A Practical Methodology to Learn Computer Architecture, Assembly Language, and Operating System

Hiroaki Fukuda[1], Paul Leger[2] and Ismael Figueroa[3]

[1]*Department of Computer Science and Engineering, Shibaura Institute of Technology, 3-7-5 Toyosu, Koto, Tokyo, Japan*

[2]*Escuela de Ingeniería Civil, Universidad Católica del Norte, Coquimbo, Chile*

[3]*Ingeniería en Información y Control de Gestión, Universidad de Valparaíso, Valparaíso, Chile*

Keywords: Educational Methodology, Operating System, Virtual Machine, Assembly Language, Computer Architecture.

Abstract: System-level details, such as assembly language and operating systems, are important to develop/debug embedded systems and analyze malware. Therefore it is recommended to teach every topic of these subjects. However, their learning cost has been significantly increased due to current system complexities. To solve this problem, several visualization techniques have been proposed to help students in their learning process. However, observing only the computer system behaviors may be insufficient to apply it to real systems due to the lack of practical experiences and a comprehensive understanding of system-level details. To address these issues, we propose a novel methodology where students implement a virtual machine instead of using existing ones. This virtual machine needs to execute binary programs that can be run on a real operating system. Through implementing this virtual machine, students improve by experience their understanding of computer architecture, assembly languages, instruction sets, and the role of operating systems. We also provide MMVM that is a virtual machine implementation reference, and can execute the binary programs while showing the internal states of CPU (registers & flags) to users (students) to support their implementation. Finally, this paper reports the education results applying this methodology to 15 students that consist of 3rd-year students and 1st year of master students.

1 INTRODUCTION

For computer science students (shortly students), systems-level details such as computer architecture, assembly languages, and operating systems are still important. However, it is difficult to spend enough time to learn these details due to increasing their complexities and/or appearing on new topics such as Cloud Computing (Armbrust et al., 2010) and Artifical Intelligence (AI) (Russell and Norvig, 2009). Thereby students mainly learn these details from textbooks, acquiring abstract knowledge without any experience. As a result, there might be a big gap when students face real problems (*e.g.,* debugging embedded systems, analyzing computer viruses by binary dump). Some researchers propose visualize within computers such as stored values in registers, memories, and how the stored values will be changed as a program is executed on the computer (Zeng et al., 2009). Visualization techniques are useful because students can understand the inside behavior of computer systems; however, two learning issues appear. The first one is the lack of experience. Even though students understand the correct behavior, they have not learned real skills but knowledge, therefore they may fail with programming and debugging tasks in practice. The second one is the lack of comprehensive understanding. Due to the increase in complexities and specialties of each subject, giving adequate exercises is also difficult. Even though implementing an operating system helps students to understand the difference, it is too difficult because students need to know other subjects such as assembly languages, hardware (*e.g.,* CPU) and a binary format.

To solve the previous two learning issues, we propose a novel methodology that is based on the construction of a virtual machine. In this methodology, we give students a set of exercises (binary programs) that will lead to a virtual machine implementation instead of using existing ones. The virtual machine will execute binary programs that can be run on certain hardware and an operating system. Implementing this virtual machine requires students to emulate existing hardware (*e.g.,* CPU and memory) and un-

Fukuda, H., Leger, P. and Figueroa, I.
A Practical Methodology to Learn Computer Architecture, Assembly Language, and Operating System.
DOI: 10.5220/0009319503330340
In *Proceedings of the 12th International Conference on Computer Supported Education (CSEDU 2020) - Volume 1*, pages 333-340
ISBN: 978-989-758-417-6

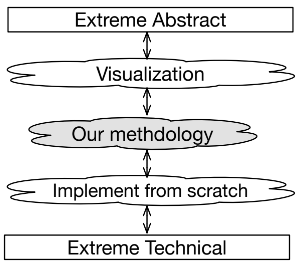

Figure 1: The position of our methodology between abstract and technical.

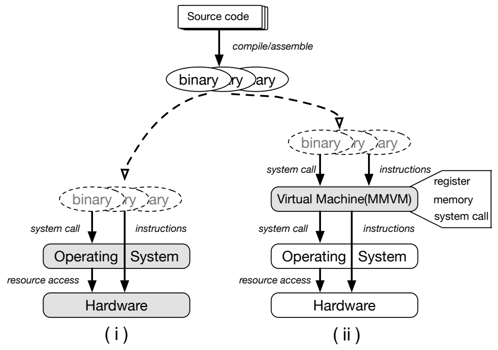

Figure 2: Connections among binaries to be executed, Virtual Machine and Operating System in our methodology.

derstand instruction sets in addition to assembly languages. Besides, we assume these binary programs are executed on userland, meaning that the virtual machine needs to emulate system calls that are usually provided by an operating system. With this methodology, students can learn systems-level details with experiences and connections among mainly computer architecture, assembly language and operating systems. For this methodology, we provide a tool called MMVM that can execute binary programs and show the internal states (registers & flags) to users (*i.e.,* students) to support a virtual machine implementation. Students can start from small binary programs that have only limited instruction sets and system calls to large programs that have many instruction sets and system calls with a step-by-step manner. As shown in Figure 1, our methodology is located in the middle between abstract and technical learning. On the one hand, our methodology offers students to implement their virtual machines instead of just observing behavior (abstract learning). On the other hand, the implementation of a virtual machine is simple using MMVM because students do not need to implement a complete operating system or virtual machine (technical learning). We show the educational results applying this methodology to 15 students as a preliminary evaluation.

The rest of this paper is organized as follows. Section 2 explains the overview of our methodology. Section 3 provides learning processes in our methodology with a tool (MMVM). Section 4 gives a preliminary evaluation and Section 5 describes related work. Finally, Section 6 gives the conclusion and future work.

2 METHODOLOGY OVERVIEW

This section explains our methodology where students can learn systems-level details in practice.

As we can see in Figure 2-(i), a set of source code will be compiled and assembled by a compiler and an assembler, generating a binary program. This binary program consists of instruction sets and system calls provide by a certain CPU and an operating system. Thereby the program is executed on the CPU directly and uses system resources via system calls. In our methodology, as shown in Figure 2-(ii), we offer to execute the program on their virtual machines that need to emulate an architecture and instruction sets of the original CPU, and also system calls of the original operating system (*i.e.,* both are shown in Figure 2-(i)). In our methodology, students have to implement their virtual machines, which is corresponding to Virtual Machine in Figure 2-(ii), resulting in partial learning about computer architecture, instruction sets, assembly languages, and operating system because their virtual machines have to emulate this behavior. We also give MMVM which can execute the binaries with showing internal states to support students' implementation.

There have been two categories to learn system-level details: *1*) preparing simulators that provide simplified execution environments (CPU and/or programming languages) of a program, and *2*) using real systems. Each category has advantages and disadvantages. Using simulators makes learning easier; however, simulators are *imitations*, which demotivate students to learn (Gondow et al., 2010). In contrast, using real systems is difficult; however real systems motivate students to learn, and knowledge and experiences will be widely applicable. Considering these advantages and disadvantages, we have chosen using a real system (Gondow et al., 2010) such as real instruction sets, an assembly language, and an operating system. Taking these points into account, we have selected a CPU (computer architecture) and the corresponding operating system. Next, we explain both selections.

2.1 CPU Selection

A variety of CPUs for computers are currently available including a series of x86 from Intel, a series of CPUs from AMD and ARM. On the one hand, the mainstream of usage in current computers is 64-bit computing, implying a difficult to learn for students from the beginning. On the other hand, we can also choose 8-bit old CPUs for the sake of education; however, the acquired knowledge might be useless when students face problems related to current CPU architectures. As a conclusion of this selection, we choose the 8086 16-bit CPU from Intel because of the following reasons. Firstly, a majority of people use a series of x86 CPUs then this trend continue until x86_64. Therefore we think the acquired knowledge and experiences of 8086 can be a base for students to learn about current CPUs such as x86_64 and ARM. Secondly, we consider the bit length of CPUs is important because: *a*) values consist of many bits (*e.g.,* 64bit) is difficult to trace when we find and fix bugs in implementations of emulators, and *b*) 8bit CPUs such as 8080 has a limited number of instructions, meaning that knowledge from 8080 is not enough to apply current CPUs.

2.2 Operating System Selection

As requirements for the selection of an operating system in our methodology, we identify three. Firstly, as students need to implement virtual machines that emulate the system calls, they should refer source code to emulate each system call correctly; meaning that a set of source code for a certain operating system needs to be open access. Secondly, the size and complexity of an operating system need to be small and simple enough to understand its behavior. Finally, hopefully, the acquired knowledge and experiences, via reading source code, can be a base for students to tackle current and industrial operating systems.

Based on operating system requirements, we thought that UNIX V6 could fulfill these requirements because it is considered as an origin of BSD Unix and its total lines of code are around 10,000. Besides, we can refer to the book (John Lions, 1977) that explains pieces of code in detail. Unfortunately, UNIX V6 was designed for PDP 11, therefore it cannot run on 8086 that we choose as a CPU. As a consequence, we chose the minix2 operating system (Tanenbaum and Woodhull, 2005). As an explanation, minix2 is designed for multiple CPUs including 8086 and it can be considered as an operating system for education, meaning that it is open source and lines of kernel code is around 30,000. In addition, minix2 follows the POSIX, meaning that the kind and behavior of its system calls are the same as current operating systems that follow POSIX.

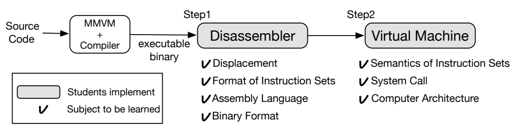

Figure 3: Learning Process.

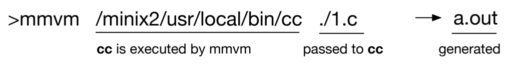

Figure 4: The usage example of MMVM.

3 LEARNING PROCESSES

This section explains how students can learn system-level details using our methodology as shown in Figure 3. We divide the entire educational processes into two steps: *Step1*) disassembler implementation and *Step2*) virtual machine implementation. This is because the disassembler implementation is almost a subset and simplified version of a virtual machine implementation. As Figure 3 shows, each step gives detailed subjects related to system-level details such as Binary Format, Assembly Languages, and System Call. Besides, in this methodology, we use two kinds of documents: 8086 16Bit Microprocessor (Intel, 1990), and Intel 64/IA-32 Architecture Software Developer's Manual (Intel, 2018). This is to confirm the available instruction sets and compile source code using MMVM to the minix2's binary, and thus, observing the correct internal states. Figure 4 shows how to use MMVM and compile source code. As MMVM can run binaries for minix2, we use cc distributed by minix2 as binary to compile source code.

3.1 Disassembler Implementation

When students start implementing their disassembler, they should learn the file format such as Executable and Linkable Format (ELF) (Committee, 1995) because they have to run binary programs which can be run on minix2. Since minix2 adopted a.out as a executable file format, students need to understand the a.out format, then extract text (*i.e.,* executable code) from the binary program. As shown in Figure 5, the a.out format consists of a header of 32 bytes of length, text and data, whose lengths are recorded as 4 bytes in the header respectively. Therefore students start from analyzing the header. We think this process is suitable because this task only requires the knowledge of

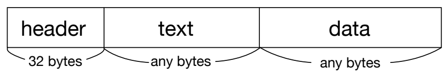

Figure 5: The binary format of a program in minix2.

the a.out format shown in Figure 5; however students have to analyze binary, resulting in a good introduction to handle binaries directly. Since almost all students have never analyzed binary programs manually, they may think to analyzing binary programs seems to be difficult. Therefore extracting text from an a.out format can break their way of thinking.

After extracting the text part, students start analyzing text part using the document (Intel, 1990). As we will explain in the next section, students can start from a small binary program. In addition, students can see the disassemble result using MMVM with an option (see Listing 6), implying students can easily find their mistakes comparing each result.

Compared to virtual machine implementation, disassembler implementation is easier because the result is always the same if the input binary program is the same. Therefore students can complete their disassembler implementation. When students finish this implementation, we can consider that students understand how to read binary such as displacement and variable-length operators following the specification, which is an initial step for virtual machine implementation.

3.2 Virtual Machine Implementation

The virtual machine implementation is an extension of the disassembler implementation because it also requires analyzing the binary file formatted by a.out to extract text and data. In addition to the tasks required by disassembler implementation, students need to prepare registers, flags, memories and extract data from the binary program. Since 8086 CPU consists of 14 registers including the flag register and program counter, the virtual machine must prepare them. Besides, the virtual machine should also allocate memories to store programs and data, the size of which is 0x10000 because 8086 is 16 bit CPU, meaning that a register can represent from 0x0000 to 0xffff.

We show an example implementation of a virtual machine in Listing 1. In Listing 1, we define necessary registers (Line1) and keep memories for text and data (Line2). The run represents the execution of CPU, then fetch_and_decode in Line 6 returns an operator by fetching and decoding an instruction from memory. The opcode represents a kind of operator, therefore the behavior of each operator is defined using switch-case statements.

```
1   unsigned short AX, BX...
2   unsigned char memory[0x10000] = {0};
3   ......
4   void run() {
5     while(true) {
6       opcode = fetch_and_decode()
7       switch(opcode) {
8         case 0x20: // interrupt
9           system_call(BX); // emulate system calls
10        case 0x88: // mov instruction
11          ....
12  }}}
```

Listing 1: An example of virtual machine implementation.

3.2.1 Instruction Sets

After loading the text and data on corresponding memory, the virtual machine starts executing the binary program. Since an instruction cycle (*i.e.,* fetch-decode-execute cycle) is the most basic operational process of a computer, the virtual machine has to emulate this cycle. In this instruction cycle, the emulations of fetch and decode have been done partially in the disassemble implementation. Therefore students can start virtual machine implementation without much stress. The virtual machine generally consists of an infinite loop that involves a large switch-case branches. Each case corresponds to each instruction, therefore students will implement each instruction's behavior referring to the document (Intel, 2018). During this implementation, students can check the correct behavior of each instruction using an option in the MMVM execution, showing all values stored in registers at every execution step. Thereby, students can find and fix their mistakes easily. Note that students do not need to implement all instructions at a time but implement requisite instructions involved in the binary program to be executed.

```
typedef struct {
  int m_source; // caller information
  int m_type;   // kind of system calls
  union {
    mess_1 m_m1;
    mess_2 m_m2;
    mess_3 m_m3;
    mess_4 m_m4;
    mess_5 m_m5;
    mess_6 m_m6;
  } m_u;    // for arguments
} Message;
```

Listing 2: The definition of struct message.

3.2.2 System Calls

As the binary programs that are executed by a virtual machine contain system calls to use system resources such as I/O, students have to interrupt and capture system call invocations and provides corresponding results to the program. The 0xcd instruction represents an interruption in 8086 and 0x20 represents the interruption for system calls in minix2. Therefore the 0xcd20 represents the system call interruption in the current environment (*e.g.,* minix2 running on 8086). In minix2, arguments of a system call are passed to the kernel using message structure defined in "/usr/include/minix/type.h" as shown in Listing 2. Then the address of the message is stored in bx register. Thereby we can access all requisite information using message. After a system call ends, its result is stored in ax register, restarting the interrupted program execution. To sum up, implementing system calls needs the following steps:

1. Find the 0xcd20 instruction.
2. Detect system calls and its arguments using message the address of which is stored in bx register.
3. Emulate the corresponding system call behavior.
4. Store the execution result in ax register.
5. Restart the program.

Note that, system calls supported by minix2 follow POSIX (Oracle Corporation and/or its affiliates, 2010); therefore we can easily find out the behavior from documents and/or using them in current operating systems that follow POSIX.

4 PRELIMINARY EVALUATION IN EDUCATION

This section discusses our experience using our educational methodology with MMVM in under/graduate students in Shibaura Institute of Technology in Japan.

4.1 Educational Methodology Details

We have applied our methodology to 15 students: 10 from 3rd-year of undergraduate and 5 from 1st year of master. Although students' skills vary, all students at least passed one programming course and can use one programming language. Besides, we do not require any special knowledge and skills about low-level details such as binary data, assembly language, and machine instruction sets. In this methodology, we firstly give two simple assembly code for students to get used to binary programs and disassembling. Listing 3 shows the simplest assembly code that we prepare. Apart from the data section (*i.e.,* after 6 line), this program has only 4 lines. Besides, Listing 4 shows the corresponding disassemble result. As we can see in this figure, the disassemble result from line 1 to 4 easily seems to be corresponding to from line 1 to 4 in Figure 3. Moreover, observing the disassembled result without any document, it can intuitively imagine that bb is corresponding to mov bx, and the next two bytes should be flipped (1000 $\rightarrow$ 0010) and might be the second arguments of mov instruction.

```
1  mov bx, #message
2  int 0x20
3  mov bx, #exit
4  int 0x20
5
6  .sect .data
7  message: .data2 1, 4, 1, 6, 0, hello, 0, 0
8  exit: .data2 1, 1, 0, 0, 0, 0, 0, 0
9  hello: .ascii "hello\n"
```

Listing 3: The first test code written by assembly language.

After having intuition from the disassembled result, students start to implement their disassemblers that will show the same result of MMVM from the same input binary program. We think that students who finish implementing this disassembler correctly (which just accepts only two kinds of binary programs that we provide) have understood the a.out format by analyzing its header, and the basic structure of a disassembler such as an infinite loop and switch-case statements.

```
1  0000: bb0000        mov bx, 0000
2  0003: cd20          int 20
3  0005: bb1000        mov bx, 0010
4  0008: cd20          int 20
```

Listing 4: The disassemble result of Listing 3 using MMVM.

4.1.1 Disassembler Implementation

After disassembling the previous two simple binary programs, students start from Step1 shown in Figure 3.

We give the document (Intel, 1990) and explanations on how to interpret it. Although the document seems to be difficult at a glance for students, they are ready to read it because they have had experiences of a simple disassembler implementation, helping students understanding. We teach several important elements such as byte order, byte/word instructions, displacement, and effective address. For the practical disassembler and virtual machine implementation, we prepare six kinds of source code (from 1.c to 6.c) written in C language, and nm (nm.c) provided by the source tree of minix2 (we reuse these pieces of code in virtual machine implementation). For example, although Listing 5 shows the source code of 1.c output of which is the same as Listing 3, the disassemble result shown in Listing 6 reaches 114 lines and students have to understand the issues that we teach (*e.g.,* byte order and/or effective address). Although the difficulty will much increase compared to the former disassembler, students can follow these difficulties in a step-by-step manner.

4.1.2 Virtual Machine Implementation

We assume that students who can complete a disassembler implementation understand how to divide a sequence of byte data (binary) into instruction sets and their arguments, meaning that they are ready to start Step2 in Figure 3. The basic structure of a virtual machine is similar to that of a disassembler (*i.e.,* an infinite loop and switch-case structure), therefore students can reuse and extend their disassemblers for virtual machine implementation.

Virtual machine implementation also starts from executing the simplest binary program. Figure 7 shows the execution result of Figure 3 using MMVM with -m option. As we can see in Listing 7, this program ends only 7 steps that consists of two instructions (mov and int) and two system calls (write and exit).

For executing this binary program, we need to implement the behavior of the two instructions, and two system calls referring documents and results of MMVM with -m option. For example in Figure 7, the value of bx register is changed from line 6 to line 7, and mov instruction is executed with 0x0010 at line 6. We can intuitively imagine what's happened in executing mov instruction, confirm its semantics using the document (Intel, 2018), resulting in the correct implementation. As for the system call implementation, we will refer the document of POSIX system calls for confirming the correct behavior, and message structure for receiving requisite arguments for each system call.

4.2 Educational Result and Discussion

We summarize the result of the preliminary evaluation in Table 1 that consists of the name of programs (Name), the number of disassemble result (Disasm), number of execution steps (Execution), the number of kinds of executed system calls (System Call) and the number of students who have completed each task (Disassembler and Virtual Machine). We applied this methodology to two cases: (1) 10 students from 3rd-year who would join our laboratory, (2) 5 students from a master course class. We mention the positive and negative results respectively as follows.

```
main() {
    write(1, "hello\n", 6);
}
```

Listing 5: The source code of 1.c.

```
0000: 31ed          xor bp, bp
0002: 89e3          mov bx, sp
0004: 8b07          mov ax, [bx]
0006: 8d5702        lea dx, [bx+2]
0009: 8d4f04        lea cx, [bx+4]
000c: 01c1          add cx, ax
........... (continue) ..........
```

Listing 6: The disassemble result of 1.c using MMVM.

```
1 AX   BX   CX   DX   SP   BP   SI   DI   FLAGS IP
2 0000 0000 0000 0000 ffdc 0000 0000 0000 ----00: bb0000 mov bx,0000
3 0000 0000 0000 0000 ffdc 0000 0000 0000 ----03: cd20   int 20
4 <write(1, 0x0020, 6)hello
5  => 6>
6 0000 0000 0000 0000 ffdc 0000 0000 0000 ----05: bb1000 mov bx,0010
7 0000 0010 0000 0000 ffdc 0000 0000 0000 ----08: cd20   int 20
8 <exit(0)>
```

Listing 7: The execution result of Listing 3 using MMVM with -m.

As positive results, as shown in Table 1, all students have completed disassembler and virtual machine implementations for almost all given programs except for nm.c. In case (1), we did not measure time students consumes[1] because we wanted to confirm whether this educational methodology could be applied to students or not. Also in case (1), one student completed executing nm.c while the rest of the students failed it. However, we thought this methodology could be applicable for systems-level education for students because at least all students could execute until 6.c. In case (2), due to the result of the case (1), we applied this method to a master course class that consists of 14 classes each class of which reserves 100 minutes. The 5 master students took this class, then 4 students completed executing all given programs within 14 classes. In each case, we observed the progress of each student, then we found that students made several misunderstanding and wrong implementation the kinds of which varied, for example displacement, effective address, sign-extended in disassembler implementation and flag register, relative address and the role of a stack in virtual machine implementation. At that time students focused on the difference of the results between their own disassembler/virtual machine and MMVM, they needed to consider the reason and refer documents, source code and sometimes implemented a small test program if needed. We think these iterative processes change their shallow knowledge to deep knowledge with experiences. Besides, we think they understood

[1] In fact, it varies from several days to a month.

Table 1: The number of students who complete each program.

Program				Complete number of Students	
Name	Disasm (Step1)	Execution (Step2)	System Call (Step2)	Disassembler	Virtual Machine
1.s	4	4	2	15	15
2.s	5	5	2	15	15
1.c	140	140	2	15	15
2.c	148	152	2	15	15
3.c	1,929	1,014	4	15	15
4.c	1,933	1,665	4	15	15
5.c	1,945	1,896	4	15	15
6.c	177	597	2	15	15
nm.c	2,912	553,722	7	15	5

systems-level details comprehensively with experiences because a function call should be emulated by instruction sets while a system call should be emulated by students in a virtual machine implementation, which are different clearly.

As negative results, 10 students could not complete executing nm.c. We think this is because the execution step of nm (553,722, see Table 1) is much different from other programs. Therefore it was difficult to find their mistakes only using MMVM with an option. In other words, the main reason might be the difference in programming skills because the programming skills of master students are better than 3rd-year students in general. We need to extend MMVM for students to find their mistakes easily.

5 RELATED WORK

There have been several types of research and educational software systems for teaching systems-level details. We divide them into two categories: operating systems and computer architecture.

Operating System. In articles like (Atkin and Sirer, 2002; Black, 2009; Brylow, 2008), we can find proposals that teach details of an operating system. These proposals generally offer students to implement an operating system from scratch or modify an educational operating system. Students can learn the main roles of an operating system such as process management, semaphore, system calls, and file systems with experiences. (Dall and Nieh, 2014) gives a tool to review the source code of operating systems for students to learn existing operating systems. These proposals can give experiences to learn operating systems partially such as scheduler, file systems; however these works sometimes might not produce good educational results because students can refer the complete code, meaning that they can copy and paste without much consideration, leading to shallow understanding. Besides, implementing operating system from scratch is too difficult because students have to understand specifications of hardware such as CPU and BIOS. Instead, implementing virtual machine in our methodology is not so difficult because we do not need to use physical hardware and the structure of a virtual machine implementation is simple. Moreover, students can use MMVM to see the internal state of the correct implementation.

Computer Architecture. Proposals found in articles (Skrien, 2001; Warford and Okelberry, 2007) are CPU simulators that allow students to create or modify the architectures being studied. Students can design their architecture and write machine language or assembly language programs and run them on the CPU. However, students cannot learn the connection between hardware (*e.g.,* CPU) and software (*e.g.,* operating system) because they focus on studying the architecture of the CPU. Besides, we cannot use the knowledge directly in practice because the CPUs are enough simplified for learning. In article (Black and Komala, 2011), the authors provide a graphical computer simulator for educations. Unlike other teaching simulators, the proposed simulator faithfully models a complete personal computer including i386 processor, memories, I/O ports interrupts, timers and a serial port. Students can execute any programs that expected to run on x86 processor including an operating system and confirm the values stored in memories and registers. Although students can learn the real CPU architecture (*i.e.,* x86), they can only confirm the values being changed as a program is being executed. However the knowledge between hardware and software is still shallow because they can only observe to be changed inside the CPU.

6 CONCLUSIONS

Due to appearing new topics in computer science, and increasing complexities in hardware and software, the learning process about systems-level details such as computer architecture, programming language, and operating system becomes difficult. To overcome the previous difficulty, we propose a methodology for students to learn systems-level details with a tool called MMVM. Unlikely existing researches and software systems, our methodology is based on the implementation of a virtual machine instead of using existing ones. We conducted a preliminary evaluation in terms of learning in which we offer 15 under/graduate students to implement a virtual machine. As a result, we found two observations. Firstly, all students could implement a disassembler for all programs, the kinds of them varies from just invoking printf to using functions, arguments and control statements (*e.g.,* for, if) and system calls, generating various kinds of 8086's instruction. Secondly, all students could execute 8 of 9 programs in the virtual machine and only one student executed all of them. Through these observations, we realized that students could learn system-level details such as assembly language, computer architecture, and operating systems as real skills. This is because if they did not understand these details, they could not finish to implement a disassembler and a virtual machine. Thereby we can claim that our methodology could address the two problems mentioned in this paper: lack of experiences and comprehensive understanding in system-level details, leading to improving students' knowledge and skills.

REFERENCES

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

Atkin, B. and Sirer, E. G. (2002). Portos: An educational operating system for the post-pc environment. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 116–120, New York, NY, USA. ACM.

Black, M. D. (2009). Build an operating system from scratch: A project for an introductory operating systems course. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 448–452, New York, NY, USA. ACM.

Black, M. D. and Komala, P. (2011). A full system x86 simulator for teaching computer organization. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 365–370, New York, NY, USA. ACM.

Brylow, D. (2008). An experimental laboratory environment for teaching embedded operating systems. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 192–196, New York, NY, USA. ACM.

Committee, T. (1995). Tool interface standard (tis) executable and linking format (elf) specification version 1.2.

Dall, C. and Nieh, J. (2014). Teaching operating systems using code review. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 549–554, New York, NY, USA. ACM.

Gondow, K., Fukuyasu, N., and Arahori, Y. (2010). Mieru-compiler: Integrated visualization tool with "horizontal slicing" for educational compilers. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 7–11, New York, NY, USA. ACM.

Intel (1990). 8086 16-bit microprocessor. http://www.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:8086-datasheet.pdf.

Intel (2018). Intel 64 and ia-32 architecture software developer's manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

John Lions (1977). *Lions' Commentary on Unix 6th Edition*. Peer to Peer Communications/ Annabook.

Oracle Corporation and/or its affiliates (2010). Posix system calls. https://docs.oracle.com/cd/E19048-01/chorus4/806-3328/index.html.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

Skrien, D. (2001). Cpu sim 3.1: A tool for simulating computer architectures for computer organization classes. *J. Educ. Resour. Comput.*, 1(4):46–59.

Tanenbaum, A. S. and Woodhull, A. S. (2005). *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Warford, J. S. and Okelberry, R. (2007). Pep8cpu: A programmable simulator for a central processing unit. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 288–292, New York, NY, USA. ACM.

Zeng, H., Yourst, M., Ghose, K., and Ponomarev, D. (2009). Mptlsim: A cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *SIGARCH Comput. Archit. News*, 37(2):2–9.