L3: Background Part 3 – VHDL

Notices & reminders

- Lecture time for Friday 29/08: 9:00 – 11:00
- I will only respond to email that requires administrative involvement by the LiC, or that raises a query about coursework that a tutor has not been able to answer
  - Email martindg@cse for web related, administrative, or organisational matters
  - Email your_tutor@cse for queries about coursework, and CC: to comp3211_tutor-list@cse
  - Send from a UNSW mail service

Overview

This lecture

1. Introduction to VHDL
   - Use in describing systems
2. Describing & modelling systems
   - Signals, events, timing, concurrency
   - Use of discrete event simulation

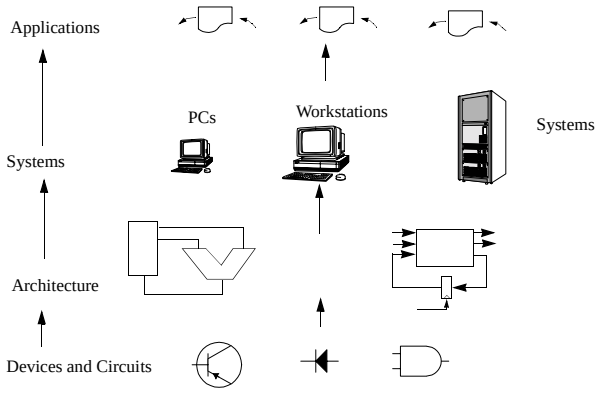
Next few lectures

3. Basic language constructs
   - Describing the interface and behaviour of components
   - Signal assignments
   - Delays
   - Modelling complex behaviour
   - Modelling state machines
   - Modelling structure
   - Subprograms, packages, and libraries

1. Introduction to VHDL

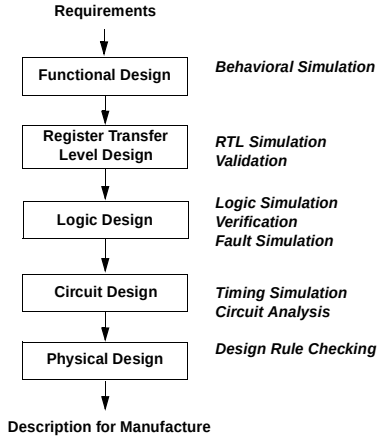
- Use in describing systems

Describing Digital Systems



- Systems may be described at multiple levels of abstraction

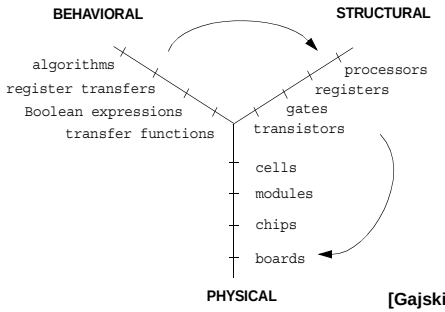
The Digital System Design Process



Why Do We Describe Systems?

- Design Specification
  - unambiguous definition of components and interfaces in a large design
- Design Simulation
  - verify system/subsystem/chip performance prior to design implementation
- Design Synthesis
  - automated generation of a hardware design

Hardware Description Languages



[Gajski and Kuhn]

- Design is structured around a hierarchy of representations
- HDLs can describe distinct aspects of a design at multiple levels of abstraction

The VHDL Language

V *Very High Speed Integrated Circuit*

H *Hardware*

D *Description*

L *Language*

- Interoperability between design tools: standardized portable model of electronic systems
- Technology independent description
- Reuse of components described in VHDL

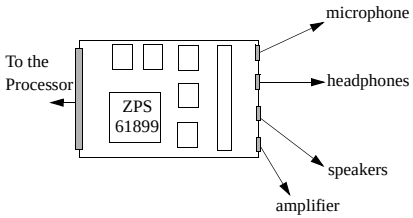
History of VHDL

- Designed by IBM, Texas Instruments, and Intermetrics as part of the DoD funded VHSIC program
- Standardized by the IEEE in 1987: IEEE 1076-1987
- Enhanced version of the language defined in 1993: IEEE 1076-1993
- Additional standardized packages provide definitions of data types and expressions of timing data
  - IEEE 1164 (data types)
  - IEEE 1076.3 (numeric)
  - IEEE 1076.4 (timing)

2. Describing & modelling systems

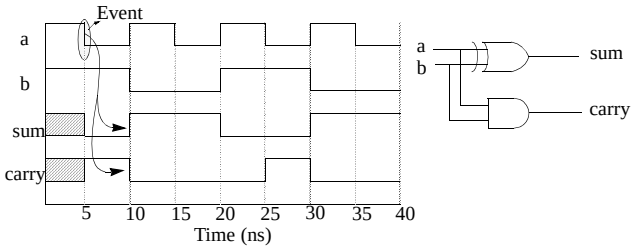
- Signals, events, timing, concurrency
- Use of discrete event simulation

Describing Systems



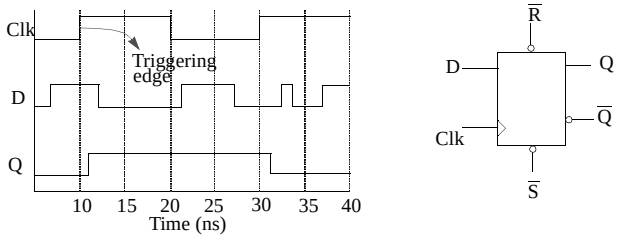
- System: “An assemblage of objects united by some form or regular interaction or dependence”
- What aspects of a digital system do we want to describe?
  - interface
  - behavior: functional and structural

Behavioral Attributes of Digital Systems



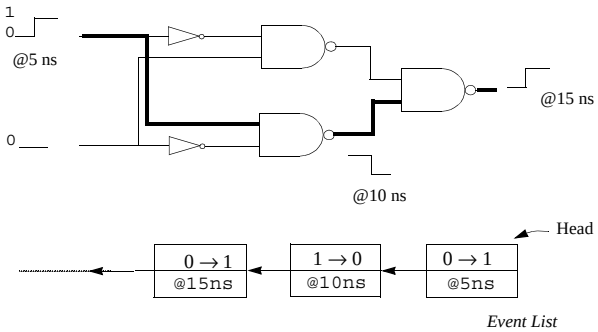
- Digital systems are about *signals* and their *values*
- *Events*, *propagation delays*, *concurrency*
- Time ordered sequence of events produces a *waveform*

Behavioral Attributes of Digital Systems



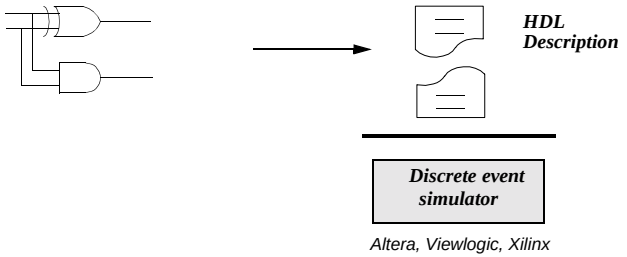
- *Timing*: computation of events takes place at specific points in time
- Timing is an attribute of both synchronous and asynchronous systems

Simulation of Digital Systems



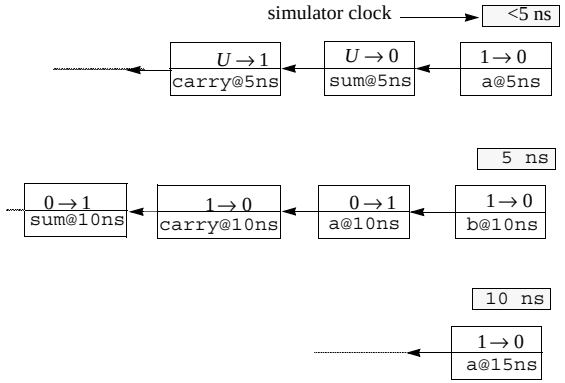
- Digital systems propagate events
- Discrete event simulations manage the generation and recording of events

Simulation Modeling



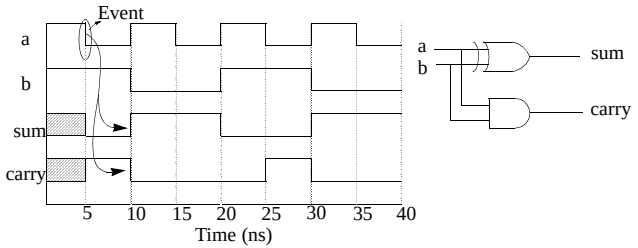
- VHDL programs describe the generation of events in digital systems
- Discrete event simulator manages event ordering and progression of time

Discrete Event Simulation: Half Adder Model



- Management of simulation time: ordering of events
- Two step model of the progression of time

Behavioral Attributes of Digital Systems



- Digital systems are about *signals* and their *values*
- *Events*, *propagation delays*, *concurrency*
- Time ordered sequence of events produces a *waveform*

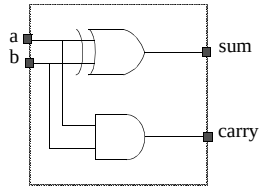
Modeling Digital Systems

- VHDL programs describe behavioral attributes of digital systems
  - events, propagation delays, concurrency
  - waveforms and timing
  - signal values
- Discrete event simulators mimic the operation of the system being described
  - two step model of time
    - assign values scheduled for signals at the current time
    - process affected components and schedule future signal values
  - ensures events are generated and processed in the correct order
  - a correct simulation generates *only and all* those events that would have been generated in the physical system

3. Basic language constructs

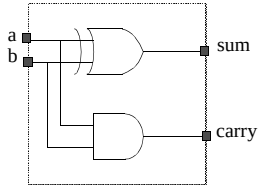
- Describing the interface and behaviour of components
- Signal assignments
- Delays
- Modelling complex behaviour
- Modelling state machines
- Modelling structure
- Subprograms, packages, and libraries

Basic Language Concepts



- What aspects of a digital system do we want to describe?
  - interface: how do we connect to it
  - behavior: what does it do?

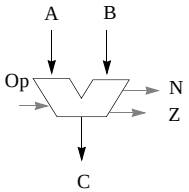
Describing the Interface: The Entity Construct



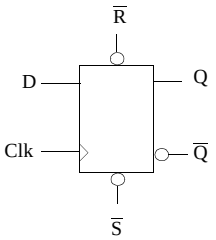

```
entity half_adder is
port ( a, b : in bit;
       sum, carry :out bit);
end  half_adder;
```

- The interface is a collection of *ports*
  - ports are a new programming object: *signal*
  - ports have a type, e.g., **bit**
  - ports have a mode: in, out, inout (bidirectional)

Example Entity Descriptions

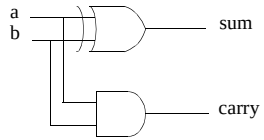



```
entity ALU32 is
port(  A, B : in bit_vector  (31 downto 0);
       C : out bit_vector  (31 downto 0);
       Op : in bit_vector  (5 downto 0);
       N, Z : out bit);
end ALU32;
```




```
entity D_ff is
port(  D, Q, Clk, R, S: in bit;
       Q, Qbar : out bit);
end D_ff;
```

Describing Behavior: The Architecture Construct

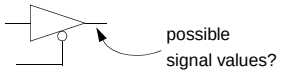



```
entity half_adder is
port (a, b : in bit;
       sum, carry :out bit);
end  half_adder;

architecture behavioral of half_adder
is
begin
sum <= (a xor b) after 5 ns;
carry <= (a and b) after 5 ns;
end behavior;
```

- Description of events on output signals in terms of events on input signals: the *signal assignment statement*
- Specification of propagation delays
- Type **bit** is not powerful enough for realistic simulation: use the IEEE 1164 value system

Behavioral Attributes of Digital Systems

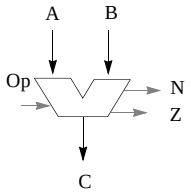
- We associate logical values with the state of a signal



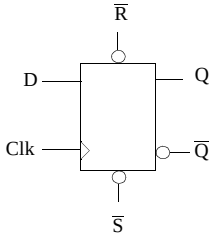
- Signal Values: IEEE 1164 Value System

Value	Interpretation
U	Uninitialized
X	Forcing Unknown
0	Forcing 0
1	Forcing 1
Z	High Impedance
W	Weak Unknown
L	Weak 0
H	Weak 1
-	Don't Care

Example Entity Descriptions: IEEE 1164

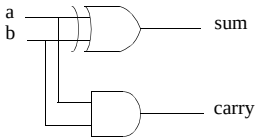



```
entity ALU32 is
port(  A, B: in std_logic_vector (31 downto 0);
       C : out std_logic_vector (31 downto 0);
       Op: in std_logic_vector (5 downto 0);
       N, Z: out std_logic);
end ALU32;
```




```
entity D_ff is
port(  D, Q, Clk, R, S: in std_ulogic;
       Q, Qbar : out std_ulogic);
end D_ff;
```

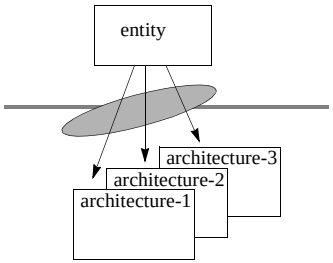
Describing Behavior: The Architecture Construct




```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port ( a, b : in std_logic;
         sum, carry :out std_logic);
end  half_adder;

architecture behavioral of half_adder
is
begin
sum <= (a xor b) after 5 ns;
carry <= (a and b) after 5 ns;
end behavior;
```

- Use of the IEEE 1164 value system simply requires inclusion of the library and package declarations statements

Design Units

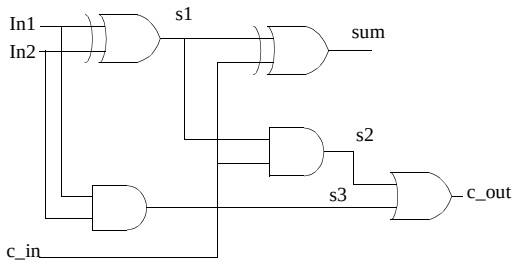


- An entity may have multiple architectures
- Separation of interface from implementation

Simple Signal Assignment

```
library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
port (in1, in2, c_in: in std_logic;
      sum, c_out: out std_logic);
end full_adder;
```

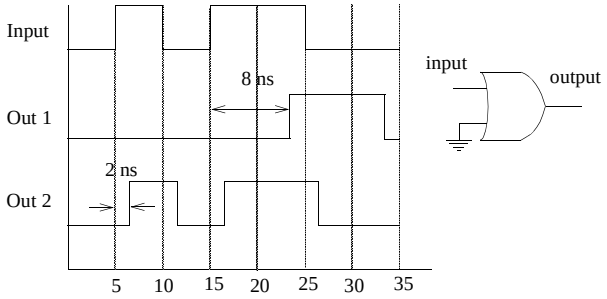
```
architecture dataflow of full_adder is
signal s1, s2, s3 : std_logic;
constant gate_delay: Time:= 5 ns;
begin
L1: s1 <= (In1 xor In2) after gate_delay;
L2: s2 <= (c_in and s1) after gate_delay;
L3: s3 <= (In1 and In2) after gate_delay;
L4: sum <= (s1 xor c_in) after gate_delay;
L5: c_out <= (s2 or s3) after gate_delay;
end dataflow;
```



Simple Signal Assignment Statement

- The *constant* programming object
  - values cannot be changed
- Use of signals in the architecture
  - internal signals used to connect components
- A statement is executed when a signal in the RHS has value assigned to it
  - 1-1 correspondence between signal assignment statements and signals in the circuit
  - order of statement execution follows propagation of events in the circuit
  - textual order *does not* imply execution order
  - trace the statement execution order when In1 changes value

Understanding Delays



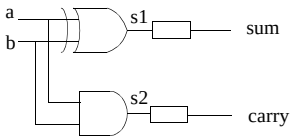
- Inertial delay model
- Transport delay model
- Delta delay model

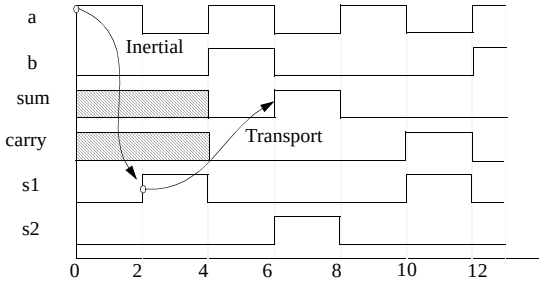
Understanding Delays

- Inertial delay
  - default delay model
  - suitable for modeling delays through devices such as gates
- Transport Delay
  - to model delays through devices with very small inertia, e.g, wires
  - all input events are propagated to output signals
- Delta delay
  - what about models where no propagation delays are specified?
  - infinitesimally small delay is automatically inserted by the simulator to preserve correct ordering of events

Transport Delays: Example

```
architecture transport_delay of
half_adder is
signal s1, s2: std_logic:= '0';
begin
s1 <= (a xor b) after 2 ns;
s2 <= (a and b) after 2 ns;
sum <= transport  s1 after 4 ns;
carry <= transport  s2 after 4 ns;
end transport_delay;
```

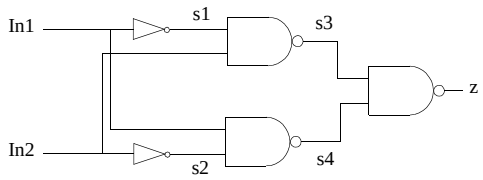




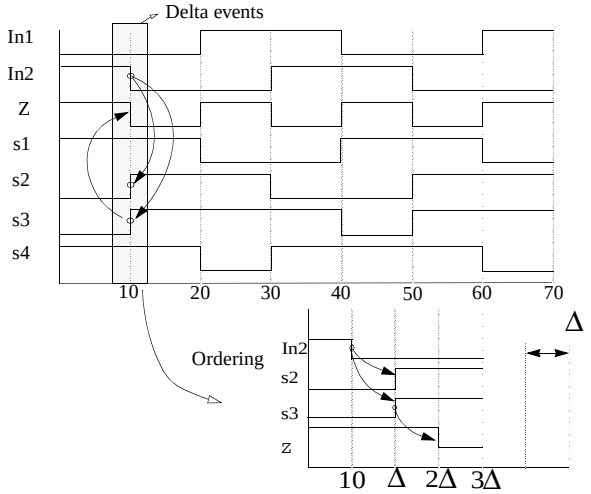
Delta Delays: Example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
port (In1, In2: in std_logic;
      z : out std_logic);
end combinational;
```

```
architecture behavior of
combinational is
signal s1, s2, s3, s4: std_logic:=
begin
s1 <= not  In1;
s2 <= not In2;
s3 <= not (s1 and In2);
s4 <= not (s2 and In1);
z <= not (s3 and s4);
end behavior;
```



Delta Delays: Example



Conditional Signal Assignment

```
library IEEE;
use  IEEE.std_logic_1164.all;
entity mux4 is
port (    In0, In1, In2, In3  : in std_logic_vector  (7 downto 0);
          Sel: in std_logic_vector(1  downto  0);
          Z  : out std_logic_vector  (7 downto 0));
end mux4;

architecture behavioral of mux4 is
begin
Z <= In0  after 5 ns when Sel = "00" else
     In1  after 5 ns when Sel = "01" else
     In2  after 5 ns when Sel = "10" else
     In3  after 5 ns when Sel = "11" else
     "00000000" after 5 ns;
end behavioral;
```

Evaluation order is important!

- First true conditional determines the output value

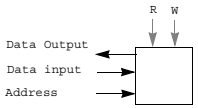
Selected Signal Assignment Statement

```
library IEEE;
use  IEEE.std_logic_1164.all;
entity mux4 is
port (    In0, In1, In2, In3  : in std_logic_vector (7 downto 0);
          Sel: in std_logic_vector(1  downto  0);
          Z  : out std_logic_vector (7 downto 0));
end mux4;

architecture behavioral-2 of mux4 is
begin
with Sel select
Z <= (In0 after 5 ns) when "00",
     (In1 after 5 ns) when "01",
     (In2 after 5 ns) when "10",
     (In3 after 5 ns) when "11"
     (In3 after 5 ns) when others;
end behavioral;
```

All options must be covered and only one must be true!

- The "when others" clause can be used to ensure that all options are covered

Modeling Complex Behavior



Memory Module

add R1, R2, R3
sub R3, R4, R5
move R7, R3
•
•
•

Instruction Set Simulation

- Concurrent signal assignment statements can easily capture the gate level behavior of digital systems
- Higher level digital components have more complex behaviors
  - input/output behavior not easily captured by concurrent signal assignment statements
  - models utilize state information
  - incorporate data structures
- We need more powerful constructs

The Process Statement

```
library IEEE;
use  IEEE.std_logic_1164.all;
entity mux4 is
port (    In0, In1, In2, In3  : in std_logic_vector (7 downto 0);
          Sel: in std_logic_vector(1  downto  0);
          Z  : out std_logic_vector (7 downto 0));
end mux4;

architecture behavioral-3 of mux4 is
begin
process (Sel, In0, In1, In2, In3)
variable Zout: std_logic;
begin
      if (Sel = "00") then Zout:= In0;
      elsif (Sel = "01") then Zout:= In1;
      elsif (Sel = "10") then Zout:= In2;
      else Zout := In3;
      end if;
      Z <= Zout;
end process;
end behavioral;
```

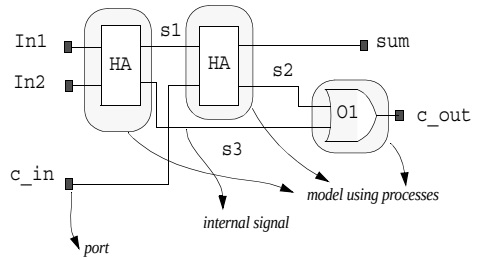
Sensitivity List

Variable Assignment

The Process Construct

- Statements in a process are executed sequentially
- A process body is structured much like conventional C or Pascal programs
  - declaration and use of variables
  - *if-then*, *if-then-else*, *case*, *loop* and *while* constructs
  - process can contain signal assignment statements
- A process executes concurrently with other concurrent signal assignment statements
- A process takes 0 simulation time to execute and may schedule events in the future: we can think of a process as a complex signal assignment statement!

Concurrent Processes: Full Adder



- Each of the components of the full adder can be modeled using a process
- Processes execute concurrently
- Processes communicate via signals

Concurrent Processes: Full Adder

```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port (In1, c_in, In2 : in std_logic;
      sum, c_out : out std_logic);
end full_adder;

architecture behavioral of
full_adder is
signal s1, s2, s3: std_logic;
constant delay :Time:= 5 ns;
begin
HA1: process (In1, In2)
begin
s1 <= (In1 xor In2) after delay;
s3 <= (In1 and In2) after delay;
end process HA1;

HA2: process(s1,c_in)
begin
sum <= (s1 xor c_in) after
delay;
s2 <= (s1 and c_in) after delay;
end process HA2;

OR1: process (s2, s3) -- process
describing the two-input OR gate
begin
c_out <= (s2 or s3) after delay;
end process OR1;
end behavioral;
```

Concurrent Processes: Half Adder

```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port (a, b : in std_logic;
sum, carry : out std_logic);
end half_adder;

architecture behavior of
half_adder is
begin
sum_proc: process(a,b)
        begin
          if (a = b) then
              sum <= '0' after 5 ns;
          else
              sum <= (a or b) after 5 ns;
          end if;
        end process;

 carry_proc: process (a,b)
begin
case a is
when '0' =>
carry <= a after 5 ns;
when '1' =>
carry <= b after 5 ns;
when others =>
carry <= 'X' after 5 ns;
end case;
end process carry_proc;
end behavior;
```

The Wait Statement

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is
port (D, Clk : in std_logic;
      Q, Qbar : out std_logic);
end dff;

architecture behavioral of dff is
begin
output: process
begin
wait until (Clk'event and Clk = '1'); -- wait for edge

   Q <= D after 5 ns;
   Qbar <= not D after 5 ns;
end process output;
end behavioral;
```

signifies a value change on signal clk

- wait for <*time expression*>, wait on <*signal*>, wait until <*boolean expression*>

The Wait Statement: Waveform Generation

```
library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out
std_logic);
end two_phase;

architecture behavioral of
two_phase is
begin
rproc: reset <= '1', '0' after 10 ns;
clock_process: process
begin
phi1 <= '1', '0' after 10 ns;
phi2 <= '0', '1' after 12 ns, '0' after
18 ns;
wait for 20 ns;
end process clock_process;
end behavioral;
```

The Wait Statement

- A process can have multiple wait statements
- A process cannot have both a wait statement and a sensitivity list (it should have one or the other)

VHDL Identifiers, Objects, Types, & Operators

- Identifiers
  - Used as names for variables, signals, constants, and design units such as entities, architectures, etc.
  - Composed of alphanumeric characters and underscore
  - Must start with a letter; may not end with underscore
  - Are not case-sensitive

Data Objects

- Classes: signals, variables, constants, and files
- Range of permissible values determined by type
  - Signals represent wires
  - Signals differ from variables in that:
    - Signals are scheduled to receive values at some time by the simulator, and one can schedule multiple values at distinct instants in the future
    - Variables are assigned during execution of assignment statements and can only be assigned one value at a time – variables are essentially equivalent to their conventional programming language counterparts
  - Constants must be declared & initialized at start of simulation & cannot be modified during the simulation

Data Types

- Specify the range of values an object may take and the set of operations that can be performed on it:

Type	Range of values	Example declaration
integer	implementation dependent	**signal** index:**integer**:=0;
real	implementation dependent	**variable** val:**real**:=1.0;
boolean	(TRUE,FALSE)	**variable** test:**boolean**:=TRUE;
character	defined in package STANDARD	**variable** term:**character**:='@';
bit	0, 1	**signal** In1:**bit**:='0';
bit_vector	array of bit	**variable** PC:**bit_vector**(31 **downto** 0);
time	implementation dependent	**variable** delay:**time**:=25 **ns;**
string	array of char	**variable** name:**string(**1 **to** 10):="model name";
natural	0 to maxint	**variable** index:**natural**:=0;
positive	1 to maxint	**variable** index:**positive**:=1;

Enumerated Types

- The standard set provided by VHDL can be augmented through user-defined types
- Consider troble with "bit" signals, thus we have:

```
type std_ulogic is ('U','0','1',
    'Z','W','L','H','-');
```

  then we can declare

```
signal carry:std_ulogic:='U';
```

- The definition explicitly enumerates all possible values that an object of this type can assume

Another example

- Define new type:

```
type instr_opcode is
    ('add','lw','beq',..);
```

  then can have

```
case opcode
when beq =>..
```

- Can place such type declarations in declarative region of architecture, process, or package

Array Types

- Very common in hardware, e.g., a word is an array of bits, memory an array of words, a data bus an array of data lines...
- E.g.

```
type byte is
    array (7 downto 0) of bit;
```

Physical Types

- Motivated by need to represent physical quantities such as time, voltage, power, etc.
- E.g., **type** time **is range** 0 **to** 1000000

```
units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;...
    hr = 60 min;
end units;
```


Operators

- Used in expressions involving objects

logical operators	**and**	**or**	**nand**	**nor**	**xor**	**xnor**
relational operators	**=**	**/=**	**<**	**<=**	**>**	**>=**
shift operators	**sll**	**srl**	**sla**	**sra**	**rol**	**ror**
addition operators	**+**	**-**	**&**			
unary operators	**+**	**-**				
multiplying operators	*****	**/**	**mod**	**rem**		
miscellaneous operators	****	**abs**	**not**			

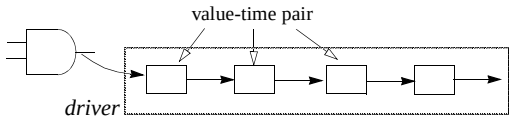
- There is increasing precedence from top to bottom
- All operators within the same class have the same precedence and are evaluated in textual order from left to right
- Use parentheses to control evaluation order

Attributes

- Data can be obtained about VHDL items such as types, arrays and signals.

`object'attribute`

- Example: consider the implementation of a signal



  - what types of information about this signal are useful?
    - occurrence of an event
    - elapsed time since last event
    - previous value

Value Kind Attributes

- Return a constant value

type `statetype` **is** `(state0, state1, state2 state3);`

  - `state_type`'**left** = `state0`
  - `state_type`'**right** = `state3`

Value attribute	Value
`type_name`'**left**	returns the left most value of `type_name` in its defined range
`type_name`'**right**	returns the right most value of `type_name` in its defined range
`type_name`'**high**	returns the highest value of `type_name` in its range
`type_name`'**low**	returns the lowest value of `type_name` in its range
`array_name`'**length**	returns the number of elements in the array `array_name`

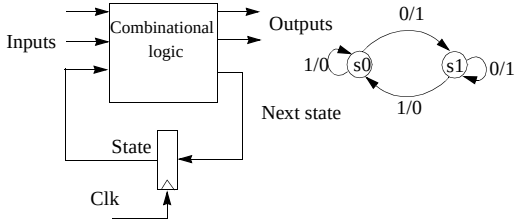
Function Kind Attributes

- Use of attributes invokes a function call which returns a value
  if (`Clk`'**event and** `Clk` = '1')

  function call
- Function signal attributes

Function attribute	Function
`signal_name`'**event**	Return a Boolean value signifying a change in value on this signal
`signal_name`'**active**	Return a Boolean value signifying an assignment made to this signal. This assignment may not be a new value.
`signal_name`'**last_event**	Return the time since the last event on this signal
`signal_name`'**last_active**	Return the time since the signal was last active
`signal_name`'**last_value**	Return the previous value of this signal

State Machines



- Basic components
  - combinational component: output function and next state function
  - sequential component
- Natural process implementation

Example: State Machine

```
library IEEE;
use IEEE.std_logic_1164.all;
entity state_machine is
port(reset, clk, x : in std_logic;
     z : out std_logic);
end state_machine;

architecture behavioral of state_machine is
type statetype is (state0, state1);
signal state, next_state : statetype := state0;
begin
comb_process: process (state, x)
begin
--- process description here
end process comb_process;

clk_process: process
begin
-- process description here
end process clk_process;
end behavioral;
```

Example: Output and Next State Functions

```
comb_process: process (state, x)
begin
 case state is -- depending upon the current state
when state0 => -- set output signals and next state
      if x = '0' then
      next_state <= state1;
      z <= '1';
   else next_state <= state0;
      z <= '0';
   end if;
when state1 =>
if x = '1' then
      next_state <= state0;
      z <= '0';
   else next_state <= state1;
      z <= '1';
   end if;
end case;
end process comb_process;
```

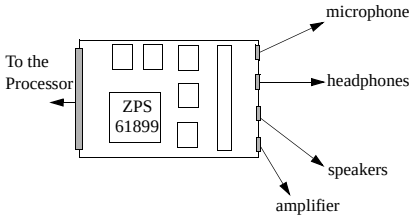
- Combination of the next state and output functions

Example: Clock Process

```
clk_process: process
begin
wait until (clk'event and clk = '1'); -- wait until the rising edge
   if reset = '1' then -- check for reset and initialize state
   state <= statetype'left;
   else state <= next_state;
   end if;
end process clk_process;
end behavioral;
```

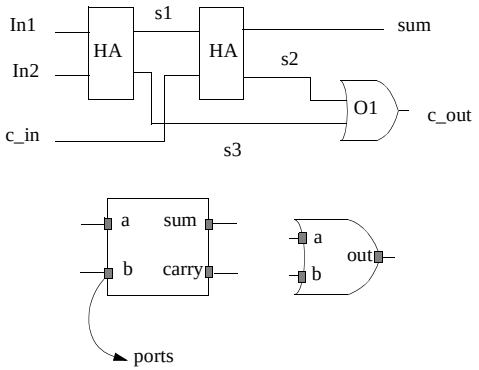
- Use of asynchronous reset to initialize into a known state

Modeling Structure



- Structural models describe a digital system as an interconnection of components
- Descriptions of the components must be available as structural or behavioral models

Modeling Structure



- Define the components used in the design
- Describe the interconnection of these components

Modeling Structure

```
architecture structural of full_adder is
component half_adder                    -- the declaration
port (a, b: in std_logic;                -- of components you will use
      sum, carry: out std_logic);
end component;

component or_2
 port (a, b : in std_logic;
       c : out std_logic);
end component;

signal s1, s2, s3 : std_logic;
begin
H1: half_adder port map (a => In1, b => In2, sum=>s1, carry=>s3);
H2:half_adder port map (a => s1, b => c_in, sum =>sum,
                        carry => s2);
O1: or_2 port map (a => s2, b => s3, c => c_out);
end structural;
```

unique name of the components
component type
interconnection of the component ports
component instantiation statement

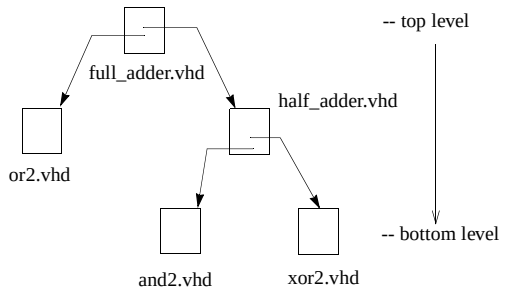
- Entity/architecture for half_adder and or_2 must exist

Hierarchy and Abstraction

```
architecture structural of half_adder is
component xor2
 port (a, b  : in std_logic;
       c : out std_logic);
end component;
component and2
port (a, b : in std_logic;
        c  : out std_logic);
end component;
begin
EX1: xor2 port map (a => a, b => b, c => sum);
AND1: and2 port map (a=> a, b=> b, c=> carry);
end structural;
```

- Structural descriptions can be nested
- The half adder may itself be a structural model

Hierarchy and Abstraction



- Nested structural descriptions to produce hierarchical models
- The hierarchy is flattened prior to simulation
- Behavioral models of components at the bottom level must exist

Generics

```
library IEEE;
use IEEE.std_logic_1164.all;

entity xor2 is
generic (gate_delay : Time:= 2 ns);
port(In1, In2 : in std_logic;
     z : out std_logic);
end xor2;

architecture behavioral of xor2 is
begin
z <= (In1 xor In2) after gate_delay;
end behavioral;
```

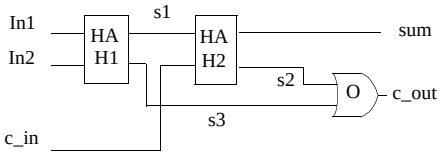
- Enables the construction of parameterized models

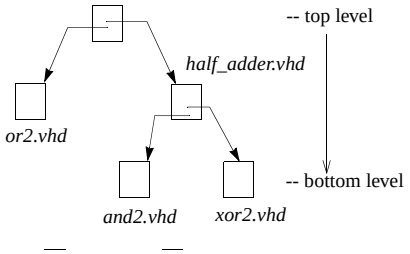
Generics in Hierarchical Models

```
architecture generic_delay of half_adder is
component xor2
generic (gate_delay: Time);
port (a, b  : in std_logic;
      c : out std_logic);
end component;

component and2
generic (gate_delay: Time);
port (a, b : in std_logic;
      c  : out std_logic);
end component;

begin
EX1: xor2 generic map (gate_delay => 6 ns)
           port map(a => a, b => b, c => sum);
A1: and2 generic map (gate_delay => 3 ns)
          port map(a=> a, b=> b, c=> carry);
end generic_delay;
```

- Parameter values are passed through the hierarchy

Example: Full Adder





Example: Full Adder

```
...
H1: half_adder generic map (gate_delay => 6 ns)
        port map(a => In1, b => In2, sum => s1, carry=>s2);
H2: half_adder ...
```

from full_adder.vhd

6 ns

6 ns

half_adder.vhd

```
...
EX1: xor2 generic map
(gate_delay⇒gate_delay)
...
A1: and2 generic map
(gate_delay⇒gate_delay)
```

half_adder.vhd

```
...
EX1: xor2 generic map
(gate_delay⇒gate_delay)
...
A1: and2 generic map
(gate_delay⇒gate_delay)
```

6 ns

6 ns

6 ns

6 ns

6 ns

xor2.vhd

```
entity and2 is
generic (gate_delay : Time:= 2 ns);
...
```

and2.vhd

```
entity xor2 is
generic (gate_delay :Time := 2 ns);
...
```

Precedence of Generic Declarations

```
architecture generic_delay2 of half_adder is
component xor2
generic (gate_delay: Time);
 port(a,b : in std_logic;
        c : out std_logic);
end component;

component and2
generic (gate_delay: Time:= 6 ns);
port (a, b : in std_logic;
   c : out std_logic);
end component;

begin
EX1: xor2 generic map (gate_delay => gate_delay)
        port map(a => a, b => b, c => sum);
A1: and2 generic map (gate_delay => 4 ns)
        port map(a=> a, b=> b, c=> carry);
end generic_delay2;
```

takes precedence

- Generic map takes precedence over the component declaration

Example: N-Input Gate

```
entity generic_or is
generic (n: positive:=2);
port (in1 : in std_logic_vector((n-1) downto 0);
      z : out std_logic);
end generic_or;

architecture behavioral of generic_or is
begin
process (in1)
variable sum : std_logic:= '0';
begin
sum := '0';  -- on an input signal transition sum must be reset t
for  i  in 0 to (n-1) loop
sum := sum or in1(i);
end loop;
z <= sum;
end process;
end behavioral;
```

- Map the generics to create different size OR gates

Example: Using the Generic N-Input OR Gate

```
architecture structural of full_adder is
component  generic_or
generic (n : positive);
port (in1 : in std_logic_vector((n-1) downto 0);
      z : out std_logic);
end component;
...
... -- remainder of the declarative region from earlier example
...
 begin
H1: half_adder port map (a => In1, b => In2, sum=>s1, carry=>s3);
H2:half_adder port map (a => s1, b => c_in, sum =>sum,
                                carry => s2);
O1: generic_or generic map (n => 2)
           port map (a => s2, b => s3, c => c_out);

end structural;
```

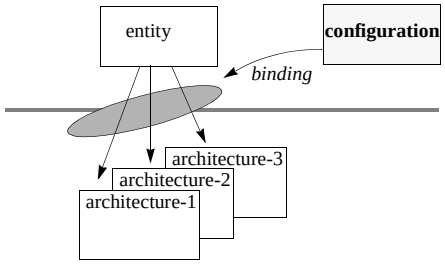
- Full adder model can be modified to use the generic OR gate model via the **generic map ()** construct

Example: N-bit Register

```
entity generic_reg is
generic (n : positive:=2);
port ( clk, reset, enable : in std_logic;
       d : in std_logic_vector (n-1 downto 0);
       q : out std_logic_vector (n-1 downto 0));
end generic_reg;

architecture behavioral of generic_reg is
begin
reg_process: process (clk, reset)
begin
 if reset = '1' then
    q <= (others => '0');
 elsif (clk'event and clk = '1') then
    if enable = '1' then q <= d;
   end if;
  end if;
end process reg_process;
end behavioral;
```

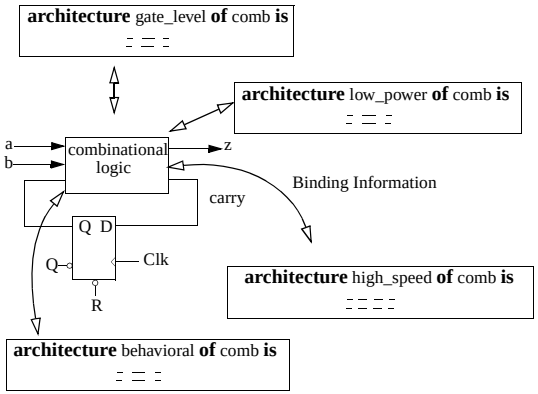
- This model is used in the same manner as the generic OR gate

Configurations



- A design entity can have multiple alternative architectures
- A configuration specifies the architecture that is to be used to simulate a design entity

Component Binding



- Concerned with configuring the architecture and not the entity
- Enhances sharing of designs: simply change the configuration

Default Binding Rules

```
architecture structural of  serial_adder  is
component comb
 port (a, b, c_in  : in std_logic;
          z, carry : out std_logic);
end component;

 component dff
port (clk, reset, d :in std_logic;
       q, qbar :out std_logic);
end component;

signal s1, s2 : std_logic;
begin
C1: comb  port map (a => a, b => b, c_in => s1, z =>z, carry => s2);
D1: dff port map(clk => clk, reset =>reset, d=> s2, q=>s1, qbar =>open);
end structural;
```

- Search for entity with the same component name
- Bind the last compiled architecture for that entity
- How do we get more control over binding?

Configuration Specification

```
architecture structural of full_adder is
--
--declare components here
signal s1, s2, s3: std_logic;
--
-- configuration specification
for H1: half_adder use entity WORK.half_adder (behavioral);
for H2: half_adder use entity WORK.half_adder (structural);
for O1: or_2 use entity POWER.lpo2 (behavioral)
generic map(gate_delay => gate_delay)
port map (I1 => a, I2 => b, Z=>c);

begin       -- component instantiation statements
H1: half_adder port map (a =>In1, b => In2,
                         sum => s1, carry=> s2);
H2: half_adder port map (a => s1, b => c_in,
                         sum => sum, carry => s2);
O1: or_2 port map(a => s2, b => s3, c => c_out);
end structural;
```

library name, *entity name*, *architecture name*

- We can specify any binding where ports and arguments match

Configuration Declaration

```
configuration Config_A of full_adder is  -- name the configuration
                                         -- for the entity
  for structural   -- name of the architecture being configured
  for H1: half_adder use entity WORK.half_adder (behavioral);
  end for;
  --
  for H2: half_adder use entity WORK.half_adder (structural);
  end for;
  --
  for O1: or_2 use entity POWER.lpo2 (behavioral)
  generic map(gate_delay => gate_delay)
  port map (I1 => a, I2 => b, Z=>c);
  end for;
  --
  end for;
  end Config_A;
```

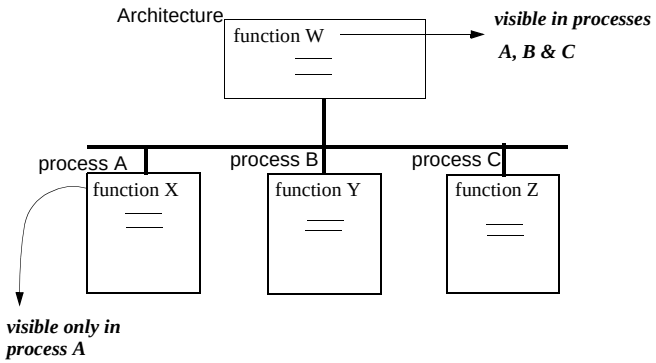
- Separate design unit
- Can be written to span a design hierarchy

Essentials of Functions

```
function rising_edge (signal clock: std_logic) return boolean is
--
--declarative region: declare variables local to the function
--
begin
-- body
--
return (expression)
end rising_edge;
```

- Formal parameters and mode
- Functions cannot modify parameters
- Types of formals and actuals must match except for formals which are constants (default)
- Wait statements are not permitted in a function!

Placement of Functions



- When used by multiple design entities place functions in packages

Function: Example

```
architecture behavioral of dff is
function rising_edge  (signal clock : std_logic) return boolean is
variable edge : boolean:= FALSE;
begin
edge := (clock = '1' and clock'event);
return (edge);
end rising_edge;

begin
output: process
begin
wait until (rising_edge (Clk) );

    Q <= D after 5 ns;
    Qbar <= not D after 5 ns;

end process  output;
end behavioral;
```

Architecture Declarative Region

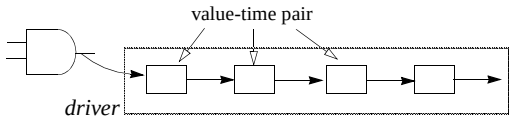
Function: Example

```
function to_bitvector (svalue : std_logic_vector) return bit_vector is
variable outvalue : bit_vector (svalue'length-1 downto 0);
begin
for i in svalue'range loop -- scan all elements of the array
case svalue  (i) is
when '0' => outvalue  (i) := '0';
when '1' => outvalue  (i) := '1';
when others => outvalue  (i) := '0';
end case;
end loop;
return outvalue;
end to_bitvector
```

- Type conversion functions
  - interoperability
- Use of attributes for flexible function definitions
- Browse the vendor supplied packages for many examples

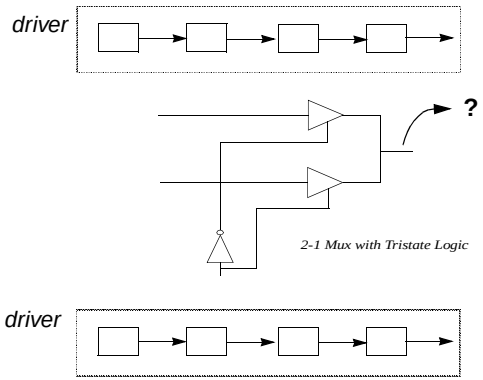
Implementation of Signals

- The structure of a signal assignment statement
  - *signal* <= (*value expression* **after** *time expression*)
  - RHS is referred to as a *waveform element*
- Every signal has associated with it a *driver*



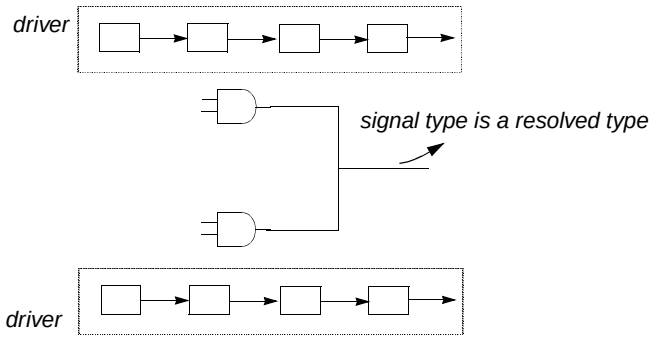
  - holds the current and future values of the signal - a *projected waveform*
  - signal assignment statements modify the driver of a signal
  - value of a signal is the value at the head of the driver
  - note the hardware analogy

Shared Signals



- How do we model the state of a wire?
- Rules for determining the signal value is captured in the resolution function

Resolved Signals



- Resolution function is invoked whenever an event occurs on this signal
- Resolution must be an associative operation

Resolved Types: std_logic

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

supports only single drivers

```
function resolved (s : std_ulogic_vector) return
std_ulogic;

subtype std_logic is resolved std_ulogic;
```

supports multiple drivers

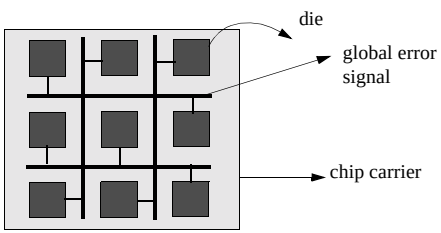
Resolution Function: std_logic & resolved()

- resolving values for std_logic types

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

- Pairwise resolution of signal values from multiple drivers
- Resolution operation must be associative

Example



- Multiple components driving a shared error signal
- Signal value is the logical OR of the driver values

A Complete Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mcm is
end mcm;

architecture behavioral of mcm is
function wire_or (sbus :std_ulogic_vector)
return std_ulogic;
begin
for i in sbus'range loop
 if sbus(i) = '1' then
return '1';
end if;
end loop;
return '0';
end wire_or;

subtype wire_or_logic is wire_or
std_ulogic;
signal error_bus : wire_or_logic;
begin
Chip1: process
begin
--..
error_bus <= '1' after 2 ns;
--..
end process Chip1;
Chip2: process
begin
--..
error_bus <= '0' after 2 ns;
--..
end process Chip2;
end behavioral;
```

- Use of unconstrained arrays and associative operations

Summary: Essentials of Functions

- Placement of functions
- Formal parameters
- Check the source listings of packages for examples of many different functions

Essentials of Procedures

```
procedure read_v1d  (variable f: in text; v :out std_logic_vector)
--declarative region: declare variables local to the procedure
--
begin
-- body
--
 end read_v1d;
```

- parameters may be of mode **in** (read only) and **out** (write only)
- default class of input parameters is constant
- default class of output parameters is variable
- variables declared within procedure are initialized on each call

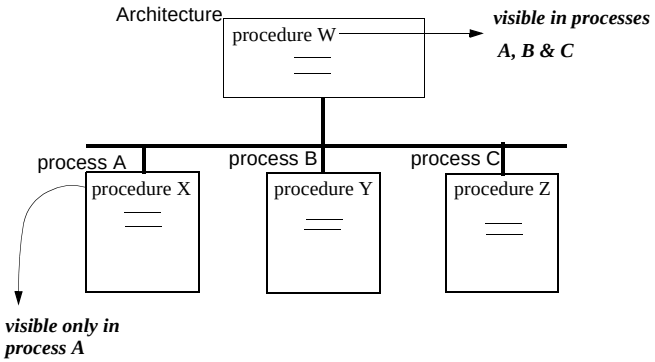
Procedures: Placement

```
architecture behavioral of cpu is
--
-- decalarative region
-- procedures can be placed in their entirety here
--
begin
   process_a: process
   -- declarative region of a process
   -- procedures can be placed here
   begin
   --
   -- process body
   --
   end process_a;
   process_b: process
   --declarative regions
   begin
   -- process body
   end process_b;
end behavioral;
```

visible to all processes

visible only within process_a

Placement of Procedures



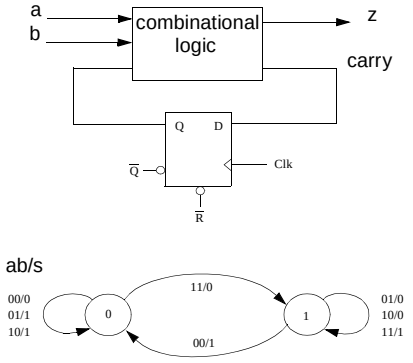
- placement of procedures determines visibility in its usage

Procedures and Signals

```
procedure mread (address : in std_logic_vector (2 downto 0);
signal R : out std_logic;
signal S : in std_logic;
signal ADDR : out std_logic_vector (2 downto 0);
signal data : out std_logic_vector (31 downto 0)) is
begin
ADDR <= address;
R <= '1';
wait until S = '1';
data <= DO;
R <= '0';
end mread;
```

- procedures can make assignments to signals passed as input parameters
- procedures may not have a wait statement if the encompassing process has a sensitivity list
- procedures may modify signals not in the parameter list, e.g., ports

Concurrent vs. Sequential Procedure Calls



- Example: bit serial adder

Concurrent Procedure Calls

```
architecture structural of serial_adder is
component comb
 port (a, b, c_in  : in std_logic;
          z, carry : out std_logic);
end component;

procedure dff (signal d, clk, reset : in std_logic;
               signal q, qbar : out std_logic) is
begin
 if (reset = '0') then
     q  <= '0' after 5 ns;
     qbar <= '1' after 5 ns;
   elsif (clk'event and clk = '1') then
   q  <= d after 5 ns;
   qbar <= (not D) after 5 ns;
 end if;
end dff;

signal s1, s2 : std_logic;
```

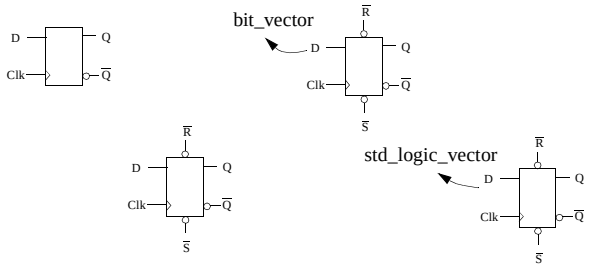
```
begin
C1: comb  port map (a => a, b => b,
c_in => s1, z =>z, carry => s2);
--
-- concurrent procedure call
--
dff(clk => clk, reset =>reset, d=> s2,
q=>s1, qbar =>open);
end structural;
```


Equivalent Sequential Procedure Call

```
architecture structural of serial_adder is
component comb
 port (a, b, c_in  : in std_logic;
          z, carry : out std_logic);
end component;

procedure dff (signal d, clk, reset : in std_logic;
               signal q, qbar : out std_logic) is
begin
 if (reset = '0') then
     q  <= '0' after 5 ns;
     qbar <= '1' after 5 ns;
   elsif (clk'event and clk = '1') then
   q  <= d after 5 ns;
   qbar <= (not D) after 5 ns;
 end if;
end dff;

signal s1, s2 : std_logic;
```

```
begin
C1: comb  port map (a => a, b => b,
c_in => s1, z =>z, carry => s2);
--
-- sequential  procedure call
--
process
begin
dff(clk => clk, reset =>reset, d=> s2,
q=>s1, qbar =>open);
wait on clk, reset,s2;
end structural;
```


Subprogram Overloading



- hardware components differ in number of inputs and the type of input signals
- model each component by a distinct procedure
  - procedure naming becomes tedious

Subprogram Overloading

- consider the following procedures for the previous components
  - dff_bit (clk, d, q, qbar)
  - asynch_dff_bit (clk, d,q,qbar,reset,clear)
  - dff_std (clk,d,q,qbar)
  - asynch_dff_std (clk, d,q,qbar,reset,clear)
- all of the previous components can use the same name --> subprogram overloading
- the proper procedure can be determined based on the arguments of the call

Essentials of Packages

- Package Declaration
  - declaration of the functions, procedures, and types that are available in the package
  - serves as a package interface
  - only declared contents are visible for external use
- note the behavior of the **use** clause
- Package body
  - implementation of the functions and procedures declared in the package header

Example: Package Header std_logic_1164

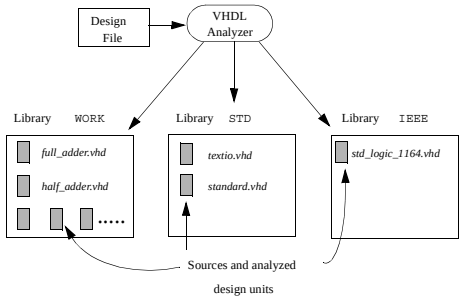
```
package std_logic_1164 is
 type std_ulogic is ('U', --Unitialized
                     'X', -- Forcing Unknown
                     '0', -- Forcing 0
                     '1', -- Forcing 1
                     'Z', -- High Impedance
                     'W', -- Weak Unknown
                     'L',  -- Weak 0
                     'H', -- Weak 1
                     '-' -- Don't care
                    );
type std_ulogic_vector is array (natural range <>) of std_ulogic;

function resolved (s : std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_ulogic;

type std_logic_vector is array (natural range <>) of std_logic;

function "and" (l, r : std_logic_vector) return std_logic_vector;
--..<rest of the package definition>
end std_logic_1164;
```


Example: Package Body

```
package body my_package is
--
-- type definitions, functions, and procedures
--
end my_package;
```

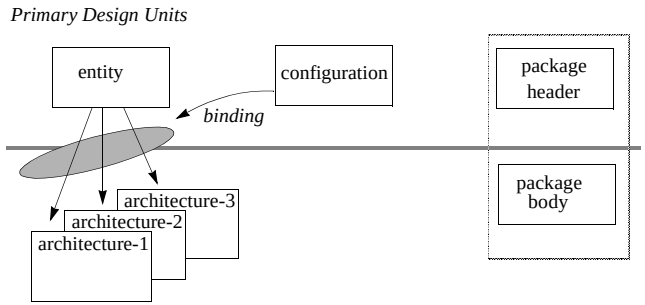
- Packages are typically compiled into libraries
- New types must have associated definitions for operations such as logical operations (e.g., and, or) and arithmetic operations (e.g., +, *)
- Examine the package std_logic_1164 stored in library IEEE

Essentials of Libraries



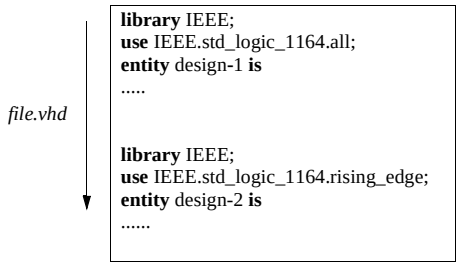
- Design units are analyzed (compiled) and placed in libraries
- Logical library names map to physical directories
- Libraries STD and WORK are implicitly declared

Design Units



- Distinguish the primary design units

Visibility Rules



- when multiple design units are in the same file visibility of libraries and packages must be established for each ***primary*** design unit (entity, package header, configuration) separately!
- The **use** clause may selectively establish visibility, e.g., only the function rising_edge() is visible within entity design-2

Further reading

- VHDL Starter's Guide, Sudhakar Yalamanchili
- Check the VHDL references link from the course web page
- Look at the on-line documentation available with ISE – there is a huge VHDL manual
- Read/copy the online manual placed in the VHDL manual link of the course web page

Homework

1. Download and install WebPack and ModelSim as described on the VHDL references course web page,

OR

Access the ISE Project Manager from a *vmware* enabled school Linux workstation;

AND

2. Familiarize yourself with the design environment by completing the Tutorial and testing the designs we have discussed in lectures.