

# ECSE-323

# Digital System Design

**VHDL Lecture #1**

# Your first day on the job after graduating from McGill...



**the boss**



Hey, rookie!  
We need to design a *new*,  
*inexpensive*, *fast*  
embedded digital system  
and we need it *yesterday*!  
Get on it!



You went to McGill and took DSD so you say  
**Don't worry boss, I got this!**

**The design needs to be**

- 1. - verified*
- 2. - documented*
- and**
- 3. - implemented*



# Design Verification

*How do you know the design is correct?*

*(Note: the **boss** doesn't like incorrect designs. Actually, he doesn't like losing money, and incorrect designs waste time, and time is money!)*

Answer: *Simulate* the design using some *description* of the target implementation and *compare* the results to the *specified* behaviour.

*(assumes that the specifications are correct)*

So, for *verification*, we need:

- A way to *describe* an implementation
- A way to *simulate* the implementation based on the description.
- A way to *compare* simulation results with specifications.



# Design Documentation

*How are you going to **communicate** information about your device to the **consumer**?*

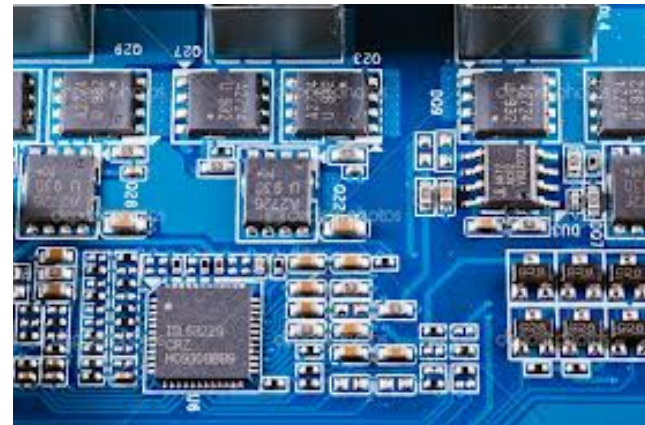
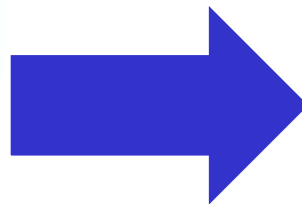
*How will you communicate this information to other members of the design team (e.g. for later modifications).*

Kill 2 birds with 1 stone –

Use the implementation *description*  
developed for verification purposes for  
*documentation* as well!

# Design Implementation

*How are you going to transfer your design ideas to a hardware implementation?*



# Design Synthesis

Using your *description* of the hardware, map, or *synthesize*, the design onto the desired target hardware (such as a gate-array, printed circuit board, VLSI chip).

Notice that each requirement of the design process:

*verification,*

*documentation,*

*implementation,*

needs a **description** of the hardware being designed.

Some smart engineers got tired of having their bosses get mad at them, so many companies developed various forms of **hardware description languages**.



The two Hardware Description Languages that are most often used in industry are:

*VHDL* ← you will learn this one

*Verilog* (has a more ‘C-like’ syntax)

These are similar and if you learn one you can pick up the other quickly.

# VHDL

**V** is for *VHSIC*

(*V*ery *H*igh *S*peed *I*ntegrated *C*ircuit)

**H** is for *Hardware*

**D** is for *Description*

**L** is for *Language*





# VHDL

H stands for *Hardware*

This course is brought to you by the letter **H**!

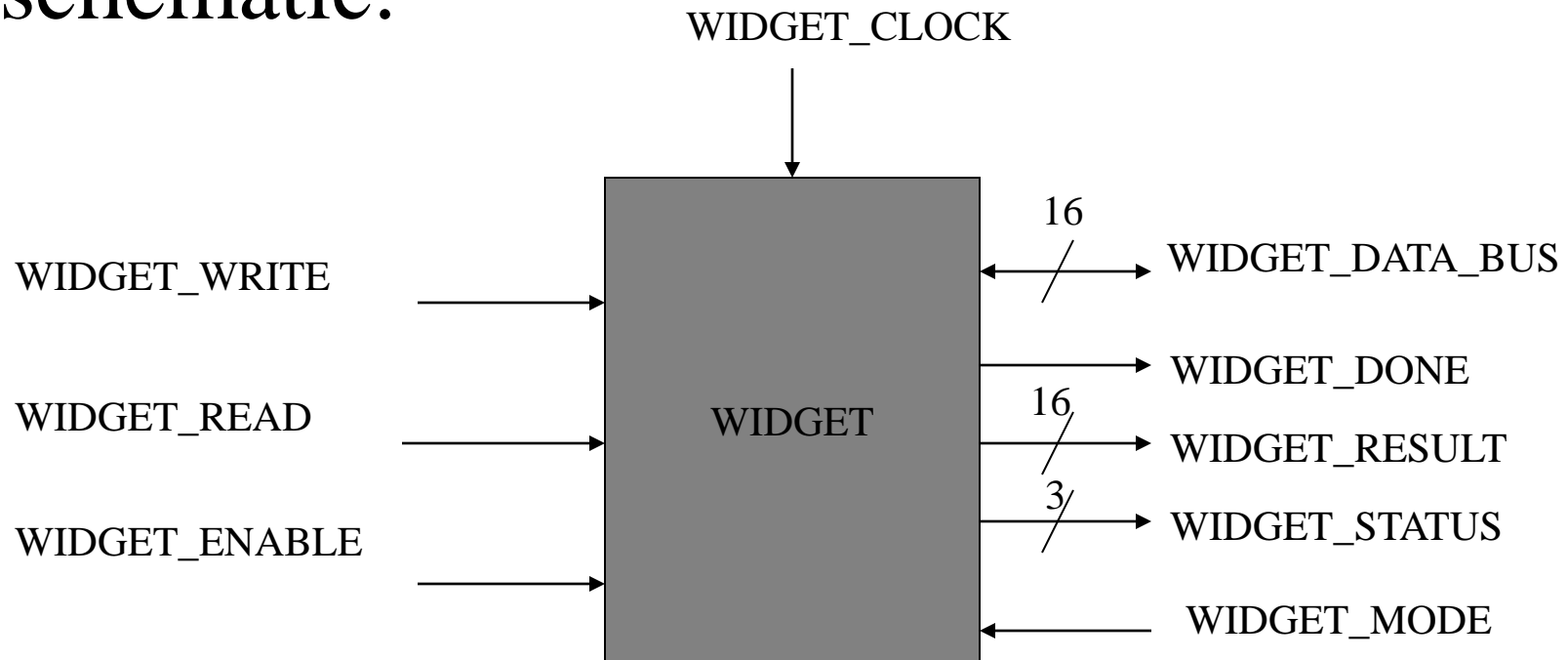
VHDL Descriptions consist of two parts:

- The *entity* declaration
- The *architecture* declaration

The combination of these two parts is called a *design entity*

The *entity declaration* describes the circuit as it appears from the "outside" – from the perspective of its *inputs* and *outputs*.

You could think of the entity declaration as being analogous to a block symbol on a schematic.



*The actual operation of the circuit is not defined in the entity declaration.*

This is just as in a schematic diagram, where the symbol for a part gives no details about what is going on *inside* the part.

Example of an entity declaration:

```
entity example is
  port( A, B: in bit;
        EQ: out bit);
end example;
```

**NAME of the design entity**

**SIGNAL TYPES**

**PORT MODES**

**SIGNAL NAMES**

The *port statement* defines *all* of the signals that will be visible to external design entities.

Each of the ports is given a *port mode*, a *signal name*, and a *signal type*.

# ***PORT MODES***

There are 4 different port modes:

**in** - the associated signal *can only be read*, and not set

**out** - the associated signal *can only be set*, and not read

**inout** - the associated signal can be read and set

**buffer** - indicates a port which can be used for both input and output, and it can have only one source. A buffer port can only be connected to another buffer port or to a signal that also has only one source.

We will mostly use **in** ports and **out** ports.



# ***SIGNALS***

Signals in VHDL are objects whose values may be changed and have a time dimension. In VHDL software they are implemented as *lists of time-stamped events* which represent digital waveforms on wires in a circuit.

Signals are physically realizable (i.e. there will exist wires in the synthesized hardware on which these signals reside)

## ***SIGNAL TYPES***

Signals in VHDL have *types*, which are defined by the possible values which the signal can take on.

Signals of type “**BIT**” can take on values from the set  $\{0, 1\}$

Signals can also be arrays of values. An example is the “**BIT\_VECTOR**” type.

## Example of bit\_vector signal type usage

**entity** example **is**

```
port( A, B: in bit_vector(7 downto 0);  
      EQ: out bit);
```

**end** example;

Signals A and B are defined as 8-element arrays with each array element being of type bit.

The **MSB** is element 7 and the **LSB** is the element with index 0

The “BIT” and “BIT\_VECTOR” types are *built-in* to VHDL (i.e. are defined by the VHDL standard)

Some other built-in signal types are:

**BOOLEAN**

**CHARACTER, STRING**

**INTEGER**

**REAL**

## Integer Signal Type

The *integer* signal type is very useful for implementing counters.

```
signal    x : INTEGER range -16 to 15;
```

If the range is left out, the default range is used:

$$-(2^{31} - 1) \text{ to } (2^{31} - 1)$$

The number of bits in an integer signal is not specified explicitly. The compiler determines how many bits to allocate to the signal.

```
signal    x : integer range -32768 to 32767;
```

In the above example, 16 bits would be allocated to the signal **X**.

One is not limited to the signal types that are built-in to VHDL (i.e. are defined by the VHDL standard).

Other signal types can be user-defined.

(these are often defined in packages that are included into your code).



## The **STD\_LOGIC** Type

- Designed to model electrical signals on single wires
- Used to represent signals driven by:
  - active drivers* (forcing strength)
  - resistive drivers* (pull-ups and pulldowns – weak strength)
  - tri-state drivers* (which add a high-impedance state)

There are *9 different values* that a signal of type **std\_logic** can take on. Operators for signals of this type must handle all of the possible combinations.

```
type std_logic is (  
    'U', -- Uninitialized  
    'X', -- Forcing unknown  
    '0', -- Forcing zero  
    '1', -- Forcing one  
    'Z', -- High Impedance  
    'W', -- Weak Unknown  
    'L', -- Weak zero  
    'H', -- Weak one  
    '-' ); -- Don't care
```

## **STD\_LOGIC\_VECTOR** Type

Unconstrained array type for vectors of standard-logic values  
(std\_logic values)

Similar to **bit\_vector** signals but are arrays of **std\_logic** signals.

The ranges of **bit\_vector** and **std\_logic\_vector** signals can be defined in two different ways:

## Example of Style 1:

```
C : in std_logic_vector(0 to 2);
```

This way is often used to represent a collection of wires that are not intended to represent a number.

In this example **C(0)** is the MSB and **C(2)** is the LSB.

## Example of Style 2:

```
C : in std_logic_vector(2 downto 0);
```

This way is preferred when the signal represents a binary number.

In this example **C(0)** is the LSB and **C(2)** is the MSB.

## VHDL Libraries

Signal definitions can be gathered together into libraries, which can be re-used in many different designs.

*Always declare all used libraries before the design entity*

```
library IEEE;  
use std_logic_1164.all;  
  
entity my_design is  
.....
```

## IEEE STD\_LOGIC Library

This is a set of packages defining commonly used data types and operations.

### Std\_Logic\_1164 Multivalue Logic System

This library defines types and operations to deal with strong, weak and high-impedance strengths, and unknown values.

- Std\_logic
- Std\_logic\_vector

*Always include this library in your VHDL designs*

**the boss**



Here are the **functional specifications** for the new system...

It shouldn't take a hotshot like you very long to come up with the design!



Oh no! How can we translate our boss's functional specifications into hardware?

So far, we haven't said anything about how we can describe what actually goes on in the circuit.

In VHDL, this is taken care of in the *Architecture declaration and body*

The architecture body consists of two parts:

- **Declarations area**
  - declare internal (non-port) signals
  - declare component types
  - declare user-defined signal types
  - declare constants
  - other declarations
- **Concurrent statements area**
  - signal assignments

Architecture name  
Associated entity name

```
architecture implementation1 of example is  
  signal C : bit; }  
begin  
  C <= A and B;  
  EQ <= A or C; }  
end implementation1;
```

Declarations area

Concurrent statements area

## Signal Declarations

A signal declaration includes—in this order:

- the *reserved word* "*signal*",
- the *name* of the signal,
- the *type* of the signal,
- optionally, an indication of the signal's kind (which must be either "register" or "bus"),
- optionally, an expression specifying the initial value of the signal.

## Some examples of signal declarations:

```
signal data, reset : bit;  
signal clk : std_logic := '0';  
signal memory_bus : bit_vector(7 downto 0);
```

Signal names

Signal types

Initial values

## The Concurrent Statements Area

The *concurrent statements area* is where you describe the *functionality* of the circuit.

This place is found between the **begin** and **end** statements of an *architecture body*.

```
architecture implementation1 of example is
  signal C : bit; }
begin
  C <= A and B; }
  EQ <= A or C; }
end implementation1;
```

Declarations area

Concurrent statements area

## *Concurrency*

VHDL models physical circuits, therefore there is no natural ordering of signals and signal assignments. Many events can happen at the same time in a physical circuit.

**All signal assignments are therefore considered to be *concurrent*, and can be *written in any order*.**

There can still be *causal* ordering, however.





## This one?

## This one?

**It is a  
meaningless  
question.**

# Signal Assignment Statements

*Circuit functionality* is described using various types of *signal assignment statements*.

## *Signal Assignments*

...specify how *events* on some signals are created in response to *events* on other signals.

## Forms of concurrent statements:

- *Simple Signal* Assignment
- *Selected Signal* Assignment
- *Conditional Signal* Assignment
- *Component* Instantiation
- *Process* Statement
- *Generate* Statement
- *Concurrent Function* Call
- *Concurrent Procedure* Call
- *Block* Statement

## *Simple Signal Assignment Statement*

```
signal  <=  expression;
```

Examples:

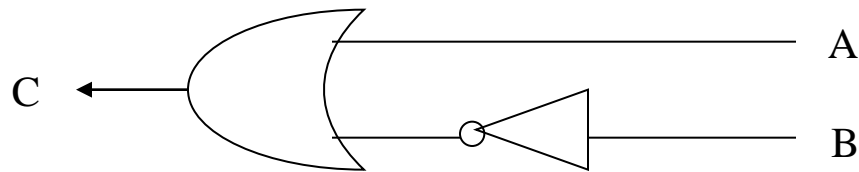
```
A_Out  <=  not  (  A_In  or  B_in  )  and  Enable;
```

```
Data  <=  X  nor  D2  xor  (flag_A  and  flag_B);
```

- The “not” operator has the highest precedence.
- Operators in parentheses are evaluated first.
- All binary operators have equal precedence.
- Operators with the same precedence are evaluated left-to-right.

Time for an example...

Let us make a VHDL description of the following circuit using simple signal assignment statements:



```
entity example is
    port( A, B: in std_logic;
          C: out std_logic);
end example;
```

```
architecture implementation1 of example is
    signal I1 : std_logic;
begin
    C <= A or I1;
    I1 <= not B;
end implementation1;
```

The order of these statements is not important!