Combinational Logic Design with Verilog

ECE 152A – Winter 2012

Reading Assignment

- Brown and Vranesic
  - 2 Introduction to Logic Circuits
    - 2.10 Introduction to Verilog
      - 2.10.1 Structural Specification of Logic Circuits
      - 2.10.2 Behavioral Specification of Logic Circuits
      - 2.10.3 How *Not* to Write Verilog Code

Reading Assignment

- Brown and Vranesic (cont) *1st edition only!*
  - 4 Optimized Implementation of Logic Functions
    - 4.12 CAD Tools
      - 4.12.1 Logic Synthesis and Optimization
      - 4.12.2 Physical Design
      - 4.12.3 Timing Simulation
      - 4.12.4 Summary of Design Flow
      - 4.12.5 Examples of Circuits Synthesized from Verilog Code

Programmable Logic

- Provides low cost and flexibility in a design
  - Replace multiple discrete gates with single device
  - Logical design can be changed by reprogramming the device
    - No change in board design
  - Logical design can be changed even after the part has been soldered onto the circuit board in modern, In-system programmable device
  - Inventory can focus on one part
    - Multiple uses of same device

Programmable Logic

- Evolution of Programmable Logic
  - Both in time and complexity
  - ROM's and RAM's
    - Not strictly programmable logic, but useful in implementing combinational logic and state machines
  - PAL's
    - PAL's – Programmable Array Logic
    - PLA's – Programmable Logic Array
    - GAL's – Generic Logic Array

Programmable Logic

- PLD's
  - Programmable Logic Device
    - PLDs are (in general) advanced PALs
- CPLD's
  - Complex Programmable Logic Device
    - Multiple PLDs on a single chip
- FPGA's
  - Field Programmable Gate Array

Design Entry

- In previous examples, design entry is schematic based
    - TTL implementation using standard, discrete integrated circuits
    - PLD implementation using library of primitive elements
- Code based design entry uses a hardware description language (HDL) for design entry
    - Code is synthesized and implemented on a PLD

Verilog Design

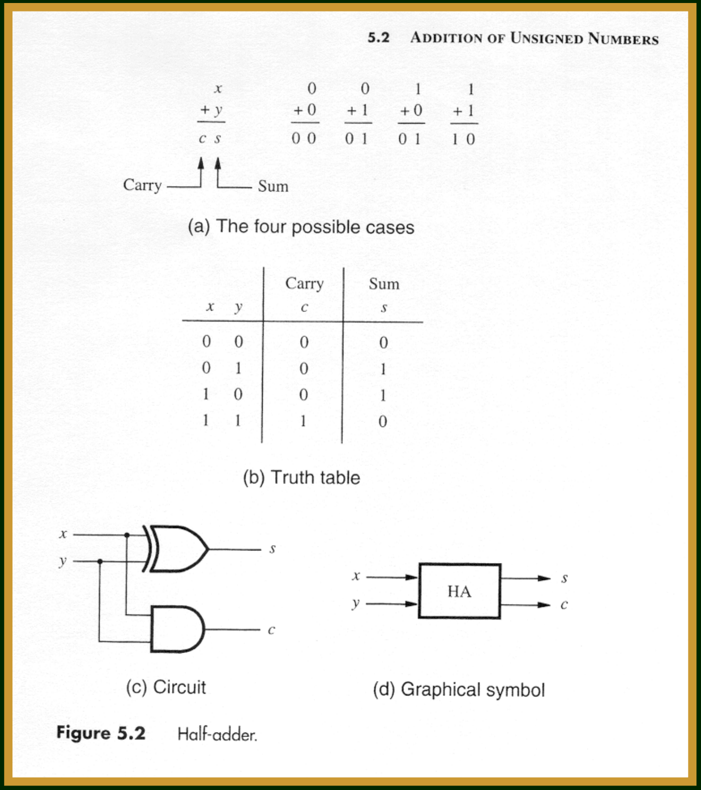
- Structural Verilog
  - Looks like the gate level implementation
    - Specify gates and interconnection
  - Text form of schematic
    - Referred to as “netlist”
  - Allows for “bottom – up” design
    - Begin with primitives, instantiate in larger blocks

Verilog Design

- RTL (Register Transfer Level) Verilog
  - Allows for “top – down” design
  - No gate structure or interconnection specified
  - Synthesizable code (by definition)
    - Emphasis on synthesis, not simulation
      - vs. high level behavioral code and test benches
    - No timing specified in code
    - No initialization specified in code
      - Timing, stimulus, initialization, etc. generated in testbench (later)

Half Adder - Structural Verilog Design

- Recall Half Adder description from schematic based design example
  - Operation
  - Truth table
  - Circuit
  - Graphical symbol



5.2 Addition of Unsigned Numbers

$$\begin{array}{r} x \\ +\,y \\ \hline c\;s \end{array} \qquad \begin{array}{r} 0 \\ +\,0 \\ \hline 0\;0 \end{array} \quad \begin{array}{r} 0 \\ +\,1 \\ \hline 0\;1 \end{array} \quad \begin{array}{r} 1 \\ +\,0 \\ \hline 0\;1 \end{array} \quad \begin{array}{r} 1 \\ +\,1 \\ \hline 1\;0 \end{array}$$

(a) The four possible cases

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table

(c) Circuit

(d) Graphical symbol

Figure 5.2 Half-adder.

Verilog Syntax

- Modules are the basic unit of Verilog models
  - Functional Description
    - Unambiguously describes module's operation
      - Functional, i.e., without timing information
  - Input, Output and Bidirectional ports for interfaces
  - May include instantiations of other modules
    - Allows building of hierarchy

Verilog Syntax

- Module declaration
  - module ADD_HALF (s,c,x,y);
    - Parameter list is I/O Ports
- Port declaration
  - Can be input, output or inout (bidirectional)
    - output s,c;
    - input x,y;

Verilog Syntax

- Declare nodes as wires or reg
  - Wires assigned to declaratively
  - Reg assigned to procedurally
    - More on this later
  - In a combinational circuit, all nodes can, but don't have to be, declared wires
      - Depends on how code is written
    - Node defaults to wire if not declared otherwise
    - wire s,c,x,y;

Verilog Syntax

- Gates and interconnection
  - xor G1(s,x,y);
  - and G2(c,x,y);
    - Verilog gate level primitive
      - Gate name
    - Internal (local) name
      - Instance name
    - Parameter list
      - Output port, input port, input port…

Gate Instantiation

- Verilog Gates
  - Note: *notif* and *bufif* are tri-state gates

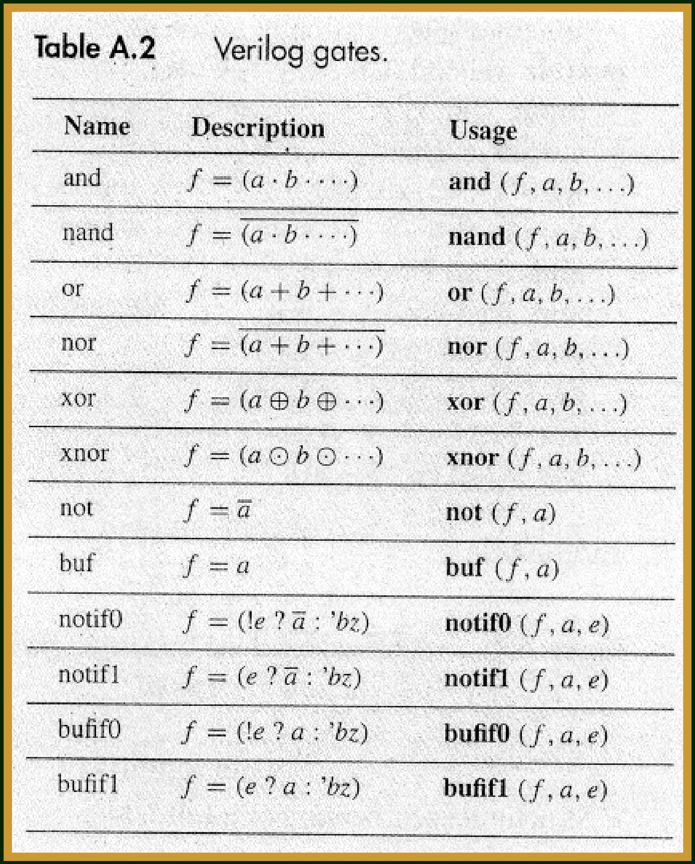


Table A.2 Verilog gates.

Name	Description	Usage
and	$f = (a \cdot b \cdots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \overline{a}$	**not** (f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e \ ? \ \overline{a} : \text{'}bz)$	**notif0** (f, a, e)
notif1	$f = (e \ ? \ \overline{a} : \text{'}bz)$	**notif1** (f, a, e)
bufif0	$f = (!e \ ? \ a : \text{'}bz)$	**bufif0** (f, a, e)
bufif1	$f = (e \ ? \ a : \text{'}bz)$	**bufif1** (f, a, e)

Verilog Syntax

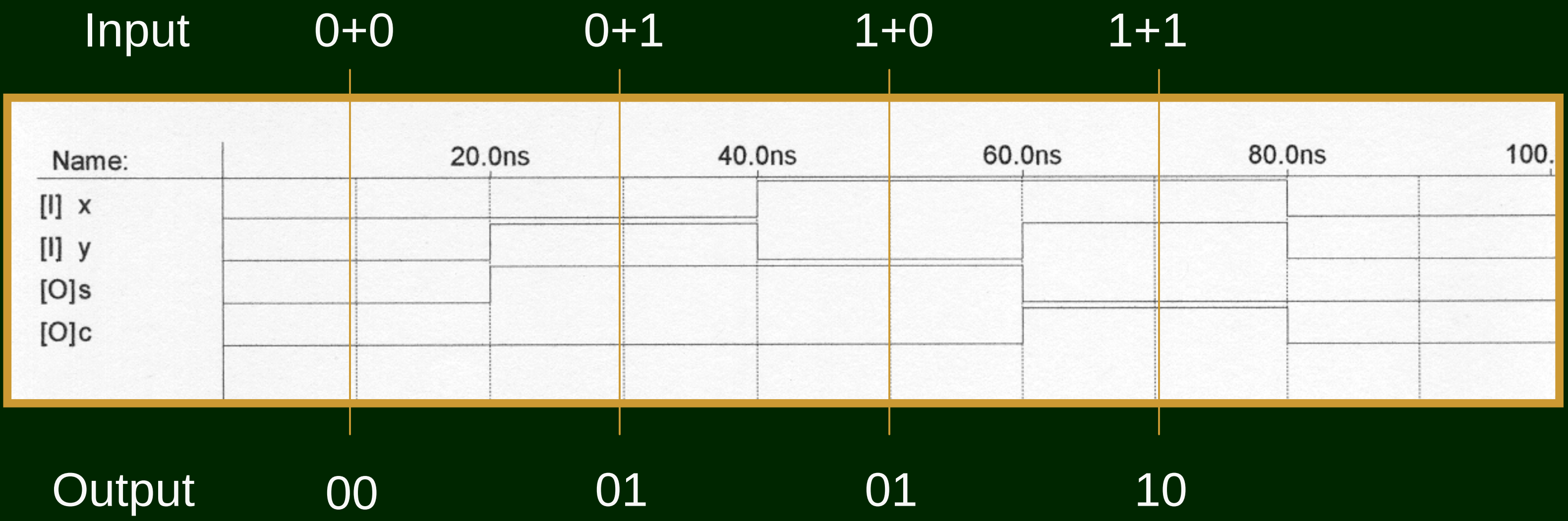
- Close the module definition with
  - endmodule
- Comments begin with //

Half Adder - Structural Verilog Design

```
module ADD_HALF (s,c,x,y);

    output s,c;
    input x,y;

    wire s,c,x,y;
          // this line is optional since nodes default to wires

    xor G1 (s,x,y);  // instantiation of XOR gate
    and G2 (c,x,y);  // instantiation of AND gate

endmodule
```

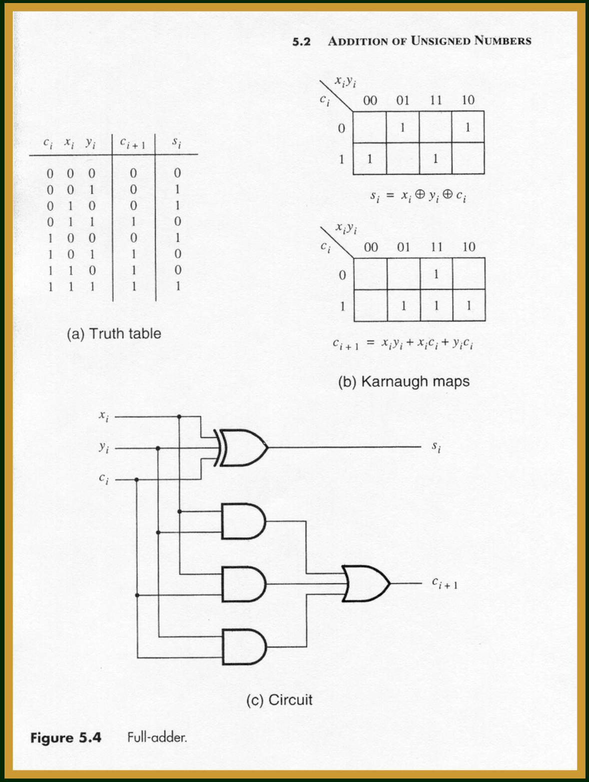
Half Adder – PLD Implementation

- Functional Simulation



Full Adder – Structural Verilog Design

- Recall Full Adder description from schematic based design example
  - Truth table
  - Karnaugh maps
  - Circuit



c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

c_i \ x_iy_i	00	01	11	10
0		1		1
1	1		1	

$$s_i = x_i \oplus y_i \oplus c_i$$

c_i \ x_iy_i	00	01	11	10
0			1	
1		1	1	1

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

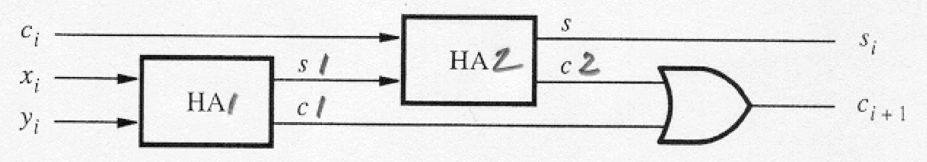
(b) Karnaugh maps

x_i
y_i
c_i
s_i
c_{i+1}

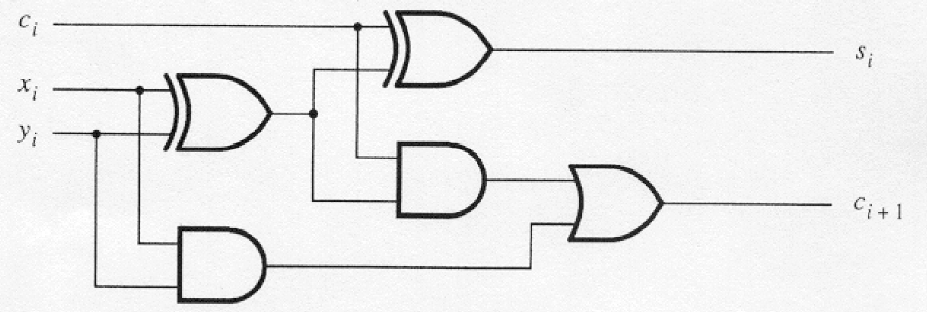
(c) Circuit

Figure 5.4 Full-adder.

Full Adder from 2 Half Adders



(a) Block diagram



(b) Detailed diagram

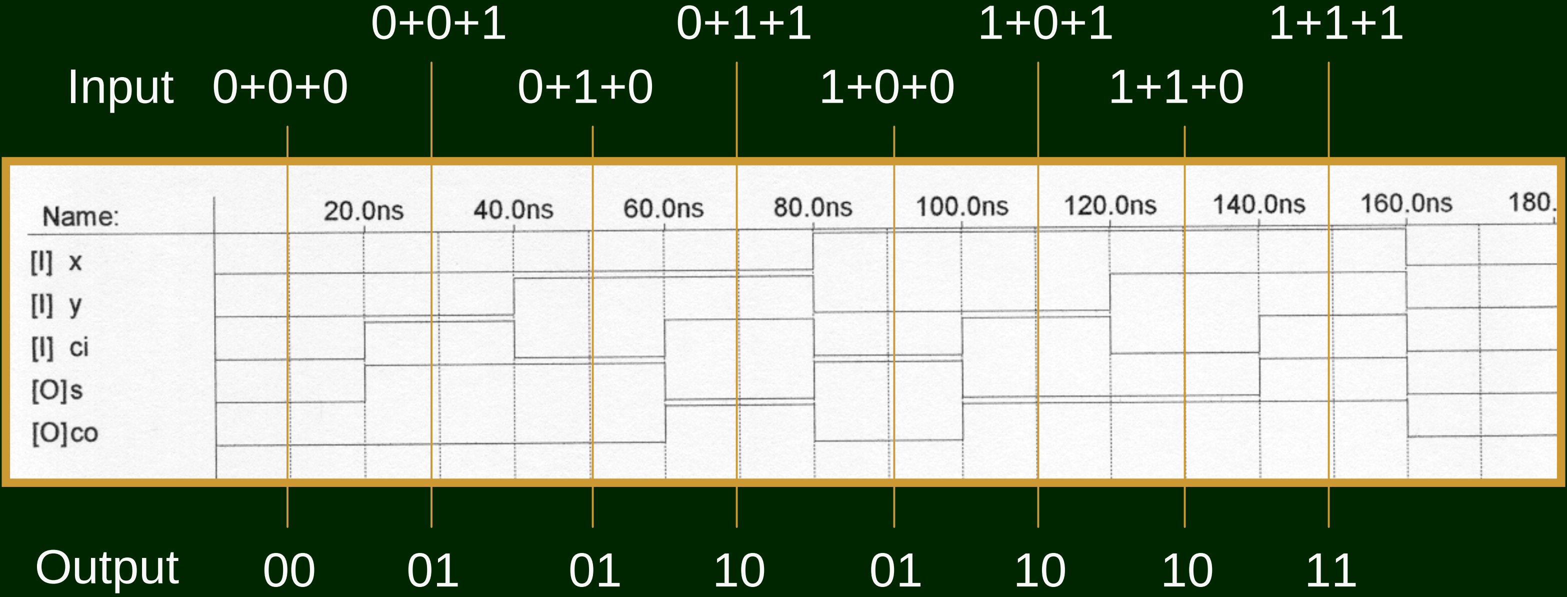
Figure 5.5 A decomposed implementation of the full-adder circuit.

Full Adder – Structural Verilog Design

```
module ADD_FULL (s,cout,x,y,cin);

    output s,cout;
    input x,y,cin;

    //internal nodes also declared as wires
    wire cin,x,y,s,cout,s1,c1,c2;

    ADD_HALF HA1(s1,c1,x,y);
    ADD_HALF HA2(s,c2,cin,s1);
    or (cout,c1,c2);

endmodule
```

Full Adder – PLD Implementation

- Functional Simulation



Verilog Operators

- The Verilog language includes a large number of logical and arithmetic operators
  - Bit length column indicates width of result

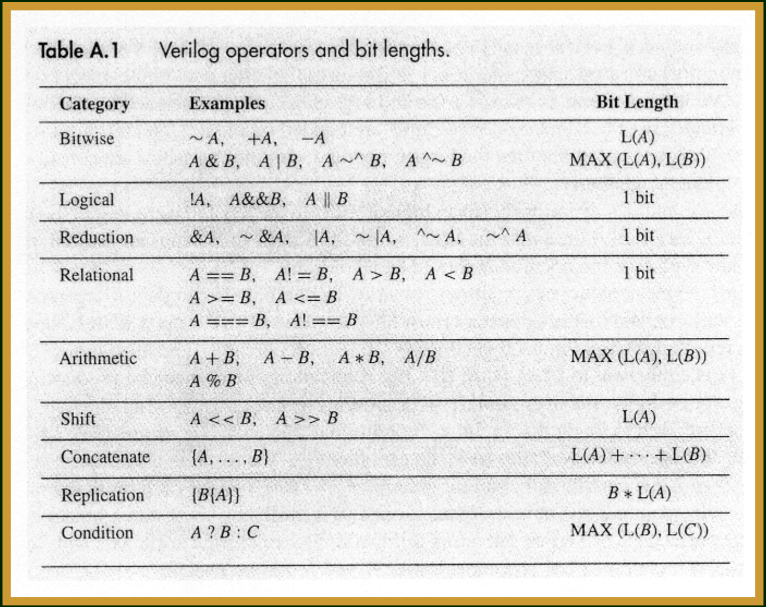


Table A.1 Verilog operators and bit lengths.

Category	Examples	Bit Length
Bitwise	~A, +A, −A A & B, A | B, A ~^ B, A ^~ B	L(A) MAX (L(A), L(B))
Logical	!A, A&&B, A || B	1 bit
Reduction	&A, ~&A, |A, ~|A, ^~A, ~^A	1 bit
Relational	A == B, A! = B, A > B, A < B A >= B, A <= B A === B, A! == B	1 bit
Arithmetic	A + B, A − B, A ∗ B, A/B A % B	MAX (L(A), L(B))
Shift	A << B, A >> B	L(A)
Concatenate	{A, . . . , B}	L(A) + · · · + L(B)
Replication	{B{A}}	B ∗ L(A)
Condition	A ? B : C	MAX (L(B), L(C))

Behavioral Specification of Logic Circuits

- Continuous Assignment Operator
  - assign sum = a ^ b;
    - “Assign” to a wire (generated declaratively)
    - Equivalent to
      - xor (sum,a,b);
  - Continuous and concurrent with other wire assignment operations
    - If a or b changes, sum changes accordingly
    - All wire assignment operations occur concurrently
      - Order not specified (or possible)

Full Adder from Logical Operations

```
module ADD_FULL_RTL (sum,cout,x,y,cin);

    output sum,cout;
    input x,y,cin;

    //declaration for continuous assignment
    wire cin,x,y,sum,cout;

    //logical assignment
    assign sum = x ^ y ^ cin;
    assign cout = x & y | x & cin | y & cin;

endmodule
```

Full Adder from Arithmetic Operations

```
module ADD_FULL_RTL (sum,cout,x,y,cin);

    output sum,cout;
    input x,y,cin;

    //declaration for continuous assignment
    wire cin,x,y,sum,cout;

    // concatenation operator and addition
    assign {cout, sum} = x + y + cin;

endmodule
```

Procedural Verilog Statements

- Recall:
  - Wires assigned to declaratively
    - Continuous / concurrent assignment
  - Reg “variables” assigned to procedurally
    - Value is “registered” until next procedural assignment
      - Continuous assignment (wires) occurs immediately on input change
    - Enables clocked (synchronous) timing

Procedural Verilog Statements

- The "always" block
  - Syntax is "always at the occurrence *(@)* of any event on the *sensitivity list*, execute the statements inside the block (in order)"

```
always @ (x or y or cin)
    {cout, sum} = x + y + cin;
```

RTL Design of Full Adder

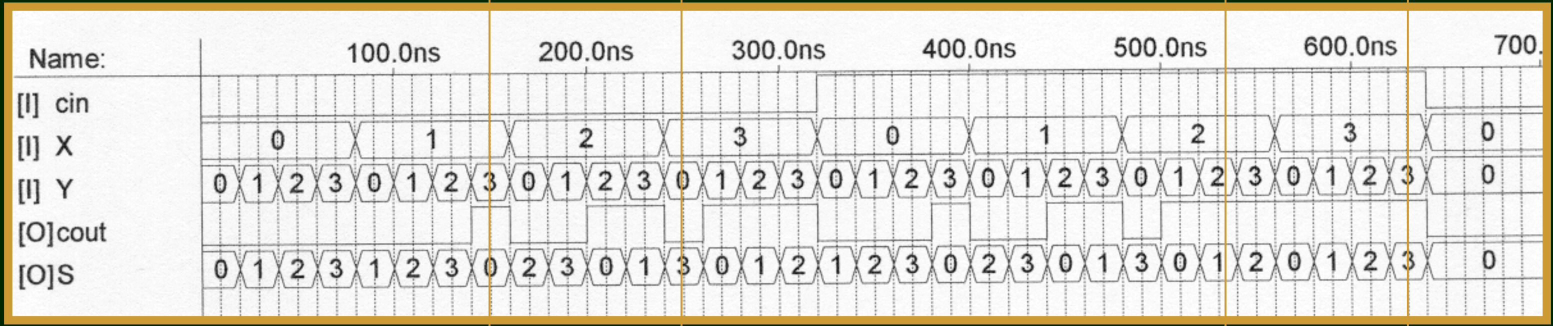
```
module ADD_FULL_RTL (sum,cout,x,y,cin);

    output sum,cout;
    input x,y,cin;

    //declaration for behavioral model
    wire cin,x,y;
    reg sum,cout;

    // behavioral specification
    always @ (x or y or cin)
             {cout, sum} = x + y + cin;

endmodule
```

Two-bit, Ripple Carry Adder – Structural Verilog

```
module TWO_BIT_ADD (S,X,Y,cin,cout);

    input cin;
    input [1:0]X,Y;     // vectored input
    output [1:0]S;      // and output signals
    output cout;

    wire cinternal;

    ADD_FULL AF0(S[0],cinternal,X[0],Y[0],cin);
    ADD_FULL AF1(S[1],cout,X[1],Y[1],cinternal);

endmodule
```

Two-bit, Ripple Carry Adder – PLD Implementation

- Functional Simulation
  - Base-4 Bus Representation of X, Y and Sum



$0+1+3 = 4 = 10_4 \rightarrow$

$1+2+2 = 5 = 11_4 \rightarrow$

$0+3+0 = 3 = 03_4 \rightarrow$

$1+3+3 = 7 = 13_4 \rightarrow$

Verilog Test Bench

- Device Under Test (DUT)
  - Circuit being designed/developed
    - Full adder for this example
- Testbench
  - Provides stimulus to DUT
    - Like test equipment on a bench
- Instantiate DUT in testbench
  - Generate all signals in testbench
  - No I/O (parameter list) in testbench

Full Adder Testbench Example

```
module ADDFULL_TB;

reg a,b,ci;
wire sum,co;

initial begin
        a = 0;
        b = 0;
        ci = 0;
end

always begin
        #5 a = ~a;
end

always begin
        #10 b = ~b;
end

always begin
        #20 ci = ~ci;
end

ADD_FULL  AF1(sum,co,a,b,ci);

endmodule
```