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**

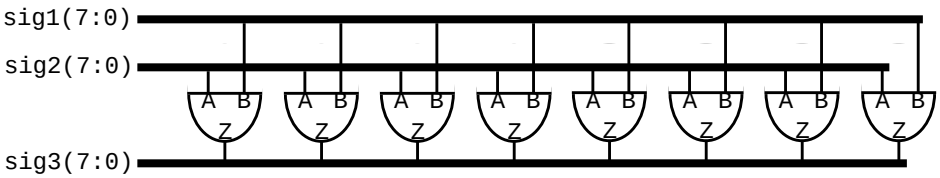
FOR Scheme Format:

```
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.

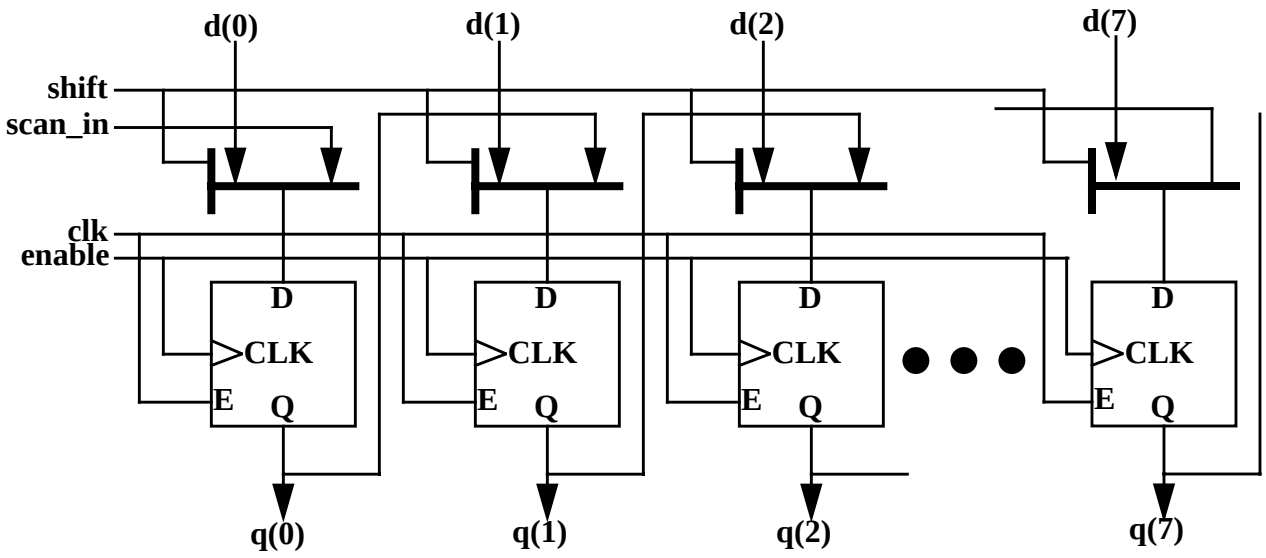
IF Scheme Format:

```
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 : 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 <u>not</u> 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.

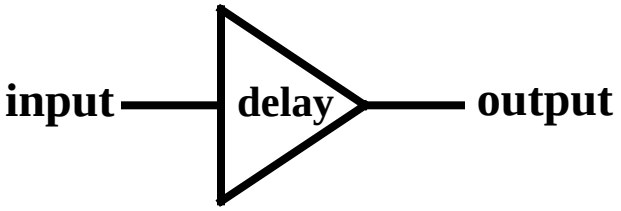
The basic idea is to make a model work, and then make it work at the desired speed. Only experience will help you determine how fast your HDL code will eventually run.

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.

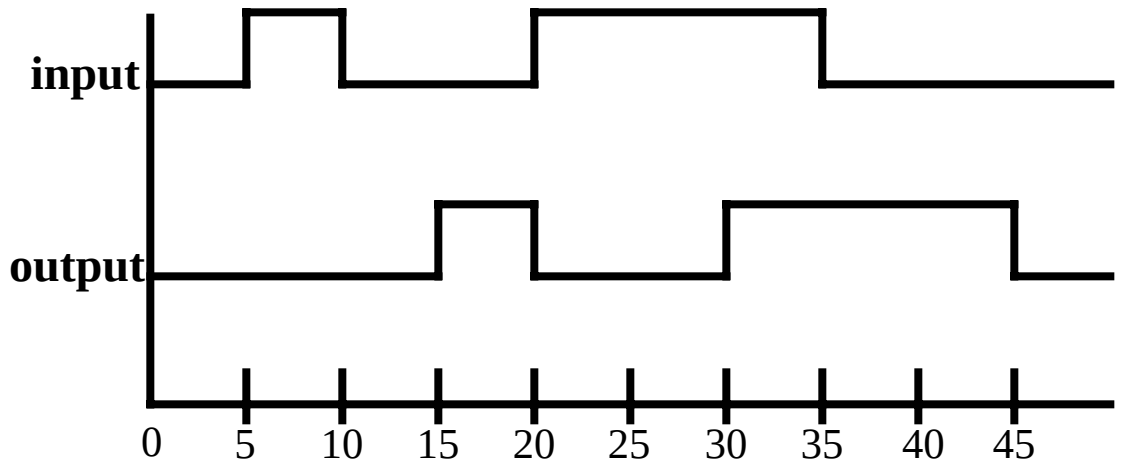
Delay Types - Transport

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.

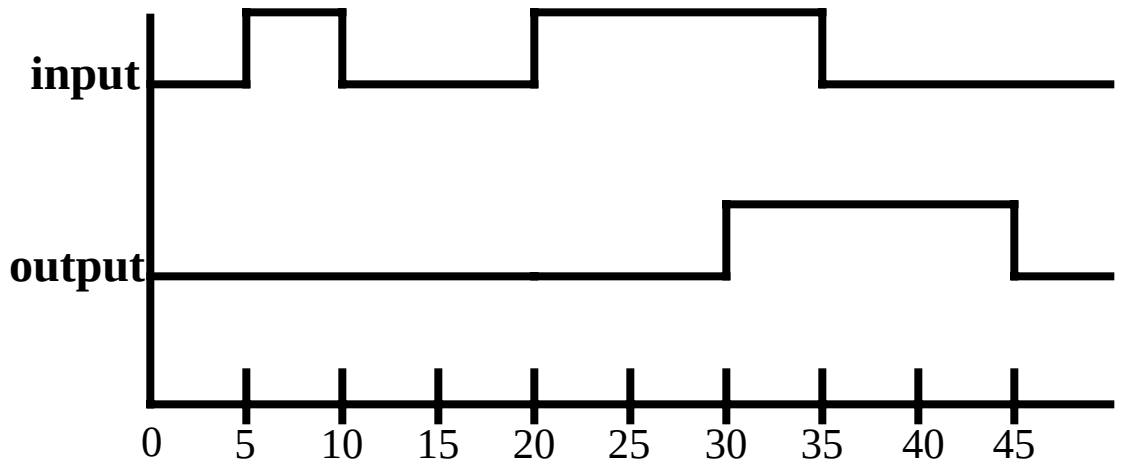
Inertial delay provides for specification of input pulse width, i.e. 'inertia' of output, and propagation delay.

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.

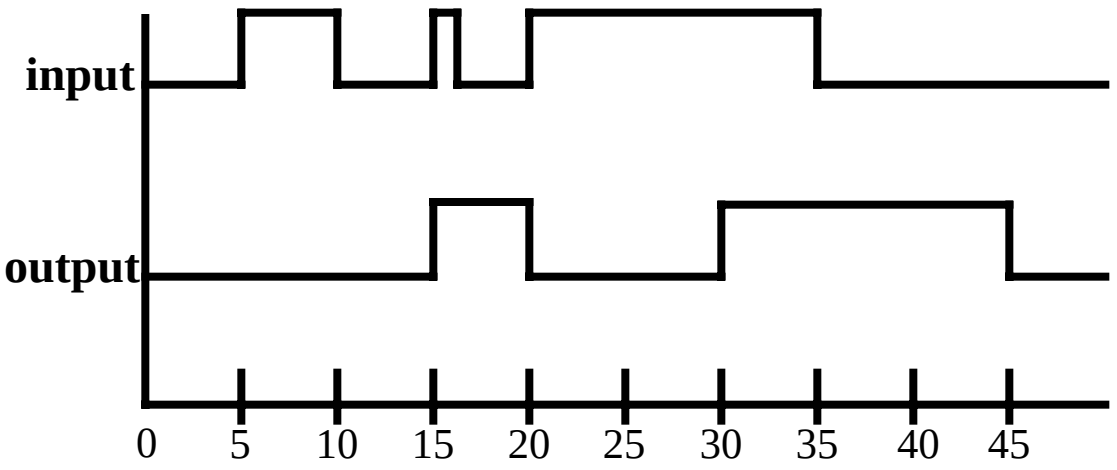
Inertial delay acts like a real gate. It "eats" pulses narrower in width than the propagation delay.

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 <u>infinite</u> number of delta times equals <u>zero</u> 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:

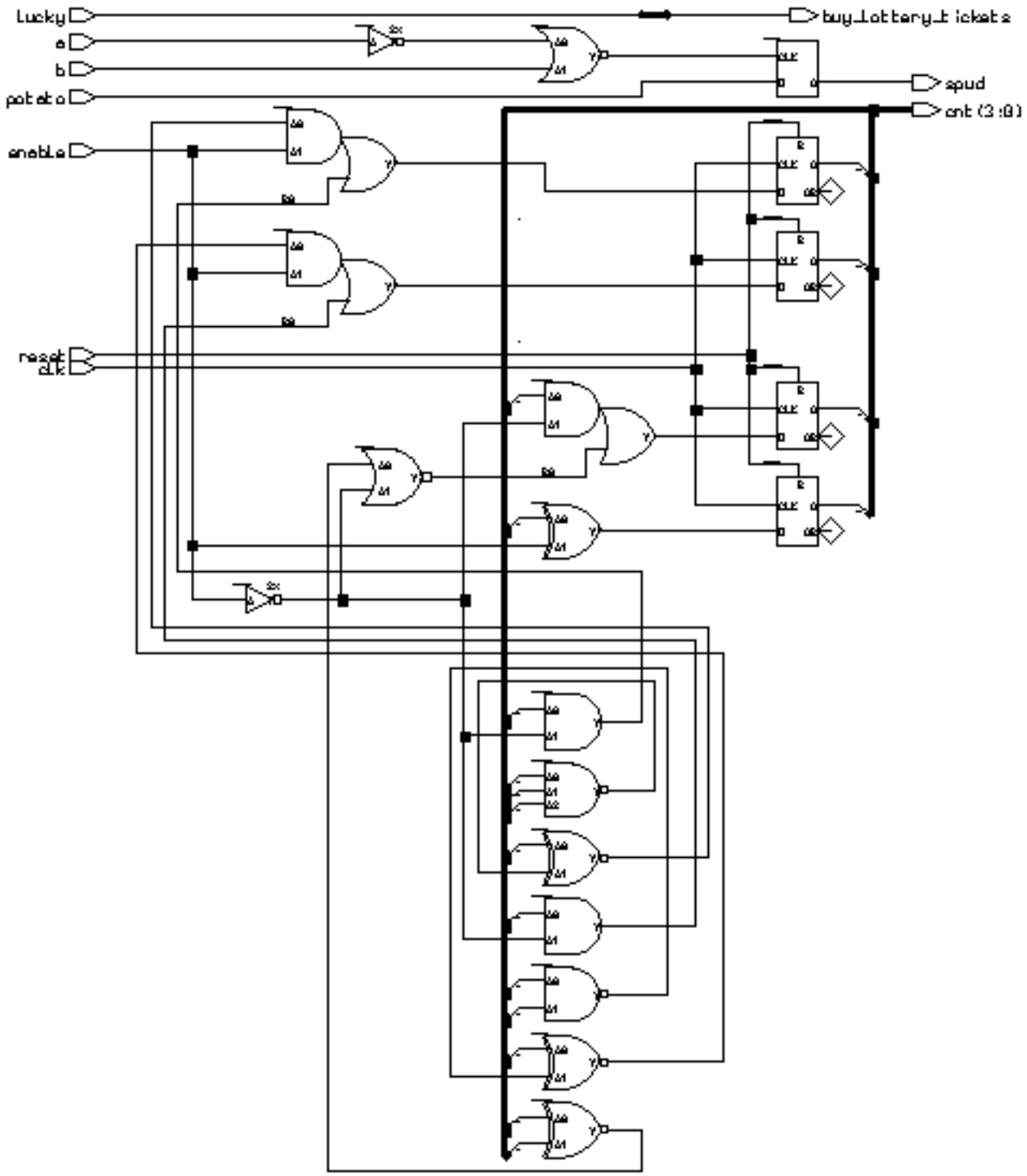
```
--enabled latch
IF (a = ‘1’ AND b = ‘0’) THEN
  spud <= potato;
END IF;
```

```
--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`

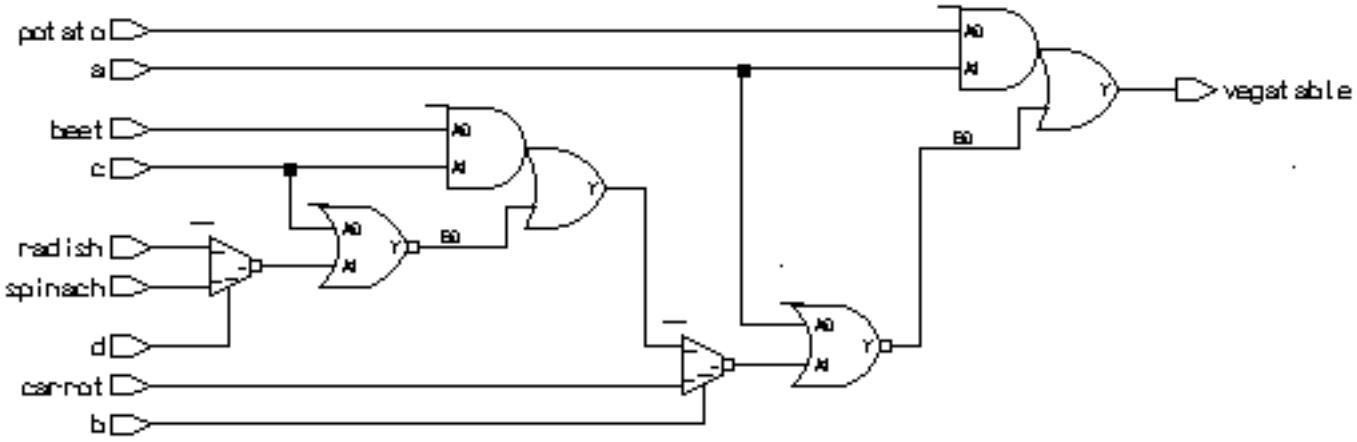
Synthesized example from previous page



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.

<u>OperatorOperation</u>

```
=                    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.