Verilog

VLSI Systems Design
SS23 | CEID, UPatras

Useful Links

- https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md
- https://github.com/lowRISC/style-guides/blob/master/DVCodingStyle.md
- http://www.sunburst-design.com/papers/CummingsSNUG2002Boston_NBAwithDelays.pdf

Introduction to Verilog

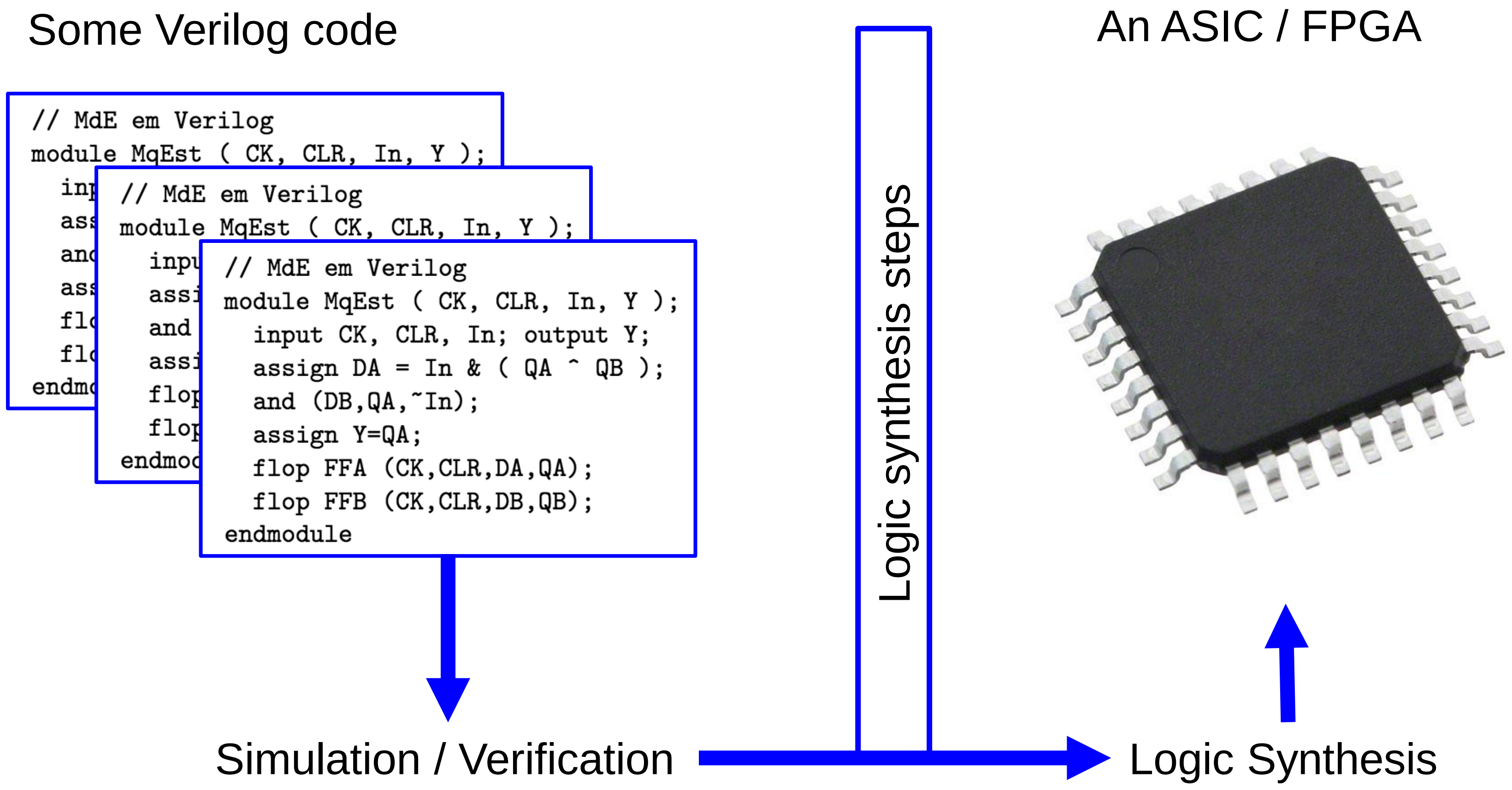
Definition of Verilog

- The language that describes the hardware functionality is called Verilog. Verilog is a Hardware Description Language(HDL)
  - Verilog is a language, it has syntax, variables, loops and other language-related characteristics
  - Verilog code “translates” to a hardware (e.g., ASIC, FPGA)

- Verilog is the specification language for logic synthesis
  - Verilog can be simulated
  - Verilog goes through some stages to be fully “translated” into a circuit. These steps are the logic synthesis steps

- Verilog is a tool that allows to abstract away from the detailed implementation of the circuit
  - If users know the functional description of the circuit only, then they can create the description in Verilog and synthesize the hardware from it

Definition of Verilog (2)

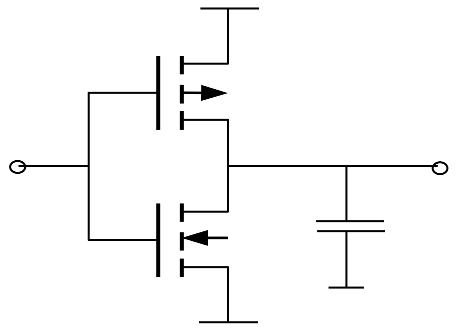
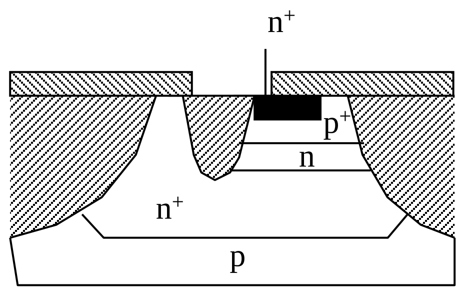


```
// MdE em Verilog
module MqEst ( CK, CLR, In, Y );
  input CK, CLR, In; output Y;
  assign DA = In & ( QA ^ QB );
  and (DB,QA,~In);
  assign Y=QA;
  flop FFA (CK,CLR,DA,QA);
  flop FFB (CK,CLR,DB,QB);
endmodule
```

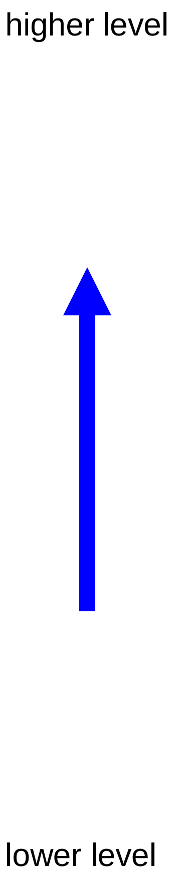
Design Levels

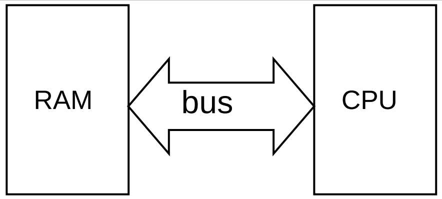
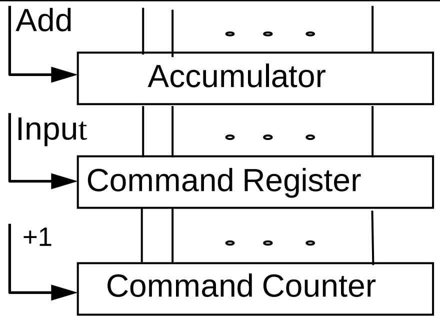
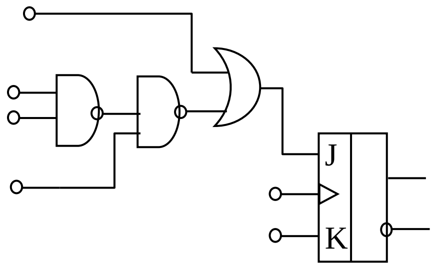
higher level

lower level

Level	Modeling Object	Example of Modeling Object
Circuit	Electrical Circuit	
Device	IC Components	

Design Levels (2)

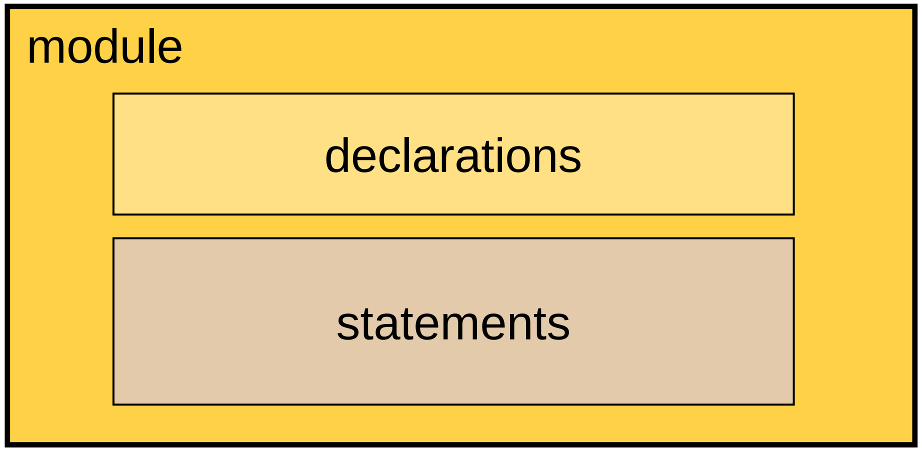


Level	Modeling Object	Example of Modeling Object
System	Structural Circuit	
Register-Transfer	Functional Circuits on the level of multibit devices	 Add Accumulator Input Command Register +1 Command Counter
Gate	Circuit on the level of gates and flip-flops	

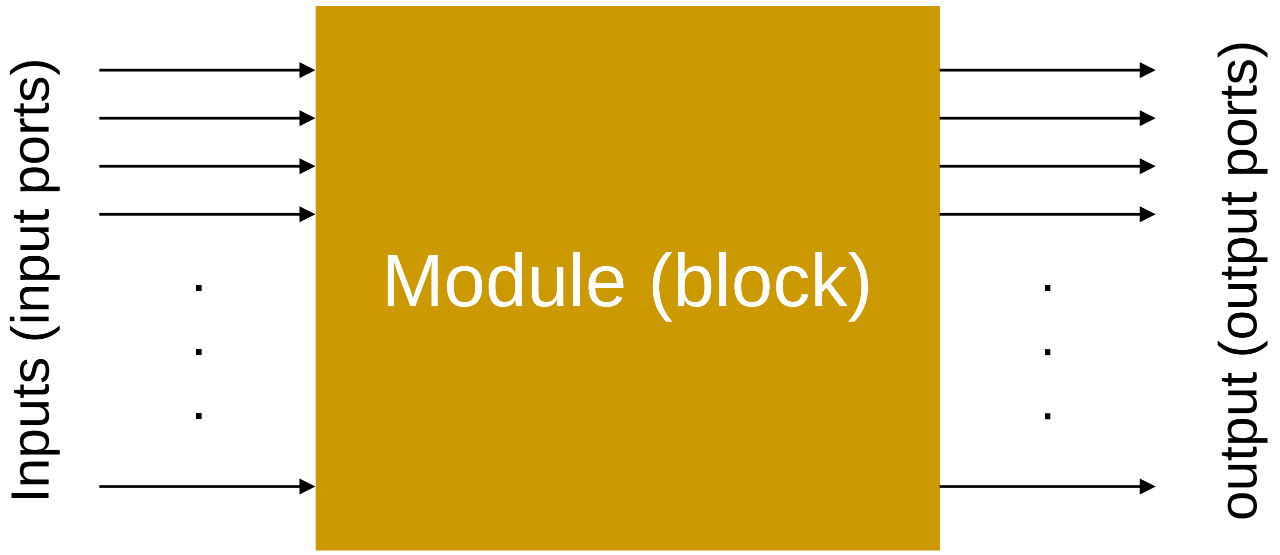
Basic concepts, syntax

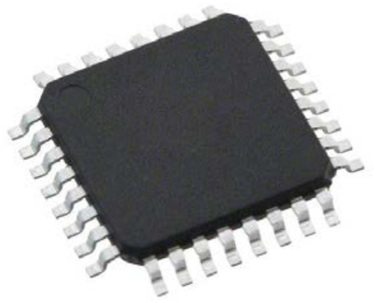
Basic Concepts

- Basic Unit – A module
  - Everything is a module in Verilog
- The Module
  - Describes the functionality of the design
  - States the input and output ports, contains variable declarations and statements



Basic concepts (2)




The block diagram of a module looks like a packaged circuit

- The module is a block of Verilog code
- It can hide the internal implementation
- Can contain other modules

The Syntax of Module Declaration

```
module  module_name(a, b, c);
        input  a, b;
        output c;
        inout declarations;
        net declarations;
        variable declarations;
        parameter declarations;
        function declarations;
        task declarations;

        behavior/functionality description; // c = a and b

endmodule
```

a semicolon…

can be changed

declarations section

the circuit description

Keywords are labeled in blue color.

More keywords (reference)

always	else	input	not
and	end	integer	or
assign	endmodule	module	output
begin	for	nand	parameter
case	If	nor	real
posedge	negedge	forever	repeat
reg	wire	endcase	initial

Data types, parameter types

Data Types

- Net Types: Physical Connection between structural elements
  - wire, tri, wor, trior, wand, triand, supply0, supply1
- Register Type: Represents an abstract storage element
  - reg, integer, time, real, realtime
- Default Values
  - 4 values

Two Main Data Types

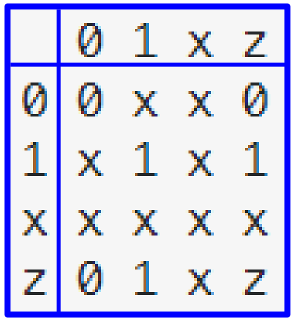
- wire: Nets represent connections between structural elements
  - Do not hold their value
  - Take their value from a driver such as a gate or other module
  - Cannot be assigned in *some* blocks

- reg: Regs represent data storage
  - Behave exactly like memory in a computer
  - Hold their value until explicitly assigned
  - Never connected to something
  - Can be used to model latches, flip-flops, etc., but do not correspond exactly

Values in Verilog

Verilog is a 4 valued language

- {0, 1}
  - Logic 0 and 1. As Verilog describes the behavior of a digital circuit, then its variables can hold digital values
- Z
  - Output of an undriven I/O
  - The case where nothing is setting a wire’s value
- X
  - Models the case where the simulator can’t decide the value of the net
  - Initial state of registers
  - When a wire is being driven to 0 and 1 simultaneously
  - Output of a gate with z inputs



	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Value Representation in Verilog

Numbers are specified by

<size>'<base><number>

<base> - binary (b), decimal (d), hexadecimal (h), octal(o)

5’d3	Equals to 5’b00011
16’h4xa	Equals to 16’b0000_0100_xxxx_1010
6’oz5	Equals to 6’bzzz101
-8’d7	2’s complement of 7, held in 8 bits

An x declares 4 unknown bits in hexadecimal, 3 in octal and 1 in binary. z declares high impedance values similarly.

Alternatively, z, when used in numbers, can be written as ? This is advised in case expressions to enhance readability.

Value Representation in Verilog (2)

base	binary	decimal	octal	hexadecimal
digits	0,1	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
representation	1’b1	2’d3	5’07	8’hA 9
representation in binary	1	11	00 $\boxed{111}$	$\boxed{1010}$ $\boxed{1001}$

The base and the bit number do not have to correspond!
e.g., there can be an octal number with only 6-bit length

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	:
'b10	:
4'b100	:
4'bx	:
8'hfx	:
16'h8xc	:
5'd3	:
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	:
4'b100	:
4'bx	:
8'hfx	:
16'h8xc	:
5'd3	:
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number $10=2_{10}$
4'b100	:
4'bx	:
8'hfx	:
16'h8xc	:
5'd3	:
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number $10=2_{10}$
4'b100	: binary number $0100=4_{10}$
4'bx	:
8'hfx	:
16'h8xc	:
5'd3	:
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number 10=2_{10}
4'b100	: binary number 0100=4_{10}
4'bx	: xxxx
8'hfx	:
16'h8xc	:
5'd3	:
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number 10=2_{10}
4'b100	: binary number 0100=4_{10}
4'bx	: xxxx
8'hfx	: 1111xxxx
16'h8xc	:
5'd3	:
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number $10=2_{10}$
4'b100	: binary number $0100=4_{10}$
4'bx	: xxxx
8'hfx	: 1111xxxx
16'h8xc	: 0000 1000 xxxx 1100
5'd3	:
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number $10=2_{10}$
4'b100	: binary number $0100=4_{10}$
4'bx	: xxxx
8'hfx	: 1111xxxx
16'h8xc	: 0000 1000 xxxx 1100
5'd3	: 00011
-8'd3	:
-12’h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number $10=2_{10}$
4'b100	: binary number $0100=4_{10}$
4'bx	: xxxx
8'hfx	: 1111xxxx
16'h8xc	: 0000 1000 xxxx 1100
5'd3	: 00011
-8'd3	: 1111 1101 (two's complement -3)
-12'h50	:

Value Representation in Verilog: Examples

0	: number 0
1	: number 1
10	: decimal number 10
'b10	: binary number $10=2_{10}$
4'b100	: binary number $0100=4_{10}$
4'bx	: xxxx
8'hfx	: 1111xxxx
16'h8xc	: 0000 1000 xxxx 1100
5'd3	: 00011
-8'd3	: 1111 1101 (two’s complement -3)
-12’h50	: 111110110000 (two’s complement -80)

Module Example

- General definition

```
module module_name ( port_list );
    port declarations;
    …
    variable declaration;
    …
    description of behavior
endmodule
```

- Example

```
module half_adder(a, b, sum, carry);
    input a, b;
    output sum, carry;

    assign sum = a ^ b;  // ^ denotes XOR
    assign carry = a & b; // & denotes AND
endmodule
```

1-bit wires

Module Example

- **The following two codes are functionally identical**

```
module test ( a, b, y );
  input  a;
  input  b;
  output y;

endmodule
```

```
module test (
  input  a,
  input  b,
  output y
);
endmodule
```

port name and direction declaration can be combined

Module Example

- The following two codes are functionally identical

```
module test ( a, b, y );
  input  a;
  input  b;
  output y;

endmodule
```

```
module test (
  input  a,
  input  b,
  output y
);
endmodule
```

port name and direction declaration can be combined

Parameters, numbers, strings

- Parameters
  - Constants in Verilog are called parameters
  - parameter is a keyword
- Numbers (integer & real)
  - Integer is a variable with 32-bits length. It holds an integer value
  - Real is a variable that can store floating-point numbers
- Strings
  - Are stored in regs
  - Each string character is an ASCII value and has a length of 1 byte(8bits)
  - The reg has to be large enough to store a string(e.g., 4-character string requires 4 bytes of length, i.e., 4*8 bits)

Parameters, numbers, strings: Declaration

- Parameters

  ```
  parameter a = 5;
  parameter var = 9.5;
  ```

- Numbers (integer & real)

  ```
  integer num = 32'b1; // num = 784
  real var = 10.24;
  ```

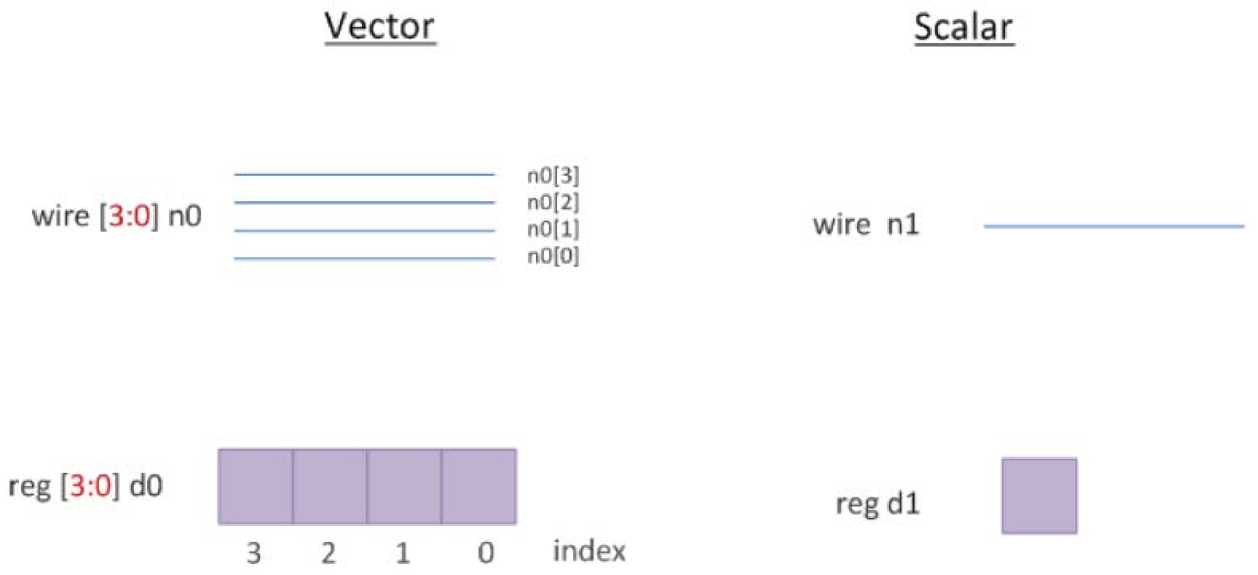
- Strings

  ```
  reg[8*12:1] str_var = "Hello World!"
  ```

- Comments
  - // single line comment
  - /* multi
    line comment*/

Scalars & Vectors

- Scalars are 1-bit wide wire or reg declarations without a range specification
- Vectors are wire and reg declarations with a range specification. They have “width”.



image(c) chipverify.com

Vectors: Declaration

type [msb:lsb] variable_name;

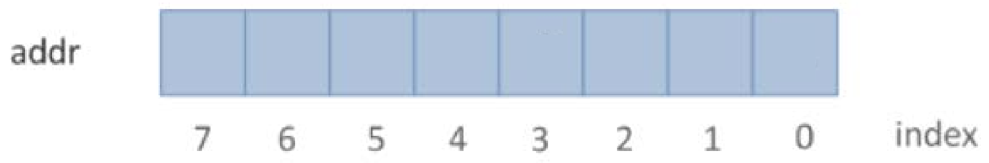
- wire [12:0] s = 12; // 32-bit decimal number → "truncated" to 13 bits
- wire [12:0] z = 13’d12; // 13-bit decimal number
- wire [3:0] t = 4'b0101; // 4-bit binary number
- wire [63:0] u = 64'hdeadbeefcafebabe; // 64-bit hexadecimal number
- reg out; // 1 bit register
- reg [3:0] out; // 4-bit register
- reg [0:100] n; // 101-bit register
- integer loop_count; // loop_count is a 32-bit integer

Can also be explicitly declared as signed numbers

- reg signed [3:0] a; // 4-bit signed register
- wire signed [15:0] s; // 16-bit signed wire

Vectors: <u>bit-select</u> and part-select

```
wire [7:0] addr;
addr = 8'b10011100;
```



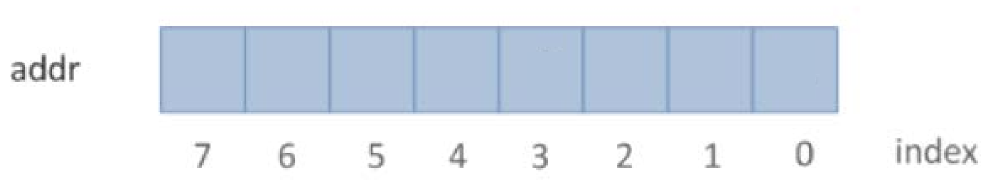
```
wire [7:0] addr;
addr[0] = 1'b1;
addr[3] = 1'b0;
addr[10] = 1'b1;
```

image(c) chipverify.com

Vectors: bit-select and part-select

```
wire [7:0] addr;
addr = 8’b10011100;
```

declare an 8-bit wire



```
wire [7:0] addr;
addr[0] = 1’b1;
addr[3] = 1’b0;
addr[10] = 1’b1;
```

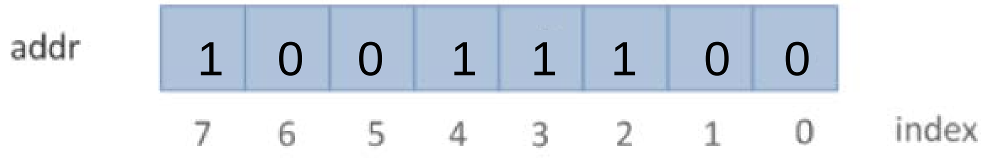
image(c) chipverify.com

Vectors: <u>bit-select</u> and part-select

```
wire [7:0] addr;
addr = 8’b10011100;
```

declare an 8-bit wire

assign an 8-bit value



```
wire [7:0] addr;
addr[0] = 1’b1;
addr[3] = 1’b0;
addr[10] = 1’b1;
```

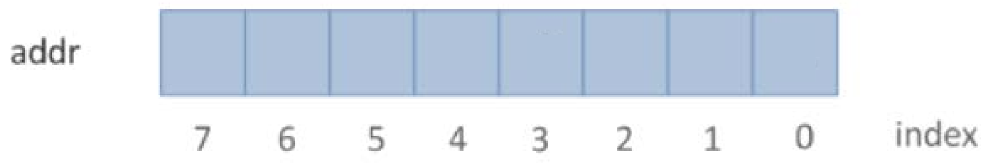
image(c) chipverify.com

Vectors: bit-select and part-select

```
wire [7:0] addr;
addr = 8'b10011100;
```

declare an 8-bit wire

assign an 8-bit value



```
wire [7:0] addr;
addr[0] = 1'b1;
addr[3] = 1'b0;
addr[10] = 1'b1;
```

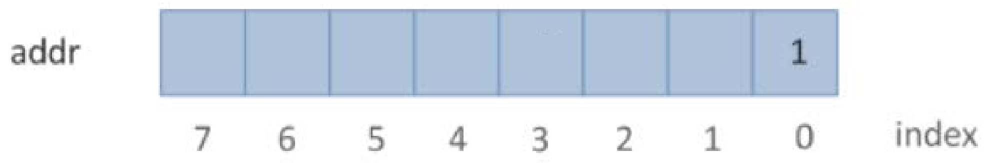
image(c) chipverify.com

Vectors: bit-select and part-select

```
wire [7:0] addr;
addr = 8'b10011100;
```

declare an 8-bit wire

assign an 8-bit value



```
wire [7:0] addr;
addr[0] = 1'b1;
addr[3] = 1'b0;
addr[10] = 1'b1;
```

assign 1 to 0th bit of addr

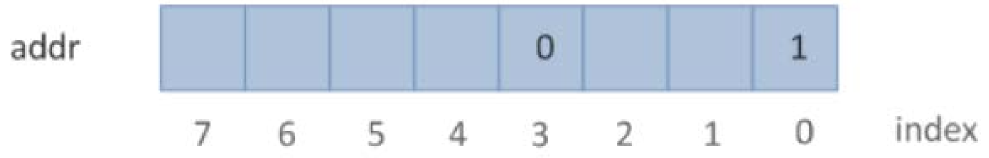
image(c) chipverify.com

Vectors: bit-select and part-select

```
wire [7:0] addr;
addr = 8’b10011100;
```

declare an 8-bit wire

assign an 8-bit value



```
wire [7:0] addr;
addr[0] = 1’b1;
addr[3] = 1’b0;
addr[10] = 1’b1;
```

assign 1 to 0th bit of addr

assign 0 to 3rd bit of addr

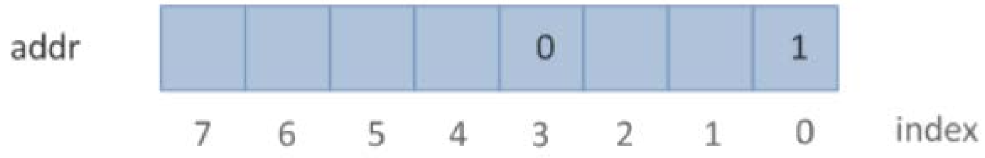
image(c) chipverify.com

Vectors: bit-select and part-select

```
wire [7:0] addr;
addr = 8'b10011100;
```

declare an 8-bit wire

assign an 8-bit value



```
wire [7:0] addr;
addr[0] = 1'b1;
addr[3] = 1'b0;
addr[10] = 1'b1;
```

assign 1 to 0th bit of addr

assign 0 to 3rd bit of addr

no 10th bit to assign a value, error

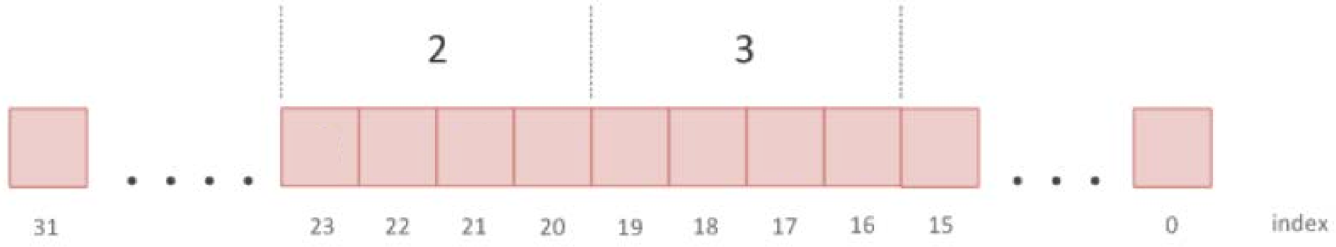
image(c) chipverify.com

Vectors: bit-select and part-select (1)

```
wire [31:0] addr;
addr = 32'b1;
```

declare a 32-bit wire

assign a 32-bit value



```
wire [31:0] addr;
addr[23:16] = 8'h23;
```

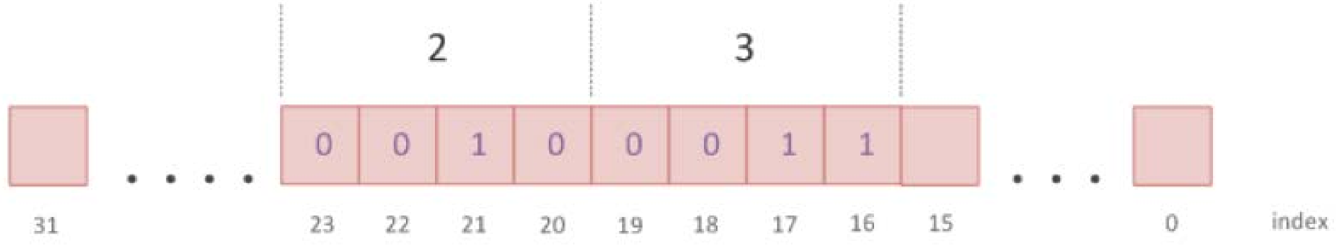
image(c) chipverify.com

Vectors: bit-select and part-select (1)

```
wire [31:0] addr;
addr = 32’b1;
```

declare a 32-bit wire

assign a 32-bit value



```
wire [31:0] addr;
addr[23:16] = 8’h23;
```

assign 8’h23 to 16-23 bits of addr

image(c) chipverify.com

Vectors: bit-select and part-select (2)

[start-bit +:width]

[start-bit -:width]

Vectors: bit-select and part-select (2)

[start-bit +:width]

starting(increment) from a <start-bit>, take a chunk of a vector with <width> length

[start-bit -:width]

Vectors: bit-select and part-select (2)

[start-bit +:width]

starting(increment) from a <start-bit>, take a chunk of a vector with <width> length

[start-bit -:width]

starting(decrement) from a <start-bit>, take a chunk of a vector with <width> length

Vectors: bit-select and part-select (2)

```
[start-bit +:width]
[start-bit -:width]
```

starting(increment) from a <start-bit>, take a chunk of a vector with <width> length

starting(decrement) from a <start-bit>, take a chunk of a vector with <width> length

```
reg [31:0] addr;
parameter num = 1;
addr = 32'hCAFE_FACE;

addr[0 +:8];
addr[8 * num +:8];
addr[31 -:8];
```

Vectors: bit-select and part-select (2)

[start-bit +:width]

[start-bit -:width]

starting(increment) from a <start-bit>, take a chunk of a vector with <width> length

starting(decrement) from a <start-bit>, take a chunk of a vector with <width> length

```
reg [31:0] addr;
parameter num = 1;
addr = 32'hCAFE_FACE;

addr[0 +:8];
addr[8 * num +:8];
addr[31 -:8];
```

first 1 byte from lsb -> 8'hCE

Vectors: bit-select and part-select (2)

[start-bit +:width]

[start-bit -:width]

starting(increment) from a <start-bit>, take a chunk of a vector with <width> length

starting(decrement) from a <start-bit>, take a chunk of a vector with <width> length

```
reg [31:0] addr;
parameter num = 1;
addr = 32'hCAFE_FACE;

addr[0 +:8];
addr[8 * num +:8];
addr[31 -:8];
```

first 1 byte from lsb -> 8'hCE

second 1 byte from lsb -> 8'hFA

Vectors: bit-select and part-select (2)

```
[start-bit +:width]
[start-bit -:width]
```

starting(increment) from a <start-bit>, take a chunk of a vector with <width> length

starting(decrement) from a <start-bit>, take a chunk of a vector with <width> length

```
reg [31:0] addr;
parameter num = 1;
addr = 32'hCAFE_FACE;

addr[0 +:8];
addr[8 * num +:8];
addr[31 -:8];
```

first 1 byte from lsb -> 8'hCE

second 1 byte from lsb -> 8'hFA

first 1 byte from msb -> 8'hCA

Arrays

Arrays allow to create multi-dimensional objects in Verilog

reg var [7:0];

var is a **scalar** with an 8-bit depth

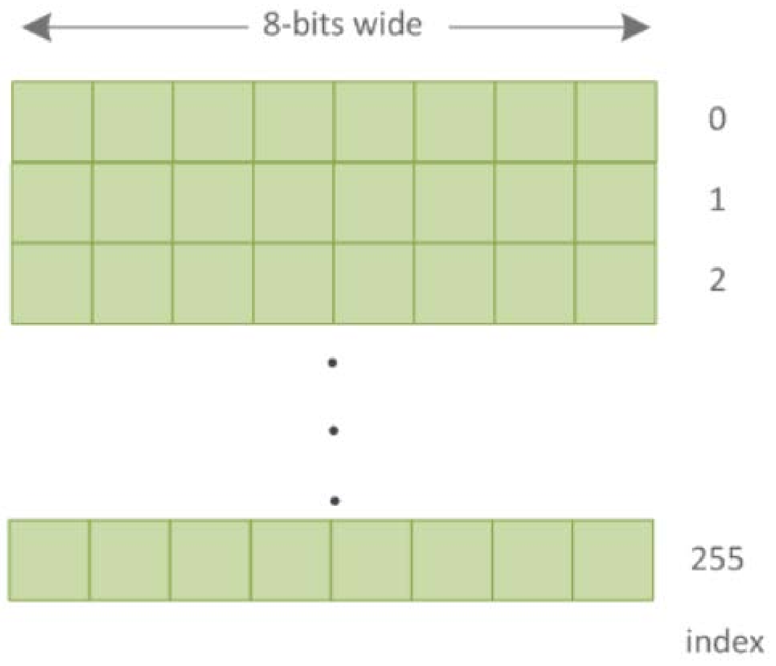
wire [3:0] wires [7:0];

wires is a 4-bit **vector** with an 8-bit depth

wire [3:0] nets [7:0][15:0];

nets is a 4-bit **vector** 2D array where columns=8 & rows=16

reg [7:0] addr [255:0];



image(c) chipverify.com

Verilog Operators

Verilog Operators

- Arithmetic
  - These are the operators that help to perform some for of arithmetic operation
- Relational
  - These operators help to compare the values and make decisions in code based on the comparison
- Logical
  - Logical operators check for logical 0 or 1 result for the operation
- Bitwise
  - These operators perform bit-by-bit operation on the values
- Shift
  - Shift operators perform bit-shifting operation on numbers
- Concatenation
  - Allows to concatenate the bits

Verilog Operators (2)

Type	Symbol	Description	Note
Arithmetic	+	add	
	-	subtract	
	*	multiply	
	/	divide	may not synthesize
	%	modulus (remainder)	may not synthesize
	**	power	may not synthesize

Verilog Operators (3)

Type	Symbol	Description	Note
Bitwise	~	not	
	|	or	
	&	and	
	^	xor	
	~& or &~	nand	mix two operators
Relational	>	greater than	
	<	less than	
	>=	greater than or equal	
	<=	less than or equal	
	==	equal	
	!=	not equal	

Verilog Operators (4)

Type	Symbol	Description	Note
Logical	!	negation	
	||	logical OR	
	&&	logical AND	
	==	logical equality	
Shift operators	>>	right shift	
	<<	left shift	
	>>>	right shift with MSB shifted to right (sign)	
	<<<	same as <<	
Concatenation	{…, …}	concatenates bits	{1’b1, 1’b0}

Verilog Operators: Examples

```
module test;
        reg [7:0] num1;
        reg [7:0] num2;

                …
                num1 = 5;
                num2 = 7;
                $display("Addition: %d", num1 + num2);
                $display("Subtraction: %d", num1 - num2);
                $display("And: %d", num1 & num2);
                $display("Power: %d", num1 ** num2);
                …
endmodule
```

Verilog Operators: Examples

```
module test;
        reg [7:0] num1;
        reg [7:0] num2;

                …
                num1 = 5;
                num2 = 7;
                $display(“Addition: %d”, num1 + num2);
                $display(“Subtraction: %d”, num1 - num2);
                $display(“And: %d”, num1 & num2);
                $display(“Power: %d”, num1 ** num2);
                …
endmodule
```

no ports for this block

Verilog Operators: Examples

```
module test;
    reg [7:0] num1;
    reg [7:0] num2;

        …
        num1 = 5;
        num2 = 7;
        $display("Addition: %d", num1 + num2);
        $display("Subtraction: %d", num1 - num2);
        $display("And: %d", num1 & num2);
        $display("Power: %d", num1 ** num2);
        …
endmodule
```

no ports for this block

Verilog system task $display

Verilog Operators: Examples

```
module test;
        reg [7:0] num1;
        reg [7:0] num2;

                …
                num1 = 5;
                num2 = 7;
                $display(“Addition: %d”, num1 + num2);
                $display(“Subtraction: %d”, num1 - num2);
                $display(“And: %d”, num1 & num2);
                $display(“Power: %d”, num1 ** num2);
                …
endmodule
```

no ports for this block

Verilog system task $display

a placeholder for a number

Verilog Operator Precedence

In case we use more then one operator, the operators execute in special order:

1. Bitwise/logical negation, and, or, xor, etc.
   - ~, !, &, |, ^, etc.
2. Concatenation {}
3. Multiply, divide, modulus *, /, %
4. Arithmetic plus, minus +, -
5. Shift operations <<, >>
6. Relational <, >, <=, >=
7. Relational ==, !=

Verilog Operator Precedence

Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	logical bit-wise reduction reduction reduction reduction reduction reduction
+ -	unary (sign) plus unary (sign) minus	arithmetic arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
* / %	multiply divide modulus	arithmetic arithmetic arithmetic
+ -	binary plus binary minus	arithmetic arithmetic
<< >>	shift left shift right	shift shift
> >= < <=	greater than greater than or equal to less than less than or equal to	relational relational relational relational
== !=	logical equality logical inequality	equality equality

===
!==
	case equality
	case inequality
		equality
		equality

&	bit-wise AND	bit-wise
^ ^~/~^	bit-wise XOR bit-wise XNOR	bit-wise bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

src:https://class.ece.uw.edu/cadta/verilog/operators.html

Verilog Assignments

Verilog Assignments

If we want to “write” a value to a net or a register, we need to assign a value to it. There are two main types of assignments in Verilog:

- **Continuous**
- **Procedural**

Assignment general schema:

LHS = RHS

```
wire a = 1’b1;
```

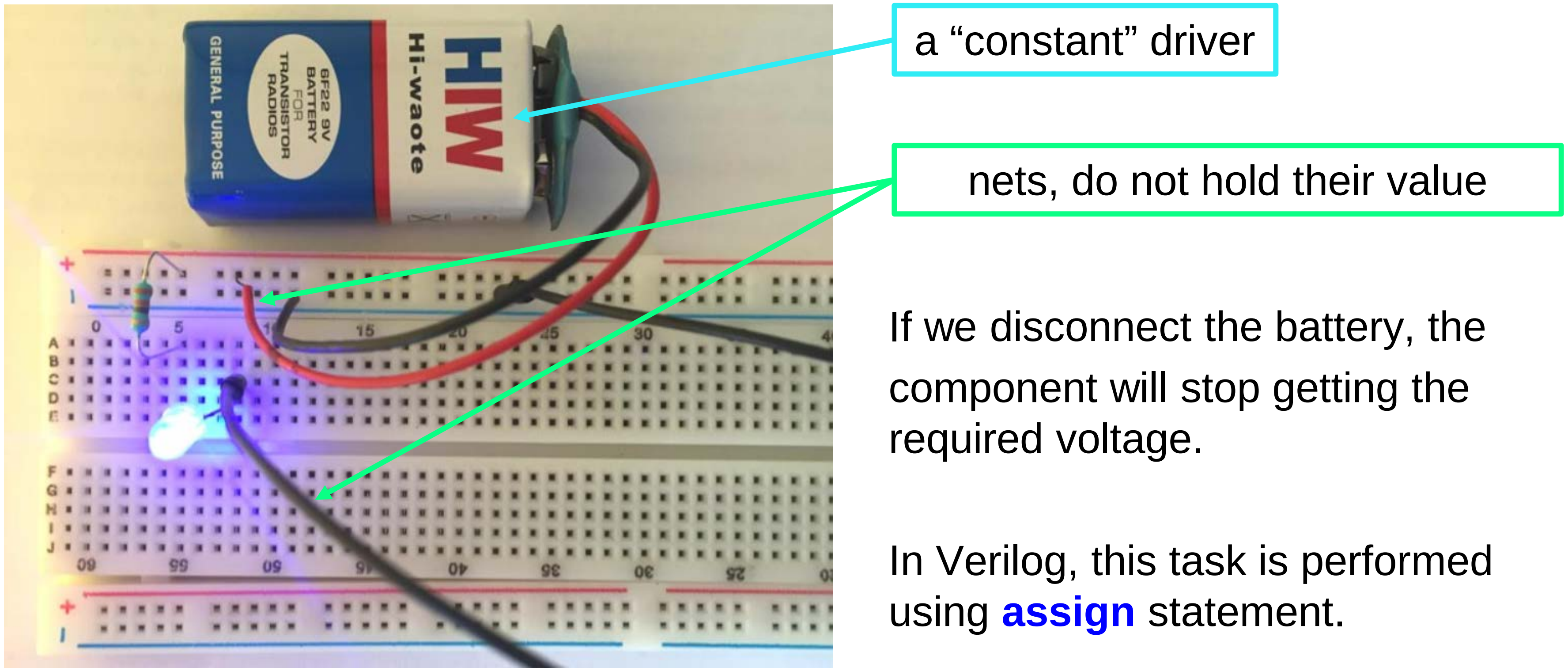
LHS = Left Hand Side
a wire or a reg

RHS = Right Hand Side
a value or an expression
that evaluates into a value

Verilog Assignments and Blocks: assign

Verilog "assign" statement

Assignment of some value to a net(to a wire) requires the assignment to be performed **continuously**.



If we disconnect the battery, the component will stop getting the required voltage.

In Verilog, this task is performed using **assign** statement.

image(c) circuitdigest.com

Verilog “assign” statement (2)

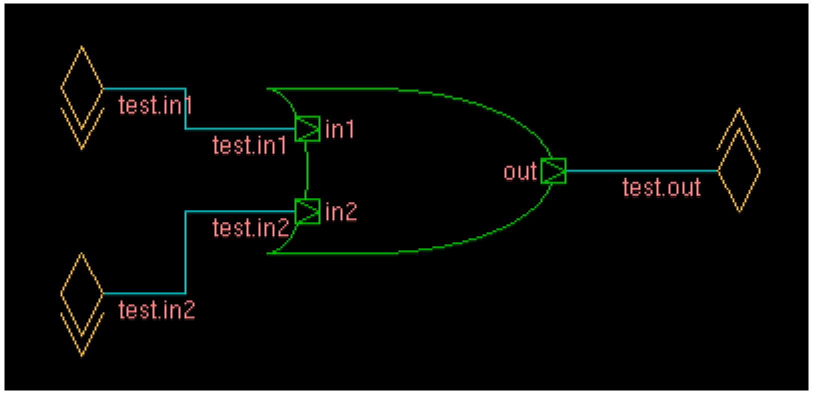
The generic syntax of assign statement:

assign <a net> **=** <expression or a constant value>;

- Starts with **assign** keyword
- LHS is a scalar or vector **net(wire)**, it can’t be a register
- RHS can be a wire or a reg
- The assign statements perform assignment whenever the RHS value changes
- Assign statement is an assignment of a continuous type and is always active during execution
- If there are multiple assign statements in a Verilog code, they all execute in parallel

Verilog “assign” statement (2)

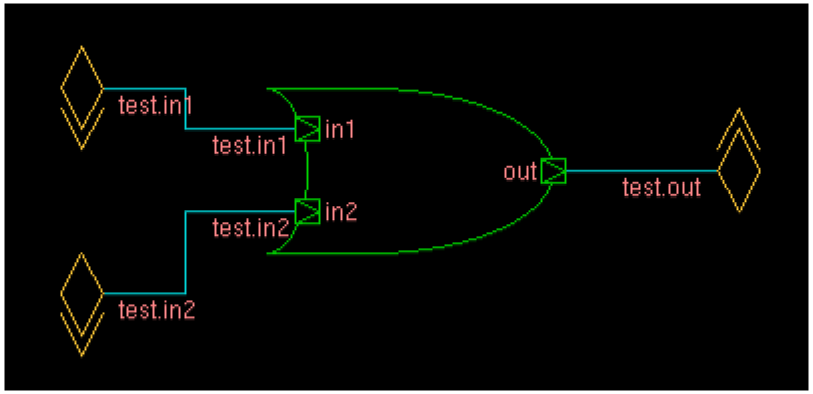
```
module test(
  input in1, in2,
  output out
);
        assign out = in1 | in2;
endmodule
```



Verilog "assign" statement (2)

```
module test(
  input in1, in2,
  output out
);
        assign out = in1 | in2;
endmodule
```

wire declarations

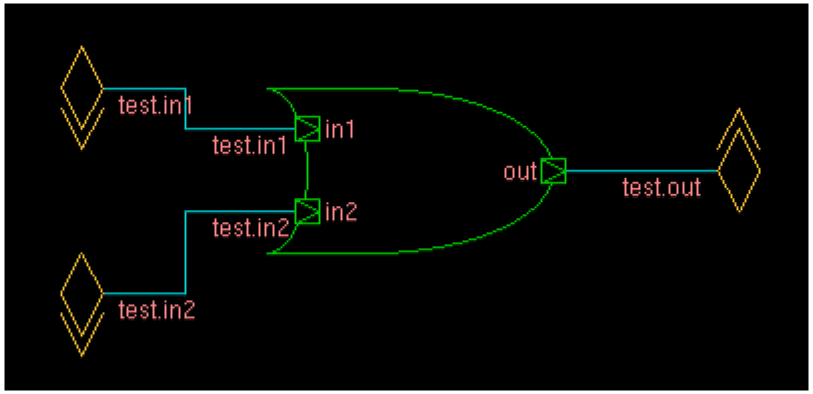


Verilog “assign” statement (2)

```
module test(
  input in1, in2,
  output out
);
          assign out = in1 | in2;
endmodule
```

wire declarations

a continuous logical or



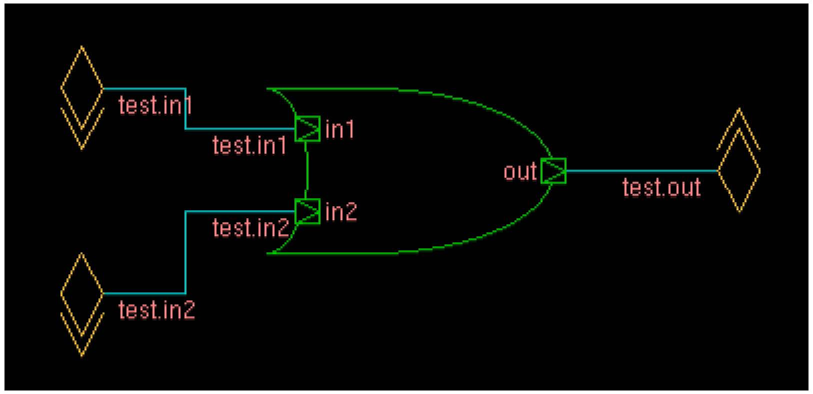
Verilog “assign” statement (2)

```
module test(
  input in1, in2,
  output out
);
          assign out = in1 | in2;
endmodule
```

wire declarations

a continuous logical or

gets assigned a new value whenever in1 or in2 change

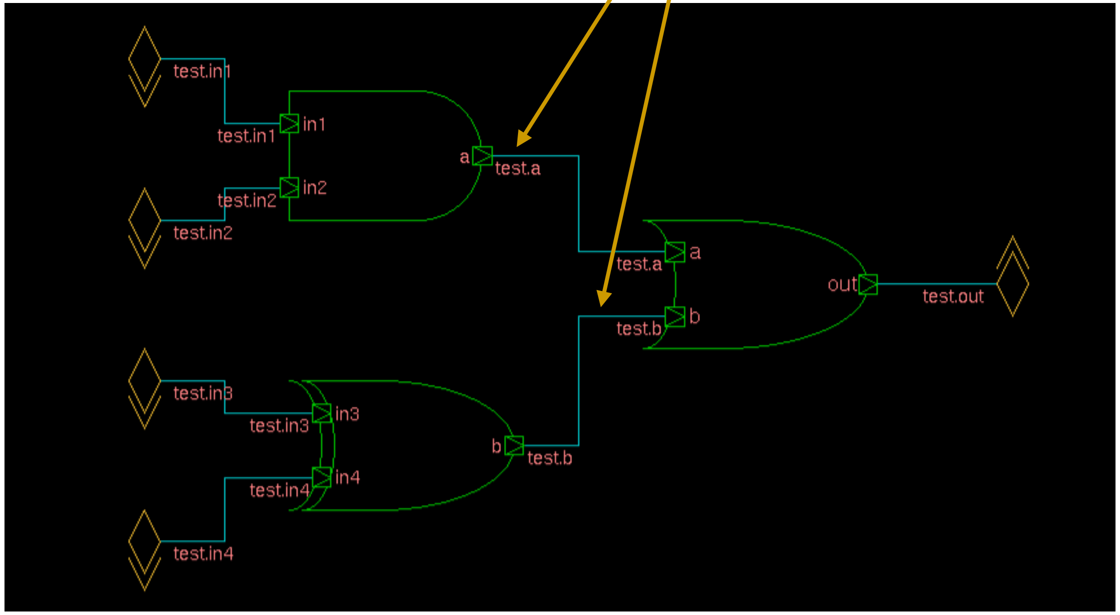


Verilog “assign” statement (3)

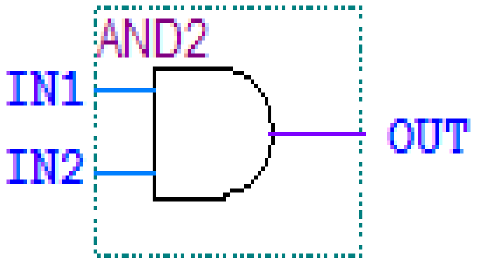
```
module test(
  input in1, in2, in3, in4,
  output out
);
          wire a, b;

          assign a = in1 & in2;
          assign b = in3 ^ in4;
          assign out = a | b;
endmodule
```

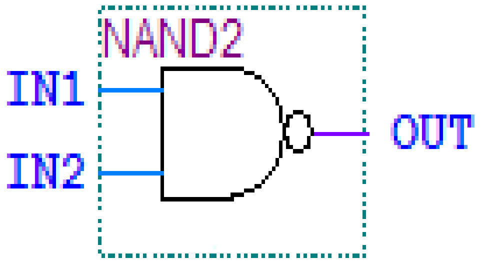
interconnect wires

Combinational circuit has to be continuously driven to produce and maintain the output



Gate primitives

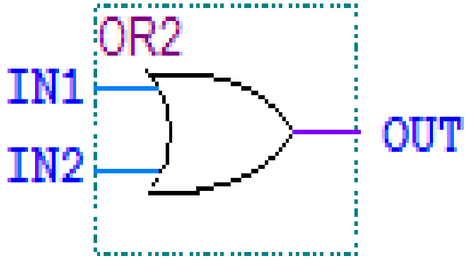


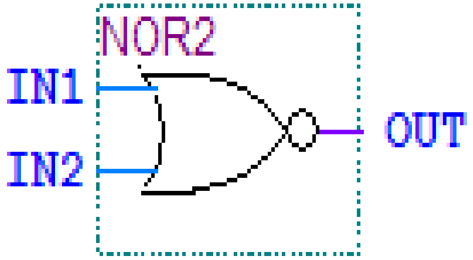


	AND2	IN1: 0	1	x	z
IN2	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

	NAND2	IN1: 0	1	x	z
IN2	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

Gate primitives: OR2, NOR2

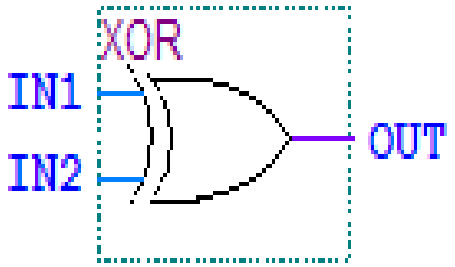


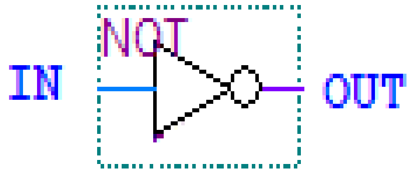


		IN1			
OR2		0	1	x	z
IN2	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

		IN1			
NOR2		0	1	x	z
IN2	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

Gate primitives: XOR2, NOT





		IN1			
	XOR	0	1	x	z
IN2	0	0	1	x	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

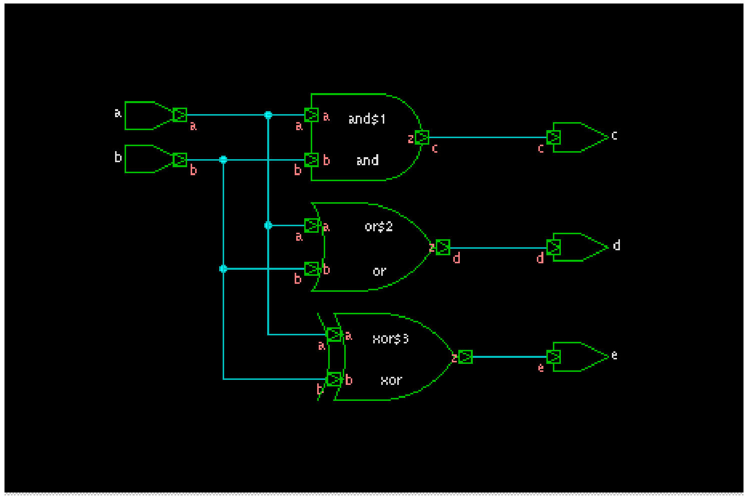
	NOT	OUT
IN	0	1
	1	0
	x	x
	z	x

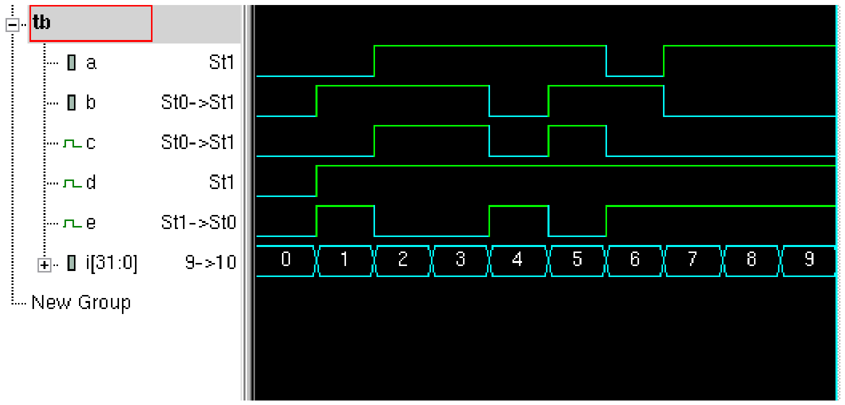
AND / NAND Using Assign

```
module AND2(output OUT, input  IN1, IN2);
        assign OUT = IN1 & IN2;
endmodule

module NAND2(output OUT, input IN1, IN2);
        assign OUT = ~(IN1 & IN2);
endmodule
```

AND/OR/XOR Gates

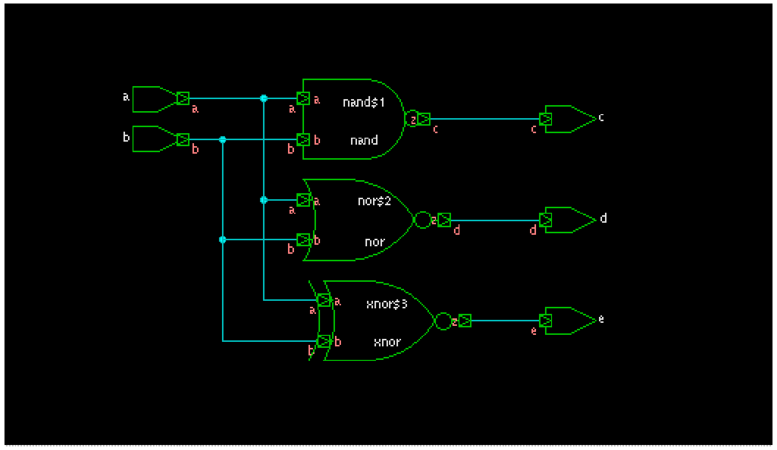
```
module gates(input a, b, output c, d, e);
        and(c, a, b);   // c is the output, a and b are the inputs
        or(d, a, b);    // d is the output, a and b are the inputs
        xor(e, a, b);   // e is the output, a and b are the inputs
endmodule
```

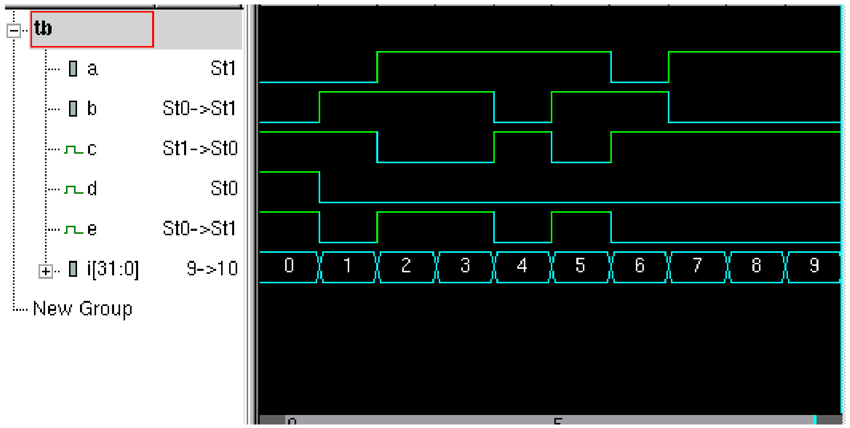




NAND/NOR/XNOR Gates

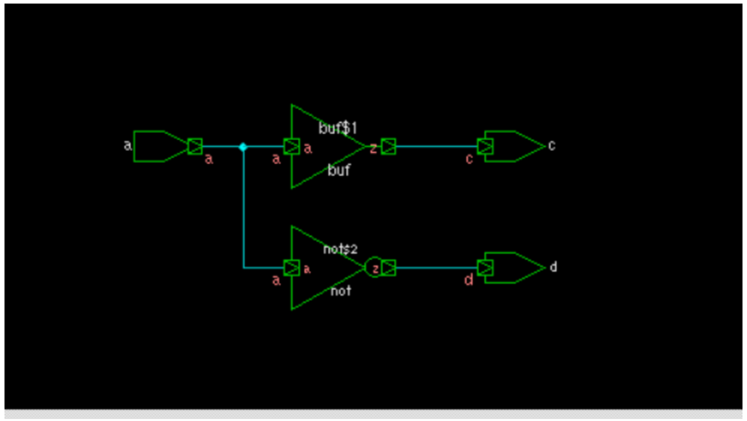
```
module gates(input a, b, output c, d, e);
        // Use nand, nor, xnor instead of and, or and xor
        // in this example
        nand(c, a, b);    // c is the output, a and b are the inputs
        nor(d, a, b);     // d is the output, a and b are the inputs
        xnor(e, a, b);    // e is the output, a and b are the inputs
endmodule
```



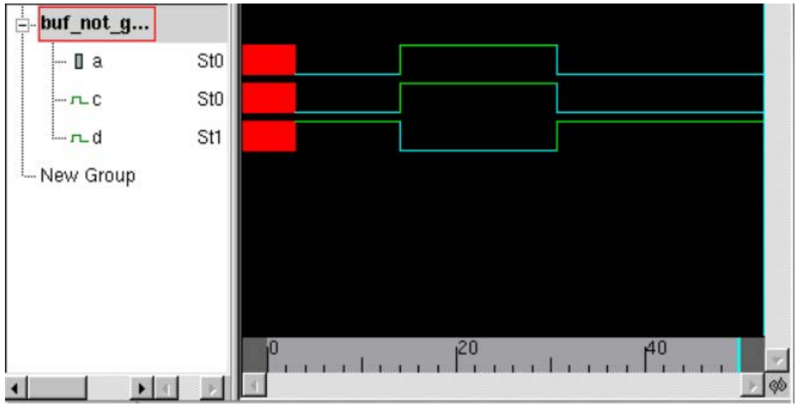


BUF/NOT Gates

```
module gates (input a, output c, d);
        buf (c, a);        // c is the output, a is the input
        not (d, a);        // d is the output, a is the input
endmodule
```



a
a
a
a
buf$1
buf
z
c
c
a
a
not$2
not
z
d
d



buf_not_g...
a St0
c St0
d St1
New Group
0
20
40