CESE4005: Hardware Fundamentals

2024-2025, lecture 4

Introduction to Verilog: Part 1

Computer Engineering Lab

Faculty of Electrical Engineering, Mathematics & Computer Science

2024-2025

Anteneh Gebregiorgis

TU Delft

Delft University of Technology

Objectives

- To become familiar with Hardware Description Language (HDL) to specify designs

Objectives

- To become familiar with Hardware Description Language (HDL) to specify designs

- Be able to understand Verilog HDL description

Objectives

- To become familiar with Hardware Description Language (HDL) to specify designs
- Be able to understand Verilog HDL description
- Be able to write a simple Verilog HDL description using a limited set of syntax and semantics

Objectives

- To become familiar with Hardware Description Language (HDL) to specify designs
- Be able to understand Verilog HDL description
- Be able to write a simple Verilog HDL description using a limited set of syntax and semantics
- Understanding the need for a “hardware view” when reading and writing an HDL

Overview

- HDL introduction
- Verilog basics and syntax
- Modules
- Assignments
- Initial and always blocks
- Structural modeling
- Dangers of Verilog
- Delays
- Testbenches

Introduction

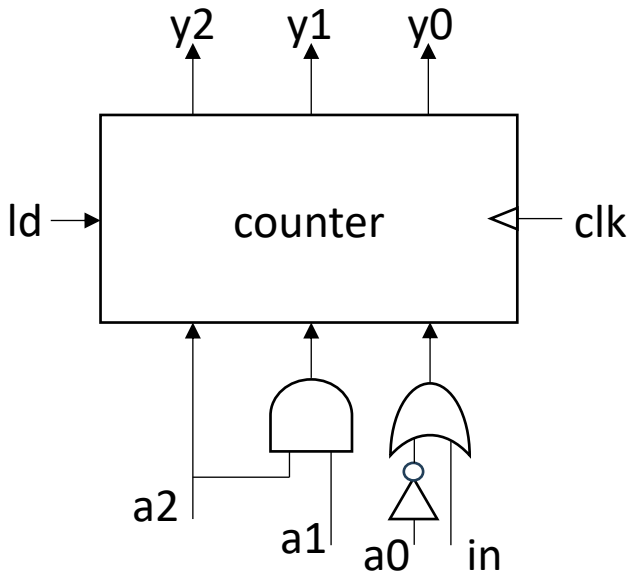
- Hardware description language (HDL) is a language to:
  - Specify logic function at different abstraction levels.

Introduction

- Hardware description language (HDL) is a language to:
  - Specify logic function at different abstraction levels.
  - Simulate the intended hardware behavior
  - Generate the final hardware (netlist with all components)

Introduction

- Hardware description language (HDL) is a language to:
  - Specify logic function at different abstraction levels.
  - Simulate the intended hardware behavior
  - Generate the final hardware (netlist with all components)

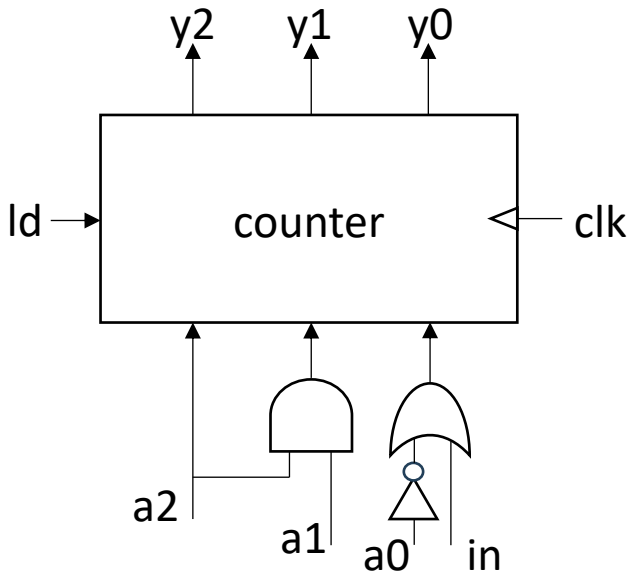



```
module counterdesign (
        input clk, ld, a2, a1, a0, in,
        output y2, y1, y0);

   wire n1, n0, ninv;

   counter inst1 (clk, ld, y2, y1, y0, a2, n1, n0);
   assign n1 = a2 & a1;
   assign n0 = ninv | in;
   assign ninv = ~a0;
endmodule;
```

Verilog Example

Introduction

- Hardware description language (HDL) is a language to:
  - Specify logic function at different abstraction levels.
  - Simulate the intended hardware behavior
  - Generate the final hardware (netlist with all components)
- Most commercial designs built using HDLs

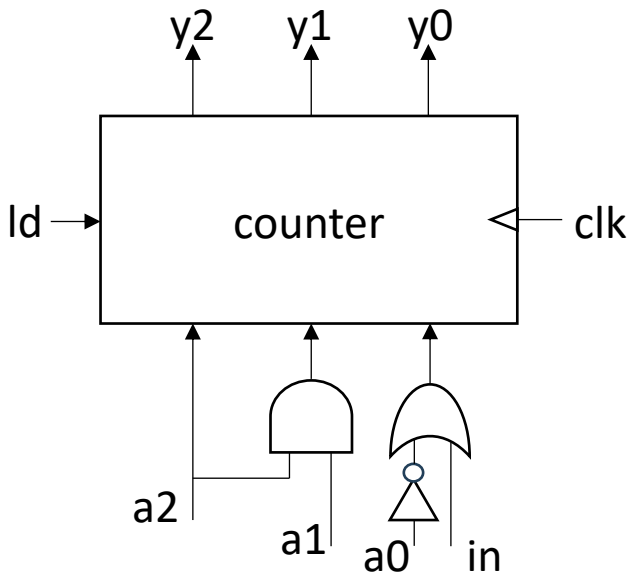



```
module counterdesign (
        input clk, ld, a2, a1, a0, in,
        output y2, y1, y0);

   wire n1, n0, ninv;

   counter inst1 (clk, ld, y2, y1, y0, a2, n1, n0);
   assign n1 = a2 & a1;
   assign n0 = ninv | in;
   assign ninv = ~a0;
endmodule;
```

Verilog Example

Introduction

- Hardware description language (HDL) is a language to:
  - Specify logic function at different abstraction levels.
  - Simulate the intended hardware behavior
  - Generate the final hardware (netlist with all components)
- Most commercial designs built using HDLs
- Leading HDLs:
  - **Verilog/SystemVerilog**
    - Developed in 1984 by Gateway Design Automation
  - **VHDL**
    - Developed in 1981 by the US Department of Defense

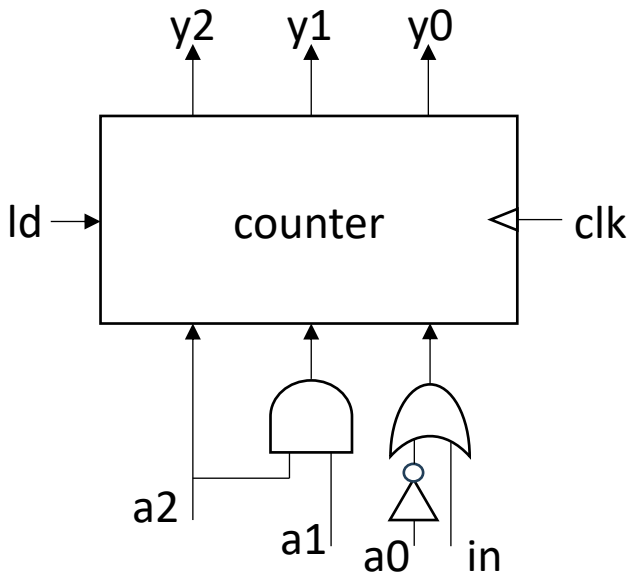



```
module counterdesign (
        input clk, ld, a2, a1, a0, in,
        output y2, y1, y0);

    wire n1, n0, ninv;

    counter inst1 (clk, ld, y2, y1, y0, a2, n1, n0);
    assign n1 = a2 & a1;
    assign n0 = ninv | in;
    assign ninv = ~a0;
endmodule;
```

Verilog Example

Introduction

- Hardware description language (HDL) is a language to:
  - Specify logic function at different abstraction levels.
  - Simulate the intended hardware behavior
  - Generate the final hardware (netlist with all components)
- Most commercial designs built using HDLs
- Leading HDLs:
  - **Verilog/SystemVerilog**
    - Developed in 1984 by Gateway Design Automation
  - **VHDL**
    - Developed in 1981 by the US Department of Defense
- We use Verilog, not VHDL,
  - VHDL is more verbose and cumbersome




```
module counterdesign (
        input clk, ld, a2, a1, a0, in,
        output y2, y1, y0);

    wire n1, n0, ninv;

    counter inst1 (clk, ld, y2, y1, y0, a2, n1, n0);
    assign n1 = a2 & a1;
    assign n0 = ninv | in;
    assign ninv = ~a0;
endmodule;
```

Verilog Example

HDL role in hardware design

The two major purpose of HDLs are logic simulation and synthesis

HDL role in hardware design

The two major purpose of HDLs are logic simulation and synthesis

- **Simulation of hardware behavior**
  - Inputs applied to circuit

HDL role in hardware design

The two major purpose of HDLs are logic simulation and synthesis

- **Simulation of hardware behavior**
  - Inputs applied to circuit
  - Simulator simulates the output signals
  - Outputs checked for correctness

HDL role in hardware design

The two major purpose of HDLs are logic simulation and synthesis

- **Simulation of hardware behavior**
  - Inputs applied to circuit
  - Simulator simulates the output signals
  - Outputs checked for correctness
  - Millions of euros saved by debugging in simulation instead of hardware

HDL role in hardware design

The two major purpose of HDLs are logic simulation and synthesis

- **Simulation of hardware behavior**
  - Inputs applied to circuit
  - Simulator simulates the output signals
  - Outputs checked for correctness
  - Millions of euros saved by debugging in simulation instead of hardware
- **Synthesis**
  - Logic synthesizer transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

HDL role in hardware design

The two major purpose of HDLs are logic simulation and synthesis

- **Simulation of hardware behavior**
  - Inputs applied to circuit
  - Simulator simulates the output signals
  - Outputs checked for correctness
  - Millions of euros saved by debugging in simulation instead of hardware
- **Synthesis**
  - Logic synthesizer transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)
  - Optimization is performed to reduce the amount of hardware.
  - The output netlist maybe a Verilog file, or a schematic to visualize the circuit.

HDLs vs. Software programming languages

- **Have syntactically similar constructs:**
  - Data types, variables, assignments, if statements, loops, …

HDLs vs. Software programming languages

- **Have syntactically similar constructs:**
  – Data types, variables, assignments, if statements, loops, …
  Similarity ends here

HDLs vs. Software programming languages

- **Have syntactically similar constructs:**
  - Data types, variables, assignments, if statements, loops, …

  Similarity ends here

- **Very different mentality and semantic model: everything runs in parallel, unless specified otherwise**
  - Statement models hardware
  - Hardware is inherently parallel

HDLs vs. Software programming languages

- **Have syntactically similar constructs:**
  - Data types, variables, assignments, if statements, loops, …
  - **Similarity ends here**

- **Very different mentality and semantic model: everything runs in parallel, unless specified otherwise**
  - Statement models hardware
  - Hardware is inherently parallel

- **Hardware descriptions are composed of modules (mostly)**
  - A hierarchy of modules connected to each other
  - Modules are active at the same time

HDLs vs. Software programming languages

- **Have syntactically similar constructs:**
  - Data types, variables, assignments, if statements, loops, …

  Similarity ends here

- **Very different mentality and semantic model: everything runs in parallel, unless specified otherwise**
  - Statement models hardware
  - Hardware is inherently parallel

- **Hardware descriptions are composed of modules (mostly)**
  - A hierarchy of modules connected to each other
  - Modules are active at the same time

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

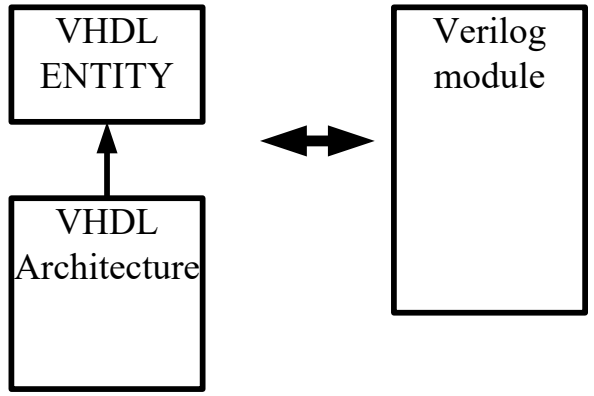
Verilog vs VHDL vs SystemVerilog

- **VHDL ➔ description language**
  - Strongly typed ➔ more compiler errors
  - verbose and deterministic language

Verilog vs VHDL vs SystemVerilog

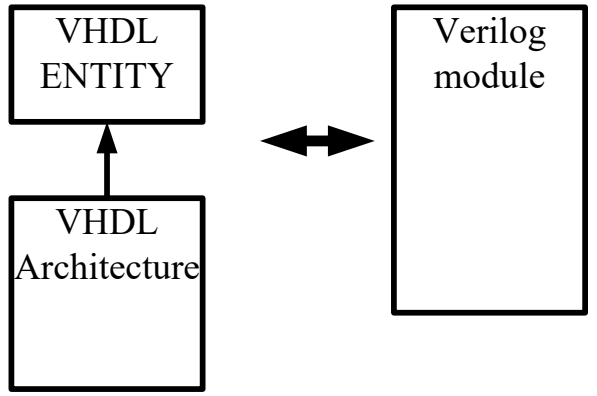
- **VHDL ➔ description language**
  - Strongly typed ➔ more compiler errors
  - verbose and deterministic language

- **Verilog ➔ modeling language**
  - weakly typed ➔ less compiler error
  - C-like, easy to learn
  - Less code than VHDL for the same task



Verilog vs VHDL vs SystemVerilog

- **VHDL ➔ description language**
  - Strongly typed ➔ more compiler errors
  - verbose and deterministic language



- **Verilog ➔ modeling language**
  - weakly typed ➔ less compiler error
  - C-like, easy to learn
  - Less code than VHDL for the same task

- **SystemVerilog ➔ based on Verilog with more features**
  - it is like C++ of Verilog
  - More datatypes & feature
  - Easy to learn/switch to from Verilog

Verilog basics and syntax

- **Case sensitive**
  - **Example:** **reset** and **Reset** are not the same signal.
  - Be consistent in your use of capitalization and underscores in signal and module names.

Verilog basics and syntax

- **Case sensitive**
  - **Example: reset** and **Reset** are not the same signal.
  - Be consistent in your use of capitalization and underscores in signal and module names.
- **No** identifier that **start with numbers**
  - Example: **2mux** is an **invalid** identifier

```
integer var_a;          // Identifier contains alphabets and underscore -> Valid
integer $var_a;         // Identifier starts with $ -> Invalid
integer v$ar_a;         // Identifier contains alphabets and $ -> Valid
integer 2var;           // Identifier starts with a digit -> Invalid
integer var23_g;        // Identifier contains alphanumeric characters and underscore -> Valid
integer 23;             // Identifier contains only numbers -> Invalid
```

Verilog basics and syntax

- **Case sensitive**
  - **Example: reset** and **Reset** are not the same signal.
  - Be consistent in your use of capitalization and underscores in signal and module names.
- **No** identifier that **start with numbers**
  - Example: **2mux** is an **invalid** identifier

```
integer var_a;          // Identifier contains alphabets and underscore -> Valid
integer $var_a;         // Identifier starts with $ -> Invalid
integer v$ar_a;         // Identifier contains alphabets and $ -> Valid
integer 2var;           // Identifier starts with a digit -> Invalid
integer var23_g;        // Identifier contains alphanumeric characters and underscore -> Valid
integer 23;             // Identifier contains only numbers -> Invalid
```

  - Keywords ➔ special identifier to define Verilog construct
    - E.g., always, and, assign, generate, endgenerate, event, for, forever, fork, function, etc…

Verilog basics and syntax

- **Case sensitive**
  - **Example: reset** and **Reset** are not the same signal.
  - Be consistent in your use of capitalization and underscores in signal and module names.
- **No** identifier that **start with numbers**
  - Example: **2mux** is an **invalid** identifier

```
integer var_a;          // Identifier contains alphabets and underscore -> Valid
integer $var_a;         // Identifier starts with $ -> Invalid
integer v$ar_a;         // Identifier contains alphabets and $ -> Valid
integer 2var;           // Identifier starts with a digit -> Invalid
integer var23_g;        // Identifier contains alphanumeric characters and underscore -> Valid
integer 23;             // Identifier contains only numbers -> Invalid
```

  - Keywords ➔ special identifier to define Verilog construct
    - E.g., always, and, assign, generate, endgenerate, event, for, forever, fork, function, etc…
- **Whitespace (spaces, tabs, newlines) are ignored**
  - Nevertheless, proper indenting and use of blank lines is helpful to make nontrivial designs readable.

Verilog basics and syntax

- **Case sensitive**
  - **Example: reset** and **Reset** are not the same signal.
  - Be consistent in your use of capitalization and underscores in signal and module names.
- **No** identifier that **start with numbers**
  - Example: **2mux** is an **invalid** identifier

```
integer var_a;          // Identifier contains alphabets and underscore -> Valid
integer $var_a;         // Identifier starts with $ -> Invalid
integer v$ar_a;         // Identifier contains alphabets and $ -> Valid
integer 2var;           // Identifier starts with a digit -> Invalid
integer var23_g;        // Identifier contains alphanumeric characters and underscore -> Valid
integer 23;             // Identifier contains only numbers -> Invalid
```

  - Keywords ➔ special identifier to define Verilog construct
    - E.g., always, and, assign, generate, endgenerate, event, for, forever, fork, function, etc…
- **Whitespace (spaces, tabs, newlines) are ignored**
  - Nevertheless, proper indenting and use of blank lines is helpful to make nontrivial designs readable.
- **Statements are ended with a ;**
- **List elements are separated with a ,**

Verilog basics and syntax

- **Case sensitive**
  - **Example: reset** and **Reset** are not the same signal.
  - Be consistent in your use of capitalization and underscores in signal and module names.
- **No** identifier that **start with numbers**
  - Example: **2mux** is an **invalid** identifier

```
integer var_a;          // Identifier contains alphabets and underscore -> Valid
integer $var_a;         // Identifier starts with $ -> Invalid
integer v$ar_a;         // Identifier contains alphabets and $ -> Valid
integer 2var;           // Identifier starts with a digit -> Invalid
integer var23_g;        // Identifier contains alphanumeric characters and underscore -> Valid
integer 23;             // Identifier contains only numbers -> Invalid
```

  - Keywords ➔ special identifier to define Verilog construct
    - E.g., always, and, assign, generate, endgenerate, event, for, forever, fork, function, etc…
- **Whitespace (spaces, tabs, newlines) are ignored**
  - Nevertheless, proper indenting and use of blank lines is helpful to make nontrivial designs readable.
- **Statements are ended with a ;**
- **List elements are separated with a ,**
- **Comments: // single line comment**
  - **/* multiline**

    comment */

Verilog operators

There are three types of operators

Verilog operators

There are three types of operators

- **Unary operators**

  - Operators that take only one operand

Verilog operators

There are three types of operators

- **Unary operators**
  - Operators that take only one operand
  - Shall appear to the left of the operand
  - e.g., `x = ~y;` // ~ is a unary *negation* operator

Verilog operators

There are three types of operators

- **Unary operators**
  - Operators that take only one operand
  - Shall appear to the left of the operand
  - e.g., `x = ~y;` // ~ is a unary *negation* operator

- **Binary operators**
  - Operators that take two operands
  - Shall appear between the operands
  - e.g., `x = y & z;`

Verilog operators

There are three types of operators

- **Unary operators**
  - Operators that take only one operand
  - Shall appear to the left of the operand
  - e.g., `x = ~y;` // ~ is a unary *negation* operator

- **Binary operators**
  - Operators that take two operands
  - Shall appear between the operands
  - e.g., `x = y & z;`

- **Ternary or conditional operators**
  - have two operators that separate three operands
  - e.g., `x = (y>5) ? w : z;`

Verilog operators

There are three types of operators

- **Unary operators**
  - Operators that take only one operand
  - Shall appear to the left of the operand
  - e.g., x = ~y; // ~ is a unary *negation* operator

- **Binary operators**
  - Operators that take two operands
  - Shall appear between the operands
  - e.g., x = y & z;

- **Ternary or conditional operators**
  - have two operators that separate three operands
  - e.g., x = (y>5) ? w : z;

Another way of grouping operators

Operator types	examples
Bitwise	~, &, |, ^, ^~
Arithmetic	+, -, *, /, %, >>, <<
Relational/equality	>, <, >=, <=/ ==, !=
logical	&&, ||, !

Operator precedence

Operators	Meaning
~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Operator precedence

Highest

Operators	Meaning
~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Operator precedence

Highest

Operators	Meaning
~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Lowest

Operator precedence

Highest

Operators	Meaning
~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Lowest

E.g., `~a & b + c` is evaluated as `(~a) & (b + c)`

Verilog lexical conventions

Concatenation & Replication Operators

{id_1, id_2, …} ➔ Concatenates id_2 at the end of id_1

{n{id}} ➔ replicates id *n* times

- Examples: assign conc_reg = {REG_IN[6:0],Serial_in},

{8 {1'b0}} ➔ *00000000*

Verilog lexical conventions

Concatenation & Replication Operators

{id_1, id_2, …} ➔ Concatenates id_2 at the end of id_1

{n{id}} ➔ replicates id *n* times

- Examples: assign conc_reg = {REG_IN[6:0],Serial_in},

{8 {1'b0}} ➔ *00000000*

Binary, number and string specification

- Binary values ➔ 0, 1, X and Z (X is *unknown value and* Z is *high impedance state)*

Verilog lexical conventions

Concatenation & Replication Operators

{id_1, id_2, …} ➔ Concatenates id_2 at the end of id_1

{n{id}} ➔ replicates id *n* times

- Examples: assign conc_reg = {REG_IN[6:0],Serial_in},

{8 {1'b0}} ➔ *00000000*

Binary, number and string specification

- Binary values ➔ 0, 1, X and Z (X is *unknown value and* Z is *high impedance state)*
- *Number specification ➔ 3 ways to specify*

Verilog lexical conventions

Concatenation & Replication Operators

{id_1, id_2, …} ➔ Concatenates id_2 at the end of id_1

{n{id}} ➔ replicates id *n* times

- Examples: assign conc_reg = {REG_IN[6:0],Serial_in},

{8 {1'b0}} ➔ *00000000*

Binary, number and string specification

- Binary values ➔ 0, 1, X and Z (X is *unknown value and* Z is *high impedance state)*
- *Number specification ➔ 3 ways to specify*
  - *Simple decimal (unbased) specification ➔ sequence of digits 0-9 e.g., 729,-365 etc*

Verilog lexical conventions

Concatenation & Replication Operators

{id_1, id_2, …} ➔ Concatenates id_2 at the end of id_1

{n{id}} ➔ replicates id *n* times

- Examples: assign conc_reg = {REG_IN[6:0],Serial_in},

{8 {1'b0}} ➔ *00000000*

Binary, number and string specification

- Binary values ➔ 0, 1, X and Z (X is *unknown value and* Z is *high impedance state)*
- *Number specification ➔ 3 ways to specify*
  - *Simple decimal (unbased) specification ➔ sequence of digits 0-9 e.g., 729,-365 etc*
  - *Based sized specification ➔ <sign> <size> ' <base format> <number>*
    - *4'b1111 ➔ 4-bit binary number*
    - *-12'habc & -12s'habc ➔ 12-bit negative hexadecimal number for unsigned & signed operation*

Verilog lexical conventions

Concatenation & Replication Operators

{id_1, id_2, …} ➔ Concatenates id_2 at the end of id_1

{n{id}} ➔ replicates id *n* times

- Examples: assign conc_reg = {REG_IN[6:0],Serial_in},

{8 {1'b0}} ➔ *00000000*

Binary, number and string specification

- Binary values ➔ 0, 1, X and Z (X is *unknown value and* Z is *high impedance state)*
- *Number specification ➔ 3 ways to specify*
  - *Simple decimal (unbased) specification ➔ sequence of digits 0-9 e.g., 729,-365 etc*
  - *Based sized specification ➔ <sign> <size> ' <base format> <number>*
    - *4'b1111 ➔ 4-bit binary number*
    - *-12'habc & -12s'habc ➔ 12-bit negative hexadecimal number for unsigned & signed operation*
  - *Based unisized specification ➔ <sign> ' <base format> <number>*
    - *'o7460 ➔ is an octal number*

Verilog lexical conventions

Concatenation & Replication Operators

{id_1, id_2, …} ➔ Concatenates id_2 at the end of id_1

{n{id}} ➔ replicates id *n* times

- Examples: assign conc_reg = {REG_IN[6:0],Serial_in},

{8 {1'b0}} ➔ *00000000*

Binary, number and string specification

- Binary values ➔ 0, 1, X and Z (X is *unknown value and* Z is *high impedance state)*
- *Number specification ➔ 3 ways to specify*
  - *Simple decimal (unbased) specification ➔ sequence of digits 0-9 e.g., 729,-365 etc*
  - *Based sized specification ➔ <sign> <size> ' <base format> <number>*
    - *4'b1111 ➔ 4-bit binary number*
    - *-12'habc & -12s'habc ➔ 12-bit negative hexadecimal number for unsigned & signed operation*
  - *Based unisized specification ➔ <sign> ' <base format> <number>*
    - *'o7460 ➔ is an octal number*
- *String values ➔ any value within " "*

Numbers

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b10101011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010

Numbers

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b10101011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011

Numbers

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b10101011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	unsized	decimal (default)	42	00…0101010

Data types: Nets

- Used to connect between hardware entities e.g., gates
- They don’t store any values
- Their values is driven by the output of entities they are connected to

Data types: Nets

- Used to connect between hardware entities e.g., gates
- They don’t store any values
- Their values is driven by the output of entities they are connected to
- A net data type must be used when a signal is:
  - Driven by the output of some devices
  - It is declared as an input or in-out port
  - On a left hand of a continuous assignment

Data types: Nets

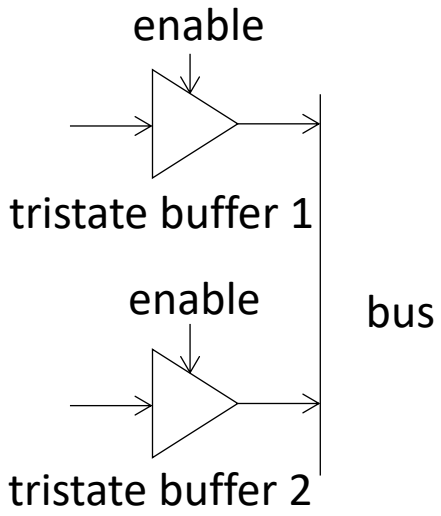
- Used to connect between hardware entities e.g., gates
- They don’t store any values
- Their values is driven by the output of entities they are connected to
- A net data type must be used when a signal is:
  - Driven by the output of some devices
  - It is declared as an input or in-out port
  - On a left hand of a continuous assignment
- Some net data types are ***wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1,*** and ***trireg***

Data types: Nets

- Used to connect between hardware entities e.g., gates
- They don’t store any values
- Their values is driven by the output of entities they are connected to
- A net data type must be used when a signal is:
  - Driven by the output of some devices
  - It is declared as an input or in-out port
  - On a left hand of a continuous assignment
- Some net data types are ***wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1,*** and ***trireg***
- Default value of net is ‘Z’ (except the “trireg” net, which defaults to x)

Z's and X's

- HDLs use z to indicate a floating value (not connected to a source).
- Similarly, HDLs use x to indicate an invalid logic level (or unknown/uninitialized value).
- z is particularly useful to describe a tristate buffer, whose output floats when the enable is 0.
- If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is x, indicating contention.
- If all of the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by z.



Resulting value of bus as a function of the buffer outputs

buffer 1 →

↓ buffer 2

	z	0	1
z	z	0	1
0	0	0	X
1	1	X	1

Data types: Registers

- Registers represent data storage element
- They retain the value until another value is placed to them
- Reg is a Verilog variable type and does not necessarily imply a physical register

Data types: Registers

- Registers represent data storage element
- They retain the value until another value is placed to them
- Reg is a Verilog variable type and does not necessarily imply a physical register
- Some register data types are ***reg***, **integer**, **time**, and ***real***
  - **Reg** is used to describe logic ➔ its default value is ‘x’
  - An **integer** is general-purpose variable to store signed numbers e.g., parameters, constants, loop-indices
  - **Real** in system modules.
  - **Time** and **realtime** for storing simulation times in test benches.

Data types: Registers

- Registers represent data storage element
- They retain the value until another value is placed to them
- Reg is a Verilog variable type and does not necessarily imply a physical register
- Some register data types are ***reg***, **integer**, **time**, and ***real***
  - **Reg** is used to describe logic ➔ its default value is ‘x’
  - An **integer** is general-purpose variable to store signed numbers e.g., parameters, constants, loop-indices
  - **Real** in system modules.
  - **Time** and **realtime** for storing simulation times in test benches.
- Verilog register do not need clock as hardware registers do

Data types: Vectors and Strings

- **Vectors**
  - Nets or reg data types can be declared as vectors (multiple bit widths)
  - Vectors can be declared at [high#:low#] or [low#:high#], but the left number in the squared brackets is always the most significant bit of the vector

Data types: Vectors and Strings

- **Vectors**
  - Nets or reg data types can be declared as vectors (multiple bit widths)
  - Vectors can be declared at [high#:low#] or [low#:high#], but the left number in the squared brackets is always the most significant bit of the vector

- **Strings**
  - Each character in a string represents an *ASCII* value and requires 1 byte
  - Strings are stored in reg

Data types: Vectors and Strings

- **Vectors**
  - Nets or reg data types can be declared as vectors (multiple bit widths)
  - Vectors can be declared at [high#:low#] or [low#:high#], but the left number in the squared brackets is always the most significant bit of the vector

- **Strings**
  - Each character in a string represents an *ASCII* value and requires 1 byte
  - Strings are stored in reg
  - The width of the reg variable has to be large enough to hold the string
    - If the variable's size is smaller than the string, then Verilog truncates the leftmost bits of the string
    - If the variable's size is larger than the string, then Verilog adds zeros to the left of the string

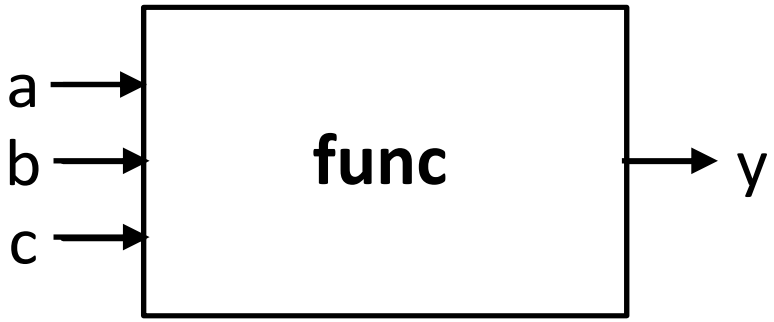
Verilog modules

- A module is the basic building block in Verilog

Verilog modules

- A module is the basic building block in Verilog
- Interfaces with outside using ***ports***

Example

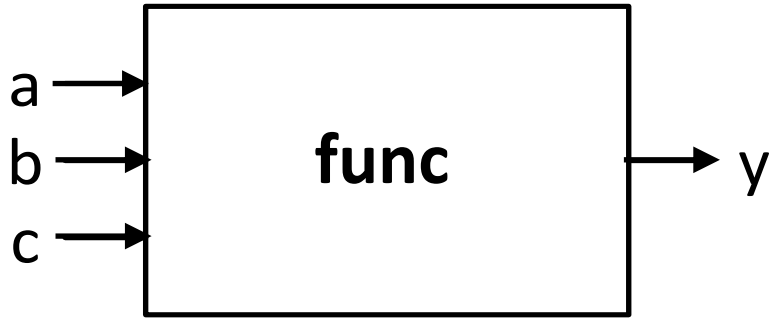



```
module func(input a, b, c,
            output y);

  // description of contents

endmodule
```

Verilog modules

- A module is the basic building block in Verilog
- Interfaces with outside using ***ports***
- Module can be an element or collection of lower-level design blocks
- Module cannot be nested

Example

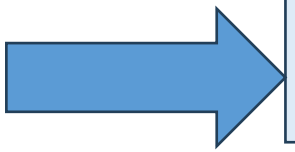



```
module func(input a, b, c,
            output y);

  // description of contents

endmodule
```

Verilog modules

Components of Verilog module

Module definition /specification

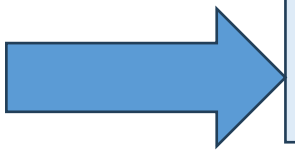


Module name, port list, pot declarations (if ports present)
Prameters (optional)

Verilog modules

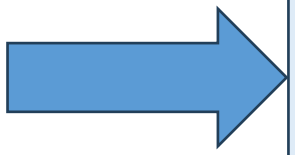
Components of Verilog module

Module definition /specification



Module name, port list, pot declarations (if ports present)
Prameters (optional)

Module implementation

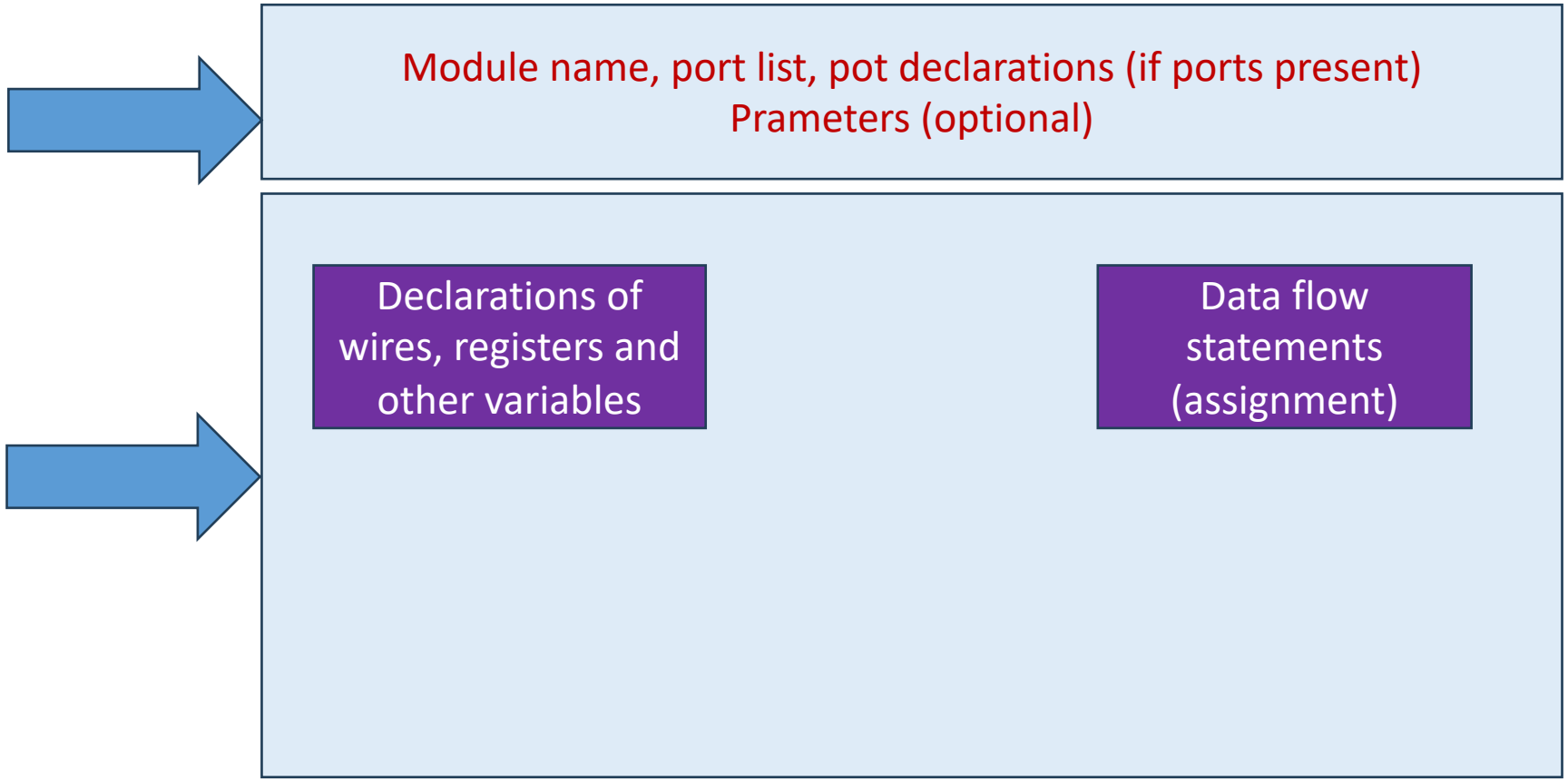


Verilog modules

Components of Verilog module

Module definition /specification

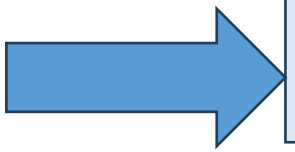
Module implementation



Verilog modules

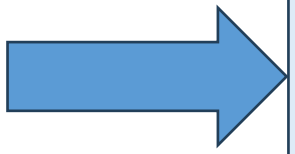
Components of Verilog module

Module definition /specification



Module name, port list, pot declarations (if ports present)
Prameters (optional)

Module implementation



Declarations of wires, registers and other variables

Instantiation of lower level modules

Data flow statements (assignment)

Always and initial blocks behavioral statements

Verilog modules

Components of Verilog module

Module definition /specification

Module name, port list, pot declarations (if ports present)
Prameters (optional)

Module implementation

Declarations of wires, registers and other variables

Data flow statements (assignment)

Instantiation of lower level modules

Always and initial blocks behavioral statements

Tasks and functions

Verilog modules

Components of Verilog module

Module definition /specification

Module name, port list, pot declarations (if ports present)
Prameters (optional)

Module implementation

Declarations of wires, registers and other variables

Data flow statements (assignment)

Instantiation of lower level modules

Always and initial blocks behavioral statements

Tasks and functions

End module

Endmodule statement

Signals and assignments

- Signals (ports and internal nodes) can be scalar (single-bit) or vector (multi-bit busses):

Signals and assignments

- Signals (ports and internal nodes) can be scalar (single-bit) or vector (multi-bit busses):

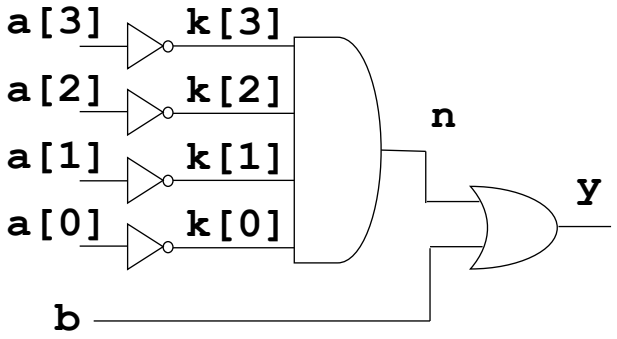
```
module func1(input [3:0] a,    // creates a[3], a[2], a[1] and a[0]
             input b,
             output y);

   wire [3:0] k;  // creates k[3], k[2], k[1] and k[0]
   wire n;

   assign k = ~a;  // implies k[3] = ~a[3], k[2] = ~a[2] etc.
   assign n = k[3] & k[2] & k[1] & k[0];
   assign y = b | n;

endmodule
```

Signals and assignments

- Signals (ports and internal nodes) can be scalar (single-bit) or vector (multi-bit busses):

```
module func1(input [3:0] a,    // creates a[3], a[2], a[1] and a[0]
             input b,
             output y);

   wire [3:0] k;  // creates k[3], k[2], k[1] and k[0]
   wire n;

   assign k = ~a;  // implies k[3] = ~a[3], k[2] = ~a[2] etc.
   assign n = k[3] & k[2] & k[1] & k[0];
   assign y = b | n;

endmodule
```

Equivalent circuit



Verilog modules

Two types of content descriptions within Modules:

Verilog modules

Two types of content descriptions within Modules:

- Behavioral: describe what the module does using programming language like constructs

Verilog modules

Two types of content descriptions within Modules:

- Behavioral: describe what the module does using programming language like constructs

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign n1 = a & b & c;
   assign y = ~n1;
endmodule
```

n1 is an intermediate signal

Verilog modules

Two types of content descriptions within Modules:

- Behavioral: describe what the module does using programming language like constructs

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign n1 = a & b & c;
   assign y = ~n1;
endmodule
```

n1 is an intermediate signal

- Structural: describe how it is built from other modules

Verilog modules

Two types of content descriptions within Modules:

- Behavioral: describe what the module does using programming language like constructs

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign n1 = a & b & c;
   assign y = ~n1;
endmodule
```

n1 is an intermediate signal

- Structural: describe how it is built from other modules

```
module nand3(input a, b, c
                output y);
  wire n1, n2;

  and3 i1 (a, b, c, n1);
  inv  i2 (n1, n2);
  assign y = n2;
endmodule
```

and3 and **inv** are other modules that are described elsewhere

Verilog modules

In general, the order of statements is not important (they just describe contents):

Verilog modules

In general, the order of statements is not important (they just describe contents):

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign n1 = a & b & c;
   assign y = ~n1;
endmodule
```

 is equivalent to

Verilog modules

In general, the order of statements is not important (they just describe contents):

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign n1 = a & b & c;
   assign y = ~n1;
endmodule
```

is equivalent to

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign y = ~n1;
   assign n1 = a & b & c;
endmodule
```

Verilog modules

In general, the order of statements is not important (they just describe contents):

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign n1 = a & b & c;
   assign y = ~n1;
endmodule
```

is equivalent to

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign y = ~n1;
   assign n1 = a & b & c;
endmodule
```

```
module nand3(input a, b, c
                output y);
   wire n1, n2;

   and3 i1 (a, b, c, n1);
   inv  i2 (n1, n2);
   assign y = n2;
endmodule
```

is equivalent to

Verilog modules

In general, the order of statements is not important (they just describe contents):

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign n1 = a & b & c;
   assign y = ~n1;
endmodule
```

is equivalent to

```
module nand3(input a, b, c,
             output y);
   wire n1;

   assign y = ~n1;
   assign n1 = a & b & c;
endmodule
```

```
module nand3(input a, b, c
             output y);
   wire n1, n2;

   and3 i1 (a, b, c, n1);
   inv  i2 (n1, n2);
   assign y = n2;
endmodule
```

is equivalent to

```
module nand3(input a, b, c
             output y);
   wire n1,  n2;

   inv  i2 (n1, n2);
   and3 i1 (a, b, c, n1);
   assign y = n2;
endmodule
```

Bitwise operator module example

- *Bitwise* operators act on single-bit signals or on multi-bit busses:

Bitwise operator module example

- *Bitwise* operators act on single-bit signals or on multi-bit busses:

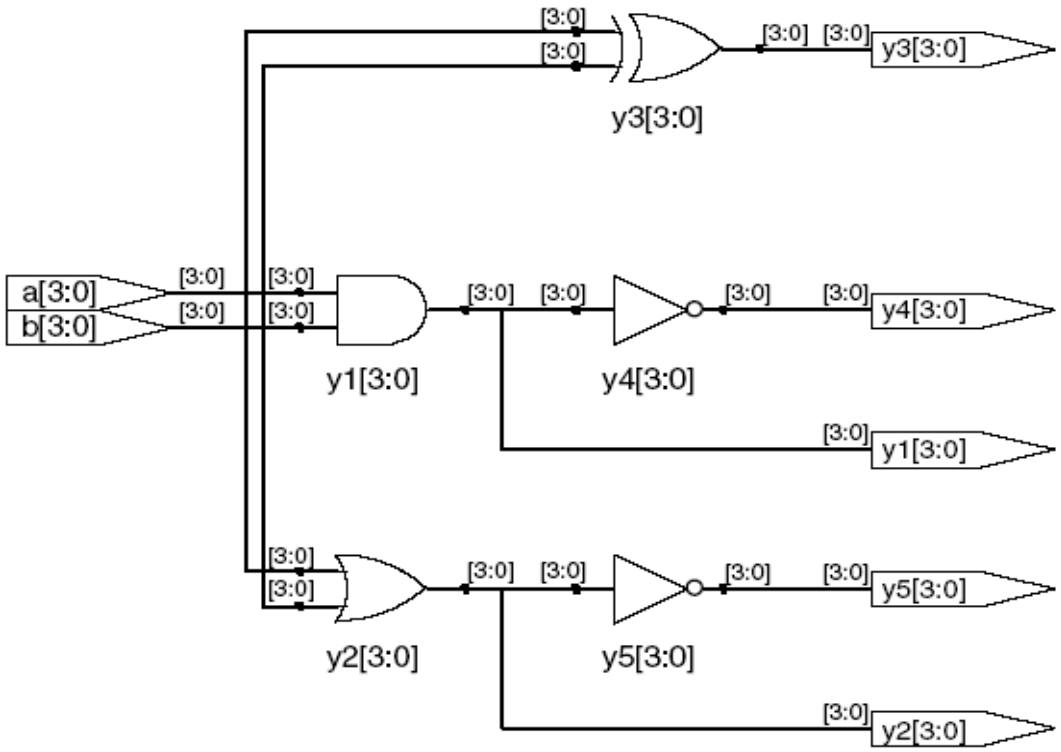
```
module gates(input [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

Bitwise operator module example

- *Bitwise* operators act on single-bit signals or on multi-bit busses:

```
module gates(input [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

Synthesis:



Reduction operator module example

- *Reduction* operators imply a multiple-input gate acting on a single bus.

Reduction operator module example

- *Reduction* operators imply a multiple-input gate acting on a single bus.

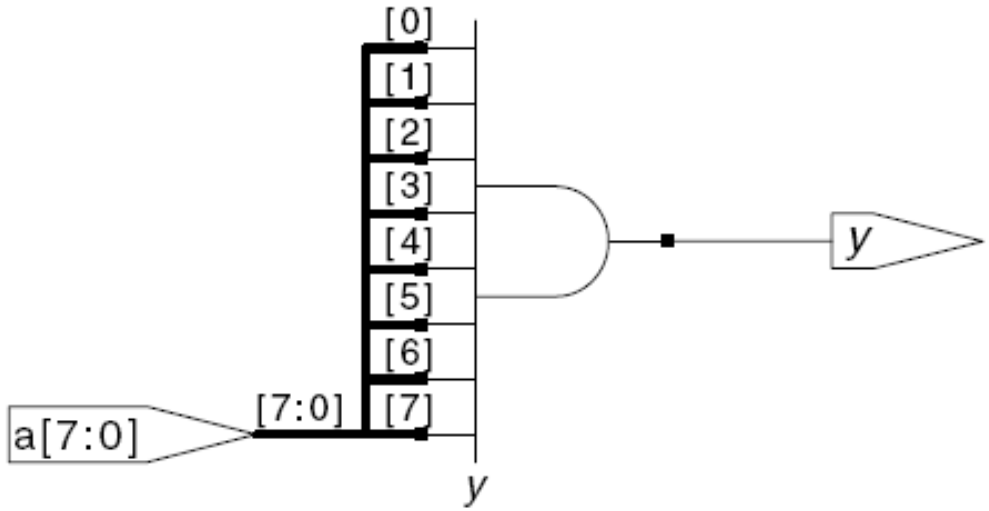
```
module and8(input [7:0] a,
            output y);
   assign y = &a;   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```

Reduction operator module example

- *Reduction* operators imply a multiple-input gate acting on a single bus.

```
module and8(input [7:0] a,
            output y);
   assign y = &a;   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```

Synthesis:



Conditional assignment

- *Conditional assignments* select the output from alternatives based on an input called the *condition.*

Conditional assignment

- *Conditional assignments* select the output from alternatives based on an input called the *condition.*

```
module mux2(input [3:0] d0, d1,
            input s,
            output [3:0] y);

   assign y = s ? d1 : d0;

endmodule
```

If s is 1,then y = d1 else y = d0.

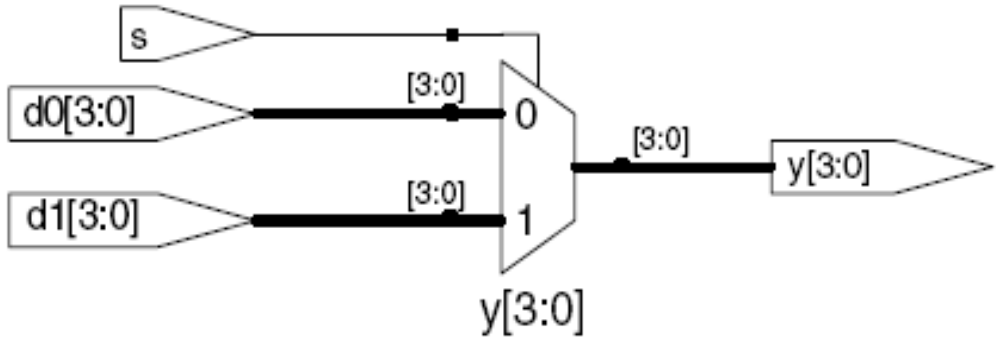
`? :` is also called a *ternary operator* because it operates on 3 inputs: `s`, `d1`, and `d0`.

Conditional assignment

- *Conditional assignments* select the output from alternatives based on an input called the *condition.*

```
module mux2(input [3:0] d0, d1,
            input s,
            output [3:0] y);

   assign y = s ? d1 : d0;

endmodule
```

Synthesis:



`If s is 1,then y = d1 else y = d0.`

`? :` is also called a *ternary operator* because it operates on 3 inputs: `s`, `d1`, and `d0`.

Blocking and non blocking assignment statements

Non blocking assignment

Blocking assignment

Blocking and non blocking assignment statements

Non blocking assignment

- It is represented with the sign “<=“
- Its execution is concurrent with the previous or next assignment statements

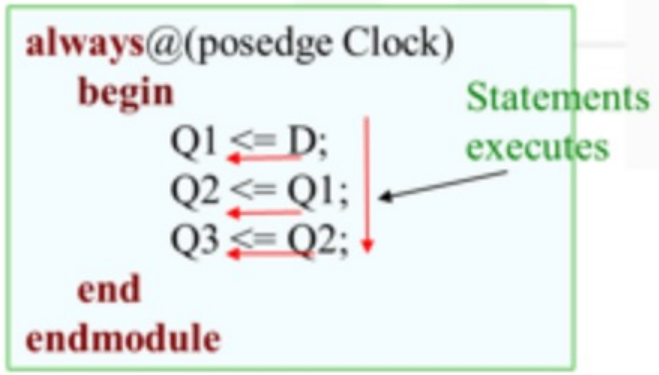
Blocking assignment

- It is represented with the sign “=“
- Its execution is *sequentially* i.e., one statement is executed at a time

Blocking and non blocking assignment statements

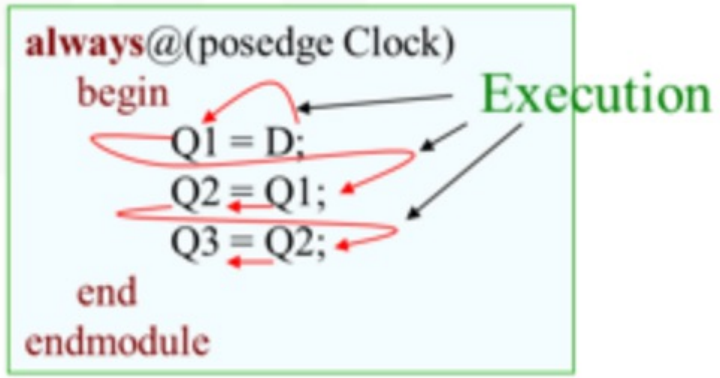
Non blocking assignment

- It is represented with the sign “<=“
- Its execution is concurrent with the previous or next assignment statements



Blocking assignment

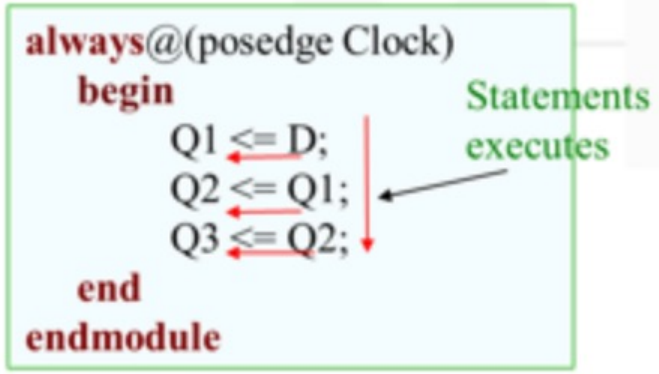
- It is represented with the sign “=“
- Its execution is *sequentially* i.e., one statement is executed at a time



Blocking and non blocking assignment statements

Non blocking assignment

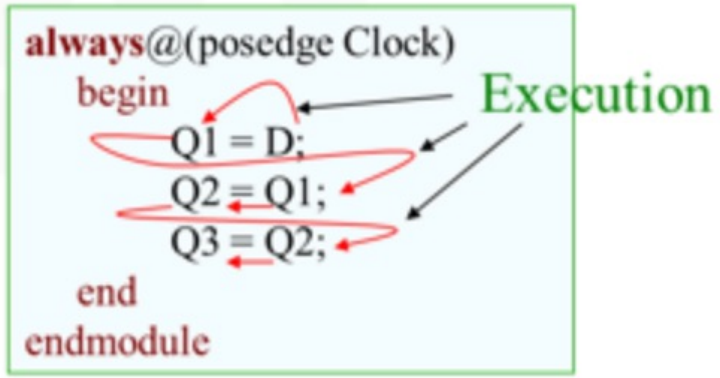
- It is represented with the sign “<=“
- Its execution is concurrent with the previous or next assignment statements



- It is illegal to use non blocking assignments in a continuous assignment statement or in a net declaration.

Blocking assignment

- It is represented with the sign “=“
- Its execution is *sequentially* i.e., one statement is executed at a time



- It is illegal to use blocking assignments in a continuous assignment statement or in a net declaration.

Internal variables

- Often it’s convenient to break a complex function into intermediate steps and introduce internal nodes.

Internal variables

- Often it's convenient to break a complex function into intermediate steps and introduce internal nodes.

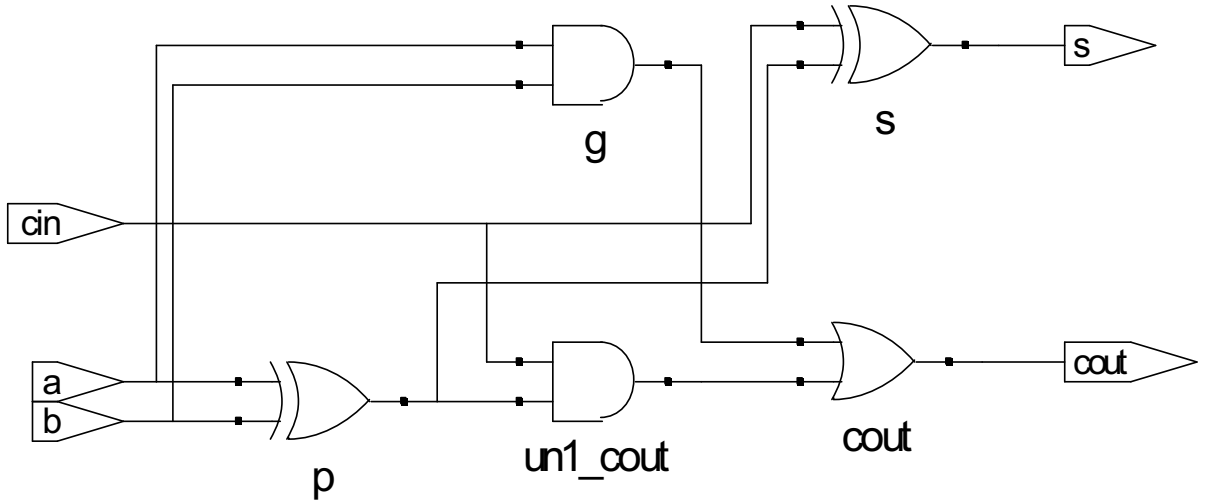
```
module fulladder(input a, b, cin,
                 output s, cout);
  wire p, g;   // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Internal variables

- Often it's convenient to break a complex function into intermediate steps and introduce internal nodes.

```
module fulladder(input a, b, cin,
                 output s, cout);
  wire p, g;    // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Synthesis:



During synthesis, additional internal nodes (e.g. un1_cout) may be created

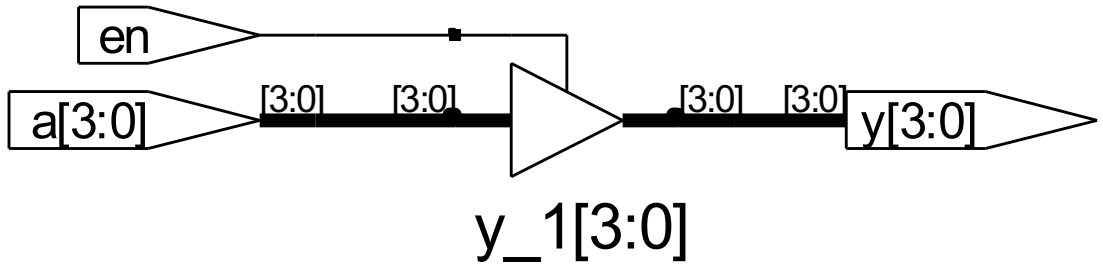
Z's and X's

```
module tristate(input [3:0] a,
                input en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

Z's and X's

```
module tristate(input [3:0] a,
                input en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

Synthesis:



The initial and always blocks

- The always block executes freely/whenever one of the signals in the list has a transition according to the way it is written

The initial and always blocks

- The always block executes freely/whenever one of the signals in the list has a transition according to the way it is written
- Could also write it with ANDs such that all the signals used must have a transition

The initial and always blocks

- The always block executes freely/whenever one of the signals in the list has a transition according to the way it is written
- Could also write it with ANDs such that all the signals used must have a transition

- The initial block executes only once

The initial and always blocks

- The always block executes freely/whenever one of the signals in the list has a transition according to the way it is written
- Could also write it with ANDs such that all the signals used must have a transition

- The initial block executes only once

- Both blocks starts execution at the beginning of the simulation (time zero)

Structural Verilog

- The previous sections discussed behavioral modeling: describing a module in terms of the relationships between inputs and outputs.

Structural Verilog

- The previous sections discussed behavioral modeling: describing a module in terms of the relationships between inputs and outputs.

- This section examines structural modeling: describing a module in terms of how it is composed of simpler modules.

Structural Verilog

- The previous sections discussed behavioral modeling: describing a module in terms of the relationships between inputs and outputs.

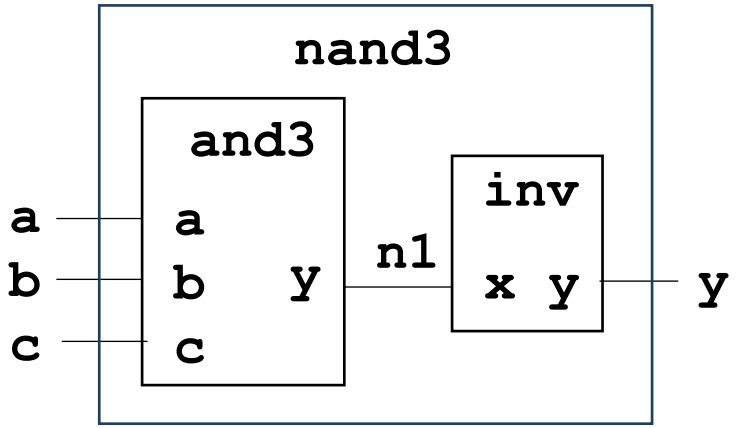
- This section examines structural modeling: describing a module in terms of how it is composed of simpler modules.

- At the lowest level (leaf modules) you always need to use behavioral modelling.

Structural Verilog

Example

- The previous sections discussed behavioral modeling: describing a module in terms of the relationships between inputs and outputs.
- This section examines structural modeling: describing a module in terms of how it is composed of simpler modules.
- At the lowest level (leaf modules) you always need to use behavioral modelling.

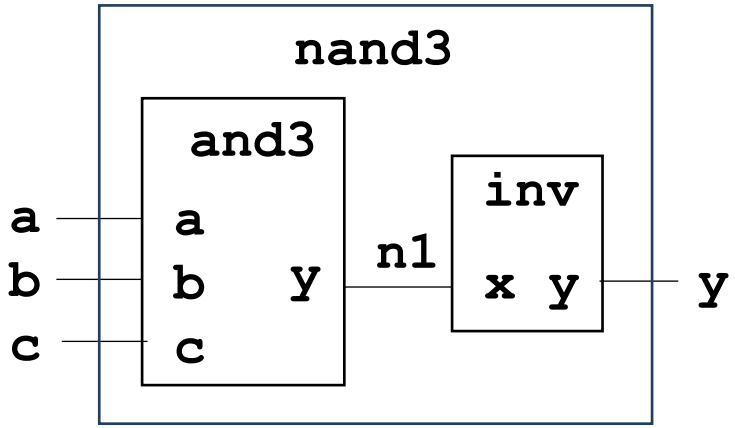


Structural Verilog

- The previous sections discussed behavioral modeling: describing a module in terms of the relationships between inputs and outputs.
- This section examines structural modeling: describing a module in terms of how it is composed of simpler modules.
- At the lowest level (leaf modules) you always need to use behavioral modelling.

Example

```
module and3(input a, b, c,
            output y);
  assign y = a & b & c;
endmodule
```

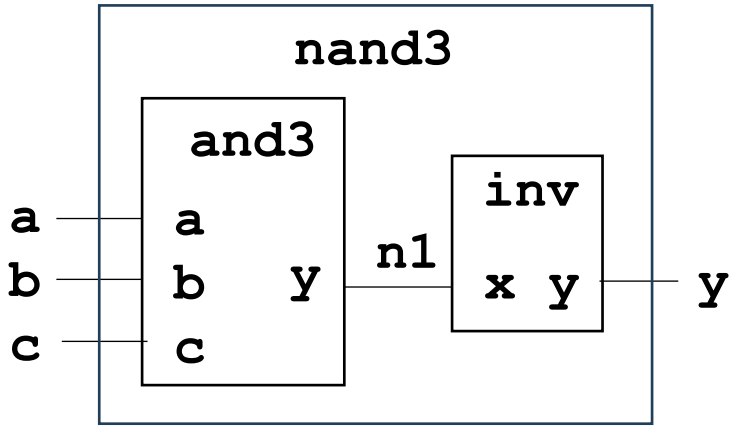


Structural Verilog

- The previous sections discussed behavioral modeling: describing a module in terms of the relationships between inputs and outputs.
- This section examines structural modeling: describing a module in terms of how it is composed of simpler modules.
- At the lowest level (leaf modules) you always need to use behavioral modelling.

Example

```
module and3(input a, b, c,
            output y);
  assign y = a & b & c;
endmodule

module inv(input x,
           output y);
  assign y = ~x;
endmodule
```

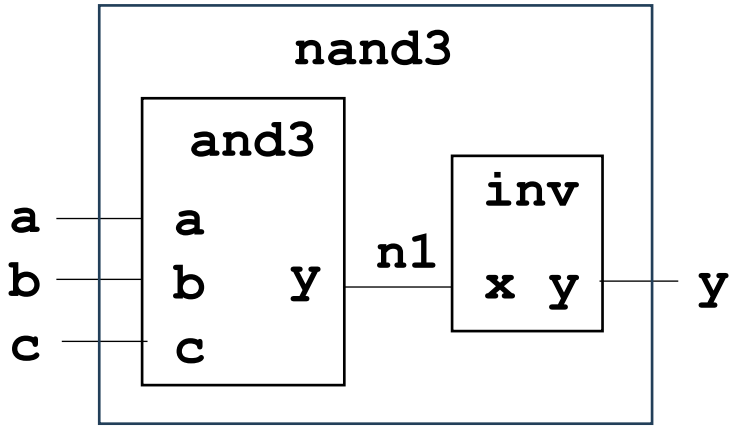


Structural Verilog

- The previous sections discussed behavioral modeling: describing a module in terms of the relationships between inputs and outputs.
- This section examines structural modeling: describing a module in terms of how it is composed of simpler modules.
- At the lowest level (leaf modules) you always need to use behavioral modelling.

Example

```
module and3(input a, b, c,
            output y);
  assign y = a & b & c;
endmodule

module inv(input x,
           output y);
  assign y = ~x;
endmodule

module nand3(input a, b, c,
             output y);
  wire n1, n2;                    // internal signal

  and3 i1(a, b, c, n1);  // instance of and3
  inv  i2(n1, n2);        // instance of inv
  assign y2 = n2
endmodule
```



Structural Verilog

- Alternative specification of port connections

Structural Verilog

- Alternative specification of port connections

```
module and3(input a, b, c,
            output y);
  assign y = a & b & c;
endmodule

module inv(input x,
           output y);
  assign y = ~x;
endmodule
```

Structural Verilog

- Alternative specification of port connections

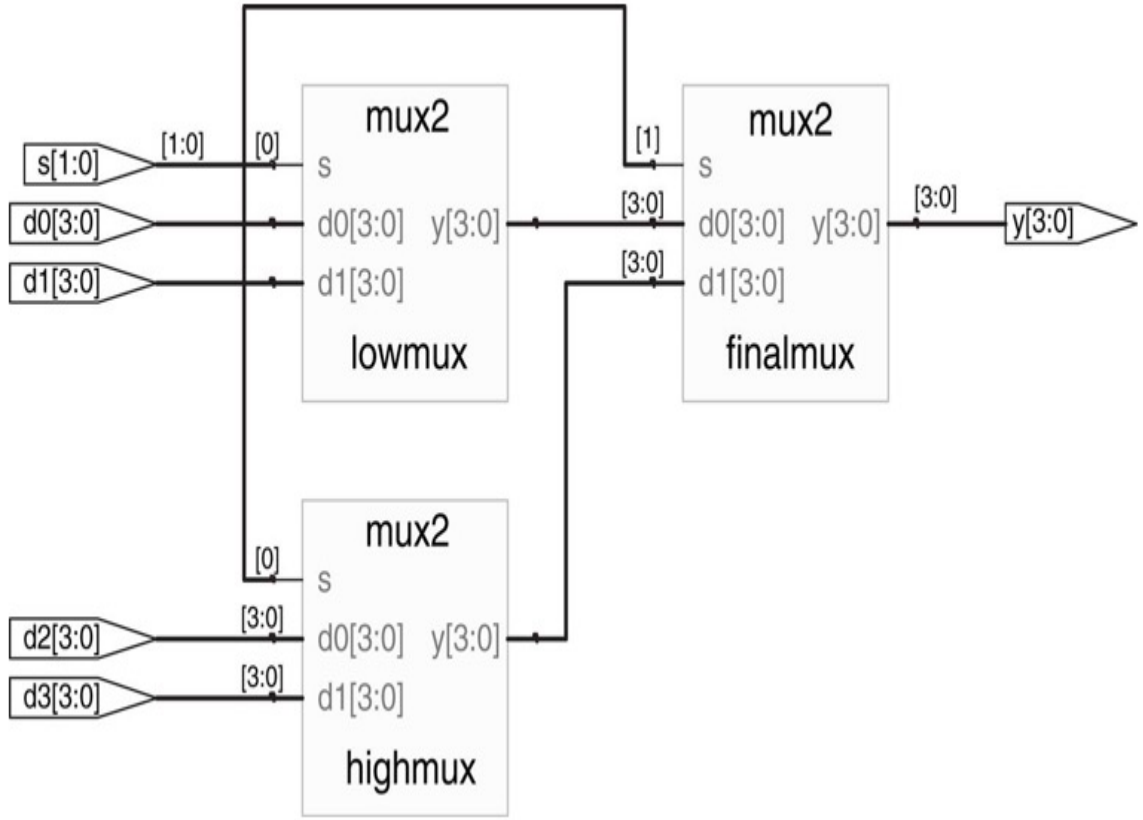
```
module and3(input a, b, c,
            output y);
  assign y = a & b & c;
endmodule

module inv(input x,
           output y);
  assign y = ~x;
endmodule

module nand3(input a, b, c,
             output y);
  wire n1;

  and3 i1(.a(a), .b(b), .c(c), .y(n1));
  inv  i2(.x(n1), .y(y));
endmodule
```

Structural modelling: 4:1 multiplexer

```
module mux4(input [3:0] d0, d1, d2, d3,
            input [1:0] s,
            output [3:0] y);

   wire [3:0] low, high;

   mux2 lowmux(d0, d1, s[0], low);
   mux2 highmux(d2, d3, s[0], high);
   mux2 finalmux(low, high, s[1], y);
endmodule
```

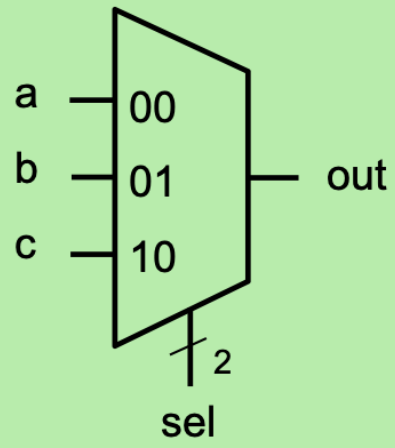
Structural modelling: 4:1 multiplexer

```
module mux4(input [3:0] d0, d1, d2, d3,
            input [1:0] s,
            output [3:0] y);

   wire [3:0] low, high;

   mux2 lowmux(d0, d1, s[0], low);
   mux2 highmux(d2, d3, s[0], high);
   mux2 finalmux(low, high, s[1], y);
endmodule
```



Dangers of Verilog: Incomplete specification

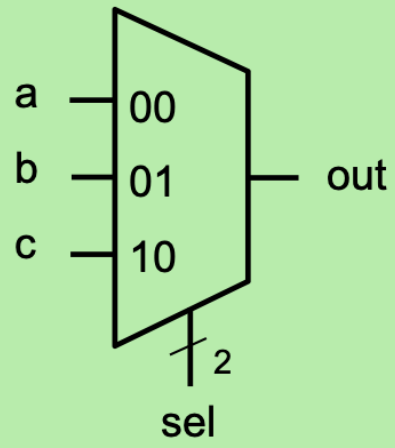
Intended design



3-to-1 MUX
('11' input is a don't-care)

Dangers of Verilog: Incomplete specification

Intended design



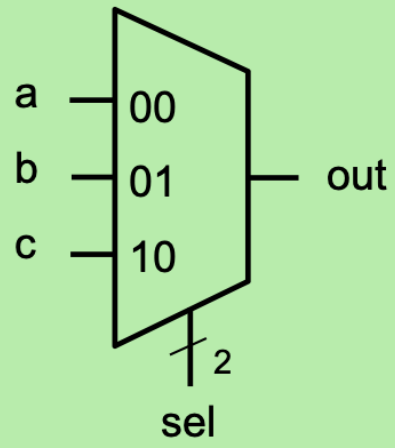
3-to-1 MUX
('11' input is a don't-care)

Written Verilog Code:

```
module mux3(input a, b, c,
            input [1:0] sel,
            output out);
reg out_;
assign out = out_;
always@(a or b or c or sel)
begin
        case (sel)
           2'b00: out_ = a;
           2'b01: out_ = b;
           2'b10: out_ = c;
        endcase
end

endmodule
```

Dangers of Verilog: Incomplete specification

Intended design



3-to-1 MUX
('11' input is a don't-care)

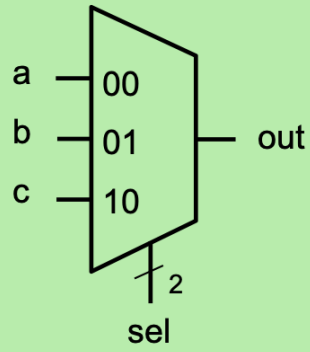
Written Verilog Code:

```
module mux3(input a, b, c,
            input [1:0] sel,
            output out);
reg out_;
assign out = out_;
always@(a or b or c or sel)
begin
        case (sel)
           2'b00: out_ = a;
           2'b01: out_ = b;
           2'b10: out_ = c;
        endcase
end

endmodule
```

Is this a 3-1 multiplexer?

Dangers of Verilog: Incomplete specification

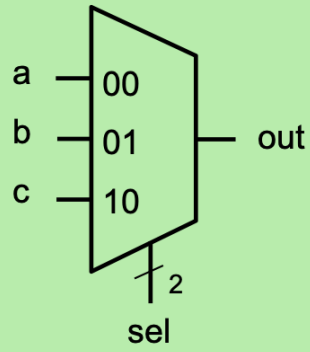
Intended design



3-to-1 MUX
('11' input is a don't-care)

Dangers of Verilog: Incomplete specification

Intended design



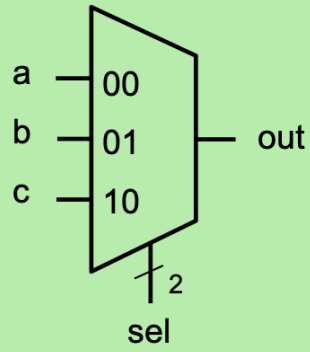
3-to-1 MUX
('11' input is a don't-care)

Written Verilog Code:

```
module mux3(input a, b, c,
            input [1:0] sel,
            output out);
reg out_
assign out = out_
always@(a or b or c or sel)
begin
       case (sel)
          2'b00: out_ = a;
          2'b01: out_ = b;
          2'b10: out_ = c;
       endcase
end
endmodule
```

Dangers of Verilog: Incomplete specification

Intended design

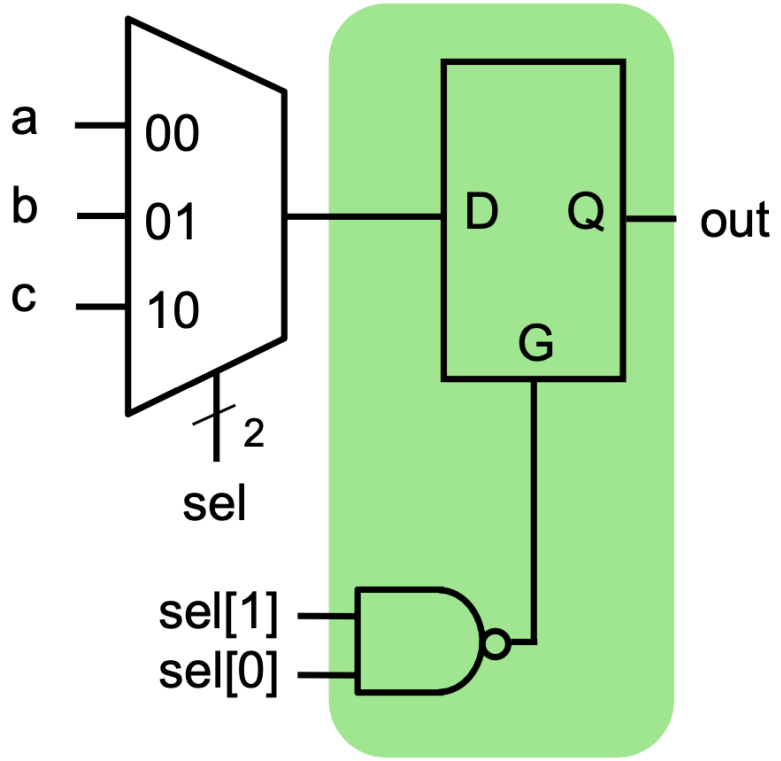


3-to-1 MUX
('11' input is a don't-care)

Written Verilog Code:

```
module mux3(input a, b, c,
            input [1:0] sel,
            output out);
reg out_
assign out = out_
always@(a or b or c or sel)
begin
        case (sel)
            2'b00: out_ = a;
            2'b01: out_ = b;
            2'b10: out_ = c;
        endcase
end
endmodule
```

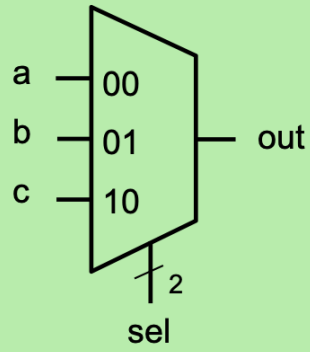
Synthesized result



if `out` is not assigned during any pass through the always block, then the previous value must be retained!

Dangers of Verilog: Incomplete specification

Intended design

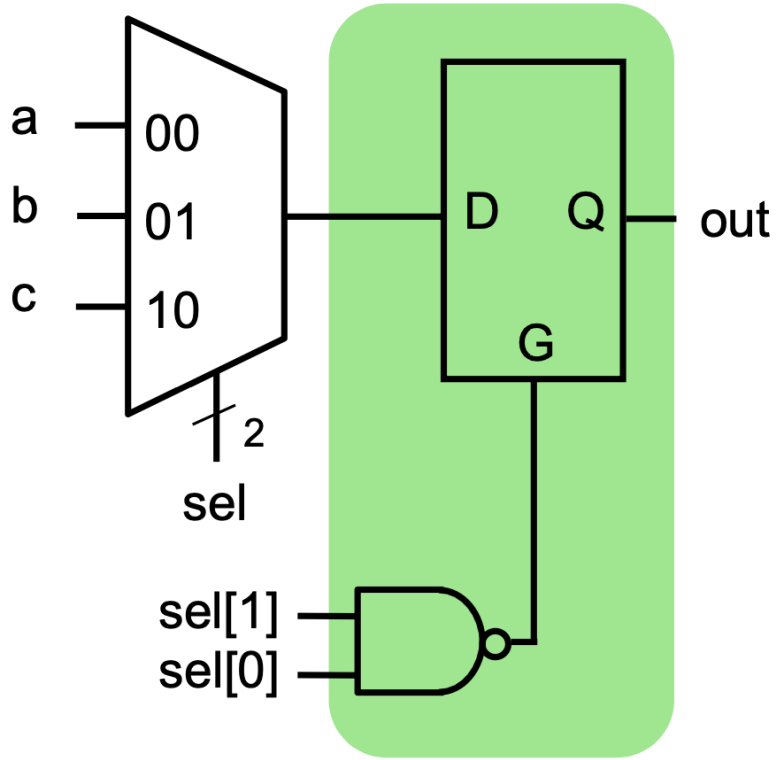


3-to-1 MUX
('11' input is a don't-care)

Written Verilog Code:

```
module mux3(input a, b, c,
            input [1:0] sel,
            output out);
reg out_
assign out = out_
always@(a or b or c or sel)
begin
        case (sel)
            2'b00: out_ = a;
            2'b01: out_ = b;
            2'b10: out_ = c;
        endcase
end
endmodule
```

Synthesized result



if `out` is not assigned during any pass through the always block, then the previous value must be retained!

Latch memory "latches" old data when G=0

Avoiding incomplete specification

```
always@(a or b or c or sel)
begin
        out_ = 1'bx;
        case (sel)
           2'b00: out_ = a;
           2'b01: out_ = b;
           2'b10: out_ = c;
        endcase
end
```

Avoiding incomplete specification

- **Precede all conditionals with a default assignment for all signals assigned within them...**

```
always@(a or b or c or sel)
begin
        out_ = 1'bx;
        case (sel)
           2'b00: out_ = a;
           2'b01: out_ = b;
           2'b10: out_ = c;
        endcase
end
```

```
always@(a or b or c or sel)
begin
        case (sel)
           2'b00: out_ = a;
           2'b01: out_ = b;
           2'b10: out_ = c;
           default: out_ = 1'bx;
        endcase
end
```

Avoiding incomplete specification

- **Precede all conditionals with a default assignment for all signals assigned within them...**

```
always@(a or b or c or sel)
begin
        out_ = 1'bx;
        case (sel)
           2'b00: out_ = a;
           2'b01: out_ = b;
           2'b10: out_ = c;
        endcase
end
```

- **...or, fully specify all branches of conditionals and assign all signals from all branches**

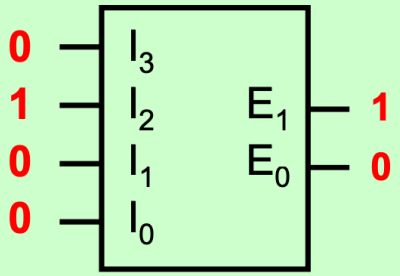
For each `if`, include `else` For each `case`, include `default`

```
always@(a or b or c or sel)
begin
        case (sel)
           2'b00: out_ = a;
           2'b01: out_ = b;
           2'b10: out_ = c;
           default: out_ = 1'bx;
        endcase
end
```

Dangers of Verilog: Priority logic

Intended design

4-to-2 Binary Encoder

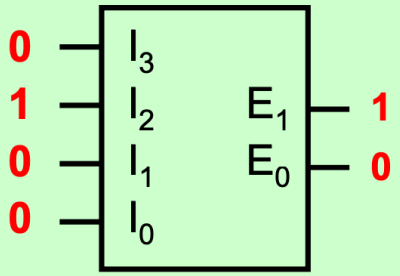


I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Dangers of Verilog: Priority logic

Intended design

4-to-2 Binary Encoder



I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

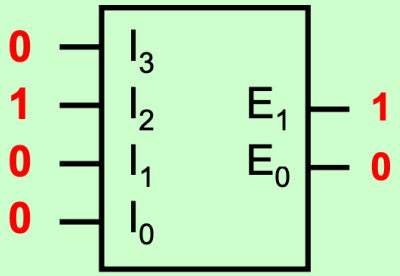
Written Verilog Code:

```
module encoder(input [3:0]i,
                    output [1:0] e);
        reg [1:0]e_;
        assign e<=e_;
always @(i)
begin
        if (i[0]) e = 2'b00;
        else if (i[1]) e_ = 2'b01;
        else if (i[2]) e_ = 2'b10;
        else if (i[3]) e_ = 2'b11;
        else e_ = 2'bxx;
end
endmodule
```

Dangers of Verilog: Priority logic

Intended design

4-to-2 Binary Encoder



I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Written Verilog Code:

```
module encoder(input [3:0]i,
                    output [1:0] e);
        reg [1:0]e_;
        assign e<=e_;
always @(i)
begin
        if (i[0]) e = 2'b00;
        else if (i[1]) e_ = 2'b01;
        else if (i[2]) e_ = 2'b10;
        else if (i[3]) e_ = 2'b11;
        else e_ = 2'bxx;
end
endmodule
```

What is the resulting circuit?

Dangers of Verilog: Priority logic

Intent: **if more than one input is 1, the result is a don't-care.**

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Dangers of Verilog: Priority logic

Intent: **if more than one input is 1, the result is a don't-care.**

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Code: **if i[0] is 1, the result is 00 regardless of the other inputs.** ***i[0] takes the highest priority.***

```
if (i[0]) e = 2'b00;
else if (i[1]) e = 2'b01;
else if (i[2]) e = 2'b10;
else if (i[3]) e = 2'b11;
else e = 2'bxx;
end
```

Dangers of Verilog: Priority logic

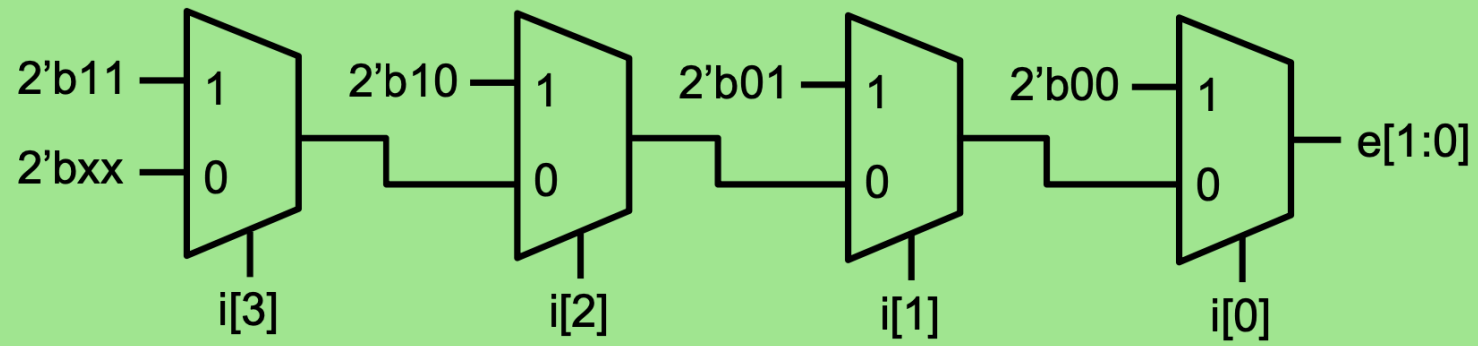
Intent: **if more than one input is 1, the result is a don't-care.**

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Code: **if i[0] is 1, the result is 00 regardless of the other inputs.** ***i[0] takes the highest priority.***

```
if (i[0]) e = 2'b00;
else if (i[1]) e = 2'b01;
else if (i[2]) e = 2'b10;
else if (i[3]) e = 2'b11;
else e = 2'bxx;
end
```

Inferred Result:



Dangers of Verilog: Priority logic

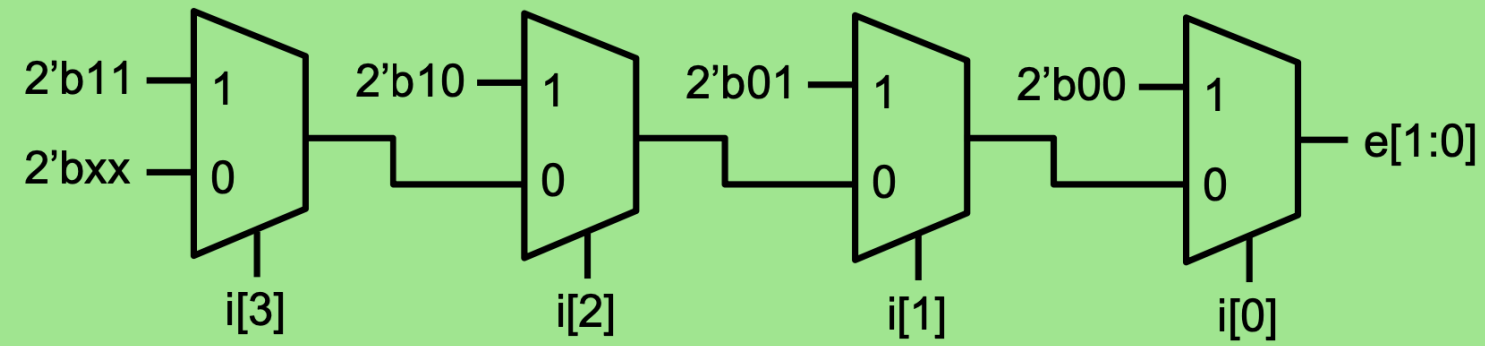
Intent: **if more than one input is 1, the result is a don't-care.**

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Code: **if i[0] is 1, the result is 00 regardless of the other inputs.** ***i[0] takes the highest priority.***

```
if (i[0]) e = 2'b00;
else if (i[1]) e = 2'b01;
else if (i[2]) e = 2'b10;
else if (i[3]) e = 2'b11;
else e = 2'bxx;
end
```

Inferred Result:



`if-else` **and** `case` **statements are interpreted literally! Beware of unintended priority logic.**

Avoiding unintended priority logic

Make sure that if-else and case statements are *parallel*

If mutually exclusive conditions are chosen for each branch...

...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Avoiding unintended priority logic

Make sure that if-else and case statements are *parallel*

If mutually exclusive conditions are chosen for each branch...

...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module encoder(input [3:0]i,
               output [1:0] e);
       reg [1:0]e_;
       assign e<=e_;
always @(i)
begin
       if (i = 4'b0001) e = 2'b00;
       else if (i = 4'b0010) e_ = 2'b01;
       else if (i = 4'b0100) e_ = 2'b10;
       else if (i = 4'b1000) e_ = 2'b11;
       else e_ = 2'bxx;
end
endmodule
```

Avoiding unintended priority logic

Make sure that if-else and case statements are *parallel*

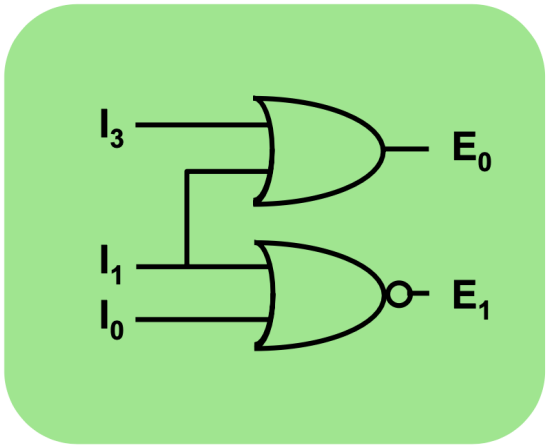
If mutually exclusive conditions are chosen for each branch...

...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module encoder(input [3:0]i,
               output [1:0] e);
       reg [1:0]e_;
       assign e<=e_;
always @(i)
begin
       if (i = 4'b0001) e = 2'b00;
       else if (i = 4'b0010) e_ = 2'b01;
       else if (i = 4'b0100) e_ = 2'b10;
       else if (i = 4'b1000) e_ = 2'b11;
       else e_ = 2'bxx;
end
endmodule
```

Minimized synthesis result



Avoiding unintended priority logic

Make sure that if-else and case statements are *parallel*

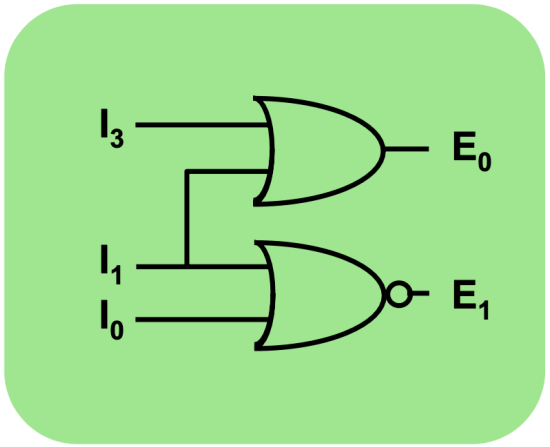
If mutually exclusive conditions are chosen for each branch...

...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module encoder(input [3:0]i,
               output [1:0] e);
       reg [1:0]e_;
       assign e<=e_;
always @(i)
begin
       if (i = 4'b0001) e = 2'b00;
       else if (i = 4'b0010) e_ = 2'b01;
       else if (i = 4'b0100) e_ = 2'b10;
       else if (i = 4'b1000) e_ = 2'b11;
       else e_ = 2'bxx;
end
endmodule
```

Minimized synthesis result



Optimization

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Avoiding unintended priority logic

Make sure that if-else and case statements are *parallel*

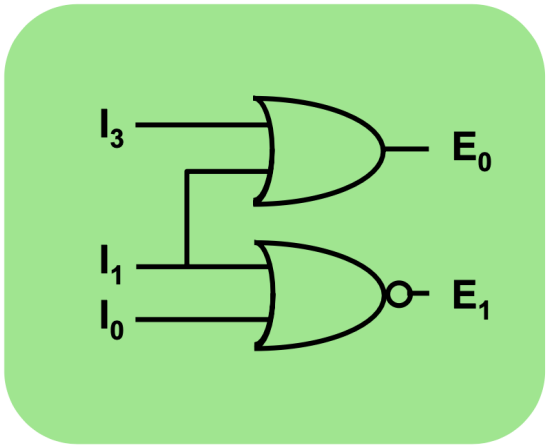
If mutually exclusive conditions are chosen for each branch...

...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module encoder(input [3:0]i,
               output [1:0] e);
       reg [1:0]e_;
       assign e<=e_;
always @(i)
begin
       if (i = 4'b0001) e = 2'b00;
       else if (i = 4'b0010) e_ = 2'b01;
       else if (i = 4'b0100) e_ = 2'b10;
       else if (i = 4'b1000) e_ = 2'b11;
       else e_ = 2'bxx;
end
endmodule
```

Minimized synthesis result



Why I_2 is missing?

Optimization

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Avoiding unintended priority logic

Make sure that if-else and case statements are *parallel*

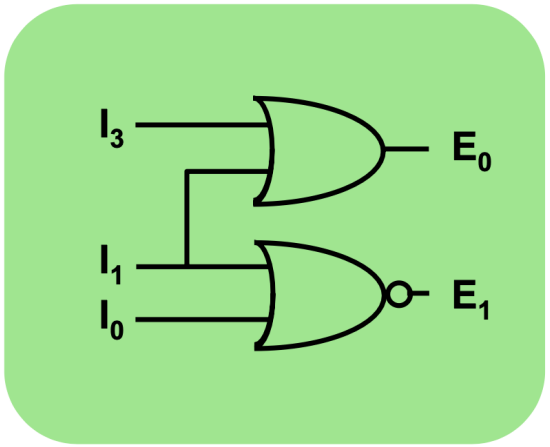
If mutually exclusive conditions are chosen for each branch...

...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module encoder(input [3:0]i,
               output [1:0] e);
       reg [1:0]e_;
       assign e<=e_;
always @(i)
begin
       if (i = 4'b0001) e = 2'b00;
       else if (i = 4'b0010) e_ = 2'b01;
       else if (i = 4'b0100) e_ = 2'b10;
       else if (i = 4'b1000) e_ = 2'b11;
       else e_ = 2'bxx;
end
endmodule
```

Minimized synthesis result



Why I_2 is missing?

Optimization

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

E_0 =1 when only I_1 or I_3 is 1

E_1 =1 when both I_0 and I_1 are 0

Avoiding unintended priority logic

Make sure that if-else and case statements are *parallel*

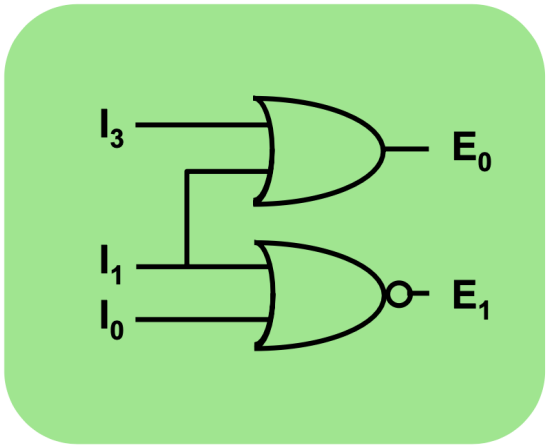
If mutually exclusive conditions are chosen for each branch...

...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module encoder(input [3:0]i,
               output [1:0] e);
       reg [1:0]e_;
       assign e<=e_;
always @(i)
begin
       if (i = 4'b0001) e = 2'b00;
       else if (i = 4'b0010) e_ = 2'b01;
       else if (i = 4'b0100) e_ = 2'b10;
       else if (i = 4'b1000) e_ = 2'b11;
       else e_ = 2'bxx;
end
endmodule
```

Minimized synthesis result



Why I_2 is missing?

Optimization

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

E_0 =1 when only I_1 or I_3 is 1

E_1 =1 when both I_0 and I_1 are 0

Therefore, I_2 can be safely optimized

Delays

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Delays

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

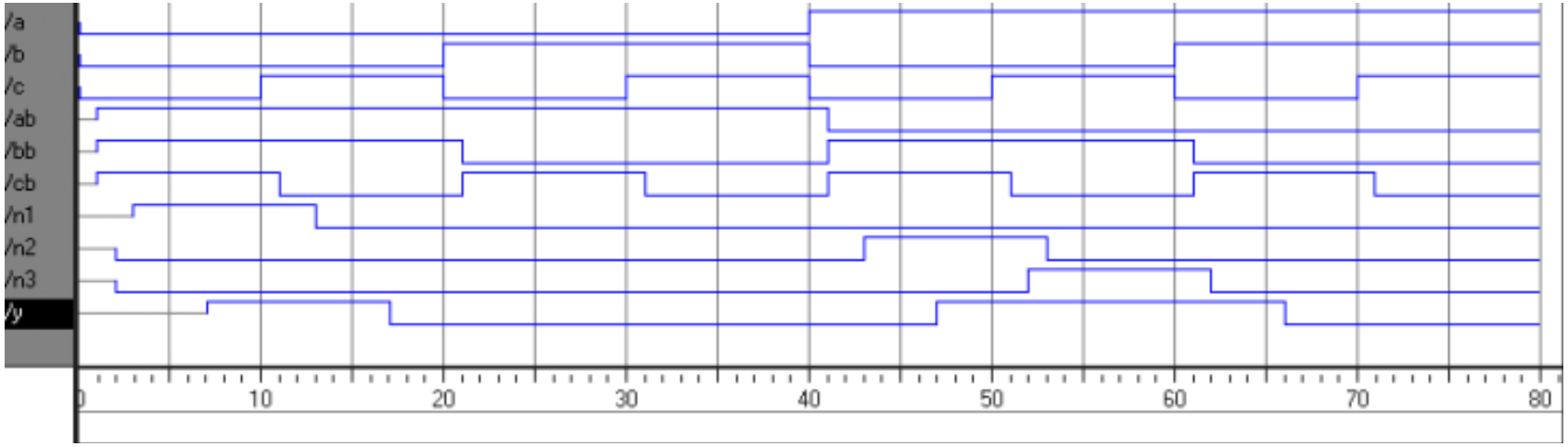
- The delay for each assignment is specified: time between change of input and update of output.
- Delays are for simulation only! They do not determine the delay of the circuit after synthesis.

Delays

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

- The delay for each assignment is specified: time between change of input and update of output.
- Delays are for simulation only! They do not determine the delay of the circuit after synthesis.

Simulation

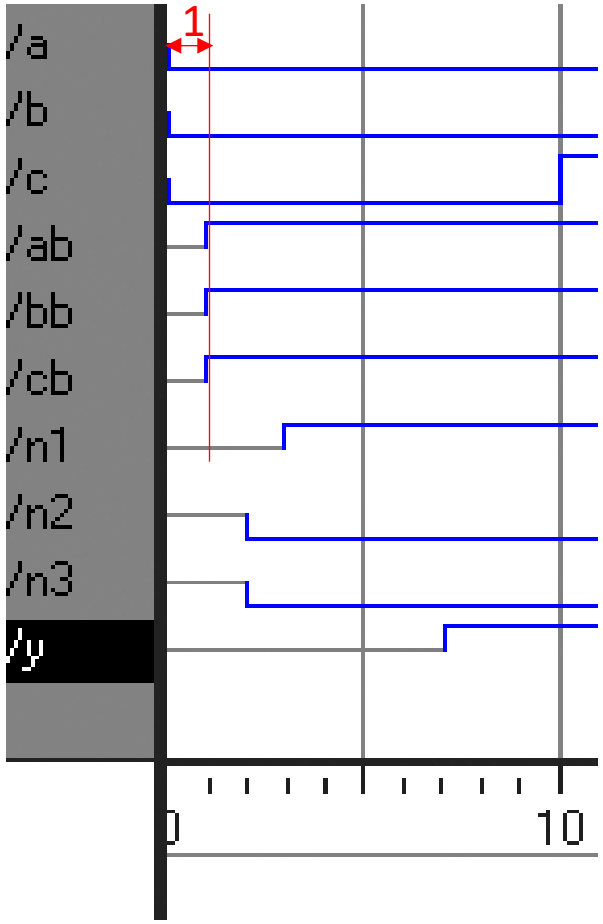


Delays

Updates triggered by events on a, b, c

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #3 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation

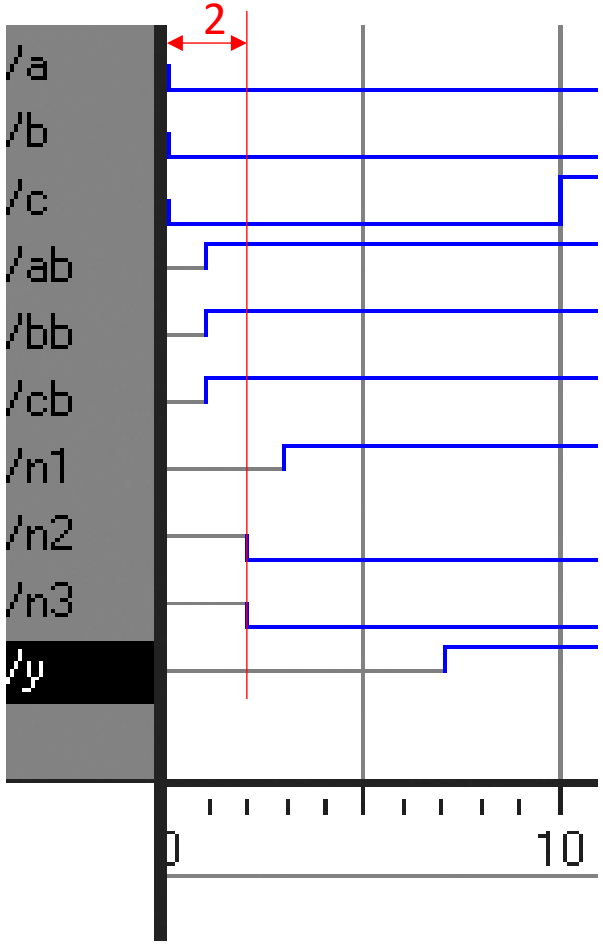


Delays

Updates triggered by event on a

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #3 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation

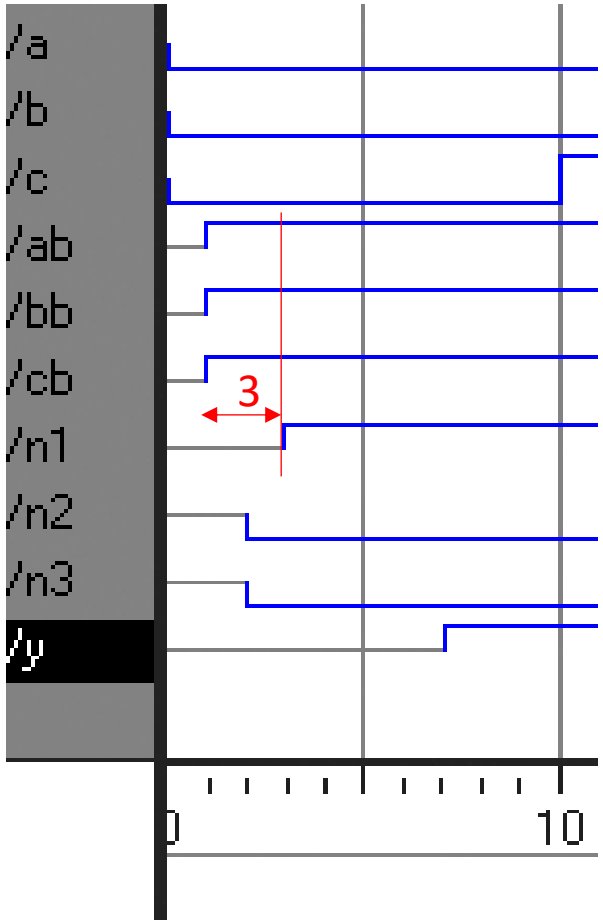


Delays

Update triggered by events on ab, bb, cb

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #3 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation

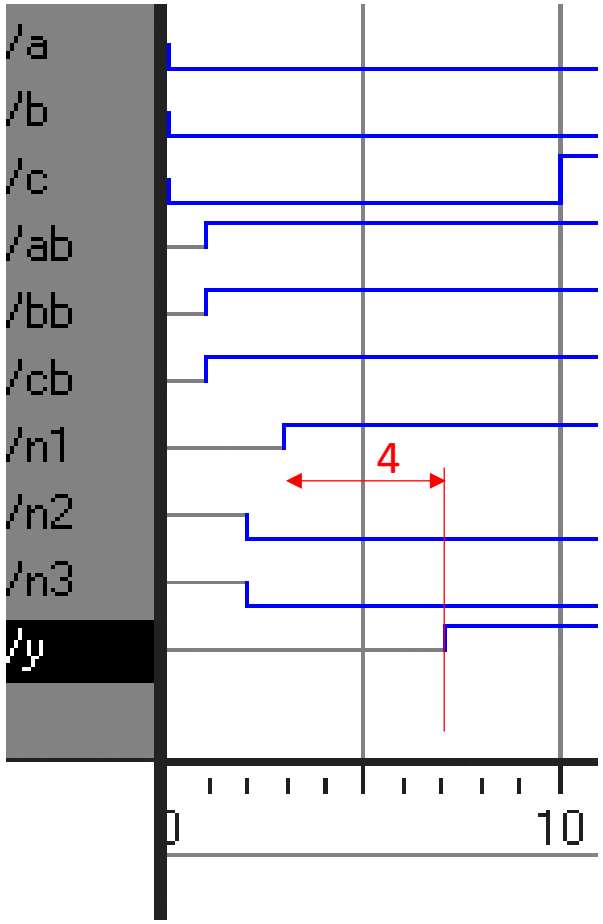


Delays

Update triggered by event on n1

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #3 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation



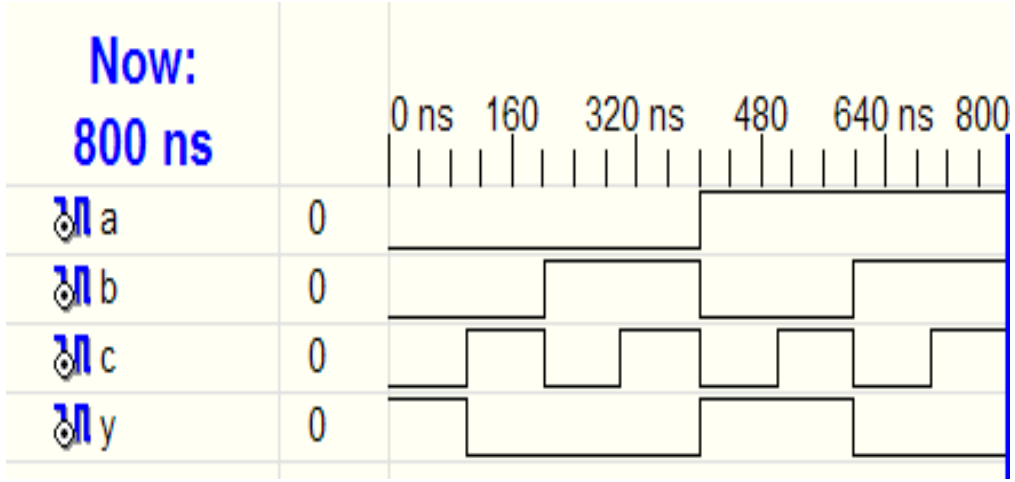
Verilog simulation & synthesis

```
module example(input a, b, c,
               output y);
   assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Verilog simulation & synthesis

```
module example(input a, b, c,
               output y);
   assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

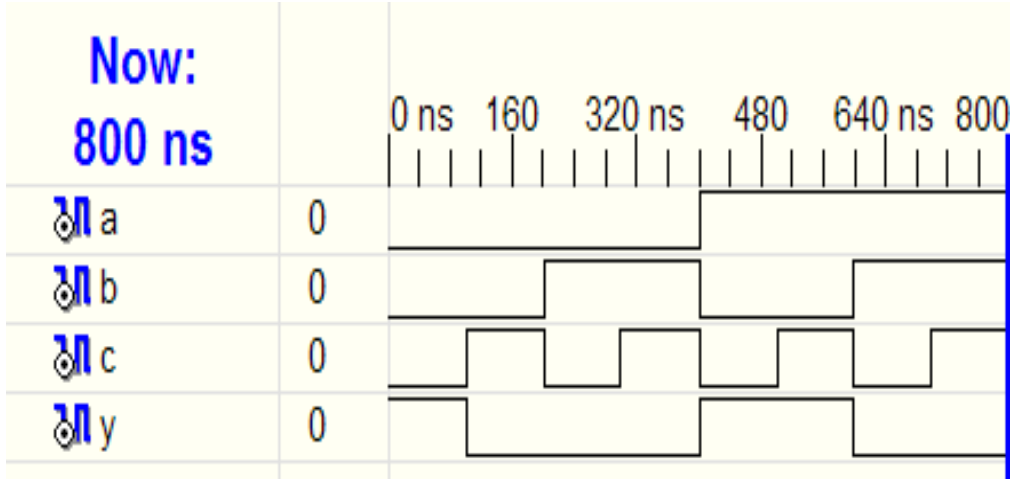
Simulation Results



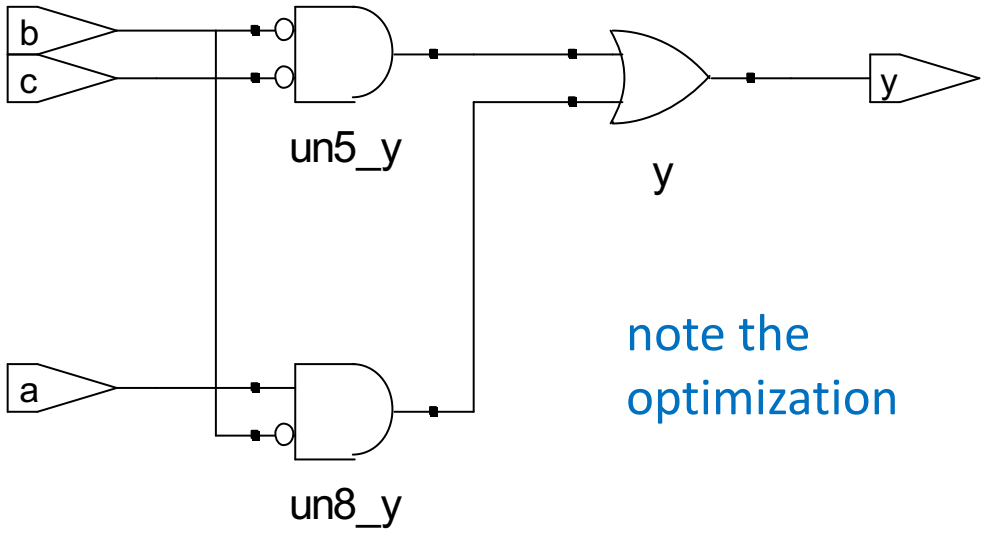
Verilog simulation & synthesis

```
module example(input a, b, c,
               output y);
   assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Simulation Results



Synthesis Results



note the optimization

Verilog testbenches

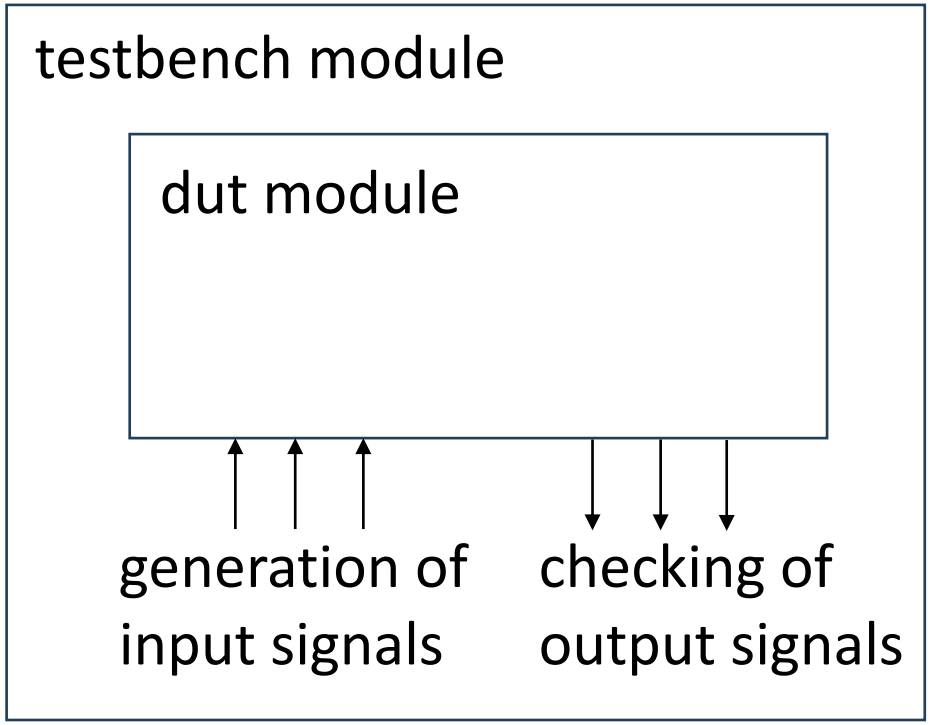
- HDL module that **tests another module**: *device under test* (dut)

Verilog testbenches

- HDL module that **tests another module**: *device under test* (dut)
- **Not synthesizable**

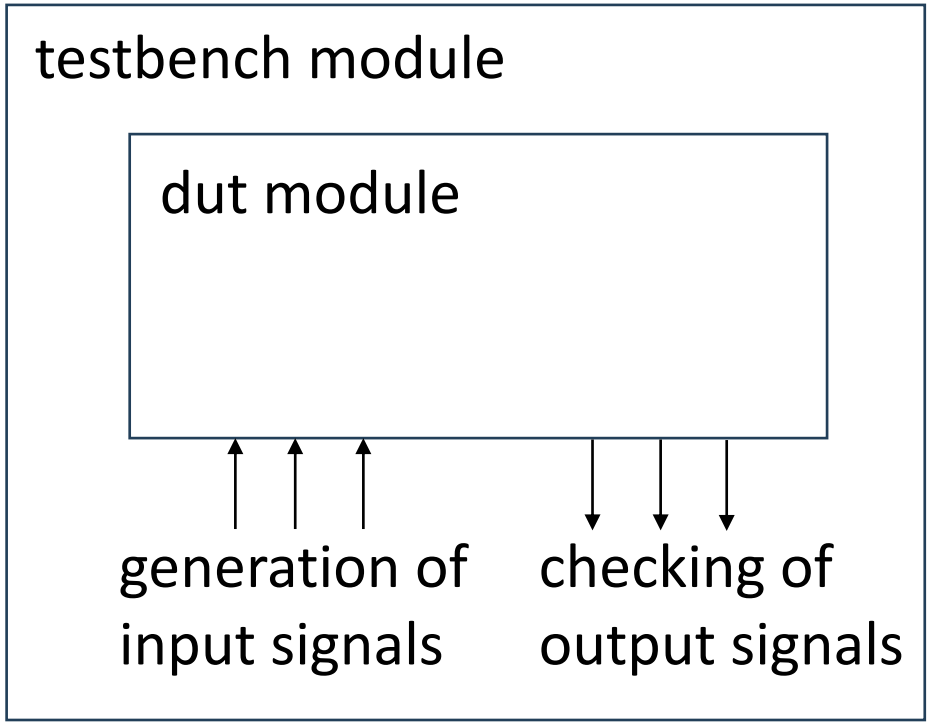
Verilog testbenches

- HDL module that **tests another module**: *device under test* (dut)
- **Not synthesizable**



Verilog testbenches

- HDL module that **tests another module**: *device under test* (dut)
- **Not synthesizable**
- **Types:**
  - Simple
  - Self-checking



Testbenches

Write Verilog code to implement the following function in hardware. Name the module **`myfunction`**.

`y = bc + ab`

```
module myfunction(input a, b, c,
                  output y);
  assign y = ~b & ~c | a & ~b;
endmodule
```

Testbench 1: Simple testbench

```
module tb1();
```

Testbench 1: Simple testbench

```
module tb1();
  reg a, b, c;
  wire y;
```

Testbench 1: Simple testbench

```
module tb1();
  reg a, b, c;
  wire y;
  // instantiate device under test
  myfunction dut(a, b, c, y);
```

Testbench 1: Simple testbench

```
module tb1();
  reg a, b, c;
  wire y;
  // instantiate device under test
  myfunction dut(a, b, c, y);
  // apply new set of inputs every 10 time-units
  initial begin
    a = 0; b = 0; c = 0; #10; //apply inputs, wait 10ns
```

Testbench 1: Simple testbench

```
module tb1();
  reg a, b, c;
  wire y;
  // instantiate device under test
  myfunction dut(a, b, c, y);
  // apply new set of inputs every 10 time-units
  initial begin
    a = 0; b = 0; c = 0; #10; //apply inputs, wait 10ns
                  c = 1; #10; //change the value of c, wait 10ns
```

Testbench 1: Simple testbench

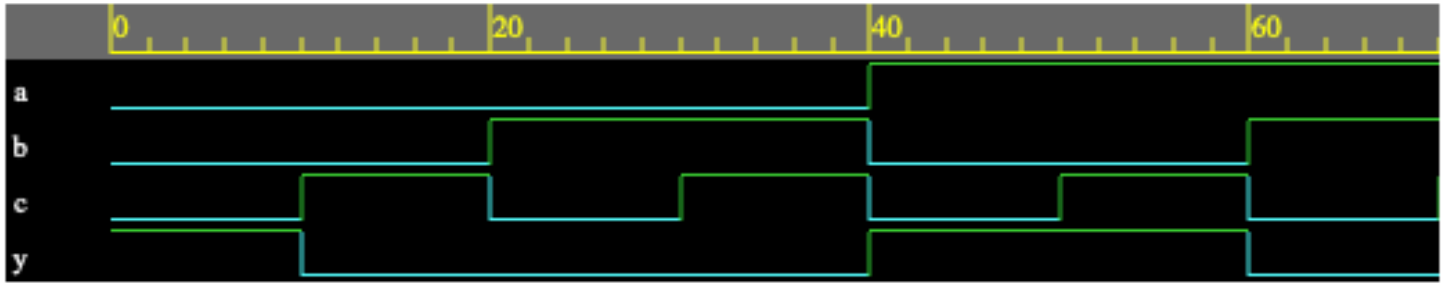
```
module tb1();
  reg a, b, c;
  wire y;
  // instantiate device under test
  myfunction dut(a, b, c, y);
  // apply new set of inputs every 10 time-units
  initial begin
    a = 0; b = 0; c = 0; #10; //apply inputs, wait 10ns
                  c = 1; #10; //change the value of c, wait 10ns
           b = 1; c = 0; #10; //change the values of b & c, wait 10ns
```

Testbench 1: Simple testbench

```
module tb1();
  reg a, b, c;
  wire y;
  // instantiate device under test
  myfunction dut(a, b, c, y);
  // apply new set of inputs every 10 time-units
  initial begin
    a = 0; b = 0; c = 0; #10; //apply inputs, wait 10ns
                  c = 1; #10; //change the value of c, wait 10ns
           b = 1; c = 0; #10; //change the values of b & c, wait 10ns
                  c = 1; #10; //etc
    a = 1; b = 0; c = 0; #10;
                  c = 1; #10;
           b = 1; c = 0; #10;
                  c = 1; #10;
  end
endmodule
```

Simulation result

```
module tb1();
  reg a, b, c;
  wire y;
  // instantiate device under test
  myfunction dut(.a(a), .b(b), .c(c), .y(y));
  // apply new set of inputs every 10 time-units
  initial begin
    a = 0; b = 0; c = 0; #10; //apply inputs, wait 10ns
                  c = 1; #10; //change the value of c, wait 10ns
           b = 1; c = 0; #10; //change the values of b & c, wait 10ns
                  c = 1; #10; //etc
    a = 1; b = 0; c = 0; #10;
                  c = 1; #10;
           b = 1; c = 0; #10;
                  c = 1; #10;
  end
endmodule
```



Testbench 2: Self-checking testbench

```
module tb2();
  reg  a, b, c;
  wire y;
  // instantiate dut
  myfunction dut(.a(a), .b(b), .c(c), .y(y));
  // apply inputs, check results one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;  if (y !== 1) $display("000 failed.");
                   c = 1; #10;  if (y !== 0) $display("001 failed.");
           b = 1; c = 0; #10;  if (y !== 0) $display("010 failed.");
                   c = 1; #10;  if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;  if (y !== 1) $display("100 failed.");
                   c = 1; #10;  if (y !== 1) $display("101 failed.");
           b = 1; c = 0; #10;  if (y !== 0) $display("110 failed.");
                   c = 1; #10;  if (y !== 0) $display("111 failed.");
  end
endmodule
```

Thank you