A Virtual Memory Management Scheme for Simulation Environment

Swapnajit Mittra

WIPRO Infotech Inc.

Abstract: *In this paper, a virtual memory management scheme is proposed for integrating large memory in the simulation environment. This is a novel extension to the Verilog-XL simulator from Cadence Design Systems. The scheme provides a user-transparent mechanism to instantiate a large chunk of memory without being limited by the main-memory of the simulating machine.*

I. Introduction

As modern day systems are gradually becoming more and more complex due to their wide functionalities, simulation of such systems have almost become indispensable before their implementation. Simulation gives the designer better scope of viewing the end-results of the design without going through the process of prototyping, thus avoiding valuable cycle time and money. The accuracy and precision of the results predicted by the simulation directly depends on the accuracy and versatility of the models of the individual component involved. Being an integral part of any automated system in general and computer system in particular, memory plays an important role in the performance of the system [1] and hence an efficient model of memory is desirable for simulating the system.

This paper describes a detailed environment for simulation with the memories using the Verilog-XL simulator from Cadence Design Systems. In the original simulator, the memory, as a component, can only be declared as a statically allocated variable [2]. This means the amount of memory component that can be declared and used in a Verilog code is directly limited by the amount of memory the simulation environment has (whose upper limit will be the amount of main-memory of the simulating computer). The proposed environment is a novel extension of the original simulator. It works on a virtual memory management scheme [3] which gives the designer the flexibility of choosing the amount of memory to be allocated statically yet does not restrict the access on ranges beyond that. It also features other advantages of virtual memory management scheme, e.g. user-transparency to addressing mechanism, theoretically infinite memory space etc.

This paper is divided into the following sections. Section II gives the background of the work. Section III details the working principle of the scheme and Section IV draws the conclusion thereof.

II. Background

The Verilog-XL simulator provides a mechanism for instantiating memory as a component in a Verilog code. It is declared as a two-dimensional array of Verilog primitive *reg*, each of which stores one bit of information. Figure 1 shows a typical declaration of memory. During the simulation, in this case, a total of *O((width_high ~ width_low) * (depth_high ~ depth_low))* bits will be stored statically. For simulating a system, having a large memory component this may be a hindrance depending upon the availability of the main memory in the simulating environment.

Also the programmer needs to use *$readmemb* (or *$readmemh*) systems calls each time a bulk change in the content of the memory is required, probably from different data files.

```
        module MemoryExample;
        reg [width_high : width_low] mymemory
[depth_high : depth_low];

        initial begin
                $readmemb("mem.txt",mymemory); /*
"mem.txt" is the datafile */

                        [ other staff ]
        end
        endmodule
```

Figure 1 : Example of declaration of memory in Verilog

0-8186-7082-7/95 $04.00 © 1995 IEEE

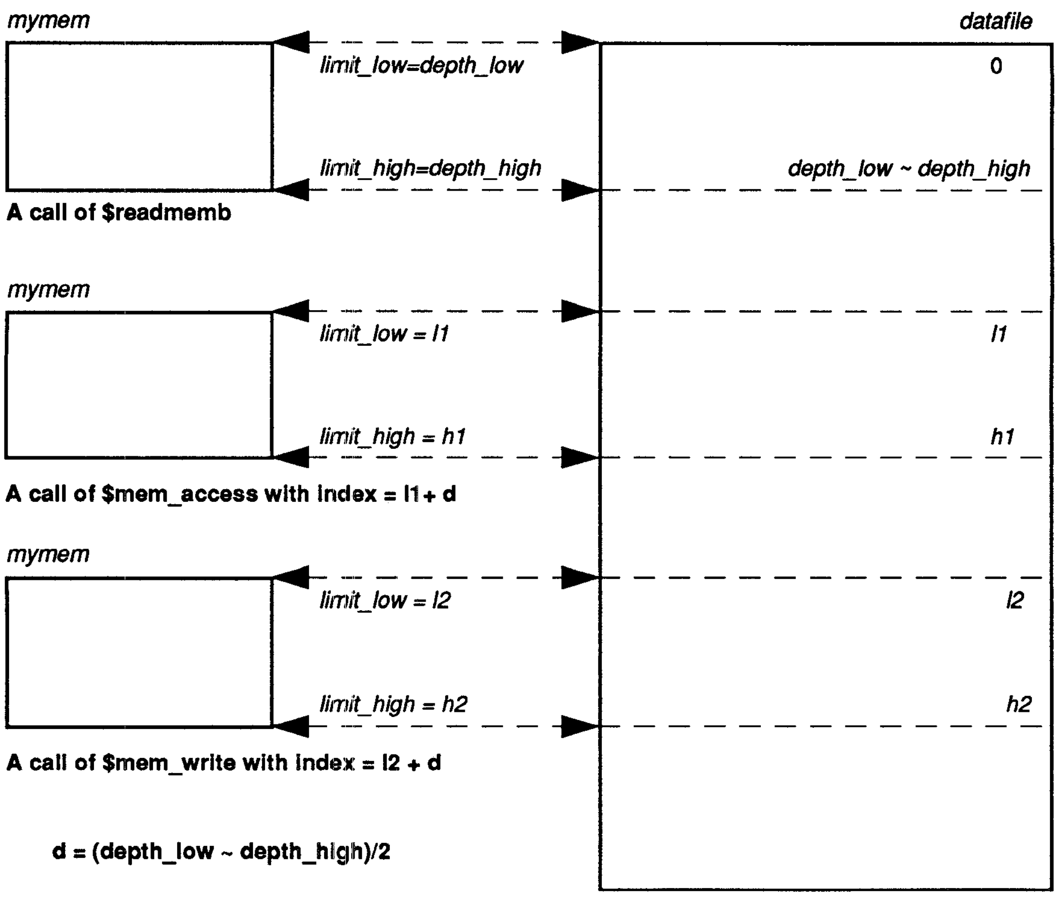

Figure 2 : The principle of operation of the scheme.

The proposed scheme augments the basic infrastructure for modeling the memory component by adding new functions to the simulator. The programmer has the flexibility to choose the amount of memory he wants to allocate statically. A reference to an element beyond the range of the currently occupied parts of memory will automatically upgrade it from an auxiliary secondary storage. Also data coherency is maintained throughout the operation to avoid the possibility of stealth data.

III. Modeling the Virtual Memory

The virtual memory management scheme for a memory component in the proposed environment works in a manner similar to cache memory protocol in a computer system. Figure 2 shows the basic principle of operation of the scheme.

The content of the memory is stored in a datafile in secondary storage (hard-disk). If the memory is uninitialised, this file is supposed to contain all x values. The programmer decides the amount of memory he wants to allocate statically by choosing proper value of *(depth_low ~ depth_high)* in the declaration of the memory component. Besides there are two variables *limit_low* and *limit_high*, which the programmer needs to declare and which are responsible to keep track of the range of the memory over the simulation time.

The programmer loads the content of the memory from the datafile using system call *$readmemb* (or *$readmemh*). Since the datafile is supposed to be larger than the size of the memory, hence only the beginning part of the file will be loaded into it. Also the variables *limit_low* and *limit_high* should be initialized to 0 and (*depth_low* ~ *depth_high*) in this case.

Once the memory component is initialized, there may be two cases, where it will be referred. They are

- Reading some particular location of the memory. This indexed element may not be currently loaded inside memory component.
- Changing some location of the memory. Here again the index may be out of range.

Each of these two cases are described separately below.

IIIA. Reading a Memory Location

Reading a particular memory location may lead to, as mentioned earlier, one of the two possible situations. If the required element is already loaded into the memory, it can be directly accessed. However, if it is not, we require to load that part from the datafile before accessing it.

To make this whole procedure transparent to the programmer, the custom task *$mem_access* is provided in the simulation environment. This task takes eight parameters as input. They are the name of the memory component, the name of the datafile, the variables *limit_low* and *limit_high*, values of *depth_low* and *depth_high*, the index of the element of the element to be read and the variable to which this value will be assigned.

This task follows an algorithm as shown in Figure 3. It checks the current availability of the data in the memory. If it is not available, goes to the datafile and reads the value. It is a common observation that in most of the cases, once the memory is referenced the subsequent memory references in a computer system are all localized. Thus to avoid too many references to the datafile wasting time on i/o operations, the task also loads the context of the present reference into the memory component. Once loaded with new data, the index of a particular location of the memory component no longer denotes the index of the data it contains. So *mem_access* changes the values of *limit_low* and *limit_high* to denote the current range of data which will be used in the subsequent call.

IIIB. Writing a Memory Location

Writing data in memory works in a way parallel to reading. The custom task *$mem_write* takes care of the user-transparency to the current availability of the data in the memory. But during a write, unlike the reading, even if the data to be changed is currently in the memory, to maintain the data coherency, the datafile should be updated.

Figure 4 shows the algorithm of a memory write. It takes the same set of eight inputs as is in *$mem_access* but this time the last parameter contains the value to be assigned to the destination location selected by the index.

The algorithm adopted here is similar to the write-through cache coherency protocol in a computer system. Since in a simulation environment different modules can ask for the access to the memory, the write-through scheme is preferred over the write-back one to avoid the need of multi-level caching. The task irrespective of the availability of the data, goes to the datafile and updates its content. Then it checks the content of the memory. If the required location is within the range between *limit_low* and *limit_high* then it is changed with the new value, otherwise that location (with its new content) is loaded from the datafile along with its context. Here also the values of *limit_low* and *limit_high* are properly updated, if required.

Example : Figure 5 shows an example of Verilog code using the custom tasks *$mem_access* and *$mem_write*. Initially the first part of the file is loaded using *$readmemb* and *limit_high* and *limit_low* are initialised to 0 and *depth_low* ~ *depth_high* (which in this case is 3-0 = 3). Next *index* is set to 0 and one *$mem_access* is

```
input :memory_name, datafile_name, limit_low,
limit_high, depth_low, depth_high, index, read_data;
output : read_data;
procedure :
        if (index lies between limit_low and limit_high )
          read_data = memory_name[index];
        else
          begin
          open the datafile;
          retrieve the indexed element and assign it to
read_data;
          load the context into the memory;
          limit_low = lower index of the context;
          limit_high = upper index of the context;
          close the datafile;
          end
```

Figure 3 : Algorithm for a memory read

```
input : memory_name, datafile_name, limit_low,
limit_high, depth_low, depth_high, index, write_data;
output : write_data;
```

```
procedure :
        open the datafile;
        go to the indexed location;
        modify the content of the location with
write_data;
        close the datafile;
        if (index lies between limit_low and limit_high )
          memory_name[index] = write_data;
        else
      begin
      open the datafile;
      retrieve the indexed element and assign it to
read_data;
      load the context into the memory;
      limit_low = lower index of the context;
      limit_high = upper index of the context;
      close the datafile
      end
```

Figure 4 : Algorithm for a memory write

done. Since this element is already loaded in the memory so the datafile is not referred. After this another *$mem_access* call with *index = 5* loads the 5th element along with the context into the memory. It also changes the values of *limit_low* and *limit_high* to 4 and 7 respectively. Similarly, a call of *$mem_write* to change the value of 0th element will again reload the previous context with *limit_low* and *limit_high* getting values 0 and 3 respectively.

```
`define depth_low 0
`define depth_high 3
module VirtualMemoryExample;
integer index, limit_low, limit_high;
reg [7:0] mymem[`depth_high:`depth_low];
reg [7:0] variable;

initial begin
        $readmemb("mem.txt",mymem);
        limit_low = `h0;
        limit_high = `depth_high - `depth_low ;

        index = `h0;
        $mem_access(mymem[index-
limit_low],"mem.txt",limit_low,limit_high,
`depth_low,`depth_high,index,variable);
                    [... other staff ...]
        index = `h5;
        $mem_access(mymem[index-
limit_low],"mem.txt",limit_low,limit_high,
depth_low,`depth_high,index,variable);
                    [... other staff ...]
        index = `h0;
        variable = 8`b10010010; /* some value I want
to write at loc. 0 */
        $mem_write(mymem[index-
limit_low],"mem.txt",limit_low,limit_high,
`depth_low,`depth_high,index,variable);
        $finish;
end
endmodule
```

Figure 5 : Using the custom tasks

IV. Conclusion

A virtual memory management scheme for simulation environment is proposed in this paper. the proposed scheme provides the capability of having theoretically infinite memory space with flexibility of user-defined static allocation for it. However, since swapping the data between the memory and the secondary storage is a slow process, hence the choice of the amount of memory to be allocated statically for the component will be a trade-off between the avilability of the main-memory of the simulating machine and the desired simulation speed.

For universal applicability, the interface with the datafile is directly implemented inside Verilog-XL using the Programming Language Interface (PLI). PLI is used to write two custom system calls, one each for reading from and writing to the memory. These system calls provide an easy-to-use interface to the user-transparent virtual memory mechanism and can be used in simulating a system with large amount of memory, without being limited by the main memory of the simulating machine.

Acknowledgement

I am greatful to Stephen Tell of University of North Carolina at Chapel Hill and Marco Zelada of Microchip Technology Inc. for their valuable suggestions regarding PLI routines. Indebted I also remain to the QCICE-team in Sequent Computer Systems Inc and VLSI-design group of WIPRO Infotech Ltd, India for all the helps and supports I got. Finally my sincere thanks to Mudit Mathur and Gary Hornbuckle for their help with Verilog grammar.

References

1. Knuth, D. E., " The Art of Computer Programming 1: Fundamental Algorithms",Section 2.5, Addision Wesley, Reading, MA.

2. Verilog-XL Version 2.0, Reference Manual, Cadence Design System, March, 1994.

3. Denning, P. J., "Virtual Memory", Computing Surveys 2, 3 (September), pp.153-189.