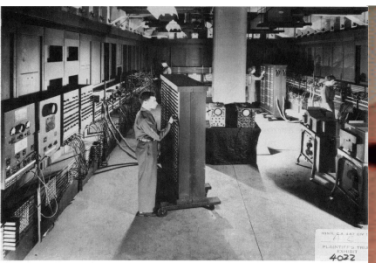
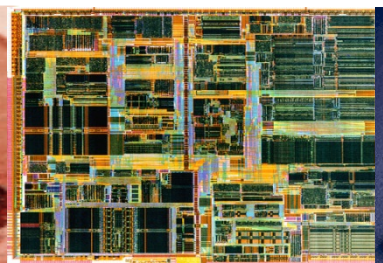
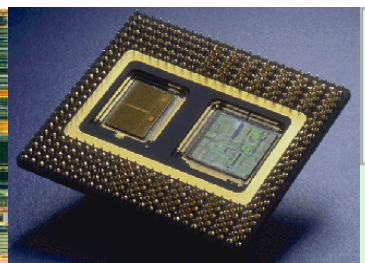
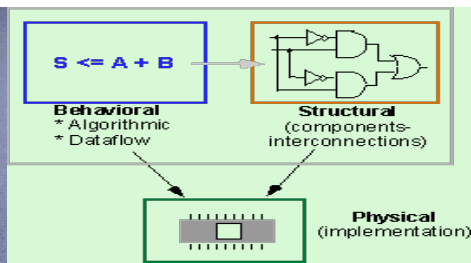
CPE 690: Introduction to VLSI Design

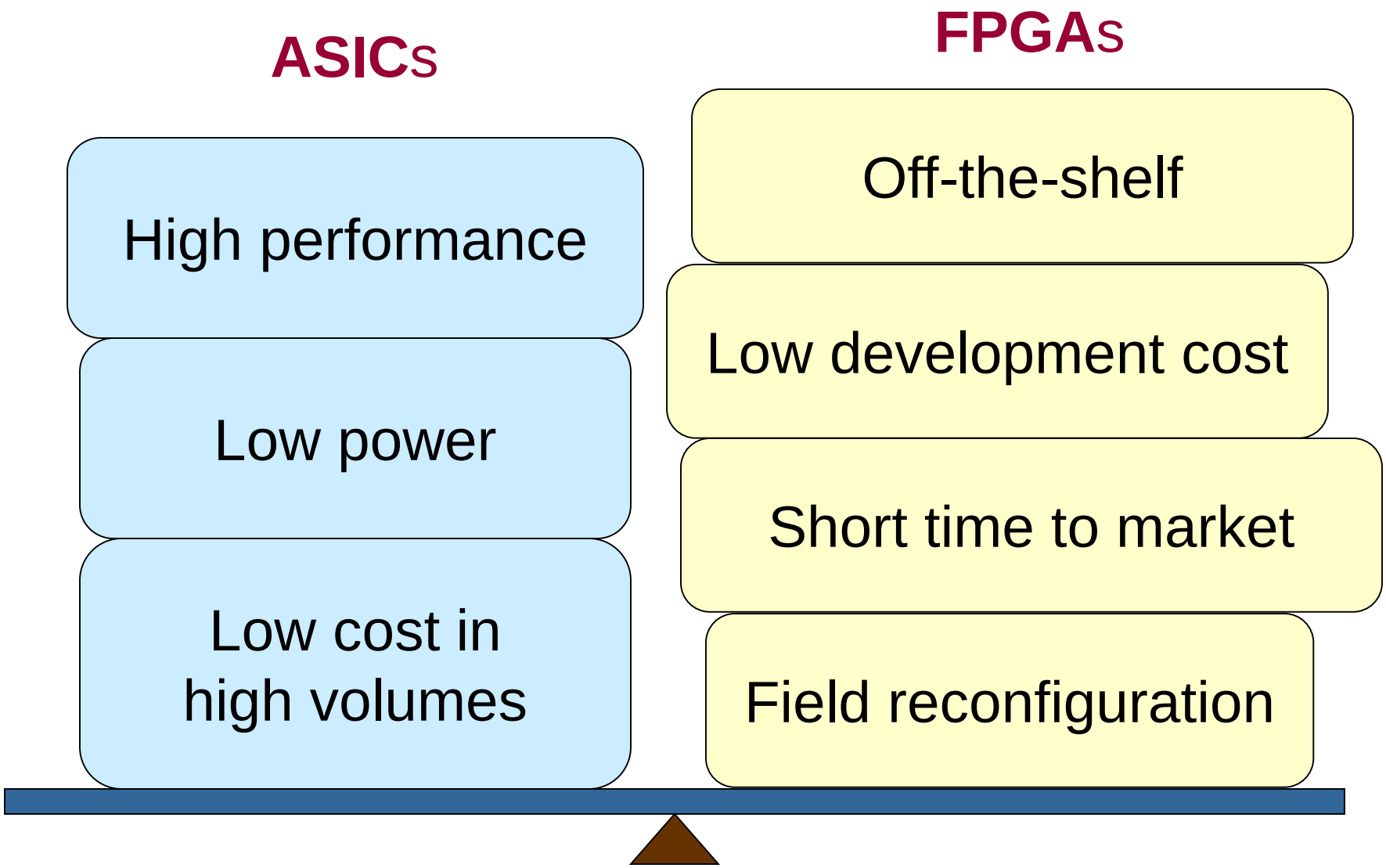
Lecture 2
FPGA Design and VHDL – Part I

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030



Two competing implementation approaches



Managing Design Complexity

To be successful, designer (or design team) must manage placement and interconnect of up to 10^8 components…

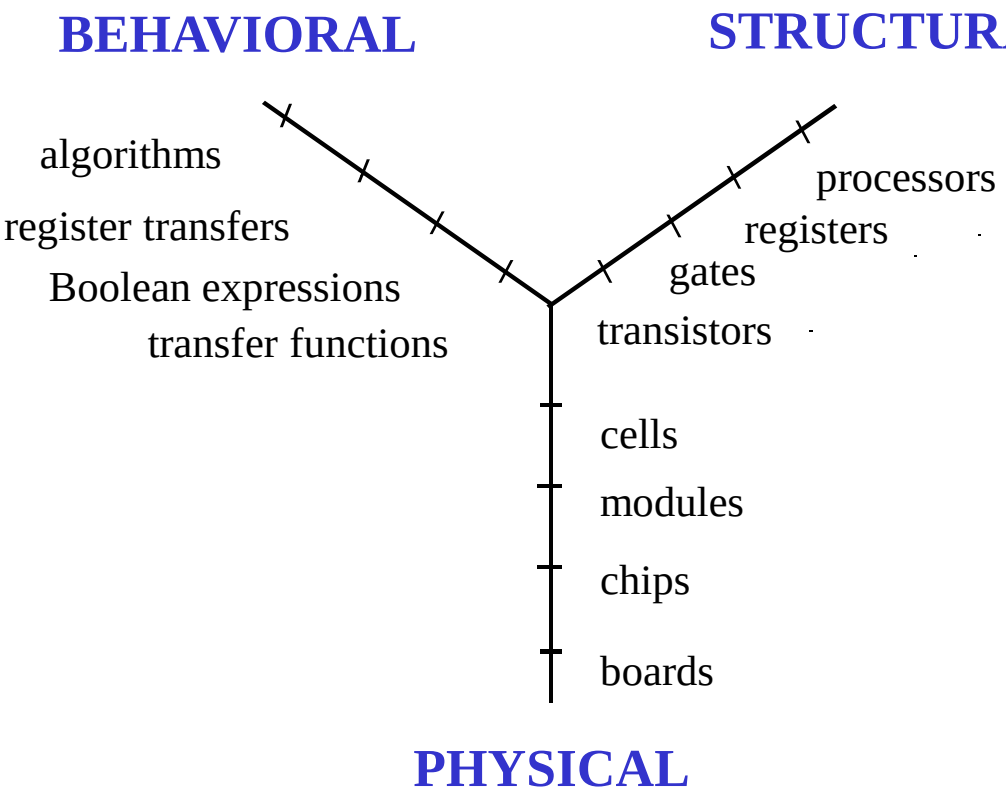
That meet the **original design specification**

i.e. function + performance

while

- Minimizing chip area (die cost)
- Maximizing yield (die cost)
- Minimizing power dissipation (battery life)
- Maximizing reliability (design margin)
- Minimizing test time (product cost)
- Minimizing design cost (**N**on **R**ecurring **E**xpense)
- Minimizing time to market (market share)

Design Space



- Three fundamentally different ways (views) of representing a design
- Each view can be formed at many different levels of abstraction (amount of detail)
- Design process is one of moving from highest behavioral level (specification) to lowest (most detailed) physical level

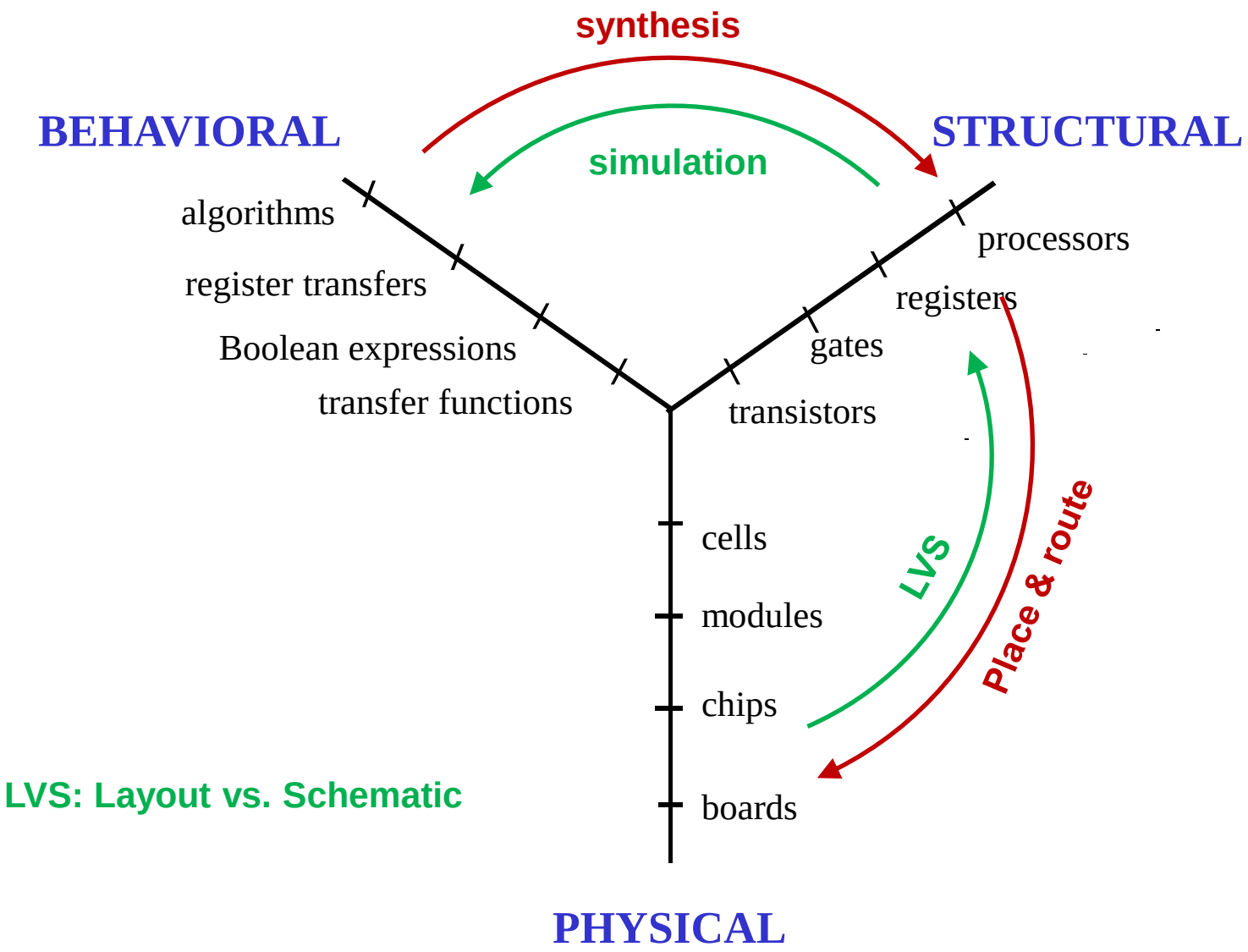
Source: D. Gajski and R. H. Kuhn

Managing Design Process

Taking a complex design from high level behavioral description (spec.) to detailed physical implementation is accomplished using:

- Hierarchy, Modularity & Regularity
  - Break design into manageable pieces
  - Pieces that have well defined functionality and simple interface
  - Pieces that can be re-used elsewhere in the hierarchy
  - Gradually refine design to greater levels of detail
- Set of computer aided design (CAD) tools that
  1. Capture design data (e.g., hardware description languages, text editors, schematic & layout editors)
  2. Translate from one representation to another (e.g., synthesis, component mapping, place & route)
  3. Verify correctness of translation (simulation, timing analysis, design rule check)
- Design Methodology
  - Recipe (or plan) of how to move from one design representation to another, which tools to use and how to rigorously verify each design step

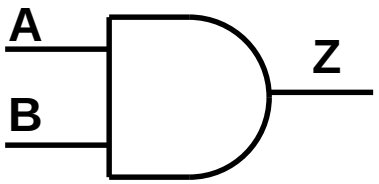
Examples of Computer Aided Design Tools



LVS: Layout vs. Schematic

History of Digital Design Specification

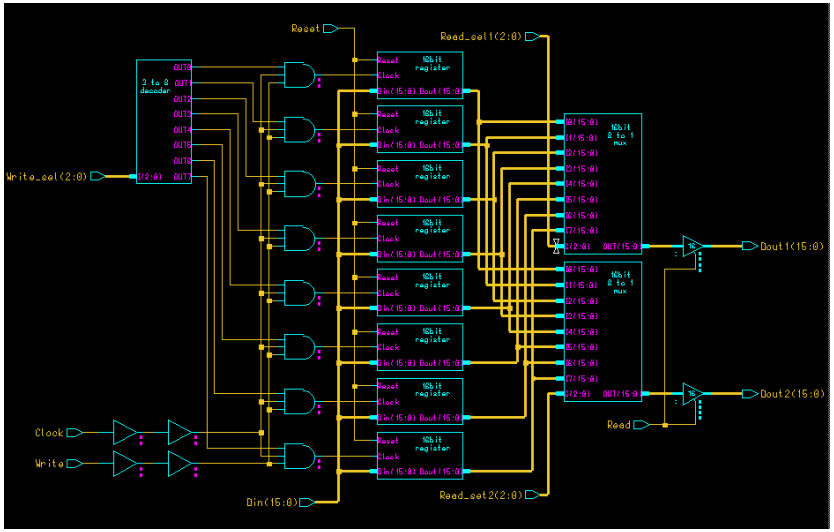
- Boolean Equations: function represented by truth tables & logic equations



Z = A.B

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

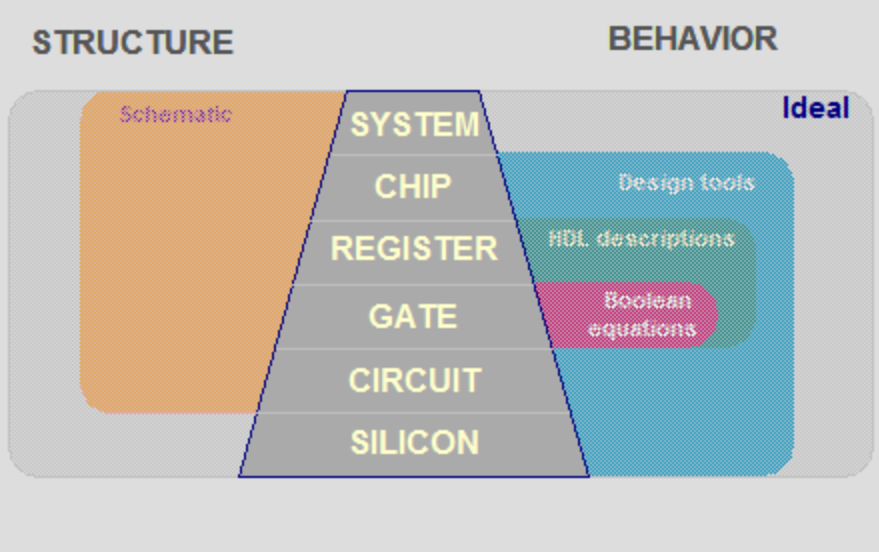
- Schematic Capture



Captures structure but behavior must be inferred

Digital Design – Hardware Description Languages

- Hardware Description Language (HDL) captures behavior and/or structure that can be compiled into simulation or physical implementation (e.g. gate array, FPGA)
- Early Hardware Description languages targeted at Register Transfer or Gate Level behavior



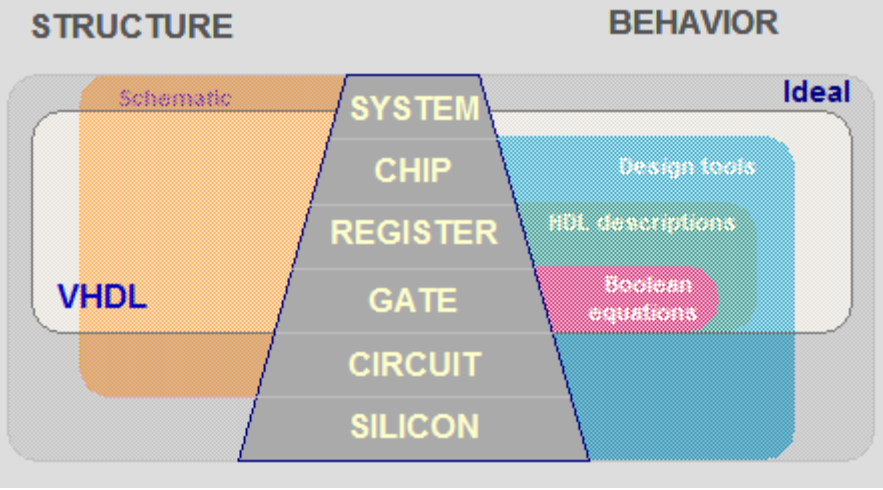
Evita Tutorial

- Different languages used at different levels of abstraction and with different tool vendors leads to lost productivity

Digital Design Specification – VHDL

- **V**HSIC **H**ardware **D**escription **L**anguage
  (VHSIC = Very High Speed Integrated Circuit)

- *Standardized language that can represent behavior and structure at many levels of abstraction*



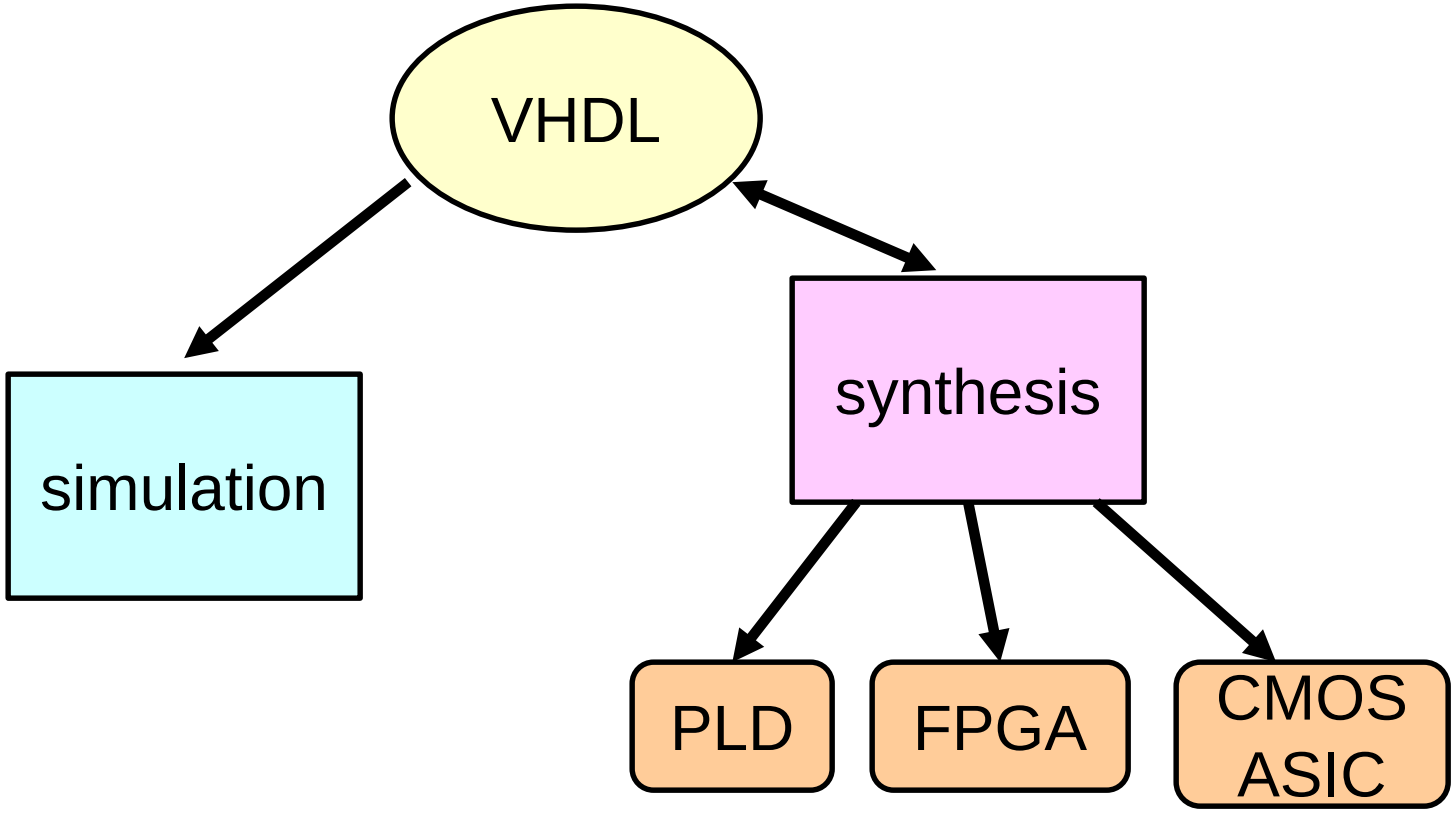
Evita Tutorial

Features of VHDL

- **VHDL can represent:**
  - **behavior** (what the system does) **or**
  - **structure** (how the components are connected) **or**
  - a combination of these.
- **VHDL can be used at different levels of abstraction:**
  - Switch level (switching behavior of transistors)
  - Gate level
  - Register transfer level (registers, multiplexers, alu's etc.)
  - High level architecture (e.g. functional behavior of microprocessor)
- **Technology independent** (ASIC, FPGA, PCB)
- **IEEE Standard** (Interoperability across tool vendors)
- **Provides executable design documentation**

Simulation & Synthesis

- VHDL “program” can be used to drive:
  - **simulation** (functional verification, performance)
  - **synthesis** (translating behavior into physical structure) **or**
  - a combination of these.

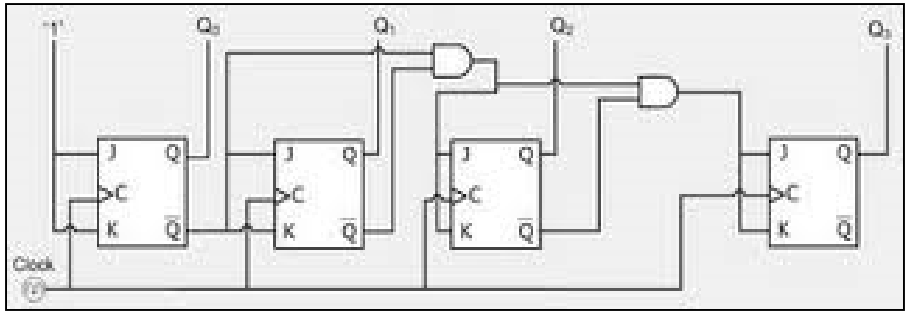


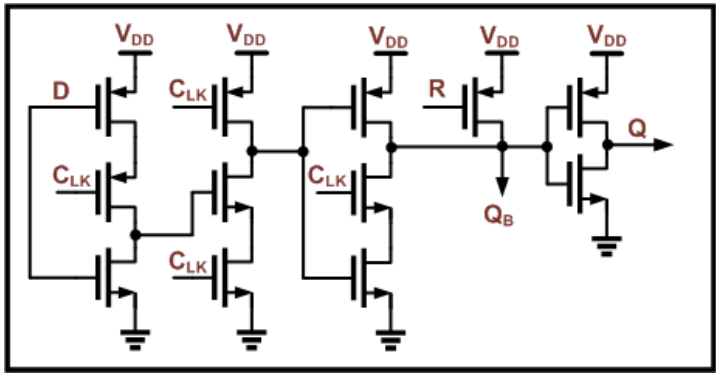
VHDL vs. Regular Programming Language

- Procedural programming languages implement an algorithm or recipe
  - for computation & data manipulation
  - essentially single sequential thread (Program Counter)
    - order of statements determines execution sequence
  - no intrinsic concept of time
  - program operates on variables
- VHDL describes a hardware system
  - from different points of view: behavior, structure, dataflow
  - can model highly concurrent operation
  - intrinsic concept of time
    - timed events determines execution sequence
  - program operates on variables and signals

Nature of Digital Systems

- *At all levels of abstraction*, electronic systems are composed of sub-systems interconnected by signals*:





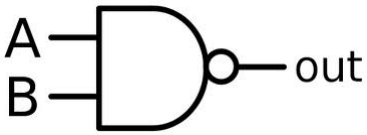
* Signals include wires, optical links & wireless links

VHDL Model of Digital Systems

- *At all levels of abstraction:*
- VHDL names and declares the interface to each (sub-)system using a programming abstraction known as an **entity**.
- VHDL models the operation (i.e. the behavior or the internal structure) of a (sub-)system using an abstraction known as an **architecture.**
- VHDL describes the (timed) information flow between (sub-)systems using an abstraction known as a **signal**.

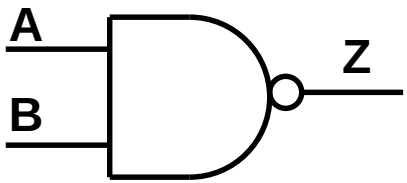
VHDL Entity

- An entity describes *the* external view of a system
- An entity specifies:
  - Name of the system
  - Parameters of the system *(get to this later)*
  - Connections to the system (external signals)



- Each of these could be an entity

Entity Example – 2-input NAND gate



name

```
entity nand2 is
port(a,b: in bit;
     z: out bit);
end entity nand2;
```

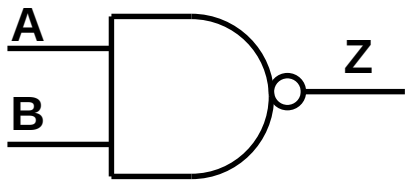
external connections

Note: words in **bold** are reserved keywords

VHDL Architecture

- An architecture describes *an* internal view of a system
- An architecture describes the behavior or internal structure of a declared entity
- There may be many architectures associated with each entity e.g:
  - structure vs behavior,
  - different levels of abstraction (gate vs RTL)
  - different timing constraints
- There is exactly one entity for each architecture

Architecture Example – 2-input NAND gate

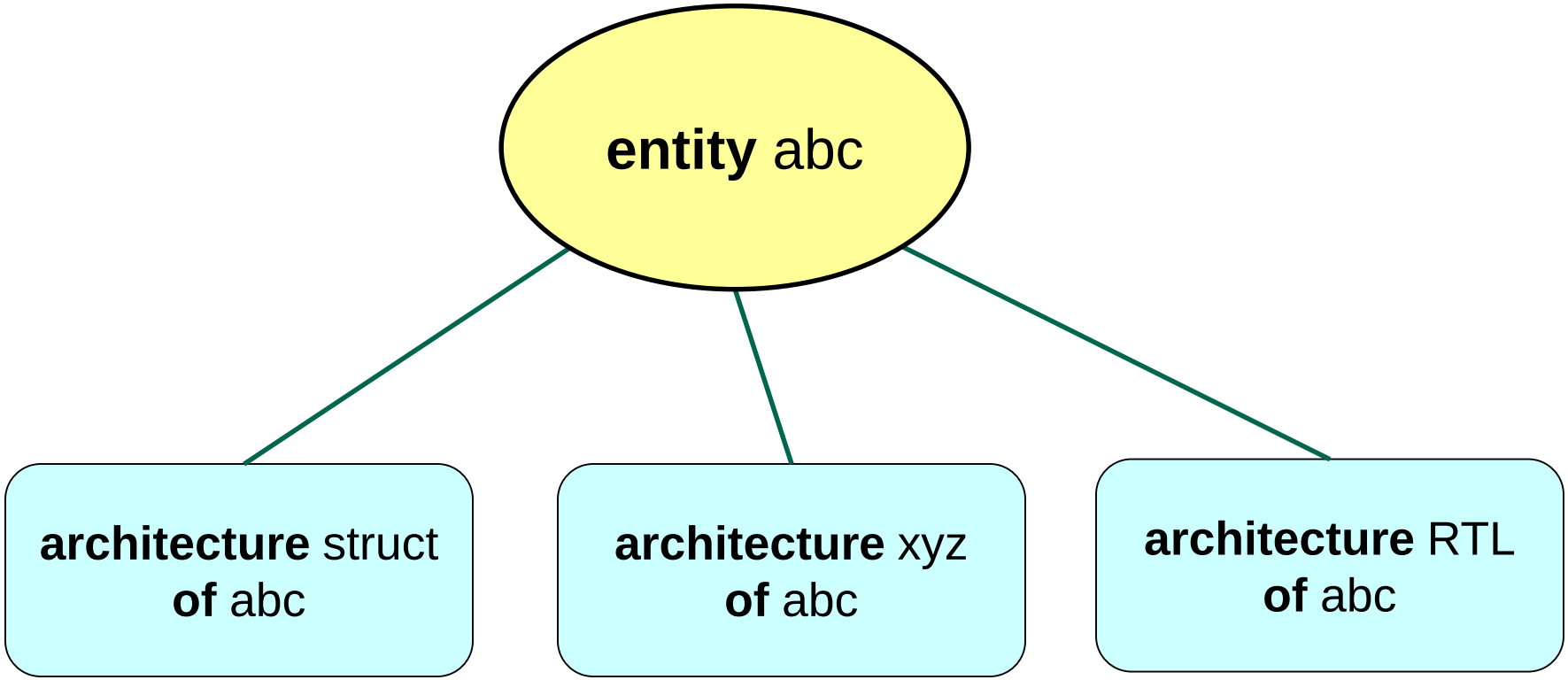



```
entity nand2 is
port(a,b: in bit;
     z: out bit);
end entity nand2;

architecture ngate of nand2 is
begin
     z <= not(a and b) after 5 ns;
end architecture ngate;
```

architecture name →

behavior

One Entity – Multiple Architectures

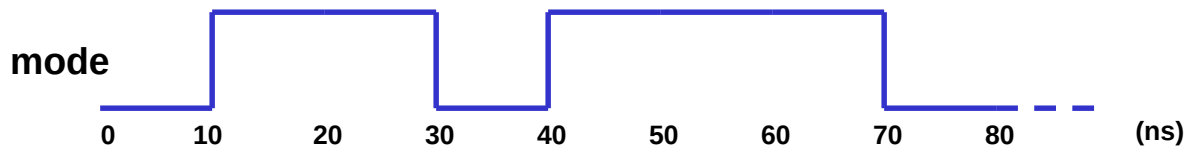


- Different architectures may describe different representations (behavior, structure) of entity at different levels of abstraction

VHDL - Signal

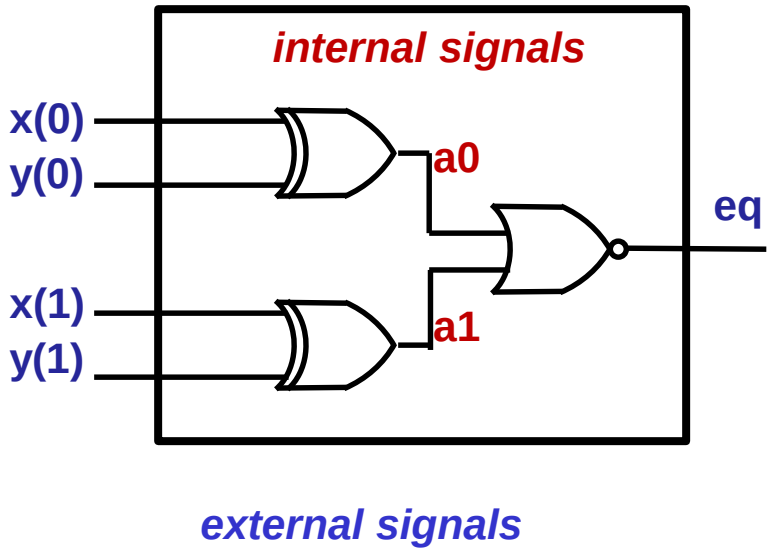
- Like conventional programming languages VHDL manipulates basic objects such as constants and variables.
- VHDL introduces a new class of object: **signal**
- Signal is a sequence of value-time pairs
- A signal will be assigned a value at a specific time
  - It will retain that value until a new value is assigned at a future point in time.

signal mode**: bit;**
mode<=‘0’,‘1’ **after** 10ns,‘0’ **after** 30ns,‘1’ **after** 40ns,‘0’ **after** 70ns;



External & Internal Signals

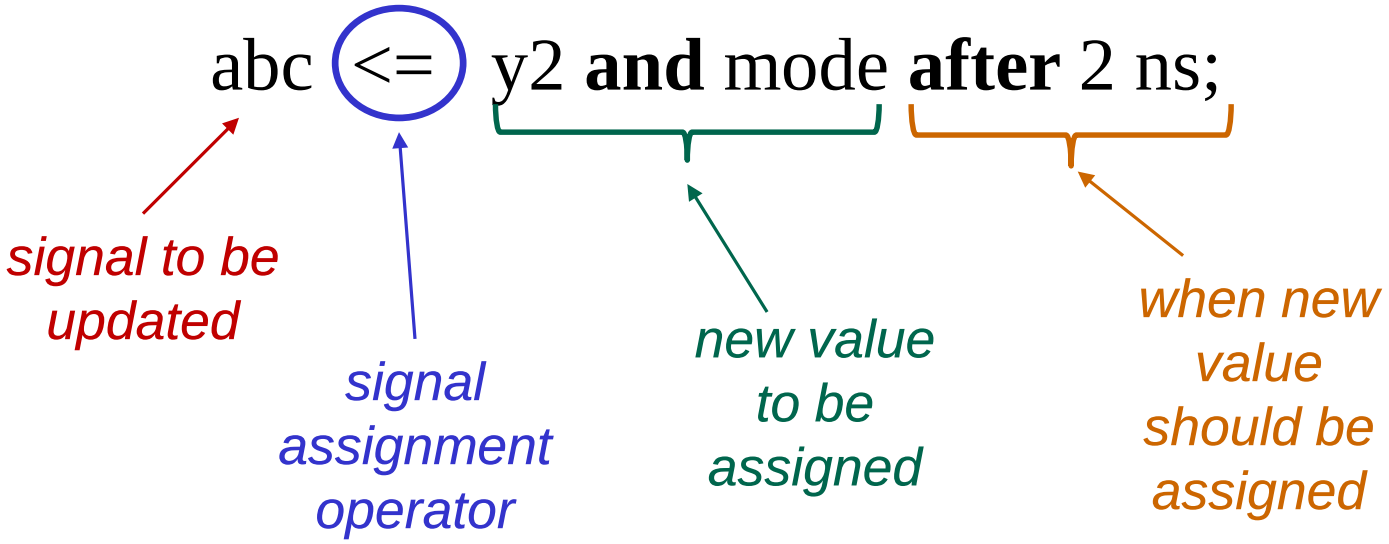
- Ports are external signals, visible inside & outside the system
- Signals declared in an architecture are internal signals
  - manipulated by programming constructs within the architecture
  - not visible outside the system




```
entity compare is
port(x,y: in bit_vector(1 downto 0);
     eq: out bit);
end entity compare;

architecture cmp_alt of compare is
signal a1, a0: bit;
begin
   a1 <=  x(1) xor y(1) after 2 ns;
   a0 <=  x(0) xor y(0) after 2 ns;
   eq <=  a0 nor a1 after 3 ns;
end architecture cmp_alt;
```

Signals Assignment Statement

- Signal assignment statement assigns a new value to a signal at a specified (future) time



- Signal assignment statements can be concurrent or sequential
  - describes when they are executed (more on this later)
- If assignment time is not specified, defaults to “after 0 ns”

Signal Types

- Like variables, all signals are typed e.g.:
  - boolean *(false, true)*
  - integer *(…,-3,-2,-1,0,1,2,3,…)*
  - character *(…,'A','B','C',…)*
  - bit *('0','1')*
- In digital circuits, often much more convenient to represent a signal as a bus, rather then individual bits.
- Type bit_vector is an array of type bit
  - bit positions are number left (msb) to right (lsb)

```
signal abus: bit_vector (0 to 7);    --  8-bit bus, abus(0) is msb
signal instr: bit_vector (15 downto 0);    --  16-bit bus, addr(15) is msb
signal opcode: bit_vector (6 downto 3);    --  4-bit bus, opcode(6) is msb

opcode <= instr(12 downto 9);        --  bit_vector used in assignment statement
```

VHDL - Packages

- Standard VHDL has limited set of types, operators, functions.
- This set can be expanded through the use of packages.
- Packages contain definitions of types, functions & procedures that can be shared by multiple designers.
- A very popular package is IEEE std_logic 1164

```
library IEEE;                          -- IEEE is a library of packages
use IEEE.std_logic_1164.all            -- load this package from IEEE library

entity half_adder is                   --  half_adder can use std_logic_1164
    •
    •
```

Std_logic type

- Standard bit type can only take on values ‘0’ or ‘1’
- In logic simulation, we often require a richer set of values
- IEEE std_logic type can take on 9 different values:
  - ‘U’ Uninitialized
  - ‘X’ Forcing unknown
  - ‘0’ Forcing 0
  - ‘1’ Forcing 1
  - ‘Z’ High impedance
  - ‘W’ Weak unknown
  - ‘L’ Weak 0
  - ‘H’ Weak 1
  - ‘-’ Don’t care

- Like bit, std_logic has vector extension e.g.:

signal addr: std_logic_vector (0 **to** 7);

signal dout: std_logic_vector (31 **downto** 16);

More on Types

- VHDL is a strongly typed language
- Every object (signal, variable, constant) has a type
- The type determines which operations can be applied to the object
- In assignment statements, LHS target must be of same type as the RHS expression
  - Type conversion functions (often defined in packages) allow explicit casting from type to another:

```
signal addr: std_logic_vector (0 to 7);
signal index: integer;
begin
    addr <= conv_std_logic_vector(index,8);
```

Some Scalar Types

- Numeric types:
  - **type** integer **is range** -2147483647 **to** +21474843647;
  - **subtype** Positive **is** integer **range** 1 **to** +21474843647;
  - **type** real_voltage **is range** 0.0 to 3.3;
- Enumerated types - explicitly listed set of allowable values
  - **type** BOOLEAN **is** (FALSE, TRUE);
  - **type** BIT **is** (‘0’,’1’);
  - **type** COLOR **is** (red, orange, yellow, green, blue, indigo, violet);
  - **type** STD_ULOGIC is (‘U’,’X’,’0’,’1’,’Z’,’W’,’L’,’H’,’-’);
- Default value is “left-hand-side” of range of object’s type
- Uninitialized objects take on default value at start of simulation

Array Types

- An array is a collection of one or more objects of the same type
- For example, *std_logic_vector* is an array of objects of type *std_logic*
- More examples:

```
type data_16 is array (15 downto 0) of std_logic;
type reg_file is array (31 downto 0) of data_16;
signal abc: reg_file;
```

note that: abc(30) is of type data_16
whereas: abc(30)(5) is of type std_logic

```
type row is array (1 to 4) of integer;
type matrix is array (1 to 4) of row;
```

- Note: If an array is indexed by signal (or variable), the index must be of type integer

Assignments to Arrays

- VHDL provides many different ways to assign values to array objects:

```
signal abc: std_logic_vector (1 to 5);
signal xx, yy: std_logic;

abc <= "01001";                          -- string literal
abc <= ('0', '1', '0', '0', '1');        -- positional
abc <= (2=>'1', 5=>'1', others => '0')   -- named

abc <= (others=>'0') ;                   -- sets all bits to '0'

abc <= (1=> xx, 4=> '0', others=> yy);   -- other signals
```

Entity Template

- An entity names a system and defines its interface (input/output signals):

entity *name_of_entity* **is**
[generic *generic_declarations*);] -- will these cover later
port (*signal_names*: **mode** *type*;
signal_names: **mode** *type*;
:
signal_names: **mode** *type*);
end [entity] [*name_of_entity***] ;**

- Entities, signals (and variables, constants, architectures etc) are named using identifiers

VHDL Basic Identifiers

- an identifier can be any length (but no spaces)
- may contain only alpha-numeric characters (A to Z, a to z, 0-9) and the underscore “_” character
- must start with a letter
- may not end with underscore “_”.
  - no successive underscores“__”.
- VHDL is case insensitive
  - (eg. And2 and AND2 or and2 are the same)
- cannot be a reserved keyword

Ports

- In an entity description, the port construct defines the input/output signals that make up the system interface
- Each port declaration has the form:

sig_name1, *sig_name2*, …,*sig_name_n* : **mode** *type* [:= *init_value*];

- Each port must have a mode and a type
- Can optionally be assigned an initial value
- Mode is one of:
  - **in** signal is an input: read-only port within the architecture
  - **out** signal is an output: write-only port within the architecture
  - **inout** signal can be an input or an output: bidirectional port
  - **buffer** signal is an output whose value can be read inside the entity's architecture

Port Examples

```
entity example is
        port ( a,b: in std_logic:='0';
        c: in integer:= 15;
        x,y: out std_logic;
        z: out std_logic_vector (7 downto 0);
        q: buffer integer);
end entity example;

architecture exarc1 of example is
signal s1, s2: std_logic; signal k: integer;
begin
        s1 <=  a and b;                  -- read in ports
        y <= s1 nor a;                   -- assign out port
        q <= c + 3;                      -- assign buffer port
        k <= q – 1;                      -- read buffer port
        s2 <= a xor y;                   -- ERROR – cannot read out port
        b <= s2 and s1;                  -- ERROR – cannot assign in port
        z <= c or a;                     -- ERROR – type mismatch
end architecture exarc1;
```

Comments

- A comment line in VHDL is represented by two adjacent hyphens “--”.
- A comment extends from “--” to the end of the line.
- Can appear anywhere within a description;

Examples:

```
--  The following entity is a 32-bit ALU
--  Designer: Fred Bloggs 10/14/08

A <= B and C;   -- A comment explaining this operation

-- X <= Y-15;     -- Commenting out an operation
```

Architecture Template

- An architecture of an entity is one representation of the internal behavior and/or structure of the entity

architecture *arch_name* **of** e*ntity_name* **is**

-- Declarations

-- components declarations
-- signal declarations
-- constant declarations
-- function declarations
-- procedure declarations
-- type declarations

begin

-- Architecture body:

--Statements that describe behavior and/or structure

end [architecture] *arch_name***;**

Architecture Body

- The body of an architecture specifies behavior and/or structure using any of the following modeling styles:

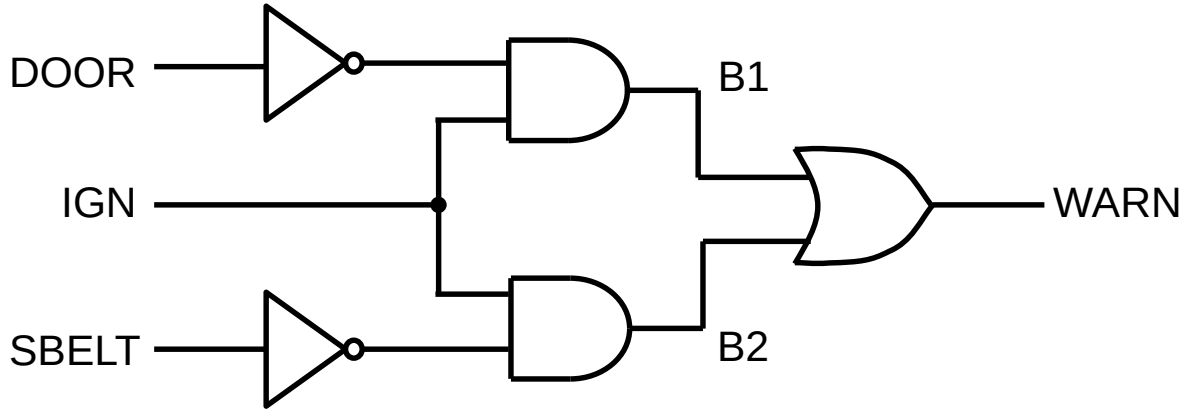
1. A set of concurrent assignment statements (to represent dataflow)

2. A set of sequential assignment statements (to represent behavior)

3. A set of interconnected components (to represent structure)

4. Any combination of the above three

Putting it All Together…




```
library IEEE;
use IEEE.std_logic_1164.all;

entity sb_alarm is
      port ( door, ign, sbelt:in std_logic;
      warn: out std_logic);
end entity sb_alarm;
```

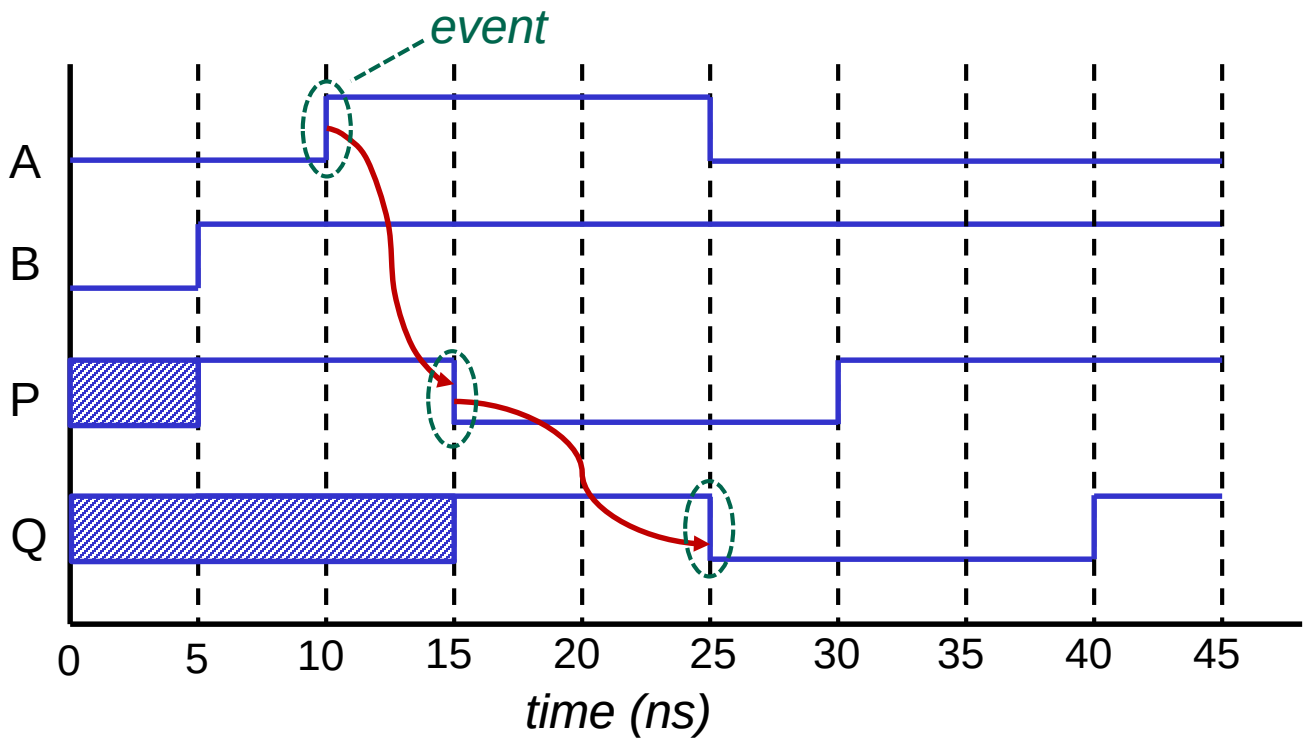
```
architecture gates of  sb_alarm is
signal b1, b2: std_logic;
begin
      b1<= (not door) and ign after 7 ns;
      b2<=(not sbelt) and ign after 7 ns;
      warn <= b1 or b2 after 9 ns;
end architecture gates;
```

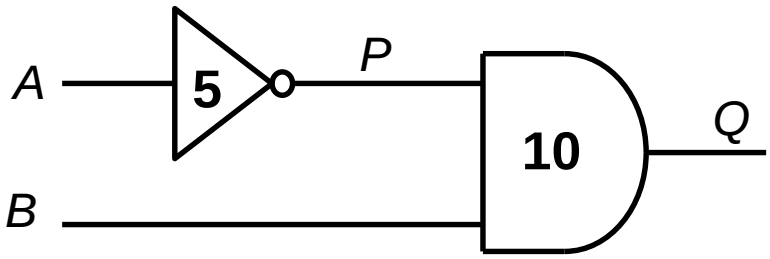
Dataflow Modeling

Concurrent Operation

- Conventional programming languages operate on variables with no intrinsic concept of time
  - Execution order is determined by order of programming statements (+ branch, subroutine calls etc.)
  - Only one operation is happening at a time
- Hardware is concurrent
  - All components execute in parallel
  - In digital circuits, output signals change in response to changes in input signals
  - When a signal changes, we say that an event occurs on that signal

Events in Digital Circuits





P <= **not** A **after** 5 ns;
Q <= P **and** B **after** 10 ns;

Concurrent Signal Assignment Statements

- Signal assignment statements in the body of the architecture are called concurrent signal assignment statements (CSA's), e.g.:

```
architecture abc of xyz is
…
begin
    …
    a <= b and c after 10 ns;
    …
end architecture abc;
```

- This CSA will be executed whenever an event occurs on *b* or *c*
  - Suppose b changes at time 100ns.
  - The value (b **and** c) will be calculated
  - This new value will be assigned to *a* at (100ns + 10ns) = 110ns

CSA Execution triggered by Events

```
a <= b and c after 10 ns;
x <= a or c after 20 ns;
```

is the same as:

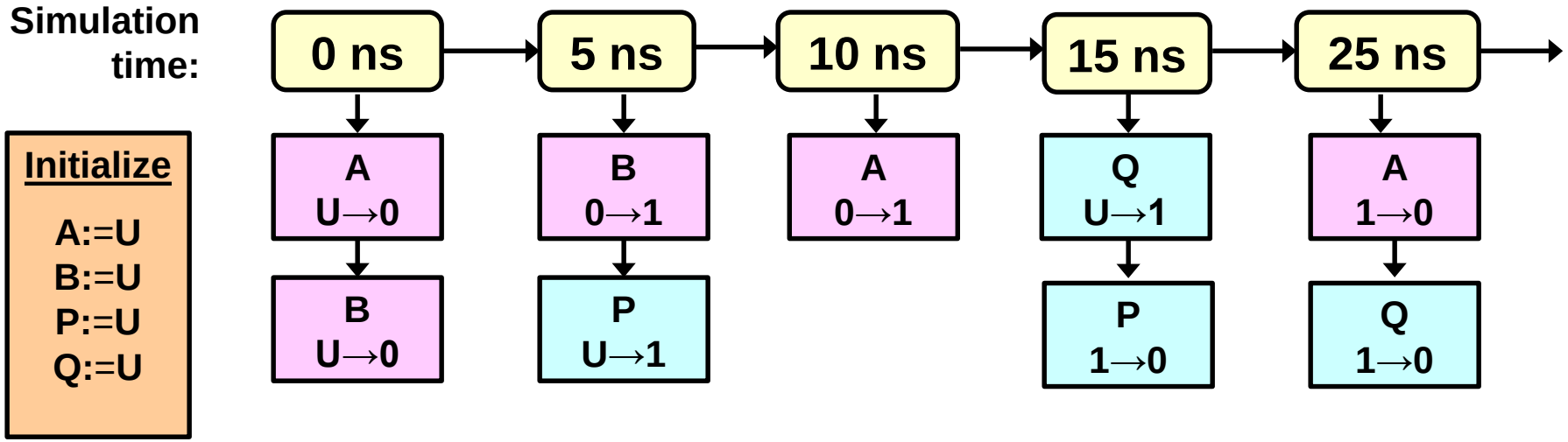
```
x <= a or c after 20 ns;
a <= b and c after 10 ns;
```

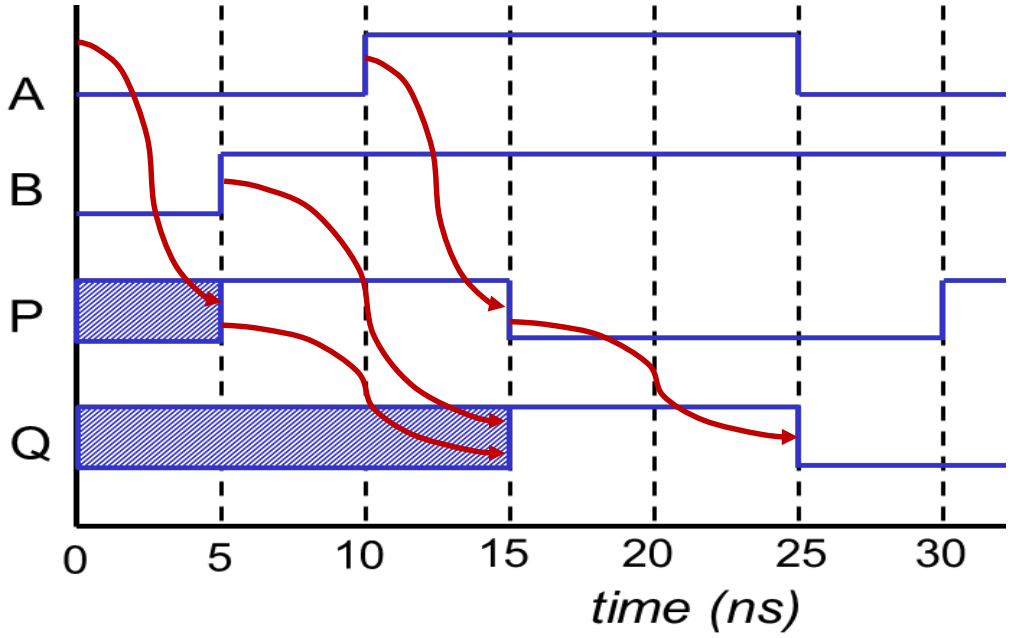
- order of concurrent signal assignments is not important
  - An event on c at 100ns may lead to an event on a at 110ns and an event on x at 120ns
  - An event on a at 110 ns may, in turn, lead to a second event on x at 130 ns
  - Note that a *signal assignment only generates a new event if the value of the signal changes*

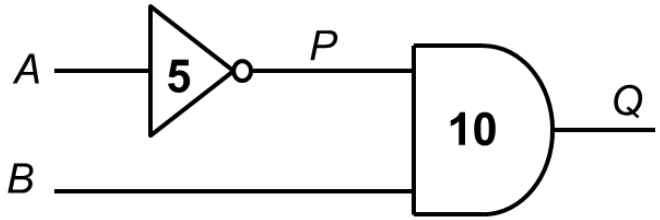
Event Driven Simulation Model

- VHDL is simulated using event driven simulation
- Simulator maintains variable current time
- Simulator also maintains event queue
  - Time ordered list of all future events
  - Each event consists of:
    - name of signal
    - change in value
    - time at which the change takes place
- Operation of simulator:
  1. Advance *current time* to earliest event on event queue
  2. Adopt all events at *current time* (i.e. update signals)
  3. Execute simulation models affected by these events
  4. Schedule future events generated by these models
  5. Repeat until queue is empty or time-limit reached

Event Driven Simulation of Simple Circuit







Constants

- Only data object we have considered so far is signal
- VHDL statements operate on 4 basic classes of objects
  - Signals
  - Constants
  - Variables (later)
  - Files (later)
- Constants are objects that are assigned a value once, when declared, and do not change their value during simulation.
- Constants are useful for creating more readable design descriptions, and they make it easier to change the design at a later time.
- Examples:
  - **constant** delay: **time** := 10 ns;
  - **constant** bus_width : **integer** := 8;

Concurrent Signal Assignment

- Simple CSA's execute whenever an event occurs on a signal on the RHS of the assignment statement

  z <= (a **and** (**not** b)) **xor** (c **or** (a **and** (**not d**))) **after** 3 **ns;**

  target-signal <= *waveform-elements*;

- Useful for describing gate level combinational logic
  - (when output is a function of current input values only)

- Not suited for modeling at higher levels of abstraction

- Does not capture sequential logic behavior
  - (when output is a function of current & previous input values)

Conditional Signal Assignment Statement

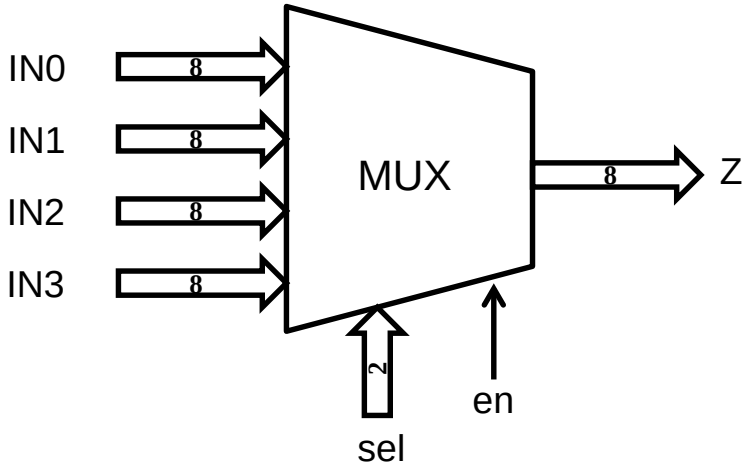
- Conditional signal assignment statement selects different values for the target signal based on the various specified conditions – it is like an if-then-else statement.

target-signal <= [*waveform-elements* **when** *condition* **else**]
[*waveform-elements* **when** *condition* **else**]
...
[*waveform-elements* **when** *condition* **else**]
[*waveform-elements*];

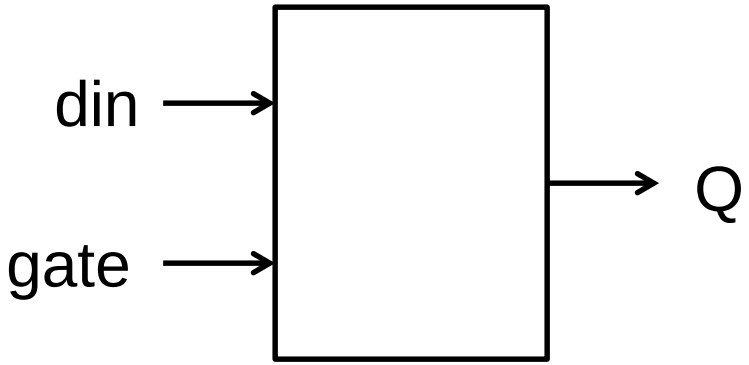
- Will be executed whenever an event occurs on a signal used in any of the waveform expressions, or in any of the conditions.
- Only the first clause found to be true is executed
  - Order of these clauses matters!

4-way 8-bit Multiplexer

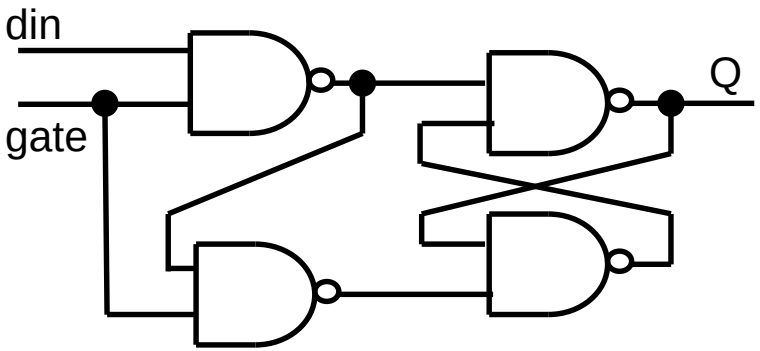
```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
        port ( IN0, IN1, IN2, IN3: in std_logic_vector (7 downto 0);
        sel: in std_logic_vector (1 downto 0);
        en: in std_logic;
        Z: out std_logic_vector (7 downto 0));
end entity mux4;

architecture behavioral of mux4 is
begin
        Z <= "00000000" after 5 ns when en='0' else
                IN0 after 5 ns when sel="00" else
                IN1 after 5 ns when sel="01" else
                IN2 after 5 ns when sel="10" else
                IN3 after 5 ns when sel="11" else
                "XXXXXXXX" after 5 ns;
end architecture behavioral;
```



1-bit Latch



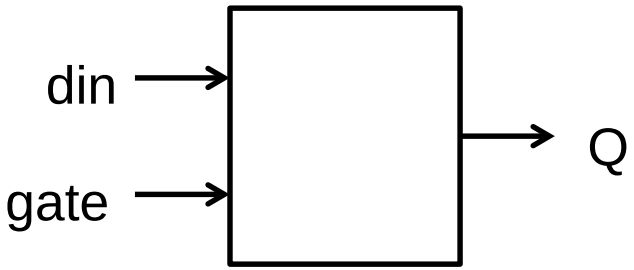
din	gate	Q
0	1	0
1	1	1
0	0	*no change*
1	0	*no change*



- Sequential circuit

- Gate level structure describes a possible implementation but does not explicitly capture behavior

1-bit Latch using Conditional Assignment




```
library IEEE;
use IEEE.std_logic_1164.all;

entity latch is
      port (din, gate: in std_logic;
      Q: out std_logic);
end entity latch;

architecture lat_1 of latch is
begin
      Q <= din after 3 ns when gate='1';
end architecture lat_l;
```

- There are combinations of inputs which will not trigger execution ⇨ memory of previous state

Selected Signal Assignment Statement

- Alternative form of conditional assignment
- Selects different values for a target signal based on the value of a select expression.
  - more like a case statement.

with *expression* **select**

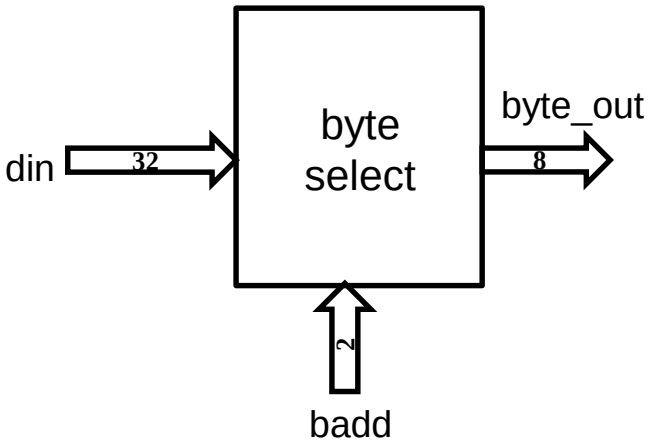
target-signal <= *waveform-elements* **when** *choices*,
waveform-elements **when** *choices*,
...
[*waveform-elements* **when others]**;

- Executed when event occurs on any signal in the select expression or on any signal used in any waveform expression.
- Choices must be mutually exclusive
  - all choices are evaluated – order does not matter

Select Signal Assignment: Byte Selector

- Select a specified byte from 32-bit word:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity bytesel is
        port ( din: in std_logic_vector (31 downto 0);
        badd: in std_logic_vector (1 downto 0);
        byte_out: out std_logic_vector (7 downto 0));
end entity bytesel;
```




```
architecture dataflow of  bytesel is
begin
        with badd select
                byte_out <= din(7 downto 0) after 3 ns when “00”,
                        din(15 downto 8) after 3 ns when “01”,
                        din(23 downto 16) after 3 ns when “10”,
                        din(31 downto 24) after 3 ns when “11”,
                        “XXXXXXXX” after 3 ns when others;
end architecture dataflow;
```

Sidebar: Concatenation Operator

- & operator can be used to concatenate a j-bit word and a k-bit word to produce a (j+k) bit word.

- For example:

if a= "0010" and b="1010" and of type std_logic_vector(3 downto 0):

a & b = "00101010"

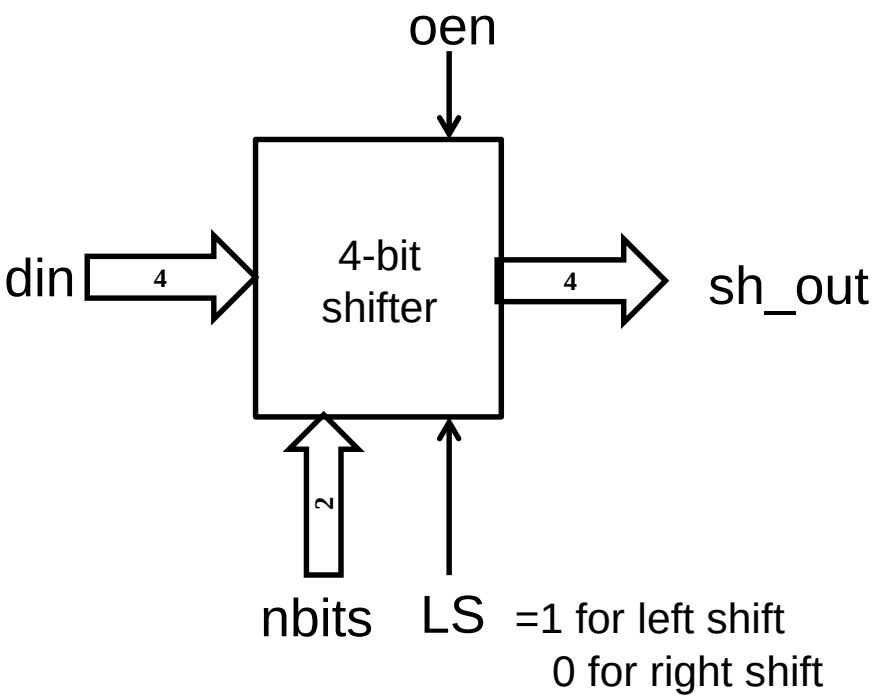
a & b & a = "001010100010"

a(3 downto 2) & "01" =

b(3) & a(1 downto 0) =

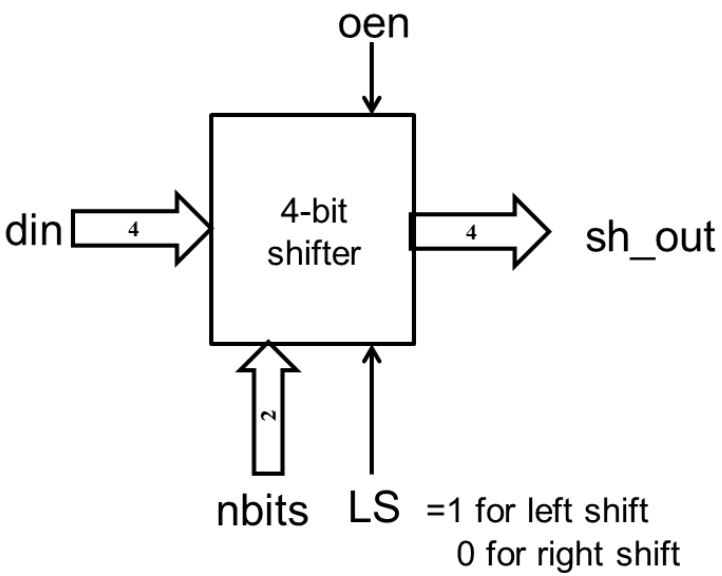
Example: 4-bit Logical Shifter

- Use Conditional Signal Assignment to describe a circuit that logically shifts a 4-bit word 0-3 bits to the right or left. Include tri-state output.



Example: 4-bit Logical Shifter

```
library IEEE;
use IEEE.std_logic_1164.all;

entity shift4 is
    port(din: in std_logic_vector (3 downto 0);
         nbits: in std_logic_vector (1 downto 0);
         LS, oen: in std_logic;
         sh_out: out std_logic_vector (3 downto 0);
end shift4;

architecture dataflow of shift4 is
begin
    sh_out<= "ZZZZ" when oen='0' else
                din when nbits="00" else
                din(2 downto 0) & '0' when nbits="01" and LS='1' else
                din(1 downto 0) & "00" when nbits="10" and LS='1' else
                din(0) & "000" when nbits="11" and LS='1' else
                '0' & din(3 downto 1) when nbits="01" and LS='0' else
                "00" & din(3 downto 2) when nbits="10" and LS='0' else
                "000" & din(3) when nbits="11" and LS='0' else
                "XXXX";
end architecture dataflow;
```

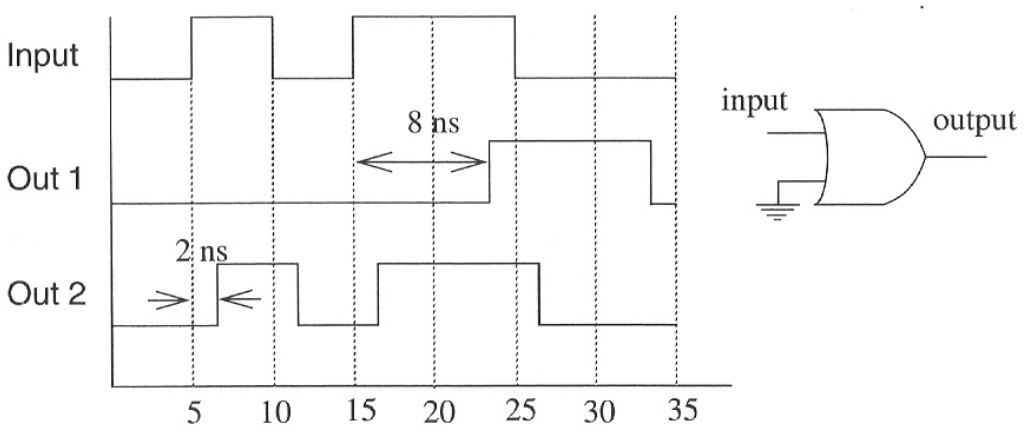


Inertial Delay

- By default, gate delays are considered inertial
  - due to intrinsic speed limitation of device
- What happens if a gate sees a very short pulse at its input?
- Inertial delay will filter out pulses shorter than the specified gate delay
  - Default rejection window = gate delay

out1 <= input **or** '0' **after** 8 ns.

out2 <= input **or** '0' **after** 2 ns.



Inertial Delay with Rejection Window

- We can override default rejection window:

  z <= **reject** 3ns **inertial** (x xor y) **after** 5ns;

- This gate has a delay of 5ns, but will only reject input pulses shorter than 3 ns.
- Inertial delay will also reject output pulses shorter than the reject window.
- For example, using xor gate (above) with following inputs:

  x <= '0', '1' **after** 10ns, '0' **after** 20ns;

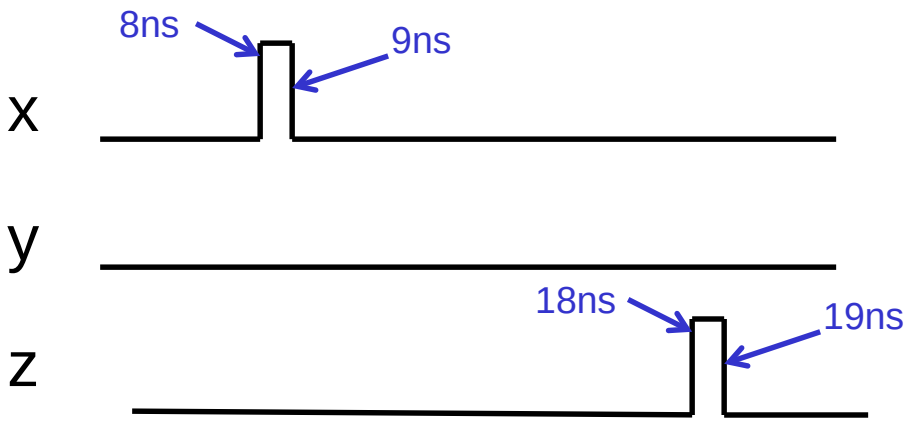
  y <= '0', '1' **after** 12ns, '0' **after** 18ns;

  generates no output on z

Transport Delay

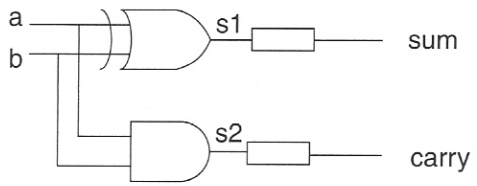
- Signals also experience delays through wires
  - Wires can change state very quickly (fast rise & fall time)
  - If wire is long, transport delay can be much greater than the wire rise/fall time.
  - Even a very long wire, may transmit very short pulses (with an appropriate delay
- Transport delay <u>does not</u> filter out short pulses:

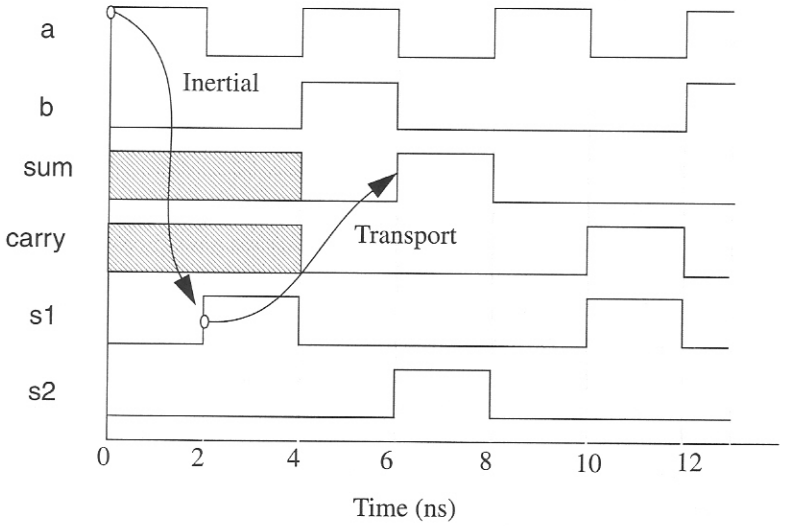
```
sum <= transport (x xor y) after 10ns;
```



Inertial + Transport Delay

```
library IEEE;
use IEEE.std_logic_1164.all;
entity half_adder is
port(a, b: in std_logic;
      sum, carry: out std_logic);
end entity half_adder;

architecture transport_delay of half_adder is
signal s1, s2: std_logic:= '0';
begin
s1 <= (a xor b) after 2 ns;
s2 <= (a and b) after 2 ns;
sum <= transport s1 after 4 ns;
carry <= transport s2 after 4 ns;
end architecture transport_delay;
```



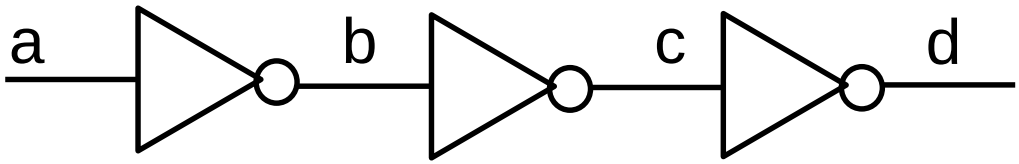


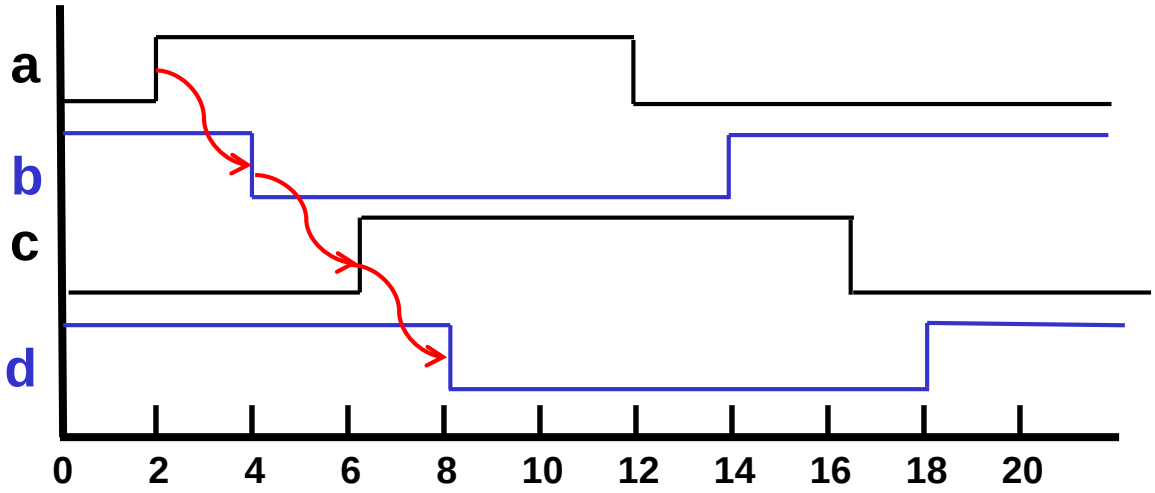
- Allows us to independently model gate (inertial) and wire (transport) delays

Dataflow Delays

- Dataflow describes process of events flowing from one device to another

```
b <= not a after 2 ns;
c <= not b after 2 ns;
d <= not c after 2 ns;
```

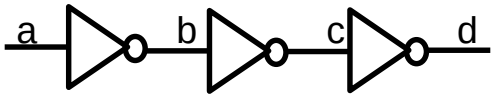


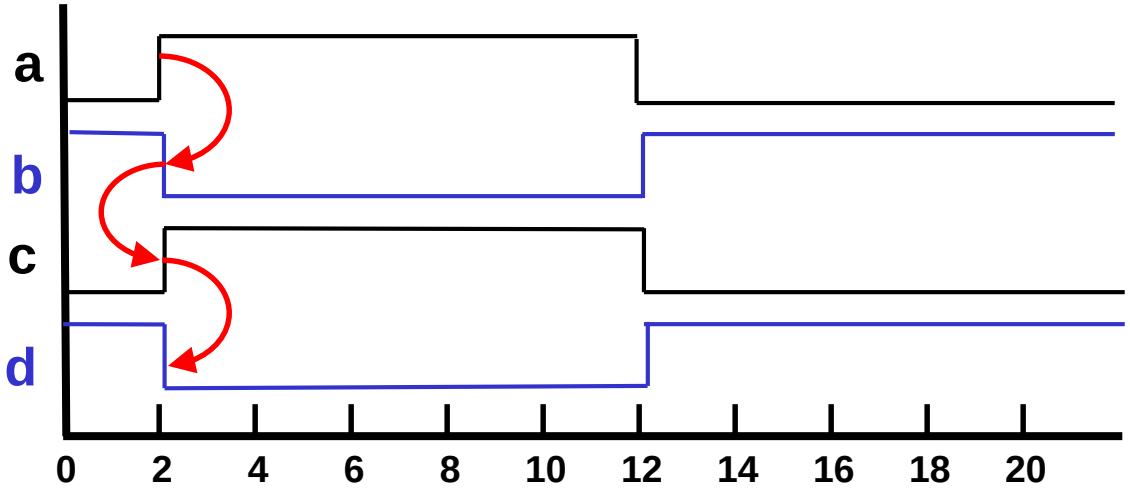


Zero Delays

- What happens when delay is zero?

b <= **not** a **after** 0 ns; *or* b <= **not** a;

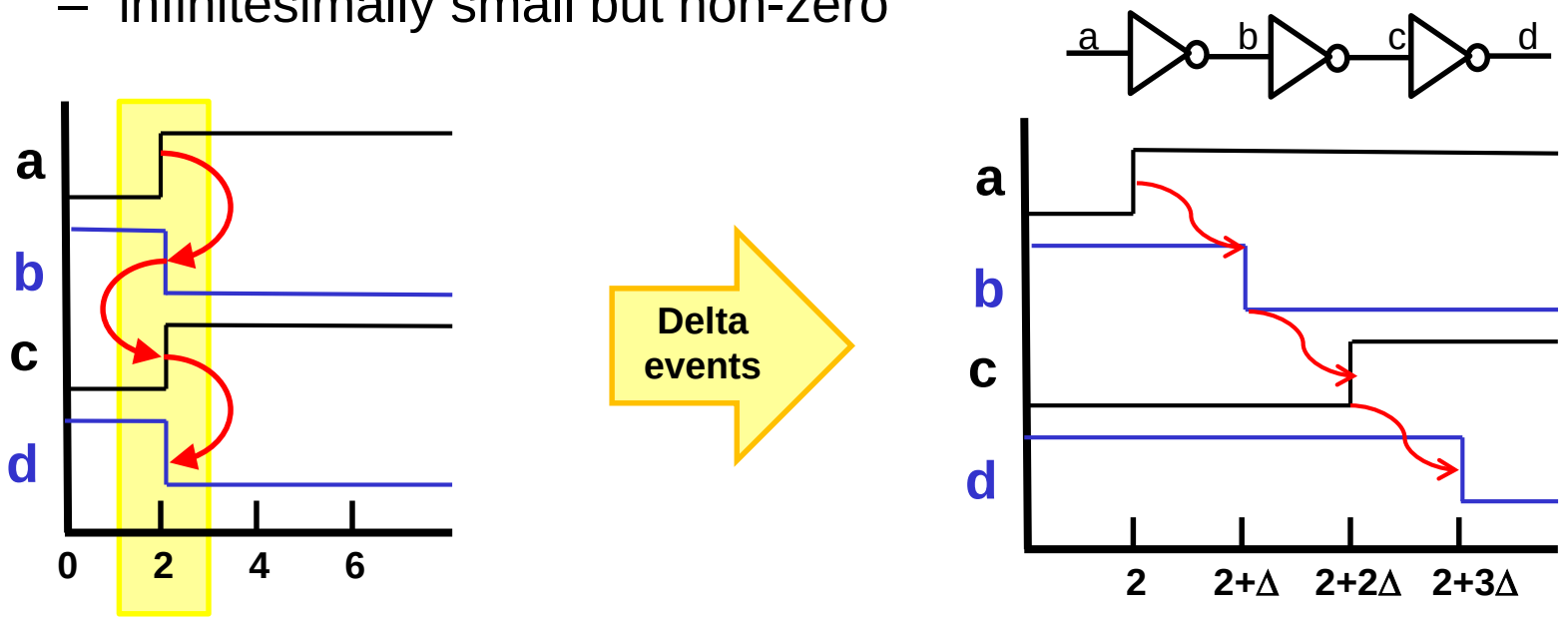




- Are these events really happening in zero time?
- What impact does this have on concurrency?

Delta Delays

- When no delay is specified, simulator adds small (delta) delay Δ when scheduling the output event
- Δ is smaller than any physical delay
  - infinitesimally small but non-zero



- This maintains correct data flow and ensures events processed in correct order
  - without introducing physical delay

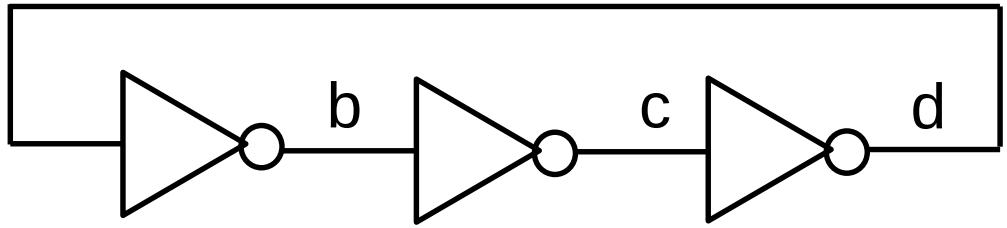
Signal Delays - Summary

- VHDL signals do not change instantaneously.
- A scheduled change for a VHDL signal never occurs at the present time but is always delayed until some future time.
  - this is the basis of concurrency
  - execution of one CSA cannot affect the execution of another CSA *at the present time*
- The future time at which the change is to take affect can be explicitly stated. If no time is specified for a signal change, the default future time is the present time plus an infinitesimally small time called delta time.

Example: Ring Oscillator

- What will happen if the output of the inverter chain is fed back to the input?

```
b <= not d after 2 ns;
c <= not b after 2 ns;
d <= not c after 2 ns;
```

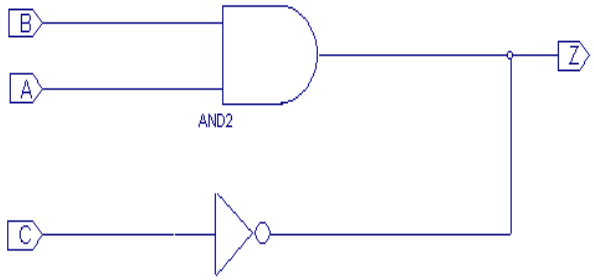


- What will happen if no delay is specified?

Multiple Drivers

- Each concurrent signal assignment statement creates a driver for the signal being assigned
- Can there be more that one driver for a signal?
  - depends on the type of the signal

```
z <= a and b after 10 ns;
z <= not c after 5 ns;
```

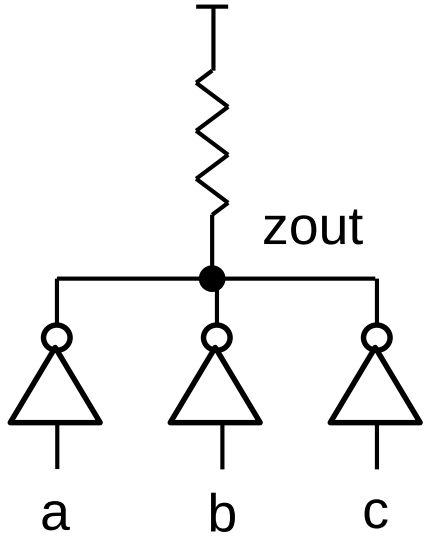


- With standard (unresolved) types (e.g. bit, std_ulogic, integer), this is illegal and will cause either a compiler or a run-time error
- With resolved types (e.g. std_logic, std_logic_vector) a resolution function is invoked to determine correct result (more on this later).

Std_logic Resolution Table

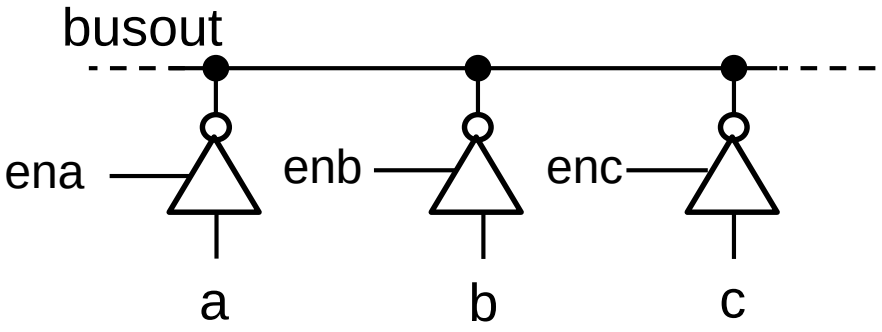
	U	X	0	1	Z	W	L	H	-
U (uninitialized)	U	U	U	U	U	U	U	U	U
X (unknown)	U	X	X	X	X	X	X	X	X
0 (forcing ‘0’)	U	X	0	X	0	0	0	0	X
1 (forcing ‘1’)	U	X	X	1	1	1	1	1	X
Z (high impedance)	U	X	0	1	Z	W	L	H	X
W (weak unknown)	U	X	0	1	W	W	W	W	X
L (weak ‘0’)	U	X	0	1	L	W	L	W	X
H (weak ‘1’)	U	X	0	1	H	W	W	H	X
- (don’t care)	U	X	X	X	X	X	X	X	X

Multiple Driver Examples



signal zout has 4 drivers
(3 open-drain buffers plus resistor)

buffers output ‘0’ or ‘Z’
resistor outputs ‘H’



tri-state bus

each buffer outputs ‘0’, ‘1 or ‘Z’
(only one driver active at a time)

Concurrent Assertion Statement

- During simulation and debugging it is useful to be able to check and report on signal values, e.g:
  - Illegal combination of inputs
  - setup or hold time violations
  - unexpected condition
- Assert statement provides mechanism for testing state of system and reporting results on simulator console

  assert *boolean-expression*
  [**report** *string-expression*]
  [**severity** *expression*];

- If the value of the *boolean-expression* is false, the report message is printed along with the severity level
  - executed when event occurs on any signal in *boolean-expression*

Severity Levels

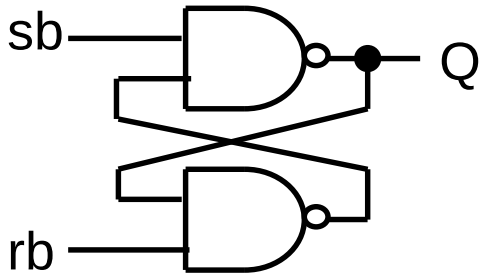
- Implementation dependent
- Common values:
  - NOTE
  - WARNING
  - ERROR
  - FAILURE *(this level will abort Xilinx Isim simulator)*

for example:

assert (a=b) **or** (a=c)
[**report** “a is not equal to b or c”]
[**severity** WARNING];

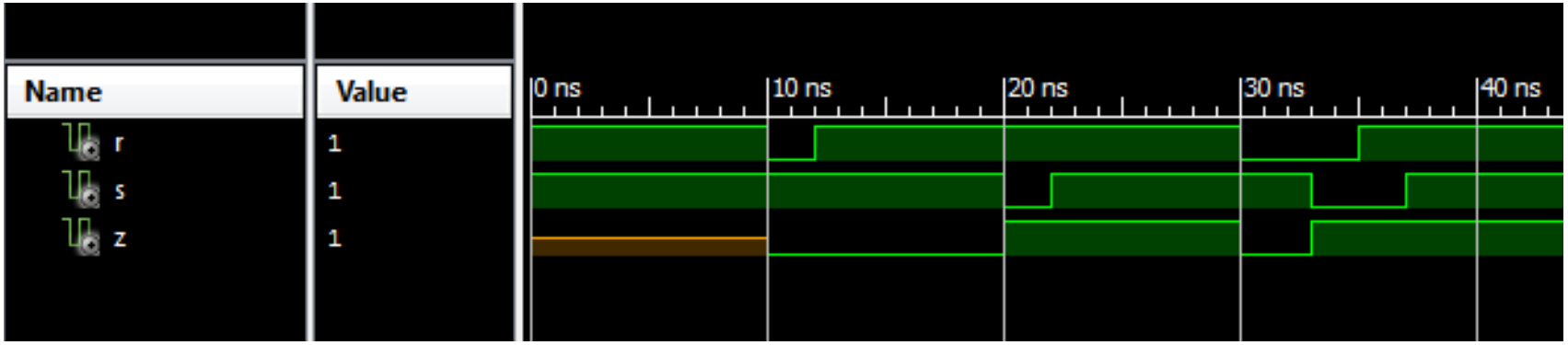
Example: RS Latch

```
entity rslat is
port(rb, sb: in std_logic;
    Q:out std_logic);
end entity rslat;

architecture rsa1 of rslat is
begin
    assert rb='1'
    report  "reset initiated"
    severity NOTE;

    assert (rb='1') or (sb='1')
    report  "rb and sb both zero"
    severity ERROR;

    z <= '1' when sb='0' else
         '0' when rb='0';

end architecture rsa1;
```



rb	sb	Q
0	1	0
1	0	1
1	1	*no change*
0	0	*illegal*

RS Latch: Simulation Output



Console

```
# run 1000 ns
Simulator is doing circuit initialization process.
Finished circuit initialization process.
at 10 ns: Note: reset initiated (/rs_assert_tb/uut/).
at 30 ns: Note: reset initiated (/rs_assert_tb/uut/).
at 33 ns: Error: Both r and s are zero
ISim>
```