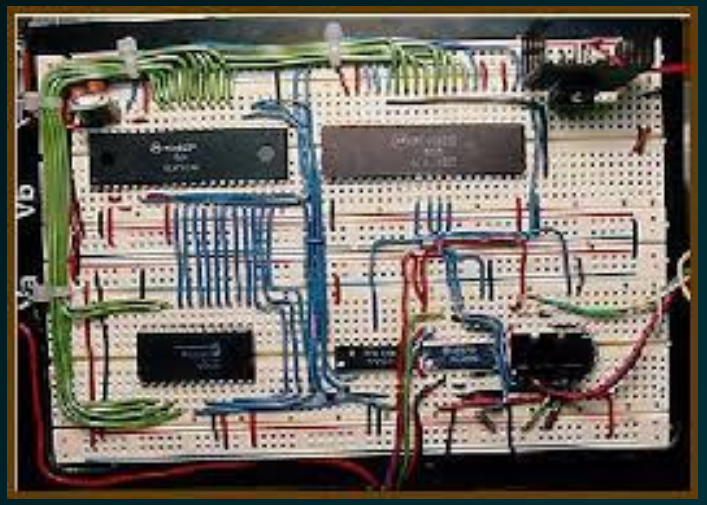
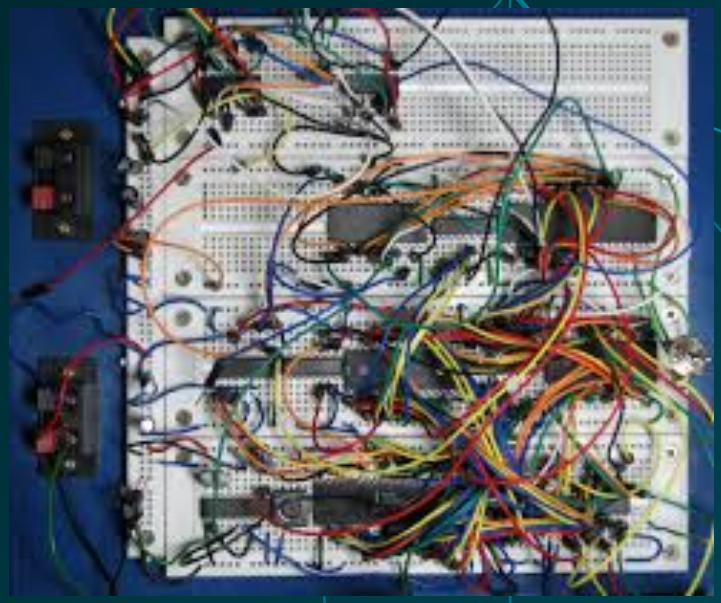
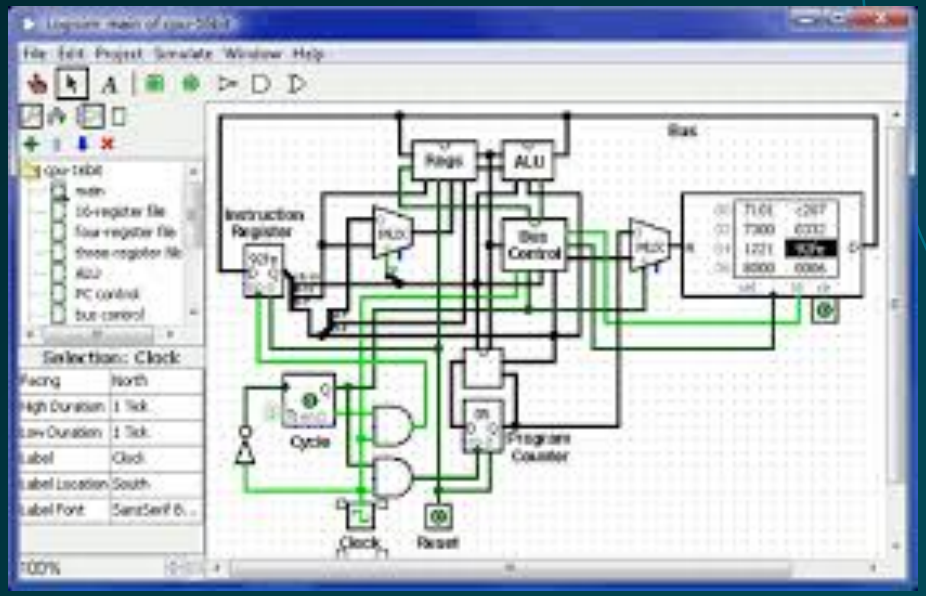
CS225/CS226

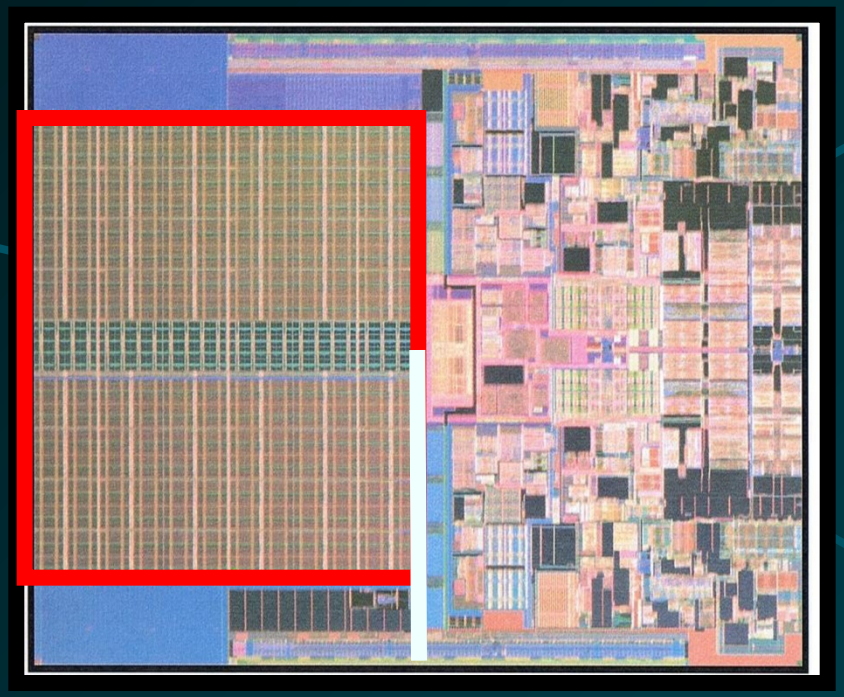
So far



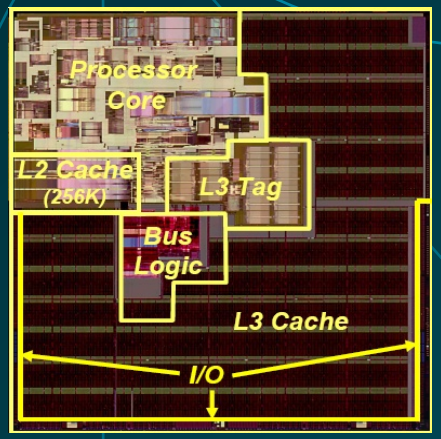



Bus
Regs
ALU
Instruction Register
Bus Control
Cycle
Program Counter
Clock
Reset

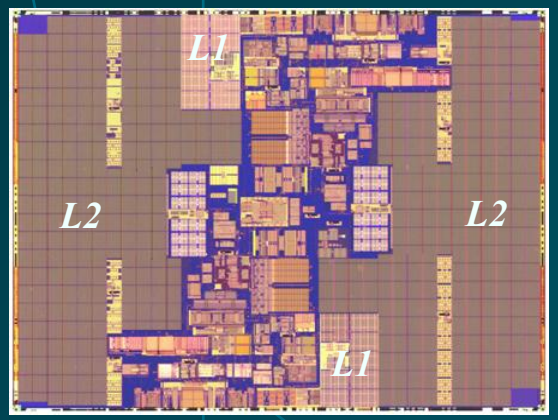
Modern System



Penryn (dual-core)
45nm Technology



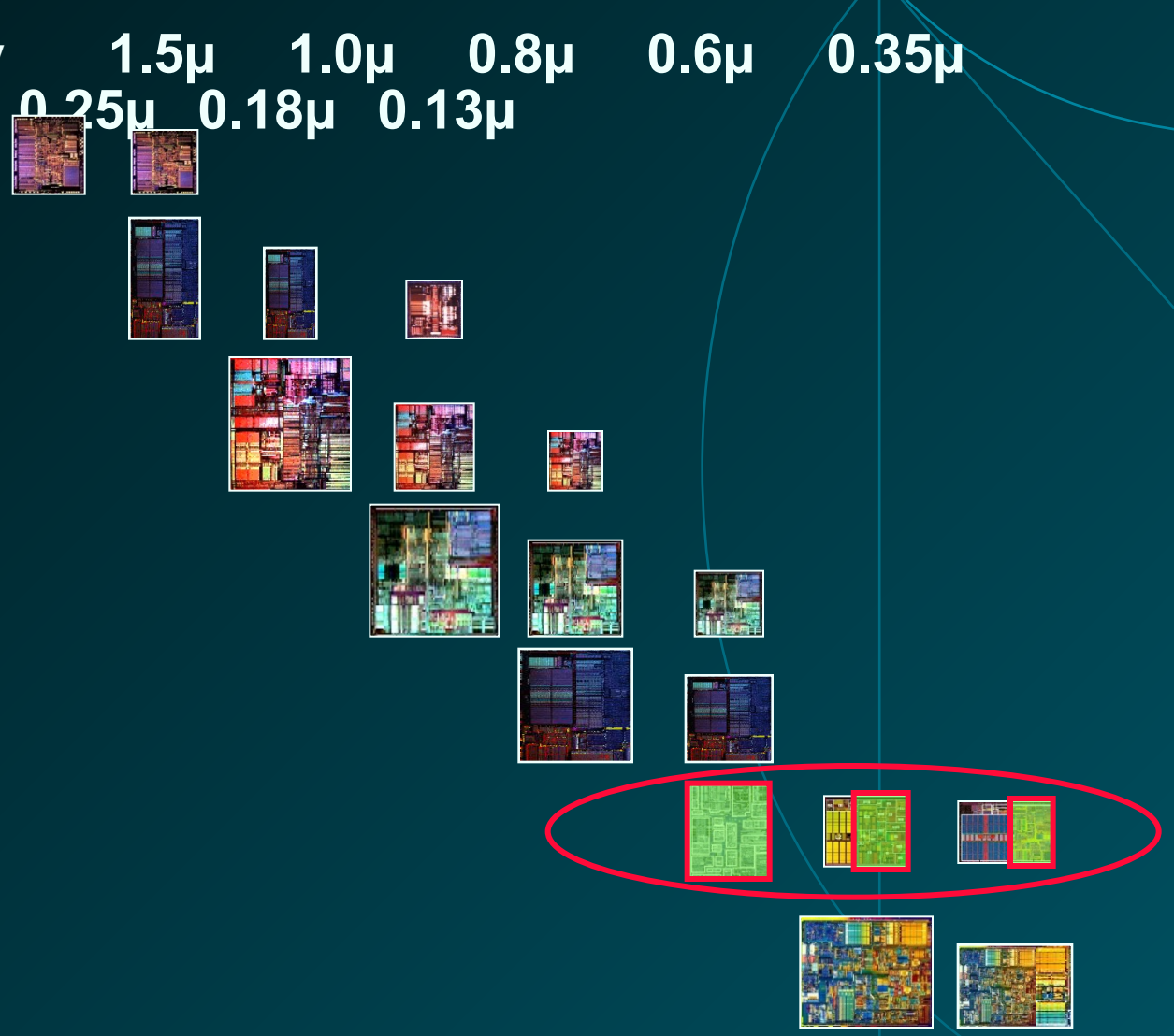
Itanium 2* (L3-9MB)
130nm Technology



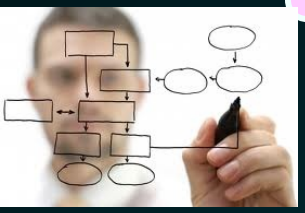
Montecito* (dual-core)
(L2-24MB, L1-2MB)
90nm Technology

V. George *et al.*, "Penryn: 45-nm Next Generation Intel® Core™ 2 Processor," *Proceedings of the IEEE Solid-State Circuits Conference*, *Intel

Different versions



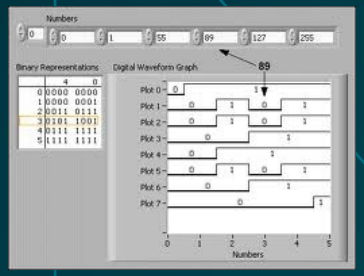
Basic Design Methodology



Requirements

RTL Model

Simulate



Synthesize

Gate-level Model

Simulate

Test Bench

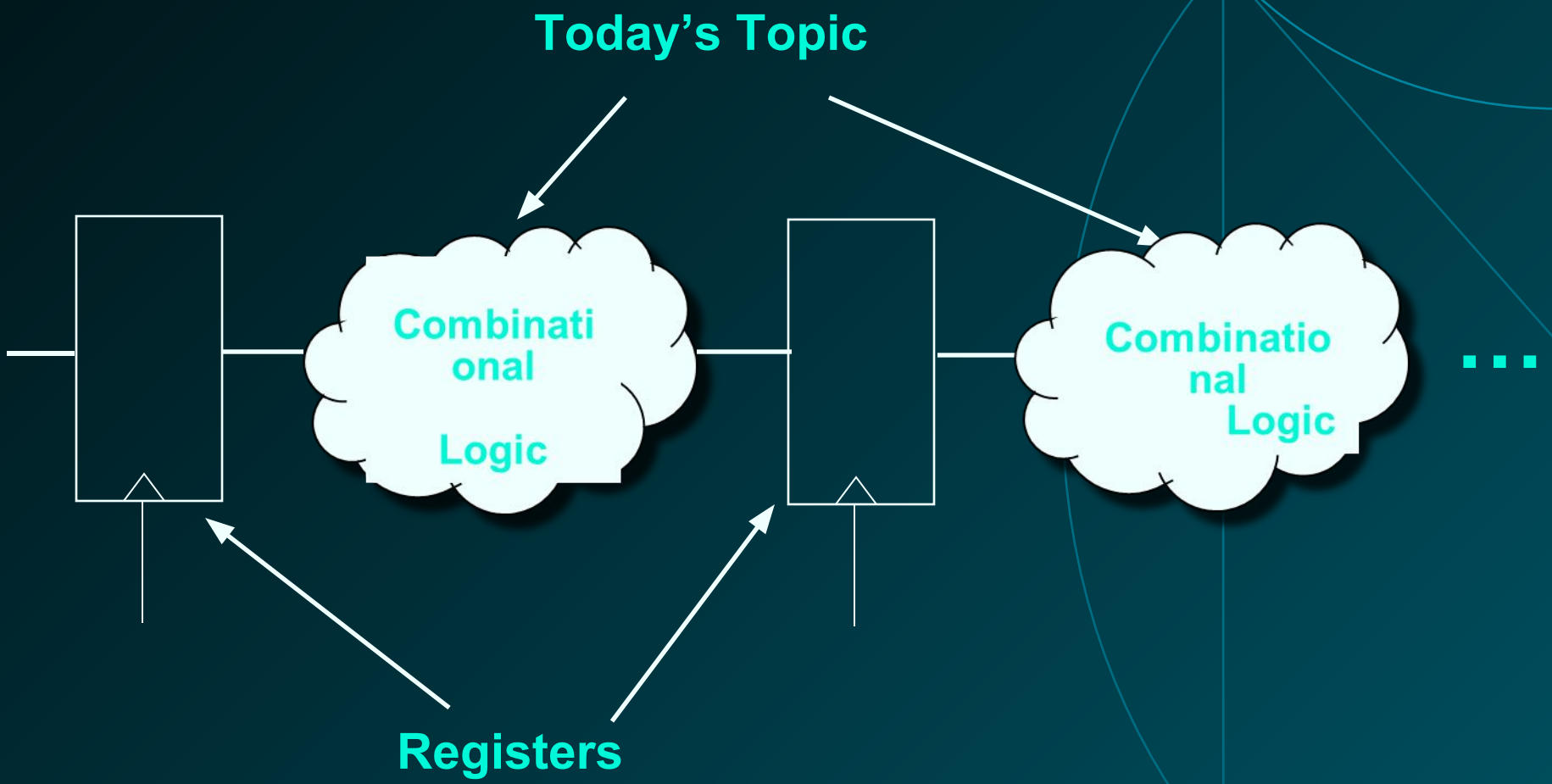
ASIC or FPGA

Place & Route

Timing Model

Simulate

Register Transfer Level (RTL) Design Description

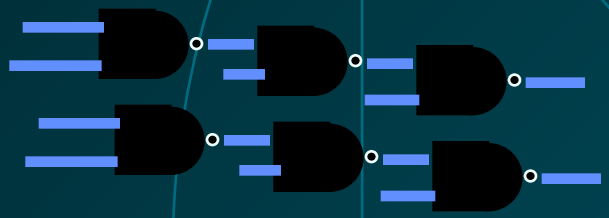


Basic Design Methodology

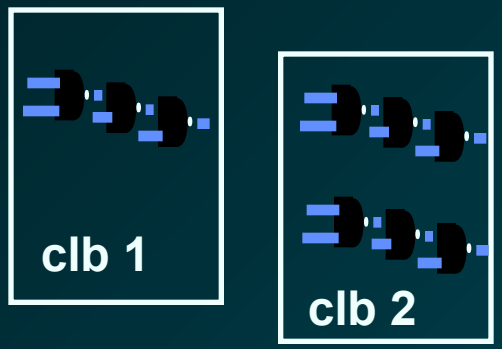
Verilog

```
Always
    inst1
    inst2
    inst3
```

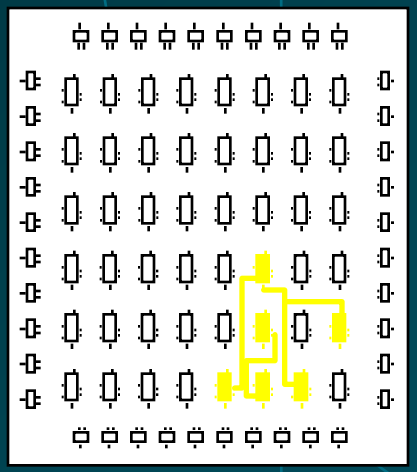
Synthesis



Mapping



Place and Route



What is Verilog HDL?

- Verilog Hardware Description Language(HDL)?
  - A high-level computer language can model, represent and simulate digital design
    - Hardware concurrency
    - Parallel Activity Flow
    - Semantics for Signal Value and Time
  - Design examples using Verilog HDL
    - Intel Pentium, AMD K5, K6, Atheon, ARM7, etc
    - Thousands of ASIC designs using Verilog HDL

VERILOG HDL

- **Basic Unit – A module**
- **Module**
  - **Describes the functionality of the design**
  - **States the input and output ports**
- **Example:**


```
module and_gate(out, a,b);
output out ;
input a, b;
assign out = a & b;
endmodule
```

Testing AND gate

- **Instantiate a module**
  - **apply test**

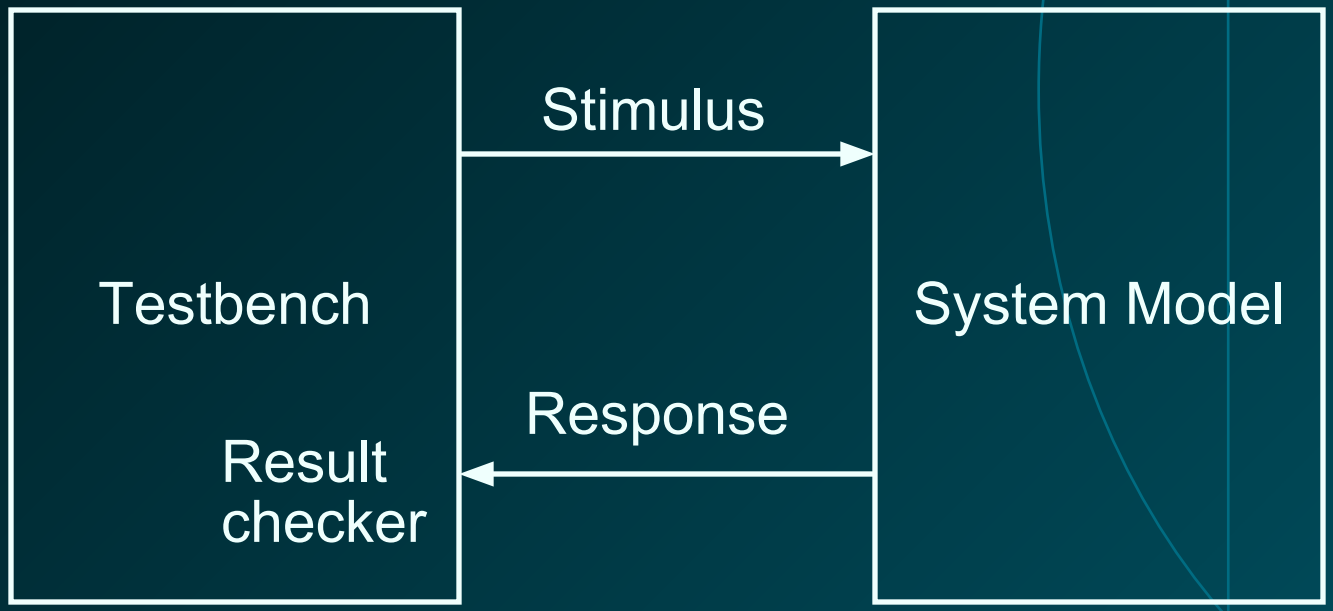
```
module tb_and_gate();
reg a,b;
wire out;
and_gate  uut (out,a,b); // this instantiates a and gate, uut is a label
initial
begin
a = 1'b0;                // here we apply inputs to the gate
b = 1'b0;
#10;
a = 1'b0;
b = 1'b1;
#10;
a = 1'b1;
b = 1'b1;
#10;
a = 1'b1;
b = 1'b0;
#10;
End
endmodule
```

Different Levels of Abstraction

- Algorithmic
  - the function of the system
- RTL
  - the data flow
  - the control signals
  - the storage element and clock
- Gate
  - gate-level net-list
- Switch
  - transistor-level net-list

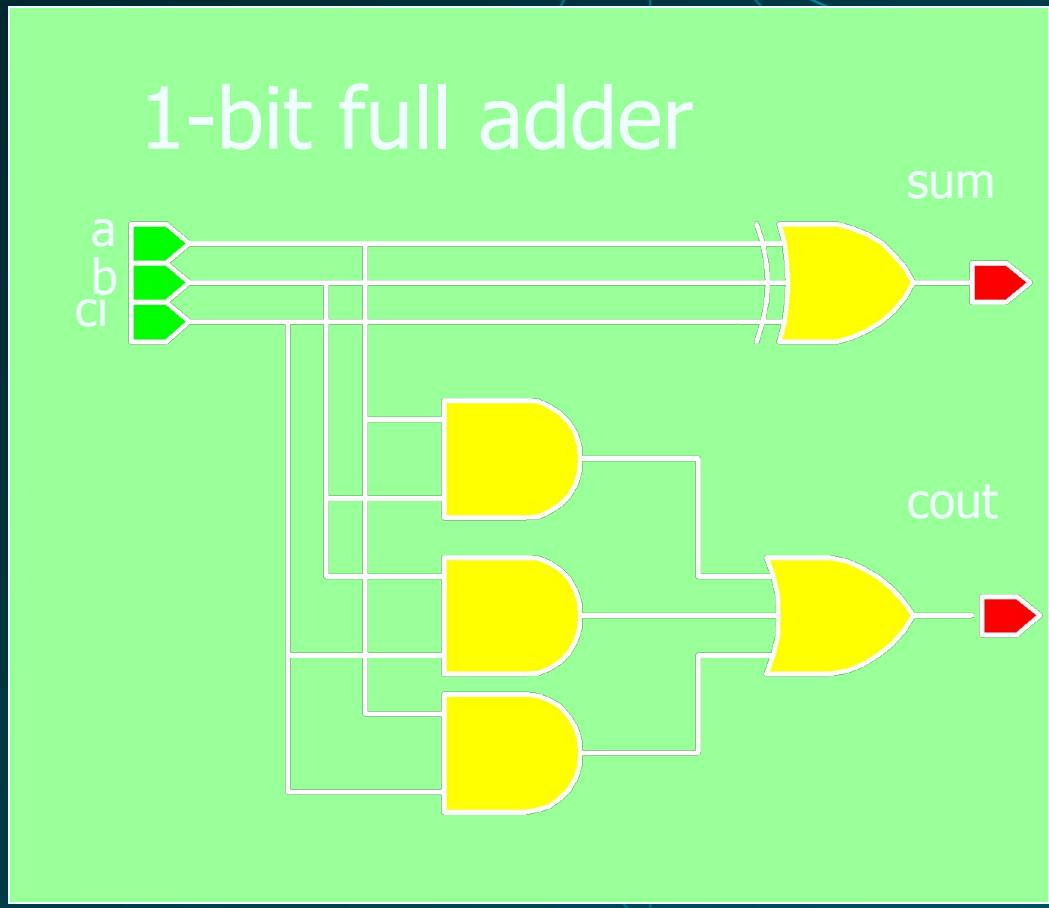
How Are Simulators Used?

- Testbench generates stimulus and checks response
- Coupled to model of the system
- Pair is run simultaneously



Sample Design

```
module fadder ( sum, cout, a, b , ci );
// port declaration
output sum, cout;
input   a, b, ci;
reg     sum, cout;
// behavior description
always @( a or b or ci )
begin
    sum = a ^ b ^ ci;
    cout = ( a&b ) | ( b&ci ) | ( ci&a);
end
endmodule
```



Writing Testbenches

```
module test;
reg a, b, sel;

mux m(y, a, b, sel);

initial begin
  $monitor($time,, "a = %b b=%b sel=%b y=%b",
                   a, b, sel, y);
  a = 0; b= 0; sel = 0;
  #10 a = 1;
  #10 sel = 1;
  #10 b = 1;
end
```

Inputs to device under test

Device under test

$monitor is a built-in event driven "printf"

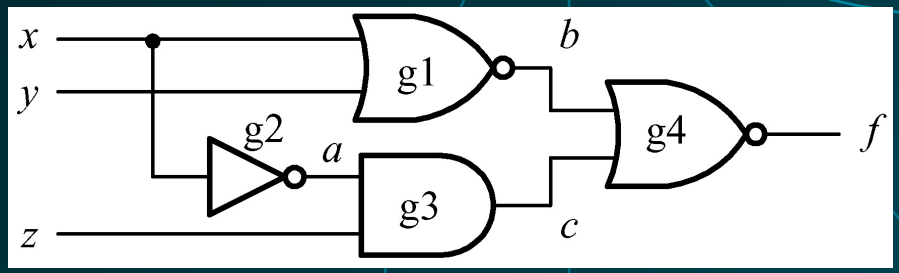
Stimulus generated by sequence of assignments and delays

Structural Modeling

- When Verilog was first developed (1984) most logic simulators operated on netlists
- Netlist: list of gates and how they're connected
- A natural representation of a digital logic circuit

- Not the most convenient way to express test benches

Module basic_gates




```
module basic_gates (x, y, z, f) ;
input   x, y, z;
output f ;
wire a, b, c;
// Structural modeling
    nor  g1  (b, x, y);
    not  g2  (a, x);
    and g3  (c, a, z);
    nor g4  (f, b, c);
endmodule
```

Behavioral Modeling

- A much easier way to write testbenches
- Also good for more abstract models of circuits
  - Easier to write
  - Simulates faster
- More flexible
- Provides sequencing

- Verilog succeeded in part because it allowed both the model and the testbench to be described together

Two Main Components of Verilog

- Concurrent, event-triggered processes (behavioral)
  - *Initial* and *Always* blocks
  - can perform standard data manipulation tasks (assignment, if-then, case)
  - Processes run until they delay for a period of time or wait for a triggering event
- Structure (Plumbing)
  - Verilog program build from modules with I/O interfaces
  - Modules may contain instances of other modules
  - Modules contain local signals, etc.
  - Module configuration is static and all run concurrently

Two Main Data Types

- Nets represent connections between things
  - Do not hold their value
  - Take their value from a driver such as a gate or other module
  - Cannot be assigned in an *initial* or *always* block
- Regs represent data storage
  - Behave exactly like memory in a computer
  - Hold their value until explicitly assigned in an *initial* or *always* block
  - Can be used to model latches, flip-flops, etc., but do not correspond exactly

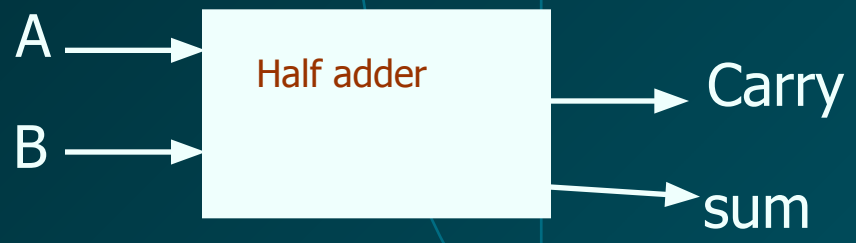
Description Styles

- **General definition**

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

- Example

```
module HalfAdder (A, B, Sum Carry);
input A, B;
output Sum, Carry;
assign Sum = A ^ B;
//^ denotes XOR
assign Carry = A & B;
// & denotes AND
endmodule
```



Description Styles (cont.)

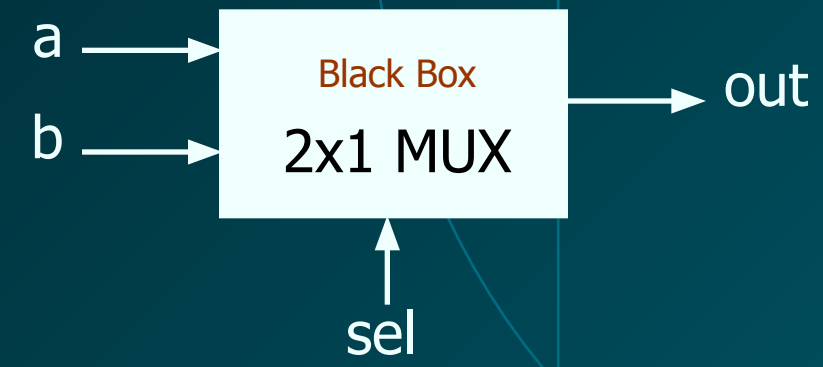
```
module mux1( select, d, q );
input[1:0] select;
input[3:0] d;
output     q;
wire       q;
wire[1:0] select;
wire[3:0] d;

assign q = d[select];
endmodule
```

Description Styles (cont.)

- Behavioral: Algorithmically specify the behavior of the design

- Example:

```
always @( select or a or b ) begin
    if (select == 0) begin
        out = b;
    end
    else if (select == 1) begin
        out = a;
    end
end
```



Description Styles (cont.)

```
module mux1( select, d, q );
input[1:0] select;
input[3:0] d;
output     q;
wire       q;
wire[1:0] select;
wire[3:0] d;

assign q = d[select];
endmodule
```

Description Styles (cont.)

```
module mux2( select, d, q );
input[1:0] select;
input[3:0] d;
output     q;
reg        q;
wire[1:0]  select;
wire[3:0]  d;

always @(d or select)
    q = d[select];
endmodule
```

Description Styles (cont.)

```
module mux3( select, d, q );
input[1:0] select;
input[3:0] d;
output q; reg q;
wire[1:0] select;
wire[3:0] d;
always @( select or d ) begin
   if( select == 0)
      q = d[0];
   if( select == 1)
      q = d[1];
   if( select == 2)
      q = d[2];
   if( select == 3)
      q = d[3];
end  endmodule
```

Description Styles (cont.)

```
module mux4( select, d, q );
input[1:0] select;
input[3:0] d;
output     q;
reg        q;
always @( select or d )
begin
   case( select )
      0 : q = d[0];
      1 : q = d[1];
      2 : q = d[2];
      3 : q = d[3];
   endcase
end
endmodule
```

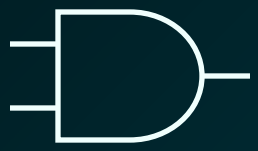
Four-valued Data

- Verilog's nets and registers hold four-valued data

- 0, 1
  - Obvious
- Z
  - Output of an undriven tri-state driver
- X
  - Models when the simulator can't decide the value
  - Initial state of registers
  - When a wire is being driven to 0 and 1 simultaneously

Four-valued Logic

- **Logical operators work on three-valued logic**



	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

Output 0 if one input is 0

Output X if both inputs are gibberish

Nets and Registers

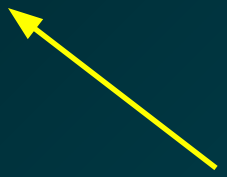
- Wires and registers can be bits, vectors, and arrays
- wire b[3:0];

```
wire a;                 // Simple wire
reg [5:0] vec;          // Six-bit register
integer imem[0:1023]; // Array of 1024 integers
reg [31:0] dcache[0:63];   // A 32-bit memory
reg [31:0] regfile [0:63];    // 64 registers each 32 bit
```

Modules and Instances

- Basic structure of a Verilog module:

```
module mymod(output1, output2, … input1, input2);
output output1;
output [3:0] output2;
input input1;
input [2:0] input2;
…
endmodule
```



Verilog convention lists outputs first

Instantiating a Module

- Instances of

```
module mymod(y, a, b);
```

- look like

```
mymod mm1(y1, a1, b1);      // Connect-by-position
mymod (y2, a1, b1),
          (y3, a2, b2);          // Instance names omitted
mymod mm2(.a(a2), .b(b2), .y(y2));  // Connect-by-name
```

Gate-level Primitives

- Verilog provides the following:

and	nand	logical AND/NAND
or	nor	logical OR/NOR
xor	xnor	logical XOR/XNOR
buf	not	buffer/inverter

Initial and Always Blocks

- **Basic components for behavioral modeling**

```
initial
  begin
statements …
  end
```

Runs when simulation starts

Terminates when control reaches the end

Good for providing stimulus

initial	c
c	statement
c	…
c	…
c	…
c	…
c	…

```
always
  begin
statements …
  end
```

Runs when simulation starts

Restarts when control reaches the end

Good for modeling/specifying hardware

always	c
c	statement
c	…
c	…
c	…
c	…
c	…

Initial and Always

- Run until they encounter a delay

```
initial begin
  #10 a = 1; b = 0;
  #10 a = 0; b = 1;
end
```

- or a wait for an event

```
always @(posedge clk)
 q = d;
always
begin wait(i); a = 0;
 wait(~i); a = 1;
end
```

Binary Literals

- Valid digits:0,1,z,Z,?,x,X

(one bit per digit)

- Zero-filled to 32 bits
  - Filled with z or x if most significantt bit
- Syntax ‘b<digit> or ‘B<digit>
- Example

‘b0 0000 0000 … 0000

‘b1 0000 0000 … 0001

‘B1 0000 0000 … 0001

‘bx xxxx xxxx … xxxx

‘b10x 0000 0000 … 010x

Decimal Literals

- Valid digits:0, 1, ..9

(No X or Z)

- No bitwise correspondence
- Zero filled to 32 bits
- Syntax ‘d<digit> or ‘D<digit>
- Example

‘d0 0000 0000 … 0000

‘d1 0000 0000 … 0001

‘D1 1 0000 0000 … 1011

Hexadecimal Literals

- Valid digits:0-9, a-f, A-F, z, Z, ?, x, X
- Four bits per digit
- Zero filled to 32 bits
- Syntax ‘h<digit> or ‘ H<digit>
- Example

‘h0 0000 …00000000

‘h1 0000 …00000001

‘h1 1 0000 …00010001

‘hx xxxx …xxxxxxxx

‘h17xz 000 … 0010111xxxxzzzz

Size of Literals

- By default 32 bit:
- Example

‘b0	0000 0000 … 0000
‘b1	0000 0000 … 0001
‘h1	0000 0000 … 0001
‘h0F	0000 0000 … 1111
‘d1	0000 0000 … 0001
‘d14	0000 0000 … 1110

Sized Literals

- To specify number of bits different from 32
- For any literal base
- Extra bits are truncated
- example

1’b0	1
‘b1	0000 0000 … 0001
8‘b1	0000 0001
8‘h0F	00001111
8‘d256	0000 0000
5‘d14	01110

Integer/Real Literals

- Valid digits: 0-9
- 2's compliment
- example

1 0000000 … 0000000001

-1 11111111…11111111111

35 0000000 … 0000100011

Real : Double precision (64 bits) floating point

Syntax: [-] <digit>.<digit> E[-] <digit>

example: 0.0

1.0

1.0E-9

Logical Values

- Any non-zero value is TRUE
- Anything else if FALSE
  - Including ‘x’ and ‘z’
- examples

1’b0	FALSE
1’b1	TRUE
1’bX	FALSE
3’b000	FALSE
3’b10X	TRUE
3’b00X	FALSE

Operators

Arithmetic	Bitwise	Reduction	Relational
+: add - : subtract * : multiply / : divide % : modulus **: exponent	~ : NOT &: AND | : OR ^: XOR ~^, ^~: XNOR	&: AND |: OR ~&: NAND ~|: NOR ^: XOR ~^, ^~: XNOR	>: greater than <: less than >= : greater than or equal <=: less than or equal
Shift	**case equality**	**Logical**	**Miscellaneous**
<< : left shift >> : right shift <<< : arithmetic left shift >>>: arithmetic right shift	===: equality !==: inequality	&&: AND || : OR ! : NOT	{ , }: concatenation {c{ }}: replication ? : conditional
	Equality		
	==: equality !=: inequality		

Bitwise Operators

- **~ <expr>** *bitwise not*
- **<expr> & <expr>** *bitwise and*
- **<expr> | <expr>** *bitwise or*
- **<expr> ^ <expr>** *bitwise xor*

- **examples**

~3'b101 3'b010

3'b101 & 3'b111 3'b101

3'b101 | 3'b111 3'b111

3'b101 ^ 3'b111 3'b010

Reduction Operators

- & <expr> *and*- reduction
- | <expr> *or*- reduction
- ^ <expr> *xor*- reduction
- examples

& 3'b011	1b0
& 3'b111	1b1
& 3'b11x	1bx
& 3'b01x	1b0

| 3'b011	1b1
| 3'b000	1b0
| 3'b11x	1b1
| 3'b00x	1bx

^ 3'b001	1b1
^ 3'b111	1b1
^ 3'b11x	1bx

Vector Operators

- **<expr> << <expr>** *Logic shift left*
- **<expr> >> <expr>** *Logic shift right*
- **{<expr> , <expr>}** *Concatenation*
- **{<cont-expr>{expr}}** *Replication*

- **examples**

3'b101 << 1	3'b010
3'b101 >> 2	3'b001
{3'b111, 1'b0}	4'b1110
{3 {2'b01}}	6'b010101

Relational Operators

- **<expr> == <expr>** **equal**
- **<expr> !> <expr>** **not equal**
- **<expr> === <expr>** ***identical***
- **<expr> !== <expr>** ***not identical***
-

- **examples**

3'b101 == 3'b010 '1b0 (False)

3'b101 != 3'b010 '1b1 (True)

3'b0x1 =≈ 3'b0x1 '1b0 (False)

3'b0x1 ===≈ 3'b0x1 '1b1 (True)

Relational Operators

- **<expr> < <expr>** *less than*
- **<expr> <= <expr>** *less than or equal*
- **<expr> > <expr>** *greater than*
- **<expr> !== <expr>** *greater than or equal*

- **examples**

3'b101 < 3'b010 '1b0

3'b111 > 3'b010 '1b1

-1 < 1 '1b1

Logical Operators

- ! <expr> logical *not*
- <expr> && <expr> logical *and*
- <expr> || <expr> logical *or*

- examples

! 3'b001 '1b0 (False)

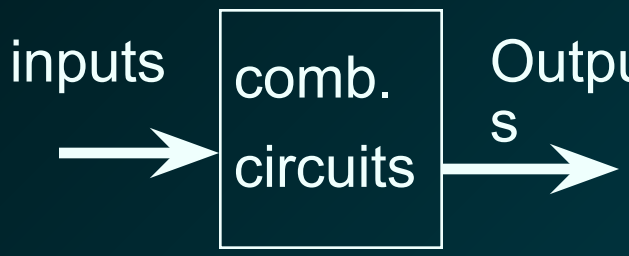
3'b001 && 3'b100 '1b1 (True)

3'b001 & 3'b100 '3b000 (False)

3'b001 || 3'b100 '1b1 (True)

Combinational Circuit Design

- Outputs are functions of inputs

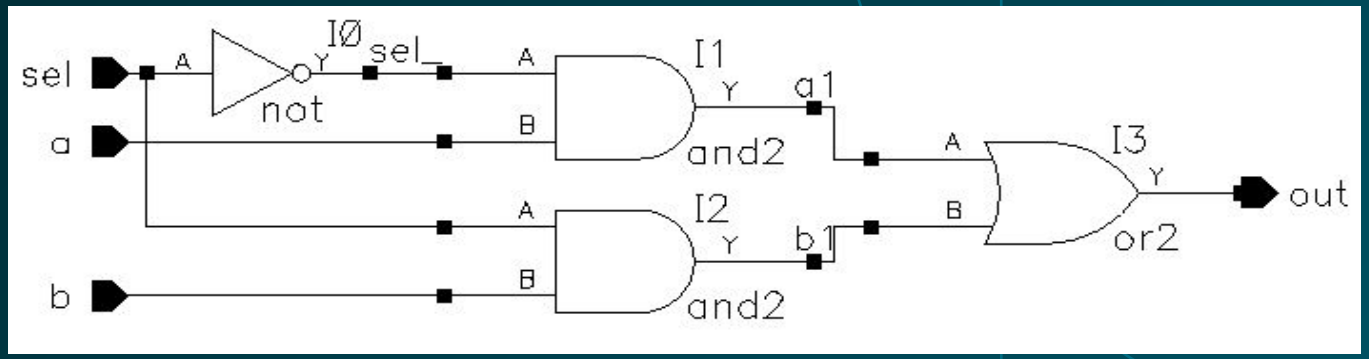


- Examples
  - MUX
  - decoder
  - priority encoder
  - adder

Multiplexor

- **Net-list (gate-level)**

```
module mux2_1 (out,a,b,sel) ;
    output out ;
    input a,b,sel ;
    not (sel_, sel) ;
    and (a1, a, sel_) ;
    and (b1, b, sel) ;
    or (out, a1, b1) ;
endmodule
```



Multiplexor

- **Continuous assignment**

```
module mux2_1 (out,a,b,sel) ;
    output out ;
    input a,b,sel ;
    assign out = (a&~sel)|(b&sel) ;
endmodule
```

- **RTL modeling**

```
always @(a or b or sel)
if(sel)
        out = b;
else
        out = a;
```

Multiplexor

- 4-to-1 multiplexor

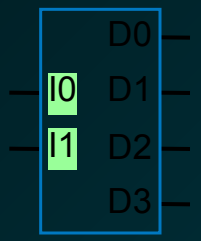
```
module mux4_1 (out, in0, in1, in2, in3, sel) ;
  output out ;
  input in0,in1,in2,in3 ;
  input [1:0] sel ;
    assign out = (sel == 2'b00) ? in0 :
                 (sel == 2'b01) ? in1 :
                 (sel == 2'b10) ? in2 :
                 (sel == 2'b11) ? in3 :
                   1'bx ;
endmodule
```

Multiplexor

```
module mux4_1 (out, in, sel) ;
  output out ;
  input [3:0] in ;
  input [1:0] sel ;
  reg out ;
    always @(sel or in) begin
        case(sel)
            2'd0: out = in[0] ;
               2'd1: out = in[1] ;
            2'd2: out = in[2] ;
            2'd3: out = in[3] ;
            default: 1'bx ;
        endcase
      end
endmodule
```

Top-Down Design – Combinational Behavior to Structure
Procedures with If-Else Statements

- **Q: Create procedure describing behavior of a 2x4 decoder using if-else-if construct**



2x4 decoder

Order of assignment statements does not matter.

Placing two statements on one line does not matter.

To execute multiple statements if expression is true, enclose them between "begin" and "end"

```
`timescale 1 ns/1 ns

module Dcd2x4(I1, I0, D3, D2, D1, D0);

   input I1, I0;
   output D3, D2, D1, D0;


    reg D3, D2, D1, D0;

    always @(I1, I0)
    begin
       if (I1==0 && I0==0)
       begin
          D3 <= 0; D2 <= 0;
          D1 <= 0; D0 <= 1;
       end
       else if (I1==0 && I0==1)
       begin
          D3 <= 0; D2 <= 0;
          D1 <= 1; D0 <= 0;
       end
       else if (I1==1 && I0==0)
       begin
          D3 <= 0; D2 <= 1;
          D1 <= 0; D0 <= 0;
       end
       else
       begin
          D3 <= 1; D2 <= 0;
          D1 <= 0; D0 <= 0;
       end
    end
endmodule
```

Decoder

- 3-to 8 decoder with an enable control

```
module decoder(o,enb_,sel) ;
output [7:0] o ;
input enb_ ;
input [2:0] sel ;
reg [7:0] o ;
always @ (enb_ or sel)
  if(enb_)
    o = 8'b1111_1111 ;
  else
    case(sel)
    3'b000 : o = 8'b1111_1110 ;
    3'b001 : o = 8'b1111_1101 ;
    3'b010 : o = 8'b1111_1011 ;
    3'b011 : o = 8'b1111_0111 ;
    3'b100 : o = 8'b1110_1111 ;
    3'b101 : o = 8'b1101_1111 ;
    3'b110 : o = 8'b1011_1111 ;
    3'b111 : o = 8'b0111_1111 ;
    default : o = 8'bx ;
    endcase
endmodule
```

Priority Encoder

```
always @ (d0 or d1 or d2 or d3)
    if (d3 == 1)
            {x,y,v} = 3'b111 ;
    else if (d2 == 1)
            {x,y,v} = 3'b101 ;
    else if (d1 == 1)
            {x,y,v} = 3'b011 ;
    else if (d0 == 1)
            {x,y,v} = 3'b001 ;
    else
            {x,y,v} = 3'bxx0 ;
```

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Parity Checker

```
module parity_chk(data,
    parity);
  input [0:7] data;
  output parity;
  reg parity;
```

```
always @ (data)
    begin: check_parity
      reg partial;
      integer n;
      partial = data[0];
      for ( n = 0; n <= 7; n = n + 1)
       begin
         partial = partial ^ data[n];
       end
      parity = partial;
    end
endmodule
```

Left shift 2

```
module leftshift2_32(input
  [31:0] a, output [31:0] y);

assign y = {a[29:0], 2'b0};

endmodule
```

```
module zeroext4_8(input [3:0] a,
output [7:0] y);

  assign y = {4'b0, a};

endmodule
```

Test Bench

```
module testbench2();  reg  a, b, c;   wire y;
    example dut(a, b, c, y);
  // apply inputs one at a time;   // checking results
  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
```

Test Bench

```
module testbench3();
  reg         clk, reset;
  reg         a, b, c, yexpected;
  wire        y;
  reg  [31:0] vectornum, errors;
  reg  [3:0]  testvectors[10000:0];

  // instantiate device under test
  example dut(a, b, c, y);
  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end
  // at start of test, load vectors    // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end
```

```
// apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip cycles during reset
      if (y !== yexpected) begin  // check result
        $display("Error: inputs = %b",
                 {a, b, c});
          $display("  outputs = %b (%b expected)",
                 y, yexpected);
          errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $finish;
      end
    end
endmodule
```

Test Bench

Adder

- RTL modeling

```
module adder(c,s,a,b) ;
output c ;
output [7:0] s ;
input [7:0] a,b ;
      assign {c,s} = a + b ;
endmodule
```

- Logic synthesis
  - CLA(carry look ahead) adder for speed optimization
  - ripple adder for area optimization

Tri-State

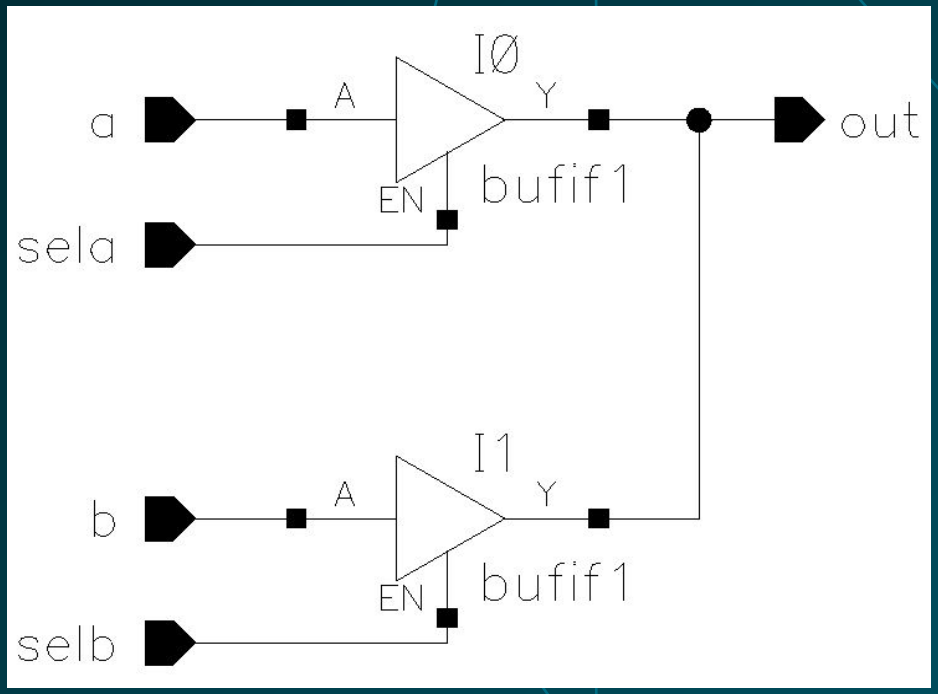
- The value z

```
always @ (sela or a)
    if (sela)
                out  = a ;
    else
                out = 1'bz ;
```

- Another block

```
always @(selb or b)
    if(selb)
                out  =b ;
    else
                out = 1'bz ;
```

assign out = (sela)? a: 1'bz ;



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