Hardware/Software Co-design is a starting point in Embedded Systems Architecture Education

Patrick Schaumont
Bradley Department of Electrical and Computer Engineering
Virginia Tech
schaum@vt.edu

ABSTRACT

Embedded Systems Architectures are hard to design, and there is no generally accepted method of doing it. In recent years, this problem has become even harder because of the wide variety of programmable components (FPGA, ASIP, DSP, …). We propose hardware/software codesign as a starting point for teaching the topic. Codesign helps designer-students to think about architecture design in terms of a trade-off between performance and flexibility. Our senior-level undergraduate course in hardware/software codesign includes a hands-on project that requires students to optimize embedded system architecture across the traditional boundaries of hardware and software. We describe a lab series that combines system modeling with refinement on an FPGA board, and that concludes with a class-wide hardware/software codesign contest. The results of the contest clearly illustrate the strengths of 'systems thinking' over 'component thinking'.

1. INTRODUCTION

Embedded Systems Architecture design is the task of selecting and programming a suitable configuration of components for a given system application. Programmable chip companies, with the help of Moore's law, are providing us with amazing selection of components to do this. Traditionally, the creation of embedded system architectures used to be relatively straightforward: use a microcontroller for flexibility and add hardware peripherals for specialized functions. Nowadays, designers can apply multiple component types (e.g. Field Programmable Gate Arrays, Digital Signal Processors, and Application-Specific Instruction-set Processors) to find the optimum over multiple design objectives, including system flexibility, power consumption, design cost, and design time.

Building embedded system architectures is not an easy task. Each programmable component comes with its own design flow and tools, and with its own programming model. Each one presents a separate learning curve to the designer.

This contribution considers how embedded-system educators can help future engineers to prepare for this complex architecture design space. Obviously, it is not feasible to train students in each possible programmable technology – there are too few hours in a semester to do that. In current practice, educators select a single component type (e.g. FPGA), and then teach students how to map and optimize an example application for this component. Educators thus use a thematic, application-driven approach to train students [1]. In order to cover a broader problem space (more component types or more applications), a structured approach to teaching embedded systems architecture may be preferable. This is an important motivation for developing a structured introduction to hardware/software codesign [2].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'04, Month 1–2, 2004, City, State, Country.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

A central idea in hardware/software codesign is to merge two design processes: hardware design uses spatial decomposition and is well suited for performance, while software design uses temporal decomposition and is well suited for flexibility. A successful combination of hardware and software enables designers to obtain solutions that are the right combination of flexibility and performance. Thus, we think of hardware/software co-design as a simplified version of the more complex trade-off that needs to be made during embedded systems architecture design, namely the partitioning between platform architecture and platform function. For this reason, we think that hardware/software co-design is the proper starting point for education in this area.

Among programmable components, FPGA platforms have been very successful in providing a target that equally suits software design and hardware design. Several courses have explored this in the context of codesign [3] [4]. We also note that there is a complementary view to embedded systems design which starts from a software-centric system view (rather than a hardware-centric system view). In that case, the problem being addressed is how to teach architecture-specific software. The Embedded Software consortium in Taiwan, for example, has defined a software curriculum because of the high add-on value that software can bring to hardware design [5]. Vanderbilt University has defined and embedded-software and systems concentration in their engineering curriculum to address the specific needs of embedded software that interacts with electrical, mechanical and other hybrid systems [6].

The rest of this paper elaborates on the need for - and our approach to - embedded systems architecture education. The following section enumerates some of the difficulties for 'newbies' in hardware/software codesign, and we point out possible causes. Section 3 discusses the approach we have followed. Section 4 explains the hands-on project we used in an undergraduate course on hardware/software codesign. The project demonstrated the importance of system-level thinking in embedded system architecture design, and the role that hardware/software codesign plays in it. We conclude the paper with a few open challenges and a positive note.

2. NEWBIE CODESIGN PROBLEMS

A modern undergraduate curriculum in computer engineering tends to use a rather strict partitioning in hardware-oriented and software-oriented topics. Figure 1 shows a typical example. After an introductory programming course, students take courses with a hardware focus or a software focus. It is not until the final-year capstone project that students will experience the full problem space of embedded systems architecture design. This may be too late. In our experience, senior computer-engineering students that follow a curriculum as shown in Figure 1 tend to develop a 'bias' towards hardware design or software design, and this hampers the development of good systems-architecture thinking. The following subsections illustrate some of the difficulties faced by aspiring codesign students.

2.1 Combining Modeling Languages

Although there has been a tremendous research effort in system-level languages over the past decade, these efforts have not yet made their full impact on the curriculum. Therefore, the hardware-branch and software-branch in Figure 1 use different, incompatible modeling – and programming languages (for example, C and VHDL). The semantic gap between these design languages is very large, and it reflects fundamental differences in thinking about design. Consider the following illustrations of the difference between writing C and modeling hardware using RTL (VHDL or Verilog).

- The concept of *time* in RTL and C differs enormously. Software designers write untimed C and hardware designers write event-driven RTL. A common ground between hardware timing and software timing would be to count time based on clock cycles or maybe instructions. Instead, RTL designers insist on event-driven modeling for the occasional asynchronous gate, and C programmers don't want to add timing details that destroy portability. Students are forced to choose their camp.
- The notion of *model* and *implementation* is very different in C and RTL. For all practical purposes, a program in C *is* an implementation. An RTL program on the other hand is a simulation artifact, and the implementation is only available after logic synthesis. In RTL, what you write is not necessarily what you get, which is hard to grasp for designers with a software mindset.
- Nearly identical syntax in C and RTL may mean very different things. A `for`-loop in C is a control-flow construct. A `for`-loop in RTL, on the other hand, is syntactical sugar that is unrelated to the control-flow in the implementation. In fact, if we discount the modeling of state machines using `case` statements, RTL does not offer a good means to model control.

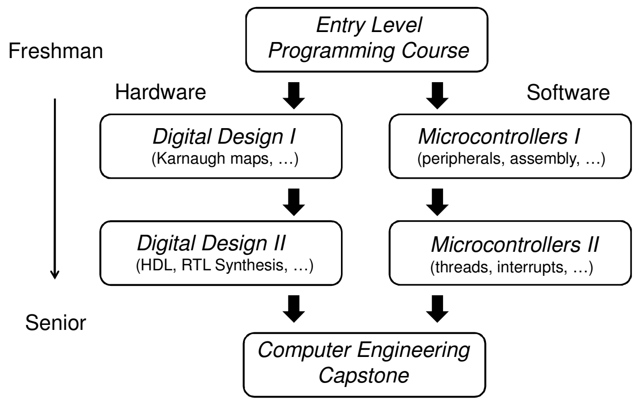

Figure 1: Separate Software and Hardware Tracks in Typical Computer Engineering Curricula

The language differences between C and RTL do not stop designers from excelling in either hardware design or else software design. However, the differences make a combined mastering of C and RTL very hard.

2.2 Designing Interfaces

A second hurdle for students in embedded systems architecture design is the design of interfaces. The efficiency of an embedded system is critically dependent on the efficiency of interfaces (between hardware and software, between coprocessors and processors, etc). Thus, embedded system architects should think of these interfaces as essential design features. Instead, the curriculum in Figure 1 puts the focus on the individual domains, and not on the links that interconnect the domains. As a result, interfaces become a second-class citizen in embedded architecture design.

As a contrasting example, computer science students study the interface between instruction-set architecture and micro-architecture during an entire introductory course ('Introduction to Computer Architecture') [7]. This prepares them to deal with many computer-architecture issues such as pipeline-stalls, memory-bottleneck, etc. For embedded-system engineering students no similar introductory and structured interfacing-course exists.

2.3 Design Tools versus Design Methodology

A final issue for students in embedded system architecture design is the myriad of design tools they need to use. Each programmable component comes with its own design environment, often incompatible with others. The curriculum in Figure 1 promotes the use of such specialized tools, but it does not teach how to combine their use. A second issue is that many aspects of modern design cannot be covered with design tools alone. Some examples include good debugging practice, defensive programming, design for observability, version control, and divide-and-conquer problem solving. These issues, as well as many others, can be collected under the common theme of design *methodology*: the recipe to transform design ideas into design implementation.

Therefore we believe that an embedded systems architecture student will benefit more from a sound design methodology using

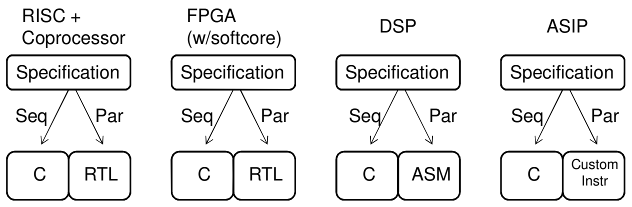

Figure 2: Hardware/Software Codesign is a generic formulation of a problem with multiple instances

simple tools then from fancy and automated design tools that operate stand-alone that that abstract out crucial details of the problem.

3. CODESIGN AS BASIS FOR EMBEDDED ARCHITECTURE DESIGN

In this section, we briefly motivate and outline our course, which is targeted to seniors and first-year graduate students. We present hardware/software codesign to students as a generic solution for a design problem that re-appears in many different forms during the design of embedded system architectures. Specifically, hardware/software codesign targets the combination of a generic processing engine and a specialized processing engine. A designer then maps a specification so as to optimize the efficiency (power, preformance, utilization ...) of the overall architecture.

In all of the cases shown in Figure 2, the generic processing engine is a machine that runs C. The specialized processing engine is, depending on the case, a coprocessor, an FPGA (-coprocessor), a DSP instruction-set, or an ASIP instruction. Although the target platform is different in each case, the underlying design concepts are strongly related, and it makes sense to address them in the context of a structured introduction to hardware/software codesign. The next subsection describes the course topics, while the subsection after that addresses the issue of design tools.

3.1 Course Topics

The hardware/software codesign course contains three parts.

Fundamentals: The first part introduces fundamental ideas in embedded system architectures. On top is a discussion on concurrent specifications and parallel implementation. We use Synchronous Dataflow (SDF, [8]) as an introduction to concurrent specifications. SDF semantics are very well suited for this because of their formal properties in combination with their practical applications (signal processing). Besides a discussion of SDF, we also teach the students how to analyze the control flow and the data flow in a C program. This analysis is very useful when considering architectural alternatives for a C program.

Custom Architecture Design Space: A second part in the course is an in-breadth discussion of the custom architecture space. As illustrated in Figure 3, we start with finite-state-machine-with-datapath models, and gradually proceed to System-on-Chip

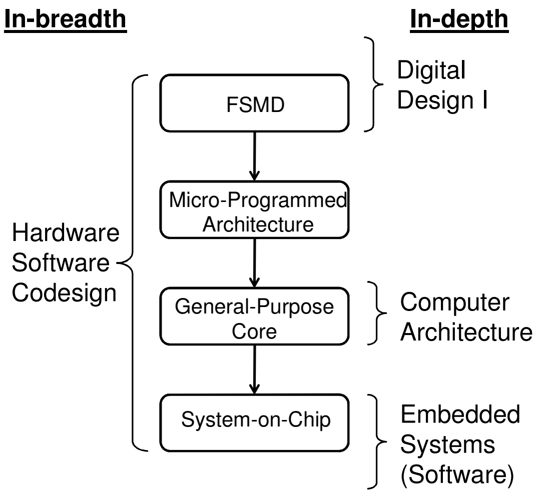

Figure 3: Covering the custom architecture-space in-breadth rather than in-depth.

architectures. The in-breadth discussion of architectures starts with typical 'hardware' targets and proceeds to typical 'software' targets. This way, students learn that there is only a single design space. It is easy to show that each step in the sequence of Figure 3 is an improvement over earlier targets in terms of flexibility. Obviously, in-breadth discussions imply that we cannot address the full details of each architecture. However, our discussion is complementary to the in-depth approach of existing courses: FSMD, general-purpose cores and system-on-chip may be addressed in detail in a digital-design course, a computer architecture course, and an embedded systems software course respectively.

Hardware/Software Interfaces: The final part of the course is a discussion on hardware/software interfaces. We describe how C communicates with a specialized processing component such as a coprocessor or a custom datapath. This is a broad topic, and it includes a description of data communication from C to hardware (interfaces and buses), synchronization methods (handshakes, mailboxes, queues), encapsulation of software (custom instructions, API's) and encapsulation of hardware modules.

3.2 Tools and Design Flow

The course topics are augmented with an intensive hands-on part. Because of the difficulties with a mixed-language approach based on C and RTL, we are using GEZEL, a codesign modeling - and cosimulation environment [9]. GEZEL combines a design language for FSMD with custom interfaces to instruction-set simulators. GEZEL provides cycle-based co-simulation and a path to implementation by converting the FSMD models into VHDL. Students write co-designed models by combining C and GEZEL language. In comparison with the use of RTL and C for codesign, we address the following issues:

- Hardware is expressed using cycle-based, single-clock FSMD modeling. These models result in a compact syntax, close to implementation. In GEZEL, a `reg` variable is really a flip-flop. In our experience, a simple mechanism to express hardware models is very important to enable students to concentrate on methodology and design. Some students enter

```
1.  //-----------------------------------
2.  // ARM core with FSL interface in GEZEL
3.  ipblock arm1 {
4.    iptype "armsystem";
5.    ipparm "exec = exec.elf";
6.  }
7.
8.  ipblock fsl1(out data   : ns(32);
9.               out exists : ns(1);
10.              in  read   : ns(1)) {
11.   iptype "armfslslave";
12.   ipparm "core=arm1";
13.   ipparm "write=0x80000000";
14.   ipparm "status=0x80000004";
15. }
16.
17. //------------------------------------
18. // GEZEL FSMD, reads from FSL, accumulates
19. dp avg_fsmd    (in  rdata   : tc(32);
20.                 in  exists  : ns(1);
21.                 out read    : ns(1)) {
22.
23. reg rexists : ns(1);
24. reg acc : tc(32);
25.
26.   always        { rexists = exists; }
27.   sfg doread    { read  = 1;
28.                   acc  = acc + rdata; }
29.   sfg dontread  { read  = 0; }
30. }
31.
32. fsm fsm_avg_fsmd(avg_fsmd) {
33.   initial s0;
34.   @s0 if (rexists) then (doread)   -> s0;
35.                    else (dontread) -> s0;
36. }
37.
38. //-------------------------------------
39. // C driver for ARM core with FSL interface
40. void sendarray (int *in,
41.                 unsigned length) {
42.   volatile unsigned int
43.     *wchannel_data   = (int *) 0x80000000;
44.   volatile unsigned int
45.     *wchannel_status = (int *) 0x80000004;
46.   int i;
47.
48.   // send content of in[]  to FSL link
49.   for (i=0; i<length; i++) {
50.     while (*wchannel_status == 1) ;
51.     *channel_data = in[i];
52.   }
53. }
```

Listing 1: This HW/SW model accumulates a data stream.

our class without previous knowledge on RTL programming (Verilog is only covered in an elective course).

- GEZEL offers access to HW/SW interfaces as library blocks in the language. These library blocks expose the pin-out of interfaces without burdening the model with unneeded internals. The library blocks reflect actual interfaces from prototyping environments; their use guarantees that the GEZEL FSMD can be easily connected to the prototype once they are converted to VHDL. Listing 1 illustrates one type of HW/SW interface (Fast Simplex Link), connected to a software driver and an FSMD. The overall operation of this model is to send an array of integers from software to hardware over an FSL link, and to accumulate the sum of these integers in the hardware model. Some features of the model are as follows. Lines 8-15 show the FSL link. This interface is modeled after the tightly-coupled Fast Simplex Link found in Xilinx' MicroBlaze processor, and has a data port and two handshake signals `exists` and `read`. The interface is attached to a core `arm1` (line 12), and this core will control the interface through memory locations `0x80000000` and `0x80000004` (a memory-based emulation of the FSL protocol is needed since the ARM simulation model used by the cosimulator does not have dedicated instructions for the FSL interface). The FSMD module that connects to this hardware-software interface is shown on lines 19-36. The datapath has an `always` instruction that executes every clock cycle (line 26), as well as two instructions `doread` and `dontread` (lines 27-29) that will only execute when told so by the FSM controller on line 32-36. The top-level hardware module, which interconnects the FSMD and the cosimulation interface, is not shown in Listing 1. Finally, a software driver that communicates with the FSMD through the hardware/software interface is shown on lines 40-53. The use of `volatile int` pointers ensures that the C compiler does not optimize the apparently redundant memory-read and memory-write operations.
- The cosimulation environment is interactive, and command-line driven. The model in Listing 1 would be captured in two files (e.g. `sw.c` and `hw.fdl`) and would be simulated by cross-compiling the embedded software, and next by running the cosimulator:

```
> arm-linux-gcc sw.c -o exec.elf
> gplatform hw.fdl
```

The design environment used by the students contains the following elements:

- A KNOPPIX CDROM with a pre-installed GEZEL-based codesign environment, including all cross-compilation and simulation tools.
- A Xilinx-based EDK+ISE environment, which is used as a backend for code created in the codesign environment.
- A Spartan-3E Starter Kit with a baseline configuration including Microblaze, on-chip timer, off-chip DDR RAM memory.

4. A LAB SERIES TO INTRODUCE CODESIGN

As described in the previous section, the students in the codesign course make use of a KNOPPIX cdrom (for modeling and cosimulation) and an FPGA board with design software (for prototyping) [10]. Figure 4 shows the organization of the hands-on experiments based on these tools. An initial lab series familiarizes students with the use of the tools. That experience then converges into a *Codesign Challenge*, a competition that challenges students into building the fastest possible implementation of a given C program onto an FPGA.

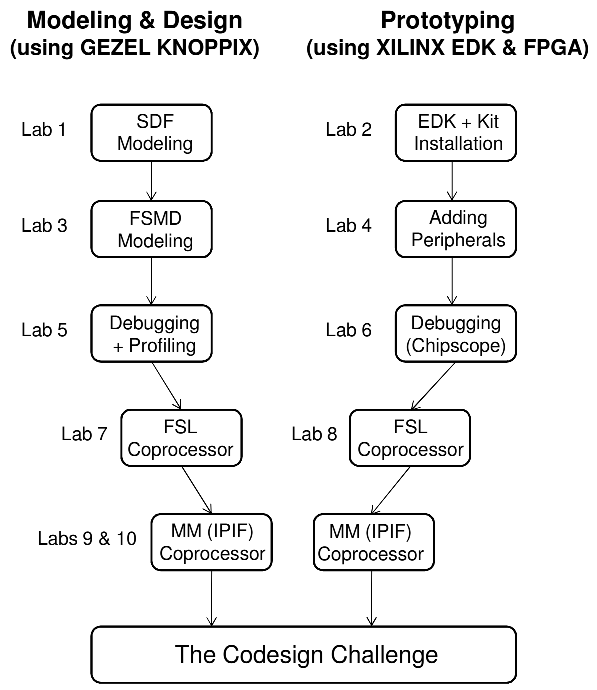

Figure 4: The Lab Series combines modeling and prototyping and merges into a 'Codesign Challenge'

4.1 Lab Series

A set of 10 lab projects prepare students for the codesign challenge. Five lab projects cover modeling and design; five additional projects cover the use of the FPGA prototyping environment. The two tracks of lab projects gradually converge, so that the codesign environment is coupled to the FPGA design flow. Students then are able to complete the following tasks: convert a single C program into a combination of a C program and a hardware coprocessor; verify the resulting design using cosimulation; port the coprocessor and C program to the FPGA platform; and verify the performance of the design in the resulting prototype.

The modeling assignments include SDF & FSMD modeling, profiling of embedded software using an instruction-set simulator, and two coprocessor designs. The coprocessors use a tightly-coupled Fast Simplex Link and a general-purpose Memory-mapped interface.

In the complementary prototyping assignments, students learn to take the output of the modeling flow (software driver and generated coprocessor VHDL) and connect that into the FPGA environment. They also familiarize themselves with the numerous available platform architecture parameters (location and size of memories, configuration of buses, optimization during software compilation and hardware synthesis, etc).

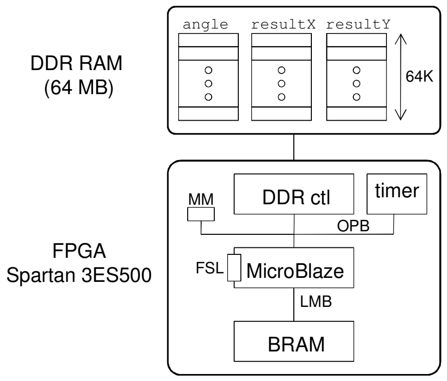

Figure 5: The 'Codesign Challenge' in fall 2007 was to optimize 64K CORDIC rotations.

The combined use of GEZEL cosimulation and EDK synthesis is done for several reasons. First, EDK has a steep learning curve compared to the simplicity of GEZEL. Second, even though EDK can hide many of the details of the underlying machine, the cosimulation mechanism is still quite cumbersome as it requires elaboration of the hardware model. Finally, the students that take the codesign course may not have taken a course on VHDL or Verilog yet, since the latter is an elective at the authors' institution. However, it should also be clarified that the author has not yet attempted to organize this lab sequence solely using EDK.

4.2 Codesign Challenge

The Codesign Challenge is deliberately set up as an open-ended assignment to improve the performance of a given C specification as much as possible. The constraints are the reference specification, the resulting platform, and the design time (two weeks). The initial ranking of results is based on absolute performance of the student designs, even though the in-class discussion of the results is based on Pareto-optimality of performance and resource usage. In the following, we describe the results of the Codesign Challenge we ran in the fall semester of 2007. In that class, 28 students participated in the final project.

Figure 5 shows the target platform for the Codesign Challenge. A DDR RAM memory contains a vector `angle` with 65536 values. The design in the FPGA must read this vector, and transform each element in the vector to an (X,Y) tuple representing the sine and cosine of that angle. The result is stored in the DDR RAM. A CORDIC algorithm with 20 successive rotations is used for this transformation [11]. The reference design in the FPGA includes a MicroBlaze processor with local memory and a peripheral bus system with timer. The students also received a software implementation of the CORDIC design for this reference platform. The performance of their design is measured as the wall-clock time needed to rotate 64K vectors stored in off-chip memory.

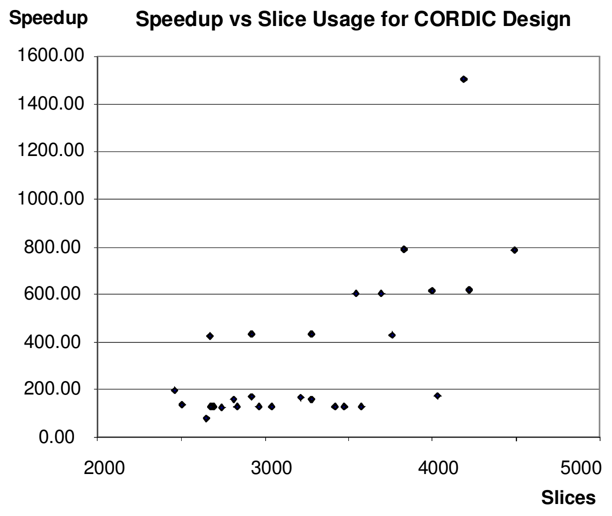

Figure 6: The results of the Codesign Challenge. Design performance is measured as a speedup over the intiial reference implementation. Each dot is one student.

After two weeks of design time, the students turned in their results. Figure 6 reflects the speedup of their design (over the reference software implementation) versus resource usage. A wide variation can be observed for each parameter - as could be expected in such an open-ended assignment. Figure 6 shows that the student designs can be partitioned into in two groups: those with a speedup lower than 200, and those with a better speedup. All of the students implemented a hardware accelerator for the CORDIC computations, with a typical design computing a rotation in 20 clock cycles. However, as can be observed from the system architecture in Figure 5, each rotation requires three off-chip memory accesses, and the performance of the reference design was around 20 cycles per memory access. The designs bounded at a speedup of 200 are those that did not take this bottleneck into account.

About one third of the class figured that hardware acceleration alone would not save the day and that further optimization of the system architecture was necessary. The results obtained in their designs illustrate the case that embedded system architecture design goes beyond hardware/software partitioning. Among the optimizations performed by this group of students are the following:

- Allocate multiple coprocessors, and exploit the parallelism available in 65536 independent CORDIC rotations;
- Write driver software that overlaps input/output communication with coprocessor computation;
- Move the `.text` segment of the code to on-chip memory and free up the system communication bus.

Two students from those with previous Verilog experience decided to develop hardware directly in Verilog (rather than in GEZEL) for compactness and performance; however, the resulting system was harder to debug because cosimulation at RTL is inadequate and time-consuming.

Another interesting issue is how students allocated design time under a limited time budget. The best designs were those that focused (almost exclusively) on the embedded system architecture and the implementation of system-level data streams. Those top students were also able to decide how much effort a given optimization was worth, because they started by estimating the performance limits in their platform.

5. CONCLUSIONS & FUTURE WORK

The breakneck speed of technological development enables senior students to complete in a class project what was considered a high-end design 10 years ago. The technologies and tools are available, and they are cheap. However, the computer engineering curriculum is lagging. The idea that digital design education should start with digital gates is losing its relevance when designers are no longer concerned with individual gates. Educators are in need of more effective design abstractions. We have used hardware/software codesign as a step towards structured thinking about embedded systems architecture design. It is not possible to cover each programmable technology that is being proposed today. Hence, our objective is to produce engineers that can quickly adapt to new programmable systems architectures.

As a positive note, the embedded systems design space has never been more interesting and offered more opportunities, for students and educators alike. There are significant opportunities available in the computer engineering curriculum for new tools, subjects, and course projects.

6. REFERENCES

[1] M. Grimheden, M. Torngren, ”What is Embedded Systems and How Should It Be Taught?” ACM Trans. on Embedded Computing Systems (TECS), 4(3): 633–651, ACM Press, NY, 2005.

[2] P. Schaumont, "A Senior-level Course in Hardware/sofwtare Codesign,", MSE 07, 7-8, June 2007.

[3] R. Chamberlain, J. Lockwood, S. Gayen, R. Hough, and P. Jones, "Use of a soft-core processor in a hardware/software codesign laboratory," in Proc. IEEE Int. Conf. MicroElectronics System Design Education, Anaheim, CA, 97-98, June 2005.

[4] C. Bieser, K.D. Muller-Glaser, and J. Becker, "Hardware/software co-training lab: From VHDL bit-level coding up to case-tool based system modeling," in Proc. IEEE Int. Conf. MicroElectronics System Education, Anaheim, CA, 134-135, June 2005.

[5] S. Tsao, T. Huang, C. King, "The Development and Deployment of Embedded Software Curricula in Taiwan," ACM SIGBED Review, 64-72, 2007.

[6] J. Sztipanovits, G. Biswas, K. Frampton, A. Gokhale, L. Howard, G. Karsai, J. Koo, X. Koutsoukos, D. Schmidt, "Introducing Embedded Software and Systems Education and Advanced Learning in an Engineering Curriculum," ACM Trans. on Embedded Computing Systems, 4(3):549-568.

[7] D. Patterson, J. Hennessy, "Computer Organization and Design: The Harwdare/Software Interface," MKP Publishers.

[8] E. Lee, D. Messerschmitt, "Synchronous Data Flow," Proc. IEEE, September 1987.

[9] GEZEL homepage, `http://rijndael.ece.vt.edu/gezel2`

[10] P. Athanas, C. Patterson, "A Holistic Approach towards a Unified CpE Laboratory Platfrom", MSE 07, 73-64, Juen 2007.

[11] R. Andraka, "A Survey of CORDIC Algorithms for FPGA's," Proc. FPGA 1998, 191-200, 1998

7. APPENDIX A: Course Topics

The following enumerates the lecture topics of the course. This material is covered in approximately 22 lectures, excluding reviews, midterms, and exam.

Fundamentals

- What is hardware-software codesign?
  - Flexibility versus performance
  - Concurrent and sequential specifications, parallel implementations
  - Modeling abstraction levels
- Synchronous Data-flow Modeling (as an example of concurrent specification)
  - Semantics of SDF
  - Analysis of SDF
  - Implementing SDF in Software
  - Implementing SDF in Hardware
- Control Edges and Data Edges (of a C program)

The Custom Architecture Design Space

- FSMD: a systematic but inflexible model for hardware
- Micro-programmed Architectures: Flexible, scalable control, but hard to program and optimize (e.g. pipelining)
- General-purpose Embedded Cores: Flexible, scalable control, easy to program, but hard to specialize
- System-on-Chip: Flexible, scalable control, easy to program, feasible to specialize

The Custom-Hardware/Software Interface

- On-chip busses
- Memory-mapped interfaces
- Coprocessor (dedicated processor-HW) interfaces
- Control Design in Co-processors
- Intellectual-property Interfaces
- Advanced solutions
  - ASIPs
  - Using C for Hardware Design