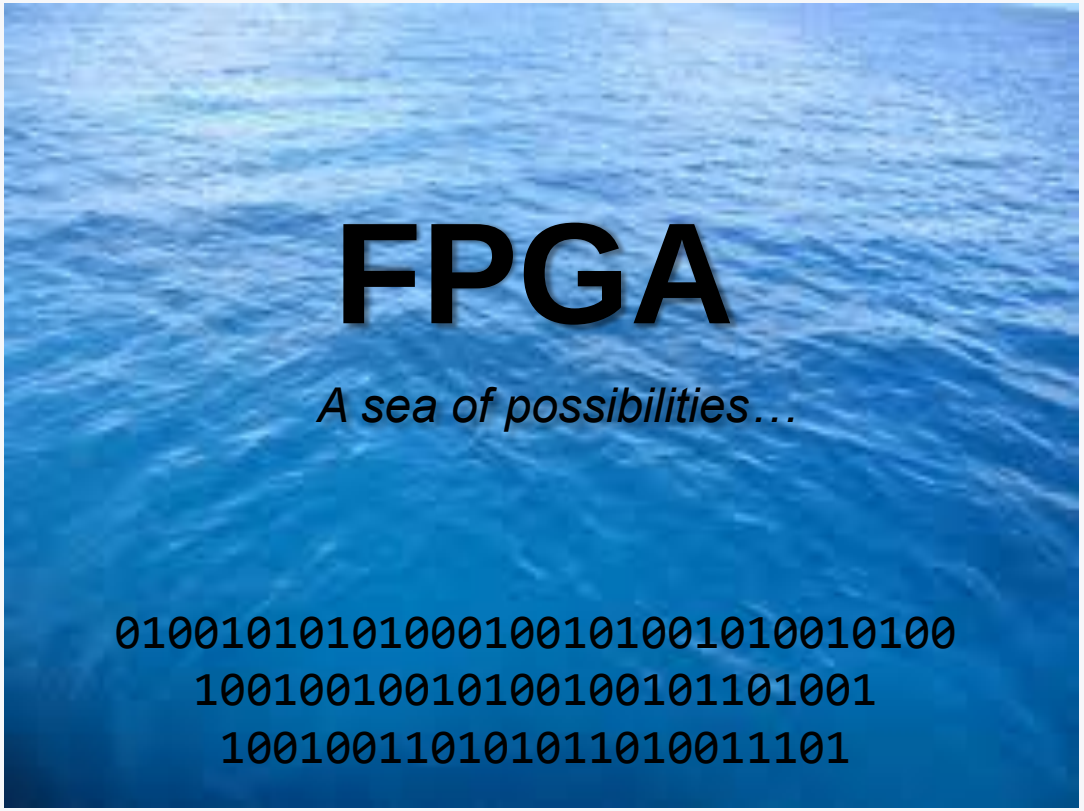
Although the principles in these lectures also apply for programming modern PLAs and CPLDs devices which are much more affordable that FPGAs.

SO WHAT?

What is so special about FPGAs?

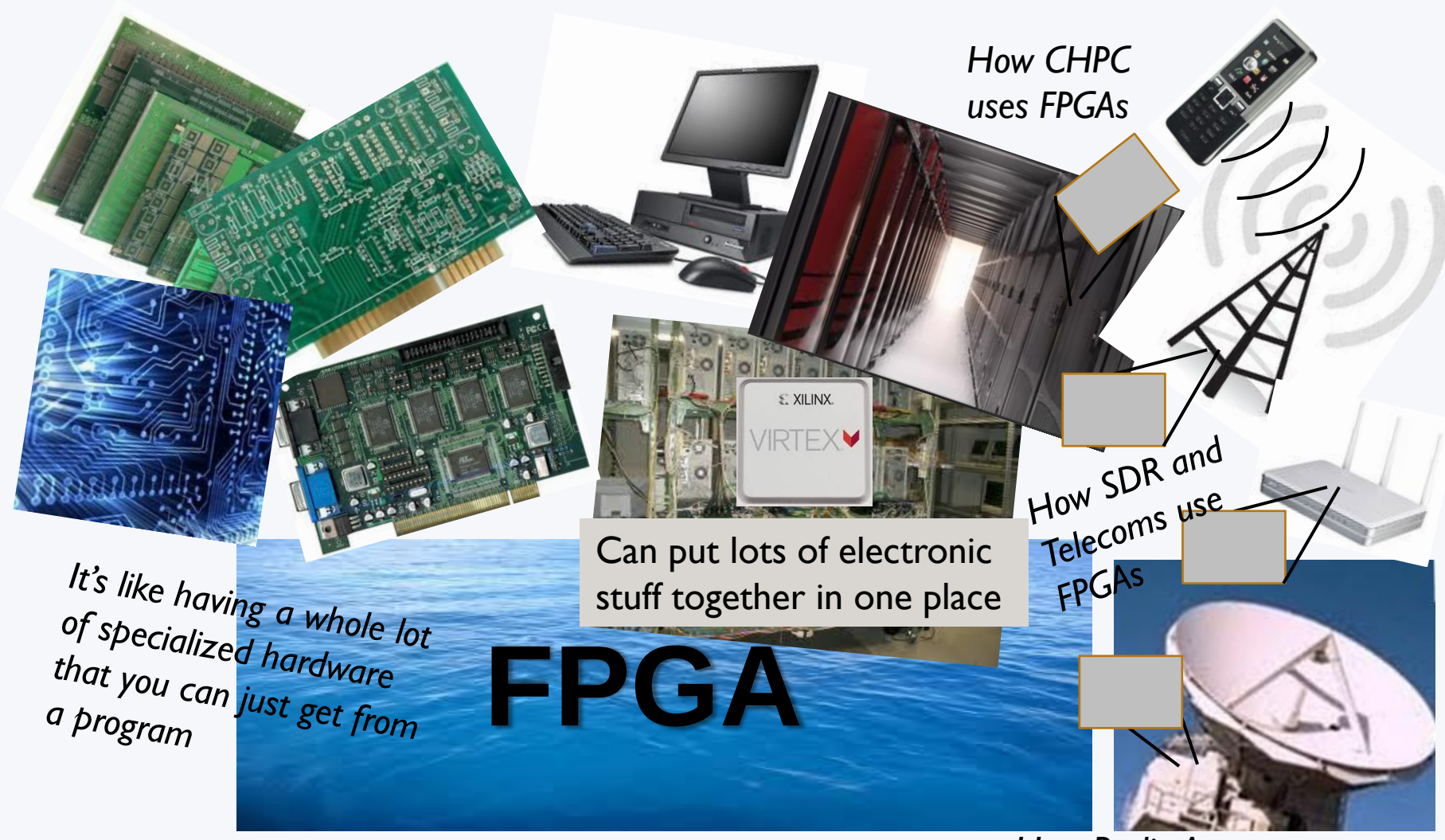
Why should we be bothered?

Because they are so flexible and highly parallel



all in a little chip!

WHAT IS SO SPECIAL ABOUT FPGA?



FPGA INTERNS:
A BASIC VIEW ON HOW FPGAS WORK

although I'm not saying to much about it now…

FPGA INTERNAL STRUCTURE

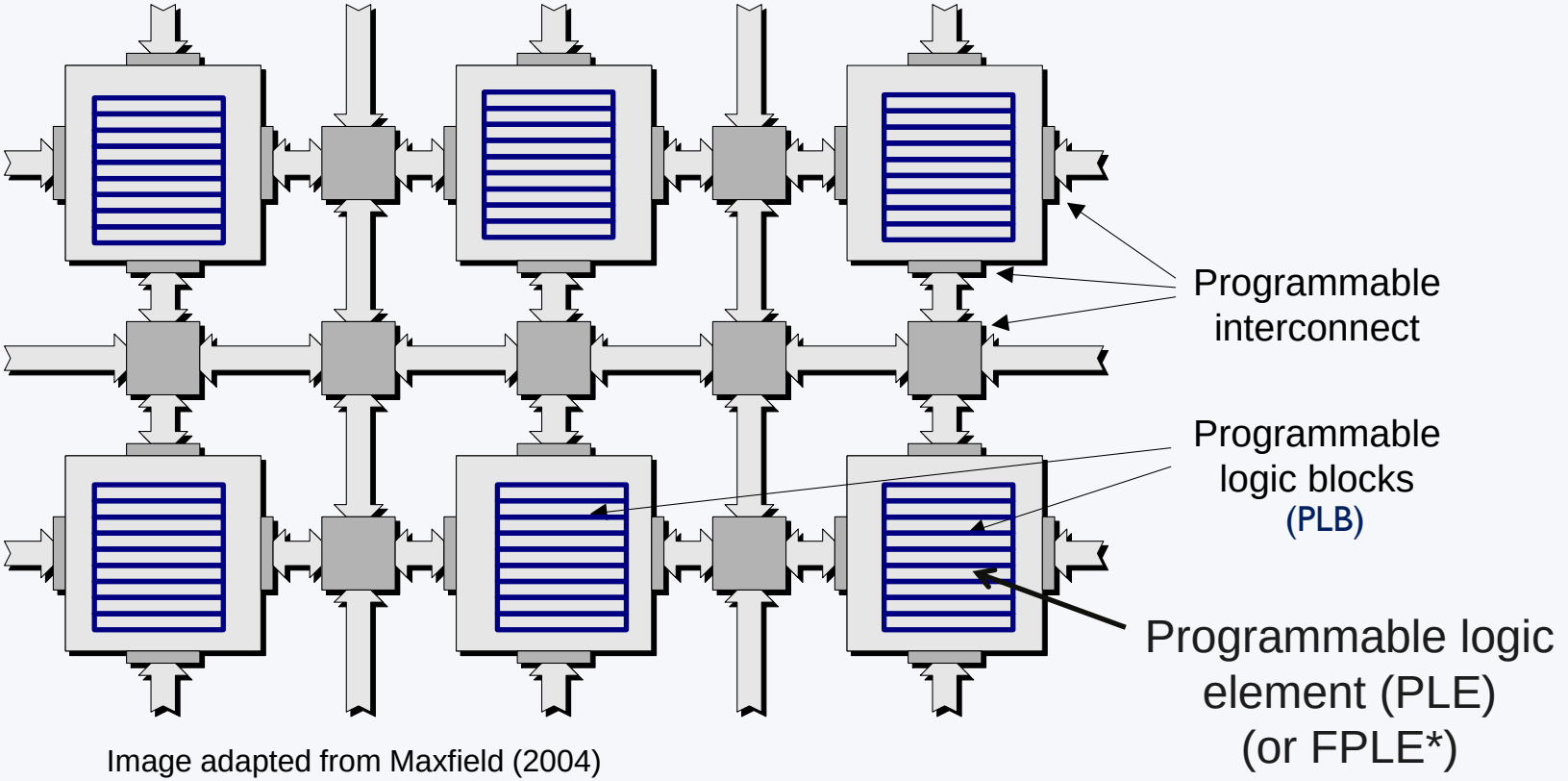


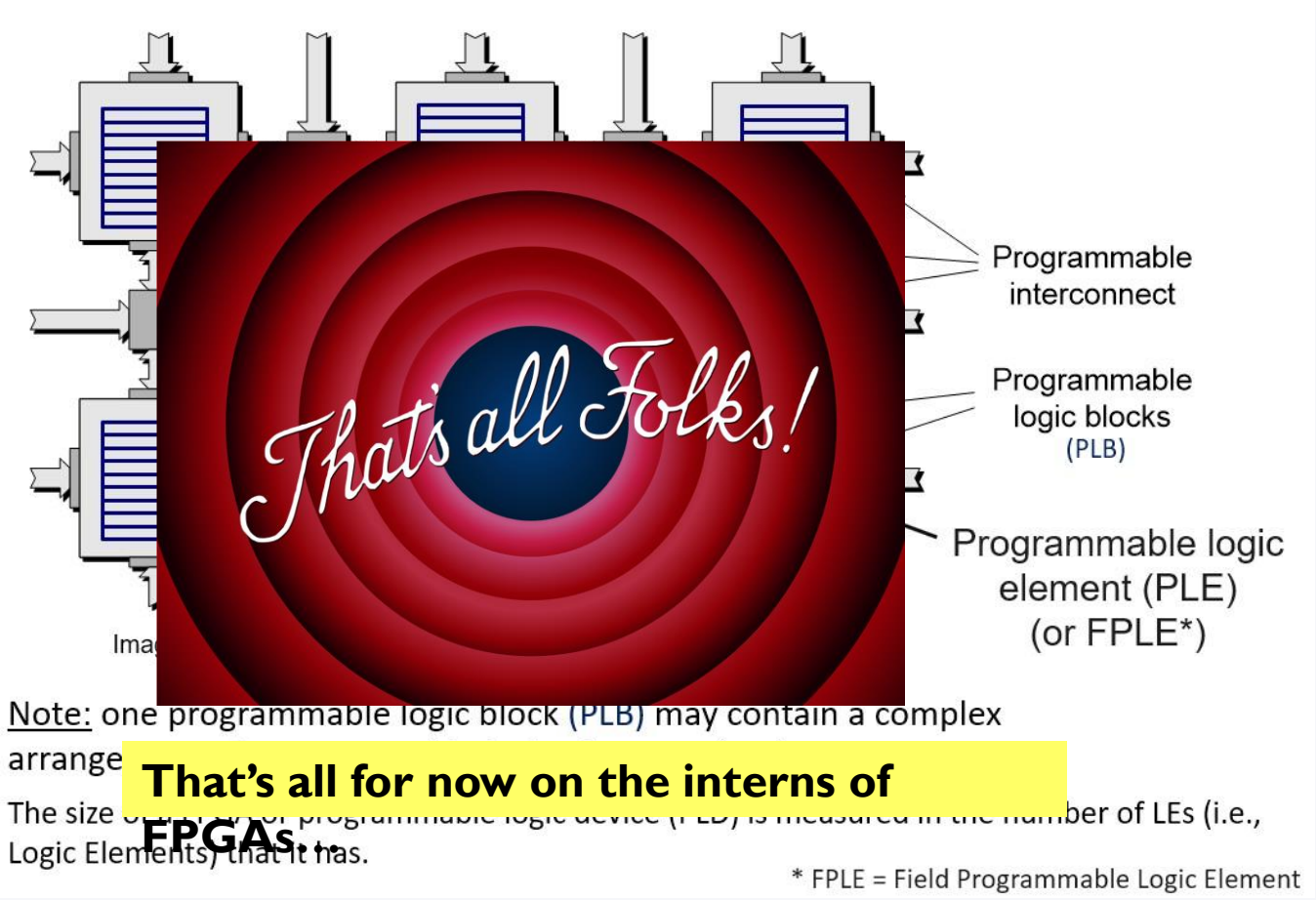
Image adapted from Maxfield (2004)

Note: one programmable logic block (PLB) may contain a complex arrangement of programmable logic elements (PLE).

The size of a FPGA or programmable logic device (PLD) is measured in the number of LEs (i.e., Logic Elements) that it has.

* FPLE = Field Programmable Logic Element

FPGA INTERNAL STRUCTURE



In EEE4120F we dive much deeper into FPGA issues if you might be interested in pursuing that.

PLD/FPGA DEVELOPMENT FLOW

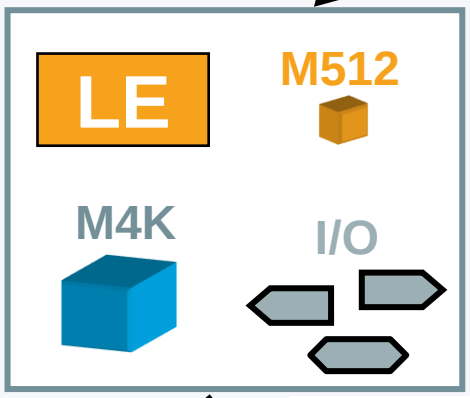
Design Specification

Design and HDL / RTL Coding

- Behavioral or Structural Description of Design
- Writing VHDL, deciding i/o, formulating tests

RTL Simulation

- Functional Simulation
- Verify Logic Model & Data Flow
- View model-specified timing



Synthesis

- Translate Design into Device Specific Primitives
- Optimization to meet Area & Performance Constraints

Place and Route (PAR)

- Map primitives to specific locations inside FPGA with reference to area & performance constraints
- Specify routing resources to use

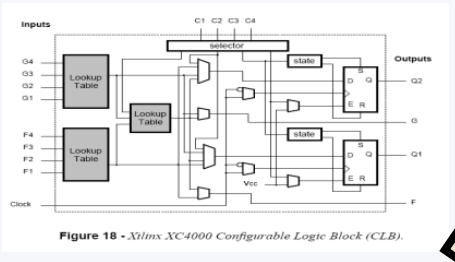


Figure 18 - *Xilinx XC4000 Configurable Logic Block (CLB).*

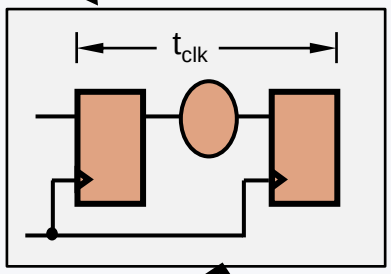
... PTO ...

This development cycle diagram is an adaptation of that prepared by Dr. Junaid Ahmed Zubairi,

Avail: http://www.cs.fredonia.edu/zubairi/training/fpga.ppt

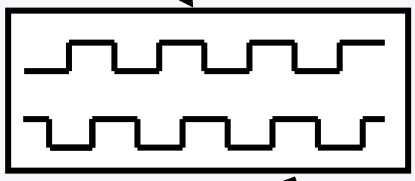
PLD/FPGA DEVELOPMENT FLOW (CONT)

Place and Route (PAR)



Timing Analysis

- Verify performance specifications
- Static timing analysis



Gate Level Simulation (GLS)

- Timing simulation
- Verify design will work on target platform

Program and test on hardware

- Generate bit file
- Program target device
- Activate the system

PLD/FPGA DEVELOPMENT FLOW (CONT)

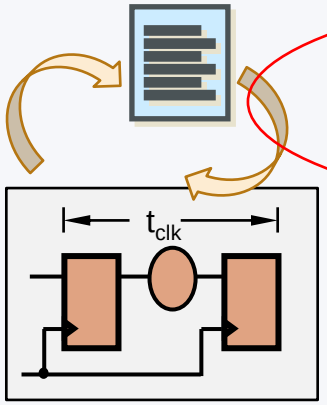
Where is the most time spent?

Every development project is different

But in my own experience, most of the time is probably spent …

Engineer's time

PC's time



Design and RTL Coding
- Behavioral/Structural Description of Design
- Writing HDL, deciding i/o, formulating tests

Timing Analysis
- Verify performance specifications
- Static timing analysis

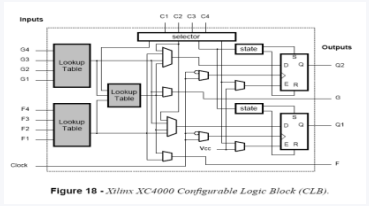
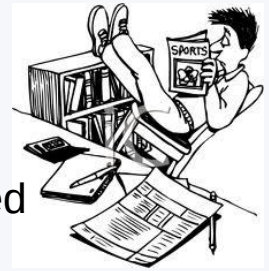


Figure 18 - *Xilinx XC4000 Configurable Logic Block (CLB).*

Place and Route (PAR)
- Map primitives inside FPGA
- Specify routing resources used



Little glimpse of FPGAs

That's all for now...