

Introduction to Hardware Description Languages - Verilog

Luca Macchiarulo

Verilog Core Element

- Verilog: Module:



Typical bits and bit arrays are predefined in Verilog – no need for package inclusions

```
module simple(a, b, c ,y);  
input a;  
input b;  
input c;  
output y;
```

Module header –
Contains the interface of the component – inputs and outputs – and possibly some passing parameters

```
assign y = (a | b) & c;  
endmodule
```

Module core–
Contains the description of how the output depend on the inputs – there can be only one core per module – no need for configuration

Another example – 32 bit adder



Basic arithmetic is already included
– no need for packages

```
module simple_adder (a, b, y);  
input [31:0] a;  
input [31:0] b;  
output [31:0] y;  
    assign y = a + b;  
endmodule
```

Verilog 2001 C-style headers

Verilog 2001 introduced the simpler format for header where input output and size can be embedded in the module interface

```
module simple(input a, b, c, output y);
```

```
module simple_adder (
```

```
input [31:0] a,
```

```
input [31:0] b,
```

```
output [31:0] y);
```

Combinational Logic

- In verilog simple combinational logic (logic that is not registered or controlled by a clock) can be expressed by a simple **signal assignment statement** **assign ... = ...;**
 - assign y = ~a;
- Implicitly, all the signal used in the right hand side expression are used to decide when the expression is evaluated (technically, they are in the “sensitivity list” of the assignment)
- From the point of view of synthesis, this means that the hardware will not contain any FF or latch

Bitwise operators

- The operation can be repeated on all bits in a `std_logic_vector`:

`assign y = ~a;`

is the same (if y and a are [2:0] arrays) as

`y[0] = ~a[0];`

`y[1] = ~a[1];`

`y[2] = ~a[2];`

- Verilog has also useful **reduction operators**:

`assign x = |a;` //where x is a single bit

(equivalent to `assign x = a[2] | a[1] | a[0];`)

Conditional Assignment

- Classical C-style conditional expression:
assign `y = s ? d1 : d0;` // `s` is a single bit
Note that as in C, 1 is equivalent to “condition true”)
- Does not have multiple tests (VHDL’s with or when), but they can – somewhat inelegantly – be faked by a chain of conditional expressions

Internal variable

- Normally in a complex architecture you need intermediate values. They need to be declared before used – anywhere in the module:

```
module simple(input a, b, c , output y);  
  wire a,b,c,y; //not necessary – signals are “wire” by default  
  wire p; // also not necessary for single bits – but strongly  
           //recommended  
      assign p =a | b;  
      assign y =p & c;  
endmodule
```

Note that all the architecture is still combinational – can be seen as 2 gates connected together – the synthesizer will try to fit it in the smaller number of logic blocks in the target technology (for FPGA, LUTs)

Expressions and precedence

- Inversion: \sim
- Multiplicative: $*$ / $\%$
- Additive: $+$ -
- Logical shifts $\ll \gg$
- Arithmetic shifts $\lll \ggg$
- Relative comparison: $< \leq \geq >$
- Equality comparison: $== !=$
- And, nand: $\& \sim\&$
- Xor xnor: $\wedge \sim\wedge$
- Or nor: $| \sim|$
- Conditional: $\dots? \dots: \dots$

Numbers

- Integer and real numbers are written normally (130, 0.12)
 - There is no reason to use real numbers but in testbenches for time expressions – they are most clearly non-synthesizable
 - Integers should be used sparingly, as they can take a lot of space (typically 32 bits) but they are ok if used as constants in expressions
- Binary numbers can be written conveniently (it is good practice to always indicate the bit number):
 - 8'b10110010
 - 8'hB2
- Single bits do not require single quotes 0, 1

High impedance, uninitialized

- Verilog “bits” are more complex than digital bits. Besides '0' and '1' a bit can be:
 - 'u' : undefined (at start of simulation before any assignment)
 - 'z' : high impedance (to model tristate buffers)
- Useful in simulation, ignored in synthesis
 - There is a special equality operator `===` that checks equality of Xs that are normally ignored

Bit swizzling

- It is easy to break up and build a new vector:
assign $y[5:3] = \{a[2:1], 1'b0\}$;
- The strange ordering of vector indices makes sure numeric vectors are read correctly with constants:
 - assign $y[3:0] = 4'b1100$; /* assigns the (decimal) number 12 to the 4 Lsbits of y */
 - There is a useful repetition operator:
 - $\{a, \{3\{d[0]\}\}$ is equivalent to $\{a, d[0], d[0], d[0]\}$

Delays

- For simulation (but NOT for synthesis) it is possible to fix the delays in assignments:
 - `assign #2 y = a;`
 - `assign #10 z = b + c;`
- The timescale is defined at the beginning of the file:
 - ``timescale 1ns/1ps /* unit = 1ns simulation resolution = 1ps */`
- Synthesis does not use them at all, and if you want to set a timing constraint, you need to follow a different route (see the next days).

Structural modelling

- How to describe very complex systems?
 - By having a hierarchy of modules, in which one is used in a higher module
 - An module that is used in a higher level module is instantiated by name – only one module with the same name – no need for configurations
 - The instance syntax is somewhat different than VHDL : the name of the component comes first, then the instance, then the port map.
 - `Comp_name inst_name (.f1(a1), .f2(a2),.f3(a3));` where f are formal parameters, a actual signals in the upper module

Example – leaf module

```
21 module Mux2_v(a,b,s,y);  
22   input a,b,s;  
23   output y;  
24  
25   assign y = s ? b : a;  
26  
27 endmodule  
28
```

Example – hierarchy top

```
21 module Mux4_v(a,b,c,d,s,y
22     );
23     input a,b,c,d;
24     input [1:0] s;
25     output y;
26
27     wire m0, m1;
28
29     Mux2_v Mux_I(
30         .a(a),
31         .b(b),
32         .s(s[0]),
33         .y(m0)
34     );
35
36     Mux2_v Mux_II(
37         .a(c),
38         .b(d),
39         .s(s[0]),
40         .y(m1)
41     );
42
43     Mux2_v Mux_y(
44         .a(m0),
45         .b(m1),
46         .s(s[1]),
47         .y(y)
48     );
49
50 endmodule
```

No need to declare a module:
just use its name

Assignments of .formal(actual):
c is a signal in Mux4, a is its name in Mux2

Side note – strange/confusing conventions

- Verilog has some different conventions and symbols than VHDL:
 - Colons inside a port, semicolons as a statement/instance separator
 - Module x(a, b, c);
 - Test for equality is ==
 - Test for inequality is !=
 - There are 3 types of assignments:
 - = in continuous assignments
 - = inside always or initial (blocking assignments)
 - <= inside always or initial (non-blocking assignments)
 - If statements ALWAYS require parenthesis, and no **then** and **end if**;
 - Alternative if is **else**
 - Comments are introduced by // or /* ... */ as in C

Sequential Logic

- In Verilog this needs to be described using special **always** statements
 - always @(posedge clk)
 - Signals in always statements cannot be wires, they need to be declared reg:
 - reg clk;
- The key for the synthesizer to recognize sequential logic is the fact that the assignments are not continuous – there are conditions in which nothing happens now – see the D-FF of the following page

D-FF

```
21 module D_FF_v(clk, d, q);  
22   input clk, d;  
23   output q;  
24   reg q;  
25  
26   always @(posedge clk)  
27   begin //begin end could be omitted as single statement  
28     q<=d;  
29   end  
30  
31 endmodule  
32
```

All signals on the LHS
Of an assignment inside always and initial
Need to be declared reg (even if not sequential!)

Sensitivity list: only when there is
a change in the listed signals
the process is active

This is happening only at the rising edge of clk, all other times q remembers the old value

Resettable registers

```
21 module Reg_res_v(clk,rst,d,q);
22   input clk, rst;
23   input [7:0] d;
24   output [7:0] q;
25   reg [7:0] q;
26
27   always @(posedge clk or posedge rst)
28   if(rst)
29     q<=0;
30   else
31     q<=d;
32
33 endmodule
```

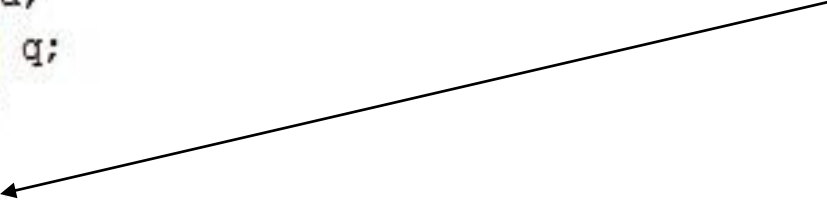
Verilog2001 allows
(posedge clk, posedge rst)

Asynchronous reset:
It operates independently of clk

Synchronizer

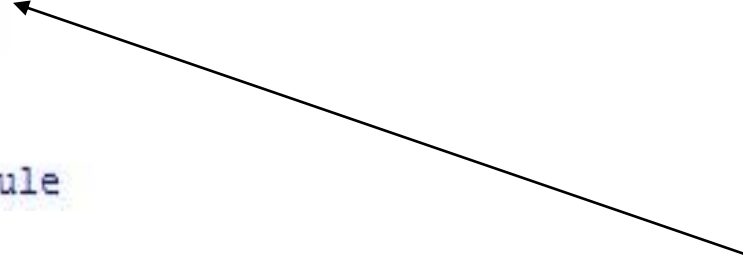
```
21 module synchronizer_v(clk,d,q);
22 input clk;
23 input [7:0] d;
24 output [7:0] q;
25 reg [7:0] q;
26
27 reg [7:0] n;
28
29 always @(posedge clk)
30 begin
31 n<=d;
32 q<=n;
33 end
34
35 endmodule
```

Also n needs to be declared reg



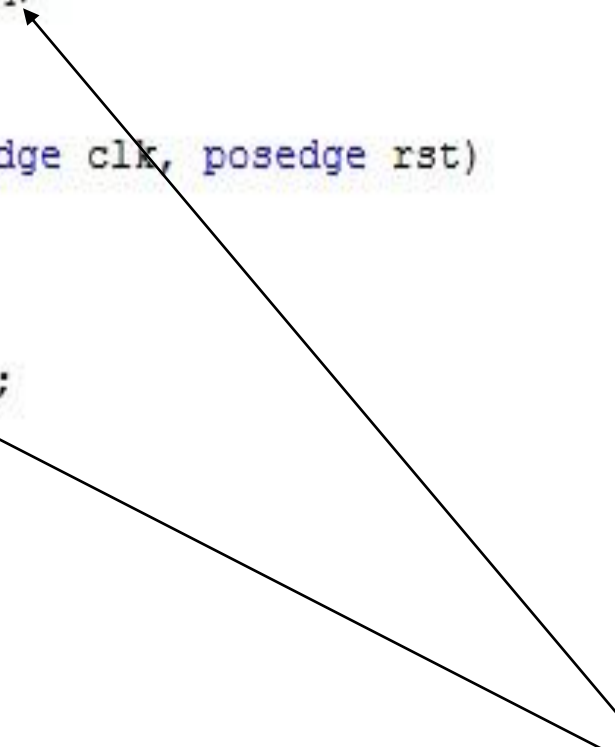
IMPORTANT!

The assignments are performed using the d and n values BEFORE the clock transition so this introduces an extra clock delay!



Counter

```
21 module counter_v(clk, rst, q);  
22   input clk, rst;  
23   output [7:0] q;  
24   reg [7:0] q;  
25  
26   always @(posedge clk, posedge rst)  
27   if(rst)  
28     q <= 0;  
29   else  
30     q <= q + 1;  
31  
32 endmodule
```



No need for internal signal as Verilog has no constraints on outputs being used inside an architecture

Latch

```
21 module Latch_v(load, d, q);  
22   input load, d;  
23   output q;  
24   reg q;  
25  
26   always @(load or d)  
27       if(load) q<=d;  
28  
29 endmodule
```

Note the sensitivity list -
No “posedge”



NOTE: Xilinx discourages the use of latches and asynchronous resets – mostly for timing closure issues, but they are occasionally handy (to record one time flags, for example)

Combinational processes

- Processes can be also used to describe complex combinational logic, not only sequential:
 - The trick is to make sure that ALL possible changes in the used signals are actually taken into consideration
 - This requires having all “input” signals in the sensitivity list (if not, the system might work but the simulation might not match the synthesis)
 - Verilog2001 has a shortcut to guarantee it in always statements: `always @(*)`
 - And also making sure that all “outputs” are assigned a value in the process no matter the value of any signal

Example of wrong “combinational” process

```
always @(a)
begin
    if a == 2'b00
        y <= 1'b0;
    else if a == 2'b01
        y <= b;
    else if a == 2'b10
        Y <= b;
end
```

b missing from sensitivity list

Ooops – what happens if a = “11”?

The diagram illustrates a Verilog code snippet for a combinational process. It features three annotations with arrows pointing to specific parts of the code: 1. An arrow from the text 'b missing from sensitivity list' points to the variable 'b' in the assignment 'y <= b;' within the 'else if a == 2'b01' block. 2. An arrow from the text 'Ooops – what happens if a = “11”?' points to the assignment 'Y <= b;' in the 'else if a == 2'b10' block. 3. An arrow from the text 'always @(a)' points to the sensitivity list '(a)' in the 'always' statement, highlighting that variable 'b' is not included in the list of variables that trigger the process.

Corrected “combinational” process

```
always @(a or b) // in Verilog 2001 also always @(a,b)
begin
    if a = 2'b00
        y <= 1'b0;
    else if a = 2'b01
        y <= b;
    else if a = 2'b10
        y <= b;
    else
        y <= b;
end
```

All used signals in sensitivity list

Else guarantees all cases are covered

Types of assignments inside a process

- There are 2 types of signal assignments in a process:
 - Non-blocking: signalled by the use of `<=`
 - Same meaning as in VHDL – all changes are done based on the values before entering the always block. One assignment “does not block” the next.
 - Blocking: use the `=`
 - The execution makes sure that the assignment is performed before the next are executed, so in practice the assigned signals behave as VHDL variables, and can be used to force immediate evaluation

Example of blocking assignment

```
reg [7:0] v1;
```

```
reg [7:0] v2;
```

```
always @(posedge clk)
```

```
begin
```

```
    v1 = a + b;
```

```
    v2 = c + d;
```

```
    y <= v1 + v2;
```

```
end
```

At every clock cycle, y will be updated with the current value of a+b+c+d. What would have Happened if v1 and v2 were assigned using <=?

Statements allowed inside a always or initial

- Case statements:

```
case (a)
```

```
    2'b00 : y<= 1'b0;
```

```
    2'b01 : y<= b;
```

```
    2'b10 : y<= b;
```

```
    default : y<=b;
```

```
endcase
```

Statements allowed inside an always or initial

- Some statements are very useful for simulation but are meaningless and should not be used for synthesis:
 - #number → used to introduce a finite delay between one statement and the next – no code with this will be synthesizable (we'll see it in testbenches)
 - No assert statements, but can be mimicked with if checks

FSM descriptions

- FSM can be described in various ways in Verilog. The most common ways are trying to mimic the HW implementation of FSMs:
 - 2 processes, 1 updating the state at the clock tick, 1 computing the future state and outputs
 - 3 processes, 1 updating the state at the clock tick, 1 computing the future state, 1 computing the output (can easily distinguish Moore and Mealy machines).
 - A single process, updating a single state signal inside a posedge clock always statement – only for Moore machines
- It typically uses:
 - case statements to distinguish between the current state
 - Verilog does not have enumeration types for states – they need to be assigned

FSM example – 2 processes

```
21 module FSM_v(clk, a, o);
22   input clk, a;
23   output o;
24
25   reg o;
26
27
28   parameter S0 = 2'b00;
29   parameter S1 = 2'b01;
30   parameter S2 = 2'b10;
31
32   reg [1:0] current_state = S0;
33   reg [1:0] next_state = S0;
34
35   always @(posedge clk)
36     current_state <= next_state;
37
38   always @(*)
39   begin
40     case(current_state)
41     S0 : begin
42         if(a) next_state<=S1;
43         else next_state <=S0;
44         o<= 1'b1;
45     end
46     S1 : begin
47         if(a) next_state<=S2;
48         else next_state <=S1;
49         o<= 1'b0;
50     end
51     S2 : begin
52         if(a) next_state<=S0;
53         else next_state <=S2;
54         o<= 1'b0;
55     end
56     default : begin
57         next_state<=S0;
58         o<=1'b1;
59     end
60   endcase
61 end
62
63 endmodule
```

Verilog does not have enumerative types: we can “fake” it using parameters (constants at compile time)

This process needs to be completely combinational – next_state needs to be assigned for any possible Input – sensitivity list @(*) guarantees that all signals are listed – case statement and if statements cover all cases (check!)

Default guarantees that the FSM will always Land in a legal state, also in simulation

Testbenches

- Once a module is written and passes syntax checks, how do we test it behaves as expected?
 - Simulation of course, but how do we feed the inputs?
 - Building a **testbench**: a Verilog code that can be simulated and that provides all appropriate inputs and relative timing
- A testbench:
 - Is an empty entity (no inputs or outputs – and no port!)
 - uses # constructs to add delays and synchronize the inputs with clocks

Example

Testbench for the FSM

```
25 module tb_FSM_v;  
26  
27     // Inputs  
28     reg clk;  
29     reg a;  
30  
31     // Outputs  
32     wire o;  
33  
34     // Instantiate the Unit Under Test (UUT)  
35     FSM_v uut (  
36         .clk(clk),  
37         .a(a),  
38         .o(o)  
39     );  
40  
41     initial forever  
42     begin  
43         clk = 0;  
44         #1;  
45         clk = 1;  
46         #1;  
47     end  
48     initial begin  
49         // Initialize Inputs  
50         clk = 0;  
51         a = 0;  
52  
53         // Wait 100 ns for global reset to finish  
54         #100;  
55         a = 1;  
56         #2;  
57         a = 0;  
58         #2;  
59         a = 1;  
60         #2;  
61         a = 0;  
62         #2;  
63         a = 1;  
64     end  
65  
66 endmodule
```

Instance of “Unit Under Test”

Sequence of inputs

No further change from here