VHDL

- **Agenda**

1. VHDL History
2. VHDL Systems and Signals
3. VHDL Entities, Architectures, and Packages
4. VHDL Data Types
5. VHDL Operators
6. VHDL Structural Design
7. VHDL Behavioral Design
8. VHDL Test Benches

VHDL History

- **VHDL**

  V = Very High Speed Integrated Circuit
  H = Hardware
  D = Description
  L = Language

  - Originally a Department of Defense sponsored project in the 80's
  - Original Intent was to Document Behavior (instead of writing system manuals)
  - Original Intent was NOT synthesis, that came later
  - Simulation was a given, since the designs were already in text and we had text compilers
  - Designed by IBM, TI, Intermetrics (all sponsored by DoD)

VHDL History

- **VHDL & IEEE**

  - In 1987, IEEE published the "VHDL Standard"

    - IEEE 1076-1987 = First formal version of VHDL
    - Strong "Data Typing"

      - each signal/variable is typed (bit, bit_vector, real, integer)

      - assignments between different types NOT allowed

    - Did not handle multi-valued logic

VHDL History

- **VHDL & IEEE**

  - What is multi-valued logic?

    - when there are more possible values than 0 and 1

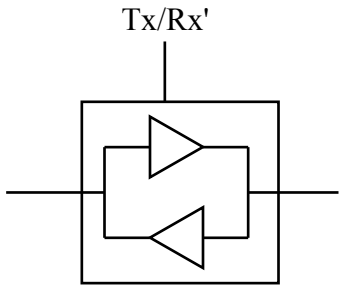
    - we need this for real world systems such as buses

      - a bus is where multiple circuits drive and receive information
      - only one agent drives the bus (low impedance)
      - all other agents listen (high impedance)

    - how can something drive AND receive?

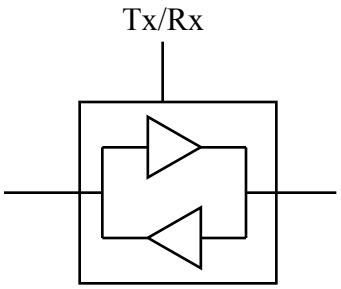
    - a "transceiver" has both a transmit (i.e., a gate facing out) and receive (i.e., a gate facing in)

    - we can draw it as follows:

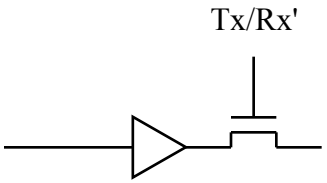


VHDL History

- VHDL & IEEE
  - What is multi-valued logic?
    - but that circuit doesn't actually work because the driving gate will always be driving?



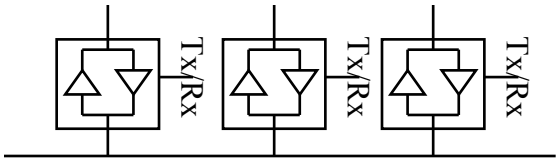
    - in reality it looks like this:



    - what does this look like when it is "OFF"? High Impedance

VHDL History

- VHDL & IEEE
  - High Impedance



    - it is how circuits behave, strong drivers will control the bus when everyone is High-Z
    - When nobody is driving the bus, the bus is High-Z
    - So for true behavior, VHDL has to model High-Z
    - VHDL's built in types (bit and bit_vector) can only be 0 or 1, these don't cut it.
  - Weak/Strong
    - Some busses have multiple drivers but some are weaker than others (i.e., MCAN)?
    - We should model these too

VHDL History

- **VHDL & IEEE**

  - VHDL allows users to come up with their own data types. Since the world needed multi-valued logic, everyone started creating their own add-on packages.

  - this created a lot of confusion when multiple vendors worked together (i.e., Fab Shop and Designer)

  - In 1993, IEEE published an Upgrade

    - IEEE 1164 - added support for Multi-Valued Logic through the "STD_LOGIC" package
      - better syntax consistency

  - Every time there is a need for a data type, industry will start to create add-ons. Then IEEE will create a standard to reduce confusion

  - Other package standards that were added to VHDL

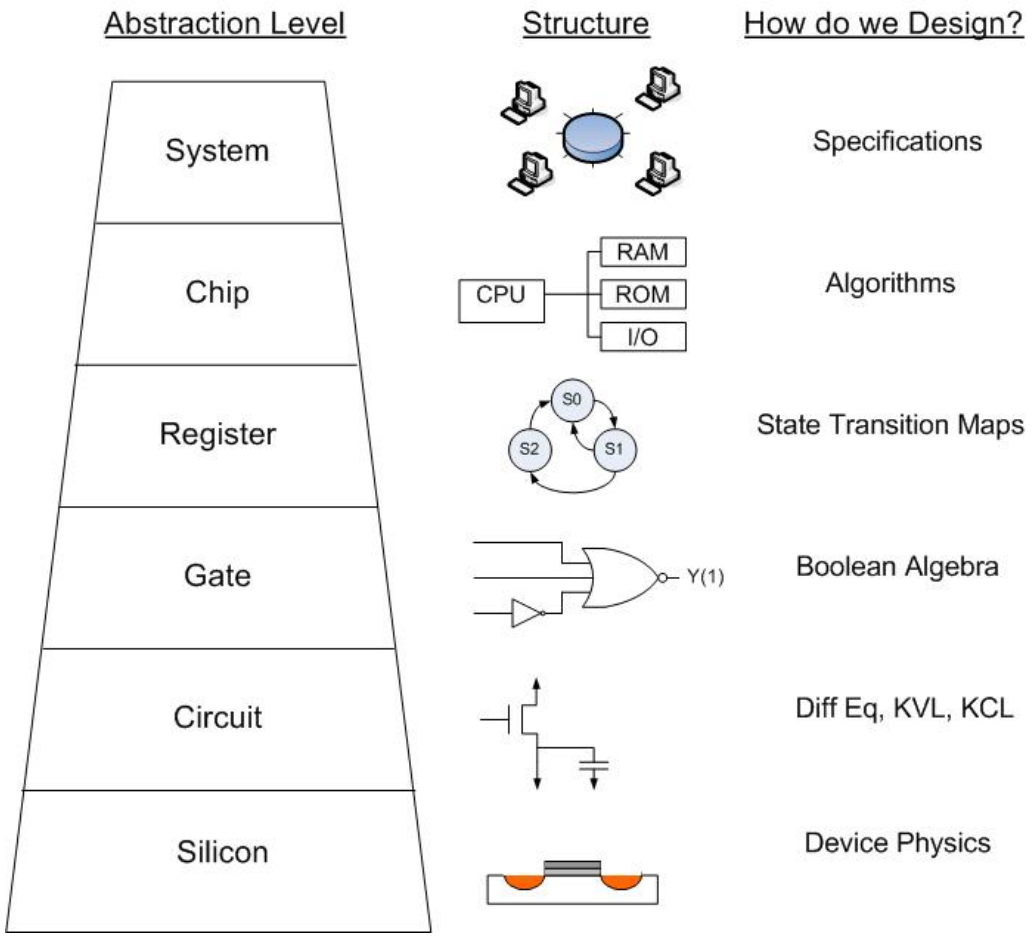
    - 1076.2 = "Real and Complex Data Types"
    - 1076.3 = "Signed and Unsigned Data Types"

  - The last rev of VHDL in 2003 (1076.3) is considered by most to be the more recent major relea

  - Although people are talking about VHDL 2006 (which now has turned into VHDL 200x)

VHDL Design

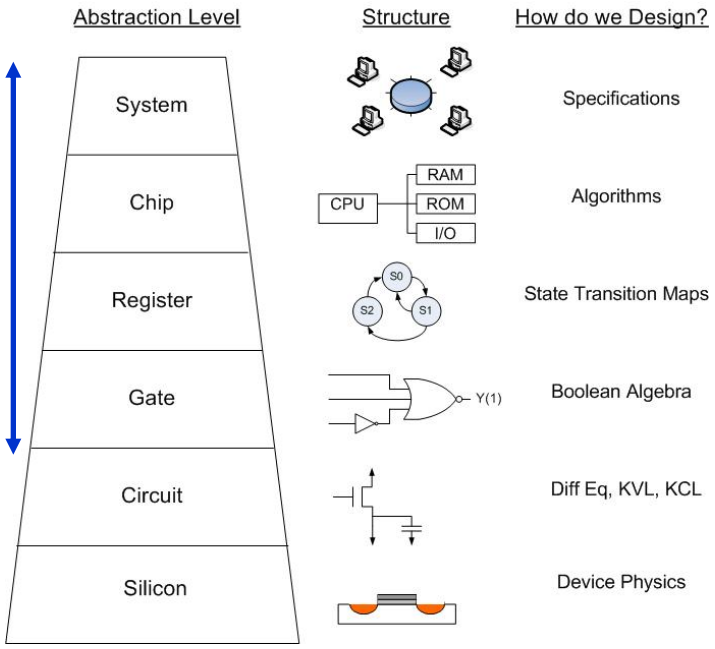
- At What level can we design?



VHDL Design

- **What does abstraction give us?**
  - The higher in abstraction we go, the more complex & larger the system becomes
  - But, we let go over the details of how it performs (speed, fine tuning)
  - There are engineering jobs at each level
  - Guru's can span multiple levels
- What does VHDL model?
  - System : Chip : Register : Gate
  - VHDL let's us describe systems in two ways:

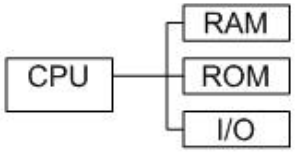
    1) Structural (text netlist)
    2) Behavioral (requires synthesis)



VHDL Systems and Signals

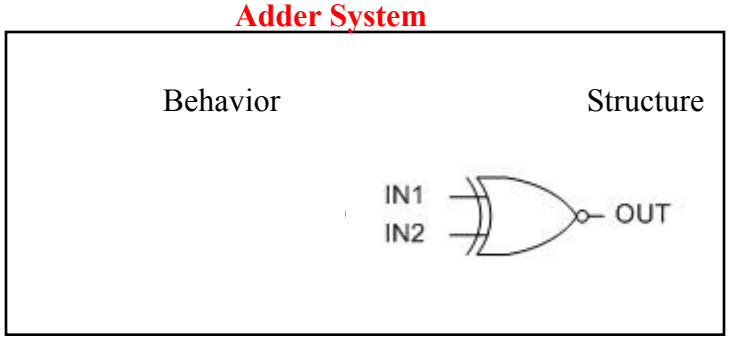
- **Systems**

  - The world is made up of systems communicating with each other



  - Systems are made up of other Systems

  - A System has a particular "Behavior" and "Structure"

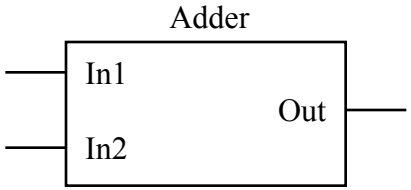


  - We can describe an "Adder" system in multiple ways and at multiple levels of abstraction

VHDL Systems and Signals

- **System Interface**

  - We must first describe the system's Interface to connect it to other systems



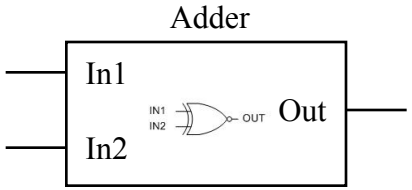
  - An "Interface" is a description of the Inputs and Outputs

  - We also call these "Ports"

VHDL Systems and Signals

- **System Behavior**

  - We then must describe the system's behavior (or functionality)



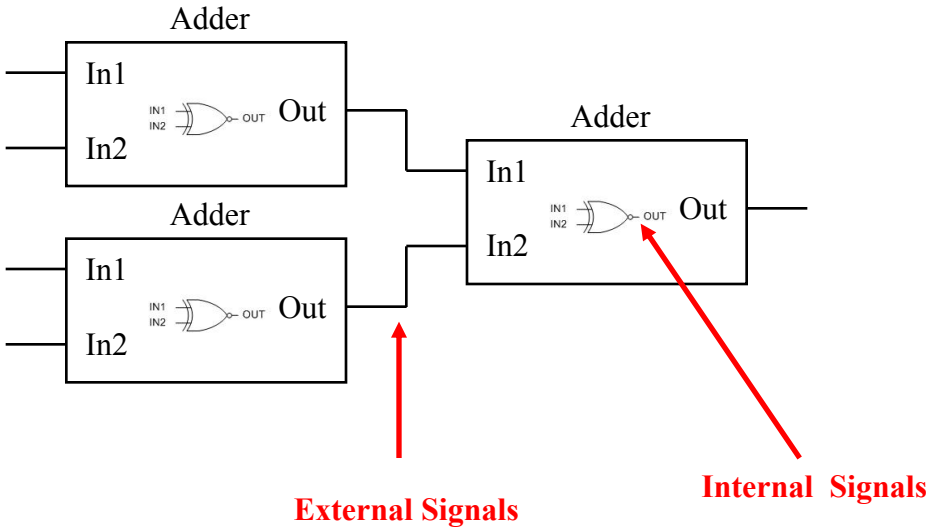
  - There are many ways to describe the behavior in VHDL

  - When describing a system, we must always describe its:

    1) Interface
    2) Behavior

VHDL Systems and Signals

- **Signals**
  - Multiple Systems communicate with each other using signals



VHDL Entity

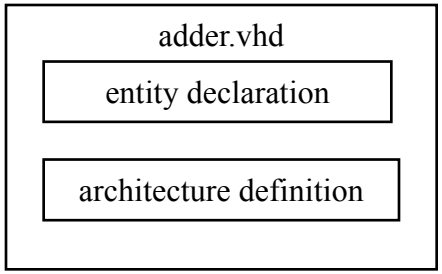
- VHDL

Entity
- used to describe a system's interface
- we call the Inputs and Outputs "Ports"
- creating this in VHDL is called an "Entity Declaration"

Architecture
- used to describe a system's behavior (or structure)
- separate from an entity
- an architecture must be tied to an entity
- creating this in VHDL is called an "Architecture Definition"

Syntax Details we'll follow:

- we put the entity and architecture together in one text file
- we name the text file with the system name used in the entity
- the post fix for VHDL is *.vhd



VHDL Entity

- **More Syntax Notes**

  - VHDL is NOT case sensitive
  - Comment text is proceeded with "--"
  - Names must start with an alphabetic letter (not a number)
  - Names can include underscore, but not two in a row (i.e., __) or as the last character.
  - Names cannot be keywords (in, out, bit, ….)

VHDL Entity

- **Entity Details**

  - an entity declaration must possess the following:

    1) entity-name — user selected, same as text file

    2) signal-names — user selected

       - mode — direction of signal (in, out, buffer, inout)

    3) signal-type — what type of data is it?
       (bit, STD_LOGIC, real, integer, signed,…)

       - this is where VHDL is strict!

       - we say it is a "strong type cast" language

       - there are built in (or pre-defined) types

         (bit, bit_vector, boolean, character, integer, real, string, time)

       - we can add more types for realistic behavior (i.e., buses)

VHDL Entity

- **Entity Syntax**

```
entity  entity-name  is

        port (signal-name   :   mode   signal-type;
              signal-name   :   mode   signal-type;

              signal-name   :   mode   signal-type);

end entity entity-name;
```

NOTES:

- the keywords are entity, is, port, end
- multiple signal-names with the same type can be comma delimited on the same line
- the port definition is contained within parenthesis
- each signal-name line ends with a ";"

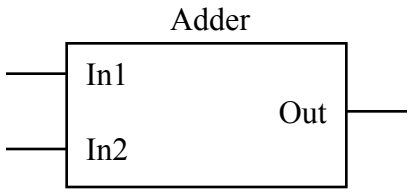
except

- the last line (watch the ");" at the end, this will get you every time!)

VHDL Entity

- **Entity Example**

```
entity  adder  is

        port (In1, In2   :   in   bit;
              Out        :   out   bit);

end entity adder;
```



NOTES: – we can also put "Generics" within an entity, which are dynamic variables

ex) generic (BusWidth : Integer := 8);

more on generics later….

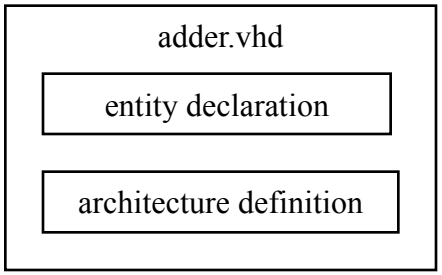
VHDL Entity

- **Systems in VHDL**

  - Systems need to have two things described

    1) Interface (I/O, Ports…)
    2) Behavior (Functionality, Structure)

  - In VHDL, we do this using entity and architecture



Entity
- used to describe a system's interface
- we call the Inputs and Outputs "Ports"
- creating this in VHDL is called an "Entity Declaration"

Architecture
- used to describe a system's behavior (or structure)
- separate from an entity
- an architecture must be tied to an entity
- creating this in VHDL is called an "Architecture Definition"

VHDL Architecture

- **Architecture Details**

  - an architecture is always associated with an entity (in the same file too)

  - an architecture definition must possess the following:

    1) architecture-name - user selected, different from entity
       - we usually give something descriptive (adder_arch, and 2_arch)
       - some companies like to use "behavior", "structural" as the names

    2) entity-name - the name of the entity that this architecture is associated with
       - must already be declared before compile

    3) optional items…
       - types
       - signals : internal connections within the architecture
       - constants
       - functions : calling predefined blocks
       - procedures : calling predefined blocks
       - components : calling predefined blocks

    4) end architecture - keywords to signify the end of the definition
       - we follow this by the architecture name and ";"

VHDL Architecture

- **Architecture Syntax**

architecture *architecture-name* of *entity-name* is

type…
signal…
constant…
function…
procedure…
component…

begin

…behavior or structure

end architecture *architecture-name*;

NOTE: - the keywords are architecture, of, is, type…component, begin, end
- there is a ";" at the end of the last line

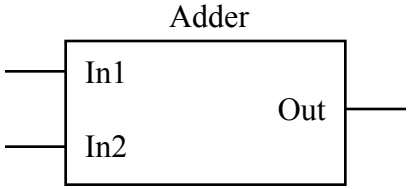
VHDL Architecture

- **Architecture definition of an AND gate**

```
architecture  and2_arch of and2 is

begin
        Out1 <= In1 and In2;

end architecture and2_arch;
```

- **Architecture definition of an ADDER**

```
architecture  adder_arch of adder is

begin
        Out1 <= In1 + In2;

end architecture adder_arch;
```



VHDL Packages

- **VHDL is a "Strong Type Cast" language…**

  - this means that assignments between different data types are not allowed.

  - this means that operators must be defined for a given data types.

  - this becomes important when we think about synthesis

    ex) string + real = ???

    - can we add a string to a real?
    - what is a "string" in HW?
    - what is a "real" in HW?

  - VHDL has built-in features:

    1) Data Types
    2) Operators

  - built-in is also called "pre-defined"

VHDL Packages

- **Pre-defined Functionality**

  ex) there is a built in addition operator for integers

  integer + integer = integer

  - the built-in operator "+" works for "integers" only
  - it doesn't work for "bits" as is

- **Adding on Functionality**

  - VHDL allows us to define our own data types and operators
  - a set of types, operators, functions, procedures… is called a "Package"
  - A set of packages are kept in a "Library"

VHDL Packages

- **IEEE Packages**

  - when functionality is needed in VHDL, engineers start creating add-ons using Packages

  - when many packages exist to perform the same function (or are supposed to) keeping consistency becomes a problem

  - IEEE publishes "Standards" that give a consistent technique for engineers to use in VHDL

  - we include the IEEE Library at the beginning of our VHDL code

    syntax: | library *library-name*

  - we include the Package within the library that we want to use

    syntax: | use *library-name.package.function*

  - we can substitute "ALL" for "function" if we want to include everything

VHDL Packages

- **Common IEEE Packages**

  - in the IEEE library, there are common Packages that we use:

    STD_LOGIC_1164
    STD_LOGIC_ARITH
    STD_LOGIC_SIGNED

Ex)

```
library  IEEE;
use      IEEE.STD_LOGIC_1164.ALL;
use      IEEE.STD_LOGIC_ARITH.ALL;
use      IEEE.STD_LOGIC_SIGNED.ALL;
```

  - libraries are defined before the entity declaration

VHDL Design

- **Let's Put it all together now…**

```
library  IEEE;                                          -- package
use      IEEE.STD_LOGIC_1164.ALL;
use      IEEE.STD_LOGIC_ARITH.ALL;
use      IEEE.STD_LOGIC_SIGNED.ALL;

entity  and2  is                                        -- entity declaration

    port (In1, In2       :   in     STD_LOGIC;
          Out1           :   out    STD_LOGIC);

end entity and2;

architecture  and2_arch of and2 is                              -- architecture definition

  begin
    Out1 <= In1 and In2;

end architecture and2_arch;
```

VHDL Design

- **Another Example…**

```
library  IEEE;                                                    -- package
use      IEEE.STD_LOGIC_1164.ALL;

entity  inv1  is                                                  -- entity declaration

    port (In1                    :   in     STD_LOGIC;
          Out1          :   out   STD_LOGIC);

end entity inv1;

architecture inv1_arch of inv1 is                                 -- architecture definition

  begin
    Out1 <= not In1;

end architecture inv1_arch;
```

- **The Pre-defined features of VHDL are kept in the STANDARD library**
  - but we don't need to explicitly use the STANDARD library, it is automatic

VHDL Data Types

- **Signals**

  - a single bit is considered a Scalar quantity

  - a bus (or multiple bits represented with one name) is called a Vector

  - in VHDL, we can define a signal bus as:

```
data_bus   : in    bit_vector (7 downto 0);   -- we will use "downto“ for descending order
                or
data_bus   : in    bit_vector (0 to 7);          --ascending order
```

  - the Most significant Bit (MSB) is ALWAYS on the left of the range description:

```
ex) data_bus : in    bit_vector (7 downto 0);
                                        data_bus(7) = MSB

ex) data_bus : in    bit_vector (0 to 7);
                                        data_bus(0) = MSB
```

VHDL Data Types

- **Signals**

  - there are "Internal" and "External" signals

  Internal - are within the Entity's Interface

  External - are outside the Entity's Interface and connect it to other systems

VHDL Data Types

- **Scalar Data Types (Built into VHDL)**

  - scalar means that the type only has one value at any given time

  Boolean
  - values {TRUE, FALSE}
  - not the same as '0' or '1'

  Character
  - values are all symbols in the 8-bit ISO8859-1 set (i.e., Latin-1)
  - examples are '0', '+', 'A', 'a', '\'

  Integer
  - values are whole numbers from -2,147,483,647 to +2,147,483,647
  - the range comes from +/- 2^{32}
  - examples are -12, 0, 1002

  Real
  - values are fractional numbers from -1.0E308 to +1.0E308
  - examples are 0.0, 1.134, 1.0E5

  Bit
  - values {'0', '1'}
  - different from Boolean
  - this type can be used for logic gates
  - single bits are always represented with single quotes (i.e., '0', '1')

VHDL Data Types

- **Array Data Types (Built into VHDL)**

  - array is a name that represents multiple signals

  Bit_Vector
  - vector of bits, values {'0', '1'}
  - array values are represented with double quotes (i.e., "0010")
  - this type can be used for logic gates

  ex) Addr_bus : in BIT_VECTOR (7 downto 0);

  - unlimited range
  - first element of array has index=0 (i.e., Addr_bus(0)…)

  String
  - vector of characters, values{Latin-1}
  - again use double quotes
  - define using "to" or "downto" ("to" is easier for strings)

  ex) Message : string (1 to 10) := "message here…"

  - first element in array has index=1, this is different from BIT_VECTOR

VHDL Data Types

- **Physical Data Types (Built into VHDL)**

  - these types contain object value and unites
  - NOT synthesizable

  Time — range from -2,147,483,647 to +2,147,483,647
  - units: fs, ps, ns, us, ms, sec, min, hr

- **User-Defined Enumerated Types**

  - we can create our own descriptive types, useful for State Machine
  - no quotes needed

  ex) type States is (Red, Yellow, Green);

VHDL Operators

- **VHDL Operators**
  - Data types define both "values" and "operators"
  - There are "Pre-Determined" data types

    Pre-determined = Built-In = STANDARD Package
  - We can add additional types/operators by including other Packages
  - We'll first start with the STANDARD Package that comes with VHDL

VHDL Operators

- **Logical Operators**

  - works on types BIT, BIT_VECTOR, BOOLEAN

  - vectors must be same length

  - the result is always the same type as the input

    not
    and
    nand
    or
    nor
    xor
    xnor

VHDL Operators

- **Numerical Operators**

  - works on types INTEGER, REAL

  - the types of the input operands must be the same

+	"addition"
-	"subtraction"
*	"multiplication"
/	"division"
mod	"modulus"
rem	"remainder"
abs	"absolute value"
**	"exponential"

ex) Can we make an adder circuit yet?

```
A,B     : in     BIT_VECTOR (7 downto 0)
Z       : out    BIT_VECTOR (7 downto 0)

        Z <= A + B;
```

VHDL Operators

- **Relational Operators**

  - used to compare objects

  - objects must be of same type

  - Output is always BOOLEAN (TRUE, FALSE)

  - works on types: BOOLEAN, BIT, BIT_VECTOR, CHARACTER, INTEGER, REAL, TIME, STRING

=	"equal"
/=	"not equal"
<	"less than"
<=	"less than or equal"
>	"greater than"
>=	"greater than or equal"

VHDL Operators

- **Shift Operators**

  - works on one-dimensional arrays

  - works on arrays that contain types BIT, BOOLEAN

  - the operator requires

    1) An Operand (what is to be shifted)
    2) Number of Shifts (specified as an INTEGER)

  - a negative Number of Shifts (i.e., "-") is valid and reverses the direction of the shift

sll	"shift left logical"
srl	"shift right logical"
sla	"shift left arithmetic"
sra	"shift right arithmetic"
rol	"rotate left"
ror	"rotate right"

VHDL Operators

- **Concatenation Operator**

  - combines objects of same type into an array

  - the order is preserved

    & "concatenate"

    ex) New_Bus <= (Bus1(7:4) & Bus2(3:0))

VHDL Operators

- **Assignment Operators**
  - The assignment operator is <=
  - The Results is always on the Left, Operands on the Right
  - Types need to all be of the same type
  - need to watch the length of arrays!

Ex) x <=y;

a <= b or c;

sum <= x + y;

NewBus <= m & k;

VHDL Operators

- **Delay Modeling**

  - VHDL allows us to include timing information into assignment statements

  - this gives us the ability to model real world gate delay

  - we use the keyword "after" in our assignment followed by a time operand.

    Ex) B <= not A after 2ns;

  - VHDL has two types of timing models that allow more accurate representation of real gates

    1) Inertial Delay (default)

    2) Transport Delay

VHDL Operators

- **Inertial Delay**

  - if the input has two edge transitions in less time than the inertial delay, the pulse is ignored

    said another way…

  - if the input pulse width is smaller than the delay, it is ignored

  - this models the behavior of trying to charge up the gate capacitance of a MOSFET

    ex) B <= A after 5ns;

    any pulses less than 5ns in width are ignored.

VHDL Operators

- **Transport Delay**

  - transport delay will always pass the pulse, no matter how small it is.

  - this models the behavior of transmission lines

  - we have to explicitly call out this type of delay using the "transport" keyword

```
ex)     B <= transport A after 5ns;

        B <= transport not A after t_delay;        -- here we used a constant
```

Generics vs. Constants

- **Generics vs. Constants**

  - it is very useful to be able to design using variables/parameters instead of hard coded values

    ex) width of bus, delay, loop counters,

  - VHDL Provides two methods for this functionality

    1) Generics
    2) Constants

  - These are similar but have subtle differences

Generics vs. Constants

- **Generics**

  - declared in Entity

  - design can be compiled without initialization

  - global variable which can be altered at run-time

  - is visible to all architectures below that entity

  syntax:

  generic (gen-name : gen-type := init-val)

  NOTE: init-val is optional

  ex)

```
entity  inv_n  is

   generic    (WIDTH  : integer := 7);

   port (In1    :  STD_LOGIC_VECTOR (WIDTH downto 0);
         Out1   :  STD_LOGIC_VECTOR (WIDTH downto 0) );

end entity inv_n;
```

Generics vs. Constants

- **Constants**

  - declared in Architecture

  - needs to be initialized

  - only visible to the architecture it is defined in

  syntax:

  constant (const-name : const-type := init-val)

  NOTE: init-val is NOT optional

  ex)

```
architecture inv_n_arch of inv_n is
    constant (t_dly : time := 1ns);
    begin
        Out1 <= not In1 after t_dly;
end architecture inv_n_arch;
```

VHDL Concurrent Signal Assignments

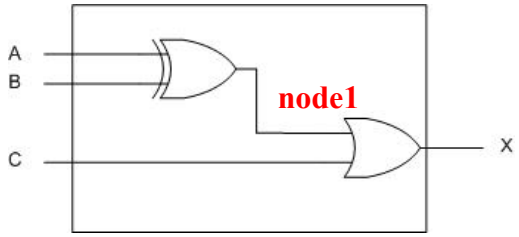
- **Concurrency**
  - the way that our designs are simulated is important in modeling real HW behavior
  - components are executed concurrently (i.e., at the same time)
  - VHDL gives us another method to describe concurrent logic behavior called

    "Concurrent Signal Assignments"
  - we simply list our signal assignments (<=) after the "begin" statement in the architecture
  - each time any signal on the Right Hand Side (RHS) of the expression changes, the Left Hand Side (LHS) of the assignment is updated.
  - operators can be included (and, or, +, …)

VHDL Concurrent Signal Assignments

- **Concurrent Signal Assignment Example**

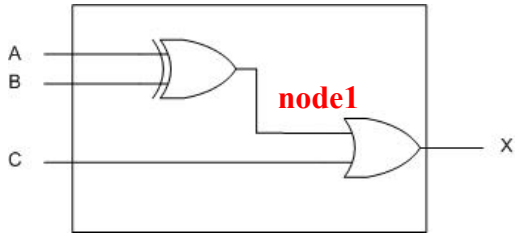
```
entity TOP is
        port (A,B,C      :  in   STD_LOGIC;
                X        :  out            STD_LOGIC);
end entity TOP;


architecture TOP_arch of TOP is

  signal    node1      :      STD_LOGIC;

  begin

        node1      <= A xor B;
        X          <= node1 or C;

end architecture TOP_arch;
```



VHDL Concurrent Signal Assignments

- **Concurrent Signal Assignment Example**

```
node1   <= A xor B;
X       <= node1 or C;
```



- if these are executed concurrently, does it model the real behavior of this circuit?

  Yes, that is how these gates operate. We can see that there may be timing that needs to be considered….

  - When does C get to the OR gate relative to (A ⊕ B)?
  - Could this cause a glitch on X? What about a delay in the actual value?

VHDL Concurrent Signal Assignments

- **Conditional Signal Assignments**

  - we can also include conditional situations in a concurrent assignment

  - the keywords for these are:

    "when" = if the condition is TRUE, make this assignment
    "else" = if the condition is FALSE, make this assignment

    ex)
    ```
    X <= '1' when A='0' else '0';
    Y <= '0' when A='0' and C='0' else '1';
    ```

  - X and Y are evaluated concurrently !!!

  - notice that we are assigning static values (0 and 1), this is essentially a "Truth Table"

  - if using this notation, make sure to include every possible input condition,
    or else you haven't described the full operation of the circuit.

VHDL Concurrent Signal Assignments

- **Conditional Signal Assignments**

  - We can also assign signals to other signals using conditions

  - this is similar to a MUX

    ex) X <= A when Sel='0' else B;

  - Again, make sure to include every possible input condition, or else you haven't described the full operation of the circuit.

  - If you try to synthesis an incomplete description, the tool will start making stuff up!

VHDL Concurrent Signal Assignments

- **Selected Signal Assignment**

  - We can also use a technique that allows the listing of "choices" and "assignments" in a comma delimited fashion.

  - this is called "Selected Signal Assignment" but it is still CONCURRENTLY assigned

  syntax:

```
with expression select

        signal-name <= signal-value when choices,
                       signal-value when choices,
                              :
                       signal-value when others;
```

  - we use the term "others" to describe any input condition that isn't explicitly described

VHDL Concurrent Signal Assignments

- **Selected Signal Assignment Example**

  Describe the following Truth Table using Selected Signal Assignments:

Input	X
000	0
001	1
010	1
011	0
100	1
101	1
110	0
111	0

```
begin
   with Input select
            X<= '0' when "000",
                '1' when "001",
                '1' when "010",
                '0' when "011",
                '1' when "100",
                '1' when "101",
                '0' when "110",
                '0' when "111";
```

VHDL Concurrent Signal Assignments

- **Selected Signal Assignment Example**

  - we can shorten the description by using "others" for the 0's

  - we can also use "|" delimited choices

Input	X
000	0
001	1
010	1
011	0
100	1
101	1
110	0
111	0

```
begin
   with Input select
            X<= '1' when "001" | "010" | "100" | "101",
                '0' when others;
```

VHDL Structural Design

- **Structural Design**

  - we can specify functionality in an architecture in two ways

    1) Structurally : text based schematic, manual instantiation of another system

    2) Behaviorally : abstract description of functionality

  - we will start with learning Structural VHDL design

- **Components**

  - blocks that already exist and are included into a higher level design

  - we need to know the entity declaration of the system we are calling

  - we "declare" a component using the keyword "component"

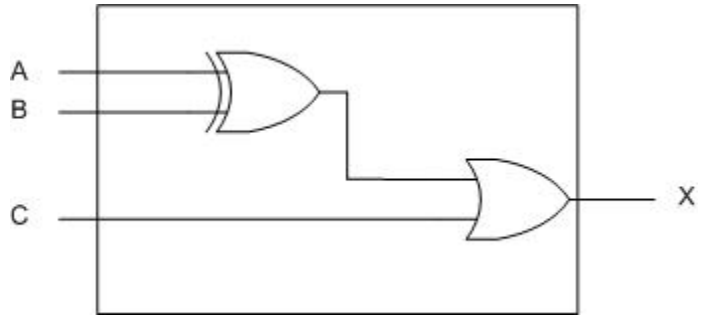
  - we declare the component in the architecture which indicates we wish to use it

VHDL Structural Design

- **Component Syntax**

```
component  component-name

        port (signal-name  :  mode  signal-type;
              signal-name  :  mode  signal-type);   -- exactly the same as the Entity declaration

end component;
```

- **Let's build this…**



VHDL Structural Design

- **Component Example**

  - let's use these pre-existing entities "xor2" & "or2"

```
entity xor2 is

        port (In1, In2    :  in   STD_LOGIC;
              Out1        :  outSTD_LOGIC);

end entity xor2;
```

```
entity or2 is

        port (In1, In2    :  in   STD_LOGIC;
              Out1        :  outSTD_LOGIC);

end entity or2;
```

VHDL Structural Design

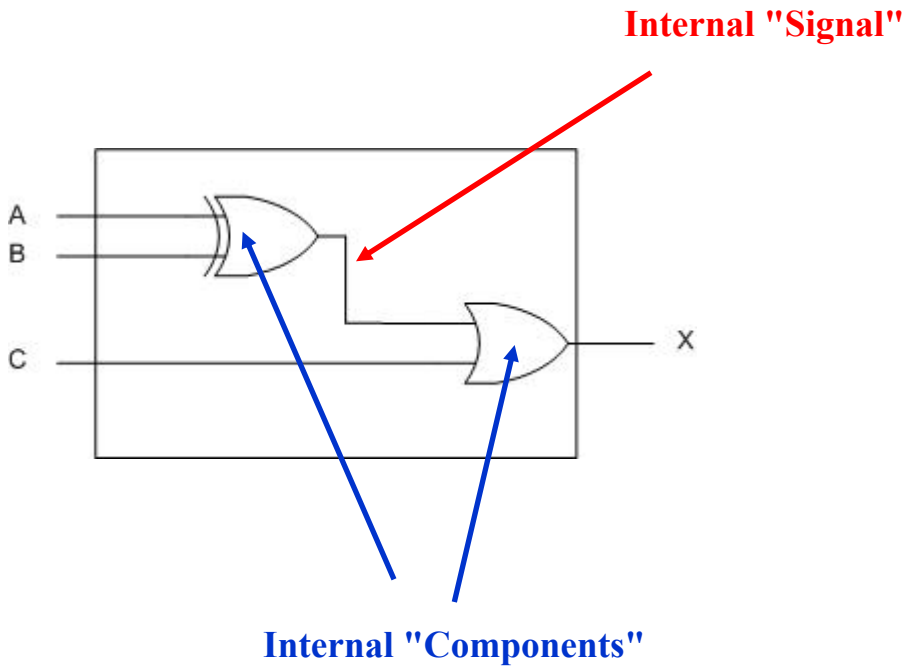
- **Component Example**

  - now let's include the pre-existing entities "xor2" & "or2" into our "TOP" design

```
entity TOP is
        port (A,B,C       :  in    STD_LOGIC;
                X             :  out              STD_LOGIC);
end entity TOP;

architecture TOP_arch of TOP is

  component xor2                                          -- declaration of xor2 component
        port (In1, In2    :  in   STD_LOGIC;
                Out1      :  outSTD_LOGIC);
  end component;

  component or2 is                                        -- declaration of or2 component
        port (In1, In2    :  in   STD_LOGIC;
                Out1      :  outSTD_LOGIC);
  end component;

  begin
        …..
```

VHDL Structural Design

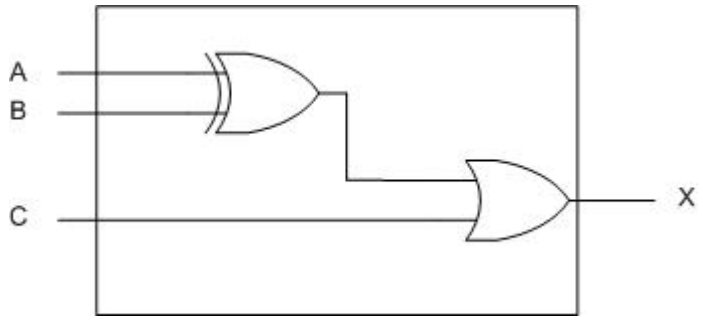
- **Signals**
  - now we want to connect items within an architecture, we need "signals" to do this
  - we defined signals within an architecture



VHDL Structural Design

- **Signal Syntax**

```
architecture TOP_arch of TOP is

        signal      signal-name : signal-type;
        signal      signal-name : signal-type;
```

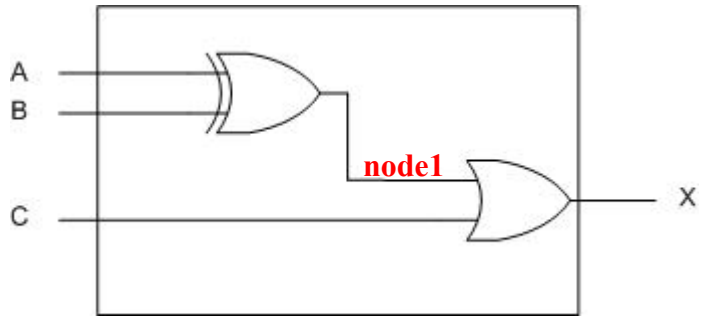


VHDL Structural Design

- **Let's put the signal declaration into our Architecture**

  - now let's include the pre-existing entities "xor2" & "or2" into our "TOP" design

```
architecture TOP_arch of TOP is

  signal    node1       :       STD_LOGIC;

  component xor2                                        -- declaration of xor2 component
        port (In1, In2    :  in   STD_LOGIC;
              Out1        :  outSTD_LOGIC);
  end component;

  entity or2 is                                         -- declaration of or2 component
        port (In1, In2    :  in   STD_LOGIC;
              Out1        :  outSTD_LOGIC);
  end component;

  begin
        …..


end architecture TOP_arch;
```



VHDL Structural Design

- **Component Instantiation**

  - after the "begin" keyword, we can start adding components and connecting signals

  - we add components with a "Component Instantiation"

  syntax:

  label : component-name port map (port => signal, ……) ;

  NOTE:

  - "label" is a unique reference designator for that component (U1, INV1, UUT1)

  - "component-name" is the exact name as declared prior to the "begin" keyword

  - "port map" is a keyword

  - the signals with in the ( ) of the port map define how signals are connected to the ports of the instantiated component

VHDL Structural Design

- **Port Maps**

  - There are two ways describe the "port map" of a component

    1) Positional
    2) Explicit

- **Positional Port Map**

  - signals to be connected to the component are listed in the exact order as the components port order

  ex) U1 : xor2 port map (A, B, node1);

- **Explicit Port Map**

  - signals to be connected to the component are explicitly linked to the port names of the component using the "=>" notation (Port => Signal, Port => Signal, ….)

  ex) U1 : xor2 port map (In1 => A, In2 => B, Out1 => node1);

VHDL Structural Design

- **Execution**

  - All components are executed CONCURRENTLY

  - this mimics real hardware

  - this is different from traditional program execution (i.e., C/C++) which is executed sequentially

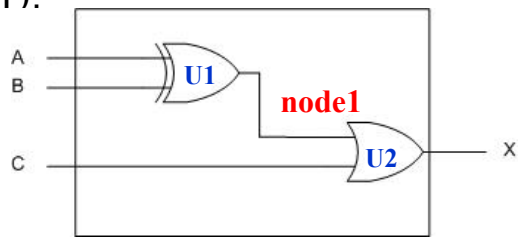
    because

  We are NOT writing code, we are describing hardware!!!

VHDL Structural Design

- **Let's put everything together**

```
architecture TOP_arch of TOP is

  signal     node1       :       STD_LOGIC;

  component xor2                                   -- declaration of xor2 component
        port (In1, In2    :  in   STD_LOGIC;
                  Out1    :  outSTD_LOGIC);
  end component;

  component or2 is                                 -- declaration of or2 component
        port (In1, In2    :  in   STD_LOGIC;
                  Out1    :  outSTD_LOGIC);
  end component;

  begin
        U1   : xor2 port map (In1=>A, In2=>B,       Out1=>node1):
        U2   : or2   port map (In1=>C, In2=>node1, Out1=>X);


end architecture TOP_arch;
```



VHDL Behavioral Design

- **Behavioral Design**

  - we've learned the basic constructs of VHDL (entity, architecture, packages)

  - we've learned how to use structural VHDL to instantiate lower-level systems and to create text-based schematics

  - now we want to go one level higher in abstraction and design using "Behavioral Descriptions" of HW

  - when we design at the Behavioral level, we now rely on Synthesis tools to create the ultimate gate level schematic

  - we need to be aware of what we CAN and CAN'T synthesis

  - Remember, VHDL was invented to model systems, not for synthesis

  - This means we can simulate a lot more functionality than we could ever by synthesized

VHDL Behavioral Design

- **Processes**

  - a way to describe interaction between signals
  - a process executes a SEQUENCE of operations
  - the new values in a process (i.e., the LHS) depend on the current and past values of the other signals
  - the new values in a process (i.e., the LHS) do not get their value until the process terminates
  - a process goes in the architecture after the "begin" keyword

syntax:

```
name : process (sensitivity list)

        declarations
        begin
            sequential statements
end process name;
```

VHDL Behavioral Design

- **Process Execution**

  - Real systems start on certain conditions

  - they then perform an operation

  - they then wait for the next start condition

    ex) Button pushed?
    Clock edge present?
    Reset?
    Change on Inputs?

  - to mimic real HW, we want to be able to START and STOP processes

  - otherwise, the simulation would get stuck in an infinite loop or "hang"

VHDL Behavioral Design

- **Process Execution**

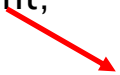
  - Processes execute in Sequence (i.e., one after another, in order)

  - these are NOT concurrent

  - this is a difficult concept to grasp and leads to difficulty in describing HW

ex)

```
name : process (sensitivity list)

   begin
            sequential statement;
            sequential statement;
            sequential statement;
end process name;
```



  - these signal assignments are called "Sequential Signal Assignments"

    (as opposed to "Concurrent Signal Assignments")

VHDL Behavioral Design

- **Starting and Stopping a Process**

  - There are two ways to start and stop a process
    1) Sensitivity List
    2) Wait Statement

- **Sensitivity List**

  - a list of signal names
  - the process will begin executing if there is a change on any of the signals in the list

ex)
```
FLOP : process (clock)

 begin
        Q <= D;
end process FLOP;
```

  - each time there is a change on "clock", the process will execute ONCE
  - the process ends after the last statement

VHDL Behavioral Design

- **Wait Statements**

  - the keyword "wait" can be used inside of a process to start/stop it

  - the process executes the sequences 1-by-1 until hitting the wait statement

  - we don't use "waits" and "sensitivity lists" together

ex)
```
DOIT : process
  begin
        statement 1;
        statement 2;
        statement 3;
end process DOIT;
```

(No Start/Stop Control, loops forever)

```
DOIT : process
                begin
        statement 1;
        statement 2;
        wait;
end process DOIT;
```

(w/ Start/Stop Control, executes until "wait" then stops)

  - we need to have a conditional operator associated with the wait statement, otherwise it just stops the process and it will never start again.

VHDL Behavioral Design

- **Wait Statements**

  - the wait statements can be followed by keywords "for" or "until" to describe the wait condition

  - the wait statement can wait for:

    1) type-expression — ex) wait for 10ns;
    wait for period/2;

    2) condition — ex) wait until Clock='1'
    wait until Data>16;

VHDL Behavioral Design

- **Signals and Processes**

  - Rules of a Process

    1) Signals cannot be declared inside of a process

    2) Assignment to a Signal takes effect only after the process suspends. Until it suspends, signals keeps their previous value

    3) Only the last signal assignment to a signal in the list has an effect. So there's no use making multiple assignments to the same signal.

```
ex)  DOIT : process (A,B)                    -- initially A=2, B=2… then A changes to 7

     begin                                   -- Y = 7 + 2      NOT      Y = 7 + 0
          A <= '0';
          B <= '0';
          Y <= A+B;

     end process DOIT;
```

VHDL Behavioral Design

- **Signals and Processes**

  - But what if we want this behavior?

```
ex) DOIT : process (A,B)                    -- initially A=2, B=2… then A changes to 7

    begin
        A <= '0';                           -- we WANT A to be assigned '0'
        B <= '0';                           -- we WANT B to be assigned '0'
        Y <= A+B;                           -- we WANT Y to be assigned A + B = 0

    end process DOIT;
```

  - we need something besides a Signal to hold the interim value

  - we need a "Variable"

Variables

- **Variables**

  - Signals in processes are only assigned their value when the process suspends

  - this makes multiple assignments to a signal meaningless

ex)
```
DOIT : process (A,B)          -- a change on A or B will trigger this process

  begin
        A <= 2;               -- B gets its value from the previous value of A,
        B <= A + 1;           -- not from the A <= 2 assignment

end process DOIT;
```

  - Variables allow us to assign values during the sequence of statements

Variables

- **Variables**

  - Variables are defined within a process

  syntax:
  variable var-name : var-type := init value

  - assignments to variables are made using ":=" instead of "<="

  - assignments take place immediately

ex) DOIT : process (A,B) -- a change on A or B will trigger this process

```
variable temp : integer := 0;

begin
    temp := 2;
    B <= temp + 1;

end process DOIT;
```

Variables

- **Signal vs. Variable**

Signal	Variable
has type (type, value, time)	has type (type, value)
assignment with <=	assignment with :=
declared outside of the process	declared inside of process
assignment takes place when process suspends	assignment is immediate
always exists	only exists when process executes

If-Then Statements

- **If / Then Statements**

  - Used ONLY within a process. VHDL has the following:

    - *if, then*
    - *if, then, else*
    - *if, then, elsif, then*
    - *if, then, elsif, then, else*

  syntax:

  if boolean-exp then seq-statement
  elsif boolean-exp then seq-statement
  else seq-statement

  - parenthesis are allowed, but not required

  - multiple sequential statements allowed, they are separated by a ";" and can be on different lines

  - logical operators allowed in Boolean Expression

If-Then Statements

- **If / Then Statements**

ex) Design a 2-to-1 MUX

```
architecture mux_2to1_arch of mux_2to1 is

begin
        MUX : process (A,B,Sel)
          begin
               if (Sel = '0') then
                  Out1 <= A;
               elsif (Sel = '1') then
                  Out1 <= B;
               else
                  Out1 <=A;          -- this isn't necessary, just for illustration
               end if;
          end process MUX;

end architecture mux_2to1_arch;
```

Case Statements

- **Case Statements**

  - used ONLY within a process

  - better for larger input combinations, If/Then's can get too long

  syntax:

```
case  expression is
    when  choices  =>  seq-statement;
    when  choices  =>  seq-statement;
          :
end case;
```

  - the keyword "others" is available for input combinations not explicitly called out

Case Statements

- **Case Statements**

ex) Design a 2-to-1 MUX

```
architecture mux_2to1_arch of mux_2to1 is

begin
        MUX : process (A,B,Sel)
          begin
               case (Sel) is
                  when '0'                  => Out1 <= A;
                  when '1'                  => Out1 <= B;
                  when others => Out1 <= A;  -- this isn't necessary, just for illustration
               end case;
          end process MUX;

end architecture mux_2to1_arch;
```

- the case statement works nice on vectors

- if you want to combine individual signals to form a vector, you can use variables and the concatenation operator

Conditional Loops

- **Conditional Loops**

  - There are multiple loop structures we can use within VHDL

    1) Loop
    2) While
    3) For

- **Loops**

  - "Loop" is a keyword that starts a loop

  - creates an infinite loop

  - useful for modeling process that go forever (i.e, clocks, time)

Conditional Loops

- **Loops**

ex)
```
CLOCK_GEN : process
        begin
          clock <= '0';

          loop
            clock <= '1' after 1ns;
            clock <= '0' after 1ns;
          end loop;

end process CLOCK_GEN;
```

- the loop is ended using the keywords "end loop;"

Conditional Loops

- **While Loops**

  - a Boolean condition is tested at the beginning of the loop

  - the loop only executes if the condition is true

ex)

```
CLOCK_GEN : process
        begin
          clock <= '0';

          while (EN = '1')
            clock <= not clock after 1ns;
          end loop;

end process CLOCK_GEN;
```

Conditional Loops

- **For Loops**

  - a loop with a counter

  - the loop executes the # of times in the range that is specified

  syntax:

```
for identifier in range loop

        seq-statement
        seq-statement

end loop;
```

  - the "identifier" is the loop variable.

    - It is implicitly declared when included in the "for" statement.
    - It is automatically the same type as the "range"
    - it will step through ALL values in range

Conditional Loops

- **For Loops**

  - the "range" needs to be previously defined. All types are allowed

  - Supporting all types is powerful for enumerated lists in state machines

    (i.e., state_list = idle, go, stop, ….)

ex)

```
for state in state_list loop

    if (current_state =  state) then

            valid_state = TRUE;
    end if;
end loop;
```

Attributes

- **Attributes**

  - ability to get more information about a signal other than its current value

  - attributes allow access to the signal's history

    - previous value
    - time since last change

  - this is how we can specify "edge triggered" events in sequential logic

  - we put the attribute keyword after the signal name using the apostrophe (')

  - there are many attributes, the most commonly used are:

    1) event
    2) transaction
    3) last_value
    4) last_event

Attributes

- **"event" Attribute**

  - tells us when there was a change on the signal

  - useful for edge detection

  ex) "rising edge"

  ```
  if (Clock'event and Clock='1')
  ```

- **"transaction" Attribute**

  - tells us when there was an assignment is made to a signal

  - the signal value does not need to change (i.e., 0 to 0)

  ex) process (A'transaction)

  ```
  statement if anybody ever assigns to A
  ```

Attributes

- **"last_value" Attribute**

  - tells us the last value of a signal (before most recent assignment)

- **"last_event" Attribute**

  - gives TIME since last event

  - good for tracking timing violations (Setup/Hold, signals changing too fast)

  ex) process (Data'event)

```
begin
        if (Data'last_event < 0.5ns) then
            too_fast <= TRUE;
        else
            too_fast <= FALSE;
        end if;
```

VHDL : Test Benches

- **Test Benches**
  - We need to stimulate our designs in order to test their functionality
  - Stimulus in a real system is from an external source, not from our design
  - We need a method to test our designs that is not part of the design itself
  - This is called a "Test Bench“
  - Test Benches are VHDL entity/architectures with the following:
    - We instantiate the design to be tested using components
    - We call these instantiations "Unit Under Test" (UUT) or "Device Under Test".
    - The entity has no ports
    - We create a stimulus generator within the architecture
    - We can use reporting features to monitor the expected outputs

VHDL : Test Benches

- **Test Benches**

  - Test Benches are for Verification, not for Synthesis!!!

    - this allows us to use constructs that we ordinarily wouldn't put in a design because they are not synthesizable

- **Let's test this MUX**

```
entity  Mux_2to1 is

        port (A, B, Sel      : in       STD_LOGIC;
              Y              : out      STD_LOGIC);

entity Mux_2to1;
```

VHDL : Test Benches

```
entity  Test_Mux is
end entity Test_Mux;                                    -- the test bench entity has no ports

architecture Test_Mux_arch of Test_Mux is

        signal      In1_TB, In2_TB      : STD_LOGIC;    -- setup internal Test Signals
        signal      Sel_TB              : STD_LOGIC;    -- give descriptive names to make
        signal      Out_TB              : STD_LOGIC;    --  apparent they are test signals

        component Mux_2to1                              -- declare any used components
            port (A, B, Sel     : in        STD_LOGIC;
              Y                 : out       STD_LOGIC);
        end component;

        begin

          UUT : Mux_2to1                                 -- instantiate the design to test
            port map ( A    => In1_TB,
                       B    => In2_TB,
                       Sel => Sel_TB,
                       Y    => Out_TB);
```

VHDL : Test Benches

```
        STIM : process                                  -- create process to generate stimulus
          begin
            In1_TB <= '0'; In2_TB <= '0'; Sel_TB <= '0' wait for 10ns    -- we can use wait
            In1_TB <= '0'; In2_TB <= '1'; Sel_TB <= '0' wait for 10ns    -- statements to control
            In1_TB <= '1'; In2_TB <= '0'; Sel_TB <= '0' wait for 10ns    -- the speed of the stim

                                    :
                                    :
                                    :
            In1_TB <= '1'; In2_TB <= '1'; Sel_TB <= '1' wait for 10ns    -- end with a wait…

        end process STIM;


end architecture Test_Mux_2to1;
```

VHDL : Test Benches

- **Test Bench Reporting**

  - There are reporting features that allow us to monitor the output of a design

  - We can compare the output against "Golden" data and report any differences

  - This is powerful when we evaluate our designs across power, temp, process...

- **Assert**

  - the keyword "assert" will check a Boolean expression

  - if the Boolean expression is FALSE, it will print a string following the "report" keyword

  - Severity levels are also reported with possible values {ERROR, WARNING, NOTE, FAILURE}

    ex)

```
A<='0'; B<='0'; wait for 10ns;
assert (Z='1') report "Failed test 00" severity ERROR;
```

  - The message comes out at the simulator console.

VHDL : Test Benches

- **Report**

  - the keyword "report" will always print a string

  - this is good for outputting the process of a test

  - Severity levels are also reported

ex)

```
report "Beginning the MUX test" severity NOTE;

A<='0'; B<='0'; wait for 10ns;
assert (Z='1') report "Failed test 00" severity ERROR;
```