Crucial Topics in Computer Architecture Education and a Survey of Textbooks and Papers

Abdullah Yildiz, Sezer Gören, H. Fatih Ugurdag, Barış Aktemur, and Taylan Akdogan

***Abstract*—We have been teaching undergraduate computer architecture since 2012 in an unconventional way. Most undergraduate computer architecture courses are based on microprocessors, and they quickly move into advanced topics such as instruction pipelining, forwarding, branch prediction, cache, and even memory management unit. We instead spend only the last one-third of our course on these topics. The first two thirds of the course is devoted to microcontrollers, i.e., simple-minded processors with no memory hierarchy, no branch prediction, sometimes even no pipelining. Our claim is that it is very hard to truly grasp the advanced topics without full grasp of the basics. Equipped with the above approach, this article comes up with an all-inclusive list of crucial topics for computer architecture education, and it surveys 25 computer architecture textbooks as well as 38 computer architecture education papers to see how much they cover these topics. In addition to that, the article contains a concise description of the perspective of our course. One of the pillars of our course is a working CPU on FPGA. We have so far had around 600 students design their own unique CPUs using Verilog given a complete instruction set, close to 70% of them with complete success.**

***Index Terms*—computer architecture education; computer organization education; FPGA; microcontroller versus microprocessor; assembly language; instruction set completeness; one-instruction CPU; instruction set design; memory-mapped I/O; self-modifying code; memory banks; memory hierarchy; instruction pipelining; CPU customization**

I. Introduction

SINCE 2012, we have been teaching undergraduate computer architecture with a grounds-up approach at Ozyegin University. We have also been using part of the material in advanced digital design and microprocessors courses at Yeditepe University as well as Ozyegin University. The computer architecture education material in question has been developed at both of these universities. Some of the development work has been carried out through a PhD thesis at Yeditepe University and a funded research project run at Yeditepe University and Ozyegin University. There have also been some altruistic contributions from several graduate students at both universities. Undergraduate students have also contributed through course and senior projects.

Usually, even undergraduate computer architecture/organization courses rapidly move into advanced-level architectural techniques such as pipelining, forwarding, branch prediction, cache policies, virtual memory. They spend the whole semester on these advanced topics without ensuring that the students have the necessary underlying low-level digital design background to appreciate the respective trade-offs.

Advanced-level architectural techniques aim to improve Instructions Per Cycle (IPC) while assuming Instructions Per Second (IPS or MIPS for Million IPS) will also improve. IPS equals IPC times the clock frequency. For IPS to improve when IPC improves, the clock frequency should not go down or should go down less than the IPC goes up. However, advanced-level architectural techniques that boost IPC usually make the hardware more complex, thus lowers the clock frequency (i.e., making the critical path of the logic circuit longer). What is worse is that it is not easy to see the extent of reduction in clock frequency from the architectural/algorithmic trade-off being made, even when a person is experienced in digital design.

Given the above perspective, we have a different view of what is crucial in an undergraduate computer architecture course. This article contributes a *crucial topic list* to the literature for computer architecture education. We also *surveyed* 25 computer architecture textbooks plus 38 computer architecture education papers to see how much they cover these topics. The article also contains a description of our course. We claim it is crucial that students are able to design a working (though simple) *CPU on FPGA* at the end of the semester. So far, around 600 of our students had to design a CPU. 70% of them successfully tested it in simulation and on FPGA.

This article is organized as follows. In the next section, we outline our computer architecture course and teaching philosophy as well as the CPU it is based on. The following section lists and defines the computer architecture topics we or other instructors deem critical, some of which are basic and some of which are advanced topics. The same section does an analysis of each topic in terms of how much it is covered in the related textbooks and articles we found. We then draw some conclusions in the final section.

II. Our Computer Architecture Course and VSCPU

The first time we taught our undergraduate (sophomore level) computer architecture course (in 2012), it was a small class (16 students). As a consequence, we were able to have more interaction and unrehearsed discussions. In one of those discussions, we really tried to define what a "computer" is. It is basically a calculator, as the name has the verb "compute", which is not too different from "calculate". However, the calculation commands of a computer come from a memory in an automated fashion as opposed to them being entered manually by a human. There, we made a further observation that we should call it a computer only if the available calculation commands allowed us to implement an arbitrary algorithm.

Manuscript received February 21, 2020 and revised June 3, 2020. This work was supported by TÜBİTAK under grant no. 117E090.

A. Yildiz is with the Dept. of Computer Engineering, Yeditepe University, Istanbul, 34755, Turkey e-mail: ayildiz@cse.yeditepe.edu.tr.

S. Gören is with the Dept. of Computer Engineering, Yeditepe University, Istanbul, 34755, Turkey. H. F. Ugurdag and T. Akdogan are with School of Engineering, Ozyegin University, Istanbul, 34794, Turkey. Barış Aktemur is currently a senior software engineer at Intel, Germany; his contributions to this work were carried out while he was a faculty member at Ozyegin University.

Hence, we realized that we are in fact looking for a set of commands that serve as the "basis vectors" of the space of all algorithms. Basis vectors are such that by using their multiples and superimposing them we can arrive at any point in the vector space. Another way of looking at it is that the commands implemented by a computer (a.k.a. instructions) are like the elements in the periodic table. They are the atoms from which molecules are made and in turn matter is made. Then, we naively asked what those fundamental elements in our case (i.e., instructions) are. In other words, we were curious about the smallest set of instructions one can come up with, namely, the question of "instruction set completeness". Therefore, the whole classroom at this point agreed that we could call a compute engine a computer only if its instructions possessed "instruction set completeness".

To paraphrase it, the question we posed was "what is the minimum number of instructions necessary and what are they?" so that we can do any set of discrete computations (see the crucial topic of Instruction Set Completeness in the next section). It is an easy question but does not have a simple answer.

Some students had the right intuition and said one instruction is enough, and that is the `NAND` instruction. Their reasoning was that in the prerequisite Digital Systems course, we taught them that NAND is a universal gate that can implement any logic. We obviously told them that they are forgetting about conditionals and loops (but yet a one-instruction CPU is possible). With all the naiveness of this initial edition of our course, we designed a 16-instruction CPU, which is instruction-set-complete (otherwise, it could not be called a CPU).

Having said that, we believe the most important topic in computer architecture education is "instruction set completeness" as it is how one can define a "computer". And as you will see in our survey section below, hardly any textbook or article gives it the emphasis it deserves.

We called the CPU we designed to support our computer architecture course "VerySimpleCPU" and VSCPU in short. There is also the "Very Simple CPU" of Carpinelli [1]. It is just a coincidence that we have the same name. Otherwise, the two are completely different, even to an extent that Carpinelli's Very Simple CPU is not instruction-set-complete. To reduce the confusion between the two CPUs, we will call ours VSCPU.

VSCPU [2] is a microcontroller (MCU) such as PIC16. Hence, it does not have memory hierarchy or even a pipeline. It is a bare-bones CPU. It has Von Neumann architecture (although we have also implemented its Harvard variants). Its memory words are 32-bit wide. Its instruction words (`IW`) have a 3-bit opcode, a 1-bit immediate flag, and two 14-bit arguments (see Fig. 1(a)). Including the immediate flag, it has 16 instructions. If we just had a 3-bit opcode (hence no immediate flag), then either one of the two arguments would be 15 bits or we would have an unused bit in the `IW`. By the way, we could have had 1 or 3 arguments (see Fig. 1(b)) instead of 2 arguments. Having 2 arguments strikes a good balance between addressed memory space and code density.

Our computer architecture course has 3 parts. Part 1 teaches the basic architecture of VSCPU and its assembly programming. Part 2 deals with designing it in Verilog and mapping it to FPGA. Part 3 is a shorter version of traditional courses and hence teaches some of the advanced topics listed earlier. There are also lab sessions. The lab supports Part 1 and 2. On top of it, the students work on a project and design a CPU called ProjectCPU (a cross between VSCPU and PIC16) and make it work on FPGA.

We developed a set of tools and components for and around VSCPU:

- Two different instruction sets (one also supports floating-point arithmetic)
- Assembler
- Instruction Set Simulator (ISS)
- Web-based ISS
- C compiler
- PIC16 to VSCPU assembly converter
- FPGA debug interface
- Worst Case Execution Time (WCET) profiler
- Hundreds of synthesizable Verilog implementations (including pipelined versions)
- Several peripherals
- Several customized versions

Table I lists the instructions of VSCPU v1, which has only unsigned integer instructions. Although there are a total of 16 instructions, we list only the 8 fundamental ones, which are enough for doing any computation. Instructions that are not fundamental, hence not listed, are `ADDi`, `NANDi`, `SRLi`, `LTi`, `CPi`, `BZJi`, `MUL`, `MULi` and can be implemented using the fundamental instructions.

`MUL` is multiply and is not a necessary instruction as it can be implemented in software as a function. We have it just because we have a spare opcode. `ADDi`, `NANDi`, `SRLi`, `LTi`, `CPi`, `MULi` are such that we replace `*B` with `B` in the function of the respective instructions in Table I. For ex., `ADDi A B` does `*A = *A + B`. The "i" in an instruction name denotes that it is the *immediate* version of the respective instruction. `BZJi` reads as Branch on Zero or Jump immediate and acts as a Jump instruction. We did not want to violate the rule that every instruction has an immediate version. Also note that all instructions except `BZJ` and `BZJi` implicitly perform `PC = PC + 1` (not shown in Table I), where `PC` stands for Program Counter, same as Instruction Pointer.

It is possible to have an instruction-set-complete CPU with only one instruction. We will later discuss our version of a one-instruction CPU, but then it is not practical as it has poor assembly code density. VSCPU, on the other hand, has good assembly code density, is easy to code and read at assembly level. It has even better code density, readability, memory space size, and performance than even PIC16 (a commercial MCU). That is because PIC16 has one operand whereas VSCPU has two operands, and PIC16 has 8-bit data words whereas VSCPU has 32-bit data words.

VSCPU is perfect for assembly coding so that a student can understand CPU functionality at a very low-level and hence can design it in Verilog on FPGA. Every single student designs both their own VSCPU on FPGA in the Lab and what we call a ProjectCPU (yet another instruction set, one argument, and is similar to PIC16) for their project assignment. The students are expected to come up with unique implementations of ProjectCPU. We correlate their Verilog using MOSS [3] and verify the uniqueness of their work.

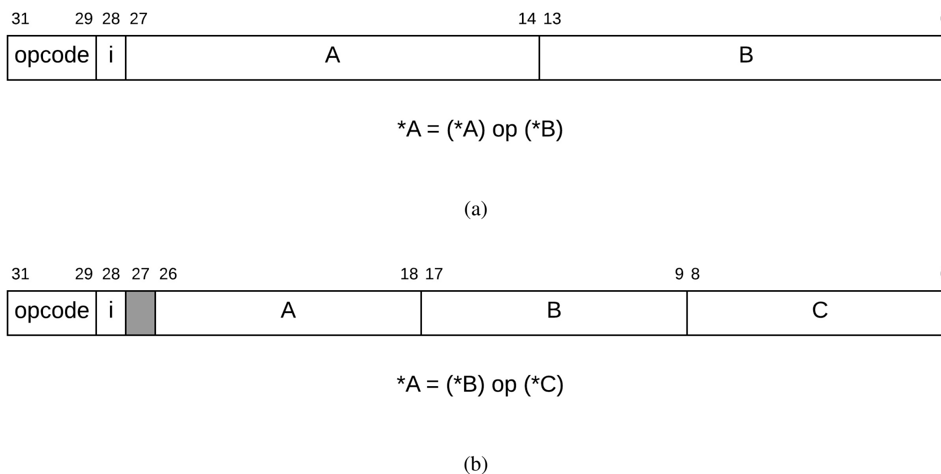


Fig. 1. (a) VSCPU IW. (b) An alternate IW with 3 operands.

We will further discuss our VSCPU and the course around it in the next two sections, when we discuss the crucial topics in computer architecture education and their coverage by textbooks and education articles as well as our course.

III. Crucial Topics in Computer Architecture Education

This section is the main contribution of this article to the literature. We list here 18 crucial topics that may be considered in computer architecture or related courses. We list not only the topics we regard important but also topics that are covered in courses that are more traditional than ours.

- Assembly Coding
- Instruction Set Completeness
- One-Instruction CPU
- Tradeoffs in Instruction Design
- Memory-Mapped I/O
- Indirect Memory Access
- Detailed Explanation of Harvard vs von Neumann
- Self-Modifying Code
- Granularity of Simulation Environment
- Working CPU on FPGA
- Memory Banking
- Memory Hierarchy
- Virtual Memory
- Pipelining and Hazards
- Forwarding
- Branch Prediction
- Out-of-Order and Multiple Issue
- CPU Customization

We did a comprehensive survey of computer architecture textbooks [4]–[28] and papers on computer architecture education [1], [29]–[65] to find out which of them cover the individual crucial topics listed above. Figure 2 summarizes how much the textbooks and the papers we surveyed cover the individual topics.

In this section, we present the results of our survey for these crucial topics by first giving a definition of each topic, and then we evaluate our findings on what percentage of the textbooks and papers cover each topic.

A. Assembly Coding

The ultimate goal of a computer architecture course would be to design a working CPU on FPGA. It is only possible to design a system if one truly understands the relationship between the inputs and outputs. The input of a CPU is the program (i.e., a sequence of instructions) and input data, and the output is the output data. At the center of all this, it is the instructions, which are of key importance. The set of instructions is the true programming language of the CPU. To design a CPU, we first need to speak its language, and hence be able to code in its assembly language.

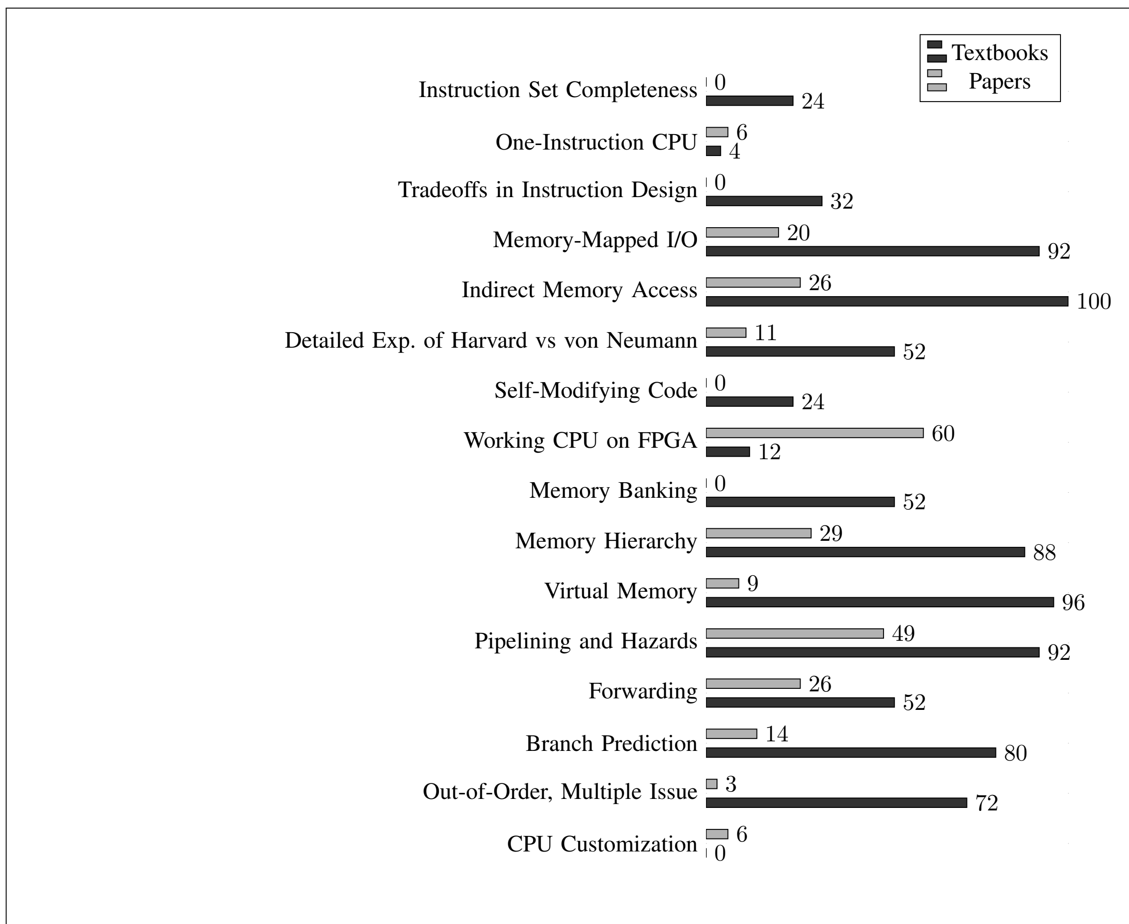
Assembly language of a CPU goes with what we call the Instruction Set Architecture (ISA) of the CPU, namely, the set of instructions and architecture the instructions assume. The internal design of a CPU can be done in hundreds of different ways but they would all be the same CPU as long as the assembly language (hence the ISA) is the same.

We quickly give the ISA in the first two weeks of the course, which we explain as a state machine (processor core) and a memory. We state the ISA not as something written in concrete but instead as something that could have been designed differently. We spend the rest of first one third of the course solving programming problems using assembly code. Most courses use assembly coding to a certain extent. What is critical is the extent of assembly coding in the course. For us, it is a pillar of the course.

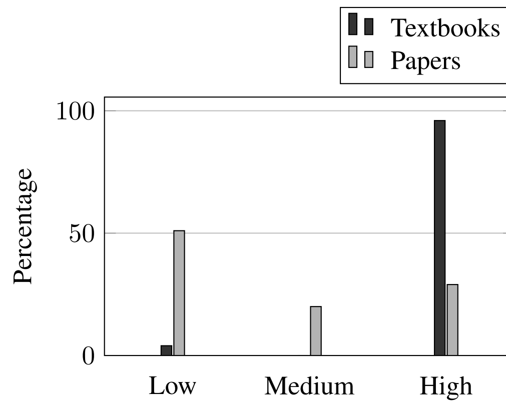
We classified the emphasis on assembly level coding as three different categories: high, medium, and low. Based on our findings, about 50% of the papers [1], [30], [32], [34], [38]–[41], [43], [46], [49], [51], [54], [56], [60], [62]–[65] put low emphasis, about 20% of them [33], [37], [44], [45], [53], [55], [58], [59], [61] put medium emphasis, and 30% of them [29], [31], [35], [36], [42], [47], [48], [50], [52], [57] put high emphasis on the importance of assembly-level coding. Almost every textbook [4]–[23], [25]–[28] puts high emphasis on assembly-level programming for the respective processor they introduce.

B. Instruction Set Completeness

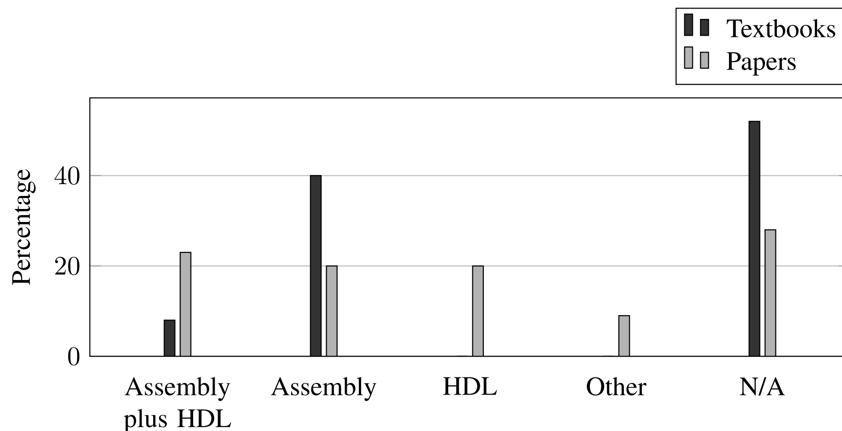
Computer Architecture courses are mostly about general-purpose CPUs. A CPU has an instruction set and a memory



(a) Coverage of topics



(b) Emphasis on assembly coding



(c) Granularity of simulation environment

Fig. 2. Comparison of textbooks and papers on teaching processor design.

TABLE I
VSCPU V1 INSTRUCTION SET.

Instruction	Description	Functionality
ADD A B	ADDition	*A = *A + *B
NAND A B	bitwise NAND	*A = ∼(*A & *B)
SRL A B	Shift Right or Left	*A = (*B < 32) ? (*A >> *B) : (*A << (*B−32))
LT A B	Less Than	*A = *A < *B
CP A B	CoPy	*A = *B
CPI A B	CoPy Indirect	*A = **B
CPIi A B	CoPy Indirect immediate	**A = *B
BZJ A B	Branch on Zero or Jump	PC = (*B == 0) ? *A : (PC + 1)

architecture around it, hence the ISA. As we have tried to point out before, if we are speaking of a general-purpose CPU, we need to define what a CPU is. A CPU needs to have a "complete" instruction set (not an arbitrary set of instructions), hence the question of instruction set completeness.

A complete instruction set makes any computation possible. A Universal Turing Machine (UTM) can compute anything. Therefore, proving that a given ISA can emulate a UTM would prove its instruction set completeness (that it is Turing-complete). However, Turing machines are cumbersome machines, and this approach is just a theoretical exercise. Instead, we recommend the approach of assuming a well-established CPU to be instruction-set-complete and writing a converter that converts assembly programs of the well-established CPU to the assembly of the CPU in question. We have, for ex., a rigorous assembly converter that converts from PIC16 to VSCPU. Obviously, the converter has to handle the differences between the two architectures as well.

We previously mentioned that VSCPU has 16 instructions but actually 8 of them (see Table I) form a complete set. In our course, we also discuss the fact that some of these 8 instructions can be implemented using others, hence making a smaller subset a complete instruction set. For ex., we have previously asked on an exam the implementation of `CPI` and `CPIi` using self-modifying code and `CP`, `NAND`, `SRL` instructions. On the other hand, `BZJ` can be implemented with `CP`, `NAND`, `LT` instructions if we memory-map `PC` (Program Counter register in the Core). `ADD`, however, can be implemented using, `CP`, `NAND`, `SRL`, `CPI`, `CPIi`, `BZJ`.

Based on our survey, there is no paper that mentions this issue. When we looked at the textbooks, only one [20] mentions the relationship between an instruction set and Turing machine. However, 20% of the textbooks [6], [7], [9], [19], [25] discuss the completeness of an instruction set in terms of the necessity of arithmetic, logic, branch, and memory instructions within a processor.

C. One-Instruction CPU

In our computer architecture lectures, once we start discussing instruction sets and whether they are complete or not, the following question comes up: "What is the minimum number of instructions possible in a complete instruction set?"

Some students quickly say that it is possible to have a complete instruction set with only one instruction. When we ask them what that single instruction would be, they quickly reply "NAND". Their justification is that in a previous digital design course, we told them that we can construct any logic with a two-input NAND gate (i.e., a universal gate like NOR and MUX).

It is clear that any truth table (hence any combinational logic cloud) can be implemented with NAND gates. However, a program is like a hardware state machine. State machines are simply combinational logic plus flip-flops, and flip-flops can also be implemented with NAND gates. However, there are physical issues there. And more importantly, the NAND gates in a flip-flop cannot be simply replaced with `NAND` instructions. The correct approach would be to start with a complete instruction set longer than one instruction and implement the instructions there with `NAND` instructions. The tricky part there is the implementation of branch and jump instructions.

One-instruction CPUs are called "One Instruction Set Computer" (OISC) in the literature [20]. Although this is not a perfect name, we guess it has been adopted due to its similarity to RISC. Design of OISCs has been so far mainly an educational exercise. Recently, OISC architecture has been used as an enabler for encrypted data computation [66], [67].

The earliest publication that describes an OISC is Mavaddat and Parhami [63]. The respective CPU is called URISC for Ultimate RISC. The single instruction that does it all is *subleq* and has three addresses as argument. It subtracts the contents of the first two addresses and branches to the third address if the result is negative. The nice thing about URISC (and all other subleq CPUs) is that they are word-oriented. That is, they process multiple bits with each instruction. The criticism is that this is actually multiple instructions in disguise of a single instruction, namely, subtract, less than, and branch.

It is also possible to have an OISC around only a *copy* instruction (mostly called *move*). This is possible if the ALU ports are tied to certain memory addresses (i.e., memory-mapped). This can be regarded as moving the ALU from the CPU core to the peripherals. In other words, the ALU becomes a coprocessor. This is also a multi-instruction machine in disguise, as with this approach, we can have hundreds of instructions and yet call the machine an OISC. This basically gets us into the area of "Transport Triggered Architectures" (TTAs) [68], which is recently popular again, but yet is beyond the scope of this article.

In our lectures, the question that comes up is not only if an OISC is possible but also if it can be done with a `NAND` instruction. The same question can be asked in regards to a `NOR` instruction. And "The NOR Machine [69]" partially answers that. Its `NOR` instruction is word-oriented like *subleq*. The NOR Machine is partially a TTA

as "shifting" is handled on a memory-mapped coprocessor. Branching is similar although no coprocessor is needed. It is simply that PC is memory-mapped.

Since the students' questions center on NAND and since, in our opinion, other OISCs are not truly one-instruction CPUs, we designed a NAND Machine. As our NAND based OISC is bit-oriented, it does not need a coprocessor that shifts bits. However, PC is memory-mapped just like the NOR Machine. On the other hand, we built a single-operand machine just like PIC16 family of MCUs and unlike the three-operand NOR Machine.

The NAND machine has a single instruction like other OISCs, hence no opcode is needed. And since it has a single operand, each instruction is nothing but simply an address. It has memory with a wordsize of one bit. The machine has a W register in the core just like PIC16, and we instead call it R. The single instruction NANDs the bit at the address pointed indirectly by the instruction (A) with the bit in R and writes it into both R and memory location [A] (i.e., `R=*[A]=!(R&*[A])`). [A] refers to the number formed by bits `(*A, *(A-1), *(A-2), ..., *(A-n+1))`, where n is the bitwidth of the memory address.

Our approach in proving that this machine can compute anything has been to create layers of more and more high-level instructions. In the process of doing that, we call the single fundamental instruction "0th order instruction". Using this instruction, we create "1st order instructions". Using those, "2nd order instructions" are created and so on.

The 1st order instructions are what we call Z, T, and A. These also represent addresses as in our case, i.e., an instruction is nothing but an address. With a program, we can always initialize data. Z is a location in memory that is guaranteed to be or initialized to zero, whereas T is a location that is true (1). A is an instruction where the address is an address other than Z or T. A is not a particular address. It can be anything.

Using Z, T, and A, we can implement `R=0`, `R=1`, `*[A]=0`, `*[A]=1`, `R=*[A]`, `*[B]=*[A]`, `*[A]=!*[A]`, `*B=!*[A]`, which we call the 2nd order instructions. The 3rd order instructions are 3-operand logic instructions. At the very top, we built a scripting language that even supports functions. We have a program that converts our scripting language to our NAND machine's instructions.

Last but not least, our OISC implements jump capability by mapping PC to particular n locations in the memory (n being the address bitwidth). When the MSB of these bits is written, the jump happens. If it is a branch that we are implementing, the address we write into that special location has the possibility of being the next instruction location.

Only 6% of the papers [46], [63] propose single instruction processors although they do not analyze them to show that they are instruction-set-complete. Only one of the textbooks [20] surveyed analyzes the topic of one-instruction CPU in detail and introduces a theorem to determine whether a processor is instruction-set-complete as well.

D. Tradeoffs in Instruction Design

VSCPU instructions have 2 operands. However, in our computer architecture course, we consider 3-operand instructions as well. We start with a Von Neumann architecture and use a memory with 32-bit wordsize. We justify 32 bits by saying it is a commonly used wordsize in SRAMs. We subtract 4 bits for the VSCPU opcode (including the immediate flag) from 32. That leaves us with 28 bits for 3 operands, i.e., two 9-bit operands and one 10-bit operand. There is no point in having an operand wider than the others. And with 9 bits we can address 512 locations, which may be somewhat small for quite a few programs. Although memory size can be addressed with banking, at this point the students do not know about memory banking, and also it introduces some complications. We then say if we had 2 operands in each instructions, then they could each be 14 bits and could address 16K locations. That is a memory of 64KB since each location is 32 bits. 64KB is a decent memory size for an MCU.

As a a course project, the students design what we call ProjectCPU, which is a cross between VSCPU and PIC16. ProjectCPU is 1-operand just like PIC16. Therefore, the students see the tradeoffs between different number of operands. As the number of operands gets smaller, the program gets longer, because we need more instructions to the same thing. However, the number of memory locations we can address gets larger or memory wordsize can be smaller hence giving us more locations for the same memory size.

None of the papers we surveyed mention this topic, while about 30% of the textbooks [9], [11], [14], [21], [23]–[25] cover this topic.

E. Memory-Mapped I/O

Mapping special registers or special hardware blocks to particular memory locations is a very crucial concept in computer architecture. Because of that, we had to use this concept in the prior subsections above without fully explaining it.

A basic CPU is nothing but a machine (i.e., the Core) that keeps reading the Memory and writing it, continually transforming the numbers in the Memory. The only interface of a basic CPU is an interface between a host and the Memory, through which the Host can inject a program and initial data values and can offload the final values of output variables from the Memory. Therefore, a basic CPU only supports batch processing.

How do we use a CPU in interaction with a physical world and run interactive real-time programs on it? For such interaction, we need additional pins on the CPU other than the ones for the host interface. If these pins come out of the Core, that means the new core with additional pins would need to have additional instructions. If each version of the CPU has a different instruction set, that means every version of the CPU would need modifications in the compiler.

The solution is that these additional pins are memory-mapped, i.e., each pin has a separate address as shown in Fig. 3. As a result of this, setting and resetting these pins or reading them can be implemented with the *copy* instruction. The only hardware change would be redesigning the Memory Decoder in the Memory (see Fig. 3).

Note that sometimes CPU pins are driven by some dedicated logic (in a way coprocessors). In that case, what are memory-mapped are not the pins but the configuration registers of the logic, which is usually called a *peripheral*.

About 10% of the papers [41], [45], [57], [65] briefly mention memory-mapped I/O model and about 8% of them

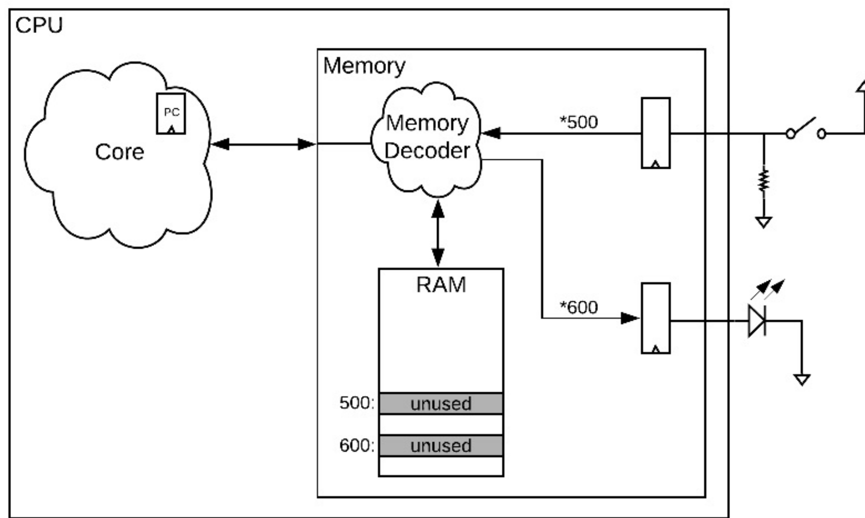


Fig. 3. Memory-mapped I/O.

[47], [50], [52] strongly emphasize memory-mapped I/O model. Within the textbooks, we observed that almost every textbook [4]–[10], [12]–[19], [21]–[28] covers the topic of memory-mapped I/O.

F. Indirect Memory Access

In a regular CPU instruction, there is an opcode as well as flag(s) and operands. An operand is either an immediate number or an address. However, neither of these can handle arrays unless self-modifying code is written (note that self-modifying code requires von Neumann architecture or modified Harvard). To handle arrays, "indirect memory access" is needed. That is about operands in certain instructions that point to addresses that contain an address. VSCPU has indirect addressing through one operand in `CPI` and `CPIi` instructions (see Table I and Fig. 4). VSCPU also, in a way, has indirect addressing in `BZJ` and `BZJi` instructions.

In the case of PIC16, for ex., indirect addressing is not implied through particular opcodes but a particular address. For that particular address, the value of not the address itself but the value at the address pointed by it is used. And to set that particular location itself, another special address is used.

About 25% of the papers [36], [38], [40]–[42], [45], [50], [58], [65] touch upon indirect addressing mechanism, whereas every textbook [4]–[28] discusses indirect memory access mechanism in detail.

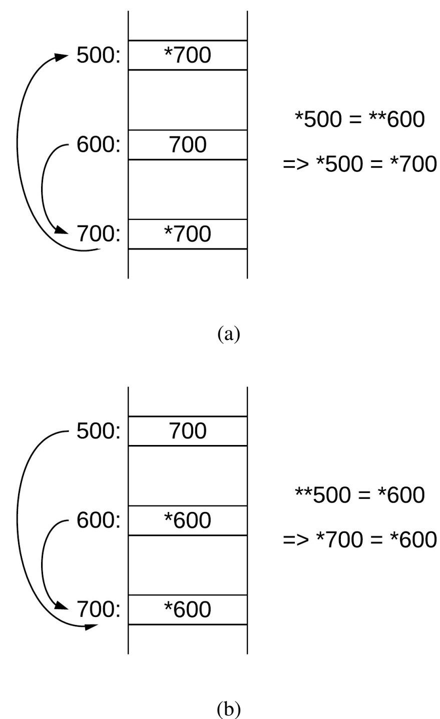


Fig. 4. (a) CPI 500 600. (b) CPIi 500 600.

G. Detailed Explanation of Harvard vs von Neumann

A system that employs Harvard architecture has separate memory spaces for instruction and data. On the other hand, a system that employs von Neumann architecture uses the same memory space for both instruction and data.

The main advantage of Von Neumann is that it can boot itself. That is, it does not need a host to load the program. Also, it offers "simplicity". We do not have two different types of addresses. One other advantage is that program and data can be intermixed. Another one is that it allows self-modifying programs although this is not a significant advantage.

On the other hand, Harvard has the advantage that the total memory space is smaller and hence the data addresses (i.e., instruction operands) are smaller in bitwidth compared to Von Neumann. As a result, the instruction width can be smaller thus reducing the program memory size. It should be noted that if the ISA has branch and jump instructions with fixed address operands, then those instructions include program space addresses unlike the rest of the instructions. In the case of VSCPU, we do not have this inconsistency in the Harvard version of our ISA as our `BZJ` and `BZJi` take as an operand an address in the data memory space, which in turn, points to the program memory space.

There is also a performance aspect. Harvard (a single-port program memory and a single-port data memory) is equivalent to a 2-port Von Neumann. However, a 2-port memory is slower and more expensive compared to 2 single-port memories.

Most microcontrollers are Harvard, while microprocessors are Von Neumann from a main memory perspective and Harvard from a cache perspective.

We found that the comparison between Harvard and von Neumann architectures was briefly mentioned in about 10% of the papers [39]–[41], [65]. On the other hand, about 50% of the textbooks [7], [10], [13], [16]–[19], [21], [23], [24], [26]–[28] discuss this topic.

H. Self-Modifying Code

As explained before, for code to be self-modifying, either the processor should be Von Neumann or modified Harvard. Modified Harvard is such that a few special addresses in the data memory space serve as a gateway/bridge to the program memory space.

Self-modification can be used to make programs smaller and faster. However, such programs are very difficult to debug. Therefore, self-modifying code is not a desired feature. It rather allows for theoretically interesting cases. For ex., the ability of programs modifying themselves in an ISA has an impact on instruction set completeness. In VSCPU, if self-modifying code is written, `CPI` and `CPIi` are not needed as they can be emulated with a few lines of code using the rest of the instructions. Figure 5 shows an example on how to emulate VSCPU `CPI` instruction with a set of other VSCPU instructions.

On a similar note, in ISAs with branch and jump instructions with fixed target address, self-modifying code can be used to turn branch and jump instructions to instructions with variable target address.

One convincing benefit of self-modifying code can be making reverse engineering difficult.

None of the papers discuss self-modifying code. However, this topic is covered in about 25% of the textbooks [6], [7], [10], [17], [24], [27].

```
100: NAND 103 601
101: NAND 103 103
102: ADD 103 600
103: CP 500 0

601: 0xFFFFC000
```

Fig. 5. This code does CPI 500 600 (“100:” for ex. indicates the instruction is stored in address 100).

I. Granularity of Simulation Environment

It is indisputable that simulation tools are a must for computer architecture education. When the point is the simulation of a processor, the level of simulation affects the learning curve of students in understanding the working mechanism of a processor. Computer architecture courses based on complex architectures such as x86, ARM, MIPS, etc. usually prefer to use an ISS due to the course length. The drawback of this approach is that anyone who runs the simulator can only examine the current status of registers of the processor and memory for the instruction that is currently executed. As a result of that, it can be difficult to track what is happening in the processor over time while instructions are executing.

A better approach is to use a cycle-accurate simulator for the processor and base the teaching on a less complex processor architecture. If such is the case, usually a model of the processor written in a Hardware Description Language (HDL) is used. Thus, it becomes possible to track changes that take place inside the processor during the execution of a program at the level of individual clock cycles.

While 23% of the papers [30], [32], [35], [38], [43], [44], [47], [58] discuss employing an environment that supports both assembly and HDL-level simulation within the courses, 20% of the papers [1], [29], [36], [42], [48], [50], [60], [65] discuss assembly-based approach, and 20% of the papers [31], [33], [34], [37], [39], [40], [53], [56], [61] discuss HDL-based approach. Within the textbooks, only 8% of them [15], [20] address both assembly and HDL-level simulation, and about 40% of them [4]–[9], [11], [14], [17], [21] only prefer to base things on assembly-based simulation.

J. Working CPU on FPGA

FPGAs are very helpful for instructors of computer architecture in showing the students to put their hardware modifications in action on a CPU working real-time. Students can design any hardware on their computers by writing its description in HDL such as Verilog and then download it on to the FPGA in a few minutes. The change from the ability to design processors on paper to within a simulation software is huge, while the transition to students implementing their own CPUs (even individually customized) on a chip is even bigger.

Using FPGAs to support computer architecture education was addressed by 60% of the papers [30], [31], [33]–[37], [39]–[41], [43], [44], [46], [47], [49], [50], [52]–[54], [56], [58], [61], [64]. Interestingly, only 12% of the textbooks [5], [8], [12] surveyed mention FPGA-assisted learning method.

K. Memory Banking

Memory banking is about implementing a memory with multiple physical memories. Memory banking comes up in different contexts. It can be used to improve memory access time (as in the case of DRAMs) by creating a pipelined access, though at the cost of additional area. However, in our courses, we approach it from a slightly different perspective (i.e., microcontroller perspective).

The address part of an instruction in a microcontroller is not too many bits, hence the microcontroller cannot address a huge memory space. In the case VSCPU, an address is 14 bits, and the addressable Von Neumann memory space is $2^{14} = 16K$ locations. If that space is not enough to hold all of the program and data, we may want to modify our CPU by making the addresses 15 bits. That lets us address $2^{15} = 32K$ memory locations. However, there is a side-effect. The instructions have to grow from 32 (=3+1+14+14) bits to 34 (=3+1+15+15) bits. VSCPU uses 3 bits for opcode and 1 bit for immediate flag. In a simple and efficient CPU, an instruction has to fit in one location (i.e., one word) of the memory. And since VSCPU is Von Neumann the whole memory should grow also in width besides its growth in height (i.e., number of locations). This is the case even when all variables still fit in a wordsize of 32 bits.

One approach is to use a Harvard architecture and decouple instruction and data memory from each other and let them have different wordsizes.

However, a better approach is “Memory Banking”. This can be used in conjunction with a Von Neumann or Harvard architecture. We can use 2 or more banks. Let us suppose we use 2 banks. That is, we put a second memory with another $2^{14} = 16K$ locations in the case of VSCPU. The core of the CPU thinks there are only 16K locations (hence a 14-bit address) and accesses the bank on the top (see Fig. 6). If the programmer wants the CPU core to access the other bank, the other bank has to be brought to the top. Memory-Mapping comes to the rescue at this point. The CPU designer picks a special address (1 in Fig. 6). This location serves as a single-bit value and selects between the two banks. One way of looking at this is as follows. In a way, we still increase the address to 15 bits from 14 bits. However, the CPU core (i.e., the instruction set) is not aware of this. The only people that know this and are in coordination are the memory designer and programmer. Memory designer is the person that puts the memory banks there and designs the memory decoder, both of which are internal to the memory block. The programmer writes the appropriate bank number in location 1 (with a `CPi` instruction), when there is a need for changing the memory bank.

However, there is a small problem. This trick works with data memory but not instruction memory. There is a problem both with Von Neumann and Harvard architectures. Let us look at how we can solve the problem when the architecture is Von Neumann. We do not want the bank number to change for instructions when the bank number changes for data. Also, for instructions, we cannot simply use a `CPi` to change the bank number. Instruction address (`PC`) has to either keep incrementing or should jump.

Here is a solution for this problem. We add an additional bit to the address port of the memory. This bit indicates if the memory access is for an instruction or data. If it is a data access, what we suggested earlier in terms of banking works. If it is an instruction, the address is actually an offset. That is either 1 (for most instructions) or the jump amount as a distance from the current instruction to the jump/branch target. All of this implies that the current instruction address (`PC`) is kept in the memory block (maybe in addition to the CPU core). If the jump/branch target addresses are kept as offsets, this approach is more easily implemented. However, even without it, this is a feasible approach.

When we look at the literature, we see that none of the papers mention memory banking, while about 50% of the textbooks [5]–[7], [11], [13]–[16], [18], [23], [24], [26], [27] cover the topic.

L. Memory Hierarchy

Microcontrollers (for ex. VSCPU) do not have memory hierarchy, while microprocessors (i.e., sophisticated processors that even have hardware support for easily running operating systems) have memory hierarchy. Memory hierarchy is a solution to a dilemma. The dilemma is as follows.

Consider a basic CPU such as VSCPU, which has a single memory space and has a single physical memory. The smaller the memory, the faster it is, and hence the CPU is faster.

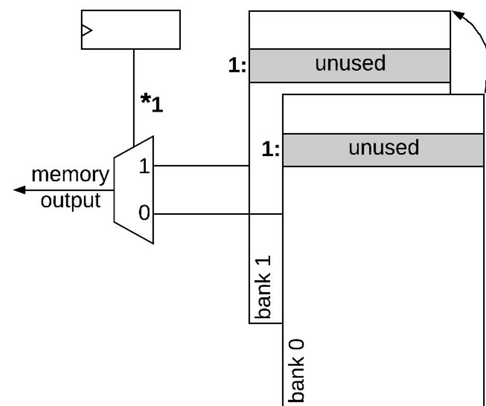


Fig. 6. Memory banking.

However, small memory does not support big programs. To address that, if we make the memory large, then the CPU slows down. The question is how we can have the best of both worlds.

The answer to this question is introducing “Memory Hierarchy”. That is, CPU core most often interacts with a small and hence fast memory. It every once in a while interacts with a bigger (main) memory and fills the smaller memory. This brings us to a “Load/Store Architecture”. That is, all instructions operate on the locations of the smaller memory (address space 1), except load and store instructions, which serve as a bridge between address space 1 and the address space 2 of a bigger (but slower) memory. A register file and a main memory (with an option of cache included) is an example for this approach (see Fig. 7).

Another approach is the cache (abbrev. as $) approach, which is a hardware based approach for memory hierarchy, which we may also call “big but yet fast” or “big and small” approach. In this approach also, the CPU core interacts with a small memory, which is supplied from a bigger memory. However, the difference is that the smaller memory does not interact with the big memory through load and store instructions, instead they interact through an automated hardware mechanism. And one other difference is that there is only one address space.

In modern microprocessors, both of the mechanisms explained above are employed. Therefore, they have a memory hierarchy with register file at the top, then cache, and then main memory.

We motivate the concept of memory hierarchy with examples such as refrigerator/super market and bookcase/library.

In our courses, we also discuss the full memory hierarchy including the disk (as well as virtual memory and swap space). In addition, we discuss the usual physical implementations of the types of memory in the memory hierarchy. One interesting observation we share with students (especially those with EE background) when covering this topic is that memory hierarchy mechanism resembles “impedance matching”.

We observed that about 30% of the papers [32], [43], [47], [48], [51], [52], [54], [57]–[59] discuss including memory hierarchy in a typical curriculum, and about 90% of the textbooks [4]–[14], [16]–[18], [21]–[28] at least cover cache memory in detail.

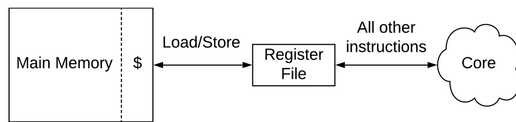


Fig. 7. Memory hierarchy.

M. Virtual Memory

"Virtual memory" is such that each program (i.e., process) thinks that it has a dedicated memory space. This prevents programs from corrupting each other or the operating system. The instructions point to addresses in the virtual memory space, which need to be translated to the "physical memory" space. The physical memory addresses are the addresses fed to the pins of the physical memory. The translation between the two types of addresses has to be made in real-time. That is why modern microprocessors employ a hardware unit called Memory Management Unit (MMU).

Problems that arise as a result of this address translation problem are fragmentation, cache access efficiency, and the size of tables in the MMU. The concept "page" and "page tables" come to the rescue at this point.

The concept of virtual memory not only allows every program to have its memory space but also allows hard disk to act as an extension of DRAM (i.e., main memory), thus the "swap space".

We have seen that about 10% of the papers [47], [48], [57] mention this topic, while almost all of the textbooks [4]–[18], [20]–[28] cover it.

N. Pipelining and Hazards

Today almost every processor employs instruction pipelining technique since it improves clock frequency and Instructions Per Cycle (IPC) of the processor. However, pipelining a datapath of a processor has some consequences, which are commonly called as hazards. There are three types of pipeline hazards as structural, data, and control hazards. A basic pipeline addresses a structural hazard, i.e., the same hardware unit being busy for more than one cycle.

For a student to fully understand pipelining of a processor, he/she needs to have a strong logic design background. Although we teach the basic idea of pipelining in our logic design courses, we do a quick overview in computer architecture and other advanced courses. We introduce cut-lines to a combinational (i.e., feed-forward) logic and put flops on the cut-lines, thus reducing the critical path and increasing the clock frequency. Then, we study and contrast "latency" and "cycle-time" (a.k.a., initiation interval). We give examples featuring a water pipe, exam grading, and a computer network with servers thousands of miles apart. We later introduce "retiming", which balances cycle-time. Then, we introduce "functional pipelining", which then leads us to "software pipelining". Eventually, the student understands that pipelining is a very high-level concept and simply means that we initiate (hence overlap) a new instance of a repetitive computation without waiting for the previous instance to end.

Later, we study "stallable pipelines" and the fact that a processor's pipeline needs to support stalls. This brings us to stalls that result from data and control hazards (as well as those due to cache misses), which are the topics of the following subsections.

In our computer architecture course, we also make the connection between pipelining and state machine design. For a simple CPU implementation, the number of states for different instructions can be different. However, the first thing we do for a pipelined CPU implementation is to implement all instructions with the same number of states. In doing that for a particular instruction, the trick is to do nothing in states that do not serve any purpose for that instruction. The minimum number of states we use for the pipelined implementation of VSCPU is 4 states. That is because there are a maximum of 4 memory accesses in VSCPU instructions and every instruction requires a minimum latency of 2 cycles. As for the 4 memory accesses, we have one read for the instruction, one read for operand A, one read for operand B, and one write for the result to be deposited in address A. This leads us into the issue of a multi-port memory. At steady state, in a pipeline with no stalls, there is a need for 3 read ports and 1 write port. This can be implemented 3 double-port block RAMs of an FPGA. In such discussions, we also visit the need for a host interface.

About 50% of the papers [30]–[32], [34], [35], [37], [38], [43], [44], [47], [52], [54], [56]–[59], [64] support teaching pipelining mechanism and about 90% of the textbooks [4]–[14], [16]–[18], [20]–[28] cover pipelining mechanism in detail.

O. Forwarding

Read-after-write data dependencies (i.e., data hazards) between consecutive instructions can cause the pipeline to stall. Depending on the depth and exact sequence of steps in the pipeline, there can be a data dependency problem between instructions that are apart. In our computer architecture course, we briefly touch on "Forwarding" and pictorially explain how the pipeline can be kept running without stalls in the existence of data dependencies between neighboring instructions. However, we deal with an HDL implementation of Forwarding in more advanced courses.

Forwarding is mentioned in about 25% of the papers [30], [34], [35], [37], [43], [44], [47], [52], [58], while about 50% of the textbooks [4], [5], [8], [10]–[13], [16], [21], [23]–[25], [28] cover forwarding mechanism.

P. Branch Prediction

When a branch instruction (or an indirect jump instruction as in VSCPU) enters the pipeline, its outcome and hence the next instruction is not known until the later stages of the pipeline. In such case, the straight-forward approach would be to stall the pipeline until the branch instruction's target (i.e., address of the next instruction to be executed) is resolved. Unlike most other computer architecture courses, we also look into implementations of stallable pipeline besides "Branch Prediction" approach, which keeps the pipeline running.

Although a stallable pipeline is an acceptable approach, it wastes clock cycles and reduces the overall IPC of the processor. Instead, the processor could continue to fetch instructions by guessing the outcome of the branch instruction.

The simplest approach would be to continue with the next instruction (in the address right after the current instruction) and jump to the target, once the branch condition's value and the target are known. At the point these are resolved and it turns out that the branch is taken, the instructions executed after the branch have to be undone. If the pipeline is ordered such that a branch instruction is fully resolved before the "write-back stage" of the pipeline, where the result is written back, then the undo mechanism is simple. It requires modifying the `PC` and canceling (i.e., invalidating) the instructions in the pipeline stages before the pipeline stage where branch resolution is complete. In cases where the branch is fully resolved after write-back, complex register renaming schemes are required.

Today, modern micprocessors employ dynamic and complex branch predictors instead of static and simple ones in order to lower the misprediction rate. Branch predictors are placed in the "Instruction Fetch Unit" of a processor, and they can be quite complex and hence making the fetch unit one of the most complex pipeline stages.

This topic is only covered in about 15% of the papers [29], [35], [43], [51], [59], while about 80% of the textbooks [4]–[8], [10]–[14], [16], [17], [21]–[28] cover the topic of branch prediction.

Q. Out-of-Order and Multiple Issue

In our computer architecture course, we do not discuss this dimension much. However, we touch on it when we discuss the topic of "Delay Slots" in branch prediction. Although delay slots are usually filled at compile-time, they could be filled in run-time as well.

Out-of-order execution normally refers to reordering of executions by the processor core at run-time. Any bubbles (i.e., stalls) that result in the pipeline due to hazards or cache misses can be filled by keeping a queue of instructions in a buffer and identifying those instructions that are ready to be deployed.

Although "Multiple Issue" is a separate topic, it is looked at together with out-of-order execution, as they are both employed in superscalar processors. Multiple issue is also employed in Very Long Instruction Word (VLIW) processors. The asymptotic value of IPC for a regular RISC processor is 1, however, multiple-issue can achieve higher IPC.

These are both advanced topics, which should be reserved for advanced architecture courses, however, we at least mention them and create some awareness. Multiple issue for ex. requires instructions to be fetched in groups and introduces an extra complexity when the group of instructions fetched has a branch instruction. There is additional complexity when a branch target points to the middle of a group. As for out-of-order execution, in some approaches, instruction execution is out-of-order but instructions are retired (i.e., write-back step is completed) in-order.

Only one paper [38] covers this topic, while about 70% of the textbooks [4]–[8], [10]–[14], [16], [17], [20]–[24], [26] cover the topic.

R. CPU Customization

Customizing a CPU can be useful in situations where the same program runs on the CPU indefinitely and performance, area, and/or power consumption matters.

For ex., if the instruction set of a processor does not include a multiply instruction but the application contains too many multiplication operations, the processor will spend a lot of time running a subroutine that implements multiplication. Consequently, there will be a huge degradation in performance and power consumption with respect to a processor that contains multiply instruction. By modifying the instruction set of the processor to add a multiply instruction, one can both save power and increase the performance of the application. The impact on this on area can be in either direction as ALU area increases but the area of program memory is reduced since multiply is moved from software to hardware.

On the other hand, in some applications unused instructions can be eliminated thus making the ALU smaller with no impact on program memory. In the case of rarely used instructions, they can be moved to software from hardware, thus making the ALU smaller but program memory bigger.

A customized microcontroller is easy to design and can be frequently utilized in FPGA design.

This topic is covered in about 6% of the papers [43], [56], and interestingly, none of the textbooks discuss CPU customization.

IV. Student Evaluation

In the context of this research, we did an exam to evaluate the students' performance and also conducted a survey to evaluate the students' learning and perception of our VSCPU based computer architecture education course. The exam and survey were carried out on 64 students of Computer Engineering in Yeditepe University that were enrolled CSE224-Introduction to Digital Systems course in Spring 2020.

We asked students to answer the following types of questions within the exam:

- **VSCPU assembly programming:**
  - given a C program, write its corresponding VSCPU assembly code.
  - write an assembly program that solves a particular problem (e.g., bubble sort).
- **Memory tracing of a running program on VSCPU:** fill in the table to show the changes in memory for each instruction in a given program.
- **Design of VSCPU in Verilog HDL:** design VSCPU state machine that could execute a particular subset of instructions.

Figure 8 shows the average percentage grade for each type of question. As can be seen from the results, students performed well on the exam with an average score of at least 80% on all types of questions.

On the other hand, we asked students to answer a survey in the context of term project. Survey questions are given in Table II and the results of the survey are given in Figure 9, 10 and 11, respectively. The following is a summary of the survey responses:

1) Almost every student thinks that his/her effort in the term project was at least satisfactory.
2) About 85% of the students think that they gained at least sufficient confidence on the design of a CPU at the end of the project.

TABLE II
STUDENT SURVEY QUESTIONS.

Question
Q1. Level of effort you put into the term project (Poor/Satisfactory/Excellent)
Q2. Contribution to learning CPU Design (Poor/Satisfactory/Excellent)
a. Level of skill/knowledge at the start of project
b. Level of skill/knowledge at the end of project
c. Level of skill/knowledge required to complete the project
d. Contribution of project to your skill/knowledge
Q3. Tools (Poor/Satisfactory/Excellent)
a. Instruction set simulator
b. C compiler
c. Assembler
d. User manual
Q4. Project content (Strongly disagree/Disagree/Neutral/Agree/Strongly agree)
a. Learning objectives were clear
b. Project content was organized and well planned
c. Project workload was appropriate
d. Project organized to allow all students to participate fully
Q5. When you compare VSCPU to other CPUs with respect to learning computer architecture (select all that apply)
a. VSCPU ISA is simpler than other CPUs
b. VSCPU makes understanding how pipelining, interrupt, etc. mechanisms work on a CPU easier
c. Writing programs in assembly language for VSCPU requires less effort than other CPUs
d. Implementing VSCPU on hardware (i.e., FPGA) and running a program on it help me better understand the course
Q6. What aspects of this project were most useful or valuable? (select all that apply)
a. Design and implementation of a CPU from scratch
b. Improving practice of writing assembly programs
c. Understanding how a CPU communicates with peripherals
d. Understanding how a C program translates (i.e., compiles) to instructions of CPU
Q7. How would you improve this project? (select all that apply)
a. Customizing VSCPU by adding new instructions (e.g., multiply-add, division, etc.)
b. Writing programs with a subset of VSCPU instructions (e.g., using only NAND, ADD, and BZJ)

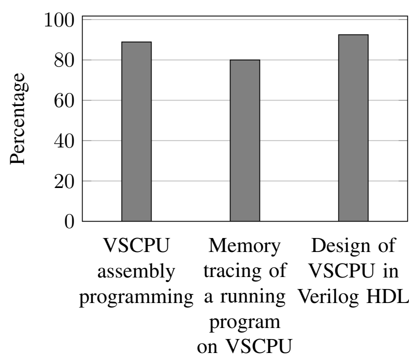


Fig. 8. Exam results.

3) About 70% of the students agree that the project content and workload was satisfactory.
4) About 70% of the students find VSCPU most valuable to design a CPU from scratch.
5) About 90% of the students feel at least comfortable with using the tools.
6) About 56% of the students think that VSCPU is a contributing factor with respect to other CPUs in learning computer architecture.
7) About 70% of the students find VSCPU useful to design and implement a CPU, about 58% of them think that VSCPU ISA improves their ability to write assembly programs, about 65% of them think that VSCPU does help them to understand how a CPU accesses I/O devices, and 19% of them find VSCPU useful to understand how a compiler works.

Based on these feedback from the students, we can conclude that using VSCPU as a course material was effective in motivating them to learn about how a processor and computer works. However, we also noticed that some students didn't find some part of our VSCPU based course as a contributing factor to learn computer architecture (especially coding in assembly language). For this reason, we are planning to modify the content of our course in order to improve students' performance and better meet their expectations.

V. CONCLUSIONS

In this paper, we analyzed 25 textbooks on computer architecture and 38 papers on computer architecture educa-

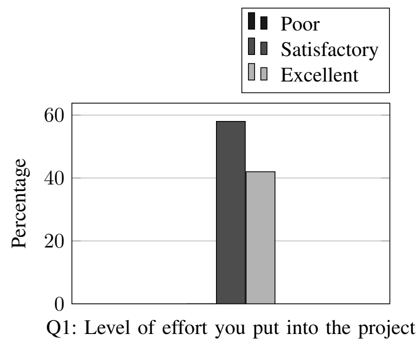


Fig. 9. Survey results - part I.

tion from the perspective of 18 critical topics covered in our references. Along the way, we also explained our own approach on computer architecture teaching, which is based on our own processor called VSCPU.

Below is a sorted list of the 18 topics from the one covered most in our references to the one covered the least (with the average percentage of textbook and paper coverage in parentheses):

- Assembly Coding (100%)
- Pipelining and Hazards (70%)
- Indirect Memory Access (63%)
- Memory Hierarchy (60%)
- Memory-Mapped I/O (59%)
- Granularity of Simulation Environment (56%)
- Virtual Memory (55%)
- Branch Prediction (48%)
- Forwarding (38%)
- Out-of-Order and Multiple Issue (37%)
- Working CPU on FPGA (36%)
- Detailed Explanation of Harvard vs von Neumann (30%)
- Memory Banking (25%)
- Tradeoffs in Instruction Design (15%)
- Self-Modifying Code (13%)
- Instruction Set Completeness (12%)
- One-Instruction CPU (5%)
- CPU Customization (3%)

On the other hand, we deem the below topics as basic topics, which must be covered even in an introductory computer architecture course:

- Assembly Coding
- Instruction Set Completeness
- One-Instruction CPU
- Tradeoffs in Instruction Design
- Memory-Mapped I/O
- Indirect Memory Access
- Detailed Explanation of Harvard vs von Neumann
- Self-Modifying Code
- Granularity of Simulation Environment
- Memory Banking

Unfortunately, the following are covered in 30% or fewer of the textbooks and papers:

- One-Instruction CPU (5%)
- Instruction Set Completeness (12%)
- Self-Modifying Code (13%)
- Tradeoffs in Instruction Design (15%)
- Memory Banking (25%)
- Detailed Explanation of Harvard vs von Neumann (30%)

Most computer architecture courses are based on microprocessors, and they quickly move into advanced topics such as pipelining, forwarding, branch prediction, cache, virtual memory without spending enough time on the above basic concepts.

Out of the above basic concepts, we see “Instruction Set Completeness” as the top priority since it is what separates a general-purpose computer from a non-general-purpose computer. In discussing instruction set completeness, the question of a `NAND` instruction being sufficient or not always comes up as it is taught to be a gate sufficient to implement all logic functions (remember your basic logic design course). Later, the discussion naturally leads a computer architecture class to “Tradeoffs in Instruction Design”. While the core of a processor implements the instructions, the other half of a processor is the memory, and a discussion of Von Neumann versus Harvard architectures is where we usually end up. Such discussion should include self-modifying code even though it is against the current practice to write such code. Then, there should be a discussion of what the memory size should be and what we need to do if we exceed the size dictated by the address size in the instructions. That is the topic of “Memory Banking”.

Last but not least, we need to state that implementing a CPU on FPGA adds extraordinary value to a computer architecture course although it is considered here to be an advanced topic. At that point, we see that implementing customized CPUs on FPGA for particular applications is relatively trivial once you have a working CPU on FPGA.

Acknowledgment

This work was supported by TÜBİTAK under grant no. 117E090.

References

[1] J. D. Carpinelli and T. Zaman, “Instructional Tools for Designing and Analysing a Very Simple CPU,” *Int. Journal of Electrical Engineering & Education, SAGE*, vol. 43, pp. 261–270, 2006.
[2] A. Yıldız, H. F. Ugurdag, B. Aktemur, D. İskender, and S. Gören, “CPU design simplified,” in *Proc. of Int. Conf. on Computer Science and Engineering (UBMK)*. IEEE, 2018, pp. 630–632.
[3] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local Algorithms for Document Fingerprinting,” in *Proc. of Int. Conf. on Management of Data (SIGMOD)*. ACM, 2003, pp. 76–85.
[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Elsevier, 2017.
[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.
[6] W. Stallings, *Computer Organization and Architecture: Designing for Performance*. Pearson, 2003.
[7] A. S. Tanenbaum, *Structured Computer Organization*. Pearson, 2016.
[8] S. Harris and D. Harris, *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2015.
[9] J. G. De Lamadrid, *Computer Organization: Basic Processor Structure*. Chapman and Hall/CRC, 2018.
[10] A. Clements, *Computer Organization & Architecture: Themes and Variations*. Cengage Learning, 2013.
[11] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge University Press, 2012.
[12] V. C. Hamacher *et al.*, *Computer Organization and Embedded Systems*. McGraw-Hill, 2012.
[13] D. Comer, *Essentials of Computer Architecture*. Chapman and Hall/CRC, 2017.

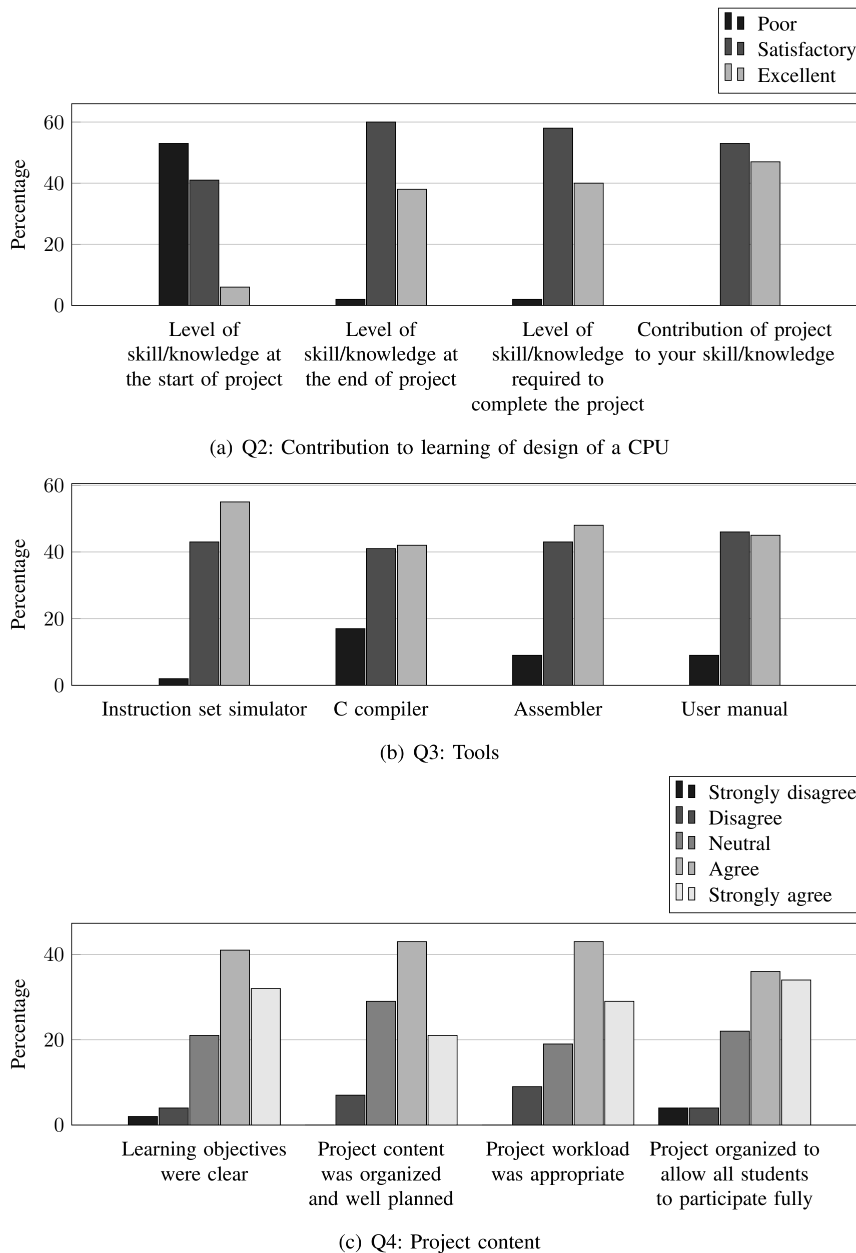


Fig. 10. Survey results - part II.

[14] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*. Jones & Bartlett, 2014.

[15] N. Nisan and S. Schocken, *The Elements of Computing Systems: Building A Modern Computer from First Principles*. MIT Press, 2005.

[16] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of Computer Organization and Architecture*. John Wiley & Sons, 2005.

[17] A. S. Berger, *Hardware and Computer Organization*. Newnes, 2005.

[18] I. East, *Computer Architecture and Organization*. CRC Press, 2004.

[19] H. A. Farhat, *Digital Design and Computer Organization*. CRC Press, 2003.

[20] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective*. Springer, 2003.

[21] S. P. Dandamudi, *Fundamentals of Computer Organization and Design*. Springer, 2003.

[22] J. Y. Hsu, *Computer Architecture: Software Aspects, Coding, and Hardware*. CRC Press, 2001.

[23] S. G. Shiva, *Computer Organization, Design, and Architecture*. CRC Press, 2007.

[24] H. G. Cragon, *Computer Architecture and Implementation*. Cambridge University Press, 2000.

[25] M. M. Mano, *Computer System Architecture*. Prentice Hall, 1993.

[26] M. Balch, *Complete Digital Design: A Comprehensive Guide to Digital Electronics and Computer System Architecture*. McGraw-Hill, 2003.

[27] P. Juola, *Principles of Computer Organization and Assembly Language*. Pearson, 2006.

[28] B. Chalk, A. T. Carter, and R. W. Hind, *Computer Organisation and Architecture: An Introduction*. Macmillan Int. Higher Education, 2017.

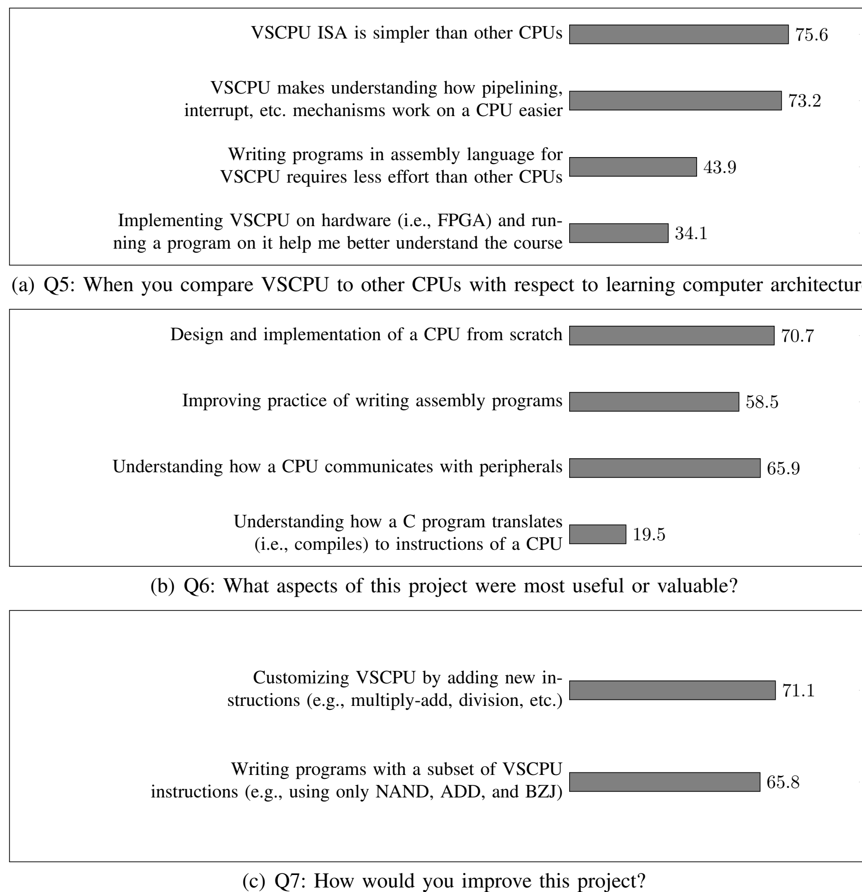
[29] A. Clements, "ARMs for the poor: Selecting a processor for teaching computer architecture," in *Proc. of Frontiers in Education Conf. (FIE)*. IEEE, 2010, pp. T3E/1–6.

[30] V. Rubio and J. Cook, "A FPGA implementation of a MIPS RISC processor for computer architecture education," Master's thesis, New Mexico State University, Las Cruces, New Mexico, 2004.

[31] C. M. Kellett, "A project-based learning approach to programmable logic design and computer architecture," *IEEE Transactions on Education*, vol. 55, pp. 378–383, 2012.

[32] B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic, "A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization," *IEEE Transactions on Education*, vol. 52, pp. 449–458, 2009.

[33] K. Nakano, K. Kawakami, K. Shigemoto, Y. Kamada, and Y. Ito, "A tiny processing system for education and small embedded systems on the FPGAs," in *Proc. of Int. Embedded and Ubiquitous Computing (EUC)*. IEEE, 2008, pp. 472–479.



(a) Q5: When you compare VSCPU to other CPUs with respect to learning computer architecture:

(b) Q6: What aspects of this project were most useful or valuable?

(c) Q7: How would you improve this project?

Fig. 11. Survey results - part III.

[34] Y. Li and W. Chu, "Aizup-a pipelined processor design and implementation on Xilinx FPGA chip," in *Proc. of Symp on FPGAs for Custom Computing Machines (FCCM)*. IEEE, 1996, pp. 98–106.

[35] P. Bulić, V. Guštin, D. Šonc, and A. Štrancar, "An FPGA-based integrated environment for computer architecture," *Computer Applications in Engineering Education, Wiley*, vol. 21, pp. 26–35, 2013.

[36] H. Oztekin, F. Temurtas, and A. Gulbag, "BZK.SAU.FPGA10.1: A modular approach to FPGA-based micro computer architecture design for educational purpose," *Computer Applications in Engineering Education, Wiley*, vol. 22, pp. 272–282, 2014.

[37] A. K. Uht, J.-C. Lo, Y. Sun, J. C. Daly, and J. Kowalski, "Building real computer systems," *IEEE Micro*, vol. 20, pp. 48–56, 2000.

[38] A. Clements, "Computer architecture education," *IEEE Micro*, vol. 20, pp. 10–12, 2000.

[39] M. D. L. Á. Cifredo-Chacón, Á. Quirós-Olozábal, and J. M. Guerrero-Rodríguez, "Computer architecture and FPGAs: A learning-by-doing methodology for digital-native students," *Computer Applications in Engineering Education – Wiley*, vol. 23, pp. 464–470, 2015.

[40] A. Hernandez Zavala, O. Camacho Nieto, J. A. Huerta Ruelas, C. Domínguez, and R. Arodí, "Design of a general purpose 8-bit RISC processor for computer architecture learning," *IPN Computación y Sistemas*, vol. 19, pp. 371–385, 2015.

[41] D. Šulík, M. Vasilko, and P. Fuchs, "Design of a RISC microcontroller core in 48 hours," *Journal of Electrical Engineering*, vol. 52, pp. 171–176, 2001.

[42] J. Djordjevic, B. Nikolic, and A. Milenkovic, "Flexible web-based educational system for teaching computer architecture and organization," *IEEE Transactions on Education*, vol. 48, pp. 264–273, 2005.

[43] J. Gray, "Hands-on computer architecture: teaching processor and integrated systems design with FPGAs," in *Proc. of Workshop on Computer Architecture Education (WCAE)*. ACM, 2000, pp. 17–24.

[44] B. Hatfield and L. Jin, "Improving learning effectiveness with hands-on design labs and course projects for the operating model of a pipelined processor," in *Proc. of Frontiers in Education Conf. (FIE)*. IEEE, 2010, pp. F1E/1–6.

[45] S. Yamazaki, T. Satoh, T. Jiromaru, N. Tachi, and M. Iwano, "Instructional Design of a Workshop "How a Computer Works" Aimed at Improving Intuitive Comprehension and Motivation," in *Proc. of IIAI International Conference on Advanced Applied Informatics*. IEEE, 2014, pp. 338–341.

[46] V. Guštin and P. Bulić, "Learning computer architecture concepts with the FPGA-based "Move" microprocessor," *Computer Applications in Engineering Education, Wiley*, vol. 14, pp. 135–141, 2006.

[47] S. L. Harris, D. M. Harris, D. Chaver, R. Owen, Z. L. Kakakhel, E. Sedano, Y. Panchul, and B. Ableidinger, "MIPSfpga: using a commercial MIPS soft-core in computer architecture education," *IET Circuits, Devices & Systems*, vol. 11, pp. 283–291, 2017.

[48] D. Ellard, D. Holland, N. Murphy, and M. Seltzer, "On the design of a new CPU architecture for pedagogical purposes," in *Proc. of Workshop on Computer Architecture Education (WCAE)*. ACM, 2002, pp. 6–12.

[49] R. Brennan and M. Manzke, "On the introduction of reconfigurable hardware into computer architecture education," in *Proc. of Workshop on Computer Architecture Education (WCAE)*. ACM, 2003, pp. 15–21.

[50] H. Oztekin, F. Temurtas, and A. Gulbag, "On the improvement of the teaching quality and learning effectiveness in the computer organization course through FPGA and modular centered microcomputer design," *Computer Applications in Engineering Education, Wiley*, vol. 26, pp. 1825–1840, 2018.

[51] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A binary instrumentation tool for computer architecture research and education," in *Proc. of Workshop on Computer Architecture Education (WCAE)*. ACM, 2004, pp. 22–29.

[52] J. H. Lee, S. E. Lee, H. C. Yu, and T. Suh, "Pipelined CPU design with FPGA in teaching computer architecture," *IEEE Transactions on Education*, vol. 55, pp. 341–348, 2012.

[53] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an FPGA," in *Proc. of Int. Conf. on Parallel and Distributed Systems (ICPADS)*. IEEE, 2008, pp. 723–728.

[54] W. Wolf, "Rethinking embedded microprocessor education," in *Proc. of ASEE Annual Conf. & Expo*, 2001, pp. 861–866.

[55] J. Qian, R. Wang, S. Shi, Y. Zhu, and Z. Xie, "Simplifying and integrating experiments of hardware curriculums," in *Proc. of Int. Conf. on Computer Science and Information Technology (ICCSIT)*. IEEE, 2010, pp. 610–614.

[56] V. Bonato, R. Menotti, E. Simões, M. M. Fernandes, and E. Marques, "Teaching embedded systems with FPGAs throughout a computer

science course," in *Proc. of Workshop on Computer Architecture Education (WCAE)*. ACM, 2004, pp. 8–14.
[57] W. Zhang and U. Z. Zhang, "Teaching the Introductory Computer Architecture Course with a Systematic View," in *Proc. of Midwest Section Conf. of ASEE*, 2007, pp. 1–10.
[58] V. Angelov and V. Lindenstruth, "The educational processor Sweet-16," in *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE, 2009, pp. 555–559.
[59] A. Clements, "The undergraduate curriculum in computer architecture," *IEEE Micro*, vol. 20, pp. 13–21, 2000.
[60] J. D. Carpinelli, "The very simple CPU simulator," in *Proc. of Frontiers in Education (FIE)*. IEEE, 2002, pp. T2F/11–14.
[61] R. Nakamura, Y. Ito, and K. Nakano, "TinyCSE: Tiny Computer System for Education," in *Proc. of Int. Symp. on Computing and Networking*. IEEE, 2013, pp. 639–641.
[62] Y. Chen and P. Cao, "Toy CPU: An innovative curriculum design," in *Proc. of Int. Conf. on Computer Science & Education (ICCSE)*. IEEE, 2012, pp. 1690–1693.
[63] F. Mavaddat and B. Parhami, "URISC: The ultimate reduced instruction set computer," *Int. Journal of Electrical Engineering Education, SAGE*, vol. 25, pp. 327–334, 1988.
[64] X. Wang, "Using FPGA-based configurable processors in teaching hardware/software co-design of embedded multiprocessor systems," in *Proc. of Int. Conf. on Microelectronic Systems Education (MSE)*. IEEE, 2011, pp. 114–117.
[65] L. Ribas-Xirgo, "Yet another simple processor (YASP) for introductory courses on computer architecture," *IEEE Transactions on Industrial Electronics*, vol. 57, pp. 3317–3323, 2010.
[66] N. G. Tsoutsos and M. Maniatakos, "Investigating the Application of One Instruction Set Computing for Encrypted Data Computation," in *Proc. of Int. Conf. on Security, Privacy, and Applied Cryptography Engineering (SPACE)*. Springer, 2013, pp. 21–37.
[67] ——, "HEROIC: homomorphically EncRypted one instruction computer," in *Proc. of Design, Automation & Test in Europe Conf. & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.
[68] H. Corporaal, "Design of transport triggered architectures," in *Proc. of Great Lakes Symposium on VLSI (GLSVLSI)*. IEEE, 1994, pp. 130–135.
[69] A. Demin, "The NOR machine," In *PragPub magazine*, accessed through https://pragprog.com/magazines/2012-03/the-nor-machine, 2012.

Abdullah Yildiz is a PhD candidate at the Dept. of Computer Engineering, Yeditepe University. He received his BS degree in EE from Uludag University in 2008 and MS degree in EE from Ozyegin University in 2012. His research interests include computer architecture education, digital design and verification, and embedded systems.

Sezer Gören received the BS and MS degrees in EE from Bogazici University and the PhD degree in Computer Engineering from the University of California, Santa Cruz. She was a senior engineer in Silicon Valley from 1998 to 2004 at Syntest, Cadence, Apple, PMC-Sierra, and Aarohi Communications. She is currently a full professor and chair of the Dept. of Computer Engineering, Yeditepe University. Her research interests include reconfigurable computing, design automation, design verification, test, computer arithmetic, vehicular technologies, and embedded systems design.

H. Fatih Ugurdag is a full professor at Ozyegin University. He received his MS and PhD from Case Western Reserve University in EE in 1989 and 1995, respectively. He received his BS in EE with a double major in Physics from Bogazici University in 1986. He did an MS thesis on machine vision and a PhD dissertation on parallel hardware design automation. He worked in the industry in the USA between 1989-2004 at companies such as GE, GM, Lucent, Juniper, and Nvidia as a machine vision engineer, EDA software developer, and chip designer. In late 2004, he joined academia. His research interests include real-time hardware/software design in the areas of video processing, communications, and automotive systems.

Barış Aktemur is a senior software engineer at Intel, Munich. Previously he was an assistant professor of computer science at Ozyegin University. He received his BS in Computer Engineering from Bilkent University in 2003 and MSc and PhD degrees in Computer Science from University of Illinois at Urbana-Champaign in 2005 and 2009, respectively. His research interests include programming languages, compilers and debugging tools, and software engineering.

Taylan Akdogan is the Dean of Faculty of Engineering at Ozyegin University. He received his PhD degree from Massachusetts Institute of Technology (MIT) in Physics in 2003, and BS degrees from Bogazici University in EE and Physics in 1995. He did a PhD dissertation on nucleon-nucleon interactions. His research expertise includes nucleon-nucleon interactions, electromagnetic structure of nucleons and light nuclei, neutrino physics, computational and mathematical physics, data acquisition systems. He contributed to discoveries of both tau neutrino and Higgs boson, two fundamental particles of the Standard Model. He started his academic career as a research scientist at MIT in 2003. He joined the faculty of Bogazici University in 2006 and served as the Dean of Faculty of Arts and Sciences between 2014-2016, then joined Ozyegin University in 2017.