7th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS
Suceava, Romania, May 27 – 29, 2004

SOME CONSIDERATIIONS ON 8-LEVEL HDL STACK IMPLEMENTATION

Iuliana PATENTARIU
Alin Dan POTORAC
"Stefan cel Mare" University of Suceava
str.Universitatii nr.13, RO-720225 Suceava
iuliap@eed.usv.ro, alinp@eed.usv.ro

Abstract. *This paper presents an 8-level stack implementation in hardware description language. Applied to 8-level stack design, Verilog is used for logic synthesis, for test analysis, for timing analysis and for verification through simulation. The stack design is described using the concept of a 'module' in behavioral specification. The operations that we performed on the stack, in this implementation, are: stack push, stack pop and stack reset. The stack module it was tested by using the idea of the stimulus module and the results was monitored to verify the design.*
Keywords: *Verilog, stack, push, pop, test vectors, results analysis.*

1. Introduction

Verilog Hardware Description Language

There are now two industry standard hardware description languages, VHDL and Verilog. The complexity of ASIC and FPGA designs has meant an increase in the number of specialist design consultants with specific tools and with their own libraries of macro and mega cells written in either VHDL or Verilog. As a result, it is important that designers know both VHDL and Verilog and that EDA tools vendors provide tools that provide an environment allowing both languages to be used in unison [1].
The Verilog HDL is an IEEE standard - number 1364. The standard document is known as the Language Reference Manual, or LRM. This is the complete authoritative definition of the Verilog HDL. IEEE Std 1364 also defines the Programming Language Interface, or PLI. This is a collection of software routines which permit a bidirectional interface between Verilog and other languages (usually C).
In the mid-80's, Gateway Design Automation developed a logic simulator, Verilog-XL, to simulate designs described using their proprietary Verilog HDL. Cadence have since bought Gateway and retained the Verilog-XL simulator, but the language, Verilog HDL, is now maintained by Open Verilog International (OVI) [5].

Evolution of HDL Concepts

The history of the Verilog HDL goes back to the 1980s, when a company called Gateway Design Automation developed a logic simulator, Verilog-XL, and with it a hardware description language [5].
Cadence Design Systems acquired Gateway in 1989, and with it the rights to the language and the simulator. In 1990, Cadence put the language (but not the simulator) into the public domain, with the intention that it should become a standard, non-proprietary language [4]. Cadence was motivated to open the language to the Public Domain with the expectation that the market for Verilog HDL-related software products would grow more rapidly with broader acceptance of the language. Cadence realized that Verilog HDL users wanted other software and service companies to embrace the language and develop Verilog-supported design tools [2]. The Verilog HDL is now maintained by a non profit making organization, Open Verilog International (OVI). OVI had the task of taking the language through the IEEE standardization procedure. In december 1995 Verilog HDL became IEEE Std. 1364 -1995 [5].

2. Stack Implementation

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. Modules can either be specified behaviorally or structurally (or a combination of the two). A *behavioral specification* defines the behavior of a digital system (module) using traditional programming language constructs, e. g., *if* assignment statements. A *structural specification* expresses the behavior of a digital system (module) as a hierarchical interconnection of submodules. At the bottom of the hierarchy the components must be primitives or specified behaviorally. Verilog primitives include gates, e.g., *nand*, as well as pass transistors (switches) [2].
In this section we will design a 8-level stack in Verilog language and we describe it in behavioral specification. Stack is a simple data structure that may be implemented in hardware description languages, as Verilog.
Below is a logic diagram for an 8-level stack.

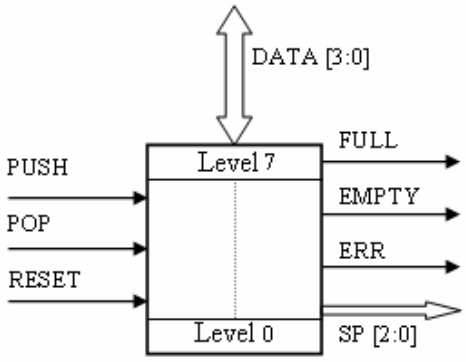

Figure 1. The diagram for 8-level stack

The design is described using the concept of a *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 – Push, Pop, Reset, DataIO, SP, Full, Empty, Err.

```
// Verilog code for stack
module Stack (DataIO, Reset, Push,
Pop, SP, Full, Empty, Err);
/* declare input, output and inout
ports */
inout [3:0] DataIO;
input Push,Pop,Reset;
output Full, Empty,Err;
output [2:0] SP;
// declare registers
reg Full,Empty,Err;
reg [2:0] SP;
reg [3:0] Stack[7:0];
reg [3:0] DataR;
/* continuous assignment of  DataIO to
DataR register, with delay 0 */
wire [3:0]  #(0) DataIO = DataR;
...
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 – *Stack*, 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** - *Push*, *Pop*, *Reset*;
- **output** - *SP*, *Empty*, *Full*, *Err*;
- **inout** - *DataIO*.

The module definition is terminated by the keyword *endmodule*.

Stack Description

At any given instant in time, only the item on the top of the stack is accessible. Any other item in the stack is blocked from convenient access by all of the items above it. To access items below the item on the top of the stack, we must sequentially remove items from the top until we reach the item we want to access.
In this implementation the stack needs to be eight entries deep and four bits wide. That means we will need a collection of 8 registers.
There are two operations we can perform on the stack, *push* and *pop*. One way to implement a stack is to actually move data between "adjacent" registers and another way to do it, as shown in this implementation, is to keep the data stationary and adjust a pointer (Stack Pointer - SP).

There are three commands:

- **Push** – this input should cause the four-bit binary input to be pushed onto the stack;
- **Pop** – this input should cause the top item on the stack to be popped and stored in *DataR* register;
- **Reset** – this input should initialize the entire stack to a known state;

The statements from the inside of the stack will be made by two blocks *always* that contains statements which are only executed when any of the variables in the list of variables change:
- the first *always* is controlled by positive edge of Push, Pop and Reset; in this block will be initialize the stack and will be push and pop data;

```
always @ (posedge Push or posedge Pop
or posedge Reset)
begin
...
end
```

- the second *always* is controlled by negative edge of Pop; in this block the *DataR* register goes in High Impedance and the stack will be able to receive data in case of push instruction.

```
always @ (negedge Pop)
begin
      DataR = 4'bzzzz;
end
```

The list of variables is called the sensitivity list, because this construct is sensitive to their change. The block *always* will loop until simulation is over.

Stack PUSH

A push instruction will cause the 4-bit input (*DataIO*) to be pushed onto the stack and the stack pointer *SP* will be incremented.
After a depth of 7 is reached, the stack is full and the *Full* output goes HIGH. If further pushes are attempted when the stack is full, the *Error* output goes HIGH, and the stack information at the top of the stack (7) will be over-written.

```
if (Push==1)  begin
    // when the stack is empty
     if (Empty==1)
     begin
      Stack[SP] = DataIO;
      Empty = 0;
      if (Err==1)
          Err=0;
     end
     else   // when the stack is full
     if (Full==1)
     begin
      Stack[SP] = DataIO;
      Err = 1;
     end
     else
     begin
      SP = SP +1;
      Stack [SP] = DataIO;
      if (SP == 3'b111)
          Full = 1;
     end
end
```

Stack POP

A pop instruction will cause the top item on the stack to be popped and the stack pointer *SP* will be decremented. Further pops from a not empty stack will place the bottom data on the *DataR* register. If further pops are attempted when the stack is empty, the *Error* and *Empty* output goes HIGH.

```
if(Pop==1) begin
/* if SP indicates the last location
but the stack is not empty */
    if ((SP == 3'b000) && (Empty!=1)
    begin
      DataR = Stack[SP];
      Empty = 1;
    end
    else   // if the stack is emtpy
    if(Empty==1)
    begin
      DataR = Stack[SP];
      Err = 1;
    end
    else
    begin
          DataR = Stack[SP];
      if (SP != 3'b000)
          SP = SP-1;
      // if the stack is full
      if (Err==1)  Err = 0;
      if (Full==1) Full = 0;
    end
end
```

Stack RESET

The reset instruction will initialize the entire stack to a known state: the *DataR* output goes in High Impedance and the other outputs (Stack Pointer, Full, Empty, Err) goes LOW.

```
if (Reset==1)
begin
      DataR = 4'bzzzz;
      SP = 3'b0;
      Full = 0;
      Empty = 0;
      Err = 0;
  end
```

The test bench

Once the *stack* module has been designed it can be tested by applying test inputs. This is idea of the stimulus module. It calls the design module and uses its functionality then results can be monitored to verify its design. Below is the stimulus for the 8-level stack.

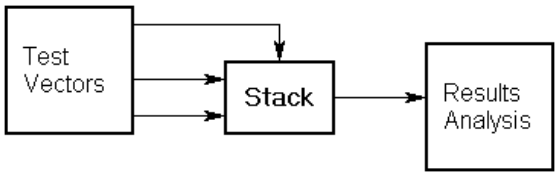

Figure 2. The stimulus for the stack

A. 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 wave forms (clock pulses) and processes which are executed once in a simulation.
An *initial* block consists of a statement or a group of statements which will be executed only once at simulation time 0. The *initial* blocks execute concurrently and independently.

```
module StackTest;
/* Nothing else calls it to use its
functionality, so it doesn't need a
port list */
...
initial
begin
// Stimulus
...
end
initial
begin
// Analysis
...
end
endmodule
```

The wires used in continuous assignments must be declared. We want to be able to assign values to the inputs and values to be driven into the output. It follows that the inputs must be *reg* data types and the output must be a *wire*.

```
reg [3:0] DataR;
reg Push, Pop, Reset;
wire Full, Empty, Err;
wire [2:0] SP;
/* continuous assignment of DataIO to
DataR register, with delay 0 */
wire [3:0] #(0) DataIO = DataR;
```

In a *module instance*, the ports defined in the module interface are connected to wires in the instantiating module through the use of port mapping.
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. *Stack*), an instance name (unique to that instance within the current module - *StackTest*) 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.

```
Stack StackTest (DataIO, Reset, Push,
Pop, SP, Full, Empty, Err);
```

In the stimulus *initial* block, we need to generate waveform on the *Push*, *Pop*, *Reset* inputs and initialize *DataR* register.

```
initial  begin
// initialize  registers
    NrIter = 8;
    # 0  Reset = 1;
/* #2 means do after two unit of
simulation  time */
```

```
    # 2  Reset = 0;
// after reset the stack is empty
   Pop = 1;
   # 1 Pop = 0;
// initialize register DataR
   DataR = 4'b0000;
// push data onto the stack
   for (i=0; i<=NrIter; i=i+1)
   begin
        # 2 Push = 1;
        # 1 Push = 0;
        DataR = DataR +1;
   end
// getting data from the stack
   # 1 DataR = 4'bzzzz;
/* pops the stack and stores the
contents  in DataR register */
   for (i=0; i<=NrIter; i=i+1)
   begin
      # 2 Pop = 1;
      # 1 Pop = 0;
   end
end
```

B. Results Analysis

The Response initial block can be described very easily in Verilog as we can benefit from a built-in Verilog system tasks.

```
initial  // Response
begin
$display("Push   Pop   Reset   DataIO
Empty  Full  Error  SP");
$monitor($time, ,Push, ,Pop, ,Reset,
,   ,DataIO,   ,Empty,   ,Full,   ,Err,
,SP);
end
```

$display system task allows the designer to print a message.

$monitor is a system task that is part of the Verilog language. Its mission is to print values to the screen. The $monitor task is executed whenever any one of its arguments changes.

$time is a system function and it returns the current simulation time. In the above example, $time is an argument to $monitor. However, $time changing does not cause $monitor to execute.

The space at position 2 in the argument list ensures that a space is printed to the screen after the value of $time each time $monitor is executed.

This is the outputs created by $monitor in *StackTest* testbench:

Time	Push	Pop	Reset	DataR	DataIO	Empty	Full	Error	SP
0	x	x	1	x	x	0	0	0	0
2	x	1	0	x	x	1	0	0	0
3	x	0	0	0	0	1	0	0	0
5	1	0	0	0	0	0	0	0	0
6	0	0	0	1	1	0	0	0	0
8	1	0	0	1	1	0	0	0	1
9	0	0	0	2	2	0	0	0	1
11	1	0	0	2	2	0	0	0	2
12	0	0	0	3	3	0	0	0	2
14	1	0	0	3	3	0	0	0	3
15	0	0	0	4	4	0	0	0	3
17	1	0	0	4	4	0	0	0	4
18	0	0	0	5	5	0	0	0	4
20	1	0	0	5	5	0	0	0	5
21	0	0	0	6	6	0	0	0	5
23	1	0	0	6	6	0	0	0	6
24	0	0	0	7	7	0	0	0	6
26	1	0	0	7	7	0	1	0	7
27	0	0	0	8	8	0	1	0	7
28	0	0	0	z	z	0	1	0	7
30	0	1	0	z	7	0	0	0	6
31	0	0	0	z	z	0	0	0	6
33	0	1	0	z	6	0	0	0	5
34	0	0	0	z	z	0	0	0	5
36	0	1	0	z	5	0	0	0	4
37	0	0	0	z	z	0	0	0	4
39	0	1	0	z	4	0	0	0	3
40	0	0	0	z	z	0	0	0	3
42	0	1	0	z	3	0	0	0	2
43	0	0	0	z	z	0	0	0	2
45	0	1	0	z	2	0	0	0	1
46	0	0	0	z	z	0	0	0	1
48	0	1	0	z	1	0	0	0	0
49	0	0	0	z	z	0	0	0	0
51	0	1	0	z	0	1	0	0	0
52	0	0	0	z	z	1	0	0	0

Another system task is **$stop** and it puts the simulator into a halt mode and passes control to the user.

```
initial begin
/* Will stop the execution after
100 simulation units */
   #100  $stop;
end
```

The figures show how the initial block has created a waveform sequence for the stack signals, for push and pop instruction. The *Push* instruction pushes the contents data onto the stack and when the stack pointer (SP) is reached 7, the stack is full and the *Full* output goes HIGH.

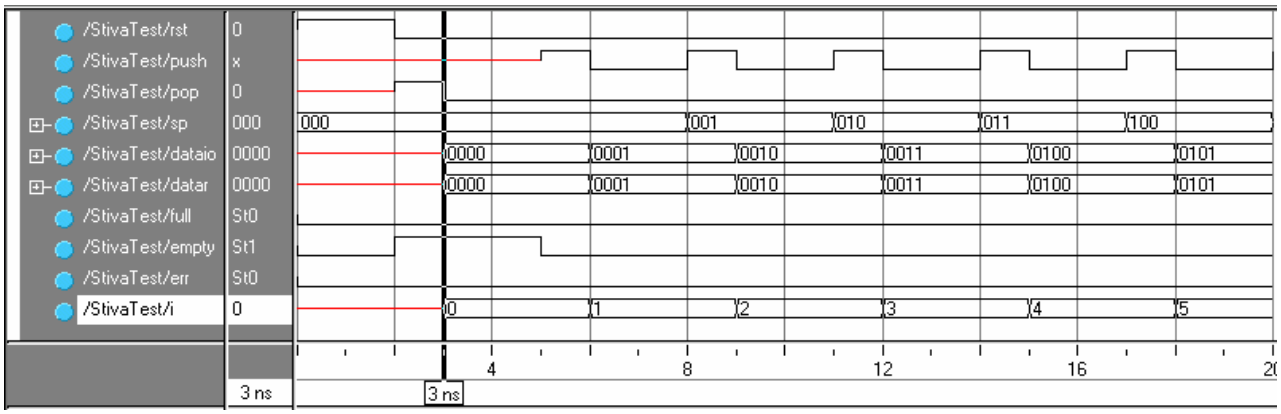

Figure 3. Stack Push

The *Pop* instruction pops the stack and stores the contents in *DataR* register and when the stack pointer (SP) is reached 0 the stack is empty and the output *Empty* goes HIGH.

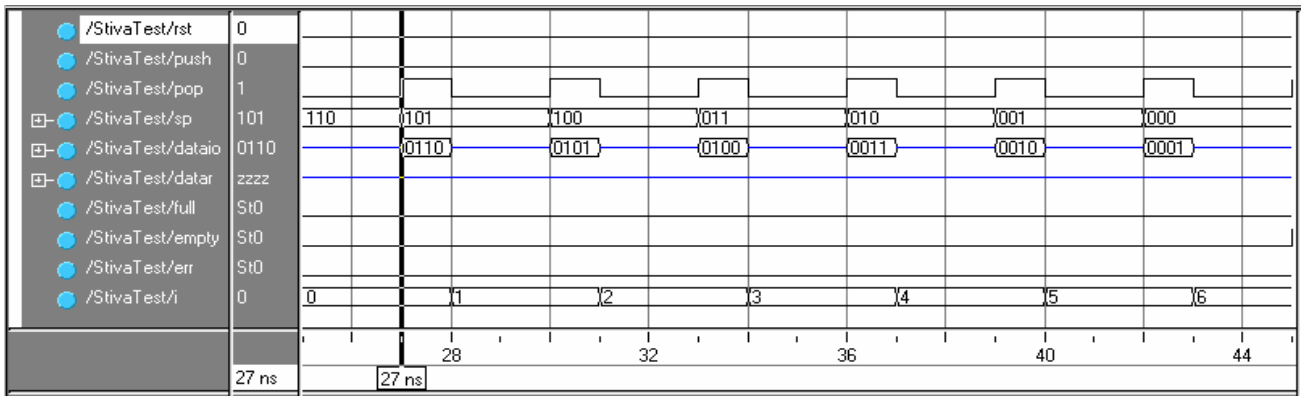

Figure 4. Stack Pop

3. Conclusion

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 stack 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.

References

[1] Iuliana Patentariu, Alin Dan Potorac (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), ISSN 1582-7445

[2] Hyde, D.C. (1997) *Handbook on Verilog HDL*, Bucknell University, Lewisburg, USA

[3] Pellerin D. (1998) *An Introduction to HDLs for Simulation and Synthesis*, Protel Technology Inc., Provo, USA

[4] Smith, M.J.S. (1997) *Application-Specific Integrated Circuits*, ISBN: 0-201-50022-1, Addison Wesley Longman

[5] www.doulos.com

[6] www.verilog.net