A Comparison of Two VHDL Memory Modeling Techniques

Richard Munden
Acuson, A Siemens Company
650-694-5523
munden@acuson.com

Introduction

In February 2001 a new version of the VHDL Initiative Toward ASIC Modeling (VITAL) became official. VITAL 2000 includes many improvements over its predecessor, VITAL 95. However, the most dramatic change is the addition of the VITAL memory package. This package approximately doubles the number of lines of code in the VITAL packages. It provides special table formats, path delays and, timing checks for memories as well as optimized storage for memory arrays.

In this paper I examine how the new package can be used to model a generic SRAM. The model is contrasted with one of identical functionality written without the package. Then the two models are compared for memory usage and simulation speed.

Importance

VITAL'95 provided VHDL with accelerated simulation primitives. It included procedure calls for checking timing constraints and adding propagation delays. And it allowed all timing information to be annotated to the simulation through SDF files. But VITAL'95 did not address memory modeling.

There are many ways memory can be modeled in VHDL. However, since there are so many ways, simulator developers could not optimize any of them for performance or memory usage. Since board level designs frequently contain hundreds of megabytes of memory, the amount of computer memory required to simulate those designs can limit the practicality of performing board level simulations.

VITAL'2000 defines standard methods for memory modeling. With it, tool developers can optimize their compilers and simulators to use fewer resources and have faster execution.

The Old Way

The Free Model Foundry models memories using what we refer to as the Shelor method presented in an article by Charles Shelor published in VHDL Times in 1998. The Shelor method uses an array of type integer to store memory values as shown in figure 1..

```
-- Memory array declaration
TYPE MemStore IS ARRAY (0 to TotalLOC) OF INTEGER
                    RANGE  -2 TO MaxData;
```

FIGURE 1.

Normal memory values are represented as unsigned integers in the range of 0 to MaxData. The reason for extending the range to -2 is to enable the representation of uninitialized memory locations and corrupted locations.

The behavioral part of the model is done as ordinary behavioral VHDL as in figure 2. However, timing constraint checks and path delays take advantage of VITAL'95 procedures.

The New Way

The VITAL'2000 memory package provides a standard method for declaring memory. Because it is known in advance to the tools developers, they have been able to optimize storage such that a 1 byte word occupies only 2 bytes of computer memory.

The VITAL'2000 memory declaration is shown in figure 3. It has 5 parameters. These specify the number of words in the memory; the number of bits per word; if the memory can be accessed as

```
IF (CE_nwv = '1' AND CENeg_nwv = '0') THEN
        IF (OENeg_nwv = '0' OR WENeg_nwv = '0') THEN

            Location := To_Nat(AddressIn);

            IF (OENeg_nwv = '0' AND WENeg_nwv = '1') THEN
               DataTemp  := MemData(Location);
               IF DataTemp >= 0 THEN
                  DataDrive := To_slv(DataTemp, DataWidth);
               ELSIF DataTemp = -2 THEN
                  DataDrive := (OTHERS => 'U');
               ELSE
                  DataDrive := (OTHERS => 'X');
               END IF;
            ELSIF (WENeg_nwv = '0') THEN
               IF Violation = '0' THEN
                  DataTemp := To_Nat(DataIn);
               ELSE
                  DataTemp := -1;
               END IF;
               MemData(Location) := DataTemp;
            END IF;
         END IF;
      END IF;
```

FIGURE 2.

subwords, the size of the subwords; the external file from which the memory may be initialized; and, whether or not that file is in a binary format.

```
-- VITAL Memory Declaration
      VARIABLE Memdat : VitalMemoryDataType :=
         VitalDeclareMemory (
            NoOfWords          => TotalLOC,
            NoOfBitsPerWord    => DataWidth,
            NoOfBitsPerSubWord => DataWidth,
--            MemoryLoadFile     => MemLoadFileName,
            BinaryLoadFile     => FALSE
         );
```

FIGURE 3.

Another major difference is in the way the behavior of the model is described. Instead of ordinary behavioral code, the memory package provides for the use of tables. Tables have two big advantages: they can be optimized for performance by the compiler; and, they are general enough to be placed in a package in a library from where they can be reused.

A portion of the table used in our generic memory model is illustrated in figure 4.

The path delay portion of the model is also changed. Using VITAL'95, path delays were scalar. As the size of the data bus increased, we were compelled to place the path delay procedure in a sepa-

```
CONSTANT Table_generic_sram : VitalMemoryTableType := (

        -- ------------------------------------------------------
        -- CE, CEN, OEN, WEN, Addr, DI, act, DO
        -- ------------------------------------------------------
        -- Address initiated read
          ( '1', '0', '0', '1', 'G', '-', 's', 'm' ),
          ( '1', '0', '0', '1', 'U', '-', 's', 'l' ),

        -- Output Enable initiated read
          ( '1', '0', 'N', '1', 'g', '-', 's', 'm' ),
          ( '1', '0', 'N', '1', 'u', '-', 's', 'l' ),
          ( '1', '0', '0', '1', 'g', '-', 's', 'm' ),
```

FIGURE 4.

rate process inside a generate statement in order to reduce the number of lines in the model, as shown in figure 5.

```
-------------------------------------------------------------------------
-- Path Delay Processes generated as a function of data width
-------------------------------------------------------------------------
    DataOut_Width : FOR i IN HiDbit DOWNTO 0 GENERATE
      DataOut_Delay : PROCESS (D_zd(i))
        VARIABLE D_GlitchData:VitalGlitchDataArrayType(HiDbit Downto 0);
      BEGIN
        VitalPathDelay01Z (
          OutSignal       => DataOut(i),
          OutSignalName   => "Data",
          OutTemp         => D_zd(i),
          Mode            => OnEvent,
          GlitchData      => D_GlitchData(i),
          Paths           => (
            0 => (InputChangeTime => OENeg_ipd'LAST_EVENT,
                  PathDelay       => tpd_OENeg_D0,
                  PathCondition   => TRUE),
            1 => (InputChangeTime => CENeg_ipd'LAST_EVENT,
                  PathDelay       => tpd_CENeg_D0,
                  PathCondition   => TRUE),
            2 => (InputChangeTime => AddressIn'LAST_EVENT,
                  PathDelay => VitalExtendToFillDelay(tpd_A0_D0),
                  PathCondition   => TRUE)
          )
        );

      END PROCESS;
    END GENERATE;
```

FIGURE 5.

The VITAL'2000 memory package has its own path delay procedures. They are vector based so they do not grow with the size of the data bus. This simplifies and potentially shortens the model. They also allow for additional functionality such as output retention timing that was difficult to

model with VITAL’95. The path delay procedures from the VITAL’2000 model are shown in figure 6.

```
VitalMemoryInitPathDelay (
        ScheduleDataArray   => DSchedData,
        OutputDataArray     => D_zd
    );

    VitalMemoryAddPathDelay (           -- #11
        ScheduleDataArray     => DSchedData,
        InputSignal           => AddressIn,
        OutputSignalName      => "D",
        InputChangeTimeArray  => AddrChangeArray,
        PathDelayArray        => Addr_D_Delay,
        ArcType               => CrossArc,
        PathCondition         => true
    );

    VitalMemoryAddPathDelay (           -- #14
        ScheduleDataArray     => DSchedData,
        InputSignal           => OENegIn,
        OutputSignalName      => "D",
        InputChangeTime       => OENegChange,
        PathDelayArray        => OENeg_D_Delay,
        ArcType               => CrossArc,
        PathCondition         => true,
        OutputRetainFlag      => false
    );

    VitalMemoryAddPathDelay (           -- #14
        ScheduleDataArray     => DSchedData,
        InputSignal           => CENegIn,
        OutputSignalName      => "D",
        InputChangeTime       => CENegChange,
        PathDelayArray        => CENeg_D_Delay,
        ArcType               => CrossArc,
        PathCondition         => true,
        OutputRetainFlag      => false
    );

    VitalMemorySchedulePathDelay (
        OutSignal           => DataOut,
        OutputSignalName    => "D",
        ScheduleDataArray   => DSchedData
    );
```

FIGURE 6.

Finally, the memory package has its own setup and hold and period/pulse width timing constraint check procedures. These were not used in the model tested.

The Difference

Two models of a generic SRAM were coded. The SRAM contained 4,194,304 words of 8 bits each. One model was written using traditional behavioral code and the Shelor method of memory declaration. The other model was written using the VITAL’2000 memory package including a state table based description of behavior.

Identical testbenches were written for each model. The testbenches ran for about 60 microseconds simulation time or about 3600 memory access cycles. In order to reduce the effect of the testbench on the measured performance, each testbench contained two instantiations of its model. Because

the result simulation times were still too fast to be measured accurately on an Ultra10, a 167MHz Ultra1 cpu was used for the comparison. The results are shown in table 1.

ModelSim version 5.5b was used. Times include SDF timing annotation.

	Shelor	VITAL'2000
compiled size of single model	23MB	15MB
compiled size of models and testbench	39MB	24MB
run time (167MHz Ultra1)	4.2 sec.	7.7 sec.

Table 1:

Conclusion

VITAL'2000 clearly provides a smaller memory footprint for memory simulation. This should result in a significant increase in capacity for board-level simulation of designs containing memory. However, run times are considerably slower. Since 5.5 is the first release of ModelSim to support the VITAL'2000 memory package, I assume performance will be improved in the future.

Model acquisition, not simulation performance has been the bottleneck in board-level simulation. I believe the VITAL'2000 memory package will make memory modeling easier and more accurate.