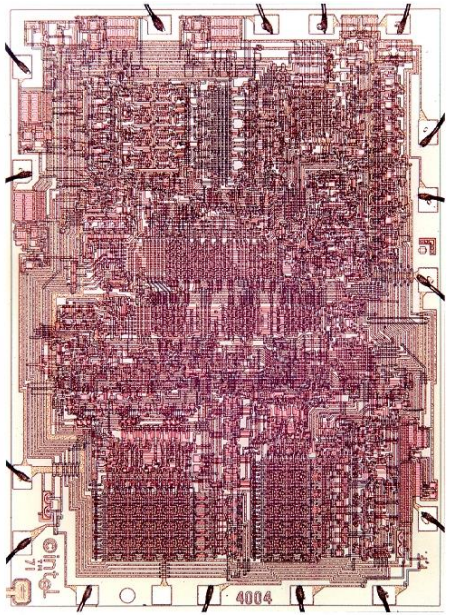
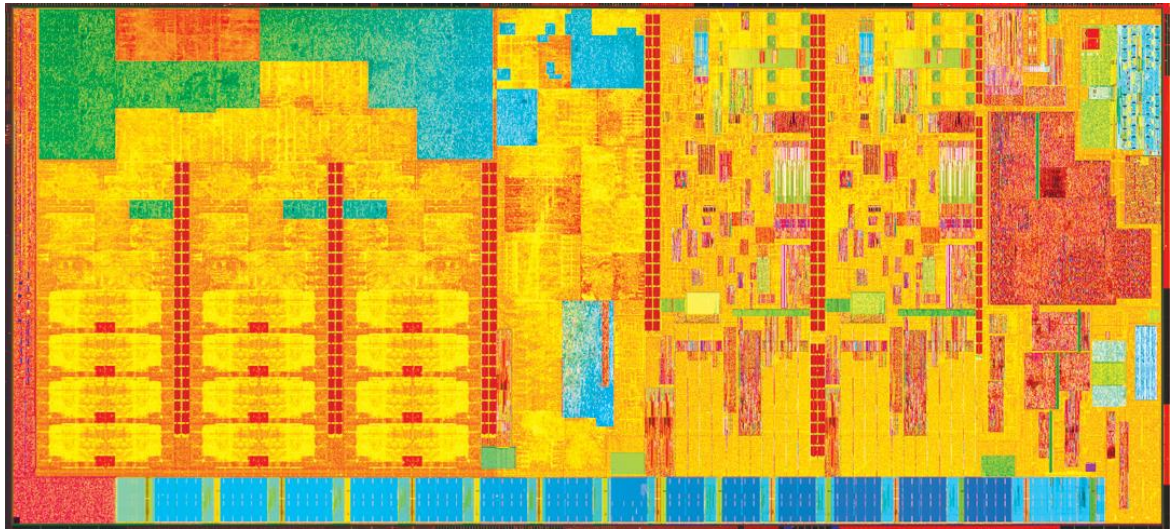
Verilog HDL

044252 - Digital Systems and Computer Structure

Advancements Over the Years





- © Intel 4004 Processor
- Introduced in 1971
- 2300 Transistors
- 108 KHz Clock

- © Intel core i7 Broadwell-E
- Introduced in 2016
- 3.2 Billion Transistors
- 3GHz Clock

Ways to Represent Hardware:

- Draw schematics
  - Hand-drawn
  - Machine-drawn

- Write a netlist
  - Z52BH I1234 (N123, N234, N4567);
- Write primitive Boolean equations
  - AAA = abc DEF + ABC def
- Use a Hardware Description Language (HDL)
  - assign overflow = c31 ^ c32;

Agenda

- Hardware Description Languages and coding styles
- Verilog Building Blocks:
  - Keywords
  - Modules
  - Values / Number representation
  - Data Types
  - Operators
- Assignments and procedural blocks
- Test benches and simulation

Hardware Description Languages (HDLs)

- Textual representation of a digital logic design
  - Can represent specific gates, like a netlist, or more abstract logic
- HDLs are not "programming languages"
  - No, really. Even if they look like it, they are not.
  - For many people, a difficult conceptual leap
- Similar development chain
  - Compiler: source code ➔ assembly code ➔ binary machine code
  - Synthesis tool: HDL source ➔ gate-level specification ➔ hardware

Why Use an HDL?

- Easy to write and edit
- Compact
- Don’t have to follow a maze of lines
- Easy to analyze with various tools

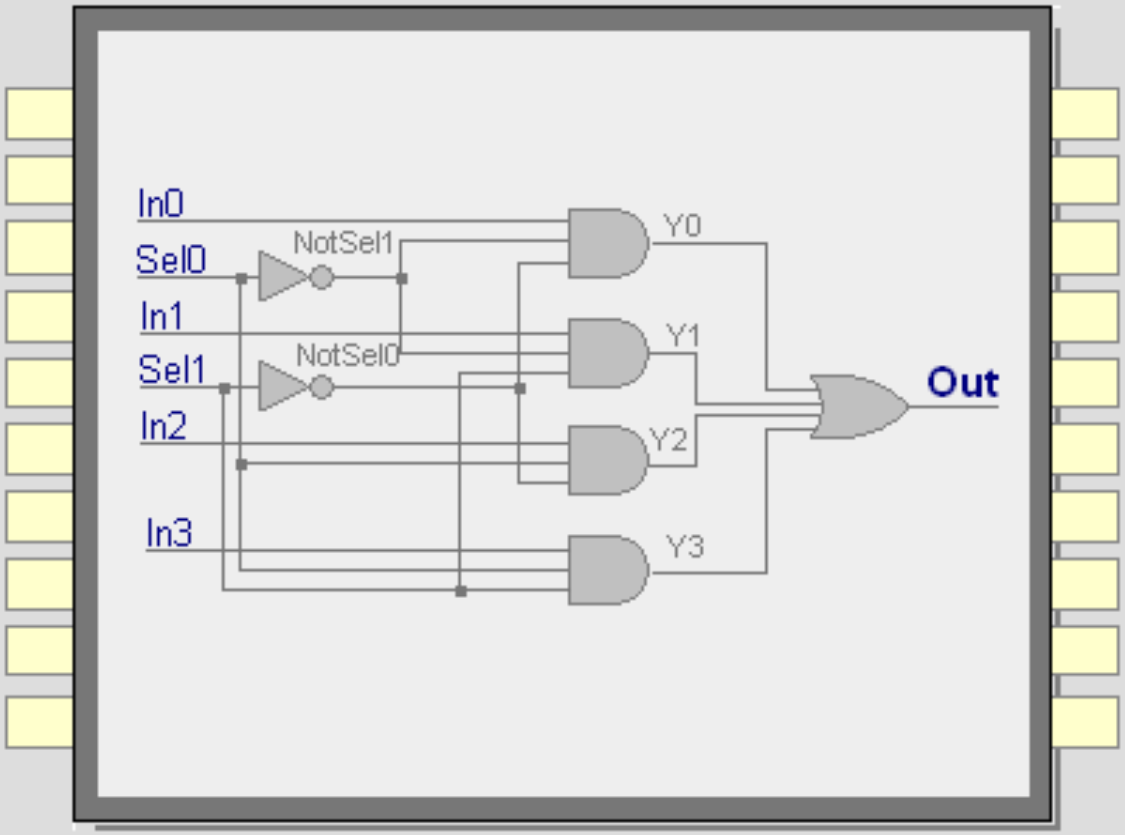
Why not to use an HDL

- You still need to visualize the flow of logic
- A schematic can be a work of art
  - But often isn’t!

Verilog

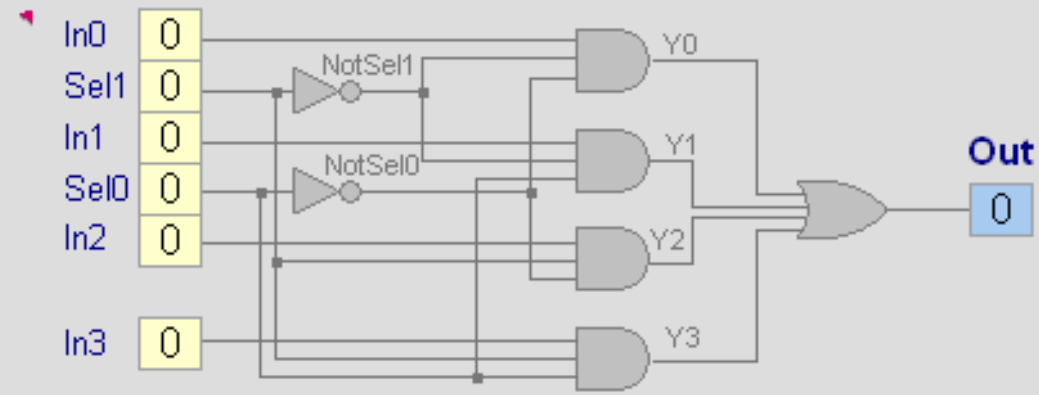
- One of the two leading HDLs - VHDL is the other.
- Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages
- Verilog was introduced in 1985 by Gateway Design System Corporation (Today Cadence)
- IEEE 1364-2005 is the latest Verilog HDL standard
- Verilog is case sensitive (Keywords are in lowercase)
- Verilog is both a behavioral and a structural language

One Language, Many Coding Styles



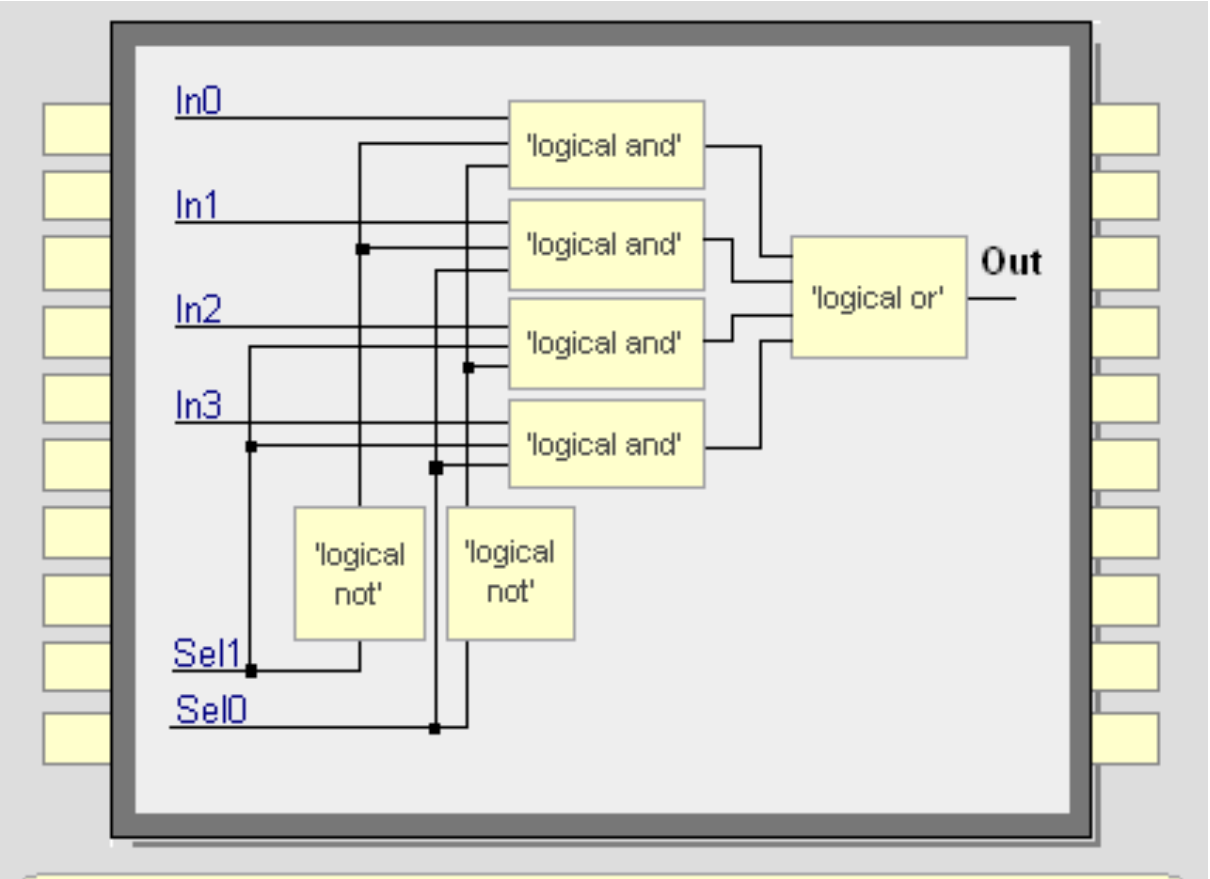
Structural Style. The circuit is specified in terms of instantiations of lower level components (in this case logic gates, which are Verilog primitives) connected with internal signals. The translation of such a specification into a physical circuit is straightforward.

Structural Style: Verilog Code



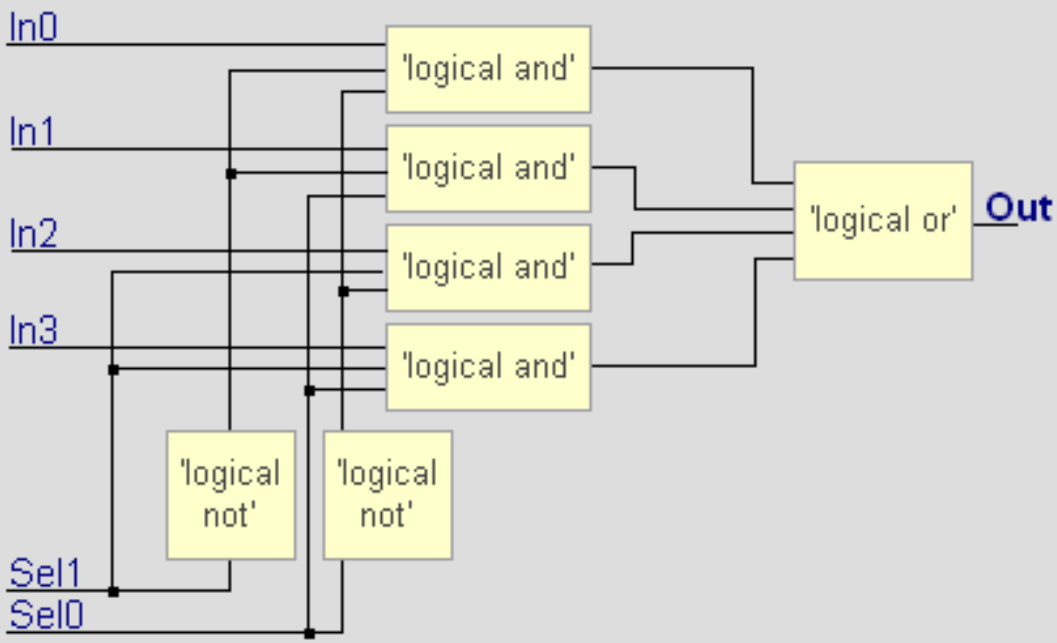

```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;

wire NotSel0, NotSel1;
wire Y0, Y1, Y2, Y3;

not (NotSel0, Sel0);
not (NotSel1, Sel1);
and (Y0, In0, NotSel1, NotSel0);
and (Y1, In1, NotSel1, Sel0);
and (Y2, In2, Sel1, NotSel0);
and (Y3, In3, Sel1, Sel0);
or (Out, Y0, Y1, Y2, Y3);

endmodule
```

One Language, Many Coding Styles (contd.)



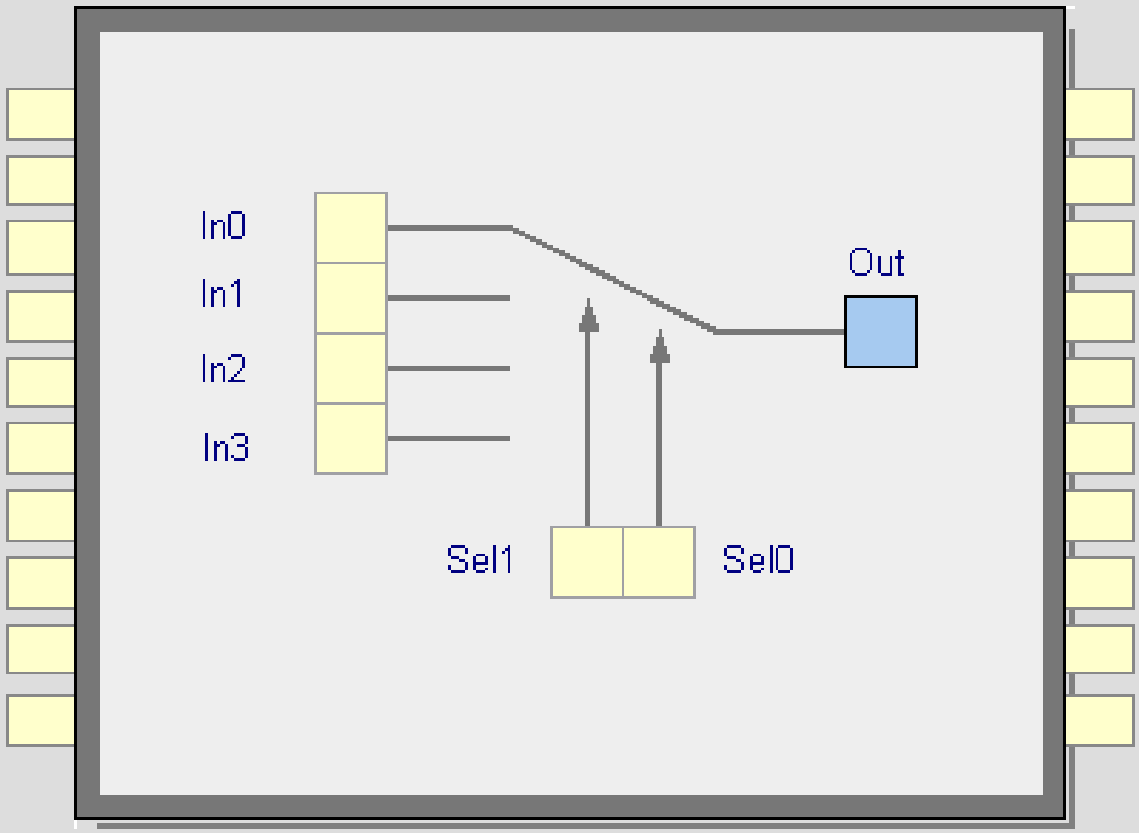
Dataflow Style. This style is similar to logical equations, although in general it is not limited to logical values only. The specification is comprised of expressions made up of input signals and assigned to outputs. In most cases, such an approach can be quite easily translated into a structure and then implemented.

Dataflow Style: Verilog Code



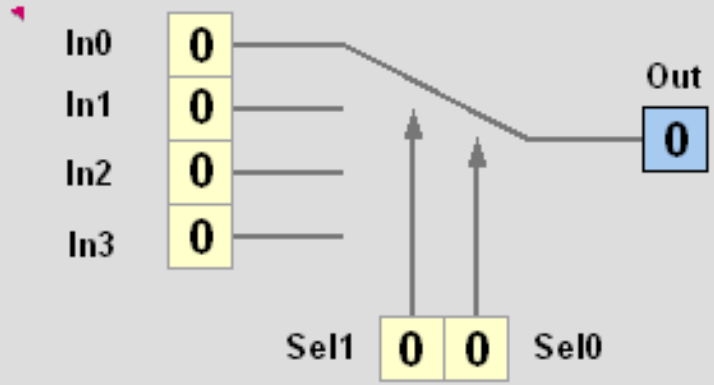

```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);

output Out;
input In0, In1, In2, In3, Sel0, Sel1;

assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)
             | (Sel1 & ~Sel0 & In2)| (Sel1 & Sel0 & In3);

endmodule
```

One Language, Many Coding Styles (contd.)



Behavioral Style. It specifies the circuit in terms of its expected behavior. It is the closest to a natural language description of the circuit functionality, but also the most difficult to synthesize.

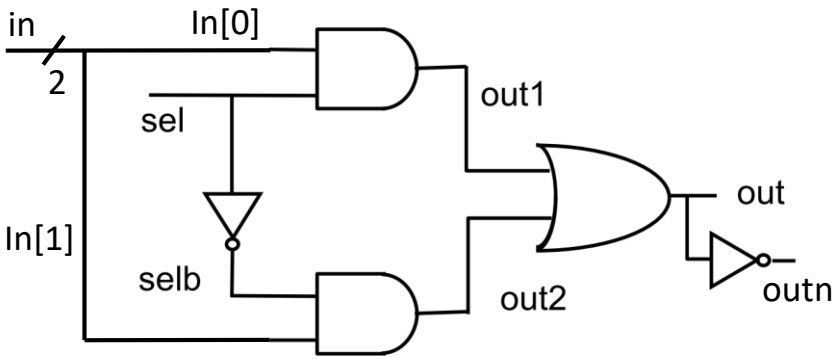
Behavioral Style: Verilog Code




```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;
reg Out;

always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)
  begin
    case ({Sel1, Sel0})
    2'b00 : Out = In0;
    2'b01 : Out = In1;
    2'b10 : Out = In2;
    2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end

endmodule
```

Combinational Example

```
module mux (a, b, out, outbar, sel);
        input [1:0]         in;
        input               sel;
        output              out, outn;
        wire                out1, out2, selb;

        and a1 (out1, in[0], sel);
        not i1 (.O(selb), .I(sel));
        and a2 (out2, in[1] , selb);
        always @(*) begin
                out = out1 | out2;
        end
        assign outn = ~out;
endmodule
```



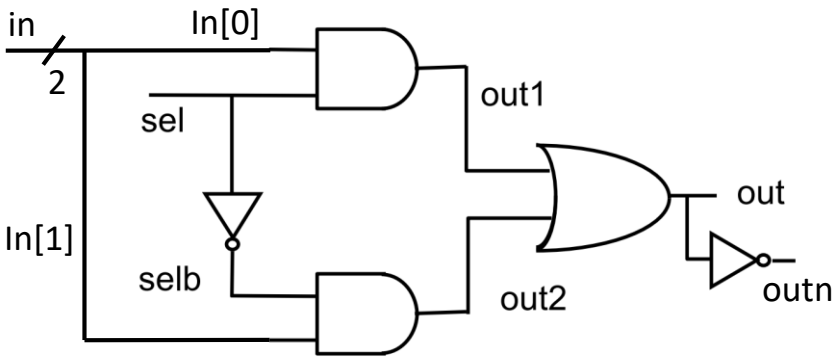
Combinational Example

```
module mux (a, b, out, outbar, sel);
        input [1:0]         in;
        input               sel;
        output              out, outn;
        wire                out1, out2, selb;

        and a1 (out1, in[0], sel);
        not i1 (.O(selb), .I(sel));
        and a2 (out2, in[1] , selb);
        always @(*) begin
                out = out1 | out2;
        end
        assign outn = ~out;
endmodule
```

module "name" (port list)

Opens every module in design

Closed with **endmodule**

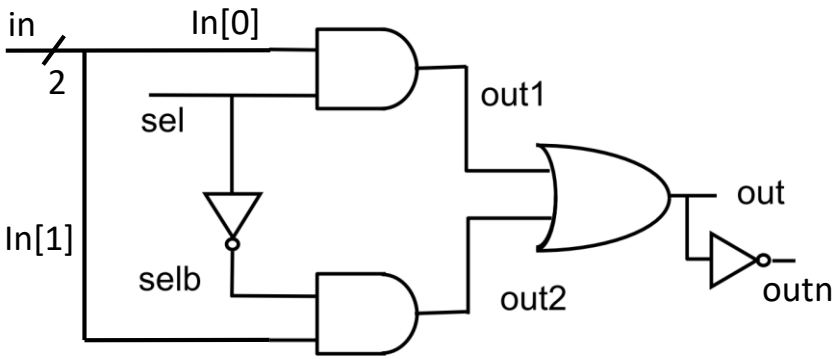


Combinational Example

```
module mux (a, b, out, outbar, sel);
        input [1:0]         in;
        input               sel;
        output              out, outn;
        wire                out1, out2, selb;

        and a1 (out1, in[0], sel);
        not i1 (.O(selb), .I(sel));
        and a2 (out2, in[1] , selb);
        always @(*) begin
                out = out1 | out2;
        end
        assign outn = ~out;
endmodule
```

input/output <range> "names"

Declares port directions

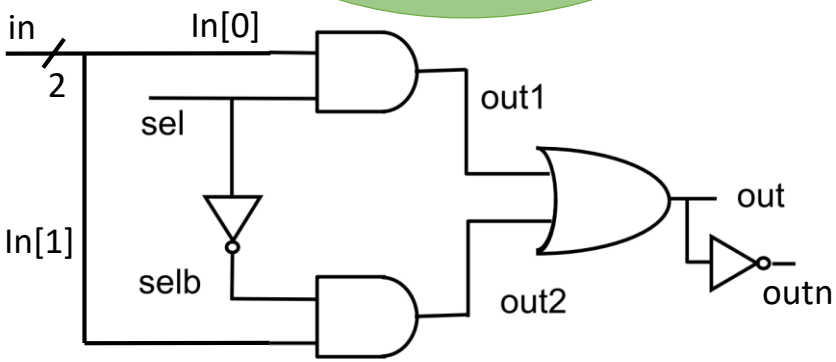


Combinational Example

```
module mux (a, b, out, outbar, sel);
        input [1:0]         in;
        input               sel;
        output              out, outn;
        wire                out1, out2, selb;

        and a1 (out1, in[0], sel);
        not i1 (.O(selb), .I(sel));
        and a2 (out2, in[1] , selb);
        always @(*) begin
                out = out1 | out2;
        end
        assign outn = ~out;
endmodule
```

module instance (ports)

Instantiating logic using modules.
Port binding by name/place.

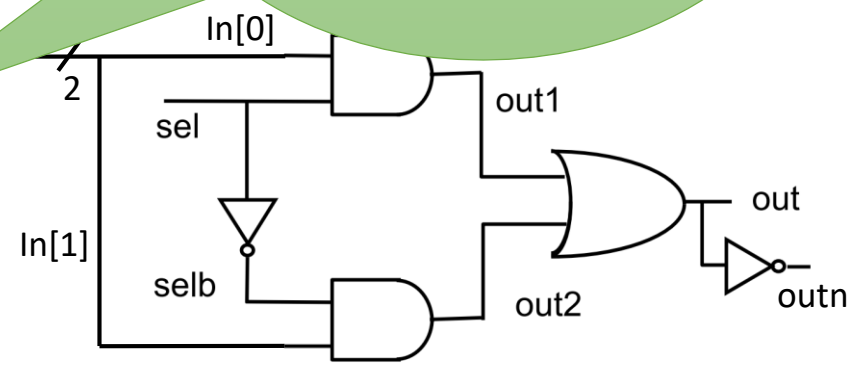


Combinational Example

```
module mux (a, b, out, outbar, sel);
        input [1:0]         in;
        input               sel;
        output              out, outn;
        wire                out1, out2, selb;

        and a1 (out1, in[0], sel);
        not i1 (.O(selb), .I(sel));
        and a2 (out2, in[1] , selb);
        always @(*) begin
                out = out1 | out2;
        end
        assign outn = ~out;
endmodule
```

always @(*)

Procedural assignment
Used for complex behavioral modeling of HW

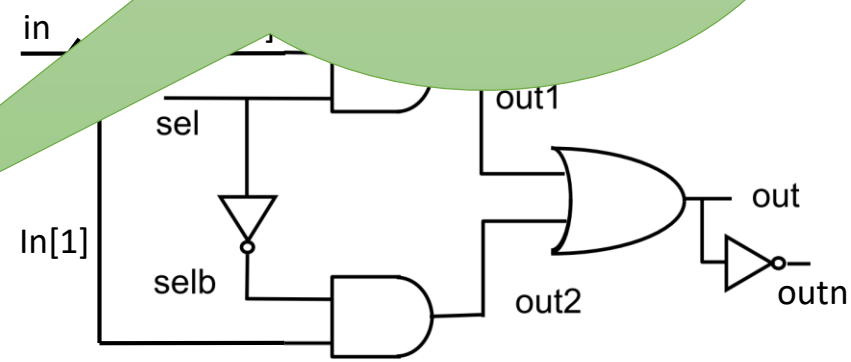


Combinational Example

```
module mux (a, b, out, outbar, sel);
        input [1:0]         in;
        input               sel;
        output              out, outn;
        wire                out1, out2, selb;

        and a1 (out1, in[0], sel);
        not i1 (.O(selb), .I(sel));
        and a2 (out2, in[1] , selb);
        always @(*) begin
                out = out1 | out2;
        end
        assign outn = ~out;
endmodule
```

assign

Concurrent assignment
Used for simple assignment of values to signals



Agenda

- Hardware Description Languages and coding styles
- Verilog Building Blocks:
  - Keywords
  - Modules
  - Values / Number representation
  - Data Types
  - Operators
- Assignments and procedural blocks
- Test benches and simulation

Keywords

- Note : All keywords are defined in lower case
- Examples :
- module, endmodule
- input, output, inout
- reg, integer, real, time
- not, and, nand, or, nor, xor
- parameter
- begin, end
- fork, join

Keywords (contd.)

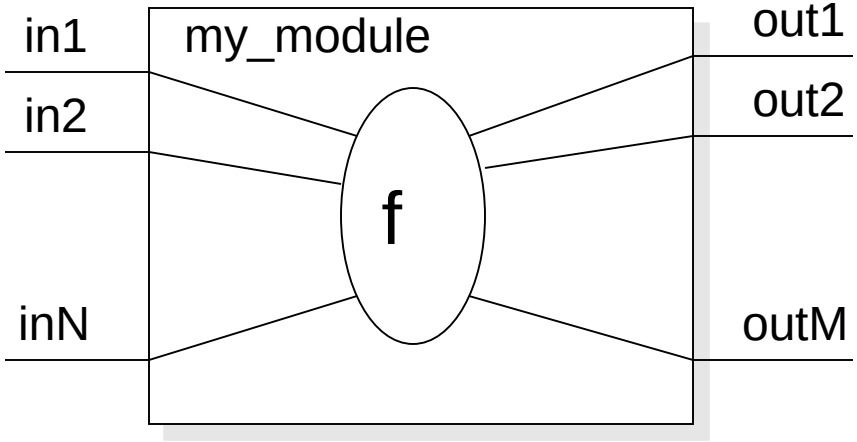
- `module` – fundamental building block for Verilog designs
  - Used to construct <u>design hierarchy</u>
  - Cannot be <u>nested</u>
- `endmodule` – ends a module – not a statement

  => no "`;`"
- Module Declaration
  - `module` *module_name* `(`*module_port*, *module_port*, …`);`
  - Example: `module full_adder (A, B, c_in, c_out, S);`

Keywords (contd.)

- Input Declaration
  - Scalar
    - `input` *list of input identifiers*;
    - Example: `input A, B, c_in;`
  - Vector
    - `input`[*range*] *list of input identifiers*;
    - Example: `input[15:0] A, B, data;`
- Output Declaration
  - Scalar Example: `output c_out, OV, MINUS;`
  - Vector Example: `output[7:0] REG_IN, data_out;`

Module

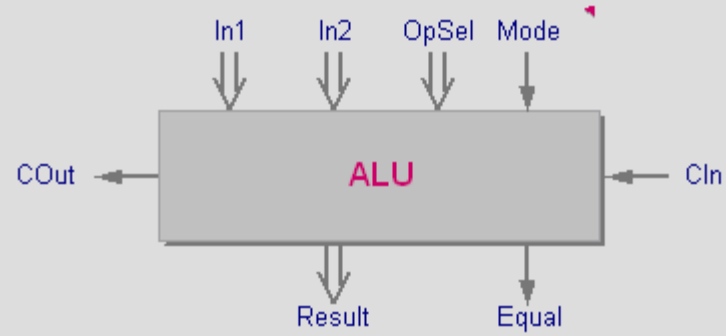



```
module my_module(out1, .., inN);
output out1, .., outM;
input in1, .., inN;

.. // declarations
.. // description of f (maybe
.. // sequential)

endmodule
```

Everything you write in Verilog must be inside a module
exception: compiler directives

The Module Interface

- Port List
- Port Declaration




```
module ALU (Result, COut, Equal, In1, In2,
            OpSel, CIn, Mode);

output [3:0] Result;   // operation result
output       COut;     // carry out
output       Equal;    // when 1, In1 = In2
input  [3:0] In1;      // first operand
input  [3:0] In2;      // second operand
input  [3:0] OpSel;    // operation select
input        CIn;      // carry in
input        Mode;     // mode arithm/logic;
                       // arithm when 0
 . . .

endmodule
```

Hierarchical Verilog Example

- Build up more complex modules using simpler modules
- Example: 4-bit wide mux from four 1-bit muxes
  - Just “drawing” boxes and wires

```
module mux2to1_4(
   input [3:0] A,
   input [3:0] B,
   input Sel,
   output [3:0] O );

   mux2to1 mux0 (Sel, A[0], B[0], O[0]);
   mux2to1 mux1 (Sel, A[1], B[1], O[1]);
   mux2to1 mux2 (Sel, A[2], B[2], O[2]);
   mux2to1 mux3 (Sel, A[3], B[3], O[3]);
endmodule
```

Connections by Name

- Can (should) specify module connections by name
  - Helps keep the bugs away
  - Example

```
mux2to1 mux1 (.A (A[1])
              .B (B[1]),
              .O (O[1]),
              .S (Sel)   );
```

  - Verilog won't complain about the order (but it is still poor practice to mix them up):

Verilog Logic Values

- 0: zero, logic low, false, ground
- 1: one, logic high, power
- X: unknown
- Z: high impedance, unconnected, tri-state

```
!false
```

It’s funny because it’s true

Vectors

- Wire/Reg vectors:

```
wire [7:0] W1;      // 8 bits, w1[7] is MSB
```

  - Also called “buses”

- Operations
  - Bit select: W1[3]
  - Range select: W1[3:2]
  - Concatenate:

```
vec = {x, y, z};
{carry, sum} = vec[0:1];
```

  - e.g., swap high and low-order bytes of 16-bit vector

```
wire [15:0] w1, w2;
assign w2 = {w1[7:0], w1[15:8]}
```

Repeated Signals

- Previously we discussed vector concatenation

```
assign vec = {x, y, z};
```

- Can also repeat a signal n times

```
assign vec = {16{x}}; // 16 copies of x
```

- Example uses (what does this do?):

```
wire [7:0] out;
wire [3:0] A;
assign out = {{4{0}}, A[3:0]};
```

- What about this?

```
assign out = {{4{A[3]}}, A[3:0]};
```

Numbers in Verilog

<width>’<signed?><radix> value (”<>” indicates optional part)

No of **bits**

s,S – Signed (Two’s complement)

Binary → b or B
Octal → o or O
Decimal → d or D
Hexadecimal → h or H

Consecutive chars 0-f, x, z

- Defaults
  - Width = 32
  - Unsigned
  - Radix = Decimal

Numbers in Verilog - Examples

- `14` - ordinary decimal number
- `–14` - 2's complement representation
- `12'b0000_0100_0110` - binary number with 12 bits (_ is ignored)
- `12'h046` - hexadecimal number with 12 bits
- Unless **both** operands are **signed** – results are *unsigned*
  - e.g., `C[4:0] = A[3:0] + B[3:0];`
  - if A = 0110 (6) and B = 1010(-6)
    C = 10000 not 00000
    i.e., B is zero-padded, not sign-extended

Data Types in Verilog

- Nets
  - Nets are physical connections between devices
  - Many types of nets, but all we care about is **`wire`**
  - Cannot be assigned in an *initial* or *always* block
  - **Declaring a net**
    `wire <range> net_name;`
- Registers
  - Implicit storage-holds its value until a new value is assigned to it.
  - Register type is denoted by **`reg`**
  - Hold their value until explicitly assigned in an *initial* or *always* block
  - Is *NOT* necessarily a register in the circuit
  - **Declaring a register**
    `reg <range> reg_name;`

➢ *Parameters* are not variables, they are constants.

Parameters

- Allow per-instantiation module parameters
  - Use “parameter” statement
- modname #(10, 20, 30) instname(in1, out1);
- Example:

```
module mux2to1_N(Sel, A, B, O);
    parameter N = 1
    input [N-1:0] A;
    input [N-1:0] B;
    input Sel;
    output [N-1:0] O;
    // Mux instantiation
endmodule
…
Mux2to1_N #(4) mux1 (S, in1, in2, out)
```

Local Parameters

- Contain constant values
- Can not be assigned during instantiation - Not to be confused with parameters
- Example:

```
module mux2to1_N(Sel, A, B, O);
    localparam N = 2'd2
    input [N-1:0] A;
    input [N-1:0] B;
    input Sel;
    output [N-1:0] O;
    // Mux instantiation
endmodule
```

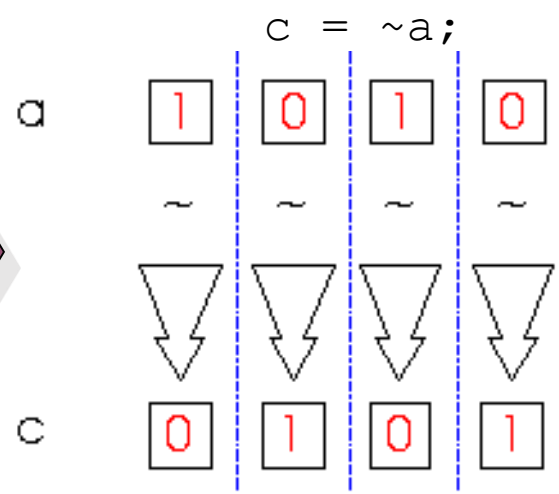
Bitwise Operators (i)

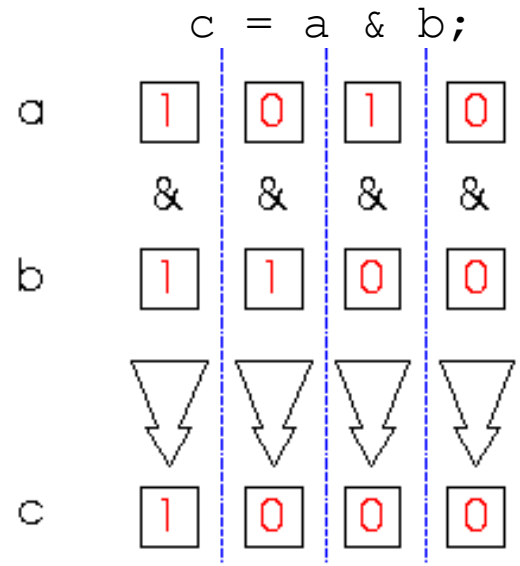
- & → bitwise AND
- | → bitwise OR
- ~ → bitwise NOT
- ^ → bitwise XOR
- ~^ or ^~ → bitwise XNOR

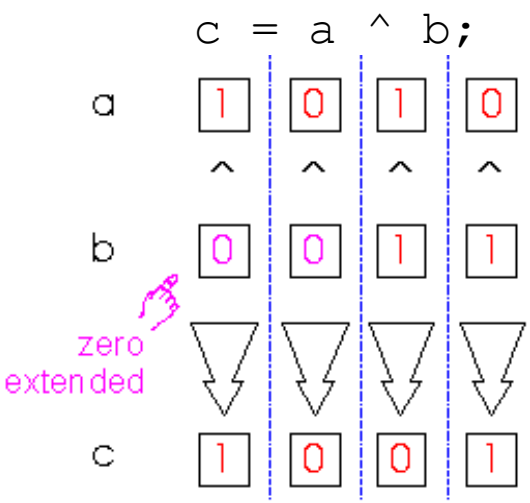
- Operation on bit by bit basis

Bitwise Operators (ii)

```
a = 4'b1010;
b = 4'b1100;
```








```
a = 4'b1010;
b = 2'b11;
```

Logical Operators

- `&&` → logical AND
- `||` → logical OR
- `!` → logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
- Result is ONE bit value: *0*, *1* or *x*

```
A = 1;
B = 0;
C = x;
```

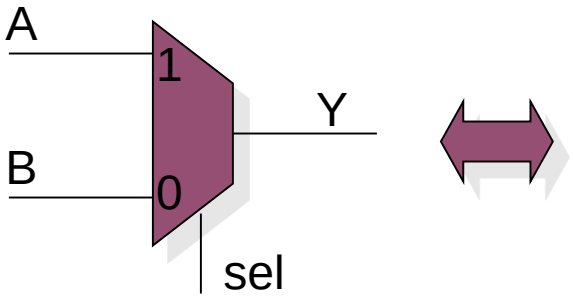
```
A && B → 1 && 0 → 0
A || !B → 1 || 1 → 1
C || B → x || 0 → x
```

but C&&B=0

Shift, Conditional Operators

- >> → shift right
- << → shift left
- `a = 4'b1010;`
  - `b = a >> 2; // b = 0010`
  - `c = a << 1; // c = 0100`

- `cond_expr ? true_expr : false_expr`



`Y = (sel)? A : B;`

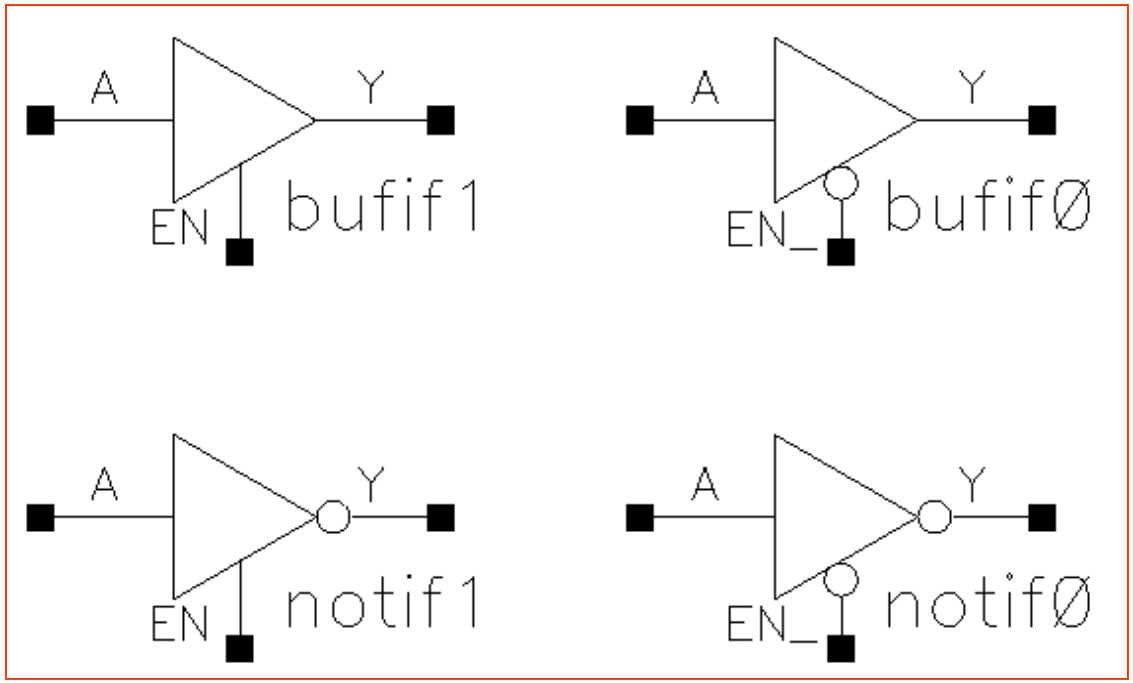
Relational Operators

- `<, <=` → a <[=] b; // is a less than [or equal to] b?
- `>, >=` → a >[=] b; // is a greater than [or equal to] b?
- `==` → c == a ; // is c equal to a?
- `!=` → c != a ; // is c not equal to a?

- All of these return a 1-bit logical value (true/false)

Verilog Primitives

- Basic logic gates available for direct instantiation
  - and
  - or
  - not
  - buf
  - xor
  - nand
  - nor
  - xnor
  - bufif1, bufif0
  - notif1, notif0



- All of these can be implemented by using operators

Verilog Operators

Verilog 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
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

Agenda

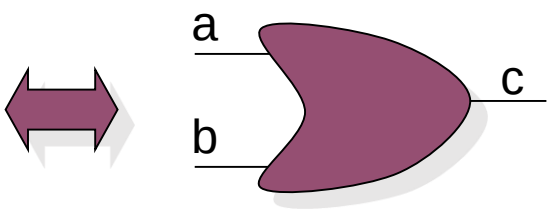
- Hardware Description Languages and coding styles
- Verilog Building Blocks:
  - Keywords
  - Modules
  - Values / Number representation
  - Data Types
  - Operators
- Assignments and procedural blocks
- Test benches and simulation

Continuous Assignments

Wire/Vector assignment are “continuous assignments”

- Connect combinational logic block or other wire to wire input
- When right-hand-side changes, it immediately flows through to left
- **Order of statements not important to Verilog**, executed totally in parallel
- Designated by the keyword **assign**

```
wire c;
assign c = a | b;
wire c = a | b; //same thing
```



- Common errors:
  - Not assigning a wire a value
  - Assigning a wire a value more than once

Procedural Blocks

- Two Procedural Constructs
  - initial Statement - Executes only once
  - always Statement - Executes in a loop
- Sequential: All statements within the block are executed sequentially
- Assign variables using procedural assignments
- Example

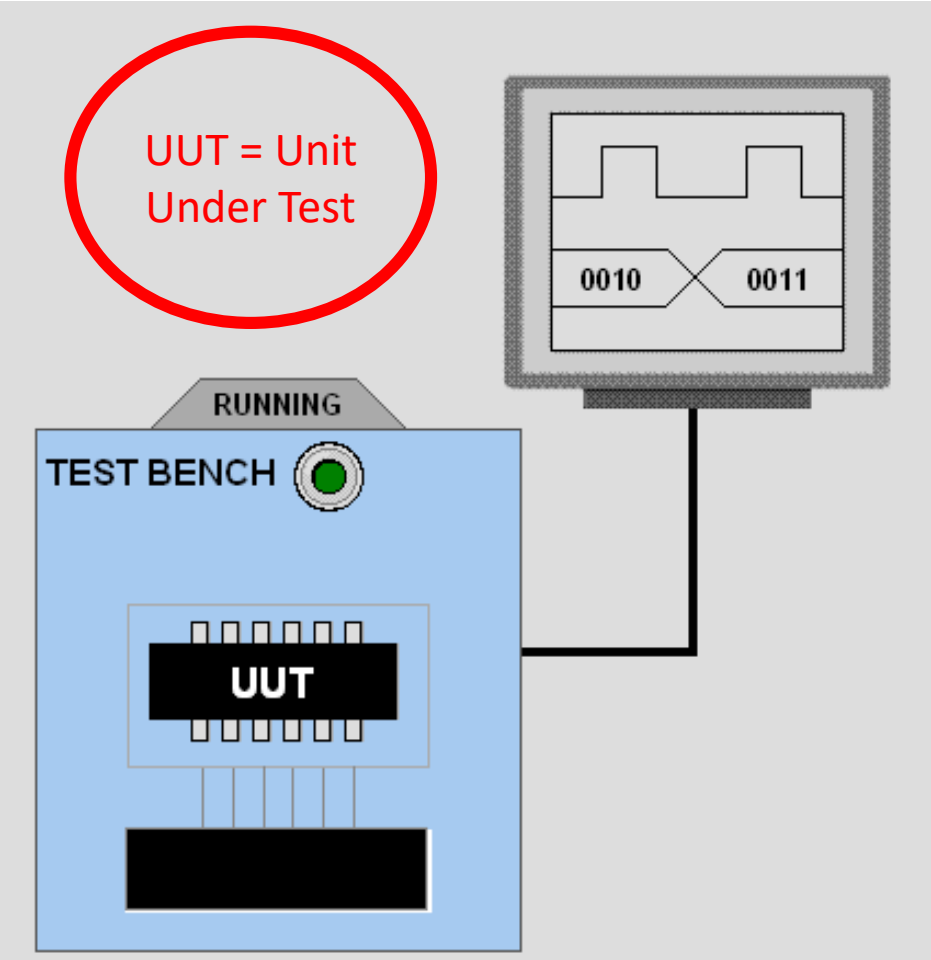
```
...
initial begin
  Sum = 0;
  Carry = 0;
end
...
```

```
...
always ... begin
  Sum = A ^ B;
  Carry = A & B;
end
...
```

Agenda

- Hardware Description Languages and coding styles
- Verilog Building Blocks:
  - Keywords
  - Modules
  - Values / Number representation
  - Data Types
  - Operators
- Assignments and procedural blocks
- Test benches and simulation

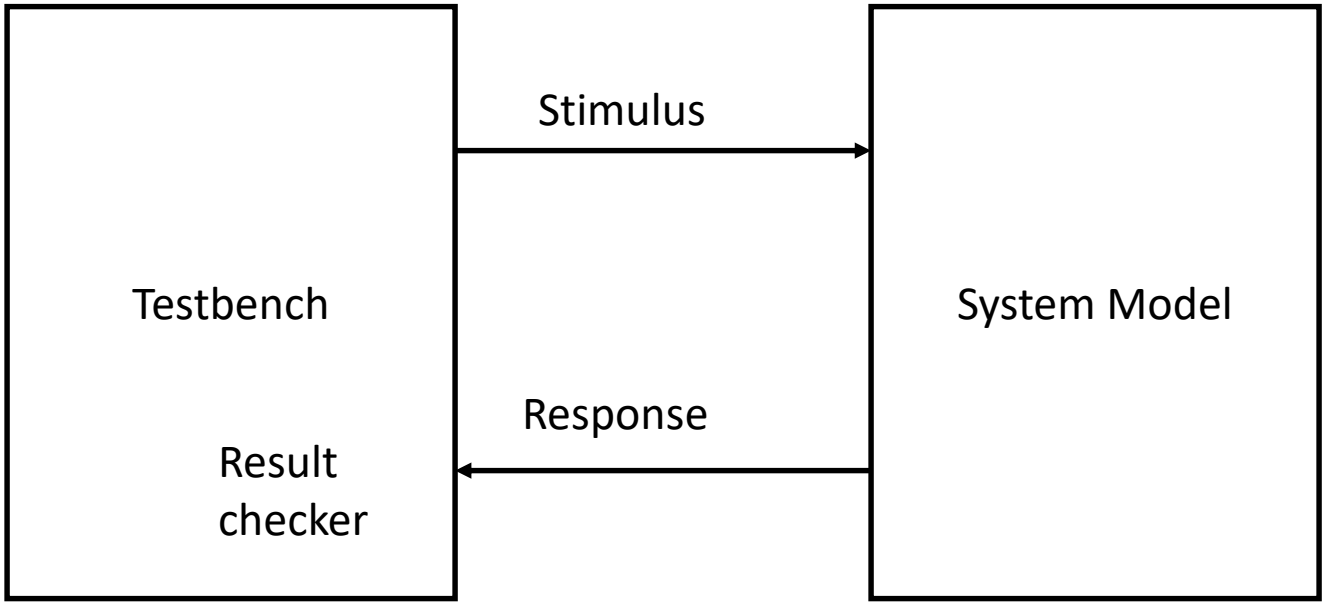
Test Bench




```
module main;
reg  a, b, c;
wire sum, carry;

fulladder add(a,b,c,sum,carry);
initial
begin
  a = 0; b = 0; c = 0;
  #5
  a = 0; b = 1; c = 0;
  #5
  a = 1; b = 0; c = 1;
  #5
  a = 1; b = 1; c = 1;
  #5
end
endmodule
```

How Are Simulators Used?

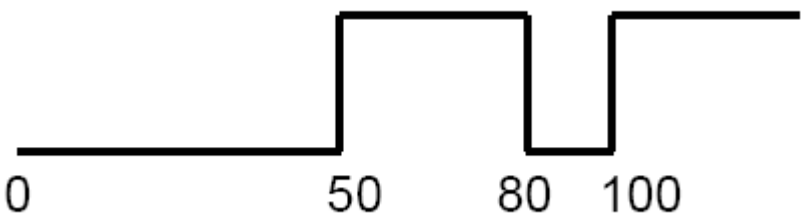
- Test bench **generates stimuli** and checks response
- Coupled to model of the system
- Pair is run simultaneously



Test Bench - Generating Stimuli

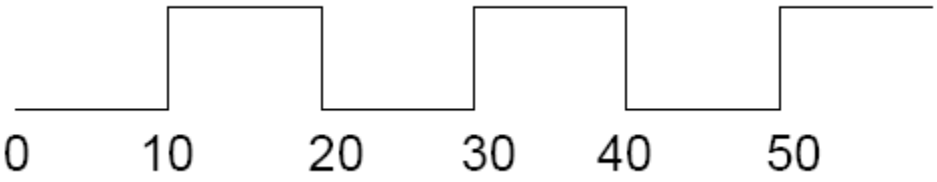
- Data: A sequence of values

```
initial begin
  Clock = 0;
  #50 Clock = 1;
  #30 Clock = 0;
  #20 Clock = 1;
End
```



- Repetitive Signals (clock)

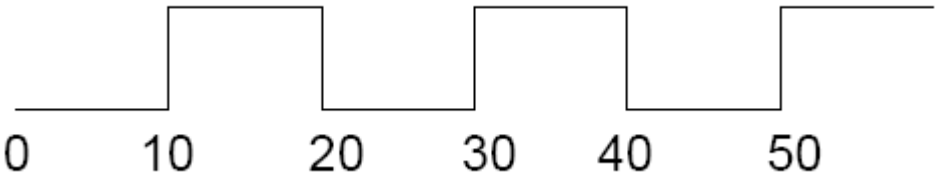
```
initial begin
  Clock = 0;
end
…
always begin
#10 Clock = ~ Clock;
end
```



Test Bench - Generating Clock

- Repetitive Signals (clock)

Clock



- A Simple Solution:

```
wire Clock;
assign #10 Clock = ~ Clock
```

- Caution:
  - Initial value of Clock (***wire*** data type) = z
  - ~z = x and ~x = x

Test Bench - Generating Clock (cont.)

- Initialize the Clock signal

```
initial begin
   Clock = 0;
end
```

- Caution: Clock is of data type ***wire***, cannot be used in an ***initial*** statement
- Solution:

```
reg Clock;
…
initial begin
  Clock = 0;
end
…
always begin
#10 Clock = ~ Clock;
end
```

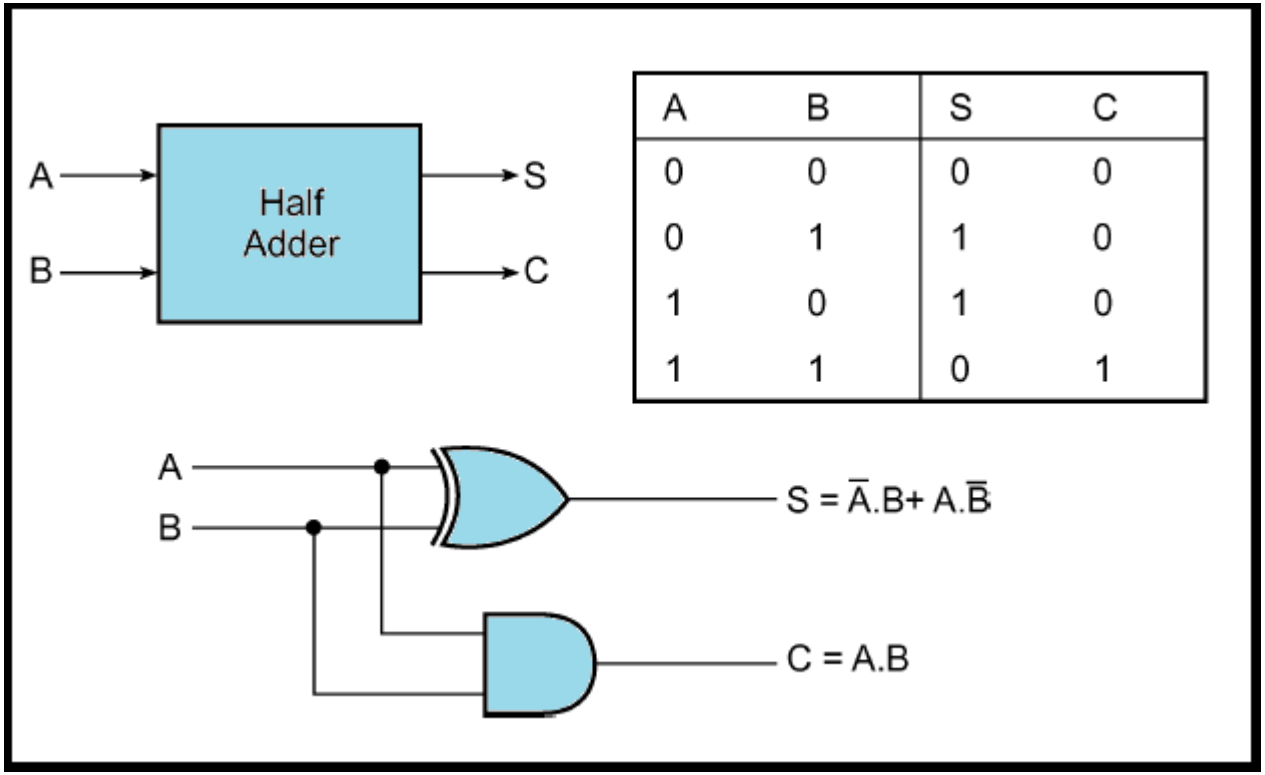
forever loop can also be used to generate clock

System Tasks

- Display tasks
  - $display : Displays the entire list at the time when statement is encountered
  - $monitor : Whenever there is a change in any argument, displays the entire list at end of time step
- Simulation Control Task
  - $finish : makes the simulator exit
  - $stop : suspends the simulation
- Time
  - $time: gives the simulation time

“Hands on” Example

Full adder implementation using two half adders



A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

“Hands on” Example

Full adder implementation using two half adders