2006-556: VERIFICATION OF HARDWARE DESCRIPTION LANGUAGE DESIGNS

Joanne DeGroat, Ohio State University

Dr. Joanne DeGroat is an Associate Professor at The Ohio State University in the Department of Electrical and Computer Engineering. She received her BS degree in Engineering Science from Penn State University, her MSEE from Syracuse University, and her Ph.D. in Electrical and Computer Engineering from the University of Illinois. Her research interests are in computer architecture, VLSI, mixed signal VLSI, hardware description languages (HDLs), and verfication of HDL designs. Recently she has been conducting research in the areas of HDL verification, FPGA architectures, and RF VLSI design.

Verification of Hardware Description Language Designs

Joanne E. DeGroat
Department of Electrical and Computer Engineering
The Ohio State University

Abstract:

The field of HDL verification is relatively new and addresses the need to verify that the HDL model does indeed implement what has been specified. This is especially important as almost all digital integrated circuits are synthesized from HDL descriptions. This paper outlines the content of the course Verification of Hardware Description Language (HDL) Models at Ohio State University. This course currently consists of lectures and four verification projects. This paper discusses the course and the four projects.

I. Overview

The design of modern digital integrated circuits has changed dramatically in the last 15 years. Technology has advanced to the point to where we are able to reliably produce chips with millions of logic gates on a single integrated circuit die. This translates into very significant logic function for a single chip. The only way that design of chips capable of effectively using this much functionality is possible is with advanced tools and design methodology. Part of the methodology is a rigorous partitioning and structuring of the design. One has only to look at a photomicrograph (photo of the circuitry on an IC) of a chip from the early or mid 1970s to the photomicrograph of a modern processor to see this. The early chips looked like a bowl of spaghetti. Modern chip are well organized and are a structure of interconnected blocks.

This is well supported by the current design methodology that involves the use of Hardware Description Languages (HDLs) for the design of the IC. Modern HDLs are hierarchical and support the partitioning of the design into logical functioning blocks. The use of HDLs in design has allowed more significant verification of the design before it is first fabricated. In the early 1990s the focus was on getting a chip that fabricated and functioned. With the advent of HDL synthesis, getting functioning first run chips was no longer an issue. The emphasis quickly evolved into to not only producing a chip that functioned, but that functioned as desired in the environment for which it was being designed. This has given rise to the field of HDL Verification. And this field has grown rapidly. It has reached the point where numerous corporations not only have design engineers but also now have an equal number of verification engineers.

This paper will highlight the course content of "Verification of Hardware Description Language (HDL) Models" at The Ohio State University. The course was first offered in SPRING 2001 and was one of the first HDL Verification courses taught anyplace. It is the second course in the HDL sequence. The first course covers modeling with a Hardware Description Language at various levels of abstraction. It starts with modeling at the data flow level with a one-to-one correspondence of HDL statement to the physical hardware being modeled. The level of abstraction that designs are modeled at, increases until at the end of the course the modeling is at the algorithmic level, i.e., the behavioral level. In summary, in the first course of the sequence the students write a model of the design and the testbench, with the tests that verifies their model, is provided to them.

This second course, “Verification of Hardware Description Language (HDL) Models” continues from where the first course leaves off. In this course the students are provided the design specification and a HDL data flow model of an implementation of a design intended to meet that specification. They must write the testbench and construct a test suite to verify that the design meets the specification. The final assignment of the first HDL course is the modeling of an IEEE Floating Point Multiplier at the behavioral level. The first assignment of the verification course is the verification of an IEEE Floating Point Addition/Subtraction unit. The course then continues with the verification of a design that transforms through 3 stages of development.

II. Course Structure

The course consists of lecture and lab components. It meets three times per week. Two meetings are lectures and one is lab where the students get to work on the projects. There are currently four projects over the ten weeks of the quarter. These consist of the Floating Point Adder project, a multifunction calculator, a modification to the multifunction calculator, and then turning the calculator design into a datapath capable of out-of-order execution.

The course concentrates on functional verification. The lectures also cover topics such as formal verification, random test generation, and assertion based verification but focuses primarily on the topic of functional generation. For functional verification the testbench both applies the tests and verifies the results produced by the design under test. The tests are usually contained in a vector file rather than coded into the HDL of testbench as this makes test generation significantly easier. Placing the test vectors in a file also allows easy expansion of tests and incorporation of new tests once the first test results are evaluated. Lectures also cover several other verification topics such a white box versus black box verification. The first design the students work on is white-box verification. White box verification means that the students see the HDL code they are verifying. In some cases the HDL code is not available to be seen.

II.A. The Projects

II.A.1 Floating Point Adder Functional Unit

For the first project the students work in groups of two. As modern HDL design is done in teams the verification course uses teamwork for all the assignments. However, the teams are changed for each assignment. The second assignment is also done in groups of two but the students must pair with a different partner. The third assignment is done in groups of three as is the final project.

As has been stated previously, the first project is the verification of an IEEE Floating Point Add/Subtract functional unit that handles all aspects of the floating point standard, including denormalized numbers. They are provided with two models of the design for this assignment. One is a behavioral model the code of which is included in figures 1 and 2. The second is the model that requires verification. It is a VHDL dataflow description of the floating point adder that synthesizes well. The behavioral model is algorithmic and readily understandable. However, it would not synthesize well. The dataflow model follows a design that does not quite have a one-to-one correspondence with the logic generated from synthesis but is close.

```
-----------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use WORK.fpa_support.all;
entity fpa is
   PORT (A,B : IN std_logic_vector (31 downto 0);
             latch, drive: IN std_ulogic;
             C : OUT std_logic_vector (31 downto 0));
end fpa;
----------------------------------------------------------------
```

Figure 1. Floating Point Adder Entity

```
architecture behavioral of fpa is

signal Aint,Bint,Cint : std_logic_vector (31 downto 0);
signal srexp : std_logic_vector (7 downto 0);
signal saman,sbman,srman : std_logic_vector (23 downto 0);
signal s_mul_res : std_logic_vector (47 downto 0);
signal s_res_exp_int : integer;

BEGIN

latch_in : PROCESS
BEGIN
wait until rising_edge(latch);
Aint <= A; Bint <= B;
END PROCESS latch_in;

floating_pt_add : PROCESS (Aint,Bint)
variable asign,bsign,rsign : std_logic;
variable aexp,bexp,rexp : std_logic_vector (7 downto 0);
variable aman,bman,rman : std_logic_vector (23 downto 0);
variable a_arg,b_arg : std_logic_vector (24 downto 0);
variable res_build : std_logic_vector (31 downto 0);
variable add_res : std_logic_vector (47 downto 0);
variable res_exp_int : integer;
variable aexp_int, bexp_int,shift_dist : integer;
```

Figure 2. Floating Point Adder Behavioral Code

```
BEGIN
asign := Aint(31);
bsign := Bint(31);
rsign := asign;
aexp := Aint(30 downto 23);
bexp := Bint(30 downto 23);
-- handle NaNs by putting them into a format like normalized num
-- or put into normalized format
IF ((aexp = exp_zero) and (Aint(22 downto 0) /= man_zero(22 downto 0)))
THEN aman := Aint(22 downto 0) & '0';
ELSIF (aexp /= exp_zero AND aexp /= exp_ones)
THEN aman := '1' & Aint(22 downto 0);
ELSE aman := '0' & Aint(22 downto 0);
END IF;
```

Figure 2. Floating Point Adder Behavioral Code

```
IF ((bexp = exp_zero) and (Bint(22 downto 0) /= man_zero(22 downto 0)))
THEN bman := Bint(22 downto 0) & '0';
ELSIF (bexp /= exp_zero AND bexp /= exp_ones)
THEN bman := '1' & Bint(22 downto 0);
ELSE bman := '0'&Bint(22 downto 0);
END IF;
-- handle special case of result a NaN
IF((aexp = exp_ones) and (not (aman = man_zero))) or -- A a NaN
((bexp = exp_ones) and (not (bman = man_zero))) or -- B a NaN
((aexp = exp_ones) and (aman = man_zero) and -- +inif + -inif
(bexp = exp_ones) and (bman = man_zero) and (asign /= bsign))
THEN Cint <= NAN;
-- handle special case of A input Inifinity
ELSIF((aexp = exp_ones) and (aman = man_zero))
THEN Cint <= Aint;
-- handle special case of B input Inifinity
ELSIF((bexp = exp_ones) and (bman = man_zero))
THEN Cint <= Bint;
-- handle special case of A equal +/- 0
ELSIF((aexp = exp_zero) and (aman = man_zero))
THEN Cint <= Bint;
-- handle special case of B equal +/- 0
ELSIF((bexp = exp_zero) and (bman = man_zero))
THEN Cint <= Aint;
ELSE
-- add the numbers
-- make a the large of the two numbers
IF ((bexp > aexp) OR ((bexp = aexp) AND (bman > aman)))
THEN -- b is the larger number so exchange them
rsign := bsign; bsign := asign; asign := rsign;
rexp := bexp; bexp := aexp; aexp := rexp;
rman := bman; bman := aman; aman := rman;
END IF; -- a now holds the larger of the two numbers
```

Figure 2 (continued). Floating Point Adder Behavioral Code

```
-- determine the exponent difference by converting the
-- exponents to integer
aexp_int := std8_2_int(aexp);
bexp_int := std8_2_int(bexp);
shift_dist := aexp_int - bexp_int;
-- shift the b mantissa by shift distance
IF (shift_dist>0) THEN
FOR I in 1 to shift_dist LOOP
bman := '0' & bman(23 downto 1);
END LOOP;
END IF;
-- add the mantissas if the signs are the same or subtract if diff.
a_arg := '0' & aman;
b_arg := '0' & bman;
IF (asign = bsign) THEN -- add
std_logic_add(b_arg,a_arg); -- result will be in a_arg
ELSE
b_arg := not b_arg; -- ones complement of B
std_logic_add(const_one,b_arg);
std_logic_add(b_arg,a_arg); -- result in a_arg
END IF;
-- now normalize the result of the add
normalize (aexp_int,a_arg,rexp,rman);
IF (rman = man_zero) THEN -- zero result
Cint <= rsign & exp_zero & man_zero(22 downto 0);
ELSIF (rexp = exp_zero) THEN
-- denorm result
Cint <= rsign & rexp & rman(23 downto 1);
ELSE
CINT <= rsign & rexp & rman(22 downto 0);
END IF;

END IF;

END PROCESS floating_pt_add;

drive_out : PROCESS (drive)
BEGIN
IF drive = '0'
THEN C <= Cint;
ELSE C <= HighZ;
END IF;
END PROCESS drive_out;

END behavioral;
```

Figure 2 (continued). Floating Point Adder Behavioral Code

The students start this project by writing a test plan which outlines the tests to be applied, the expected result of the test, and outlines the methodology of how the tests are to be applied and checked. Students then must write the testbench which both applies the tests and checks the results. They must also generate the tests to be applied. They are provided with the tests applied to the Floating Point Multiplier that they algorithmically modeled in ECE 762, the HDL modeling course. For that assignment they are provided with a file of the tests that the testbench applies. Class lecture covers the floating point multiplier test vector set and the coverage provided by the test set. They use the testset of that design as a basis for designing tests for the floating point adder. They are pointed to sites on the web which deal with testing floating point units and several groups have located and used programs for IEEE Standard floating point unit test generation downloaded from the web.

In generating the tests to apply students can hard code the test into the testbench, which is not recommended, enter them into a file with an editor, or have them generated by a program. The ability to generate the test vectors to apply to the model from a C/C++ program allows the generation of a significant number of test cases. This results in better test coverage and more confidence in the results of the verification effort. It also introduces the use of randomly generated tests and directed random test.

The coverage being referred to here is functional coverage. An RTL model of the design, the best coding style for synthesis of the final circuit, will have near 100% code coverage with only a few tests. Functional coverage is coverage of the regions of the design that the test should address. The use of the floating-point-adder is excellent for discussing coverage. When designing the tests for the adder you would start by taking the various classes on each of the inputs as shown below in figure 3.

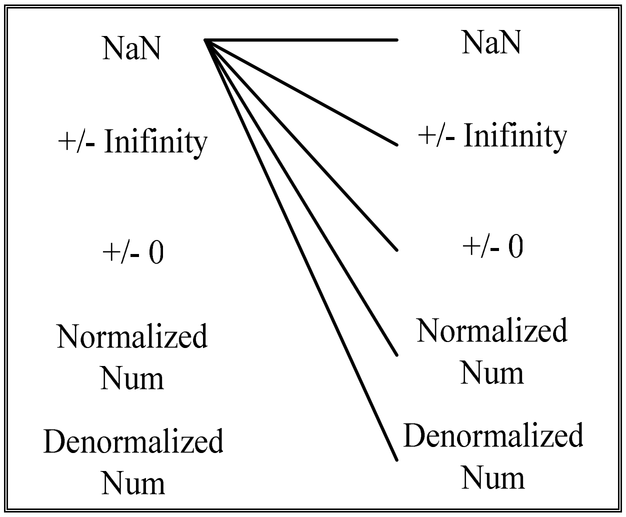

Figure 3. Test Matrix

The tests start with taking a NaN input on the A input to the adder and having it operate with an input on the B input from each of the other classes. As illustrated that means having the NaN on A added with an NaN, +Inifinity, -Inifinity, +0, -0, normalized numbers, and denormalized numbers. In the cases of the B input being all but a normalized number or a denormalized number only one test is needed. In the case of normalized and denormalized number several cases would be needed and it is here that some random "directed" tests would be useful. The cases of Inifinity and 0 are similar.

The case of a normalized number on each input is the situation where directed random test is very useful. One can randomly generate numerous normalized number values for both the A and B input. The same holds true for denormalized numbers on both inputs. And then to complete the functional coverage, the cases where a normalized added/subtracted with denormalized results in a normalized, where a a normalized added/subtracted with denormalized number results in a denormalized numbers, etc. are also needed. Also required are the cases where two denormalized numbers are added and result in a normalized number, the addition of two normalized numbers overflows to Infinity, etc. Covering the central cases randomly and these corners of the design explicitly, provides one with a high level of confidence that the design is correct.

In modern designs of which this might only be a part it is impossible to run all possible test vectors through the design. In the case of the floating point adder exhaustive tests of the design would require 264 test vectors. This is simply not possible. So the verification engineer runs enough vectors that the level of confidence in the design is high. These details are covered in the test plan which the students write.

Additionally, the students analyze the functional coverage of the tests on the code of both the behavioral model and the dataflow model to insure all portions of the code are exercised. Lectures also cover the value and merit of functional coverage and code coverage.

The product due from this assignment is a report which contains the test plan, the code of the testbench, and a summary of the results of running the tests. The model is purposefully injected with errors so that there are errors to be found. The report needs to identify the errors found and the recommended fix.

II.A.2 The Calculator

A project entitled "The Calculator" is the next assignment. This assignment is based on a model provided by IBM. The block structure of the design is shown in figure 4. In this design there are four input instruction/data streams and 4 output data streams (instruction results). Each input is constrained such that the output will appear on the corresponding output, i.e. OUT1 will be the output of the data and operations that came in on IN1. The calculator is capable of four operations on the data input to it. It can adder or subtract using the adder or perform both left and right shifts of the data using the shifter. The shifter is capable only of shifts of 2 places.

The priority scheme implemented by the priority logic is that all four requestors (input streams) have equal priority and when inputs arrive simultaneously they are serviced in ascending order of request number. After the operation is complete the response is temporarily held in the ALU output and then routed to the correct output by the MUX. Add and shift operations complete in one cycle. Overall timing is such that the next operation does not begin until the output from the current operation appears on the ouput.

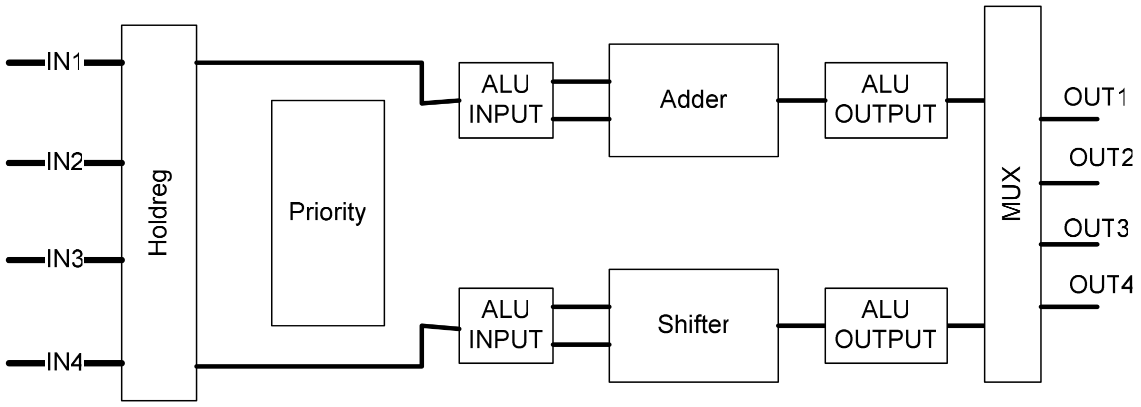

Figure 4. The Calculator Design

Students are given the synthesizable dataflow VHDL code for this architecture. They again work in groups of two. As with the first design they must complete a verification plan and then implement it. And as with the first design, error are injected into the design and the grading is based on the errors found, the quality of the test plan, and the quality of the implementation of the test plan.

II.A.3 Calculator, version II

The next assignment is an extension to the calculator I design. The initial design allowed only one command from each of the four ports at a time. All ports needed to wait until the calculator completed execution of the current commands before another command could be sent on any port. In this version of the calculator, up to four commands can be sent to the calculator on each of the four ports. Hence, the calculator could have up to 16 commands queued up at a single time.

This single design change has major implications to the system and significantly increases the complexity of the verification task. Since there are two internal arithmetic functions units, one for addition/subtraction and one for shifts, and with priority imposed, it is quite possible for commands to be executed out of order. For example, if the four ports all send in 3 add commands followed by a shift command, the shift commands will likely complete prior to the latter add commands. However, commands from the same port that use the same execution unit will complete in order especially since the priority scheme is such that all four initial add instructions will complete before the second add instruction for any stream can be executed.

In order to relate the response to the correct command a tag is added to the input and output protocols. The tag is a unique identifier for each of the commands on each port stream. This tag maintains the order of the instructions. Another tag is used to identify the port that the instruction/data belongs to. Additionally, when instructions are dispatched to the hold register the operands are valid. There are no data dependencies in dispatched instructions. Data dependencies would be handled by the instruction dispatch unit which is not part of design two.

As with the other designs students must complete a verification plan and then implement it. One difference is that students now work in groups of three. Grading of this assign is similar to the previous assignments. Additionally, student groups are required to present their results to the class in a short presentation.

II.A.4 Calculator Design 3 – Transformation into an ALU

The third and final assignment sees this design transformed into a datapath capable of two simultaneous operations each cycle. The significant hardware addition is a set of registers that can be written to, receive the results from either execution unit, send values to either execution unit for an instruction, or send the contents of the register to the output through a path alongside one of the execution units. The additions can be seen in Figure 5 which is a high level diagram of the Calculator Design 3.

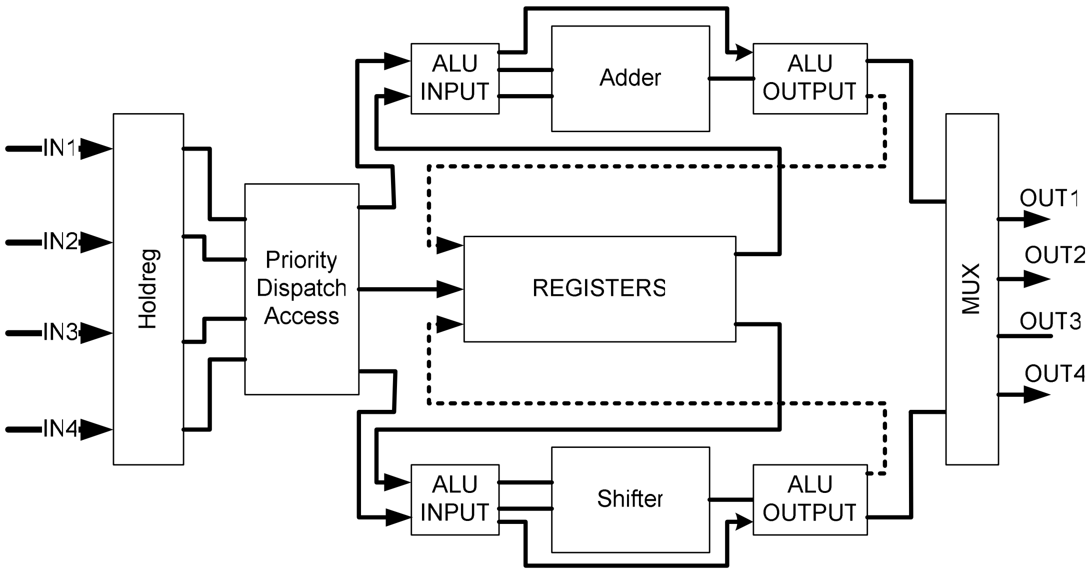

Figure 5. Calculator 3 Block Diagram

This change to the architecture transforms the calculator into a complex datapath. There are now instructions to load data into the registers and output the contents of a register. Operations are now of the form where one operand is in the instruction. In the case of two operand instructions the other operand comes from a register with the results of the operation written back to the source register.

The task of verifying the architecture is now orders of magnitude more complex than the previous architecture. Not only is data tagged as to the stream it belongs to, but instructions are also tagged to relate instructions to the correct steam and order them. An added complication arises when an instruction uses the computed result of a previous instruction. The design must insure that the new instruction does not use the data until it is written back. This, and the possibility of out of order execution, presents significant challenges to the student. At the conclusion of this assignment, each student group submits a report and gives an oral presentation.

It is easy to see that it is impossible to even come close to exhaustively testing the design. This is the major challenger verification engineers are faced with. Exhaustive test is not possible. This means that a verification engineer must design the tests to be applied in a logical fashion and increases the probability that the design meets specification to an acceptable level.

III. Conclusions

This paper outlines the contents of the Verification Course at Ohio State University. At the 2005 Design and Verification Conference (DVCon) it was noted in the keynote address that within the next 5 years industry will be looking for a significant number of verification engineers. The ratio of verification engineers to design engineers in many corporations is currently about one verification engineer to one design engineer. At DVCon 2006 one company was reporting that they had a ratio of five verification engineers to each design engineer. It is quite likely that this ratio will shift toward verification engineers in the near future for all companies.

The verification course was first offered in the Spring of 2001. The first course teaches modeling and verification is not included as that course could benefit from more time as there is easily more that could be added to it. The verification course is packed and has no room to add more content. A third course that takes a design from specification, through modeling, through verification and to an FPGA implementation is envisioned.

The content of what will be in the course in the Spring 2006 offering has, needless to say, changed somewhat. The use of a reference model was introduced in the 2004 offering. For the coming offering a modification for the inclusion of assertion-based-verification is in progress. The course will continue to change as the field changes. When the course started design verification languages such as VERA were new. Today there are numerous verification languages such as VERA and modifications to design languages to support verification such as is done with System Verilog. These were discussed and will be included in the course over time as the field becomes stable.

Bibliography

1. Peet James, “Verification Plans,” Kluwer Academic Publishers, 2004.

2. “System Verilog Golden Reference Guide,” Doulos Ltd.,May 2004

3. “System C Golden Reference Guide,” Doulos Ltd.,January 2005

4. Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, “System Verilog Assertions Handbook, VhdlCohen Publishing, 2005.

5. Janick Bergeron, “Writing Testbenches:Functional Verification of HDL Models, 2nd Edition,” Academic Publishers, 2004.