VHDL:

Language Basics

Goals

- To study the rules on identifiers and know the VHDL reserved words.
- To differentiate between signal and variable assignments.
- To learn the different port signal modes.
- To get familiar with the delta delay concept.
- To differentiate between inertial and transport delays.
- To differentiate between transactions and events on signals.
- To learn some signal attributes.
- To get familiar with using constants in VHDL.
- To learn different VHDL data types (scalar and composite) and operations.
- To get started on using the std_logic type and know how it is resolved.
- To know how to declare a new data type in VHDL.
- To get acquainted with object visibility rules in VHDL.

Identifiers (Names)

- An identifier may contain letters `'A'--'Z'`, decimals `'0'--'9'`, and the underscore `'_'`.
- An identifier must start with a letter.
- An identifier must fit in a single line.
- An identifier cannot have an underscore at the beginning or the end or two successive underscores.
- A VHDL reserved keyword cannot be used as an identifier.
- VHDL is case insensitive: B $\equiv$ b.

Example: 1 (Home) *Show whether each of the following identifiers is legal or not. Explain!*

```
4plus4      A$1      V-3
The__state  _State   State_
```

VHDL Reserved Keywords

abs	access	after	alias	all	and
architecture	array	assert	attribute	begin	block
body	buffer	bus	case	component	configuration
constant	disconnect	downto	else	elsif	end
entity	exit	file	for	function	generate
generic	group	guarded	if	impure	in
inertial	inout	is	label	library	linkage
literal	loop	map	mod	nand	new
next	nor	not	now	null	of

VHDL Reserved Keywords (Cont.)

on	open	or	others	out	package
port	postponed	procedure	process	pure	range
record	register	reject	rem	report	return
rol	ror	select	severity	signal	shared
sla	sll	sra	srl	subtype	then
to	transport	type	unaffected	units	until
use	variable	wait	when	while	with
xnor	xor				

Objects (Value Holders)

- An object is something that can hold a value.
- There are three types of objects in VHDL:
  1. Variables ✓
  2. Signals ✓
  3. Constants ✓
- Each object has a **type** associated with the value it is holding.
- In hardware, value holders are:
  1. ✓ Wires
  2. ✓ Latches
  3. ✓ Flip-Flops
- VHDL objects are mapped to hardware value holders.

Variables

- Variables can only be declared inside sequential bodies, e.g., processes.
- We declare a variable in VHDL as follows:

```
VARIABLE memory_size:  integer;
VARIABLE delay:  time := 5 ns;
VARIABLE temp:  real;
VARIABLE reset:  boolean := false;
```

- Some of the above variables were declared with **initial** values.
- Because variables are local objects to processes, they cannot be used for carrying information between processes.
- At start of simulation, all variables are assigned their **initial** values, if they have any. Otherwise, they get the **default** value of their corresponding types.

Declarative Part Usage

	entity	architecture	process	package	package body	configuration
configuration		X				
use clause	X	X		X	X	X
component		X		X		
variable			X			
signal	X	X		X	X	
constant	X	X	X	X	X	
type and subtype	X	X	X	X	X	
subprogram declaration	X	X	X	X	X	
subprogram body	X	X	X		X	
generic	X	X		X		

Variable Assignment Statements

Effect of variable assignment (:=) is immediate.

Example: 2 (Home) *Show whether the following two pieces of code are equivalent or not by evaluating the value of s at the end of the code. Comment!*

```
   SIGNAL s:  integer;
p1:  PROCESS IS
   VARIABLE a,b,c,d:  integer;
BEGIN
   a := 2; b := 5; c := 3;
   c := a * b;
   d := c + 2;
   s <= d;
   wait;
END PROCESS p1;
```

a b c d s
2 5 3 0 0
10
12
0
12

$\boxed{5 \mid 12 \mid 0}$

```
   SIGNAL s:  integer;
p2:  PROCESS IS
   VARIABLE a,b,c,d:  integer;
BEGIN
   a := 2; b := 5; c := 3;
   d := c + 2;
   c := a * b;
   s <= d;
   wait;
END PROCESS p2;
```

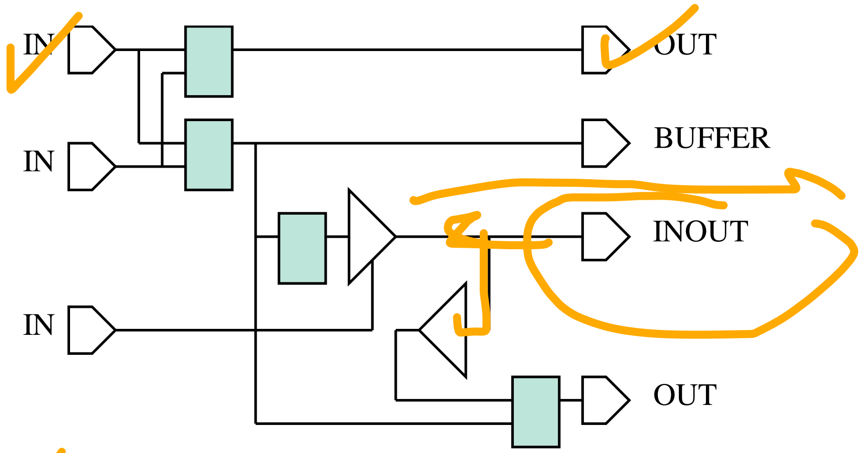
d=5
c=10

$\boxed{5 \mid 5 \mid 0}$

Signals

- Signals are complex compared to variables.
- Signals cannot be declared inside processes.
- Only signals can be used for transferring information between modules (e.g., processes).
- VHDL signals are of two types:
  1. **Ports:** these are the pins of chips. They have directions (modes) and are declared in entities. Input signals may be initialized in the entity declaration. This initialization is ignored at synthesize time.
  2. **Internals:** these are the architecture interconnections. They have no direction and are declared and used in architectures.
- At start of simulation all signals are assigned their **initial** values, if they have any. Otherwise, they get the **default** value of their corresponding types.

Port Signal Modes



1. Ports of mode **OUT** cannot be read (used on a RHS of signal assignment (<=) statement).
2. Ports of mode ~~**BUFFER**~~ carry data out of a module, but can also be read within the module (allowing for internal feedback). However, the port cannot be driven from outside the entity, so it cannot be used for data input.
3. Best to use internal signals and **IN**, **OUT**, and **INOUT** ports. ~~**BUFFER**~~ ports are not real. They are not supported in synthesis.

Port Signal Modes (Cont.)

Example: 3 (Home) *What is wrong with the following process:*

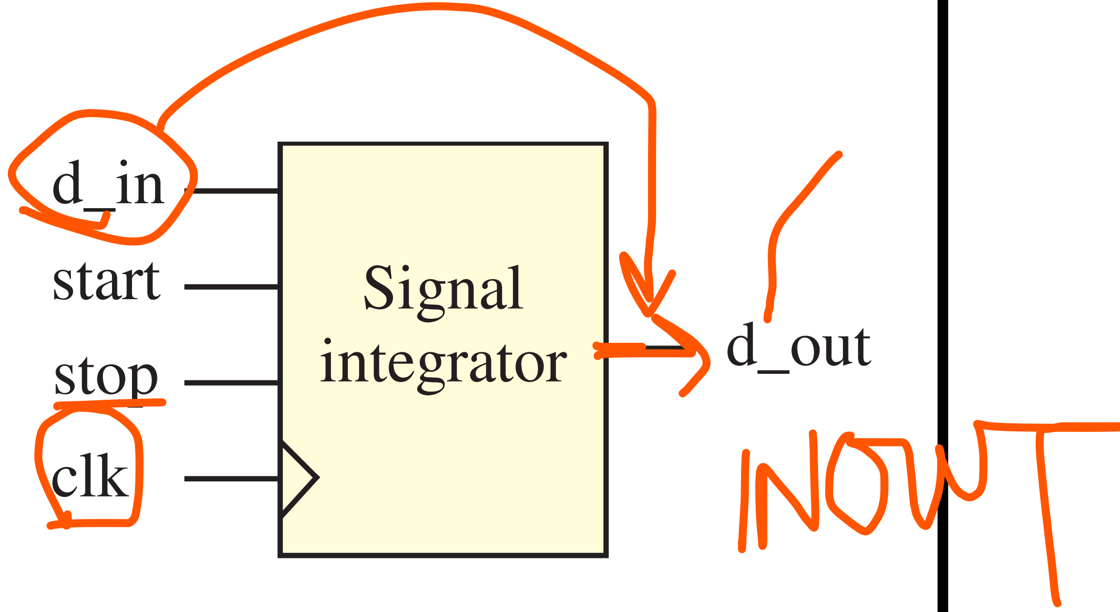
```
ENTITY wrong IS
   PORT( a, b:  IN bit;
         c:  OUT bit);
END ENTITY wrong;
ARCHITECTURE wrong OF wrong IS BEGIN
   pw:  PROCESS (a, b, x) IS
      VARIABLE x:  bit;
   BEGIN
      x <= a OR b AND c;
   END PROCESS pw;
END ARCHITECTURE wrong;
```

Example: 4 (Class) *Mention two different ways to read an output signal.*

1. Define the output port signal mode as INOUT or BUFFER.
2. Define the output port signal mode as OUT and use an internal signal/variable to hold a copy of the output signal value. Do all intermediate operations on the internal signal/variable and finally assign it to the actual output port signal.

Port Signal Modes (Cont.)

Example: 5 (Class) *Write VHDL code for the shown signal integrator, with d_out $= \sum$ d_in*



We cannot use a statement like

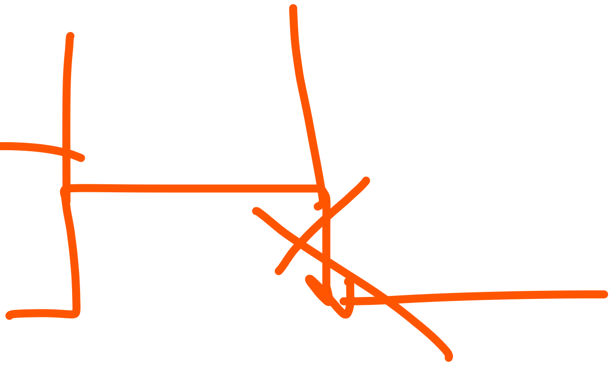
```
d_out <= d_out + d_in
```

if d_out is a signal of type OUT, since it cannot be used on the RHS of a signal assignment statement.

Port Signal Modes (Cont.)

Example 5: (Cont.)

```
ENTITY integrator IS
   PORT( d_in:  IN real := 0.0;
         clk, start, stop:  IN bit;
         d_out:  OUT real);
END ENTITY integrator;
ARCHITECTURE behav OF integrator IS BEGIN
   pi:  PROCESS (clk) IS
      VARIABLE temp:  real := 0.0;
   BEGIN
      IF clk = '1' AND clk'event THEN
         IF start = '1' THEN
            temp := 0.0;
         ELSIF stop = '1' THEN
            d_out <= temp;
         ELSE
            temp := temp + d_in;
         END IF;
      END IF;
   END PROCESS pi;
END ARCHITECTURE behav;
```



Port Signal Modes (Cont.)

Example: 6 (Home) *How will the system in Example 5 behave if both start and stop are set to '1' at the same time? Comment! Change the code to assert that both start and stop are not set to '1' at the same time and if this is violated, get the code to generate a warning without responding to this input combination.*

Example: 7 (Home) *Which one of the following sentences properly describes the integrator in Example 5? Comment! (Can you answer this question by only inspecting the sensitivity list?)*

1. *The integrator has asynchronous start and stop inputs.*
2. *The integrator has synchronous start and stop inputs.*

If the design has asynchronous start and stop inputs, can you change it so it has synchronous start and stop inputs and vice versa?

Port Signal Modes (Cont.)

Example: 8 (Home) *Which one of the following sentences properly describes the integrator in Example 5?*

1. *Intermediate outputs can be read out of the system while the input data is being supplied.*
2. *Only final output can be read out of the system when the system stops.*

If it is possible to read intermediate outputs out of the system, can you change it to only be able to read final output and vice versa?

Signal Delays

- Signals describe data waveforms versus time.
- Signals emulate real electrical signals traveling on wires in hardware designs, which require certain amounts of delay to change values.
- Delays arise mainly due to charge and discharge of parasitic capacitors. A small delay component is due to ballistic velocity of electrons in devices.
- A signal value does not change as soon as the signal assignment statement is executed.
- A signal assignment inside a process takes effect only when the process suspends on a wait statement or waiting for signals in the sensitivity list to change. Until that happens, the signal keeps its previous value.
- A signal assignment statement schedules a transaction for the signal, which is applied after the process suspends.
- A process does not see the effect of a signal assignment until the next time it resumes.

Signal Delays (cont.)

Example: 9 (Home) *Evaluate the value of x, y, and z at the end of the following code. Comment!*

```
ARCHITECTURE a of e IS
   SIGNAL y:  integer := 10;
   SIGNAL z:  integer := 20;
BEGIN
   p1:  PROCESS IS
      VARIABLE x:  integer := 5;
   BEGIN
      y <= z + 5;
      z <= x;
      x := 7 + y + z;
      z <= x + y;
      wait;
   END PROCESS p1;
END ARCHITECTURE a;
```

x y z

5 10 20

37

y 25

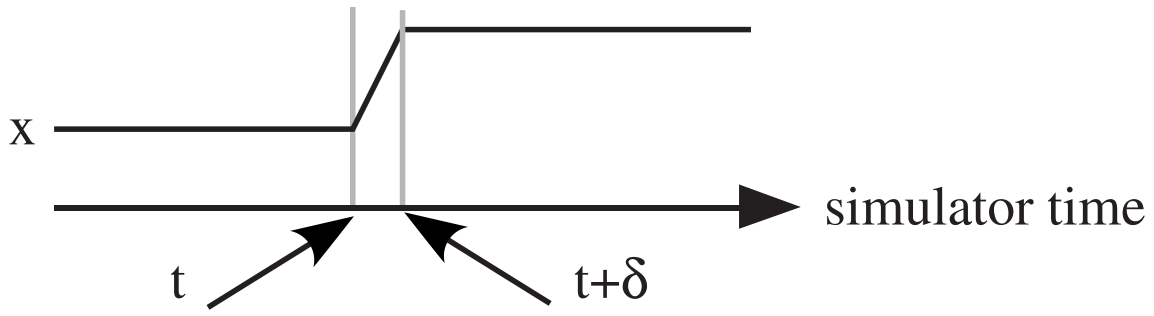
z 5

z 47

x=37, y=25 z=47

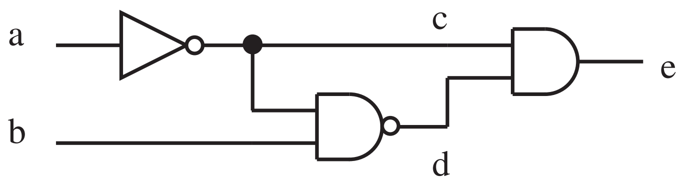
Delta Delay (δ)

- Signal assignments take effect after a specified amount of delay, which defaults to the so called **delta delay (δ)**, when no explicit reference to time is made in the signal assignment statement.
- δ could be thought of as an infinitesimal, but quantized, unit of time that ensures that signals are updated after the current simulation time.
- δ itself is not equal to a fixed time value; such that an infinite number of δ's could be accommodated between simulation intervals.
- In most VHDL simulators, δ is called the **iteration cycle**.
- A process executes all its statements in a δ time unit.

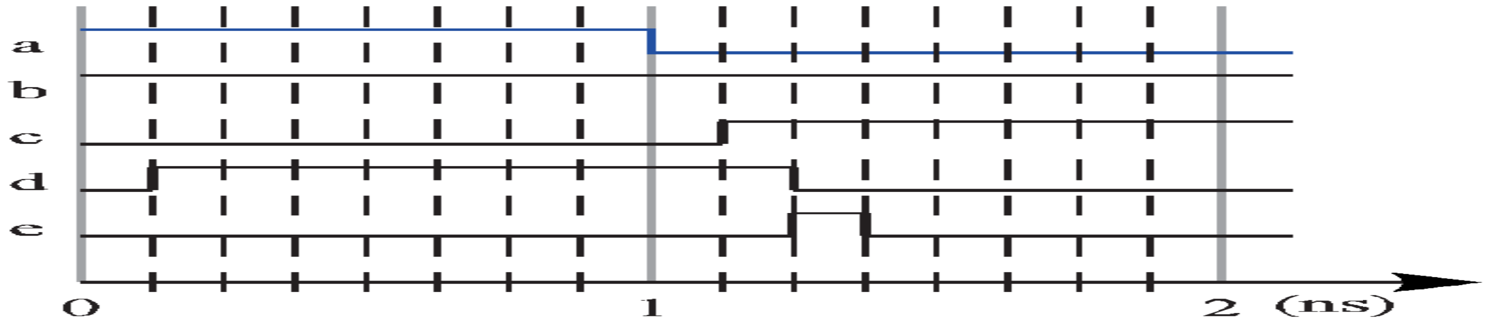


Delta Delay (δ) (cont.)

Example: 10 (Class) *Assume that signals `a` and `b` take the shown waveforms. Show the waveform generated for signal `e`. All signals are of type bit.*



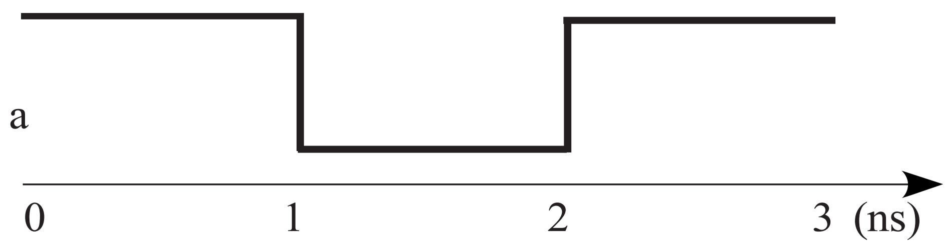
	Iteration cycle			Iteration cycle					
t	0	δ	2δ	1	$1+\delta$	$1+2\delta$	$1+3\delta$	$1+4\delta$	$\cdots$
a	1	1	1	0 *	0	0	0	0	$\cdots$
b	1	1	1	1	1	1	1	1	$\cdots$
c	0	0	0	0	1 *	1	1	1	$\cdots$
d	0	1 *	1	1	1	0 *	0	0	$\cdots$
e	0	0	0	0	0	1 *	0*	0	$\cdots$



Delta Delay (δ) (Cont.)

Example: 11 (Home) *Given the waveform of port a (of mode IN and type bit), generate the waveforms of signals b and c from the codes below.*

```
ARCHITECTURE w OF m IS
   SIGNAL b, c:  bit;
BEGIN
   b <= a;
   c <= b;
END ARCHITECTURE w;
```



```
ARCHITECTURE x OF m IS
   SIGNAL b, c:  bit;
BEGIN
   p:  PROCESS (a,b) IS
   BEGIN
      b <= a;
      c <= b;
   END PROCESS p;
END ARCHITECTURE x;
```

```
ARCHITECTURE y OF m IS
   SIGNAL b, c:  bit;
BEGIN
   q:  PROCESS (a) IS
   BEGIN
      b <= a;
      c <= b;
   END PROCESS q;
END ARCHITECTURE y;
```

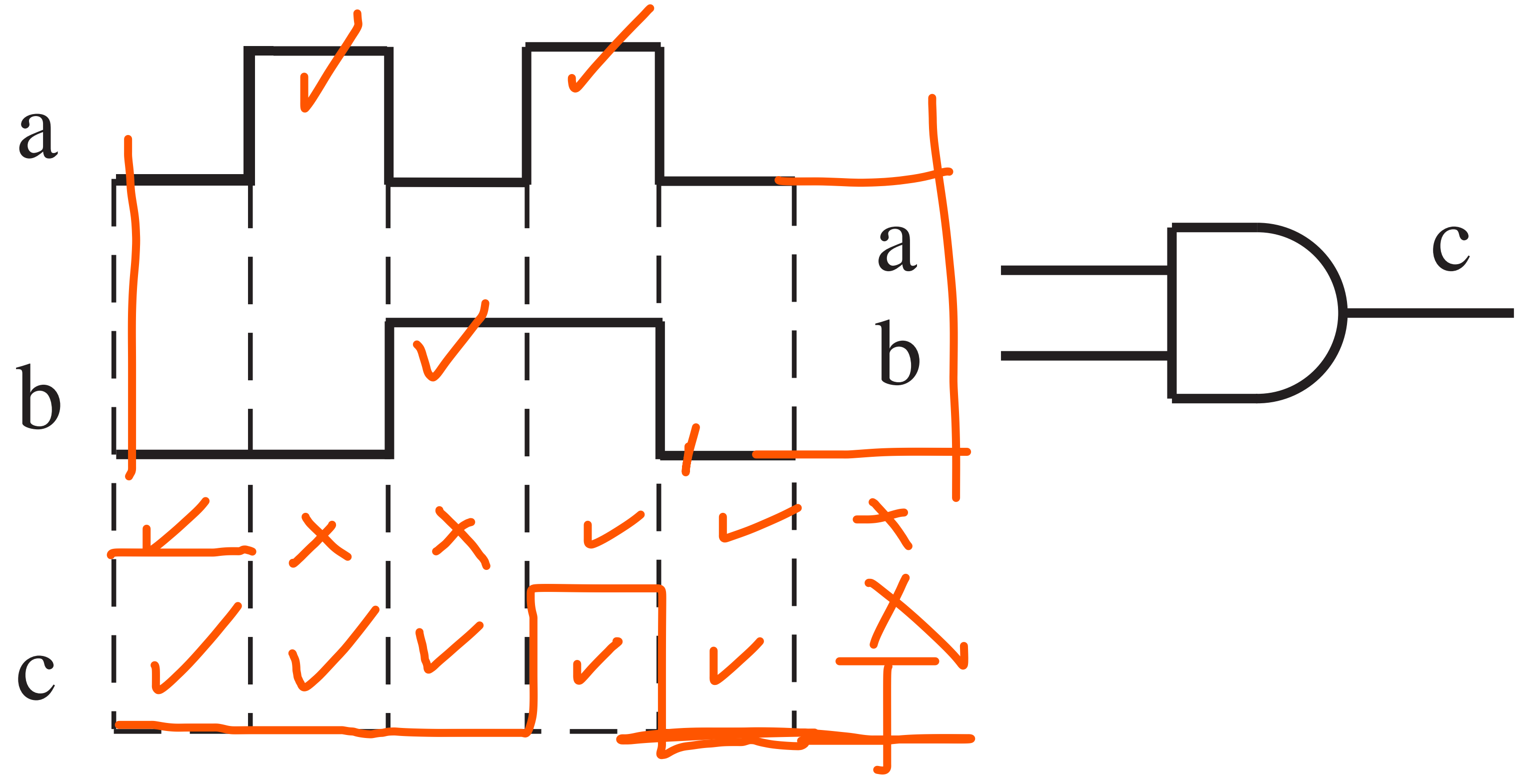
```
ARCHITECTURE z OF m IS
   SIGNAL c:  bit;
BEGIN
   r:  PROCESS (a) IS
      VARIABLE b:  bit;
   BEGIN
      b := a;
      c <= b;
   END PROCESS r;
END ARCHITECTURE z;
```

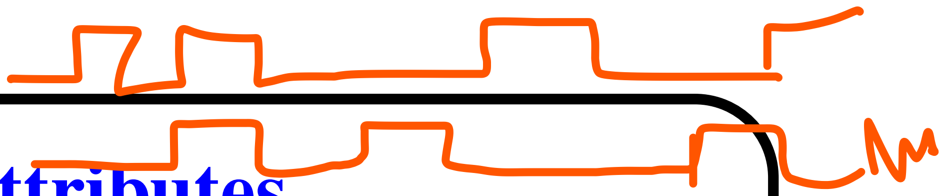
Transactions and Events on Signals

- Whenever a value is scheduled to be assigned to a signal s, a **transaction** is said to have been scheduled for this signal.
- When a value is actually assigned to the signal, a **transaction** is said to have occurred on s.
- If the new value of s is same as the old value of s, no change in the value takes place and no **event** is said to have occurred.
- If, however, the new value of s is different from its old value, an **event** is said to have occurred on s.

Transactions and Events on Signals (Cont.)

Example: 12 (Home) *Trace the waveform of the signal c and determine when transactions and events occur on this signal.*





Some Signal Attributes

`s'delayed(T)`	Copy of s delayed by time T.
`s'stable(T)`	A Boolean signal that is true if there has been no event on s in time interval T up to current time, otherwise false.
`s'quiet(T)`	A Boolean signal that is true if there has been no transaction on s in time interval T up to current time, otherwise false.
`s'transaction`	A signal of type bit that changes value from `'0'` to `'1'` or vice versa each time there is a transaction on s.
`s'event`	A Boolean signal that is true if there is an event on s in the current simulation cycle, otherwise false.
`s'active`	A Boolean signal that is true if there is a transaction on s in the current simulation cycle, otherwise false.
`s'last_event`	The time interval since the last event on s.
`s'last_active`	The time interval since the last transaction on s.
`s'last_value`	The value of s just before the last event on s.

Transport and Inertial Delay Mechanisms

- If there are pending transactions scheduled for a signal when a signal assignment is executed, the new transactions are merged with them in a way that depends on the delay mechanism used in the signal assignment.
- There are two types of signal delays in VHDL:
  1. Transport delays
  2. Inertial delays

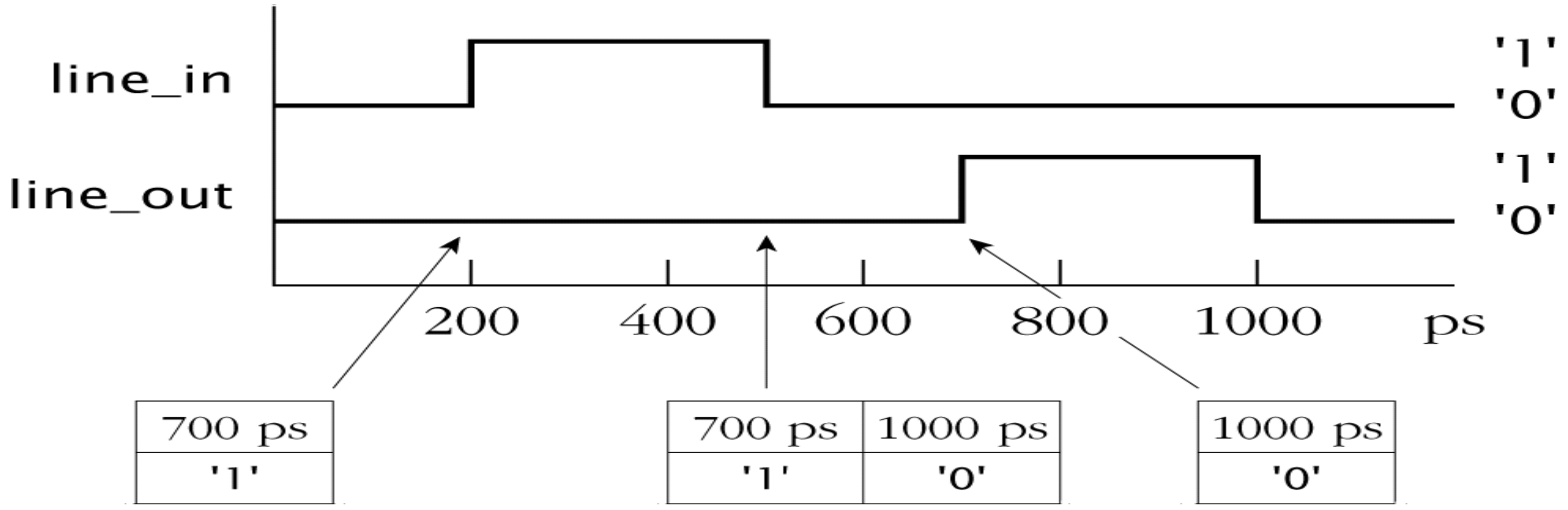
Transport Delay Mechanism

- This delay models an ideal device with infinite frequency response, in which any input pulse, no matter how short, produces an output pulse.
- It mimics delays across wires where glitches are delayed but not absorbed.
- An example of such a device is an ideal transmission line, which transmits all input changes delayed by some amount.
- Identified by the keyword **TRANSPORT**.
- When a new transaction is scheduled for a simulation time that is later than the pending transactions queued by the driver, they are queued behind the pending ones.
- If the input changes twice or more within a period shorter than the transport delay, the scheduled transactions are simply queued by the driver until the simulation time at which they are to be applied.

Transport Delay Mechanism (Cont.)

Example: 13 (Class) *Write a process that models an ideal transmission line with delay 500 ps. Show the transactions queued by the driver on the output (line_out) in response to the changes shown on the input (line_in).*

```
transmission_line:  PROCESS (line_in) IS BEGIN
   line_out <= TRANSPORT line_in AFTER 500 ps;
END PROCESS transmission_line;
```



When simulation time reaches 1000 ps, and the final transaction is applied, the driver queue is left empty.

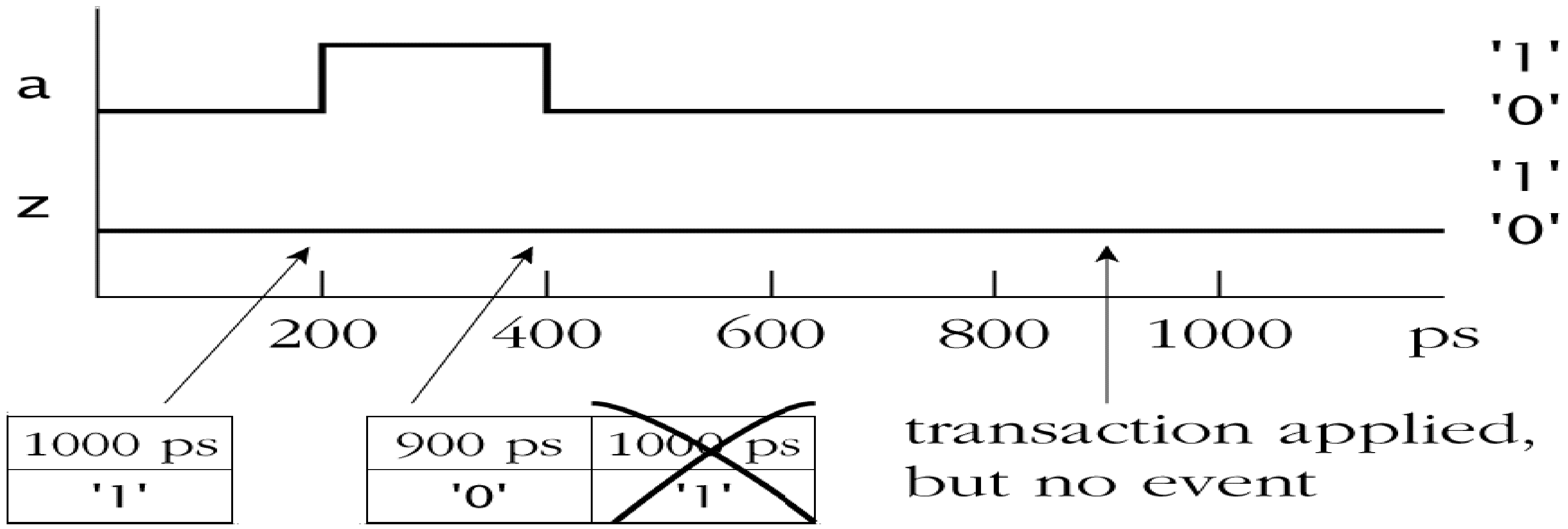
Transport Delay Mechanism (Cont.)

- When **variable delays** are used, we cannot schedule a transaction for an earlier time than a pending transaction.
- If there are pending transactions on a driver that are scheduled for a time later than or equal to a new transaction, those later transactions are deleted.

Transport Delay Mechanism (Cont.)

Example: 14 (Class) *Write a process that models an asymmetric delay element, with different delay times for rising (800 ps) and falling (500 ps) transitions. If we apply an input pulse of only 200 ps duration, show the transactions queued by the driver on the output (z).*

```
asym_delay:  PROCESS (a) IS BEGIN
   IF a THEN
      z <= TRANSPORT a AFTER 800 ps;
   ELSE
      z <= TRANSPORT a AFTER 500 ps;
   END IF;
END PROCESS asym_delay;
```



Transport Delay Mechanism (Cont.)

Example: 15 (Class) *Show the transactions queued by the driver on the signal s after executing the following two signal assignment statements.*

```
transport_waveform:  PROCESS IS
BEGIN
   s <= TRANSPORT 1 AFTER 1 ns, 3 AFTER 3 ns, 5 AFTER 5 ns;
   s <= TRANSPORT 3 AFTER 4 ns, 4 AFTER 5 ns;
   WAIT;
END PROCESS transport_waveform;
```

After statement 1	**s**	**(1,1 ns)**	**(3,3 ns)**	**(5,5 ns)**
				delete

After statement 2	**s**	**(1,1 ns)**	**(3,3 ns)**	**(3,4 ns)**	**(4,5 ns)**

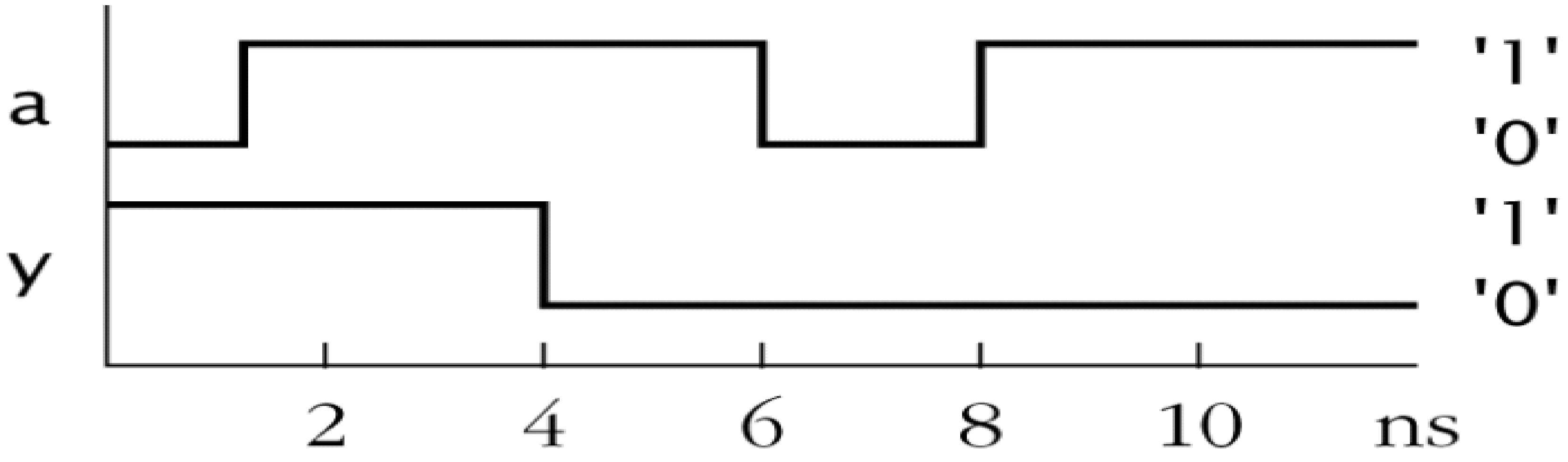
Inertial Delay Mechanism

- Most real electronic circuits do not have infinite frequency response.
- In real devices, changing the values of internal nodes and outputs involves moving electronic charge around in the presence of capacitance, inductance, and resistance. This gives the device some inertia. It tends to stay in the same state unless we force it by applying inputs for sufficiently long duration.
- The inertial delay mechanism allows us to model devices that reject input pulses too short to overcome their inertia.
- It filters out pulses shorter than the delay through a signal assignment. From this property inertial delay gets its name.
- It mimics delay in gates where very short glitches are absorbed.
- It is the mechanism used by default in a signal assignment, or we can specify it explicitly by including the optional keyword **INERTIAL**.

Inertial Delay Mechanism (Cont.)

Example: 16 (Class) *Given the shown waveform of signal a, generate the waveform of signal y from the following code (a and y are of type bit).*

```
inv:  PROCESS (a) IS BEGIN
   y <= INERTIAL NOT a AFTER 3 ns;
END PROCESS inv;
```



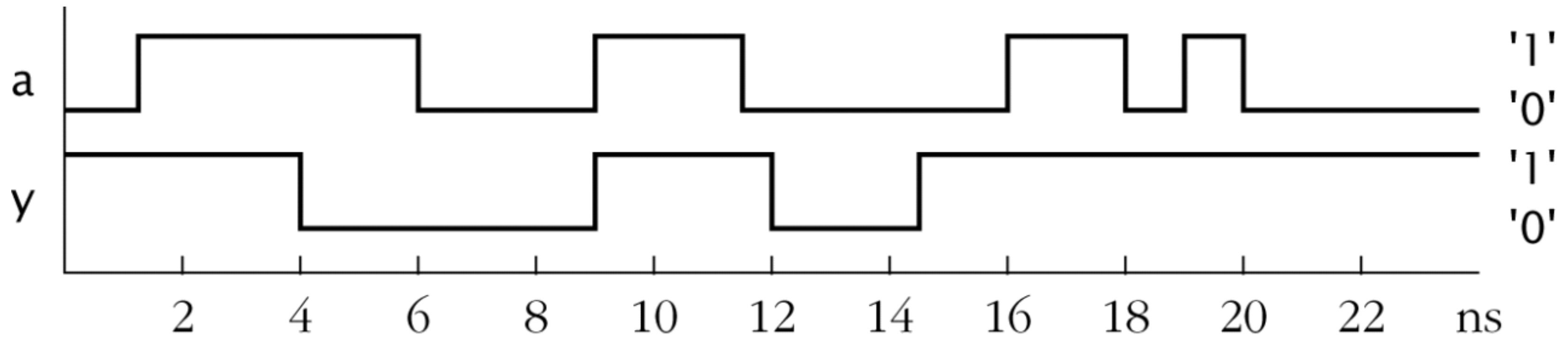
Inertial Delay Mechanism Cont.)

- If a signal assignment produces an output pulse shorter than (or equal to) the **pulse rejection limit**, the output pulse does not happen:
  - The pulse rejection limit specified must be between 0 and the delay specified in the signal assignment.
  - Omitting a pulse rejection limit is the same as specifying a limit equal to the delay.
  - Specifying a limit of 0 is the same as specifying transport delay.

Inertial Delay Mechanism (Cont.)

Example: 17 (Class) *Given the shown waveform of signal a, generate the waveform of signal y from the following code (a and y are of type bit).*

```
inv:  PROCESS (a) IS BEGIN
   y <= REJECT 2 ns INERTIAL NOT a AFTER 3 ns;
END PROCESS inv;
```



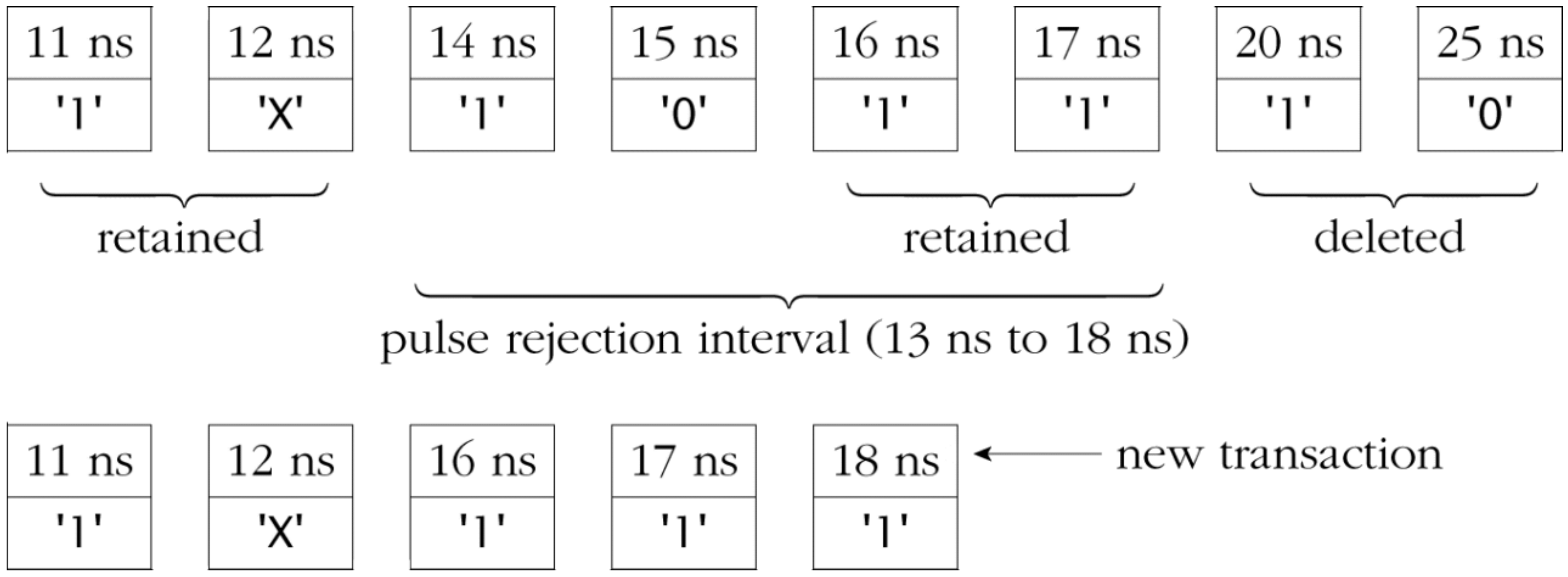
Inertial Delay Mechanism (Cont.)

- When inertial delay time and pulse rejection limit in different signal assignments applied to the same signal vary, we need to examine the pending transactions on a driver when adding a new transaction.
- Suppose a signal assignment schedules a new transaction for time t_{new}, with a pulse rejection limit of t_r:
  - Any pending transactions scheduled for a time later than or equal to t_{new} are deleted and the new transaction is added to the driver.
  - Any pending transactions scheduled in the interval $t_{new} - t_r$ to t_{new} are examined:
    - ∗ If there is a run of consecutive transactions immediately preceding the new transaction with the same value as the new transaction, they are kept in the driver.
    - ∗ All other transactions in the interval are deleted.
  - Any pending transactions scheduled for a time earlier than $t_{new} - t_r$ are retained.

Inertial Delay Mechanism (Cont.)

Example: 18 (Class) *Suppose a driver for signal* `s` *contains pending transactions as shown at the top of the figure below, and the the process containing the driver executes the following signal assignment statement at time 10 ns. Show the pending transactions after this statement.*

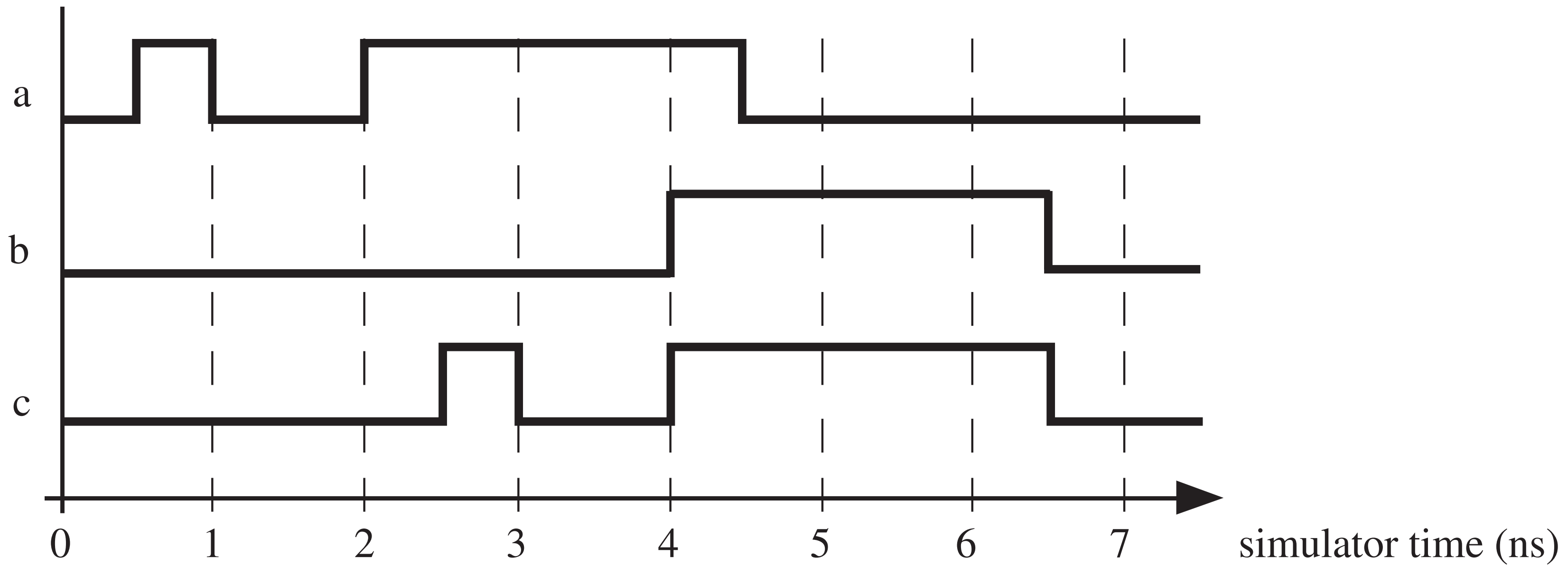
```
s <= REJECT 5 ns INERTIAL '1' AFTER 8 ns;
```



Inertial and Transport Signal Delays (Cont.)

Example: 19 (Class) *Given the shown waveform of signal a, generate the waveforms of signals b and c from the following code (all signals are of type bit).*

```
b <= a AFTER 2 ns;
c <= TRANSPORT a AFTER 2 ns;
```



Inertial and Transport Signal Delays (Cont.)

- **Corollary:** When a number of signal assignment statements are executed on the same signal in a process with no delay explicitly stated, the last assignment statement is the only effective one.
- **Important note:** VHDL does not wait for the delay to pass (inertial or transport) before going to the next instruction. It installs a driver, for example (0, 2 ns), on a signal and directly moves to the following statement.
- If a number of waveform elements are included:
  - The specified mechanism only applies to the first element.
  - All subsequent elements schedule transactions using transport delay.
  - Since the delays for the multiple waveform elements must be in ascending order, all of the transactions after the first are just added to the driver transaction queue in the order written.

Inertial and Transport Signal Delays (Cont.)

Example: 20 (Class) *Show the transactions queued by the driver on the signal s after executing the following two signal assignment statements.*

```
inertial_waveform:  PROCESS IS
BEGIN
   s <= 1 AFTER 1 ns, 3 AFTER 3 ns, 5 AFTER 5 ns;
   s <= 3 AFTER 4 ns, 4 AFTER 5 ns;
   WAIT;
END PROCESS inertial_waveform;
```

After statement 1	**s**	**(1,1 ns)**	**(3,3 ns)**	**(5,5 ns)**
		delete		**delete**
After statement 2	**s**	**(3,3 ns)**	**(3,4 ns)**	**(4,5 ns)**

Inertial and Transport Signal Delays (Cont.)

Example: 21 (Home) *What is the difference between the waveforms of signal a generated from the following two processes?*

```
p:  PROCESS IS
BEGIN
  a <= '1';
  a <= '0' AFTER 10 ns;
  a <= '1' AFTER 20 ns;
  WAIT;
END PROCESS p;
```

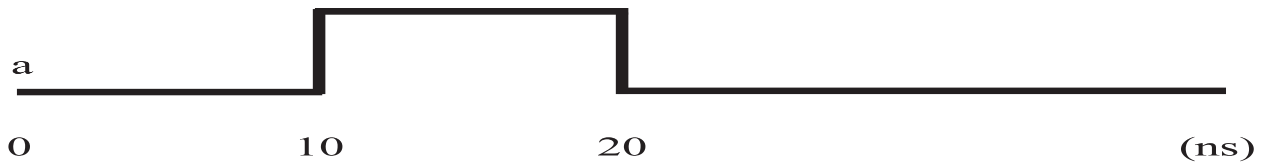
```
q:  PROCESS IS
BEGIN
   a <= '1', '0' AFTER 10 ns, '1' AFTER 20 ns;
   WAIT;
END PROCESS q;
```

Example: 22 (Home) *Show the waveform of signal a generated from the following process.*

```
r:  PROCESS IS
BEGIN
  a <= '0', '1' AFTER 10 ns, '0' AFTER 20 ns;
  WAIT FOR 15 ns;
  a <= '1';
  WAIT;
END PROCESS r;
```

Inertial and Transport Signal Delays (Cont.)

Example: 23 (Class) *Generate the shown pulse using two different ways.*



```
p1:  PROCESS IS
BEGIN
   a <= '0';
   WAIT FOR 10 ns;
   a <= '1';
   WAIT FOR 10 ns;
   a <= '0';
   WAIT;
END PROCESS p;

p2:  PROCESS IS
BEGIN
   a <= '0', '1' AFTER 10 ns, '0' AFTER 20 ns;
   WAIT;
END PROCESS p2;
```

The Simulation Cycle

- **The simulation cycle consists of two phases:**
  1. **A signal update phase** where simulation time is advanced to the time of the earliest scheduled transaction, and the values in all transactions scheduled for this time are applied to their corresponding signals. This may cause events to occur on some signals.
  2. **A process execution phase** where all processes that respond to these events are resumed and execute until they suspend again on a wait statement or waiting for signals in the sensitivity list to change.
- After these two phases, the suimulator repeats the simulation cycle.

Declarative Part Usage

	entity	architecture	process	package	package body	configuration
configuration		X				
use clause	X	X		X	X	X
component		X		X		
variable			X			
signal	X	X		X	X	
constant	X	X	X	X	X	
type and subtype	X	X	X	X	X	
subprogram declaration	X	X	X	X	X	
subprogram body	X	X	X		X	
generic	X	X		X		

Constants

- Constants are used to:
  - Specify the size of complex objects (e.g., arrays or buses),
  - Control loop counters,
  - Define timing parameters: delays, setup times, hold times, switching times, etc.,
  - etc.
- We declare a constant in VHDL as follows:

```
CONSTANT memory_size:  integer := 7;
CONSTANT delay:  time := 5 ns;
```

- It is preferable to declare constants in a package and use them later.

Declarative Part Usage

	entity	architecture	process	package	package body	configuration
configuration		X				
use clause	X	X		X	X	X
component		X		X		
variable			X			
signal	X	X		X	X	
constant	X	X	X	X	X	
type and subtype	X	X	X	X	X	
subprogram declaration	X	X	X	X	X	
subprogram body	X	X	X		X	
generic	X	X		X		

VHDL Types and Operations

- Scalar data types
  1. Numeric (integer and real). Can be used for simulation only. They are converted to bits at synthesize time.
  2. Character ✓
  3. String ✓
  4. Physical
  5. Enumeration (e.g., bit, three_level, boolean)
- Composite data types
  1. Array (e.g., bit_vector, std_logic, std_logic_vector)
  2. Record
- VHDL is a **declarative language**. All used signals or variables should be first declared (assigned a data type).

Numbers (Integers and Reals)

- Integers do not have a decimal point, e.g., 0, 23, 4E5, 1146.
- Reals have a decimal point, e.g., 0.0, 32.1, 1.2E-8.
- Reals have the range: -1.0E+38 – 1.0E+38;
- We can use bases other than 10, in the range 2–16, where exponent represents the power of the base:

```
2#111010# -- Base 2 integer
16#FD#    -- Base 16 integer
2#0.10#   -- Base 2 real (0.5)
8#0.4#    -- Base 8 real (0.5)
12#0.6#   -- Base 12 real (0.5)
2#1#E10   -- Base 2 integer
16#4#E2   -- Base 16 integer
```

Numbers (Integers and Reals) (Cont.)

- Two integer subtypes are defined:

```
SUBTYPE natural IS integer RANGE 0 TO integer'high;
SUBTYPE positive IS integer RANGE 1 TO integer'high;
```

- We can do the following **numerical operations** on integers:

  + - * / REM MOD ABS ** (integer exponent)

- We can do the following **numerical operations** on reals:

  + - * / ABS ** (integer exponent)

Example: 24 (Class) *What is the difference between REM and MOD.*

`x REM y` -- has the same sign as x.

`x MOD y` -- has the same sign as y.

REM and MOD are equivalent if x and y have the same sign, but differ if x and y have different signs.

Type Conversion

```
x := real (123);        -- converts integer to real
y := integer (3.6);     -- converts real to integer (rounding)
```

Characters

- Specified using single quote marks, e.g., `'0'`, `'A'`, `'9'`.
- 9 is a number, whereas `'9'` is a character.

Strings

1. Character strings: represent sequences of characters, e.g., `"THIS IS A STRING"`.
2. Bit strings: represent register values, e.g., `"00111001"` or equivalently `"0011_1001"`.
3. We can specify the base of a bit string to be:
   - b: binary (default),
   - o: octal, or
   - x: hexadecimal.

   For example, `b"1111_0010_0001"` ≡ `x"F21"`

   7

Physical Types

These types are used to represent real physical quantities; e.g.,

```
TYPE time IS RANGE 0 TO 1E20
   UNITS
      fs ;-- base unit 10**-15 s
      ps = 1000 fs;
      ns = 1000 ps;
      us = 1000 ns;
      ms = 1000 us;
      s = 1000 ms;
END UNITS time;

TYPE length IS RANGE 0 TO 1E5
   UNITS
      um ;-- base unit 10**-6 m
      mm = 1000 um;
      cm = 10 mm;
      mil = 254 um;
      inch = 1000 mil;
END UNITS length;
```

Physical Types (Cont.)

- The primary unit is the only mandatory unit in the type declaration.
- Range specified in the header of a physical type declaration refers to the primary unit only.
- All secondary units are defined with respect to the primary unit and as integral multiples of it.
- The physical types are not synthesizable.
- We can do the following **numerical operations** on physical types:

  +   -   ABS
- Examples:

```
x = 5 mm; y = 6 cm;
z <= x + y; -- z = 65 mm
```

- We can also multiply or divide a physical type by an integer or real.

Enumeration Type

- Allows using proper values for an object from a set of valid values.
- The synthesizer uses the proper number of bits for each value.

Example: 25 (Individual) *Define a signal to represents the summer months.*

```
TYPE summer IS (may, june, july, august);
SIGNAL s:  summer := may;
```

Default value for summer is may.

Signal s requires 2 bits for its representation.

Value	Binary equivalent
may	"00" ✓
june	"01" ✓
july	"10" ✓
august	"11" ✓

Enumeration Type (Cont.)

```
TYPE bit IS ('0', '1');

TYPE three_level IS ('0', '1', 'Z');
```

- We can do the following **logical operations** on bit type:

  NOT AND OR NAND NOR XOR XNOR

Example: 26 (Individual) *Define x to be a three_level signal and show how to assign it to high impedance.*

```
SIGNAL x:  three_level;
        ⋮
   x <= 'Z';
```

Enumeration Type (Cont.)

```
TYPE boolean IS (false, true);
```

- We can do the following **logical operations** on boolean type:

  NOT AND OR NAND NOR XOR XNOR

- We can do the following **relational operations** to compare two objects of the same type. The result is boolean:

  = /= < <= > >=

Example: 27 (Class) *Evaluate the following relational expression:*

`"1011" < "1100"` *Comment!*

The expression evaluates to **true**.

Important Note: When two bit vectors are compared, they do not need to be of the same length. Both operands are left justified and trailing zeros are added to the shorter.

Composite Types: Arrays and Records

- Composite data objects consist of a collection of related data elements in the form of an array or record.
- Before we can use such objects one has to declare the composite type first.
- Typically, we lump composite type declarations in a package that we use later.

Arrays and Array Attributes

An array type is declared as follows:

```
TYPE my_word IS ARRAY (15 DOWNTO 0) OF bit;
TYPE var IS ARRAY (0 TO 7) OF integer;
```

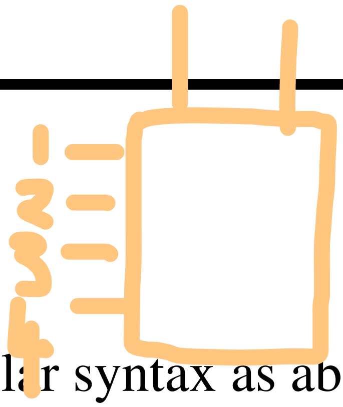
We can now declare objects of these data types:

```
SIGNAL mem_addr:  my_word;
CONSTANT settings:  var := (2, 4, 6, 8, 10, 12, 14, 16);
```

Some of the array attributes are:

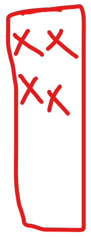
```
mem_addr'low = 0                  mem_addr'high = 15
mem_addr'right  = 0               mem_addr'left  = 15
mem_addr'range = 15 DOWNTO 0      mem_addr'reverse_range = 0 TO 15
mem_addr'length = 16
```

Arrays (Cont.)



Multidimensional arrays are declared by using a similar syntax as above:

```
TYPE mat4x2 IS ARRAY (1 TO 4, 1 TO 2) OF integer;
VARIABLE m4x2:  mat4x2 := ((0,2), (1,3), (4,6), (5,7));
```



To access an element, specify the index, e.g., mat4x2 (3,1) returns 4.

Sometimes it is more convenient not to specify the dimension of the array when the array type is declared. This is called an unconstrained array type. The range is specified when the array object is declared:

```
TYPE vector IS ARRAY (integer RANGE <>) OF integer;
VARIABLE vec6:  vector (2 DOWNTO -3) := (3, 5, 1, 4, 7, 6);
```

Bit_Vector Type

```
TYPE bit_vector IS ARRAY (integer RANGE <>) OF bit;
```

A bit_vector is an array with each element of type bit:

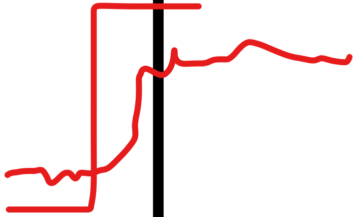
```
SIGNAL s:   bit_vector (7 DOWNTO 0);           -- s requires 8 bits.
VARIABLE v:   bit_vector (0 TO 15)             -- v requires 16 bits.

SUBTYPE t IS bit_vector (26 DOWNTO 15);
VARIABLE v1:   t;                    -- v1 requires (26-15)+1=12 bits.

CONSTANT c:   bit_vector (0 TO 2) := "001";    -- c requires 3 bits.
```

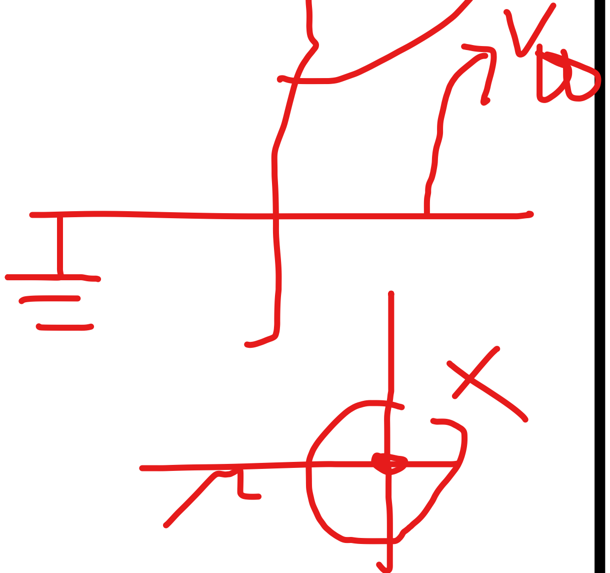
- A bit represents a wire (single-line signal).
- A bit_vector represents a bus (multiple-line signal).

Std_Ulogic Type

- The bit type cannot model real signals.
- IEEE designed a new **package** `std_logic_1164` to fix that.
- Inside the package the type `std_ulogic` is declared.

```
TYPE std_ulogic IS ('U', -- uninitialized, default value
                    'X', -- forcing unknown
                    '0', -- forcing zero
                    '1', -- forcing one
                    'Z', -- high impedance
                    'W', -- weak unknown
                    'L', -- weak zero
                    'H', -- weak one
                    '-'  -- don't care
                   );
```





- **forcing:** signal coming from an active driver.
- **weak:** signal coming from resistive driver.

Std_Logic Type

Type **std_logic** is derived from **std_ulogic**:

```
SUBTYPE std_logic IS RESOLVED std_ulogic;
-- *** industry standard logic type ***


-- unconstrained array of std_ulogic
TYPE std_ulogic_vector IS ARRAY (natural RANGE <>) OF std_ulogic;


-- unconstrained array of std_logic
TYPE std_logic_vector IS ARRAY (natural RANGE <>) OF std_logic;
```

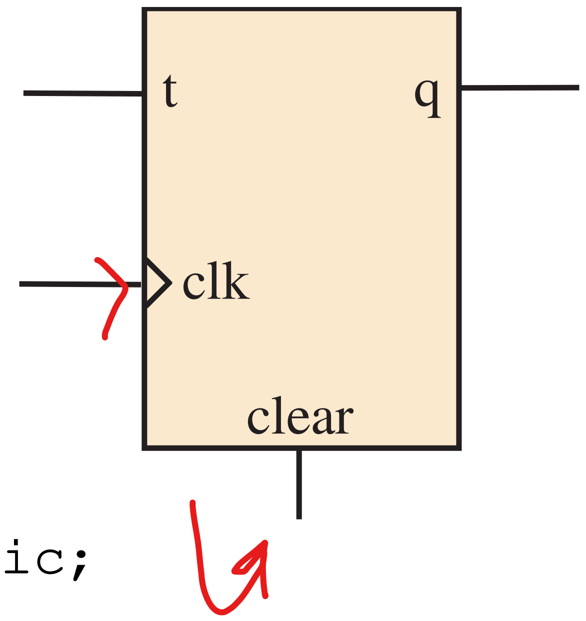
Std_Logic Type (Cont.)

- We can do the following **logical operations** on std_logic's:
  AND OR NOT NOR XOR XNOR
- We can do the following **numerical operations** on std_logic's.
  SLL SRL SLA SRA ROL ROR
- These operations return a std_logic result of
  `'U'` `'X'` `'0'` `'1'`
- The package std_logic_1164 includes the functions **rising_edge** and **falling_edge** to test for edges on signals of type std_logic:
  - rising_edge is true when clock changes from `'0'` or `'L'` to `'1'` or `'H'`.
  - falling_edge is true when clock changes from `'1'` or `'H'` to `'0'` or `'L'`.

Std_Logic Type (Cont.)

Example: 28 (Class) *Model a positive edge-triggered T flip-flop with an asynchronous clear.*

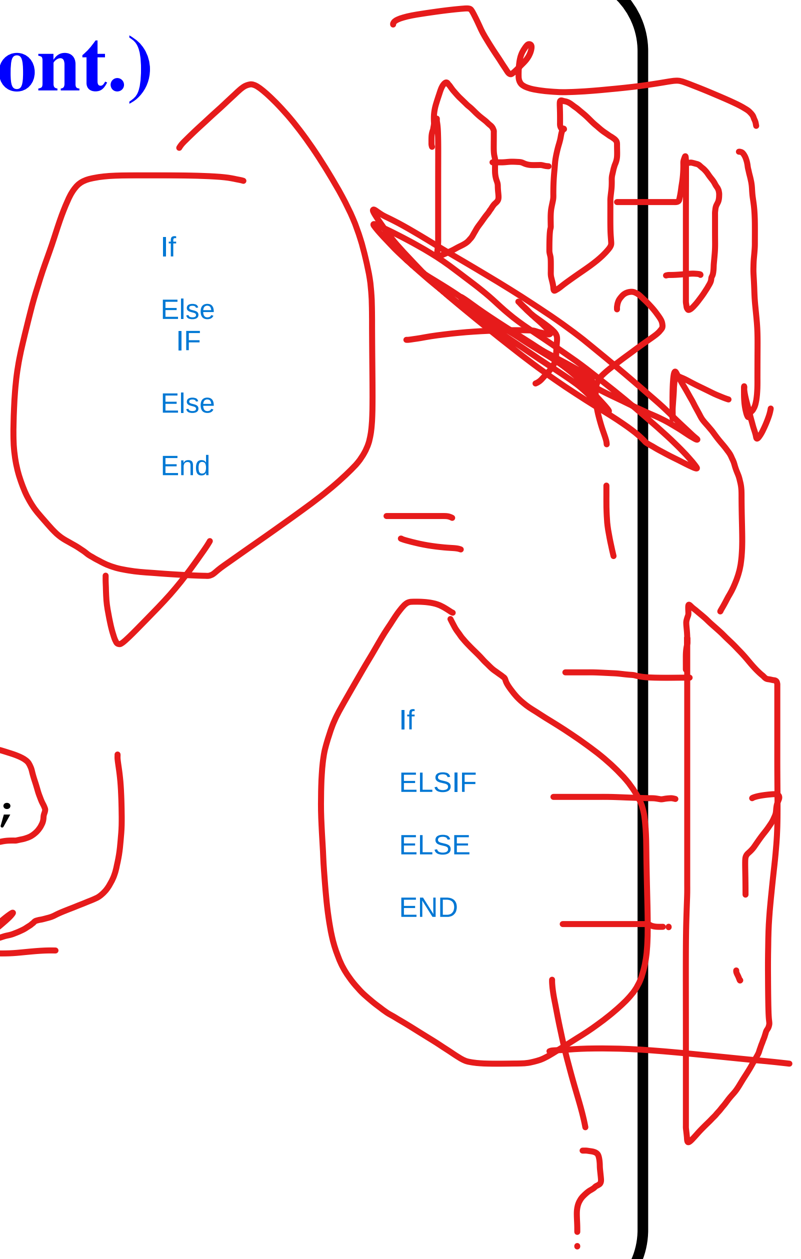
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY t_ff IS
   PORT( t:   IN std_logic;
         clk, clear:   IN std_logic;
         q:   OUT std_logic);
END ENTITY t_ff;
         ⋮
```



Std_Logic Type (Cont.)

Example 28: (Cont.)

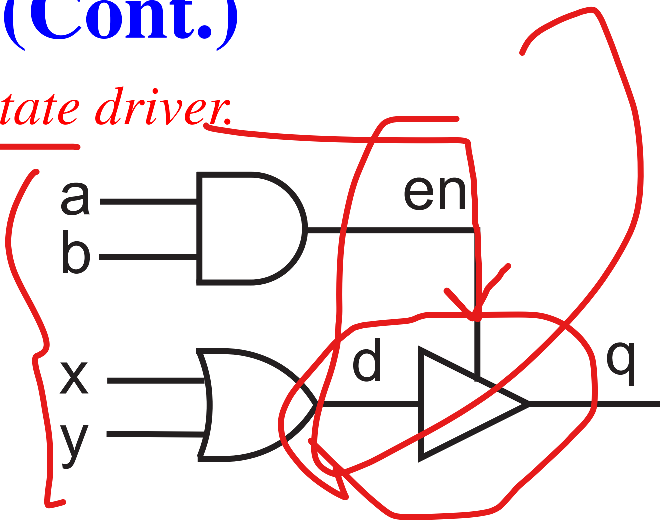
```
         ⋮
ARCHITECTURE async_clear OF t_ff IS
   SIGNAL internal_t: std_logic;
BEGIN
   TFF: PROCESS (clk, clear) IS BEGIN
      IF clear = ’1’ THEN
         internal_t <= ’0’;
      ELSIF rising_edge (clk) THEN
         IF t = ’1’ THEN
            internal_t <= NOT internal_t;
         ELSIF t = ’0’ THEN
            internal_t <= internal_t;
         END IF;
      END IF;
   END PROCESS TFF;
   q <= internal_t;
END ARCHITECTURE async_clear;
```



Std_Logic Type (Cont.)

Example: 29 (Class) *Model the shown tristate driver.*

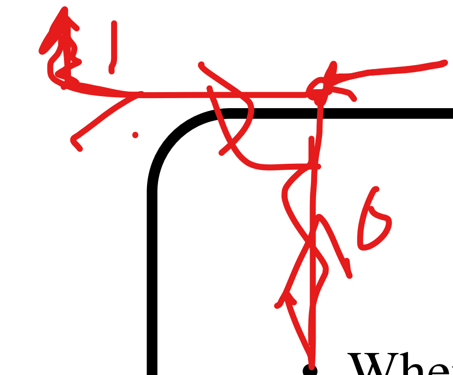
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY and2_std IS
   PORT( a, b:   IN std_logic;
         c:   OUT std_logic);
END ENTITY and2_std;
ARCHITECTURE and2_std OF and2_std IS BEGIN
   p1:  PROCESS (a, b) IS BEGIN
      c <= a AND b;
   END PROCESS p1;
END ARCHITECTURE and2_std;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
PACKAGE and_std_Package IS
   COMPONENT and2_std IS
      PORT( a, b:   IN std_logic;
            c:   OUT std_logic);
   END COMPONENT and2_std;
END PACKAGE and_std_Package;
         :
```



Std_Logic Type (Cont.)

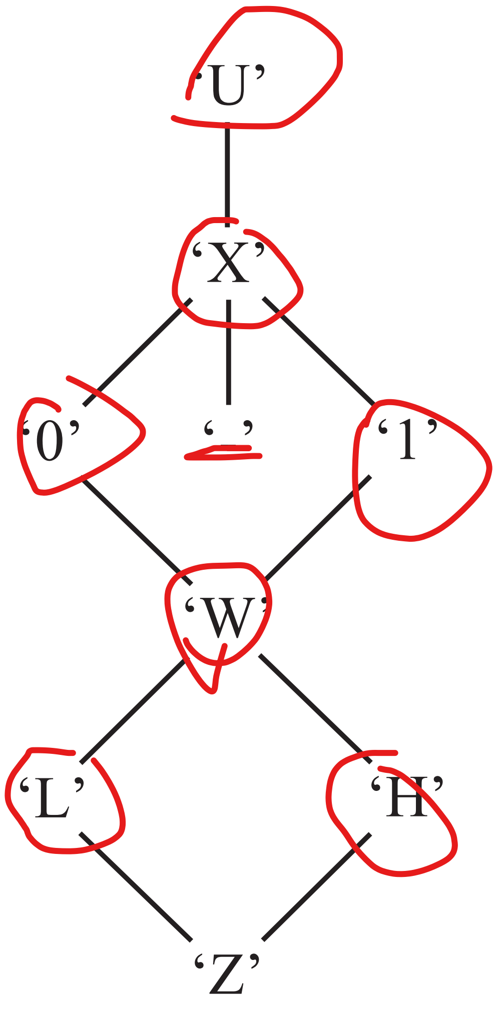
Example 29: (Cont.)

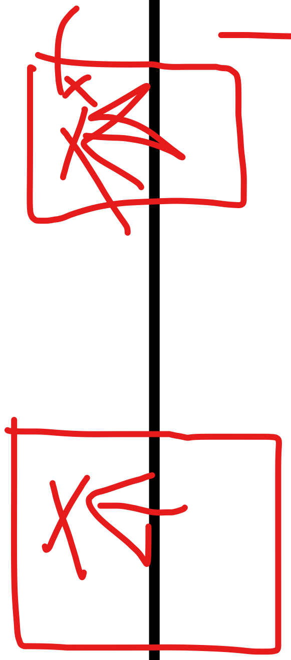
```
        ⋮
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.and_std_Package.ALL;

ENTITY driver IS
   PORT( a, b, x, y:  IN std_logic;
         q:  OUT std_logic);
END ENTITY driver;

ARCHITECTURE tri OF driver IS
   FOR gate:  and2_std USE ENTITY WORK.and2_std (and2_std);
   SIGNAL d, en:  std_logic;
BEGIN
   gate:  and2_std PORT MAP (a, b, en);
   q <= d  WHEN en = '1' ELSE 'Z';
   op:  PROCESS (x, y) IS BEGIN
      d <= x OR y;
   END PROCESS op;
END ARCHITECTURE tri;
```



Std_Logic Resolution

- When two processes try to assign two different values to a single signal, a resolution mechanism is needed to resolve this conflict.
- Conflict between drivers could be resolved using the lattice shown opposite.
- The value nearest the top always wins.
- When the conflict is between two values at the same level (except with `'-'`), the result is the next higher value.
- Any conflict with `'-'` results in `'X'`, except when `'-'` conflicts with `'U'` the result is `'U'`.





Std_Logic Resolution (Cont.)

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

Std_Logic Resolution (Cont.)

- Inside the IEEE `std_logic_1164` package, the resolution_table constant is declared.

```
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
----------------------------------------------------------------------
-- resolution function
----------------------------------------------------------------------
CONSTANT resolution_table :  stdlogic_table := (
--       -------------------------------------------------------------
--       |  U    X    0    1    Z    W    L    H    -          |    |
--       -------------------------------------------------------------
         ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
         ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
         ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
         ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
         ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
         ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
         ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
         ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
         ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) --  | - |
         );
```

Std_Logic Resolution (Cont.)

Example: 30 (Home) *Show the waveform of signal a generated from the following architecture.*

```
ARCHITECTURE dataflow OF s IS
   SIGNAL a:  std_logic;
BEGIN
   a <= '0';
   a <= '1' AFTER 2 ns;
END ARCHITECTURE dataflow;
```

Rule: Do not allow more than one process, concurrent signal assignment, or component instance to drive (assign values) to the same signal. Otherwise, a resolution function will be required.

Example: 31 (email) *Simulate the RS-flip-flop model below by applying the operations: set, no change, reset, no change, and set, in sequence. Explain the simulation results. How would the output look like if only one of the two processes is included. Suggest how to fix any problem in the code with minimal changes.*

Std_Logic Resolution (Cont.)

Example 31: (Cont.)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY rs_ff IS
   PORT( r, s:  IN std_logic;
         q:  OUT std_logic);
END ENTITY rs_ff;
ARCHITECTURE wrong OF rs_ff IS BEGIN
   ps:  PROCESS (r, s) IS BEGIN
      IF (  s = '1' AND r = '0') THEN
            q <= '1';
      END IF;
   END PROCESS ps;

   pr:  PROCESS (r, s) IS BEGIN
      IF (  s = '0' AND r = '1') THEN
            q <= '0';
      END IF;
   END PROCESS pr;
END ARCHITECTURE wrong;
```

Signed and Unsigned Types

- VHDL does not enable us to access bits of integer or real data.
- **IEEE 1076.3 standard** defines two functionally equivalent packages:
  1. **numeric_bit**: based on the bit array type
  2. **numeric_std**: based on the std_logic array type
- Each of these packages defines two types, unsigned and signed, to represent unsigned and signed integer values, respectively:
  - **unsigned** is interpreted as a positive integer.
  - **signed** is interpreted as 2's complement number.
- Signed and unsigned data types enable us to access and manipulate bits of numerical data and, thus, simulate arithmetic hardware.
- Signed and unsigned types and operations are suitable for synthesis.

Signed and Unsigned Types

- Each of these packages defines the following arithmetic operations on integers represented using vectors of **bit** and **std_logic** elements, respectively: arithmetic, comparison, conversion, shift and rotate, resize, clock edge detection, and logical.
- Include appropriate package:

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;
```

Signed and Unsigned Types (Cont.)

- In the case of **numeric_bit** package:

```
TYPE unsigned IS ARRAY (natural RANGE <>) OF bit;
TYPE signed IS ARRAY (natural RANGE <>) OF bit;
```

- In the case of **numeric_std** package:

```
TYPE unresolved_unsigned IS ARRAY (natural RANGE <>) OF std_ulogic;
TYPE unresolved_signed IS ARRAY (natural RANGE <>) OF std_ulogic;

ALIAS u_unsigned IS unresolved_unsigned;
ALIAS u_signed IS unresolved_signed;

SUBTYPE unsigned IS (RESOLVED) unresolved_unsigned;
SUBTYPE signed IS (RESOLVED) unresolved_signed;
```

- An **alias** declaration defines an alternate identifier to refer to the named data object. We can refer to the object using the new identifier.

Signed and Unsigned Types (Cont.)

Examples of signed and unsigned types:

```
LIBRARY ieee;
USE ieee.numeric_bit.ALL;

SIGNAL sig:  signed (0 TO 7);          -- sig(0) is sign bit
SIGNAL sgp:  signed (3 DOWNTO 0);      -- sgp(3) is sign bit
SIGNAL ugi:  unsigned (0 TO 15);       -- ugi(0) is MSB
SIGNAL ugp:  unsigned (31 DOWNTO 0);  -- ugp(31) is MSB
```

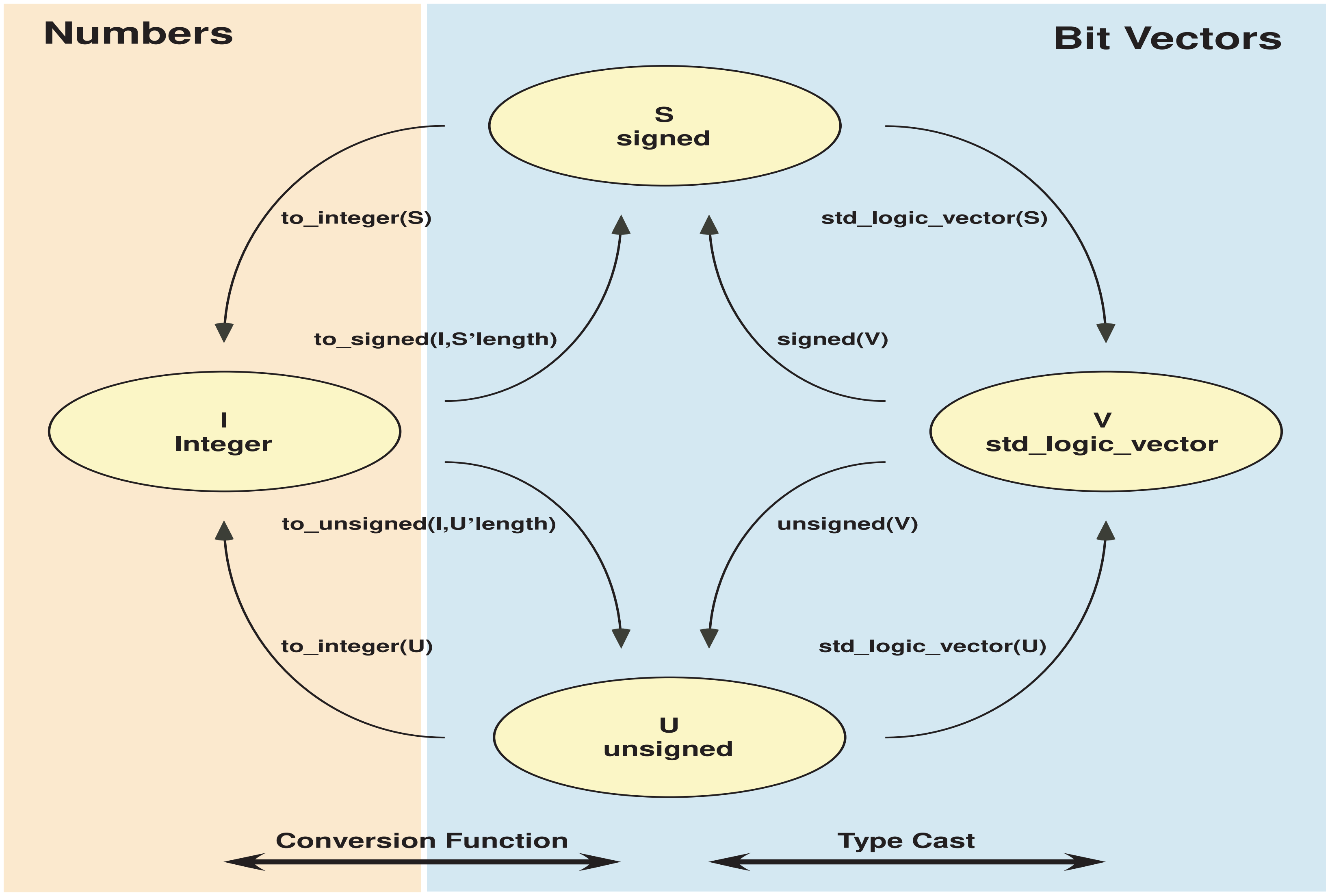
Some useful functions are also defined on signed and unsigned data type:

```
FUNCTION "+" (L, R: signed) RETURN signed;
FUNCTION "*" (L: signed, R: signed) RETURN signed;
FUNCTION "<" (L, R: signed) RETURN boolean;
```

Signed and Unsigned Conversion Functions

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

SIGNAL u:  unsigned (7 DOWNTO 0);
SIGNAL s:  signed (7 DOWNTO 0);
SIGNAL v:  std_logic_vector (7 DOWNTO 0);
SIGNAL n:  integer;

n <= to_integer (u);             n <= to_integer (s);
u <= to_unsigned (n, 8);         s <= to_signed (n, 8);
u <= unsigned (s);               s <= signed (u);
u <= unsigned (v);               s <= signed (v);
v <= std_logic_vector (u);       v <= std_logic_vector (s);
```

Signed and Unsigned Conversion Functions



Signed and Unsigned Conversion Functions (Cont.)

- The `std_logic_arith` library defines extra type conversion functions for representing integers in standard ways.

- The `conv_integer` function from the std_logic_arith library converts its argument (signed, unsigned, or std_logic) to an integer.
- The conv_integer function provided by the std_logic_arith library cannot convert a std_logic_vector to an integer because it is impossible to determine if it represents an unsigned or signed value.
- The conv_integer functions included in the `std_logic_unsigned` and `std_logic_signed` libraries can convert a std_logic_vector to an integer.

Signed and Unsigned Conversion Functions (Cont.)

- The `conv_std_logic_vector` function from the std_logic_arith library converts its argument (integer, signed, unsigned, or std_logic) to a std_logic_vector value with a specific size in bits.
- If conv_std_logic_vector argument is unsigned or positive, it is treated as an unsigned value; if it is negative, it is converted to 2's complement signed form.
- The `conv_unsigned` function from the std_logic_arith library converts its argument (integer, signed, or std_logic) to a unsigned value with a specific size in bits.
- The `conv_signed` function from the std_logic_arith library converts its argument (integer, unsigned, or std_logic) to a 2's complement signed value with a specific size in bits.

Signed and Unsigned Conversion Functions (Cont.)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

SIGNAL u:  unsigned (7 DOWNTO 0);
SIGNAL s:  signed (7 DOWNTO 0);
SIGNAL v:  std_logic_vector (7 DOWNTO 0);
SIGNAL n:  integer;

n <= conv_integer (u);          n <= conv_integer (s);
n <= conv_integer (v);
v <= conv_std_logic_vector (u, 8);
v <= conv_std_logic_vector (s, 8);
v <= conv_std_logic_vector (n, 8);
u <= conv_unsigned (s, 8);      u <= conv_unsigned (n, 8);
s <= conv_signed (u, 8);        s <= conv_signed (n, 8);
```

Bits and Bytes

We have several options for defining a byte or groups of bits:

1. `TYPE byte IS ARRAY (7 DOWNTO 0) OF bit;`
2. `SIGNAL byte:  bit_vector (7 DOWNTO 0);`
3. `SIGNAL byte:  std_logic_vector (7 DOWNTO 0);`
4. `SIGNAL byte:  unsigned (7 DOWNTO 0);`
5. `SIGNAL byte:  signed (7 DOWNTO 0);`

Aggregates

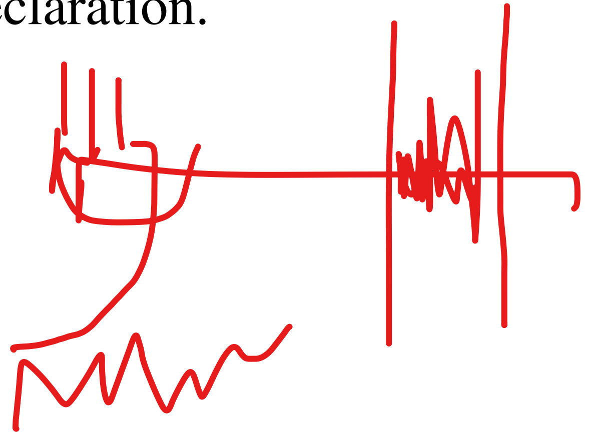
- An aggregate is a list of expressions, separated by commas and enclosed in brackets.
- The expressions in an aggregate can include operators and even other aggregates, so long as they yield a value of the type of the array.
- An array aggregate can contain an **OTHERS** association, as the last association in the aggregate. OTHERS supplies a value for all of the elements that have not been explicitly associated in the aggregate.

```
TYPE bcd5 IS ARRAY (4 DOWNTO 0) OF bit_vector (3 DOWNTO 0);
VARIABLE u:  bcd5 := ("1001", "1000", "0111", "0110", "0101");
VARIABLE v:  bcd5 := ("1001", "1000", OTHERS => "0000");
VARIABLE w:  bcd5 := (3 => "0110", 0 => "1001", OTHERS => "0000");
VARIABLE x:  bcd5 := (OTHERS => "0000");
VARIABLE a:  bit_vector (3 DOWNTO 0) := (OTHERS => '1');
```

Slicing

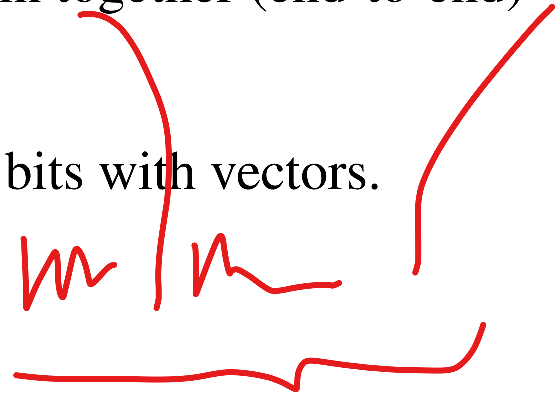
- The name of a vector denotes the entire vector~~, e.g., v.~~
- An indexed name denotes one element of a vector, e.g., v(1).
- A slice name denotes part of vector with contiguous index values, e.g., v(2 TO 5).
- Slice names can be used to read or assign part of a vector.
- The direction of the slice name (TO or DOWNTO) has to be consistent with the direction given in the declaration.

```
VARIABLE v:  bit_vector (0 TO 7);
VARIABLE f:  bit_vector (3 DOWNTO 0);
         .
         .
         .
   f := v (2 TO 5);
   f (1 DOWNTO 0) := "11";
   v (0 TO 1) := f (3 DOWNTO 2);
```



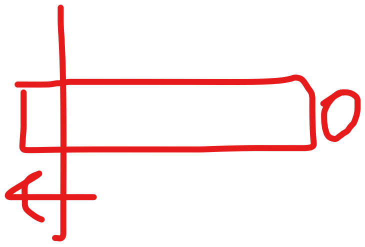
Concatenation

- The concatenation operator & is used to join together (end-to-end) two vectors to form a single longer vector.
- We can also concatenate bits with bits and bits with vectors.



```
SIGNAL a, b:  bit_vector (7 DOWNTO 0);
SIGNAL f, g:  bit_vector (15 DOWNTO 0);
        ⋮
   f <= a & b;
   g <= '0' & a(7) & b(6 DOWNTO 1) & b(7) & a(6 DOWNTO 0);
```

Example: 32 (Individual) *Show how slicing and concatenation could be used together to implement a logical shift left.*



```
SIGNAL a:  bit_vector (7 DOWNTO 0);
        ⋮
a <= a (6 DOWNTO 0) & '0';
```

Record Type

A record consists of multiple elements that may be of different types.

```
TYPE my_module IS
    RECORD
        rise_time:  time;
        size:  integer RANGE 0 TO 200;
        data:  bit_vector (15 DOWNTO 0);
    END RECORD my_module;

SIGNAL a, b:  my_module;
```

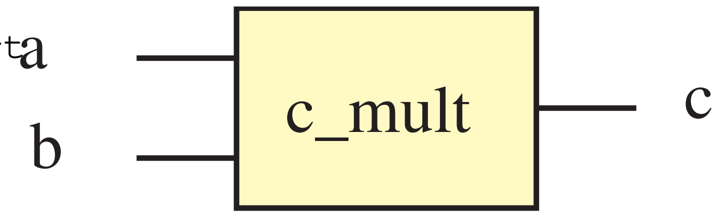
To access values or assign values to records, one can use one of the following methods:

```
a.rise_time <= 5 ns;
b <= a;
```

Record Type (Cont.)

Example: 33 (Class) *Model a complex number multiplier.*

```
PACKAGE my_types IS
   TYPE complex IS RECORD
      r_part:  integer; -- real part
      i_part:  integer; -- imaginary part
   END RECORD;
END PACKAGE my_types;

USE WORK.my_types.ALL;
ENTITY c_mult IS
   PORT( a, b:  IN complex;
         c:  OUT complex);
END ENTITY c_mult;
ARCHITECTURE c_mult OF c_mult IS
BEGIN
   c.r_part <= a.r_part * b.r_part - a.i_part * b.i_part;
   c.i_part <= a.r_part * b.i_part + a.i_part * b.r_part;
END ARCHITECTURE c_mult;
```



Declaring a New Type

```
TYPE twos_complement_single IS RANGE -128 TO 127;
TYPE twos_complement_double IS RANGE 255 DOWNTO -256;
```

Typically, we lump all our type declarations in a package that we use later.

Example: 34 (Individual) *Declare a small_int type, taking values in the range 0 to 255. Show the entity declaration for a small_int adder.*

```
PACKAGE my_types IS
   TYPE small_int IS RANGE 0 TO 255;
END PACKAGE my_types;

USE WORK.my_types.ALL;
ENTITY small_adder IS
   PORT( a, b:  IN small_int;
         s:  OUT small_int);
END ENTITY small_adder;
```

Subtypes

- Sometimes we need to assign values to objects from a restricted range of a certain type.
- **Subtype** defines a restricted set of values of a **base type**.

Example: 35 (Individual) *Declare an eight_bit subtype, taking values in the range -128 to 127. Show how to divide two numbers of this subtype, assuming non-zero divisors.*

```
PACKAGE new_types IS
   SUBTYPE eight_bit IS integer RANGE -128 TO 127;
END PACKAGE new_types;
USE WORK.new_types.ALL;
ENTITY eight_bit_divider IS
   PORT( a, b:  IN eight_bit;
         q:  OUT eight_bit);
END ENTITY eight_bit_divider;
ARCHITECTURE divide OF eight_bit_divider IS BEGIN
   q <= a/b;
END ARCHITECTURE divider;
```

Object Visibility Rules

The visibility of each VHDL object is determined by the place in which it is declared:

- A signal, a constant, a type, or a subtype declared or defined in a package are visible in all design units which use this package.
- A port signal, a constant, a type, or a subtype declared or defined in an entity are visible in all architectures assigned to this entity.
- A signal, a constant, a type, or a subtype declared or defined in an architecture are visible only inside this architecture.
- A variable, a constant, a type, or a subtype declared or defined in a process are visible only inside this process.

Object Visibility Rules (Cont.)

Example: 36 (Class) *The package below includes definitions of some constants and subtypes. The entity mem uses the defined package.*

```
PACKAGE my_package IS
   CONSTANT word_size : positive := 32;
   SUBTYPE address_bus IS bit_vector (word_size-1 DOWNTO 0);
   SUBTYPE data_bus IS bit_vector (word_size-1 DOWNTO 0);
END PACKAGE my_package;

USE WORK.my_package.ALL;
ENTITY mem IS
   PORT( addr:  IN address_bus;
         d_in:  IN data_bus;
         d_out:  OUT data_bus);
END ENTITY mem;
         ⋮
```

Object Visibility Rules (Cont.)

```
        ⋮
ARCHITECTURE sample OF mem IS
   CONSTANT pi:  real := 3.14159;
BEGIN
   p1:  PROCESS (sensitivity_list) IS
      VARIABLE counter:  integer;
   BEGIN
      -- statements using pi, counter, addr, d_in, d_out
   END PROCESS p1;
END ARCHITECTURE sample;
```

- Types address_bus and data_bus are seen by all design units that use package my_package.
- Constant word_size is seen by all design units that use package my_package.
- Constant pi is seen by all processes and statements inside architecture sample.
- Variable counter is seen only by process p1.

Declarative Part Usage

	entity	architecture	process	package	package body	configuration
configuration		X				
use clause	X	X		X	X	X
component		X		X		
variable			X			
signal	X	X		X	X	
constant	X	X	X	X	X	
type and subtype	X	X	X	X	X	
subprogram declaration	X	X	X	X	X	
subprogram body	X	X	X		X	
generic	X	X		X		