A Behavioral Design Approach in Verilog Hardware Description Language

Chiuchisan, Iuliana
Potorac, Alin Dan

1st October 2007

"Stefan cel Mare" University of Suceava
13, University Street, RO-720225 Suceava
iuliap@eed.usv.ro, alinp@eed.usv.ro

Abstract

The paper presents a behavioral design of a random-access memory (RAM) using Verilog as hardware description language. As IP part of a larger project, the memory design is described here using the concept of a "module" in a behavioral specification at the RTL level, trying to push the description to a more abstract approach. The operations that we performed on the memory, in this implementation, are: reading the information that is saved in a external file and writing in locations of the memory and then saving results in a external file. The RAM module was tested by using the stimulus module and the results were monitored to verify the design.

Keywords: *Verilog, Very-large-scale integration, Random-Access Memory, Behavioral design, Module, Test vectors, Results analysis.*

1 The background

The behavioral description level in Hardware Description Languages is a modern concept usually associated with most abstract languages as VHDL or ABEL. The paper is demonstrating the ability of using behavioral description of a logic design, usually associated with abstract languages, in a less abstract environment as Verilog is. The language supports the early conceptual stages of design with its

behavioral level of abstraction and the later implementation stages with its structural abstractions. The language includes hierarchical constructs, allowing the designer to control a description's complexity.
Since Verilog was originally designed in the winter of 1983/84 as a proprietary verification/simulation product, no behavioral tools were included. Later, several other proprietary analysis tools were developed around the language, including a fault simulator and a timing analyzer. More recently, Verilog has also provided the input specification for logic and behavioral synthesis tools. The Verilog language has been instrumental in providing consistency across these tools.
The paper is presenting a behavioral Verilog implementation for a memory block included in a larger project. It can be reused as IP module in future development.

2 Motivation

Digital systems are highly complex. At their most detailed level, they may consist of millions of elements, as would be the case if we viewed a system as a collection of logic gates or pass transistors. From a more abstract viewpoint, these elements may be grouped into a handful of functional components such as cache memories, floating point units, signal processors, or real-time controllers. Hardware description languages have become a helpful tool to design systems with large number of elements and wide range of electronic and logical abstractions.
The creative process of digital system design begins with a conceptual idea of a logical system to be built, a set of constraints that the final implementation must meet, and a set of primitive components from which to build the system. The design is typically divided into many smaller subparts and each subpart is further divided, until the whole design is specified in terms of known primitive components [5].
The Verilog language describes a digital system as a set of modules. Modules can represent bits of hardware ranging from simple gates to complete systems, e. g. a microprocessor [2].
A module represents a logical unit that can be described either by specifying its internal logical structure — for instance describing the actual logic gates it is comprised of, or by describing its behavior in a program-like manner — in this case focusing on what the module does.
A *behavioral* model of a module is an abstraction of how the module works. The outputs of the module are described in relation to its inputs, but no effort is made to describe how the module is implemented in terms of structural logic gates. The behavioral model can be the starting point for synthesizing several alternate structural implementations of the behavior [6].

3 Random-Access Memory Implementation

In this sub-design we will describe, using Verilog environment and its behavioral level, a RAM with four addresses lines and four bidirectional data lines, $2^4 \times 4$ bits. The figure 1 shows a logic diagram for a 16-word by 4-bit random access read/write memory.

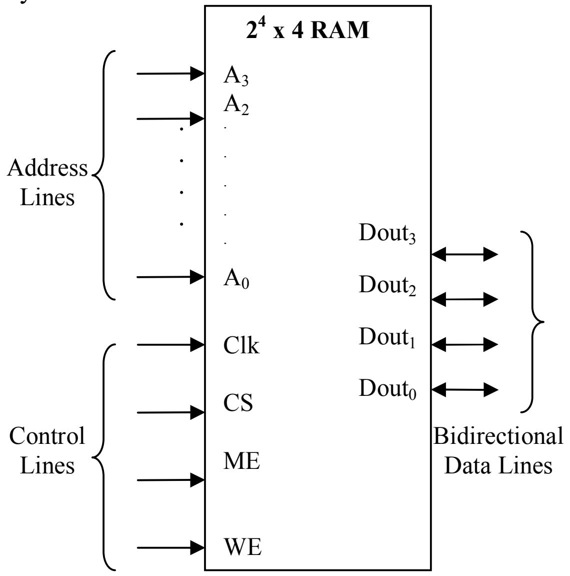

Figure 1. The diagram for RAM with $2^4 \times 4$ bits

The design is described using the concept of *module*. The module is conceptualized as consisting of two parts: the port declarations and the module body.
The *port declarations* represent the external interface to the module. Inputs to the memory consist of four address lines, four data input lines, a write enable line, a chip select and a memory enable line. The four binary address inputs are decoded internally to select each of the 16 possible word locations.

```
module RAM(Address,Clk,CS,ME,WE,Dout);
input[3:0] Address;
input Clk;
input CS;
input ME;
input WE;
inout[3:0] Dout;
reg[3:0] memory[15:0];  // Memory block 16k x 4
reg[3:0] Data_Read;
assign Dout=Data_Read;
…
endmodule
```

The module body represents the internal description of the module - its behavior, in this case. The name of the module is just an arbitrary label invented by the user – *RAM*, and it does not correspond to a name pre-defined in a Verilog component library.
The ports may correspond to a pin on an IC, an edge connector on a board, or any logical channel of communication with a block of hardware.
Each port declaration includes the name of one or more ports and the direction that information is allowed to flow through the ports:

- input – Address, Clock Signal (Clk), Chip Select (CS), Memory Enable (ME), Write Enable (WE);
- inout – Bidirectional Port (Dout).

Address Operation: Address inputs must be stable to the rising edge of memory enable input.
Write Operation: Information present at the data inputs is written into the memory at the selected address by bringing write enable high.
For write operation the data inputs can be:

- read from a external file (“init.dat”);
- equal with the address values;
- random values;
- all zero values;
- unknown values.

```
integer i;
initial begin
if(init=="file")
      $readmemb("init.dat", memory);
else begin
if(init=="address") begin
for(i=0;i<=15;i=i+1)
      memory[i]=i;
end
else begin
if(init=="random") begin
for(i=0;i<=15;i=i+1)
      memory[i]= $random;
end
else begin
if(init=="zero") begin
for(i=0;i<=15;i=i+1)
      memory[i]=0;
end
else begin
if(init=="x") begin
for(i=0;i<=15;i=i+1)
```

```
      memory[i]=4'bx;
end
else begin
      $display("Erorr: Incorrect parameter init=%s",init);
      $stop;
end   end   end   end   end    end
```

Read Operation: The information which was written into the memory is read out at the four outputs. This is accomplished by selecting the desired address and bringing memory enable high and write enable low. When the device is writing or disabled the outputs assumes a TRI-STATE (Hi-Z) condition.

```
always@(posedge Clk) begin
if(CS==1) begin
if(WE==1) begin
      Data_Read<=4'bz;
     //Write in the memory
      memory[Address]<=Dout;
     end
else begin
if(ME==1)
     //Read from the memory
      Data_Read<=memory[Address];
end     end
else Data_Read<=4'bz;
end
```

The basic essence of this behavioral model is the *process.* A process can be thought of as an independent thread of control, involving only one repeated action. The basic Verilog statement for describing a process is the *always* construct.

4 Simulation

The design described earlier is simulated for functionality and fully debugged. Translation of the debugged design into the corresponding hardware circuit (using FPGA or ASIC) is called synthesis.
Testing is essential for the VLSI design process as with any hardware circuit. It has two dimensions to it – functional tests and timing tests. Testing or functional verification is carried out by setting up a “test bench” for the design. The test bench will have the design instantiated in it and will generate necessary test signals and apply them to the instantiated design. The outputs from the design are brought back to the test bench for further analysis. The input signal, waveforms and sequences required for testing are all to be decided in advance and the test bench configured based on the same [6].

4.1 The test bench

Once the RAM design is completed, it must be tested for all its functional aspects. The functionality of the design block can be tested by applying stimulus and checking results. The test bench is done at the behavioral level. The constructs are flexible enough to allow all types of test signals to be generated.
For the stimulus block we use a number of tasks for write and read in/from memory that facilitates control and flow of the testing process.
Verilog *tasks* are as constructs analogous to subroutine in a software program and it allows for the behavioral description of a module to be broken into more-manageable parts. A task is defined within a module and can be called as many times as desired within a procedural block.

```
task Read;
      input[3:0] Address_task;
begin
      Data_Write=4'bz;
      Address<=Address_task;
//Control signals for read operation
      CS<=1'b1;
      WE<=1'b0;
      ME<=1'b1;
      @(posedge Clk);
      #5 $display("%tRead from the memory:Address=%d,Data=%h
                  H",$time, Address_task, Dout);
end    endtask

task Write;
      input[3:0] Address_task;
      input[3:0] Data_W;
begin
      $display("%t Write in the memory: Address=%d, Data=%h
               H",$time, Address_task, Data_W);
      Address<=Address_task;
// The value who is written at the specified location
      Data_Write<=Data_W;
//Control signals for write operation
      CS<=1'b1;
      WE<=1'b1;
      ME<=1'b1;
      @(posedge Clk);
end     endtask
```

4.2 Test Vectors

In this code fragment, the stimulus and response capture are going to be coded using a pair of *initial* blocks used for monitoring, generating waveforms (clock pulses) and processes which are executed once in a simulation.

4.3 Instances

A module provides a template from which we can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters and I/O interface. The process of creating objects from a module template is called *instantiation*, and the objects are called *instances* [7].
Each instance is an independent, concurrently active copy of a module. Each module instance consists of the name of the module being instanced (e.g. *RAM*), an instance name (unique to that instance within the current module – *memory*) and a port connection list.
The module port connections can be given in order (positional mapping), or the ports can be explicitly named as they are connected (named mapping). Named mapping is usually preferred for long connection lists as it makes errors less likely [9].

```
// instantiate the design block
RAM memory(Address,Clk,CS,ME,WE,Dout);
defparam memory.init="file";
/* "file": all locations are filled with the data stored in a
external file named "init.dat"
"address": all locations are filled with the address values
"random": all locations are filled with the random values
"zero": all locations are filled with the zero value
"x": all locations are filled with the unknown values    */
initial begin
      @(posedge Clk);
      Display_Memory("results.dat");
      Write(4'd10,4'b0001);
      Write(4'd11,4'b0010);
      Write(4'd12,4'b0011);
      Write(4'd13,4'b0100);
      Read(4'd10);
      Read(4'd11);
      Read(4'd12);
      Read(4'd13);
      Display_Memory("results.dat");
      @(posedge Clk);
      $display("%t Finished Simulation", $time );
      $stop;  end
```

The initial block above does six controlling activities during the simulation run:

- Initializes the memory with the information, using the parameter *init*;
- Displays the initial content of the memory in a external file named “results.dat”;
- Writes another information in the memory;
- Reads the changed information;
- Displays the content of the memory in “results.dat”;
- Stops the simulation at the specified time.

5 Results Analysis

For the results analysis we described a Verilog task that reads data from the memory and displays the results in a external file named “results.dat”.

```
task Diplay_Memory;
      input[0:100] file;
      integer k;
      integer file_id;
begin
      file_id=$fopen(file);
      $display ("%t Read from the memory", $time);
      for(k=0;k<=15;k=k+1) begin
      $fdisplay(file_id, "Address=%d, Data=%h H", k,
                memory. memory [k]);
      $display("Address=%d, Data=%h H", k, memory.memory[k]);
end
      $fclose(file_id);
end
endtask
```

File used to write in the memory (init.dat)	File used to read from the memory (results.dat)
1111	Address= 0, Data= f H
1110	Address = 1, Data= e H
1101	Address = 2, Data= d H
1100	Address = 3, Data= c H
1011	Address = 4, Data= b H
1010	Address = 5, Data= a H
1001	Address = 6, Data= 9 H
1000	Address = 7, Data= 8 H
0111	Address = 8, Data= 7 H
0110	Address = 9, Data= 6 H
0101	Address = 10, Data= 1 H
0100	Address = 11, Data= 2 H

0011	Address =12, Data= 3 H
0010	Address =13, Data= 4 H
0001	Address =14, Data= 1 H
0000	Address =15, Data= 0 H

Simulation results can alternately be viewed as waveforms. The code for the test bench is simulated using HDL simulator. The figure 2 shows how the Verilog tasks described to write and read in/from the memory, has created a waveform sequence for the RAM signals.

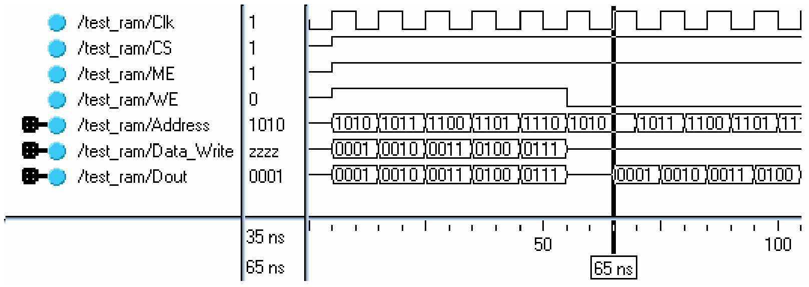

Figure 2. Write and read operations

6 Conclusion

The behavioral model can be the starting point for synthesizing several alternate structural implementations of the behavior. Behavioral models are useful early in the design process. At this point, a designer is more concerned with simulating the system's intended behavior to understand its gross performance characteristics with little regard to its final implementation. Later, structural models with accurate detail of the final implementation are substituted and re-simulated to demonstrate functional and timing correctness. In terms of the design process, the key point is that it is often useful to describe and simulate a module using a behavioral description before deciding on the module's actual structural implementation.
Simulation of the Verilog source before synthesis allows a direct form of testing the design and finding simple run-time bugs before being tested in hardware. To allow ease of simulation, the RAM was replaced with accurate timing model and file to represent their behavior and storage. Functionality could easily be tested by writing programs in byte-code and saved as a file to be automatically run by the simulation. Behavioral implementation of a RAM module demonstrates the Verilog environment possibility to host abstract descriptions and accordingly to be used for complex abstract designs.

Acknowledgment

The work presented in this paper has been supported by the national founding grant Ref. No. 56-CEEX/2006. project TERAPERS.

References

[1] Nicula, D., Toacse, Gh. (2005) „Electronica Digitala” , vol. 2, Tehnic Ed. Bucharest, pp. 202-210

[2] Hyde, D.C. (1997) „Handbook on Verilog HDL”, Bucknell University, Lewisburg, USA, pp 4-11

[3] Pellerin, D. (1998) „An Introduction to HDLs for Simulation and Synthesis”, Protel Technology Inc., Provo, USA, pp 5-20

[4] Smith, M.J.S. (1997) „Application-Specific Integrated Circuits”, Addison Wesley Longman, pp 5-20

[5] Thomas, D.E., Moorby P.R. (2002) „The Verilog Hardware Description Language”, ECE Department, Carnegie Mellon University, Pittsburgh, USA, pp 1- 36

[6] Padmanabhan, T.R. Sudari, B. (2003) „Design Through Verilog HDL”, IEEE Press, USA, pp 11-27

[7] Palnitkar, S. (1996) „Verilog HDL A guide to Digital Design and Synthesis”, SunSoft Press, USA, pp 2-26

[8] Lee, W.F. (2003) “Verilog Coding for Logic Synthesis”, USA, pp 3-40

[9] Patentariu, I., Potorac, A. D. (2004) „Some Consideration On 8-level HDL Stack Implementation”, The 7th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS, Faculty of Electrical Engineering, Ştefan cel Mare University of Suceava, pp 303-308.

[10] Patentariu, I., Potorac, A. D. (2003) “Hardware Description Languages, A Comparative Approach”, Advances in Electrical and Computer Engineering, Faculty of Electrical Engineering, “Ştefan cel Mare” University of Suceava, vol.3 (10), no.1 (19), pp 84-89

[11] Wakerly, J. F. (2002) “Digital Design, Principles and Practices”, 3rd Edition, Teora, pp 847-854

[12] www.doulos.com

[13] www.verilog.net