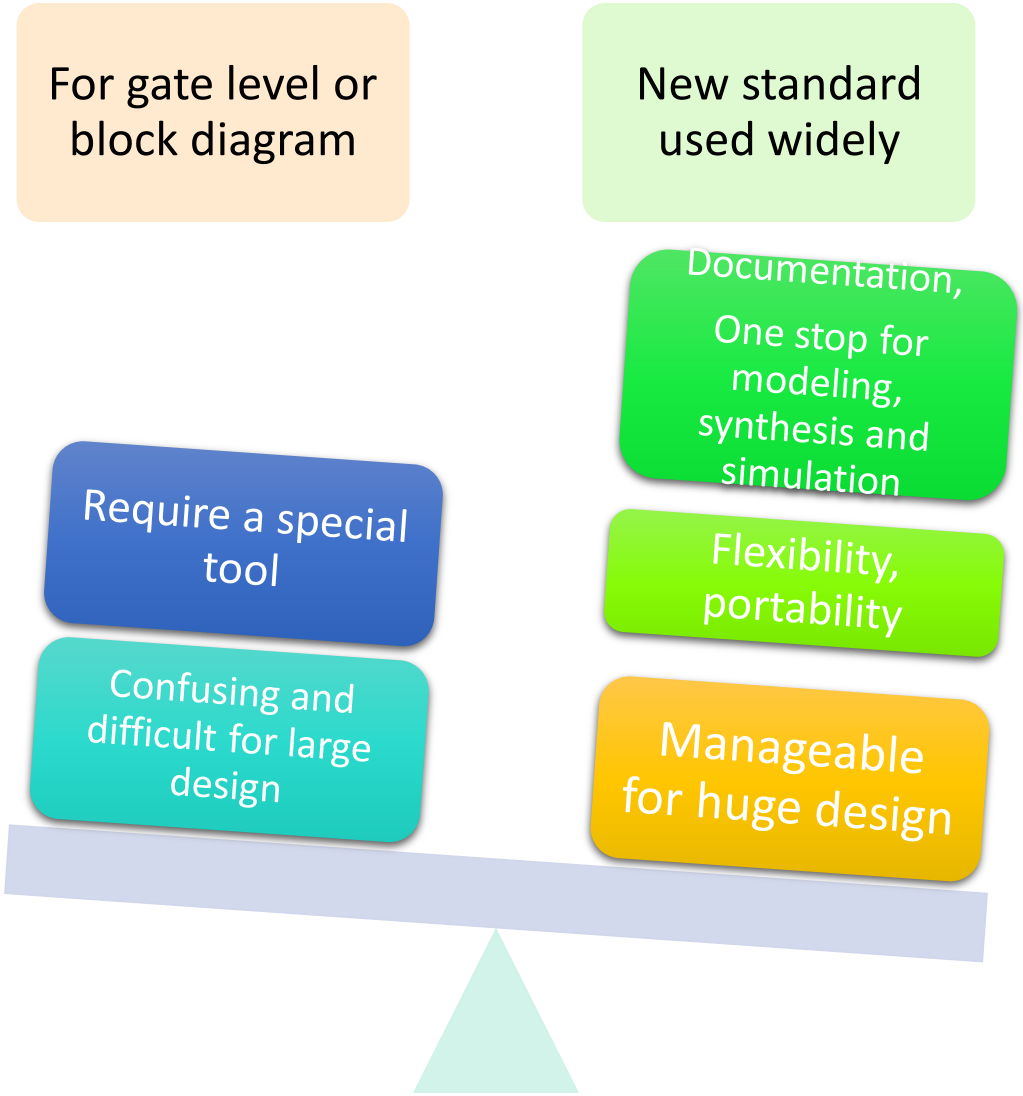
# Introduction to Verilog HDL

# Outline

- Introduction to HDL Coding
- History of Verilog
- Schematic versus Hardware Description Language (HDL)
- Representation of numbers in Verilog
- Operators in Verilog
- Continuous & Sequential Assignments
- Verilog Modelling Styles
- Introduction to Dataflow Modelling
- Introduction to Structure Modelling
- Introduction to Behavioural Modelling

# Schematic versus HDL



# History of Verilog

- **1983**
  - Originated at Automated Integrated Design Systems (renamed Gateway Design Automation)
- **1985**
  - Enhanced Verilog called Verilog-XL
- **1989**
  - Acquired by Cadence
- **1990**
  - Verilog is open to public
  - An IEEE working group is set up develop the standard
- **1993**
  - IEEE Standard 1394-1995
- **2001**
  - IEEE Standard 1364-2001
- **2005**
  - Minor Revision

**Simulation language**

- Subset of the Verilog constructs & operators

**Added with synthesis function**

- All Verilog constructs & operators

## Verilog

- Open to public in 1990
- Preferred by Silicon Valley's companies

## VHDL

- Open to public since 1980's
- Preferred by universities, companies at Japan and Europe

# HDL Coding for Synthesis

- What does the word **Synthesis** mean?

  *the combining of the constituent elements of separate material or abstract entities into a single or unified entity*
  *(oppose to analysis) the separating of any material or abstract entity into its constituent elements.*

  i.e. *creating a circuit*

- Default industry standard hardware description languages for modeling digital systems:

VHSIC (Very High Speed Integrated Cirduit) HDL or known simply as **VHDL.** First standardized in 1987 by the IEEE standard 1076. Ever since then there are 4 versions of IEEE 1076 – in 1987, 1993, 2002 and 2008

**Verilog HDL**. Simply known as **Verilog** . Adopted as IEEE Standard 1364-1995 in 1995. Enhanced version in 2001 called as Verilog-2001 as IEEE Standard 1364-2001. A further minor revision done in 2005 known as Verilog-2005 commonly known to HDL designer as **System Verilog**.

# HDL Coding for Synthesis

- The focus of this course is tailored towards the RTL design and synthesis using HDL (Verilog, in specific), and not the HDL language itself.
- Therefore, we do not attempt to cover Verilog language in detail.
- We use Verilog just as a mean to do RTL design and synthesis.
- We use Verilog to describe (to model) the intended hardware implementation, to develop Verilog code that accurately describes the underlying hardware architecture and to provide correct information to ensure the synthesis tool can generate efficient implementations.

# HDL Coding for Simulation and Synthesis

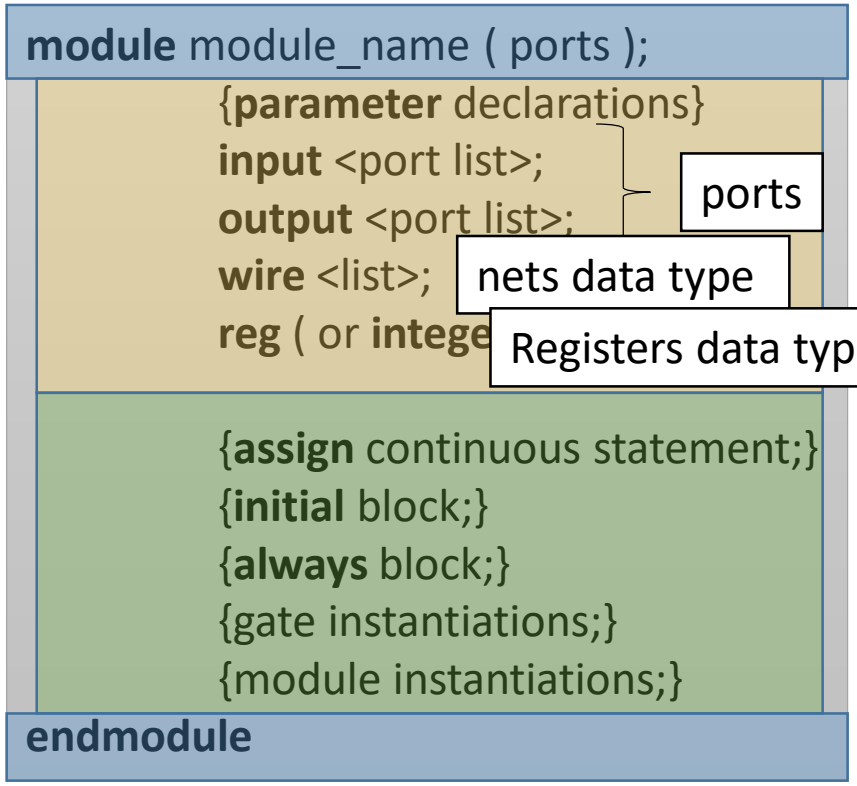
In simulation mode - Design is realized only in virtual environment i.e. software simulator. All constructs and operators in Verilog can be used for simulation

For hardware implementation or synthesis-

- Not all constructs but only a subset of Verilog HDL can be used for synthesis.
- The synthesis tools understands only a subset of Verilog, called “RTL Verilog”
- Verilog constructs such as file operations and assertions cannot be used to describe (build) a circuit in Verilog HDL. Thus named as unsynthesizable codes known as “non-RTL Verilog “. But still they are useful in testing, verification and simulation.

# Module

- Verilog describes a digital circuit or a system as a set of modules.
- Thus, the basic design unit used in Verilog description of hardware components is a module. (In VHDL it is called **entity**)
- General structure and syntax for a module (Verilog)




A verilog design consists of 3 main parts:

**Registers Data Type**

**Registers** data type is declared by keyword ***reg***

*A* ***reg*** data type is assigned for assignment that can hold and store value

Can also be used to declare values, integers. Much like variables in software languages

Module item

describes the behavior of the circuit/ system

# Example of port, net & variable declarations in a module

```
module circuitA (Cin, x, y, X, Y, Cout, s, S, out, Bus);
    //{parameter declarations}
    input  Cin, x, y;
    input  [3:0]  X, Y;
    output  Cout, s;
    output  [3:0] S;
    Inout  [7:0] Bus;
    wire  d;
    reg  e;

    {assign continuous statement;}  // behavior statements
    {initial block;}
    {always block;}
    {gate instantiations;}
    {module instantiations;}
endmodule
```

# Representation of Numbers (1)

**Four-value logic system**

- 0 – logic 0 , *Gnd* or false condition
- 1 – logic 1 or true condition
- x, X – unknown logic value
- z, Z - high-impedance state or *float*

**Number formats**

- binary - **b**, **B**
- decimal (defult) - **d**, **D**
- hexadecimal - **h, H**
- octal - **o**, **O**

**For sized numbers**, the format is:

**<size-in-bits> ' <radix identifier> <value>**

**16'H789A**

– Number in 789Ah in 16-bit number (in hex format )

**1'b0**

– number 0 in 1-bit binary format

**For unsized numbers**, the format is :

**' <radix identifier> <value>**

**123**

- Unsized decimal

**'h 12**

- Unsized hex 12 ( decimal 18).

# Representation of Numbers (2)

Examples:

- $100010110_2$ in binary format
  - 9'b100010110
  - or 9'b 1_0001_0110
  - Stored pattern = 100010110
- 116h in 9-digit hex format
  - 9'h 116
  - Stored pattern = 1_0001_0110
- 3'o5
  - Number $5_8$ in 3-digit octal format
  - Stored pattern = 101
- 12'h8A9
  - Number $8A9_8$ in 12-digit hex format
  - Stored pattern = 1000 1010 1001

# Representation of Numbers (2)

## Examples:

- -4'b101
  - $-5_{10}$ in 4-digit 2's complement format
  - Stored pattern = 1011
- -8'd6
  - $-6_{10}$ in 8-digit format
  - Stored pattern = 11111010
- 8'hz3
  - 3h in 8-bit format
  - Stored pattern = zzzz0011

```
module constants;
parameter H12_UNSIZED = 'h 12; // Unsized hex 12 = decimal 18.
parameter H12_SIZED = 6'h 12; // Sized hex 12 = decimal 18. //
Note: a space between base and value is OK. //
Note: `' (single apostrophes) are not the same as the ' character.
parameter D42 = 8'B0010_1010; // bin 101010 = dec 42 // OK to use underscores
to increase readability.
 parameter D123 = 123; // Unsized decimal (the default).
parameter D63 = 8'o 77; // Sized octal, decimal 63.
// parameter ILLEGAL = 1'o9; // No 9's in octal numbers!
// A = 'hx and B = 'ox assume a 32 bit width.
parameter A = 'h x, B = 'o x, C = 8'b x, D = 'h z, E = 16'h ????;
// Note the use of ? instead of z, 16'h ???? is the same as 16'h zzzz.
// Also note the automatic extension to a width of 16 bits.
reg [3:0] B0011,Bxxx1,Bzzz1;
real R1,R2,R3;
integer I1,I3,I_3;
parameter BXZ = 8'b1x0x1z0z;

initial begin B0011 = 4'b11; Bxxx1 = 4'bx1; Bzzz1 = 4'bz1;
// Left padded. R1 = 0.1e1; R2 = 2.0; R3 = 30E-01;
// Real numbers. I1 = 1.1; I3 = 2.5; I_3 = -2.5;
// IEEE rounds away from 0.
end
```

# Representation of Numbers (3)

By default, Values are unsigned

- e.g., **C[4:0] = A[3:0] + B[3:0];**
- if A = 0110 (6) and B = 1010(-6)

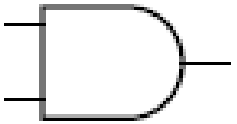
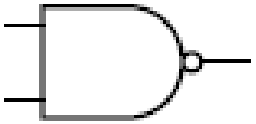
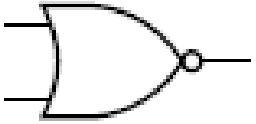
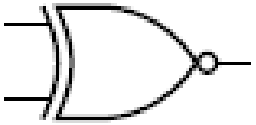
  C = 10000 not 00000 i.e., B is zero-padded, not sign-extended

- **wire signed [31:0] x;**
- Declares a signed (2's complement) signal array.

# Operators in Verilog

| Operator type | Operator symbol | Operation |
|---|---|---|
| Arithmetic | +, -, *, /, %, ** | add, sub, multiply, divide, modulus, power |
| Binary Bitwise | ~, &, ~&, \|, ~\|, ^, ~^, ^~ | not, and, nand, or, nor, xor, xnor, xnor |
| Unary Reduction | &, ~&, \|, ~\|, ^, ~^ | Reduction and, nand, or, nor, xor, xnor |
| Logical | !, &&, \|\| | Logical negation, logical and, logical or |
| Equality | ==, !=<br>===, !== | Logical equality, logical inequality, case equality, case inequality |
| Relational | >, <, >=, <= | Gt, Lt, Gt or eq, Lt or eq |
| Logical Shift | >>, << | Right shift, Left shift |
| Arithmetic Shift | >>>, <<< | |
| Concatenation | { , } | |
| Replication | { { } } | |
| Conditional | ? : | |

# Example: Bitwise Operators

| | | |
|---|---|---|
| a & b | $a \cdot b$ |  |
| a \| b | $a + b$ |  |
| ~(a & b) | $\overline{a \cdot b}$ |  |
| ~(a \| b) | $\overline{a + b}$ |  |
| a ^ b | $a \oplus b$ |  |
| a ~^ b | $\overline{a \oplus b}$ |  |
| ~a | $\overline{a}$ |  |

# Bitwise Operation

Bitwise operations will produce the same number of bits as the operands

Each bit in an operand will be operated with the relevant of the other operand . Example if A= a1a0, B= b1b0, and C=c1c0 then
C = A|B will result in c1=a1|b1 and c0=ao|b0

## Examples:

- A=1001, B=0001, A&B=0001
- A=1001, B=x001, A&B=x001
- A=1001, B=z001, A&B=x001
- A=0001, B=z001, A~^B=x111

# Logical Operator

There are 3 logical operators :

- ! (Logical NOT)
- && (Logical AND)
- || (Logical OR)

- Verilog converts the whole number into either a 1 (if it contains a nonzero bit) or 0 (if it only contains zero), then performs the equivalent bitwise operation. Thus, the answer is also one bit.

  Examples:

  !0000 = ~0 = 1
  !1101 = ~1 = 0
  !1000 = ~1 = 0

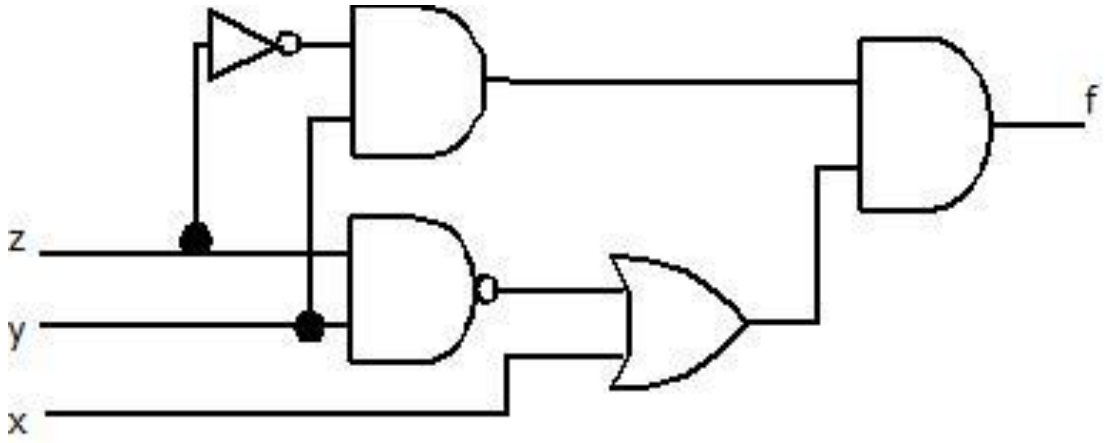
- !(any nonzero binary number) = ~1 = 0

- 0000 && 1101 = 0 & 1 = 0
  0010 && 1101 = 1 & 1 = 1

- 0000 || 0110 = 0 | 1 = 1
  0000 || 0000 = 0 | 0 = 0

# Example: Logical Operator

$$f = (x + (y \cdot \bar{z})) \cdot \overline{(y \cdot z)}$$




```
module circuit (f,  x, y, z );

assign f = (x | ~(y & z)) & (y & ~z);

endmodule
```

# Reduction Operator / Unary Operator
# Unary Reduction

Example:

Operate on single vector and produce one-bit result

- A=0101, B=1111, C=0000
  - &A = 0 & 1 &0 & 1 = 0
  - | B= 0 | 1 | 0 | 1 = 1
  - ~&C= 0 ~& 1 ~& 0 ~& 1 = 0

- D= x001, E=z001
  - ^D = x
  - ~^E = x

# Example: Equality ==, !=, ===, !==

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length
- Result is 0 (false) or 1 (true)
- For the == and != operators, the result is x, if either operand contains an x or a z
- For the === and !== operators, bits with x and z are included in the comparison and must match for the result to be true
- Example:
  - A=z0x1, B=z0x1;  A===B will produce **true (1)**
  - A=z0x1, B=z001;  A===B will produce **false(0)**
  - A=x0x1, B=x001;  A!==B will produce **true (1)**
  - A=z0x1, B=z001;  A!== B will produce **true(1)**
  - A=z0x1, B=z0x1;  A==B will produce **x**
  - A=0011, B=0011; A==B will produce **true(1)**

# Shift Operators
# Logical Shift and Arithmetic Shift

- Shift operator shifts a vector operand left or right by a specified number of bits, filling vacant bit positions with zeros.
- Shifts do not wrap around.
- Arithmetic shift uses context to determine the fill bits.
- Examples when x=4'b1100
  - y = x >> 1; // y is 4'b0110
  - y = x << 1; // y is 4'b1000
  - y = x << 2; // y is 4'b0000

- Examples when x is signed and x=4'b1100
  - y = x >>> 1; // y is 4'b1110 where new bit follow sign bit
  - y = x <<< 1; // y is 4'b1000 where new bit is 0
  - y = x <<< 2; // y is 4'b0000 whre new bits are 0

# Concatenation, Replication, Conditional

- **Concatenation**
  - A=1'b1; B=2'b00; C=6'b101001
  - {A,B} = 3'b100;
  - {C[5:3], A} =4'b1011

- **Replication**
  - A=1'b1; B=2'b01
  - {4{A}} = 4'b1111;
  - {B, 4{A}} = 6'b011111

- **Conditional**
  - Z = A? B:C

  Z=B if A is true. Otherwise, Z=C

# Verilog Procedures and Assignments

- Inside behavior statements of a module - there are procedures and assignments
- A Verilog **procedure** can comprised of an **always** block, **initial** block, a **task** , or a **function** .
- The statements within a sequential block (statements that appear between a **begin statement** and an **end** statement ) which is part of a procedure execute sequentially in the order in which they appear,
- But more than one procedure can exist in a single module. The procedures executes concurrently with other procedures.
- This is a fundamental difference from computer programming languages. Think of each procedure as a microprocessor running on its own and at the same time as all the other microprocessors (procedures).
- Verilog has 2 different types of **assignment** statements.
  (i) Continuous assignment statements – which appear outside procedures
  (ii) Sequential assignment statements which appear inside procedures

# Verilog Assignment Statements

- Verilog 2 different types of **assignment** statements.

  (i) Continuous assignment statements - outside procedures

  (ii) Sequential assignment statements – inside procedures

  The difference between **continuous** and **procedural** assignment is, continuous assignments take place in parallel while procedural assignments take place one after the other (sequential). Continuous assignment can model only combinational logic while procedural assignment can model both Combinational and Sequential logic.

# Verilog Assignment Statements (2)

Example of both continuous assignment and procedural assignment statements in a module

Here is an example explaining both continuous and procedural statement.

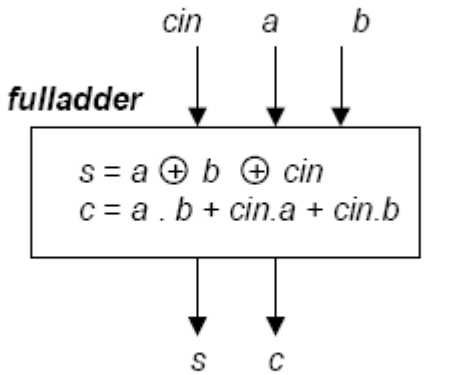
## cont_proc.v

```
1  module cont_proc(in1, in2, out1_cont, out2_cont, out1_proc, out2_proc);
2  input in1, in2;
3  output out1_cont, out2_cont, out1_proc, out2_proc;
4
5  wire in1, in2 ,out1_cont, out2_cont;
6  reg out1_proc, out2_proc;
7  //continuous assignment
8  assign #2 out1_cont = in1 | in2;
9  assign #1 out2_cont = in1 | in2;
10
11 always@(in1, in2) begin
12 //procedural assignment
13  #2 out1_proc = in1 | in2;
14  #1 out2_proc = in1 | in2;
15 end
16 endmodule
```

# Continuous Assignment Statements (1)

- Continuous assignments should appear **outside procedures** (always, initial, functions, tasks, etc). Refer to slide 26
- It overrides any procedural assignment.
- Only combinational logic can be modelled using Continuous Assignment.

```
module fulladder (Cin, x, y, S, Cout) ;
    input Cin, x, y;
    output S, Cout ;
    assign  S  = ( x ^ y ^ Cin ) ;
    assign Cout = (x & y) | (Cin & x) | (Cin & y);
endmodule
```




Block Diagram of *fulladder*

# Continuous Assignment Statements (2)

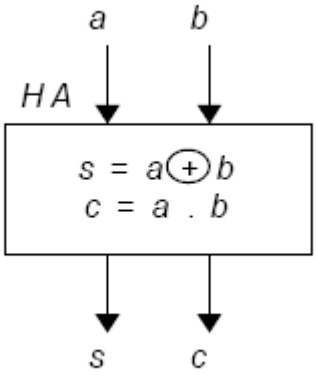
- Executed concurrently (in parallel) and the order of the statements in the module does not matter.
- By Verilog syntax, a continuous assignment statement begins with the keyword **assign**.
- Used in “dataflow modeling to describe combinational logic.
- Continuous assignment can only drive **wire** or **tri** . Which means left-side data type should be **net data type**.
- A **wire** should be assigned only once using continuous assignment.

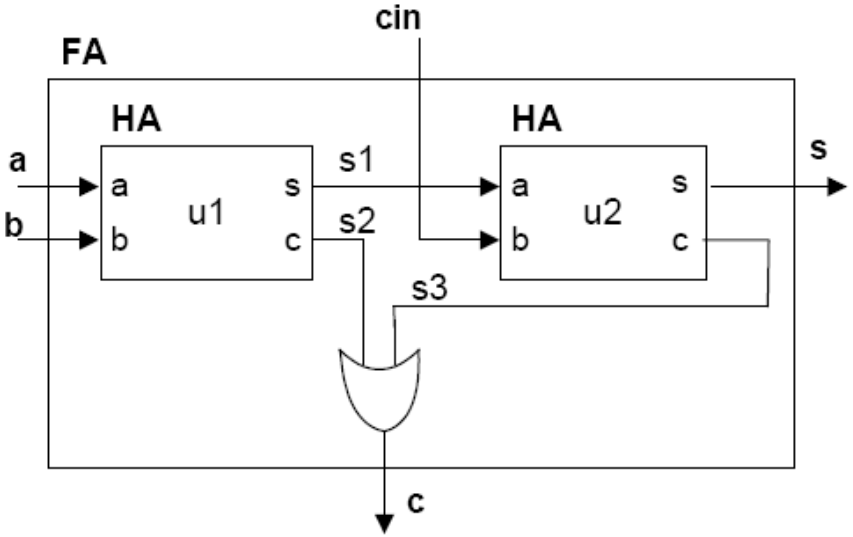
# Continuous Assignment Statements (3)

- Drive values into the nets continuously
- It executes each time the right hand side expression changes.
- Example:

```
wire [3:0] A, X,Y,R,Z;
wire [7:0] P;
wire r, a, cout, cin;
assign r = &X;
assign R = (a == 1'b0) ? X : Y;
assign P[7:0] = {4{X[3]}, X[3:0]};
assign {cout, R} = X + Y + cin;
assign Y = A << 2;
```

# Continuous Assignment Statements (4)




Functional Block of FA

```
module HA ();
input cin, inA, inB;
output outS, outC;

assign s = a^b;
assign c = a & b;

endmodule
```

```
module FA();
input cin, a, b;
output s, c;
wire s1, s2, s3

HA u1 (.inA(a), .inB(b), .outS(s1), .outC(s2);
HA u2 (.inA(s1), .inB(cin), .outS(s), .outC(s3);

assign c = s2 | s3

endmodule
```

# Procedural Assignment Statements (1)

- Also known as sequential assignment.
- Verilog syntax requires procedural assignments to appear **inside** procedures (or **procedural blocks**); **always** block and **initial** block.
- Sequential statements are used in behavioral modelling.
- A sequential assignment statement describes a behavioral function described on the RHS of the statement and assigned to the LHS
- **Sequential assignments are evaluated according to its order in the procedures**
- Procedural assignment can drive only **reg, integer, real and time data type**. Which means left-side data type cannot be nets data type.
- It can model both combinational and sequential logic.

# Procedural Assignment Statements (2)

Example of both continuous assignment and procedural assignment statements in a module

cont_proc.v

```
1  module cont_proc(in1, in2, out1_cont, out2_cont, out1_proc, out2_proc);
2  input in1, in2;
3  output out1_cont, out2_cont, out1_proc, out2_proc;
4
5  wire in1, in2 ,out1_cont, out2_cont;
6  reg out1_proc, out2_proc;
7  //continuous assignment
8  assign #2 out1_cont = in1 | in2;
9  assign #1 out2_cont = in1 | in2;
10
11 always@(in1, in2) begin
12 //procedural assignment
13  #2 out1_proc = in1 | in2;
14  #1 out2_proc = in1 | in2;
15 end
16 endmodule
```

Note :
LHS elements declared as output ports
RHS elements declared as input ports

# Procedural (Sequential) Assignment Statements

- 2 types of procedural blocks:
  - **Initial** - executed once only at initial time (time 0)
  - **Always** – executed always based on the conditions
- Examples of **Initial block** (note: the code is evaluated according to the order):

```
module initial_example ();
  reg clk, reset, enable, data;

  initial begin
    clk = 0;
    reset = 0;
    enable = 0;
    data = 0;
  end
endmodule
```

```
module initial_example ();
  reg clk, reset;
  wire enable, data;

  initial begin
    clk = 0;
    reset = 0;
    enable = 0;
    data = 0;
  end
endmodule
```

# Procedural (Sequential) Assignment Statements

- **Always** – always blocks loop to execute over and over again; in other words, it executes always based on the conditions (as long as the condition is true)

- Examples always block :

```
module always_example ( );
reg clk, reset,enable,q_in,data;
always @ (posedge clk)
if (reset) begin
   data <= 0; 7
   end else if (enable) begin
    data <= q_in;
    end
endmodule
```

Sensitive list

Sensitive list

```
module always_example();
input a, b, sel;
output y;
reg y;
always  @ (a or b or sel)
begin
    y = 0;
    if (sel == 0)  begin
      y = a;
    end else
    begin
      y = b;
    end
 end
endmodule
```

Sensitive list

Sensitive list