Chapter 3
Entities and Architectures

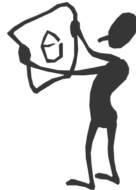
VHDL

Course Objectives Affected

- Write functionally correct and well-documented VHDL code, intended for either simulation or synthesis, of any **combinational** or **sequential** logic design.
- Define and use the three major styles of writing VHDL code (**structural**, **dataflow**, and **behavioral**).
- Write VHDL code that can be implemented efficiently in a given technology device.
- Programming and testing the device.

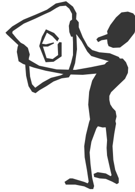
Outline

- ➡ **VHDL Basics:**
- Entity - Architecture Structure
- Entity Declarations (Ports)
- Architecture Body
- VHDL Styles
- Effects of Style on Synthesis



Reserved Word - IEEE 1076-1993

abs	downto	library	postponed	sri
access	else	linkage	procedure	subtype
after	elsif	literal	process	then
alias	end	loop	pure	to
all	entity	map	range	transport
and	exit	mod	record	type
architecture	file	nand	register	unaffected
array	for	new	reject	units
assert	function	next	rem	until
attribute	generate	nor	report	use
begin	generic	not	return	variable
block	group	null	rol	wait
body	guarded	of	ror	when
buffer	if	on	select	while
bus	impure	open	severity	with
case	in	or	signal	xnor
component	inertial	others	shared	xor
configuration	inout	out	sla	
constant	is	package	sll	
disconnect	label	port	sra	



Identifiers

- **Identifiers = names of things you create**
  - signals, variables, constants
  - architecture names, entity names, component names
  - process names
  - function names, procedure names, etc.
- **Rules**
  - Cannot be reserved words
  - Uppercase and lowercase equivalent
  - Only letters, numbers, and underscore (_)
    - First character is letter
    - First and last character NOT underscore
    - Two underscores in succession illegal

Identifiers Example

- **Which are legal identifiers?**

footer_5	7bits
_load	Execute14_more
doitnow_	Endif
add__subtract	process
don’t_do_it	exor#and
StrongFuzzyLogicDriver	carry/

Extended Identifiers

- An extended identifier is a sequence of characters written between two backslashes. Any of the allowable characters can be used, including characters like., !, @, ',and $. Within an extended identifier, lower-case and upper-case letters are considered to be distinct. Examples of extended identifiers are:
  - **\TEST** -- Differs from the basic identifier TEST.
  - **\2FOR$**
  - **\process** -- Distinct from the keyword process.
  - **\7400TTL**
  - Two consecutive backslashes represents one backslash \→\\.

Comments

- Comments in a description must be preceded by two consecutive hyphens (**--**); the comment extends to the end of the line. Comments can appear anywhere within a description. Examples are:

```
-- This is a comment; it ends at the end of this line.
-- To continue a comment onto a second line, a separate
-- comment line must be started.
```

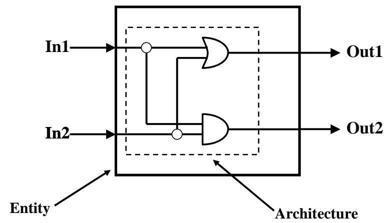
- **entity** UART **is end**; --This comment starts after entity declaration.
- Equivalent to // in C & C++.

Objects

- **Objects** are things that hold values (containers)
  - Have a **class** and a **type**
- Class determines the kind of operations possible for an object
- Type determines the legal values for an object
- Classes:
  - **signal** - value changes as function of time, has a driver; physical wire
  - **variable** - value changes instantly, no concept of time
  - **constant** - value cannot be changed
  - **file** - values accessed from external disk file

Tutorial

How to generate a basic Boolean functions in VHDL?



Entity

Architecture

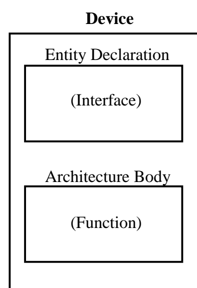
Tutorial

Basic Boolean functions in VHDL?

```
library ieee;
use ieee.std_logic_1164.all;

entity MyFirstProg is port(
   IN1, IN2:          in    std_logic;
   OUT1, OUT2:     out  std_logic);
end MyFirstProg;

architecture DataFlow of  MyFirstProg  is
begin
            OUT1 <= IN1 or IN2;
            OUT2 <= IN1 and IN2;
end DataFlow;
```

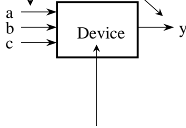

Code Model

Device



Device Model

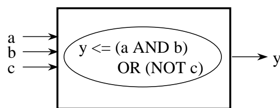
Ports - external connection points that have a *class*, *type,* and *mode*



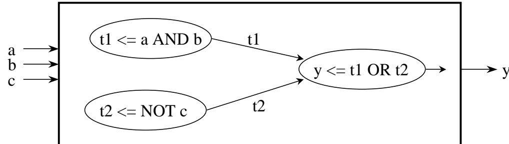
Entity - collection of concurrent *processes & statements* that describe the function

Device Model

Single-process Entity:



Multiple-process Entity:



Entity Declaration

- **Entity Declaration**
  - defines ***external interface*** to this entity or device
  - comes before the architecture section
- **Basic** entity syntax:

Direction of data flow

```
ENTITY entity_name IS
  PORT (port1, port2: MODE TYPE;
        port3, port4: MODE TYPE;
        port5: MODE TYPE);
END entity_name ;
```

What kind of values it can have

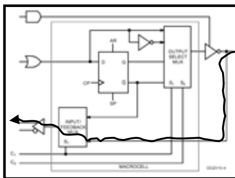
Ports

- Port Declaration is primary content of the Entity Declaration
- Each port represents either
  - external pin(s) of the device, or
  - wire(s) connecting two or more entities within a complete device
- Each port has
  - **Port name** (identifier you create)
  - **Mode** (direction - In, Out, Buffer, InOut)
  - **Type** (kind of values possible)

Port Modes

- **In**:
- Data flows only INTO this entity, driven by another entity.
- Used on right side of assignments.

z <= a NAND inport;

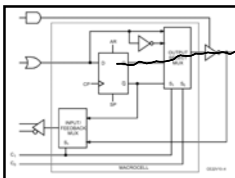


```
library ieee;
use ieee.std_logic_1164.all;

entity MyFirstProg is port(
   IN1, IN2:          in    std_logic;
   OUT1, OUT2:        out   std_logic);
end MyFirstProg;

architecture DataFlow of  MyFirstProg  is
begin
          OUT1 <= IN1 or IN2;
          OUT2 <= IN1 and IN2;
end DataFlow;
```


Port Modes

- **Out**:
- Data flows only OUT of this entity, into other entities.
- Cannot be read by this entity (internal feedback)
- Used on left side of assignments.

outport <= x OR y;



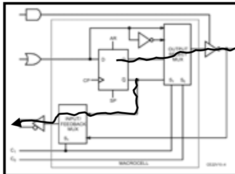
```
library ieee;
use ieee.std_logic_1164.all;

entity MyFirstProg is port(
   IN1, IN2:          in    std_logic;
   OUT1, OUT2:        out   std_logic);
end MyFirstProg;

architecture DataFlow of  MyFirstProg  is
begin
          OUT1 <= IN1 or IN2;
          OUT2 <= IN1 and IN2;
end DataFlow;
```

Port Modes

- **Buffer**:
- Data flows only OUT of the entity, into other entities.
- Can be read by this entity (internal feedback).
- Can only have one driver.
- Used on both sides of assignments.

```
bufport <= z;
y <= bufport;
```

* The bufport is the same in both lines.



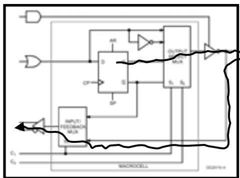
```
library ieee;
use ieee.std_logic_1164.all;

entity MyFirstProg is port(
   IN1, IN2:          in    std_logic;
   Buff1, Buff2:      buffer  std_logic);
end MyFirstProg;

architecture DataFlow of  MyFirstProg  is
begin
         Buff1 <= IN1 or IN2;
         Buff2 <= Buff1 and IN2;
end DataFlow;
```


Port Modes

- **InOut**:
- Data can flow **bidirectionally**, either **in** or **out** or both.
- Can be read by this entity (external input).
- Can also be driven by external driver.
- Used on both sides of assignments.

```
biport <= z;
y <= biport;
```

* The biport is not the same in both lines.



```
library ieee;
use ieee.std_logic_1164.all;

entity MyFirstProg is port(
   IN1, IN2:          in    std_logic;
   BiDir:             inout  std_logic);
end MyFirstProg;

architecture DataFlow of  MyFirstProg  is
begin
         BiDir <= IN1 or IN2;
         X <= Bidir and IN2;
end DataFlow;
```


Port Types

- Ports are always **signals**
  (*object class* - others are variables, constants, files)
- **Types** useful for synthesis and simulation
  - bit, bit_vector
  - std_logic, std_logic_vector
  - std_ulogic, std_ulogic_vector
  - boolean
  - integer
- Types only useful for simulation
  - real
  - time

Architecture Body

- **Architecture Body**
  - defines the ***function*** of the entity
  - follows the entity declaration

- Basic architecture syntax:

```
ARCHITECTURE arch_name OF entity_name IS
BEGIN
  CONCURRENT_STATEMENT1;
  CONCURRENT_STATEMENT2;
  PROCESS();
  PROCEDRURES();
END arch_name ;
```

No meaning for the order

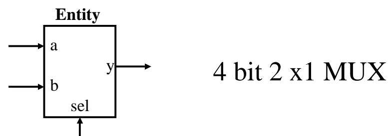
Architecture Body

- Levels of abstraction (VHDL styles)
  - **behavioral**
  - **dataflow**
  - **structural**

- An architecture may be written entirely in one style, or in a mixture of styles (more common)

- Various styles allow designer to describe the function of a device in the simplest or most natural form

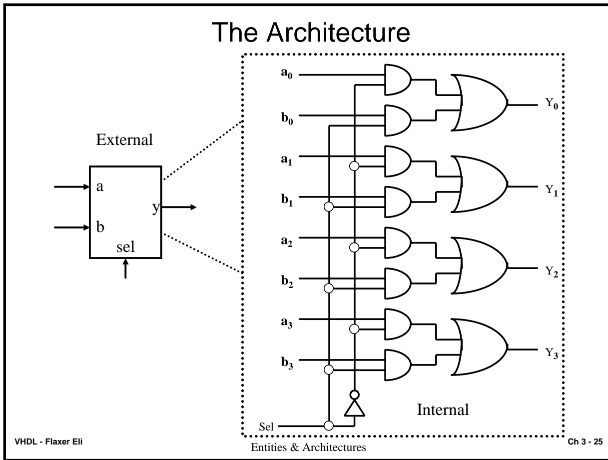
Levels of Abstraction



4 bit 2 x1 MUX

```
LIBRARY ieee;
   USE ieee.std_logic_1164.all;
   -------------------------------------------
   ENTITY mymux1 IS
     PORT (a,b: IN bit_vector(3 DOWNTO 0);
           sel: IN bit;
           y:   OUT bit_vector(3 DOWNTO 0));
   END mymux1;
    ------------------------------------------
```

The Architecture



Behavioral Style

- High-level, Algorithmic
- Easy to write and understand (like high-level language code)
- VHDL Process with sequential statements - order is important!
- Executes in zero simulation time

```
ARCHITECTURE behavior OF mymux1 IS
BEGIN
  mux: PROCESS (a,b,sel)
  BEGIN
    IF sel = '0' THEN
          y <= a;
    ELSE
          y <= b;
    END IF;
  END PROCESS mux;
END behavior;
```


Behavioral Style

- Another behavioral description of same device
- Is this exactly the same ??? **YES !!!**
- Why ??? (signals are update at the end of the process).

```
ARCHITECTURE behavior2 OF mymux1 IS
BEGIN
  PROCESS (a,b,sel)
  BEGIN
    y <= a;
    IF sel = '1' THEN
       y <= b;
    END IF;
  END PROCESS;
END behavior2;
```

Dataflow Style

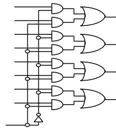
- Mid-level, Data transfers and transformations
- Also called RTL (Register Transfer Language) style
- May be harder to write and understand (like assembly code)
- No VHDL Process
- Multiple concurrent signal assignment statements
- Executes in non-zero simulation time

```
ARCHITECTURE dataflow OF mymux1 IS
BEGIN
  y <= a WHEN (sel = '0') ELSE b;
END dataflow ;
```


Boolean Dataflow Style

- Another dataflow description of same device
- Concurrent statements - order is irrelevant

```
ARCHITECTURE boolean_dataflow OF mymux1 IS
BEGIN
  y(3) <= (a(3) AND NOT sel) OR (b(3) AND sel);
  y(2) <= (a(2) AND NOT sel) OR (b(2) AND sel);
  y(1) <= (a(1) AND NOT sel) OR (b(1) AND sel);
  y(0) <= (a(0) AND NOT sel) OR (b(0) AND sel);
END boolean_dataflow ;
```



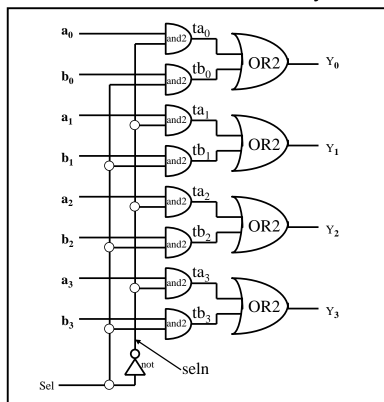
Structural Style

- Low-level, VHDL netlist - component instantiations and wiring
- Essentially the text version of a schematic
- Hierarchical
- Uses a **package** of pre-defined lower-level components
- May be hard to write and understand (very detailed and low level)
- No VHDL Process or concurrent signal assignment statements

Structural Style

```
USE work.gates_pkg.all;
ARCHITECTURE structural OF mymux1 IS
  SIGNAL ta, tb: bit_vector (3 downto 0);
  SIGNAL seln: bit;
BEGIN
  u0:  and2 PORT MAP (a(3), seln, ta(3));
  u1:  and2 PORT MAP (a(2), seln, ta(2));
  u2:  and2 PORT MAP (a(1), seln, ta(1));
  u3:  and2 PORT MAP (a(0), seln, ta(0));
  u4:  and2 PORT MAP (b(3), sel, tb(3));
  u5:  and2 PORT MAP (b(2), sel, tb(2));
  u6:  and2 PORT MAP (b(1), sel, tb(1));
  u7:  and2 PORT MAP (b(0), sel, tb(0));
  u8:  or2  PORT MAP (ta(3), tb(3), y(3));
  u9:  or2  PORT MAP (ta(2), tb(2), y(2));
  u10: or2  PORT MAP (ta(1), tb(1), y(1));
  u11: or2  PORT MAP (ta(0), tb(0), y(0));
  u12: not  PORT MAP (sel, seln);
END structural;
```


Schematic Structural Style



Effects of Style on Synthesis

- For complex circuits, the style of description affects the implementation by synthesizer and fitter (or place-and-route) - Why?
- The VHDL code may not be an optimal description of the function
- The synthesizer may not generate logic descriptions in the best form for a particular device's fitter tool
- The fitter may not choose the best device resources to use, even though it receives the optimal description from the synthesizer.

Effects of Style on Synthesis

- General recommendations:
- Start with behavioral and dataflow code
  - easiest to write and understand and debug
- Use structural code where design naturally decomposes into separate functional blocks
- If this does not satisfy area/speed goals, add synthesis directives and constraints
- If result still does not satisfy goals, add more detailed RTL descriptions and/or use vendor-specific library components.