Essential and Elective Topics: A Proposal for the Content of Reconfigurable Computing Courses

Ron Sass, Member, IEEE and David Andrews, Member, IEEE

Abstract—The introduction of FPGAs around twenty years ago was a watershed moment for the field of Reconfigurable Computing. Since then, steady gains in device capabilities and the supporting technology has sparked wide-spread interest. While some have begun offering special-topic courses (and one-time seminar courses abound), relatively few institutions offer regularly-scheduled, permanent courses. That is likely to change, as this workshop presciently observes.

If the emerging discipline of reconfigurable computing was more mature and more courses existed, then gathering empirical data about pedagogical techniques, topical coverage, and lab-related exercises would be possible. However, this is not the case. At this stage, we must resort to reasoned arguments, limited experience teaching the topic in ad hoc settings, and gut instinct. This article proposes a set of essential and elective topics in a Reconfigurable Computing course based on anecdotal evidence and the authors’ experience as both students and teachers in the field for over the last decade. The article does not attempt to argue that the set of topics is optimal; rather the goal is to present a starting point. By initiating a dialog in the international community, the aim is to speed the process to consensus. Such a consensus has several advantages, including the ability share materials and experiences and setting up a marketplace of ideas to steadily improve the teaching of the field. These advantages could be slowed if consensus is not reached early in the development of the discipline.

Index Terms—reconfigurable computing education, reconfigurable computing topics, curriculum development, graduate and undergraduate education

I. INTRODUCTION

FPGAs are rapidly becoming first-class design components in modern computing systems. Promoted from its prior role as a behind-the-scenes convenience (i.e., ’glue logic’) to its broader role as a compute/communication assistant, systems today are making FPGAs available to users as programmable elements in the architecture [1], [2]. However, the idea of using a reconfigurable logic to develop applications is so dramatically different from the skills or theory presented in existing courses, it demands its own treatment in a new course.

As is the case with any emerging technology, numerous special topic, seminar, and tutorial classes have been offered at various institutions and venues. Often these classes are designed to meet a specific need: a local RC research program, an embedded systems/signal processing project, or perhaps a desire to accelerate a particular high-performance computing application. While these courses address specific and immediate needs, there is a subtle long-term issue that they fail to address. Namely, there is no uniformity from institution to institution (or from semester to semester, for that matter). There are advantages and disadvantages to uniformity (see the next section) but if, for the moment, one assumes that uniformity is desirable, then the question is, ’what belongs in such a course?’ This includes (i) what are the core topics? and (ii) how much uniformity should be expected?

The contribution of this article is two-fold. (1) We argue that a common core of essential topics is good for the community and the advancement of the field. (2) We propose a set of core Reconfigurable Computing (RC) topics and a sample of topics that can be used to specialize the course. Note that we do not presume that the proposed topical coverage is either ideal or optimal. Rather, the aim of this work is to begin a dialog in the community. Other disciplines have had the luxury of many years (in some cases, hundreds of years) of engineering education in which to develop a set of best practices. In those disciplines, many individuals have invented new teaching tools, introduced core topics, and experimented with student exercises. Over a long period of time, the most successful features of these courses become permanent and migrate to other schools. That is, a standard emerges. However, by initiating this dialog, we wish to accelerate this process by artificially seeding the marketplace of ideas. Our hope is that we can avoid waiting dozens of years for consensus to be reached in the emerging discipline of Reconfigurable Computing.

The rest of this short article is organized as follows. In the next section, we expand on the basic premise — the need for a set of common topics. In section III, a core set of common topics followed by a set of optional topics that might appear in a course with ‘reconfigurable computing’ in its name are detailed. A specific example of how this proposed curriculum might be implemented is described in section IV. The paper concludes with a restatement of the goals and contributions of this work. As an aside, we include an appendix of important, but unanswered questions regarding RC education.

II. MOTIVATION

Why bother to propose a set of core material for an RC course? Standardization stifles creativity and, for sure, a flawed standard could cause a great deal of grief to the community. Those points notwithstanding, there are several compelling reasons for developing a core set of topics.

First, developing a common vocabulary offers a big advantage to the discipline. By agreeing to what belongs in a
basic "reconfigurable computing" course, it is much easier to communicate. Nothing prevents courses from being tailored to specific niches but if the course title contains the term, then someone unfamiliar with the course or institution would still at least have an idea of the minimal set of topics covered. For example, someone in the automotive industry may look at a prospective employee's university transcript and see that they took a Reconfigurable Computing class. The class may have emphasized computational biology but the interviewer could reasonable expect that certain topics, such as pipelining, were covered. Of course, there are numerous things that might cause the student to be unprepared for the job: the quality of school, the proficiency of the teacher, and the individual student, for example. However, having a universal understanding of what should be covered is a marked advantage over no definition at all.

A second advantage has to do with training versus education. It is a natural tendency for each individual department (and instructor) to adapt a course for their own agenda. This happens subtly and perhaps unconsciously in established courses all of the time. The instructor's agenda manifests itself in the assigned laboratory exercises, in-class examples, and in how the material is emphasized. In general, this is good for established courses because variation leads to innovation. However, there is a danger that every instructor faces when developing a new course that the resulting course produces narrowly-equipped students that specially targeted a specific, local project (such as the professor’s research program). Even if the goal is to "get a bunch of new students up to speed," having a portion of standard topics covered has long-term advantages. By educating the student in Reconfigurable Computing, the course can be designed to both serve the immediate agenda and also the broader goal of an RC education. It also has the benefit of being able to reuse material developed elsewhere. From the students perspective, it has the potential of aiding them later in their career — perhaps in unintended ways. This is the nature of college education.

A third — and perhaps the most compelling — reason to agree on a common core, is that it facilitates a marketplace of ideas. By having a framework of key concepts that is widely taught across many different institutions (and perhaps in slightly different courses), then different educators can explore different ways of explaining the ideas. Individual instructors will design different exercises (on paper and in the lab) to challenge the students’ understanding of the material. Details such as topic order, depth of coverage, and pre-requisites are variables that different instructors and institutions will adapt, based on their teaching style, experience, and local curricula. With a common core, these decisions and their results can be disseminated to others teaching in the field. There, they can be evaluated and incorporated. Competitive forces will help shape the ideal basis for a Reconfigurable Computing course. Evaluating the effect of an adaptation may be self-evident: “ah ha, that’s a much better first-lab-exercise than the one we are using.” Alternatively, it may be measurement based: “a larger percentage of students met the expected outcomes of the course.” However, if there was no common set of topics, then different ideas cannot be exchanged and the advancement of the field is handicapped.

III. PROPOSED COURSE TOPICS

Since students, teachers, and curricula vary from institution to institution, the focus of a common core should be restricted to just 8–10 weeks of a 15 week curriculum. This allows for adding introductory material at the beginning or advanced, specialized material at the end. Each of the four main topics are described below followed by a list of specialized topics.

A. Composition (six 50-minute lectures)

Behavioral design has largely supplanted the schematic-capture or structural styles of designing basic computational cores. This is partially due to ease-of-use and productivity issues but also because it is becoming increasingly important for designers to rely on the (behavioral) design compiler to find and synthesize optimal implementations. However, once these basic cores have been designed, larger systems have to be constructed. These systems are still predominantly assembled structurally using hierarchy and instantiation to compose encapsulating designs from simpler ones.

While there are tools from various vendors to build these systems, it is an active area of research (which may end up transforming how we do system composition). The current state-of-the-art is pretty much just that — an art of balancing complexity and efficiency. Competency in this area means being able to design basic digital cores and then compose systems from those simple cores.

1) Core Design: At the heart of composition is the design of basic, computational cores. There is a variety of styles (sometimes called models) that are used to express the desired computation. They exist on a spectrum that ranges from a state machine-driven data path of components to a desynchronized network of components. The former is commonly used for control-oriented operations (and is a common target for high-level languages since it closely matches the operation of a simple processor). The latter is common in data intensive computation such as signal processing and high-performance computing applications. In practice, it is common to combine the two by building networks of state machine-driven components.

2) Hierarchical Design: As we approach billions of effective logic gates on an FPGA, the system designer has to find productive ways of managing the complexity of options. This mirrors the situation faced by software programmers in the 1970s. Faced with millions of lines of code, system analysts had to start thinking about appropriate abstractions that captured functionality and interface in a mnemonic package. Part of teaching abstraction to students includes the ideas of information hiding and the cohesiveness of the design. The next idea is that of instantiation. To students familiar with (discrete component) hardware design, this is a fairly easy concept, but expressing it in a textual manner is important. The associated issue of how to interface with components is part of this. Finally, the key concept of explicitly designing for reuse should be included. This means teaching the student the costs and benefits of building productivity-enhancing components.
3) **Exercises:** The challenge in the classroom here is to develop spatial design maturity. Most students will have had a computer programming course, and some may extensive software experience. However, it is crucial that they break away from the cycle-dependent model of sequential operations and begin to get comfortable with the idea of simultaneous operations.

There is an enormous challenge when designing exercises for these concepts because the tools and technical training to simply generate a design are complex and time-consuming. It will not be enough to give the student a lab tutorial where they type in the various options, press the right buttons, and then out pops a working system. That relies on the student to recognize the choices that have been made and infer their consequences of choosing an alternative. It does not help them make value-based choices. Towards this end, the exercises need to both train in the tools, expose the students to the consequences of bad choices, and help them realize the advantages of good design choices. Typical exercises might include:

- cost of building a higher level component from first-principles versus simpler components
- bus-based approach to adding high-level cores
- simple state machine/data path designs
- network of state machines
- inefficiency of bad abstractions

**B. Partitioning (nine 50-minute lectures)**

Not to be confused with hardware/software co-design, partitioning is a subset of the well-known research area. Partitioning is concerned with what parts of the application will be implemented in a conventional, von Neumann style execution engine (i.e. processor) and what parts will be implemented in a spatial network of components (i.e. configurable logic blocks or “reconfigurable hardware”). Since the logic blocks are all digital in nature, this side steps co-design issues related to mixed-signal design and analog components altogether. Despite being simpler, it is still a fundamental challenge for which all students should be equipped to find reasonable solutions.

Two main concepts are key to the partitioning problem.

1) **State:** The first is how to maintain consistent global state, and that usually involves careful attention to interfaces and marshaling mechanisms to explicitly move data from the registered memory or cache of one component to another component.

2) **Performance Gain:** The second key concept is the idea of performance gain. Performance is a multi-faceted metric (including execution rate, power, jitter, and others) and as such there are a number of ways of estimating or measuring the performance gains of reconfigurable hardware implementation. (For example, execution rate can be estimated by running a software reference design with profiling turned on.) Then students can be shown how performance gain per resources required can be used to guide the partitioning decision. Finally, as part of partitioning, students should be exposed to any number of pragmatic techniques used in practice, since this is far from an exact science.

3) **Exercises:** There are several typical exercises that help a student understand these concepts. For example,

- several marshaling exercises are possible — a specific exercise includes identifying effected state
- using profiling tools to characterize impact
- restructuring applications to maximize profiling (and performance) impact

**C. Parallelism (six 50-minute lectures)**

**D. Throughput (nine 50-minute lectures)**

One of the most crucial, and often overlooked, aspects of building a high performance reconfigurable computing design is maximizing throughput. The issue arises when a path bifurcates and then later rejoins in a network of operations. If the parallel paths have a different latency (not the same number of pipeline stages) then the resulting performance of a continuous transfer of state will be decimated.

The term parallelism as a compute model comes from electrical circuits where two components can be arranged spatially in parallel — as with the two resistors of Figure 1(a) — or in series, as shown in Figure 1(b). In the physical world, all of the components are active, all of the time. However, in the compute model, information flow is discretized and simply being “in parallel” does not guarantee that operations happen at the same time, all of the time.

The principle concept of this chapter is rooted in network flows; in particular, the dual of the max-flow, min-cut theorem. This provides a simple, polynomial-time algorithm for determining the number of buffers needed to guarantee maximum throughput. Issues such as how to handle variable latency paths, FIFO v. buffer, and implementation techniques (flip-flops v. shift registers) are important to this segment.

Teaching throughput analysis is an important and oft-overlooked topic of Reconfigurable Computing. The problem
arises in systems that are composed of networks of pipelined components. It is easily understood with a simple example. Consider the network shown in Figure 2(a). It is designed to compute \( \sqrt{x} + x \) where \( \sqrt{x} \) is a 16-stage, pipelined core. That is, it takes 16 cycles to produce a result but it can accept a new argument every cycle. Both networks in (a) and (b) take 17 cycles to produce a result for a single \( x \) value. However, if more than one \( x \) value is needed, then we are more interested in the throughput of the network. Consider computing a whole vector of values, \( \vec{x} \). The throughput of the network (a) is one result every 17 cycles. However, because of the FIFO in the lower path, the network (b) can produce a result every cycle. As \( |\vec{x}| \rightarrow \infty \), the latency becomes insignificant and throughput dominates the overall performance.

1) **Buffer Insertion:** Fortunately, there is a polynomial-time algorithm that finds the minimum number of buffers needed for any fixed network of fixed latency components. It is useful to introduce this concept from its Integer Linear Programming foundation before developing the polynomial-time algorithm that solves this. It is also worthwhile to show the development step of mapping buffers to the edges of the network, which has a huge impact on resources of an FPGA. (For example, two discrete 2-deep FIFOs use more resources than one 16-deep FIFO.)

2) **Variable Rate Components:** For many digital signal processing applications, loop bounds and other aspects of the problem are fixed and known at compile-time. This is a necessary condition for the polynomial-time algorithm to work. However, in general, variable-latency components exist. These components arise in a number of situations such as when a component iterates until a tolerance is met, or when some components are designed to "short-circuit" for special cases (such as when a divider is asked to "divide by one"). It also arises when the components are composed hierarchically. Thus, students need to be exposed to effective heuristics to design high-throughput networks that involve handling variable latency components.

3) **Component Selection:** A third built-in assumption of the initially posed problem is that every desired function has exactly one component that implements it. In fact, many basic cores are designed with generic parameters that allow the person using the core to choose the number of pipeline stages during synthesis. Also, there are numerous ways of implementing, for example, addition. A bit-serial adder is very efficient in space and — if it is on a path that is parallel with a long-latency path — then we save adder resources and buffer resources with no loss of throughput! Moreover, there are other high level optimizations (such as algebraic manipulations) that impact throughput. Most of these can be posed as an Integer Linear Program although in current practice, implementation appears to rely on designer intuition and experience.

4) **Exercises:** Typical exercises might include:

- network flow formulation and problems
- percentage of resources needed
- placement of buffers and resource minimization

**E. Specialization Topics**

In addition to the core, many institutions will find it necessary to include preliminary material. Furthermore, courses designed to support a research focus area are likely to include research-specific topics. Finally, teachers interested in using FPGAs in their domain will want to augment the course with application-specific material. Examples of these topics are given below.

- HDLs and Simulation.
- Low-Level Device Details and Synthesis.
- Taxonomy/History of RC Architectures.
- Complex Performance Metrics (such as designing for power or designing for real-time constraints or designing for high-performance computing).
- Electrical Interfacing (such as low-level networking and interfacing to standard microcomputer buses).
- Real-Time Operating Systems.
- Run-Time Reconfiguration.
- Evolutionary Computing Models.
IV. HIGH-PERFORMANCE EMBEDDED COMPUTING

To see how these topics might be realized in a course, consider the following prototypical 15-week course designed to teach embedded systems design with Platform FPGAs. The audience is assumed to be seniors or first-year graduate students with a computer science or computer engineering background. As such, they will have a fairly strong background in software, systems, and architecture but perhaps only one or two courses in digital design. The goal is for the students to become proficient in the design and implementation of embedded systems that are composed of software and a collection of special-purpose cores. (It is not expected that such a course would replace an advanced digital design course.)

A. Preliminaries

The first four topics covered introduce the main ideas and issues of RC. The first is a broad introduction to the subject. Since the presumed pre-requisite material is a basic computer science/engineer background, the next topic emphasizes the issues relevant to embedded systems design. For example, it would include exposure to making cost-effective engineering decisions in terms of development costs, manufacturing, time-to-market, as well as execution speed. The third preliminary topic introduces the complexity and intricacies of a Platform FPGA. This includes the FPGA fabric, the plethora of diffused IP, and a review of structural design. The fourth topic addresses software: compiling applications and system software for an embedded (soft or hard) processor on an FPGA.

B. Common Core

The common core, as described in the previous section, would form the heart of the course. This would include segments on system composition, partitioning between hardware and software, the different forms of parallelism, and throughput analysis.

C. Specialization

To round out an embedded systems course, two or three additional topics are needed. Clearly, a segment on real-time operating systems and the various scheduling policies is needed. A second segment on networking is also critical, since so many embedded systems now have communication components to them. A third potential segment might be an introduction to run-time reconfigurable systems.

While this topical outline is appropriate for an embedded systems course, it seems likely that other domains would be able to combine the common core with a selection of specialized topics to meet their immediate needs while maintaining the uniformity which was the proposed goal of this effort.

V. CONCLUSION

The field of Reconfigurable Computing is growing rapidly, and the Field-Programmable Gate Array is at the forefront. While a number of traditional digital design courses use FPGAs — either for prototyping or as a ‘glue logic’ — there is no standard for teaching the principles and practices of deploying reconfigurable computing systems with FPGAs playing a central role.

This article describes a proposal for a set of essential and elective topics for Reconfigurable Computing courses and argues that a degree of uniformity across institutions is valuable. By initiating a dialog in the international community, the aim is to create a marketplace of ideas that will help the education community keep pace with the rest of this rapidly advancing field.

APPENDIX

- What department does the course reside? Or, more specifically, do you assume a CS background — or, a computer engineering background?
- How much attention is payed to practical tools?
- As research ideas transition from testable hypotheses to stable facts in the body of knowledge, courses get updated. However, with the genesis of a new course, where do you draw the line between ideas under investigation and those that have been vetted?
- Devices are rapidly evolving — indeed, in just four years we witnessed FPGAs change from uniform gate arrays to irregular Platform FPGAs. Can you ignore the heterogeneous aspect of FPGAs and only focus on the configurable logic?
- How do you separate tool training from tool concept? Building a system today requires so many practical steps and so much practical knowledge, how do you convey concepts without being so generic that it is meaningless?
- What proportion of structural to behavioral style HDL should you include?
- With greater complexity in the devices, how do you manage the diverging devices in the field?
- Hardware needs a design/complexity metric like O() in the software world. What should it be and how do you teach it?

ACKNOWLEDGMENT

The author would like to thank the many talented students in the lab that have worked in the field without a guide: much of what the author has learned about how to teach this field should be credited to those ad hoc sessions. The author would also like to thank Xilinx in general for its continued support and, specifically, for initiating the book project. Tiffany Gasbarrini and Denise Penrose, at Morgan-Kaufmann and Imprint of Elsevier, have also provided valuable support and feedback, thank you.

REFERENCES