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:

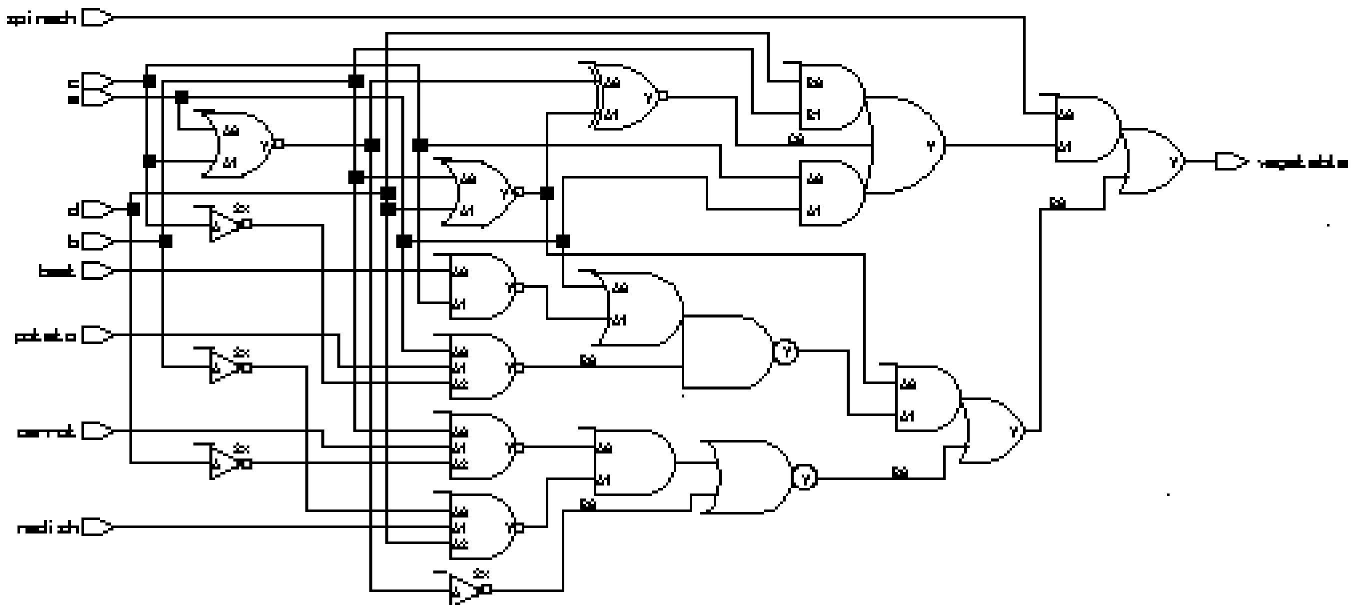
```
--a four to one mux
mux: PROCESS (sel, a, b, c, d)
BEGIN
  CASE sel IS
     WHEN “00”  => out <= a;
     WHEN “01”  => out <= b;
     WHEN “10”  => out <= c;
     WHEN “11”  => out <= d;
     WHEN OTHERS => out <= ‘X’;
  END CASE ;
END PROCESS mux;
```

Either every possible value of *expression_value* must be enumerated, or the last choice <u>must</u> 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;
```



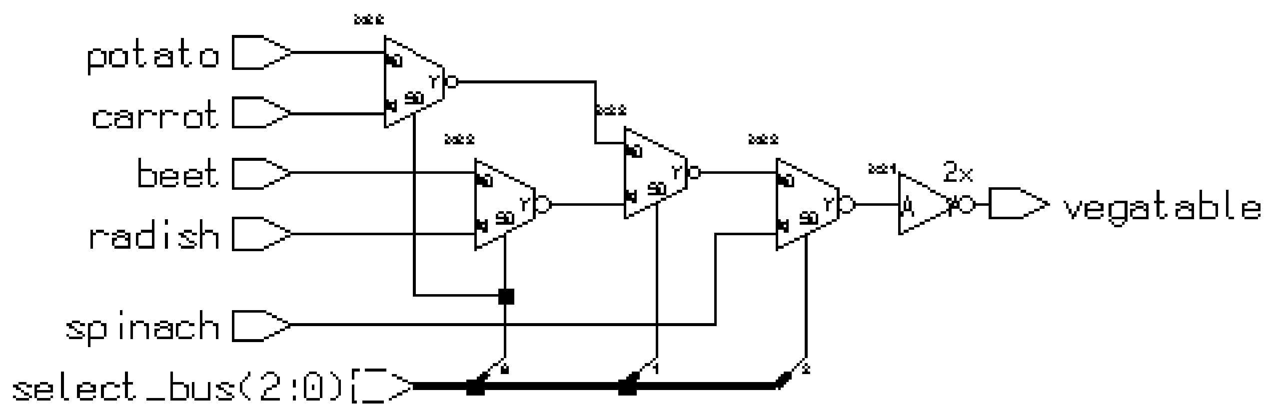
With the exception of spinach, the number of gate delays from each signal input to output is four. The gate delays in the **IF** example varied from 1 to 8 gate delays. However, this function for CASE could be coded better.

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.