



EEE4120F



High Performance Embedded Systems

The background details to FPGAs were covered in Lecture 14. This lecture launches into HDL coding.

Lecture 15 Coding in Verilog

I'm going to go through the main points of this lecture to help you make an effective start on the HDL prac for the course.

```
module myveriloglecture ( wishes_in, techniques_out );  
    ...  
    // implementation of today's lecture  
    ...  
endmodule
```

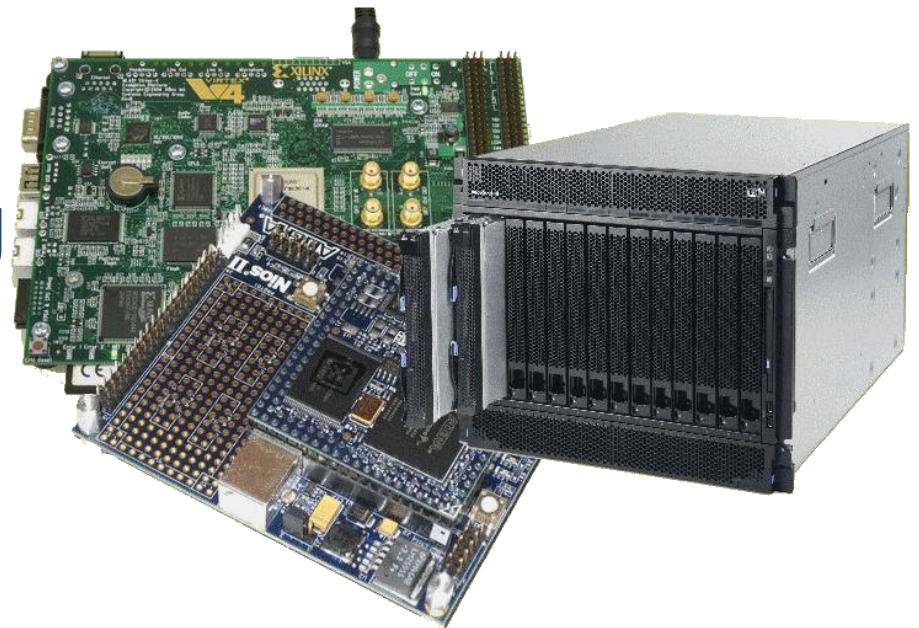
Lecturer:
Simon Winberg

*Learning Verilog with
Xilinx Vivado, Icarus Verilog
or Intel Altera Quartus II*

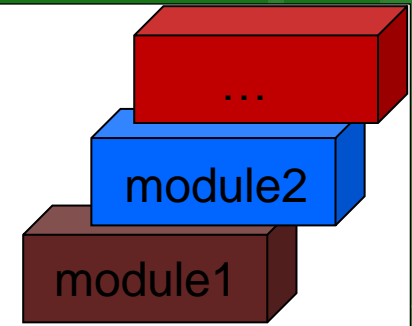


Lecture Overview

- Basics of Verilog coding
- Exercise
- Verilog simulators
- Intro to Verilog in ISE/Vivado
- Test bench
- Generating Verilog from Schematic Editors



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 ($77_8 = 63_{10}$)

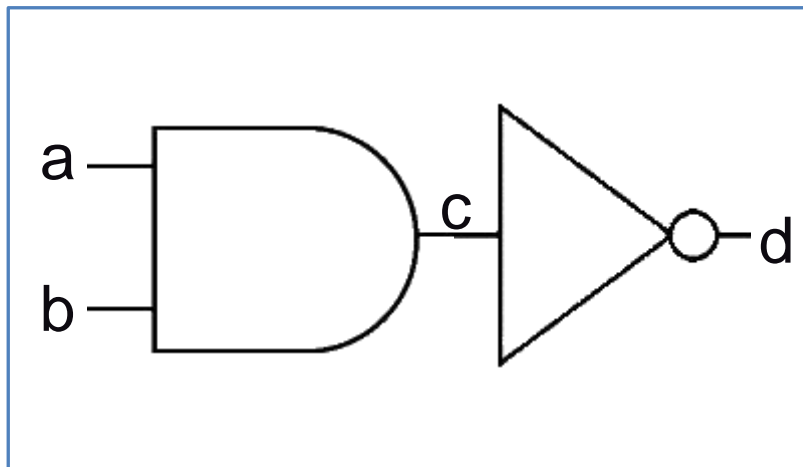
Syntactic issues:

Constant Values in Verilog

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

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

Verilog Parameters & Initial block

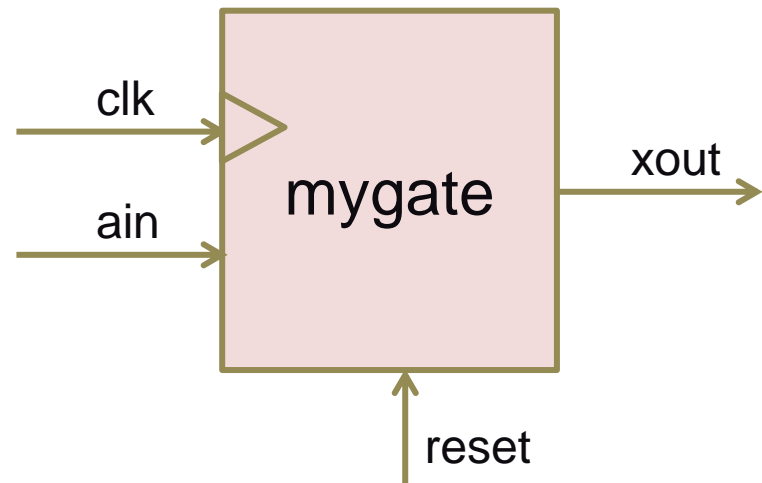
- **Parameter:** the rather obscurely named 'parameter' works more like a constant in C (or **generic** in VHDL)
 - Parameters can be used in implementation of both synthesisable and simulation code
- **Initial:** used to initialize parameters or registers or describe a process for initializing a module (i.e. like constructor in C++)
 - An initial block is effectively an always@ block that is triggered on the start of simulation (NB: it is not synthesisable)
 - Initial can only be used in simulation code

Ports

- The tradition is to **list input ports first** and then output ports. This makes reading of code easier. i.e.:

ModuleName (<input ports> <output ports>);

```
module mygate (  
    reset, // reset line if used  
    clk ,  // clock input  
    xout,  // 1 bit output  
    ain ); // a 1 bit input  
// define inputs  
input reset, clk, ain;  
// define outputs  
output xout;  
... rest of implementation ...  
endmodule
```



Register Output Ports

- These are output port that hold their value. An essential feature needed to construct things like timers and flip flops

```
module mycounter (  
    clk,          // Clock input of the design  
    count_out // 8 bit vector output of the  
);  
// Inputs:  
input clk;  
output [7:0] count_out; // 8-bit counter output  
// All the outputs are registers  
reg [7:0] count_out;  
  
...  
endmodule
```

Instantiating modules and connecting up ports

- These two tasks usually done in one go...
- Modules are instantiated within modules

Syntax: <module name> <instance name> (<arguments>)

EXAMPLE:

```
// Multiplexer implemented using gates only*
```

```
module mux2to1 (a,b,sel,y);
```

```
  input a,b,sel;
```

```
  output y;
```

```
  wire sel, asel, bsel, invsel;
```

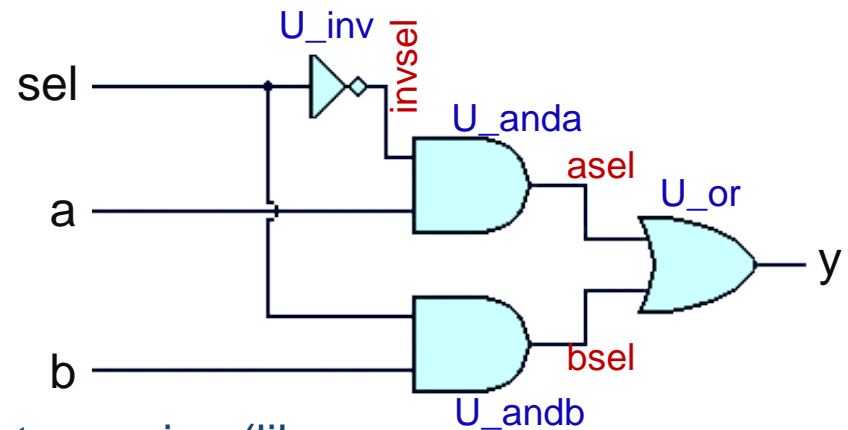
```
  not U_inv (invsel,sel);
```

```
  and U_anda (asel,a,invsel),
```

```
      U_andb (bsel,b,sel);
```

```
  or U_or (y,asel,bsel);
```

```
endmodule
```



Module instance names

Port mapping (like arguments in a C function call)

Instantiating modules

- Why give instances names?

- In Verilog 2001 you can do:

```
module mux2to1 (input a, input b, input sel, output y);  
  ...  
  and (asel,a,invsel), // can have unnamed instance  
  ...  
endmodule
```

Major reason for putting a name in is when it comes to debugging: Xilinx tends to assign instance names arbitrarily, like the and above might be called XXYY01 and then you might get a error message saying something like “cannot connect signals to XXYY01” and then you spend ages trying to track down which gate is giving the problem.

Verilog Primitive Gates

and or not
nand nor xor

Examples:

```
and a1 (OUT,IN1,IN2);  
not n1 (OUT,IN);
```

Buffer (i.e. 1-bit FIFO or splitter)

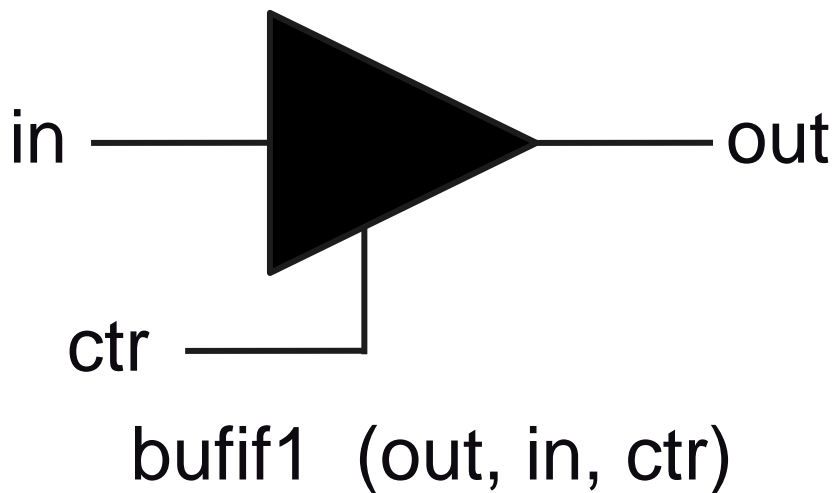
buf

Example:

```
buf onelinkbuf (OUT, IN);  
buf twolinkbuf (OUT1, OUT2, IN);
```


Bufif (hardware if gate)

Tri-state buffer. Can choose to drive *out* with value of *in* (if *ctr* = 1) or don't drive anything to *out* (i.e. if *ctr* = 0 connect high impedance to *out*)



bufif1 operation

	ctr →			
⊘	0	1	x	z
0	z	0	L	L
1	z	1	H	H
x	z	x	x	x
z	z	x	x	x

See also notif (works in the apposite way: if *ctr*=0 then drive out with *in*)

Where to go from here...

- The preceding slides have given a brief recap of Verilog, but covered much of the major things used most commonly.
- It's best to get stuck into experimenting and testing code in order to learn this language ... which is a major reason for the YODA project.

Some thoughts for experimenting to do soon...

Verilog Recommended Coding Styles

- Consistent indentation
- Align code vertically on the = operator
- Use meaningful variable names
- Include comments (i.e. C-style // or /**/)
 - brief descriptions, reference to documents
 - Can also be used to assist in separating parts of the code (e.g. indicate row of /*****/ to separate different module implementations)

Code Example 1 : MUX

```
//-----  
// Design Name : mux_using_assign  
// File Name   : mux_using_assign.v  
// Function    : 2:1 Mux using Assign  
// Coder       : Deepak Kumar Tala  
//-----  
module mux_using_assign(  
    din_0 , // Mux first input  
    din_1 , // Mux second input  
    sel   , // Select input  
    mux_out // Mux output  
);  
//-----Input Ports-----  
input din_0, din_1, sel ;  
//-----Output Ports-----  
output mux_out;  
//-----Internal Variables-----  
wire mux_out;  
//-----Code Start-----  
assign mux_out = (sel) ? din_1 : din_0;  
  
endmodule //End Of Module mux
```

Do get into a habit of providing a preamble for each file.

Do make use of divider lines to separate different pieces of the code

Do try to provide useful comments especially if the argument names are not very obvious

For older versions of Verilog (before 2001)

Adapted from source: http://www.asic-world.com/code/hdl_models/mux_using_assign.v

Try it on EDA Playground : <https://www.edaplayground.com/> (run HDL code using online simulators)

Testbenches

- A testbench is essentially code that is written to **test your design**, or exercise a module you're building
- Basically you set up a testbench to run a series of test vectors or manipulate pins to see what happens.
- They are usually written in the same language as the module under test, but not necessarily... I often use a combination of Verilog, Matlab and/or C in my testbenches (Matlab or C to generate test vectors and a Verilog testbench module as the interface to this)

Verilog 4-bit counter with testbench

Testbench Example

EEE4120F



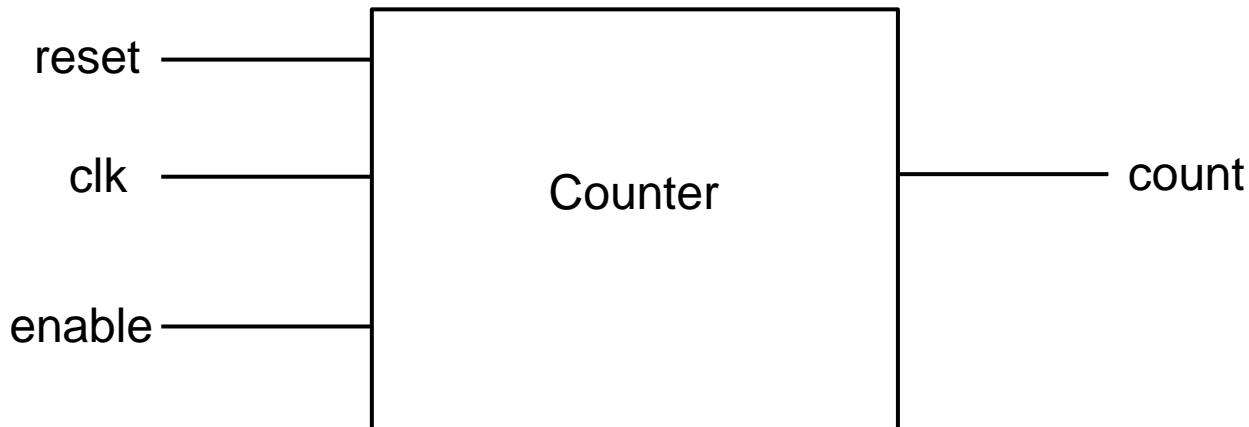
This is a brief version of testbench development that was given in EEEE3096S

Counter Design

Before you jump into coding, you should do some design...

Let's think about what a 4-bit counter needs...

- (1) A module
- (2) interfaces: some inputs... reset and clock
- (3) interfaces: an output... count value
- (4) Maybe further embellishments ... like enable line



OK, that sounds like enough for now.. Let's code it!

Code Example2 : Counter

```
//-----  
// Design Name : counter  
// File Name : counter.v  
// Function : 4 bit up counter  
// Adapted from http://www.asic-world.com/examples/verilog/counters.html  
//-----  
module counter (clk, reset, enable, count);  
// Define port types and directions  
input clk, reset, enable;  
output [3:0] count;  
reg [3:0] count;  
  
always @ (posedge clk)  
if (reset == 1'b1) begin  
    count <= 0;  
end else if ( enable == 1'b1) begin  
    count <= count + 1;  
end  
endmodule
```

Do get into a habit of providing a preamble for each file.

Here's our *port interface*, including enable and reset lines. Count is the current count value that will increase with each *clock*.

i.e. the output port count holds it value

Note: always name your .v file the same as the main module in that code file (i.e. if counter is the entry point module then name the file counter.v)

Adapted from source: <http://www.asic-world.com/examples/verilog/counters.html>

Let's do it in iVerilog



Icarus Verilog

- With iVerilog you basically need a good text editor
- Should install gnuplot too, there are ways to graph waveforms

Verilog compiles the .v code into an executable. To do so:

```
iverilog -ooutputfile inputfile.v
```

Generates an executable file called *outputfile*

So lets do: `iverilog -ocount count.v`


And amazingly we see a counter file generated...

Ooooh how fun! What exciting stuff will happen if we run it?!!!

Nothing!
Because we don't have a
testbench

Code Example2 : Counter

Do get into a habit of providing a preamble for each file.



```
//-----  
// Design Name : counter  
// File Name : counter.v  
// Function : 4 bit up counter  
// Adapted from http://www.asic-world.com/examples/verilog/counters.html  
//-----  
module counter (clk, reset, enable, count);  
// Define port types and directions  
input clk, reset, enable;  
output [3:0] count;  
reg [3:0] count;  
  
always @ (posedge clk)  
if (reset == 1'b1) begin  
    count <= 0;  
end else if ( enable == 1'b1) begin  
    count <= count + 1;  
end  
endmodule
```

Note: always name your .v file the same as the main module in that code file (i.e. if counter is the entry point module then name the file counter.v)

Adapted from source: <http://www.asic-world.com/examples/verilog/counters.html>

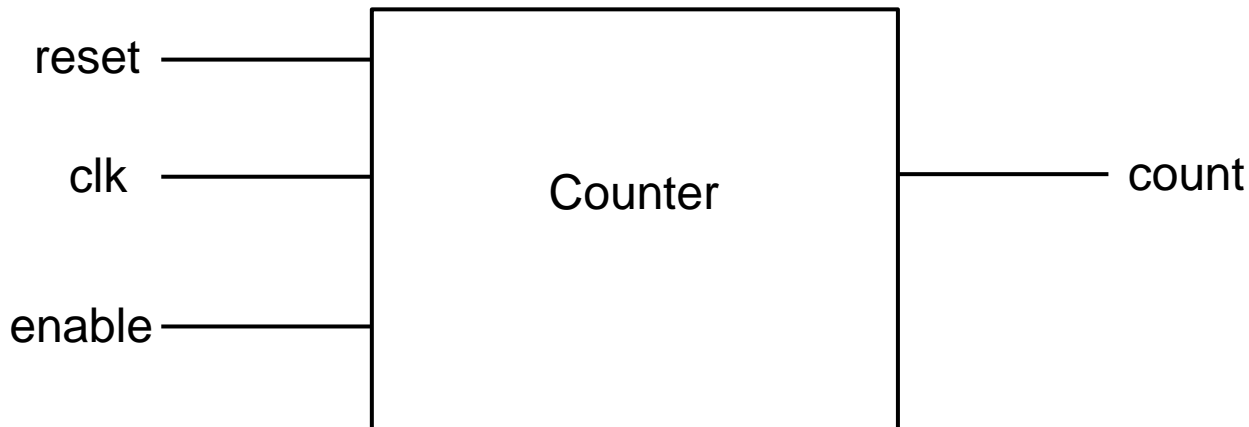
Counter Testbench Design

So again before you jump into coding, do some more design...

Let's think about how to test a 4-bit counter...

Basically you need:

1. \$monitor the lines you want to see.
2. Do a reset high and toggle clock (because it is an active reset)
3. Then continue on setting reset low and enable high and continuously toggle the clock



OK, that sounds like a plan... let's do it!

Code Example2 : Counter_tb1

```
// Counter Test bench version 1
// This just hooks up the test bench

module counter_tb;           // this will become the TLM
    reg clk, reset, enable; // define some regs, like global vars
    wire [3:0] count;       // just need a wire for count as it is stored
                             // within the counter module
    counter U0 (             // instantiate the counter (U0 = unit under test)
        .clk    (clk),      // these are explicit port maps (usually one
        .reset  (reset),    // doesn't both to do this in Verilog).
        .enable (enable),
        .count  (count)
    );

endmodule
```

But this will of course still not do anything in the simulator...
we need to exercise some pins!

What we get out:

```
$ ./counter_tb2
time,    clk,    reset,    enable,    count
   0,    0,    0,    0,    x
   5,    1,    1,    0,    0
  10,    0,    1,    0,    0
```

Code Example2 : Counter_tb3

Monitor pins and change
some of their values

```
// Counter test bench 3
// Set up a monitor and change some pins
// Coder: S. Winberg

// 4-bit Upcounter testbench
module counter_tb;
    reg clk, reset, enable;
    wire [3:0] count;

    counter U0 (
        .clk (clk), .reset (reset), .enable (enable), .count (count) ); // instantiate the module

    initial
    begin
        // Set up a monitor routine to keep printing out the
        // Now exercise the pins!!!
        clk = 0;  reset = 0;  enable = 0;
        #5 clk = !clk; // The # says pause for x simulation steps
        reset = 1;
        #5 clk = !clk; // Let's just toggle it again for good measure
        reset = 0; // Lower the reset line
        enable = 1; // now start counting!!

        repeat (10) begin
            #5 clk = !clk; // Let's just toggle it a few more times
        end

    end
endmodule
```

But First...

Test Your Knowledge:

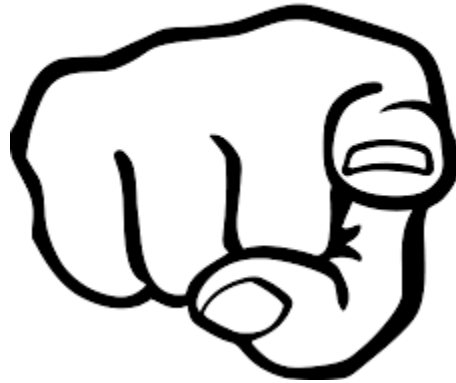
What will *count* count up to with this code??

Note: you need to tell iverilog all the files to include, so use:
iverilog -o counter_tb3 counter_tb3.v counter.v

What we get out:

```
swinberg@forge:~/Verilog$ ./counter_tb3
time,      clk,      reset,      enable,      count
  0,       0,       0,       0,          x
  5,       1,       1,       0,          0
 10,       0,       0,       1,          0
 15,       1,       0,       1,          1
 20,       0,       0,       1,          1
 25,       1,       0,       1,          2
 30,       0,       0,       1,          2
 35,       1,       0,       1,          3
 40,       0,       0,       1,          3
 45,       1,       0,       1,          4
 50,       0,       0,       1,          4
 55,       1,       0,       1,          5
 60,       0,       0,       1,          5
```

It will count up to 5 because in each iteration of the repeat it does half a clock



Over to you
to experiment further with iverilog

Additional info: the follow slides are provided as optional additional guides

End of Term 1

Happy a happy vacation!!

Guidelines on using Xilinx Vivado / ISE

The follow slides are provided as optional additional guides, you can have a look over these to get a sense of actions to be done in Prac3. However, I do suggest going directly into Prac3 as it is planned around being a tutorial to help you become familiar with Vivado and its simulation functionality.

OPTIONAL
ADDED READING

Learning Verilog By Example

EEE4120F

Learning Verilog

- The best approach is starting small, and there are lots of example Verilog programs on line that you can test, have a look at sites such as:
 - <http://www.asic-world.com/examples/verilog/>
 - <http://www.edaboard.com/>
- Free for students Active-HDL (includes nice simulator, takes less space than ISE)
 - https://www.aldec.com/en/products/fpga_simulation/active_hdl_student
- You can also generate Verilog from the schematic editor, which can help in deciding the syntax to use... short example of how to do this follows... (at least this can be useful for quickly generating gate-based / architectural combinational logic designs)

Counter Module in Vivado

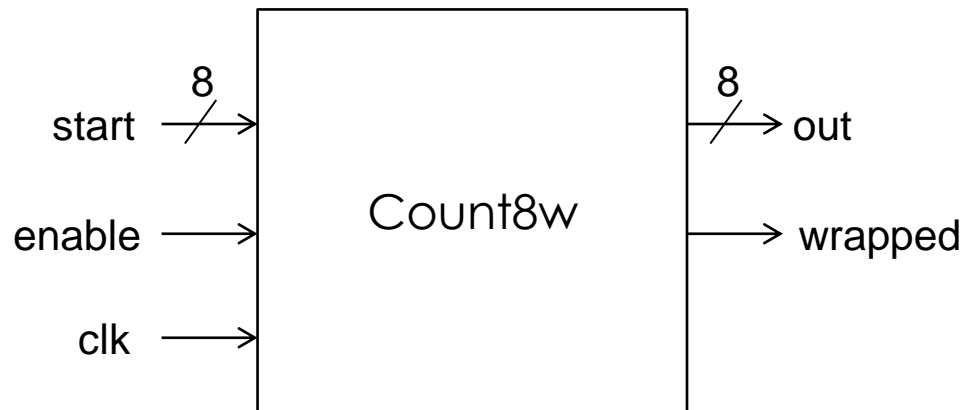
EEE4120F



Learning Verilog with Xilinx Vivado

- If you are using Vivado to practice you can do so with or without a FPGA platform connected
- BUT **you do still need to set up a desired target platform** in order to start a project (so might as well specify a commonly used training platform; this example uses a Nexys3 but you could select pretty much any option you have in your Vivado installation)

Implementing an 8-bit counter



Requirements:

INPUTS

- Want a counter that counts up for each positive edge clock pulse on *clk*
- Input line *enable* that to enables (1) or disables (0) the count operation
- An 8-bit *start* value that specifies the starting value for the counter and is loaded when *enable* is 0

OUTPUTS

- B-bit output *out* that provides the current counter value.
- *Wrapped* changes from 0 to 1 each time the counter wraps (i.e. goes from 255 to 0).

Reference Manual for FPGA Platform

If you use a board, then you need to get the right reference manual for it. For example, if using a Nexys board (see Prac5a) you need to get the right pin assignments and other configuration information from the reference manual. Digilent Inc. uses useful names for these manuals e.g. “Nexys3_rm_V2.pdf” for the Nexys3 manual.

Nexys3™ Board Reference Manual

Revision: April 10, 2013



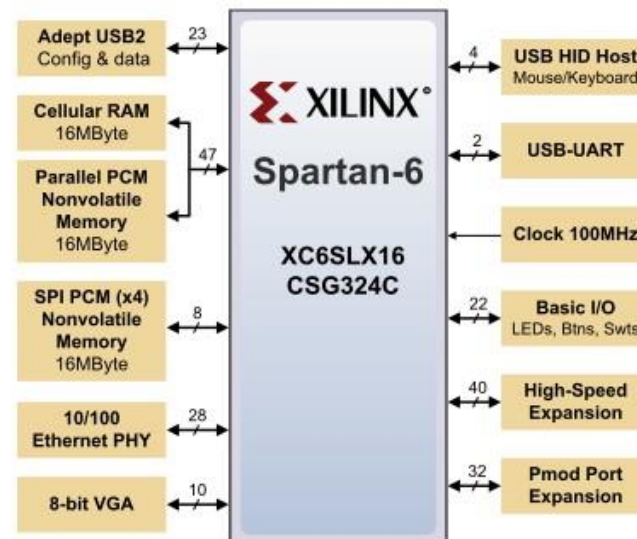
1300 Henley Court | Pullman, WA 99163
(509) 334 6306 Voice and Fax

Overview

The Nexys3 is a complete, ready-to-use digital circuit development platform based on the Xilinx Spartan-6 LX16 FPGA. The Spartan-6 is optimized for high performance logic, and offers more than 50% higher capacity, higher performance, and more resources as compared to the Nexys2's Spartan-3 500E FPGA. Spartan-6 LX16 features include:

- 2,278 slices each containing four 6-input LUTs and eight flip-flops
- 576Kbits of fast block RAM
- two clock tiles (four DCMs & two PLLs)
- 32 DSP slices
- 500MHz+ clock speeds

In addition to the Spartan-6 FPGA, the Nexys3 offers an improved collection of peripherals including 32Mbytes of Micron's latest Phase Change nonvolatile memory, a 10/100 Ethernet PHY, 16Mbytes of Cellular RAM, a USB-UART port, a USB host port for mice and keyboards, and an improved high-

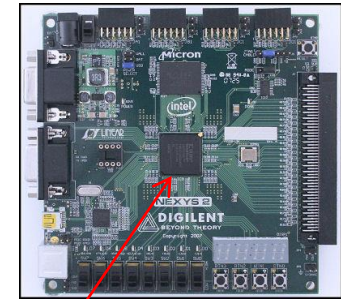
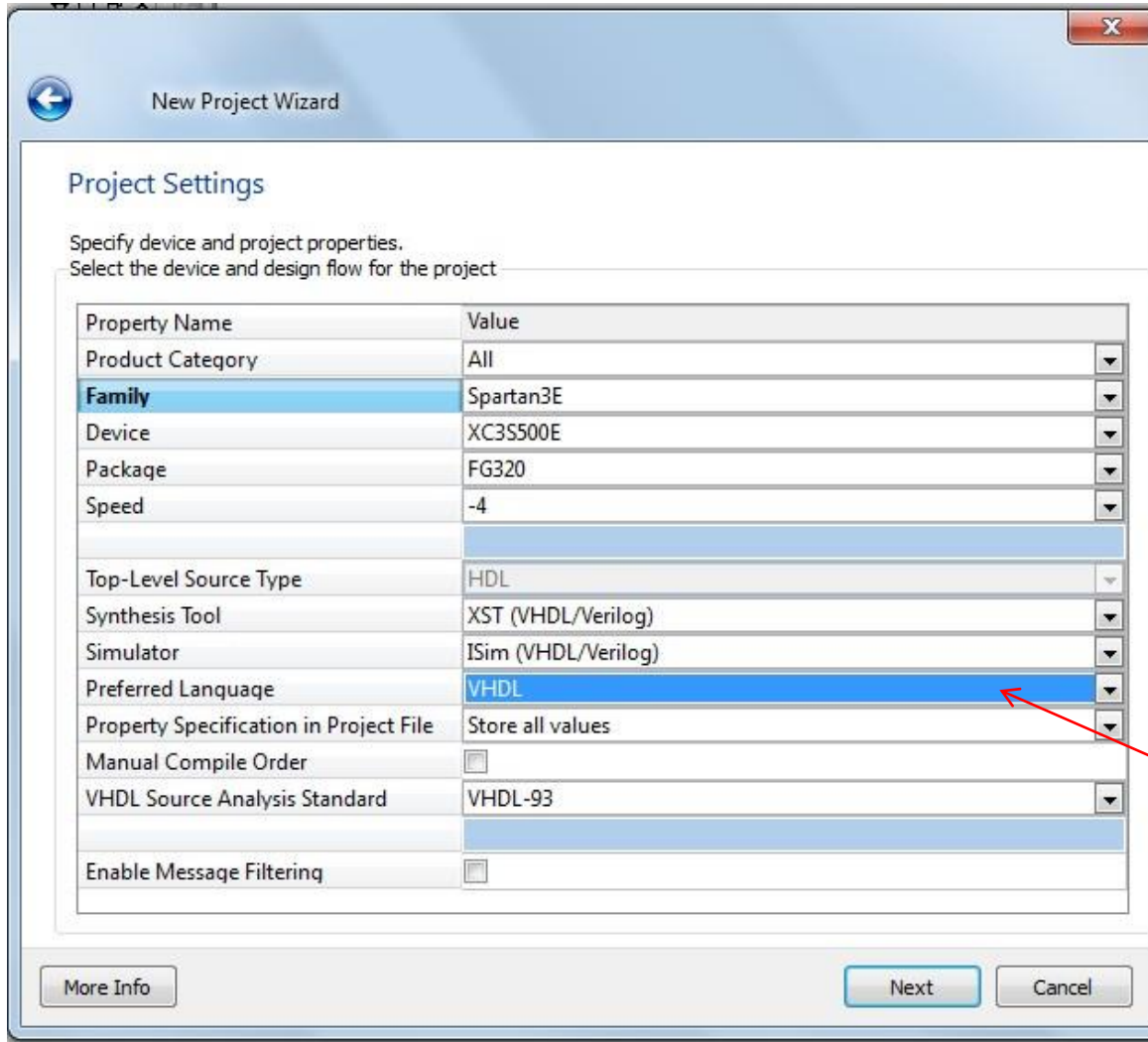


- Xilinx Spartan-6 LX16 FPGA in a 324-pin BGA package
- 16Mbyte Cellular RAM (x16)

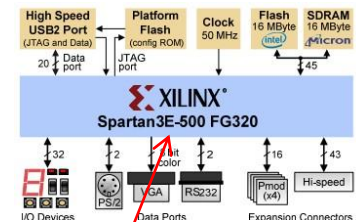
Choose Verilog in ISE/Vivado

Starting with a new project... (can use an existing project also)

Example using a Nexys2



Can read off FPGA device name here

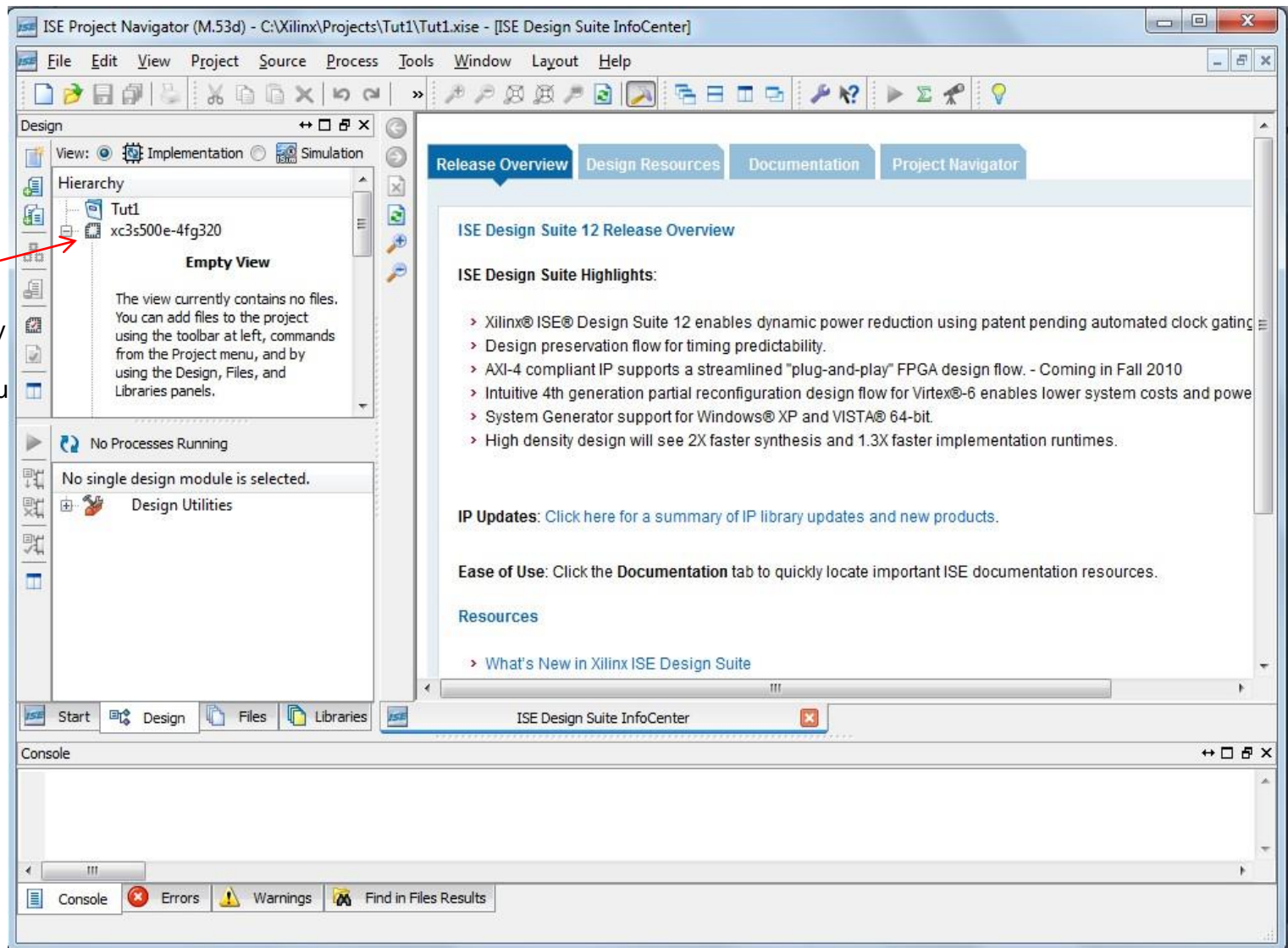


Or pg1 of reference manual should have the device name indicated

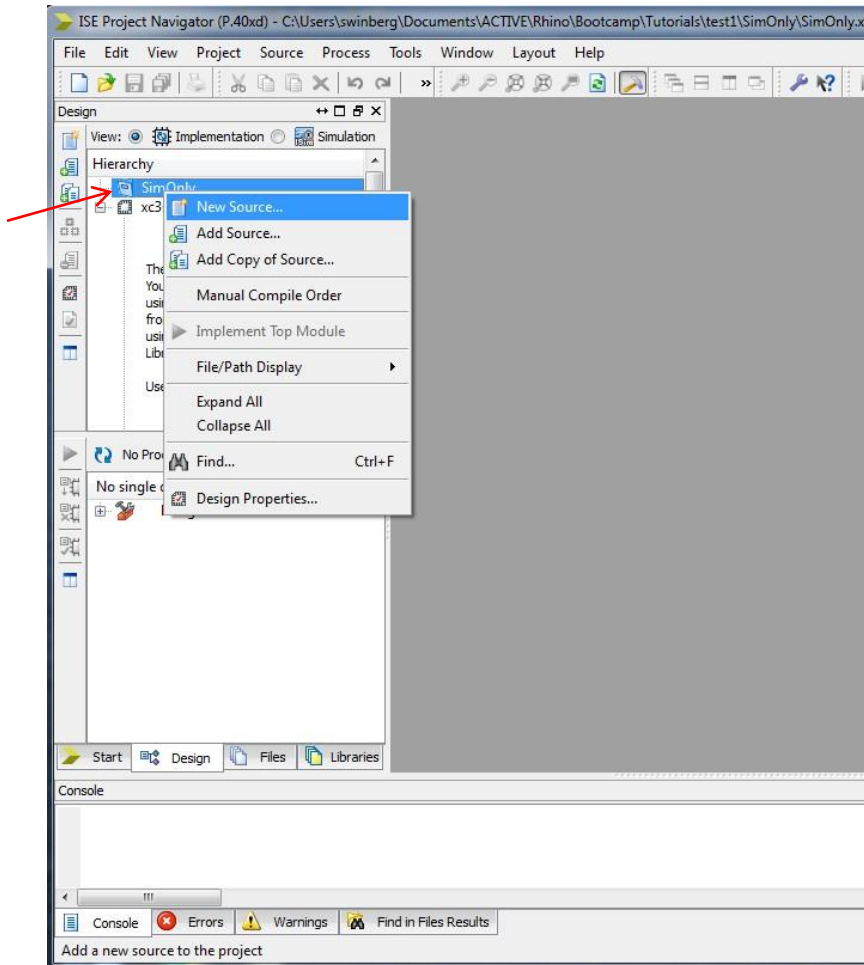
Change to Verilog (optional as you can add in Verilog to a project with preferred language VHDL)

Should get something like this displayed...

Implementation view of hierarchy should confirm which device you are using

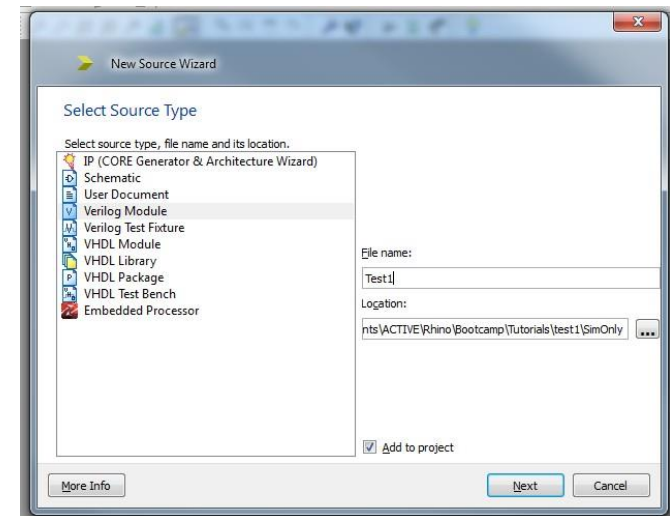


Add a Verilog file...

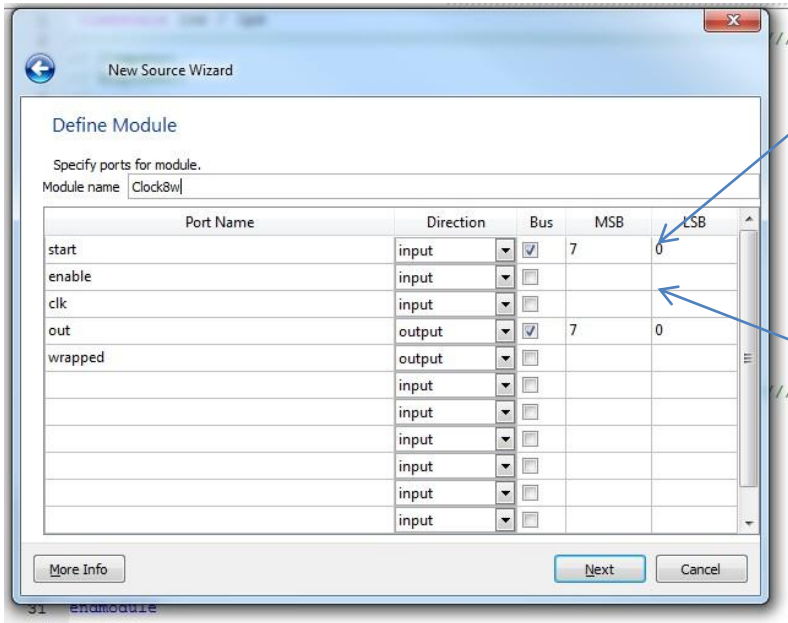


Add in a new file by right-clicking On the project object in the design hierarchy view

Select add a Verilog file



You can use the Define Module form if you want to specify the ports without typing in manually, but you may prefer to skip this and define the ports in the code (especially as this is something that may need to be edited later)



Specifies some busses

Generates a starting file like this



These two are just single bit / wire ports

Simulation configuration setting:
timescale <reference_time>/<precision>
Each 1ns step simulated with 1ps precision *

```

`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
/
// Company:
...
// Design Name:
// Module Name: Count8w
// Project Name:
...
//
/////////////////////////////////////////////////////////////////
/
module Count8w(
    input [7:0] start,
    input enable,
    input clk,
    output [7:0] out,
    output wrapped
);

endmodule
    
```

* Example of timescale use:
`timescale 1ns/1ps
means time scale is 1ns with resolution or least count of 1ps
#1 ; // 1ns delay
#0.001; // 0.001 ns this is the minimum delay at this time scale
#0.0001; // give 0 ns delay!! (not simulating to this fine a resolution)

Expand on the Verilog design...

```
// Additional Comments: counter with start input and wrap detection

module Count8w(
    input [7:0] start,
    input enable,
    input clk,
    output reg [7:0] out, // this one needs to be a register to keep its data
    output reg wrapped    // set to true if gone past start
);
    reg org_start; // save the original start value
    // start an always loop -- is activated for each clk positive edge
    always @(posedge clk)
    if (~enable) begin
        out <= start; // the output is set to start if enable if low
        org_start <= start;
        wrapped <= 0;
    end else begin
        out <= out + 1;
        if (out == start) begin
            wrapped <= 1;
        end
    end
end
endmodule
```

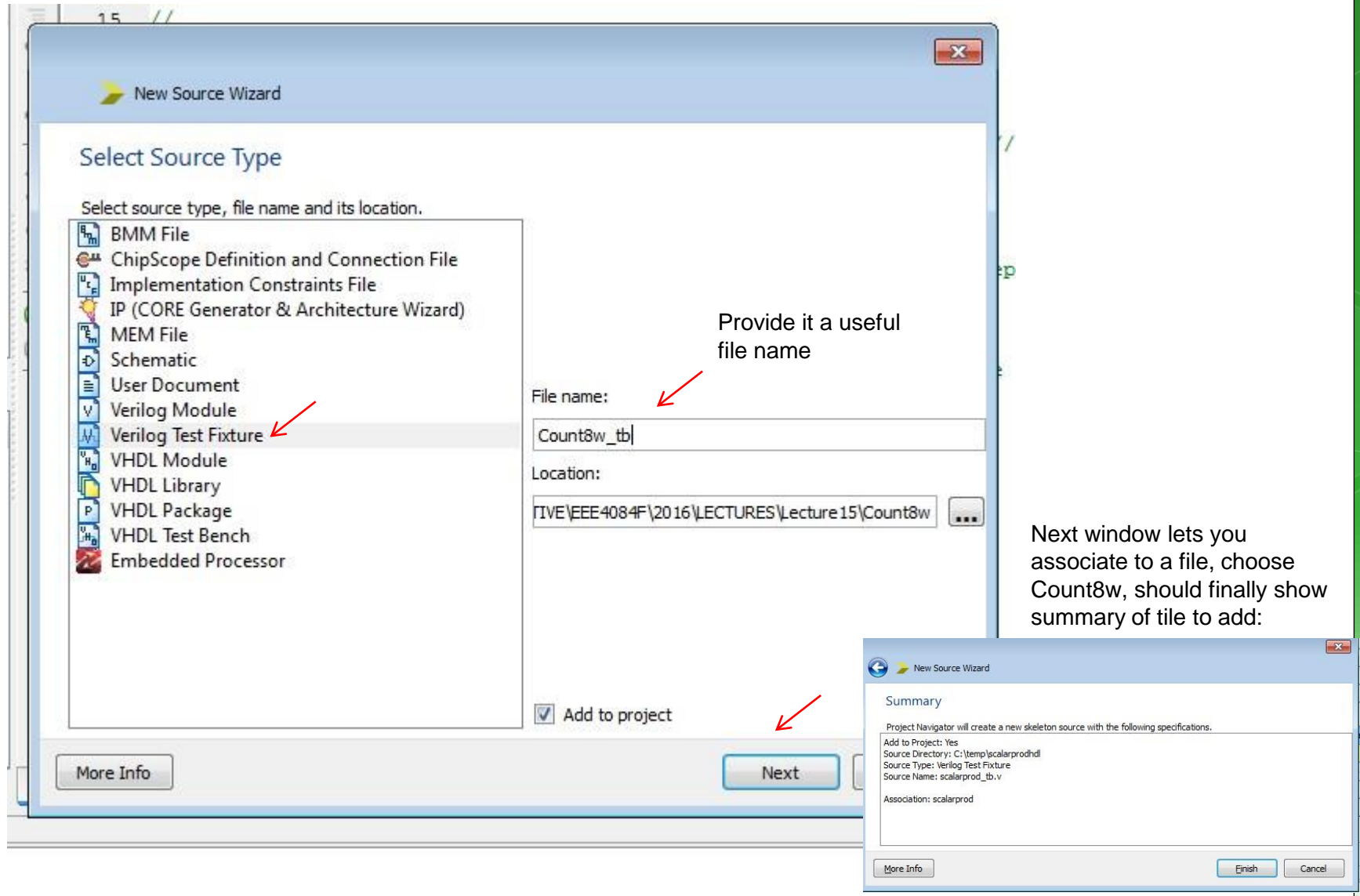
Test Bench

- A Test Bench is a HDL program that verifies the functional correctness of the hardware design.
- The test bench program checks whether the hardware model does what it is supposed to do.
- Used with the simulator, tends to need addition of simulator commands such as using the delay (#n) operation

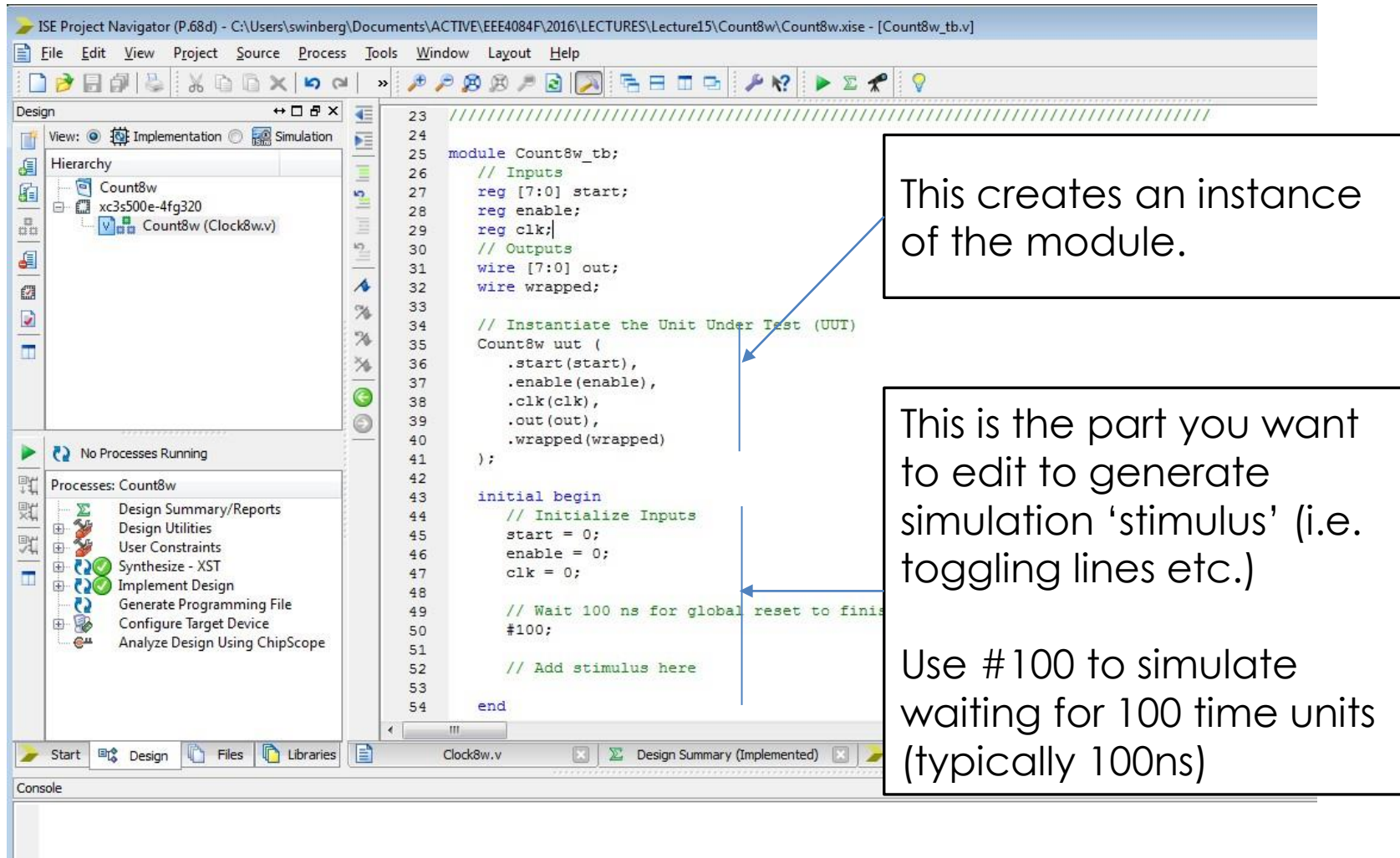
Adding this to the simulator...

Creating a Verilog Test Fixture

.. And put in a suitable name for the resultant file, usually it is followed by `_tb` to show it is a test bench.



This is test bench file generated... And put in a suitable name for the resultant file, usually
Creating a Verilog Test Fixture It is followed by _tb to show it is a test bench.



The screenshot shows the ISE Project Navigator interface. The main window displays the Verilog code for a test bench named `Count8w_tb.v`. The code defines a module `Count8w_tb` with inputs `start`, `enable`, and `clk`, and outputs `out` and `wrapped`. It instantiates the `Count8w` module (UUT) and includes an `initial` block to initialize the inputs and wait for 100 time units before adding stimulus.

```
23 ///////////////////////////////////////////////////////////////////
24 ///////////////////////////////////////////////////////////////////
25 module Count8w_tb;
26     // Inputs
27     reg [7:0] start;
28     reg enable;
29     reg clk;
30     // Outputs
31     wire [7:0] out;
32     wire wrapped;
33
34     // Instantiate the Unit Under Test (UUT)
35     Count8w uut (
36         .start(start),
37         .enable(enable),
38         .clk(clk),
39         .out(out),
40         .wrapped(wrapped)
41     );
42
43     initial begin
44         // Initialize Inputs
45         start = 0;
46         enable = 0;
47         clk = 0;
48
49         // Wait 100 ns for global reset to finish
50         #100;
51
52         // Add stimulus here
53
54     end
```

Annotations in the image point to specific parts of the code:

- A box points to the module instantiation (lines 35-41) with the text: "This creates an instance of the module."
- A box points to the `initial` block (lines 43-53) with the text: "This is the part you want to edit to generate simulation 'stimulus' (i.e. toggling lines etc.)"
- A box points to the `#100` delay (line 50) with the text: "Use #100 to simulate waiting for 100 time units (typically 100ns)"

Example test bench

```
`timescale 1ns / 1ps // simulation precision
...
initial begin
    // Initialize Inputs
    start = 0;
    enable = 0;
    clk = 0;

    // Wait 100 ns for global reset to finish
    #100;

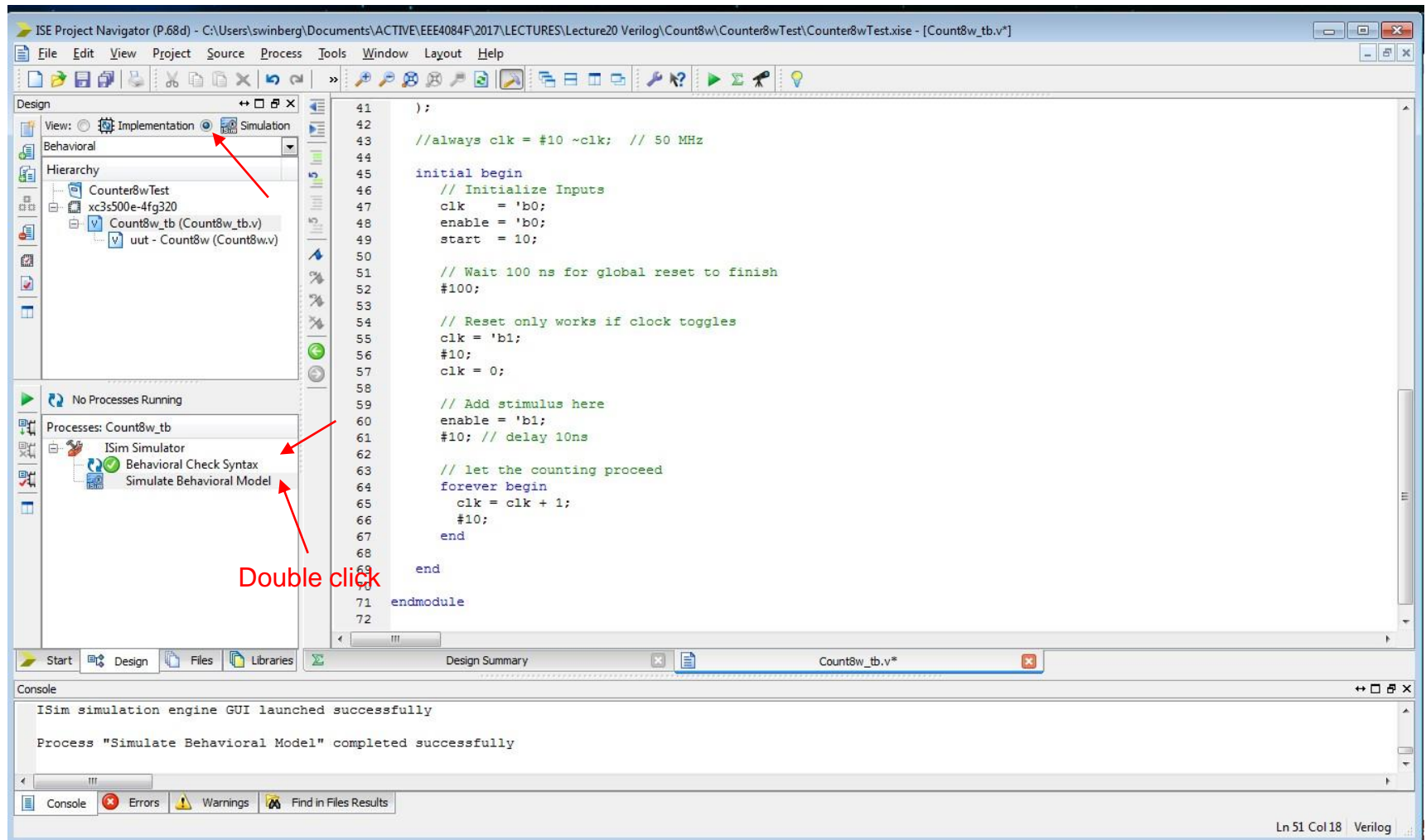
    // Add stimulus here
    start = 'd10;
    enable = 'b0;
    clk = 1;
    #10; // delay 10ns

    // Add stimulus here
    enable = 'b1;
    clk = 0;
    #10; // delay 10ns

end
```

In the Altera simulator the waveform editor allows initial conditions of lines to be set and addition of clock lines, but in test bench code you need to implement this behaviour

Click on Simulator and then double click simulate behavioural model



Testbench can be further refined to simulate behaviour of other inputs and continue the clock for longer (e.g. using always or 'forever begin' simulation commands).

ISim (P.68d) - [Default.wcfg]

File Edit View Simulation Window Layout Help

1.00us Re-launch

Source Files

- Count8w.v
- Count8w_tb.v
- gbl.v

Objects

Simulation Objects for Count8w_tb

Object Name	Value
out[7:0]	00110111
wrapped	0
start[7:0]	00001010
enable	1
clk	1

Name	Value
out[7:0]	00001011
wrapped	0
start[7:0]	00001010
enable	1
clk	1

X1: 120,001 ps

Default.wcfg

Console

Time resolution is 1 ps
 Simulator is doing circuit initialization process.
 Finished circuit initialization process.
ISim>

Compilation Log

Compilation Log Breakpoints Find in Files Results Search Results

Further examples to try using simulators or on the actual hardware:

More <http://www.asic-world.com/examples/verilog> (these also provide examples that can be run using iVerilog)

Generating Verilog from the Schematic Editor

It can occasionally be useful to generate a Verilog code file from an existing schematic, e.g. if you started with a schematic and then wanted to change to using the Verilog code directly.

Starting with a new project... (can use an existing project also)

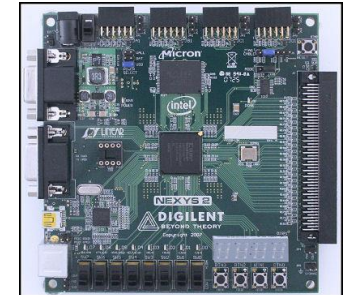
New Project Wizard

Project Settings

Specify device and project properties.
Select the device and design flow for the project

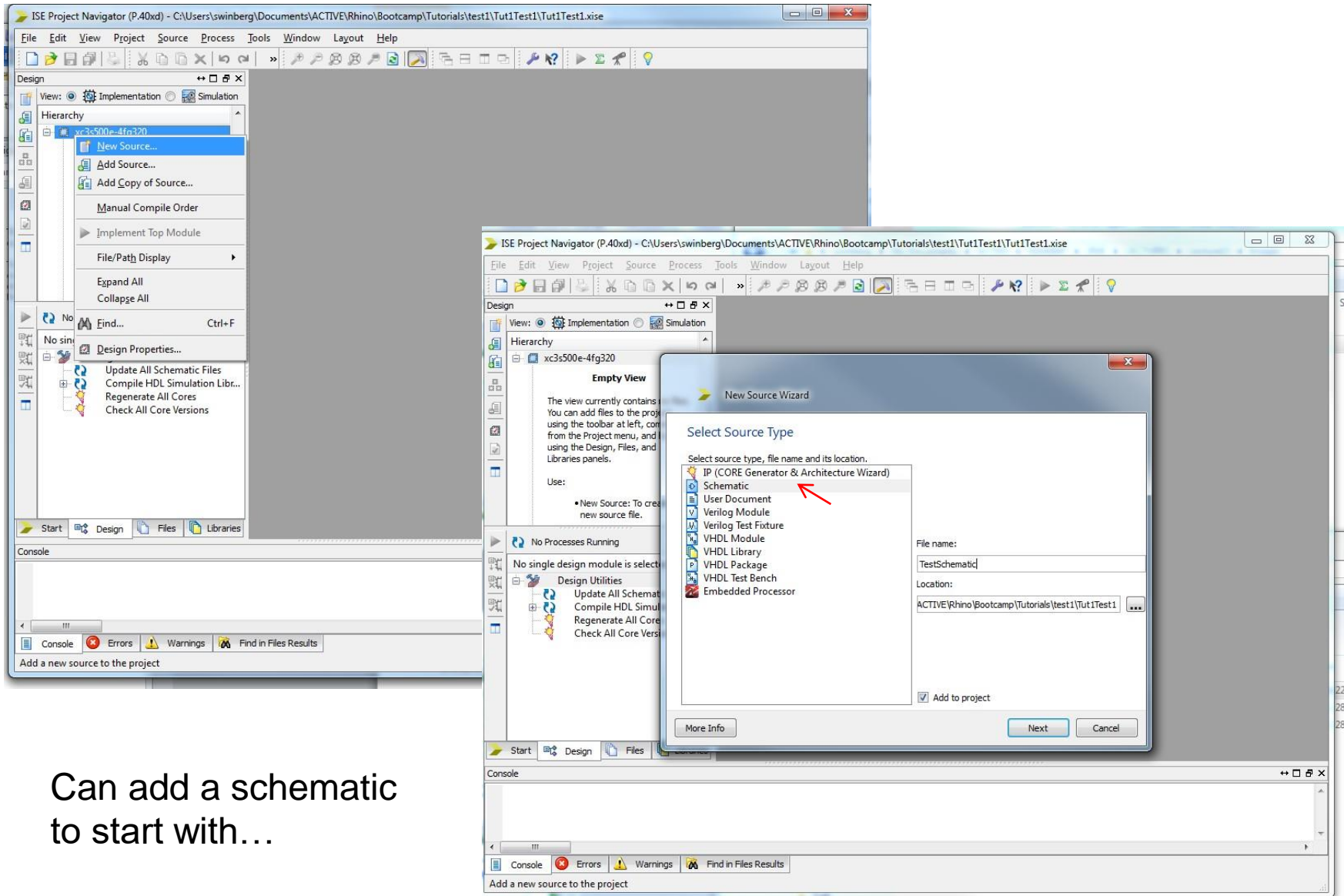
Property Name	Value
Product Category	All
Family	Spartan3E
Device	XC3S500E
Package	FG320
Speed	-4
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	VHDL
Property Specification in Project File	Store all values
Manual Compile Order	<input type="checkbox"/>
VHDL Source Analysis Standard	VHDL-93
Enable Message Filtering	<input type="checkbox"/>

More Info Next Cancel



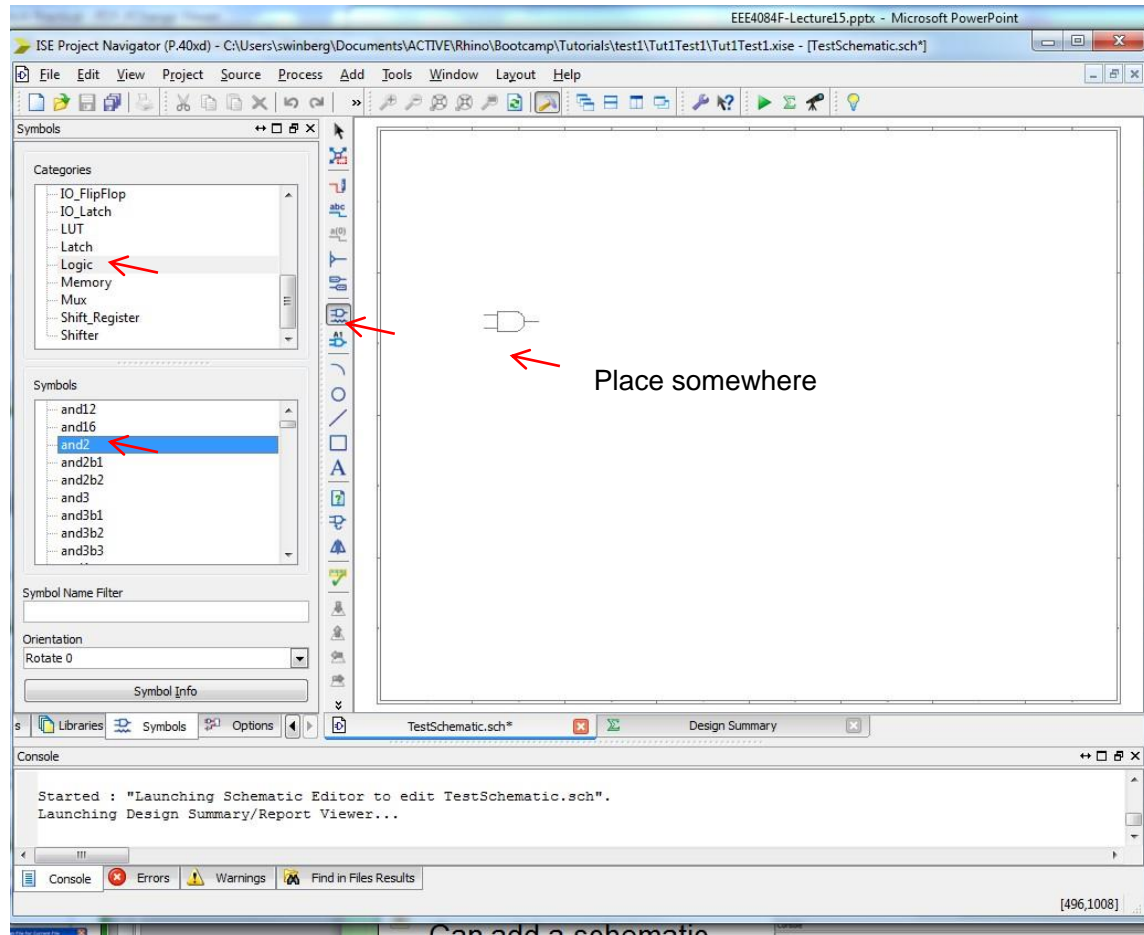
Change to Verilog
(optional as you can add in Verilog to a project with preferred language VHDL)

Click to create a new source file



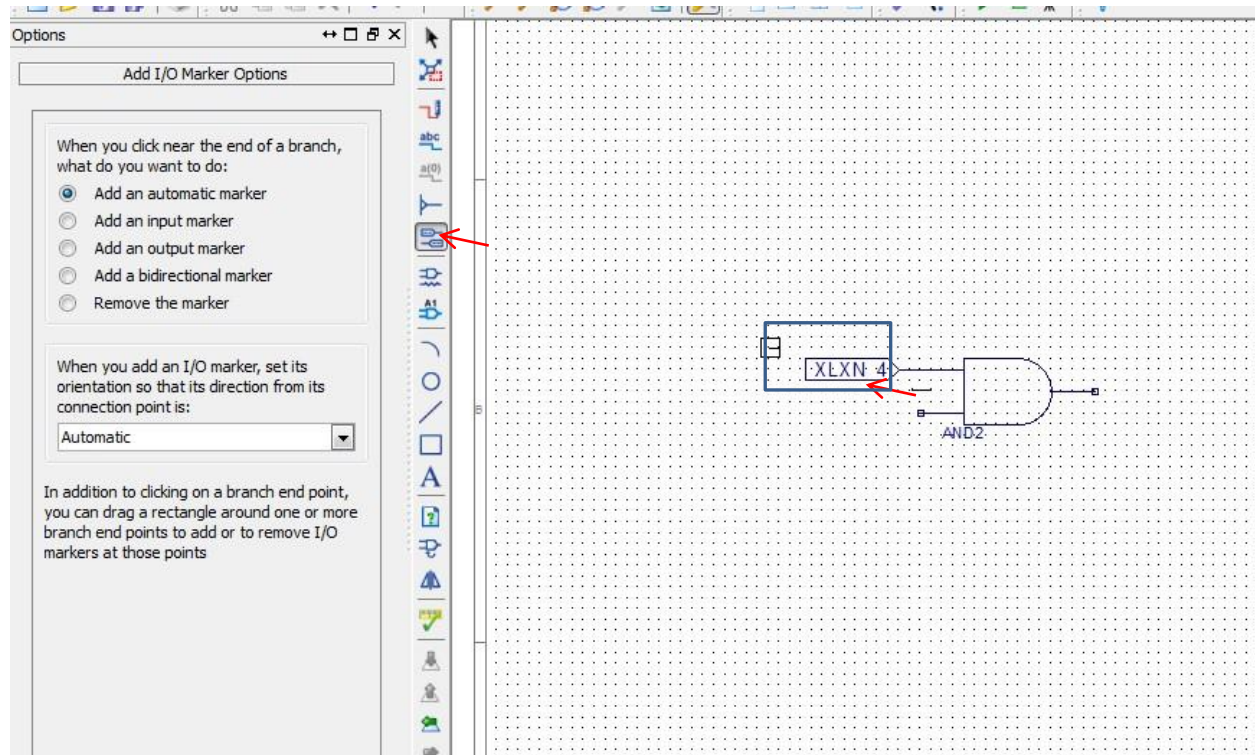
Can add a schematic to start with...

Add a symbol...

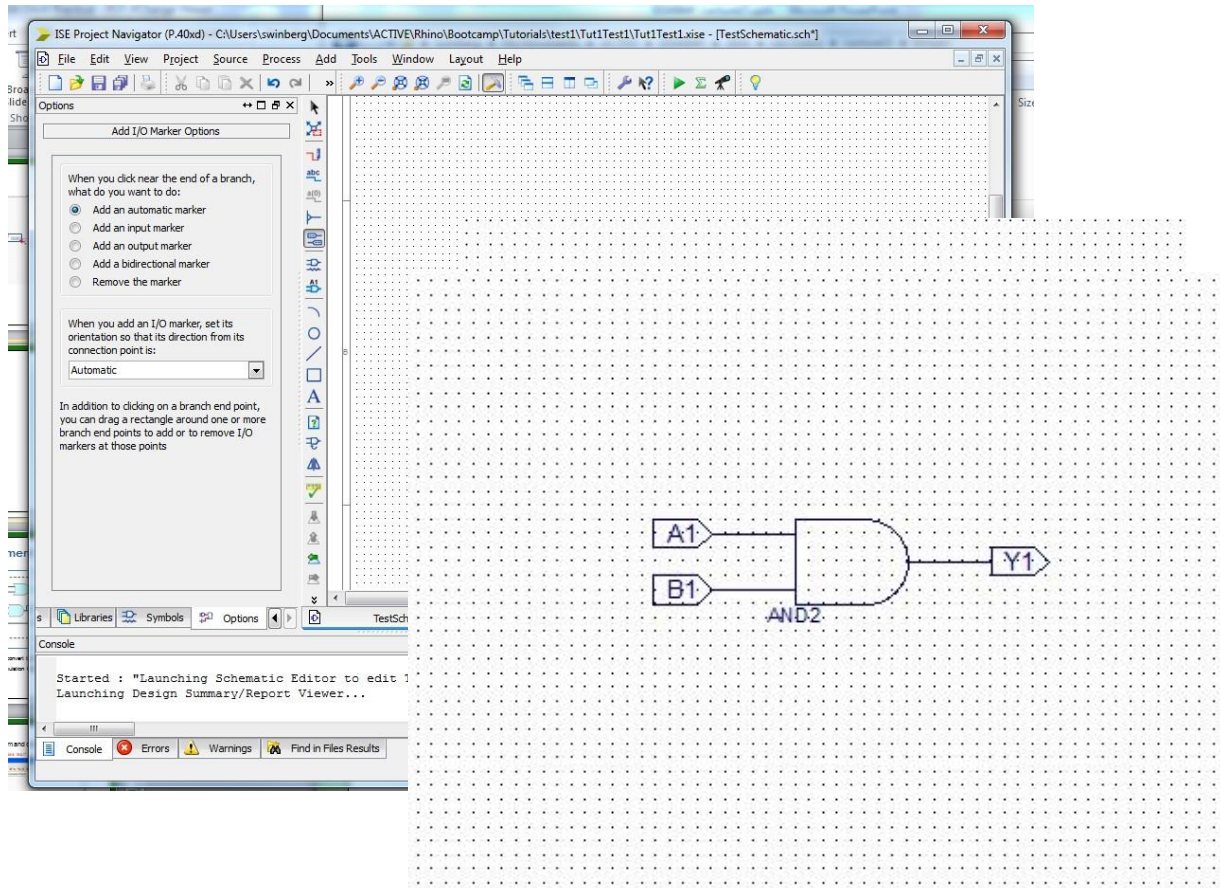


Can add a schematic.

Add an IO marker or three



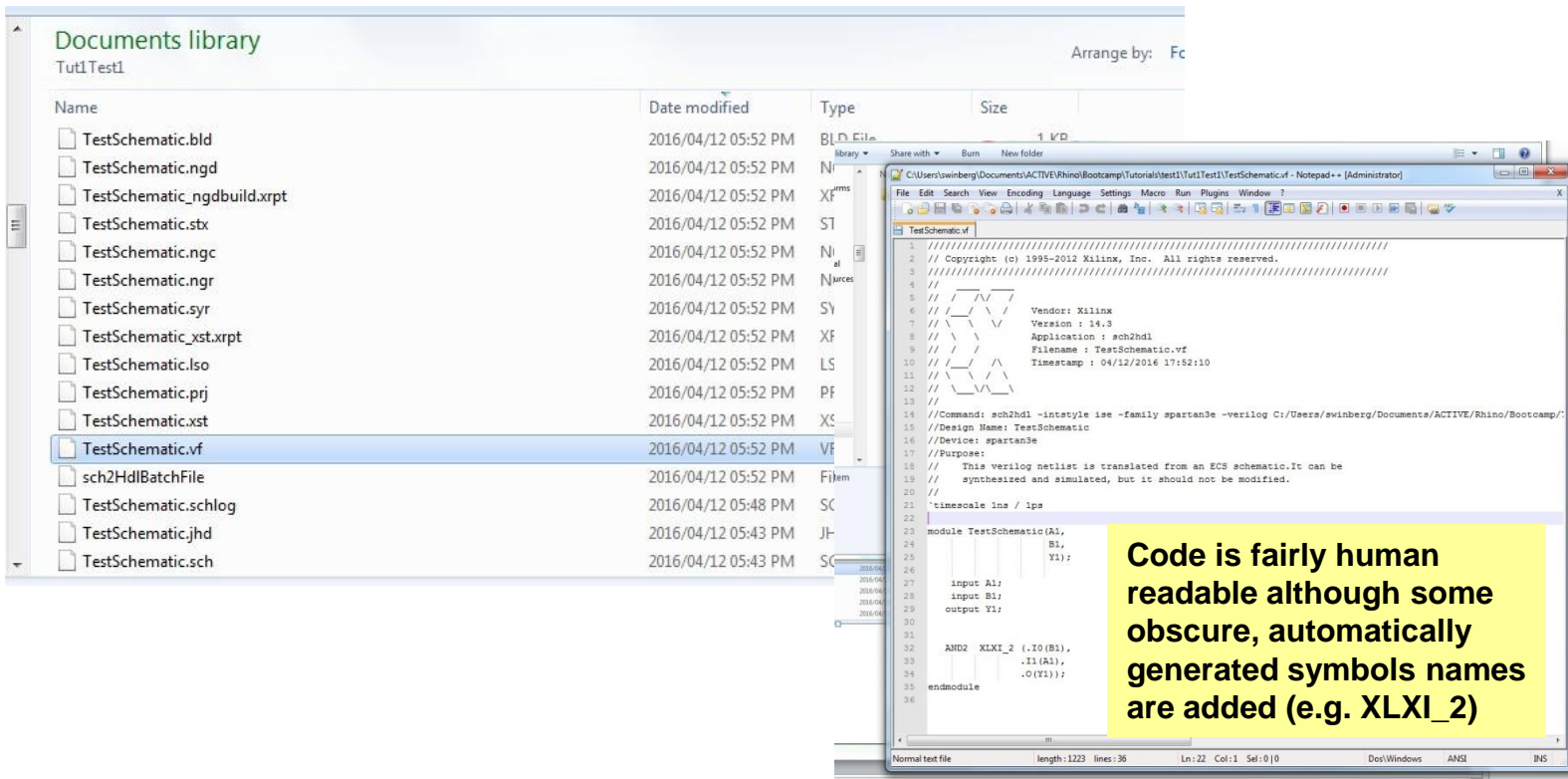
Put in some better names



Generating the Verilog...

You need to run synthesis to generate the Verilog code (only works if you chose Verilog as preferred language)

Once done, look for the VF file with the same name as the schematic file.



The screenshot shows a Windows file explorer window titled 'Documents library' with the path 'Tut1\Test1'. The file list includes various files such as 'TestSchematic.bld', 'TestSchematic.ngd', 'TestSchematic.ngdbuild.xrpt', 'TestSchematic.stx', 'TestSchematic.ngc', 'TestSchematic.ngr', 'TestSchematic.syr', 'TestSchematic.xst.xrpt', 'TestSchematic.iso', 'TestSchematic.prj', 'TestSchematic.xst', 'TestSchematic.vf', 'sch2HdlBatchFile', 'TestSchematic.schlog', 'TestSchematic.jhd', and 'TestSchematic.sch'. The file 'TestSchematic.vf' is selected and highlighted in blue.

An inset window shows the Verilog code generated from the schematic. The code is as follows:

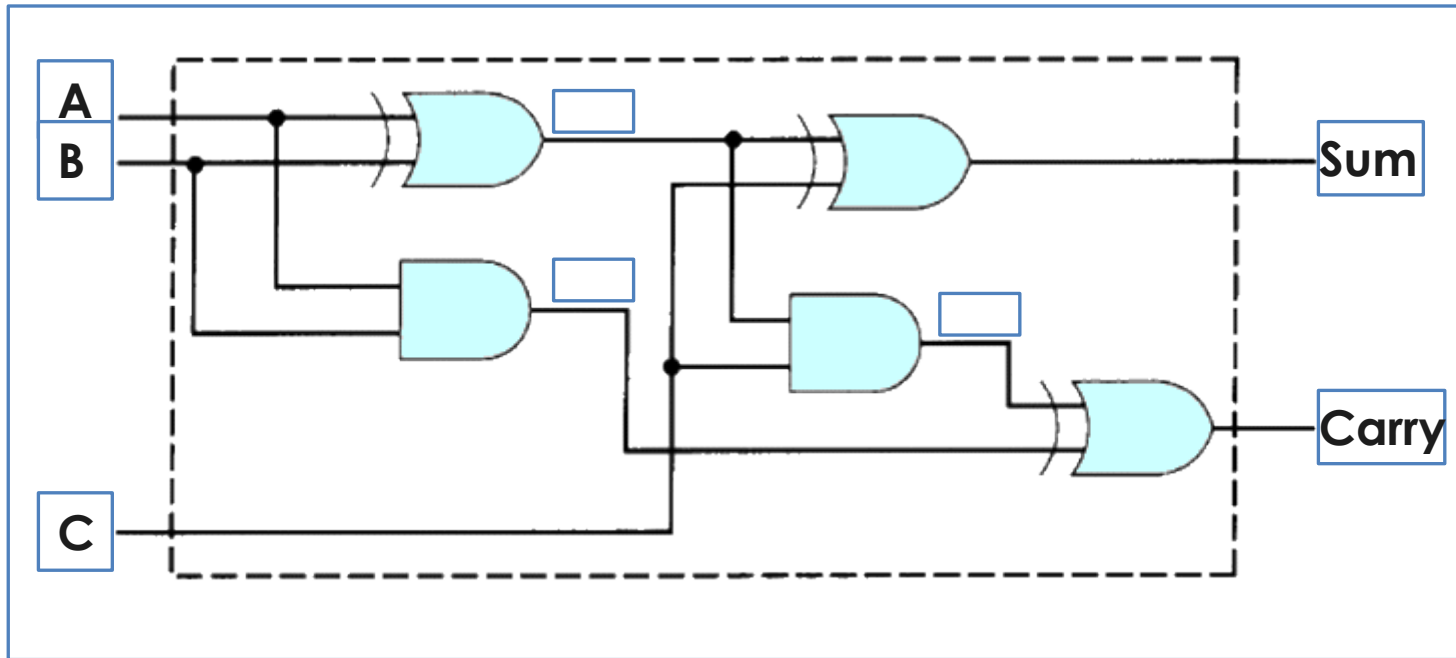
```
1 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2 // Copyright (c) 1998-2012 Xilinx, Inc. All rights reserved.
3 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
4 //
5 //
6 // Vendor: Xilinx
7 // Version : 14.3
8 // Application : sch2hdl
9 // Filename : TestSchematic.vf
10 // Timestamp : 04/12/2016 17:52:10
11 //
12 //
13 //
14 // Command: sch2hdl -intstyle ise -family spartan3e -verilog C:/Users/swinberg/Documents/ACTIVE/Rhino/Bootcamp/
15 // Design Name: TestSchematic
16 // Device: spartan3e
17 // Purpose:
18 // This verilog netlist is translated from an ECS schematic. It can be
19 // synthesized and simulated, but it should not be modified.
20 //
21 timescale 1ns / 1ps
22
23 module TestSchematic(A1,
24 B1,
25 Y1);
26
27 input A1;
28 input B1;
29 output Y1;
30
31
32 AND2 XLXI_2 (.IO(B1),
33 .I1(A1),
34 .O(Y1));
35
36 endmodule
```

Code is fairly human readable although some obscure, automatically generated symbols names are added (e.g. XLXI_2)

Further discussion about Schematic to Verilog: <http://www.edaboard.com/thread217131.html>

and Schematic to VHDL: <http://stackoverflow.com/questions/8968982/how-to-generate-vhdl-code-from-a-schematic-in-xilinx>

Suggested assignment



This is a 4-bit adder design. Try to convert this into Verilog.

Try to run on one or a few of the simulation tools presented in next slides...

Supplement

If you have QuartusII installed already you could of course also use it for experimenting and generating Verilog...

You may find the QuartusII simulator easier to use, which could be a reason to use this tool.

Learning Verilog in QuartusII

One approach is using a block diagram and converting to Verilog HDL.

E.g. using Altera Quartus II

(See test1.zip for example Quartus project)

The screenshot displays the Quartus II interface with the 'Create HDL Design File for Current File' dialog box open. The dialog box contains the following elements:

- File type:** Two radio buttons are present: 'VHDL' (unselected) and 'Verilog HDL' (selected).
- Add VHDL Statements...:** A button is visible next to the VHDL option.
- File name:** The text 'C:/Temp/delme/Testverilog.v' is entered in the field.
- Buttons:** 'OK' and 'Cancel' buttons are at the bottom of the dialog.

The background shows the Quartus II workspace with a block diagram of an AND gate (AND2) and a counter component (lpm_counter). The 'File' menu is open, showing options like 'New...', 'Open...', 'Save', and 'Create / Update'. The 'Create / Update' submenu is also visible, listing various file creation options.

Learning Verilog

One approach is using a block diagram and converting to Verilog HDL.
E.g. using Altera Quartus II

The screenshot displays the Altera Quartus II IDE interface. The main window shows the Verilog code for a testbench named 'Testverilog.v'. The code is as follows:

```
18
19 module Testverilog(
20     Ain, Bin, ClrCnt, Clk, Xout,
21     SumOut
22 );
23 |
24 input  Ain;
25 input  Bin;
26 input  ClrCnt;
27 input  Clk;
28 output Xout;
29 output [7:0] SumOut;
30
31 wire   [7:0] SYNTHESIZED_WIRE_0;
32
33 assign Xout = Ain & Bin;
34
35 altaccumulate0 b2v_inst34
36     .clock(Clk),
37     .data(SYNTHESIZED_WIRE_0),
38     .result(SumOut));
39
40
```

Two red arrows point to specific parts of the code:

- One arrow points to the parameter list in the module definition (lines 19-21), with the text: "See how param types are specified".
- Another arrow points to the instantiation of the 'altaccumulate0' module (line 35), with the text: "See how in QuartusII included modules are instantiated and ports explicitly mapped (Xilinx chose not to do the explicit port mapping when converting from schematic to code)".

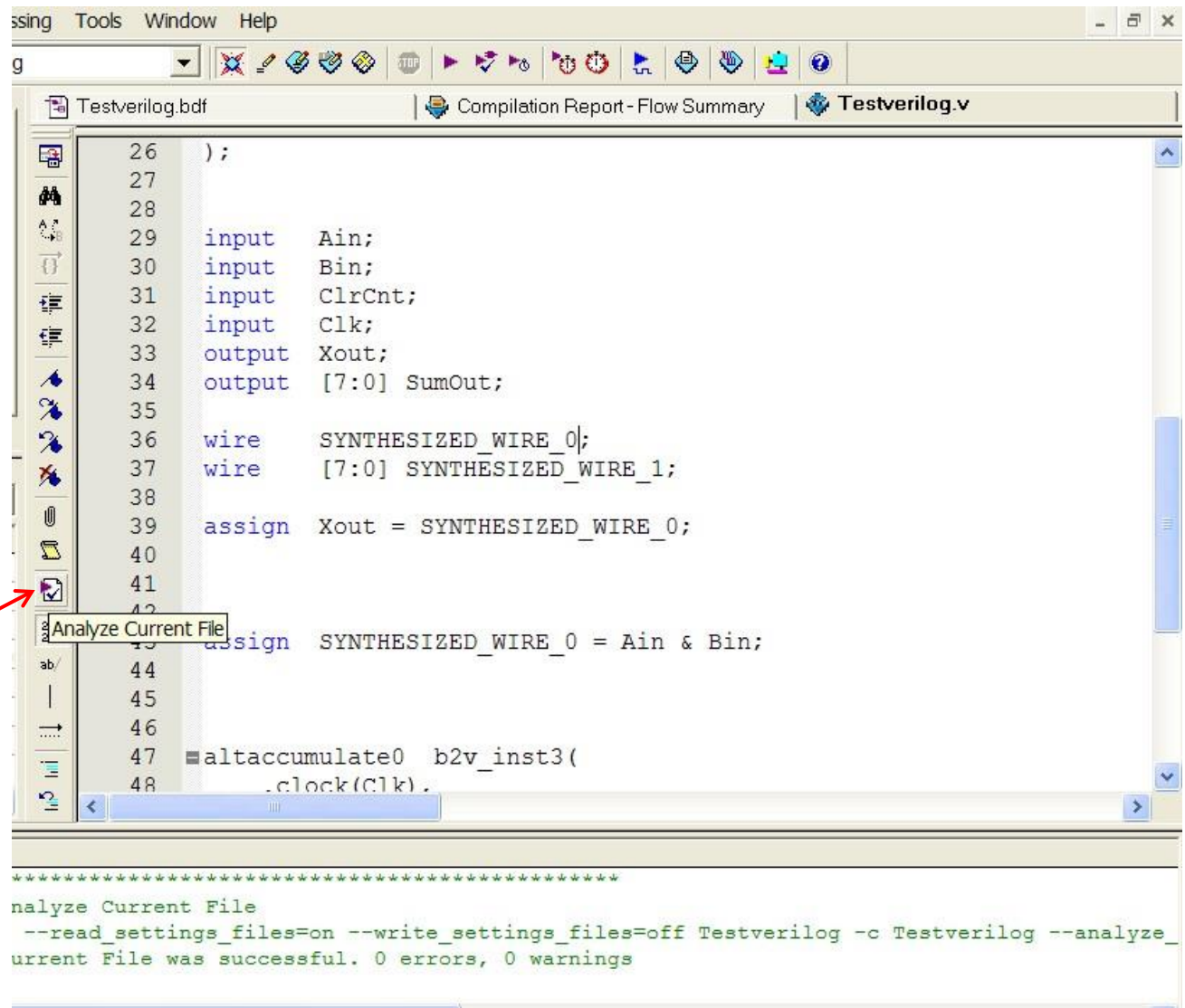
The bottom panel of the IDE shows a message log with the following information:

Type	Message
Info	Running Quartus II Create Verilog File
Info	Command: quartus_map --read_settings_files=on --write_settings_files=off Testverilog -c Testverilog --conver
Info	Found 1 design units, including 1 entities, in source file Testverilog.bdf
Info	Elaborating entity "" for the top level hierarchy
Info	Quartus II Create Verilog File was successful. 0 errors, 0 warnings

At the bottom of the IDE, the status bar shows "Ln 23, Col 1" and "Idle".

Checking syntax

I find a handy tool is the file analyser tool in Quartus II. This can be used to check the syntax of the file without having to go through the whole build process.



Testing

(See test2.zip for example Quartus project that contains only Verilog files and waveform file)

The screenshot displays the Quartus II interface with the following components:

- Project Navigator:** Shows the project hierarchy for 'Cyclone: EP1C12F324C8' and 'mynand.v'.
- Tasks:** A list of tasks including 'Compile Design', 'Analysis & Synthesis', 'Fitter (Place & Route)', 'Assembler (Generate programming file)', 'Classic Timing Analysis', and 'EDA Netlist Writer'. The 'Compile Design' task is currently selected.
- Simulation Waveforms:** A window titled 'Simulation Waveforms' showing a timing diagram. The 'Master Time Bar' is set to 10.925 ns. The diagram shows three signals: 'ain' (B 1), 'bin' (B 0), and 'xout' (B 1). A red arrow points to the simulation control buttons.
- Messages:** A window showing system messages, including 'Info: Option to preserve fewer signal tra...', 'Info: Simulation partitioned into 1 sub-s...', 'Info: Simulation coverage is 100.00 %', 'Info: Number of transitions in simulation', and 'Info: Quartus II Simulator was successful'.

Running the simulation should allow you to verify the design is working as planned (i.e. NANDing)

Load the T
make sure
level Entity

For Help, press F1

Suggested study ideas...

- See **Verilog tutorials** online, e.g.:
 - <http://www.verilogtutorial.info/>
- **Icarus Verilog** – An open-source Verilog compiler and simulator
 - <http://iverilog.icarus.com/>
 - Try **iverilog** on forge.ee
- **Gplcver** – Open-source Verilog interpreter
 - <http://sourceforge.net/projects/gplcver/>
 - Try **cver** on forge.ee
- **Verilator** – An open-source Verilog optimizer and simulator
 - <http://www.veripool.org/wiki/verilator>



Comprehensive list of simulators:

<http://www.asic-world.com/verilog/tools.html>

Icarus Verilog



Probably the easiest free open-source tool available
Excellent for doing quick tests.

Takes very little space (a few megs) & runs pretty fast.

Installed on forge.ee

For Ubuntu or Debian you can install it (if you're linked to the leg server), using: `apt-get install iverilog`

Iverilog parsing the Verilog code and generates an executable the PC can run (called a.out if you don't use the flags to change the output executable file name)

I suggest the following to get to know iverilog... upload mynand.v example to forge.ee, compile it with iverilog. Run it. Try changing the testbest code, put in some more operations

<http://iverilog.icarus.com/>

```
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$
```

More Experimenting

Try **test3** or **mycounter.v** as a more involved program and test
Experiment with using both Altera Quartus II, Icarus Verilog, and Xilinx ISE ISim

The image shows a screenshot of the Quartus II software interface during a simulation. The main window displays a timing diagram for a counter circuit. The diagram shows three signals: 'count_out', 'reset', and 'clk'. The 'clk' signal is a square wave with a period of 10 ns. The 'reset' signal is a single pulse of 10 ns width at 11.775 ns. The 'count_out' signal shows the counter output, which increments from 0 to 1, then 0, then 1, then 2, and finally 3, corresponding to the clock edges. The simulation mode is set to 'Timing'.

Below the timing diagram, a terminal window titled 'Bitwise xterm - forge.tlp - forge.ee.uct.ac.za:22' shows the execution of the Icarus Verilog simulator. The terminal output is as follows:

```
swinberg@forge:~/TestVeri$ iverilog mycounter.v
swinberg@forge:~/TestVeri$ ./a.out
out= x, clk=0, reset=1
out= 0, clk=1, reset=1
out= 0, clk=0, reset=1
out= 0, clk=0, reset=0
out= 1, clk=1, reset=0
out= 1, clk=0, reset=0
out= 2, clk=1, reset=0
out= 2, clk=0, reset=0
swinberg@forge:~/TestVeri$
```

Intro to Xilinx ISE using simulation

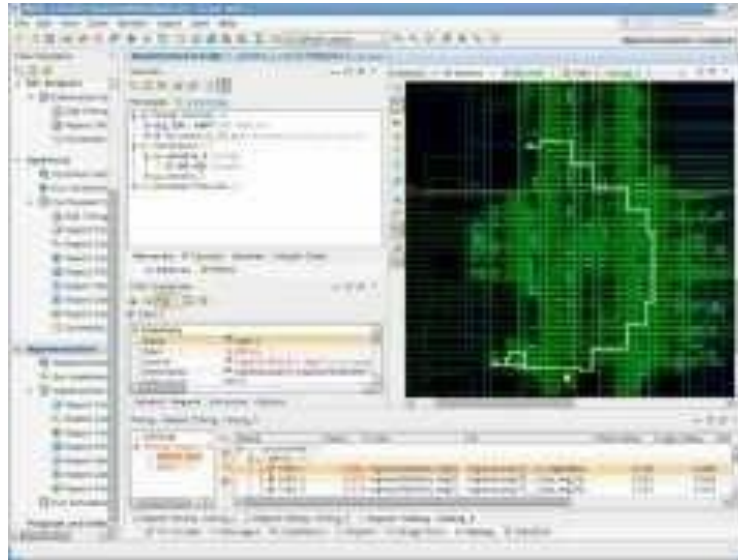


Xilinx ISE Simulation Tutorial.mp4

Short video

<https://www.youtube.com/watch?v=pkJAWpkaiHg>

Intro to Xilinx Vivado



Short video

<http://www.youtube.com/watch?v=H6W4HKbjnaQ>

Disclaimers and copyright/licensing details

I have tried to follow the correct practices concerning copyright and licensing of material, particularly image sources that have been used in this presentation. I have put much effort into trying to make this material open access so that it can be of benefit to others in their teaching and learning practice. Any mistakes or omissions with regards to these issues I will correct when notified. To the best of my understanding the material in these slides can be shared according to the Creative Commons “Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)” license, and that is why I selected that license to apply to this presentation (it’s not because I particularly want my slides referenced but more to acknowledge the sources and generosity of others who have provided free material such as the images I have used).

Image sources:

man working on laptop – flickr

scroll, video reel, big question mark – Pixabay <http://pixabay.com/> (public domain)

References: Verilog code adapted from

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

