Seamless Integration of Digital Circuits and Assembly Language

Prof. Yumin Zhang, Southeast Missouri State University

Yumin Zhang is a professor in the Department of Engineering and Technology, Southeast Missouri State University. His research interests include semiconductor devices, electronic circuits, neural networks, and engineering education.

Seamless Integration of Digital Circuits and Assembly Language

Yumin Zhang

Department of Engineering and Technology
Southeast Missouri State University
Cape Girardeau, MO 63701

Abstract

Digital circuits, with their ability to manipulate binary data, form the foundation of modern electronic devices. On the other hand, assembly language, operating as a low-level programming language, provides nuanced control over a computer's hardware. The seamless convergence of these two realms empowers students with a deep comprehension of the interface between hardware and software, unlocking pathways for adept and streamlined code development at more advanced levels.

A course with the title of "Computer Systems and Assembly Language" is designed for undergraduate students majoring in Computer Science. The first half of the semester focuses on introducing combinational and sequential logic circuits. A free web-based circuit simulator allows students to design a basic CPU, which facilitates the formulation of an instruction set, empowering students to execute machine codes for fundamental operations. In the second half of the semester, the curriculum delves into assembly language. Through online simulators, students explore the fetch-decode-execute cycle and gain insights into implementing essential structures such as the for-loop and if-else, commonly used in high-level computer languages.

Introduction

In recent years, enrollment in the Computer Science (CS) department has surged dramatically. Consequently, faculty members with backgrounds in Electrical and Computer Engineering have been called upon to teach certain CS courses. From their standpoint, there appears to be a disparity between software and hardware, with CS students often lacking fundamental awareness of the digital circuits that form the backbone of software operations. While courses in computer organization and architecture are the most relevant in addressing hardware aspects, their primary focus tends to be on register-transfer level optimizations [1-3].

To furnish CS students with a comprehensive understanding of computer systems from the ground up, we offer a course that commences with the fundamentals of basic logic gates. As shown in Fig. 1, it covers layers 4-7 on *logic gates*, *digital circuits*, *micro-architecture*, and *ISA* that includes both instruction set and assembly language. The first half of the semester focuses on the exploration of combinational and sequential logic circuits, which correspond to layers four and five in Fig. 1. Thereafter, students can design an Arithmetic Logic Unit (ALU) circuit capable of executing fundamental arithmetic and logic operations. This design exercise can aid students in grasping how binary opcodes can effectively govern digital circuits. Furthermore, students also design a functional CPU circuit using the essential components provided by the

circuit simulator. This practical laboratory experience contributes to a thorough comprehension of the operations associated with basic assembly code.

To furnish CS students with a comprehensive understanding of computer systems from the ground up, we offer a course that commences with the fundamentals of basic logic gates. As shown in Fig. 1, it covers layers 4-7 on *logic gates*, *digital circuits*, *micro-architecture*, and *ISA* that includes both instruction set and assembly language. The first half of the semester focuses on the exploration of combinational and sequential logic circuits, which correspond to layers four and five in Fig. 1. Thereafter, students can design an Arithmetic Logic Unit (ALU) circuit capable of executing fundamental arithmetic and logic operations. This design exercise can aid students in grasping how binary opcodes can effectively govern digital circuits. Furthermore, students also design a functional CPU circuit using the essential components provided by the circuit simulator. This practical laboratory experience contributes to a thorough comprehension of the operations associated with basic assembly code.

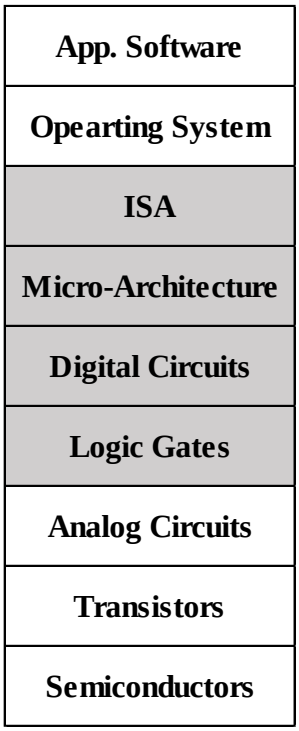


Fig. 1. Layers of abstraction.

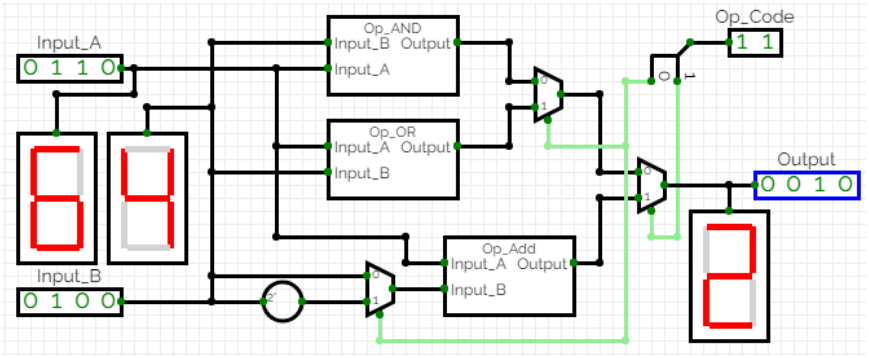
With the CPU circuit designed and simulated, the focus transitions to assembly language. An integral aspect is the correlation between assembly code and machine code, enabling direct interaction with the digital circuits in CPU. The course adopts the RISC-V ISA, renowned for its straightforward and systematic instruction set. The online simulator, emulsiV, proves highly beneficial in illustrating the intricacies of executing assembly code. However, it also exhibits certain limitations. Hence, the incorporation of alternative simulators for assembly language becomes essential, particularly for handling advanced projects.

ALU Design

At the core of a CPU lies the Arithmetic Logic Unit (ALU), responsible for executing arithmetic and logic operations. Fig. 2 illustrates the structure of a 4-bit ALU featuring four distinct operations, each encoded with 2-bit opcodes. This circuit comprises three operational modules, executing the functions of AND, OR, and ADD. The design of logic modules are straightforward, which simply include four copies of AND/OR gates. The adder module at the bottom engages four built-in single-bit adders. Fortunately, the CircuitVerse simulator [4] includes a component, the circle at the bottom, facilitating the conversion of a number into its 2's complement form, simplifying the implementation of the subtraction operation.

The control stemming from the opcode is implemented through three multiplexers. The Most Significant Bit (MSB) of the opcode regulates the rightmost multiplexer: when "00" or "01" is selected, it connects the output to the two logic modules, whereas "10" and "11" direct the output to the adder module. Simultaneously, the Least Significant Bit (LSB) of the opcode governs the selection between two results within the same category (logic or arithmetic). In Fig. 2, the opcode is set to "11," indicating the operation Y = A – B. Initially, the input B undergoes the

conversion to its 2's complement format, followed by addition to A, and the result is showcased in the hex-display on the right. Modifying the opcode will accordingly reflect the corresponding results.



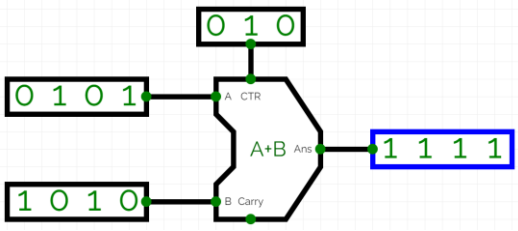
Opcode	Operation
00	AND
01	OR
10	ADD
11	SUB

Fig. 2. ALU Circuit.

With 4-bit operands, the numbers can be conveniently displayed with the hex-displays. In Fig. 2, A = 6, B = 4, and Y = 2. In addition, wire color is coded with the voltage level: *light green* signifies a high voltage level, while *dark green* corresponds to a low voltage level. This effect will also be demonstrated in Fig. 4-5.

CPU Design

The circuit simulator provides an ALU component, as depicted in Fig. 3. In contrast to the ALU circuit illustrated in Fig. 2, this version is more potent, featuring a 3-bit opcode that allows it to perform up to eight distinct operations. For this lab assignment, we opted for a 4-bit bus width, wherein the two input numbers are sourced from the left, and the resulting output is showcased on the right. Positioned at the top is the opcode, and the associated operations are detailed in the table on the right, accessible through the documentation website [5].



Opcode	Opcode Value	Operation
000	0	A & B
001	1	A | B
010	2	A + B
100	4	A & ~B
101	5	A | ~B
110	6	A - B
111	7	SLT

Fig. 3. ALU component with opcode table.

Regrettably, the circuit simulator lacks a register as a built-in component, necessitating students to design a register circuit using D Flip-Flops and multiplexers [6], as illustrated in Fig. 4. This circuit is transformable into a component, featuring input pins on the left and an output pin on the right. Activation of the "Load" pin facilitates the loading of a new 4-bit data into the register. Moreover, it also accommodates the "Clear" operation.

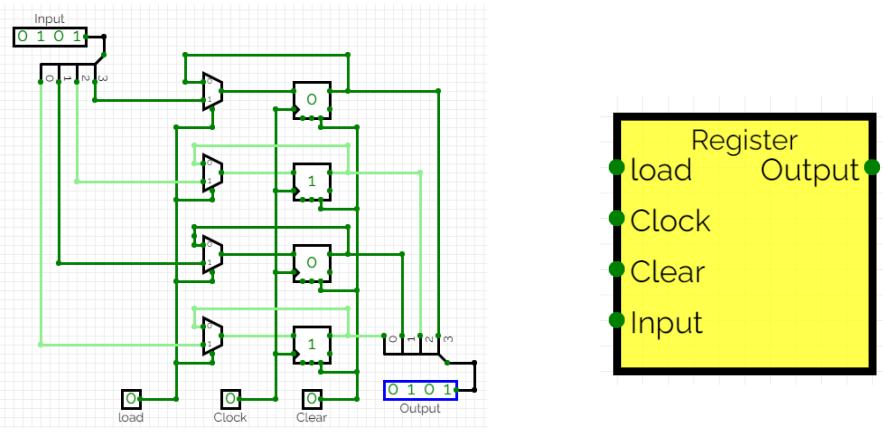


Fig. 4. Register circuit and component.

Furthermore, students are required to construct a one-hot decoder circuit employing basic logic gates, as depicted in Fig. 5. This decoder circuit plays a pivotal role in governing the data path between the ALU and the registers, ensuring that only one output is set to '1' while the other three remain '0'. While this circuit is very simple and can be manually designed, CircuitVerse offers a convenient tool (combinational analysis) capable of generating digital circuits based on truth tables.

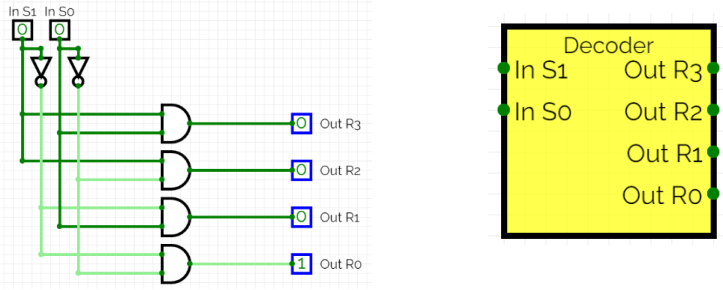


Fig. 5. One-hot decoder circuit and component.

Having designed the register and decoder circuits, the construction of the CPU circuit is now feasible, as illustrated in Fig. 6. Tri-state gates regulate the data paths, capable of being open or shorted based on the output signals from the decoder circuits. Additionally, hex-displays are connected to the inputs and output of the ALU, along with the three registers. In the RISC-V architecture, the first register, X0, is hardwired to zero, thus necessitating only three registers for this CPU. If students are unable to finish the design within one lab session, they can save the incomplete design for future work. In CircuitVerse, the designs can be saved either online or offline.

Adhering to the RISC-V ISA convention, the machine code format for this CPU is structured as follows: | RS2 | RS1 | RD | Opcode |. Given the presence of four registers, only 2 bits are required for their addressing. Additionally, the opcode uses 3 bits, resulting in a 9-bit machine code for this CPU. For instance, the machine code 001101010 can be decoded as follows: RS2 = X0, RS1 = X3, RD = X1, Operation = addition, indicating the operation X1 = X3 + X0. This operation aligns with the associated assembly language instruction: *add x1, x3, x0*. Interestingly,

the sequence of the components in a line of assembly code is reversed when compared to the machine code.

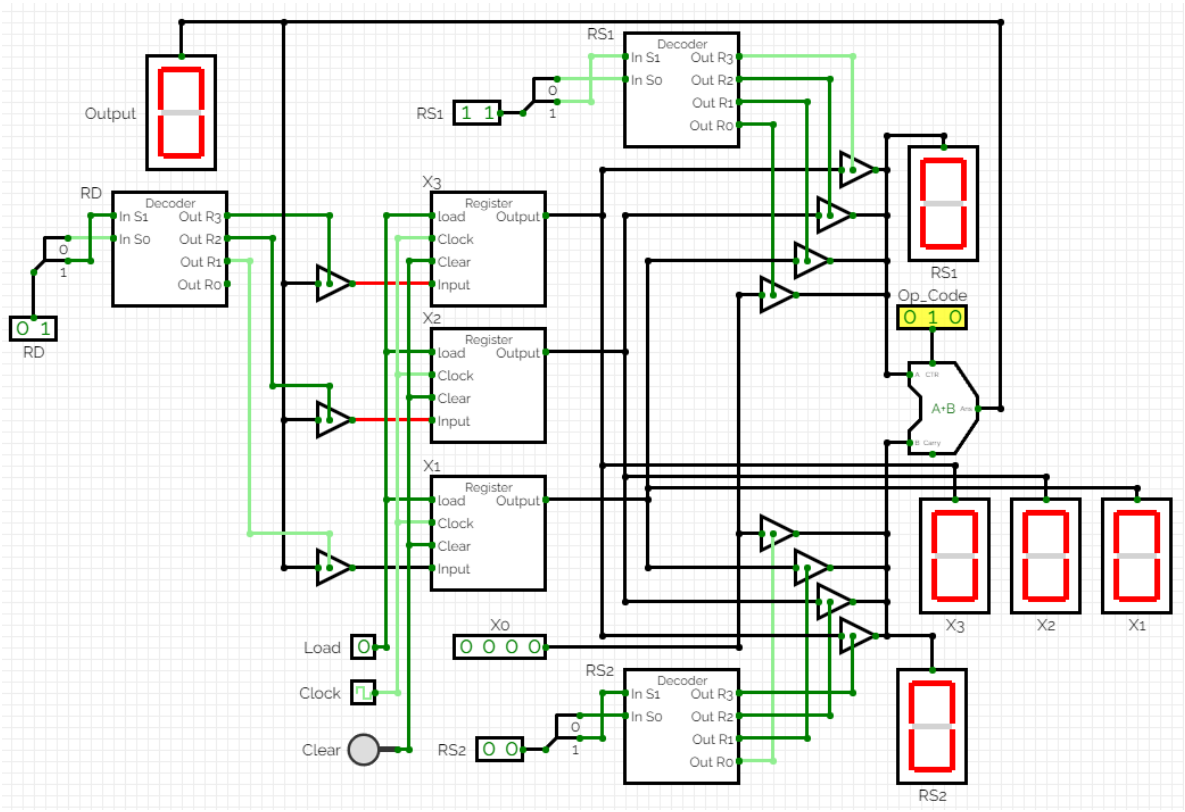


Fig. 6. CPU circuit.

CPU Operations with Machine Code

Upon activating the "Clear" button located at the bottom, all data stored in the three registers are reset to zero, as depicted in Fig. 6. Starting from this initial state, various operations can be subsequently executed.

1) X1 = X0 |~ X0 → 000001101: RS2 = 00, RS1 = 00, RD = 01, Opcode = 101, and the result is X1 = “1111”, which is essentially the inversion of X0.

2) X2 = ‘1’ if (X0 > X1) → 010010111 (RS2 = 01, RS1 = 00, RD = 10, Opcode = 111), and the result is X2 = “0001”, since the number “1111” is equal to -1 in 2’s complement format. The opcode 111 (SLT) means “set the output if input A is less than input B”.

3) X3 = X1 – X2 → 100111110 (RS2 = 10, RS1 = 01, RD = 11, Opcode = 110), and the result is X3 = “1110”.

4) X3 = X1 & X2 → 100111000 (RS2 = 10, RS1 = 01, RD = 11, Opcode = 000), and the result is X3 = “0001”. In this operation, X2 serves as a mask to detect the least significant bit (LSB)

of X1. One application is to identify whether a number is even or odd, as the LSB of even numbers is 0, while for odd numbers, it is 1.

RISC-V Assembly Language

There are several widely used instruction set architectures (ISAs), including X86, ARM, MIPS, and RISC-V. In contrast to conventional proprietary ISAs, RISC-V stands out for its open accessibility, enabling the implementation of processors based on its specifications without incurring any licensing fees. A fundamental aspect of engaging with RISC-V architecture involves proficiency in RISC-V Assembly Language. This proficiency is crucial for programmers and hardware designers seeking to leverage the capabilities and adaptability offered by the RISC-V architecture.

The RISC-V ISA offers a direct interface to the underlying hardware, enabling efficient resource utilization across diverse computing environments. With the expanding RISC-V ecosystem, expertise in RISC-V Assembly Language becomes progressively more valuable for individuals engaged in the advancement of next-generation computing systems. Among its various advantages, a notable characteristic of the RISC-V ISA is its simplicity and regularity, as illustrated in Fig. 7 showcasing the RV32I instruction format.

Fig. 7. RV32I Instruction Format.

Broadly speaking, basic assembly code can be correlated with machine code, and an online simulator (emulsiV) [7] adeptly facilitates this association. Illustrated in Fig. 8, as the assembly code is entered on the right-hand side, the corresponding machine code is automatically generated. To the left of the machine code section also lists the addresses associated with the machine codes.

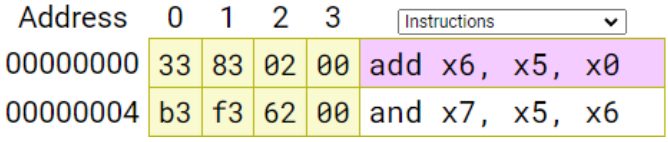


Fig. 8. Machine code and assembly code.

Unfortunately, the machine code is arranged in big-endian format, necessitating the reordering of the bits. For instance, the machine code for the first line of assembly code is 0x00028333, which can be transformed into the R-type binary format with the three register regions highlighted as

bold: |0000000|**00000**|**00101**|000|**00110**|0110011|. The R-type machine code's operations can be deciphered by examining the three remaining sections: function7 section, function3 section, and the opcode section on the right. A comparison with Table 1 confirms that the operation is "ADD." Similarly, one can analyze the machine code for the second line of the assembly code, revealing the AND operation: |0000000|**00110**|**00101**|111|**00111**|0110011|.

Table 1. R-type RISC-V machine code.

31					0	
funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**	
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

The online simulator (emulsiV) offers an animation mode, a valuable tool aiding students in comprehending CPU operations. In this mode, as the code is executed step by step, the program counter (PC) progresses through memory addresses, and the instruction register (IR) fetches the machine code stored in memory. Simultaneously, the right-hand side of the simulator displays the register values.

However, certain limitations exist with this simulator. Firstly, it only accepts authentic assembly codes, despite pseudo codes being more user-friendly. Secondly, the absence of support for labels makes it challenging to implement branching and procedures. Consequently, emulsiV serves primarily for introductory purposes with simple assignments, while Venus [8] emerges as a more powerful online simulator suitable for more advanced projects.

Discussion

In this course, all the simulators used are web-based, ensuring convenient access for students without the need for installations on their computers. However, the trade-off lies in the limited functionality and relatively slow speed of these simulators. While these drawbacks may not pose significant challenges in the context of an introductory course on computer systems and assembly language, it is important to acknowledge the trade-off between accessibility and limitations associated with online simulators.

As depicted in Fig. 6, the online simulator CircuitVerse is sufficiently effective for designing a basic CPU, aiding students in comprehending the interaction between binary machine code and digital circuits. Furthermore, despite its relatively basic nature, emulsiV was utilized for a lab

assignment involving the generation of a dataset and subsequent sorting using the bubble sort algorithm. Therefore, these simulators serve this course quite effectively.

Before teaching this course for the first time in Fall 2023, the author anticipated that CS students might struggle with learning digital logic circuits due to their lack of background in basic electric circuits. However, it turned out that this concern was unfounded, as the students encountered no difficulties in designing and simulating the circuits. In fact, some students displayed great enthusiasm during the lab sessions, and the divide between software and hardware was easily bridged. The feedback from students was overwhelmingly positive, particularly considering that this course had been regarded as the most challenging in the CS curriculum in the past. Regrettably, there is currently no quantitative assessment data available to demonstrate the effectiveness of this approach.

Summary

In a course titled "Computer Systems and Assembly Language," students delve into the design of digital circuits from the ground up. Following the introduction of basic combinational and sequential logic circuits, students progress to designing ALU and CPU circuits. This hands-on process enables students to grasp how binary machine code seamlessly interacts with digital circuits. The second part of the semester introduces assembly language, where the online simulator emulsiV proves invaluable in elucidating the fetch-decode-execute cycle. Additionally, students can also gain insights into the implementation with assembly language of higher-level language structures, such as for-loops, while-loops, if-else statements, functions, etc.

References

[1] David Patterson, John Hennessy, *Computer Architecture: A Quantitative Approach*, 6th ed. Amsterdam, Netherland: Morgan Kaufmann, 2017. ISBN: 978-0128119051.

[2] David Patterson, John Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd ed. Amsterdam, Netherland: Morgan Kaufmann, 2020. ISBN: 978-0128203316.

[3] Jim Ledin, Dave Farley, *Modern Computer Architecture and Organization: Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers*, 2nd ed. Birmingham, UK: Packt Publishing Ltd, 2022. ISBN: 978-1803234519.

[4] CircuitVerse simulator: https://circuitverse.org/

[5] CircuitVerse documentation: https://docs.circuitverse.org/#/

[6] David Harris, Sarah Harris, *Digital Design and Computer Architecture*, RISC-V Edition, Amsterdam, Netherland: Morgan Kaufmann, 2021. ISBN: 978-0128200643.

[7] RISC-V assembly language simulator emulsiV: https://eseo-tech.github.io/emulsiV/

[8] RISC-V assembly language simulator Venus: https://venus.kvakil.me/