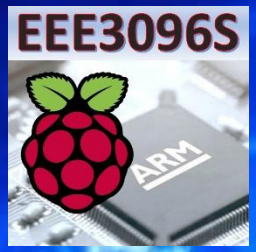


Hardware Description Languages

a taste in 5-lectures Part1

Embedded Systems II

L37

Dr Simon Winberg

Electrical Engineering
University of Cape Town

Hardware Description Languages: *a taste in 5-lectures*

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

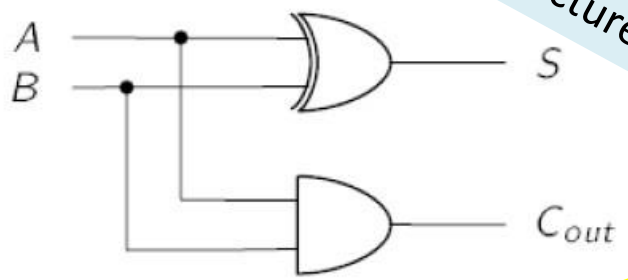
- To learn the essential syntax
- To understand simple HDL code
- To know what HDL is used for

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

These slides are adapted mainly Simon's *FPGA Bootcamp* and also from EEE4084F 2018 Verilog Intro

Digital Circuit

Lectures 23-27



The Half Adder Circuit

These slides will make use of EEE3096S Lectures 23-27

Verilog

Lectures 30-34

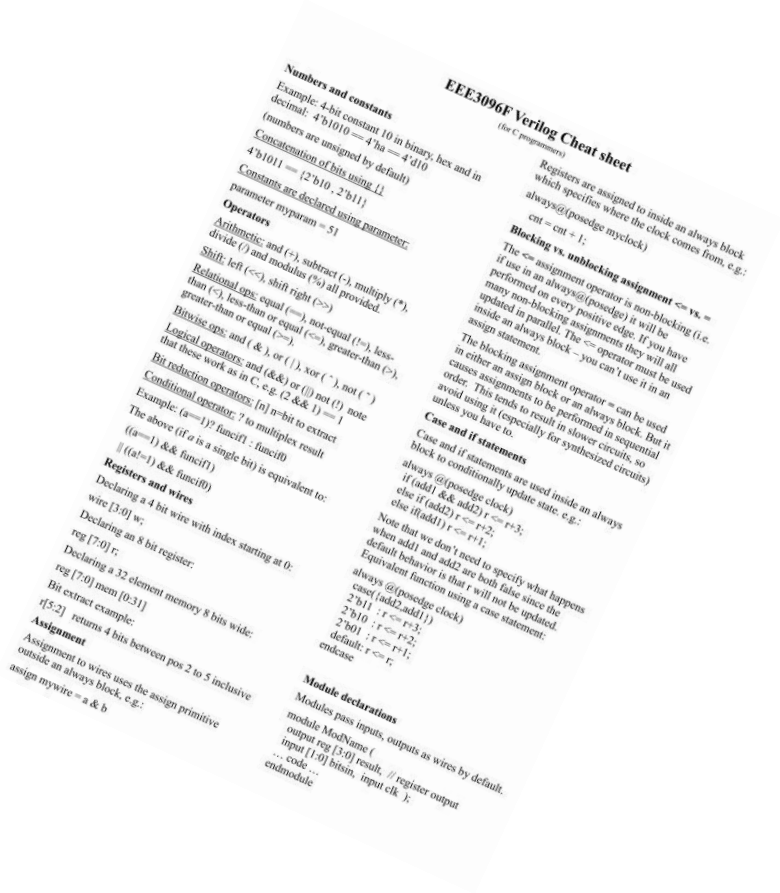
```
module half_adder (A,B,S,Cout);
  input  A, B;
  output S, Cout;

  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 lot like C / Python (Verilog95 syntax is probably simpler actually).

So without further ado…

Let's get started on Verilog HDL!!

Handout:



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 (<), greater-than (>), greater-than or equal (>=), less-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

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

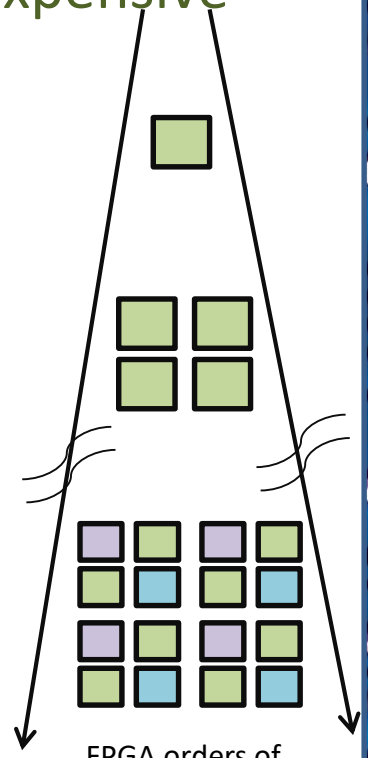
Programmable Chips

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*



FPGA orders of magnitude larger than CPLD

Impressiveness of FPGAs

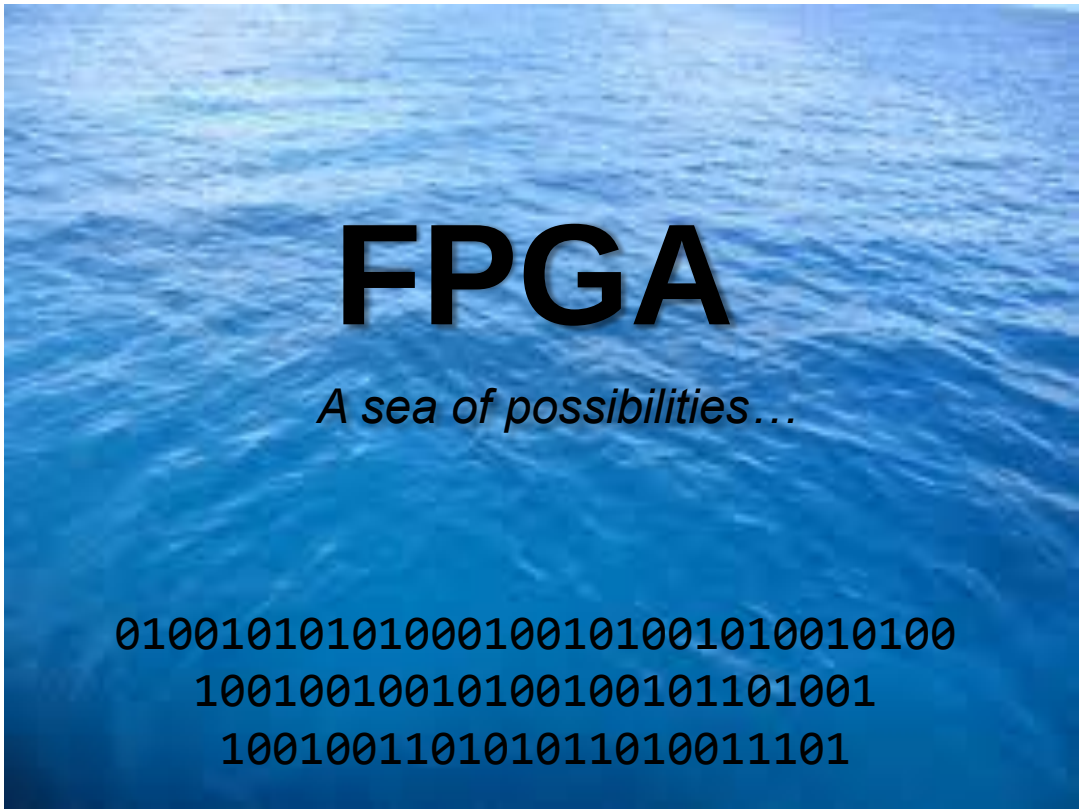
Although the principles in these lectures also apply for programming modern PLAs and CPLDs devices which are much more affordable that FPGAs.

SO WHAT?

What is so special about FPGAs?

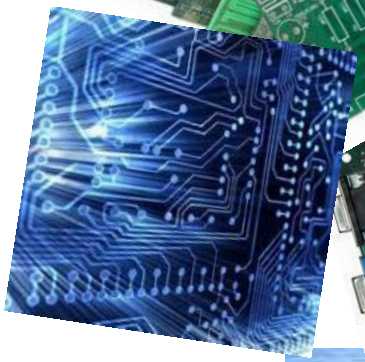
Why should we be bothered?

Because they are so flexible and highly parallel



all in a little chip!

What is so special about FPGAs?



How CHPC uses FPGAs

How SDR and Telecoms use FPGAs

Can put lots of electronic stuff together in one place

FPGA

It's like having a whole lot of specialized hardware that you can just get from a program

How Radio Astronomy use FPGAs

FPGA Interns:
a basic view on how FPGAs work

I’m not saying much about it now…

FPGA internal structure

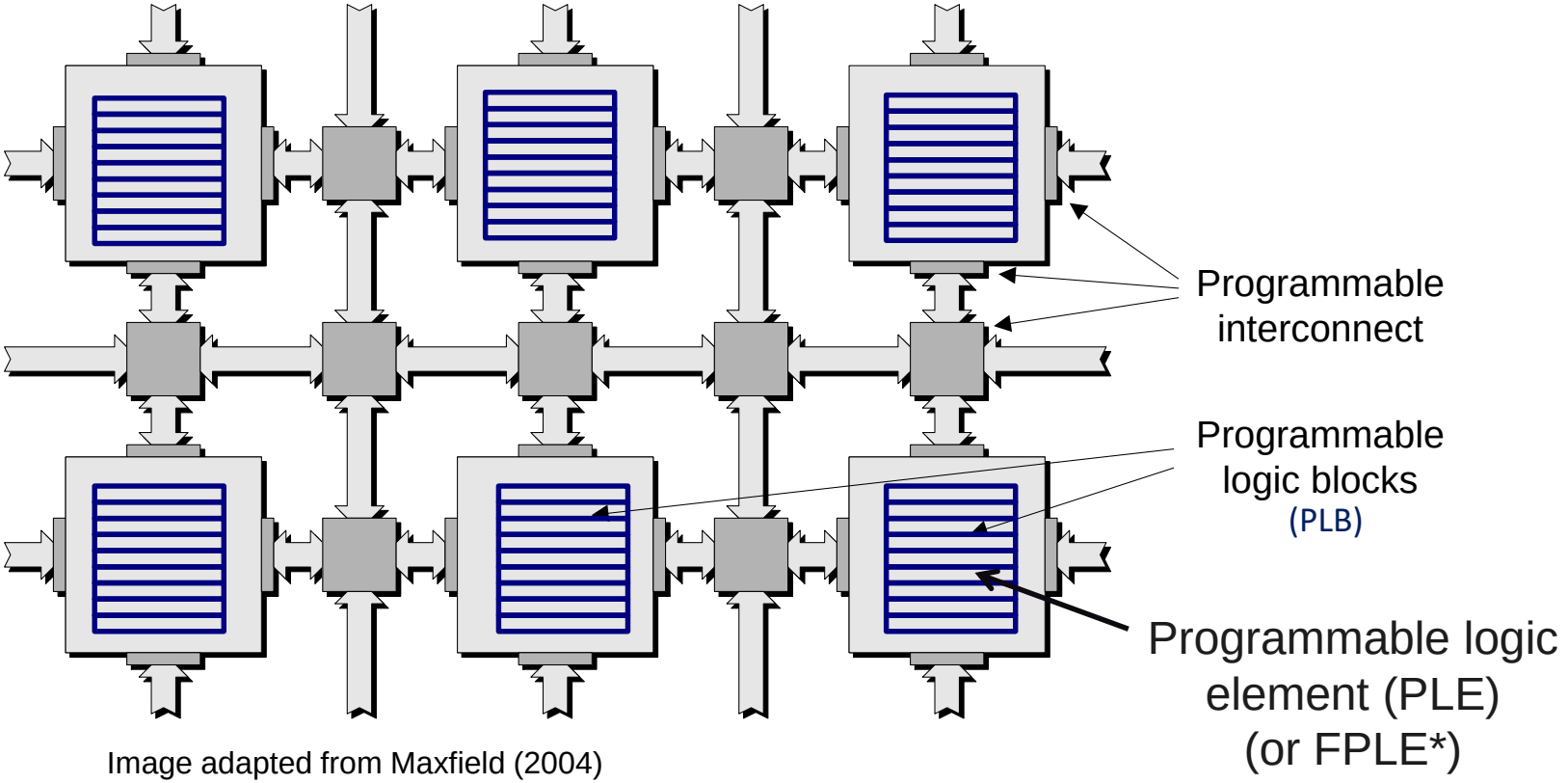


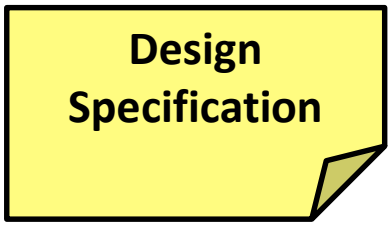
Image adapted from Maxfield (2004)

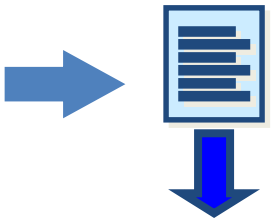
Note: one programmable logic block (PLB) may contain a complex arrangement of programmable logic elements (PLE).

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

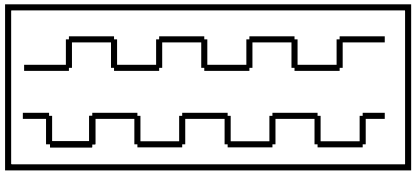
PLD/FPGA Development Flow





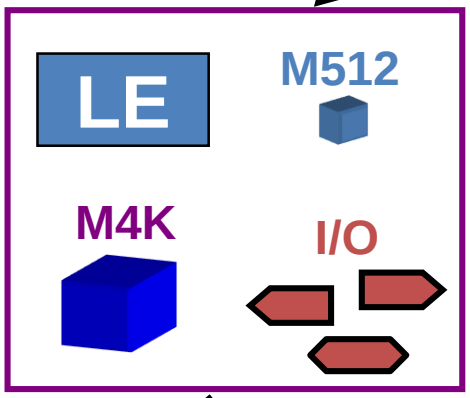
Design and RTL Coding

- 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

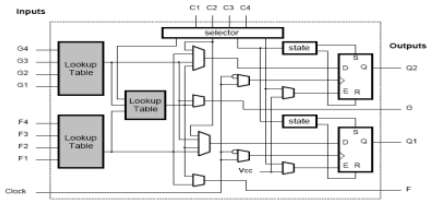


Figure 18 - *Xilinx XC4000 Configurable Logic Block (CLB).*

Place and Route (PAR)

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

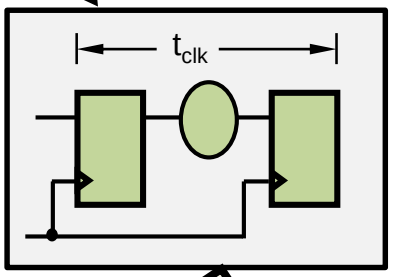
... PTO ...

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

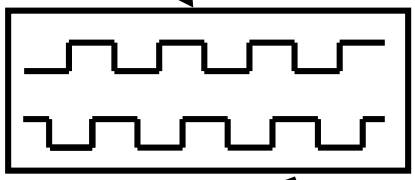
Avail: http://www.cs.fredonia.edu/zubairi/training/fpga.ppt

PLD/FPGA Development Flow (cont)

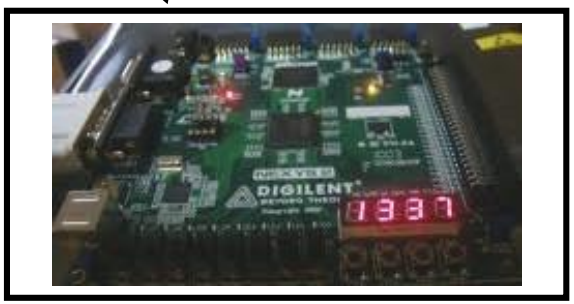
Place and Route (PAR)



Timing Analysis
- Verify performance specifications
- Static timing analysis



Gate Level Simulation
- Timing simulation
- Verify design will work on target platform



Program and test on hardware
- Generate bit file
- Program target device
- Activate the system

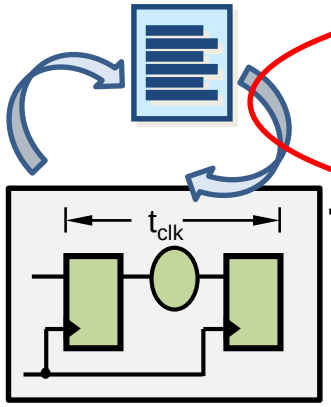
PLD/FPGA Development Flow (cont)

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

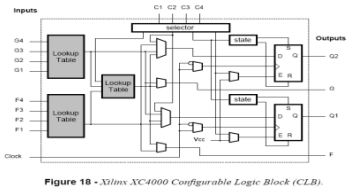
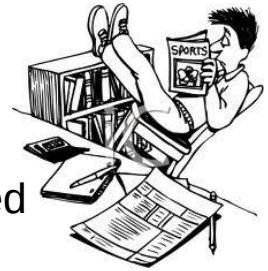


Figure 18 - *Xilinx XC4000 Configurable Logic Block (CLB).*

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



Little glimpse of FPGAs

Starting on Verilog

Learning Verilog

Best way to learn HDL…

is through practice coding with it!

See the Verilog Cheat Sheet included in resources

See practice coding exercises on Vula

Online learning and reference sources for Verilog programming

Since we are not going in-depth into Verilog an appropriate reference is the one provided by **Wikibooks** (these links are concise resources which would be adequate for introduction level what we are doing) :

https://en.wikibooks.org/wiki/Programmable_Logic

https://en.wikibooks.org/wiki/Programmable_Logic/Verilog

ASIC World is an excellent web resource, the goto place for useful support and examples (at a introductory to intermediate level) :

http://www.asic-world.com/verilog/index.html

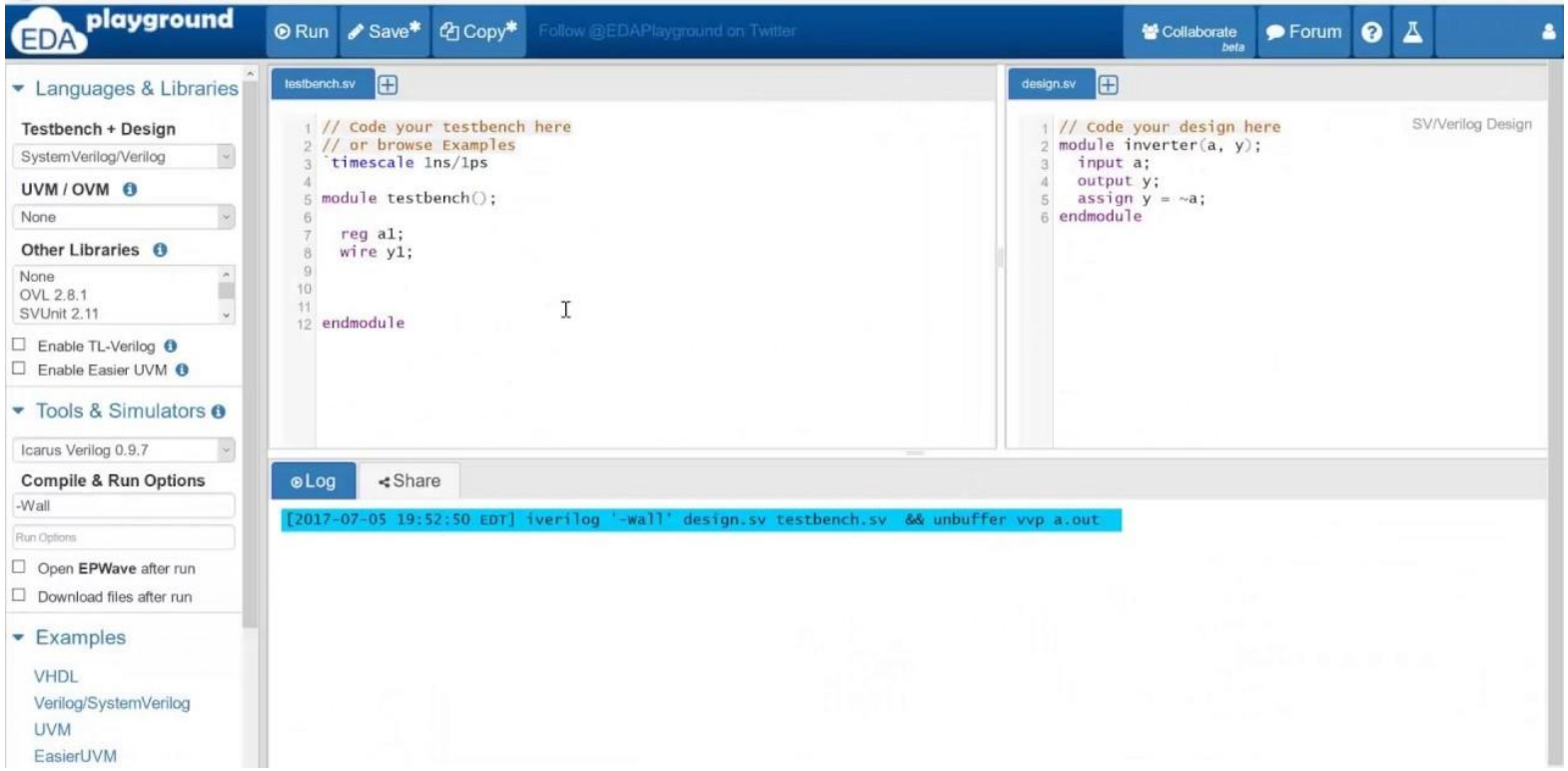
http://www.asic-world.com/examples/verilog/

ASIC WORLD

EDA Playground

one of Simon's favourite!

An easy to access tool that you can get started with! Why not try it out.



https://www.edaplayground.com/

NOTE: How to set it up…

- Log in via Gmail or create a **register a new login** so that you can save designs
- Choose the simulator: from Tools & Simulators select **Icarus Verilog 0.10.0**

iVerilog : Icarus Verilog

If you would prefer using Verilog offline, or run simulations on your own computer, or develop more substantial designs, then you may want to use Icarus Verilog. It's free!

iVerilog is a compiler:

iVerilog will parse the Verilog code and generate an executable that the PC can run (called a.out if you don't use the flags to change the output executable file name)

```
swinberg@forge:~/TestVeri$ iverilog mynand.v
swinberg@forge:~/TestVeri$ ./a.out
A = 0, B = 0, Nand output w = 1

A = 0, B = 1, Nand output w = 1

A = 1, B = 0, Nand output w = 1

A = 1, B = 1, Nand output w = 0

swinberg@forge:~/TestVeri$
```

http://iverilog.icarus.com/

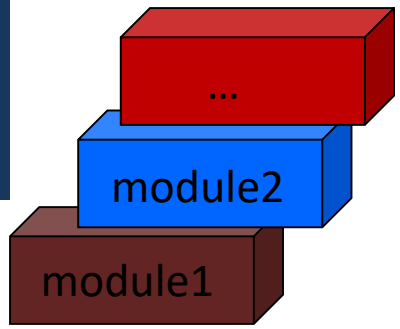
Available on Vula in Resources/Software/iverilog-10.1.1-x64_setup

For Ubuntu or Debian you can install it (if linked to the leg server) using:

```
apt-get install iverilog
```

one of Simon's favourites!

Module: Building block of Verilog Programs



Module: the basic block that does something and can be connected to (i.e. equivalent to entity in VHDL)

Modules are hierarchical. They can be individual elements (e.g. comprise standard gates) or can be a composition of other modules.

SYNTAX: **module** <*module name*> (<*module terminal list*>);

...

<*module implementation*>

...

endmodule

Module Abstraction Levels

- Switch Level Abstraction (lowest level)
  - Implementing using only switches and interconnects.
- Gate Level (slightly higher level)
  - Implementing terms of gates like (i.e., AND, NOT, OR etc) and using interconnects between gates.
- Dataflow Level
  - Implementing in terms of dataflow between registers
- Behavioral Level (highest level)
  - Implementing module in terms of algorithms, not worrying about hardware issues (much). Close to C programming.

Arguably the best thing about Verilog!!

Syntactic issues: Constant Values in Verilog

- Number format:

  <size>'<base><number>

- Some examples:
  - 3'b111 – a three bit number (i.e. 7_{10})
  - 8'ha1 – a hexadecimal (i.e. $A1_{16} = 161_{10}$)
  - 24'd165 – a decimal number (i.e. 165_{10})

Defaults:

100 – 32-bit decimal by default if you don't have a '

'hab – 32-bit hexadecimal unsigned value

'o77 – 32-bit hexadecimal unsigned value ($77_8 = 63_{10}$)

Syntactic issues: Constant Values in Verilog

Constant	Hardware Condition
0	Low / Logic zero / False
1	High / Logic one / True
x	Unknown
z	Floating / High impedance

NB!

```
// Example for linking a data bit to a bus if module
// is  asked to put value on the bus via a link input
module myfunc (output busbit, input link  );
  reg dat; // stores result worked on
  assign busbit = link ? dat : 1'bz;
  // do stuff to compute dat
  assign dat = 1'b1;
endmodule
```

bitbus
(1-bit bus)

can selectively drive a logic voltage (0 or 1) or **z** to output a high-impedance that effectively disconnects myfunc from the bus

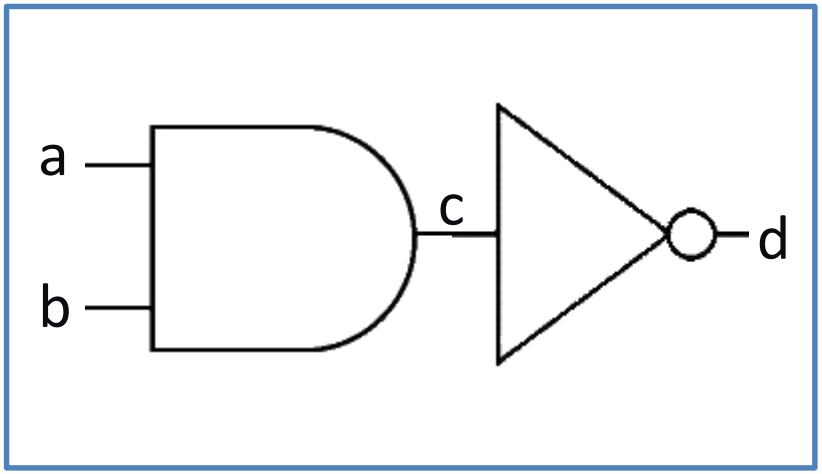
```
// Testbench
module myfunc_tb;
  reg y, lk;
  myfunc f1 (y,lk);
  initial
   begin
    $monitor("y=%b link=%b\n", y,lk);
    lk= 0;  #10   // initially y not linked
    lk=1; #10    // now y is linked
    $finish;
   end
endmodule
```

Next lecture introduces Verilog program structure and test benches

Try it online: https://www.edaplayground.com/x/3tUK

Wires

- Wires (or nets) are used to connect elements (e.g. ports of modules)
- Wires have values continuously driven onto them by outputs they connect to




```
// Defining the wires
// for this circuit:

wire a;
wire a, b, c;
```

Registers

- Registers store data
- Registers retain their data until another value is put into them (i.e. works like a FF or latch)
- A register needs no continuous driver

```
reg myregister;  // declare a new register  (defaults to 1 bit)

myregister = 1'b1;  // set the value to 1
```

Vectors of wires and registers

```
// Define some wires:
wire a;  // a bit wire
wire [7:0] abus;  // an 8-bit bus
wire [15:0] bus1, bus2; // two 16-bit busses
// Define some registers
reg active; // a single bit register
reg [0:17] count; // a vector of 18 bits
```

Non-synthesisable Data types

These datatypes are used both during the compilation and simulation stages to do various things like checking loops, calculations.

- Integer 32-bit value

  integer i; // e.g. used as a counter

- Real 32-bit floating point value

  real r; // e.g. floating point value for calculation

- Time 64-bit value

  time t; // e.g. used in simulation for delays

Memory jogger…

Q: Explain the Verilog value 10'h10 …

A:

(a) 10-bit value 10_{10}

(b) Arbitrary size value 16_{10}

(c) 10-bit value 10_{16}

(d) array [16,16,16,16,16,16,16,16,16,16]

That’s all for now…

The Next Episode…

Lecture L38

- Building modules in Verilog
- Built-in parts and basic building blocks
- Types of Verilog file
- Coding style