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:

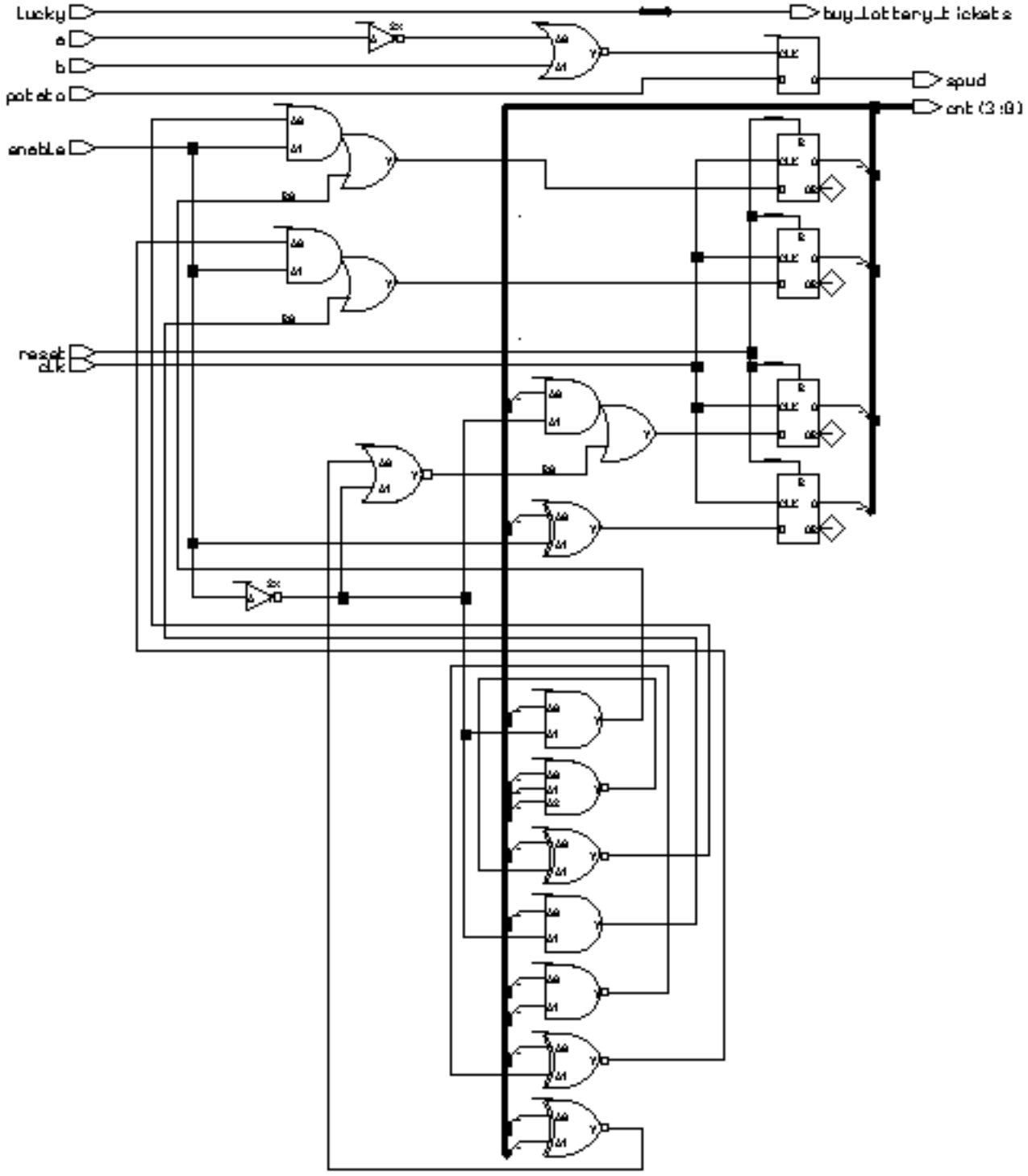
```
--enabled latch
IF (a = ‘1’ AND b = ‘0’) THEN
  spud <= potato;
END IF;
```

```
--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`

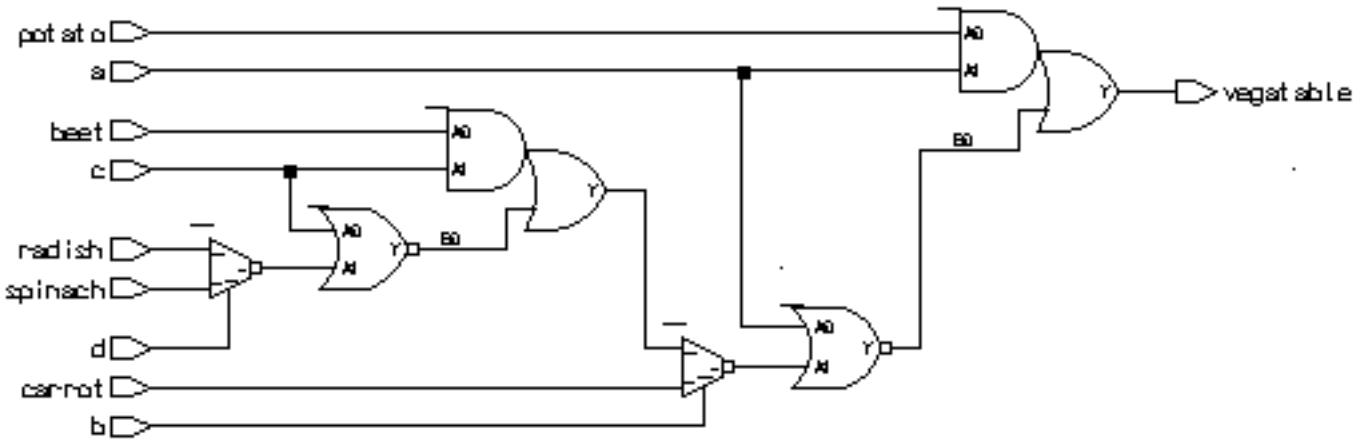
Synthesized example from previous page



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:

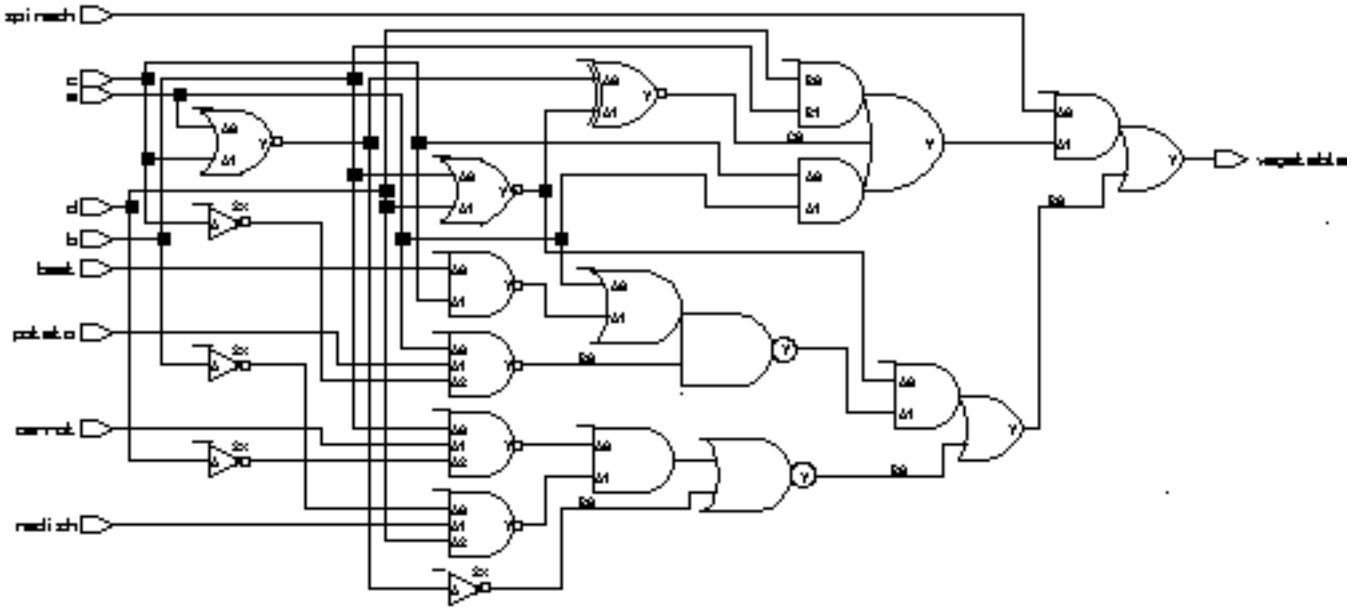
```
--a four to one mux
mux: PROCESS (sel, a, b, c, d)
BEGIN
  CASE sel IS
     WHEN “00”  => out <= a;
     WHEN “01”  => out <= b;
     WHEN “10”  => out <= c;
     WHEN “11”  => out <= d;
     WHEN OTHERS => out <= ‘X’;
  END CASE ;
END PROCESS mux;
```

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;
```



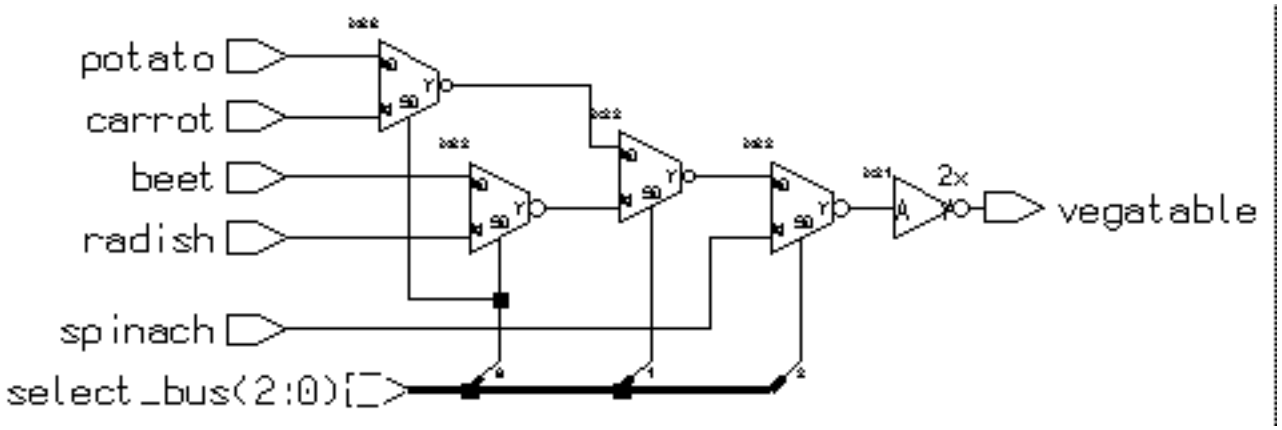
With the exception of spinach, the number of gate delays from each signal input to output is four. The gate delays in the **IF** example varied from 1 to 8 gate delays. However, this function for CASE could be coded better.

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*.

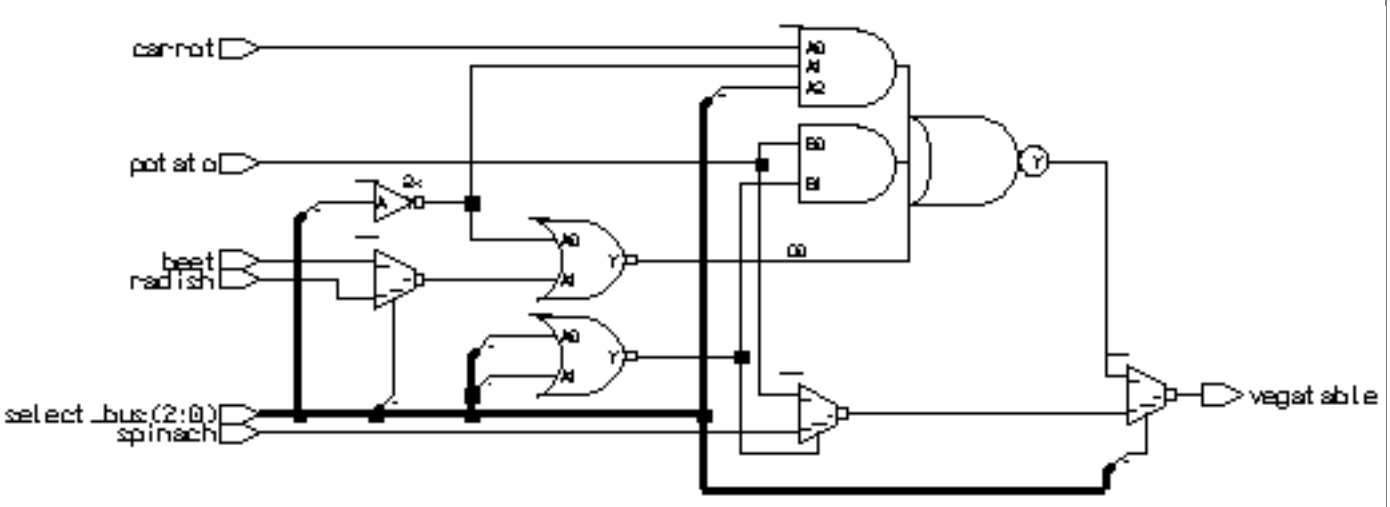
If we chose another valid input for the OTHERS clause, the error (‘X’ output from a flip flop) in the simulation will not be propagated to downstream logic. It will stop or be lost at the mux input because the select_bus value “0X1” maps to a valid input. At the next clock cycle the flip flop may transition 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.

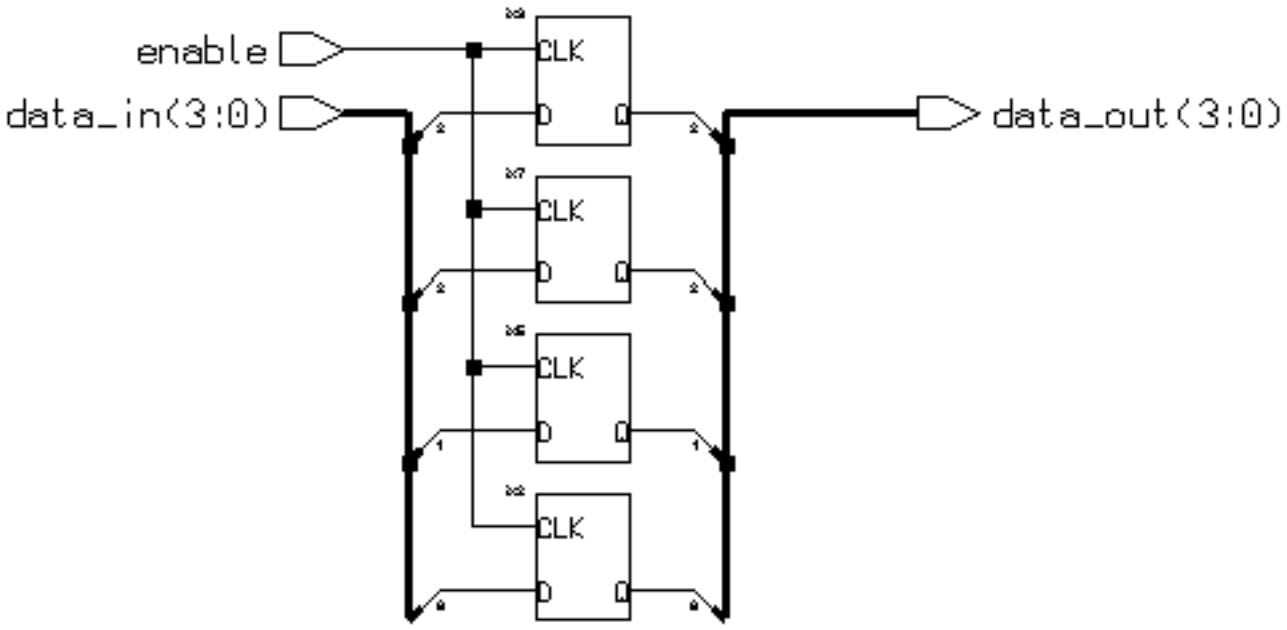
```
--infer 4-bit wide latch
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_vector_arith.ALL;


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.

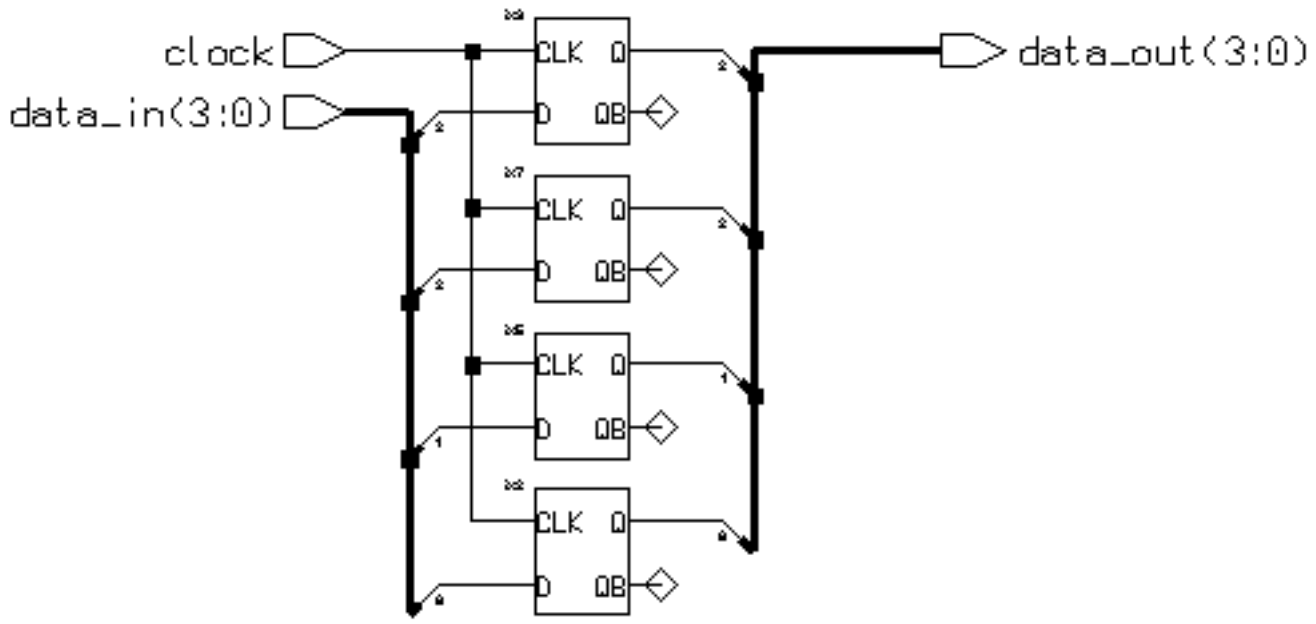
Inferring D-type Flip Flops

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:

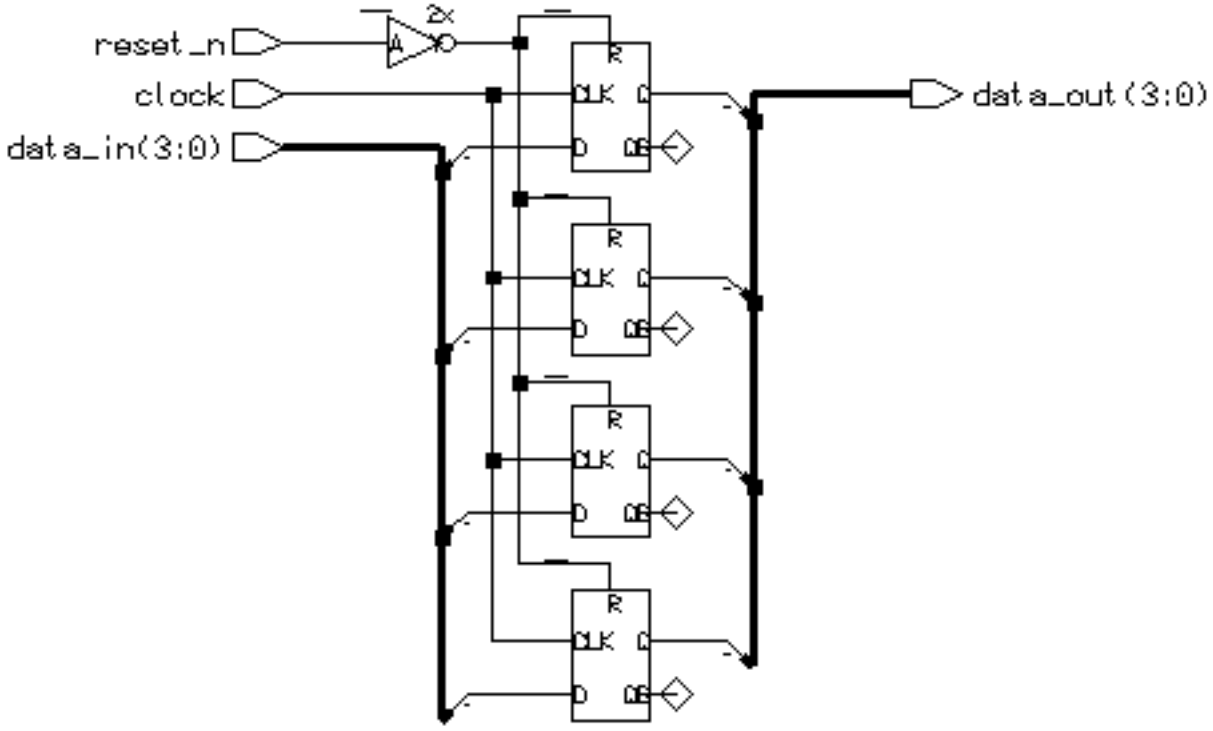
```
infer_dff:
PROCESS (clock, data_in)
  BEGIN
    IF (clock'EVENT AND clock = '1') THEN
      data_out <= data_in;  --get new value
    ELSE
      data_out <= data_out; --hold old value...UNNECESSARY
    END IF;
END PROCESS infer_dff;
```

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.