A Free Online Textbook Introducing Computer Architecture Topics

Tia Newhall
Swarthmore College
Swarthmore, PA, USA
newhall@cs.swarthmore.edu

Suzanne J. Matthews
US Military Academy
West Point, NY, USA
suzanne.matthews@westpoint.edu

Kevin C. Webb
Swarthmore College
Swarthmore, PA, USA
kwebb@cs.swarthmore.edu

ABSTRACT

This paper describes the computer architecture content in *Dive into Systems*, our free, online textbook that introduces a broad range of computer systems topics. *Dive into Systems* assumes only a CS1 background of the reader, and includes numerous examples and illustrations to foster a reader's understanding of its content. Our textbook is designed to be used as a primary textbook for a range of courses that introduce computer systems and computer architecture topics. It also serves as a supplementary text in upper-level undergraduate and graduate level courses to provide background material on computer architecture, systems, and parallel computing. In addition to presenting the details about our book's coverage of computer architecture topics, we also discuss the overarching themes of our textbook and our motivations for writing a free online textbook to introduce computer systems topics. Our book is currently used by more than 45 institutions in a wide range of courses, including undergraduate computer architecture courses.

CCS CONCEPTS

• **Applied computing** → **Education**; *E-learning*; • **Computer systems organization** → **Architectures**.

KEYWORDS

computer architecture, textbook, CS education

ACM Reference Format:
Tia Newhall, Suzanne J. Matthews, and Kevin C. Webb. 2023. A Free Online Textbook Introducing Computer Architecture Topics. In *WCAE '23: Workshop on Computer Architecture Education, June 17, 2023, Orlando, FL.* ACM, New York, NY, USA, 8 pages. https://doi.org/X

1 INTRODUCTION

The expense of modern college textbooks is often a barrier to student learning. According to the U.S. Bureau of Labor Statistics, the average cost of a college textbook in 2023 is $107.30 [26]. A recent survey of college students [15] indicated that nearly two-thirds of students skipped buying or renting an assigned textbook or access code (or both) due to the prohibitive costs. Another survey found that approximately one in four students opt not to purchase a required course material, and that students who skipped materials were more likely to consider dropping out [16].

College faculty increasingly recognize that the costs of materials represent a serious barrier to their students [22], and they have accordingly begun adopting free materials [16]. These free materials include open access resources and large, digital subscriptions that are paid by the institution ("Inclusive Access programs") so students do not have to directly pay the cost. In one survey, 73% of students reported that their instructors assigned at least one material that was free (or not paid for by students) [16]. Faculty are also more inclined to assign digital resources in the wake of the COVID-19 pandemic, which represented a seismic shift in the way professors taught courses. As of this writing, fully in-person instruction has not returned to pre-pandemic levels [22], with many faculty still teaching in online-only or blended in-person/online modalities.

Thus, as computational thinking and programming increasingly become desired skills, it is imperative that computer science faculty have access to high quality, online, open-access resources to teach their students. However, much of the effort in producing free online materials for teaching computer science concentrates on introductory programming courses.

In this paper, we describe the content in *Dive into Systems* that is relevant to modern computer architecture courses and how it aligns to common computer architecture curricula standards, such as CS2023 [2] and TCPP2020 [24]. We also reflect on our lessons learned and future directions of the project. We hope that our experience will inspire others to create open access resources for teaching computer architecture concepts, thereby reducing a critical barrier toward making computing resources accessible to all.

2 *DIVE INTO SYSTEMS* DEVELOPMENT

In 2018, we began the *Dive into Systems* project [12] in an effort to create a free, high-quality, online textbook that covers introductory computer systems topics (including several chapters that are extremely relevant to students taking introductory computer architecture courses). We presently know of at least 45 institutions that currently are using our textbook in their courses. Most are using it in intermediate-level courses like computer organization or computer systems. However, several are also using it in upper-level courses, including six who are using *Dive into Systems* in their computer architecture courses.

Throughout the development of *Dive into Systems*, we have sought community help in evaluating its content and its use to help us meet our goal of making it a useful and broadly applicable resource. In the initial writing, we had multiple external editors provide feedback on the content and presentation of each chapter.

WCAE '23, 17 June, 2023, Orlando, FL

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/X

These editors were volunteers, primarily faculty at other institutions, who we recruited via our own professional networks and by organizing ad hoc meetings at SIGCSE conferences. People were eager to help, primarily because our book filled a coverage gap for a broad range of systems topics at the introductory level. Many of the reviewers were interested to adopt it in their own courses, but they also cited that volunteering naturally matched the spirit of a free online textbook. We ultimately received invaluable feedback from reviewers, and this process greatly strengthened the book.

Once we had a beta version of the textbook completed, we ran an early adopters program in the 2019-20 academic year. Adopters were faculty using our textbook as a required textbook in their courses who agreed to give a survey about the book to students in their course and to take a similar instructor survey as part of participation in this program. We received a SIGCSE Special Projects grant to give faculty a very small stipend for participating. We recruited participants through our book's mailing list and through SIGCSE and other mailings lists. Again, the feedback we received helped us to strengthen the book's content. Details of the early adopter program are summarized in [13].

After releasing the online version, we recognized that a small but vocal number of students prefer non-digital materials [13, 22]. We have since contracted with No Starch Press to produce a low-cost print edition of the book [14] for the students who prefer it. Reflecting on our decision to produce a print edition of the book, it was important that we contracted with a reputable publisher that committed to produce a low-cost print edition of *Dive into Systems*. We also benefited from the copy-editing process of a computing publisher, their printing and distribution process, and offloaded challenges like preventing unauthorized translations and violations of copyright. Additionally, having a book published with a well-known publisher increases the esteem of the work in the eyes of institutional tenure and promotion boards, which are accustomed to the more traditional publishing models for textbooks.

3 CONTENT OVERVIEW

Dive into Systems introduces a broad range of computer organization, architecture, systems, and parallel computing topics. It assumes only a CS1 background of the reader and is designed to be a primary textbook for a range of intermediate-level undergraduate courses that first introduce these topics. Thus, the set of topics we cover, and the depth in which we cover them, is determined by *Dive into Systems*'s target use being as a primary textbook that first introduces these topics in undergraduate introductory systems courses. It also can serve as an auxiliary textbook for providing background material in a range of topics for upper-level undergraduate or even graduate-level courses which may use other primary textbooks such as those by Hennesy and Patterson [8, 18].

Because we want our textbook to be applicable to such a broad range of courses and uses, we strive to limit chapter dependencies as much as possible so that an instructor can mix and match content to best fit the needs of their particular course. The book's topic coverage is guided by three main themes that unify the topics and inform the topic coverage. These three themes are:

(1) *How a computer system is designed to run programs.* Our book takes the reader through a vertical slice through a computer system, explaining how a program expressed in a high-level programming language is executed by the low-level circuitry of the computer hardware, from translation to assembly, binary representation, CPU and system architecture design, and OS abstractions for running programs.
(2) *How to evaluate systems costs associated with a program's performance.* We focus on the memory hierarchy and CPU caching, but also evaluate performance in the context of OS abstractions, systems overheads, and parallel computing.
(3) *How to leverage the power of parallel computers.* We focus our introduction to parallelism on multi-core architectures and on shared memory parallelism, including detailed coverage of programming with pthreads. We also include coverage of other parallel architectures and programming models.

Table 1: CS2023 Coverage

Knowledge Unit	Book Chapters
AR/Digital Logic & Digital Systems	5
AR/Machine-Level Data Representation	4
AR/Assembly-Level Machine Org	2, 5, 7-9, 13, 14, 15
AR/Memory Hierarchy	11, 13
AR/Interfacing & Communication	11, 13
AR/Functional Organization	5
AR/Heterogeneous Architectures	11, 14, 15
OS/Memory Management	11, 13
PDC/Programs and Execution	14
SEP/History	5

The primary coverage of computer architecture topics are in Chapter 5 on computer architecture and in Chapter 11 on the memory hierarchy and CPU caching. We additionally cover some architecture and related topics in Chapter 4 on binary representation, Chapter 13 on operating systems, Chapter 14 on shared memory parallelism, and Chapter 15 on other types of parallel systems.

Our book's coverage of computer architecture topics map to recommendations made by several curricula published by CS professional organizations. For example, the ACM/IEEE-CS Joint Task Force on Computer Science Curricula recent 2023 report (CS2023) [2] includes an Architecture and Organization (AR) knowledge area with increased emphasis on parallelism. CS2023 separates their topics into two key categories: CS core, the set of topics that *every* computer science graduate must know; and KA core, the set of topics that must be covered if the knowledge area is covered. In this context, the AR knowledge area contains 9 hours of CS core topics and 16 hours of AR KA core topics. CS2023 also suggests topics from related knowledge areas to be included in an introductory computer architecture course. Table 1 illustrates that *Dive into Systems* covers 78% of the AR knowledge area in CS2023, including all knowledge units suggested for inclusion in an introductory architecture course. *Dive into Systems* covers many other topics that CS2023 identifies as valuable for an introductory architecture course, including topics from the OS, PDC, and SEP knowledge areas.

The increased emphasis on parallel computing topics is partially driven by a new requirement [1] by the Accreditation Board for Engineering and Technology (ABET) that all undergraduate students enrolled in accredited CS programs are exposed to PDC. The

Table 2: 2020 NSF IEEE/TCPP Computer Architecture Topics

TCPP Area (sub area): specific topics	Book Chapters
Pervasive: concurrency, dependency, locality, asynchrony, performance	Ch. 5, 11, 13, 14
Arch (Underlying Mechanisms): caching, atomicity, consistency, coherence, false sharing, interrupts, process ID	Ch. 11, 13
Arch (Classes of Parallelism): Flynn's taxonomy, ILP, SIMD/Vector, MIMD, multi-threading, heterogeneous, GPU, pipelines, data / control hazards, buses, snooping, shared memory, multi-core, multi-threading	Ch. 5, 11, 13, 14, 15
Arch (Performance, Scaling, & Others): bandwidth, cost of data movement across memory hierarchies, floating point repr	Ch. 4, 11, 13, 14

Table 3: CSTA Coverage

Standard	Book Locations
Computing Systems(CSys)/Describe how internal & external computing devices function to form a system.	Ch.5, Ch. 11
Computing Systems/Model how computer hardware and software work together as a system to accomplish tasks.	Ch. 5
Data and Analysis/Represent data using multiple encoding schemes.	Ch. 4
CSys/Use appropriate terminology in identifying and describing the function of common physical components of computing systems (hardware).	Ch. 0, Ch. 5, Ch. 11
CSys/Compare levels of abstraction and interactions between application SW, system SW, and hardware layers.	Ch. 4, Ch. 5, Ch. 13
Data Analysis (DA)/Evaluate tradeoffs in how data are organized & stored.	Ch. 11
DA/Translate between different bit representations of real-world phenomena, like characters, numbers, images.	Ch. 4
CSys/Categorize the roles of operating system software.	Ch. 13

CS2023 PDC knowledge area is influenced by the 2012 draft of the NSF IEEE/TCPP Curriculum on Parallel and Distributed Computing [23], which identified over a hundred PDC topics that can be covered in an undergraduate computing program (including Architecture as one of its four main areas). *Dive into Systems* covers many of the topics in the Architecture area of their curriculum, as well several topics in their Pervasive PDC Concepts area. Table 2 lists a summary of *Dive into Systems*'s coverage of topics in TCPP's 2020 revision of their curriculum (TCPP2020) [24]. Again, our textbook is an introduction to these topics, so our coverage may not be sufficiently deep to completely satisfy their recommendations for coverage of all of these listed topics, but many are.

Lastly, we believe that *Dive into Systems* can also be used to teach students core hardware and architecture concepts in middle and high school. The Computer Science Teachers Association (CSTA) maintains a list of key learning objectives for teaching K-12 computer science. *Dive into Systems* covers several learning outcomes listed in the CSTA K-12 Standards related to computer architecture topics (e.g., the devices, hardware and storage subconcepts); a mapping is shown in Table 3. As a result, we believe our book has the potential to be a very useful resource for teaching beginning architecture concepts to a broad population of students.

4 MAIN ARCHITECTURE CONTENT

The main chapter on computer architecture (Chapter 5) is presented in the context of addressing the first main theme of our textbook: how a computer runs a program. Specifically, we focus on how a single core CPU architecture is designed to execute binary program instructions on binary data. We begin our coverage with some definitions including the CPU, Instruction Set Architecture, microarchitecture, and some discussion of RISC and CISC instruction set architectures (ISAs). We then provide a brief overview of the history of modern architectures and introduce the Von Neumann architecture to frame much of our later presentation of modern CPU design. We then proceed to build a simple single core CPU starting from basic logic gates. It includes an explanation of the clock driven execution of an instructions through four execution stages, showing how the bits of the instruction and data are used in different stages of the architecture to execute the instruction.

We additionally touch on topics related to our second main theme of systems costs and efficiency, specifically via CPU caching, the memory hierarchy, pipelining, and parallel architectures including multi-core, multi-threading, IPC, and other features of modern CPUs. We also introduce parallel system designs beyond shared memory, including accelerators (primarily GPU architectures).

4.1 History of Modern Architectures

We provide an overview of the history of computers, starting with a brief discussion of pre-modern computer history, including work by Charles Babbage and Ada Lovelace. We then focus on modern computer history that led to today's general-purpose, stored program model of a computer. We discuss the origins that lead to Alan Turing's Logical Computing Machine work and his work defining the universal Turing machine. We discuss early electronic computers, including Colossus, ENIAC, and the Z3. And we conclude with the development of the Von Neumann Architecture. Throughout, we highlight contributions by women, particularly focusing on their important roles in developing programming and algorithms for these early machines. To the best of our knowledge, we are the only textbook that sheds light on this important (and often overlooked) contribution of women in computer history.

4.2 Von Neumann Architecture

We introduce the Von Neumann Architecture as the model of modern architectures. Our coverage begins with the five main functional units (processing, control, memory, input, and output) and their role in executing stored program instructions. We introduce buses

interconnecting the units and explain how they communicate program instructions, data, and control signals between the units. We then describe how the units collectively execute program instructions by describing the Fetch-Decode-Execute-Writeback cycle of program execution. We also introduce how two special registers, the program counter (PC) and the instruction register (IR), facilitate this process. Later in this chapter, when we discuss clock driven program execution and simple pipelining, we revisit these same four phases of execution on our simple CPU. In addition to providing important coverage of Von Neumann Architectures, by covering this topic early, we foreshadow the design of the simple CPU that we'll be building in the remainder of the chapter.

4.3 Basic CPU Architecture

The focus of our main chapter on computer architecture is understanding how a single core CPU is designed to execute program instructions. Throughout our presentation we reinforce the idea of building up layers of abstraction as we construct a full CPU from featureful component units that themselves abstract away details of their low-level implementations. For example, we show how an ALU is a building block of a CPU that is itself built from sub-circuits combined together to create the higher-level ALU functionality. We start our coverage at the lowest level, focusing on the process of building a simple 1-bit circuit from basic logic gates starting with its truth table. We also show how to extract a truth table from simple 1-bit circuit built from basic logic gates. The goal of this coverage is to illustrate how, at the lowest layers, simple circuits can be constructed from simple truth tables describing the circuit. Then once built, these circuits can be used as components for building higher functionality, which in turn can be used as building blocks for even higher functionality, and so on, ultimately creating a full CPU.

4.3.1 Logic Gates. This section marks the beginning of our main CPU architecture coverage. It serves as an introduction to combinatorial circuits, and it sets up a strong emphasis on abstraction throughout the remainder of the chapter. We begin by introducing logic gates and their corresponding truth tables. We use the gates AND, OR, and NOT as the set of basic building blocks of all circuits since readers have seen these logical operators in a CS1 courses and because this subset is easier to reason about (versus using NAND or NOR as the minimal subset). We include as an aside how logical gates are implemented by transistors. However, we do not cover the details of how each logic gate is implemented with transistors, but instead treat a logic gate as the smallest building block. After explaining 1-bit versions of gates we show how to build multi-bit versions by directing individual bits of a N-bit input through N 1-bit gates. In the context of describing minimal subsets of logic gates, we additionally introduce NAND, NOR, and XOR gates and their truth tables, showing how these gates can be created from AND, OR, and NOT gates.

4.3.2 Circuits. We begin our coverage of digital circuits by discussing the role they play in implementing an ISA. We start by detailing how to create simple circuits, introducing the notion of abstraction that treats the simple circuit as a building block unit with defined inputs and outputs, leaving its detailed implementation abstracted from its use. We organize our presentation in terms of three different types of circuit, which also mirror parts of the Von Neumann architecture: arithmetic and logic circuits; control circuits; and storage/memory circuits. Our coverage of circuits begins with simple combinatorial circuits built from basic logic gates. We defer introducing sequential circuits to our discussion of storage circuits, which comes later.

Arithmetic/Logic Circuits. We motivate this section by introducing the ALU part of the processor and describing how individual arithmetic/logic circuits make up the components of an ALU. We start with 1-bit versions of simple circuits built from AND, OR, and NOT gates, and we use these to build up more complicated functionality. Specifically, we discuss the following algorithm for creating a 1-bit circuit:

(1) Create a truth table for the circuit: determine the number of inputs and outputs, enumerate all possible input permutations, and determine the output for each one.
(2) Using the truth table, derive expressions for each row with a 1 output using combinations of AND, OR, NOT. At this step the resulting logical expressions may be simplified with the goal of reducing the number of gates, possibly by adding NAND, NOR, XOR gates in place of combinations of AND, OR, and NOT (we do not present formal methods for minimizing circuit design, but illustrate some examples of simplification).
(3) Translate the logical expressions into combinations of logic gates to create the resulting circuit.

We then demonstrate this algorithm by applying it to create a 1-bit equality circuit ($A == B$). We also discuss how single bit versions of NAND, NOR and XOR circuits could be similarly constructed from AND, OR, and NOT gates by applying this algorithm.

Next, we apply the same process to create a 1-bit adder circuit ($A+B$) with two outputs (sum, and C_{out}). Seeing this process applied to create an arithmetic circuit is often not obvious to students.

Finally, we demonstrate the construction of an N-bit ripple adder circuit, motivating the need for an additional input (C_{in}) to each 1-bit adder circuit building block. We describe building a 4-bit ripple carry adder circuit from 1-bit adder building blocks. This example illustrates how two 4-bit binary data input values, combined with C_{out} and C_{in}, propagate through a sequence of 1-bit adder circuits to produce the correct sum and C_{out} outputs for the multi-bit adder. We then abstract the 4-bit adder circuit with three inputs (4-bit A, 4-bit B, 1-bit C_{in}), and two outputs (4-bit Sum and 1-bit C_{out}). Finally, we show how this multi-bit adder circuit can be used as a building block in combination with a bit-flipper circuit to implement a subtraction circuit (this links the reader back to the relationship between binary addition and subtraction covered in chapter 4).

Control Circuits. Our coverage of control circuits starts with some examples of the types of actions control circuits may perform. We focus in detail on building multiplexor circuits, starting with a 1-bit two-way multiplexer, applying our circuit building algorithm to construct its truth table based on three inputs (A, B, and S, the selection bit). We illustrate how the resulting circuit works by tracing through it with different example inputs. Next, we abstract the 1-bit two-way MUX circuit into a building block from which we build a 4-bit, two-way MUX. We also present how to build a 1-bit, four-way MUX with four 1-bit data input and one 2-bit select input, and trace through some example inputs on the resulting circuit to

illustrate its output, and we discuss the number of selection bits needed for an N-way multiplexer. We also briefly cover demultiplexer and decoder circuits as other control circuits, showing their circuit diagrams and truth tables that define their functionality.

Storage Circuits. Our first sequential circuit is presented in the context of our discussion of storage circuits as part of the CPU architecture. We briefly introduce SRAM versus DRAM as different memory that the reader may know from our coverage of the memory hierarchy in Chapter 11. We discuss that a memory circuit needs a feedback loop to store a consistent value and that it needs functionality to change the value stored in the circuit (to write a new value). We focus our coverage on a single type of memory circuit to reinforce understanding how it works rather than covering many different ones. We choose the R-S latch, and then use it to build a gated-D latch. Much of the coverage in this section is devoted to illustrating how the R-S latch works, stepping through multiple examples of how its inputs determine its outputs, and how its inputs are set when the latch stably and continuously stores a value. We also show how its inputs are set to enable writing either a 0 or 1 value into the latch, showing how the written value propagates through the circuit to become the new stably stored value. We then build a gated D latch from the R-S latch, demonstrating how it ensures that R and S input values are never both 0, and how enabling the write-enable (W) input to the circuit triggers writing to the latch. Finally, our abstraction of a 1-bit gated D latch circuit is used as a building block to create a 32- bit CPU register, that we abstract as a storage circuit with one 32-bit data input, one 1-bit W input, and one 32-bit data output.

At this point, we have demonstrated multi-bit circuits for arithmetic/logic, control, and storage functionality, and are ready to use them to build a full simple CPU circuit.

4.3.3 Building a CPU and Instruction Execution. We begin this section reviewing the functional components of the Von Neumann architecture. We then create an ALU circuit combining N-bit control and arithmetic/logic circuits. We show how a MUX circuit inside the ALU uses the opcode bits of an instruction to choose which of the ALU's arithmetic/logic sub-circuits ultimately determines the ALU's output. We also illustrate how the instruction bits encode the two data inputs to the ALU and the destination of the result, give an example instruction with two immediate operand values, and show how the operands are fed as input to the ALU. We also introduce ALU condition code outputs, and discuss examples how these may be used to execute conditional instructions, like if statements, with some examples. Finally, we abstract the ALU as a circuit with two 32-bit data inputs, one multi-bit select input, one 32-bit data output, and several 1-bit condition code outputs.

We next create a register file circuit from eight 32-bit register circuits, a demultiplexer circuit that chooses which register's WE is set (if any), and two multiplexer circuits, each of which chooses a register's output as the output for one of the two data outputs from the register file. The resulting circuit has eight 32-bit registers, a 32-bit data input, a 1-bit write enabled input, 3 multi-bit select inputs: one to the DMUX to select which register to write to (if the WE input is also set); and two to 2 MUX circuits that select which registers to read from for the two data output of the register file.

Finally, we put these circuits together with control circuitry and buses to create a simple CPU circuit with an ALU, register file, and two special purpose registers, the IR and PC. We focus on the opcode bits of the instruction stored in the IR and illustrate how they determine the output of the ALU, stressing how the values flow through all ALU sub-circuits, but the control circuitry determines the output values. We also show how the data output from the register file can be one of several inputs to the ALU, and how the ALU output can be one of several input sources to the register file.

With our simple CPU circuit, we are able to discuss how it is used to execute program instructions. We start by looking at a generic binary instruction dividing its bits into opcode, two source, and one destination bits, and we use an example instruction that encodes registers in the source and destination bits to illustrate how instruction operand values are read from, and operation result values are written to, specific registers in the register.

We then revisit the 4 stages of instruction execution based on the Von Neumann model (Fetch, Decode, Execute, and WriteBack), and step through each illustrating it on the CPU circuit. We begin with showing how the PC value is used to read the bits of the next instruction from memory into the IR register in the Fetch stage. We then demonstrate how the bits of the instruction in the IR register are used during the Decode phase to select the outputs from the register file to be input into the ALU. Next, we show how the opcode bits in the IR are used to select the ALU output during the Execute phase. And finally, we show how the destination bits in the IR are used to select the register to write the ALU output to in the WriteBack phase. For simplicity, we do not add a fifth Memory stage of execution, and instead just focus on example instructions with all register operands to illustrate how the CPU executes instructions.

With CPU circuitry in hand, we introduce clock circuits and explain how they drive values through circuits, discussing clock edges and propagation delay from initial input change to when a stable output value can be read. We discuss the details of the CPU's clock-driven execution of instructions, introducing the measure of cycles per instruction (CPI) in the context of our simple CPU taking 4 cycles to execute an instruction, one for each stage. We finish this section with a figure of a modern digital computer, showing the parts of the CPU we built in relation to other main parts including cache memory, RAM, I/O devices, and the buses connecting them.

At this point we have achieved the main focus of our introduction to computer architecture: understanding how the CPU architecture is designed to execute program instructions.

4.4 Pipelining

After covering the details of a simple single-core CPU, and explaining how it is designed to execute program instructions in four stages, we introduce pipelining in the context of making the CPU faster. This is our first coverage of parallel execution by overlapping the execution of multiple instructions, each executing in a different stage of execution and each using a different parts of the CPU circuit in each stage.

Additionally, we cover some advanced pipelining issues in a separate section. Here we use a 5-stage execution model, adding a Memory stage to our simple 4-stage model. We discuss data and control hazards that are due both to CISC instructions that use

different numbers of stages and dependencies among instructions in the pipeline. We talk about pipeline bubbles, techniques for getting operand data early, and briefly introduce speculative execution. Our coverage of advanced pipelining issues is not exhaustive, but it introduces the challenges of pipelining in the context of executing a sequence of program instructions.

4.5 Parallel Architectures

Our main chapter on computer architecture (Chapter 5) concludes with a section about the design of today's processors, focusing on support for parallel execution beyond pipelining. We introduce Moore's Law and the power wall in the context of motivating many of today's processor designs. Additionally, in Chapter 15 we introduce accelerators, focusing on GPU architectures. We also cover parallel performance metrics at the end of our chapter on shared memory parallel computing (Chapter 14).

At the end of our main architecture chapter, we present features of modern processor design starting with ILP, including vector processors, superscalar, and VLIW designs that follow from our coverage of pipelining. We then discuss issues with these designs and the power wall that led to the development of architectures that require explicit parallel programming in order to make a program to run faster. We focus in detail on multi-core architectures throughout our textbook and use them as the architecture for motivating our coverage of parallelism and parallel computing. We finish the chapter with a discussion of some examples of current architectures that incorporate many of these features, including a discussion of hardware multi-threading.

The last chapter of our text book (Chapter 15) is a "looking ahead" overview of some other types of parallel systems and parallel programming beyond our book's main focus of shared memory parallelism. We introduce Flynn's Taxonomy and categorize some of the systems we discuss in this chapter as SIMD or MIMD examples. The main architecture-specific part of this chapter introduces heterogeneous computing using accelerators like FPGAs, cell processors, and GPUs. We present some details of the architecture of a generic GPU processor consisting of multiple SM units, each with several SP cores, their own register file, cache memory, and a thread warp scheduler, and a larger GPU memory shared by all SM units. We also introduce CUDA for GPGPU programming, showing some example CUDA programs and explaining how they run on the GPU architecture. Our coverage in this chapter is meant only as an introduction to these other models, with more detailed and in-depth coverage of parallel architectures and programming left for more advanced textbooks on these specific topics.

We also cover some parallel architecture issues in a section on cache coherency protocols for multi-core (and SMP) systems in our chapter on the memory hierarchy (11.6). We first revisit the multi-core processor design using an example with private L1 and shared L2 on-chip caches, which we use to motivate coherency issues. We introduce the cache coherency problem, and coherency protocols as a solution, and step through the details of a simple coherency protocol (MSI), demonstrating how cache block meta data are updated on read and write accesses and how they are used to trigger coherency actions. We briefly discuss implementing these protocols, introducing snooping. In our chapter on shared memory parallelism, we revisit cache coherency in more depth (14.5), focusing on false sharing and some techniques to avoid it.

Finally, our coverage of parallel computing (Chapter 14) revisits its multi-core architecture and introduces OS threads to motivate shared memory parallel systems. This chapter also includes a section on performance measures, including Amdahl's law, speed-up, efficiency, Gustafson-Barsis Law, and scalability.

4.6 Memory Hierarchy and Caching

Our book has a full chapter (Chapter 11) that introduces storage devices, the memory hierarchy, and caching. It begins by describing the memory hierarchy and the trade-offs between storage density, cost, and access time. Next, it characterizes storage devices in more detail based on their performance properties (e.g., capacity, latency, transfer rate, etc.). For each device, we classify it as primary or secondary storage and describe how it connects to the rest of the system. We also briefly discuss the mechanics of hard disk drives and the latency implications of spinning platters and moving arms.

Next, we cover the concept of locality (both temporal and spatial) in general terms using real-world examples. We then show C code examples and highlight instances of locality in the code's memory access patterns, with a focus on how taking advantage of locality can help to improve program performance. These examples motivate the general principles of caching with a scenario of a student storing books on her desk, a shelf down the hall, and a library.

We then dive into the details of CPU caches, starting with basic terminology (hit, miss, cache block, metadata, etc.). Following that, we begin analyzing how memory addresses map to locations in the cache via the *index* bits. We then describe how to identify which address's data is stored in a cache line by verifying the *tag* bits. Finally, we illustrate how the remaining *offset* bits identify a specific byte within the resulting cache data block.

Armed with address division information, we explain direct-mapped caches with an extensive example, followed by distinguishing between write-through and write-back policies for write operations. After exploring direct-mapped caches, we introduce set associativity and the necessary metadata to keep track of replacement policy decisions (i.e., LRU bits). We round out the section with another full example on a two-way set associative cache.

Having characterized caches, we demonstrate Valgrind's tool for evaluating cache performance in a real program, `cachegrind`. Specifically, we contrast the cache hit and miss rates when accessing a two-dimensional array in row-major vs. column-major order. Finally, we wrap up the chapter by characterizing cache coherency behavior on multi-core processors (described above in Section 4.5: Parallel Architectures).

5 OTHER RELATED CONTENT

5.1 Binary

An early chapter of the book (Chapter 4) covers data representation using binary and hexadecimal numbers. Starting with unsigned integers, it describes how to convert from one format to another. We then introduce signed integers, focusing primarily on two's complement and relating those representations to C's type system.

Next we illustrate how to perform arithmetic, including addition, subtraction, multiplication, division, and bitwise operations

on binary integers. Throughout, we discuss the ways in which arithmetic operations might cause overflows, including the implications of overflows how to detect them. Finally, we briefly present byte ordering and representations for real numbers (both fixed- and floating-point) at the end of the chapter.

5.2 Assembly

Dive into Systems also exposes readers to three real-word instruction set architectures across three chapters: 32-bit x86 (IA32), 64-bit x86 (x64), and ARM version 8A (AArch64). We deliberately chose ISAs that students had easy access to (either through laptops, desktops, or single board computers), in keeping with our philosophy of teaching students about their own computers. Each chapter follows a similar structure, covering registers, addressing modes, and various instructions for loading and storing data, accessing memory, implementing conditional control, stack management, and function calls. The high consistency across the chapters enables instructors to choose a single chapter to teach a specific ISA, or adopt multiple chapters to compare and contrast multiple ISAs.

5.3 Parallel Computing

The primary focus of our two parallel computing chapters is to describe how programmers can leverage the power of parallel architectures. Due to the ubiquity of multicore systems, we devote the first chapter to shared memory programming, with an emphasis on POSIX threads. This chapter first compares and contrasts how processes and threads run on single-core vs multi-core systems (with implications for program speedup) before moving on to thread creation, race conditions, synchronization constructs, condition variables, deadlocks, cache coherency and false sharing, and performance topics like speedup, efficiency, scalability, and Amdahl's and Gustafson-Barsis laws. We close the chapter with a discussion of OpenMP, explaining how a compiler uses pragmas to allow programmers to incrementally add parallelism to their programs, while providing a layer of abstraction to the programming process.

The last chapter of the book, in addition to discussing various parallel architectures, presents an overview of the languages and libraries typically used to program them. The scalar multiplication example (which we first presented to the reader in our shared memory chapter), is reintroduced in this chapter, and we use it to illustrate how to parallelize scalar multiplication in both CUDA and MPI. Lastly, we discuss the architectural differences between supercomputers and data centers, discuss the three pillars of cloud computing, and briefly discuss the MapReduce paradigm. We close the chapter with a discussion of some emerging architectures, such as Internet of Things and the rise of domain-specific architectures (like the TPU). We point interested readers to the Turing Award lecture [9] by Hennessy and Patterson for a broader view of the future of computer architecture.

5.4 C Programming

C is used as the high-level programming language for code examples used throughout our book to illustrate topics and to show implementations of functionality. We introduce C programming assuming the reader has a CS1 background in some other language and cover almost all of the C language over two main chapters. In a third chapter, we cover debugging with GDB and Valgrind. Currently, we have two versions of our first chapter on C where we compare some basic C language syntax to syntax in a language the reader may already know (one for Python programmers and the other for Java programmers). Either version of the chapter is presented in a way that is understandable to a reader who does not know the specific comparison language; the language examples that help illustrated C's similarity to the language they know are just less useful in this case. Our second C chapter provides complete and in-depth coverage of the C programming language. Throughout our coverage we stress the parts of program memory, scope, and type, and we describe our many of our C code examples with illustrations of their execution effects on stack and heap memory.

5.5 Operating Systems

Our chapter introducing operating systems (Chapter 13) focuses on the OS's role in running programs (theme 1 of our textbook), and on how the OS is designed to efficiently manage system hardware and software resources (theme 2 of our textbook). It also briefly presents a first introduction to parallel computing topics (theme 3 of our textbook). We primarily focus on what the OS is in terms of its role in implementing the computer system, and on the main abstractions it implements for running programs on the computer, namely processes and virtual memory. In our coverage of the process abstraction we introduce concurrency and asynchrony via signals and signal handling. We present in detail some features of implementing and supporting the process abstraction, including process IDs, process state, fork-exec, wait, kill, address spaces, and multiprogramming. Our presentation of virtual memory revisits the meaning of addresses, distinguishing between virtual addresses from program binary instructions to the physical addresses used to address RAM. We describe in detail a single-level page table implementation of paged virtual memory that includes coverage of address translation hardware and the TLB. We also present some coverage of IPC (including signals, message passing, and shared memory), and we introduce the thread abstraction in the context of explicit parallel computing for multi-core architectures. Throughout our coverage of the OS, we discuss where efficient implementation of its functionality and abstractions require some hardware support, including hardware support for interrupts and user/kernel mode, and for supporting parts of virtual memory address mapping and some IPC functionality.

6 RELATED WORK

The CS education community recognizes many high-quality traditional textbooks that cover computer architecture topics [6, 20, 21, 25, 28], however their cost and availability may be prohibitive to many students [27]. There are not many free, online, complete textbooks for teaching computer architecture. Tarnoff's *Computer Organization Design Fundamentals* [7] covers digital logic, the memory hierarchy, processor architecture and some assembly fundamentals. *Computer Science from the Bottom Up* [10] presents computer systems concepts from digital logic and operating system fundamentals up to the application level, and contains many topics of interest in a computer architecture course. However, some of the coverage remains rather general and/or incomplete. The operating

systems text *Operating Systems: Three Easy Pieces* [5] contains several topics of interest to computer architecture courses. We also note that *The Elements of Computer Systems (Nand2Tetris)*, while not fully free, does freely share the first six chapters online [17], including a chapter on computer architecture.

Several high-quality computer architecture teaching resources consist of smaller modules that are freely available online. The Elements of Logic [11] is a free online guide for logic design and Verilog coding. The Teaching Undergrads Collaborative and Heterogeneous Computing (ToUCH) [19] project contains modules that introduce heterogeneous computing, CPU architecture (ARM), the memory hierarchy (emphasis on GPUs), and programming to take advantage of multiple cores. The CSinParallel [3] and LearnPDC [4] projects contain many modules related to parallel computing. In the context of architecture, they cover heterogeneous computing, GPU architecture, and programming for multi-core systems.

7 DISCUSSION AND ONGOING WORK

While the first edition of *Dive into Systems* was officially released in 2022, the book project remains in active development. One of the advantages of creating an online textbook is that it is easy to make updates and corrections fairly rapidly. Thus far, our book has been strengthened by the level of community involvement in the project, and we plan to continue a community-oriented approach.

For our next major addition, we have an NSF grant that supports the development of interactive exercises and visualization content for our book. In this process, we are working with numerous faculty to create free, interactive exercises that will allow students to validate their knowledge of important concepts in the browser. Our grant includes stipends that pay external collaborators to develop exercises for each chapter. We have recruited exercise developers from various presentations and SIGCSE conference associated events, as well as posting to community mailing lists. Recently, we ran an Affiliated Event at SIGCSE'23 to brainstorm ideas and to recruit more participants. Consistent with our early adopter experience, we have found participants eager to help (some regardless of a stipend) a textbook project that they want to adopt in their own classes, and one that is committed to remaining freely available online. We also think faculty appreciate being able to contribute to a project that benefits the community at large.

At present, we are generating exercises for the C and Assembly chapters. Next year, we will begin generating exercises for Binary and the Memory Hierarchy, with more computer architecture and parallel computing exercises to appear in the coming years. We strongly believe that having faculty input from educators from a wide swathe of institutions has helped improve the clarity of the examples and ensured that the book helps as diverse a student body as possible.

While creating *Dive into Systems* was (and continues to be) a large undertaking, we believe it is ultimately good and necessary to produce free online resources. With textbook costs simply adding additional weight on top of the already heavy cost of a college education, we believe that open access textbooks will be critical to making a computing education attainable for all, especially those from communities with limited resources. Faculty have found *Dive into Systems* to be a useful addition to their courses. For students new to computer architecture, we hope that *Dive into Systems* will spark their interest in computer systems and architecture, and hopefully help inspire the next generation of computer architects.

REFERENCES

[1] ABET Computing Accreditation Commission. 2018. CRITERIA FOR ACCREDITING COMPUTING PROGRAMS. https://www.abet.org/wp-content/uploads/2018/02/C001-18-19-CAC-Criteria-Version-2.0-updated-02-12-18.pdf.

[2] ACM/IEEE-CS Joint Task Force. 2023. Computer Science Curricula 2023. https://csed.acm.org/wp-content/uploads/2023/03/Version-Beta-v2.pdf.

[3] Joel Adams, Richard Brown, Suzanne Matthews, and Libby Shoop. 2013. CSinParallel. https://csinparallel.org/.

[4] Joel Adams, Richard Brown, Suzanne Matthews, and Libby Shoop. 2021. LearnPDC. https://www.learnpdc.org/.

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2020. Operating Systems: Three Easy Pieces. pages.cs.wisc.edu/~remzi/OSTEP.

[6] David Patterson and John Hennessy. 2013. *Computer Organization and Design (5th ARM Edition).* Morgan Kaufmann.

[7] David Tarnoff. 2007. *Computer Organization and Design Fundamentals.* lulu.com. 432 pages. Available at: https://faculty.etsu.edu/tarnoff/138292/.

[8] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture A Quantitative Approach, Sixth Edition.* Morgan Kaufmann Publishers.

[9] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (jan 2019), 48–60. https://doi.org/10.1145/3282307

[10] Ian Wienand. 2020. Computer Science from the Bottom Up. https://www.bottomupcs.com/.

[11] Shing Kong. 2001. The Elements of Logic Design, (Revised by Milo Martin, 2006). https://acg.cis.upenn.edu/milom/elements-of-logic-design-style/.

[12] Suzanne J. Matthews, Tia Newhall, and Kevin C. Webb. 2020. Dive into Systems. https://diveintosystems.org/.

[13] Suzanne J. Matthews, Tia Newhall, and Kevin C. Webb. 2021. Dive into Systems: A Free, Online Textbook for Introducing Computer Systems. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1110–1116. https://doi.org/10.1145/3408877.3432514

[14] Suzanne J. Matthews, Tia Newhall, and Kevin C. Webb. 2022. *Dive into Systems: A Gentle Introduction to Computer Systems.* No Starch Press.

[15] Cailyn Nagle and Kaitlyn Vitez. 2020. *Fixing the Broken Textbook Market* (2 ed.). U.S. PIRG Education Fund. https://pirg.org/wp-content/uploads/2022/07/Fixing-the-Broken-Textbook-Market_June-2020_v2-5.pdf.

[16] National Association of College Stores. 2022. NACS Student Watch Report: Course Materials Spending Dropped. https://www.nacs.org/nacs-student-watch-report-course-materials-spending-dropped.

[17] Noam Nisan and Shimon Schocken. 2008. *The Elements of Computing Systems: Building a Modern Computer from First Principles.* MIT Press. https://www.nand2tetris.org/course

[18] David A. Patterson and John L. Hennessy. 2009. *Computer Organization and Design: The Hardware/Software Interface, 4th Edition.* Morgan Kaufmann Publishers.

[19] Apan Qasem, David Bunde, and Phil Schielke. 2019. ToUCH: Teaching Undergrads Collaborative and Heterogeneous Computing. https://github.com/TeachingUndergradsCHC/modules.

[20] Randal Bryant and David O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective (3rd Edition).* Pearson.

[21] Sarah L. Harris and David Harris. 2021. *Digital Design and Computer Architecture, RISC-V Edition.* Morgan Kaufmann.

[22] Julia E. Seaman and Jeff Seaman. 2023. *Turning Point for Digital Curricula: Educational Resources in U.S. Higher Education, 2022.* Bay View Analytics. https://www.bayviewanalytics.com/reports/turningpointdigitalcurricula.pdf.

[23] The NSF/IEEE-TCPP Curriculum Working Group. 2012. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates. http://www.cs.gsu.edu/~tcpp/curriculum/.

[24] The NSF/IEEE-TCPP Curriculum Working Group. 2020. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates (Version 2.0 Beta). http://www.cs.gsu.edu/~tcpp/curriculum/.

[25] Umakishore Ramachandran and William Leahy. 2010. *Computer Systems: An Integrated Approach to Architecture and Operating Systems.* Pearson.

[26] United States Bureau of Labor Statistics. 2023. College tuition and fees up 4.7 percent since February 2020. TED: Th Economics Daily, https://www.bls.gov/opub/ted/2023/college-tuition-and-fees-up-4-7-percent-since-february-2020.htm.

[27] Will Jarvis. 2019. A Textbook Giant Is Going 'Digital First.' That Might Not Be Good for Affordability. *The Chronicle of Higher Education* (July 2019).

[28] Yale Patt and Sanjay Patel. 2019. *Introduction to Computing Systems (3rd Edition).* McGraw-Hill.