Embedded Systems (digital design)

Verilog and SystemVerilog

- basics on combinatorial design -

Prof. Davide Zoni (davide.zoni@polimi.it)

2022

Let’s start from the beginning

- Verilog 2001: The Basic “Ingredients” -

Verilog 2001: Data Types

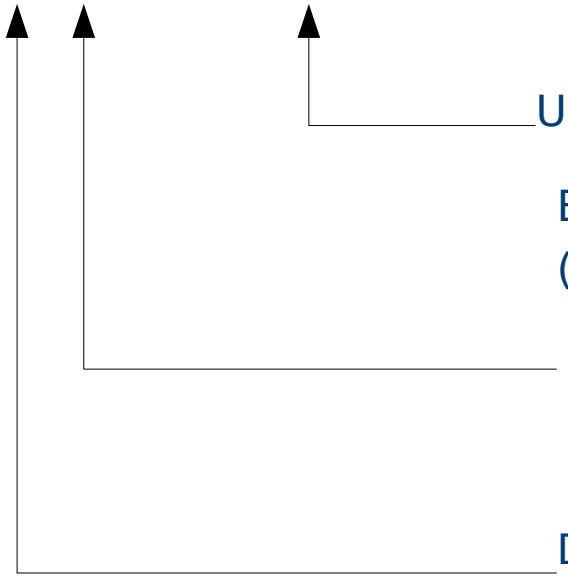
- **Vector of bits is the only data type we consider in these classes**
  - No struct, nor float data types
  - Integer is a 32-bit vector (the code, e.g., 2's complement)
  - String can be declared as an n-bit vector where each char is 8-bit long
- **Each bit of the vector can take on one of four values**

Value	Meaning Simulation	Meaning Synthesis
0	Logic zero	Logic zero
1	Logic one	Logic one
X	Unknown	Don't care
Z	High impedance	High Impedance

- Set X in simulation where we want don't care, thus **helping the synthesis** results.
- Moreover, **it helps finding bugs**

Bit literals

8’b0101_1111



Underscores are ignored

Base format
(d – decimal
b – binary
o – octat
h – hexadecimal)

Decimal number
representing the size in bits

- **Binary literals**
  - 8’b0000_0000
  - 8’b0xx0_1xx1
- **Hexadecimal literals**
  - 32’h0123_cdef
  - 16’haxxx
- **Decimal literals**
  - 32’d58

Verilog 2001: wire and reg (and logic)

- **wire:**
  - used to denote a hardware net
    - wire [15:0] instruction;
    - wire [7:0] byte_bus;

- **reg:**
  - It is a variable that can be used to implement both combinational and sequential logic
  - In general, it does not identify a hardware register.
    - The procedure of inferring a hardware register is subject to the way we describe the component

Verilog 2001 and SystemVerilog 2012

- **Verilog 2001:**
  - no type safety checks
  - only checks the width of the two connecting signals. A width mismatch is solved by trimming down the wider signal to the width of the shorter one
- **SystemVerilog2012:**
  - Backward compatible with Verilog 2001
  - The ***logic*** keyword to overcome the distinction between ***reg*** and ***wire***
  - The ***logic*** keyword can be used in place of ***reg*** and ***wire***
  - ***reg*** and ***wire*** keywords can be still used in SystemVerilog design
- **Question: why teaching a defunct HDL, i.e., Verilog 2001?**
  - Several commercial designs make use of Verilog 2001
  - Commercial CAD tools
  - Netlists are in Verilog 2001
- **In the next:**
  - I will use Verilog and SystemVerilog interchangeably
  - We start by mixing Verilog and SystemVerilog to gracefully move to almost pure SystemVerilog

Verilog 2001: Boolean Operators

- **Bitwise operators:** perform bit-sliced operations on vectors
  - ~(4'b1010)={~1,~0,~1,~0}=4'b0101
  - 4'b1010 & 4'b0011 = 4'b0010
- **Logical operators:** return one-bit result
  - a && b
- **Reduction operators:**
  - bit-wise operation on a single operand returning one-bit result
  - &(4'b0101) = 0 & 1 & 0 & 1 =1'b0
- **Comparison**: boolean test on two args

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a~^b	XNOR

Reduction

&a	AND
~&a	NAND
|a	OR
~|	NOR
^	XOR

Logical

!a	NOT
a && b	AND
a || b	OR

Comparison

a<b a>b a>=b	Relational
a==b a!=b	(in)equality. Returns x when x or z in bits. Else 0 or 1, bit-wise comparison
a === b a !== b	Case (in)equality. Returns 0 or 1on a bit-2-bit comparison. x and z values matter

Shift operators

a	a>>2	a>>>2	a<<2	a<<<2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100

Synthesis

- If both operands of a shift operator are signals, e.g., a<<b, the inferred operator is a barrel shifter
- If the shift amount is fixed the inferred design only requires proper routing of the input signals

Functionality

- Logical shift (<< and >>) shifts in 0's
- Arithmetic shift (<<< and >>>) shifts in the sign bit (MSB) for >>> and 0's are shifted in for <<<
- There is no difference between << and <<<. the latter is included for completeness

Bitwise reduction and logical operators (all for synth) 1/2

a	b	a&b	a|b	a&&b	a||b
0	1	0	1	0 (false)	1 (true)
000	000	000	000	0 (false)	0 (false)
000	001	000	001	0 (false)	1 (true)
011	001	001	011	1 (true)	1 (true)

4 basic bitwise operators: and (&), or (|), xor (^), and ~ (not)

- Operation performed at bit-granularity

3 basic logical operators: logical and (&&), logical or (||), logical not (!)

- If no x or z values are used the operands are false when all bits are 0s
- If x or z values are used (see next slide)

Bitwise reduction and logical operators (all for synth) 2/2

a	b	a&b	a|b	a&&b	a||b
0	x	0	x	0	x
0	z	0	x	0	x
1	x	x	1	x	1
1	z	x	1	x	1
0000	01xz	0000	01xx	0	1
1001	01xz	000x	11x1	1	1
1001	00xz	000x	10x1	x	1
1001	00zz	000x	10x1	x	1

Use z and x values carefully to avoid propagating x values

Relational and equality operators

Synthesizble

- 4 relational operators: <, >, <=, >=
- 2 equality operators
- Functional – compars two operands and return either false (1-bit 0) or true (1-bit 1)
- Synthesis – infers comparators

Non-synthesizable

- 2 equality operators: case equality (===) and case inequality (!==)
- Functional – perform the comparison accounting for x and z values

Precedence of the operators

Operator
!, ~, +, - (unary)
*, *
*, /, %
+, - (binary)
<<, <<<, >>>, >>
==, !=, ===, !==
&
^
&&
||
?:

Use parenthesis to:

- alter the default rules on the precedence of the operators
- Improve HDL readability

Example

- a+b>>1; performs the **(1)** addition and **(2)** right logical shift
- a+(b>>1); performs the right logical shift of b at first, and then the addition

Expression and bit-length adjustment (1/4)

Problem: In general, signals in SV have different number of bits, i.e., bit lengths or widths

- **SystemVerilog allows to combine signals of different widths in the same statement**
- **The adjustment for mismatching lengths is ruled by a set of rules:**

1) Determine the maximum bit length of the operands in the context[Note] considering both the right- and the left-hand-sides
2) Extend the bit lengths of the operands on the right-hand-side to the maximum
3) Evaluate the expression
4) Assign the result to the left-hand-side signal. Truncate the MSBs if the left-hand-side signal is smaller

Example 1

```
logic [7:0] a,b;
assign a=8'b0000_0000;  //both left- and right-hand-side have the same width, i.e., 8 bits
assign b=0;             //32-bit zero constant is truncated to 8 bits while assigned
```

Note: usually the context is the single SV statement. However, the parenthesis defines sub-contexts

Expression and bit-length adjustment (2/4)

Example 2

```
logic [7:0] a,b; logic [7:0] sum8; logic [8:0] sum9; logic carry;

/* OP1 - 8-bit is the maximum length in the statement. a+b is computed and the carry bit
   is discarded at the time of the assignment to sum8 */
assign sum8=a+b;

/* OP2 - a and b are extended to 9 bits due to sum9. The a+b is performed, and the result
   is correctly assigned to the 9-bit signal sum9*/
assign sum9=a+b;

/* OP3 - a and b are extended to 9 bits due to the length of the concatenation of
   {carry,sum8}. The rest is equivalent to OP2 */
assign {carry,sum8}=a+b;
```

Expression and bit-length adjustment (3/4)

Example 3

```
logic [7:0] a,b,sum1,sum2;
Logic [8:0] sum9_tmp;

/* OP1 - 8-bit is the maximum length in the statement. a+b is computed and the carry bit is
   discarded. When the RLS is performed 0 is shifted into the MSB.*/
assign sum1=(a+b)>>1;

/* OP2 - a and b are extended to 32 bits due to 0 constant. The 0+a+b is performed, and the
   result is a 32bit value. The 32-bit value is RLS by 1 bit and then assigned to sum2 */
assign sum2=(0+a+b)>>1;

/* OP3 – functionally equivalent to OP2 but more readable. a and b are extended to 9 bits
   due to the length of the concatenation of {carry,sum8}. The rest is equivalent to OP2 */
assign sum9_tmp={1'b0,a}+{1'b0,b};
assign sum2=sum_ext[9:1];
```

Expression and bit-length adjustment (4/4)

Guidelines

- SV automatically adjusts bit lengths of the expressions
  - It is a features that can easily lead to bugs and subtle errors. DO NOT LEVERAGE ON IT

- Except for trivial adjustments always adjust the bus-length manually

The module

- Verilog 2001: The Basic “Ingredients” -

The Module declaration

- **Modules are the basic building blocks in Verilog and SystemVerilog**
  - Hardware blocks: made of synthesizable Verilog/SV code
  - Testing blocks: made of non-synthesizable Verilog/SV code
- **Clear separation between synthesizable and non-synthesizable construct**
  - We will discuss coding styles from both families
- **Verilog designs consist of a hierarchy of modules**
  - A module can instantiate other modules as children

```
module dut(in1,in2...,out1,out2...);
    input in1;
    input in2;
    output out;

<impl of the module>

endmodule
```

The module's behavior can be described in many different ways, but it should not matter from outside

Different coding styles for module declaration

```
module(input in1,input in2,…,output o1, output o2);

begin: [optional name]
    [impl of the module];
    ...
end
```

```
module(in1,in2,…,o1, o2);
input in1; input in2; … output o1; output o2;
begin: [optional name]
    [impl of the module];
    ...
end
```

Better for use with parameters in packages

SystemVerilog

– 3 coding styles to describe and verify digital hardware –

Hardware Description: coding styles

Behavioral

- High level description of the functionality
- **Used for:**
  - Non-synthesizable logic /verification infrastructure
  - Efficient high-level description of complex modules
  - Huge optimization space for the synthesizer

Data-Flow

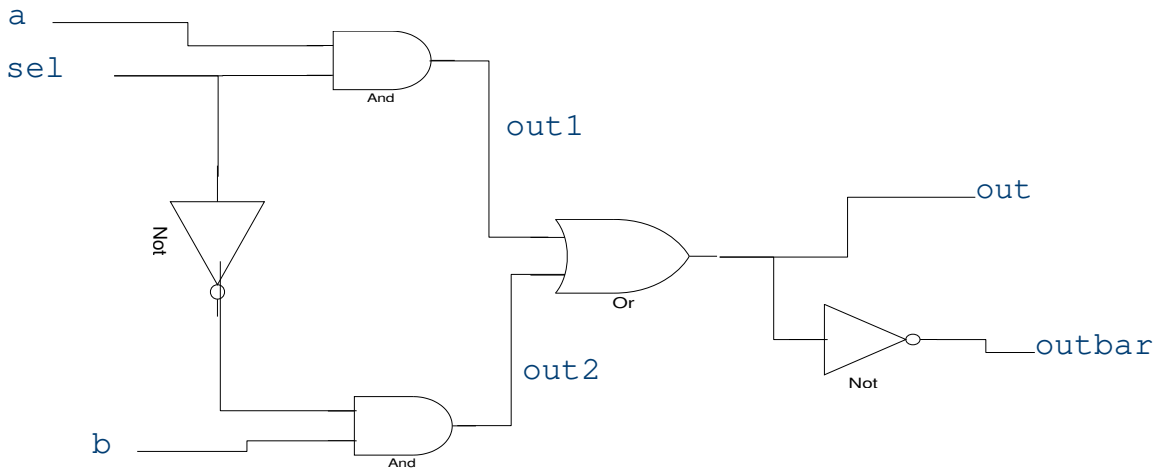
- Combinational logic only
- **Used for:**
  - The design highlights a clear data flow structure

Gate-Level

- Design made of basic gates. Both combinational and sequential logic
- **Used for:**
  - Constraint to a specific implementation

A simple example: the multiplexer (gate-level)

```
module muxgate
   (a,b,out,outbar,sel);

input a,b,sel;
output out,outbar;
wire out1,out2,selb;
   and a1(out1,a,sel);
   not i1(selb,sel);
   and a2(out2, b selb);
   or  o1 (out,out1,out2);
   not i2(outbar,out);
endmodule
```

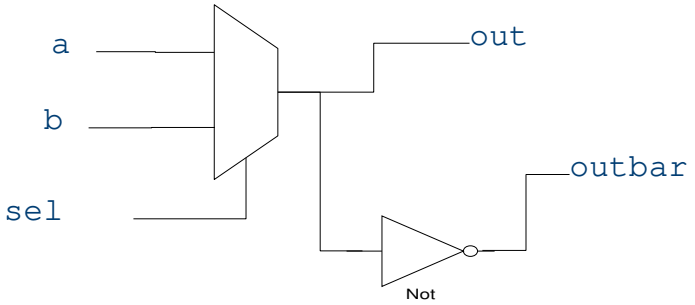


Verilog supports for basic logic gates as primitives:

- and, nand, or, nor, xor, xnor, not, buf
- The net between each pair of gates is represented by means of a wire

A simple example: the multiplexer (dataflow)

```
module muxdataflow (a,b,out,outbar,sel);
input a,b,sel;
output out,outbar;
  assign out = sel ? a : b;
  assign outbar = ~out;
endmodule
```



Continuous assignments by means of the `assign` keyword

- models combinational logic in an easy way
- The left side MUST BE a wire
- The right side CAN BE reg/wire/logic

Dataflow operator

- Conditional operator: (cond_expression) ? (value_if_true) : (value_if_false);
- Arithmetic operator: +, -, (use *, **, /, with care, in synthesizable code)
- Boolean operator: ~, &, | , ^ [unary and binary version]
- Nested conditional operator (4:1 mux)
  - assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);

A simple example: the multiplexer (behavioral)

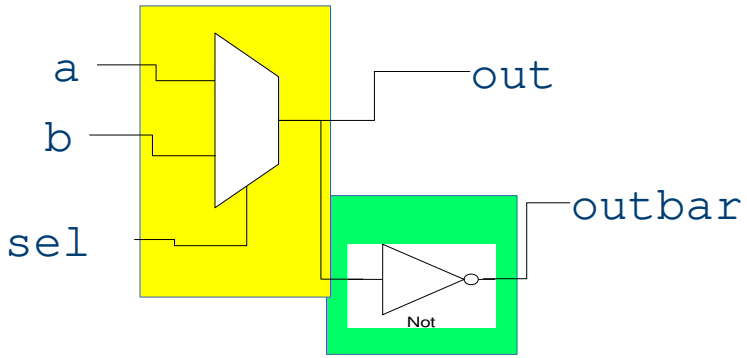
```
module muxbehavioral (a,b,out,
                      outbar,sel);
input a,b,sel;

output reg out, outbar;

always@(a,b,sel)
begin
  if(sel)
       out = a;
  else
       out = b;
  outbar = ~out;
end
endmodule
```

Comments

- Anything assigned in an always block must be declared as **reg**
- always block is processed once whenever a signal in the **sensitivity list** changes value. Since Verilog-2001 the sensitivity list can be substituted with a * to get all the wire/variables to whom the always has to be “sensitive” at
- Statements in the always block are executed sequentially. (Order matters!)
- The begin/end keyword pair is used to group multiple statements within the same always block. Actually they are equivalent with the { / } in C/C++ code

Combine assignments

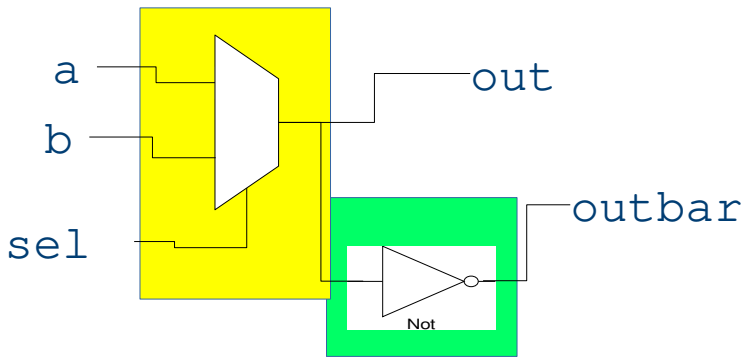
```
module mux (a,b,out,outbar,sel);
input a,b,sel;
output reg out;
output outbar;
  always@(a,b,sel)
  begin
    if(sel) out=a;
    else out=b;
  end
  assign outbar = ~out;
endmodule
```



- **Digital design:** procedural blocks and continuous assignments used together
- **Functional verification**: we maximize the use of procedural blocks
- **Simulation hints:** continuous assignments and procedural blocks are evaluated in parallel

SystemVerilog always_comb for combinational logic

```
module mux (a,b,out,outbar,sel);
input a,b,sel;
output logic out;
output outbar;
  always_comb
  begin
    if(sel) out=a;
    else out=b;
  end
  assign outbar = ~out;
endmodule
```



SystemVerilog defines the **always_comb** process to model combinational logic

Two advantages (compared to traditional Verilog description):

- Sensitivity list is automatically inferred
- Raise a warning in case a sequential circuit is inferred, i.e., checks complete output assignments

The always block for comb circuits (1/2)

Template of a Verilog always block

```
always @([sensitivity_list])
begin: [optional name]
    [procedural statement];
    [procedural statement];
    ...
end
```

- The **[sensitivity_list]** term is a list of signals and events to which the always block reacts
  - **Comb circuits** – all the inputs, i.e., used signals, to the always block must be added
- The **begin/end** delimiter groups the statements in the body of the always block
- The **@([sensitivity_list])** term is a timing control construct. It is the only timing control construct in a synthesizable always block
- The procedural statements (see next slides)

The always block for comb circuits (2/2)

From the **synthesis** point of view:

- The **always block** is a black box
- The behavior of the **always block** is described by its procedural statements

We only analyze the subset of procedural statements meant for synthesis

- (Non-)blocking assignments
- If statement
- Case statement
- Simple loop statement

SystemVerilog introduces 3 always block variants to enhance the quality of the synthesis

- **always_comb** – to infer a combinational circuit
- **always_ff** – to infer a flip flop or a register
- **always_latch** – to infer latches

The `if-else` Statement

```
if [boolean_expr_1]
begin
  [procedural statement]+;
end
else if [boolean_expr_2]
begin
  [procedural statement]+;
end
. . .
else
begin
  [procedural statement]+;
end
```

- The `boolean_expr_1` is evaluated first; if true its procedural statements are executed, otherwise the the first else if expression is evaluated
- It is possible to have multiple cascading `else if`
- The `if` construct infers a priority routing network

The if-else Statement: 4-request priority encoder

```
module prio_enc_if (input [3:0] r, output reg [2:0] y);

 always_comb
  if(r[3]==1'b1)
     y=3'b100;
  else if(r[2]==1'b1)
     y=3'b011;
  else if(r[1]==1'b1)
     y=3'b010;
  else if(r[0]==1'b1)
     y=3'b001;
  else
     y=3'b000;
endmodule
```

Input	Output
1---	100
01--	011
001-	010
0001	001
0000	000

- The code checks the **r[4]** first and eventually cascades to the subsequent else if statements one after the other until one of them matches or the else statement is reached
- Note: both **r** and **y** are 4 values, i.e., **0, 1, Z, X**
- The final else statement allows to complete the **if-else** statement specification to prevent subtle, non-easily detectable bugs

The `case` Statement

```
case [case_expr]
[item1]:
  begin
    [procedural statement]+;
  end
[item2]:
  begin
    [procedural statement]+;
  end
[item3]:
  begin
    [procedural statement]+;
  end
. . .
default:
  begin
    [procedural statement]+;
  end
endcase
```

- The `case` statement is a multi-way decision statement that compares the `[case_expr]` with a number of items and execute the procedural statements of the first matching item
- In the case of multiple matching items only the procedural statements of the first one are executed
- It does not infer a priority network like the `if-else` statement
- The default keyword is matched each time no item is matched. It is not mandatory to specify all the possible items

The `case` Statement: Example

```
module prio_enc_case(input [3:0] r, output
   reg [2:0] y
always@(*)
 case(r)
   4'b1000,4'b1001,4'b1010,4'b1011,4'b1100,4
   'b1101,4'b1110,4'b1111:
      y=3'b100;
   4'b0100,4'b0101,4'b0110,4'b0111:
      y=3'b011;
   4'b0010,4'b0011:
      y=3'b010;
   4'b0001:
      y=3'b001;
   4'b0000:
      y=3'b000;
 endcase
endmodule
```

- The `case` statement can mimic the priority enforced by the `if-else` statement
  - it seems a stretch
- The 0 and 1 values are not the only values the r and items can assume (hints: X and Z)
  - Maybe a more flexible case statement can fit the task if the priority is necessary
  - The non complete specification can be dangerous... (discussed after)

The casez and casex Statements

```
casez (expr)
 3'b1zx: …   // expr[1] dont' care
 3'b01?: …   // expr[0] don't care
 3'b0??: …   // expr[1:0] don't care
endcase
```

```
casex (expr)
 3'b1zx: … // expr[1:0] dont' care
 3'b01?: … // expr[0] don't care.
 3'b0??: … // expr[1:0] don't care
endcase
```

NOTE: *casex* and *casez* can infer a priority network, since multiple items can overlap

- The `casez` statement is a multi-way decision statement that allows the z value and ? character in the item expression as don't-cares
  - It ignores each bit position with a Z
  - Use it for synthesis if necessary
- The `casex` statement is a multi-way decision statement that allows the z and x values as well as the ? character in the item expression as don't-cares.
  - It ignores each bit position with a Z or X
  - don't use the `casex` statement due to its subtleties in treating x values that can thus mask bugs
  - `x` value semantic is different between simulation and synthesis
- **NOTE:**
  - `casex` and `casez` use the identity operator (===)to compare the `case_espr` with each `item`
  - `casex and casez` are symmetric since it does not matter if the special value/character appears in the `item` or in the `case_espr`
  - `?` is a placeholder / character; `X / Z` are values
  - The use of ? In the `item` is preferred than `X /Z`

if-then-else and case: priority and unique attributes

```
1  module tb;
2    logic [4:0] v = 10;
3    initial
4    begin
5      unique case (v)
6        0: $display ("0");
7        1: $display ("1");
8        2: $display ("2");
9      endcase
10     unique if (v == 0 | v == 1)
11       $display ("0 or 1");
12     unique case (v)
13       3       : $display ("3");
14       4       : $display ("4");
15       default: $display ("DEF");
16     endcase
17     priority casez (v)
18       5'b00???: $display ("CASE1");
19       5'b0????: $display ("CASE2");
20     endcase
21   end
22 endmodule
23
24 // OUTPUT:
25 // WARNING: Ons: no condition is true
26 // WARNING: Ons: no condition is true
27 // DEF
28 // CASE2
```

unique and priority - usage rules

- Cannot be used at the same time
- Can be used only with `if...else` and `case` statements
- No warning if `default` statement is used (see `line 15`)

unique

- Exactly one matching branch
- No overlapping/multiple branches allowed
- Otherwise, run-time warning (see `lines 5, 10`)

priority

- At least one matching branch
- Overlapping branches allowed
- If multiple matching branches, only the first one is executed
- Otherwise, run-time warning

Inferred Unintended Latches in Comb. Circuits

```
module mux_wrong(input [1:0] sel, output reg
   [1:0] y
always@(*)
 case(sel)
   2'b00: y=2'b00;
   2'b01: y=2'b01;
   2'b10: y=2'b10;
 endcase
endmodule
```

Use the default special case item

```
module mux_correct(input [1:0] sel, output reg [1:0] y
always@(*)
 case(sel)
   2'b00: y=2'b00;
   2'b01: y=2'b01;
   2'b10: y=2'b10;
   default: y=2'bxx;
 endcase
endmodule
```

- What we have when sel == 2'b11?
  - The old value. Is that what we intended to infer? (probably not!)
  - We inferred an unintended latch to eventually keep the old value
- The `default` special item can help preventing such a behavior
  - Simulation: propagates the X and Z values thus eases the bug detection
  - Synthesis: X values are optimized as treated as don't-care
- The same for the `if-else` statement
  - Use the else statement to catch all not listed possibilities

Routing structures of conditional control constructs

Priority and multiplexing routing networks

Facts

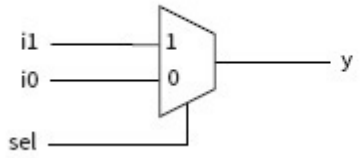
1) SystemVerilog offers several conditional control constructs to design combinational circuits:
   - Continuous assignments - ?: operator
   - Blocking assignments – if/then/else, case

2) Always block allows to the model combinational circuit through a procedural description of the algorithm

3) Combinational circuits have no sequential/procedural control

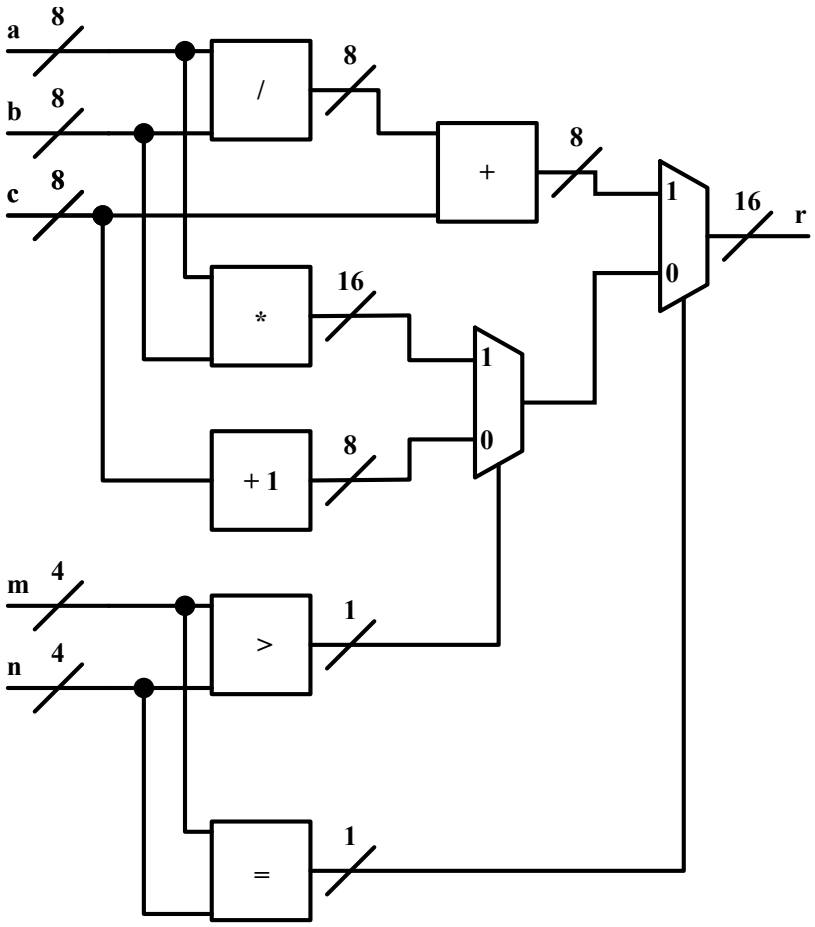
Solution - The hardware implements **routing networks** to synthesize the procedural semantic generated through the use of SV conditional control constructs

Priority routing network



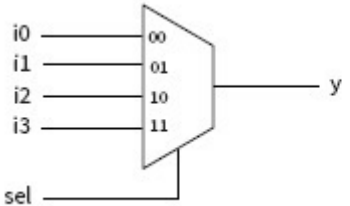
sel	y
0 (false)	i0
1 (true)	i1

```
1  module priority_routing_network (
2      a, b, c, m, n,
3      r
4  );
5      input   logic   [7:0]   a,b,c;
6      input   logic   [3:0]   m,n;
7      output  logic   [15:0]  r;
8
9      always_comb
10     begin
11         if (m == n)
12             r   =   (a / b) + c;
13         else if (m > n)
14             r   =   a * b;
15         else
16             r   =   c + 1;
17     end
18 endmodule: priority_routing_network
```

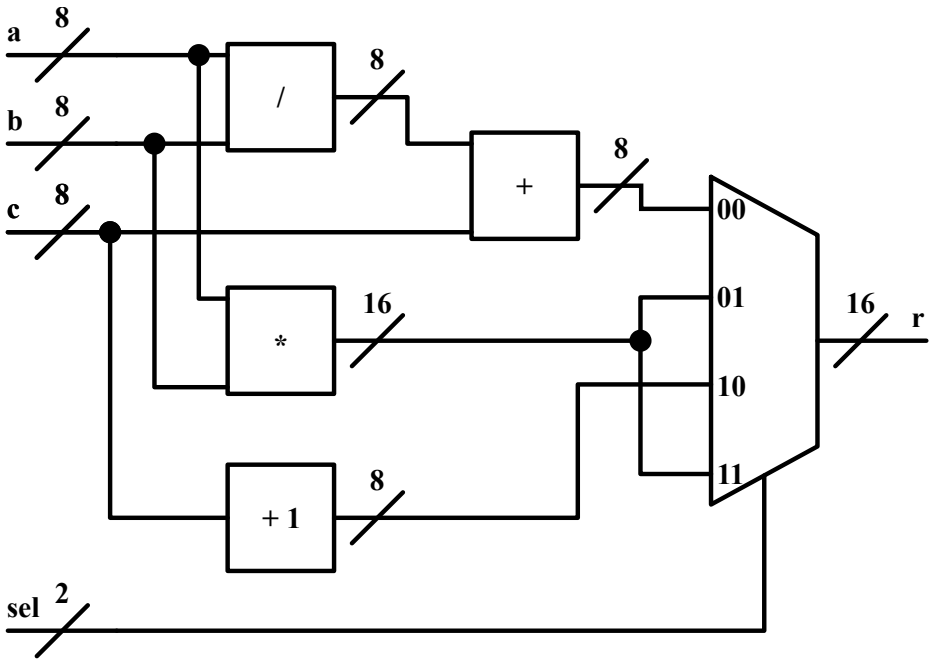


Multiplexing routing network

```
1  module multiplexing_network (
2      a, b, c, sel,
3      r
4  );
5      input   logic   [7:0]   a,b,c;
6      input   logic   [1:0]   sel;
7      output  logic   [15:0]  r;
8
9      always_comb
10     begin
11         case (sel)
12             2'b00:   r = (a / b) + c;
13             2'b10:   r = c + 1;
14             default: r = a * b;
15         endcase
16     end
17 endmodule: multiplexing_network
```



sel	y
00	i0
01	i1
10	i2
11	i3



Guidelines for synthesis of combinational circuits

Guidelines for the synthesis of comb circuits

- SV is meant for both modeling and synthesis -

the always block is the most complex construct that can describe real hardware

3 common errors in modeling combinational circuits for synthesis:

1) Assign signals in multiple always blocks or continuous assignments

2) Incomplete sensitivity lists

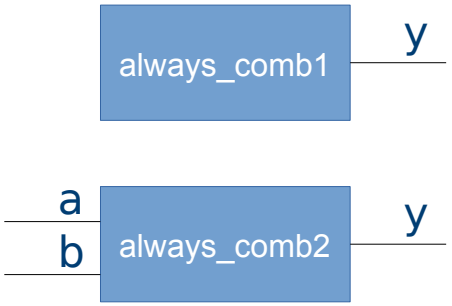
3) Incomplete branch and incomplete output assignments

Guidelines for the synthesis of comb circuits (rule 1)

1) Common error - Signal assigned in multiple always blocks

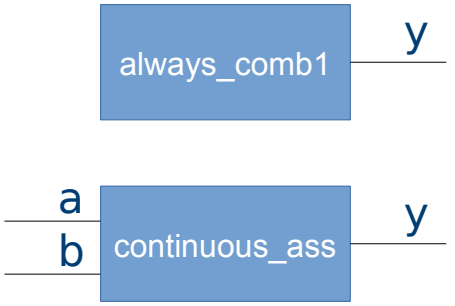
Wrong code 1

```
logic y;
...
always_comb
  y=1'b0;
...
always_comb
  y=a&b;
...
```



Wrong code 2

```
logic y;
...
always_comb
  y=1'b0;
...
assign y=a&b;
...
```



Solution – draw the coarse-grained schematic of the modeled circuit

Guidelines for the synthesis of comb circuits (rule 2)

2) Common error – incomplete sensitivity list

The output of a combinational circuit is function of the inputs of the circuit itself

- If not all inputs are listed in the sensitivity list the HDL code is syntactically correct but functionally not correct

- Verilog allows the **`always@(*)`** to:
  - automatically include all inputs in the sensitivity list
  - **`always@(*)` DOES NOT** check incomplete output assignments

- SystemVerilog introduces **`always_comb`** to:
  - automatically include all inputs in the sensitivity list
  - check incomplete output assignments

```
always@(a)
  y=a&b;
```

Wrong code

In general synthesis tools generate a warning + correct AND gate. However, simulators perform the simulation of the wrong HDL description raising functional a mismatch

```
always@(*)
  y=a&b;
```

Correct Verilog code

```
always_comb
  y=a&b;
```

Correct SV code

Guidelines for the synthesis of comb circuits (rule 3.1)

3) Common error – incomplete branch and output assignment

Combinational circuits should not contain any internal state, i.e., memory elements

- SystemVerilog specifies that:

  a **(1)** non-assigned, **(2)** output signal of an **(3)** always block → **keeps its previous value**

RULES (for correct synthesis of always blocks for comb circuits)

1) include all the if branches/case statements
2) assign a value to every output signal in every branch

```
always_comb
  if(a>b)
    gt = 1'b1;
  else if (a==b)
    eq = 1'b1
```

Wrong code

The example violates both rules …

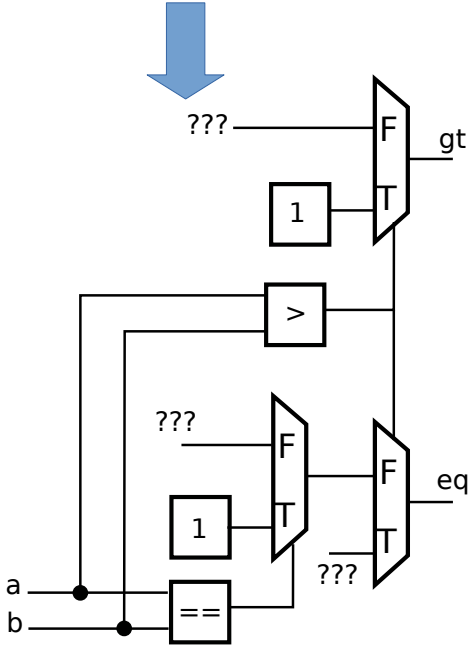
How to check and fix the issues?

Guidelines for the synthesis of comb circuits (rule 3.2)

First rule violation check/fix –

Draw the schematic of the routing network

- For each output signal draw the routing network (from the output)
- If the drawing cannot reach the input signals for all the branches there is a design error



Wrong code (original)

```
always_comb
  if(a>b)
    gt = 1'b1;
  else if (a==b)
    eq = 1'b1
```

Second rule violation check/fix

- identify all outputs in the always block
- Assign a default value to each of them at the beginning of the always block
  - **Note:** alternatively, we can assign all the output signals in all the branches of the always block

```
always_comb
begin
  gt=0;eq=0;
  if(a>b)
    gt = 1'b1;
  else if (a==b)
    eq = 1'b1
end
```

Use `parameters` to improve the design flexibility

```
module mux #(parameter W=2)
      (input [1:0] sel,
       input [W-1:0] a,
       input [W-1:0] b,
       input [W-1:0] c,
       input [W-1:0] d,
       output reg [W-1:0] y);
 always@(*)
  case(sel)
   2'b00: y=a;
   2'b01: y=b;
   2'b10: y=c;
   2'b10: y=d;
   default: y={W{1'bx}};
  endcase
endmodule
```

- The `parameter` keyword allows for a viable way to reuse the same design:
  - Same behavior but different signal widths
  - Passing options to a design at the elaboration time

- **NOTE:** Sometimes the parameter alone is not enough to make the module flexible
  - Hints: think about an N-input mux where N is a parameter. The sel signal width becomes parametric ...

Nested modules and parameters

```
module mux #(parameter W=2)
      (input [1:0] sel,
       input [W-1:0] a, input [W-1:0] b,
       input [W-1:0] c, input [W-1:0] d,
       output reg [W-1:0] y);
. . . //(defined as before)
endmodule

`define WGLOBAL 2
module parent(…)
  wire [`WGLOBAL-1:0] a1,b1,c1,d1,y1;
  wire [1:0] sel1;
  mux #(.W(`WGLOBAL))
      mux0(
       .a(a1), .b(b1), .c(c1), .d(d1),
       .sel(sel1), .y(y1)
       );
 … //rest of the parent module logic
endmodule
```

- The `parameter` keyword allows for a viable way to reuse the same design:
  - Same behavior but different signal widths
  - Passing options to a design at the elaboration time

- **NOTE:** Sometimes the parameter alone is not enough to make the module flexible
  - Hints: think about an N-input mux where N is a parameter. The sel signal width becomes parametric …

Multi-dimensional arrays

SystemVerilog multidimensional arrays (1)

Verilog array

- Collection of variables of the same data type
- Verilog 2001 limits to single dimension arrays declared in the module interfaces.
- SystemVerilog enhances the flexibility and control over memory allocation of the array collection

SystemVerilog unpacked arrays

- Traditional arrays
- Impossible to read/write at slice granularity
- declared as: “type name [width];” e.g. “**reg vect [3:0];**”

SystemVerilog packed arrays

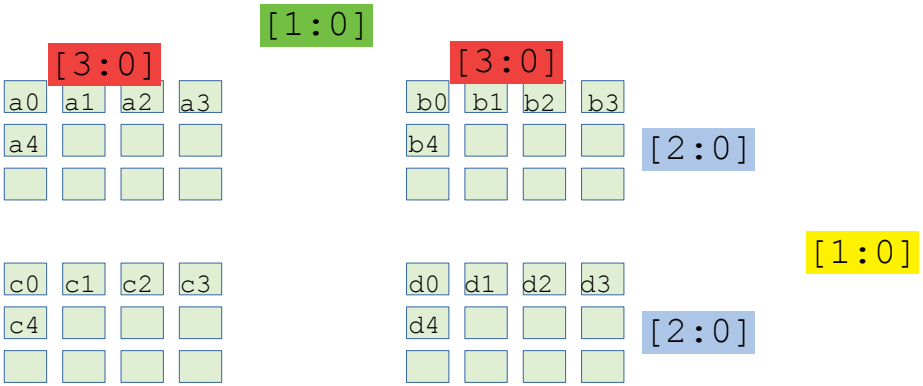
- Array elements are stored contiguously
- Can read/write any element or slice or the whole array
- declared as: “type [width] name;” e.g. “**reg [3:0] vect;**”
- Maximum size of a packed array 2^16

SystemVerilog multidimensional arrays (2)

2D unpacked array of 2D packed array

```
logic [2:0][3:0] a [1:0][1:0]; // also wire or reg
```

packed **unpacked**



Each packed slice is stored in memory in consecutive addresses.
adrA, adrB, adrC, adrD are base addresses in memory

adrA	a0
adrA+1	a1
	a2
	a3
	...
adrA+11	a11
...	...
adrC	c0
adrC+1	c1
	c2
	c3
	...
adrC+11	c11

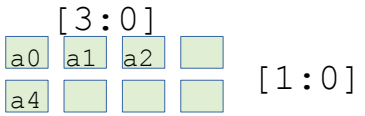
adrB	b0
adrB+1	b1
	b2
	b3
	...
adrB+11	b11
...	...
adrD	d0
adrD+1	d1
	d2
	d3
	...
adrD+11	d11

SystemVerilog multidimentional arrays (3)

Packed vs unpacked arrays

```
module tb;
  reg [1:0][3:0] data; reg [7:0] byteData;
initial begin
  data='0;
  data[1][3:2]='1;
  byteData = data[1];
  #1 $display("byteData=%2h, byteData);
end
endmodule
```

```
module tb;
  reg data [1:0][3:0]; reg [7:0] byteData;
initial begin
  data='0;
  data[1][3:2]='1;          //cannot assign to a slice
  byteData = data[1];       //cannot assign unpacked to packed
  #1 $display("byteData=%2h, byteData);
end
endmodule
```



adrA	a1,0
adrA+1	a1,1
	a1,2
	a1,3
	a0,0
	a0,1
	a0,2
	a0,3

Note:

- Endianness is not important for SystemVerilog since the endianness depends on the declaration [1:0] or [0:1]
- Always use packed arrays when allowed

Replicated structures
(generate-for statement **and** procedural-for statement)

Replicated structures

Many digital systems show a well-patterned structure

- Replica of a functional units in the CPUS
- Replicated cores in multi-cores
- Parametrized combinational elements, e.g., arbiters, ...

SystemVerilog offers 2 constructs to model such scenarios

`generate-for` statement (outside always)

```
generate
  genvar [index_variables];
  for ([initial_assign];[expr];[step_assign])
  begin [:label]
    [concurrent_constructs];
    [concurrent_constructs];
    ...
  end[:label]
endgenerate
```

`procedural-for` statement (inside always)

```
...
for ([initial_assign];[expr];[step_assign])
begin [:label]
  [concurrent_constructs];
  [concurrent_constructs];
  ...
end[:label]
...
```

The generate-for statement

Definition – outside an always block, generate of a set of hardware objects by repeating the loop body for a fixed number of times

Constraints - loop boundaries has to be static

Motivational example – (even if SV has the reduced-xor operator)

Use the generate-for to implement a parametric reduce-xor operator

y=d[0]^d[1]^d[2]^…^d[N-1]

Example – HDL code

```
module xor_gen #(parameter N=8)(d,y);
  input [N-1:0] d;
  output logic y;
  //temp wires
  logic [N-1:0] tmp;
  //setup first stage input
  assign tmp[0]=d[0];
  //replicated structure
  generate
    genvar i;
    for(i=1;i<N;i=i+1)
      assign tmp[i] = d[i]^tmp[i-1];
  endgenerate
  //connect to last stage output
  assign y=tmp[N-1];
endmodule
```

generated statements

```
assign tmp[0]=data[0];
assign tmp[1] = data[1] ^ tmp[0];
assign tmp[2] = data[2] ^ tmp[1];
...
assign tmp[N-1] = data[N-1] ^ tmp[N-2];
assign y=tmp[N-1];
```

The procedural-for statement

Definition – generate of a set of hardware statements by repeating the loop body for a fixed number of times

Constraints – (1) loop boundaries has to be static, **(2)** procedural statement according to the syntax of the always block

Motivational example - procedural-for to implement a parametric reduce-xor operator

y=d[0]^d[1]^d[2]^…^d[N-1]

Example – HDL code

```
module xor_gen #(parameter N=8)(d,y);
  input [N-1:0] d;
  output logic y;
  //temp wires
  logic [N-1:0] tmp;
  //setup first stage input
  assign tmp[0]=d[0];
  //replicated structure
  always_comb
    for(i=1;i<N;i=i+1)
      tmp[i] = d[i]^tmp[i-1];
  //connect to last stage output
  assign y=tmp[N-1];
endmodule
```

generated statements

```
assign tmp[0]=data[0];
-----------------------
always_comb
begin
  tmp[1] = data[1] ^ tmp[0];
  tmp[2] = data[2] ^ tmp[1];
  ...
  tmp[N-1] = data[N-1] ^ tmp[N-2];
end
-----------------------
assign y=tmp[N-1];
```

Guidelines: generate-for and procedural-for (1/2)

Generate-for

- powerful construct to make the design truly parametric and flexible
- Within the generate-for the for-loop is optional
- Within the generate-for we can insert:
  - always blocks
  - continuous assignments
  - Instantiate modules
  - `ifdef and if-else to further steer the hardware generation
- It is important to correctly solve the wiring problem, i.e., correctly assign the wires to each and all generated statements
  - p[i]=s[i]
- The naming problem in CAD tools

generate-for - template

```
parameter IMPL_MOD = "mod"
module xor_gen #(parameter N=8)(...);
  //temp wires
  logic [...] in [N-1:0];
  logic [...] out [N-1:0];
  //replicated structure
  generate
    genvar i;
    if (IMPL_MOD = "mod")
      for(i=0;i<N;i=i+1)
        mod mod1(in[i],out[i]);
    else
      for(i=0;i<N;i=i+1)
        assign out[i] = 0;
  endgenerate
  //connections between modules
  ...
endmodule
```

Guidelines: generate-for and procedural-for (2/2)

Procedural-for

- Works inside the always block to create procedural statements
- The naming problem in CAD tools
- In general, used to initialize signals

generate-for - template

```
//`define IMPL_MOD
module xor_gen #(parameter N=8)(...);
  //temp wires
  logic [...] tmp [N-1:0];
  //replicated structure
  always_comb
     for(i=0;i<N;i=i+1)
       tmp[i] = i;
  //connections between modules
  ...
endmodule
```