

## What is Verilog?

Verilog is a Hardware Description Language (HDL). It is not a sequential programming language like C.

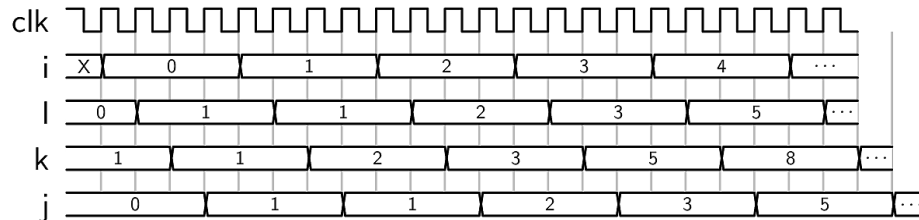
For example, consider this C code which would be executed line-by-line:

```

int j=0;
int k=1;
int l=0;
for (int i=0; i<6; i=i+1) {
    l = k;
    k = j + k;
    j = l;
}

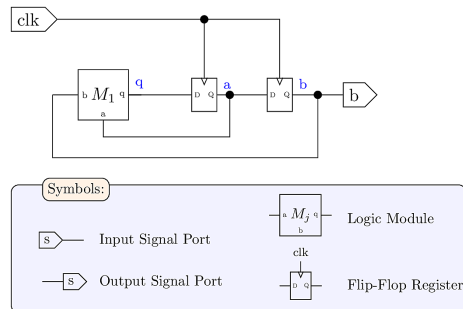
```

The variables are updated one at a time in each clock cycle:



## Modules and Circuits

An HDL describes hardware entities or modules that have their own separate existence, and interact via signals. The flow of signals is handled by registers which are updated synchronously by a master clock signal.

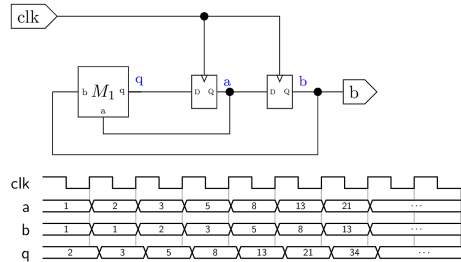


## Data Flow Example

Suppose each of the modules implements an integer addition function:

$$q := a + b,$$

and all registers are initialized with a value of one.



## Describing the Circuit in Verilog

Verilog supports modeling at several levels of abstraction, which include:

- Physical circuits (with electrical characteristics and delays)
- Structural circuits (describe schematic connections)
- Behavioral models (describe functional characteristics)
- Test and verification (many functions and tools similar to C libraries)

Modern digital designs are modeled at the Register Transfer Level (RTL), which primarily employs behavioral models that are synthesizable, and may also include structural hierarchies.

## RTL Basics

### Synthesizable RTL Design

RTL designs use a synthesizable subset of behavioral syntax.

Synthesizable syntax examples:

```

1     b <= a;           // Simple assignment
2
3     a <= a + b;      // Basic arithmetic
4
5     if (a > 0) b <= a; // Conditional logic
6     else      b <= 1;

```

### Non-synthesizable syntax examples:

```

1     real a, b;       // Real numbers work in simulation, not synthesis
2
3     #10 b = a;       // Delay 10 ``time units`` before assignment
4
5     $display("%f", b); // Like printf, displays values in the console

```

Non-synthesizable syntax is useful for testbenches, but doesn't translate to hardware.

## RTL Example

Here's a synthesizable behavioral model of our example module:

```
1 module M(clk, b);
2     input          clk;
3     output reg [7:0] b;
4
5     reg [7:0] a;
6
7     initial begin
8         a = 1;
9         b = 1;
10    end
11
12    always @(posedge clk) begin
13        a <= a + b;
14        b <= a;
15    end
16 endmodule
```

Let's go through this line by line.

## Module declarations

```
1 module M(clk, b);
2     input          clk;
3     output reg [7:0] b;
```

- Begins with the module keyword.
- The parentheses contain a comma-separated list of I/O signal ports.
- Terminated with a semicolon.
- Port direction, type, and bit-width are defined on the following lines.

## Alternative Port Declaration

An alternative syntax declares the I/O port types:

```
1 module M(
2     input          clk,
3     output reg [7:0] b
4 );
```

- Port direction, type, and bit-width are defined within the module declaration.
- This style is considered more readable, but less compact.

## I/O Ports: direction and type

- Every I/O port needs a defined direction, either input or output (or inout in special cases).
- I/O ports are signals. Every signal needs a type.
- The default type is a single-bit wire.

This line declares `clk` as one input wire:

```
1 input          clk;
```

## I/O Ports: Bus Declaration

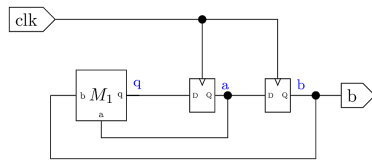
The next line declares an 8-bit bit vector or bus:

```
1 output reg [7:0] b;
```

- The `reg` keyword declares that signal ‘b’ will be defined as part of a hardware behavioral model, to be assigned within always or initial scopes (more on that later).
- The brackets `[7:0]` declare a bus with wires indexed from 7 down to 0, giving a total of eight.

## Internal Signals

In addition to I/O ports, most modules also need internal signals.



```
1 reg [7:0] a;
```

- Signal `a` is not accessible outside the module, so is declared “internally.”
- Again the keyword `reg` indicates that `a` is used in a hardware behavioral model.
- The signal is an 8-bit bus.

## Behavioral Models: Initialization

Verilog allows an initial statement to define the initial values of reg signals.

```
1 initial
2     begin
3         a = 1;
4         b = 1;
5     end
```

- In Xilinx FPGAs, the initial statement is synthesizable – it defines the powerup states of actual hardware signals.
- Multiple lines are grouped between begin and end statements.
- Initial assignments are made with the ‘=’ operator, and every line is terminated with a semicolon.
- WARNING: initial statements are not officially synthesizable. This code will work with Xilinx FPGAs, but to make portable RTL code that can be used with other tools and devices, you should define an explicit reset input and associated reset behavior to replicate the initial conditions.

## Version with Reset

The code below modifies our example to include a reset input.

```
1 module M(clk, rst_n, b);
2     input          clk;
3     input          rst_n;
4     output reg [7:0] b;
5
6     reg [7:0] a;
7
8     always @(posedge clk, negedge rst_n) begin
9         if (~rst_n) begin
10            a <= 1;
11            b <= 1;
12        end
13        else begin
14            a <= a + b;
15            b <= a;
16        end
17    end
18 endmodule
```

## Reset Considerations

- Reset signals are conventionally active low. Since a device will typically power-up with low-level signals (until something pulls them high), this helps ensure that it starts in the reset state.
- In our example, the always block is sensitive to negedge rst\_n, meaning it is evaluated when rst\_n falls to zero, and the assignments are executed when rst\_n is low:

```
1 always @(posedge clk, negedge rst_n) begin
2     if (~rst_n) begin
3         ...
4     end
```

- The subscript \_n is used here to indicate that the signal is active-low. This is done to make the code easier to understand.
- This code demonstrates an asynchronous reset, since the reset code is evaluated the instant rst\_n changes, regardless of what's happening with the clock signal. This can be important for powerup reset, since the system can reset before the clock ever starts.

## Behavioral Models: Synchronous Logic

Behavioral models are defined using the always keyword.

```
1 always @(posedge clk) begin
2     a <= a + b;
3     b <= a;
4 end
```

- The always keyword must be followed by a sensitivity list beginning with an '@' sign.
- For synchronous (i.e. clocked) logic, sensitivity is either posedge or negedge. In this case, the behavior is actuated on the rising edge (posedge) of clk.
- To make synchronous assignments, the <= operator is used. This assignment has a special meaning:
  - The assignment will be completed after the clock edge.
  - On the right side of the operator, the operations are computed before the clock edge.

## Behavioral Models: Combinational Logic

As an alternative syntax style, we can use a combinational model to make the circuit more explicit:

```

1
2 // COMBINATIONAL LOGIC:
3 reg [7:0] q;
4
5 always @(a,b) begin
6     q = a + b; // Blocking assignment
7 end
8
9 // SEQUENTIAL LOGIC:
10 always @(posedge clk) begin
11     a <= q; // Non-blocking assignments
12     b <= a;
13 end

```

- In combinational models, the sensitivity list must reference all dependencies for the assignments, in this case `a` and `b`.
- Alternatively a wildcard can be used: `always @(*)`
- Combinational assignments like `q = a+b` are blocking, interpreted line-by-line.
- Clocked assignments like `b <= a` are non-blocking, interpreted concurrently with other lines in the group.

## Behavioral Models: Continuous Assignments

As yet another alternative syntax style, we can declare `q` as a `wire` (as opposed to `reg`) and use the `assign` statement to define its combinational behavior:

```

1
2 // COMBINATIONAL LOGIC:
3 wire [7:0] q;
4
5 assign q = a + b; // continuous assignment
6
7 // SEQUENTIAL LOGIC:
8 always @(posedge clk) begin
9     a <= q; // Non-blocking assignments
10    b <= a;
11 end

```

This syntax provides a compact way to represent simple combinational assignments.

## Blocking vs Non-Blocking: Unstable Feedback

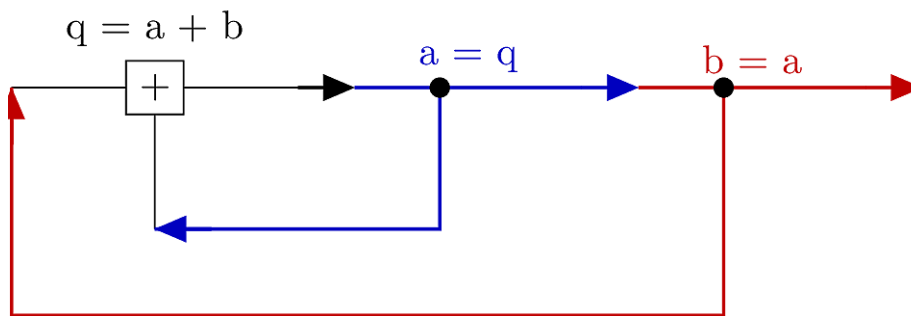
Consider this modified version that uses only blocking combinational assignments:

```

1   reg [7:0] q;
2
3   // COMBINATIONAL LOGIC:
4   always @(*)
5       q = a + b;
6       a = q;
7       b = a;
8   end

```

Within the `always @(*)` scope, each line is interpreted as a physical connection, like this:



This circuit has direct wired feedback, but nothing to regulate timing. The only stable solution is `q == a == b == 0`.

## Blocking vs Non-Blocking: Stable Timed Feedback

Our original design uses a mix of blocking and non-blocking assignments.

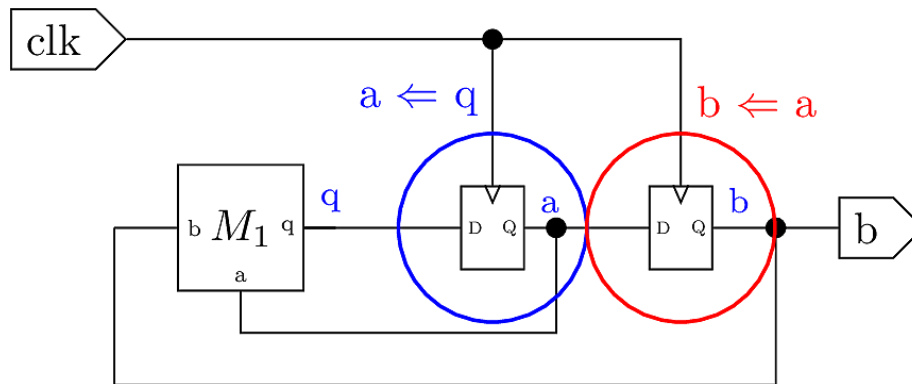
```

1   always @(a,b) begin
2       q = a + b; // Blocking assignment
3   end
4
5   always @(posedge clk) begin
6       a <= q; // Non-blocking assignments
7       b <= a; // create registers
8   end

```

- Each non-blocking assignment, when sensitive to `posedge clk`, implies a clocked D register, so data is processed on a controlled schedule.





## End of the Module

A module concludes with the endmodule statement. It should appear by itself on a line with no semicolon:

```
1 endmodule
```

After the endmodule statement, you can define additional modules in the same file if desired.

## Verilog Expressions

### Expression Syntax: Bitwise Operators

These operators are applied pairwise across all bits in a vector.

- & AND
- | OR
- ^ XOR
- ~ NOT

### Expression Syntax: Bitwise Operators

Example using continuous assignment:

```
1 input [7:0] a;
2 input [7:0] b;
3
4 wire [7:0] q;
5
6 assign q = a & b;
```

The output is:

```
q[0] = a[0] & b[0]
q[1] = a[1] & b[1]
q[2] = a[2] & b[2]
... and so on.
```

## Expression Syntax: Bitwise Operators

Example with `reg` signal:

```
1  input [7:0] a;
2  input [7:0] b;
3
4  reg [7:0] q;
5
6  always @(*) begin
7      q = a & b;
8  end
```

This is essentially the same as the `assign` example.

## Expression Syntax: Replication and Concatenation

- `{a,b}` concatenate `a` and `b` into one vector
- `{N{b}}` repeat `N` copies of `b`

example:

```
1  input [7:0] a;
2  input      b;
3  wire [7:0] q;
4
5  assign q = a & {7{b}};
```

The output is

```
q[0] = a[0] & b
q[1] = a[1] & b
q[2] = a[2] & b
... and so on.
```

## Expression Syntax: Logical vs Binary

- A logical expression is defined as a Verilog statement that is “true” or “false”.
- This is not precisely the same as binary signal values 0 or 1.

Examples:

Expression	Type	Value
a	binary signal	1
b	binary signal	0
a & b	bitwise operation	0
a == 1	logical expression	true
b == 1	logical expression	false

## Expression Syntax: Logical Operators

Logical operators are used for conditional expressions. In the syntax below, the term “exp” refers to any logical expression.

- a == b true if a equals b
- a != b true if a doesn't equal b
- exp1 && exp2 true if exp1 AND exp2 are true
- exp1 || exp2 true if exp1 OR exp2 are true
- !exp true if exp is false

Bitwise operators can be used in conditional expressions, but is bad style. Logical operators can be used in assignments, but is usually bad style.

## Expression Syntax: Conditionals

- if-else statements are supported for reg signals within always or initial scope blocks.
- Can be used within combinational or sequential logic.

```

1  input      [2:0] a;
2  output reg      b;
3
4  always @(a) begin
5      if (a>2)
6          b = 0;
7      else
8          b = 1;
9  end

```

## Expression Syntax: Conditionals

- expr ? true\_assignment : false\_assignment is supported for wire signals in continuous assignments.

```

1  input      [2:0] a;
2  output      b;

```

```

3
4   assign b = (a>2) ? 0 : 1;
5

```

## Expression Syntax: Conditionals

- case statements are supported for `reg` types in `always` scope blocks:

```

1   input      [2:0] a;
2   output    b;
3
4   always @(a) begin
5       case (a)
6           0:      b = 0;
7           1:      b = 0;
8           2:      b = 0;
9           default: b = 1;
10      endcase
11  end

```

## Inferred Latches Important!!

In combinational logic models:

- Every if should have an else
- Every case should have a default assignment.

Otherwise you will imply that the signal is stored in a latch, which is usually not what you want. Consider this example:

```

1   input      [2:0] a;
2   output reg b;
3
4   always @(a) begin
5       if (a>2)
6           b = 0;
7       else if (a==2)
8           b = 1;
9   end

```

What happens when `a` is less than 2?

## Four-Value Logic

### Four-Valued Logic: Signals

Verilog signals are binary, but can take four values:

- 0 Driven to logic low
- 1 Driven to logic high
- X Unknown or invalid signal
- Z Not driven (high impedance, floating, or disconnected)

## Four-Valued Logic: Bitwise Operators

When the inputs contain X or Z values, bitwise operations use an extended definition as shown in the truth tables below. Notice that when Z is an input, the output can still be a valid logic level, or can be X.

AND operation

	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

XOR operation

	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

OR operation

	0	1	X	Z
0	0	1	0	0
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

## Four-Valued Logic: Comparison Operators

Is this expression true or false?

```
(4'b10X1 == 4'b1011)
```

- In Verilog, this comparison returns X, neither true nor false, because one of the digits is unknown.
- To include X and Z values in the comparison, use the triple-equal operator (===).

```
(4'b10X1 === 4'b1011) (returns false)
```

```
(4'b10X1 === 4'b10X1) (returns true)
```

Similarly, to check inequality:

```
(4'b10X1 !== 4'b1011) (returns true)
```

```
(4'b10X1 !== 4'b10X1) (returns false)
```

## Arithmetic

### Arithmetic Operations

- $a + b$  Integer Addition
- $a - b$  Integer Subtraction
- $a * b$  Integer Multiplication (complex circuit)
- $a / b$  Integer Division (not synthesizable!)
- $a \% b$  Integer Modulo (not synthesizable!)
- $a \ll b$  Shift bits left
- $a \gg b$  Shift bits right

## Testing and Verification

### Testbenches

Most of the time, code should be tested in simulation before implementing it in hardware. A testbench is a module that does the following:

- Instantiate the design under test (DUT).
- Generate clock and reset signals.
- Create stimulus signals for all the DUT inputs.
- Perform verification tasks to ensure the DUT does what is expected.

A testbench module typically has no input or output ports. Automated verification may be done by comparing the DUT signals against a “golden” model of the intended design. Manual verification may be done by delivering information to the user via the console or output files.

### Example Testbench Module

Here is an example testbench for our design:

```
1 module testbench();
2     reg          clk;
3     reg          rst;
4     wire [7:0]  b;
5     integer      clock_count;
6
7     M DUT (
8         .clk(clk),
9         .rst(rst),
10        .b(b)
```

```

11     );
12
13     initial begin
14         clk          = 0;
15         rst          = 0; // Startup in reset
16         clock_count = 0;
17
18         forever #10 clk = ~clk; // Create clock signal
19     end
20
21     always @(posedge clk) begin
22         clock_count <= clock_count + 1;
23         if (clock_count > 0)
24             rst <= 1; // End reset after one clock cycle
25     end
26 endmodule

```

## Useful Testbench System Tasks

There are several non-synthesizable \$ commands called system tasks that are useful for verification and debugging:

- \$display Like printf, used to print information to the console.
- \$stop Like a breakpoint, pauses simulation.
- \$finish Ends simulation completely.
- \$reset Restart simulation from time zero.
- \$random Generate a random integer.

Example for our testbench:

```

1 always @(posedge clk) begin
2     clock_count <= clock_count + 1;
3     if (clock_count == 1)
4         rst <= 1; // End reset after one clock cycle
5     else if (clock_count == 18)
6         $finish;
7
8     $display("%d\t%d\t%d", clock_count, rst, b);
9 end

```

Like in the C printf function, '\t' indicates a tab, and %d indicates a decimal value.

## Note on Simulating Examples

When simulating small example designs, it is often more convenient to run terminal simulations rather than create a full Vivado project. Suppose your design files are in a directory named "example\_design", and the files are named

“M.v” and “testbench.v”. Open a terminal, change to that directory, and run these commands:

```
[example_design/]$ xvlog M.v testbench.v
[example_design/]$ xelab --debug typical testbench -s sim
[example_design/]$ xsim --runall sim
```

The first command, `xvlog`, compiles your code and reports any syntax errors. If your design uses all the Verilog files in the directory, you can specify `*.v` instead of listing the files. The second command, `xelab`, analyzes the hierarchy and reports any connection problems or undefined modules. You specify the top-level module name (“testbench” in this example), and provide a simulation name (“sim”). The final command, `xsim`, runs the simulation. The `-runall` argument tells it to run in batch mode; make sure your testbench has a `$finish` command so that it knows when to stop.

## Simulating with Cadence Incisiv

When you are on campus, you can utilize a faster and more powerful simulation tool from Cadence Design Systems. This tool is widely used in industry for designing electronic chips and systems. Our university license only allows it to be used on-site. To use the Cadence simulator, you need to add the tool’s directory to the `PATH` environment variable. To do this, launch a text editor like `gedit` and open the file named `.bashrc` in your home directory. Add this line to the end of the file:

```
export PATH=/opt/cadence/installs/INCISIV152/tools/bin:$PATH
```

Save the file, then launch a new terminal. Now you can use the `irun` command to compile, elaborate and simulate all in one step:

```
[example_design/]$ irun M.v testbench.v
```

This usually runs much faster than the Xilinx tools, and gives you a chance to test your code on different platforms.

## Testbench `$display` Output

Running this testbench with our module yields the output shown at right. The columns are:

1. `clock_count`
2. `rst`
3. `b`

As expected, the `rst` signal starts at 0 and rises to 1 in the next cycle after `clock_count` equals 1. Then `b` starts accumulating after another cycle delay (remember that `<=` assignments have a clock delay, because they load data into flip-flops that activate on the clock edge).



Notice that we have an overflow condition at `clock_count` 16. This is because we only allocated 8 bits to our signals, so the highest they can count is 255. We will revisit this problem shortly.

<code>clock_count</code>	<code>rst</code>	<code>b</code>
-----	---	----
0	0	1
1	0	1
2	1	1
3	1	1
4	1	2
5	1	3
6	1	5
7	1	8
8	1	13
9	1	21
10	1	34
11	1	55
12	1	89
13	1	144
14	1	233
15	1	121
16	1	98 <== ovf
17	1	219

### Alternative: `$strobe` Output

Instead of using `$display`, we can use a similar command called `$strobe`. It has the same syntax, but whereas `$display` shows values just *before* the clock edge, `$strobe` shows values just *after* the clock edge, i.e. after all the synchronous assignments are completed.

The difference is subtle. In this example it just means we start on `clock_count` 1 instead of 0.

<code>clock_count</code>	<code>rst</code>	<code>b</code>
-----	---	----
1	0	1
2	1	1
3	1	1
4	1	2
5	1	3
6	1	5
7	1	8
8	1	13
9	1	21
10	1	34

```

11    1    55
12    1    89
13    1   144
14    1   233
15    1   121
16    1    98
17    1   219
18    1    61

```

### Alternative \$write Output

Another task with the same syntax is `$write`. The only difference is that `$display` adds a newline, and `$write` doesn't. This can be useful for printing arrays, or for splitting complicated displays onto several lines, like in the example code below.

```

1    always @(posedge clk) begin
2        clock_count <= clock_count + 1;
3        if (clock_count == 1)
4            rst <= 1;
5        else if (clock_count == 18)
6            $finish;
7
8        $write("%d\t",clock_count);
9        $write("%d\t",rst);
10       $write("%d\n",b);
11    end

```

clock_count	rst	b
-----	---	----
0	0	1
1	0	1
2	1	1
3	1	1
4	1	2
5	1	3
6	1	5
7	1	8
8	1	13
9	1	21
10	1	34
11	1	55
12	1	89
13	1	144
14	1	233
15	1	121
16	1	98

## File Input and Output

Suppose instead of writing to the console, we want to write to a file. To do this, we can open a file using the `$fopen` task, and print text data using the `$fwrite` task. The `$fwrite` task has essentially the same syntax as `$write`:

```

1  integer f; // File identification number
2
3  initial begin
4      f = $fopen("output.txt","w"); // open "output.txt" for writing
5      clk          = 0;
6      rst          = 0; // Startup in reset
7      clock_count = 0;
8
9      forever #10 clk = ~clk; // Create clock signal
10 end
11
12 always @(posedge clk) begin
13     // Write a line to the file:
14     fwrite(f,"%d\t%d\t%d\n",clock_count,rst,b);
15
16     // Keep track of time, decide when to quit:
17     clock_count <= clock_count + 1;
18     if (clock_count == 1)
19         rst <= 1; // End reset after one clock cycle
20     else if (clock_count == 18) begin
21         $fclose(f); // Close file when done
22         $finish;
23     end
24
25 end

```

## Reusability, Portability, Readability

Parameters Within a module definition, you can declare parameters to make the module more flexible. The code below works exactly the same as our original version, but now the width can be adjusted. With a width of 8 bits, the maximum value is 255. We might need to support larger numbers in some applications. Using a parameter makes it easy to adjust.

```

1  module M(clk, b);
2      parameter WIDTH=8;
3
4      input          clk;
5      output reg [WIDTH-1:0] b;

```

```

6
7     reg          [WIDTH-1:0] a;
8
9     initial begin
10        a = 1;
11        b = 1;
12    end
13
14    always @(posedge clk) begin
15        a <= a + b;
16        b <= a;
17    end
18 endmodule

```

### **‘define, localparam, param**

There are several different ways to define parameters:

‘define: global constants and macros

```

1 // WIDTH will be the same everywhere in the design
2 `define WIDTH 8
3
4 module M(clk, b);
5     output reg [WIDTH-1:0] b;
6     //...
7 endmodule

```

Note that this command uses a “back tick” located at the upper left of the keyboard.

### **‘define, localparam, param**

parameter: constants can be modified for each module instance:

```

1 module M(clk, b);
2
3     // WIDTH will be constant only within each instance
4     parameter WIDTH = 8;
5
6     output reg [WIDTH-1:0] b;
7     //...
8 endmodule

```

### **‘define, localparam, param**

We can instantiate two M modules with different WIDTHS:

```

1 module testbench();
2     //...
3
4     // Instance "DUT1" has WIDTH 8
5     M #(.WIDTH(8)) DUT1 (
6         .clk(clk),
7         .rst(rst),
8         .b(b1)
9     );
10
11    // Instance "DUT2" has WIDTH 16
12    M #(.WIDTH(16)) DUT2 (
13        .clk(clk),
14        .rst(rst),
15        .b(b2)
16    );
17
18    //...
19 endmodule

```

Parameter values can be modified in the instance declaration using the syntax:

```
<module_type> #(<parameter_list>) <instance_name> (<port_list>);
```

#### **‘define, localparam, param**

If we add the 16-bit signal, b2, to the output data file, we see that the overflow does not occur at clock-count 16:

clock_count	rst	b1 (8bit)	b2 (16bit)
=====	===	=====	=====
0	0	1	1
1	0	1	1
2	1	1	1
3	1	1	1
4	1	2	2
5	1	3	3
6	1	5	5
7	1	8	8
8	1	13	13
9	1	21	21
10	1	34	34
11	1	55	55
12	1	89	89
13	1	144	144
14	1	233	233
15	1	121	377

16	1	98	610	<==== b1 overflows, not b2
17	1	219	987	

## 'define, localparam, param

localparam: constant within module, not changeable

```

1 module M(clk, b);
2
3     // WIDTH will be constant in all instances of M
4     localparam WIDTH = 8;
5
6     output reg [WIDTH-1:0] b;
7     //...
8 endmodule

```

## Tasks

A task is sort of like a module within a module. Tasks are used to package lines of code that will be reused multiple times. For example, suppose we need to reverse the bit-order of some signals:

```

1 module my_interface (
2     output reg [7:0] out1,
3     output reg [7:0] out2,
4     input [7:0] sig1,
5     input [7:0] sig2
6 );
7     integer i;
8
9     always @(sig1,sig2) begin
10         reverse_bits(sig1,out1); // Makes code more compact,
11         reverse_bits(sig2,out2); // improves readability
12     end
13
14     task reverse_bits(
15         input [7:0] sig,
16         output [7:0] out
17     );
18
19     begin
20         for (i=0; i<8; i=i+1)
21             out[7-i] = sig[i];
22     end
23     endtask // reverse_bits
24
25 endmodule

```

## Task Example Testbench

```
1 module testbench();
2     reg      clk;
3     reg [7:0] sig1;
4     reg [7:0] sig2;
5     wire [7:0] out1;
6     wire [7:0] out2;
7
8     my_interface DUT1 (
9         .sig1(sig1),
10        .sig2(sig2),
11        .out1(out1),
12        .out2(out2)
13    );
14
15    initial begin
16        clk = 0;
17        sig1 = 0;
18        sig2 = 8'hFF;
19
20        // Print column headers on the terminal:
21        $write(" sig1 \t sig2 \t out1 \t out2\n");
22        $write(" ==== \t ==== \t ==== \t ==== \n");
23
24        forever #10 clk = ~clk; // Create clock signal
25    end
26
27    //...continued on next slide...
```

## Task Example Testbench (continued)

```
1 always @(posedge clk) begin
2     sig1 <= sig1 + 1; // sig1 counts up from zero
3     sig2 <= sig2 - 1; // sig2 counts down to zero
4
5     if (sig2 == 0)
6         $finish;
7
8     // Print signal values on the terminal:
9     $write("%b\t",sig1);
10    $write("%b\t",sig2);
11    $write("%b\t",out1);
12    $write("%b\n",out2);
13    end
14 endmodule
```

## Task Example Output

sig1	sig2	out1	out2
====	====	====	====
00000000	11111111	00000000	11111111
00000001	11111110	10000000	01111111
00000010	11111101	01000000	10111111
00000011	11111100	11000000	00111111
00000100	11111011	00100000	11011111
00000101	11111010	10100000	01011111
00000110	11111001	01100000	10011111
00000111	11111000	11100000	00011111

## Functions

Functions are like tasks, with a couple of differences:

- Functions return a single value.
- Function calls look like this:  
`out_signal = function_name(in1, in2, ...);`
- Tasks allow timing syntax (`#delays`, `posedge`, `negedge`). Functions do not.
- Functions can call other functions, but cannot call tasks.
- Functions can model combinational logic assignments.

## Functions vs Tasks

The main differences are:

	task	function
Multi output	Y	N
Timing Statements	Y	N
Consumes time	Y	N
inout ports	Y	N
output ports	Y	N
can call tasks	Y	N
can call functions	Y	Y
returns a value	N	Y

Tasks are versatile, but functions can be safer since they do not affect timing.

## Function Example

Here is our task example, rewritten as a function:



```

1  module my_interface (
2      output reg [7:0] out1,
3      output reg [7:0] out2,
4      input [7:0] sig1,
5      input [7:0] sig2
6  );
7
8      always @(sig1,sig2) begin
9          out1 = reverse_bits(sig1);
10         out2 = reverse_bits(sig2);
11     end
12
13     function [7:0] reverse_bits;
14         input [7:0] sig;
15         integer i;
16         begin
17             for (i=0; i<8; i=i+1)
18                 reverse_bits[7-i] = sig[i];
19         end
20     endfunction // reverse_bits
21
22 endmodule

```