

## Two Interesting Problems in VHDL Model Design

Michael D. McKinney  
VHDL Design Services  
2113 Normandy Drive  
Irving, TX 75060  
E-mail : 74037.3515@compuserve.com

Gordon L. Sturm  
Consulting Engineer  
610 E. Bell Rd., Suite 2-228  
Phoenix, AZ 85022  
E-mail : sturm@indirect.com

### Abstract

Building models and testbenches in VHDL can be a harrowing as well as exciting and creative task. This paper presents two common VHDL structures which are well known and easily understood : multiplexers and data queues. It is hoped that the identification of some of the "gotchas" in these structures will help preclude inclusion of bad language choices into otherwise good model designs.

### Problem 1 : Multiplexer Design

A multiplexer (mux) is a hardware element designed to select from several inputs, and to send one of them to a single output. Both data inputs and data outputs may be single wires or groups (busses or vectors). The select inputs may be of arbitrary number and usually affect the number of data inputs to the mux. However, there may also be some built-in select decoding so that the width of the select inputs may not necessarily indicate the exact depth of the mux (the exact number of its data inputs).

In the problem before us, the select inputs to the mux are a set of signals which are interpreted as addresses, and the data inputs are sets of signals from various other parts of the design (Figure 1).

### Basic Issues

Three items of interest are quickly derived from a top-level scan of the VHDL source code:

First, this designer included in the select address a new bit which is called INT\_SVC. The new bit is concatenated to the low-order end of the address selectors (becomes the new bit 0) just before the select address is used in the mux structure.

Second, this designer has chosen to utilize a conditional concurrent signal assignment statement to construct the mux structure. This is a perfectly valid method.

Third, it is clear that some of the inputs are selected with specific select bus bit patterns, while others are selected with ranges of values. Again, both are acceptable methods.

### Looking in Depth

So far, a quick scan of the code has produced nothing unexpected. In truth, this code was analyzed, elaborated, simulated and even synthesized without error. However, a more in-depth look, this time for actual functionality, brings other items of interest to light:

First, functionally this designer wanted the mux to

```

sel_addr(14 downto 0) <= (addr(13 downto 0) & INT_SVC);

out_data <= CREG WHEN sel_addr = "00000000000001" ELSE
  s1REG WHEN sel_addr = "000000000000011" ELSE
  s2REG WHEN sel_addr = "000000000000101" ELSE
  s3REG WHEN sel_addr = "000000000000111" ELSE

  s4REG WHEN sel_addr = "00010000000001" ELSE
  s5REG WHEN sel_addr = "000100000000011" ELSE
  s6REG WHEN sel_addr = "000100000000101" ELSE
  s7REG WHEN sel_addr = "000100000000111" ELSE
  s8REG WHEN sel_addr = "000100000001001" ELSE
  s9REG WHEN sel_addr = "000100000001011" ELSE

  s10REG WHEN sel_addr = "00100000000001" ELSE
  s11REG WHEN sel_addr = "001000000000011" ELSE
  s12REG WHEN sel_addr = "001000000000101" ELSE
  s13REG WHEN sel_addr = "001000000000111" ELSE
  s14REG WHEN sel_addr = "001000000001001" ELSE
  s15REG WHEN sel_addr = "001000000001011" ELSE

  s16REG WHEN sel_addr = "01100000000001" ELSE
  s17REG WHEN sel_addr = "011000000000011" ELSE
  s18REG WHEN sel_addr = "011000000000101" ELSE
  s19REG WHEN sel_addr = "011000000000111" ELSE
  s20REG WHEN sel_addr = "011000000001001" ELSE

  t1REG WHEN ((sel_addr = "01110000000001") OR
              (sel_addr < "011100001000001")) ELSE
  t3REG WHEN ((sel_addr = "10000000000001") OR
              (sel_addr < "100010100000001")) ELSE
  t4REG WHEN ((sel_addr = "110000001100001") OR
              (sel_addr < "110010000000001")) ELSE

rf_out WHEN (INT_SVC = '0') ELSE
"-----";

```

-- Figure 1 --

pick from the next-to-last option whenever the order select bit (INT\_SVC) is '0', and to select some other option when this bit is '1'.

Second, perhaps not so clear, is that ANY incoming select address within the address space should select the next-to-last option when the INT\_SVC bit is '0', since the main address decoding is left out of this line.

#### Unintentional, but Deadly Errors

So far, so good. There are, however, several obviously unintentional but potentially deadly errors introduced into this mux structure, all centered on the **tReg** lines, all of which specify a range of address decoding values. These lines immediately create two major difficulties:

First, note that one-half of the available addresses in the evaluation will have the low order bit (remember, it is INT\_SVC) set to '0'. Any address in its range will be decoded by this line, which means that for one-half of the opportunities (the ones with INT\_SVC

= '0') data will be sent to the output from the wrong source.

Second, and equally important, note that because the designer used the '<' operator here, ANY ADDRESS IN THE ENTIRE ADDRESS SPACE lower than the specified value will be decoded, not just those from the previous select options. The intention was, "any address in the address space NOT ALREADY SELECTED, with INT\_SVC = '1', will be selected here" -- clearly not the actual function.

### Secondary Issues

Most VHDL design specifications require that no VHDL model should produce the '-' state. The resolution function for signals will resolve this state as if it were an 'X', and it would therefore indicate an error during simulation as it is designed. The final ELSE could therefore be 'X's rather than '-s.

From the standpoint of synthesis, it doesn't make sense to use the 'X' or '-' in the final ELSE clause. What will a particular synthesis engine do with these states? If we want the code to synthesize in a predictable manner, it would certainly be preferable to drive a suitable vector composed of '1's and/or '0's for the final ELSE clause.

This situation illustrates an interesting trade-off between synthesis and simulation issues. For simulation, the 'X' or '-' style ELSE clause is nice because it clearly shows when an illegal mux selection has been made, especially when the select inputs are not mutually exclusive. However, for synthesis a specific bit pattern would probably serve to produce its hardware more predictably.

Finally, there is the problem of simulator and hardware inefficiency produced by the use of the '<' or '>' operators. Already, the synthesis engine has had to produce at least three 15-bit adders to decode the specified range of addresses in the tReg lines. If the intention of this designer had been to decode both limits of the range (i.e, replacing '=' with '>=' in the tReg lines), the efficiency would be even worse, now making even more 15-bit adders.

### Fixing It

The key to fixing the t1Reg line is to realize that t1Reg should be selected when addr (13 downto 5) is "011100000", and without regard to the value represented by the lower address bits. So, the t1Reg line can be rewritten as follows:

```
t1Reg WHEN (addr(13 downto 5) = "011100000") AND
           (INT_SVC = '1') ELSE
```

This selection can be made by decoding a total of 10 bits instead of adding 15 bits twice.

The t3Reg line has a similiar problem, but the bit patterns are slightly more complicated. The upper limit does not fall on a nice even bit boundary, forcing the use of two terms to define it:

```
t3Reg WHEN ((addr(13 downto 9) = "10000") OR
            (addr(13 downto 7) = "1000100")) AND
           (INT_SVC = '1') ELSE
```

Although it needs two terms, at least it can be done with no adders.

We are not quite so lucky with the t4Reg selection. We can break down this address into two requirements: 1) the top 5 bits must be "1100" and 2) the next 5 bits must be >= "00011". Once again, the lower bits of the address are unnecessary:

```
t4Reg WHEN (addr(13 downto 9) = "11000") AND
           (addr(8 downto 4) >= "00011") AND
           (INT_SVC = '1') ELSE
```

We do wind up here with one 5 bit adder, but that is better than the original which required two 15-bit adders.

### Lessons Learned

A detailed interpretation of the selector decoding reveals that when INT\_SVC is '0' the correct data is sent only 20% of the time, based on the available address space. This means that the `rf_out` option fails 80% of the time, and during those times invalid data is sent to the output. An engineer would have to pay close attention to the data output to see that something was amiss. If it happens that the testbench for this design always simulates using INT\_SVC = '0' for one of the acceptable addresses, the error quite possibly would not be caught before the design was set in silicon. This, of course, would be a mistake of mammoth proportions.

Remember that adder structures in hardware are large and slow. Add vectors only when necessary, and then try to use as few bits as possible.

Perhaps for this design a complete rework of the mux would be the best course to take, evaluating the INT\_SVC signal separately from the addresses, and at least attempting to reduce or eliminate the usage of the '<' and '>' operators.

## Problem 2 : Data Queues

The second problem involves a testbench situation where it is necessary to generate a serial data stream for use as stimulus by the design under simulation. In this case the designer wanted to be able to change the data content easily, assuming that the data would always have long continuous periods of only '1's or only '0's. He chose to read in the data using normal textio processes and to store it in the simulator using **delta coding**. Delta coding stores the times that the data changes rather than every data bit.

### Basic Thoughts

In this simplified case (Figure 2) Ser\_Data is the desired serial output stream, just 1 bit wide. The data stream can contain up to 400 rising and 400 falling edges (DataOn, DataOff). The designer created a 640ns clock cycle process based on his required bit-time length, and this process triggers the `make_data` process which handles the actual signal assignments.

There are two controlling arrays, DataOff and DataOn, both of mode SIGNAL and each containing 400 elements of type TIME. DataOff contains all the times during simulation when the Ser\_Data pin should go to '0', while DataOn contains those times

when Ser\_Data should go to '1'.

The algorithm is of this structure: when the trigger clock rises, one side of the IF-ELSE structure will perform. As with any IF-ELSE structure, the IF part reacts with higher priority than the ELSE part. For each side, the lowest element of the respective **data queue** (a TIME element) is compared to present time (NOW). If present time is equal or later to the specified array time, the Ser\_Data signal is set '1' or '0'.

The next activity is to assign a slice of the DataOn or DataOff array (all elements except the lowest) to a different slice of the same array (all elements except the highest). The designer is asking the simulator to copy each element of the array down one place in the array. This will cause the bottom of the array (element LOW) to be ready for the next edge-triggered evaluation.

### Noteworthy features, both good and bad

1) The use of the LOW and HIGH attributes in the IF and assignment statements is good because it makes the code insensitive to the size of the arrays.

2) The controlling data consists of a set of high-going TIMES and low-going TIMES for the data stream. Some other code must compute these time values from the desired bit content for the output stream, and must load the arrays at the start of simulation. This is a valid method, and will not be changed.

3) For this code to work properly, the designer must store increasing time values in the DataOff and DataOn arrays. An interesting aspect of this code is that if the identical time value is placed in corresponding elements of the two arrays, a 640ns high-going pulse will be produced on Ser\_Data. This could be a nice feature or a hidden flaw, depending on the designers needs.

4) The memory requirements for this code are substantial. The simulator must contain more than 800 signals (including all their standard attributes) just to control a single data bit.

5) The simulation time inefficiency of the `make_data` process is remarkable. For each signal assignment to the output bit another 399 signal assignments (one for each array element in the slice) must be carried out by the simulator. Remember, too, that each signal

```

ARCHITECTURE behavioral OF data_gen1 IS
TYPE TimeArray IS ARRAY (1 to 400) OF Time;
SIGNAL clock      : Std_logic;
SIGNAL DataOff : TimeArray := (OTHERS => 99999999 ns);
SIGNAL DataOn  : TimeArray := (OTHERS => 99999999 ns);
BEGIN
----- Clock process -----
make_clock : PROCESS
BEGIN
  clock <= TRANSPORT '1', '0' AFTER 320 ns;
  WAIT FOR 640 ns;
END PROCESS make_clock;
----- Data Process -----
make_data : PROCESS (clock)
BEGIN
-----
-- Some code here that used textio to load the DataOn and DataOff arrays.
-----
IF (clock'EVENT AND clock = '1') THEN -- recognize a rising edge
  IF (NOW >= DataOn(DataOn'LOW)) THEN
    ser_data <= '1';
    DataOn(DataOn'LOW TO DataOn'HIGH-1) <=
      DataOn(DataOn'LOW+1 TO DataOn'HIGH);
  ELSIF (NOW >= DataOff(DataOff'LOW)) THEN
    ser_data <= '0';
    DataOff(DataOff'LOW to DataOff'HIGH-1) <=
      DataOff(DataOff'LOW+1 TO DataOff'HIGH);
  END IF;
END IF;
END PROCESS make_data;
END behavioral;

```

-- Figure 2 --

assignment carries the additional overhead of modifying all the normal standard attributes of the signal -- quite a lot of work.

Perhaps it is a tribute to the simulator in use, but despite the inefficiencies of this code it actually does work. It was used for several months on a real project.

### Improvements

The key to improving this code is to realize that it is really managing an event queue for the output bit. However, this is not necessary in a VHDL system: the simulator itself manages a perfectly good event

queue for each of its signals. This queue is called the projected output waveform (section 8.3 of the LRM). The efficiency of this code can be greatly improved by taking advantage of this built-in event queue for Ser\_Data (Figure 3).

The algorithm is now much simpler. There is no clock process, and the data process is contained in a LOOP. The required on and off times are still stored in the same arrays as before, but the arrays themselves are not manipulated. Elements are accessed by new VARIABLES of type INTEGER.

### More Noteworthy Features

```

ARCHITECTURE behavioral OF data_gen2 IS
BEGIN
----- Data Process -----
make_data : PROCESS
TYPE TimeArray is ARRAY (1 to 400) of time;
VARIABLE DataOn : TimeArray := (others => 99999999 ns); -- ON times
VARIABLE DataOff : TimeArray := (others => 99999999 ns); -- OFF times
VARIABLE ion, ioff : INTEGER := TimeArray'LOW; -- index to ON, OFF times

BEGIN
-----
-- some textio-based code here that loaded the ON and OFF arrays
-----
WHILE (ion /= TimeArray'HIGH + 1) AND (ioff /= TimeArray'HIGH + 1) LOOP
  IF DataOn(ion) < DataOff(ioff) THEN -- schedule next ON cmd
    ser_data <= TRANSPORT '1' AFTER DataOn(ion);
    ion := ion + 1;
  ELSE -- schedule next OFF cmd
    ser_data <= TRANSPORT '0' AFTER DataOff(ioff);
    ioff := ioff + 1;
  END IF;
END LOOP
WAIT;
END PROCESS make_data;
END behavioral;

```

-- Figure 3 --

1) As before, the 'LOW and 'HIGH attributes are used to specify array limits wherever possible.

2) The assignment statements to the Ser\_Data output now must use the keyword TRANSPORT. If not used, each new edge assignment would remove all other edges from the event queue, even the ones before the current edge.

3) It is important that the designer make sure to assign to Ser\_Data with increasing AFTER times. If this did not occur, and some assignments were made with earlier times, the assignment would cause all queue elements later than that time to be removed, even with the keyword TRANSPORT. In order to guarantee increasing AFTER times, the IF statement compares the next available DataOff and DataOn times, selecting the smaller (earlier) of the two times. Once the process has used an element from an array, it increments its index variable, in order not to use the same data element again.

4) Note that the behavior of the previous code when the same time value is provided in both arrays is not carried over into this code.

5) Efficiency of this code is considerably improved over the original code. The data arrays are now of mode VARIABLE rather than of mode SIGNAL, vastly decreasing memory requirements during simulation. By using INTEGER indexes, unnecessary signal assignment statements are eliminated. The make\_data process itself executes only once during the entire simulation run, reducing clock time for the simulation. All of the required assignments to Ser\_Data are made during this single process.

#### Final Thoughts

This method deliberately puts a lot of transactions into the projected waveform for Ser\_Data, the output signal. Depending on the simulator used, there might be a limit on the number of transactions allowed.

However, one simulator has allowed more than 12,000 transactions to be placed in this single queue.

Neither version of the code is synthesizable, but this type of code is usually seen in testbenches anyway, where synthesis is generally not an issue.

The efficiency of testbench code is an important consideration because it contributes to over-all simulation speed and indirectly helps to determine how much testing is practical for a given project schedule. This is particularly true when gate-level acceleration or mixed-level simulation is used. The testbench code may in fact be the main factor determining the speed of the simulations.

A small testbench was constructed which created and used 400 elements in each of the arrays, and executed to note speed and memory size differences. For memory needs, the code in Figure 3 required 130KB less than that in Figure 2. For speed, the difference was even more dramatic: the code in Figure 3 executed 37.5 times faster than that in Figure 2. Note, finally, that execution times will increase for Figure 2 as the square of the sizes of the arrays, while that in Figure 3 will increase in a linear fashion, another indication of increased efficiency. A trial run of 4000 elements per array (8000 in the projected waveform) was tried with no simulator error.

## References

IEEE Std 1076-1987 IEEE Standard VHDL Language Reference Manual, (ANSI).

IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual, (ANSI).

IEEE Std 1164-1993 IEEE Standard Multi-Valued Logic System for VHDL Model Interoperability, (Std\_Logic\_1164), (ANSI).

Perry, Douglas L., VHDL, McGraw-Hill, N.Y., 1991.

Coelho, David R., The VHDL Handbook, Kluwer Academic Publishers, Boston, 1989.

Berge, et. al., The VHDL Designers Reference, Kluwer Academic Publishers, Boston, 1992.