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.

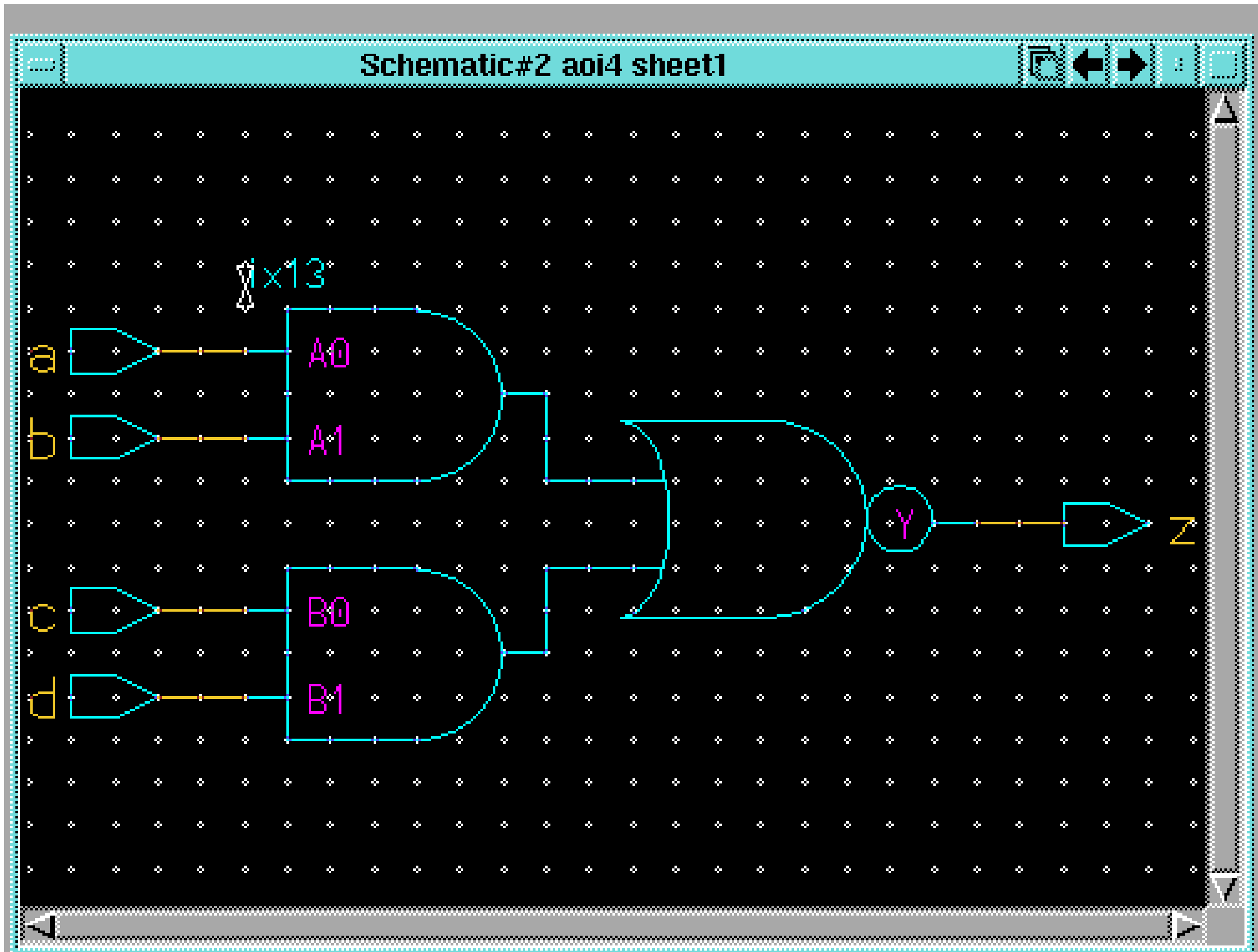
Note also, the intermediate signals temp1 and temp2 have optimized away.

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

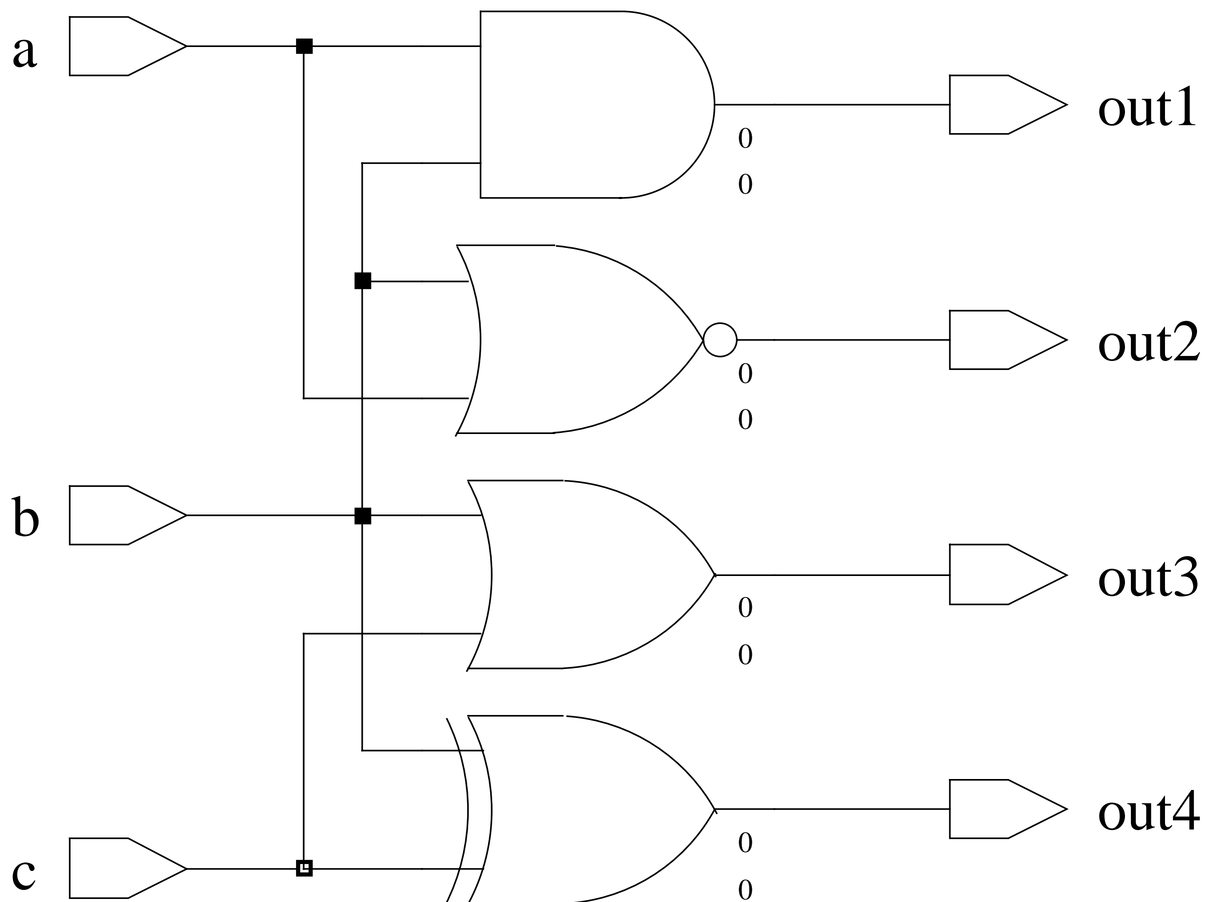
```
SIGNAL result0, signal1, signal2 : std_logic;
SIGNAL result1 : std_logic_vector(31 DOWNTO 0);
SIGNAL signal3 : std_logic_vector(31 DOWNTO 0);
SIGNAL signal4 : std_logic_vector(31 DOWNTO 0);

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:



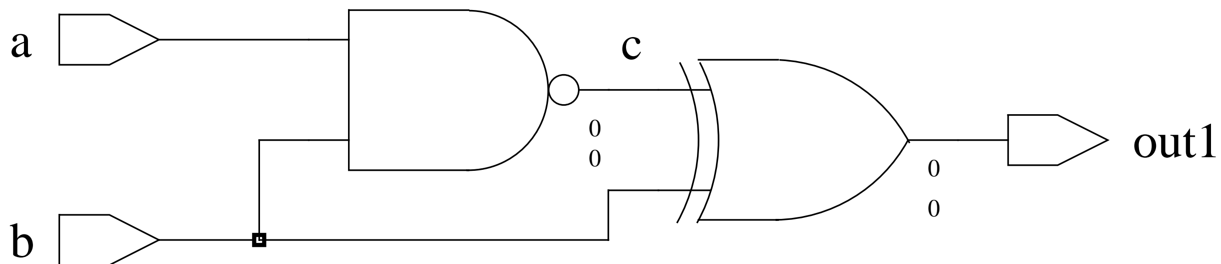

```
ARCHITETURE example of concurrent IS
BEGIN
  out1 <= a AND b;
  out2 <= a NOR b;
  out3 <= b OR c;
  out4 <= b XOR c;
END example;
```

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.

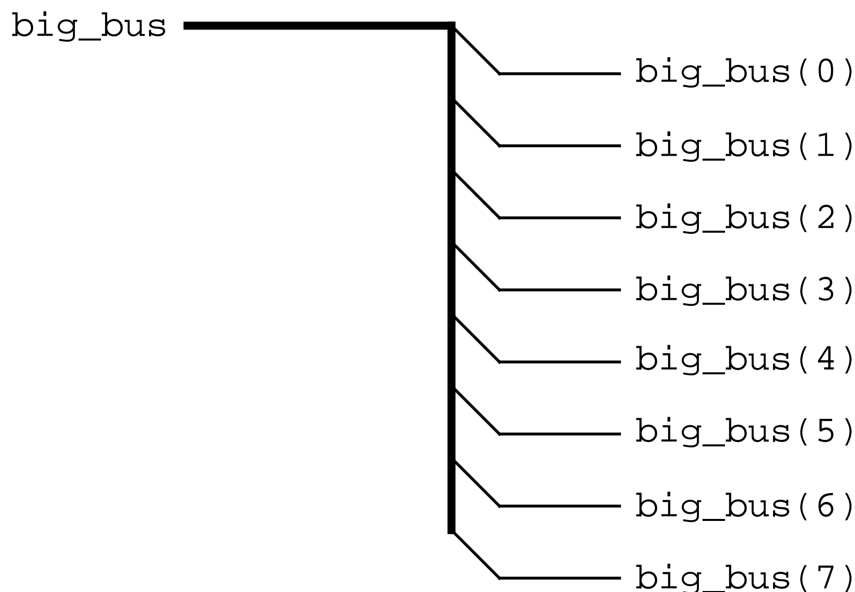
Signal Assignment with Busses

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 are 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?
```

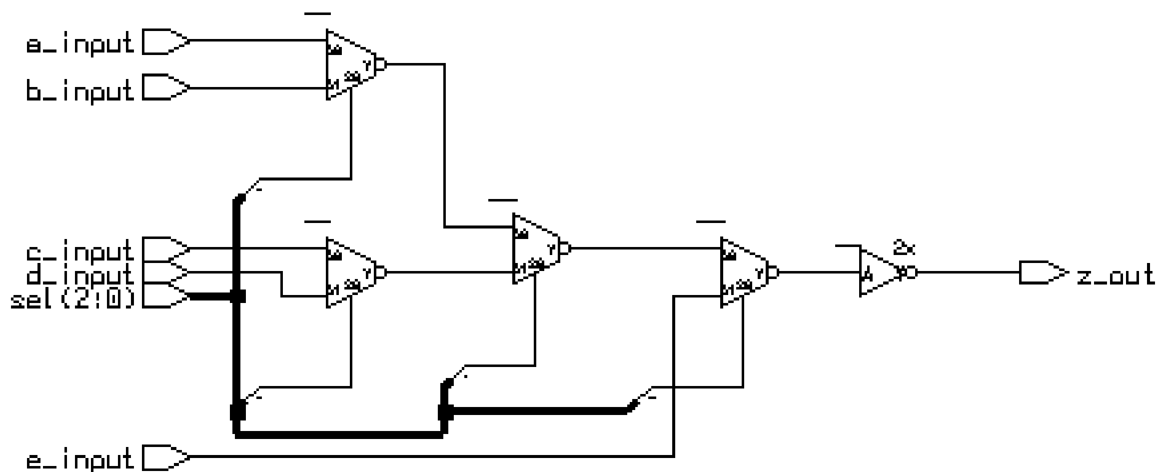
Conditional Concurrent Signal Assignment

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:

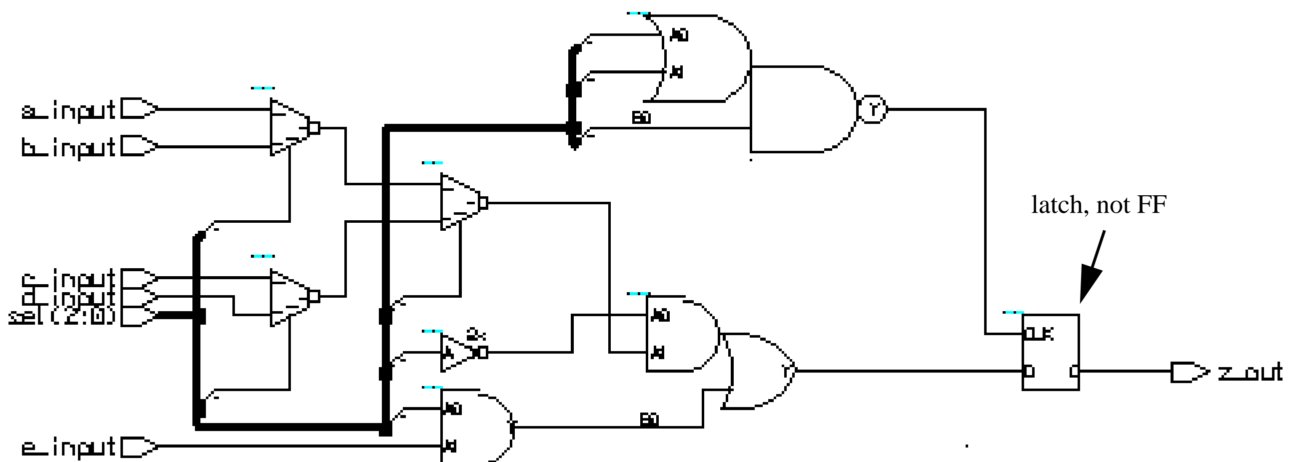


Conditional Concurrent Signal Assignment

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