

# Modeling Digital Systems with VHDL and Verilog

Reference: Roth & John text – Chapter 2  
Michael Smith text – Chapters 8 & 10

# Hardware Description Languages

---

- ▶ **VHDL** = VHSIC Hardware Description Language
  - (VHSIC = Very High Speed Integrated Circuits)
  - ▶ Developed by DOD from 1983 – based on ADA language
  - ▶ IEEE Standard 1076-1987/1993/2002/2008
  - ▶ Gate level through system level design and verification
  
- ▶ **Verilog** – created in 1984 by Phil Moorby and Prabhu Goel of Gateway Design Automation (merged with Cadence)
  - ▶ IEEE Standard 1364-1995/2001/2005
  - ▶ Based on the C language
  - ▶ IEEE P1800 “System Verilog” in voting stage & will be merged with 1364
  - ▶ Primarily targeted for design of ASICs (Application-Specific ICs)

# Related VHDL Standards

---

- ▶ **1076.1**–1999: VHDL-AMS (Analog & Mixed-Signal Extensions)
- ▶ **1076.2**–1996: Std. VHDL Mathematics Packages
- ▶ **1076.3**-1997: Std. VHDL Synthesis Packages
- ▶ **1076.4**-1995: Std. VITAL Modeling Specification (VHDL Initiative Towards ASIC Libraries)
- ▶ **1076.6**-1999: Std. for VHDL Register Transfer Level (RTL) Synthesis
- ▶ **1164**-1993: Std. Multi-value Logic System for VHDL Model Interoperability

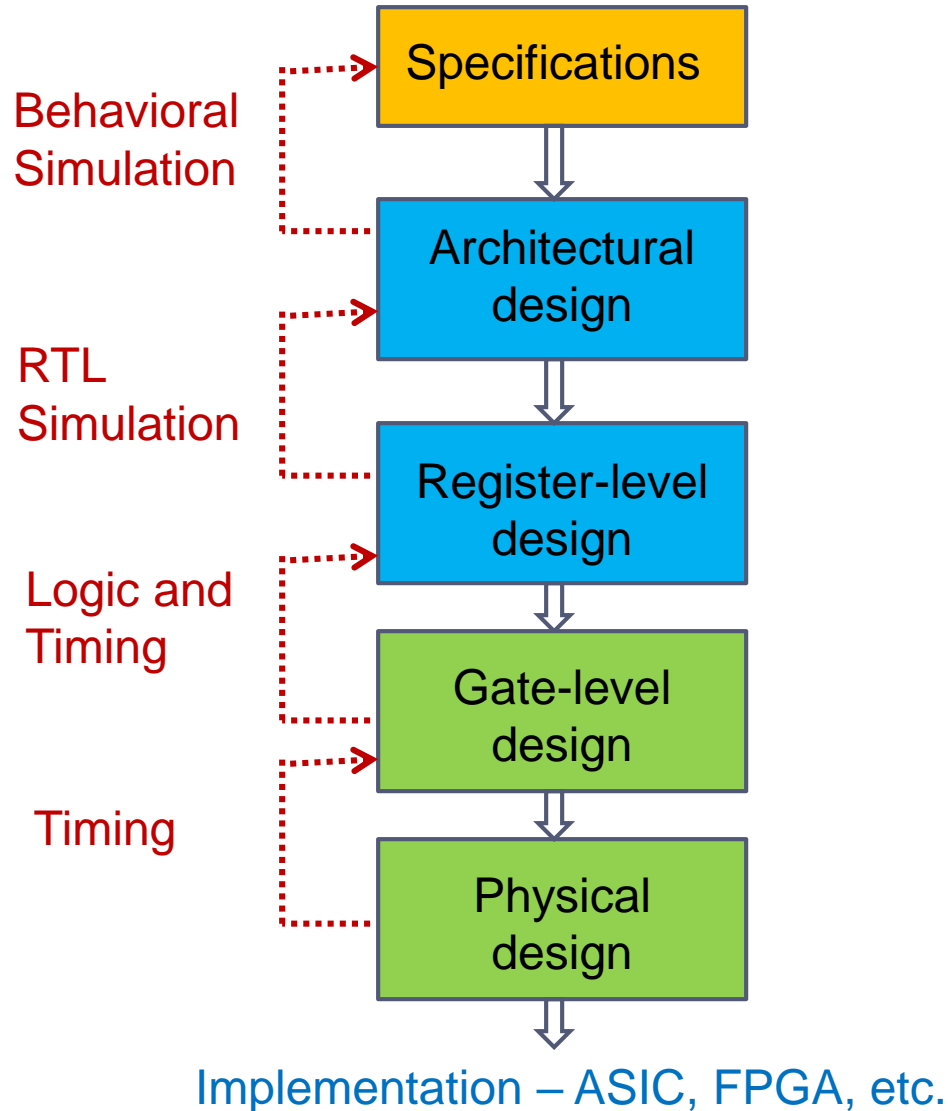
# HDLs in Digital System Design

---

- ▶ **Model** and **document** digital systems
  - ▶ **Behavioral** model
    - ▶ describes I/O responses & behavior of design
  - ▶ **Register Transfer Level (RTL)** model
    - ▶ data flow description at the register level
  - ▶ **Structural** model
    - ▶ components and their interconnections (netlist)
    - ▶ hierarchical designs
- ▶ **Simulation** to verify circuit/system design
- ▶ **Synthesis** of circuits from HDL models
  - ▶ using components from a technology library
  - ▶ output is primitive cell-level netlist (gates, flip flops, etc.)

# Typical Product Development & Design Verification Cycle Using HDLs

---



# Benefits of HDLs

---

- ▶ Early design verification via high level design verification
- ▶ Evaluation of alternative architectures
- ▶ Top-down design (*w/synthesis*)
- ▶ Reduced risk to project due to design errors
- ▶ Design capture (*w/synthesis; independent of implementation*)
- ▶ Reduced design/development time & cost (*w/synthesis*)
- ▶ Base line testing of lower level design representations
  - ▶ Example: gate level or register level design
- ▶ Ability to manage/develop complex designs
- ▶ Hardware/software co-design
- ▶ Documentation of design (*depends on quality of designer comments*)

# Designer concerns about HDLs

---

- ▶ Loss of control of detailed design
- ▶ **Synthesis may be inefficient**
- ▶ Quality of synthesis varies between synthesis tools
- ▶ Synthesized logic might not perform the same as the HDL
- ▶ Learning curve associated with HDLs & synthesis tools
- ▶ Meeting tight design constraints (time delays, area, etc.)

# Design Space Issues

---

- ▶ Area (chip area, how many chips, how much board space)
- ▶ Speed/performance
- ▶ Cost of product
- ▶ Production volume
- ▶ Design time (to meet market window & development cost)
- ▶ Risk to project (working, cost-effective product on schedule)
- ▶ Reusable resources (same circuit - different modes of operation)
- ▶ Implementation technology (ASIC, FPGA, PLD, etc.)
- ▶ Technology limits
- ▶ Designer experience
- ▶ CAD tool availability and capabilities



# DoD requirements on VHDL in mid 80s:

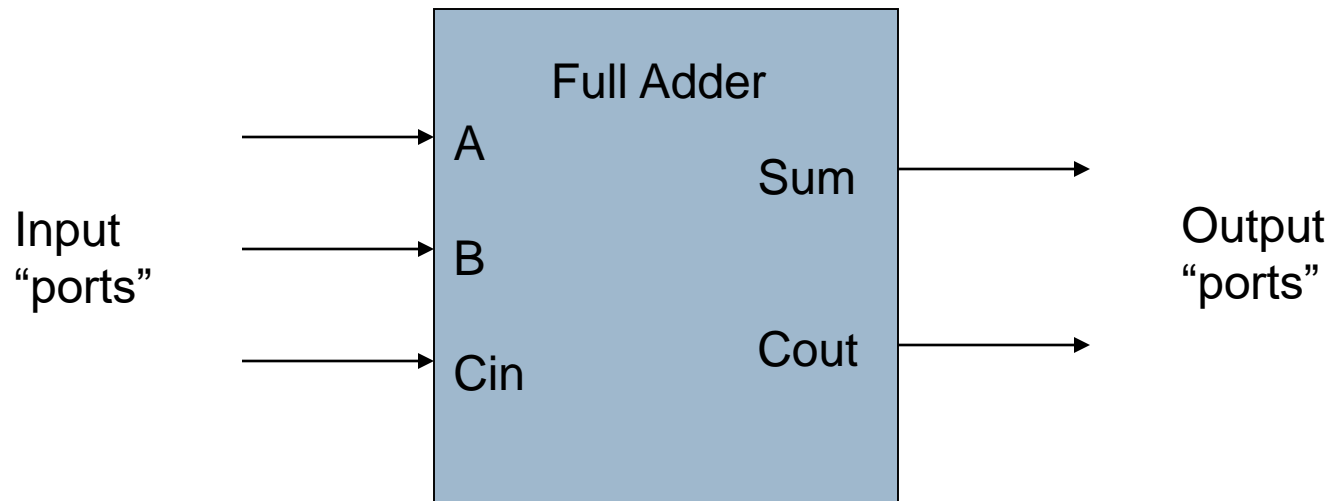
---

- ▶ Design & description of hardware
- ▶ Simulation & documentation (*with designer comments*)
- ▶ Design verification & testing
- ▶ Concurrency to accurately reflect behavior & operation of hardware (*all hardware operates concurrently*)
  - ▶ as a result, all VHDL simulation is event-driven
- ▶ Hierarchical design – essential for efficient, low-risk design
- ▶ Library support – for reuse of previously verified components
- ▶ Generic design - independent of implementation media
- ▶ Optimize - for area and/or performance
- ▶ Timing control – to assign delays for more accurate simulation
- ▶ Portability between simulators & synthesis tools (*not always true*)

# Anatomy of a VHDL model

---

- ▶ “Entity” describes the **external** view of a component
- ▶ “Architecture” describes the **internal** behavior and/or structure of the component
- ▶ Example: *1-bit full adder*



This view is captured by the VHDL “entity” (next slide)

# Example: 1-Bit Full Adder

```
entity full_add1 is                                     (keywords in green)
  port (
    a:    in  bit;  -- I/O ports
    b:    in  bit;  -- addend input
    cin:  in  bit;  -- augend input
    sum:  out bit;  -- carry input
    cout: out bit); -- sum output
end full_add1 ;                                       -- carry output
```

I/O Port Declarations

Comments follow double-dash

Signal name

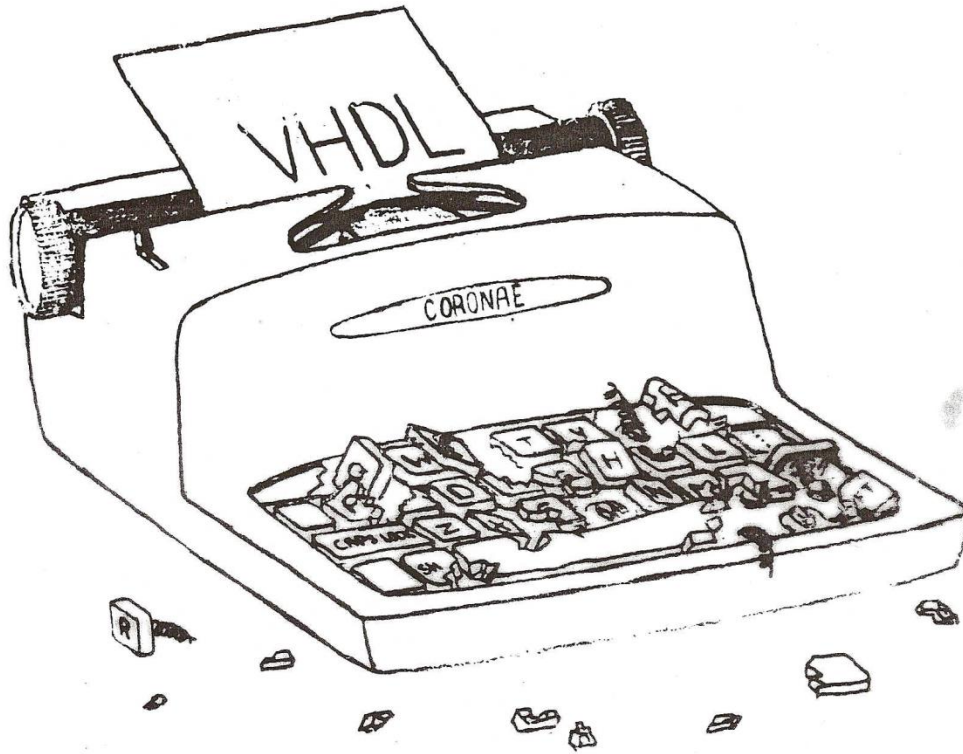
Signal direction (mode)

Signal type

# Port Format - Name: Direction Signal\_type;

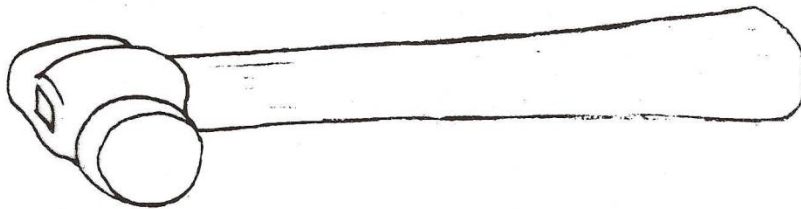
---

- ▶ **Direction**
  - ▶ **in** - driven into the entity by an external source  
*(can read, but not drive, within the architecture)*
  - ▶ **out** - driven from within the entity  
*(can drive, but not read, within the architecture)*
  - ▶ **buffer** – like “out” but can read and drive
  - ▶ **inout** – bidirectional; signal driven **both** by external source and within the architecture  
*(can read or drive within the architecture)*
- ▶ **Signal\_type**: any scalar or aggregate signal data type



*Driving signal types  
must match  
driven signal type*

VHDL: a strongly typed language



# Built-in Data Types

---

- ▶ **Scalar (single-value) signal types:**
  - ▶ **bit** – values are '0' or '1'
  - ▶ **boolean** – values are **TRUE** and **FALSE**
  - ▶ **integer** - values  $[-2^{31} \dots +(2^{31}-1)]$  on 32-bit host
- ▶ **Aggregate of multiple scalar signal types:**
  - ▶ **bit\_vector** – array of bits;
    - must specify “range” of elements

## Examples:

signal b: bit\_vector(7 downto 0);

signal c: bit\_vector(0 to 7);

b <= c after 1 ns; --drive b with value of c

c <= "01010011"; --drive c with constant value

# 8-bit adder - entity

---

-- Internally - cascade 8 1-bit adders for 8-bit adder

entity Adder8 is

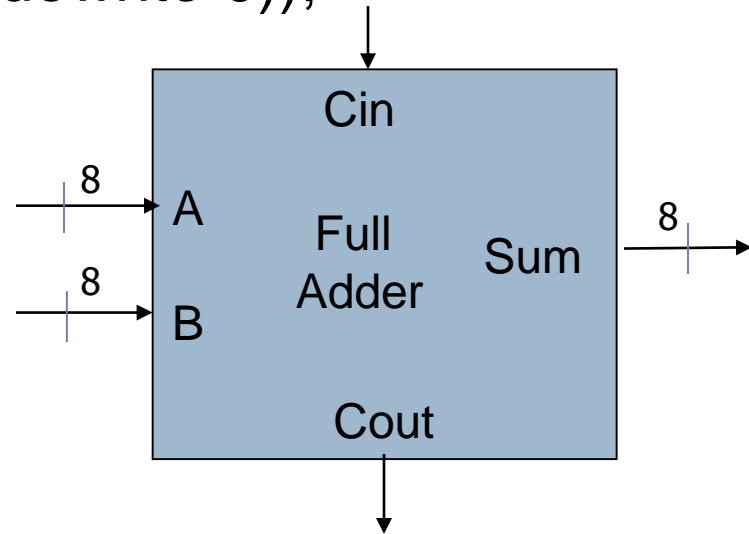
port (A, B: in BIT\_VECTOR(7 downto 0); -- or (0 to 7)

Cin: in BIT;

Cout: out BIT;

Sum: out BIT\_VECTOR(7 downto 0));

end Adder8;



# IEEE std\_logic\_1164 package

---

- IEEE std\_logic\_1164 package defines nine logic states for signal values
- models states/conditions that cannot be represented with the BIT type
- VHDL “package” similar to a C “include” file

package Part\_STD\_LOGIC\_1164 is

```
type STD_ULOGIC is ( 'U', -- Uninitialized/undefined value
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0 (drive to GND)
                    '1', -- Forcing 1 (drive to VDD)
                    'Z', -- High Impedance (floating, undriven, tri-state)
                    'W', -- Weak Unknown
                    'L', -- Weak 0 (resistive pull-down)
                    'H', -- Weak 1 (resistive pull-up)
                    '-' -- Don't Care (for synthesis minimization)
                    );
```

subtype STD\_LOGIC is resolved STD\_ULOGIC; --see next slide

type STD\_LOGIC\_VECTOR is array (NATURAL range <>) of STD\_LOGIC;

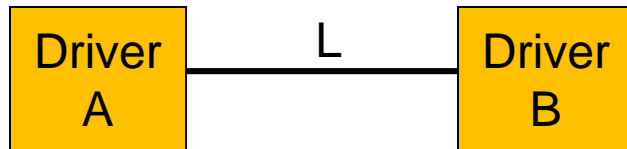
STD\_LOGIC/STD\_LOGIC\_VECTOR generally used instead of BIT/BIT\_VECTOR



# Bus resolution function

`std_logic` includes a “bus resolution function” to determine the signal state where there are multiple drivers

```
function resolved (s : STD_ULOGIC_VECTOR) return  
STD_ULOGIC;
```



Driver A:  
L <= A;

Driver B:  
L <= B;

Driver A  
value

Driver B value

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

Resolved  
Bus  
Values  
for signal  
L

## Example: 1-Bit Full Adder (VHDL)

---

```
library ieee;                                --supplied library
use ieee.std_logic_1164.all;                 --package of
definitions

entity full_add1 is
  port (                                       -- I/O ports
    a:      in  std_logic;                     -- addend input
    b:      in  std_logic;                     -- augend input
    cin:    in  std_logic;                     -- carry input
    sum:    out std_logic;                     -- sum output
    cout:   out std_logic);                   -- carry output
end full_add1 ;
```

# Architecture defines function/structure

---

**ARCHITECTURE** *architecture\_name* **OF** *entity\_name* **IS**

- data type definitions (ie, states, arrays, etc.)
- internal signal declarations
- component declarations
- function and procedure declarations

**BEGIN**

- *behavior of the model is described here using:*
  - component instantiations
  - concurrent statements
  - processes

**END;** *--optionally: END ARCHITECTURE architecture\_name;*

# Architecture defines function/structure

---

entity Half\_Adder is

port (X, Y : in STD\_LOGIC := '0';

Sum, Cout : out STD\_LOGIC); -- formals

end;

-- behavior specified with logic equations

architecture Behave of Half\_Adder is

begin

Sum <= X xor Y; -- use formals from entity

Cout <= X and Y; -- “operators” are not “gates”

end Behave;

--operators and,or,xor,not applicable to bit/std\_logic signals

# Half Adder (Verilog)

---

```
// module half_adder (Sum, Cout, X, Y); // Verilog 1995  
syntax
```

```
// output Sum, Cout;
```

```
// input X, Y;
```

```
module half_adder ( output Sum, Cout, input X,  
Y); // Verilog 2001, 2005 syntax
```

```
    assign Sum=X^Y; // ^ is the XOR operator
```

```
    assign Cout=X&Y; // & is the AND operator
```

```
endmodule
```

# 1-Bit Full Adder (VHDL)

---

```
library ieee;                                --supplied library
use ieee.std_logic_1164.all;                 --package of
definitions

entity full_add1 is
  port (
    a:      in  std_logic;                    -- addend input
    b:      in  std_logic;                    -- augend input
    cin:    in  std_logic;                    -- carry input
    sum:    out std_logic;                    -- sum output
    cout:   out std_logic);                  -- carry output
end full_add1 ;
```

# Full Adder

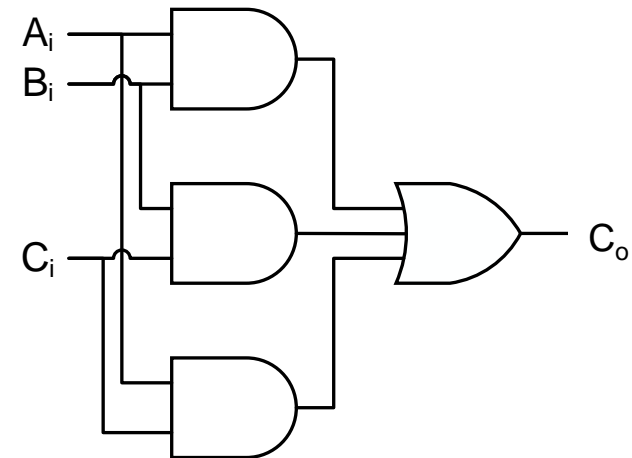
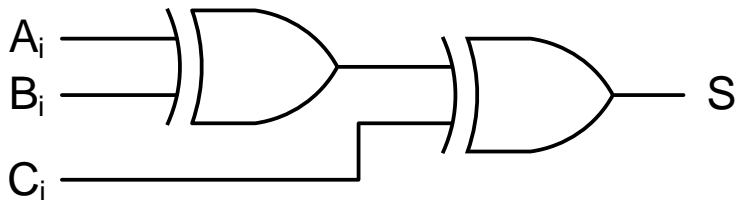
A <sub>i</sub>	B <sub>i</sub>	C <sub>i</sub>	S	C <sub>o</sub>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A <sub>i</sub> B <sub>i</sub> \ C <sub>i</sub>	0	1
00	0	1
01	1	0
11	0	1
10	1	0

A <sub>i</sub> B <sub>i</sub> \ C <sub>i</sub>	0	1
00	0	0
01	0	1
11	1	1
10	0	1

$$\begin{aligned}
 S &= A'_i B'_i C_i + A'_i B_i C'_i + A_i B_i C'_i + A_i B'_i C_i \\
 &= (A'_i B_i + A_i B'_i) C'_i + (A'_i B'_i + A_i B_i) C_i \\
 &= (A'_i B_i + A_i B'_i) C'_i + (A'_i B_i + A_i B'_i)' C_i \\
 &= A_i \oplus B_i \oplus C_i
 \end{aligned}$$

$$C_o = A_i B_i + B_i C_i + A_i C_i$$



# Full adder behavioral architectures (no circuit structures specified)

---

-- behavior expressed as logic equations

architecture dataflow of full\_add1 is

begin

```
sum <= a xor b xor cin;
```

```
cout <= (a and b) or (a and cin) or (b and cin);
```

end;

-- equivalent behavior, using an internal signal

architecture dataflow of full\_add1 is

```
signal x1: std_logic; -- internal signal
```

begin

```
x1 <= a xor b; -- drive x1
```

```
sum <= x1 xor cin; -- reference x1
```

```
cout <= (a and b) or (a and cin) or (b and cin);
```

end;



# 1 bit Full Adder (Verilog)

---

```
//module full_adder (sum, cout, a, b, cin); // Verilog 1995  
syntax
```

```
//output sum, cout;
```

```
//input a, b, cin;
```

```
module full_adder ( output sum, cout, input a, b, cin); //  
Verilog 2001, 2005 syntax
```

```
    assign sum=a ^ b ^ cin;
```

```
    assign cout=a & b | a & cin | b & cin; // | is the OR operator
```

```
endmodule
```

## Example: 8-bit full adder (VHDL)

---

```
library ieee; -- supplied library
use ieee.std_logic_1164.all; -- package of
    definitions

entity full_add8 is -- 8-bit inputs/outputs
    port ( a:    in std_logic_vector(7 downto 0);
          b:    in std_logic_vector(7 downto 0);
          cin:  in std_logic;
          sum:  out std_logic_vector(7 downto 0);
          cout: out std_logic);
end full_add8 ;
```

Can use (0 to 7) if desired.

## Example: 8-bit full adder (Verilog)

---

```
//module full_add8 (sum, cout, a, b, cin); // Verilog  
1995 syntax
```

```
//output [7:0] sum;
```

```
//output cout;
```

```
//input [7:0] a, b;
```

```
//input cin;
```

```
module full_add8
```

```
( output [7:0] sum,
```

```
output cout,
```

```
input [7:0] a, b,
```

```
input cin); // Verilog 2001, 2005 syntax
```

# Event-driven simulation (VHDL)

---

- ▶ Signal “event” = change in signal value at a specified time

`k <= b and c after 1 ns;`

- ▶ Creates a “driver” for signal k, with scheduled events
  - ▶ “Event” = (value, time) pair
  - ▶ One driver per signal (unless a bus resolution function provided)
- ▶ Data types must match (**strongly typed**)
- ▶ Delay, from current time, can (**optionally**) be specified, as above
- ▶ If no delay specified, infinitesimally-small delay “delta” inserted

`k <= b and c;`

*(To reflect that signals cannot change in zero time!)*

- ▶ **Delays are usually unknown in behavioral models and therefore omitted**

# Concurrent Statements and Event-Driven Simulation

---

- ▶ Statements appear to be evaluated **concurrently**
  - ▶ To model behavior of actual hardware elements
- ▶ Each statement affected by a signal event at time  $T$  is evaluated
  - ▶ Time  $T$  is held constant while statements are evaluated
  - ▶ Any resulting events are “scheduled” in the affected signal driver, to occur at time  $T + \text{delay}$
  - ▶ After all statements evaluated,  $T$  is advanced to the time of the next scheduled event (among all the drivers)
  - ▶ New values do not take effect until simulation time advances to the scheduled event time,  $T + \text{delay}$

# Event-Driven Simulation Example

---

a  $\leq$  b after 1ns;

c  $\leq$  a after 1ns;

<u>Time</u>	<u>a</u>	<u>b</u>	<u>c</u>
-------------	----------	----------	----------

T	'0'	'0'	'0'
---	-----	-----	-----

- assume initial values all '0' at time T

T+1	'0'	'1'	'0'
-----	-----	-----	-----

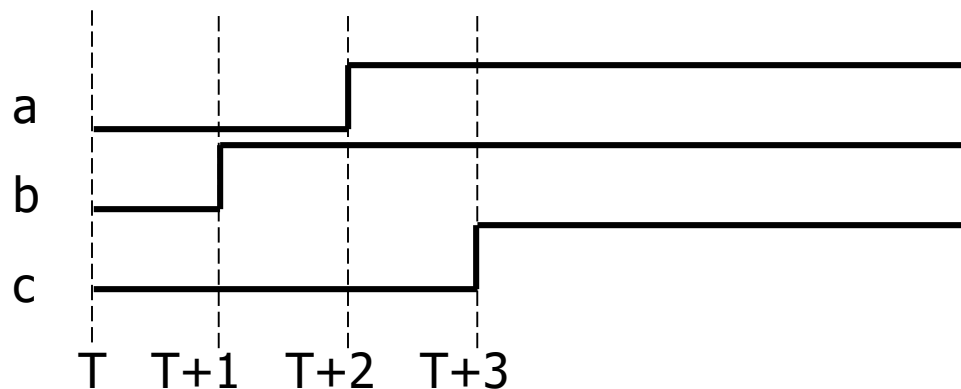
- external event changes b at time T+1

T+2	'1'	'1'	'0'
-----	-----	-----	-----

- resulting event on a

T+3	'1'	'1'	'1'
-----	-----	-----	-----

- resulting event on c



# Event-Driven Simulation Example

a <= b; -- delay  $\delta$  inserted

c <= a; -- delay  $\delta$  inserted

Time	a	b	c	
T-1	'0'	'0'	'0'	- assume initial values all '0'
T	'0'	'1'	'0'	- external event changes b at time T
T+ $\delta$	'1'	'1'	'0'	- resulting event on a after $\delta$ delay
T+2 $\delta$	'1'	'1'	'1'	- resulting event on c after 2 <sup>nd</sup> $\delta$ delay

VHDL simulators generally show time and  $\delta$  delays

