Introduction to Verilog

COE 203

Digital Logic Laboratory

Dr. Aiman El-Maleh

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

Outline

- Introduction
- Why use HDL?
- Definition of Module
- Gate Level Modeling
- Verilog Primitives
- A Full Adder
- 4-bit Adder
- Continuous Assignments
- Behavioral Description of an Adder
- Verilog Operators

Introduction

- Verilog is one of the hardware description languages (HDL) available in the industry for hardware designing.
- It allows designers to design at Behavior Level, Register Transfer Level (RTL), Gate level and at switch level.
- Parallel not serial (Not like C language).
- Verilog can describe everything from single gate to full computer system.

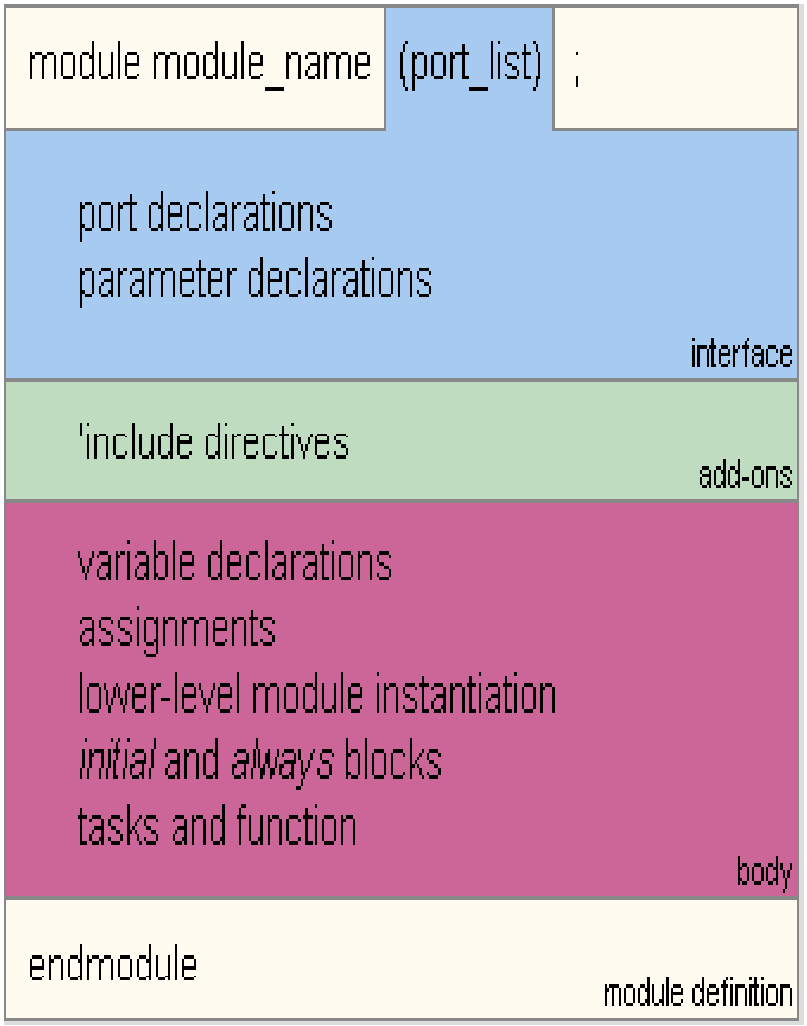
Why use HDL ?

- Digital systems are highly complex; millions of transistors.
- For large digital systems, gate-level design is very difficult to achieve in a short time.
- Verilog allows hardware designers to express their designs with behavioral constructs, deferring the details of implementation to a later stage in the final design.
- Computer-aided design tools aid in the design process.

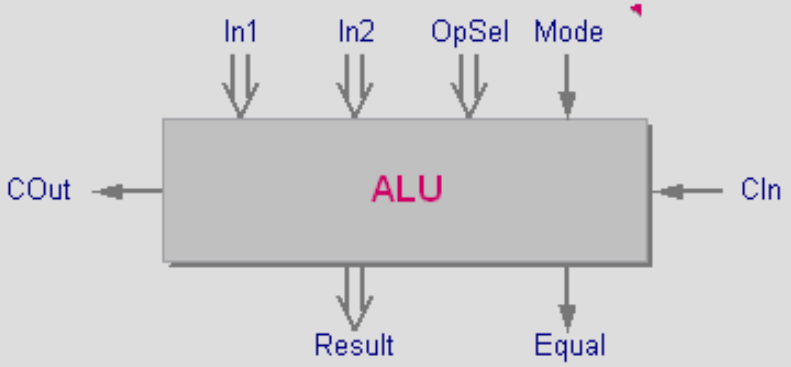
- © Intel P4 Processor
- Introduced in 2000
- 40 Million Transistors
- 1.5GHz Clock

Definition of Module

- The **<module name>** is an identifier that uniquely names the module.
- The **<port list> is** a list of input, inout and output ports which are used to connect to other modules.
- Interface: port and parameter declaration
- Body: Internal part of module
- Add-ons (optional)



The Module Interface …

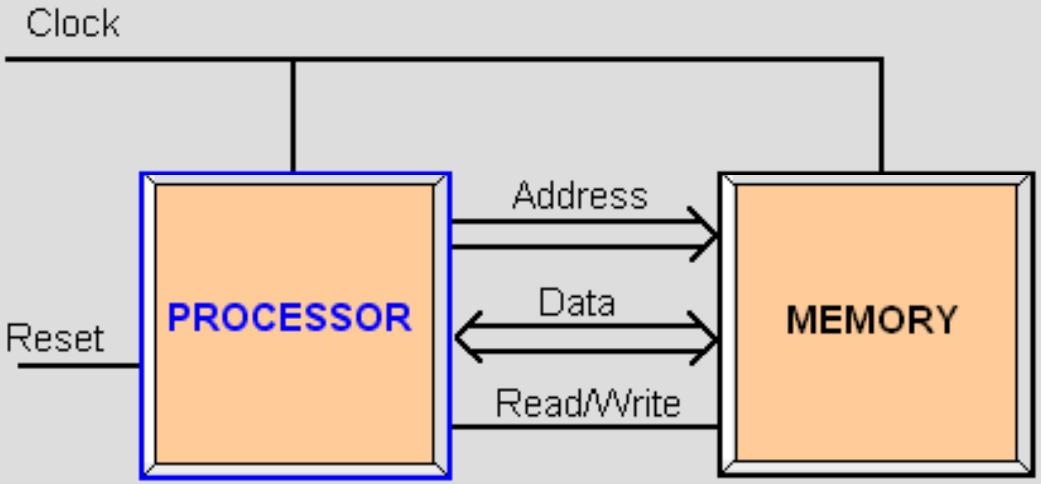


Port List

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
```

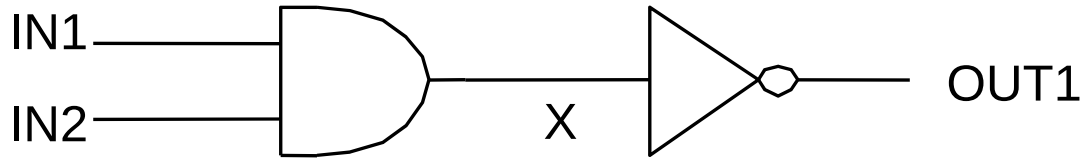
… The Module Interface




```
module Processor (Clock, Reset, Read_Write, Data, Address);
input Clock;
input Reset;
output Read_Write;
inout [15:0] Data;
output [19:0] Address;
. . .
endmodule
```

Gate Level Modeling

- Net-list description
  - built-in primitives gates



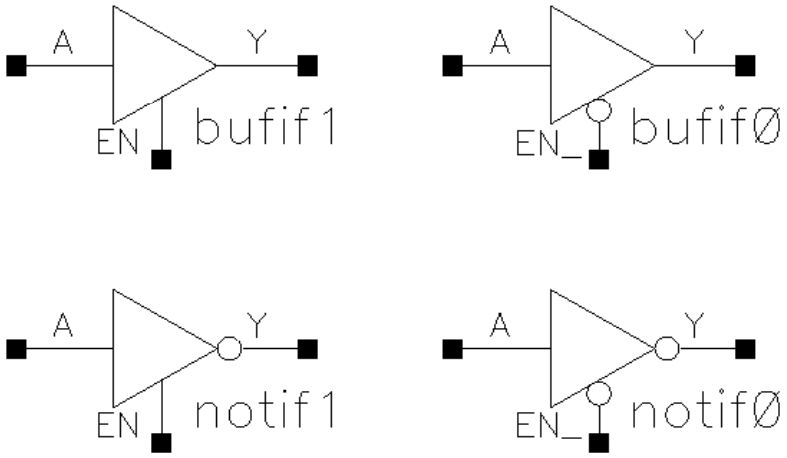

```
module my_gate(OUT1, IN1, IN2);
    output OUT1;
    input IN1, IN2;
    wire X;
    and  (X, IN1, IN2);
    not (OUT1, X);
endmodule
```

Internal Signal

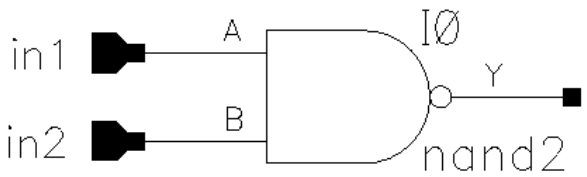
Any internal net must be defined as wire

Verilog Primitives

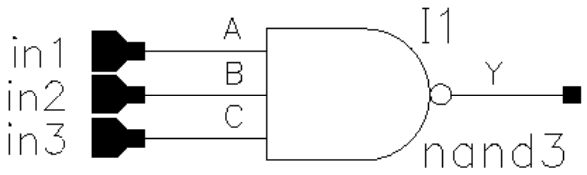
- Basic logic gates only
  - and
  - or
  - not
  - buf
  - xor
  - nand
  - nor
  - xnor
  - bufif1, bufif0
  - notif1, notif0



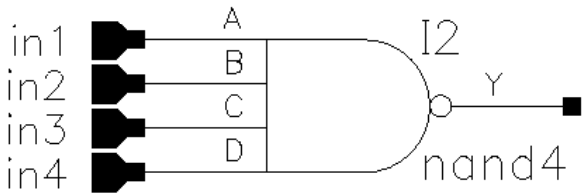
Primitive Pins Are Expandable



nand (y, in1, in2) ;



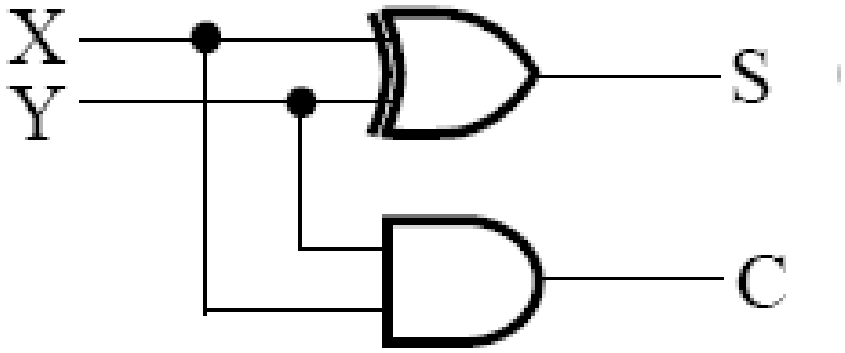
nand (y, in1, in2, in3) ;



nand (y, in1, in2, in3, in4) ;

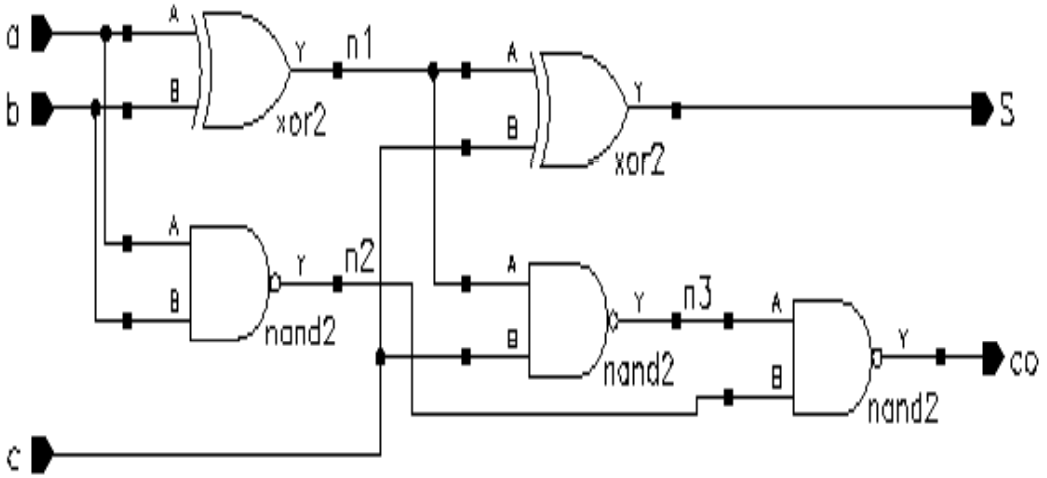
A Half Adder

```
module hadd (S, C, X, Y);
    input X, Y;
    output S, C;

    xor (S, X, Y);
    and (C, X, Y);
endmodule
```



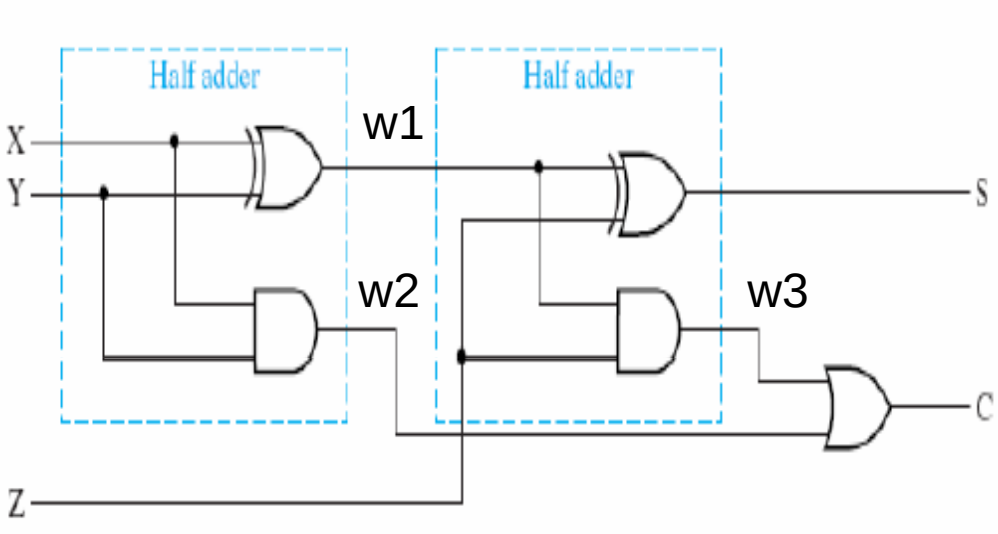
A Full Adder

```
module fadd (co, s, a, b, c);
    input a, b ,c ;
    output co, s ;
    wire n1, n2, n3;
        xor (n1, a, b) ;
        xor (s, n1, c) ;
        nand (n2, a, b) ;
        nand (n3,n1, c) ;
        nand (co, n3,n2) ;
endmodule
```



Instantiation of Modules

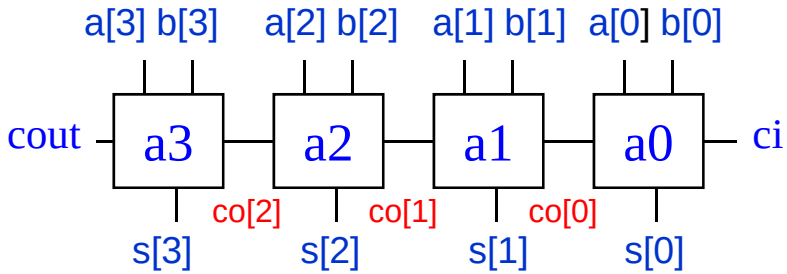
```
module fadd (S, C, X, Y, Z);
    input X, Y, Z;
    output S, C;
    wire w1, w2, w3;

    hadd M1 (w1, w2, X, Y);
    hadd M2 (S, w3, w1, Z);
    or (C, w2, w3);
endmodule
```



- Here we instantiate hadd twice. i.e., placing two hadd circuits and connecting them.
- This full adder is built from two half adders and an OR gate.

4-bit Adder

```
module add4 (s,cout,ci,a,b);
    input [3:0] a,b ;       // port declarations
    input ci ;
    output [3:0] s ;        // vector
    output cout ;
    wire [2:0] co ;
        fadd a0 (co[0], s[0], a[0], b[0], ci) ;
        fadd a1 (co[1], s[1], a[1], b[1], co[0]) ;
        fadd a2 (co[2], s[2], a[2], b[2], co[1]) ;
        fadd a3 (cout, s[3], a[3], b[3], co[2]) ;
endmodule
```



Continuous Assignments

- Describe combinational logic
- Operands + operators
- Drive values to a net
  - assign out = a&b ; // and gate
  - assign eq = (a==b) ; // comparator
  - wire #10 inv = ~in ; // inverter with delay
  - wire [7:0] c = a+b ; // 8-bit adder
- Avoid logic loops
  - assign a = b + a ;
  - asynchronous design

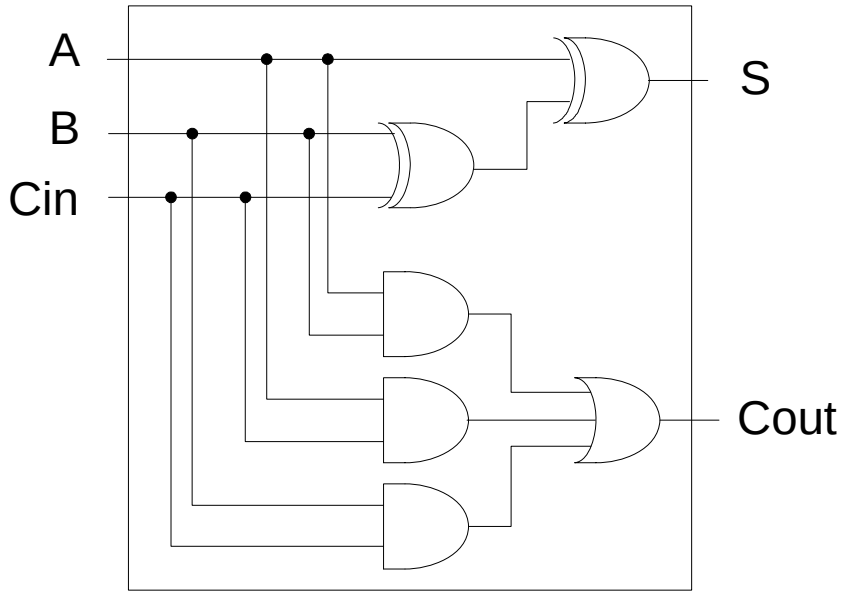
Simple XOR Gate

```
module my_xor( C, A, B );
   output C;
   input A, B;

   assign C = (A ^ B);
endmodule
```

Operation	Operator
~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

Full Adder

```
module fadd (S, Cout, A, B, Cin);
    output  S, Cout;
     input   A, B, Cin;

     assign S = A ^(B ^ Cin);

    assign Cout =   (A & B)
                  | (A & Cin)
                  | (B & Cin)  ;

endmodule
```



Behavioral Description of an Adder

```
module adder4 ( S, Cout,  A, B, Cin);
   input [3:0]  A, B;
   input Cin;
   output [3:0]  S;
   output Cout;

   assign { Cout, S } = A + B + Cin;
    // note:  Verilog treats wires as 'unsigned' numbers
endmodule
```

4-bit operands,
5-bit result

{ Cout, S } is a 5 bit bus:

Cout S[3] S[2] S[1] S[0]

Behavioral Description of an Adder

```
module adder (cout, sum, a, b, cin);
    parameter width = 2;
    input cin;
    input [width:0]      a, b;
    output [width:0]     sum;
    output cout;
    assign {cout, sum} = a + b + cin;
endmodule
```

Verilog Operators

{ }	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical NOT
&&	logical AND
||	logical OR
==	logical equality
!=	logical inequality
? :	conditional

~	bit-wise NOT
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
^~ ~^	bit-wise XNOR
&	reduction AND
|	reduction OR
~&	reduction NAND
~|	reduction NOR
^	reduction XOR
~^ ^~	reduction XNOR
<<	shift left
>>	shift right