Verilog - Modules

The module is the basic unit of hierarchy in Verilog

- Modules describe:
  - boundaries [module, endmodule]
  - inputs and outputs [ports]
  - how it works [behavioral or RTL code]
- Can be a single element or collection of lower level modules
- Module can describe a hierarchical design (a module of modules)
- A module should be contained within one file
- Module name should match the file name
- Module fadder resides in file named fadder.sv
- Multiple modules can reside within one file (not recommended)
- Correct partitioning a design into modules is critical

Verilog - Modules (cont.)

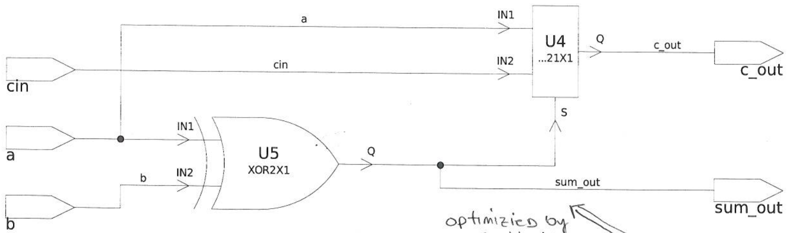
A sample module

```
//-----------------------------------------------------
//one-bit full adder module
//-----------------------------------------------------
module fadder(
  input a,          //data in a
  input b,          //data in b
  input cin,        //carry in
  output sum_out,   //sum output
  output c_out      //carry output
  );
  wire c1, c2, c3; //wiring needed
  assign sum_out = a ^ b ^ cin; //half adder (XOR gate)
  assign c1      = a * cin;     //carry condition 1
  assign c2      = b * cin;     //carry condition 1
  assign c3      = a * b;       //carry condition 1
  assign c_out   = (c1 + c2 + c3);
endmodule
```

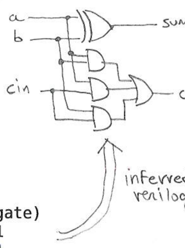
What kind of logic do you expect synthesis to produce? This is always a good question to ask.

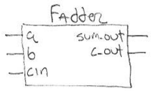
Verilog - Modules (cont.)

When fadder.sv is synthesized by dc_shell, we get...



optimizied by synthesis





```
module fadder(
  input a,         //data in a
  input b,         //data in b
  input cin,       //carry in
  output sum_out,  //sum output
  output c_out     //carry output
  );
  wire c1, c2, c3; //wiring needed

  assign sum_out = a ^ b;    //half adder (XOR gate)
  assign c1      = a * cin;  //carry condition 1
  assign c2      = b * cin;  //carry condition 1
  assign c3      = a * b;    //carry condition 1
  assign c_out   = (c1 + c2 + c3);
```

Verilog - Modules (cont.)

Two brief digressions...wire and assign

- "wire"
  - The declaration "wire" simply is what you think it is
  - A wire carries a value. It has no memory or sense of state.
  - More later about this....
- "assign"
  - The assign statements may execute in any order
  - We can consider that they are executed simultaneously
  - This is how the logic would operate

Verilog - Modules (cont.)

Modules are declared with their name and ports

```
module fadder(
  input  a,         //data in a
  input  b,         //data in b
  input  cin,       //carry in
  output sum_out,   //sum output
  output c_out      //carry output
  );
```

Verilog - Modules (cont.)

Some Lexical Conventions - Comments

- Comments are signified the same as C
- One line comments begin with "//"
- Multi-line comments start: /*, end: */

Some Lexical Conventions - Identifiers

- Identifiers are names given to objects so that they may be referenced
- They start with alphabetic chars or underscore
- They cannot start with a number or dollar sign
- All identifiers are case sensitive

Verilog - Modules (cont.)

ANSI C Style Ports Ports are declared as

- input //simple input
- output //simple output
- inout //tri-state inout

Input port

- Always understood as a net within the module
- Can only be read from [check this against SVerilog]

Output port

- Understood as a net by default
- Can be type reg if assigned in a procedural block (more later)

Inout port

- Understood as a net by default
- Is always a net

Verilog - Modules (cont.)

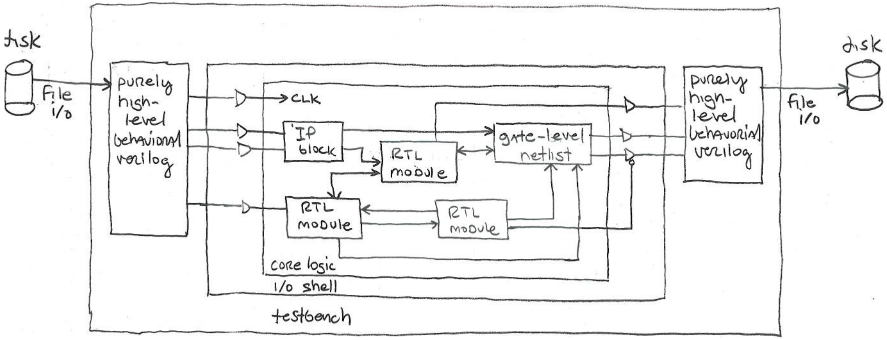
Testbench Modules

- To test a chip at the top hierarchical level, we use a testbench
- Testbench encompasses the chip at the top level
- Testbench has *no ports*,...but probably some file i/o

Verilog - Modules (cont.)

Module to Module Connections

- A hierarchical design has a top level module and lower level ones
- Lower level modules are instantiated within the higher level module
- Lower level modules are connected together with wires



Verilog - Modules (cont.)

Lower level modules can be instantiated within the upper one:

```
//-------------------------------------------------------------------
//one bit full adder - using builtin verilog gates
//-------------------------------------------------------------------
module fadder_struct(
  input a,          //data in a
  input b,          //data in b
  input cin,        //carry in
  output sum_out,   //sum output
  output c_out      //carry output
  );
  wire c1, c2, c3, psum; //wiring needed

  xor   xor_0   (a, b, psum);          //half-adder partial output
  xor   xor_1   (psum, cin, sum_out); //half-adder output
  and   and_0   (a, cin, c1);          //carry condition 1
  and   and_1   (b, cin, c2);          //carry condition 2
  and   and_2   (a, b, c3);            //carry condition 3
  or     or_0   (c1, c2, c3, c_out);  //form carry output
endmodule
```

Verilog - Modules (cont.)

Module fadder constructed with built-in Verilog gates (draw picture)

Verilog - Modules (cont.)

Momentary digression on Verilog primitive gates....

- A number of "virtual" gates are built-in to the language
  - and, or, nand, nor, xor, xnor, tristate, pullup and pulldown
- Each gate has one output, the right most argument
- Gates have as many inputs as required
- These gates are models only, no physical part exists
- Good for experimentation in absence of a cell library

Verilog - Modules (cont.)

Back to instantiation.... Typically, complex modules are connected under the top level

```
 //module instantiations
   shift_reg shift_reg_1(
     .clk           (clk_50),
     .reset_n       (reset_n),
     .data_ena      (data_ena),
     .serial_data   (serial_data),
     .parallel_data (shift_reg_out));

   ctrl_50mhz ctrl_50mhz_0(
     .clk_50   (clk_50),
     .reset_n  (reset_n),
     .data_ena (data_ena),
     .a5_or_c3 (a5_or_c3),
     .wr_fifo  (wr_fifo));

%[draw partial block diagram showing connections]
```

Verilog - Modules (cont.)

Dissection of shift_reg instantiation

```
 shift_reg shift_reg_1(     //module name and instance of the module
 .clk           (clk_50),   //the pin clk is connected to the wire clk_50
 .reset_n       (reset_n),  //the "." denotes the pin
 .data_ena      (data_ena), //the value in the parens is the wire
 .serial_data   (serial_data),
 .parallel_data (shift_reg_out));  //wires and pins do not have to match
[show with arrows and such the various parts]
```

This method of declaring pins/wires is called "named association".
This is the preferred way of doing instantiation when pins/names differ.
Comment each wire connection if its function is not obvious.

Verilog - Modules (cont.)

Its possible to connect wires to pins by position alone
Called "positional association"... just say no!

```
shift_reg shift_reg_1(clk_50,reset_n,data_ena,serial_data,shift_reg_out);
```

How do you know its connected correctly?
What if the module had 50 pins on it?
What if you wanted to add wire in the middle of the list?

Do not use Positional Association!
It saves time once, and costs you dearly afterwords

Verilog - Modules (cont.)

System Verilog instantiation short cuts
These are useful when most all pins/wires have the same name

```
shift_reg shift_reg_1(
  .clk          (clk_50       ),
  .reset_n      (reset_n      ),
  .data_ena     (data_ena     ),
  .serial_data  (serial_data  ),
  .parallel_data (shift_reg_out));
```

Implicit naming can shorten this to:

```
shift_reg shift_reg_1(
  .clk,                          //implicit .name port connection
  .reset_n,                      //implicit .name port connection
  .data_ena,                     //implicit .name port connection
  .serial_data,                  //implicit .name port connection
  .parallel_data (shift_reg_out)); //named association, pin/wire differ
```

"implicit name" and "named association" form can be mixed as required

Verilog - Modules (cont.)

Another instantiation short cut

```
ctrl_50mhz ctrl_50mhz_0(
  .clk_50   (clk_50  ),
  .reset_n  (reset_n ),
  .data_ena (data_ena),
  .a5_or_c3 (a5_or_c3),
  .wr_fifo  (wr_fifo));
```

All pins and wires have the same name, so...

```
ctrl_50mhz ctrl_50mhz_0(.*);   //wow, that's a short cut!
```

This is the most powerful form of implicit association.
Its best reserved for use at the top level testbench/top module.
Otherwise, it could hide intent.

Verilog - Modules (cont.)

Unconnected port pins may be left unconnected like this...

```
ctrl_50mhz ctrl_50mhz_0(
  .clk_50   (clk_50  ),
  .reset_n  (reset_n ),
  .data_ena (data_ena),
  .a5_or_c3 (        ), //parenthesis left empty
  .wr_fifo  (wr_fifo));
```

Unused ports may be left out of the port list...*not recommended*
Make your intent clear!

Verilog - Modules (cont.)

Parametrization of modules allows for module reuse (powerful!)

```
//-------------------------------------------------------------------
//Shift register with width parameter
//-------------------------------------------------------------------
module shift_reg #(parameter width=8) (
      input                   reset_n,      //reset async active low
      input                   clk,          //input clock
      input                   data_ena,     //serial data enable
      input                   serial_data,  //serial data input
      output reg [width-1:0] parallel_data //parallel data out
      );
  always @ (posedge clk, negedge reset_n)
    if(!reset_n)       parallel_data <= '0; //could not do "width'd0"
    else if (data_ena)
      parallel_data <= {serial_data, parallel_data[width-1:1]};
endmodule
```

In the module declaration, the keyword "parameter" is not required
It does read more clearly however

Verilog - Modules (cont.)

Paramaterized "shift_reg" could now be instantiated like this:

```
module top_module;
reg [31:0] parallel_data;
wire reset_n, clk, data_ena, serial_data;

//instantiate shift reg, overriding width to 32 bits
shift_reg #(.width(32)) sr_inst0(
      .reset_n       (reset_n       ),
      .clk           (clk           ),
      .data_ena      (data_ena      ),
      .serial_data   (serial_data   ),
      .parallel_data (parallel_data ));
endmodule
```

Verilog - Modules (cont.)

- Verilog modules provide a coarse-grain structuring mechanism
- Modules contain RTL logic or other modules
- Modules provide an abstraction mechanism via complexity hiding
- Modules provide a way to reuse designs
- How is a Verilog module different from a C function?