CLEMSON UNIVERSITY

ECE 3270 Digital Computer Design

Digital Design Using VHDL: A Systems Approach

Lecture 2:
VHDL Intro

Readings

- Lecture 1: Chapters 1, 3, 6 (Digital Abstraction)
- Lecture 2: Chapters 7, Appendix A & B (VHDL)
- Lecture 3: Chapters 4, 5 (CMOS)

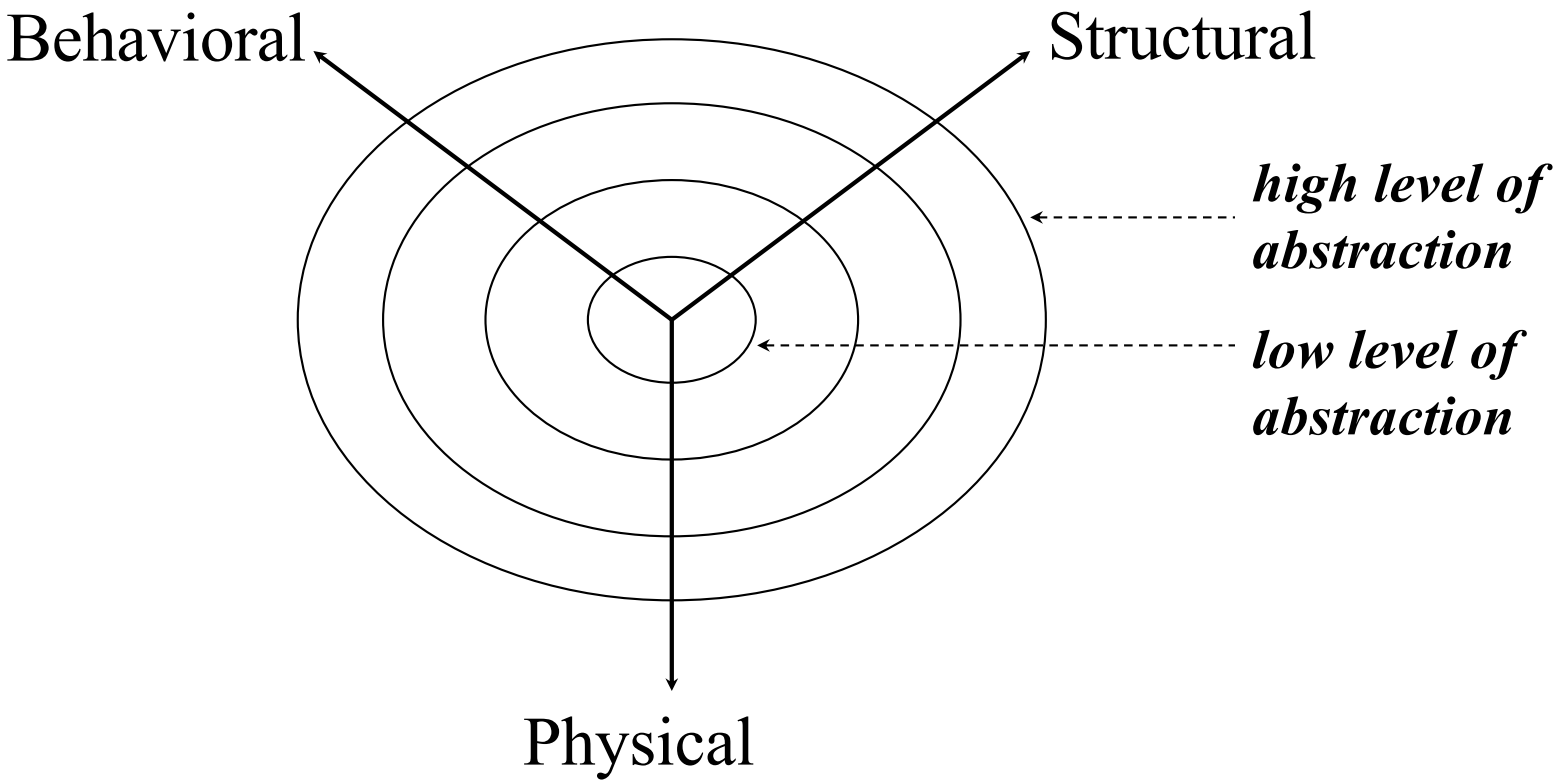
Why Use VHDL?

- Quick Time-to-Market
  - Allows designers to quickly develop designs requiring tens of thousands of logic gates
  - Provides powerful high-level constructs for describing complex logic
  - Supports modular design methodology and multiple levels of hierarchy
- One language for design and simulation
- Allows creation of device-independent designs that are portable to multiple vendors
  → Good for ASIC Migration
  - Allows user to pick any synthesis tool, vendor, or device

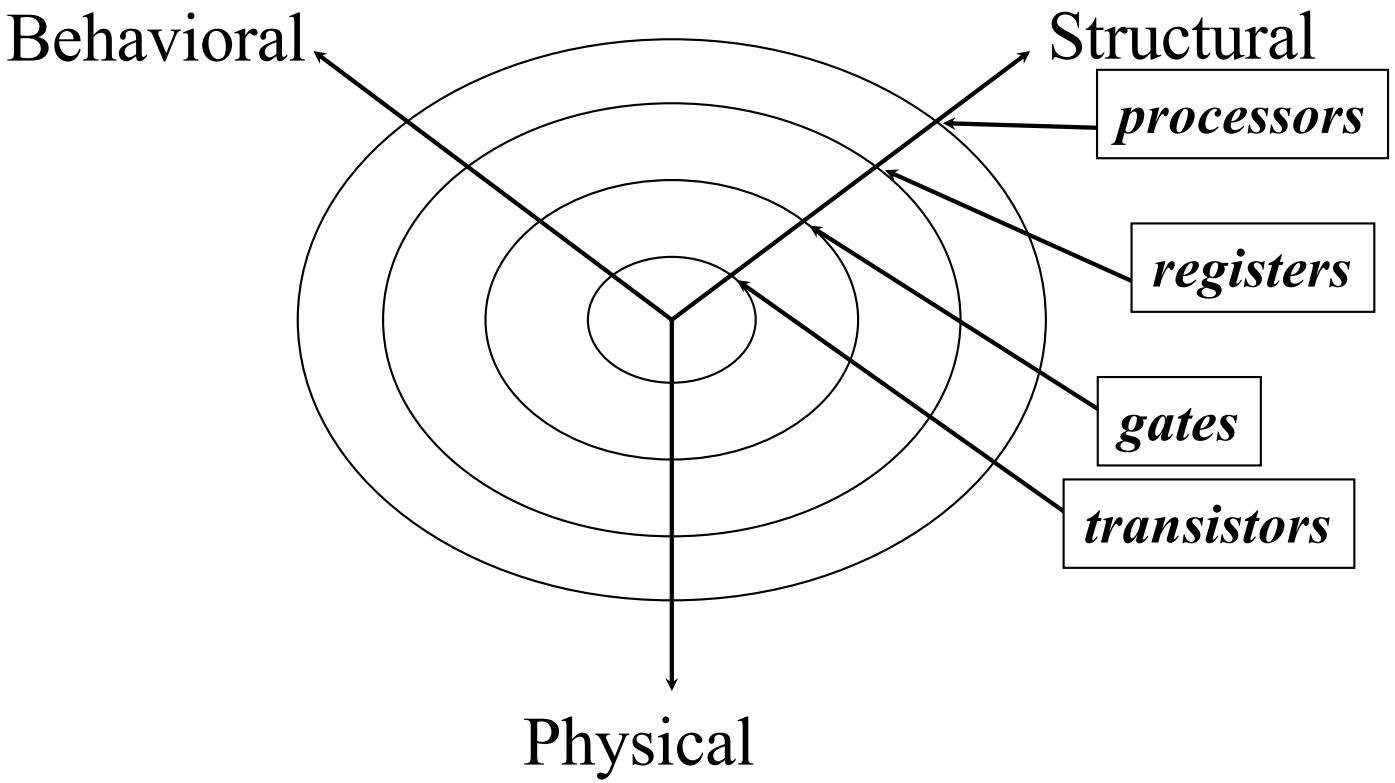
VHDL

- A *hardware description language (HDL)*
- Used as an input to:
  - Synthesis – implement hardware with gates, cells, or FPGAs
  - Simulation – see what your hardware will do before you build it
- Basic unit is a *design entity*
- Design entities have
  - Entity declaration and architecture body
  - Input and output port declarations
  - Internal signal declarations
  - Logic definition
    - Concurrent assignment statements
    - Case statements inside a process
    - Component instantiations
- Language is strongly typed and case insensitive

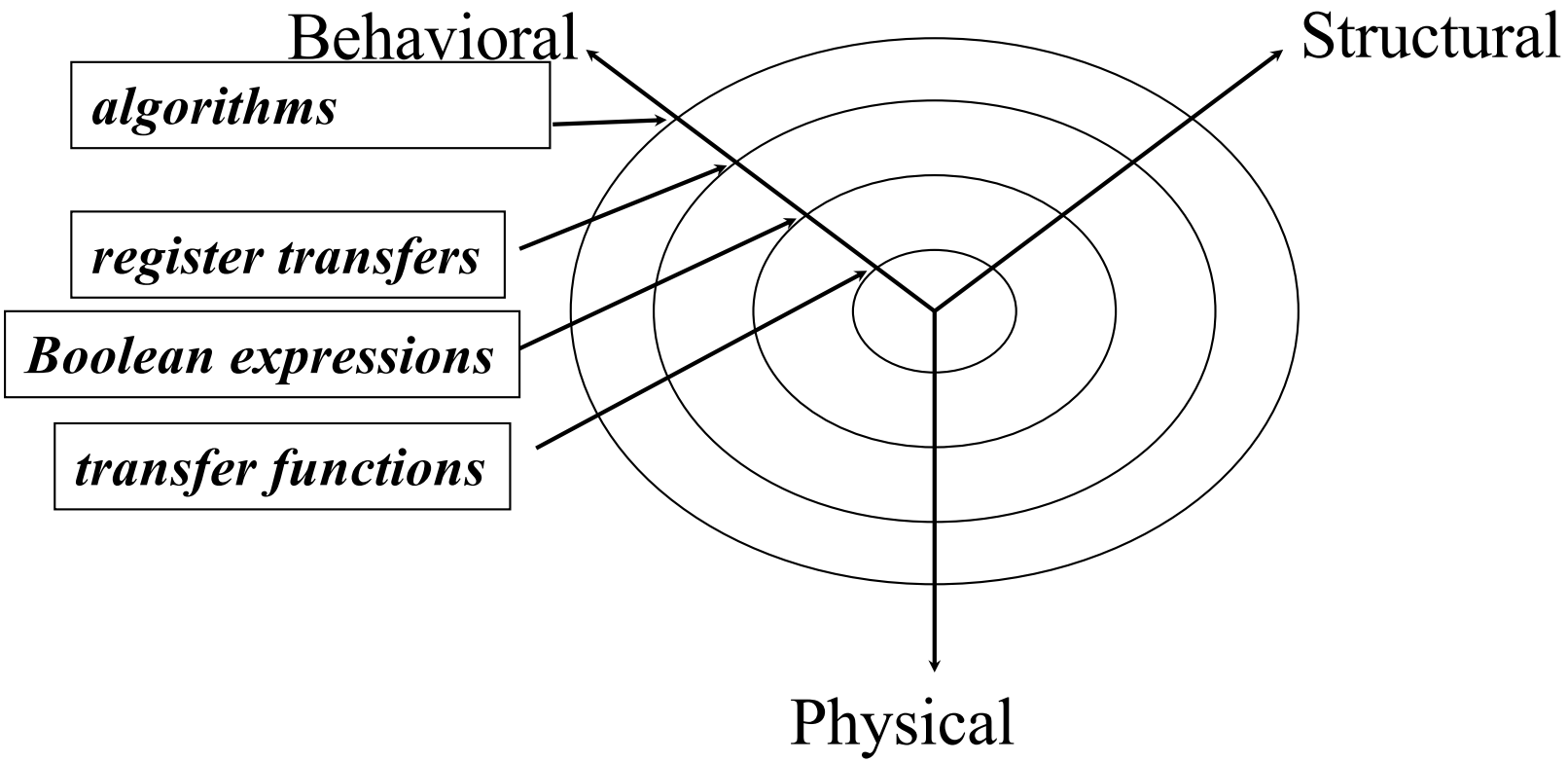
Domains and Levels of Modeling



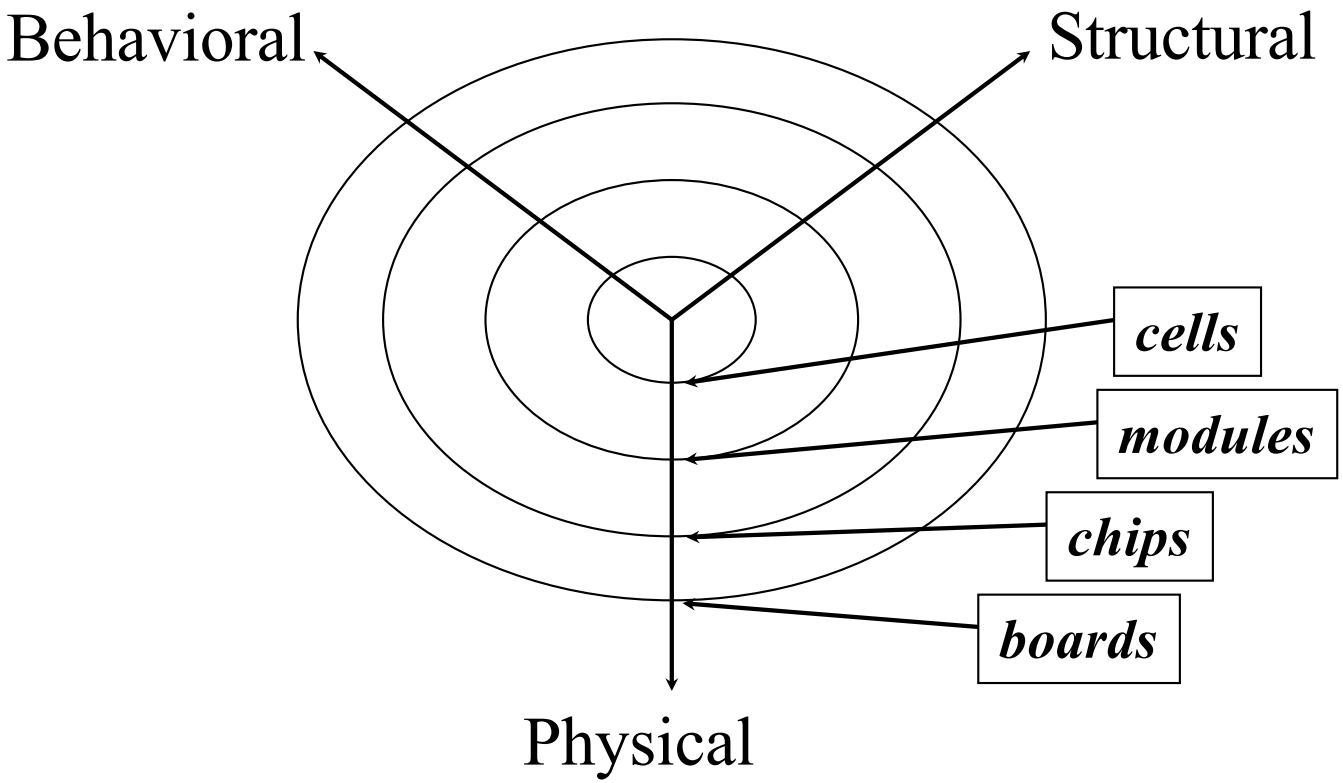
Domains and Levels of Modeling



Domains and Levels of Modeling

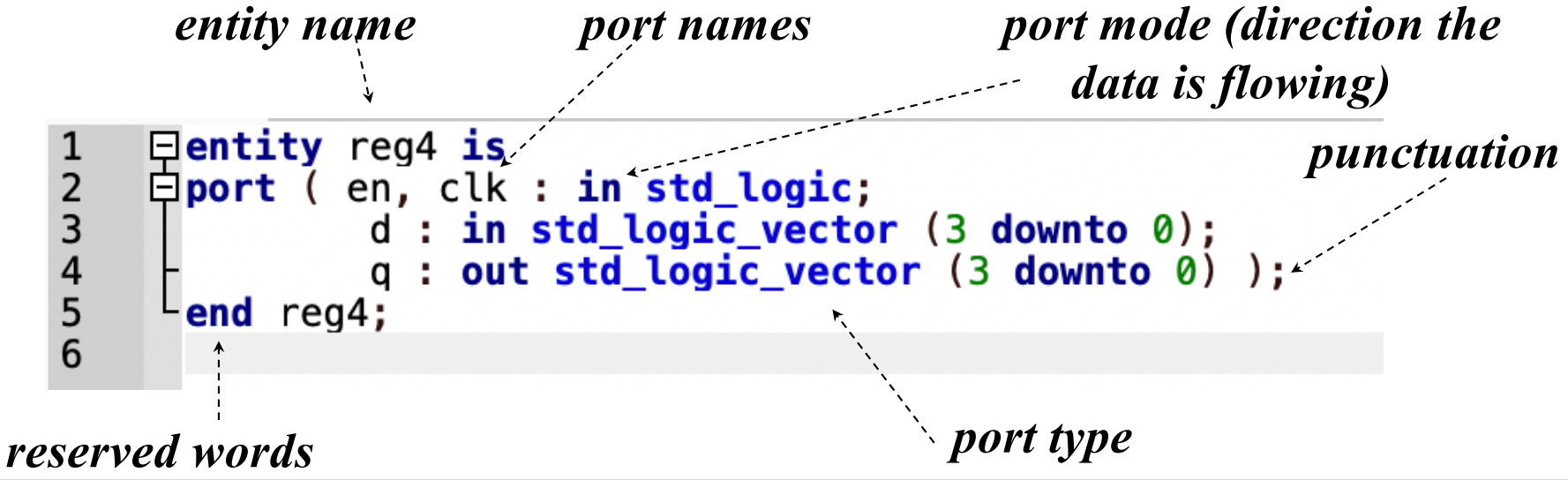


Domains and Levels of Modeling



Simple VHDL: Entity

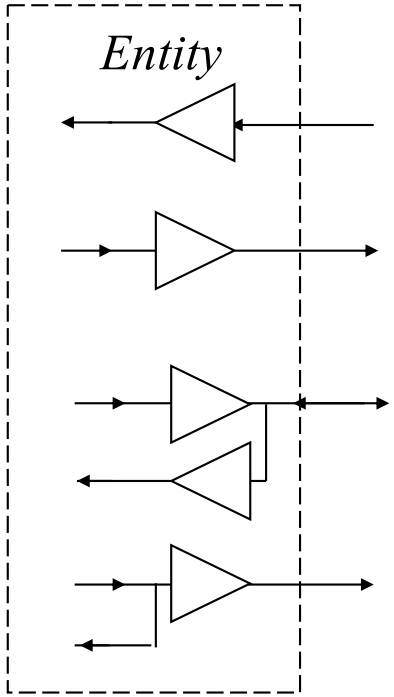
- First step in writing VHDL code is to declare the input and output signals
- Done using a construct called an ***entity***
  - describes the input/output ports of a module



Port Modes

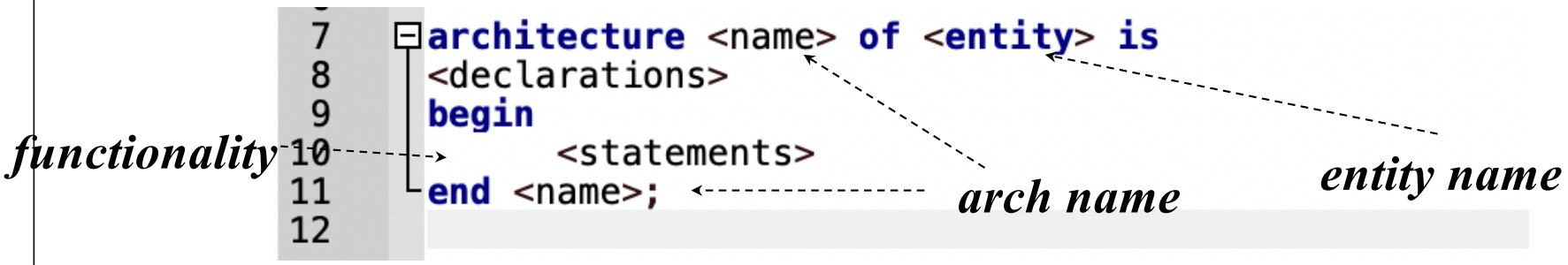
- A port's MODE indicates the direction that data is transferred:

IN	Data goes into the entity only
OUT	Data goes out of the entity only (and is not used internally)
INOUT	Data is bi-directional (goes into and out of the entity)
BUFFER	Data that goes out of the entity and is also fed-back internally



Simple VHDL: Architecture Body

- The entity specifies the inputs and outputs for a circuit, but does not describe the circuit function
- Circuit functionality is specified using a VHDL construct called an ***architecture***
  - Describes an implementation (content or function) of an entity
  - May be several per entity
  - Can be behavioral, data-flow, structural, or combination



Modeling Behavior

- *Behavioral* architecture
  - Describes the algorithm performed by the module
  - Contains *process statements*, each containing
    - *sequential statements*, including
      - *signal assignment statements* and
      - *wait statements (explicit or implicit)*

Implicit wait

```
1   architecture behav of xor_gate is
2   begin
3       new_input : process(a,b)
4       begin
5           if (a = b)
6               f <= 0 after prop_delay;
7           else
8               f <= 1 after prop_delay;
9       end process;
10  end behav;
```

Only for modeling, not synthesis

Uses High-Level programming constructs, like in C or Pascal

Modeling Data-flow Architecture

- ***Consists entirely of boolean equations***
- *References only* ***port signals***

```
 1  entity xor_gate is
 2      port (
 3          a : IN std_logic;
 4          b : IN std_logic;
 5          f : OUT std_logic );
 6  end xor_gate;
 7
 8  architecture dataflow of xor_gate is
 9  begin
10      f  <=  (a AND NOT b)  OR  (NOT a AND b);
11  end dataflow;
```

High-Level programming constructs or rtl/boolean-like descriptions

Sometimes lumped in with behavioral

Obviously not structural

Modeling Structural Architecture

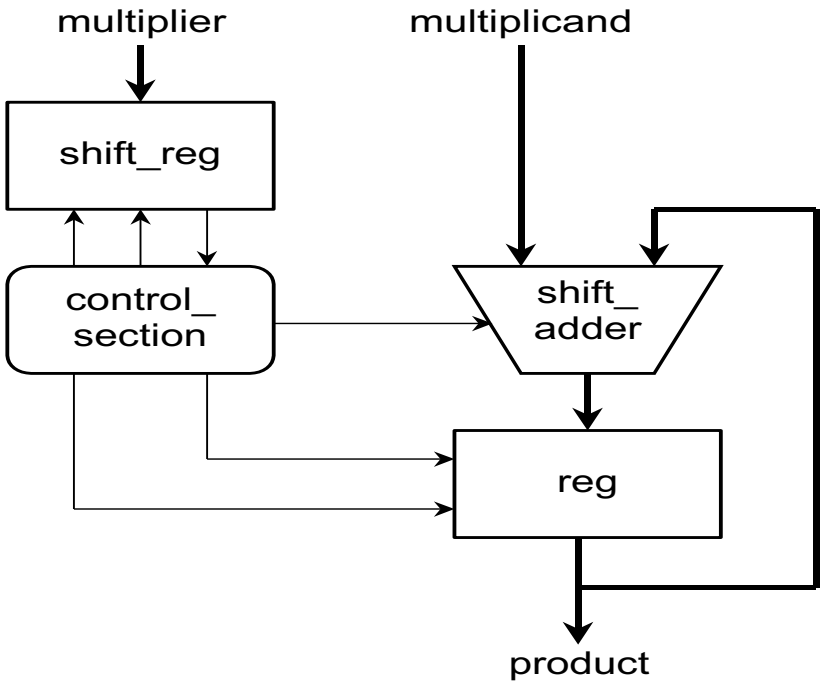
- Defines an entity using **components**
- Internal signals connect components
  - *signal declarations*, for internal interconnections
    - the entity ports are also treated as signals
  - *component instances*
    - instances of previously declared entity/architecture pairs
  - *port maps* in component instances
    - connect signals to component ports
  - *wait statements*

more like a netlist

```
1  architecture structure of xor_gate is
2  component nor_gate
3      port (x, y: IN std_logic;
4            z: OUT std_logic);
5  end component;
6
7  component and_gate
8      port (x, y: IN std_logic;
9            z: OUT std_logic);
10 end component;
11
12 signal s1, s2: std_logic;
13
14 begin
15     nor1: nor_gate PORT MAP (s1, s2, f);
16     and1: and_gate PORT MAP (a, b, s1);
17     nor2: nor_gate PORT MAP (a, b, s2);
18 end structure;
```

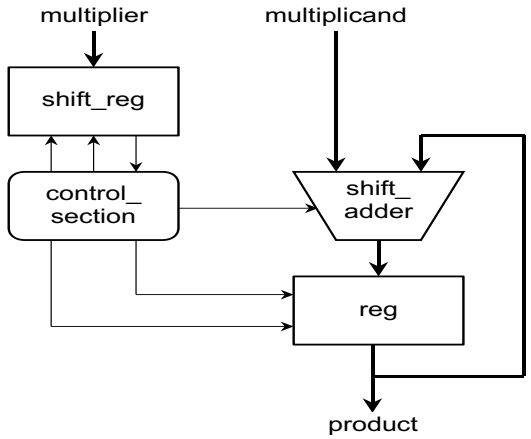
Mixed Behavior and Structure

- An architecture can contain both behavioral and structural parts
  - process statements and component instances
    - collectively called concurrent statements
  - processes can read and assign to signals
- Example: register-transfer-level model
  - data path described structurally
  - control section described behaviorally



Mixed Example

```
1  entity multiplier is
2      port ( clk, reset : in std_logic;
3             multiplicand, multiplier : in integer;
4             product : out integer );
5  end multiplier;
6
7  architecture mixed of mulitplier is
8      signal partial_product, full_product : integer;
9      signal arith_control, result_en, mult_bit, mult_load : std_logic;
10 begin
11     arith_unit : entity work.shift_adder(behavior)
12         port map ( addend => multiplicand,  augend => full_product,
13                    sum => partial_product,add_control => arith_control );
14     result : entity work.reg(behavior)
15         port map ( d => partial_product,  q => full_product,
16                    en => result_en,  reset => reset );
17     …
18     multiplier_sr : entity work.shift_reg(behavior)
19         port map ( d => multiplier,  q => mult_bit,
20                    load => mult_load,  clk => clk );
21     product <= full_product;
22     control_section : process is
23         -- variable declarations for control_section
24         -- …
25     begin
26         -- sequential statements to assign values to control signals
27         -- …
28         wait on clk, reset;
29     end control_section;
30 end mixed;
```



IEEE 1076 Types

- VHDL is a strongly typed language
  - you cannot assign a signal of one type to the signal of another type
  - bit - a signal of type bit that can only take values of '0' or '1'
  - bit_vector - a grouping of bits (each can be '0' or '1')

```
1    SIGNAL  a: BIT_VECTOR(0 TO 3);     -- ascending range
2    SIGNAL  b: BIT_VECTOR(3 DOWNTO 0); -- descending range
3            a <=  "0111"; -- double quotes used for vectors
4            b <=  "0101";
```

This means that:

$$a(0) = '0' \quad b(0) = '1'$$
$$a(1) = '1' \quad b(1) = '0'$$
$$a(2) = '1' \quad b(2) = '1'$$
$$a(3) = '1' \quad b(3) = '0'$$

IEEE 1076 Types (cont.)

- INTEGER
  - useful as index holders for loops, constants, generics, or high-level modeling
- BOOLEAN
  - can take values 'TRUE' or 'FALSE'
- ENUMERATED
  - has user defined set of possible values, e.g.,
  - TYPE traffic_light IS (green, yellow, red);

IEEE 1164

- A package created to solve the limitations of the BIT type
- Nine values instead of just two ('0' and '1')
- Allows increased flexibility in VHDL coding, synthesis, and simulation
  - Can assume several different values [0, 1, Z (high impedance), - (don't care), etc]
- STD_LOGIC and STD_LOGIC_VECTOR are used instead of BIT and BIT_VECTOR when a multi-valued logic system is required
- STD_LOGIC and STD_LOGIC _VECTOR must be used when tri-state logic (Z) is required
- To be able to use this new type, you need to add 2 lines to your code:

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

std_logic and std_logic_vector

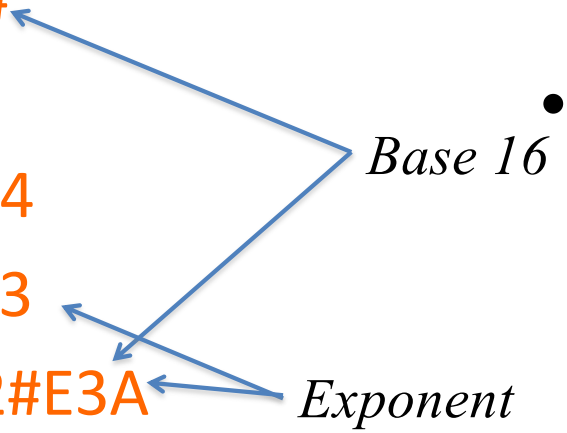
IEEE 1164 Types

- std_logic and std_logic_vector are the industry standard logic type for digital design
- Values for Simulation & Synthesis
  - '0' -- Forcing '0'
  - '1' -- Forcing '1'
  - 'Z' -- High Impedance
  - 'L' -- Weak '0'
  - 'H' -- Weak '1'
  - '-' -- Don't care
- Values for Simulation only (std_ulogic):
  - 'U' -- Uninitialized
  - 'X' -- Forcing Unknown
  - 'W' -- Weak Unknown

Not every simulatable construct is synthesizable or representable in hardware!!

VHDL Literals

- Numbers
  - 14364
  - 14_364
  - 16#FF03#
  - 16.3
  - 13_634.34
  - 1.293E-23
  - 16#3A.B2#E3A



- Bits
  - X"FF_AA"
  - B"1101_1010"
  - O"037"
- Characters
  - ' A'
  - ' _'
  - ''''
  - "This is a string"

Declarations – Signals & Components

- Explicit declaration
  - CONSTANT bus_width : INTEGER := 32;
  - CONSTANT rise_delay : TIME := 20ns;
  - VARIABLE data_val :STD_LOGIC_VECTOR (7 DOWNTO 0);
  - VARIABLE sum : INTEGER RANGE 0 TO 100;
  - VARIABLE done : BOOLEAN;
  - SIGNAL clock : STD_LOGIC;
  - SIGNAL addr_bus : STD_LOGIC_VECTOR (31 DOWNTO 0);

- Implicit declaration
  - FOR count IN 1 TO 10 LOOP –– declares count
    sum := sum + count;
    END LOOP;

- Components

```
1  component fill_reg
2      port(
3           clk :in std_logic;
4           rst :in std_logic;
5           write :in std_logic;
6           read :in std_logic;
7           data_in :in std_logic_vector(3 downto 0);
8           FFout :out std_logic;
9           data_out:out std_logic_vector(31 downto 0));
10 end component;
```

Component Instantiation

- Introduces a relationship to a component declaration
- Port map may be either *named* or *positional*
- Either way, it must match!

```
1   reg5 : fill_reg
2   port map ( clk => clk,
3        rst => rst,
4        write => write_comp,
5        read => readout,
6        data_in => dataout_comp,
7        FFout => FFout,
8        data_out => datareg5_out);
9
10  reg5 : fill_reg
11  port map (clk, rst, write_comp,
12       readout, dataout_comp,
13       FFout, datareg5_out);
```

Preferred!

Boolean Operators In VHDL

- VHDL has built-in support for the following operators
  - AND logical AND
  - OR logical OR
  - NOT logical NOT
  - NAND, NOR, XOR, XNOR
- Assignment operator: <=
  - A variable (usually an output, mode **OUT**) should be assigned the result of the logic expression on the right hand side of the operator
- VHDL does not assume any precedence of logic operators… Use parentheses in expressions to determine precedence
  - Exception to this is NOT logical operator
- In VHDL, a logic expression is called a ***simple assignment statement***. There are other types that will be introduced that are useful for more complex circuits

Native Operators

- Logical - defined for type bit, bit_vector, boolean*
  - AND, NAND
  - OR, NOR
  - XOR, XNOR
  - NOT

•The NOT operator has highest precedence
•The rest are (and, nand, xor, etc.) are of equal precedence so you just used parentheses where necessary

- Relational - defined for types bit, bit_vector, integer*
  - = (equal to)
  - /= (not equal to)
  - < (less than)
  - <= (less than or equal to)
  - > (greater than)
  - >= (greater than or equal to)

•Relational operators return a boolean value
•When comparing vectors of different lengths, the comparison is done from left to right. Therefore "111" is greater than "10111"

** overloaded for std_logic, std_logic_vector; will explain this shortly…*

Native Operators (contd.)

- Unary Arithmetic - defined for type integer*
  - - (arithmetic negate)
- Arithmetic - defined for type integer*
  - +(addition), * (multiplication)
  - - (subtraction)
  - /, mod, rem, abs, **
- Concatenation - defined for strings
  - &

- Note, a STRING is any sequence of characters, therefore a std_logic_vector is an example of a STRING

** overloaded for std_logic, std_logic_vector; will explain this shortly…*

Overloaded Operators

- In VHDL, the scope of all of the previous operators can be extended (or overloaded) to accept any type supported by the language, e.g.,

```
--      assume a declaration of a 16-bit vector as
  SIGNAL pc IS std_logic_vector(15 DOWNTO 0);
--      then a valid signal assignment is
  pc <= pc + 3;
--      assuming the '+' operator has been overloaded to
---     accept std_logic_vector and integer operands
```

- The std_logic_1164 package defines overloaded logical operators (AND, OR, NOT, etc.,) for the std_logic and std_logic_vector types

Legal VHDL Identifiers

- Letters, digits, and underscores only
  (first character must be a letter)
- The last character cannot be an underscore
- Two underscores in succession are not allowed
- Using reserved words is not allowed (the VHDL editor will highlight reserved words for this reason)
- Examples
  - Legal
    - tx_clk, Three_State_Enable, sel7D, HIT_1124
  - Not Legal
    - _tx_clk, 8B10B, large#num, case, clk_

VHDL Thoughts

- May look somewhat like a conventional programming language (e.g. C, java, etc.)
- Fundamentally different!!!!
  - In C: only one statement is active at a time and executed in sequence
  - In VHDL: ALL design entities and ALL concurrent assignment statements in EACH component are active all of the time!!!!!
    →So ALL statements are executed ALL OF THE TIME!
- Always remember that code is complied into hardware
  - Each component instantiated adds a hardware component to the design
  - Each assignment statement in each design entity adds gates to each instance of that design entity
- Very powerful but can be an impediment if the designer loses touch with the desired end product…. DON'T HACK!

VHDL design style – for synthesizable modules

1. Combinational design entities use only
    1. Concurrent assignment statements
    2. Case or Case? statements (with “when others =>”)
    3. If statements – only if all signals have a default assignment
    4. Instantiations of other combinational modules
2. Sequential design entities use only
    1. Combinational logic
    2. Explicitly declared registers (flip-flops)
3. Do not use
    1. Loops
    2. Process except for case, casex, or if
4. Do use
    1. Signal slices, e.g., a(7 downto 1) = b(6 downto 0) ;
5. Logic is organized into small design entities
    1. Leaf design entities not more than 40 lines
    2. If it could be made two design entities, it should be

VHDL design style – for synthesizable modules (page 2)

6. Use lots of comments
    1. Comments themselves
    2. Meaningful signal names – tempHigh, not th
    3. Meaningful module names – DaysInMonth, not Mod3
7. Sensitivity lists for case statements include **ALL** inputs (or use "all" keyword)
8. Constants
    1. All constants explicitly defined if used more than once
9. Signals
    1. Buses (multi-bit signals) are numbered high to low
        - e.g., bus(31 downto 0)
    2. All signals should be high-true (except primary inputs and outputs)
10. Visualize the logic your VHDL will generate.
    - If you can't visualize it, the result will not be pretty

Architecture Body May Contain:

- Concurrent Signal Assignment statements (CSAs)
  - a <= b + c;
  - Activated when any of the associated signals change their values
  - Independent from other statements in the given architecture
- Component Instantiations
  - Pairs the entity with the architecture
  - Defines a sub-component of the design
  - Associates signals or values with the ports
- Processes with behavioral descriptions
  - An independent sequential process representing the behavior of some portion of the design

Other Assignment Statements

- For ***concurrent assignment statements (or concurrent signal assignments - CSAs)*** the order in which they appear in VHDL code does not affect the meaning of the code
- VHDL provides a second category of statements, ***sequential assignment statements***, for which the ordering of the statements **may** affect the meaning of the code
  - **If then else** and **case** statements are examples of sequential stmts
- VHDL requires that sequential assignment statements be placed inside another statement, the ***process*** statement
- Therefore, all behavioral designs require caution!

What is a VHDL "Process" ?

- Processes are either awake or asleep (active or inactive)
- A process normally has a sensitivity list
  - When a signal in that sensitivity list changes value, the process wakes up and all of the sequential statements are "executed"
  - For example, a process with a clock signal in its sensitivity list will become active on changes of the clock signal
- At the end of the process, all outputs are assigned and the process goes back to sleep until the next time a signal changes in the sensitivity list

```
1  <label> process ( <sensitivity list> )
2      < declarations >
3      begin
4          < statements >
5  end process <label>;
```

Process Statement

- Process executes or is evaluated whenever an event occurs on a signal in the sensitivity list
  - Remember: No sensitivity list is required, BUT if not, wait statements must be used in the process
- Multiple processes can execute concurrently
- The statements describing the behavior are executed **sequentially**
  - This is true from a simulation standpoint
  - From a synthesized hardware point-of-view, multiple assignments to a single signal (variable) generally implies multiplexing of the assignments to produce a single output
- Assignments made inside the process are not visible outside the process until all statements in the process have been evaluated
  - If there are multiple assignments to the same signal inside a process, only the last one has any visible effect
- The <label> is optional

Process Example (from a simulation testbench)

```
1   process
2   begin
3        wait for 15 ns;
4        clk <= not(clk);
5   end process;
6
7   process (clk, rst)
8   begin
9       if rst = '1' then
10          readout <= '0';
11      elsif (clk'event and clk= '1') then
12          if (fout = '1') then
13              readout <= '1';
14          else
15              readout <='0';
16          end if;
17      end if;
18  end process;
```

```
1   <label> process ( <sensitivity list> )
2       < declarations >
3       begin
4           < statements >
5   end process <label>;
```

2 processes operating in parallel

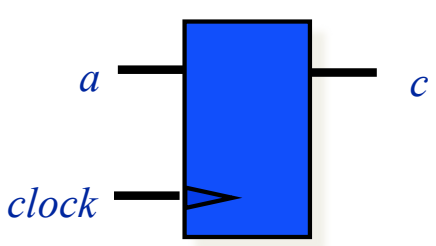
1st process generates the clock

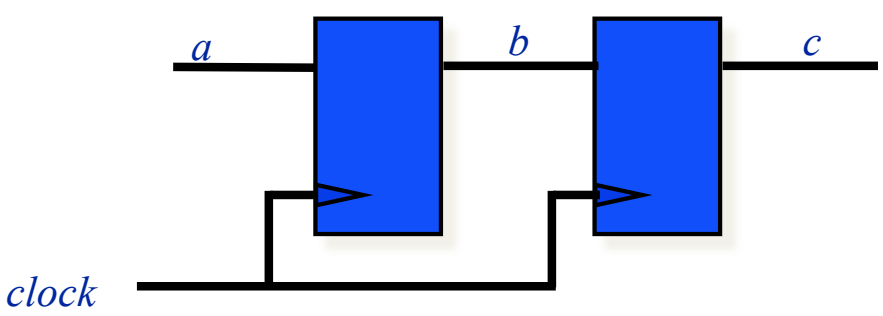
2nd process has the clock in the sensitivity list

Signal Assignment in Processes

Which Circuit do you think is Correct?

```
1   -- assume a and c are signals defined elsewhere
2
3   ARCHITECTURE arch_reg OF reg IS
4       signal b: std_logic;
5   BEGIN
6       PROCESS (a,b)
7       BEGIN
8           if (rising_edge(clk)) then
9               b <= a; -- a goes to b after clock
10              c <= b; -- b goes to c after clock
11          end if;
12      END PROCESS;
13  END arch_reg;
```





Signal Assignment in Processes

- Inside processes, signals are not updated immediately
  → Instead, they are scheduled to be updated
- Signals are updated at the END PROCESS
- Therefore, on the previous slide, two registers will be synthesized (c <= b will be the old b)
- In some cases, the use of a concurrent statement outside the process will fix the problem, but this is not always possible

- So how else can we fix this problem ?

Variables

- When a concurrent signal assignment outside the process cannot be used, the previous problem can be avoided using a variable
  - Variables are like signals, BUT they can only be used inside a given PROCESS (signals are inside the architecture)
  - They cannot be used to communicate information between processes
  - Variables can be of any valid VHDL data type
  - The value assigned to a variable is available immediately
  - Assignment of variables is done using a colon (:), like this:

    c := a AND b;

Using Variables vs. Signals

- Solution using a variable within a process:

```
1   -- assume a and c are signals defined elsewhere
2
3   ARCHITECTURE arch_reg OF reg IS
4   BEGIN
5       PROCESS (a,b)
6           VARIABLE b: std_logic;
7       BEGIN
8           if (rising_edge(clk)) then
9               b := a; -- immediate
10              c <= b; -- scheduled
11          end if;
12      END PROCESS;
13  END arch_reg;
```

Concurrency

- "Concurrent" means **nonprocedural (i.e. not in a Process statement!)**
  - The order in which the CSA stmts appear textually has nothing to do with the order in which they execute
  - They execute at the same time!!!!!!!!
- Note: `INT1` and `INT2` are internal signals, used only within the **architecture** (keep scope in mind)

```
 1  entity XOR2_OP is
 2  port (A, B : in BIT;
 3            Z : out BIT);
 4  end XOR2_OP;
 5
 6  architecture AND_OR_CONCURRENT of XOR2_OP is
 7      signal INT1, INT2 : BIT;
 8  begin
 9      INT1 <= A and not B;
10      INT2 <= not A and B;
11      Z <= INT1 or INT2;
12  end AND_OR_CONCURRENT;
```

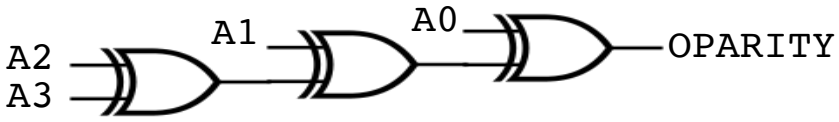
Concurrency (cont.)

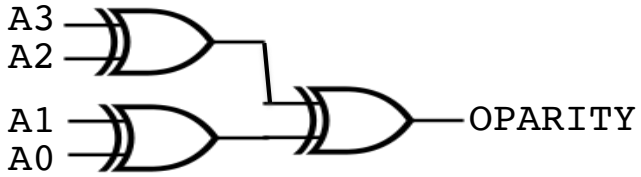
- Concurrency is fundamental to hardware and VHDL!
- Think in terms of *parallel signal transformations*
- Design tips:
  - Large procedural descriptions are ***hard to synthesize***
  - Concurrency – must think about how it is ***realized*** – "visualize the possible hardware"
  - Think about design ***decomposition*** (modular and hierarchical)

- Next example shows two designs for an odd parity detector

- Think about which one will have better performance (i.e. yield a faster design)?

VHDL Stmts & Hardware Concurrency






```
1   entity OPARITY_DETECT is
2   port (A3, A2, A1, A0 : in std_logic;
3         OPARITY : out std_logic);
4   end OPARITY_DETECT;
5
6   architecture design1 of OPARITY_DETECT is
7       signal INT1, INT2 : std_logic;
8   begin
9       INT1 <= A3 xor A2;
10      INT2 <= INT1 xor A1;
11      OPARITY <= INT2 xor A0;
12  end Design1;
```

```
1   entity OPARITY_DETECT is
2   port (A3, A2, A1, A0 : in std_logic;
3         OPARITY : out std_logic);
4   end OPARITY_DETECT;
5
6   architecture design2 of OPARITY_DETECT is
7       signal INT1, INT2 : std_logic;
8   begin
9       INT1 <= A3 xor A2;
10      INT2 <= A0 xor A1;
11      OPARITY <= INT2 xor INT1;
12  end design2;
```

Combinational Logic

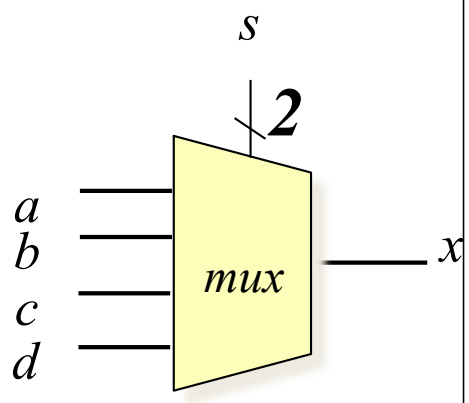
- Can be described with concurrent statements
  - boolean equations
  - when-else
  - with-select-when
  - component instantiations
- Can be described with sequential statements
  - if-then-else
  - case-when

Lets look at examples of each…

Combinational Logic w/ Boolean Equations

- Boolean Equations can be used in both concurrent and sequential signal assignment statements…
- A 4-1 multiplexer is shown below

```
1       x <=  (a AND NOT(s(1)) AND NOT(s(0))) OR
2           (b AND NOT(s(1)) AND s(0)) OR
3           (c AND s(1) AND NOT(s(0))) OR
4           (d AND s(1) AND s(0)) ;
```



concurrent

Selective Signal Assignment: with-select-when

- Assignment based on a selection signal
- WHEN clauses must be mutually exclusive
- Use a WHEN OTHERS when all conditions are not specified
- Only one reference to the signal, only one assignment operator (<=)

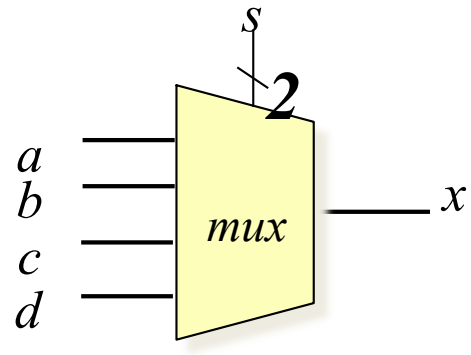
```
WITH selection_signal SELECT
signal_name <= value_1 WHEN value_1 of
  selection_signal,
        value_2 WHEN value_2 of selection_signal,
        ...
        value_n WHEN value_n of selection_signal,
        value_x  WHEN OTHERS;
```


concurrent

Combinational Logic w/ Selective Signal Assignment

- The same 4-1 multiplexer is shown below

```
1    with s select
2        x <= a when "00" ,
3             b when "01" ,
4             c when "10" ,
5             d when others ;
```

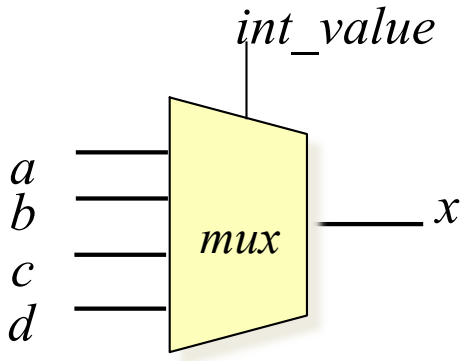


concurrent

More on with-select-when

- You can use a range of values

```
1  with int_value select
2      x <= a when 0 to 3,
3           b when 4 | 6 | 8 ,
4           c when 10 ,
5           d when others ;
```



concurrent

Conditional Signal Assignment: when-else

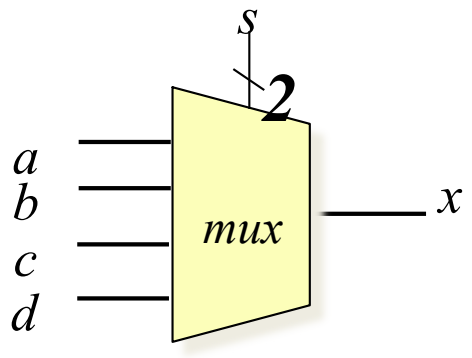
- Assign a value based on conditions
- Any simple expression can be a condition
- Priority goes in order of appearance
- Only one reference to the signal, only one assignment operator (<=)
- Use a final ELSE to avoid latches

```
signal_name <= value_1 WHEN condition1 ELSE
      value_2 WHEN condition2 ELSE
      ...
      value_n WHEN condition N ELSE
      value_x ;
```


concurrent

- The same 4-1 multiplexer is shown below

```
1    x <= a when (s = "00") else
2         b when (s = "01") else
3         c when (s = "10") else
4         d ;
```



concurrent

Combinational Logic w/ Conditional Signal Assignment

- The when conditions do not have to be mutually exclusive (as in with-select-when)
- A priority encoder is shown below
  - Order of the conditions matters!!!!!!!

```
1    j <= w when (a = '1') else
2         x when (b = '1') else
3         y when (c = '1') else
4         z when (d = '1') else
5         "000" ;
```


concurrent

Example, VHDL for Thermostat

Entity declaration

Include some standard libraries

List of I/O

3-bit wide signals

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4
5   entity thermostat is
6     port ( presetTemp, currentTemp : in std_logic_vector( 2 downto 0 );
7            fanOn : out std_logic );
8   end thermostat;
9
10  architecture impl of thermostat is
11  begin
12    fanOn <= '1' when currentTemp > presetTemp
13                 else '0';
14  end impl;
```

Architecture body

Concurrent conditional signal assignment statement updates fanOn value whenever currentTemp changes

Combinational Logic w/ Sequential Statements

- Grouped inside **Process Statements**
- Processes are concurrent with *one another* and with *concurrent signal assignment statements*
- Order of sequential statements ***does*** make a difference in synthesis

sequential

Sequential Statements: if-then-else

- Used to select a set of statements to be executed
- Selection based on a boolean evaluation of a condition or set of conditions
- Absence of ELSE results in implicit memory (FF or latch)

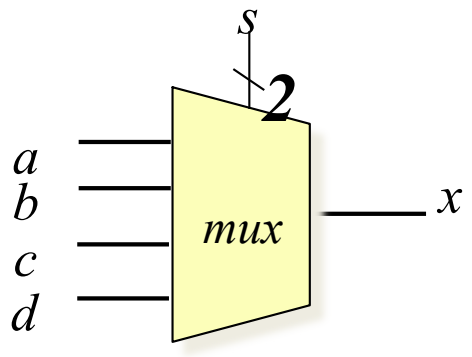
```
IF condition(s) THEN
  do something;
ELSIF condition_2 THEN-- optional
  do something different;
ELSE       -- optional
  do something completely different;
END IF ;
```


sequential

if-then-else

- 4-1 mux shown below

```
1  mux4_1: process (a, b, c, d, s)
2          begin
3              if  s = "00" then x <= a ;
4                  elsif  s = "01" then x <= b ;
5                  elsif  s = "10" then x <= c ;
6                  else   x <= d ;
7              end if;
8          end process mux4_1 ;
```



sequential

Sequential Statements: Case-When

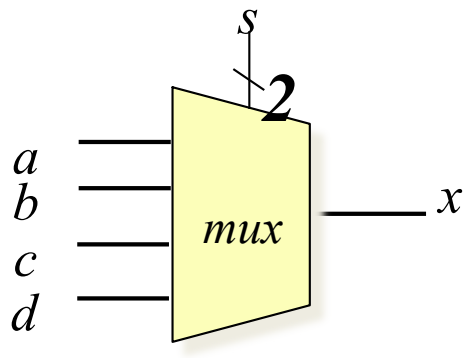
```
CASE selection_signal IS
  WHEN value_1_of_selection_signal  =>
   (do something) -- set of statements 1
  WHEN value_2_of_selection_signal  =>
   (do something) -- set of statements 2
...
  WHEN value_N_of_selection_signal  =>
   (do something) -- set of statements N
  WHEN OTHERS  =>
   (do something) -- default action
END CASE ;
```

(sort of the sequential counterpart to the with-select-when)

sequential

The CASE Statement: 4-1 Mux

```
1  ARCHITECTURE archdesign OF design IS
2      SIGNAL s: std_logic_vector(0 TO 1);
3  BEGIN
4      mux4_1: PROCESS (a,b,c,d,s)
5      BEGIN
6          CASE s IS
7              WHEN "00"  => x <= a;
8              WHEN "01"  => x <= b;
9              WHEN "10"  => x <= c;
10             WHEN OTHERS => x <= d;
11         END CASE;
12     END PROCESS mux4_1;
13 END archdesign;
```



sequential

Example, Days in Month Function

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   entity DaysInMonth is
4     port( month : in std_logic_vector(3 downto 0); -- 1=Jan, 12=Dec
5           days : out std_logic_vector(4 downto 0) ); -- # days in month
6   end DaysInMonth;
7   architecture impl of DaysInMonth is
8   begin
9     process(month) begin
10      case month is
11        when 4d"4" | 4d"6" | 4d"9" | 4d"11" => days <= 5d"30";
12        when 4d"2" => days <= 5d"28";
13        when others => days <= 5d"31";
14      end case;
15    end process;
16  end impl;
```

Process evaluates each time "sensitivity list" changes. In this case each time *month changes*

Case statement selects statement depending on value of argument. Like a truth table.

Can have multiple values per statement

Default ("when others") covers values not listed.

Sequential Statements: Case?-When

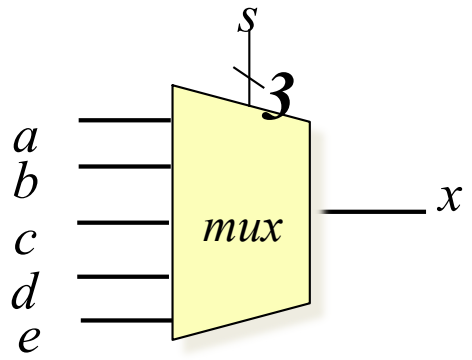
```
CASE? selection_signal IS
  WHEN value_1_of_selection_signal  =>
   (do something) -- set of statements 1
  WHEN value_2_of_selection_signal  =>
   (do something) -- set of statements 2
...
  WHEN value_N_of_selection_signal  =>
   (do something) -- set of statements N
  WHEN OTHERS  =>
   (do something) -- default action
END CASE ;
```

Matching case statement… allows don't cares

sequential

The CASE? Statement: 5-1 Mux

```
1  ARCHITECTURE archdesign OF design IS
2      SIGNAL s: std_logic_vector(0 TO 2);
3  BEGIN
4      mux5_1: PROCESS (a,b,c,d,e,s)
5      BEGIN
6          CASE? s IS
7              WHEN "00-"  => x <= a;
8              WHEN "010"  => x <= b;
9              WHEN "011"  => x <= c;
10             WHEN "11-"  => x <= d;
11             WHEN OTHERS => x <= e;
12         END CASE;
13     END PROCESS mux5_1;
14 END archdesign;
```

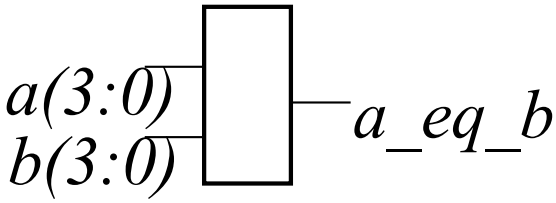


sequential

Architecture Declaration of a Comparator

- The entity declaration is as follows:

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   ENTITY compare IS PORT (
4       a, b  : IN std_logic_vector(3 DOWNTO 0);
5       a_eq_b: OUT std_logic);
6   END compare;
```



- Write an architecture that causes **a_eq_b** to be asserted when **a** is equal to **b**
- Multiple solutions exist

Three possible solutions

- *Concurrent* statement solution using a *conditional* assignment:

```
1  ARCHITECTURE df1_compare OF compare IS
2  BEGIN
3      a_eq_b <= '1' WHEN a = b ELSE '0';
4  END df1_compare;
```

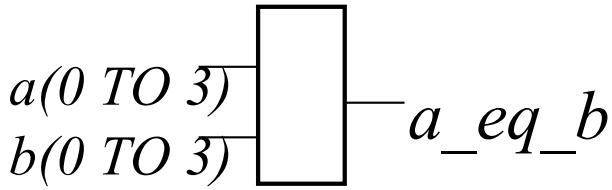
- *Concurrent* statement solution using boolean equations:

```
1  ARCHITECTURE df2_compare OF compare IS
2  BEGIN
3      a_eq_b <= NOT(
4          (a(0) XOR b(0)) OR
5          (a(1) XOR b(1)) OR
6          (a(2) XOR b(2)) OR
7          (a(3) XOR b(3)));
8  END df2_compare;
```

Three possible solutions

- Solution using a process with sequential statements:

```
1   ARCHITECTURE behv_compare OF compare IS
2   BEGIN
3   comp: PROCESS (a, b)
4       BEGIN
5           IF a = b THEN
6               a_eq_b <= '1';
7           ELSE
8               a_eq_b <= '0';
9           END IF;
10      END PROCESS comp;
11  END behv_compare;
```



ADDITIONAL EXAMPLES

4-bit Prime Number Function in VHDL Code – Using case

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

entity prime is
  port( input : in std_logic_vector(3 downto 0);
        isprime : out std_logic );
end prime;

architecture case_impl of prime is
begin
  process(input) begin
    case input is
      when x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d" => isprime <= '1';
      when others => isprime <= '0';
    end case;
  end process;
end case_impl;
```

Encoding truth table directly

4-bit Prime Number Function in VHDLCode – Using a "case?" statement

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity prime is
 5    port( input : in std_logic_vector(3 downto 0);
 6          isprime : out std_logic );
 7  end prime;
 8
 9  architecture mcase_impl of prime is
10  begin
11    process(all) begin
12      case? input is
13        when "0--1" => isprime <= '1';
14        when "0010" => isprime <= '1';
15        when "1011" => isprime <= '1';
16        when "1101" => isprime <= '1';
17        when others => isprime <= '0';
18      end case?;
19    end process;
20  end mcase_impl;
```

Each 'when clause' must not overlap

4-bit Prime Number Function in VHDLCode – Using a iIF statement

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity prime is
5     port( input : in std_logic_vector(3 downto 0);
6           isprime : out std_logic );
7   end prime;
8
9   architecture mcase_impl of prime is
10  begin
11    process(all) begin
12      if input = 4d"1" then isprime <= '1';
13      elsif input = 4d"2" then isprime <= '1';
14      elsif input = 4d"3" then isprime <= '1';
15      elsif input = 4d"5" then isprime <= '1';
16      elsif input = 4d"7" then isprime <= '1';
17      elsif input = 4d"11" then isprime <= '1';
18      elsif input = 4d"13" then isprime <= '1';
19      else isprime <= '0';
20      end if;
21    end process;
22  end mcase_impl;
```

Avoid as much as possible
Too easy to generate a sequential ckt by excluding the 'else' clause

Rule of thumb… only use if statements when specifying the next state function in state machines

4-bit Prime Number Function in VHDL Code – Using a concurrent signal assignment statement

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity prime is
5     port( input : in std_logic_vector(3 downto 0);
6           isprime : out std_logic );
7   end prime;
8
9   architecture logic_impl of prime is
10  begin
11    isprime <= (input(0) AND (NOT input(3))) OR
12               (input(1) AND (NOT input(2)) AND (NOT input(3))) OR
13               (input(0) AND (NOT input(1)) AND input(2)) OR
14               (input(0) AND input(1) AND NOT input(2)) ;
15  end logic_impl;
```

Note, no process statement

Which is a better description?

- Using “case”
- Using “case?”
- Using “concurrent signal assignment”

Test benches use a very different style of VHDL

- Process without sensitivity list
- Entity is empty
- report
- looping constructs
- wait statements

- Don’t use these constructs in synthesizable code

Test bench

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_unsigned.all;
 4  use ieee.numeric_std.all;
 5
 6  entity test_prime is
 7  end test_prime;
 8
 9  architecture test of test_prime is
10    signal input: std_logic_vector(3 downto 0);
11    signal isprime: std_logic;
12  begin
13    -- instantiate module to test
14    DUT: entity work.prime(case_impl) port map(input, isprime);
15
16    process begin
17      for i in 0 to 15 loop
18        input <= std_logic_vector(to_unsigned(i,4));
19        wait for 10 ns;
20        report "input = " & to_string(to_integer(unsigned(input))) &
21               " isprime = " & to_string(isprime);
22      end loop;
23      std.env.stop(0);
24    end process;
25  end test;
```

Entity is empty

Device under test

Simulation stimulus

Test Bench Example 2

```
 1  Library IEEE;
 2    use IEEE.STD_LOGIC_1164.all;
 3    use IEEE.STD_LOGIC_ARITH.all;
 4    use IEEE.std_logic_unsigned.all;
 5  Library WORK;
 6    use WORK.all;
 7
 8  entity testbench is
 9  end testbench;
10
11  architecture test_adder of testbench is
12    signal clk : std_logic:= '0';
13    signal rst : std_logic:= '0';
14
15    signal a : std_logic_vector(4 downto 0);
16    signal b : std_logic_vector(4 downto 0);
17    signal c : std_logic_vector(4 downto 0);
18
19    begin
20    dut : entity adder
21      port map(
22        in1 => a, in2 => b, out => c );
23
```

Entity is empty

Device under test

Test Bench Example 2 (cont)

```
24    -- ************************************
25    -- process for simulating the clock
26    process
27    begin
28      clk <= not(clk);
29      wait for 20 ns;
30    end process;
31
32    -- ************************************
33    -- This process does the RESET
34    process
35    begin
36      rst <= '1';
37      wait for 53 ns;
38      rst <= '0';
39      wait until(rst'event and rst = '1');
40    -- stops this process from happening again (this is an initial )
41    end process;
42
```

Simulation stimuli

Parallel processes!!!

Test Bench Example 2 (cont)

```
43    -- ************************************
44    -- This process provides the stimuli
45    process
46      variable i : integer;
47      variable j : integer;
48      begin
49        for i in 1 to 16 loop
50          for j in 1 to 16 loop
51            a <= CONV_STD_LOGIC_VECTOR(i, 5);
52            b <= CONV_STD_LOGIC_VECTOR(j, 5);
53            wait until (clk'event and clk='1');
54          end loop;
55        end loop;
56    end process;
57    --*****************************************
58    end test_adder;
```

converts *i* or *j* to `std_logic_vector` of size 5 bits

Summary

- VHDL
  - Describe hardware for simulation and synthesis
  - Can represent with case, case?, concurrent assignment, or structurally
  - Use representation that is readable and maintainable
    - case for truth tables
    - Concurrent assignment for equations
    - Don't try to do the logic design yourself
  - Synthesis tool will do the optimization
  - Test benches check that implementation meets its specification
    - Test benches use a different style of VHDL

General HDL modeling recommendations

- Before attempting to code a model, know and understand what it is you are modeling
- Divide your design into sub-designs when possible
  - Promotes short, easy to read, reusable code
- Make models as generic as possible
  - Parameterize bus widths
- Use meaningful signal names
  - For active low signals use **<signal_name>_n** for clearer understanding and easier debugging
- Use comments liberally
  - A header should describe each module and each signal declaration should have a comment
  - Dated, inconsistent comments are worse than none at all

VHDL Generics

- VHDL generics may be used to parameterize code making components as modular and hence reusable as possible
- For example:

```
1   ENTITY counters IS
2   GENERIC ( WIDTH : INTEGER := 8 );
3       PORT(
4           d : IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
5           clk : IN STD LOGIC;
6           clear : IN STD_LOGIC;
7           load : IN STD_LOGIC;
8           up_down : IN STD_LOGIC;
9           qd : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0));
10  END counters;
```

Generate Statement

- Whenever we write structural VHDL code and instantiate components, we often create ***instances*** of a particular ***component***
  - A multi-stage ripple carry adder made from a number of single-bit full adders might be an example
- If we need to create a large number of instances of a component, a more compact form is desired
- VHDL provides a feature called the **FOR GENERATE** statement
  - This statement provides a loop structure for describing regularly structured hierarchical code
  - NOTE: The loop cannot be terminated early

```
name : FOR N IN 1 TO 8 GENERATE
        concurrent-statements
END GENERATE name;
```

Generate Statement Example

- In this example, we will build a 32-bit wide shift register using a edge triggered DFF with clear

```
 1  entity edge_trig_Dff is
 2      port (D, clk, clr, en: in std_logic;
 3            Q: out std_logic);
 4  end entity edge_triggered_Dff;
 5
 6  architecture behav of edge_trig_Dff is
 7  begin
 8      state_change: process(clk, clr) is
 9      begin
10          if clr = '1' then
11              Q <= '0';
12          elsif clk'event and clk = '1' and en = '1' then
13              Q <= D;
14          end if;
15      end process state_change;
16  end architecture behavioral;
```

Example Cont.

- Using this DFF, we want to place 32 of them in a row and wire the i^{th} input to the $(i-1)^{st}$ output
- The first input should be wired to *shift-in* and the last output should be wired to *shift-out*
- We could do this with 32 component instantiation statements of the form:

```
1    dff1: component edge_trig_Dff
2            port map (Sin, clk, clr, en, parallel_data(1) );
3    dff2: component edge_trig_Dff
4            port map (parallel_data(1), clk, clr, en, parallel_data(2) );
5    dff3: ...
```

Example Cont.

- Or… we could use *generate* to do the work for us

```
1   shift_reg: for i in 1 to 32 generate
2   begin
3       dffx: generate
4       begin
5           dff: component edge_trig_Dff
6               port map (parallel_data(i-1), clk, clr, en, parallel_data(i) );
7       end generate dffx;
8   end generate shift_reg;
```

- Here, we are placing 32 copies of our DFF and wiring them up. There is a problem, however… There is no 0 element (*i*-1 when *i*= 1)
- Another problem exists at the 32nd element where we need to wire the shift out

32-bit Shift Register

```
1   entity shift_right_32 is
2   generic ( width : integer :=32 );
3       port (Sin, clk, en, clr: in std_logic;
4             Sout: out std_logic);
5   end entity shift_right_32;
6
7   architecture reg32 of shift_right_32 is
8       signal parallel_data: std_logic_vector(0 to width-1);
9
10      component edge_trig_Dff
11          port(D, clk, clr, en: in std_logic;
12               Q: out std_logic);
13      end component edge_trig_Dff;
14
15  begin
16      dff1: component edge_trig_Dff
17          port map (Sin, clk, clr, en, parallel_data(0) );
18
19      shift_reg: for i in 1 to width-2 generate
20      begin
21          dffx: generate
22          begin
23              dff: component edge_trig_Dff
24                  port map (parallel_data(i-1), clk, clr, en, parallel_data(i) );
25          end generate dffx;
26      end generate shift_reg;
27
28      dff32: component edge_trig_Dff
29          port map (parallel_data(width-2), clk, clr, en, Sout );
30  end architecture;
```

Sources of additional information

- Various sources of information exist that should help you learn and use good coding practice
  - Intel/Altera online help in Quartus II gives a set of guidelines for VHDL coding
  - Altera website provides numerous design examples
  - Internet
    - Here you can see questions people typically have about VHDL and programmable devices. Look at the answers/hints that other people post to **help you learn**
    - DO NOT post questions directly related to your homework/project assignments to these groups!!!!!!
    - DO NOT copy and submit code from the internet as your own!!!!

Up Next

- Lecture 3: Chapters 4, 5 (CMOS)
- Lecture 4: Chapter 8 (Programmable Devices)
- Lecture 5: Chapters 14, 15, 16, 19 (Synch Sequential Logic)