A Learning Model of Simple Computing Machine Architecture

Tai-Chi Lee and Hector Zimmermann-Ayala
Department of Computer Science and Information Systems
Saginaw Valley State University
University Center, MI 48710

Abstract

This work emerged from an independent study project, which involved building a simple machine with few registers and limited memory that could execute a program in the machine language of binary code. The purpose of this project is to learn the basic concepts and the fundamental logic design of a processor, the basic machine architecture is outlined. In order to fully understand how an instruction set gets executed, first the instruction format must be well defined and with implemented simplicity where we only limit our instruction types to the minimum to reduce size of the instruction set, which makes it easier to follow when analyzing and synthesizing the functionality of various components in a processor. For example, the arithmetic operations consist only of Addition, Subtraction, Multiplication, and Division. For data movement we have the Load and Store, and for logical operations only Compare is used. For Arithmetic operations only subtraction and addition are implemented. Multiplication and division can be achieved by repeated additions and subtractions respectively. Program flow control uses only the Jump. Furthermore, for memory management, we added a one-byte Cache. This project provides students with an easy learning experience, which is needed for their study at the introductory level of the computer architecture course, while getting a better understanding of how a processor works.

1. Introduction

Over the past five decades the computer industry has experienced enormous changes in the architectural design. Increases in device speed and reliability, as well as reduction in hardware cost and physical size have greatly enhanced computer performance. While major components such as central processing units, memories, input and output units which perform basic computations remain unchanged, issues of using cache, pre-fetch techniques, number of cores in a single chip, and choice of Reduced Instruction Set Computer (RISC) or Complex Instruction Set Computer (CISC) CPUs [1, 3] are debatable. Also, according to studies in the field, CISC CPUs do not seem to have as big an advantage as some thought they would have. So one variant of RISC, Small Instruction Set Computer (SISC) [6], being implemented in this project is introduced for the purpose of study as it seems to have less overhead for a performance advantage over CISC Processors.

2. Constructions and Designs of Basic Components

To orient the student researchers to our SISC CPU [7] for performance and simulation reasons for an Embedded System the following CPU Simulator was developed:

1) Registers

   This SISC CPU Emulator was designed with four byte size registers R0, R1, R2, R3, and R0 is the Accumulator with the ability to have some operations in internal Level 1 Cache Memory with the R1 register.

2) Main Memory

   This SISC CPU Emulator operates with an external memory of 256 bytes as it is designed for embedded systems.

3) Cache Memory

   This SISC CPU Emulator pre-fetches a byte in Level 2 Cache Memory to increase performance in case the processor requires it as data.

4) Buses

   This SISC CPU Emulator uses the data and memory buses in its design.

5) Other peripherals

   Any peripherals that would be added to the system can have interfaces through the Memory Addresses available to the CPU.

3. Instruction Set

A Small Instruction Set is provided with support for the Memory, Cache Memory, and Buses, as well as the ALU (Arithmetic Logic Unit).

3.1. Instruction Set Description

This SISC CPU has the following Instruction Set:

Mnemonic	Instruction	Length	Description.
ADD01	00000001	1 Byte	Add registers R0 and R1 and stores the result on R0 (The Accumulator). This operation also affects the Zero Flag (Whether the result is Zero) and the Overflow Flag (Whether the operation exceeded a Byte or not).
ADD	00000010	2 Bytes	Add the contents of the Accumulator (Register R0) and the data byte previously Cached, storing the result on the Accumulator (Register R0). This operation also affects the Zero Flag (whether the result is Zero) and the Overflow Flag (whether the operation exceeded a Byte or not).
ADD1	00000011	2 Bytes	Similar to ADD, but for Register 1 (R1).
ADD2	00000100	2 Bytes	Similar to ADD, but for Register 2 (R2).
ADD3	00000101	2 Bytes	Similar to ADD, but for Register 3 (R3).
SUB01	00000110	1 Byte	Subtract R1 from R0 and store the result into the Accumulator (R0). The Overflow Flag is affected to reflect whether the result exceeded one Byte or not and the Zero Flag is also affected to reflect whether the result is Zero or not.
SUB	00000111	2 Bytes	Subtracts the Cached Data from the Accumulator and places the result in the Accumulator (R0.) The Overflow Flag is affected to reflect whether the result exceeded one Byte or not and the Zero Flag is also affected to reflect whether the result is Zero or not.
SUB1	00001000	2 Bytes	Similar to Sub, but for R1.
SUB2	00001001	2 Bytes	Similar to Sub, but for R2.
SUB3	00001010	2 Bytes	Similar to Sub, but for R3.
LOAD	00001011	2 Bytes	Loads the Memory Contents with Memory Address in Cache into the Accumulator.
LOAD1	00001100	2 Bytes	Similar to LOAD, but for R1.
LOAD2	00001101	2 Bytes	Similar to LOAD, but for R2.
LOAD3	00001110	2 Bytes	Similar to LOAD, but for R3.
SEC	00001111	1 Byte	Sets the Carry Flag.
CLC	00010000	1 Byte	Clears the Carry Flag.
SEI	00010001	1 Byte	Sets the Interrupt Flag.
CLI	00010010	1 Byte	Clears the Interrupt Flag.
JMP	00010011	2 Bytes	Unconditional Jump to the address pre-fetched on the Cache.
JNZ	00010100	2 Bytes	Conditional Jump to the address pre-fetched on the Cache. Jumps if the Zero Flag is NOT set (Jump if Not Zero).
JIZ	00010101	2 Bytes	Conditional Jump to the address pre-fetched on the Cache. Jumps if the Zero Flag IS set (Jump If Zero).
JCS	00010110	2 Bytes	Conditional Jump to the address pre-fetched on the Cache. Jumps if the Carry Flag IS set (Jump if Carry Set).
JCC	00010111	2 Bytes	Conditional Jump to the address pre-fetched on the Cache. Jumps if the Carry Flag is Clear (Jump if Carry Clear).
STORE	00011000	2 Bytes	Stores the Accumulator (R0) contents into the Memory Address pre-fetched in Cache.
STORE1	00011001	2 Bytes	Similar to STORE, but for R1.

STORE2	00011010	2 Bytes	Similar to STORE, but for R2.
STORE3	00011011	2 Bytes	Similar to STORE, but for R3.
RET	11111111	1 Byte	Return / End.

3.2 Configuration of Components

The learning model presented in this paper consists of four major components including Processing Unit , Input Unit, Output Unit, and Secondary Storage (external memory), where Processing Unit is composed of Central Processing Unit (CPU) and Main Memory (internal memory). And, CPU is further divided into Arithmetic/Logic Unit (ALU) and Control Unit or Master Control Unit (CU/MCU). Figure 1 shows the configuration of the components.

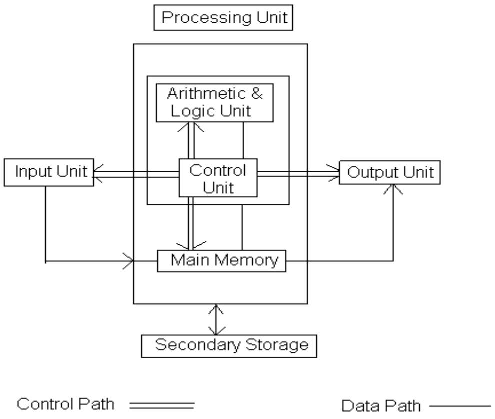


Figure 1

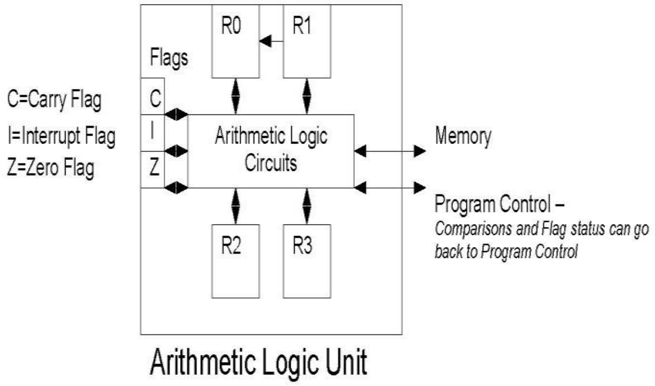


Figure 2

The following describes the role of each component.

1) ALU contains a number of registers, which performs all the arithmetic and logical operations.

2) CU or MCU is central controller, which commands the sequence of operations for all the components.

3) Main Memory is the primary storage for storing programs or data that are currently being used by the computer. It comes in two forms of memory, Random Access Memory (RAM) and Read Only Memory (ROM) [7].

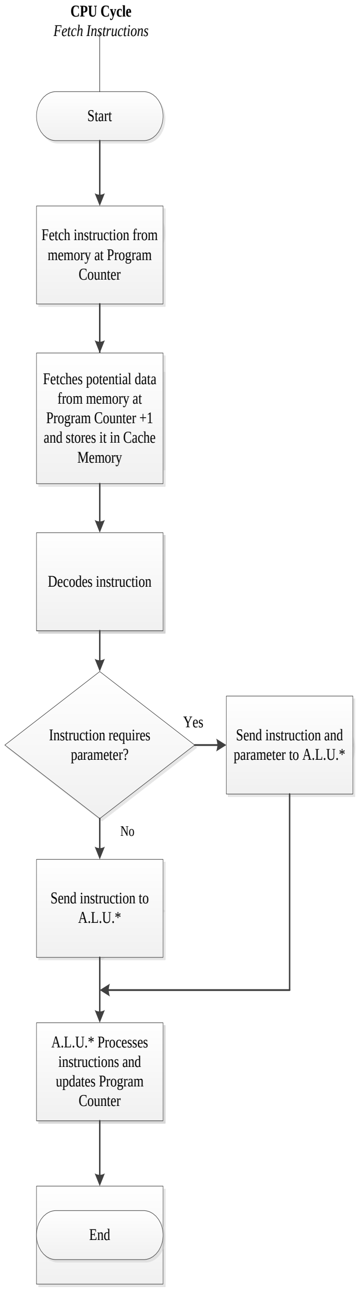
4) Input Unit is a device that allows users to enter data into the main memory. Examples are keyboard, mouse, etc…

5) Output Unit is a device that allows users to display the results from processing. Examples are printer, monitors, etc…

6) Secondary Storage is for storing and retrieving data and program. Examples are hard disk drive, CD, USB, tape, etc.…

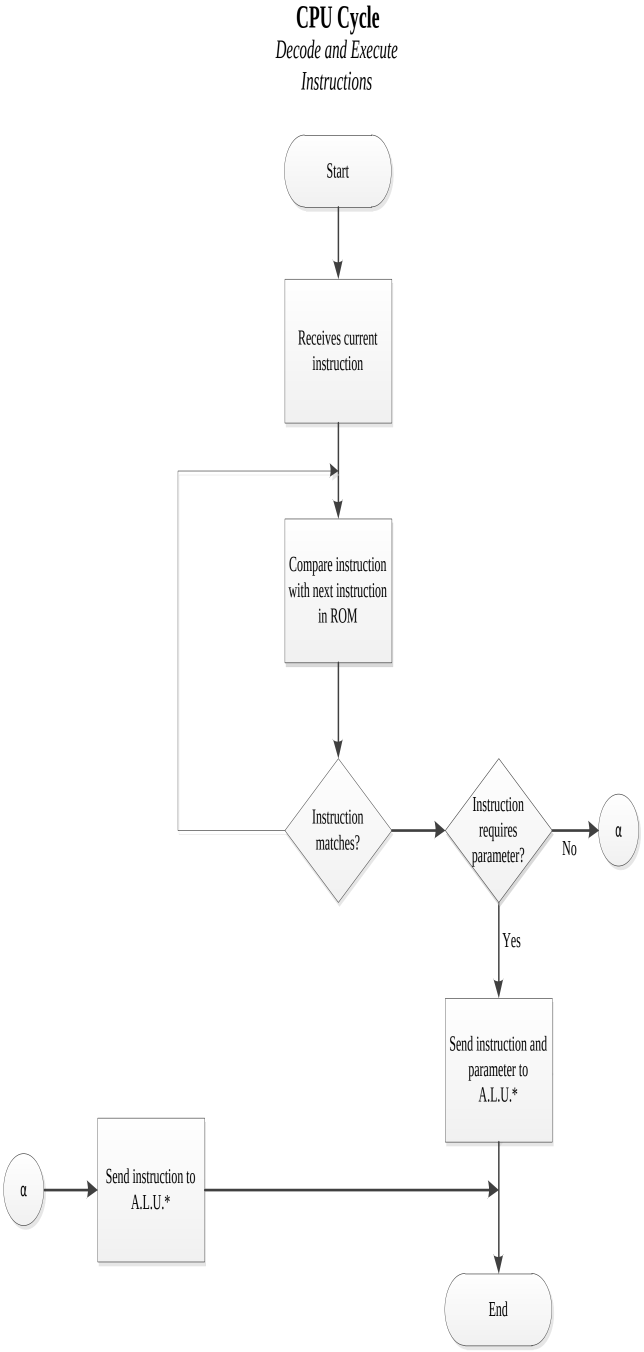
7) Bus is a path by which the electronic signal (control path) or data (data path) travels from one place to another.

In our designed model CPU will have a list of the custom instruction set it supports. When the CPU is started, it will read this list under the PC Master Control. The PC Master Control then loads the program through the Control Instruction LOADPRG, and executes a Memory Dump on screen. Afterwards, it transfers control to the PARSE instruction, which then executes a Register Dump on Screen and reads the first two bytes into Cache [7]. The Master Control scans for instructions to match the required operation and, if found, will determine whether it needs the Cached data byte or not. If it does, it will execute the instruction and send the Cached data byte as a parameter. It will then update the Program Counter accordingly and execute the instruction. After executing the instruction, it will then output a disassembly of the instruction on Screen. At the end of the run, it will perform a last Memory Dump on Screen. The following Figures 2 and 3 illustrate the flows of executions followed by a test run with results shown in Section 3.3.


CPU Cycle
Fetch Instructions
Start
Fetch instruction from memory at Program Counter
Fetches potential data from memory at Program Counter +1 and stores it in Cache Memory
Decodes instruction
Instruction requires parameter?
Yes
Send instruction and parameter to A.L.U.*
No
Send instruction to A.L.U.*
A.L.U.* Processes instructions and updates Program Counter
End

*Arithmetic Logic Unit

Figure 3


CPU Cycle
Decode and Execute Instructions
Start
Receives current instruction
Compare instruction with next instruction in ROM
Instruction matches?
Instruction requires parameter?
No
α
Yes
Send instruction and parameter to A.L.U.*
α
Send instruction to A.L.U.*
End

*Arithmetic Logic Unit

Figure 4

3.3 Test Runs and Results

The following sample program was written to test out the SISC CPU. It is expressed in decimal number system and interpreted in binary, as well as disassembled by the program for a better appreciation. Also, it provides memory dumps after loading and at the end of execution.

Input program file: 11 21 12 22 13 23 14 24 1 25 22 4 15 26 22 16 15 23 25 255 10 10 5 5 24 23 255	
Program Loaded Ok	
**** Memory dump begin ****	
1 11 21 12 22 13 23	
7 14 24 1 25 22 4	
13 15 26 22 16 15 23	
19 25 255 10 10 5 5	
25 24 23 255 0 0 0	
**** End of memory dump ****	
Running Program...	
R0:0 R1:0 R2:0 R3:0	←Initial contents in R0, R1, R2, and R3 are all 0.
1 LOAD 21	← Memory Location 21 contains **10,** which gets loaded into **R0**
R0:10 R1:0 R2:0 R3:0	
3 LOAD1 22	← Memory Location 22 also has **10**, which will be loaded into **R1**
R0:10 **R1:10** R2:0 R3:0	
5 LOAD2 23	← Memory Location 23 has 5, which will be loaded into **R2**
R0:10 R1:10 **R2:5** R3:0	
7 LOAD3 24	← Memory Location 24 has 5, which will be loaded into **R3**
R0:10 R1:10 R2:5 **R3:5**	
9 ADD01	← This will add **R0** and **R1** and will store the result into **R0**
R0:20 R1:10 R2:5 R3:5	
10 STORE1 22	←This will store the contents of **R1** into Memory Location **22**
R0:20 R1:10 R2:5 R3:5	
12 ADD2 15	← This will add **15** to **R2** and store it into **R2**
R0:20 R1:10 R2:20 R3:5	
14 STORE2 22	← This will store the contents of **R2** into Memory Location ***22***
R0:20 R1:10 R2:20 R3:5	
16 CLC	←This clears the Carry Flag
R0:20 R1:10 R2:20 R3:5	
17 SEC	← This sets the Carry Flag
R0:20 R1:10 R2:20 R3:5	
18 JCC 25	← If Carry Flag was clear it would jump to a STORE, which proves that the conditional jump JCC is operating fine on opposite condition.
20 RET	
**** Memory dump begin ****	
1 11 21 12 22 13 23	
7 14 24 1 25 22 4	
13 15 26 22 16 15 23	
19 25 255 10 ***20*** 5 5	
25 24 23 255 0 0 0	
**** End of memory dump ****	

Similarly, by changing the source program to have test runs on the remaining instruction types not contained in the above test program, we found that they all work correctly. This assures that our designed model works as they are designed for. Note that due to page limitation, the remaining test run results are not shown, but they are available upon request.

4. Conclusion

In this simple model of machine architecture, we show the basic components of a computer and the constructions of each element. For learning purposes, we keep it simple enough for beginners to follow easily. Once they have the concepts and a good understanding of the basic operations, they can carry on to expand the functionalities necessary to handle the more complex computations. Furthermore, they can also customize the design to meet the needs of individual applications. If design couples with Field Programmable Gate Array (FPGA) co-design techniques [4, 5], it may become a viable means by which to perform complex calculations quicker than general purpose processors. Our approach and design make use of the fundamental concepts of digital logic. If custom-built instructions are used, we expect the customized system to out scale similar single- purpose computing platforms due to the use of a smaller instruction set and embedded processors. The model presented here is intended to be used for an introductory course of computer architecture. More complex features such as shared memory multiprocessors [6], parallel and pipelining processing [2, 3] can be added when the level of students advances.

5. Acknowledgement

We are grateful for the support of the SVSU professional development grant for the work of this paper and the student author would like to thank Dr. Lee for providing the resources and encouragements. Also, our gratitude goes to Ann Garcia for her efforts in preparing this manuscript.

References

[1] John P. Hayes, *Computer Architecture and Organization*, 1988, McGraw-Hill, Inc.

[2] Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing,* 1984, McGraw-Hill, Inc.

[3] Richard Y. Kain, *Advanced Computer Architecture: A Systems Design Approach*, 1996, Prentice- Hall, Inc

[4] Tai-Chi Lee and Patrick Robinson, *A FPGA-Based Designed for an Image Compressor*, International Journal of Pure and Applied Math, Academic Publications, Volume 33 No.1 2006, pp 63-67.

[5] Tai-Chi Lee, Mark White, Michael Gubody, Allison Nicol, Christpher Plachta, Jeremy Strawn, and Cori Thompson, *Building An FPGA-Based Computing Platform,* The Proceedings of The 2012 International Conference on Frontiers in Education: Computer Science & Computer Engineering, pp 522-527, July 16-19, 2012, Las Vegas, NV.

[6] Julia Lobur and Linda Null, *The Essentials of Computer Organization and Architecture,* 3rd Edition, 2012, Jones & Bartlett Learning, LLC.

[7] William Stallings, *Computer Organization and Architecture: Designing for Performance,* 9th edition, 2013, Person Education., Publishing as Prentice Hall.