Efficient Computer Architecture Design Using Hardware Descriptive Language

M. Riaz Moghal, M. S. Ahmad, N. Hussain, M. S. Mirza, M. W. Mirza
University College of Engineering and Technology, Mirpur AJK, Pakistan-10250

Abstract: - We have developed an approach which allows to designing hardware components using Verilog HDL, a hardware description language. We have developed Verilog-based computational model and discussed how to realize the model in digital circuits. This work also provides computer architecture instructors the ability to teach design of architectural concepts as well the flexibility and freedom to modify the approach to integrate with their current instructional needs. Over a dozen of website have available the materials and hand out to get benefit from.

Key-Words: - Hardware Descriptive Language, Computer Architecture, VLSI design, Verilog HDL

1 Introduction

Undergraduate students in computer architecture courses need to *design* computer components in order to gain an in-depth understanding of architectural concepts. To get maximum benefit, students must be active learners, engage the material and design, i. e., produce components to meet a specific need. Miserably, computers have become so sophisticated that designing architectural components, e. g., a cache memory, in hardware is not practicable in a one semester course. This work presents an approach where students use a hardware description language (HDL), Verilog HDL and an associated simulator, to design components of computer systems and investigate architectural concepts [3-6]. To support this approach, we recommend that, student should access web-based materials including manual on Verilog HDL, literature on how to realize his Verilog-based computational model in digital circuits and perform laboratory exercises [7-9].

Engineering mentors have used hardware description languages in their courses before, but at a lower level, e. g., digital circuit design or VLSI design [5]. What is distinctive about our approach is the use of an industrial standard HDL in a computer architecture course.

Students, who consider themselves software-types, are able to design and test hardware, for example, a CPU with instruction look ahead or a floating point adder. Further, they gain valuable insight into the power of computer-aided-design tools used by hardware designers in industry.

We here stress that our objective is to formulate a computational model in the Verilog notation so we no longer need to think in digital circuits. That is, we develop a higher level of abstraction to think about digital systems that is much shorter than digital circuits, e. g., sequential machines. A few lines of Verilog code may translate into hundreds of Flip Flops, **AND**, **OR** and **NOT** gates. This Verilog model is precise and short -- the Verilog notation supplies the information for *both* the data unit and the control unit associated with the control sequence. This technique is basically the one used in industry to design digital integrated circuits, such as microprocessor chips. The students are informed that with automated tools, the Verilog code could be translated to the integrated circuit masks, say, for CMOS.

The remaining organization of the paper is as follows: Section 2 presents Institutional Context. Section 3 presents Hardware Descriptive Languages. Section 4 discusses Verilog HDL. Section 5 presents Use of Verilog. Section 6 presents Computational Model. Section 7 presents Understanding of Verilog Code in Digital Logic. Section 8 discusses "Why we use Verilog for Modeling Hardware. Section 9 presents Laboratory Work. Section 10 concludes the paper. Section 11 presents Appendix for code listing.

2 Institutional Context

We have course tested this technique at university for few semesters. The technique is used in an undergraduate computer architecture course taken primarily by both electrical engineering and computer science seniors. The prerequisite for the course is a traditional computer organization course. The architecture course is organized as two one-hour lectures and a two-hour structured laboratory session per week. During the laboratory, the students access the Verilog HDL simulator on their workstations.

The course uses the popular text Computer System Architecture, by M. Morris Mano and reference book *Computer Architecture: A Quantitative Approach*, second edition, by John L. Hennessy and David A. Patterson [1-2]. Whereas the books focus on *analyzing* a design, our approach complements the books by having the student *learn* and *do* designs as well. This design aspect is especially important to educational programs in the world seeking accreditation by the Accreditation Board of Engineering and Technology (ABET).

3 Hardware Descriptive Languages

The complex digital systems at their most detailed level may consist of millions of elements, e. g., transistors or logic gates. For many decades, logic schematics served as the *language* of logic design, but not any more. At present, hardware complexity has grown to such a degree that a schematic with logic gates is almost inadequate, as it shows only a web of connectivity and not the functionality of design. Since the 1970s, computer engineers and electrical engineers have moved toward hardware description languages (HDLs). The most famous HDLs in industry are Verilog and VHDL. Verilog is the top HDL used by thousands of designers at such hardware vendors as Sun Microsystems, Apple Computer and Motorola. Industrial designers prefer Verilog. The syntax of Verilog is based on the C language, while the syntax of VHDL is based on Ada. Since the students know C or C++, Verilog was the obvious choice. For Windows 95/NT, Windows 3.1, Macintosh, SunOS and Linux platforms, FREE versions of the VeriWell product, is available. The free versions are the same as the industrial versions except, they are restricted to a maximum of 1000 lines of HDL code.

4 Verilog Hardware Descriptive Language

The Verilog language provides the digital designer with an approach of describing a digital system at a wide range of levels of abstraction, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels.

Verilog permits hardware designers to express their design with behavioral constructs, putting off the details of implementation to a later stage of design in the design. An abstract demonstration helps the designer investigate architectural alternatives through simulations and detect design bottlenecks before detailed design begins.

Although the behavioral level of Verilog is a high level description of a digital system, it is still a precise notation. Computer aided design tools exist which will "compile" the Verilog notation to the level of circuits consisting of logic gates and Flip Flops. Verilog also permits the designer to specify designs at the logical gate level using gate constructs and at the transistor level using switch constructs. A primary use of HDLs in industry is the simulation of designs before the designer must commit to fabrication.

5 The Use of Verilog Hardware Descriptive Language

Our objective in the computer architecture course is not to create VLSI chips but to use Verilog to precisely describe the *functionality* of *any* digital system, for example, a computer.

In the course, a tiny subset of the Verilog language is used to describe and develop computer architectural concepts using a register transfer level model of computation. The course also uses the structural and gate levels of Verilog to design such things as registers from D-flip flops and adders from gates. To illustrate, we explain a simple computer in the Verilog-based model.

Assume we have a very simple computer, with 1024 words of memory (**MEM**) each 32-bits wide, a memory address register (**AR**), a memory data register (**DR**), an accumulator (**AC**), an instruction register (**IR**) and a program counter (**PC**). Since we have 1024 words of memory, the **PC** and **AR** need to be 10-bits wide. We can declare the registers and memory in Verilog as the following:

```
reg [0:31] AC, DR, IR;
reg [0:9] AR, PC;
reg [0:31] MEM [0:1023];
```

We suppose a digital system can be considered as moving vectors of bits between registers. This level of abstraction is called the *register transfer level(RTL)*. For example, we may illustrate in Verilog an *instruction fetch* by four register transfers:

```
// instruction fetch
#1 AR <= PC;
#1 DR <= MEM[AR]; // memory read
#1 IR <= DR;
#1 PC <= PC + 1;
```

The first line means to transfer the 10 bits of the **PC** into the **AR** register *after* waiting one clock period (the **#1**). Note that it is important that we use Verilog's *blocking* assignment operator (<=) rather than the *non-blocking* assignment (=). In our computational model, we suppose that trailing edge triggered D-flip flops are used for the registers. Therefore, our Verilog notation models the situation because the blocking assignment operator (<=) means to block the assignment until the *end* of the current unit in simulation time.

Suppose the operation code for a LOAD instruction is 0000 in binary and is in the first four bits of **IR**, we could design the *decode* and *execute* part of a LOAD instruction as follows:

```
// decode and execute code for a LOAD
#1 if (IR[0:3] == 4'b0000) begin
#1 AR<=IR[22:31];//last 10 bits
are address
#1 DR <= MEM[AR]; // memory read
#1 AC <= DR;
End
```

The students use Verilog to illustrate their digital systems but also to test their designs by a *simulator* running on workstations. Please, see the Appendix for the complete Verilog program list of this simple computer. The above mentioned design is very slow! A LOAD instruction would take eight clock periods. The students at later stage of the course learn to carefully analyze the Verilog code as well as introduce concurrency to improve the speed. By this means, the students learn the significance of fine tuning the hardware for maximum performance.

6 Computational Model

Register transfers are general. In reality, we can consider a computation as a specific class of register transfer.

Definition: A computation consists of placing some *Boolean* function of the contents of argument registers into a destination register.

In the course we consider, a computation is a register transfer. In computer design, register transfers are very important. Register transfers are everywhere -- in arithmetic logic units (ALU), control units (CU), memory subsystems, I/O devices and interconnection networks.

Students learn that we need only Boolean functions. We don't need arithmetic or higher order functions. To understand the Boolean functions, we design combinational logic circuits, for example, an adder, from AND, OR and NOT gates.

Though, Boolean functions have no concept of time. To include the passage of time in our model, we define computing as a sequence of several register transfers where each transfer takes one or more clock periods.

Definition: Computing is a sequence of computations.

Thus, the above Verilog code for the LOAD instruction has seven computations or register transfers. We would say that performing a LOAD instruction is computing because we do seven computations one after the other, in sequential order.

A major part of the description of a computer is a plan defining each register transfer, or computation, and specifying the order and timing in which these register transfers will take place.

In the course, students learn that the Verilog-based model describes both the *data unit* which contains the digital circuits for each register transfer and the *control unit* which sequences these register transfers at the proper times.

One of the smart parts of the Verilog language design was making register transfers look like assignment statements in other programming languages. Since many designers are pleased with a language like C or Pascal, Verilog has had a large degree of success.

7 Understanding Verilog HDL Code in Digital Logic

In the present course work we demonstrate that our subset of Verilog code can be realized in digital logic circuits. Verilog is a structured language like C++ with **sequence**, **if-then-else**, **case**, **while, repeat** and **for** constructs. We also show how each control flow construct can be easily "compiled" to a digital circuit as part of the control unit (a finite state machine). Also, we show that this translation from Verilog code to digital circuit can be automated.

8 Why We Use Verilog For Modeling Hardware

Verilog has characteristics used to model digital circuits that are not available in traditional procedural languages like C or C++. One characteristic is the **continuous assignment** statement which is active for the lifetime of the program. Whenever the arguments of the expression on the right-hand side change, the outputs change, possibly after a specified time delay. This statement is used to model combinational circuits such as an adder or a subtractor.

Another characteristic not found in C or C++ is the modeling of concurrency. For example, several register transfers can be performed in the same clock period, or concurrently. Given the following register transfers without any data dependencies:

```
#1 A <= B;
#1 C <= D;
```

We can replace these two lines with the one below and have the transfers done in the same clock period.

```
#1 A <= B; C <= D;
```

The semantics of the Verilog *blocking assignment* says to evaluate all the right-hand sides and block the assignments to left-hand sides until the *end* of the current unit of simulation time. This models our assumption that the registers are composed of trailing edge D-flip flops where the information is clocked into the flip flops at the end of the clock period.

Verilog has other language characteristics for handling concurrency. For example, a digital system with its own control unit is modeled by the **initial** (and **always**) construct. Several **initial** constructs are executed concurrently. Within an **initial** construct, a structured **fork** and **join** allows multiple threads of control within a control unit.

Also, another characteristic not found in C or C++, Verilog permits the execution of a procedural statement when triggered by a value change on a wire or a register or the occurrence of a named event. For example, this is useful to model interrupts as follows:

```
@(posedge I) Intr = line&mask;
// controlled by positive edge of I
```

9 Laboratory Experiments

The students finish a series of laboratory exercises that build on a simple four instruction computer by adding addressing modes, integer multiply, instruction lookahead, cache and floating point add. Along the way, the students explore Verilog's structural modeling to construct a carry ripple adder from **AND**, **OR** and **EXCLUSIVE OR** gates, and explore concurrency with **fork** and **join** constructs and multiple digital systems and signaling.

The students find this technique using the Verilog notation easy to relate to the Hennessy and Patterson text and their previous course work [2]. We find using a major industrial HDL is highly motivating to the students. The hardware-types see learning Verilog as a significant mark on their resumes for job opportunities. The software-types are also motivated, as they see Verilog as another programming language to learn.

10 Conclusions

We have developed a technique which permits students to design hardware components using Verilog HDL, a hardware description language. We have developed a Verilog-based computational model and discussed how to realize the model in digital circuits. Free Verilog simulator and the web-based materials supply to computer architecture instructors the ability to teach design of architectural concepts as well the flexibility and freedom to modify the approach to integrate with their current instructional needs. Dozen of websites have available the materials and hand out to get benefit from.

References:

[1] M. Morris Mano, *Computer System Architecture,* McGraw Hill, USA, 1999.

[2] John L. Hennessy and David A. Patterson,*Computer Architecture: A Quantitative Approach*, second edition, 1998.

[3] Cadence Design Systems, Inc., *Verilog-XL Reference Manual*

[4] Open Verilog International (OVI), *Verilog HDL Language Reference Manual (LRM)*, 15466 Los Gatos Boulevard, Suite 109-071, Los Gatos, CA 95032; Tel: (408)353-8899, Fax: (408) 353-8869, Email: OVI@netcom.com

[5] Sternheim, E. , R. Singh, Y. Trivedi, R. Madhaven and W. Stapleton, *Digital Design and Synthesis with Verilog HDL*, published by Automata Publishing Co., Cupertino, CA, 1993, ISBN 0-9627488-2-X

[6] Thomas, Donald E., and Philip R. Moorby, *The Verilog Hardware Description Language*, second edition, published by Kluwer Academic Publishers, Norwell MA, 1994, ISBN 0-7923-9523-9, includes DOS version of VeriWell simulator and programs on diskette.

[7] Bhasker, J., A Verilog HDL Primer, Star Galaxy Press, 1058 Treeline Drive, Allentown, PA 18103, 1997, ISBN 0-9656277-4-8.

[8] Wellspring Solutions, Inc., *VeriWell User's Guide 1.2*, August, 1994, part of free distribution of VeriWell, available online.

[9] World Wide Web Pages: FAQ for comp.lang.verilog

11 Appendix

```
//Simple Computer in Verilog HDL
module simple;
//simple computer with 4-bit op odes in
//first four bits and 10 bit address in
//last ten bits of 32-bit instructions
//0000load M;Load contents at address M
//into AC
//0001 store M; Store contents of  AC
//into address M
//0010 add M;Add contents at address Mto AC
//0011 jump M; Jump to instruction at
//address M
parameter clock = 1;
//declare registers and flip flops
reg [0:31] AC, IR, DR;
reg [0:9] PC, AR;
reg [0:31] MEM[0:1023];//1024 words of 32-
//bit memory
//The two "initial" and the "always"
//constructs run concurrently Will stop
//the execution after 80 simulation units.
initial begin: stop_at
 #(100*clock) $stop;
end
//Initialize the PC register and
//memory MEM with test program
initial begin: init
PC=10;//start of machine lang program
MEM[3]=
32'b00000000000000000000000000000010; // Data 2
MEM[4]=
32'b00000000000000000000000000000001; // Data 1
MEM[10]=
32'b00000000000000000000000000000011; // Load 3
MEM[11]=
32'b00100000000000000000000000000100; // Add 4
MEM[12]=
32'b00010000000000000000000000000101; // Store 5
MEM[13]=
32'b00110000000000000000000000001011; // Jump 11
$display("Time PC IR AR DR AC MEM[5]");
//monitor   following   registers   and   //memory
location and print when any //change
$monitor("  %0d  %h  %h  %h  %h  %h  %h",
$time, PC, IR, AR, DR, AC, MEM[5]);
end
//main_process    will    loop    until
//simulation is over
always begin: main_process
// Instruction Fetch
#clock AR <= PC;
#clock DR <= MEM[AR]; // memory read
#clock  IR  <=  DR;  AR  <=  DR[22:31];
//last ten bits are address
#clock PC <= PC + 1;
//decode and execute instruction
if(IR[0:3] == 4'b0000) begin // load
#clock DR <= MEM[AR];
#clock AC <= DR;
end
if(IR[0:3] == 4'b0001) begin // store
#clock DR <= AC;
#clock MEM[AR] <= DR;
end
if(IR[0:3]  ==  4'b0010)  begin  //
add
#clock DR <= MEM[AR];
#clock AC <= AC + DR;
end
if(IR[0:3] == 4'b0011)
begin // jump
#clock PC <= AR;
end
end
endmodule
```

Output

```
Time PC    IR       AR     DR          AC        MEM[5]
0 00a xxxxxxxx xxx xxxxxxxx xxxxxxxx xxxxxxxx
1 00a xxxxxxxx 00a xxxxxxxx xxxxxxxx xxxxxxxx
2 00a xxxxxxxx 00a 00000003 xxxxxxxx xxxxxxxx
3 00a 00000003 003 00000003 xxxxxxxx xxxxxxxx
4 00b 00000003 003 00000003 xxxxxxxx xxxxxxxx
5 00b 00000003 003 00000002 xxxxxxxx xxxxxxxx
6 00b 00000003 003 00000002 00000002 xxxxxxxx
7 00b 00000003 00b 00000002 00000002 xxxxxxxx
8 00b 00000003 00b 20000004 00000002 xxxxxxxx
9 00b 20000004 004 20000004 00000002 xxxxxxxx
10 00c 20000004 004 20000004 00000002 xxxxxxxx
11 00c 20000004 004 00000001 00000002 xxxxxxxx
12 00c 20000004 004 00000001 00000003 xxxxxxxx
13 00c 20000004 00c 00000001 00000003 xxxxxxxx
14 00c 20000004 00c 10000005 00000003 xxxxxxxx
15 00c 10000005 005 10000005 00000003 xxxxxxxx
16 00d 10000005 005 10000005 00000003 xxxxxxxx
17 00d 10000005 005 00000003 00000003 xxxxxxxx
18 00d 10000005 005 00000003 00000003 00000003
19 00d 10000005 00d 00000003 00000003 00000003
20 00d 10000005 00d 3000000b 00000003 00000003
21 00d 3000000b 00b 3000000b 00000003 00000003
22 00e 3000000b 00b 3000000b 00000003 00000003
23 00b 3000000b 00b 3000000b 00000003 00000003
25 00b 3000000b 00b 20000004 00000003 00000003
26 00b 20000004 004 20000004 00000003 00000003
27 00c 20000004 004 20000004 00000003 00000003
28 00c 20000004 004 00000001 00000003 00000003
29 00c 20000004 004 00000001 00000004 00000003
30 00c 20000004 00c 00000001 00000004 00000003
31 00c 20000004 00c 10000005 00000004 00000003
32 00c 10000005 005 10000005 00000004 00000003
33 00d 10000005 005 10000005 00000004 00000003
34 00d 10000005 005 00000004 00000004 00000003
35 00d 10000005 005 00000004 00000004 00000004
36 00d 10000005 00d 00000004 00000004 00000004
37 00d 10000005 00d 3000000b 00000004 00000004
38 00d 3000000b 00b 3000000b 00000004 00000004
39 00e 3000000b 00b 3000000b 00000004 00000004
40 00b 3000000b 00b 3000000b 00000004 00000004
42 00b 3000000b 00b 20000004 00000004 00000004
43 00b 20000004 004 20000004 00000004 00000004
44 00c 20000004 004 20000004 00000004 00000004
45 00c 20000004 004 00000001 00000004 00000004
46 00c 20000004 004 00000001 00000005 00000004
47 00c 20000004 00c 00000001 00000005 00000004
48 00c 20000004 00c 10000005 00000005 00000004
49 00c 10000005 005 10000005 00000005 00000004
50 00d 10000005 005 10000005 00000005 00000004
51 00d 10000005 005 00000005 00000005 00000004
52 00d 10000005 005 00000005 00000005 00000005
53 00d 10000005 00d 00000005 00000005 00000005
54 00d 10000005 00d 3000000b 00000005 00000005
55 00d 3000000b 00b 3000000b 00000005 00000005
56 00e 3000000b 00b 3000000b 00000005 00000005
57 00b 3000000b 00b 3000000b 00000005 00000005
59 00b 3000000b 00b 20000004 00000005 00000005
60 00b 20000004 004 20000004 00000005 00000005
61 00c 20000004 004 20000004 00000005 00000005
62 00c 20000004 004 00000001 00000005 00000005
63 00c 20000004 004 00000001 00000006 00000005
64 00c 20000004 00c 00000001 00000006 00000005
65 00c 20000004 00c 10000005 00000006 00000005
66 00c 10000005 005 10000005 00000006 00000005
67 00d 10000005 005 10000005 00000006 00000005
68 00d 10000005 005 00000006 00000006 00000005
69 00d 10000005 005 00000006 00000006 00000006
70 00d 10000005 00d 00000006 00000006 00000006
71 00d 10000005 00d 3000000b 00000006 00000006
72 00d 3000000b 00b 3000000b 00000006 00000006
73 00e 3000000b 00b 3000000b 00000006 00000006
74 00b 3000000b 00b 3000000b 00000006 00000006
76 00b 3000000b 00b 20000004 00000006 00000006
77 00b 20000004 004 20000004 00000006 00000006
78 00c 20000004 004 20000004 00000006 00000006
79 00c 20000004 004 00000001 00000006 00000006
Stop at simulation time 80
```