Report on

Workshop on using Digital Logic Instruction to Introduce Computing Concepts

Award no. 1550985

Ramachandran Vaidyanathan
Jerry L. Trahan
Suresh Rai

Louisiana State University, Baton Rouge

Summary

The purpose of this project is to address ways to thread the coverage of important topics, a little at a time, through a sequence of courses. The idea is to introduce key concepts early on, starting from lower-level courses (such as digital logic and first programming), with the purpose of reinforcing the concepts in higher-level courses throughout the curriculum. To address this objective, "The Workshop on Connecting Concepts Across the Curriculum" was held on February 4–5, 2016 at the Cook Hotel and Conference Center, Louisiana State University, Baton Rouge.

Three broad questions were addressed during the workshop:

1. Which topics are both important and difficult for students to understand?

2. Can these topics be captured through broad "concept-threads" that can be woven through a course sequence? Can these threads serve to introduce and strengthen the comprehension and retention of these concepts?

3. Are there course/level-specific ways to introduce these concept-threads in a typical curriculum?

The "target" courses/areas that were considered are Digital Logic, Programming and Algorithms, Embedded Systems, Computer Organization and Architecture and Computer Networks.

The most important computing concepts (that students also find difficult to grasp) that were identified at the workshop include: (a) Abstraction, (b) the idea of "State," (c) Addressing, (d) Trade-Offs and Optimization, (e) the difference between Data and Control, (f) the Design Process (particularly Modularity), and (f) Asymptotic Thinking and Scalability. For each of these concepts and each of the target courses, a thread of topics was identified that could be used to unfold and reinforce the underlying concept.

For example, consider the idea of state. Early on, a state could be a flip-flop output in digital logic, or a variable value in a programming or algorithmic context. In computer organization or architecture, the contents of registers and flags may be the state. In general, the idea of a state that should be constructed through the different courses is that of information from the past needed to take the next step. This is more directly understandable in the context of checkpointing and state restoration (taught possibly at a higher-level course).

In addition to the development of concept threads, the workshop featured two invited talks on "Connecting Computing Education Threads in a Coherent Active Learning Environment," and "Introducing Computer Systems from a Programmer's Perspective."

Details of the workshop, including its findings, are available at www.ece.lsu.edu/vaidy/WCCC.

1 Introduction

There has been much work on improving teaching, typically at the level of a single course. At the other extreme, there have also been efforts towards developing comprehensive and rich curricula for various computing disciplines. There is a gap, however, in understanding how individual courses in a curriculum could interact to student benefit. Typically, this interaction is specified simply as a set of prerequisites. While the downstream course (for which an upstream course is a prerequisite) assumes knowledge of certain topics, the upstream course is quite oblivious to where its coverage could be subsequently used.

The purpose of this project is to address ways to thread the coverage of important topics, a little at a time, through a sequence of courses. The idea is to introduce key concepts early on, starting from "basic" (upstream) courses, with the purpose of reinforcing them in other "advanced" (downstream) courses throughout the curriculum. One could view the approach as that of introducing "conceptual dots" in the basic courses that will be "connected" in the advanced courses.

The proposed objective of this project was to use a freshman/sophomore-level digital logic course as the basic course in which to introduce (primarily parallel and distributed) computing concepts that may be leveraged in advanced courses (such as computer organization and architecture). It was anticipated that restricting the consideration to digital logic and parallel and distributed computing would lend focus to the proposed work. This was to be facilitated through a workshop to address ways to introduce key concepts early on and utilize this downstream in the curriculum.

2 The Workshop on Connecting Concepts Across the Curriculum

During the initial planning of the workshop we found that our approach for parallel and distributed computing (PDC) topics in digital logic readily extends to several other concepts in computing, in general, with the potential for gentle introduction in other basic courses. Consequently, we expanded the scope of the project to include first digital logic and first programming as the basic courses. An expanded set of computing concepts serves a wider repertoire of advanced courses, including advanced logic/hardware design, microprocessors and embedded systems, computer networks, computer organization/architecture and algorithms.

While the objective of introducing and reinforcing key concepts across multiple courses remained the same, this objective covered a broader swath of a typical Computer Engineering or Computer Science curriculum than originally envisaged. To address this objective, "The Workshop on Connecting Concepts Across the Curriculum" was held on February 4–5, 2016 at the Cook Hotel and Conference Center, Louisiana State University, Baton Rouge.

Three broad questions were addressed during the workshop:

1. Which topics are both important and difficult for students to understand?

2. Can these topics be captured through broad "concept-threads" that can be woven through a course sequence? Can these threads serve to introduce and strengthen the

comprehension and retention of these concepts?

3. Are there course/level-specific ways to introduce these concept-threads in a typical curriculum?

To increase the chances of adoption, the discussion was framed against the following considerations for introducing new ideas in a basic course.

- The introduction of a new idea should not appreciably interfere with the content or the instruction of the basic course.
- The introduction of the new idea should enhance and/or motivate the comprehension of the material in the basic course itself.

In the remainder of this section, we describe various details of the workshop, including its structure and activities conducted. The findings of the workshop are described progressively in Sections 2.5.1–2.6. In Section 3 we outline the follow-up and continuing activity resulting from the workshop. Finally in Section 4 we make some concluding remarks.

The Appendix contains the program and a table of possible relationships between computing topics and courses. Additional details on the workshop are available at www.ece.lsu.edu/vaidy/WCCC.

2.1 The Participants

The goals of the workshop (that include identifying specific topics, threads and methods) guided the process of selecting participants. These goals were viewed in the context of a set of courses, commonly seen in many computer engineering and computer science curricula (more details appear in Section 2.2). We recognized the following considerations in inviting participants.

- A set of thought leaders is needed to guide the discussion in the workshop. They include authors of textbooks and researchers in engineering and science education.
- A set of students is needed to provide insight that is often missed from a faculty perspective.
- The invited faculty should represent a large variety of institutions ranging from leading research institutions to those focused more on teaching, from local and regional institutions to universities across the nation.
- The expected outcomes of the workshop would be a set of recommendations for how one should introduce and connect "concept dots" across courses in the curriculum. To fully understand the scope of these recommendations, they must be adopted and tried at different levels. To this end, (a) the ideas must be tried out locally (at LSU) and (b) inviting multiple faculty from the same institution increases the chances of adoption.

Once the dates, location and target courses were decided on, we proceeded to contact departmental chairs of several universities, requesting names of faculty with interest in the workshop and who teach (or have taught) the courses to be considered at the workshop. Many of the invitees to the workshop were from those recommended by chairs of departments. Independently, we also invited students, thought leaders and LSU faculty to attend the workshop.

The workshop had 25 participants, excluding the PI and Co-PIs.

1. Stephen Brown, University of Toronto
2. Randal Bryant, Carnegie Mellon University
3. Konstantin Busch, Louisiana State University
4. Murad Chowdhury, Louisiana State University
5. Pradeep Chowriappa, Louisiana Tech University
6. Gabriel DeSouza, Louisiana State University
7. Coretta Douglas, Louisiana State University
8. Geoffrey Hermann, University of Illinois
9. David Kaeli, Northeastern University
10. Justin Kilpatrick, Louisiana State University
11. David Koppelman, Louisiana State University
12. Sukhamay Kundu, Louisiana State University
13. Thomas Lavastida, Louisiana State University

14. Michael Loui, Purdue University
15. Julius Marpaung, University of Houston
16. Lu Peng, Louisiana State University
17. Michael Pratt, University of Louisiana, Lafayette
18. Jagannathan Ramanujam, Louisiana State University
19. Mohammad A. Salam, Southern University
20. Violet Syrotiuk, Arizona State University
21. Grant Thomas, Louisiana State University
22. Warren Waggenspack, Louisiana State University
23. Chao Wang, Arizona State University
24. Bill Wischausen, Louisiana State University
25. Shizhong Yang, Southern University

The participants included 3 LSU students (juniors/seniors), 10 LSU faculty, 4 faculty from other institutions in Louisiana and 8 faculty from institutions across the USA and Canada. Among the LSU faculty members invited were two from broader STEM areas in which the approach discussed at the workshop could find potential application. One of these faculty members is from Mechanical Engineering and the other is from Biological Sciences; these faculty are also active in pedagogical research.

All participants were fully supported as needed (transportation, hotel stay, food) for the duration of the workshop. Participants were also offered help related to child-care services, if needed.

For added perspective, we also invited faculty from a Community College, but did not receive a response to our invitation. We were also unable to include participants from the NSF, IEEE TCPP and others in the networking area due to scheduling conflicts; another NSF workshop was held at the same time as ours.

2.2 Workshop Structure

The broad aim of the workshop was to determine a template for the instruction of important concepts in computing by distributing the instruction through the entire curriculum, rather than in just a small set of courses. Specifically, basic (lower-level) courses would aim to introduce these concepts, typically as simple extensions to, or applications of, what is already taught in the basic course. Advanced courses (downstream in the curriculum) would leverage the superficial introduction to a concept in basic courses. To allow for a non-disruptive adoption of the ideas generated in the workshop, we required that recommended changes to existing basic courses be incremental, and, where possible, these changes should benefit the basic course itself (in addition to the advanced course). To this end, the workshop set out to determine:

1. Concepts that are important and difficult for students to grasp.

2. Topic-threads from a set of courses that can be used collectively to reinforce these concepts.

3. Course-specific strategies to teach these concepts.

To lend focus to the discussion, we used the following broad streams of computing courses/areas:

- Hardware, including digital logic, design and optimization.
- Software, including programming and algorithms.
- Computer organization and hardware.
- Embedded systems and computer networks, including elements of communication.

A typical computer engineering or computer science curriculum covers these broad areas through a set of courses.

The Workshop participants were welcomed by Prof. Judy Wornat, Interim Dean, LSU College of Engineering. It was also attended by Prof. Jagannathan Ramanujam, Director, LSU Center for Computation and Technology.

Both days of the workshop featured talks by faculty members with vast experience in using innovative approaches in undergraduate education. Broadly speaking, the first evening allowed participants to meet each other and familiarize themselves with the activity for the second day. During the second day we took on the task of determining ways to connect important concepts across courses (see list above) in the curriculum. Appendix A shows the program of the workshop.

Sections 2.3–2.6 details the activities at various parts of the workshop, including activity conducted prior to the workshop. The workshop findings are also described progressively through these sections.

2.3 Pre-Workshop Activity

After the workshop participants were finalized, we sent an invitation to each participant to consider the list of courses/areas listed above and provide an initial set of 5-10 topics that are both important and difficult for students to grasp. Participants were asked to weigh in on as many courses/areas as they were comfortable with. They were also pointed to our book chapter[1] that provides an example of how digital logic ideas could be used to introduce broader concepts in computing.

The responses from the participants were pooled together with our own ideas to generate an initial set of topics and threads. Appendix B shows this mapping of topics to courses/areas, arranged alphabetically by topic. This was mailed in advance to the participants. They were also provided with the same list of topics, ordered by the number of courses that they could potentially impact.

[1] R. Vaidyanathan, J. L. Trahan, and S. Rai, "Introducing Parallel and Distributed Computing Concepts in Digital Logic," in *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st edition, eds. S. K. Prasad et al., Elsevier-Morgan Kaufman, 2015. grid.cs.gsu.edu/ tcpp/curriculum/?q=system/files/ch5.pdf.

2.4 Invited Talk and Keynote

The first day (evening) of the workshop featured an invited talk entitled "Connecting Computing Education Threads in a Coherent Active Learning Environment," by Prof. David Kaeli, Northeastern University. The talk described a hands-on introductory course with emphasis on embedded systems and robotics, cast in the setting of a recently redesigned curriculum in the Department of Electrical and Computer Engineering at Northeastern University.

The second day began with a Keynote talk "Introducing Computer Systems from a Programmer's Perspective," By Prof. Randal E. Bryant, Carnegie Mellon University. The talk described experiences from an "Introduction to Computing Systems" course designed to teach students to be sophisticated applications programmers and to prepare them for upper-level courses.

Collectively, both talks explored, among other ideas, the benefits of using lower level courses (particularly with hands-on activity) to prime students for deeper understanding of concepts in upper level courses. The slides of the talks appear at www.ece.lsu.edu/vaidy/WCCC.

2.5 At the Workshop

The workshop had three main sessions that addressed the three questions mentioned earlier, namely, (a) which topics are important and difficult for students to grasp, (b) how can one thread these important topics (as special cases of a broader concept) through a sequence of courses, (c) what methods can be used to implement these threads in a course?

The slides outlining the process to be followed at the workshop is available at www.ece.lsu.edu/vaidy/WCCC. We describe Sessions 1–3 below.

2.5.1 Session 1: Identifying Topics

The questions mentioned above were to be answered in the context of the courses/areas listed earlier. To lend focus to the discussion, we informally divided the participants, largely based on their research and teaching interests, into four "groups," each with seven participants.

1. Digital Logic
2. Programming and Algorithms
3. Computer Organization and Architecture
4. Embedded Systems and Networking.

Each participant was given his/her group at the start of the workshop. Each group had a "leader," a faculty member with experience with pedagogical issues and textbook authorship, to lead the discussion within the group. Each group also had a student-participant. The group leaders were cued in on the nature of the expected outcome of the discussion; the organizers (R. Vaidyanathan, J. L. Trahan and S. Rai) were also part of individual groups, lending both their own opinions and providing clarification on the process as needed.

The goal of Session 1 was to identify, independently within each group, a set of important topics/concepts that are difficult for students to grasp.

Groups recorded the main points of their discussion and at the end of the session summarized the topics that their group considered important enough to pursue further in the next session.

The following broad topics were identified within each group.

Digital Logic:

- Multiple forms of logic representation: truth table, Karnaugh map, Boolean expression
- Delay
- Hazards
- Data and control paths, control units and FSMs
- Tools: debugging, design and technology
- Engineering design: specs, design, debugging, verification, iterative process.

Programming and Algorithms:

- Abstraction: moving from concept to algorithm to program
- State and state transformation
- Problem decomposition (including divide-and-conquer)
- Performance: Throughput, latency, Amdahl's law, experimental design and evaluation
- Synchronization and locking
- Debugging
- Resource management: memory, network and other shared resources
- Trade-offs: speed, power
- Big "Oh" performance

Computer Organization and Architecture:

- Impact of logic delay on cycle time and pipeline speed
- Memory: design, decoding, pointers, addressing, data representation in memory
- Distinction between code and data
- Power: busing, interfacing and I/O

Embedded Systems and Networking:

- Protocol stacks
- Flow-control, handshaking
- Data representation, distinction between control and data
- Performance and trade-off: latency, throughput, data buffering

- Addressing: physical, logical
- Abstraction: client, thread

During the break, this input was distilled and provided to the participants before the start of the next session.

2.5.2 Session 2: Identifying Threads

For Session 2, participants were relocated among groups so that each table now had members from all other groups. The group leaders remained at their original table to provide a link between the discussion in Sessions 1 and 2.

The goal of Session 2 was to identify course-specific topics through which the concepts identified in Session 1 could be emphasized/taught. The findings of Session 1 helped focus the discussion to those concepts that the participants deemed most important, yet difficult for students to understand. At this point in the discussion, abstraction, state, data vs. control, design, and trade-offs bubbled up as concepts that most participants felt were important and difficult for students.

The following broad concepts were identified by four "tables" of participants, with each table working independently.

Table 1:

- Addressing: logical and physical, conventions, translation-cost, memory topology, range and precision
- State: Finite-state machine, state representation, transitions, events
- Abstraction: levels of abstraction, black-box design, interoperability, hierarchical design
- Data vs Control: interdependency, parallelism, learning, synchronous/asynchronous

Table 2:

- Abstraction: input-output relationship; truth-tables, state table/diagram, ASM charts; instruction set, machine code; pseudo-code, flow-chart; network protocols and layering
- Optimization and Trade-offs: Gate-register, delay; CISC, RISC; throughput, latency; algorithms, data structures
- Design Process: top-down vs bottom-up; control/data portion of design; tools for specification, implementation, debugging, verification

Table 3:

- Abstraction: reasoning about complex structures, deciding the correct level of abstraction
- State: view of system as a transformer of state, state decomposition

- Addressing, including indirection
- Asymptotic thinking: relation to scalability

Table 4:

- State
- Concurrency/parallelism
- Abstraction: data types, number representation
- Robust design: problem decomposition, trade-offs, usability, testability, readability, debugging

2.5.3 Session 3: Identifying Methods

Session 2 yielded, for each of the broad concepts identified in Session 1, possible course-specific topics that could could be used to used to teach the concept. The goal of Session 3 was to find strategies for weaving these concepts through specific courses. Considering that some of the targeted concepts may not be central to the course in question, it is important to identify possible ways to integrate the target concepts into the course without affecting the flow of the course. This was the goal of Session 3.

Once again, the groups were shuffled among the tables and group leaders provided the context across the various sessions.

Unlike those of other sessions, the objective of Session 3 was much closer to a course offering than the broad concepts pursued earlier. Understandably, this session did not yield as many specific methods as we had hoped for. Nevertheless some interesting ideas emerged.

1. Knowledge of digital logic (whose aim is to produce a circuit) can be used to strengthen the understanding of mathematical logic (for reasoning).

2. A common mistake made by students in converting a binary sequence to octal or hexadecimal is to collect bits in 3's or 4's starting from the left (rather than from the lsb). This mistake can be used to emphasize the idea of framing, without which a computer can make a similar mistake.

3. A priority queue can illustrate trade-offs ranging from a linear array to a heap.

4. Implementations of network layers (for example, UDP vs TCP) can illustrate performance trade-offs and abstraction.

5. Race conditions can be illustrated in many courses ranging from digital logic (in a circuit with reconvergent paths, and metastability in latches), to operating systems (shared variables) and computer networks (multiaccess).

6. The idea of state (as a collection of information about the past needed to take the next step) can be illustrated in many contexts from FSMs and flow-control (sliding-window) protocols to simply the set of variable values in a program.

2.6 Workshop Findings

As noted in Section 2.2, the workshop findings were developed in three stages (Sessions) that addressed the following three questions.

1. In the targeted courses, what are the most important and difficult to grasp topics for students?

2. What broad concept that is threaded though a sequence of courses addresses these topics?

3. What course-specific strategies can be used to thread concepts through courses?

At the end of Session 3 the following topics/concepts emerged as some of the most important and deserving attention: (a) Abstraction, (b) State, (c) Addressing, (d) Trade-offs and Optimization, (e) Data vs. Control, (f) Design Process, and (g) Asymptotic Thinking and Scalability. We organize the ideas below by these concepts. For each of these concepts, topics are organized by (sets of) courses. Some of the topics clearly straddle multiple courses; these have also been pointed out. Depending on the level of the course at which the concept is explored, one could look to either set up the concept in a lower-level course for further development in a higher-level course or use the introduction in a lower-level course to jump-start the instruction at the higher level. The following, together with the table in Appendix B, provide a concrete method to connect the important concepts identified across multiple courses.

Abstraction: Abstract thinking is essential for problem-solving and modeling. Traditionally, students learn this important skill as a second-order activity, in the sense that very few courses directly talk about abstraction (even though they present plenty of opportunities to do so).

Digital Logic: A digital logic course begins with the abstract idea of 0 and 1 as truth values and the symbol "+" to mean the logical OR, leading to (correct) statements such as $1 + 1 = 1$; this not surprising when viewed within the right context and at the correct level of abstraction. An HDL "module" is a black box appearing to the outside only as an abstract relationship between the input and output. Digital logic also lends itself to illustrating functional and temporal abstraction through timing diagrams.

Programming and Algorithms: The process of program development takes the student through several abstract layers, starting from the algorithm (even if simple, as in a starting course) through a flow-chart and/or a pseudo-code, to the program. (An ASM chart provides a similar abstraction as a flowchart for synchronous sequential circuits). As in hardware modules, functions and procedures define black-boxes at the software level. Programming languages themselves illustrate several levels of abstraction as we move down from pseudo-code and HLLs to assembly and machine code.

Computer Organization and Architecture: The ISA (instruction set architecture) provides an abstract view of a processor's functional elements (operations, addressing modes etc.). The memory hierarchy (including virtual memory and caching) provides an abstract view of the memory space.

Embedded Systems and Networking: Layered protocol stacks are an excellent illustration of abstraction. The distinction between a sequence of bits and what it (abstractly) represents can be explored through several courses, including computer networks (packet headers and error correction code), computer organization and architecture (instruction/data, number representations) and digital logic (state assignment). Real-time control, cyber-physical systems and event-based systems present opportunities to illustrate abstraction, albeit at a more advanced level.

Finally, courses in a curriculum can themselves be viewed as an illustration of abstraction. For example, digital logic abstracts away from details of gate implementation, whereas computer organization and architecture remain at the level of registers and ALUs. Further down, a computer network is described in terms of nodes, and an algorithm typically abstracts away from many physical details of a computer.

State: While many courses use the idea of a state, students tend to view these ideas in isolation, without grasping the general concept. In the following, opportunities for reinforcing the concept of a state (as a record of what is needed/used from the past to take the next step) are identified.

Digital Logic: In constructing a state diagram, a state is indeed viewed as a record of relevant information from the past needed for the state machine to transition to the next state. This is sometimes lost on students viewing states simply as flip-flop outputs.

Programming and Algorithms: A variable's value is a record of a computation that will be consumed at a later time. A handy example is the use of a temporary storage to hold the value of one of two variables whose values are to be swapped. The difference between functional and imperative programming can also reinforce the concept of state.

Computer Organization and Architecture: Processor flags are an automatic record of features of preceding computations that are used often to decide on the next step. At a slightly higher level, a CPU can also be viewed as an FSM. Cache coherence and read/write hazards also provide opportunities to reinforce the idea of state.

Embedded Systems and Networking: Checkpointing and state restoration are important aspects of embedded (and distributed systems). States are also inherent in most protocols. The state diagram of a simple handshaking protocol is not very different from an FSM implementing the same idea across hardware module; that is, the concept of a state can be decoupled from topic-specific artifacts, such as flip-flops and program variables. At another extreme, distributed systems (such as robot swarms) whose nodes have limited state can illustrate the advantage of oblivious systems in self-stabilization.

Addressing: Addresses are used in various contexts, whose manifestation ranges from a bit sequence (for example, internet addresses, memory address) to a more abstract quantity (URL, MAC addresses, or even a variable name or pointer in a program). Here the address encompasses the idea of uniquely identifying an object (memory location, internet node etc.). Associated with this broad concept are other ideas (such as the distinction between

address and data, indirection, logical/physical address etc.) that, when tied together, can help convey the idea of addressing in a richer way. In the following, we list some of the topics that were identified as possible links to the "address" concept in various courses.

Digital Logic: A one-hot decoder is used to convert a binary address to a unary "enabling signal" (as used in most hardware). This also relates to address width bounding the largest number of uniquely identifiable entities.

Programming and Algorithms: Pointers (and associated data structures) provide a good way to distinguish data from address. Address related computations, such as in virtual memory/caching, hash tables, and routing table look-up, can be used to reinforce the concept of address.

Computer Organization and Architecture: The distinction between address and data is touched upon in various contexts in assembly programming. Indirection and address computation, together with logical and physical addresses demonstrate trade-offs between access and address space cost/size.

Embedded Systems and Networking: URL and MAC address are two examples that illustrate addresses in a more abstract sense.

Trade-offs and Optimization: This is a central topic of Engineering and Computer Science and is typically explored in several courses in the curriculum. Nevertheless, the participants felt strongly enough about its importance to include in this list. While trade-offs and optimizations occurs in different contexts in different courses, on the whole they illustrate the common idea of competing forces (trade-offs) and fine-tuning (optimization). In the following, we list some of the course-specific ideas that were brought up.

Digital Logic: Karnaugh maps illustrate optimization (in terms of gates for a two level circuit); one could however add that this optimization may fail for multiple-output circuits. In a more advanced course, a trade-off between gate cost and resilience to stuck-at faults can be illustrated by including additional implicants. In addition to gate/register count and delay (that are typically studied), wiring cost and power can be considered as well.

Programming and Algorithms: Opportunities abound in this class of courses, ranging from time/space trade-offs and the impact of data structures (for example, array vs linked lists) to algorithm design itself (for example, sorting and searching techniques, the benefit of pre-processing).

Computer Organization, Architecture and Embedded Systems: The memory hierarchy, ISA, interconnect density, and elements of system granularity (number/size of processing elements, cache line size) illustrate trade-offs.

Networking: Delay, throughput and latency are good vehicles to emphasize trade-off and optimization. Other performance measures (such as QoS) can be illustrated through protocols (such as UDP and TCP).

Data vs Control: Data and control-information manifest in different ways at various levels, some less obvious than others. Understanding the distinction between them can further comprehension of the underlying topic itself.

Digital Logic: Most digital systems have separate "control" and "data" components in which the control unit could be an FSM of fixed size that controls a data unit whose size (data-width) can be adjusted (independently of the control unit).

Programming and Algorithms: An analogous idea in programming is the idea the program-size (control) being independent of the problem size (data).

Computer Organization, Architecture and Embedded Systems: The order of instructions (control) executed over time can affect, for example, prefetching. On the other hand, the order of data accessed affects, for example, cache design. The separation of program and data (separate address spaces) is also illustrative. In general, the width of control and data words is independent.

Networking: The distinction between the payload (data to be conveyed) and header/trailer bits (that convey information about the data—error detection or correction, data delivery address etc.) captures the distinction between data and control information. Further, encapsulation that causes control information at one level to become data at the next, illustrates abstraction.

The Design Process: The aspect of the design process that was most emphasized during the workshop was modular design. Modular thinking also facilitates other directions, including abstraction, trade-offs, testing and validation.

Digital Logic, Embedded Systems, Computer Organization and Architecture: Verilog modules directly reflect modular design ideas. Logic decomposition (possibly into "control" and "data" parts) also supports modular thinking.

Programming and Algorithms: Like Verilog modules, functions and classes can be used for modular design. The scope of variables can add to the understanding of Modularity.

Networking: The design of various protocol elements and services across layers supports modular thinking.

On the whole many course sequences support modular thinking in an abstract sense; for example, digital logic deals with gates and flip-flops, whereas ideas in computer organization could be expressed in terms of registers.

Asymptotic Thinking: The ability to consider the effect of an increase in problem size is important. This impacts scalability and intractability of a general solution (that may be missed by students focused solely on local optimization).

Digital Logic: Most ideas in digital logic (truth tables, Karnaugh maps, state assignment) increase exponentially with the number of variables and offer simple ways to expose students to computational complexity.

Programming and Algorithms: Programming exercises for intractable problems (for example, Towers of Hanoi) quickly illustrate the impact of problem size increase. Most algorithm courses directly encourage asymptotic thinking.

Embedded Systems, Computer Organization and Architecture: Cache coherence, scheduling and assignment problems become significantly more difficult in a multicore environment.

Networking: In a different, but related, way, network design must allow for system expansion.

3 Follow-Up Activity

The following developments have occurred in the Division of Electrical and Computer Engineering at LSU:

Connecting Concepts: The computer group within the division met and discussed with instructors of courses on microprocessors, computer organization and architecture and singled out a specific idea that many students seem to have difficulty with: distinguishing a binary string from what it stands for (a facet of abstraction). This occurs, for example, in the context of number representations, instructions and data (in machine language). In the Fall 2016 offering of the first digital logic course, students were specifically cued into this concept, whenever the opportunity occurred. Students self reported (in increasing numbers over the course) that they understood the the difference. A final exam question indicated that over 54% of students understood the difference. It remains to be seem whether this understanding results in better comprehension and retention of concepts in downstream courses.

Digital Logic and Lab: Two ideas emphasized by both talks during the workshop were the benefits of a breadth first coverage in a beginning courses and the value of hands on experience. We have used this in a redesign of our consecutive Digital Logic and Lab sequence (where the classroom course is currently a prerequisite to the lab) into one that includes a lab experience with both courses in a two-course sequence. This uses a breadth first coverage in that the first course aims to teach students only the core fundamentals, leaving advanced (relatively speaking) material to the second course.

4 Concluding Remarks

The last session of the workshop summarized the findings and also provided an opportunity for the participants to provide feedback. On the whole the workshop was very well received. Some of the observations and suggestions we received included the following:

- The idea of concept mapping to various courses was considered a useful exercise.
- A sample syllabus of a course could be constructed to include the ideas discussed.

- The process followed in the workshop (Sessions 1–3) could be better fleshed out to make the objectives clearer.

We too recognized the importance of the last point in repeating the exercise for other STEM disciplines. Invariably each discipline has its own concept threads that hold the potential for reinforcement through the curriculum.

In our own judgment, the first two sessions yielded the results we expected for topics and threads. Session 3, however, was not as productive as we had hoped for. While specific topics (to emphasize key concepts) were identified within courses, the level of detail on how to touch on these topics, without disrupting the lower level course, was lower than we had hoped for.

On the whole, we believe that the workshop met its goals. In the coming years we expect to implement some the findings in our own curriculum.

A Workshop Program

The program for the workshop appears on the next page.

Workshop on Connecting Concepts Across the Curriculum

Parallel and Distributed Computing: From Digital Logic to Computer Architecture and Algorithms

The Cook Hotel and Conference Center, 3848 West Lakeshore Drive, Baton Rouge, LA 70808

Thursday February 4	5:00-5:30 pm	Registration
	5:30-6:00 pm	**Introduction and Welcome**
	6:00-7:00 pm	**Invited Talk** Connecting Computing Education Threads in a Coherent Active Learning Environment *Prof. David Kaeli, Northeastern University*
	7:00-7:15 pm	An outline of the Friday agenda
	7:15 PM	Dinner
Friday February 5	7:30-8:00 am	Registration
	8:00-8:45 am	**Keynote** Introducing Computer Systems from a Programmer's Perspective *Prof. Randy Bryant, Carnegie Mellon University*
	8:45-10:15 am	**Session 1** *Determine important and difficult to understand **topics** in each stream*
	10:15-10:45 am	Break
	10:45 am-12:45 pm	**Session 2** *Identify **threads** of important topics through course sequences*
	12:45-2:00 pm	Lunch
	2:00-3:30 pm	**Session 3** *Identify **methods** to implement threads across courses*
	3:30-4:00 pm	Break
	4:00-5:15 pm	**Session 4 and Closing** *Record workshop findings; Feedback*

LSU Center for Computation & Technology
Interdisciplinary | Innovative | Inventive

NSF

IEEE computer society TCPP Technical Committee on Parallel Processing

B Topic List and Possible Threads

Participants' input was distilled into the following list of topics, each with possible threads spanning the courses/areas considered in the workshop. The table in the next two pages is arranged alphabetically by topic. Participants were also provided with the same list with topics arranged by the number of courses the topic could be associated with.

Topic (alphabetical)	First digital logic + lab	Digital logic (advanced)	First programming	Microprocessors	Embedded Systems	Networking	Organization & Architecture	Algorithms	General
abstract thinking and modeling	0,1 can mean yes/no, enable/disable, true/false, numbers ….; 1+1=1?		algorithm-flowchart-code	binary string as instruction has a different meaning from same binary string as data	programming models; relation to underlying hardware	levels of protocol stack; encapsulation	ISA abstraction	abstract models (including programming models), proofs, analysis	Abstraction; "information = bits+context"
addressing modes	see cycles/instr		pointer use	main exploration of topic	content addressability		address vs data		
alternate representations/conventions (for convenience/advantage)	Truth table = K-Map; state table = state diagram		different data structures for same data (ex: incidence matrix, edge list for graphs), pre, post and infix	different representations of same data (binary-hex, unsigned, BCD)			Ex: left/right endian	different data structures. Modeling tools	
Amdahl's Law	carry propagation as a bottleneck in ripple carry adder	critical paths						parallel algorithms	bottlenecks
Arithmetic	arithmetic circuits (adders and perhaps multiplier, divider). Number representations			possible efficiencies (stemming from proper selection of operation/instruction), overflow			ALUs, possible efficiencies, stemming from proper selection of operation/instruction; Strassen's matrix multiplication		
assembly vis-a-vis hardware	Relationship between hardware, assembly/machine code and high level code			programming models; relation to underlying hardware			programming models; exploiting efficiencies		
Complexity and Analysis	Digital logic can provide a first hand look at complexity. It is intimately tied to circuit complexity and is replete with intractable problems. For example, state assignment could easily illustrate the futility of an exhaustive approach		complexity of algorithms, program efficiency		CAD tool use can provide a window to appreciating the importance of complexity in practice	Can provide a setting for probilistic and amortized analyses		Complexity theory	
Control structures (conditional, loop)			introduction at HLL	implementation in assembly			implementation in assembly		
delay	circuit delay	area/time optimization; retiming, clock skew	program speed	instruction delay	Ex: FPGA clocking rate	latency	instruction delay,pipeline delay, access time	algorithms as a general recipe, implementable in hardware or software	see performance measures
distinction between control and data	Example in MUXs, ALU etc. (data width is independent of control width). Complex circuits with control FSM enabling the operation of "data" part		control instructions		control and data hazards		indepencence in width of control and data paths		
floating point add/mult, rounding	introduction to idea that n bits can only represent 2^n distinct values	implementing floating point hardware	introduction of float	Details of floating points/standards	precision, errors	standards	instructions	precision, errors	Standards
Graphs	state diagram as a graph, can relate binary numbers to graphs	place and route, connectivity, planarity	data structures (trees, linked lists)		DAGs, task graphs	routing and congestion control		Graph algorithms	see also modeling and abstraction
hazards	logic hazards, reconvergent paths, synchronization using registers, async gates in synchronous ckts,				synchronization, handshaking		data, structural and control hazards (branch prediction),out-of-order execution, race conditions, synchronization, deadlock, livelock		
idea of "state"	From flip-flop outputs to "what needs to be remembered"		need for memory in computation (example swapping values)	CPUs as FSM	Checkpointing and state restoration; FPGA pattern matching constructs a state machine for the patterm (such as in the KMP algorithm)		Checkpointing and state restoration (particularly in distributed environment); oblivious (limited-state) computation		
instrction (cycles per instr, format,encoding)	can be introduced with circuit delay in an advanced example (that implements an instruction or an address computation)		instruction choice (example shift or multiply)	cycles/instr, instruction choice (example multiply by a constant), format	constant coefficient multipliers (via look-up tables)		instruction choice, ISA, RISC/CISC	see arithmetic	
interrupts/exceptions	enable, asynchronous inputs; event triggered Verilog execution			interrupts and exceptions, priorities				event-driven processes	
low-level paralleism	bitwise boolean operations as opposed to scans and global operations			instruction implementations	Low-level optimizations (for example in FPGAs)		ILP, supersacalar		

Topic (alphabetical)	First digital logic + lab	Digital logic (advanced)	First programming	Microprocessors	Embedded Systems	Networking	Organization & Architecture	Algorithms	General
memory	memory as an abstract idea to explain state	memory as a circuit	memory as a variable	memory as registers and RAM, different types of memory			memory hierarchy and management; memory architecture (shared, distributed); performance	memory as an algorithmic abstraction (online, oblivious)	
Models/Structures of interaction/communication	The idea of interconnects and topologies can be introduced in the first digital logic	bus (and other stucture) implementation	parameter passing, shared variables	data and control buses, interconnects and interfaces		client-server, peer-to-peer, routing, flow control, protocols and services between protocol layers	shared/distributed memory, message passing, routing,interconnection networks, topologies		also see abstract thinking
Modular design	Verilog modules, circuit decomposition		procedures, scope of variables	bit-slice	Verilog modules	layers in protocol stack, encapsulation			See also abstraction
	course sequnce itself may be viewed as an exercise in modular exposition of concepts, for example, electronics --> digital logiv --> microprocessors --> embedded systems --> computer organization --> Computer architecture								
parallel vs. sequential	First look at parallel. Circuits are inherently parallel. Many possibilities, serial to parallel conversion, carry look-ahead, barrel shifter, Verilog blocking and unblocking assignments etc.		First look at sequential		multiple pieces running sequntial code independently	Sequential/Parallel at different layers (Transport layer may deliver packets sequentially (in order), network carries packets in parallel paths/links. (see also abstraction)	Multicore environment …	parallel, sequential, distributed algorithms, concurrency	
performance measures	delay, number of gates/flip-flops	clock-speed, area, power	time (asymptotic)	processor cycles, memory	speed, power, memory, cost/form-factor	latency, throughput, quality or service	speed, power, fault-tolerance	time, space, communication cost, approximation ratio	
pipelining concept	series of flip-flops (example shift register) as an analog of progess through an ideal pipeline	pipeline implementation			Ex: image processing pipeline	pipelining packets across (frame-level) links; sliding window protocol	instruction pipeline	sofware/algorithmic pipelining	
procedures, parameter passing, scope of variables, run-time stack (see recursion also)			procedures, stacks	details of parameter passing, runtime stack		Remote Procedure Call (RPC)			see also Modular design
recursion (see procedures also)	Decompsition of MUXes, decoders etc. Carry lookahead as a recurrence	recursive harware blocks (example bitonic sorter), prefix circuits (ex: Kogge-Stone)	recursion, recursive structures (such as trees)				Recursive structures	Divide and Conquer	Recurrences
shared resources	MUX to share output among several inputs			Shared buses, hardware modules, hardware reuse (FPGA)		MAC layer, multiaccess	shared resources (memory, channels, processing elements), deadlock, semaphore, critical region		Multiplexing in other dimensions (besides time), for example frequency/wavelength
speedup	obliquely through "parallel" examples. How much is delay reduced by increasing H/W cost	area/time optimization						parallel algorithms	see tradeoffs, performance measures
Synchronization	latch, flip-flop, circuit timing				handshaking between modules	flow control, handshaking (protocol level), traffic shaping (example leaky bucket)	interfacing between modules (example CPU-RAM)	distributed systems, I/O streams and buffering, concurrency	
throughput	"throughput," for example in series parallel conversion					network throughput			see performance measures
Tools	physical device vs CAD tools		compiler, debugger	simulating, debugging, verification, testing (in different contexts)					
Trade-off	delay-area		speed-space (array-linked list)		speed-area(cost)-power		performance tradeoffs in memory hierarchy, interconnect density, scalability	space-time-communication complexity	
translating informal specs. (see also abstract thinking and modeling)	verbal/informal description to state diagram/architecure		verbal description to algorithm/flow-chart					Problem to algorithm	
verification and testing	HDL testbench, verification flow		proof of simple programs, loop invariants, induction (see also recuirsion)		HDL testbench, verification flow	protocol verification (safety, liveness)	Architecture/Software verification/testing; algorithm correctness		