# EEE3096S: Embedded Systems II

LECTURE 25:

INSIDES OF A MODULE

**Presented by:**

STANLEY MBEWE

Electrical Engineering
University of Cape Town

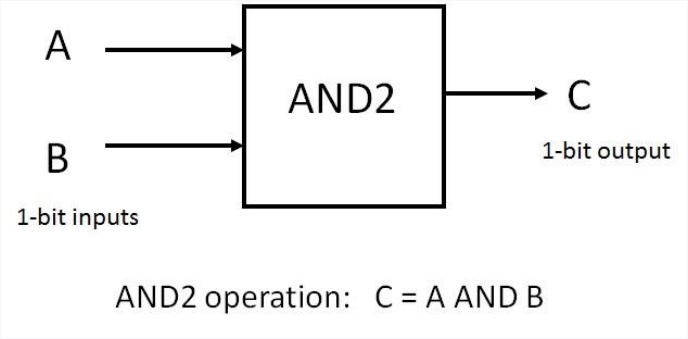
# OUTLINE OF LECTURE

- Recap of the module
- Filenames in Verilog
- The top level module (TLM)
- Module ports
- Instantiating modules
- Recommended coding style
- Basic constructs – always@ and friends
- Verilog programming take-home activity

# MODULE (RECAP)

Last lecture the module syntax was introduced. Here's a recap…

Start off planning the CLB block and its interface…




Then put together the module declaration giving suitable port names

*Verilog95*

```
module AND2(A,B,C);
  input A,B;
  output C;
  assign C = and(A,B);
endmodule
```

Then add the module implementation

*Alternatively:*

*Verilog2001*

```
module AND2(input A,B, output C);
 // start the implementation:
 assign C = A & B;
endmodule
```

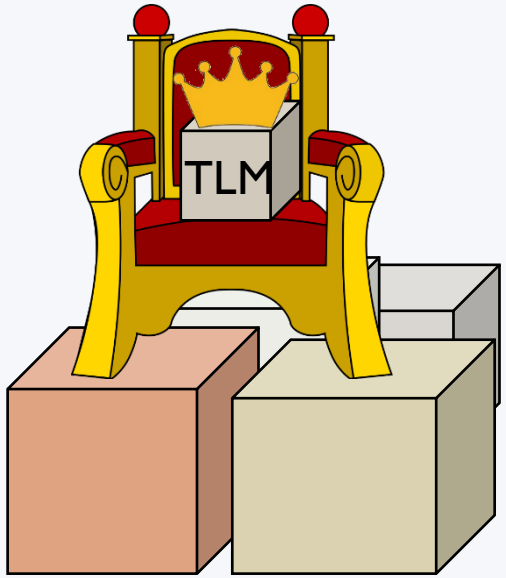
You could add some comments to improve readability

# MODULE FILE NAMES

- Verilog files have a **.v** extension
- Purists say each module should be in it's own .v file named according to the module name (e.g. if you implement a function called myadder you would put it in a file called myadder.v)
- The Verilog build system works much like Pascal.
  - The 'compliler' will read all the *.v file you list.
  - Modules will all be in global space, so you don't need to use things like #include

# TOP LEVEL MODULE

- The top level module (TLM) is the parent of all other modules (also called Top Level Entity if you are using VHDL).
- In the compiler you need to specify the top level module.
- The Verilog compiler will decide what other modules need to be compiled and connected up so that your design can be compiled.



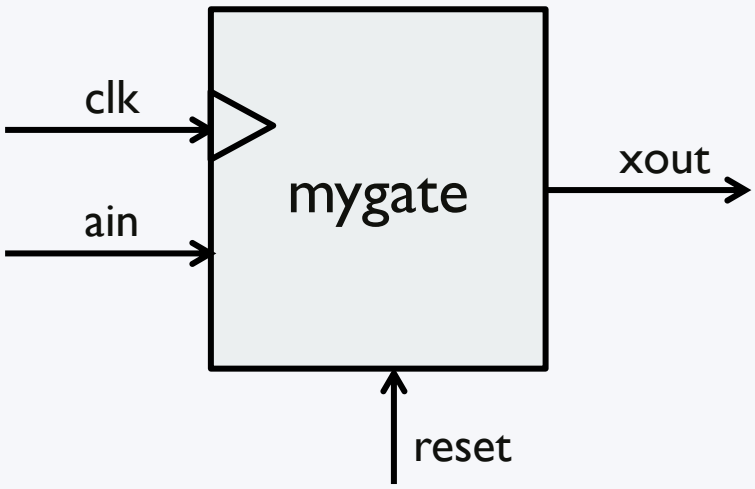
# MODULE PORTS

- The 'parameter list' (in the brackets after module) is more correctly termed the 'port list'
- The tradition is to list input ports first and then output ports. Makes reading of code easier. i.e.:

Syntax:

ModuleName ( <input_ports> <output_ports>);

**Here's an example showing this structure:**




```
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
```

# REGISTERED OUTPUT PORT

- This is an output port that holds its value.
- It is an essential feature needed to construct things like timers and flip flops that need to have some memory

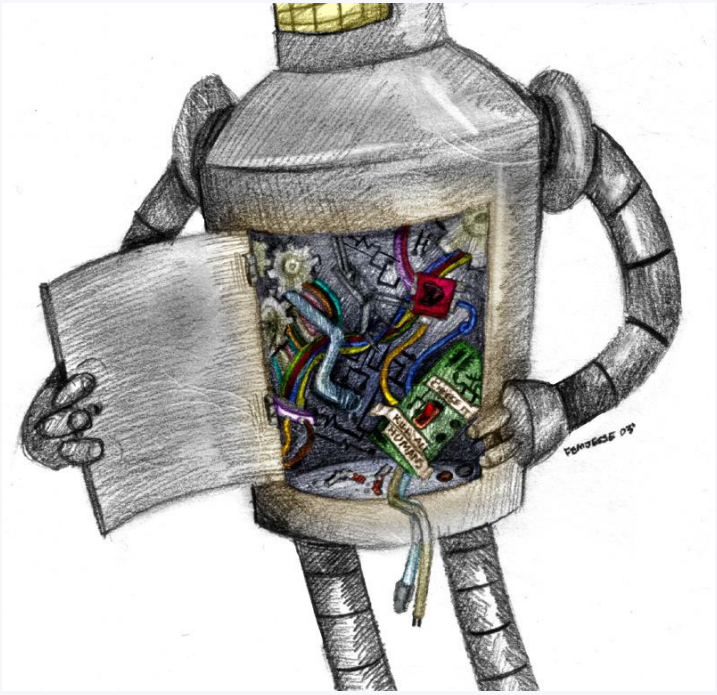
```
module mycounter (
        clk,            // Clock input of the design
        count_out  // 8 bit vector output of the
        );
        // Inputs:
        input clk;
        output [7:0] count_out; // 8-bit counter output
        // All the outputs are registers
        reg [7:0] count_out;
…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

registered output port

# Now you know modules from the outside…

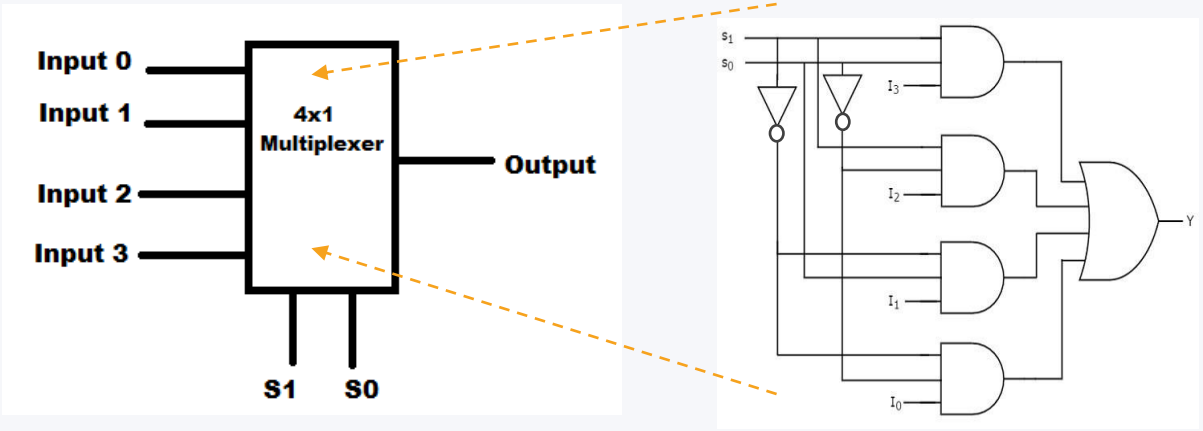
## But

## what do you do about their insides?



# PREVIOUS CLASS/TAKEHOME EXERCISE

Pseudo code thinking exercise… (see EEE3096S-L24-Activity.pdf)
Remember how a multiplexer is made using AND, NOT and OR gates?




I know you don't know the full Verilog code yet, but you know there are registers and wires that connect things (i.e. other modules) up.
Experiment with your own pseudocode on how you might express this circuit as code. You could just use binary expressions and that would likely be close enough.
*We will revisit this next lecture, using actual Verilog.*

# MY SAMPLE SOLUTION IN PSEUDO CODE

```
// Multiplexer implemented using gates only
start module mux2to1
{
  inputs:  a,b,sel.
  output: y.
  wires: sel,asel,bsel,invsel;
  invsel = ~sel;    // not
  asel = a & invsel;   // and these together, save to asel
  bsel = b & sel;  // and these also, save to bsel
  y = asel | bsel;   // or these lines together for output
}
```

Comments/questions:

*Q:* Would you get marks for this in a test if asked for Verilog?

*A:* Maybe some marks for showing some logic and understanding but you'd miss out a fair bit if not proper Verilog and appropriate syntax.

*Why* this approach: I want you to think more about the process and action of converting circuit to text, before diving down into syntax issues.

*Let us now see proper solution in Verilog….*

# INSTANTIATING MODULES

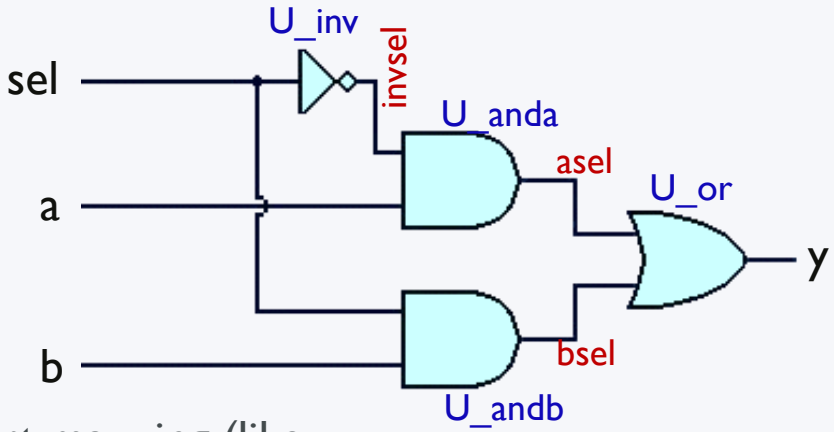
- Modules are built from other modules and/or from built-in modules. These other modules are instantiated in the module you are implementing.
- Syntax for instantiation of a module:

  <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)

* Based on source: http://www.asic-world.com/code/hdl_models/mux_2to1_gates.v

# 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 | xor | logical not * |
|---|---|---|---|---|---|
| Symbol shortcut: | & | \| | ~ | ^ | ! |

| | add | sub | mul | div* | mod* | Unary negation neg |
|---|---|---|---|---|---|---|
| Symbol shortcut: | + | - | * | / | % | *-value* |

## Examples:

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

*Same as:*

```
OUT=IN1&IN2;
OUT=~IN;
```

```
add myadd1  (a,b,c);
negate myneg1  (a,b);
```

*Same as:*

```
a=b+c;
a=-b;
```

* These operators are not always implemented you may need to import libraries.

Bit shifting and bit reordering we will discuss in a later lecture...

# 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)

Source: Coram: Verilog-A Introduction for Compact Modelers (MOS-AK Montreux 2006)

# Basic Constructs

always@ and friends

# THE ALWAYS@

- The always@ expression is used to signal that the next block will contain procedural code
- Syntax:

```
always @ (<sensitivity list>)
  <statement>;
```

OR:

```
always @ (<sensitivity list>)
 begin
  <statement>;
  <statement>;
  …
 end
```

Example:

```
always @ (posedge clk)
   count <= count + 1;
```

# ASSIGNMENT STATEMENT

- Consider a and b are register, and a=1 initially
- Blocking assignment (in an *always* block)

```
assign a = a + 1;
assign b = a + 1;
```

After this statement a will be 2

After this statement b will be 3

These are done sequentially

- Non-backing (or parallel) assignment

```
assign a <= a + 1;
assign b <= a + 1;
```

Both these statements will happen at the same time. The value of a register will change only *after* the operation completes. Therefore:
At the end of both instructions a will be 2 and b will also be 2.

# IF AND ELSE IF STATEMENT

- The if statement is a procedure statement to be used in an always block

```
always @ (*)
 if (condition)
    begin
     dosomething;
    end
end   // end if
```

```
always @ (*)
 if (condition)
    begin
     dosomething;
    end
 else if (condition)
    begin
     dosomethingelse;
    end
 else begin
  doanotherthing;
 end
```

Example:

```
always @ (posedge clk)
if (reset == 1'b1) begin
  onled <= 0;
end else begin
  onled <= 1;
end
```

# CODE EXAMPLE : MUX

*An example of good coding style ...*

```
//------------------------------------------------
// 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 Connections---------
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

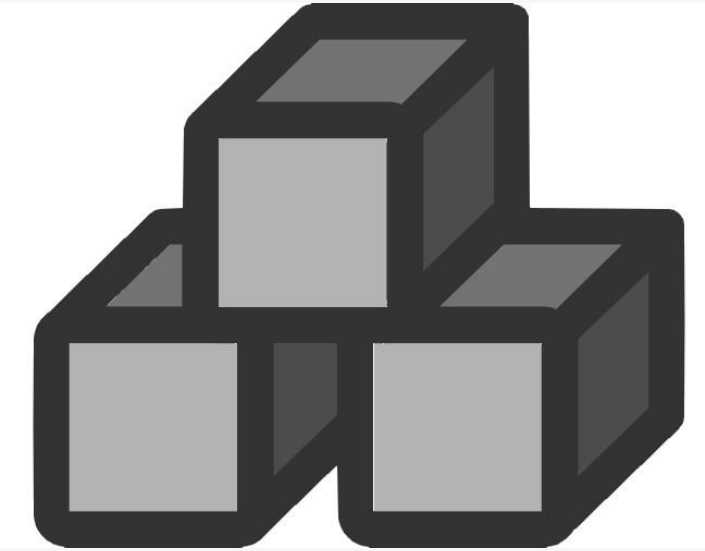
For older versions of Verilog (before 2001) you need to put datatypes after the declaration, but I strongly encourage you to clearly separate input and output (and bidir) ports clearly

I like a comment to clearly indicate the end of the module implementation

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)

# Building a module is all very well…



But how do we know that it will work, will do what we expect it to do, that it meets the specifications?

…

# That's where the Testbench is used…

# HDL TESTBENCH

- In the HDL nomenclature a testbench refers to code that will exercise and test the ‘module under test’ i.e. the module you have implemented.
- The values you set to the ports to do the testing are called test vectors or testing sequences.
- A testbench is basically another Verilog module that will implement the module under test. But it might be a non-synthesizable module that works only within a simulator.
- By all means you can also have a testbench that will run on the physical FPGA platform, but most are used in simulation
- Can have many different testbenchs, e.g.

  Test a variety of test vectors; Test different clock rates (speeds); Test exceptions; etc.

# TOWARDS A TESTBENCH EXAMPLE…

# That's where the Testbench is used…

Verilog 4-bit counter with testbench

The test bench we will only get to later once we have something to test!

# Homework / Thinking Point

- Next lecture we will focus on developing a 4-bit counter.
- Read over the description on the next slide and thinking about how you would develop a Verilog module to implement this
- Next lecture we will finish off this activity in class

**A 4-bit counter!**

# COUNTER DESIGN

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

Let's think about what a 4-bit counter needs and how to implement it…

(1) A module

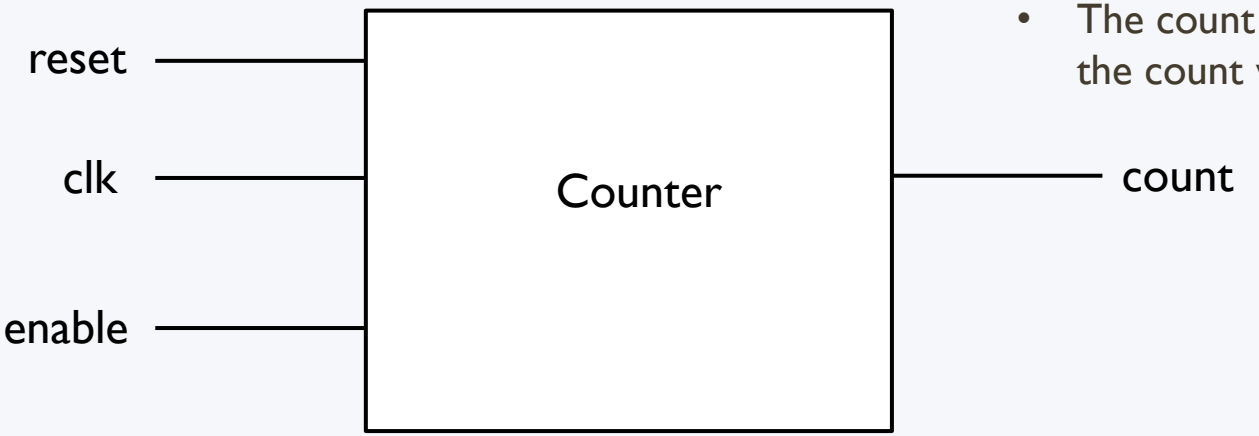
(2) interfaces: some inputs… reset and clock

(3) interfaces: an output… count value

(4) Maybe further embellishments … like enable line

Counter Operation:

- When the reset line is high the count value is set to 0
- If enable is high and there is a positive edge on clk then the count increments
- The count output gives the count value.




OK, that sounds like enough for now.. Now it's over to you to think how to do it in Verilog.

# TODO

**Complete the design of the counter module and try to implement it in Verilog for next lecture.**

Think of the test vectors to write to the ports and what outputs you should get.

(it's only about 5-10 minutes of pen-on-paper effort)

Next lecture you can see a sample solution and then we will explore simulation commands (which you need to make the simulator show results) and to implement a testbench.

Further suggestions:

Think about the testbench interface (is one actually needed for a testbench?); How would you instantiate the counter in the testbench?