Laboratory Options for the Computer Science Major

Christopher Vickery
Tamara Blain
Queens College of CUNY

Computer Science and Computer Engineering programs typically converge on the Dynamic-Static Interface (DSI) from opposite directions. Computer Science (CS) introduces students to system architecture and organization so they can have a better appreciation for the mechanisms that make their software work, whereas Computer Engineering (CE) introduces students to software design so they can have a better appreciation for the software that will be using the hardware systems they design. Mindful of this distinction between CS and CE, we chronicle the efforts of our CS department to capitalize on current trends in the design and implementation of digital systems to extend our students' expertise in this area. We summarize the current curriculum in our department, present a survey of the language options we have explored for evolving our curriculum, and conclude with a brief description of the laboratory environment we have adopted, which is centered on the Handel-C hardware implementation language.

1 Introduction

Computer Science and Computer Engineering curricula have traditionally brought significantly different perspectives to bear on what to cover and how to teach computer architecture, the point where the two disciplines meet. Broadly speaking, the CS students bring good software skills to their architecture courses, whereas the CE students bring stronger circuit design skills to theirs. The distinction carries over to the design of digital systems in industry, where a software team and a separate engineering team typically work in parallel during the development of a new product (codesign), with software/hardware integration occurring late in the development cycle.

But the inexorable advance of circuit complexity has caused the traditional engineer/programmer dichotomy to start to break down. We are not talking here about shifting the dynamic-static interface [23, 25] for a particular system design, nor about the dual roles individuals might play in a design effort. Rather, we are responding to changes in the way digital systems are developed due to changes in the functionality of programmable logic devices (FPGAs in particular) and the software tools used to develop systems using them.

Ours is a Computer Science department in a liberal arts college. There is no engineering department on campus. Although the university encourages cooperation among its member colleges, the fact remains that the closest CE courses available to our students are a 90-minute subway ride away from us. In this context of CS isolated from CE, this paper reports on the options we have considered as we adjust our curriculum to provide our undergraduate students with a better understanding of the principles and practices of implementing digital architectures.

2 Context: A CS Department

Our undergraduate curriculum prepares students in the broad areas of *i)*software design and implementation, *ii)* formal methods, *iii)* hardware design, and *iv)* applications, in roughly that order of emphasis. Our offerings in the "hardware design" area include a course in assembly language and basic logic design, and a second course that covers additional logic design and an introduction to computer architecture. We have used a number of textbooks for these courses over the years, never finding ones that both students and faculty found completely suitable. The current text for both courses is by Murdocca and Heuring [22].

Our curriculum also includes a "Hardware Laboratory" course, which has not been offered in recent years. This course was developed in the days of SSI and MSI integrated circuits and dropped by the wayside as simulators have allowed us to develop a similar degree of mastery to the old lab course without requiring the students to spend time in the lab itself. Thus, the closest our students have come to a hardware laboratory experience for the past several years has been through simulation assignments in the two courses mentioned above. The student edition of CircuitMaker [2] has served our purposes in this regard, although the free version must be installed only on the students' personal computers, not in college laboratory facilities.

Four years ago, we received NSF funding [16] to revise our curriculum to use HDLs to give our students a more realistic view of circuit design technologies. Our stated goals were, "to give all of our students some knowledge of the methods that are used in designing modern digital circuits [and] to provide those students who are interested with hands-on experience in designing and using digital logic as a method of teaching them about computer architecture."

At the time we prepared our grant proposal, VHDL seemed to be the natural vehicle for introducing CS students to logic design, leveraging their existing software skills to introduce them to hardware design techniques. There were several textbooks based on VHDL available, it was an IEEE Standard, and seemed generally well suited to our needs. Although Verilog also become an IEEE Standard in 1995, at the time of our grant proposal, VHDL seemed like the most straightforward way to go. Since then, there has been a good deal of foment in the CAD world, propelled by the need for tools to adapt to the ever-increasing complexity of digital devices. What follows is a survey of the evolving software development options we have seen.

3 Laboratory Options

A first question CS departments have to answer in planning instruction in digital design is whether to focus on simulation only, or to have the students target actual hardware devices. A second question is whether to use commercial development tools or instructional software. Once those questions are resolved, the issues of software and (if hardware devices are targeted) prototyping platforms need to be addressed.

3.1.1 Simulation or Hardware?

Simulation is a critical step in developing new hardware designs, but in an instructional environment, simulation can arguably be the end step in a student's lab experience.

There are several arguments for using only simulation for introducing CS students to the design of digital systems:

- *Cost*. Except for the computers to run the simulations, a relatively abundant commodity, simulation avoids the overhead and costs of purchasing and maintaining prototyping equipment and instrumentation for a laboratory.
- *Ease of debugging.* In addition to avoiding the issues associated with bad connections and failed components, simulation provides a software view of the system under development which is not only more familiar to CS students, but also more flexible than hardware in terms of allowing students to visualize and locate problems in their designs.
- *Simplicity.* Development tools for hardware implementations need to provide a richer feature set than instructional simulations. The result can be that students need to spend more effort learning to use the tool than studying the simulations.

On the other hand, implementing actual circuits can be much more motivating than just running "yet another program."

We have not had good luck with most of the student-oriented simulators for logic design that are available at low or no cost. Many had problems with reliable schematic entry, and many have had unnecessarily poor user interfaces. The student edition of the CircuitMaker schematic entry and simulation software from Altium cited above has been the best we have found so far [2]. Compared to a textbook-only presentation, it's far superior. But it limits the size of the designs students can implement, cannot be installed in departmental labs for free, and it doesn't provide a tie-in to actual hardware implementation.

Another option for those graduates who become interested in chip design is to send them to commercially available short courses that teach digital IC or systems design, ECAD, or hardware/software codesign. But the danger in these courses is that there is "insufficient time to address any topic in the depth required by students to gain proper insight into the subject area". Thus these courses may not adequately provide them with the skill set necessary for designing and implementing complex distributed embedded systems and Systems on Chip (SOCs) [12]. And, of course, this option begs the question of what to do in the context of a university curriculum.

The gamut of options available for implementing circuits in hardware is extremely broad, ranging from inexpensive breadboards with SSI and MSI ICs connected by jumper wires to industrial-grade prototyping systems used in the development of commercial ASIC designs at costs that are prohibitively high for virtually all instructional purposes. FPGA-based development systems strike a middle ground between these two extremes, and are particularly well suited to instructional laboratories. The boards are self-contained units requiring no assembly on the part of the student, although expansion headers are normally available for customized projects. Student designs are prepared and simulated on PCs, and downloaded to the prototyping board through a serial or parallel cable. The use of reprogrammable FPGAs as the implementation target gives students a development cycle familiar to them familiar from the software development world: edit, compile, debug.

Major FPGA vendors, notably Altera and Xilinx, provide inexpensive or free student versions of their commercial tools for FPGA development suitable for use with a variety of prototyping boards from companies such as Xess and Digilent. An inexpensive package available from Altera's University Program, for example, includes a prototyping board with a 20,000 gate

FPGA, several LEDs, displays, and switches mounted on the board, and I/O connectors for a mouse and VGA display. This kit comes packaged with a good tutorial volume [17] featuring a number of interesting projects students can do. The kit includes a student edition of Altera's MAX+Plus development software, which includes schematic, waveform, and HDL text editors for design entry. It should be noted, however, that this kit uses a relatively small FPGA by today's standards, (not large enough to implement a full CPU) and that the MAX+Plus software does not provide the same functionality as Altera's Quartus toolchain. Systems of this type are more appropriate for introductory logic design laboratories rather than CS Computer Architecture courses, where students need to explore architectural design parameters.

Hardware Description Languages (HDLs), most commonly VHDL and/or Verilog, are the most commonly used means for entering designs for platforms like those discussed so far. But today's FPGA devices can have millions of gates instead of tens of thousands, providing architecture students with hardware targets rich enough to support investigations into topics as advanced as pipelining, cache design, and multiprocessor communication, not just basic logic design. Furthermore HDL programming is evolving to deal with the complexity of these newer devices. We review some of these languages below. An appealing alternative to working with an HDL or one of its derivatives, at least for CS students who are approaching the DSI from the software side, is to use a language based on a traditional High Level Programming Language (HLL). After our survey HDLs and their derivatives, we will turn our attention to Handel-C, a hardware implementation language based on C that we are adopting for use by our CS majors.

4 HDLs and Their Derivatives

4.1.1 Verilog and VHDL

Hardware design is dominated by the use of Verilog and VHDL. They are most powerful as gate-level implementation languages [1][3]. VHDL allows a multitude of language or user-defined data types, which may mean confusing conversion functions needed to convert objects from one type to the other. All of the logical operators, NAND, NOR, XOR, etc, are included in VHDL but separate constructs, typically defined using the VITAL language, must be used to define cell primitives of ASIC and FPGA libraries. VHDL offers a great deal of flexibility in terms of its abundance of permissible coding styles. It allows for concurrent synchronization schemes, such as semaphores. VHDL is better suited than Verilog to handle very complex designs. It is relatively weaker in lower level designs but superior in higher level and system level designs, which results in slower simulations. Its wealth of constructs, attributes, and types make VHDL a good language for design and verification [7]. It is strongly typed and there are many ways to model the same circuit, features which make it more robust and powerful than Verilog but also more complex. This complexity means it is more difficult to understand and use.

Verilog has adopted many of VHDL's features, thus Verilog is moving towards increasing complexity as well [7]. Verilog is used for high-speed gate-level and register-level circuit descriptions, fast IC modeling and RTL simulation, easy synthesis, and test applications [9]. Gate simulations in Verilog are 10x to 100x faster than the same simulations in VHDL, which means shorter time to verify designs [8]. Compared to VHDL, Verilog data types are simple, easy to use and geared towards modeling hardware structure as opposed to abstract modeling. Because it is simpler, Verilog is easier to learn. On a Verilog vs. VHDL debate forum, an engineer who knows both languages cites: "If you were just taught Verilog syntax, you're in trouble. If you were taught syntax with guidelines, and warned about legal Verilog constructs that should never be used, you can gain expertise in half the time it takes to become proficient in VHDL [8]." Because of its background as an interpretive language, there are no libraries in Verilog whereas VHDL stores compiled entities, architectures, packages, and configurations. Verilog was originally developed with gate-level modeling in mind, and so has very good constructs for modeling at this level and for modeling cell primitives of ASIC and FPGA libraries. For this reason, students may find Verilog more digestible than VHDL at first. Because it is geared towards lower level modeling, it is faster in simulations and effective synthesis. It lacks, however, constructs needed for system level specifications. Verilog's simple, intuitive and effective way of describing digital circuits for modeling, simulation, and analysis purposes make it very popular in the industry.

4.1.2 ESL Design

There is a movement towards system level modeling, also called electronic system level (ESL) design. This is the design of an electronic product at the conceptual level, including hardware/software codesign; design partitioning, and specification writing [20]. It demands being able to describe requirements and functions independently of implementation, and being able to talk about interfaces and protocols without describing the actual hardware [19]. Verilog is neither object-oriented nor strongly typed, which makes it cumbersome for system level design. Also, the previously attractive flexibility of its syntax can lead to difficult to detect errors. Neither Verilog nor VHDL provides the syntax or semantics to describe a product at the system level [20]. The trend of RTL engineers moving up in abstraction and systems engineer moving down, as well as the fact that both Verilog and VHDL have shortcomings in the requirements of ESL design, has necessitated the need for either a new language, or the extension of an existing language to bridge the gap between specification and implementation. The new topic of debate is the question of which language is right for ESL design [20].

4.1.3 Extended HDLs

Superlog is an extension of Verilog that includes features that allow a more abstract description of an electronic system [20]. While most of the semantic elements added were borrowed from VHDL, it retains most of the features of Verilog, including support for hierarchy, events, timing, concurrency, and multi-valued logic [6]. Superlog's major technical advantages over VHDL are a clean and powerful interface to C that allows hardware/software codesign, and C-based constructs for system design and decomposition [1]. It borrows useful features from C and Java, including support for dynamic processes, recursion, arrays, and pointers. It also includes support for communicating processes with interfaces, protocol definition, state machines, and queuing. It has been estimated that Superlog needs one half to one third the number of lines of code to describe a function as Verilog at the same abstraction level, and Superlog can go much higher in abstraction [6].

System Verilog. A radically revised version of Verilog was presented at the 2001 International HDL Conference [15]. These changes represent a move towards an even higher level of abstraction for the language and an extension to its capability to verify large designs. SystemVerilog is a blend of Verilog, C/C++, and Superlog that allows module connections at a high level of abstraction [15]. Verilog currently allows the connection of one module to another only through module ports, which can be tedious. SystemVerilog introduces interfaces which makes it possible to begin a design without first establishing all the module connections. C-language constructs, such as globals, are another addition. In Verilog, only modules and primitive names can be global. SystemVerilog allows global variables and functions. SystemVerilog borrows abstract data types from C, such as 'bit', 'char', 'int', and 'logic', which provide more versatility then the existing 'reg' and 'net' types and allows C/C++ code to be included directly in Verilog models and verification routines. Also included is an assertion construct, similar to VHDL's, intended to do away with proprietary assertion languages. Because there's much in Superlog that is not part of SystemVerilog, Superlog will remain a superset of SystemVerilog. With its new additions, SystemVerilog may remove some of the impetus for C-language design, at least for RTL chip designers. The question of whether or not vendors will create tools to support SystemVerilog remains to be seen. [15]

4.1.4 HLL Pros and Cons

Teaching system level design in a High Level Language (HLL) is well suited to students with limited electronics or CAD backgrounds and are unfamiliar with hardware concepts such as signals, voltages, and details of the clock. By starting with either Handel-C or SystemC, hardware/software codesign becomes more accessible to students whose initial programming experience will most likely be C, C++, or Java rather than assembly language [12]. It exposes the students to concurrency, parallelism, software-to-hardware mapping, pipelining, and computer architectures as well programming principles [11]. In Handel-C, for example, each assignment statement takes one clock cycle and each expression evaluation takes no clock cycles, which makes it easy to reason about the number of clock cycles required to execute the code. This relationship encourages efficient compact code form a hardware perspective [11].

However, there is a risk in HLL-based design for the student who already has a software mindset. Specifying hardware using an HDL is not programming, but rather the building of hardware and arrays of gates. Applying general purpose programming tactics to an HDL too often makes too many gates and highly inefficient chip and logic layouts [21].

There are other shortcomings to the HLL approach. One is that it is hard to integrate outside IP with any hardware designed this way. This is due to the fact that close examination of compiler-generated circuits reveals little of their purpose or about how they were generated. Therefore, the "hooks" into the circuitry are not readily apparent. The obfuscated nature of the compiler generated circuits also makes it nearly impossible to hand optimize any of these circuits. These problems stem from the fact that the original HLL on which these new languages are based either cannot express parallelism, or their concepts of memory, methods, and objects map poorly onto real hardware. Thus the new languages are forced to include tools that include the necessary attributes, but at the expense of generating clean hardware. But as one industry expert points out, "elegance of implementation has never triumphed over timesaving hacks. Mnemonics overtook opcodes, compilers overtook assembly, and HDLs overtook schematics. Each time, the old guard maligned the inefficiency of the automated tools vs. the craftsmanship of their methods; but each time automation carried the day [26]."

Many ASIC or FPGA based products include a mixture of algorithmic processing most readily expressed in an HLL and other sets of operations most efficiently implemented directly in gates. FPGAs accommodate these designs by providing CPU cores that can be drawn from a library and implemented in the logic fabric of the FPGA as well as the emergence of devices such as Xilinx' Virtex II Pro which include one or even multiple hard CPU cores embedded directly in the device itself. In systems such as these, use of an HLL based implementation language provides a good fit for implementing the entire job [24].

An increasing amount of system functionality is expressed in embedded software; synthesis and layout are linked into one process, and the typical hardware designer is forced by complexity to work at a high level [14]. S/he would use the ultimate design system, where you wouldn't even care what goes into the hardware or software; you'd write C/C++ code and everything else would just happen under the hood because of an intelligent C/C++ compiler [1]. According to some industry experts, this future may present itself in 5 to 10 years, and those whose career paths extend that far would do well to anticipate it [14].

4.1.5 C Based Languages

SystemC. SystemC is an open source language that is more a structured class library than a language. An argument for SystemC is that the C language lacks the object-oriented facilities that some complex system designs require [19]. SystemC was developed to support system level design. Its class libraries add hardware design-specific modeling constructs that increase the power of the language to meet the needs of hardware design [3]. The class libraries provide data types appropriate for fixed-point arithmetic, communication channels, which behave like pieces of wire (signals), and modules to break down a design into smaller parts. In addition, the class library contains a simulation kernel - a piece of code that models the passing of time, and calls functions to calculate their outputs whenever their inputs change [10]. The syntax is simple and close enough to C++ that students should find it easily digestible.

SystemC partially addresses the problem that C language design presents by creating a number of classes that mimic hardware primitives and time-domain events [20]. Although at present it offers only modeling support, SystemC is moving towards broader capabilities in synthesis [5]. Future versions of the class library will be extended to cover modeling of operating systems, to support the development of models of embedded software [10].

The major drawback of SystemC is the need to convert a C/C++ based description to Verilog or VHDL in order to synthesize it [20]. The problem is that there is not yet a working behavioral synthesis tool available for commercial use that can accept C++ as it's input language. The conversion process is currently a manual decomposition of the design until the designer gets to a low enough point of abstraction such that a commercially available translator allows the use of RTL synthesis. This process, even if done automatically, is prone to errors that are difficult to find [19].

5 Handel-C

Handel-C [4] is both a subset and a superset of conventional C. It does not include functional recursion, floating-point data, or any of the Standard C runtime library functions for I/O or string operations. However, its integer type is augmented with a rich set of operators and declarations for field widths, a *par* construct for expressing parallelism, semaphores and communication channels as primitives, and multiple *main()* functions, each with its own clock [12]. Because it is a variation on C rather than on C++, Handel-C is closer to the hardware than SystemC [18].

Handel-C provides a rich set of code structures including functions, arrays of functions, inline functions, macro procedures, and macro expressions. These facilities allow the student to explore time-space tradeoffs in a design. Handel-C is not tied to any particular family of target devices, although it is clearly aimed at FPGA development in general [13].

The Handel-C development environment supports cycle accurate simulation, allowing students to see multiple statements being executed in parallel using a debugging user interface fully reminiscent of traditional software IDEs. Compiling generates an industry standard (EDIF) netlist, which is then imported into the FPGA vendor's toolkit, where VHDL or Verilog based modules can be integrated and simulated with the Handel-C part of the design if desired. The vendor's tools then perform place and route, and generate a bit stream for downloading to the target device. [26].

Handel-C appears to be an ideal development language for CS students with limited experience in hardware design. But adopting it for laboratory use introduces tradeoffs that need to be considered. In particular, prototyping kits that take full advantage of the language's ability to generate complex systems can add considerably to the cost of laboratory seats. For example, one such kit is the RC200 from Celoxica, which includes a standalone prototyping board with a 1M gate Xilinx

FPGA, audio, video, networking, and memory subsystems and peripherals such as a camera and touchscreen. Fully equipped, this kit costs as much as a complete midrange PC.

6 Conclusion

Trying to evaluate software development toolchains and/or target platforms can be as daunting a task as trying to track emerging trends in design languages. With the recent emergence of FPGA devices so powerful and fast they challenge the one-time undisputed supremacy of ASICS for high-end designs, inexpensive FPGA-based systems, such as those available from Xess and Digilent, emerge as appealing vehicles for CS programs interested in offering students exposure to mainstream technologies of the day. And the main FPGA vendors, such as Altera and Xilinx provide student versions of their development platforms on very reasonable terms for universities.

But logic design using VHDL or Verilog is not as attractive an option for CS students as C-based development languages. We found the Handel-C development environment from Celoxica Ltd. particularly appealing. Hardware is specified in Handel-C, and the resulting netlist is then imported into an FPGA project (using Altera or Xilinx tools, depending on the target), where HDL modules, including IP cores, can be integrated if desired. While inexpensive prototyping kits are available that provide support for logic designs of modest complexity, we believe the options possible using Handel-C and more complex prototyping platforms like Celoxica's RC200 are more suitable for a CS laboratory in computer architecture.

We are in the process of setting up a CS laboratory based on Handel-C and RC200 development kits and will be offering the first class using the lab during the Fall 2003 semester. A major challenge for us is to develop a set of laboratory exercises that guide students in the effective use of this complex environment in a meaningful way. We look forward to sharing our experiences.

7 References

[1] Aldec, Inc. *Evita: Advanced Verilog Tutorial with Applications*. www.aldec.com/Downloads.

[2] Altium Ltd. *CircuitMaker Student Edition.* www.circuitmaker.com.

[3] Bartlett, Joan. "The case for SystemC". *EEDesign*. 7 March 2003.

[4] Celoxica Ltd. *Handel-C Language Reference Manual.* Document Number RM-1003-3.0. 2002.

[5] Clark, Peter. "IP99: Designers see little need to move away from HDLs". *EE Times*. 4 Nov. 1999.

[6] Clark, Peter. "Startup to field next-generation design language". *EE Times*. 31 May. 1999.

[7] Cohen, Ben. *Verification Guild*. Vol. 1, No. 17. 14 Aug. 2000. janick.bergeron.com/guild.

[8] Cummings, Clifford E. *Verification Guild*. Vol. 1, No. 17. 14 Aug. 2000. janick.bergeron.com/guild.

[9] Davidmann, Simon. "It's time for a rethinking of system-on-a-chip design". *EE Times*. 25 Oct. 1999.

[10] Doulos Ltd. *A Brief introduction to SystemC*. www.doulos.com/knowhow/systemc_guide/tutorial/introduction.

[11] Downtown, A.C., Fleury, M., Self, R. P., Sangwine, S. J., and Noakes, P. D. *Hardware/Software Co-Design: A Short Course for Unbelievers*. www.celoxica.com/technical_library/files/"CEL-CUPACPGENHardware Software Co-Design - A short Course For Unbelievers-01002.pdf."

[12] Downtown, A.C., Fleury, R. P., and Noakes, P. D. *Future directions in computer architectures curricula: Silicon compilation for hardware/software co-design*. www.essex.ac.uk/ese/research/mma_lab/Handelc/21Computer.pdf.

[13] Gaffar, A. A. "A Survey on the Handel-C Language" *Surprise Project 1999*. www.iis.ee.ic.ac.uk/~frank/surp99/article1/amag97

[14] Goering, Richard. "Rank and file don't like C". *EE Times*. 15 Nov. 1999.

[15] Goering, Richard. "Standardization nears for next-generation Verilog". *EE Times*. 14 Nov. 2001.

[16] Goodman, S. G. and Vickery, C. *A Laboratory for Computer Organization and Architecture.* NSF DUE-9950364, 1999.

[17] Hamblin, J. O. and Furman, M. D. *Rapid Prototyping of Digital Systems: A Tutorial Approach*. Kluwer, 2001.

[18] Hammes, Jeffrey P. *A High Level, Algorithmic Programming Language and Compiler for Reconfigurable Systems*. www.cs.colostate.edu/cameron/Publications/hammes_enregle.pdf.

[19] Moretti, Gabe. "Get a handle on design languages". *EDN Magazine*. 5 July 2002.

[20] Moretti, Gabe. "System-level design merits a closer look." *EDN Magazine*. 21 Feb. 2002

[21] Motorsabbath. *Hardware design in JHDL*. www.slashdot.org, 16 Jan. 2002.

[22] Murdocca, M. J. and Heuring, V. P. *Principles of Computer Architecture.* Prentice Hall, 2000.

[23] Patt, Y. and Patel, S. *Introduction to Digital Systems.* McGraw-Hill, 2001.

[24] Prophet, Graham. "System-level design languages: to C or not to C?" *EDN Europe*. 14 Oct. 1999.

[25] Shen, J. P. and Lipasti, M. H. *Modern Processor Design*. McGraw-Hill, 2003.

[26] Turley, Jim. "The Death of Hardware Engineering". *Embedded.com*. 28 Feb. 2002.