Essential VHDL for ASICs

A brief introduction to design with VHDL for ASIC design.

Roger Traylor

9/7/01

Version 0.1

Revision Record

rev 0.1 : Initial rough entry of material. 9/7/01 RLT

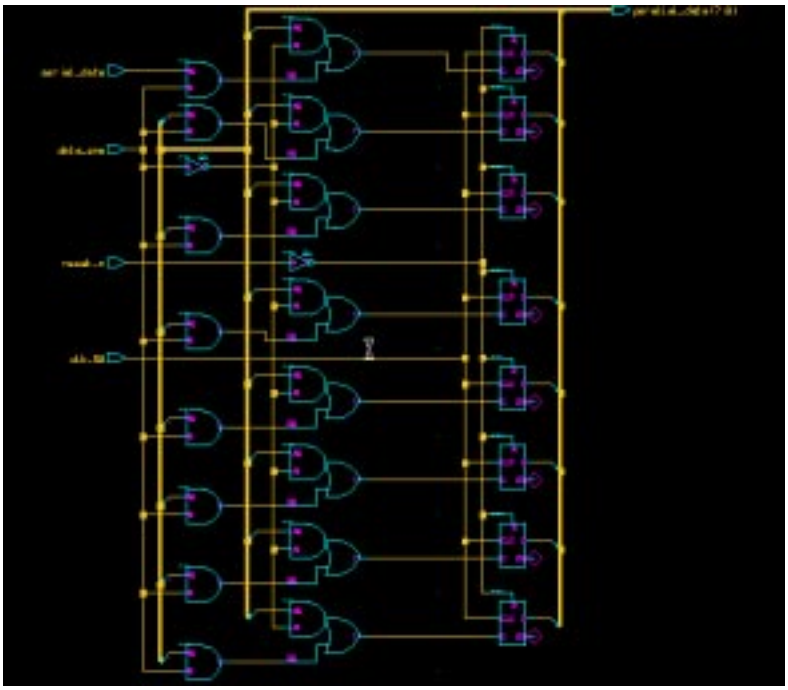
HDL Design

Traditionally, digital design was done with schematic entry.

In today's very competitive business environment, building cost-effective products in an quick fashion is best done with a top down methodology utilizing hardware description languages and synthesis.

```
shift_register:
 PROCESS (clk_50, reset_n, data_ena, serial_data, parallel_data)
   BEGIN
     IF (reset_n = '0') THEN
       parallel_data <= "00000000";
     ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
       IF (data_ena = '1') THEN
         parallel_data(7) <= serial_data;         --input gets input data
         FOR i IN 0 TO 6 LOOP
          parallel_data(i) <= parallel_data(i+1); --all other bits shift down
         END LOOP;
       ELSE
          parallel_data <= parallel_data;
       END IF;
     END IF;
   END PROCESS shift_register;
```

synthesis



HDLs - Motivation

Increased productivity

shorter development cycles, more features, but.....
still shorter time-to-market, 10-20K gates/day/engineer

Flexible modeling capabilities.

can represent designs of gates or systems
description can be very abstract or very structural
top-down, bottom-up, complexity hiding (abstraction)

Design reuse is enabled.

packages, libraries, support reusable, portable code

Design changes are fast and easily done

convert a 8-bit register to 64-bits.....
four key strokes, and its done!
exploration of alternative architectures can be done quickly

Use of various design methodologies.

top-down, bottom-up, complexity hiding (abstraction)

Technology and vendor independence.

same code can be targeted to CMOS, ECL, GaAs
same code for: TI, NEC, LSI, TMSC....no changes!

Enables use of logic synthesis which allows a investigation of the area and timing space.

ripple adder or CLA?, How many stages of look ahead?

HDLs can leverage software design environment tools.

source code control, make files

Using a standard language promotes clear communication of ideas and designs.

schematic standards?... what's that... a tower of Babel.

HDLs - What are they? How do we use them?

A Hardware Description Language (HDL) is a programming language used to model the intended operation of a piece of hardware.

An HDL can facilitate:

abstract behavioral modeling

-no structural or design aspect involved

hardware structure modeling

-a hardware structure is explicitly implied

In this class we will use an HDL to describe the structure of a hardware design.

When we use an HDL, we will do so at what is called the *Register Transfer Language level (RTL).* At this level we are implying certain hardware structures when we understand apriori.

When programming at the RTL level, we are not describing an algorithm which some hardware will execute, we are describing a hardware structure.

Without knowing beforehand what the structure is we want to build, use of an HDL will probably produce a steaming pile (think manure) of gates which may or may not function as desired.

You must know what you want to build before you describe it in an HDL.

Knowing an HDL does not relieve you of thoroughly understanding digital design.

HDL's- VHDL or Verilog

We will use VHDL as our HDL.

VHDL

more capable in modeling abstract behavior
more difficult to learn
strongly typed
85% of FPGA designs done in VHDL

Verilog

easier and simpler to learn
weakly typed
85% of ASIC designs done with Verilog (1993)

The choice of which to use is not based solely on technical capability, but on:

personal preferences
EDA tool availability
commercial business and marketing issues

We use VHDL because

strong typing keeps students from getting into trouble
if you know VHDL, Verilog can be picked up in few weeks
If you know Verilog, learning VHDL can take several months

The Bottom line...Either language is viable.

VHDL - Origins

Roots of VHDL are in the Very High Speed Integrated Circuit (VHSIC) Program launched in 1980 by the US Department of Defense (DOD).

The VHSIC program was an initiative by the DOD to extend integration levels and performance capabilities for military integrated circuits to meet or exceed those available in commercial ICs.

The project was successful in that very large, high-speed circuits were able to be fabricated successfully. However, it became clear that there was a need for a standard programming language to describe and document the function and structure of these very complex digital circuits.

Therefore, under the VHSIC program, the DOD launched another program to create a standard hardware description language. The result was the VHSIC hardware description language or VHDL.

The rest is history...

In 1983, IBM, TI and Intermetrics were awarded the contract to develop VHDL.

In 1985, VHDL V7.2 released to government.

In 1987, VHDL became IEEE Standard 1076-1987.

In 1993, VHDL restandardized to clarify and enhance the language resulting in VHDL Standard 1076-1993.

In 1993, development began on the analog extension to VHDL, (VHDL-AMS).

Extends VHDL to non-digital devices and micro electromechanical components. This includes synthesis of analog circuits.

Some Facts of Life (For ASIC designers)

The majority of costs are determined by decisions made early in the design process.

"Hurry up and make all the mistakes. Get them out of the way!"

"Typical" ASIC project: concept to first silicon about 9 months.

95% of designs work as the specification states.

60% of designs fail when integrated into the system.

The design was not the right one, but it "works".

Technology is changing so fast, the only competitive advantage is to learn faster than your competitors.

To design more "stuff" faster, your level of abstraction in design must increase.

Using HDLs will help to make digital designers successful. (and employed!)

VHDL Modeling

A VHDL models consist of an *Entity Declaration* and a *Architecture Body.*

The entity defines the interface, the architecture defines the function.

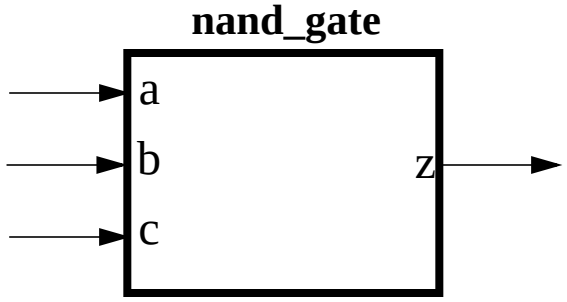
The entity declaration names the entity and defines the interface to its environment.

Entity Declaration Format:

```
ENTITY entity_name IS
    [GENERIC (generic_list);]
    [PORT (port_list);]
END ENTITY [entity_name];
```

There is a direct correspondence between a ENTITY and a block diagram symbol. For example:

```
ENTITY nand_gate IS
PORT(
        a : in   std_logic;
        b : in   std_logic;
        c : in   std_logic;
        z : out  std_logic);
END ENTITY nand_gate;
```



Port Statement

The entities *port* statement identifies the ports used by the entity to communicate with its environment

Port Statement Format:

```
PORT(
    name_list  : mode type;
    name_list  : mode type;
    name_list  : mode type;
    name_list  : mode type);
```

This is legal but poor form:

```
ENTITY nand_gate IS
  PORT(a,d,e,f : in  std_logic;
    b,j,q,l,y,v : in  std_logic;
    w,k  : in  std_logic;
    z : out: std_logic);
END nand_gate;
```

This is much less error prone:

Use one line per signal. This allows adequate comments.
Capitalize reserved names.

```
ENTITY nand_gate IS
  PORT(
    a : IN  STD_LOGIC;  --a input
    b : IN  STD_LOGIC;  --b input
    c : IN  STD_LOGIC;  --c input
    z : OUT STD_LOGIC); --nand output
END ENTITY nand_gate;
```

Port Mode:

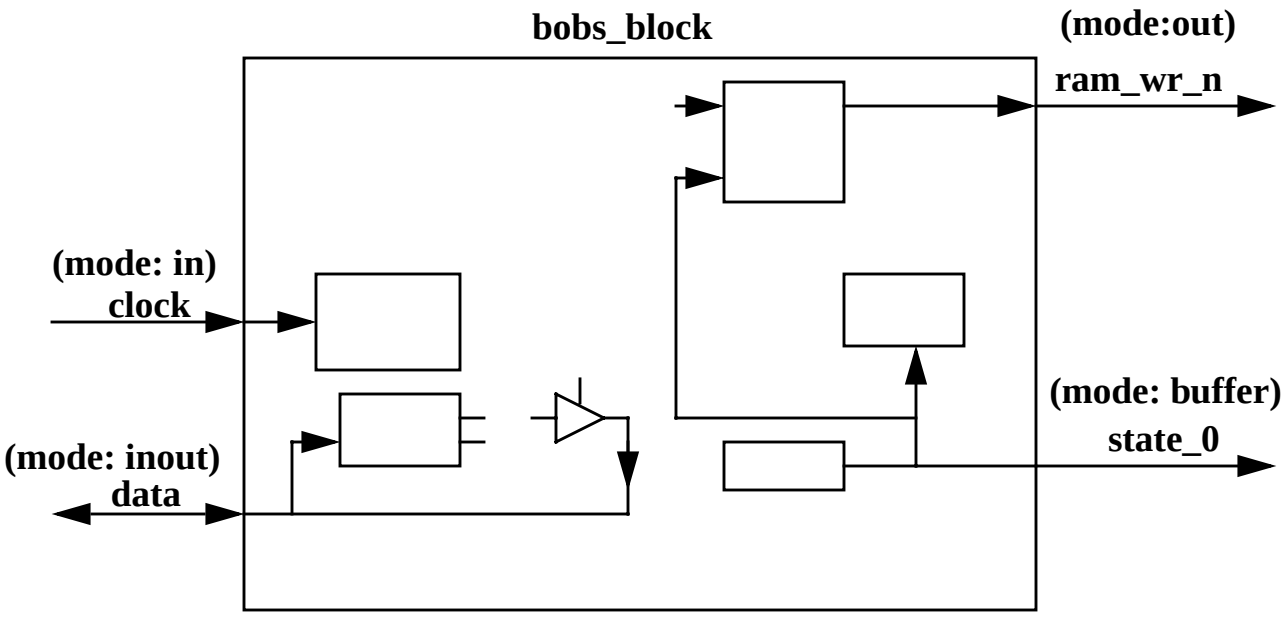
Identifies the direction of data flow through the port.

The PORT statement is optional. At the top level, none is needed.

All ports must have an identified mode.

Allowable Modes:

- **IN** **Flow is into the entity**
- **OUT** **Flow is out of the entity**
- **INOUT** **Flow may be either in or out**
- **BUFFER** **An OUTPUT that can be read from**



Architecture Body

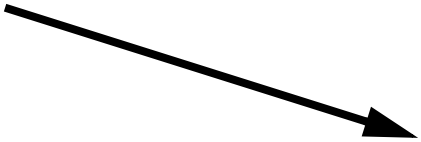
The architecture body describes the operation of the component.

Format:

```
ARCHITECTURE body_name OF entity_name IS
   --this is the ->declarative area<-
   --declare signals, variables, components,
   --subprograms
BEGIN
   --this is the ->statement area<-
   --in here go statements that describe
   --organization or functional operation of
   --the component
   --this is the “execution part” of the model
END [body_name]
```

The entity_name in the architecture statement must be the same as the entity declaration that describes the interface to the outside world.

ENTITY *entity_name* IS



ARCHITECTURE body_name OF *entity_name* IS

The “body_name” is a user-defined name that should uniquely describe the particular architecture model.

```
ARCHITECTURE beh OF nand_gate IS

ARCHITECTURE struct OF nand_gate IS
```

Note: multiple architectures are allowed.

Commenting Code

A double hyphen (--) indicates everything from that point on in that line is to be treated as a comment.

```
ARCHITECTURE example OF xor_gate IS
   --The following is a silly example of how
   --to write comments in VHDL.
BEGIN
   --comment from the beginning of a line
   a <= b XOR c; --or...comment from here on
   --
   --each line must have its own
   --comment marker unlike “C”
   --
END [body_name]
   --
   --
   --this is the end and there ain’t no more!
```

Comments can be put anywhere except in the middle of a line of code.

Important Note: The tool used to prepare this document sometimes changes the first of a pair of quotes. In VHDL, only the quote marks that lean to the right or don’t lean at all are used. For example, ‘1’ should only have single quotes that lean to the right like the second one does. The quote mark we use is on the same key as the double quote.

Entity and Architecture for a NAND gate Model

```
--
--the following is a behavioral description of
--a three input NAND gate.
--
ENTITY nand3 IS
PORT(
      a  : IN   std_logic;
      b  : IN   std_logic;
      c  : IN   std_logic;
      z  : OUT  std_logic);
END ENTITY nand3;

ARCHITECTURE beh OF nand3 IS
 BEGIN
    z <= '1'  WHEN a='0' AND b='0' ELSE
         '1'  WHEN a='0' AND b='1' ELSE
         '1'  WHEN a='1' AND b='0' ELSE
         '0'  WHEN a='1' AND b='1' ELSE
         'X';
END ARCHITECTURE beh;
```

You can create VHDL source code in any directory.

VHDL source code file may be anything.....but,
Use the name of the design entity with the extension "*.vhd*"

The above example would be in the file: nand3.vhd

Question: Why the 'X' in the above code?

Signal Assignment

The assignment operator (<=) is used to assign a waveform value to a *signal*.

Format:

```
target_object <= waveform;
```

Examples:

```
my_signal  <= '0'; --ties my_signal to "ground"
his_signal <= my_signal; --connects two wires

--vector signal assignment

data_bus <= "0010"; -- note double quote
bigger_bus  <= X"a5";  -- hexadecimal numbers
```

Declaring Objects

Declaration Format:

```
OBJECT_CLASS  identifier:  TYPE [:= init_val];
```

Examples:

```
CONSTANT  delay   :  TIME:= 10ns;
CONSTANT  size    :  REAL:=5.25;
VARIABLE  sum     :  REAL;
VARIABLE  voltage :  INTEGER:=0;
SIGNAL    clock   :  BIT;
SIGNAL    spam    :  std_logic:='X';
```

Objects in the port statement are classified as signals by default.

Objects may be initialized at declaration time.

If an object is not initialized, it assumes the left-most or minimum value for the type

Naming Objects

Valid characters:

- **alpha characters (a-z)**
- **numeric characters (0-9)**
- **underscore (_)**

Names must consist of any number of alpha, numeric, or underline characters.

Underscore must be proceeded and followed by alpha or numeric characters.

The underscore can be used to separate adjacent digits in bit strings:

```
CONSTANT big_0 : STD_LOGIC_VECTOR(15 DOWNTO 0) :=
B"0000_0000_0000_0000";
```

Names are not case sensitive. (be consistent!, use lowercase!)

Coding hints:

Use good names that are meaningful to others. If your code is good, somebody else will want to read it.

Name signals by their function. For example, if you have a multiplexor select line that selects addresses, give it a name like "`address_select`" instead of "`sel_32a`".

Name blocks by their function. If a block generates control signals for a DRAM controller, call the block "`dram_ctl`" not something obscure like "`block_d`".

A Simple Example to Recap

```
-----------------------------------------
--and-or-invert gate model
--Jane Engineer
--3/13/01
--version 0.5
-----------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY aoi4 IS
PORT(
     a  : IN    std_logic;
     b  : IN    std_logic;
     c  : IN    std_logic;
     d  : IN    std_logic;
     z  : OUT   std_logic);
END ENTITY aoi4;

ARCHITECTURE data_flow OF aoi4 IS
  SIGNAL  temp1, temp2  :  std_logic;
  BEGIN
    temp1 <= a AND b;
    temp2 <= c AND d;
    z     <= temp1 NOR temp2;
END ARCHITECTURE data_flow;
```

Simulating VHDL code

The Simulator

The simulator we will be using is the Model Technologies' *ModelSim*. It will be referred to as *vsim*. Vsim is a full featured VHDL and/or Verilog simulator with best-in-class VHDL simulation. It is also very easy to learn and use.

VHDL Libraries

Before a VHDL design can be simulated, it must be compiled into a machine executable form. The compiled image is placed into a library where the simulator expects to find the executable image. Therefore we must first create a special directory called "work".

The Library *work*

The library named work has special attributes within vsim; it is predefined in the compiler. It is also the library name used by the compiler as the default destination of compiled design units. In other words the work library is the working library.

Creating *work*

At the desired location in your directory tree, type:

```
vlib work
```

You will see a directory work created. You cannot create work with the UNIX *mkdir* command.

Simulating VHDL Code (cont.)

Compile the code

Suppose our example code is in a file called *aoi4.vhd*. At the level at which you can see the directory *work* with an *ls* command, simply type:

```
vcom -93 aoi4.vhd
```

Then you will see:

```
brilthor.ECE.ORST.EDU:vcom -93 src/aoi4.vhd
Model Technology ModelSim SE/EE vcom 5.4c Compiler 2000.08 Jul 29 2000
-- Loading package standard
-- Loading package std_logic_1164
-- Compiling entity aoi4
-- Compiling architecture data_flow of aoi4
brilthor.ECE.ORST.EDU:
```

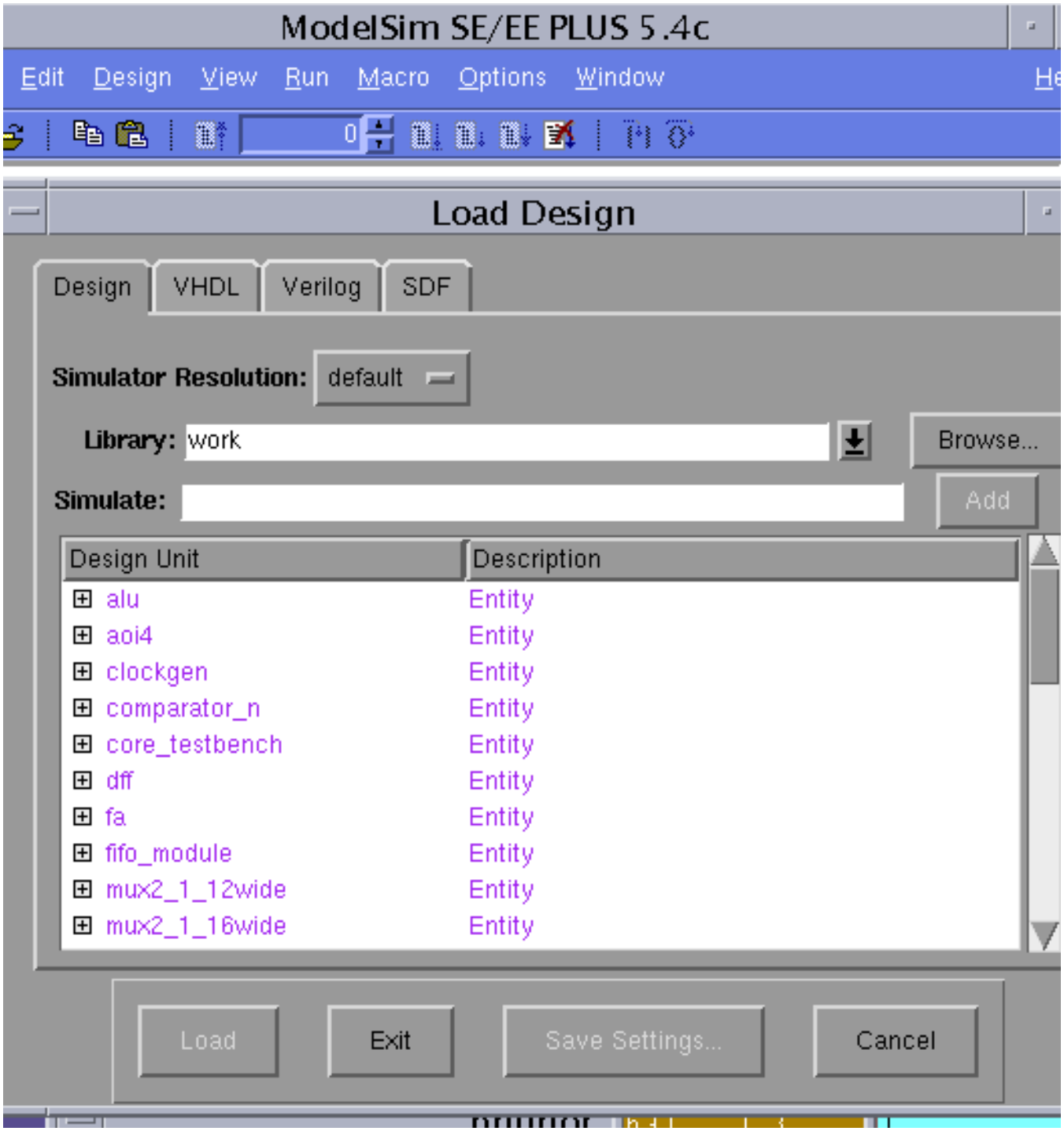
If you look in the work directory, you will see a subdirectory in work with the entity name *aoi4*. In there are the files necessary to simulate the design.

With a clean compilation, we are ready to simulate.

Simulating VHDL Code (cont.)

Simulate the design

Invoke the simulator by typing *vsim* at the UNIX prompt. You will see the *Load Design* window open over the main *vsim* window.



Simulating VHDL code (cont.)

The *Design Unit* is the name of the entity you want to load into the simulator. In this example, there are many other entities in the work directory, each corresponding to a different entity/architecture pair.

To load the design, click on *aoi4* and then *Load*. Note that *aoi4* this is not the file name, but the entity name.

The design will then load. To run a simulation, first type view * in the ModelSim window. This will open all the windows available for the simulation. You will probably want to close all but the wave, signals, source and structure windows.

To observe the signals, in the *Signals* window select:
View > Wave > Signals in Region
All the signals in the design are then visible in the Wave window.

To provide stimulation to our model we can just force the input signals and run for a short time. To do this, in the ModelSim window we can type:

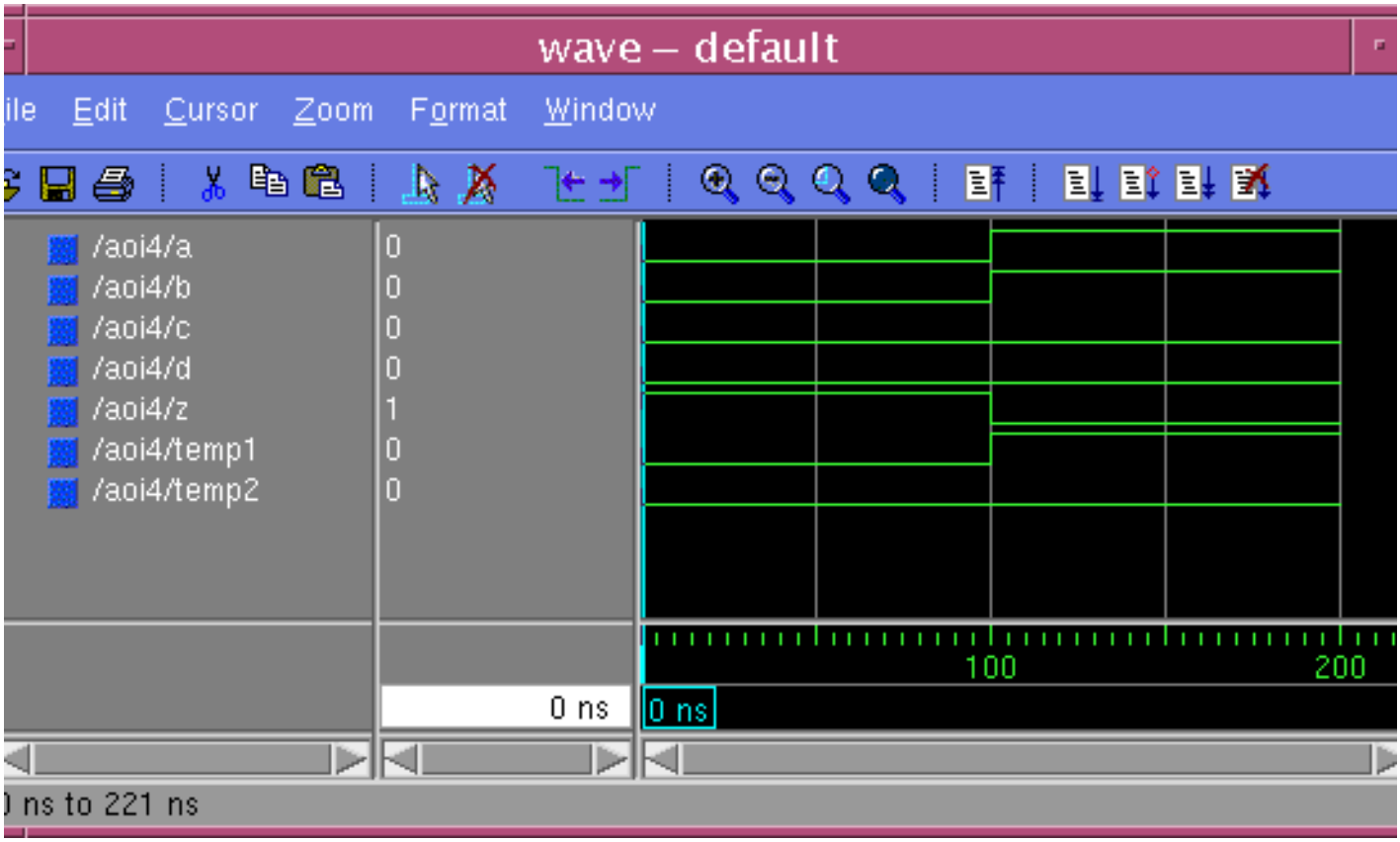
```
force a 0
force b 0
force c 0
force d 0
run 100
force a 1
force b 1
run 100
```

According to our model we should see the z output assert to a zero when either a and b or c and d both become true. We can see the correct behavior in the wave window.

HW: make 2-3 complex algebraic equations and implement them in VHDL. Students simulate and check them. Synthesize them and recheck with VHDL simulator. Print out gate level schematics.

Simulating VHDL Code

The output from the wave window looks like this:



We will make heavy usage of the *vsim* simulator. You are encouraged to explore and discover the different options and way to operate the simulator.

For example, the force commands may be applied from a “do file”. This is a text file containing all the force and run commands. Try to use a force file to exhaustively test the aoi4 design.

The documentation for the Model Technology tools may be found at:
http://www.ece.orst.edu/comp/doc/tools/mti/mti_documentation.html

What about this Synthesis thing?

Simulation is great, but one of the foremost advantages of an HDL is its ability to create gate level designs thorough a different flavor compilation....synthesis.

We can take the previous example, and synthesize the VHDL code into a gate level design and represent it at a new structural VHDL netlist or a schematic.

We will not go into the details of how synthesis is done but lets see what happens anyway.

We usually synthesize VHDL designs using a script to direct the synthesis tool. Using a GUI to do this would be very time consuming.

Helpful Hint: Running a CAD tool is not like running a web browser. Learn to use scripts and command line interfaces.

What about this “Synthesis” thing? (cont.)

Here is a simple synthesis script for *elsyn* (a synthesis tool) that synthesizes our behavioral design for the aoi4 gate.

```
#simple synthesis script
set vhdl_write_component_package FALSE
set vhdl_write_use_packages {library ieee,adk; use
ieee.std_logic_1164.all; use adk.all;}
set edifout_power_ground_style_is_net TRUE
set sdf_write_flat_netlist TRUE
set force_user_load_values TRUE
set max_fanout_load 10


load_library ami05_typ


analyze src/aoi4.vhd     -format vhdl -work work
elaborate aoi4           -architecture data_flow -work work
optimize -ta ami05_typ -effort standard -macro -area


write ./edif/aoi4.edf -format edif
write ./vhdlout/aoi4.vhd -format vhdl


#to make a schematic do this in the edif directory
#edif2eddm aoi4.edf data_flow
```

What’s important to understand here?

`load_library ami05_typ`
The synthesis tool needs a known library of logic cells (gates) to build the synthesized design from.

`analyze src/aoi4.vhd     -format vhdl -work work`
Analyze (compile) the VHDL code and do initial processing.

`elaborate aoi4           -architecture data_flow -work work`
Create a generic gate description of the design.

`optimize -ta ami05_typ -effort standard -macro -area`
Map the generic gates to the “best” ones in the library ami05.

```
write ./edif/aoi4.edf -format edif
write ./vhdlout/aoi4.vhd -format vhdl
```
Write out the results in EDIF and VHDL formats.

How is the synthesis invoked?

The script is saved in a file called script_simple.

A work directory (if not already created) is created to put the compiled images by typing:

```
vlib work
```

Create the edif and vhdlout directories where the edif and VHDL netlist will be put.

```
mkdir edif
mldir vhdlout
```

Then, from the command line type:

```
elsyn
```

Eventually you get the prompt:

```
LEONARDO{1}:
```

Then type:

```
source script_simple
```

The tool *elsyn* reads the script file and executes the commands in the script.

What does the output look like?

The synthesis tool puts a synthesized version of the design in two directories, the vhdlout and edif directories. In the vhdlout directory:

```
--
-- Definition of  aoi4
--
--      Wed Jul 18 12:31:05 2001
--      Leonardo Spectrum Level 3, v20001a2.72
--

library ieee,adk; use ieee.std_logic_1164.all; use adk.all;

entity aoi4 is
   port (
      a : IN std_logic ;
      b : IN std_logic ;
      c : IN std_logic ;
      d : IN std_logic ;
      z : OUT std_logic) ;
end aoi4 ;

architecture data_flow of aoi4 is
   component aoi22
      port (
         Y : OUT std_logic ;
         A0 : IN std_logic ;
         A1 : IN std_logic ;
         B0 : IN std_logic ;
         B1 : IN std_logic) ;
   end component ;
begin
   ix13 : aoi22 port map ( Y=>z, A0=>a, A1=>b, B0=>c, B1=>d);
end data_flow ;
```

Examine the gate level VHDL

We see that the synthesized aoi4 looks much like what we initially wrote. The entity is exactly the same.

The architecture description is *different*. The design aoi4 is now described in a different way.

Under the architecture declarative section, a gate (aoi22) from the library was declared:

```
component aoi22
  port (
    Y : OUT std_logic ;
    A0 : IN std_logic ;
    A1 : IN std_logic ;
    B0 : IN std_logic ;
    B1 : IN std_logic) ;
end component ;
```

In the statement area, we see this gate is connected to the ports of the entity with a component instantiation statement.

```
ix13 : aoi22 port map ( Y=>z, A0=>a, A1=>b, B0=>c, B1=>d);
```

We will study component instantiation in more detail later.

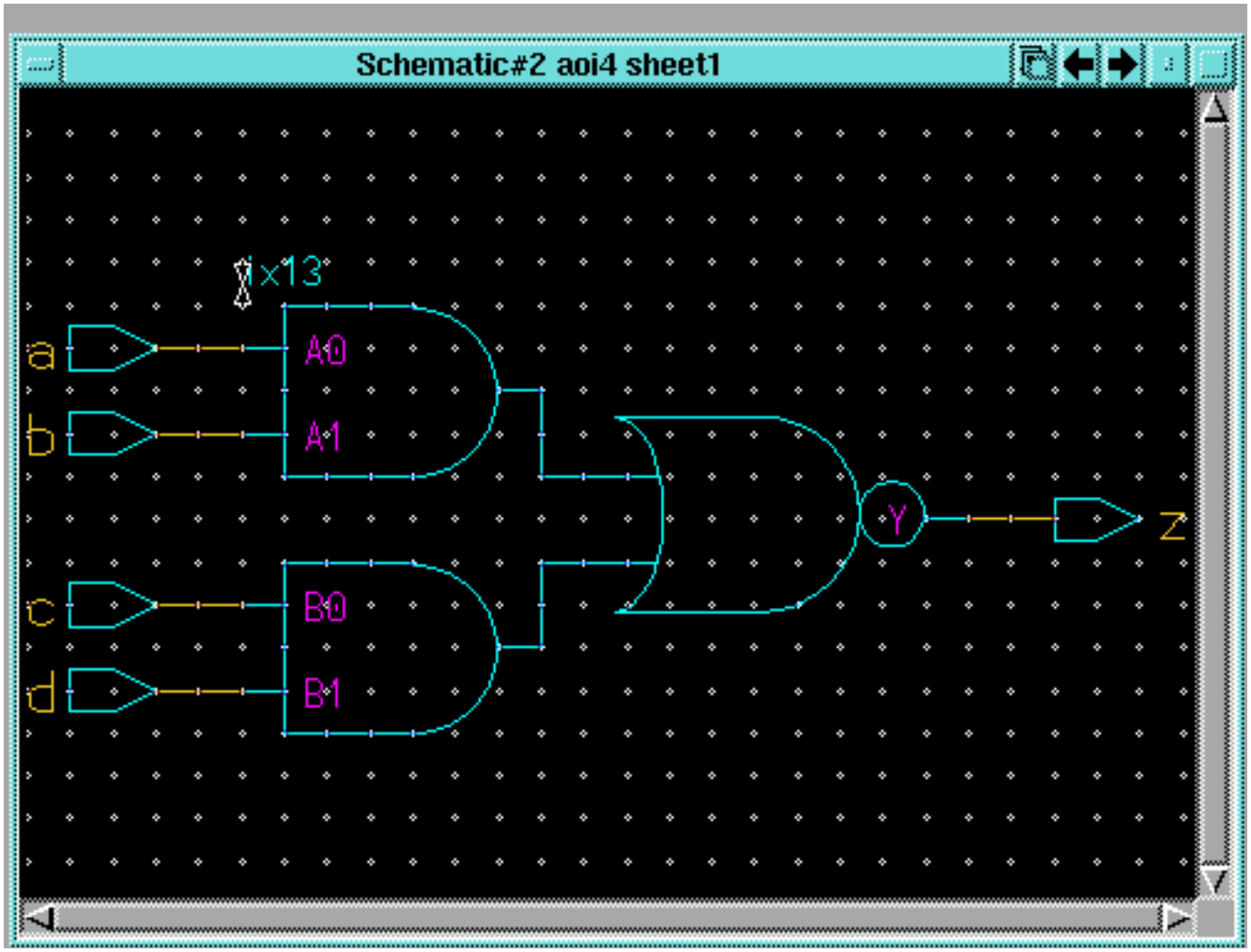
Note also, the intermediate signals temp1 and temp2 have optimized away.

Examine the schematic created by synthesis

The EDIF netlist is converted to a Mentor schematic by executing the command (in the edif directory):

```
edif2eddm  aoi4.edf  data_flow
```

When design architect is invoked upon the design we see the following:



Here we can see the direct correspondence between the gate pins and the entity pins in the statement:

ix13 : aoi22 port map (Y=>z, A0=>a, A1=>b, B0=>c, B1=>d);

The instance name (`ix13`) is also evident.

What you say is not what you get. (sometimes)

Looking at the VHDL code, one might expect something different.

```
BEGIN
  temp1 <= a AND b;
  temp2 <= c AND d;
  z     <= temp1 NOR temp2;
END data_flow;
```

This code seems to imply two AND gates feeding a NOR gate. However this is not the case. This description is a behavioral one. It does not in any way dictate what gates to use.

Two AND gates and a NOR gate would be a fine implementation, except for the fact that it is *slower, bigger,* and *consumes more power* than the single aoi22 gate.

The synthesis tool finds the “best” implementation by trying most possible implementations and choosing the optimum one.

What is a “best” implementation? Size, speed?

Data Types

Data types identify a set of values an object may assume and the operations that may be performed on it.

VHDL data type classifications:

- **Scalar: numeric, enumeration and physical objects**
- **Composite: Arrays and records**
- **Access: Value sets that point to dynamic variables**
- **File: Collection of data objects outside the model**

Certain scalar data types are predefined in a *package* called "*std*" (standard) and do not require a type declaration statement.

Examples:

- **boolean** *(true, false)*
- **bit** *('0', '1')*
- **integer** *(-2147483648 to 2147483647)*
- **real** *(-1.0E38 to 1.0E38)*
- **character** *(ascii character set)*
- **time** *(-2147483647 to 2147483647)*

Type declarations are used through constructs called *packages*.

We will use the package called *std_logic_1164* in our class. It contains the common types, procedures and functions we normally need.

A *package* is a group of related declarations and subprograms that serve a common purpose and can be reused in different parts of many models.

Using std_logic_1164

The package *std_logic_1164* is the package standardized by the IEEE that represents a nine-state logic value system known as *MVL9*.

To use the package we say:

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

The *library* clause makes a selected library containing desired packages "visible" to a model.

The *use* clause makes the library packages visible to the model.

USE clause format:

```
USE symbolic_library.pkg_name.elements_to_use
```

The name *ieee* is a *symbolic* name. It is "*mapped*" to:

```
/usr/local/apps/mti/current/modeltech/ieee
```

using the MTI utility *vmap*.

You can see all the currently active mappings by typing: *vmap*

We do not have to declare a library work. Its existence and location "./work" is understood.

Using std_logic_1164

The nine states of std_logic_1164:
(/usr/local/apps/mti/current/modeltech/vhdl_src/ieee/stdlogic.vhd)

```
PACKAGE std_logic_1164 IS
-------------------------------------------
-- logic state system (unresolved)
-------------------------------------------
  TYPE std_ulogic IS (
'U', -- Uninitialized; the default value
'X', -- Forcing Unknown; bus contention
'0', -- Forcing 0; logic zero
'1', -- Forcing 1; logic one
'Z', -- High Impedance; 3-state buffer
'W', -- Weak Unknown; bus terminator
'L', -- Weak 0; pull down resistor
'H', -- Weak 1; pull up resistor
'-'  -- Don't care; used for synthesis);
```

Why would we want all these values for signals?

VHDL Operators

Object type also identifies the operations that may be performed on an object.

Operators defined for predefined data types in decreasing order of precedence:

- **Miscellaneous: **, ABS, NOT**
- **Multiplying Operators: *, /, MOD, REM**
- **Sign: +, -**
- **Adding Operators: +, -,&**
- **Shift Operators: ROL, ROR, SLA, SLL, SRA, SRL**
- **Relational Operators: =, /=, <, <=, >, >=**
- **Logical Operators: AND, OR, NAND, NOR, XOR, XNOR**

Not all these operators are synthesizable.

Overloading

Overloading allows standard operators to be applied to other user-defined data types.

An example of overloading is the function "AND", defined as:
(/usr/local/apps/mti/current/modeltech/vhdl_src/ieee/stdlogic.vhd)

```
FUNCTION "and" (l : std_logic; r : std_logic)
RETURN UX01;

FUNCTION "and" (l, r: std_logic_vector )
RETURN std_logic_vector;
```

For Examples

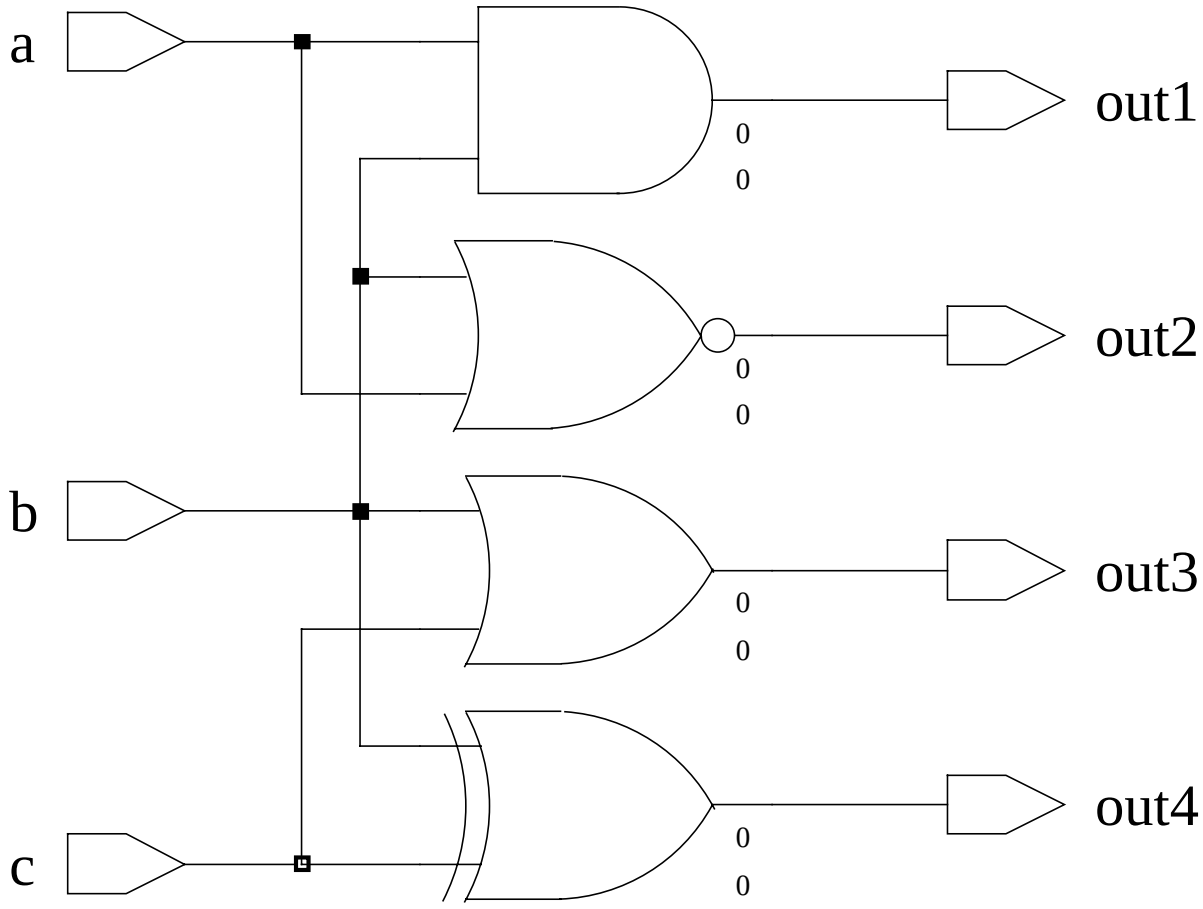
```
SIGNAL result0, signal1, signal2 : std_logic;
SIGNAL result1 : std_logic_vector(31 DOWNTO 0);
SIGNAL signal3 : std_logic_vector(31 DOWNTO 0);
SIGNAL signal4 : std_logic_vector(31 DOWNTO 0);

BEGIN
result0 <= signal1 AND signal2; -- simple AND
result1 <= signal3 AND signal4; -- many ANDs
END;
```

If we synthesize this code, what gate realization will we get?

Concurrency

To model reality, VHDL processes certain statements concurrently.

Example:



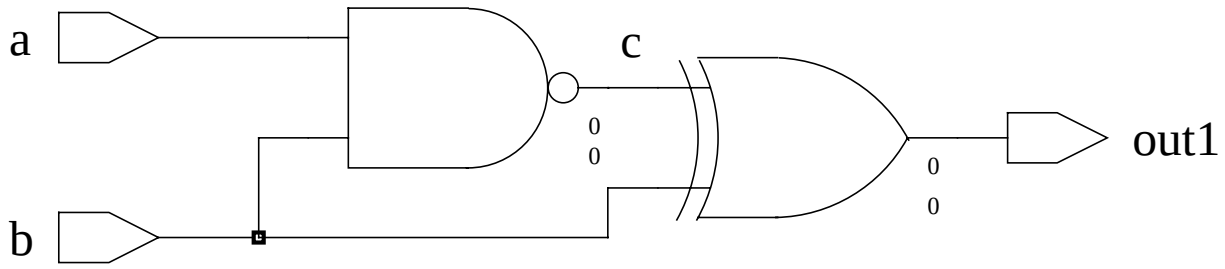

```
ARCHITETURE example of concurrent IS
BEGIN
  out1 <= a AND b;
  out2 <= a NOR b;
  out3 <= b OR c;
  out4 <= b XOR c;
END example;
```

Statement Activation

Signals connect concurrent statements.

Concurrent statements activate or "fire" when there is an event on a signal "entering" the statement.

Example:




```
ARCHITECTURE example OF concurrent IS
  SIGNAL c  :  std_logic;
  BEGIN
    c    <= a NAND b; --nand gate
    out1 <= c XOR b;  --xor gate
END example;
```

The NAND statement is activated by a change on either the a or b inputs.

The XOR statement is activated by a change on either the b input or signal c.

Note that additional signals (those not defined in the PORT clause) are defined in the architecture's declarative area.

Concurrency Again

VHDL is inherently a concurrent language.

All VHDL processes execute concurrently.

Basic granularity of concurrency is the *process*.

Concurrent signal assignments as actually one-line processes.

```
   c    <= a NAND b;  --"one line process"
   out1 <= c XOR b;   --"one line process"
```

VHDL statements execute sequentially *within a process*.

```
ARCHITECTURE example OF concurrency IS
  BEGIN
    hmmm: PROCESS (a,b,c)
    BEGIN
     c    <= a NAND b;   --"do sequentially"
     out1 <= c XOR b;    --"do sequentially"
  END PROCESS hmmm;
```

How much time did it take to do the stuff in the process statement?

Concurrency

The body of the ARCHITECTURE area is composed of one or more concurrent statements. The concurrent statements we will use are:

- **Process - the basic unit of concurrency**
- **Assertion - a reporting mechanism**
- **Signal Assignment - communication between processes**
- **Component Instantiations - creating instances**
- **Generate Statements - creating structures**

Only concurrent statements may be in the body of the architecture area.

```
ARCHITECTURE showoff OF concurrency_stmts IS
BEGIN
------concurrent club members only----------
--BLOCK
--PROCESS
--ASSERT
--a <= NOT b;
--PROCEDURE
--U1:nand1 PORT MAP(x,y,z);  --instantiation
--GENERATE
------concurrent club members only----------
END showoff;
```

Concurrent Statements - Signal Assignment

Signal assignment

We have seen the simple signal assignment statement

sig_a <= input_a AND input_b;

VHDL provides both a concurrent and a sequential signal assignment statement. The two statements can have the same syntax, but they differ in how they execute.

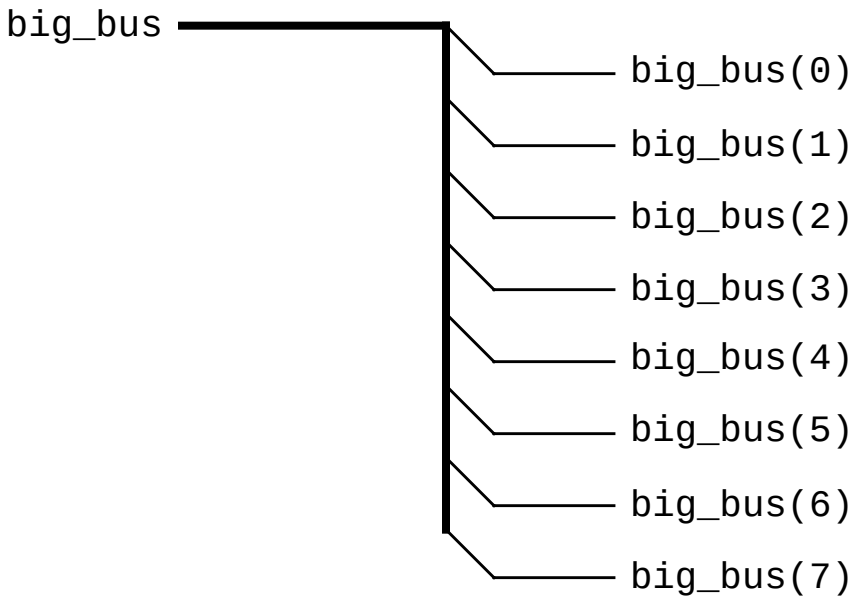
Signal Assignment with Busses

A bus is a collection of wires related in some way by function or clock domain. Examples would be an address bus or data bus.

In VHDL we refer to busses as a vector. For example:

```
--8-bit bus consisting of 8 wires carrying signals of
-- type std_logic
--all these wires may be referred to by the name big_bus
SIGNAL big_bus : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

This creates:



When we define a bus as above, the width of the bus is defined by "7 DOWNTO 0". The position of the MSB is to the left of the DOWNTO keyword. The LSB bit is to the right of DOWNTO.

The usual convention is to use DOWNTO. We will use this convention. UPTO is seldom used.

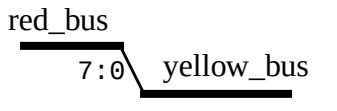
Signal Assignment with Busses (cont.)

Individual bits of a bus may be referred to like this:

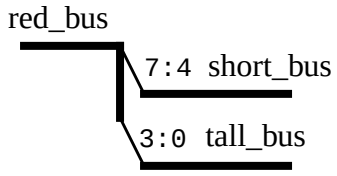
```
SIGNAL one_bit : STD_LOGIC;
SIGNAL big_bus : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
--wire called one_bit is connected to bit 6 of bus big_bus
one_bit <= big_bus(6); -- bus ripping example
```

Consider the following declarations and how they can be used.

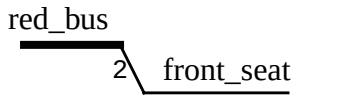
```
SIGNAL back_seat, front_seat: STD_LOGIC;
SIGNAL red_bus, yellow_bus, shift_bus
                              : STD_LOGIC_VECTOR(7 DOWNTO 0 );
SIGNAL short_bus, tall_bus, : STD_LOGIC_VECTOR(3 DOWNTO 0);
```



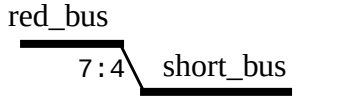

```
red_bus <= yellow_bus; -- connecting same size busses
```



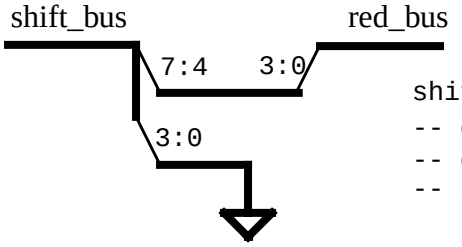

```
red_bus <= short_bus & tall_bus; -- bus concatenation
--"&" is the concatenation operator
-- MSB's of red_bus come from left most signal
```




```
front_seat <= red_bus(2); -- bus ripping
```




```
short_bus <= red_bus(7 DOWNTO 4); -- bus to bus ripping
```




```
shift_bus <= red_bus(3 DOWNTO 0) & "0000";
-- one bus created from ripping of one bus and
-- concatenation of signals connected to ground
-- shift bus is red_bus multiplied by 16
```

Bit Vector Usage

As we have seen the in the following examples VHDL has a convenient way to represent busses. A bit string literal allows us to specify the value of a bit vector. For example, the number 227_{10} could be represented as:

Binary format: B”11111010” B”1111_1010”

Hexadecimal format: X”FA”

Octal format: O”372”

The binary format may include underscores to increase readability. The underscores do not effect the value.

Values of bit string literals are inclosed in double quotes. For example: “1101”

Values of bit literals are inclosed in single quotes. For example: ‘Z’

Conditional Concurrent Signal Assignment

The conditional concurrent signal assignment statement is modeled after the "if statement" in software programming languages.

The general format for this statement is:

```
target_signal <= value1 WHEN condition1 ELSE
                 value2 WHEN condition2 ELSE
                 value3 WHEN condition3 ELSE
                            .....
                 valueN;
```

When one or more of the signals on the right-hand side change value, the statement executes, evaluating the condition clauses in textual order from top to bottom. If a condition is found to be true, the corresponding expression is executed and the values is assigned to the target signal.

The conditions must evaluate to a boolean value. i.e, True or False

Example:

```
z_out <=   a_input WHEN (select = "00") ELSE
           b_input WHEN (select = "01") ELSE
           c_input WHEN (select = "10") ELSE
           d_input WHEN (select = "11") ELSE
           'X'; -- what am I?
```

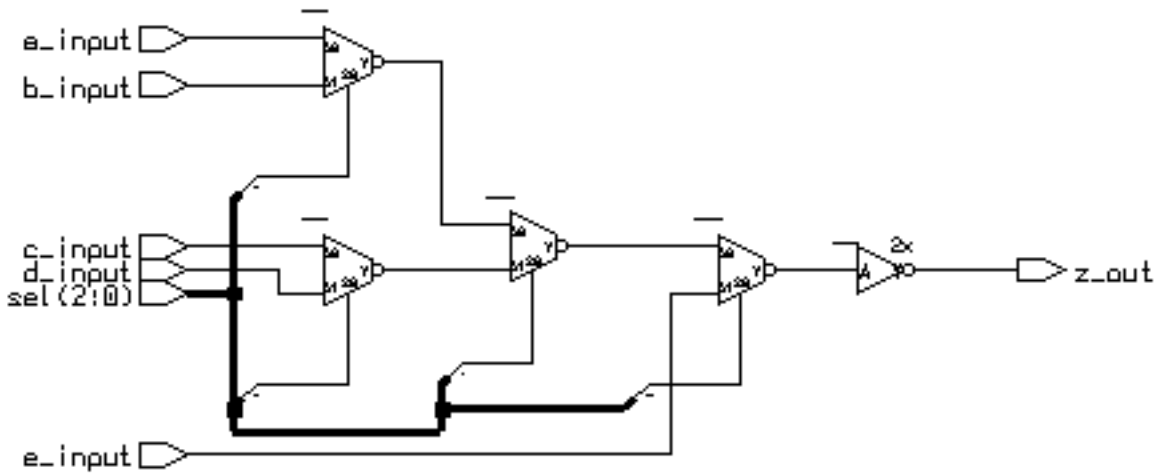
Conditional Concurrent Signal Assignment

What happens when we don't completely specify all the choices?

First, lets do it right.

```
--5:1 mux, 1 bit wide
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux5_1_1wide IS
  PORT(
     a_input   : IN STD_LOGIC;  --input a
     b_input   : IN STD_LOGIC;  --input b
     c_input   : IN STD_LOGIC;  --input c
     d_input   : IN STD_LOGIC;  --input d
     e_input   : IN STD_LOGIC;  --input e
     sel       : IN STD_LOGIC_VECTOR(2 DOWNTO 0);  --sel input
     z_out     : OUT STD_LOGIC  --data out
     );
END mux5_1_1wide;
ARCHITECTURE beh OF mux5_1_1wide IS
  BEGIN
     z_out <= a_input WHEN (sel = "000") ELSE
              b_input WHEN (sel = "001") ELSE
              c_input WHEN (sel = "010") ELSE
              d_input WHEN (sel = "011") ELSE
              e_input WHEN (sel = "100") ELSE
             'X';
 END beh;
```

When synthesized, we get:

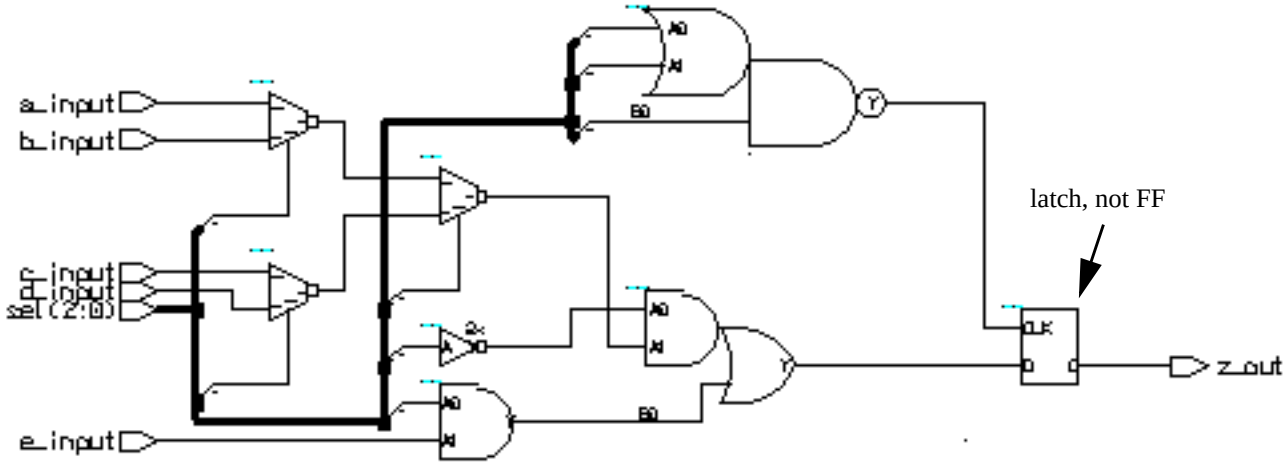


Conditional Concurrent Signal Assignment

Now let's incompletely specify the choices.

```
ARCHITECTURE noelse OF mux5_1_1wide IS
  BEGIN
     z_out <= a_input WHEN (sel = "000") ELSE
              b_input WHEN (sel = "001") ELSE
              c_input WHEN (sel = "010") ELSE
              d_input WHEN (sel = "011") ELSE
              e_input WHEN (sel = "100"); -- no ending else
 END beh;
```

When synthesized:



What happened?

- How does a transparent latch operate?
- What is the truth table for the decoder to the latch "clk" pin?

sel(2:0)	latch enable pin	behavior
000	1	latch is transparent
001	1	ditto
010	1	ditto
011	1	ditto
100	1	ditto
101	0	latch is in "hold" state
110	0	hold state
111	0	hold state

Selected Concurrent Signal Assignment

The selected concurrent signal assignment statement is modeled after the “case statement” in software programming languages.

The general form of this statement:

```
WITH discriminant SELECT
  target_signal <= value1 WHEN choice1,
                   value2 WHEN choice2,
                   value3 WHEN choice3,
                     ........
                   valueN WHEN choiceN;
                  [default_value WHEN OTHERS];
```

This statement executes when any the discriminant, value or choice expressions changes value. When it does execute, the choice clauses are evaluated. The target signal is assigned the value corresponding to the choice that matches the discriminant.

Important points for this statement:

- The discriminant must have finite discrete values. (can be enumerated).
  ```
  ERROR: Expression must return a discrete value.
  ```
- You must use or list all possible values for “choice”.
  ```
  ERROR: Case statement only covers 5 out of 729 cases.
  ```
- Only one of the choices can match the discriminant.
  ```
  ERROR: Case choice has already been specified on line 32
  ```

About “OTHERS”

The keyword **OTHERS** can be powerfully used in many situations. In general it is used to allow matching to an unspecified number of possible values of a variable. There ay be only one alternative that uses the others choice and if included in a list, it must be the last choice. In essence, it says, if a match has not yet been found and the value of the variable is within range of its type, then match with **OTHERS**.

We will see several other uses of **OTHERS** in the future.

Selected Concurrent Signal Assignment

An example from “SPAM”

```
--------------------------------------------------------------------
--2:1 mux, 16 bits wide
--------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


ENTITY mux2_1_16wide IS
 PORT(
      in_a   : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);  --input a
      in_b   : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);  --input b
      sel    : IN  STD_LOGIC;                      --select input
      output : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)   --data output
      );
  END mux2_1_16wide;

ARCHITECTURE beh OF mux2_1_16wide IS
  BEGIN
    WITH sel SELECT
      output <= in_a WHEN ‘0’,
                in_b WHEN ‘1’,
                (OTHERS => ‘X’) WHEN OTHERS;
END beh;
```

OTHERS again

Here we see **OTHERS** used to match cases where sel is not ‘1’ or ‘0’ in the **WHEN OTHERS** clause. i.e.:

(OTHERS => ‘X’) WHEN OTHERS;

OTHERS is also used to provide a shorthand method of saying, “make all the bits of the target signal ‘X” for however many bits are in target signal.

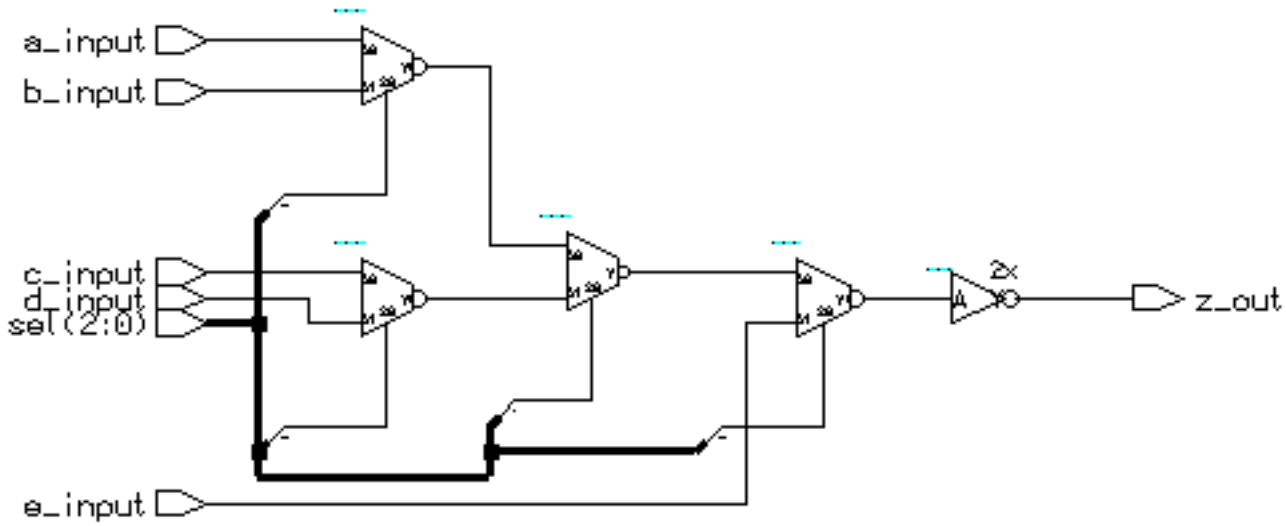
(OTHERS => ‘X’) WHEN OTHERS;

Why was ‘X ‘assigned to output when sel was neither ‘0’ or ‘1’?

Selected Concurrent Signal Assignment

A more simple example with synthesis results.

```
--5:1 mux, 1 bit wide
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux5_1_1wide IS
  PORT(
     a_input   : IN STD_LOGIC;  --input a
     b_input   : IN STD_LOGIC;  --input b
     c_input   : IN STD_LOGIC;  --input c
     d_input   : IN STD_LOGIC;  --input d
     e_input   : IN STD_LOGIC;  --input e
     sel       : IN STD_LOGIC_VECTOR(2 DOWNTO 0);  --sel input
     z_out     : OUT STD_LOGIC  --data out
     );
END mux5_1_1wide;
ARCHITECTURE beh OF mux5_1_1wide IS
  BEGIN
     WITH sel SELECT
     z_out <= a_input WHEN “000”,
              b_input WHEN “001”,
              c_input WHEN “010”,
              d_input WHEN “011”,
              e_input WHEN “100”,
             ‘X’ WHEN OTHERS; --if sel could be be 110, 111? correct?
 END beh;
```



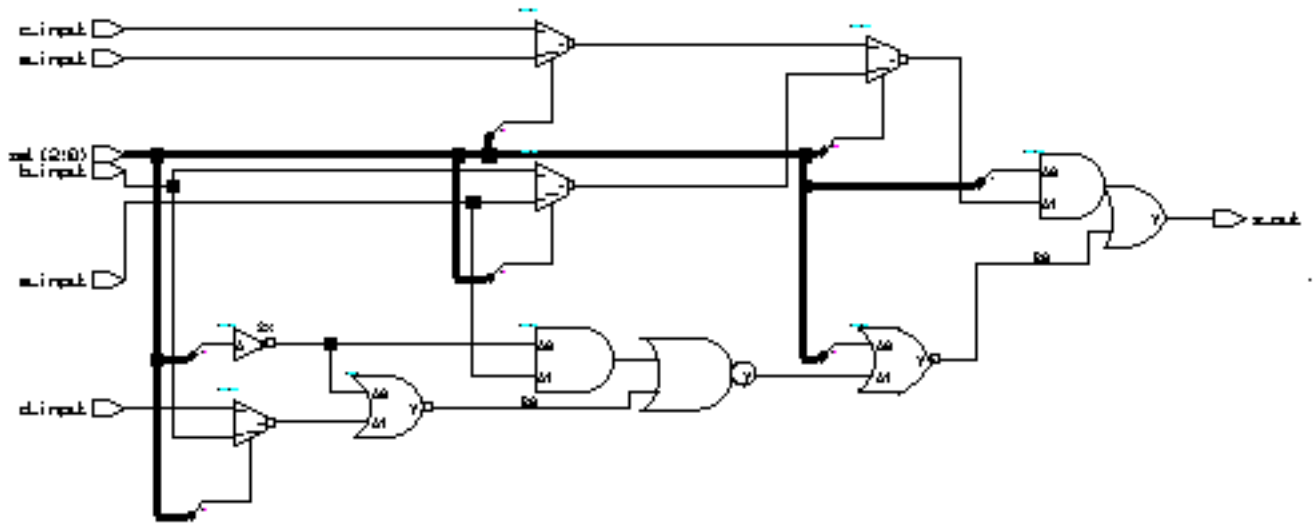
How will this circuit react to sel(2:0) values greater than “100”?

Making Choices

When we want the same target signal assignment to happen for several discriminant choices how do we specify it? Lets alter the function of our mux example as follows. The entity declaration is identical to before.

```
ARCHITECTURE beh OF mux5_1_1wide IS
  BEGIN
     WITH sel SELECT
     z_out <= a_input WHEN “000” | “001” | “111”,
              b_input WHEN “011” | “101”,
              c_input WHEN “010”,
              d_input WHEN “100”,
              e_input WHEN “110”,
             ‘X’ WHEN OTHERS;
 END beh;
```

The signal z_out gets the value of a_input when sel is equal to “000”, “001” or “111”. Signal z_out gets the value of b_input when sel is equal to “011” or “101”. The synthesized version of this mux looks like this:



As you can see, once a model is synthesized it can be hard to figure out how it works.

Concurrent Statements - Component Instantiation

Another concurrent statement is known as *component instantiation*. Component instantiation can be used to connect circuit elements at a very low level or most frequently at the top level of a design.

VHDL written in this form is known as *Structural VHDL*.

The instantiation statement connects a declared component to signals in the architecture.

The instantiation has 3 key parts:

- ***Label* - Identifies unique instance of component**
- ***Component Type* - Select the desired declared component**
- ***Port Map* - Connect component to signals in the architecture**

Example:

```
u1 : reg1 PORT MAP(d=>d0,clk=>clk,q=>q0);
```

label

component type

the pin “clk” on reg1

wire that pin “clock” is connected to

When instantiating components:

- **Local and actual must be of same data type.**
- **Local and actual must be of compatible modes.**

Locally declared signals do not have an associated mode and can connect to a local port of any mode.

Labels

Labels are used to provide internal documentation.

<u>May</u> be used with:

- **Concurrent Assertion Statements**
- **Concurrent Signal Assignments**
- **Process Statements**
- **Loop Statements**
- **Generate Statements**

<u>Must</u> be used with:

- **Component Instantiation Statements**

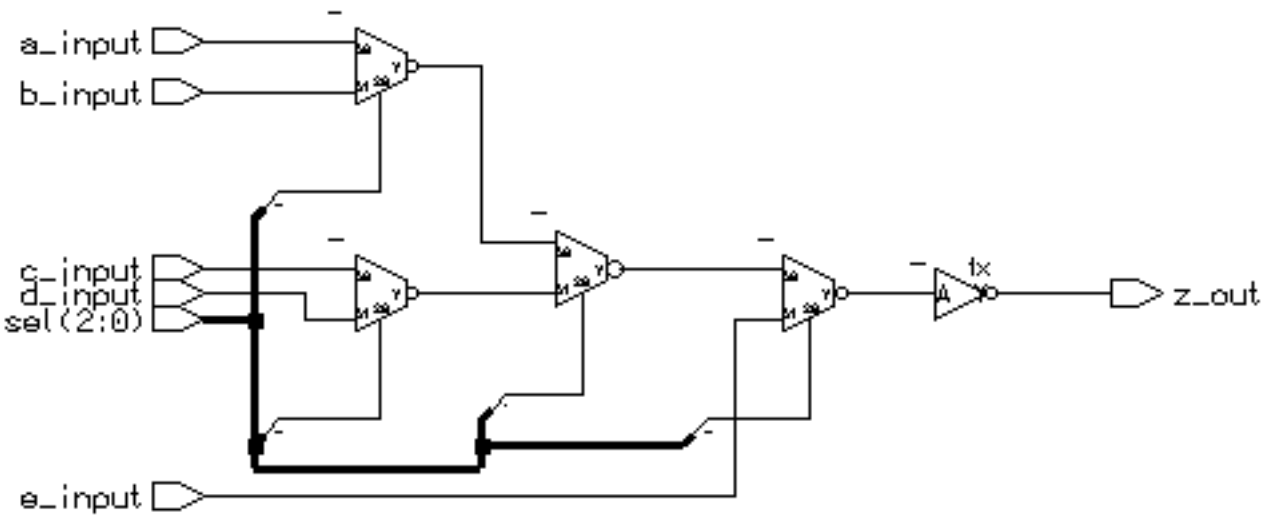
Component Instantiation

5:1 mux using component instantiaion:

```
--5:1 mux, 1 bit wide
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY adk;
USE adk.all;

ENTITY mux5_1_1wide IS
  PORT(
     a_input   : IN STD_LOGIC;  --input a
     b_input   : IN STD_LOGIC;  --input b
     c_input   : IN STD_LOGIC;  --input c
     d_input   : IN STD_LOGIC;  --input d
     e_input   : IN STD_LOGIC;  --input e
     sel       : IN STD_LOGIC_VECTOR(2 DOWNTO 0);  --sel input
     z_out     : OUT STD_LOGIC  --data out
     );
END mux5_1_1wide;
ARCHITECTURE beh OF mux5_1_1wide IS

  SIGNAL temp0, temp1, temp2, temp3 : STD_LOGIC;

  COMPONENT mux21 PORT( a0,a1,s0 : IN  STD_LOGIC;
                                y : OUT STD_LOGIC);  END COMPONENT;
  COMPONENT inv01 PORT(  a : IN  STD_LOGIC;
                         y : OUT STD_LOGIC);  END COMPONENT;
  BEGIN
  U1 : mux21 PORT MAP(a0  => a_input,
                      a1  => b_input,
                      s0  => sel(0),
                       y  => temp0);
  U2 : mux21 PORT MAP(a0  => c_input,
                      a1  => d_input,
                      s0  => sel(0),
                       y  => temp1);
  U3 : mux21 PORT MAP(a0  => temp0,
                      a1  => temp1,
                      s0  => sel(1),
                       y  => temp2);
  U4 : mux21 PORT MAP(a0  => temp2,
                      a1  => e_input,
                      s0  => sel(2),
                       y  => temp3);
  U5 : inv01  PORT MAP(a   => temp3,
                       y   => z_out);
 END beh;
```

The synthesized structural 5:1 mux

The synthesized mux is a faithful representation of our structural VHDL. (it better be!) Actually the synthesis tools “hands” are tied. The structural VHDL told exactly how the components were to be wired. It also specified exactly what logic cells were to be used. The synthesis tool actually had nothing to do except make the edif netlist and schematic.



Component Instantiation (cont.)

A few notes about the structural 5:1 mux code:

The logic cells used here were in a library called *adk*. To access these cells the declaration of this library was necessary at the top of the file.

```
LIBRARY adk;
USE adk.all;
```

Before we can use the cells in an instantiation statement, we must declare them. This is seen in the statements:

```
COMPONENT mux21 PORT( a0,a1,s0 : IN  STD_LOGIC;
                                 y : OUT STD_LOGIC);  END COMPONENT;
COMPONENT inv01 PORT(  a : IN  STD_LOGIC;
                       y : OUT STD_LOGIC);  END COMPONENT;
```

To wire the mux21 cells together, temporary signals, *temp0*, *temp1*, *temp2* and *temp3* were declared.

```
SIGNAL temp0, temp1, temp2, temp3 : STD_LOGIC;
```

Finally, the component instantiations stitch the design together.

```
U1 : mux21 PORT MAP(a0  => a_input,
                    a1  => b_input,
                    s0  => sel(0),
                    y  => temp0);
```

The PORT MAP statement describes the connections between pins of the cell and the signals. The connections are described by the format:

```
pin_on_module  =>  signal_name,
```

The first name is the module pin name, the second is the name of the signal the pin is to be connected to. This format is called *named association*.

With named association, the order of associations is not required to be in the same order as port declaration in the component.

Named vs. Positional Association

As previously mentioned, pin/signal pairs used with a PORT MAP may be associated by position. For example,

```
U1 : mux21 PORT MAP(a_input,b_input,sel(0),temp0);
```

This form is not preferred because any change in the port list (it often happens in the design phase) will be difficult to incorporate. Try doing it for entities with 50 or more signals and you'll begin to appreciate the point.

For example, some real code.....

Sample PORT MAP (w/named association)

```
dramfifo_0: dramfifo
PORT MAP(       reg_data            =>  reg_data         ,
                dram_state_ps       =>  dram_state_ps    ,
                dram_cnt_ps         =>  dram_cnt_ps      ,
                dram_cycle_type     =>  dram_cycle_type  ,
                addr_adv            =>  addr_adv         ,
                line_shift          =>  line_shift       ,
                cycle_start         =>  cycle_start      ,
                done                =>  done             ,
                any_rdgnt           =>  any_rdgnt        ,
                any_wrgnt           =>  any_wrgnt        ,
                test_mode           =>  test_mode        ,
                scl_ratio_ack       =>  scl_ratio_ack    ,
                y_wrptrlo_wen       =>  y_wrptrlo_wen    ,
                y_wrptrhi_wen       =>  y_wrptrhi_wen    ,
                u_wrptrlo_wen       =>  u_wrptrlo_wen    ,
                u_wrptrhi_wen       =>  u_wrptrhi_wen    ,
                v_wrptrlo_wen       =>  v_wrptrlo_wen    ,
                v_wrptrhi_wen       =>  v_wrptrhi_wen    ,
                wrcntrlo_wen        =>  wrcntrlo_wen     ,
                wrcntrhi_wen        =>  wrcntrhi_wen     ,
                y_rdptrlo_wen       =>  y_rdptrlo_wen    ,
                y_rdptrhi_wen       =>  y_rdptrhi_wen    ,
                u_rdptrlo_wen       =>  u_rdptrlo_wen    ,
                u_rdptrhi_wen       =>  u_rdptrhi_wen    ,
                v_rdptrlo_wen       =>  v_rdptrlo_wen    ,
                v_rdptrhi_wen       =>  v_rdptrhi_wen    ,
                rdcntrlo_wen        =>  rdcntrlo_wen     ,
                rdcntrhi_wen        =>  rdcntrhi_wen     ,
                yeol_cntr_wen       =>  yeol_cntr_wen    ,
                ueol_cntr_wen       =>  ueol_cntr_wen    ,
                veol_cntr_wen       =>  veol_cntr_wen    ,
                line_length_wen     =>  line_length_wen  ,
                ptr_rollbit_wen     =>  ptr_rollbit_wen  ,
                clk_24              =>  clk_24           ,
                clk_48              =>  clk_48           ,
                rst_24              =>  rst_24           ,
                rst_48              =>  rst_48           ,
                s_capt_en           =>  s_capt_en        ,
                vsync               =>  vsync            ,
                even_fld            =>  even_fld         ,
                qual_hsync          =>  qual_hsync       ,
                sr_sel              =>  sr_sel           ,
                current_sr          =>  current_sr       ,
                allow_rdreq         =>  allow_rdreq      ,
                allow_wrreq         =>  allow_wrreq      ,
                wr_addr             =>  wr_addr          ,
                rd_addr             =>  rd_addr          ,
                last_line_segment   =>  last_line_segment,
                start_of_video      =>  start_of_video   ,
                end_of_video        =>  end_of_video     ,
                line_length_rback   =>  line_length_rback,
                  dcu_status          =>  dcu_status);
```

Same PORT MAP (w/positional association)

```
-- dram fifo address control
dramfifo_0: dramfifo
PORT MAP(reg_data, dram_state_ps, dram_cnt_ps, dram_cycle_type,
addr_adv, line_shift, cycle_start, done, any_rdgnt, any_wrgnt,
test_mode, scl_ratio_ack, y_wrptrlo_wen, y_wrptrhi_wen, u_wrptrlo_wen,
u_wrptrhi_wen, v_wrptrlo_wen, v_wrptrhi_wen, wrcntrlo_wen,
wrcntrhi_wen, y_rdptrlo_wen, y_rdptrhi_wen, u_rdptrlo_wen,
u_rdptrhi_wen, v_rdptrlo_wen, v_rdptrhi_wen, rdcntrlo_wen,
rdcntrhi_wen, yeol_cntr_wen, ueol_cntr_wen, veol_cntr_wen,
line_length_wen, ptr_rollbit_wen, clk_24, clk_48, rst_24, rst_48,
s_capt_en, vsync, even_fld, qual_hsync, sr_sel, current_sr,
allow_rdreq, allow_wrreq, wr_addr, rd_addr, last_line_segment,
start_of_video, end_of_video, line_length_rback, dcu_status);
```

Now, lets say you need to add an extra signal in the module *dramfifo*. You want to put it just after *ueol_cntr_wen*. But let's say your signals do not necessarily have the same names as the pins. This means you would have to manually count through the list of signals to find out where to put the new one in the port map in exactly the same order. How would you know for sure its in the right position? Count through the list again! Do you have time to do this?

The Moral of the Story: Use named association.

Association lists - Some last items...

Suppose you have a module that is a four to one mux, but you only need three inputs. What do you do with the unused input? What about unused outputs?

If the module you are instantiating has a defined *default port value*, the keyword **OPEN** can be used to allow the input to be assigned the default port value. Thus the entity for a 4:1 mux with a defined default port value would look like this:

```
ENTITY mux41 IS
  PORT(
     a0    : IN STD_LOGIC := '0';  --input a0 can be left OPEN
     a1    : IN STD_LOGIC := '0';  --input a1 can be left OPEN
     a2    : IN STD_LOGIC := '0';  --input a2 can be left OPEN
     a3    : IN STD_LOGIC := '0';  --input a3 can be left OPEN
     sel   : IN STD_LOGIC_VECTOR(1 DOWNTO 0);  --sel input
     z_out : OUT STD_LOGIC  --data out
     );
END mux21;
```

The initalization expression ":= '0'" in the port declaration states that the input signals *a_input* , *b_input*, *c_input* and *d_input* will take on the default value'0' if they are left unconnected by a component instantiation.

Thus we could instantiate the 4:1 mux as follows:

```
U1 : mux41 PORT MAP(a0     => a_input,
                    a1     => b_input,
                    a2     => c_input,
                    a3     => OPEN,  --a3 is assigned the value '0'
                    sel    => sel_input),
                     z_out  => data_out);
```

Unconnected output ports are also designated by using the keyword OPEN. However, the associated design entity does not have to supply a default port value. Here is an adder with a unused carry output.

```
U17 : adder PORT MAP(a_in      => a_data,
                     b_in      => b_data,
                     sum       => output,
                    carry_out => OPEN);
```

Association lists - Some last items...

What about inputs to a module that are tied constantly high or low?

As usual with VHDL there are several solutions.

```
--four to one mux with one input tied low
logic_zero <= '0'; --a ground signal
U1 : mux41 PORT MAP(a0  => a_input,
                    a1  => b_input,
                    a2  => c_input,
                    a3  => logic_zero,
                    sel  => select,
                    y  => temp0);
```

This is a little cleaner:

```
--four to one mux with one input tied low
logic_zero <= '0'; --a ground signal
U1 : mux41 PORT MAP(a0  => a_input,
                    a1  => b_input,
                    a2  => c_input,
                    a3  => '0',
                    sel  => select,
                     y  => temp0);
```

However, you cannot do this:

```
--four to one mux with one input tied low
U1 : mux41 PORT MAP(a0  => a_input,
                    a1  => b_input,
                    a2  => c_input,
                    a3  => (a_input AND c_input),
                    sel  => select,
                     y  => temp0);
```

The expressions supplied as connections to the module or cell pins must be constant values only.

Concurrent Statements - GENERATE

VHDL provides the GENERATE statement to create well-patterned structures easily.

Any VHDL concurrent statement can be included in a GENERATE statement, including another GENERATE statement.

Two ways to apply

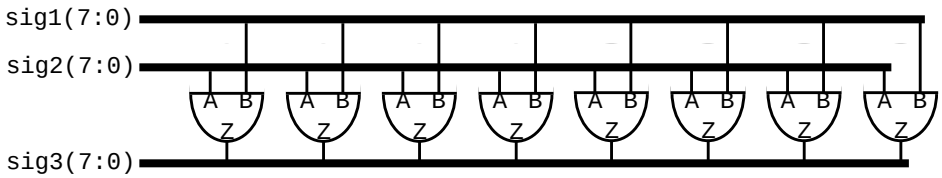
- **FOR scheme**
- **IF scheme**

FOR Scheme Format:

```
label : FOR identifier IN range GENERATE
  concurrent_statements;
END GENERATE [label];
```

Generate Statement - FOR scheme

```
ARCHITECTURE test OF test IS
COMPONENT and02
  PORT( a0 : IN  std_logic;
        a1 : IN  std_logic;
        y  : OUT std_logic);
 END COMPONENT and02;

BEGIN
 G1 : FOR n IN (length-1) DOWNTO 0 GENERATE
   and_gate:and02
   PORT MAP( a0 => sig1(n),
             a1 => sig2(n),
             y  => z(n));
  END GENERATE G1;
END test;
```



With the FOR scheme

- **All objects created are similar.**
- **The GENERATE parameter must be discrete and is undefined outside the GENERATE statement.**
- **Loop cannot be terminated early**

Note: This structure could have been created by:

```
sig3 <= sig1 AND sig2;
```

provided the AND operator was overloaded for vector operations.

Generate Statement - IF scheme

Allows for conditional creation of components.

Can’t use ELSE or ELSIF clauses.

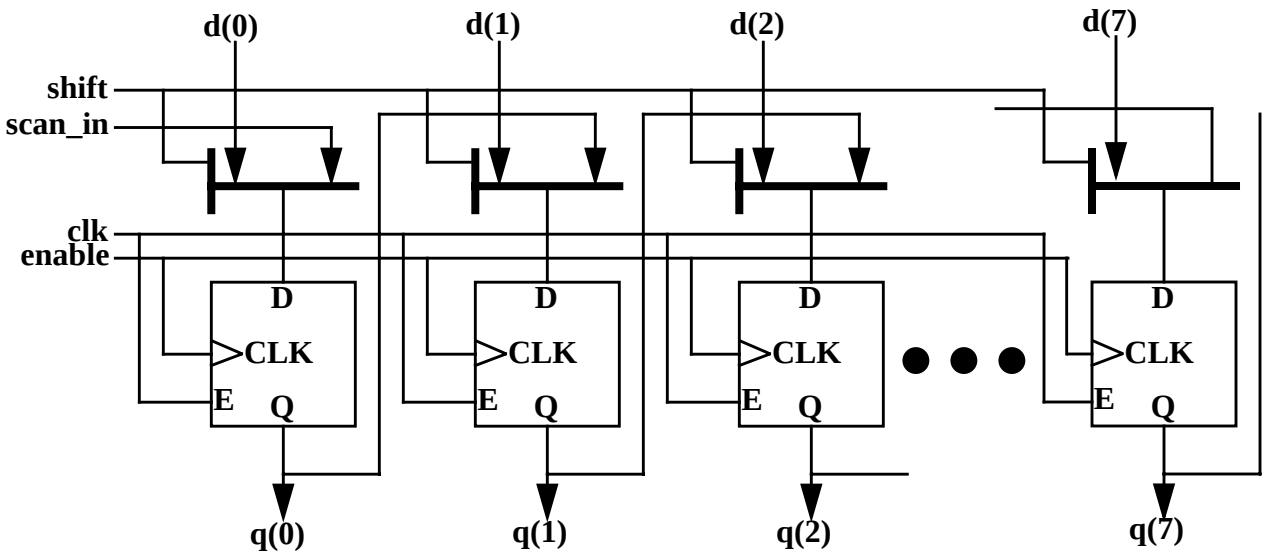
IF Scheme Format:

```
label : IF (boolean_expression) GENERATE
  concurrent_statements;
END GENERATE [label];
```

The next slide will show how we can use both FOR and IF schemes.

Use of GENERATE - An example

Suppose we want to build an 8-bit shift register.



Suppose furthermore that we had previously defined the following components:

```
ENTITY dff IS
  PORT(d, clk, en   : IN   std_logic;
       q, qn        : OUT  std_logic);
  END ENTITY dff;

ENTITY mux21 IS
  PORT(a, b, sel    : IN   std_logic;
       z            : OUT  std_logic);
  END ENTITY mux21;
```

Using GENERATE

From the block diagram we know what the entity should look like.

```
ENTITY sr8 IS
 PORT(
  din     : IN std_logic_vector(7 DOWNTO 0);
  sel     : IN std_logic;
  shift   : IN std_logic;
  scan_in : IN std_logic;
  clk     : IN sed_logic;
  enable  : IN std_logic;
  dout   : OUT std_logic_vector(7 DOWNTO 0));
```

Within the architecture statement we have to declare the components within the declaration region before using them. This is done as follows:

```
ARCHITECTURE example OF sr8 IS
--declare components in declaration area
COMPONENT dff IS
  PORT(d, clk, en  : IN   std_logic;
       q, qn       : OUT  std_logic);
  END COMPONENT;
COMPONENT mux21 IS
  PORT(a, b, sel : IN   std_logic;
       z         : OUT  std_logic);
  END COMPONENT;
```

Component declarations look just like entity clauses, except COMPONENT replaces ENTITY. Use cut and paste to prevent mistakes!

Using Generate

After the component declarations, we declare the internal signal.

```
SIGNAL mux_out : IN std_logic_vector(7 DOWNTO 0);
```

With loop and generate statements, instantiate muxes and dff's.

```
BEGIN
  OUTERLOOP: FOR i IN 0 TO 7 GENERATE
      INNERLOOP1: IF (i = 0) GENERATE
        MUX: mux21 PORT MAP(a  => d(i),
                            b  => scan_in,
                            z  => mux_out(i));
        FLOP: dff PORT MAP(d   => mux_out(i),
                           clk => clk,
                           en  => enable,
                           q   => dout(i)); --qn not listed
      END GENERATE INNERLOOP1;
      INNERLOOP2: IF (i > 0) GENERATE
        MUX: mux21 PORT MAP(a  => d(i),
                            b  => dout(i-1),
                            z  => mux_out(i));
        FLOP: dff PORT MAP(d   => mux_out(i),
                           clk => clk,
                           en  => enable,
                           q   => dout(i),
                           qn  => OPEN); --qn listed as OPEN
       END GENERATE INNERLOOP2;
  END GENERATE OUTERLOOP;
END example;
```

Concurrent Statements - ASSERT

The assertion statement checks a condition and reports a message with a severity level if the condition is not true.

Format:

```
ASSERT condition;

ASSERT condition REPORT “message”

ASSERT condition SEVERITY level;

ASSERT condition REPORT “message” SEVERITY
level;
```

Example:

```
ASSERT signal_input = ‘1’
  REPORT “Input signal_input is not 1”
    SEVERITY WARNING;
```

Severity levels are:

- **Note - general information**
- **Warning - undesirable condition**
- **Error - task completed, result wrong**
- **Failure - task not completed**

Simulators stop when the severity level matches or exceeds the specified severity level.

Simulators generally default to a severity level of “failure”

Assert Statements

Assert statements may appear within:

- **concurrent statement areas**
- **sequential statement areas**
- **statement area of entity declaration**

Example:

```
ENTITY rs_flip_flop IS
  PORT(r, s   :  IN  std_logic;
       q, qn  :  OUT std_logic);
END rs_flip_flop;

ARCHITECTURE behav OF rs_flip_flop IS
BEGIN
  ASSERT NOT (r = '1' AND s = '1')
    REPORT "race condition!"
      SEVERITY FAILURE;
               *
               *
               *
END behav;
```

<u>Remember</u>, the ASSERT statement triggers when the specified condition is *false*.

Concurrent Statements - Process Statement

The PROCESS statement encloses a set of *sequentially executed* statements. Statements within the process are executed in the order they are written. However, when viewed from the "outside" from the "outside", a process is a single concurrent statement.

Format:

```
label:
PROCESS (sensitivity_list) IS
  --declarative statements
  BEGIN
    --
    --sequential activity statements
    --only sequential statements go in here
    --
  END PROCESS [label];
```

Example:

```
ARCHITECTURE example OF nand_gate IS
  BEGIN
    nand_gate: PROCESS (a,b)
    BEGIN
      IF a = '1' AND b = '1' THEN
        z <= '0';
      ELSE
        z <= '1';
      END IF;
  END PROCESS nand_gate;
```

Why use a process? Some behavior is easier and more natural to describe in a sequential manner. The next state decoder in a state machine is an example.

Process Sensitivity List

The process *sensitivity list* lists the signals that will cause the process statement to be executed.

***Any* transition on *any* of the signals in the signal sensitivity list will cause the process to execute.**

Example:

```
ARCHITECTURE example OF nand_gate IS
  BEGIN
    bozo: PROCESS (a,b)
    -- wake up process if a and/or b changes
    BEGIN
      IF a = '1' AND b = '1' THEN
        z <= '0';
      ELSE
        z <= '1';
      END IF;
  END PROCESS bozo;
END example;
```

Signals to put in the sensitivity list:

- **Signals on the right hand side of assignment statements.**
- **Signals used in conditional expressions**

What happens if a signal is left out of the sensitivity list?
What does the synthesis tool do with the sensitivity list?

Avoid problems with sensitivity list omissions by compiling with "sythesis check" on. Like this:

```
vcom -93 -check_synthesis test.vhd
```

What about Delay?

Note that so far we haven't mentioned delay. Why not?

Both propagation delay and wiring delay is a real-world problem that must be eventually dealt with. However, at the model creation stage, it is helpful to not have to consider delay. Instead, the emphasis is to create correct functional behavior.

However, this does not mean the designer can go about designing with no concern about delay. When writing HDL code, you must have a very good idea what the structure you are creating will look like in a schematic sense. Otherwise, the synthesized circuit may have excessive delays, preventing its operation at the desired speed.

VHDL does have statements for representing several different kinds of delay. However, when describing a circuit to be synthesized, we never use them because the synthesis tool ignores them on purpose.

The aspect of delay is added to a synthesized netlist after the functionality has been proven correct. When real delays are inserted into your design (this is done automatically) often a whole world of problems crop up.

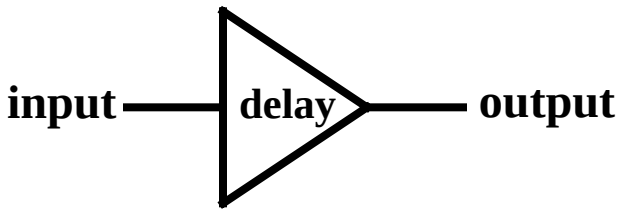
The basic idea is to make a model work, and then make it work at the desired speed. Only experience will help you determine how fast your HDL code will eventually run.

Delay Types

VHDL signal assignment statements prescribe an amount of time that must transpire before a signal assumes its new value.

This prescribed delay can be in one of three forms:

- **Transport:**
  propagation delay only
- **Inertial:**
  minimum input pulse width and propagation delay
- **Delta:**
  the default if no delay time is explicitly specified



Signal assignment is actually a *scheduling* for a future value to be placed on the signal.

Signals maintain their original value until the time for the scheduled update to occur.

<u>Any</u> signal assignment will incur a delay of one of the three types above.

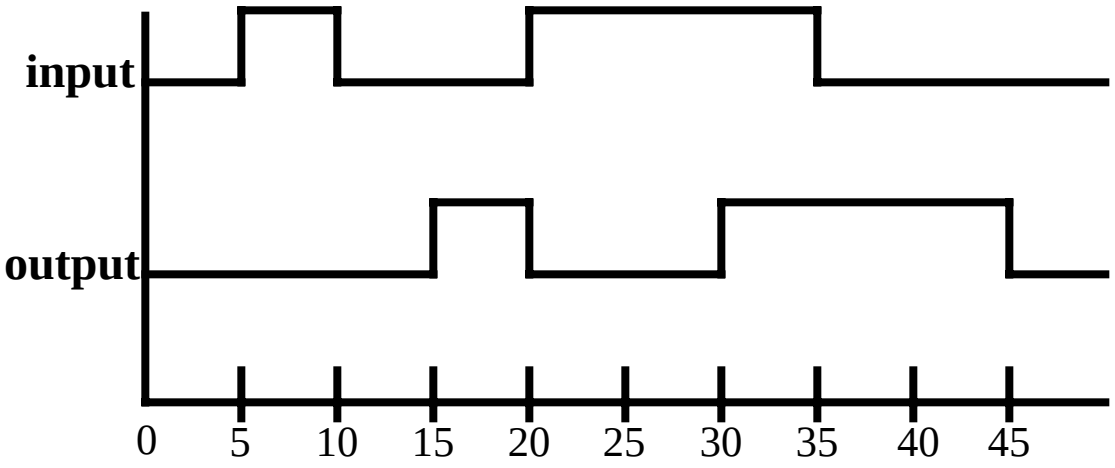
Delay Types - Transport

Delay must be explicitly specified by the user by the keyword TRANSPORT.

The signal will assume the new value after specified delay.

Example:

```
output <= TRANSPORT buffer(input) AFTER 10ns;
```



Transport delay is like a infinite bandwidth transmission line.

Delay Types - Inertial

Inertial delay is the default in VHDL statements which contain the “AFTER” clause.

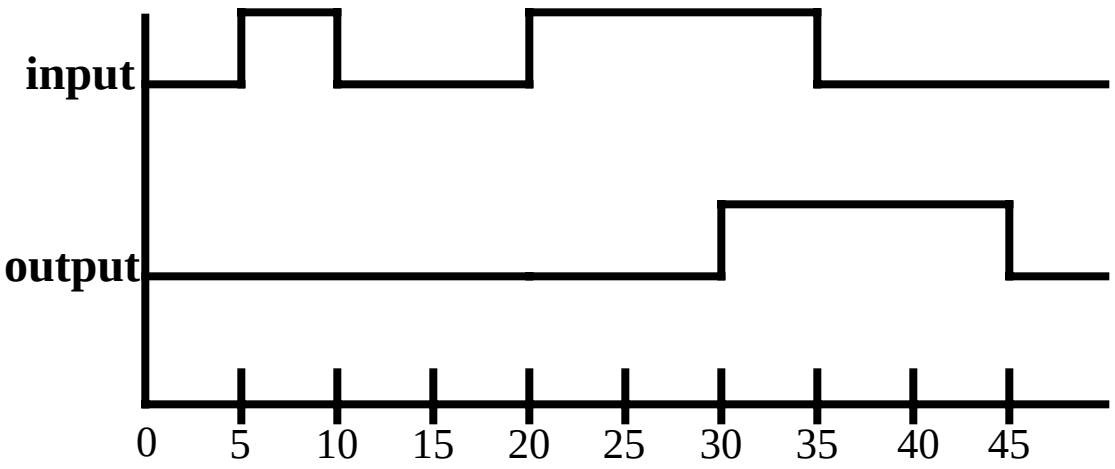
Inertial delay provides for specification of input pulse width, i.e. ‘inertia’ of output, and propagation delay.

Format:

```
target <= [REJECT time_expr] INERTIAL waveform
AFTER time
```

Example (most common):

```
output <= buffer(input) AFTER 10ns;
```



When not used, the REJECT clause defaults to the value of the AFTER clause.

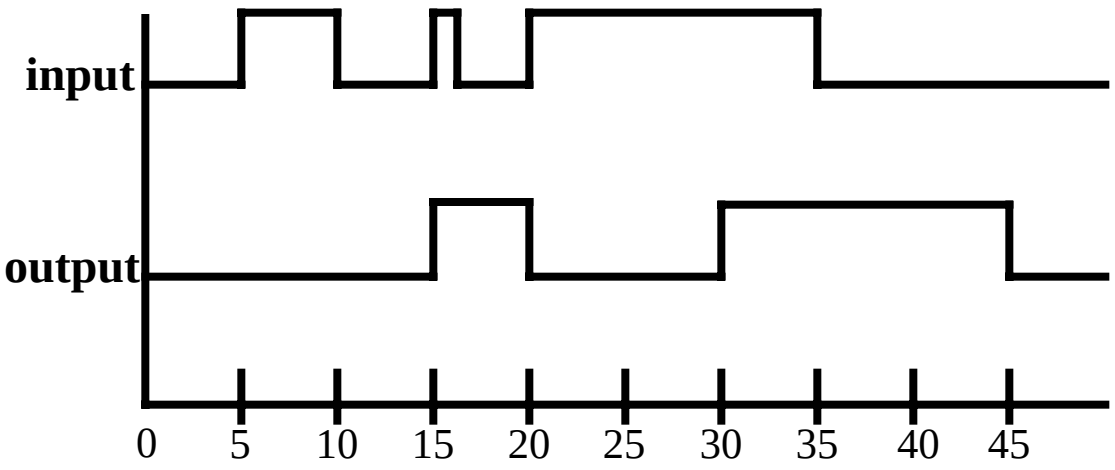
Inertial delay acts like a real gate. It “eats” pulses narrower in width than the propagation delay.

Delay Types - Inertial

Example of gate with “inertia” smaller than propagation delay:

This shows a buffer that has a prop delay of 10ns, but passes pulses greater than 5ns.

```
output <= REJECT 5ns INERTIAL buffer(input)
AFTER 10ns;
```



REJECT can be used only with the keyword INERTIAL.

Delay Types - Delta Delay

Delta delay **is the signal assignment propagation delay if none is explicitly prescribed.**

A delta time is an infinitesimal, but quantized unit of time.

An infinite number of delta times equals zero simulator time.

The delta delay mechanism provides a minimum delay so that the simulation cycle can operate correctly when no delays are stated explicitly. That is:

- **all active processes to execute in the same simulation cycle**
- **each active process will suspend at some *wait* statement**
- **when all processes are suspended, simulation is advanced the minimum time step necessary so that some signals can take on their new values**
- **processes then determine if the new signal values satisfy the conditions to proceed again from the wait condition**

Sequential Operations

Statements within processes are executed in the order in which they are written.

The sequential statements we will look at are:

- **Variable Assignment**
- **Signal Assignment***
- **If Statement**
- **Case Statement**
- **Loops**
- **Next Statement**
- **Exit Statement**
- **Return Statement**
- **Null Statement**
- **Procedure Call**
- **Assertion Statement***

***Have both a sequential and concurrent form.**

Variable Declaration and Assignment

Variables can be used only within sequential areas.

Format:

```
VARIABLE var_name : type [:= initial_value];
```

Example:

```
VARIABLE spam  : std_logic := '0';


ARCHITECTURE example OF funny_gate IS
SIGNAL c : STD_LOGIC;
  BEGIN
    funny: PROCESS (a,b,c)
    VARIABLE temp  : std_logic;
    BEGIN
      temp := a AND b;
      z <= temp OR c;
    END PROCESS funny;
 END ARCHITECTURE example;
```

Variables assume value instantly.

Variables simulate more quickly since they have no time dimension.

Remember, variables and signals have different assignment operators:

```
a <= new_value;  --signal assignment
a := new_value; --variable assignment
```

Sequential Operations - IF Statement

Provides conditional control of sequential statements.

Condition in statement must evaluate to a Boolean value.

Statements execute if boolean evaluates to TRUE.

Formats:

```
IF condition THEN           --simple IF (latch)
-- sequential statements
END IF;


IF condition THEN           --IF-ELSE
-- sequential statements
ELSE
-- sequential statements
END IF;


IF condition THEN           --IF-ELSIF-ELSE
-- sequential statements
ELSIF condition THEN
-- sequential statements
ELSE
-- sequential statements
END IF;
```

Sequential Operations - IF Statement

Examples:

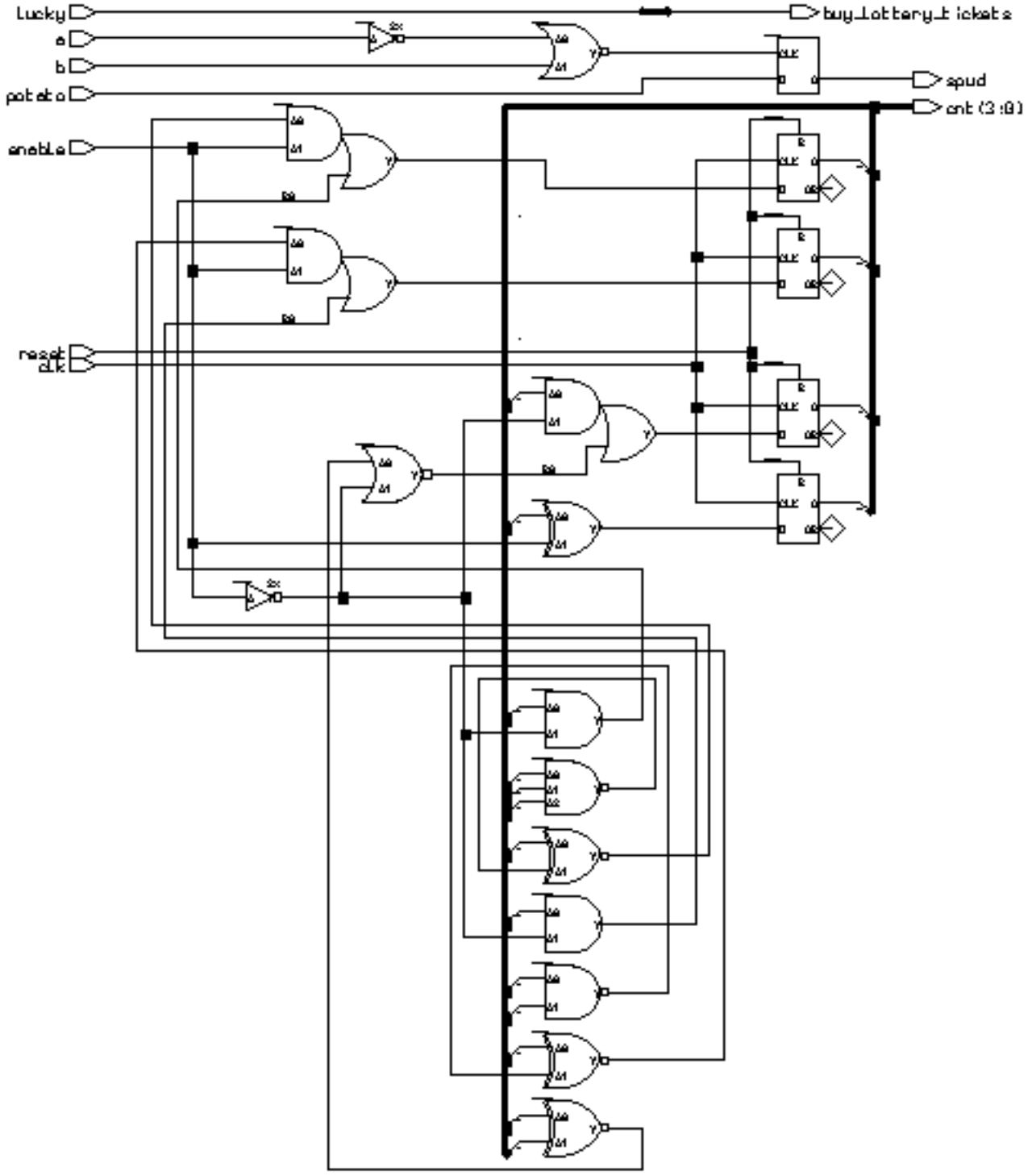
```
--enabled latch
IF (a = ‘1’ AND b = ‘0’) THEN
  spud <= potato;
END IF;
```

```
--a very simple “gate”
IF (lucky = ‘1’) THEN
  buy_lottery_tickets <= ‘1’;
ELSE
  buy_lottery_tickets <= ‘0’;
END IF;
```

```
--a edge triggered 4-bit counter with enable
--and asynchronous reset
IF (reset = ‘1’) THEN
  cnt <= “0000”;
  ELSIF (clk’EVENT AND clk = ‘1’) THEN
    IF enable = ‘1’ THEN
      cnt <= cnt + 1;
    END IF ;
END IF;
```

A Hint: Only `IF.....` needs `END IF`

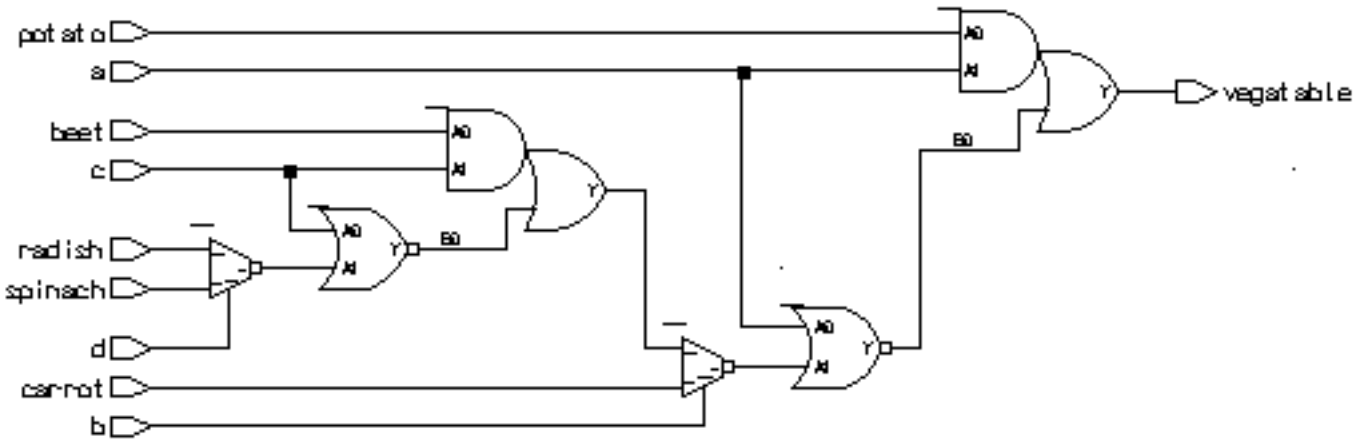
Synthesized example from previous page



IF Implies Priority

The if statement implies a priority in how signals are assigned to the logic synthesized. See the code segment below and the synthesized gates.

```
ARCHITECTURE tuesday OF example IS
  BEGIN
   wow: PROCESS (a, b, c, d, potato, carrot, beet, spinach, radish)
   BEGIN
   IF (a = '1') THEN
     vegatable  <= potato;
   ELSIF (b = '1') THEN
     vegatable <= carrot;
   ELSIF (c = '1') THEN
     vegatable <= beet;
   ELSIF (d = '1') THEN
     vegatable <= spinach;
   ELSE
       vegatable <= radish;
   END IF;
  END PROCESS wow;
END ARCHITECTURE tuesday;
```



what are the delays for each path?

Note how signal with the smallest gate delay through the logic was the first one listed. You can use such behavior to your advantage. Note that use of excessively nested **IF** statements can yield logic with lots of gate delay.

Beyond about four levels of **IF** statement, the **CASE** statement will typically yield a faster implementation of the circuit.

Area and delay of nested IF statement

We can put reporting statements in our synthesis script to tell us the number of gate equivalents and the delays through all the paths in the circuit. For this example, we included the two statements:

```
report_area -cell area_report.txt
report_delay  -show_nets  delay_report.txt
```

In *area_report.txt*, we see:

```
****************************************************
Cell: example    View: tuesday    Library: work
****************************************************
  Cell        Library  References        Total Area
  ao21       ami05_typ     2 x      1      2 gates
  mux21      ami05_typ     2 x      2      4 gates
  nor02      ami05_typ     2 x      1      2 gates

 Number of gates :                        8
```

The *delay_report.txt* has the delay information:

```
             Critical Path Report
Critical path #1   spinach to   vegatable  3.42ns
Critical path #2   radish  to   vegatable  3.41ns
Critical path #3   d       to   vegatable  3.31ns
Critical path #4   c       to   vegatable  2.88ns
Critical path #5   c       to   vegatable  2.57ns
Critical path #6   beet    to   vegatable  2.48ns
Critical path #7   carrot  to   vegatable  1.63ns
Critical path #8   b       to   vegatable  1.52ns
Critical path #9   a       to   vegatable  1.08ns
Critical path #10  a       to   vegatable  1.46ns
```

If implies priority (cont.)

The order in which the IF's conditional statement are evaluated also makes a difference in how the outputs value is assigned. For example, the first check is for (a = '1'). If this statement evaluates true, the output vegetable is assigned "potato" for any input combination where a= '1'.

If the first check fails, the possibilities narrow. If the second check (b= '1') is true, then any combination where a is '0' an b is '1' will assign carrot to vegetable.

If all prior checks fail, an ending ELSE catches all other possibilities.

Relational Operators

The IF statement uses relational operators extensively.

Relational operators return Boolean values (true, false) as their result.

OperatorOperation

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

The expression for signal assignment and less than or equal are the same. They are distinguished by the usage context.

CASE Statement

Controls execution of one or more sequential statements.

Format:

```
CASE expression IS
  WHEN  expression_value0 => sequential_stmt;
  WHEN  expression_value1 => sequential_stmt;
END CASE;
```

Example:

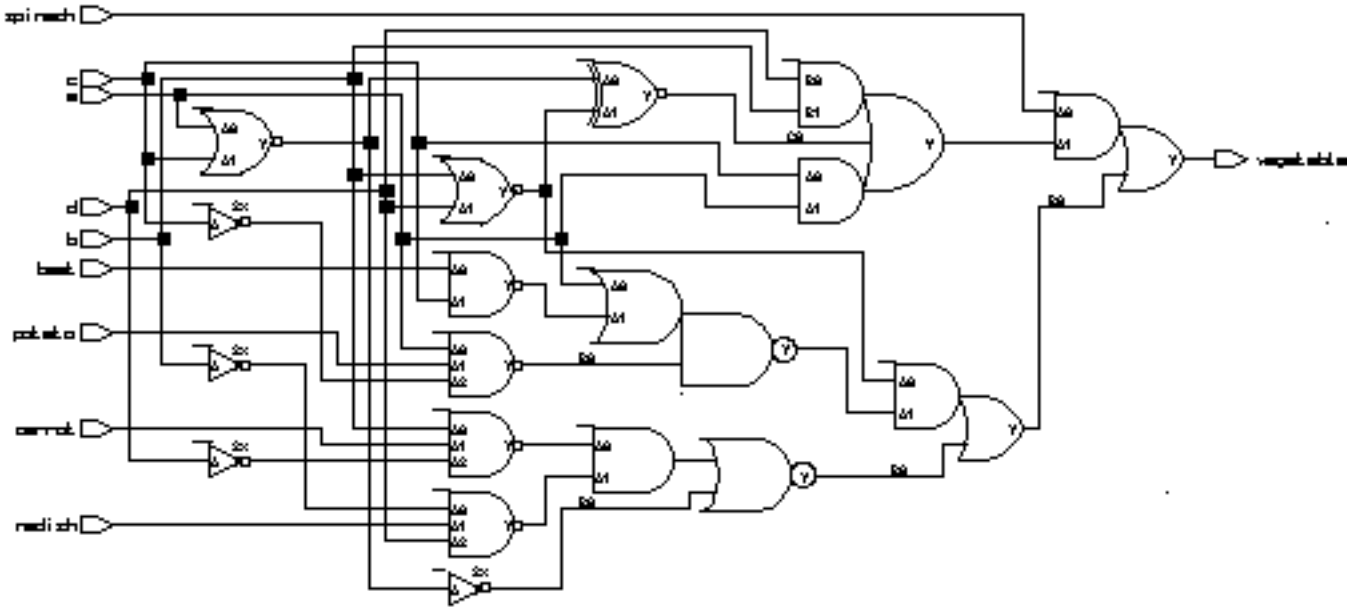
```
--a four to one mux
mux: PROCESS (sel, a, b, c, d)
BEGIN
  CASE sel IS
     WHEN “00”  => out <= a;
     WHEN “01”  => out <= b;
     WHEN “10”  => out <= c;
     WHEN “11”  => out <= d;
     WHEN OTHERS => out <= ‘X’;
  END CASE ;
END PROCESS mux;
```

Either every possible value of *expression_value* must be enumerated, or the last choice must contain an OTHERS clause.

CASE Implies equal priority

The **CASE** statement implies equal priority to how the signals are assigned to the circuit. For example, we will repeat the previous **IF** example using **CASE**. To do so, we combine the selection signals into a bus and make the output selection on the bus value as shown below.

```
ARCHITECTURE tuesday OF example IS
  SIGNAL select_bus : STD_LOGIC_VECTOR(3 DOWNTO 0);
  BEGIN
  select_bus <= (d & c & b & a); --make the select bus
   wow: PROCESS (select_bus, potato, carrot, beet, spinach, radish)
   BEGIN
     CASE select_bus  IS
       WHEN "0001" =>  vegatable <= potato;
       WHEN "0010" =>  vegatable <= carrot;
       WHEN "0100" =>  vegatable <= beet;
       WHEN "1000" =>  vegatable <= radish;
       WHEN OTHERS =>  vegatable <= spinach;
       END CASE;
  END PROCESS wow;
END ARCHITECTURE tuesday;
```



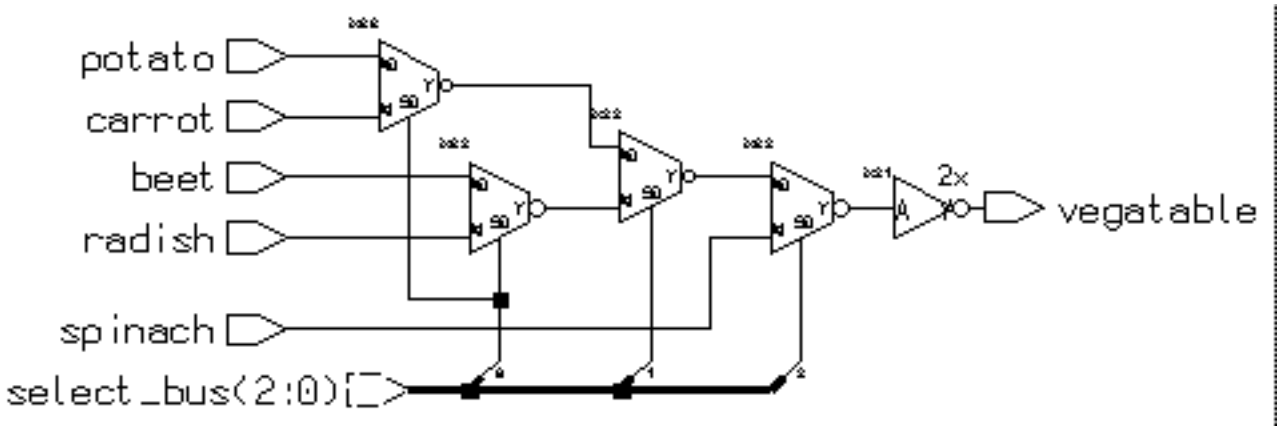
With the exception of spinach, the number of gate delays from each signal input to output is four. The gate delays in the **IF** example varied from 1 to 8 gate delays. However, this function for CASE could be coded better.

Using CASE more effectively

In the previous example, there were 5 choices to choose from. We can encode this more fully by using 3 bits. What we are creating now is a mux. Lets see how this example can be coded more efficiently:

```
ARCHITECTURE tuesday OF example IS
  BEGIN
   wow: PROCESS (select_bus, potato, carrot, beet, spinach, radish)
   BEGIN
     CASE select_bus  IS
       WHEN "000" =>  vegatable <= potato;
       WHEN "001" =>  vegatable <= carrot;
       WHEN "010" =>  vegatable <= beet;
       WHEN "011" =>  vegatable <= radish;
       WHEN "100" =>  vegatable <= spinach;
       WHEN OTHERS => vegatable <= 'X';
       END CASE;
  END PROCESS wow;
END ARCHITECTURE tuesday;
```

The synthesized circuit looks like this:



This encoding of the desired function is much cleaner, faster and smaller. Its seldom you get all three, so take it when you can. Examining the area and delay numbers between this and the **IF** implementation shows the superiority of **CASE** for this situation.

Be careful however, sometimes **CASE** may loose depending upon the circumstances! Blanket statements about synthesis results with different constructs should not be made. Examine each situation individually, and **THINK**!

Delay and area report: efficient CASE example

From area_report.txt:

```
*******************************************************
Cell: example    View: tuesday    Library: work
*******************************************************
    Cell        Library   References        Total Area
    inv02      ami05_typ     1 x      1      1 gates
    mux21       ami05_typ     4 x      2      8 gates

Total accumulated area :
Number of gates: 8
```

From delay_report.txt

```
            Critical Path Report
Critical path #1, potato         to  vegatable    1.83
Critical path #2, beet           to  vegatable    1.83
Critical path #3, carrot         to  vegatable    1.82
Critical path #4, radish         to  vegatable    1.81
Critical path #5, select_bus(0)  to  vegatable    1.72
Critical path #6, select_bus(0)  to  vegatable    1.72
Critical path #7, select_bus(1)  to  vegatable    1.19
Critical path #8, spinach        to  vegatable    0.74
Critical path #9, select_bus(2)  to  vegatable    0.64
```

The comparison between **IF** and **CASE** for this example:

```
IF:    area 8 gates,  delay 3.42ns (worst path)
CASE:  area 8 gates,  delay 1.83ns (worst path)
```

Use of OTHERS in MUXes

In the former example, the **OTHERS** clause assigned the output value of ‘X’ for inputs other than those explicitly stated. There are two main reasons for the use of ‘X’.

Simulation and debugging

Remember that we are using the 9 level logic type STD_LOGIC_1164. This type specifies that a signal can take on a “real world” set of values; 0,1,H,L,Z,X,W,U,-. All these values are included so that we simulate the behavior or “real” circuits such as resistive pullups and pulldowns, tri-state buffers and even initialized logic. An example of an uninitialized cell would be a flip flop output just after power is applied. Its output is considered unknown or ‘U’ by the simulator while if its setup or hold time is violated, the flip flop’s output becomes unknown or ‘X’ immediately after the clock edge.

If a setup violation occurs during the simulation of a circuit, a flip flop’s output will go ‘X’. If the flip flop’s output forms the select input to a mux, what input signal will be propagated to the output? In other words, if the *select_bus* signal becomes “0X1”, what input signal value will *vegatable* take on. This is known in polite circles as the *X propagation issue*.

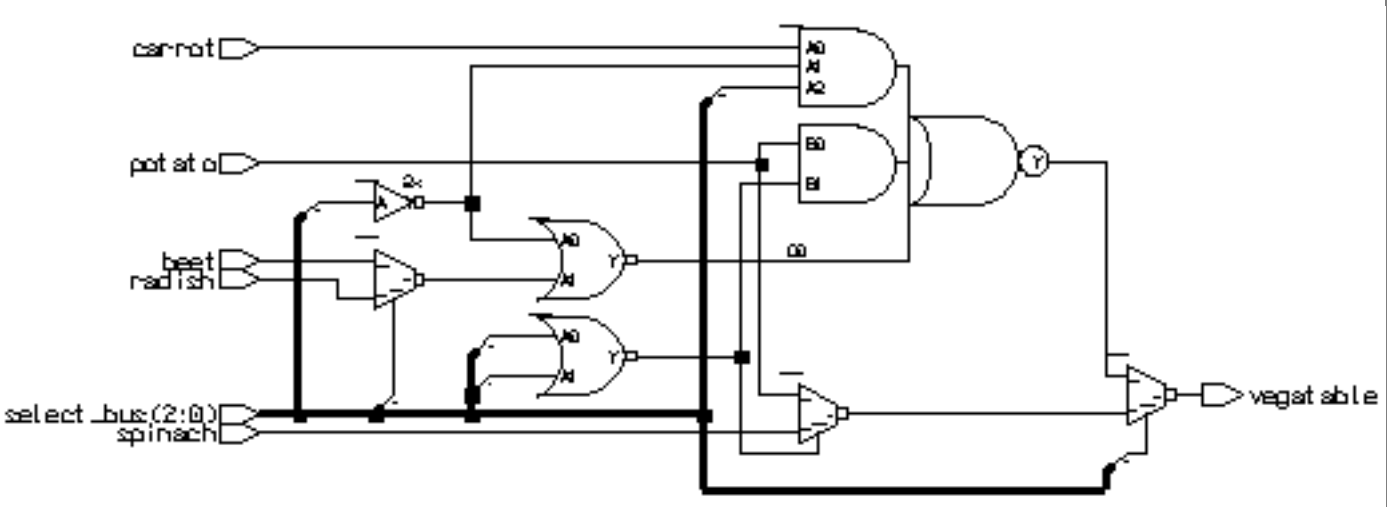
If we chose another valid input for the OTHERS clause, the error (‘X’ output from a flip flop) in the simulation will not be propagated to downstream logic. It will stop or be lost at the mux input because the select_bus value “0X1” maps to a valid input. At the next clock cycle the flip flop may transition to a valid state, the simulation will continue and the error will go unnoticed. We would rather have the ‘X’ propagate thorough the logic and “blow up” the simulation so we can catch the error.

The code below is valid and would **not** propagate the ‘X’ condition. It also represents an “overly specified” circuit. It is overly specified in the sense that surely all the possible values of *select_bus* should not map to *potato*. Giving some degree of freedom actually produces a smaller gate realization.

Use of OTHERS (cont.)

```
--overly specified mux
CASE select_bus  IS
  WHEN "000" =>  vegatable <= potato;
  WHEN "001" =>  vegatable <= carrot;
  WHEN "010" =>  vegatable <= beet;
  WHEN "011" =>  vegatable <= radish;
  WHEN "100" =>  vegatable <= spinach;
  -- output potato for all other cases
  WHEN OTHERS => vegatable <= potato;
END CASE;
```

If we synthesize this circuit we get the following:



The gate realization of this overly specified mux is obviously a little messy. This also seen in the reports from synthesis.

The worst case path from the delay_report.txt gives us:

```
Critical path #1, beet to vegatable, 2.17ns
```

The gate count from area_report.txt gives us:

```
Number of gates: 11
```

This less than optimal solution leads to the second reason for the use of ‘X’ here; logic minimization.

Use of OTHERS (cont.)

Logic minimization

The synthesis tool must choose from a library of cells to create the circuit described by the HDL code. In the case of using the statement:

```
WHEN OTHERS => vegatable <= 'X';
```

What does the synthesis tool do? There is no gate that can produce a 'X' output except when malfunctioning. How can it make a set of gates to produce an 'X' output? The answer is that it doesn't.

The *synthesizer* treats the 'X' in this case as a *don't care*. This is just like the don't care in a Karnough map. It allow the synthesis to optimize (reduce) the gate count if possible. The simulator treats the X as a value to be propagated in simulation if an error happens.

In fact, we can use another value in the mux statement; the don't care value, '-'. So we could have coded the mux as follows:

```
--don't do this!
CASE select_bus  IS
  WHEN "000" =>  vegatable <= potato;
  WHEN "001" =>  vegatable <= carrot;
  WHEN "010" =>  vegatable <= beet;
  WHEN "011" =>  vegatable <= radish;
  WHEN "100" =>  vegatable <= spinach;
  WHEN OTHERS => vegatable <= '-';
END CASE;
```

This would allow the same optimizations as the 'X' for the OTHERS case but the behavior of the simulation in the case of a '-' being propagated could be library and simulator dependent. This would **NOT** be a be a good way to code a mux even though the synthesized circuit is identical to the mux with the OTHERS statement using 'X'.

Use of OTHERS (conclusion)

By using the statement:

```
WHEN OTHERS => vegatable <= 'X';
```

the synthesizer can create a small, fast circuit that behaves properly.

One basic premise of how we want to code our designs is that we want the simulation of our code to act exactly as the gate implementation. If a real mux had a metastable (think ‘X’) input, the output would be metastable (X), not some valid (0 or 1) state.

The proper use of the don’t care operator is found in creating complex combinatorial logic and in state machine state assignments. In that context, the don’t care operator really shines. We will see some examples of this soon.

Loops

Sequences of statements that are executed repeatedly.

Types of loops:

- **For (most common usage)**
- **While**
- **Loop with exit construct (we skip this)**

General Format:

```
[loop_label:]
iteration_scheme  --FOR, WHILE
LOOP
  --sequence_of_statements;
END LOOP[loop_label];
```

For Loop

Statements are executed once for each value in the loop parameter's range

Loop parameter is implicitly declared and may not be modified from within loop or used outside loop.

Format:

```
[label:] FOR loop_parameter IN discrete_range
LOOP
--sequential_statements

END LOOP[label];
```

Example:

```
PROCESS (ray_in)
BEGIN
  --connect wires in a two busses
  label: FOR index IN 0 TO 7
  LOOP
    ray_out(index) <= ray_in(index);
  END LOOP label;
END PROCESS;
```

While Loop

Execution of statements within loop is controlled by Boolean condition.

Condition is evaluated before each repetition of loop.

Format:

```
WHILE boolean_expression
LOOP
--sequential_expressions
END LOOP;
```

Example:

```
p1:
PROCESS (ray_in)
  VARIABLE index : integer := 0;
BEGIN
  from_in_to_out:
  WHILE index < 8
  LOOP
    ray_out(index) <= ray_in(index);
    index := index + 1;
  END LOOP from_in_to_out;
END PROCESS p1;
```

Attributes

Attributes specify “extra” information about some aspect of a VHDL model.

There are a number of predefined attributes provide a way to query arrays, bit, and bit vectors.

Additional attributes may be defined by the user.

Format:

```
object_name’attribute_designator
```

The “ ‘ ” is referred to as “tick”.

Example:

```
ELSIF (clk’EVENT AND clk = ‘1’) THEN
```

Predefined Signal Attributes

signal'EVENT - returns value "TRUE" or "FALSE" if event occurred in present delta time period.

signal'ACTIVE - returns value "TRUE" or "FALSE" if activity occurred in present delta time period.

signal'STABLE - returns a signal value "TRUE" or "FALSE" based on event in (t) time units.

signal'QUIET - returns a signal value "TRUE" or "FALSE" based on activity in (t) time units.

signal'TRANSACTION - returns an event whenever there is activity on the signal.

signal'DELAYED(t) - returns a signal delayed (t) time units.

signal'LAST_EVENT - returns amount of time since last event.

signal'LAST_ACTIVE - returns amount of time since last activity.

signal'LAST_VALUE - returns value equal to previous value.

Using Attributes

Rising clock edge:

```
clk'EVENT and clk = '1'
```

OR:

```
NOT clk'STABLE AND clk ='1'
```

Falling clock edge:

```
clk'EVENT AND clk = '0'
```

Checking for too short pulse width:

```
ASSERT (reset'LAST_EVENT >= 3ns)
  REPORT "reset pulse too short!";
```

Checking stability of a signal:

```
signal'STABLE(10ns)
```

Generic Clause

Generics may be used for readability, maintenance and configuration.

They allow a component to be customized by creating a parameter to be passed on to the architecture.

Format:

```
GENERIC (generic_name:type[:= default_value]);
```

If default_value is missing, it must be present when the component is instantiated.

Example:

```
ENTITY half_adder IS
  GENERIC(
        tpd_result : delay := 4ns;
        tpd_carry  : delay := 3ns);
  PORT(
        x  IN  :  std_logic;
        y  IN  :  std_logic;
        z  OUT :  std_ulogic);
END nand_gate;

ARCHITECTURE dataflow OF half_adder
  BEGIN
    I result <= x XOR y AFTER tpd_result;
      carry  <= x AND y AFTER tpd_carry;
  END dataflow;
```

Inferring Storage Elements

In our designs, we usually use flip-flops as our storage elements. Sometimes we use latches, but not often. Latches are smaller in size, but create special, often difficult situations for testing and static timing analysis.

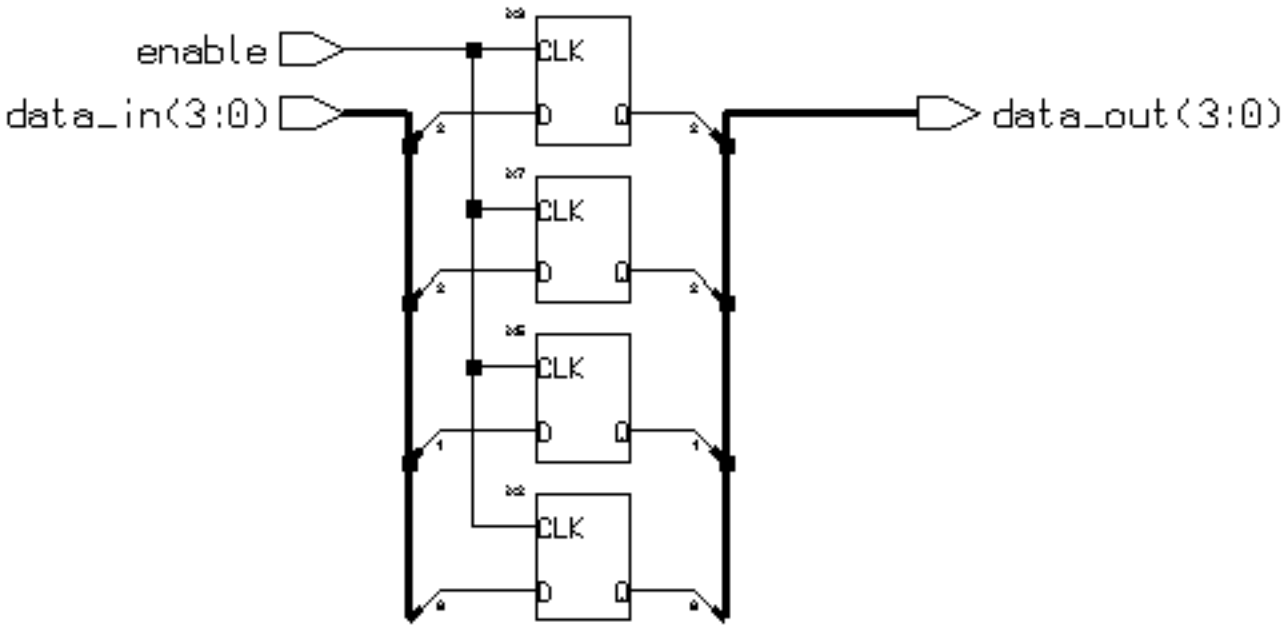
Latches are inferred in VHDL by using the IF statement without its matching ELSE. This causes the synthesis to make the logical decision to “hold” the value of a signal when not told to do anything else with it.

The inferred latch is a transparent latch. That is, for as long as enable is high, the q output “sees” the d input transparently.

```
--infer 4-bit wide latch
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_vector_arith.ALL;


ENTITY storage  IS
 PORT (
       data_in  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
       data_out : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       enable   : IN  STD_LOGIC);
END storage;

ARCHITECTURE wed OF storage IS
  BEGIN
  infer_latch:
  PROCESS (enable, data_in)
    BEGIN
      IF enable = ‘1’ THEN
        data_out <= data_in;
      END IF;  --look ma, no else!
  END PROCESS infer_latch;
END ARCHITECTURE wed;
```

When synthesized, we see the following structure:

Latch Inference



In our library, the enable is shown as going to the “CLK” input of the latch. This is misleading as the input should properly be called “EN” or something like that. If I find the time maybe I’ll change these someday.

The small size of the latches is reflected in the area report:

```
Cell      Library      References      Total Area
latch    ami05_typ        4 x      2     10 gates

Number of gates :                       10
```

This is of course relative to the size of a 2-input NAND gate. In other words, the area of each latch is about the same as 2, 2-input NAND gates!

When we synthesized, the transcript told of the impending latch inference:

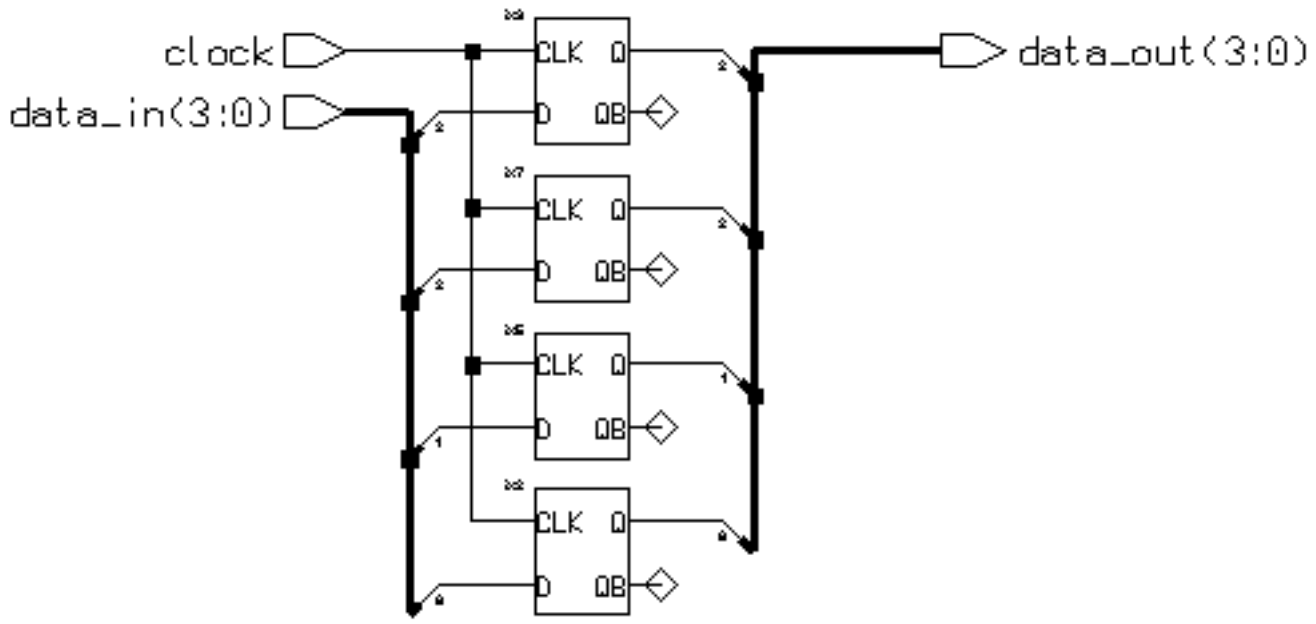
```
-- Compiling root entity storage(wed)
"/nfs/guille/u1/t/traylor/ece574/src/storage.vhd",line 8: Warning,
data_out is not always assigned. latches could be needed.
```

Always watch tool transcripts. They can be very informative. Sometime they can save your bacon.

Inferring D-type Flip Flops

Usually, we want to infer D-type, edge triggered flip flops. Here's how.

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

ENTITY storage  IS
 PORT (
       data_in  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
       data_out : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       clock    : IN  STD_LOGIC);
END storage;

ARCHITECTURE wed OF storage IS
  BEGIN
  infer_dff:
  PROCESS (clock, data_in)
    BEGIN
      IF (clock'EVENT AND clock = '1') THEN
        data_out <= data_in;
      END IF;  --look ma, still no else!.... what gives?
  END PROCESS infer_dff;
END ARCHITECTURE wed;
```



Sometime back we stated that IF with ELSE infers a latch. Well... that is usually true. Here is an exception. The line:

```
IF (clock'EVENT AND clock = '1') THEN
```

is special to the synthesis tool. The conditional statement for the IF uses the attribute which looks for a change in the signal *clock* (`clock'EVENT`). This is ANDed with the condition that *clock* is now '1' (`AND clock = '1'`). The conditional is looking for a rising edge of the signal *clock*.

Therefore, if there is a rising edge, the statement under the IF will be executed and at no other time. So when the clock rises, data_out will get the value present at data_in. Since a D flip-flop is the only cell that can satisfy this condition and can hold the value once it is acquired it is used to implement the circuit. The conditional `(clock'EVENT AND clock = '1')`really forms the recipe for a D-type rising edge flip flop.

A ELSE clause could be added to the IF statement that explicitly tells the old value to be held. This is not at all harmful, but is redundant and is ignored by the synthesis tool. An example of this is shown below:

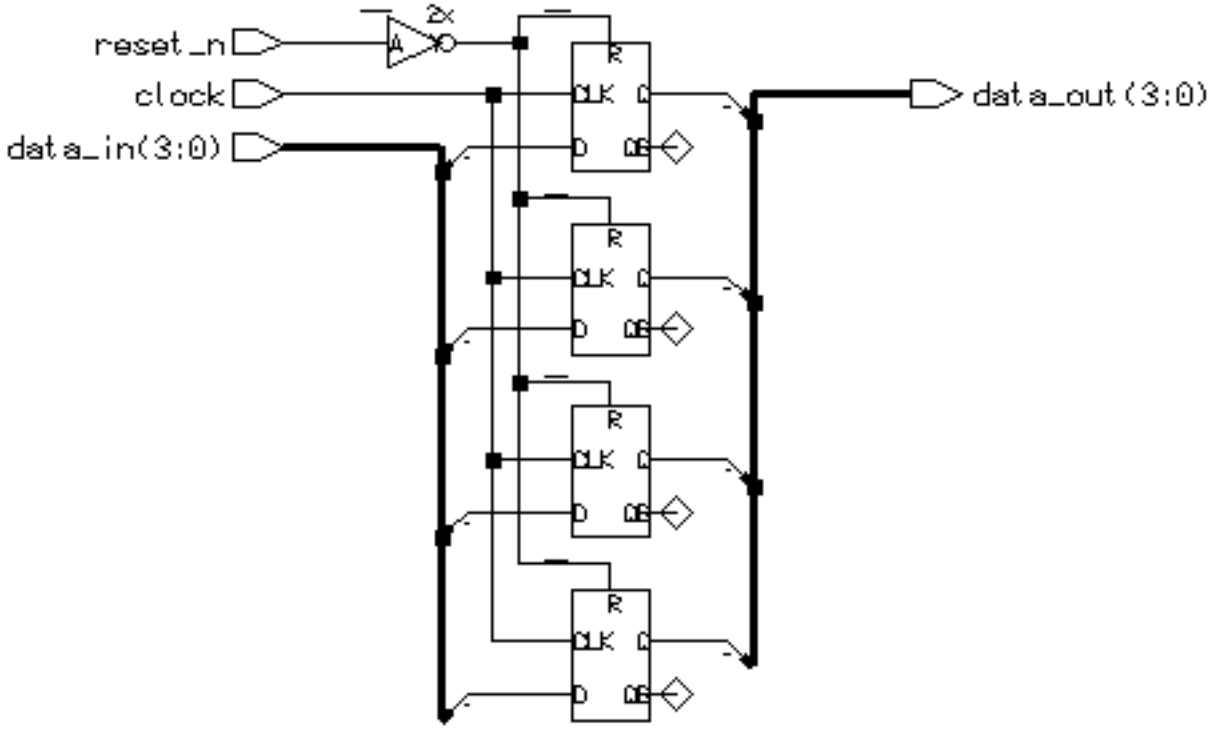
```
infer_dff:
PROCESS (clock, data_in)
  BEGIN
    IF (clock'EVENT AND clock = '1') THEN
      data_out <= data_in;  --get new value
    ELSE
      data_out <= data_out; --hold old value...UNNECESSARY
    END IF;
END PROCESS infer_dff;
```

Adding an Asynchronous Reset

We almost never want a flip flop without a reset. Without a reset, how can the simulator determine initial state? It cannot. It is very rare to find flip-flops with out a reset. Here is how to code a flip flop with a asynchronous reset:

```
ARCHITECTURE wed OF storage IS
  BEGIN
  infer_dff:
  PROCESS (reset_n, clock, data_in)
    BEGIN
      IF (reset_n = ‘0’) THEN
        data_out <= “0000”;  --aysnc reset
      ELSIF (clock’EVENT AND clock = ‘1’) THEN
        data_out <= data_in;
      END IF;
  END PROCESS infer_dff;
END ARCHITECTURE wed;
```

When synthesized, we get:



How big is a flip flop/latch?

From the area_report.txt file we see:

```
Cell      Library   References     Total Area
dffr     ami05_typ     4 x      6     24 gates
inv02    ami05_typ     1 x      1      1 gates
Number of gates :      24
```

This looks a little fishy. 24 + 1 = 24? At any rate, (assuming round off error) the flip flops are roughly 6 gates a piece.

So to summarize the relative sizes of latches and flip flops
:

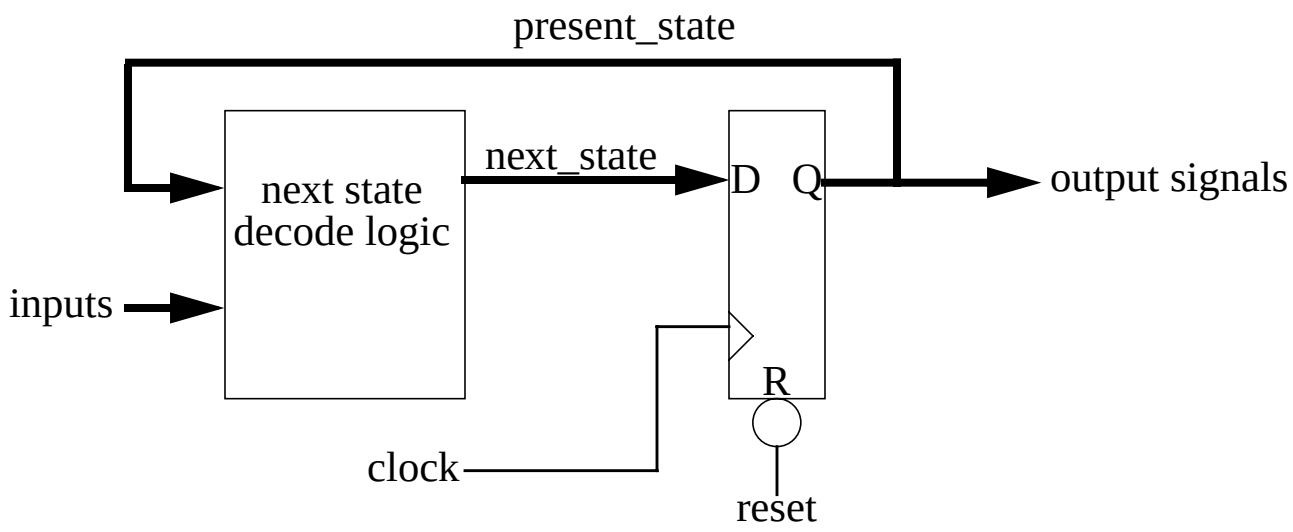
CASE	**CELL**	**SIZE**
latch no reset	latch	2 gates
latch with reset	latchr	3 gates
flip flop with no reset	dff	5 gates
flip flop with reset	dffr	6 gates

These numbers are valid only for our library. Other libraries will vary. However, the relative sizes are consistent with most any CMOS library.

State Machines in VHDL

Implementing state machines in VHDL is fun and easy provided you stick to some fairly well established forms. These styles for state machine coding given here is not intended to be especially clever. They are intended to be portable, easily understandable, clean, and give consistent results with almost any synthesis tool.

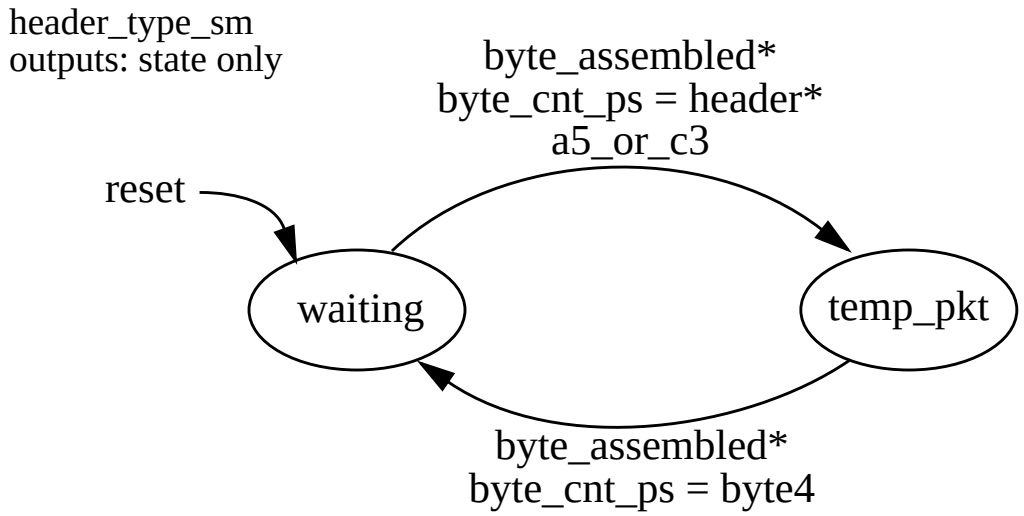
The format for coding state machines follows the general structure for a state machine. Lets look at the basic Moore machine structure.



The Moore state machine consists of two basic blocks, next state decode (or steering) logic, and some state storage usually (always for our case) D-type flip flops. Inputs are applied to the next state decode block along with the present state to create the next state output. The flip flops simply hold the value of the present state. In the example above, the only output signals are the outputs of the state flip flops. Alternatively, the flip flop outputs could be decoded to create the output signals.

For a first example we will look at the state machine in TAS which holds the state of what type of header is being received, *waiting* or *temp_pkt*. First we look at the state diagram.

State Diagram for header_type_sm



All your state machines should be documented in roughly this fashion. The name of the process holding the code for the state machine is the name of the state machine. In this case it is *header_type_sm.*

Every state machine has an arc from “reset”. This indicates what state the state machine goes to when a reset is applied. The diagram is worthless without knowing what the initial state is.

Each state in this example is given a name. In this case we are using a type for the states that is an enumerated state type. We will see what this means to the code later. For now, it provides a easy way to understand and to talk about what and how the state machine works.

Each possible transition between states is shown via an arc with the condition for the transition to occur shown. The condition need not be in VHDL syntax but should be understandable to the reader. Typically (highly recommended) logic expressions are given with active high assertion assumed.

It should be understood that all transitions occur on the clock edge.

Outputs from the state machine should be listed. The only outputs from this state machine are its present state. Most likely, some other state machine is watching this one’s state to determine its next state.

State Machines (cont.)

To use the enumerated state types in our example, we need to declare what they are. This would be done in the declarative area of the architecture as shown.

```
ARCHITECTURE beh OF ctrl_blk_50m IS
--declare signals and enumerated types for state machines

--further down we see.......

TYPE header_type_type IS (waiting, temp_pkt);
SIGNAL header_type_ps, header_type_ns: header_type_type;

--bla, bla, bla.....

BEGIN
```

The **TYPE** declaration states that we have a type called *header_type_type* and that the two only states for this type are *waiting* and *temp_pkt*. Having done this we can declare two signals for our present state and next state vectors called *header_type_ps* and *header_type_ns*. Note that the vectors get their names from the state machine they are apart of plus the *_ps* or *_ns* to distinguish present or next state vectors.

This style of state machine state coding is called enumerated state encoding. It is flexible in the sense that the synthesis tool is left to make the decision about how to assign a bit pattern to each state. More about this later.

Now using these state declarations, lets make the process that creates the state machine.

State Machine Process Body

Below we see the body of the process that creates the state machine.

```
header_type_sm:
 PROCESS (clk_50, reset_n, a5_or_c3, byte_assembled, byte_cnt_ps,
header_type_ps, header_type_ns)
 BEGIN
   --clocked part
   IF (reset_n = '0') THEN
     header_type_ps <= waiting;
   ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
     header_type_ps <= header_type_ns;
   END IF;

   --combinatorial part
   CASE header_type_ps IS
     WHEN waiting =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = header) AND
(a5_or_c3 = '1') THEN

        header_type_ns <= temp_pkt;
      ELSE
        header_type_ns <= waiting;
      END IF ;
     WHEN temp_pkt =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = byte4) THEN
        header_type_ns <= waiting;
      ELSE
        header_type_ns <= temp_pkt;
      END IF ;
   END CASE;
 END PROCESS header_type_sm;
```

First we see that the process label is consistent with the documentation and the signal names we have assigned. Also all the signals that may be read are listed in the process sensitivity list for the process.

```
header_type_sm:
 PROCESS (clk_50, reset_n, a5_or_c3, byte_assembled, byte_cnt_ps,
header_type_ps, header_type_ns)
```

Next the flip flops are created to hold the present state. This is what is commonly called the clocked or synchronous part since it is controlled by the clock.

State Machine Process Body (synchronous part)

```
--clocked part
IF (reset_n = '0') THEN
  header_type_ps <= waiting;
ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
  header_type_ps <= header_type_ns;
END IF;
```

Here we see an active low asynchronous reset that forces the present state to become the state called *waiting*. It does so without regard to the clock. That is why it is called an asynchronous reset.

Following the reset clause, the "clock tick event" clause identifies the following statements to be generating flip flops. The flip flops it creates are rising edge sensitive and cause the signal *header_type_ps* to take on the value of *header_type_ns* at the rising clock edge.

This concludes the clocked part of the process. We have created the necessary state flip flops and connected the D inputs to header_type_ns and the Q outputs to header_type_ps.

Now we will create the next state steering logic. It consists only of gates, i.e.; combinatorial logic. This part of the process is thus commonly called the combinatorial part of the process.

State Machine Process Body (combinatorial part)

```
 --combinatorial part
   CASE header_type_ps IS
     WHEN waiting =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = header) AND
         (a5_or_c3 = '1') THEN
        header_type_ns <= temp_pkt;
      ELSE
        header_type_ns <= waiting;
      END IF ;
     WHEN temp_pkt =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = byte4) THEN
        header_type_ns <= waiting;
      ELSE
        header_type_ns <= temp_pkt;
      END IF ;
   END CASE;
```

To clearly make the next state logic, a structure is created where **IF** statements are tucked under each distinct **CASE** state possibility. Each **CASE** possibility is a state in the state machine. Given a present state the **IF** statements determine from the input conditions what the next state is to be.

To further illustrate this:

The **CASE** statement enumerates each possible present state:

```
CASE header_type_ps  IS
     WHEN waiting =>
     --bla, bla, bla
     WHEN temp_pkt =>
     --bla, bla, bla
END CASE
```

In any given state the IF determines the input conditions to steer the machine to the next state. For example:

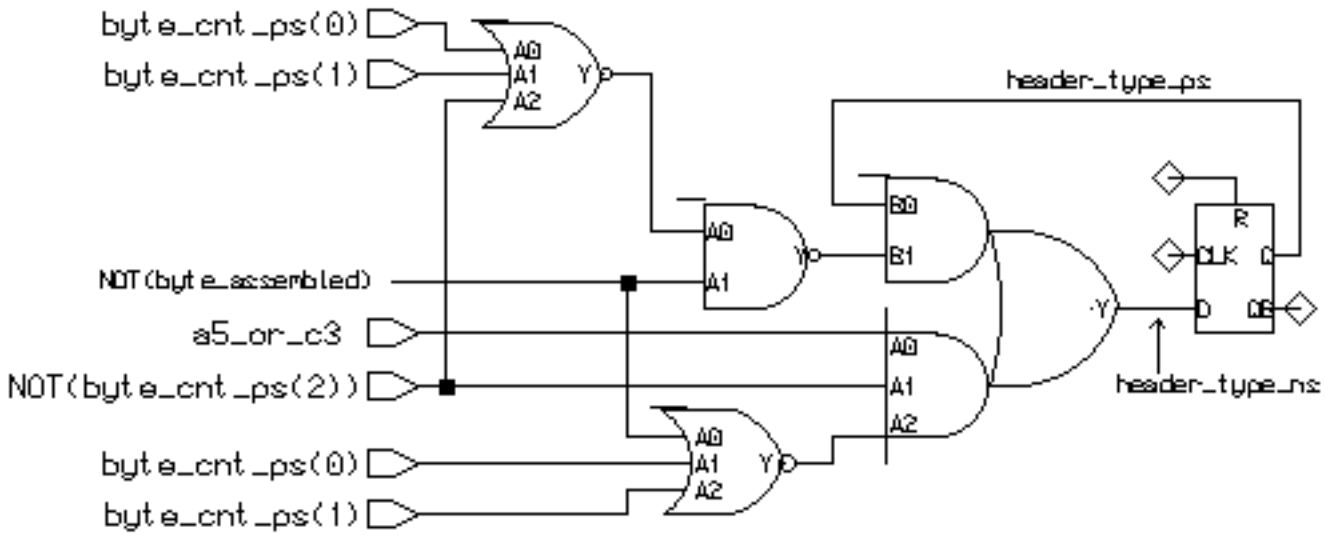
```
WHEN temp_pkt =>
   IF (byte_assembled = '1') AND (byte_cnt_ps = byte4) THEN
       header_type_ns <= waiting;  --go to waiting if IF is true
   ELSE
       header_type_ns <= temp_pkt; --else, stay put
  END IF ;
```

State Machine Synthesis

If we synthesize the state machine we see in the transcript:

```
"/nfs/guille/u1/t/traylor/ece574/src/header.vhd",line 28: Info,
Enumerated type header_type_type with 2 elements encoded as binary.

Encodings for header_type_type values
   value    header_type_type[0]
==============================
   waiting          0
   temp_pkt         1
```

This tells us that the synthesis tool selected the value of ‘0’ to represent the state *waiting*, and the value ‘1’ to represent the state *temp_pkt*. This makes since because we have only two states, thus 0 and 1 can represent them. We would furthermore expect only one flip flop to be needed. So the schematic looks like this: (clock and reset wiring omitted)



You might say, “That’s not the way I would do it.” But for the circuit this state machine was taken from, this was what it considered an optimal realization. Study the tas design file *ctrl_50m.vhd* and you can probably figure out some of the how and why the circuit was built this way.

The next state steering logic can be clearly seen to the left of the state storage (flip flop).

Enumerated Encoding

Using enumerated state encoding allows the synthesis tool to determine the optimal encoding for the state machine. If speed is the primary concern, a state machine could be created using one hot encoding. If minimum gate count is the most important criterion, a binary encoding might be best. For minimum noise or to minimize decoding glitching in outputs, grey coding might be best. Four different ways to encode a 2 bit state machine might be like this:

binary	00, 01, 10, 11
one hot	0001, 0010, 0100, 1000
grey	00, 01, 11, 10
random	01, 00, 11,10

While enumerated encoding is the most flexible and readable, there are cases where we want to create output signals that have no possibility of output glitches. Asynchronous FIFOs and DRAMs in particular.

As an example of a glitching state machine, lets build a two bit counter that has an output which is asserted in states “01” or “10” and is deasserted for states “00” and “11”. We will allow binary encoding of the counter.

Counter State Machine with Decoded Output

The code looks like this:

```
ARCHITECTURE beh OF sm1 IS
TYPE byte_cnt_type IS (cnt1, cnt2, cnt3, cnt4);
SIGNAL byte_cnt_ps, byte_cnt_ns:byte_cnt_type;
BEGIN
byte_cntr:
PROCESS (clk_50, reset_n, enable, byte_cnt_ps, byte_cnt_ns)
BEGIN
  --clocked part
  IF (reset_n = '0') THEN
    byte_cnt_ps <= cnt1;
  ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
    byte_cnt_ps <= byte_cnt_ns;
  END IF;
  --combinatorial part
  decode_out <= '0';  --output signal
  CASE byte_cnt_ps IS
    WHEN cnt1 =>  --output signal takes default value
    IF (enable = '1') THEN
       byte_cnt_ns <= cnt2;
     ELSE
       byte_cnt_ns <= cnt1;
     END IF ;
    WHEN cnt2 =>
    decode_out <= '1'; --output signal assigned
     IF (enable = '1') THEN
       byte_cnt_ns <= cnt3;
     ELSE
       byte_cnt_ns <= cnt2;
     END IF ;
    WHEN cnt3 =>
     decode_out <= '1'; --output signal assigned
     IF (enable = '1') THEN
       byte_cnt_ns <= cnt4;
     ELSE
       byte_cnt_ns <= cnt3;
     END IF ;
    WHEN cnt4 =>  --output signal takes default value
     IF (enable = '1') THEN
       byte_cnt_ns <= cnt1;
     ELSE
       byte_cnt_ns <= cnt4;
     END IF ;
  END CASE;
END PROCESS byte_cntr;
```

Specifying Outputs with Enumerated States

In the proceeding code, we see one way an output other than the present state may be created. At the beginning of the combinatorial part, before the CASE statement, the default assignment for *decode_out* is given.

```
--combinatorial part
decode_out <= '0';  --output signal
CASE byte_cnt_ps IS
```

In this example, the default value for the output *decode_out* is logic zero. The synthesis tool sees that *decode_out* is to be logic zero unless it is redefined to be logic one. In state cnt1, no value is assigned to decode_out.

```
WHEN cnt1 =>  --output signal takes default value
IF (enable = '1') THEN
   byte_cnt_ns <= cnt2;
 ELSE
   byte_cnt_ns <= cnt1;
 END IF ;
```

Thus if the present state is cnt1, decode_out remains zero. However, if the present state is cnt2, the value of decode_out is redefined to be logic one.

```
WHEN cnt2 =>
decode_out <= '1'; --output signal assigned
 IF (enable = '1') THEN
   byte_cnt_ns <= cnt3;
 ELSE
   byte_cnt_ns <= cnt2;
 END IF ;
WHEN cnt3 =>
```

We could have omitted the default assignment before the **CASE** statement and specified the value of *decode_out* in each state. But for state machines with many outputs, this becomes cumbersome and more difficult to see what is going on.

Specifying Outputs with Enumerated States

What happens if we have a state machine with an output, yet do not specify the outputs value in each state. This is similar to the situation of IF without ELSE. Latches are created on the output signal.

If we specify a “off” or default value for each output signal prior to the case statement we will never have the possibility for a latch

State Machine with Decoded Outputs

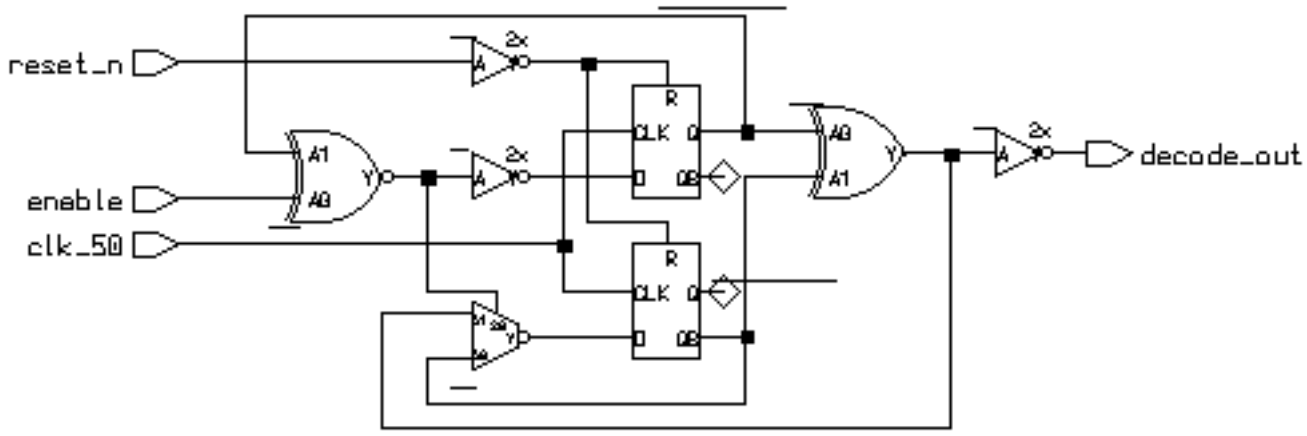
Note the declaration of the enumerated states:

```
TYPE byte_cnt_type IS (cnt1, cnt2, cnt3, cnt4);
SIGNAL byte_cnt_ps, byte_cnt_ns:byte_cnt_type;
```

Typically, (i.e., may change with tool and/or vendor) with binary encoding, the state assignments will occur following a binary counting sequence in the order in which the states are named. i.e., cnt1 = “00”, cnt2 = “01”, etc. Surely enough when the synthesis is done, we see the transcript say:

```
-- Loading entity sm1 into library work
"/nfs/guille/u1/t/traylor/ece574/src/sm1.vhd",line 23: Info,
Enumerated type byte_cnt_type with 4 elements encoded as binary.
Encodings for byte_cnt_type values
    value    byte_cnt_type[1-0]
==================================
     cnt1            00
     cnt2            01
     cnt3            10
     cnt4            11
```

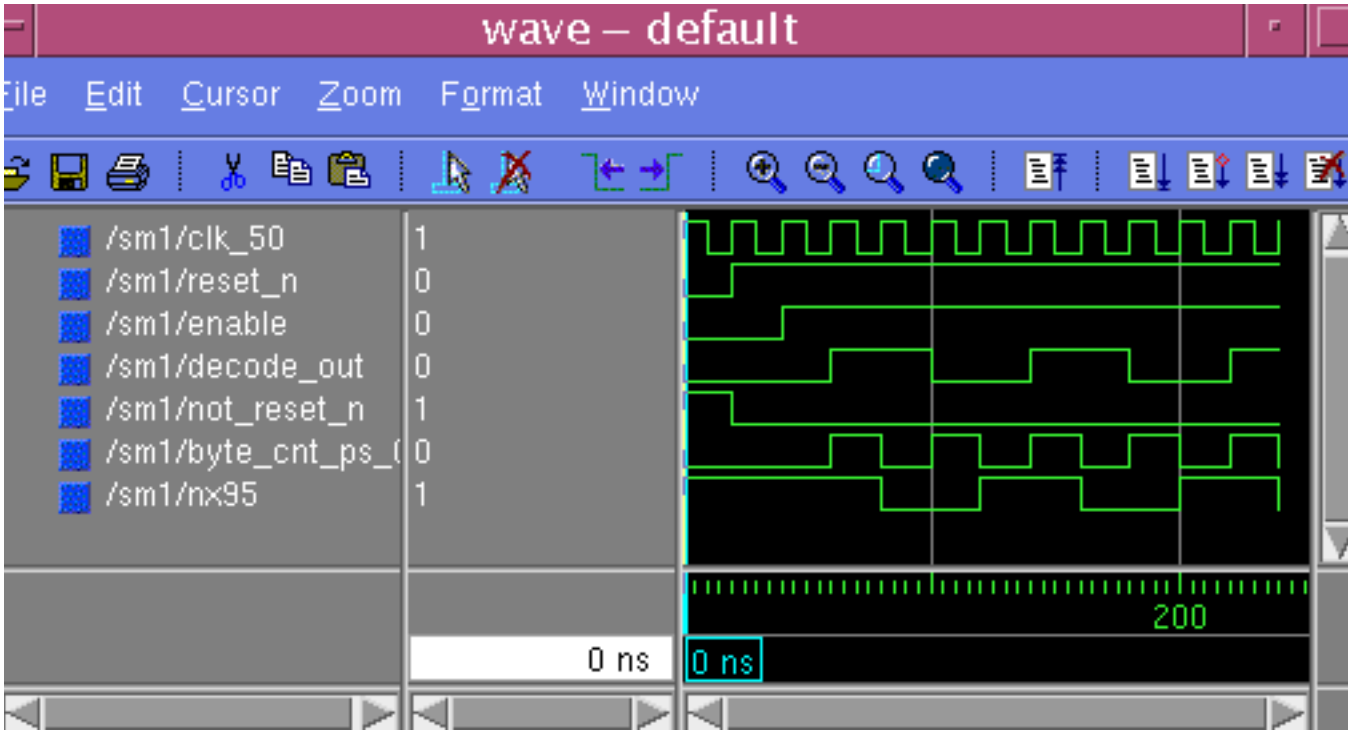
The circuit we get is shown below:



Note the output decoding created by the XOR and NOT gate. If there are unequal delays from the flip flop outputs to the XOR gate, glitches will result at the XOR output. In a “delayless” simulation this would not be seen. In a fabricated chip however, the glitches are almost a certainty.

Glitches

If we simulate this circuit without delays, we see the following. Note that the signal *decode_out* has no glitches.



When we are first creating a circuit, we usually simulate without delays. However, when we synthesize, we can specify that a "sdf" file be created. The sdf (standard delay format) file contains the delay information for the synthesized circuit.

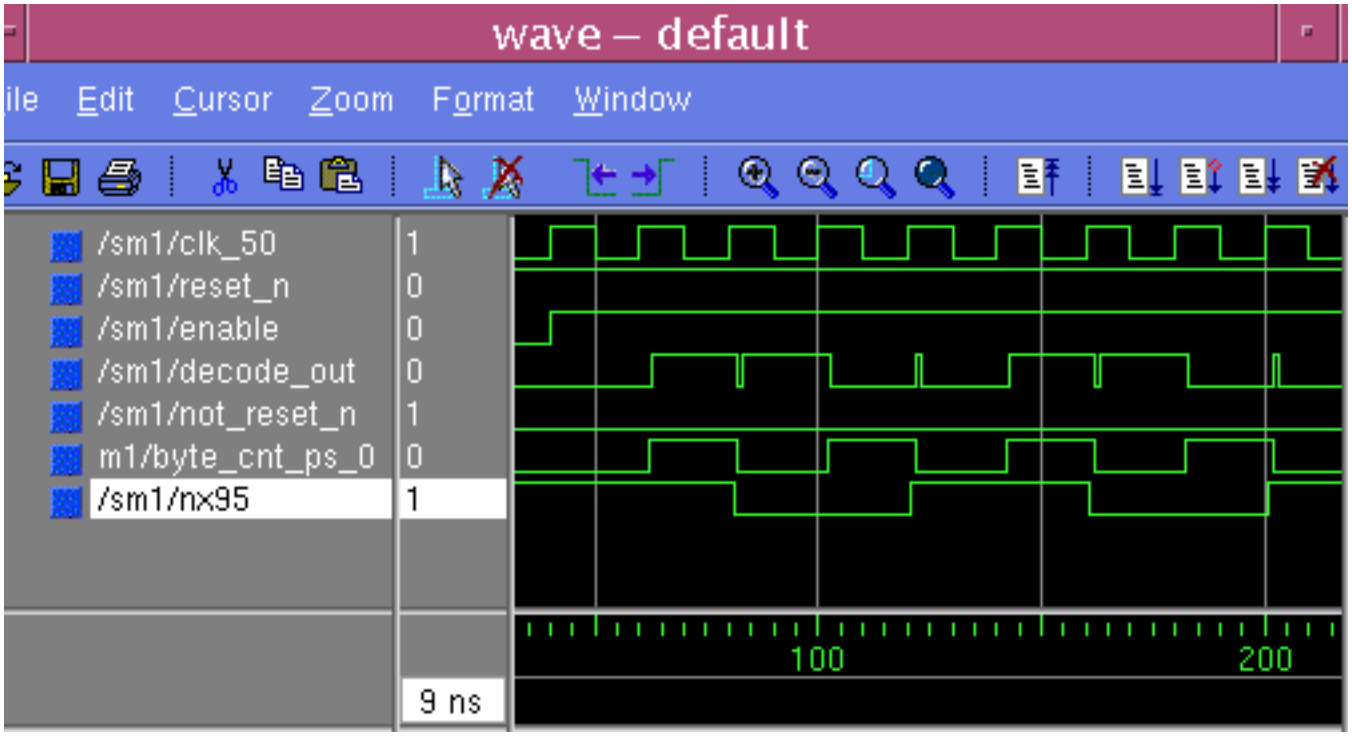
Once the synthesized circuit is created, we can invoke the vsim simulator with the sdf switch used to apply the delays to the circuit. More about this process in the latter part of the class.

So, when we invoke vsim like this:

```
vsim -sdfmax ./sdfout/sm1.sdf sm1
```

delays are added to the circuit. To make the glitch clearer, I added an additional 1ns delay to one of the flip flops by editing the sdf file. The simulation output now looks like this:

State machine output with glitches



So using enumerated types when coding state machines is a clear and flexible coding practice. However,.....the synthesizer may eat your lunch in certain situations! As folks often say, “It depends.”. If you state machine outputs go to another state machine as inputs, the glitches won’t make a bit of difference. The glitches will come only after the clock edge and will be ignored by the flip flop. But, if the outputs go to edge sensitive devices, BEWARE.

So, lets see how we can make outputs that are always clean, without glitches for those special cases.

grey coding, choosing states wisely, following flip flops, explicit states

State Encoding for Glitchless Outputs

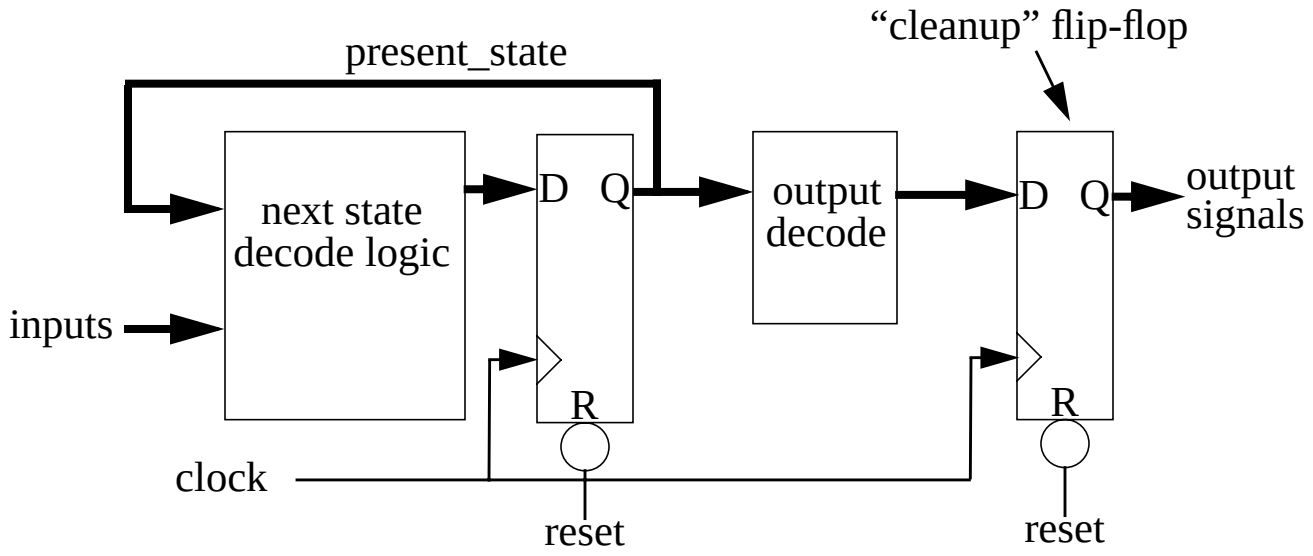
One solution to creating glitchless outputs is to strategically select the state encoding such that glitches cannot occur. This comes down to selecting states such that as the state machine goes through its sequence, the encoded states are adjacent. You can think of this as the situation in a Karnough map where two cells are directly next to each other. One of the easiest way of doing this is by using Grey coding.

If the counter advanced in the sequence “00”, “01”, “11”, “10”, no glitch would be created between states “01” and “11” as these two states are adjacent. Most synthesis tools will do grey encoding for state machines simply by setting a switch in the synthesis script.

Using grey coding in a counter makes an easy way to prevent output decoder glitches. However, in a more general state machine where many sequences may be possible and arcs extend from most states to other states, it becomes very difficult to make sure you have correctly encoded all the states to avoid a glitch in every sequence. In this situation, we can employ a more “bombproof” methodology.

Glitchless Output State Machine Methodology

The key to making glitchless outputs from our state machines is to make sure that all outputs come directly from a flip flop output. In the previous example we could accomplish this by adding an additional flip flop to the circuit.



The "cleanup" flip flop effectively removes the glitches created by the output decoder logic by "re-registering" the glitchy output signal. This is an effective solution but it delays the output signal by one clock cycle. It would be better to place the flip flop so that somehow the next state logic could create the output signal one cycle early. This is analogous to placing the output decode inside the next state decoder and adding one state bit.

The final solution alluded to above is to add one state bit that does not represent the present state but is representative only of the output signal. For the counter example we would encode the state bits like this: "000", "101", "110", "011". Thus the present state is actually broken into two parts, one representing the state (two LSB's) and the other part representing the output signal (the MSB). This will guarantee that the output signal must come from a flip flop. An another advantage is that the output signal becomes available in the same cycle as the state becomes active and with no decoder delay

Lets see how we can code this style of state machine so that the synthesis tool gives us what we want

Coding the Glitchless State Machine

Using our two-bit counter as an example here is how we could force an encoding that would allocate one flip flop output as our glitchless output.

First of all we decide up on our encoding.

present state vector consists of

"output state" bit	"present state" bits
0	00
1	01
1	10
0	11

determines the value of decode_out

keeps track of what state the counter is in in

The present state vector we will declare as STD_LOGIC_VECTOR actually consists of a one bit vector that represents the value that *decode_out* should have in the state we are in, plus two bits representing the present count state

Now, to create the present and next state vectors and assign their values as we have just stated, we do the following in the declaration area of our architecture.

```
--declare the vectors
SIGNAL byte_cnt_ns  :  STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL byte_cnt_ps  :  STD_LOGIC_VECTOR(2 DOWNTO 0);

--state encodings
CONSTANT cnt1 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "000";
CONSTANT cnt2 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "101";
CONSTANT cnt3 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "110";
CONSTANT cnt4 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "011";
```

The use of CONSTANT here allows us to use the names instead of bit vectors like our enumerated example and not be guessing what state "110" is. This becomes far more important in more complex state machines.

Coding the Glitchless State Machine

The rest of our state machine is coded identically to the enumerated example given previously with two exceptions.

Remember that the output signal is now really a part of the vector *byte_cnt_ps*. How do we associate this bit of the bit vector with the output signal? We simply rip the bus. For this example:

```
decode_out <= byte_cnt_ps(2); --attach decode_out to bit 2 of _ps
```

This piece of code can be placed in a concurrent area preferably adjacent to the process containing this state machine.

Also, since we are not covering every possibility in our **CASE** statement with our four **CONSTANT** definitions, we must take care of this. To do so we utilize the **OTHERS** statement as follows:

```
 WHEN OTHERS =>
      byte_cnt_ns <= (OTHERS => '-');
END CASE; (OTHERS => '-')
```

This code segment implies when no other case matches, the next state vector may be assigned to any value. As we do not expect (outside of catastrophic circuit failure) any other state to be entered, this is of no concern. By allowing assignment of the next state vector to any value the synthesis tool can use the assigned “don’t cares” to minimize the logic.

The mysterious portion of the above: `(OTHERS => '-')`

Really just says that for every how many bits are in the vector (all the others) assign a don’t care value. Its a handy trick that allows you to modify your code later and add or subtract bits in a vector but never have to change the **OTHERS** case in your **CASE** statement.

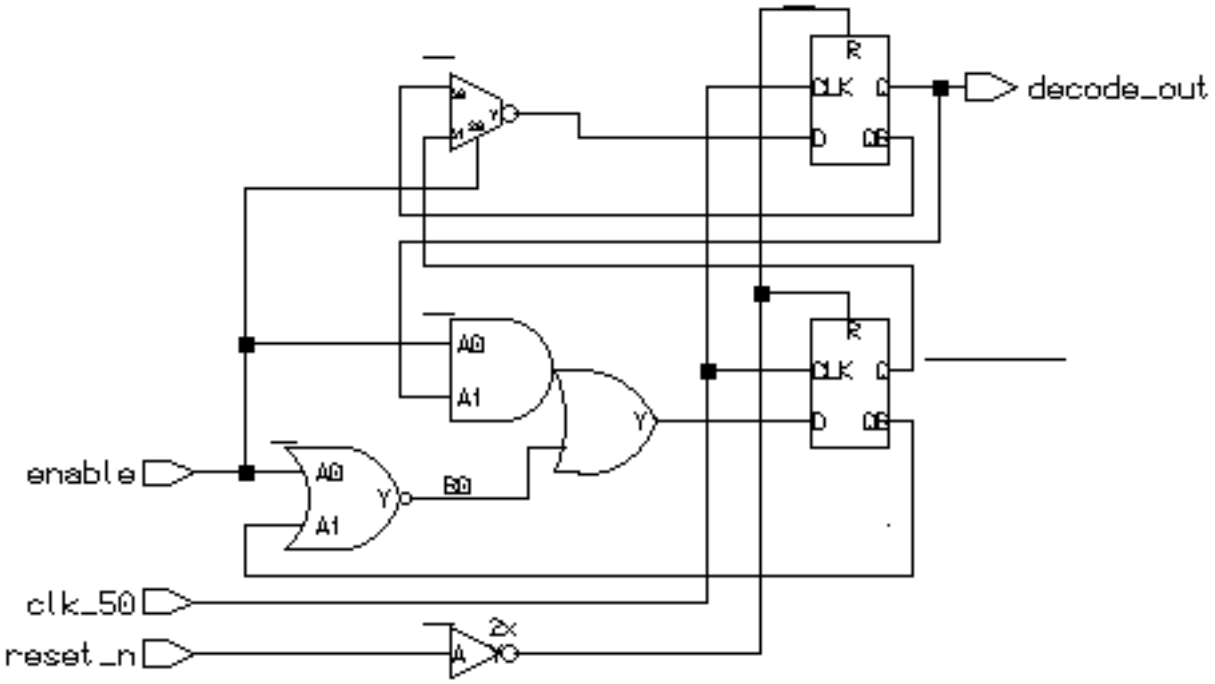
Lets look at the new and improved state machine code and the synthesized output.

Code for the Glitchless Output State Machine

```
ARCHITECTURE beh OF sm1 IS
--declare the vectors and state encodings
SIGNAL byte_cnt_ns  :  STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL byte_cnt_ps  :  STD_LOGIC_VECTOR(2 DOWNTO 0);

CONSTANT cnt1 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "000";
CONSTANT cnt2 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "101";
CONSTANT cnt3 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "110";
CONSTANT cnt4 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "011";

 BEGIN
 byte_cntr:
 PROCESS (clk_50, reset_n, enable, byte_cnt_ps, byte_cnt_ns)
 BEGIN
   --clocked part
   IF (reset_n = '0') THEN
     byte_cnt_ps <= cnt1;
   ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
     byte_cnt_ps <= byte_cnt_ns;
   END IF;
   --combinatorial part
     CASE byte_cnt_ps IS
     WHEN cnt1 =>
     IF (enable = '1') THEN
        byte_cnt_ns <= cnt2;
      ELSE
        byte_cnt_ns <= cnt1;
      END IF ;
     WHEN cnt2 =>
      IF (enable = '1') THEN
        byte_cnt_ns <= cnt3;
      ELSE
        byte_cnt_ns <= cnt2;
      END IF ;
     WHEN cnt3 =>
      IF (enable = '1') THEN
        byte_cnt_ns <= cnt4;
      ELSE
        byte_cnt_ns <= cnt3;
      END IF ;
     WHEN cnt4 =>
      IF (enable = '1') THEN
        byte_cnt_ns <= cnt1;
      ELSE
        byte_cnt_ns <= cnt4;
      END IF ;
     WHEN OTHERS =>
        byte_cnt_ns <= (OTHERS => '-');
   END CASE;
 END PROCESS byte_cntr;
 decode_out <= byte_cnt_ps(2);  --output signal assignment
```

Synthesized Glitchless Output State Machine

Here is the synthesized output for our glitchless output state machine:



Whoa, you say. That's not what I expected. Here is a case where the synthesis tool did what you "meant" but not what "you said". We sure enough got an output from a flip flop that is glitchless but the circuit still only has two flip flops. What the synthesis tool did what to rearrange the state encoding such that the bit that *decode_out* is tied to is one in states cnt2 and cnt3. In other words, it Grey coded the states to avoid the extra flip flop.

Other tools may or may not behave in the same way. Once again, it pays to checkout the transcript, take a look at the gates used and take a peek at the schematic. The synthesis transcript did mention what was done in a vague sort of way:

```
-- Compiling root entity sm1(beh)
"/nfs/guille/u1/t/traylor/ece574/src/sm1.vhd", line 27:
Info, D-Flipflop reg_byte_cnt_ps(0) is unused, optimizing...
```

The moral of the story... read the transcripts. Don't trust any tool completely. Double check everything. "The paranoid survive."..... Andy Grove

State Machines in VHDL - General Form

As seen in the previous example, most state machines in VHDL assume the following form:

```
process_name:
PROCESS(sensitivity_list)
BEGIN
--synchronous portion
  IF (reset = '1') THEN
    present_state <= reset_conditon;
  ELSIF (clk'EVENT AND clk = '1') THEN
    present_state <= next_state;
  END IF;
--combinatorial part
  CASE present_state IS
     WHEN state1 =>
        next_state <= state_value;
        other_statements;
     WHEN state2 =>
        next_state <= state_value;
        other _statements;
                    *
                    *
     WHEN OTHERS => next_state <= reset_state;
   END CASE;
END PROCESS;
```