The Cupola
Scholarship at Gettysburg College

Gettysburg College Open Educational Resources

7-22-2016

Implementing a One Address CPU in Logisim

Charles W. Kann
Gettysburg College

Follow this and additional works at: http://cupola.gettysburg.edu/oer

Share feedback about the accessibility of this item.

Kann, Charles W., "Implementing a One Address CPU in Logisim" (2016). *Gettysburg College Open Educational Resources*. 3.
http://cupola.gettysburg.edu/oer/3

This open access book is brought to you by The Cupola: Scholarship at Gettysburg College. It has been accepted for inclusion by an authorized administrator of The Cupola. For more information, please contact cupola@gettysburg.edu.

Implementing a One Address CPU in Logisim

Description

Most computer users have an incorrect, but useful, cognitive metaphor for computers in which the user says (or types or clicks) something and a mystical, almost intelligent or magical, behavior happens. It is not a stretch to describe computer users as believing computers follow the *laws of magic*, where some *magic incantation* is entered, and the computer responds with an expected, but magical, behavior.

This magic computer does not actually exist. In reality computer are machines, and every action a computer performs reduces to a set of mechanical operations. In fact the first complete definition of a working computer was a mechanical machine designed by Charles Babbage in 1834, and would have run on steam power.

Probably the biggest success of Computer Science (CS) in the 20th century was the development of abstractions that hide the mechanical nature of computers. The fact that average people use computers without ever considering that they are mechanistic is a triumph of CS designers.

This purpose of this monograph is to break the abstract understanding of a computer, and to explain a computer's behavior in completely in mechanistic terms. It will deal specifically with the Central Processing Unit (CPU) of the computer, as this is where the magic happens. All other parts of a computer can be seen as just providing information for the CPU to operate on.

This monograph will deal with a specific type of CPU, a one-address CPU, and will explain this CPU using only standard gates, specifically AND, OR, NOT, NAND and XOR gates, and 4 basic Integrated Circuits (ICs), the Decoder, Multiplexer, Adder, and Flip Flop. All of these gates and components can be described as mechanical transformations of input data to output data, and the overall CPU can then be seen as a mechanical device.

Keywords

Digital Circuits, System Architecture, Computer Organization, Integrated Circuits, Computer Logic, Central Processing Unit (CPU), Processor Architecture, Multiplexer, Decoder, Arithmetic Logic Unit, Register File, Flip-Flop, Memory, Memory Latch, Adder, Full Adder, Half Adder, State Computer, State Machine, Mod 4 Counter, 7400, 7400 Series, Digital Circuit Lab Manual, Electronic Circuits, Electronic Projects, Digital Circuit Projects, Computer Science Online, Online Laboratory Manual, Laboratory Manual

Comments

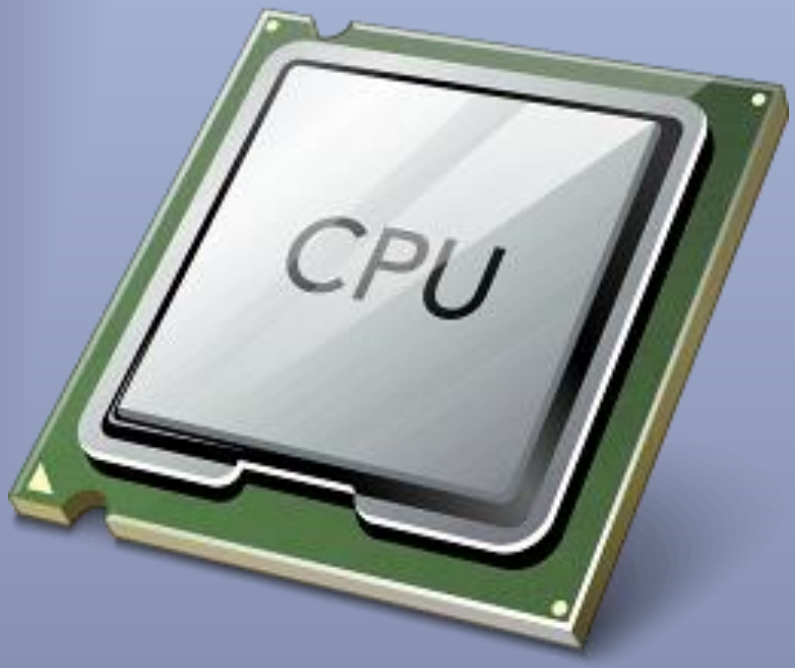
The zip file included with this entry should have all materials for the text, including the assembler, Logisim circuits, programs, and figures. As new materials are developed, they may be available as beta releases at http://chuckkann.com.

Please send me an email at ckann@gettysburg.edu if you use this book as a textbook in a class.

Creative Commons License

This work is licensed under a Creative Commons Attribution 4.0 License.

This book is available at The Cupola: Scholarship at Gettysburg College: http://cupola.gettysburg.edu/oer/3



Implementing a One-Address CPU in Logisim

Vol 1: Implementing CPUs in Logisim Series

The monograph implements a simple one-address CPU using Logisim. A working , programmable one-address CPU is created and explained, including the assembly language used for the CPU, an assembler to translate the assembly language into machine code, and how the CPU uses the machine code to implement the program.

© Charles W. Kann III
277 E. Lincoln Ave.
Gettysburg, Pa

All rights reserved.

This book is licensed under the Creative Commons Attribution 4.0 License

Last Update: Sunday, November 06, 2016

A set of problems for the computer in this text is being developed, and if you would like them email me at ckann@gettysburg.edu

This book is available for free download from: http://chuckkann.com/books/.

Other books by Charles Kann

Kann, Charles W., "Digital Circuit Projects: An Overview of Digital Circuits Through Implementing Integrated Circuits - Second Edition" (2014). *Gettysburg College Open Educational Resources.* Book 1.
http://cupola.gettysburg.edu/oer/1

Kann, Charles W., "Introduction to MIPS Assembly Language Programming" (2015). *Gettysburg College Open Educational Resources.* Book 2.
http://cupola.gettysburg.edu/oer/2

Forward

The purpose of the monographs in this series is to help students and others interested in CS to understand the mechanical nature of the most important part of a computer, the CPU. This series is intended to be used by CS students and practitioners who want a deeper understanding of the design of a CPU, but it is also targeted at computer hobbyists who are interested in how computers actually work.

This is the first monograph in a planned series documents which describe and implement different CPU architectures. This monograph implements a one-address Central Processing Unit (CPU) which the author has implemented in Logisim. Except for some simple Integrated Circuits (ICs), specifically a decoder, multiplexer, adder, and flip flop, this CPU is designed using only simple gates (AND, OR, NOT, XOR and NAND) and simple Boolean logic, and thus should be at a level that can be understood by a hobbyist who wants to delve deeper into what a CPU is, and what can a CPU do.

This CPU can also be used in classes on Computer Organization or Computer Architecture. A planned workbook to be released for this CPU will have a number of projects that students or hobbyists can implement in Logisim to modify and extend the capabilities of the CPU, to aid in the understanding of how a CPU works. These projects will include enhancing the assembly language and modifying the assembler to handle the modifications, as well as modifying the hardware to handle the new features.

This book can also be used with other books by the author to create a complete class in Computer Organization. The book Digital Circuits Projects implements the basic ICs used in this textbook using breadboards chips to illustrate how these circuits work. The book Introduction to MIPS Assembly Language Programming introduces the students to a real assembly language for the MIPS computer, and integrates the assembly language into larger programming constructs like structured programming, subprogram calling and conventions, and memory concepts such as stack, heap, static, and text memory. These three texts are used by the author in his Computer Architecture course. All are free to download (except for the workbook associated with this textbook, for which a nominal fee is asked) from the links provided. Additional resources for these textbooks can be found at the author's web site, http://chuckkann.com.

This monograph is the first in a planned series of monographs, hopefully written as student research projects, that will implement different CPU designs, including the difference between Von Neumann and Harvard architectures, and 0-Address, one-address, and 2/3-Address computer architectures. This will help readers to understand some of the design decisions that go into implementing a CPU. All of the CPU designs will be RISC based, though a CISC 3-Address architecture may be implemented to show the complexity of CISC designs, and talk about why they are no longer used.

The author is an adjunct professor at two schools, Gettysburg College and Johns Hopkins University, Engineering Professionals program. Neither school provides a large number of students to work with, so if a student using this book is interested in implementing a CPU as part of this series, I would be willing to work with them and their advisor on this as a research project. Please contact me at ckann(at)gettysburg.edu and I would be glad to discuss this.

Acknowledgements

The following have helped in producing this text.

I would like to thank Amrit Dhakal for his help in writing and debugging the assembler code, and for his ideas in the development of the CPU.

The cover design includes a picture of a generic CPU which was retrieved from the site http://www.freeiconspng.com/free-images/microprocessor-icon-9584

Contents

1 Introduction 9
1.1 Basic Components in a CPU 9
1.1.1 Boolean operations 9
1.1.2 Integrated Circuits 10
1.1.3 ALU (Adder) 11
1.1.4 Decoder 11
1.1.5 Multiplexer 12
1.1.6 Registers (D Flip Flops) and Memory 12
1.2 Comparisons of Computer Architectures 13
1.2.1 Zero-, One-, and Two/Three- Address Architecture 13
1.2.2 One-Address Architecture 15
1.2.3 Two/Three - Address Architecture 16
1.3 Von Neumann and Harvard Architectures 17
2 Assembly Language 19
2.1 What is Assembly Language 19
2.2 Assembly Language Caveats 20
2.3 Assembler Directives 21
2.4 Data types 22
2.5 Designing an Assembly Language 22
2.5.1 Transferring data from main memory to internal CPU memory 23
2.5.2 Set of valid ALU operations 23
2.5.3 Program Control (Branching) 23
2.5.4 Assembler Instructions 24
2.6 Assembler Programs 25
2.6.1 Loading a value into the AC 26
2.6.2 Adding Two Immediate values 26
2.6.3 Adding two values from memory, and storing the results 27
2.6.4 Multiplication by iterative addition 27
3 Machine Code 29
3.1 Overview of the machine code instruction format 29
4 Assembler program 32

4.1 Running a program on the One-Address CPU 33
5 CPU implementation 45
5.1 The sign extend unit 45
5.2 The ALU 45
5.3 The Control Unit (CU) 47
5.4 The CPU 47
5.4.1 The CPU – Arithmetic Subsection 48
5.4.2 The CPU – Execution Path Subsection 49
5.5 Implementing the CU 50

Figures

Figure 1-1: ALU 11
Figure 1-2: Decoder 12
Figure 1-3: Multiplexer 12
Figure 1-4: Square Wave 13
Figure 1-5: 0-address architecture 14
Figure 1-6: 3-address architecture 16
Figure 1-7: 2-address architecture 17
Figure 1-8: Difference between a Von Neumann and Harvard architecture 18
Figure 3-1: 16-bit machine instruction format 29
Figure 4-1: Assembly Process 32
Figure 4-2: Assembler overview 33
Figure 4-3: Running the assembler - step 1 34
Figure 4-4: Running the assembler - step 2 35
Figure 4-5: Running the assembler - step 3 36
Figure 4-6: Running the assembler - step 4 37
Figure 4-7: Running the assembler - step 5 38
Figure 4-8: Running the CPU - step 1 39
Figure 4-9 Running the CPU - step 2 40
Figure 4-10: Running the CPU - step 2 41
Figure 4-11: Running the CPU - step 3 42
Figure 4-12: Running the CPU - step 4 43
Figure 4-13: Running the CPU - step 5 44
Figure 5-1: The sign extend unit 45
Figure 5-2: Simple Adder 46
Figure 5-3: Adder/Subtracter 46
Figure 5-4: Adder/Subtracter with overflow 47
Figure 5-5: CPU - Arithmetic Subsection 48
Figure 5-6: CPU - Execution Path Subsection 49
Figure 5-7: Control Unit 50

Table 1-1 : Truth table for AND gate 10
Table 1-2: Truth table for NOT gate 10
Table 1-3: Truth table for AND, OR, XOR, and NAND gates 10
Table 5-1: Operations and control wires 50

1 Introduction

Most computer users have an incorrect, but useful, cognitive metaphor for computers in which the user says (or types or clicks) something and a mystical, almost intelligent or magical, behavior happens. It is not a stretch to describe computer users as believing computers follow the *laws of magic*, where some *magic incantation* is entered, and the computer responds with an expected, but magical, behavior.

This magic computer does not actually exist. In reality computer are machines, and every action a computer performs reduces to a set of mechanical operations. In fact the first complete definition of a working computer was a mechanical machine designed by Charles Babbage in 1834, and would have run on steam power.

Probably the biggest success of Computer Science (CS) in the 20th century was the development of abstractions that hide the mechanical nature of computers. The fact that average people use computers without ever considering that they are mechanistic is a triumph of CS designers.

This purpose of this monograph is to break the abstract understanding of a computer, and to explain a computer's behavior in completely in mechanistic terms. It will deal specifically with the Central Processing Unit (CPU) of the computer, as this is where the magic happens. All other parts of a computer can be seen as just providing information for the CPU to operate on.

This monograph will deal with a specific type of CPU, a one-address CPU, and will explain this CPU using only standard gates, specifically AND, OR, NOT, NAND and XOR gates, and 4 basic Integrated Circuits (ICs), the Decoder, Multiplexer, Adder, and Flip Flop. All of these gates and components can be described as mechanical transformations of input data to output data, and the overall CPU can then be seen as a mechanical device.

While it is not necessary to know the details of implementing these ICs to read this text, only how the ICs are used, the implementation of these 4 ICs is not difficult. A free book on the implementation of these integrated circuits is available from the author at http://cupola.gettysburg.edu/oer/1/. The rest of this chapter will provide a basic overview of the gates and ICs used in this text (Section 1.1) and then give an overview of different ways a CPU can be organized and architected.

1.1 Basic Components in a CPU

This section covers the basic components in a CPU. It covers the gates which are used in the CPU, and four common ICs used in a CPU, the adder, decoder, multiplexer, and register.

1.1.1 Boolean operations

Gates are hardware implementations of Boolean operations. Boolean operations are operations that take one or more binary values and calculate a result. For example, the AND operation

takes 2 binary value (with 0 = false and 1 = true) and calculates a binary output. For the AND operations, the inputs of `0 AND 0`, `0 AND 1`, and `1 AND 0` all yield `0` (false), and the input of `1 AND 1` yields `1` (true). This is normally implemented using a truth table, as follows:

Input		Output
A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

Table 1-1 : Truth table for AND gate

In this text 5 Boolean operators will be used, the AND, OR, NOT, XOR, and NAND. The NOT is a unary function (it only takes one input), and so is given in Table 1-2.

Input	Output
A	NOT
0	1
1	0

Table 1-2: Truth table for NOT gate

The AND, OR, XOR, and NAND operators are binary (taking two inputs) and shown in Table 1-3 below.

Input		Output			
A	B	AND	OR	XOR	NAND
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

Table 1-3: Truth table for AND, OR, XOR, and NAND gates

1.1.2 Integrated Circuits

An Integrated Circuit (IC) is a collection of gates that are used to build components to implement a behavior. The components of an IC are simple gates, and all of the ICs in this chapter can be reduces easily to AND, OR, NOT, XOR, and NAND gates. So just as the gates mechanically transform input into output, the ICs also do a mechanical transformation of inputs into outputs.

The ICs to be described in this chapter are the Adder, Decoder, Multiplexer, and Flip Flop.

1.1.3 ALU (Adder)

The Arithmetic Logic Unit (ALU) is the central component of the CPU. It performs all arithmetic and logical operations on the data. Everything else in the CPU is designed to provide data for the ALU to operate on.

The ALU is normally a *black box* that provides the operations for the CPU on two operands. This black box is responsible for all operations that the CPU performs, including not only integer and logical operations, but floating point calculations as well. Operations like floating point calculations are very complex, and are often implemented in coprocessors. To keep things simple, the only data types allowed for the CPU in this text will be integers, and only integer and logic operations will be allowed.

An overview of an ALU can be seen in the typical ALU shown below. An ALU takes two arguments, and implements and operation, such as add, subtract, multiply and divide operations on these two operands. The ALU also allows operations such as Boolean operations (AND, OR, XOR, etc), bit-shifting, and comparison.

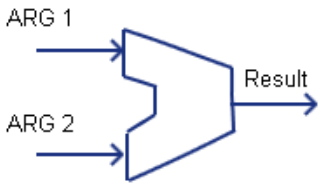


Figure 1-1: ALU

Because the only ALU operation covered in the recommended text on ICs is an adder, the ALU the used in the Logisim implementation of the CPU will contain only an adder circuit. Using an adder, both addition and subtraction are implemented. A more robust configuration for an ALU is can be found in the extra notes that can be accessed for this text.

1.1.4 Decoder

A decoder is an IC splits a n-bit number into 2^n separate output lines. For example, consider a 2-bit number, which can have 4 values, 0x0 … 0x3. A decoder would take as input 2 input lines representing the 2-bit number, and turn on one (and only one) of the four output lines. The line which is turned on corresponds to the value of the 2-bit input. So in the following diagram, if the 2-bit input has both lines high (representing “11”), and the output line 3 is turned on.

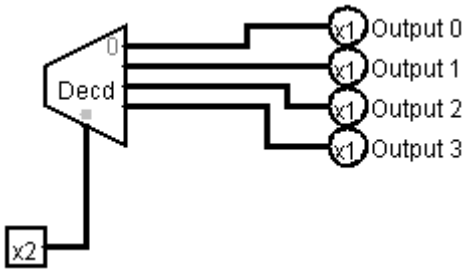


Figure 1-2: Decoder

1.1.5 Multiplexer

A multiplexer is an IC that selects between different inputs. In the following diagram, the 8 bits used by the Output can come from either Register 1 or Register 2. The MUX selects which 8-bit value to use. If Select Input is 0, Register 1 is chosen, and if Select Input is 1 Register 2 is chosen.

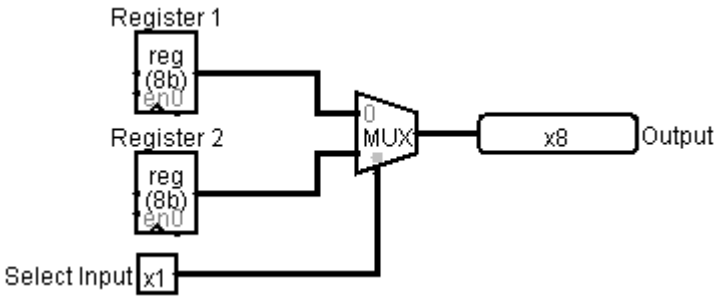


Figure 1-3: Multiplexer

1.1.6 Registers (D Flip Flops) and Memory

Memory is different than the other ICs in that it is synchronous, where synchronous means the memory cell has a value which at discrete time intervals. An example of this behavior is the `$ac` in the following program fragment:

```
clac          ← time = t0, $ac = 5
addi 5        ← time = t1, $ac = 5
addi 7        ← time = t2, $ac = 5
subi 2        ← time = t3, $ac = 5
```

This program shows that the value of the memory, `$ac`, changes discretely over time. This discrete behavior is accomplished by a system clock. A system clock is an electronic oscillator circuit that produces a square wave with a precise frequency. The following is an illustration of a square wave.

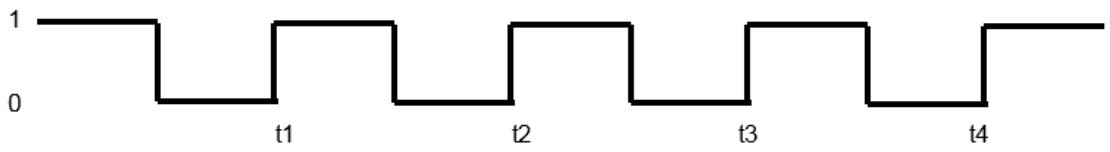


Figure 1-4: Square Wave

In a square wave, the value is always 0 or 1, and memory uses the transition from 0 to 1 (the positive edge) to change the value of all memory components. Thus memory cells have discrete values that change on each clock pulse.

Any memory cell in a CPU is normally called a register. Register memory normally consists of Static Ram (SRAM), and is implemented using Flip Flops. Main computer memory is often Dynamic Ram (DRAM)., however some memory, particularly cache memory, can be implemented with SRAM. The specifics of memory are beyond the scope of this text, and the all the reader needs to know is register memory is most typically SRAM and located within the CPU.

1.2 Comparisons of Computer Architectures

This monograph is the first in a series of monographs that will cover different types of CPUs, where the two big differences between the CPU types is the address format of the instructions, and how the instruction and data memory for the processor is divided. This next section will deal with the address format of the instructions. The section following it will cover designs where instruction and data memory are combined (Von Neumann architecture) verses separate instruction and data memory (Harvard architecture).

1.2.1 Zero-, One-, and Two/Three- Address Architecture

The major difference between 0-, 1-, and 2/3- address computer architectures is where the operands for the ALU come from. This section will outline each of these architectures.

Note that in all of these architectures, operands can come from registers/memory, or operands can be part of the instruction itself. For example, the value used in the instruction `add A` in a one-address computer comes from memory cell at an address corresponding to the label `A`, and the instruction adds *the value in* memory location `A` to the `$ac`. In the instruction `addi 5` the value of the operand is included in the instruction, and is referred to as an *immediate* value. In this series of monographs operators that use an immediate value will be appended with an "*i*". For example, as shown above, the *add* instruction uses a memory value, and the `addi` uses an immediate value.

1.2.1.1 0-Address Architecture

When discussing the address architecture of a computer, the central question is how are the arguments to the ALU retrieved, and where are the results from the ALU stored? A 0-adress architecture retrieves (pops) the two arguments from the top of an *operand stack*, performs the operation, and then stores (pushes) the result back on the operand stack. The two operands to the ALU are implied as the two operands on the top of the stack, and the operation, in this case add, does not specify any operands. Because the operator does not take any explicit operands, 0 addresses are included as part of the operation and this is called a 0-address architecture. Note that a 0-address architecture is often referred to as a stack architecture because it uses a stack for the operands to/from the ALU.

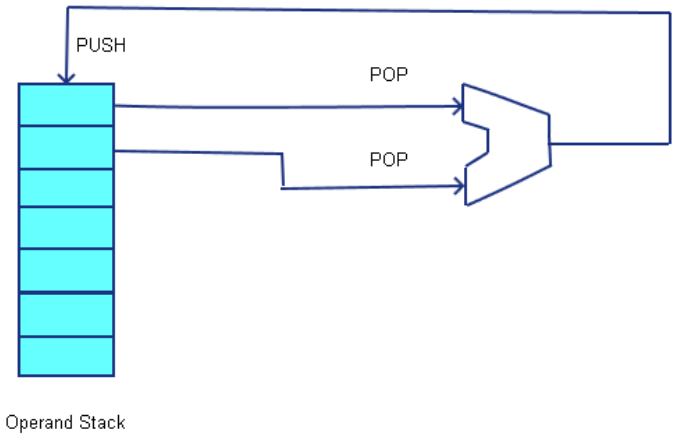


Figure 1-5: 0-address architecture

When writing assembly code to for this architecture, the operands are first pushed onto the stack (from memory or immediate values) using two *push* operations. The operation is executed, which consists of popping the two operands off the stack, running the ALU, and pushing the result back to the stack. The answer is then stored to memory by using a *pop* operation. The following program, which adds the value of variable A and value 5, then stores the result back to variable B, illustrates a simple 0-address program.

```
PUSH A
PUSHI 5
ADD
POP B
```

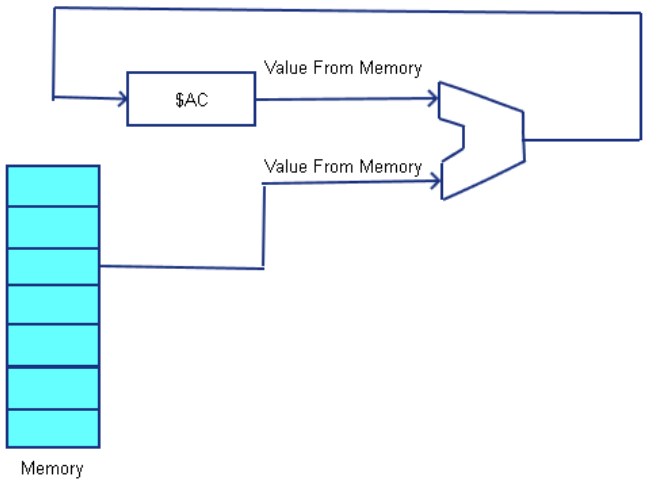
Program 1-1: 0-address program to add two numbers

Historically there have been computers implemented using 0-address architectures, such as the Burroughs 6500 and 7500 series, but it is seldom if ever used in modern hardware architectures.

However, most modern languages running on Virtual Machines (VM), such as the Java Virtual Machine (JVM) or the .Net Common Language Runtime (CLR), implement 0-address, or stack, architectures.

1.2.2 One-Address Architecture

In a one-address architecture a special register, called an *Accumulator* or `$ac` is maintained in the CPU. The `$ac` is always an implied input operand to the ALU, and is also the implied destination of the result of the ALU operation. The second input operand is the value of a memory variable or an immediate value. This is shown in the following diagram.



Program 1-2: one-address architecture

The following is a simple one-address computer program to add the value 5 and the value of variable A, and store the result back into variable B.

```
CLR    // Set the AC to 0
ADDI 5 // Add 5 to the $AC.  Since it was previously 0, this loads 5
ADD A  // Add A+5, and store the result in the $AC
STOR B // Store the value in the $AC to the memory variable B
```

Program 1-3: one-address program to add two numbers

Because only one value is specified in the ALU operator instruction, this type of architecture is called a one-address architecture. Because a one-address architecture always has an accumulator, it is also called an *accumulator* architecture.

Historically many early CPUs used one-address designs, including the Intel 8080 and PDP-8, used accumulator architectures. Because of their simplicity and can be faster than other architectures, some micro-computer designs still use an accumulator architecture, though most computers implement general purpose register designs.

1.2.3 Two/Three - Address Architecture

The two-address and three-address architectures are called *general purpose register* architectures. The two-address and three-address designs both operate in a similar fashion. Both architectures have some number of general purpose registers can be used to select the two inputs to the ALU, and the result of the ALU operation is written back to a general purpose registers. The difference is in how the result of the ALU operation (the destination register) is specified. In a three-address architecture the 3 registers are the destination (where to write the results from the ALU), R_d, and the two source registers providing the values to the ALU, R_s and R_t. This is shown in the following figure.

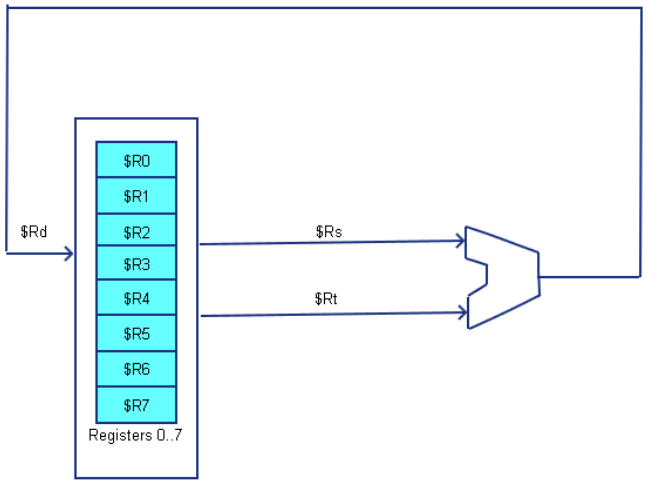


Figure 1-6: 3-address architecture

A 2-address architecture is similar to a 3-address architecture, and the only difference being that only 2 registers are specified in the instruction, the first being used for both the destination of the operation and the first source to the ALU.

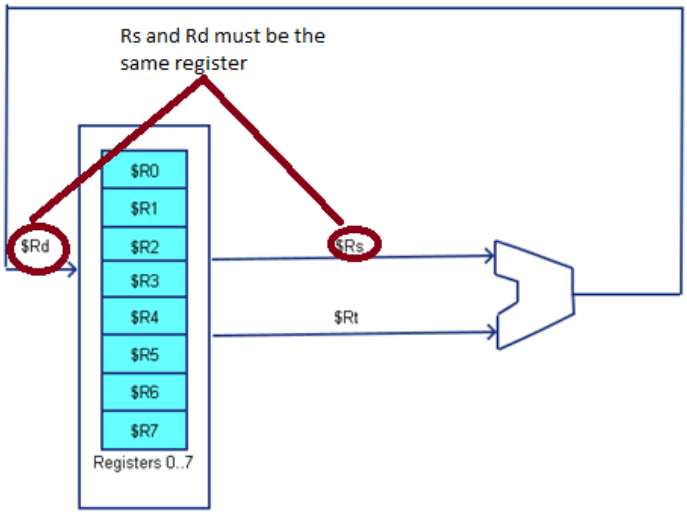


Figure 1-7: 2-address architecture

As shown in the diagrams, the CPU selects two of the general purpose registers the values to send to the ALU, and another selection is made to write the value from the ALU back to a register. In this design, all values passed to the ALU must come from a general purpose register, and the results of the ALU must be stored in a general purpose register. This requires that memory be accessed via load and store operations, and the correct name for a two/three-address computer is a "two/three-address load/store computer".

The following two programs execute the same program, B=A+5, as in the previous examples. The first example uses the 3-address format, and the second uses the 2-address format.

```
LOAD  $R0, A
LOADI $R1, 5
ADD   $R0, $R0, $R1
STORE B,   $R0
```

Program 1-4: 3-address program for adding two numbers

```
LOAD  $R0, A
LOADI $R1, 5
ADD   $R0, $R1
STORE B,   $R0
```

Program 1-5: 2-address program for adding two numbers

1.3 Von Neumann and Harvard Architectures

When discussing how memory is accessed at the CPU level, there are two designs to consider. The first is a Von Neumann architecture, and the second is a Harvard architecture. The major difference between the two architectures is that in a Von Neumann architecture all memory is

capable of storing all program elements, data and instructions; in a Harvard architecture the memory is divided into two memories, one for data and one for instructions.

For this monograph, the major issue involved in deciding which architecture to use is that some operations have to access memory both to fetch the instruction to execute, and to access data to operate on. Because memory can only be accessed once per clock cycle, in principal a Von Neumann architecture requires at least two clock cycles to execute an instruction, whereas a Harvard architecture can execute instructions in a single cycle.

The ability in a Harvard architecture to execute an instruction in a single instruction leads to a much simpler and cleaner design for a CPU than one implemented using a Von Neumann architecture. For this first monograph a Harvard implementation will be implemented. Later monographs will look at the implementation of the CPU using the Von Neumann architecture.

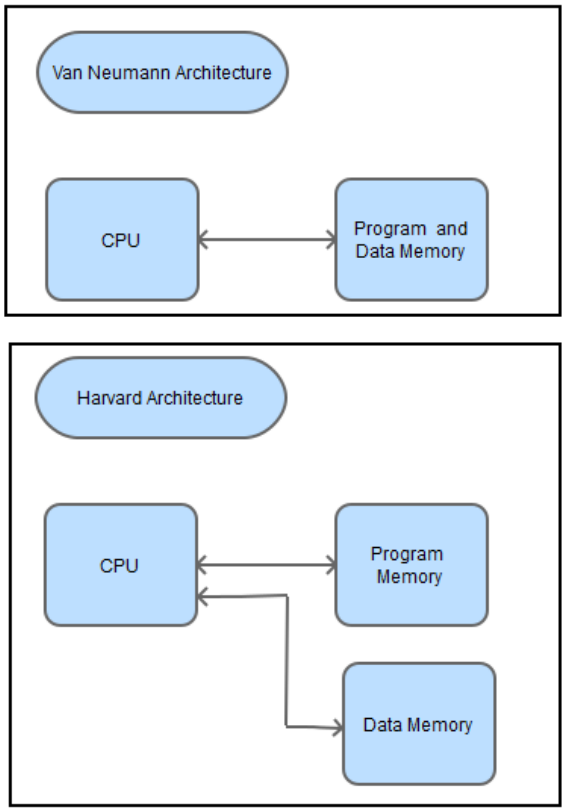


Figure 1-8: Difference between a Von Neumann and Harvard architecture

2 Assembly Language

2.1 What is Assembly Language

Assembly language is a very low level, human readable and programmable, language where each assembly language instruction corresponds to a computers machine code instruction. Assembly language programs translate directly into machine code instructions, with each assembly instruction translating into a single machine code instruction[1].

After having chosen a basic address format for the architecture, the format of the assembly language, called an Instruction Set Architecture (ISA), is defined. The next step is to design the entire assembly language that will be translated and run on the CPU.

The following are the steps in design of the CPU.

1. First, the assembly language is designed that can be used to write programs for this CPU. This language is tested by implementing simple programs in the assembly language.
2. A machine code representation for the assembly language will be written. The CPU only has the ability to interpret information that is binary data, so the assembly language needs to be translated into binary data that will be understood by the CPU.
3. The CPU is then designed to execute the machine code instructions.

This process of designing the language, creating the machine code for the language, and implementing the CPU is normally iterative, but the only final product of these steps will be included in this monograph.

The first step, the creation of the assembly language, is the topic of this chapter.

To create an assembly language there are three major constraints to the language that must be defined.

1. The definition of data that will be dealt with in this CPU need to be defined. In higher level languages this would be types, like integer, float, or string. In a CPU, types do not exist. Instead the CPU is concerned with issues such as the size of a word in the computer, and how memory addresses will be used to retrieve data.
2. A set of assembler directives to control the assembler program as it runs. Directives to define issues such as the type of memory being accessed (text or data), labels to specify addresses in the program, how to allocate and store program data, and how comments are defined.

[1] Normally this one-to-one correspondence between assembly language instructions and machine code instructions is removed. Assembly languages will define pseudo instructions that are translated into multiple machine code instructions. These pseudo instructions are designed to make it easier to write programs in the assembly language. This monograph is designed to show the reader how the CPU works, and so every assembly language instruction will correspond to a single machine language instruction.

3. A complete set of assembler instructions must be defined.

The next section of this chapter will give some caveats to programmers coming from a higher level language about issues they should consider when programming in assembly language. The 3 sections after that will define the data used in the assembler, the assembler directives, and the assembler instructions.

2.2 Assembly Language Caveats

Programmers who have learned higher level language, such as Java, C/C++, C#, or Ada, often have developed ways of thinking about a program that are inappropriate for low level languages and systems such as assembly language. This section will give some suggestions to programmers approaching assembly language for the first time.

The first thing to consider is that all instructions should implement primitive operations. Higher level languages allow a short hand that implies many instructions. For example, the statement `B=A+5` implies load operations that ready variables `A` and `5` to be sent to the ALU. Next an `add` operation by the ALU is to be performed. Finally an operation to store the result of the ALU back to variable `B` needs to be executed. In assembly the programmer must specify all of the primitive operations needed. There are no shortcuts.

The second thing to consider is that despite what you might have heard about goto statements being bad, there is no way to implement program control such as if statements or loops without using a branch instruction, which is the equivalent to a goto statement. This does not mean that structured programming constructs cannot be used effectively. If a program is confused about how to implement structured programming constructs in assembly, there is a chapter in a free book on MIPS assembly program written by the author of this monograph that explains how this can be accomplished.

The third important point about assembly language is that data has no context. In a higher level language normally the variables A and B, and the number 5, are specified as integers. The higher level language knows that these are integers, and then provides a context to interpret them. The add operation is known to be an integer add, and the compiler will generate an instruction to do the integer option and not a floating point operation. If the declaration for the numbers was changed to float, the add operation in the higher level language would be changed to a floating point add. The higher level language knows the type variables, and can provide the proper context for interpreting them.

In assembly language, there is no context for any data. Data can be an integer, a Boolean value, a floating point number, ASCII characters, or even program instructions. The assembler has no idea of a type, and simply will execute the operation specified. It is possible in assembly to do meaningless operations, such as adding two program instructions together. Assembly language will gladly let you do meaningless and completely inane things, and will in no way warn you that

it is meaningless. The assembler has no context for data, and there is no way to correct this problem because from an assembler point of view, there is no problem.

When programming in assembly language, it is important that the programmer maintain knowledge of the current program *context*. It is the programmer who knows if two data elements are integers, and thus an integer add operation is appropriate. It is up to the programmer to be aware if the values being worked with are addresses or values, and to do the proper dereferencing operations. There is nothing but the knowledge of the programmer to ensure that a program will execute correct operations on the proper datatypes.

2.3 Assembler Directives

Assembler directives are directions to the assembler to take some action or change a setting. Assembler directives do not represent instructions, and are not translated into machine code.

For this assembler, all directives begin with a "." or "#" (the comment is a #), and the directive must exist on a separate line from any other assembler directive or assembler instruction. There are 4 assembler directives and the comment tag.

- .text – The .text directive tells the assembler that the information that follows is program text (assembly instructions), and the translated machine code is to be written to the text segment of memory.

- .data – The .data directive tells the assembler that information that follows is program data. The information following a .data instruction will be data values, and will be stored in the data segment.

- .label – A label is an address in memory corresponding to either an instruction or data value. It is just a convenience so the programmer can reference an address by a name. It will be used as follow:

  .label name

  The label is a tag that can be referenced in place of an address in any assembly instruction that can take a label/adress. Labels and addresses can be used interchangeably.

- .number – The number directive tells the assembler to set aside 2 bytes of memory for a data value, and to initialize the memory to the given value. It will often be used with the .label directive to set a label to a 2-byte memory value, and initialize the value, as shown in the following code fragment.

```
.label var1
    .number 5
```

This statement allocates space for the variable `var1`, and assigns that space in memory the value 5. The data for this CPU will only work with 2-byte (16-bit) integer numbers so discrete values from -32768..32767 (inclusive) can be used. All data values are in decimal; the assembler will not recognize hex values.

- # - the # (hashtag) is used to specify a comment. Anything on a line which begins with a "#" is a comment line and ignored by the assembler.

2.4 Data types

While an assembly language has no explicit data types, there are rules for how the data is accessed and stored. This section defines the rules for data access.

In this CPU, a word is 16 bits. All memory locations are 16 bits wide, and words, not bytes, are addressable. Thus the value at the address 0 is contained in bits 0...15, the value at address 1 is contained in the bits 16...31, etc.

Each address refers to a 2-byte quantity or word. If that memory location is in data memory the value is an integer number; if the address is in text memory it is a 2-byte instruction.

There are a total of 256 memory locations (addressable words) in both the data memory and the text (program) memory. The addresses for both of these memories start at 0 and run to 255, which corresponds to an 8-bit unsigned value. Though the memory addresses overlap, the context of the request will determine which memory to use. Only the $pc will be used to access text memory, and all other addresses will refer to data memory.

When referencing values in instructions (immediate values and addresses) an 8 bit value is used. This 8-bit value can either be given as a numeric value, or as a valid label to an address somewhere in the program. When used as an address, this 8-bit value will be is unsigned, and refers to a number between 0…255. For the instructions `add`, `sub`, and `stor`, this is an addresses in the data segment. For branch instructions, `beqz`, the 8-bit address refers to the text segment.

For immediate instructions, `addi` and `subi`, the value of the operand is an 8-bit integer value, and has a value from -128…127.

2.5 Designing an Assembly Language

When designing an assembly language, a language to manipulate a CPU, there are 3 major concerns:

1. Transferring data from main memory to memory internal to the CPU (registers or operand stack).
2. The set of operations that can be performed by the ALU on the data, for example `add`, `subtract`, and, `shift`, etcetera.

3. A way to provide program control, for example to implement *branch (if)* and *looping (for* or *while)* type structures in a program. Normally control structure will be provided by a *branch* operation.

These three major concerns, and how they are addressed in the assembly language, will be discussed in the next sections. The last section of this chapter will give some programs that will illustrate how a program will be written in this assembly language.

2.5.1 Transferring data from main memory to internal CPU memory

The amount of memory directly accessible to a programmer on the CPU (e.g. registers) is very limited. In the case of the one-address architecture, only one memory slot, the `$ac`, is directly useable by a programmer. Therefore programs need to rely on main memory to store program instructions and data.

To transfer items from data memory to the $ac the instructions add, sub, and stor are used.

For the `add` and `sub` instructions, the second operand of the instruction is the label or memory address of the value to retrieved from memory and sent to the ALU. So for example, to load a value into the `$ac` from a memory location labelled `A` the following code would be used.

```
clac
add A
```

Note that the `$ac` should always be set to 0 (using the `clac`) before loading a value into the `$ac`, or the value stored in the `$ac` will be the result of adding the value at memory location A with the current value in the $ac.

For the `stor` instruction, the second operand is the label or memory address at which to `stor` the value from the `$ac`. For example, to store the value in the `$ac` to memory at the address in label `B`, the following code would be used.

```
stor B
```

2.5.2 Set of valid ALU operations

The next consideration is the set of operations which the ALU can perform on the input data. This list depends on the complexity of the ALU. The ALU in this computer is very simple, and so will only support the operations add and sub.

2.5.3 Program Control (Branching)

To do any useful program, if and loop constructs must be supported. In the implemented one-address CPU this is accomplished by the Branch-if-equal-zero (`beqz`) operation. For this operation, if the `$ac` is 0, the program will branch to the text memory address that is contained in

the branch statement. This address can be either a label representing the address, or the numeric value of the branch address. So in the following instruction, the program will branch to the address of label EndLoop if the value in the `$ac` is `0`.

```
beqz EndLoop
```

An unconditional branch statement is often used, but can be simulated by first setting the $`ac` to `0` before the branch statement. The following instruction implements an unconditional branch.

```
clac
beqz StartLoop
```

2.5.4 Assembler Instructions

Based on the criteria of the preceding section, A minimum set of assembler instructions is defined to create useful programs. These instructions are sufficient to create useful programs, and several examples will be shown at the end of this chapter.

- `add [label/address]` – Add a value from data memory (dm) to the current $ac.

  ```
  $ac <- $ac + dm[address]
  ```

  Either a label or the actual address of the value can be used in this instruction. Thus if the label `A` refers to the dm address of `5`, the following two instructions are the same.

  ```
  add A
  add 5
  ```

- `addi immediate` - Add the immediate value in this instruction to the `$ac`. The immediate value is an 8-bit integer value between -128…127

  ```
  $ac <- $ac + immediate
  ```

  An example of an `addi` instruction which adds 15 to the value in the `$ac` follows.

  ```
  addi 15
  ```

- `beqz [label/address]` – The `beqz` instruction changes the value in the Program Counter (`$pc`) to the text memory address in the instruction if the value in the `$ac` is `0`. In this CPU, the `$pc` always specifies the next instruction to execute, so this has the effect of changing the next instruction to execute to the address in the instruction. This is called a *program branch*, or simply *branch*.

  ```
  $pc <- address IF $ac is 0
  ```

  An example of the `beqz` instruction that branches to address 16 if the `$ac` is `0` follows:

  ```
  beqz 16
  ```

- `clac` – The `clac` instruction sets the $`ac` to `0`. This could be done with a set of stor and sub operations, so instruction is mostly for convenience.

  ```
  $ac <- 0
  ```

- `sub [label/address]` - Subtract a value from data memory to the current $ac.

  ```
  $ac <- $ac - dm[address]
  ```

  Either a label or the actual address of the value can be used in this instruction. Thus if the label `A` refers to the dm address of 5, the following two instructions are the same.

  ```
  sub A
  sub 5
  ```

- `subi immediate` - Sub the immediate value in this instruction to the `$ac`. The immediate value is an 8-bit integer value between -128…128.

  ```
  $ac <- $ac - immediate
  ```

  An example of an `subi` instruction which adds 15 to the value in the `$ac` follows.

  ```
  subi 15
  ```

- `stor [label/address]` – Store the current value in the `$ac` to data memory.

  ```
  dm[address] <- $ac
  ```

  Either a label or the actual address of the value can be used in this instruction. Thus if the label `A` refers to the dm address of `5`, the following two instructions are the same.

  ```
  stor A
  stor  5
  ```

- `noop` – this statement does nothing. Executing this statement does not change the value of any memory or registers (except the `$pc`) in the system. It is included so that text memory can be set to 0 and executed without changing the internal state of the computer.

2.6 Assembler Programs

The following assembly programs illustrate how the assembly language defined in this chapter can be used to implement some simple programs.

2.6.1 Loading a value into the AC

This first program loads an immediate value of `5` into the `$ac` register. After running the program, the value in the $ac will be 5.

```
.text
clac
addi 5
```

Program 2-1: Loading a value into the $ac from an immediate value

This second program loads the value from the memory address corresponding to the label `var1` into the `$ac`. Since the value at the address of `var1` is `5`, the program loads the value `5` into the `$ac`.

```
.text
clac
add var1
.data
.label var1
     .number 5
```

Program 2-2: Loading a value into the $ac memory using a label

This third program adds the value at address `0` in the data segment to the `$ac`. Since the value `5` has been loaded as the first value in the `.data` segment, the value `5` is loaded to the `$ac`.

```
.text
clac
add 0
.data
    .number 5
```

Program 2-3: Loading a value into the $ac from memory using a reference

2.6.2 Adding Two Immediate values

This program illustrates adding 2 immediate values together in the `$ac`. The `$ac` is initialize to `0`, and then the first value is loaded in the `$ac` from the immediate value of in the instruction. The immediate value in the second instruction is then added to the `$ac`, and the `$ac` contains the final result.

```
.text
clac
addi 5
addi 2
# Answer in the $ac is 7
```

Program 2-4: Adding two immediate values

2.6.3 Adding two values from memory, and storing the results

This program adds two values from memory at labels `var1` and `var2`, adds them, and stores the result back to the value at label `ans`. This program also introduces a new construct, which we will call *halt*. The halt is a set of instructions that creates an infinite loop at the end of the program so that the program does not simply continue to execute `noop` instructions until it runs out of memory. This constructs sets the `$ac` to `0`, and then branches to the same statement over and over again. The program is running, but it is not progressing the `$pc` or changing the state of the computer.

```
.text
clac
add var1
add var2
stor ans
.label halt
    clac
    beqz halt
# Answer is in data memory at address 0.
.data
.label ans
    .number 0
.label var1
     .number 5
.label var2
    .number 2
```

Program 2-5: Adding two memory values and storing back to memory

2.6.4 Multiplication by iterative addition

This program multiples two numbers by iteration. This means than n*m is calculated by adding n to itself m times, e.g. n + n + n… etc.

```
.text
# Begin the multiplication loop.
.label startLoop

# When the program is assembled the multiplier and multiplicand are
# initialized # in memory.  The counter is initialized to 0
# and incremented by 1 each time through the loop.  When the
# counter == multiplier, # the value of multiplier-counter = 0,
# and the beqz instruction branches to the end of the loop.
clac
add multiplier
sub counter
beqz endLoop  # Go to end when done

# calculate product.  The product is initially zero, but each
# time through the loop the product
```

```
# is augmented by the value of the multiplicand.  The product is
# then stored back to memory for use on the next pass.
clac
add product
add multiplicand
stor product

# The counter is incremented and the program branches
# back to the beginning of the loop for the next pass.
clac
add counter
addi 1
stor counter
clac
beqz startLoop

# When counter == multiplier, the program has completed.
# The answer is at the memory location of the label product.
.label endLoop
clac
    beqz endLoop
# result is in ans (data address 0)

.data
.label multiplicand
     .number 5
.label multiplier
    .number 4
.label counter
    .number 0
.label product
    .number 0
```

Program 2-6: Multiplication using iterative addition

3 Machine Code

Machine code is a representation of an assembly language program that the CPU hardware can understand. Since the CPU only understands binary, machine code is a binary language that controls the CPU. When we write the machine binary code, to make it easier for a human to read, the code will be collected into groups of 4 bits, and the hexadecimal (base 16) result will be written.

3.1 Overview of the machine code instruction format

All machine code instructions for our computer will consist of two 4-bit segments, and one 8-bit segment, as shown below.

Figure 3-1: 16-bit machine instruction format

The first 4-bit segment will represent the type of operation. The possible types of operations are the following:

- 0 – This is a no operation (or *noop*) instruction. It does not change the current state of the computer, and simply moves the CPU to the next instruction.
- 1 –This opcode represents an immediate operation which uses the ALU to produce a result. This instruction consists of the 4-bit opcode, a 4-bit ALU option (ALUopt) to tell the ALU what operation to execute, and an 8 bit data immediate value for an operand. As implemented the ALU only executes 2 operations, 0x0 is add and 0x1 is subtract, though the exercises at the end of the text add more operations. A maximum of 16 operations can be implemented in the CPU.

  Examples of translating these assembly instructions into machine code follow.

  The instruction:

  ```
  addi 2
  ```

  translates into the following machine code:

  ```
  0x1002
  ```

  The instruction:

```
subi 15
```

translates to the following machine code

```
0x110f
```

- 2 – This opcode represents a memory address operation which uses the ALU to produce a result. This instruction consists of the 4-bit opcode, a 4-bit ALUopt to tell the ALU what operation to execute, and an 8 bit data memory address for an operand. As implemented the ALU only executes 2 operations, 0x0 is add and 0x1 is subtract, though the exercises at the end of the text add more operations. A maximum of 16 operations can be implemented in the CPU.

  Examples of translating these assembly instructions into machine code follow.

  The instruction:

  ```
  add 2
  ```

  translates into the following machine code:

  ```
  0x2002
  ```

  The instruction

  ```
  sub 15
  ```

  translates into the following machine code

  ```
  0x210f
  ```

  Note that during assembly process labels in assembly code are translated into addresses, so labels will never appear in machine code.

- 3 – This opcode executes the clac operation (e.g. it sets the $ac to 0). In this instruction all of the subsequent bits after the 0x3 are ignored, so they can contain any value. By convention, the extra bits should always be set to 0.

  For example, the following assembly instruction

  ```
  clac
  ```

translates to

```
0x3000
```

- 4 – This opcode executes the stor operation. In this instruction the 4 bit ALU opt is not used, and should be set to 0. The address value is the address at which to store the value in the $ac. For example, the following instruction

  ```
  stor 15
  ```

  translates to:

  ```
  0x400f
  ```

- 0x5 – The opcode executes the beqz operation. In this instruction the 4 bit ALU operation is not used, and should be set to blank. The result of this operation is the $pc is set to the address value if the $ac is zero. Setting the value for the $pc causes the program to branch to that address.

  For example, the following instruction

  ```
  beqz 40
  ```

  translates to:

  ```
  0x5028
  ```

4 Assembler program

The assembler is the program that translates an assembly language program into machine code. The assembler will read a source assembly program consisting of assembly instructions. The assembler will then write two files which will be used to run the program in the Logisim CPU.

The first output file contains the machine code segment containing the machine code program that will be used in the CPU. This is a translation of the assembly instructions into machine instructions. The second output file contains the initialized data segment that will be used in the CPU. This is illustrated in the following figure.

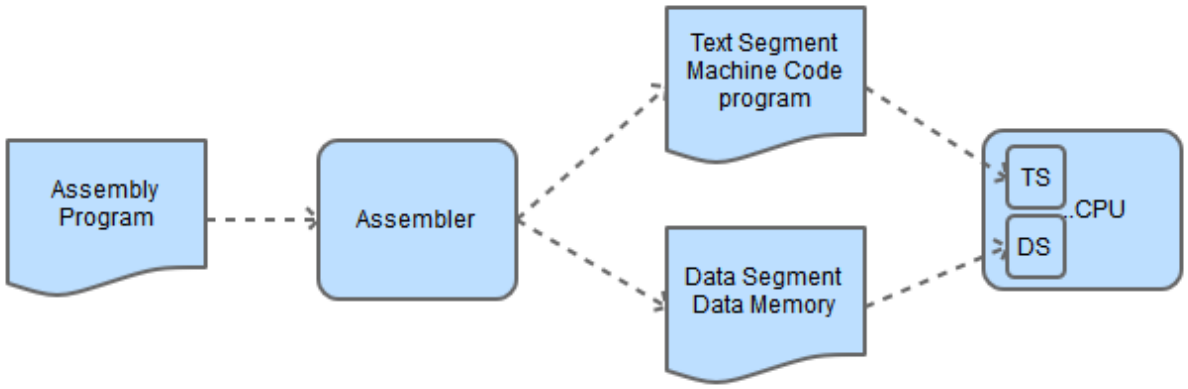


Figure 4-1: Assembly Process

The assembler is a *two-pass assembler*. A two-pass assembler reads the input file twice, or in 2 passes from the start to the end of the source assembly file. The first pass will calculate an address for each label in the program to create a symbol table. A symbol table is a list that contains labels in the program and their address in memory. The second pass will translate each instruction in the input file into machine code, and write out a file that corresponds to machine code for the data and the text segments for the program. The second pass through the file uses the symbol table to resolve any label references that are used in assembly instructions. The format of the program is shown in the following figure.

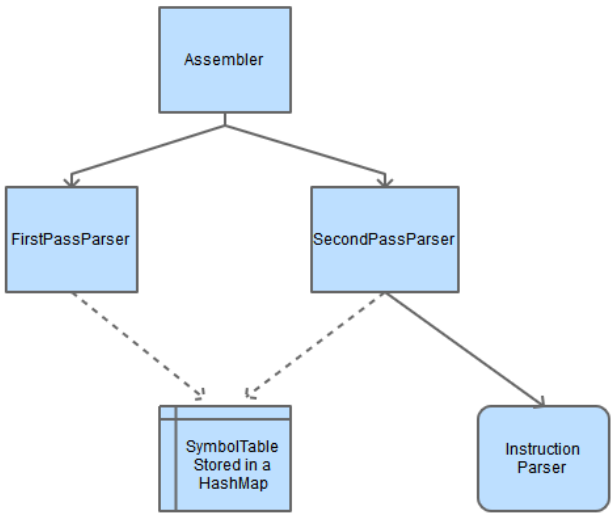


Figure 4-2: Assembler overview

The assembler is written in Java. There is no particular reason or advantage to writing this assembler in Java. The major issue in the assembler is the writing of call backs or virtual methods/functions to parse the individual instructions. These would probably be easier in C/C++, and definitely faster using function pointers instead of polymorphism. But the concept of a call back is difficult in any language for readers not familiar with the concept, and this assembler is more than fast enough.

The assembler uses static initializers with factory patterns and singleton objects to implement the polymorphic call back structure for processing the commands. This could be done in a large "if" statement, but this polymorphic solution is cleaner and easier to extend. It is also instructive for readers who are not familiar with the factory pattern, singleton pattern, and static initializers to see them. But the assembler is a relatively short and straight forward program, and readers are welcome to rewrite it using any structure or language they prefer.

The assembler can be obtained from the author's web site, http://chuckkann.com. The zip file contains the assembler, the circuit definition file for the one-address CPU, and some programs and files for the user to text and work with the CPU.

4.1 Running a program on the One-Address CPU

The following directions detail how to use the one-address CPU.

1. First download the One-AddressCPU.zip file from the author's web site, http://chuckkann.com. Unzip the file in a suitable location.

2. In the root directory of this zip is an executable JAR file named OneAddressAssembler.jar. Double click on this icon, and you should see the following screen.

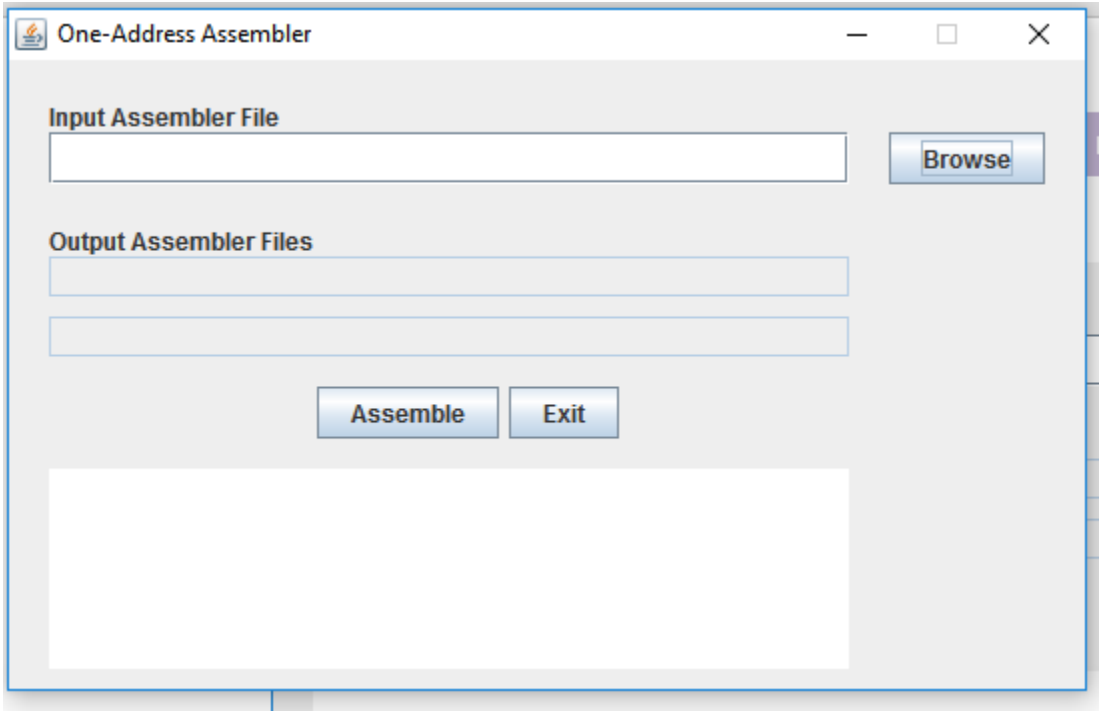


Figure 4-3: Running the assembler - step 1

3. Click the *Browse* button, and a file dialog box should appear. Select the directory AssemblyPrograms.

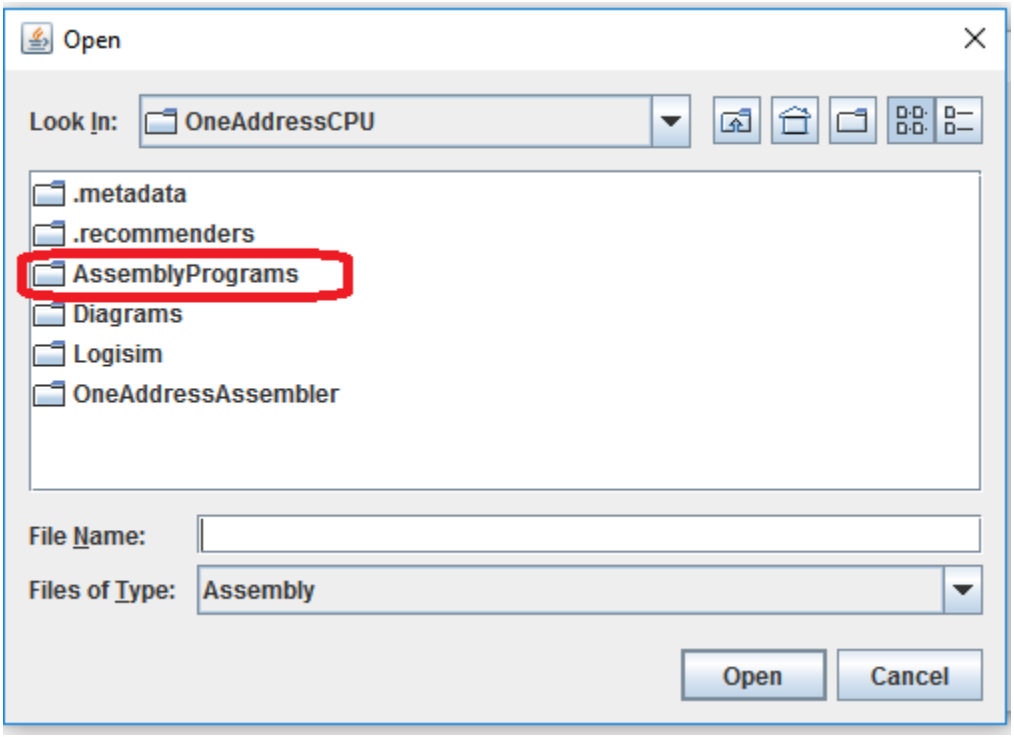


Figure 4-4: Running the assembler - step 2

4. Select the AdditionExample.asm program. This is Program 2.5 from Chapter 2, and adds two values from memory together and stores the result back in memory.

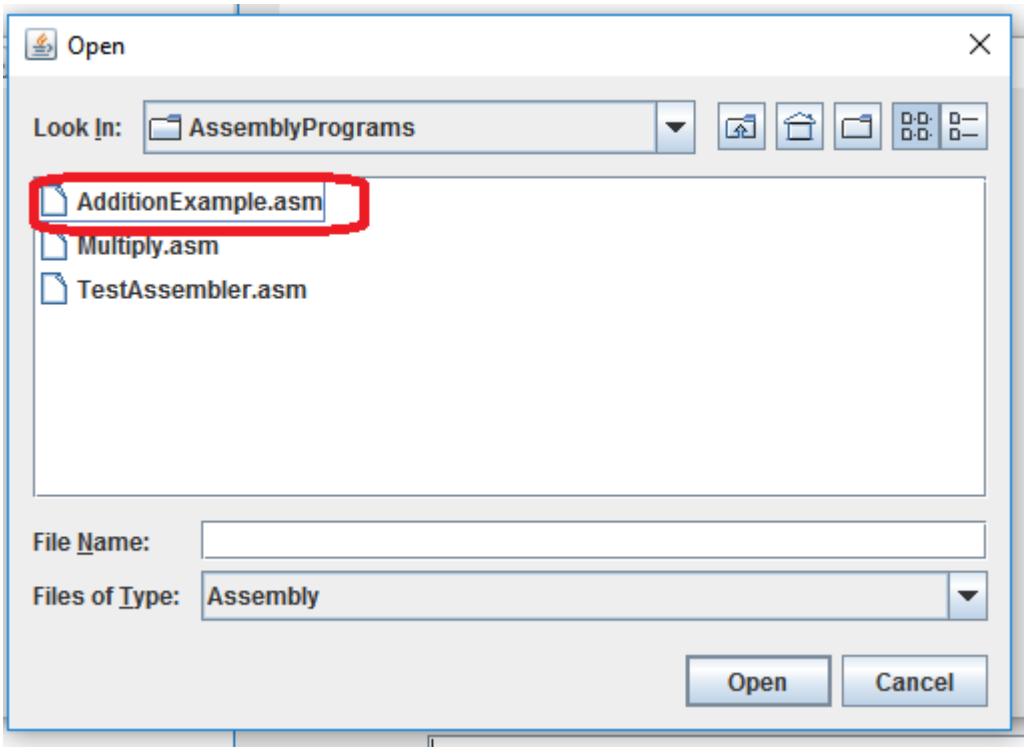


Figure 4-5: Running the assembler - step 3

5. The program will display the following screen, which says that the assembler will use as input the AdditionExample.asm file, and if it successfully completes will produce two output files, AdditionExample.mc (the machine code) and AdditionExample.dat (the data). Click the Assemble button.

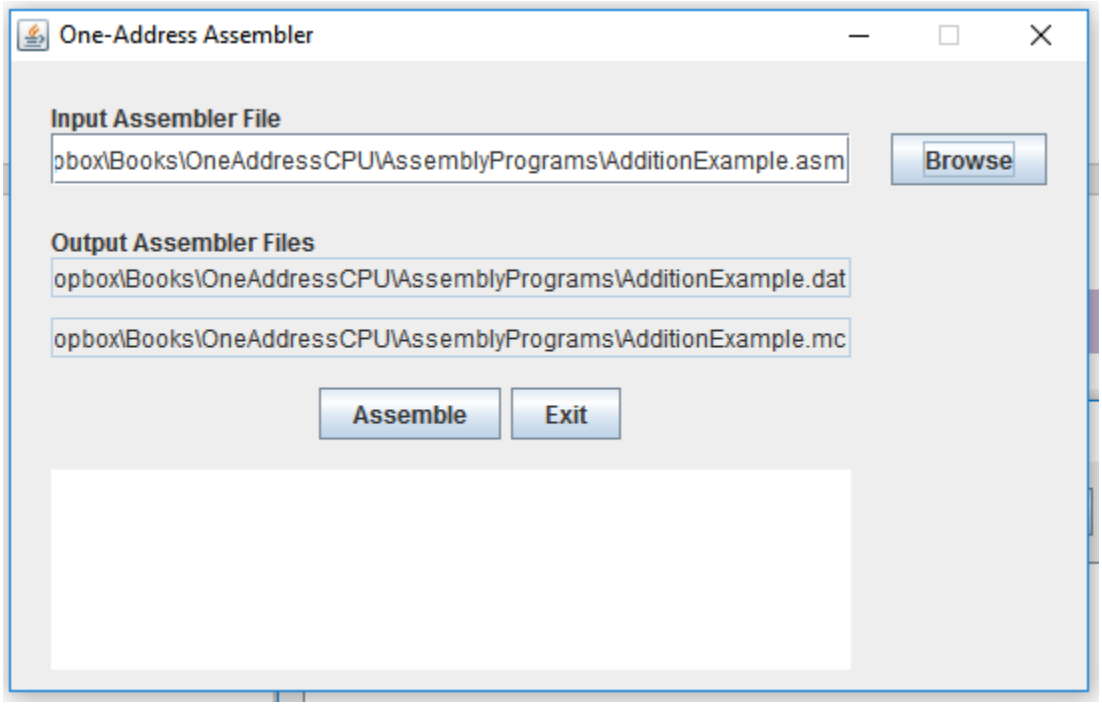


Figure 4-6: Running the assembler - step 4

6. You should get the following message that the assembly completed successfully. Click Exit to close the assembler.

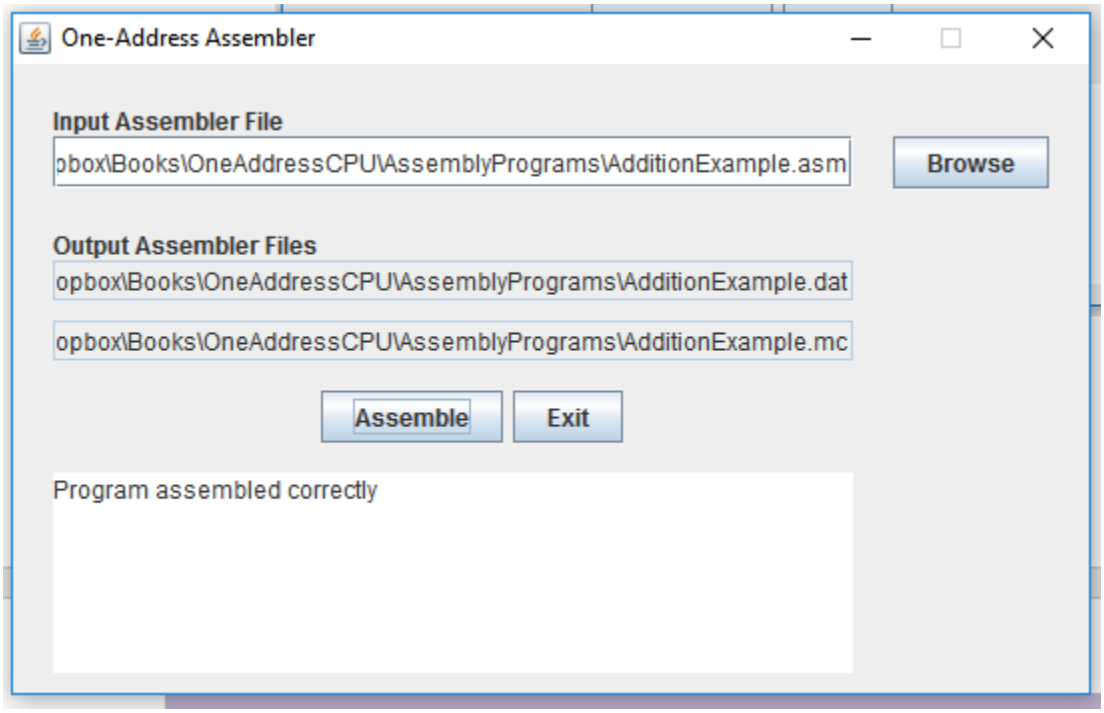


Figure 4-7: Running the assembler - step 5

7. Go to the subdirectory called *Logisim*, and click on the icon for logisim-generic-2.7.1.jar. This will open Logisim, and you should see the following screen. Choose the *File->Open* option, select the Logisim directory, and open the OneAddressCPU.circ file.

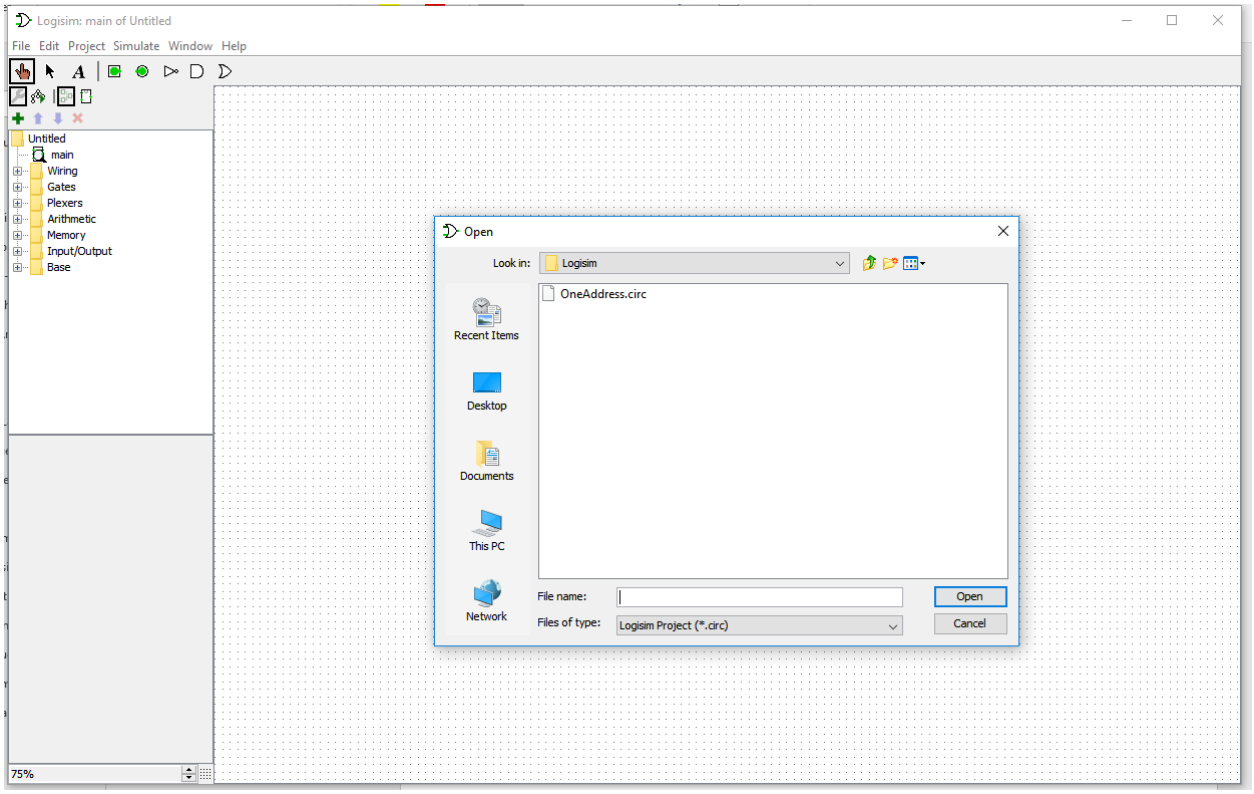


Figure 4-8: Running the CPU - step 1

8. You should now see a screen with the One-Address CPU on it. Set the Zoom size (circled in the picture) to 75% (or 50% if needed) to see the entire diagram. For now the only two areas of the CPU se are interested in are the Text Memory and the Data Memory, which are circled in the diagram.

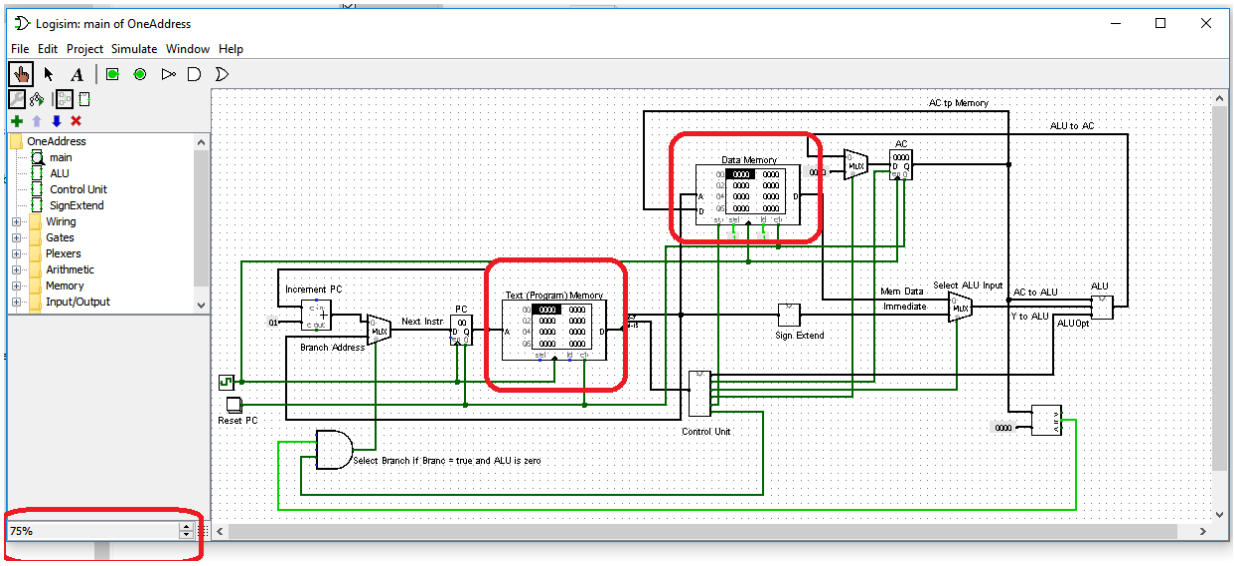


Figure 4-9 Running the CPU - step 2

9. Right click on Text Memory box, and select the *Edit Contents* option. You should see the following editor for the memory appear on your screen. Note that the input values for this memory are in hexadecimal, but we are going to fill it in from the files that were created by the assembler, and the assembler has written the files in hexadecimal.

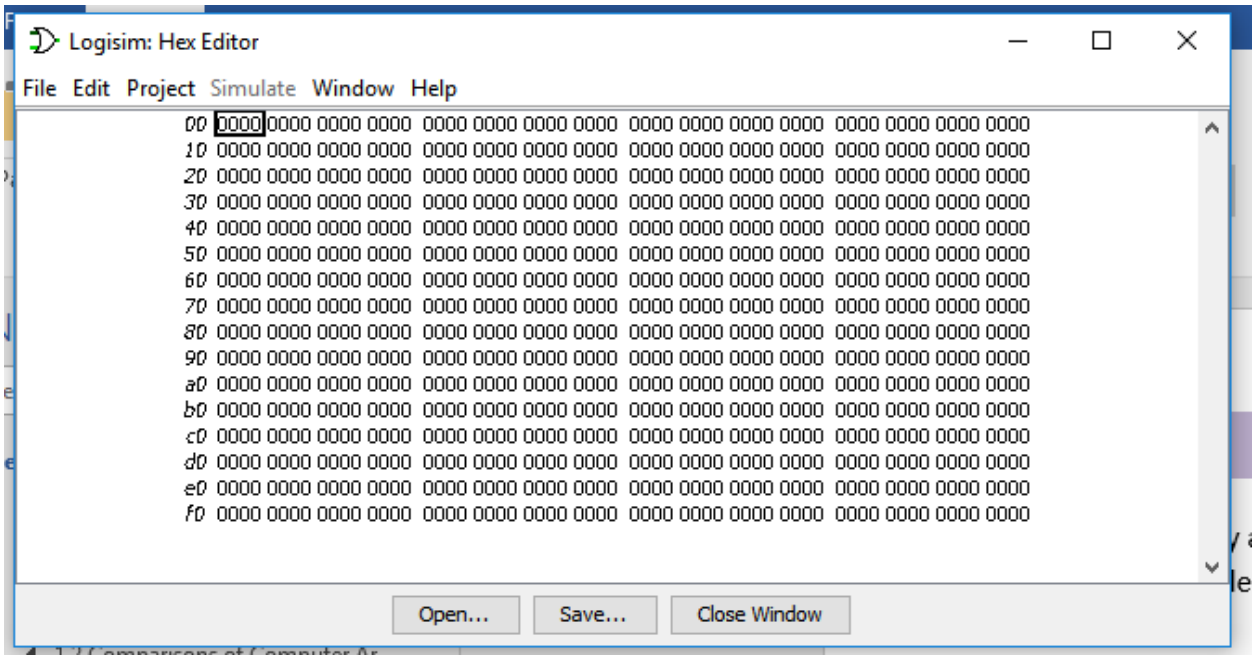


Figure 4-10: Running the CPU - step 2

10. Choose the Open option, go the AssemblyPrograms directory, and select the AdditionExample.mc file.

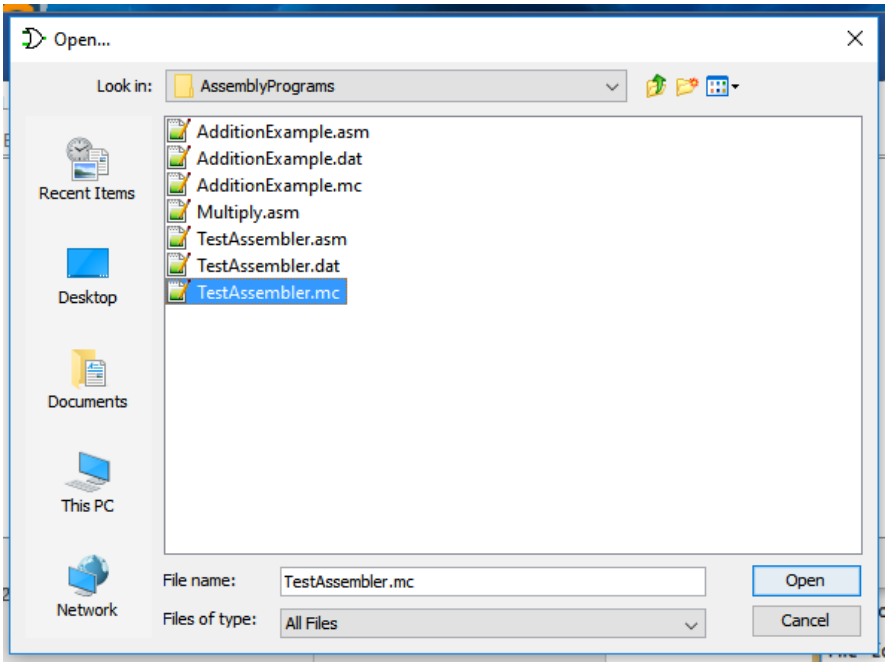


Figure 4-11: Running the CPU - step 3

11. You should now have the AdditionExample machine code in this data block, as in the following example. Select the close option, and the machine code is now available in the CPU.

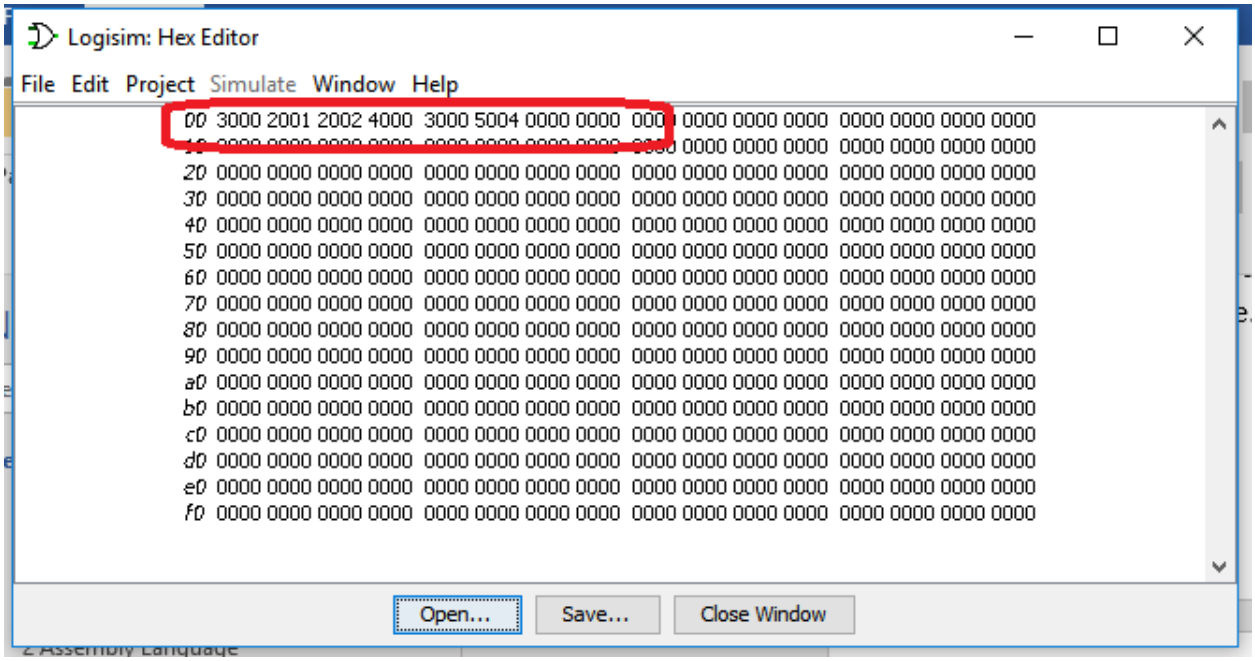


Figure 4-12: Running the CPU - step 4

12. Repeat steps 9-11 for the Data Memory using the AdditionExample.dat file. Your data and program memory should now both be initialized.

13. Clicking twice on the clock (remember that the registers and memory only change on a positive edge trigger, not a negative edge trigger), and the clac instruction will execute. This will not result in a change because the $ac is already zero. Click the clock twice more, and you should see program is at instruction 2, and the $ac change to 5.

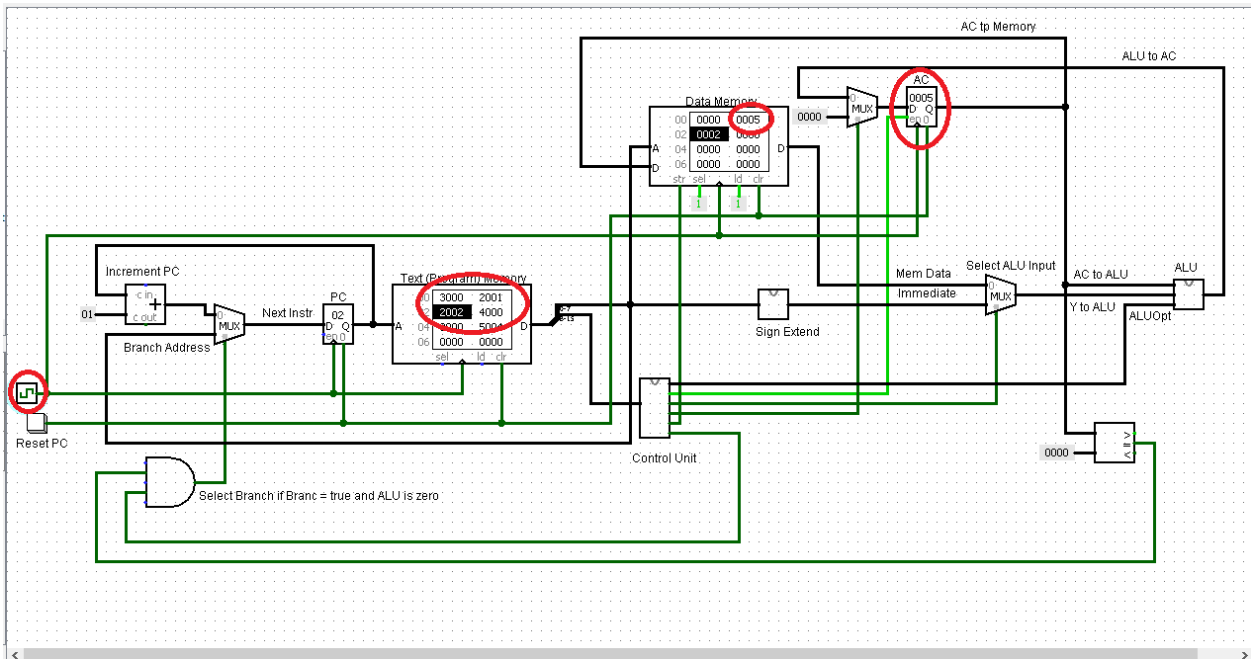


Figure 4-13: Running the CPU - step 5

14. Clicking on the clock 6 more times shows that the program has completed, and the value of 7 (5+2) has been calculated and store at memory address 0. The last two instructions, clac and beqz halt, simply put the program in an infinite loop at its end so that it does not continue to execute through the rest of memory.

In Logisim there are a number of ways to control the simulation using the *Simulation* option. Of particular use is the *tick frequency* (how often to a clock click happens), *tick enabled* (which runs the clock at the tick frequency), and *Logging* (which provides an easy way to review your results).

5 CPU implementation

This chapter will cover the Logisim implementation of the One-Address CPU. This implementation will consist of 3 Logisim subcomponents that are needed to implement the CPU, and the main component which is the CPU. The 3 subcomponents, the sign extend unit, the ALU, and the Control Unit (CU) will be explained first, and then the main component will be broken down and examined in detail.

5.1 The sign extend unit

The immediate values which can be part of an instruction are 8 bits, and can be used as an input to the ALU. However, the ALU accepts inputs which are 16 bits. Therefore, immediate values which are passed to the CPU must be expanded to fill 16 bits. The question is how to fill in the high 8 bits when expanding immediate values from 8 to 16 bits.

Remember that all the immediate values passed to the CPU are integers; the top (left-most) bit of the value determines the sign. If the 2's complement number is positive, leading 0s have no effect on the number. For example, $0101_2 = 0000\ 0101_2 = 5_{10}$. In a negative 2's complement number, leading 1s have no effect on the number. Thus $1011_2 = 1111\ 1011_2 = -5_{10}$. Thus to extend an integer the left-most bit is extended into the new binary digits. This is what the sign extend unit is doing, extending the 7^{th} bit to positions 8-15 to translate the 8-bit integer into a 16-bit integer.

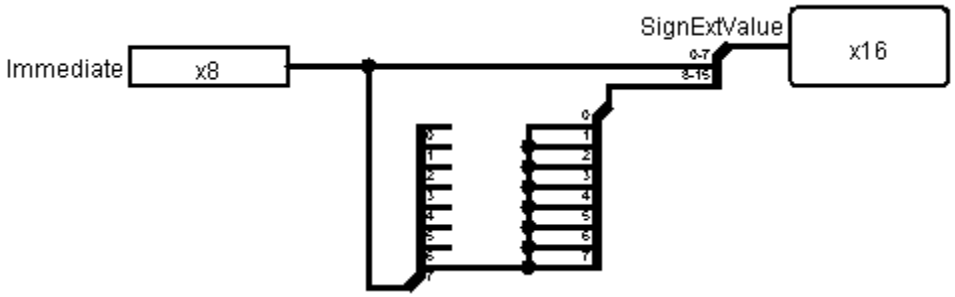


Figure 5-1: The sign extend unit

5.2 The ALU

The ALU for this unit supports addition and subtraction, and also implements a flag to tell if the current operation produced an *overflow*. And overflow occurs when two number are added which are too big to be stored in the implemented 16 bit integers in the CPU. For example, adding 27000 + 25000 = 52000, a value larger than the maximum integer that can be handled, which is 32767. Likewise -27000 + -25000 = -52000, a number that is too below the minimum integer that can be handled, which is -32768. How the ALU handles these situations will be discussed later.

A simple ALU that would implement only 16-bit addition is easy to implement, and is shown in the following figure. Two 16-bit values ($ac and Y) are sent to the CPU, and an adder is used to add the values and produce a result.

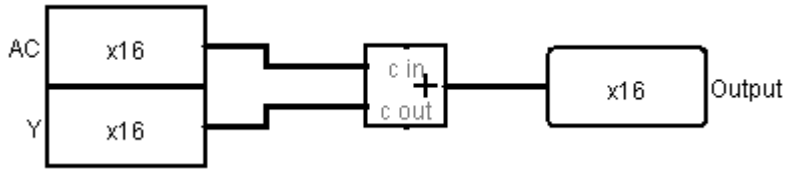


Figure 5-2: Simple Adder

To create implement subtraction, a bit of creative mathematics is used.

1. Remember that X ⊕ 0 = X; and X ⊕ 1 = X'.
2. Simple arithmetic says X + Y = X + (-Y)
3. -Y = Y' + 1 (2's complement negation operation).
4. Subtraction can be implemented by taking each bit of Y, XORing it with 1 (getting the completment), then adding 1. To add 1, pass this bit into the carry in of the adder.
5. Addition can be implemented by taking each bit of Y, XORing it with 0 (so it doesn't change) and adding 0. To add 0, pass this bit into the carry in of the adder.
6. Thus an Add/Subtract unit can be implemented by passing in a flag bit. If the bit is 0, an add operation is performed; if the bit is 1, a subtract operation is performed.

This procedure is implemented in the following Logisim circuit. Note that all it does is XOR the Y bits with the flag value 0/1, and then add the flag to the adder via the carry in to the adder.

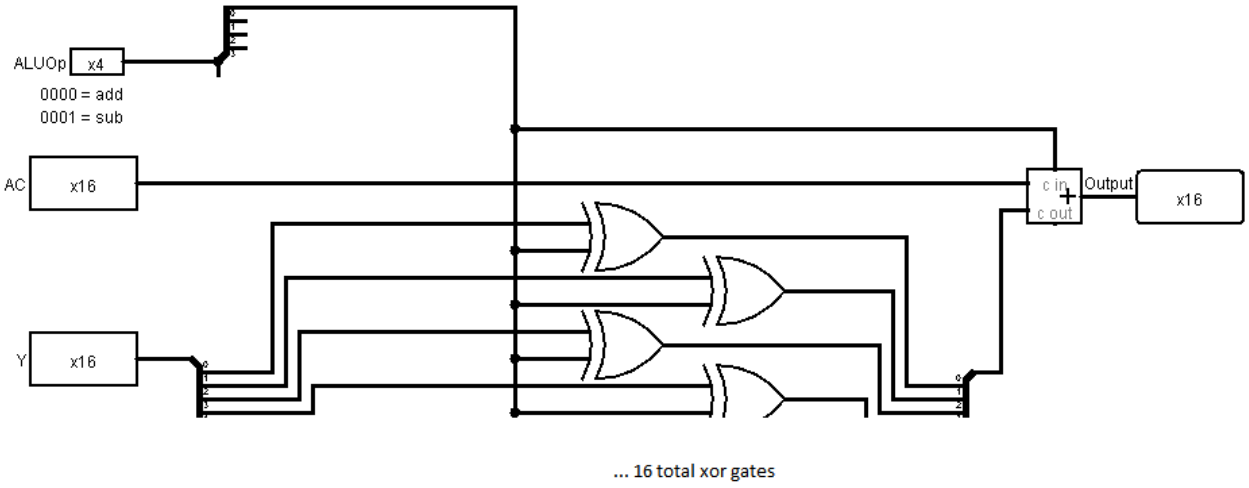


Figure 5-3: Adder/Subtracter

There is one last addition to be made to the ALU. We want to check if there is an overflow or not. The easiest way to do this is to check the carry-in and carry-out bits to the last full adder. If they are the

same, there is no overflow, but if they are different, then overflow occurred. These two bits can be checked by using an XOR gate. If they are the same (no overflow), the XOR will produce 0, and if they are different (overflow) the XOR will produce 1.

The adder from Logisim does not signal overflow, so once again some creative use of the circuit has to be done. Instead of using a 16-bit adder, the ALU will use one 15-bit adder and a 1-bit adder. This allows the checking of the carry-in and carry-out of the last bit, but requires a number of switches to get the number of lines to each component correct. However, this is the only change between the last version of the ALU, and the final version presented in the figure below.

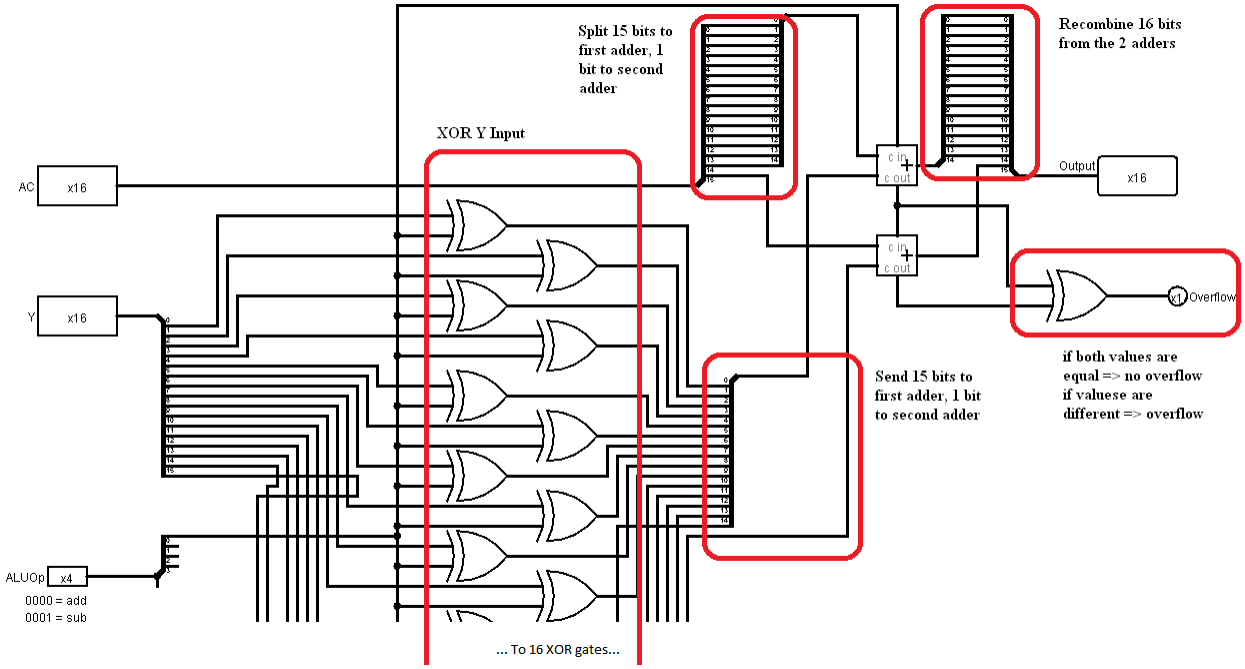


Figure 5-4: Adder/Subtracter with overflow

5.3 The Control Unit (CU)

The CU is the *brains* behind the CPU. It takes the opcode from the instruction, and sets control wires which will control how the CPU will process the instruction. Since the setting of the control wires will only make sense once the use of the control wires is understood, the explanation of the CU will be done after the CPU is explained.

5.4 The CPU

The CPU brings together all of the components into a single package to run programs. The CPU in this text consists of two subsections, and was purposefully designed to be simple enough that the control wires (other than the clock) for each subsection of the CPU are completely separate from the other.

The first subsection does all arithmetic and manages input/output to/from data memory. This subsection uses the control wires Clock, MemWr (write memory), ClrAC (clear $ac by setting it to 0),

WriteAc (write a value to $ac), ALUSrc (choose a source for the second ALU operand, either a memory or immediate value), and ALUOpt (a 4-bit value to specify what operation to run on the ALU).

The second subsection controls the execution path of the program. It uses the Clock and Beqz (branch if equal zero) control wires.

These two subsections will be looked at separately.

Before starting, a 0 in a control wire implies do nothing, hence a *noop* instruction is a 2 byte 0x0000. If a wire is not being used, by default set it to 0.

5.4.1 The CPU – Arithmetic Subsection

The arithmetic subsection of the CPU covers the $ac register, the ALU, and the data memory. This will cover the Assembly operations clac, add, addi, sub, subi, and stor. The arithmetic subsection is shown in the following diagram.

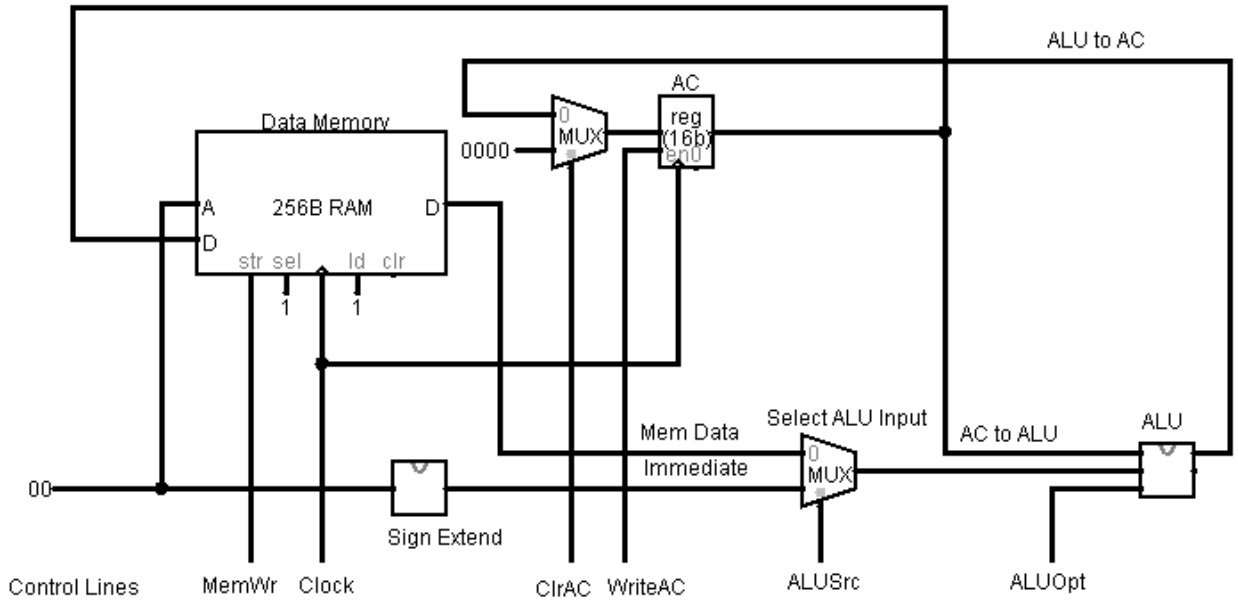


Figure 5-5: CPU - Arithmetic Subsection

The clac operation selects the constant input 0x0000 using the mux in front of the $ac to set the value of the $ac. To do this, the ClrAC line to the mux is 1 (selecting 0x0000) and WriteAC line is 1. All other control lines are 0.

For the add and sub operations, the input to the ALU is memory data, so the ALUSrc line is set to 0 to select the Mem Data. The result is stored back into the $ac, so the ClrAC line must be set to 0 to select the output from the ALU, and the WriteAC line is set to 1 to write the ALU result into the AC. The ALUOpt is set to 0000 for add, and 0001 for subtract. All other control lines are set to 0.

For the addi and subi operations, the input to the ALU is the immediate value, fo the ALUSrc line is set to 1 to select the immediate data value. All other lines are set like the add and sub operations.

For the stor operation, the MemWr is set to 1, which writes the value to the D port on the Data Memory to memory at the address specified at the A port (note the address comes from the immediate part of the instruction). All other control lines are set to 0.

Some readers might be worried that there are values passed in the CPU that are not used. For example, when the stor operation is being executed a value is still calculated in the ALU, and is sent on the wire to the $ac. However, the WriteAC line is 0, so the ALU value has no effect, and is ignored. The same is true of the Mem Data value for immediate operations like addi. The Mem Data is generated, but it is ignored as the ALUSrc chooses the immediate value. Many lines are set in every instruction, and most of them are ignored, which is why using by setting all control wires to 0 a NOOP instruction does nothing.

5.4.2 The CPU – Execution Path Subsection

The second subsection of the CPU is the execution path, which is shown in the figure below.

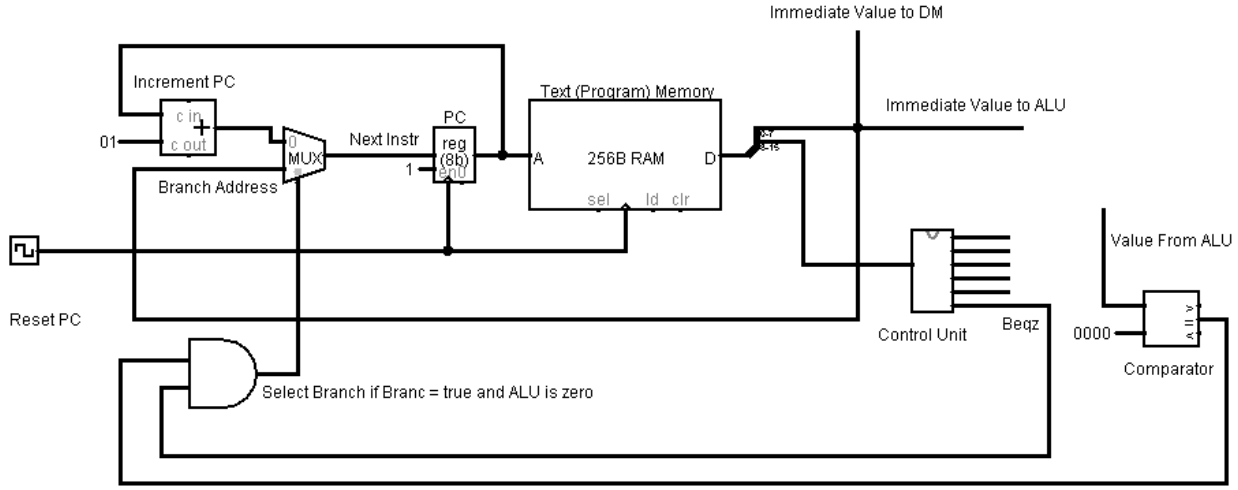


Figure 5-6: CPU - Execution Path Subsection

As this figure shows, the value of the $pc register is used to set the read address for the instruction. This instruction is then split into the control half (bits 8-15) and the immediate value (bits 0-7). The control bits are sent to the CU to set the control wires, and the immediate value is sent to the data memory or the ALU for use in the arithmetic subsection of the CPU.

Each time an instruction is executed, the $pc register is changed to point to the next instruction to execute. When the program is running sequentially, the next instruction in memory is selected by adding 1 to the $pc in the adder name Increment PC, and the multiplexor is set to 0 to select this instruction.

The only time the next instruction is not selected is if the Beqz wire is high (meaning this is a beqz instruction), AND the results of the compactor are 1 (the ALU has a value of 0). When this happens, the

mux selects the Branch Address, which is the immediate value from the instruction, and the program continues executing at the instruction at the new address.

5.5 Implementing the CU

It is now possible to specify how to set the control wires from the CU. First the ALU opt is the value of bits 8-11 of the ALU, so these are split off and sent to control the ALU.

The top 4 bits, bits 12-15, are used to set the other control wires, and from the previous discussion can be set according to the following table[2].

Operation	Code	WriteAc	ALUSrc	ClrAc	MemWr	Beqz
Immediate Operation[3]	0x1	1	1	0	0	0
Memory Operation[4]	0x2	1	0	0	0	0
clac	0x3	1	x	1	0	0
stor	0x4	0	x	x	1	0
beqz	0x5	0	x	x	0	1

Table 5-1: Operations and control wires

To implement this table, a decoder is implemented to split out the individual operations. These operations are then combined to produce the correct output behavior. The CU is shown in the figure below.

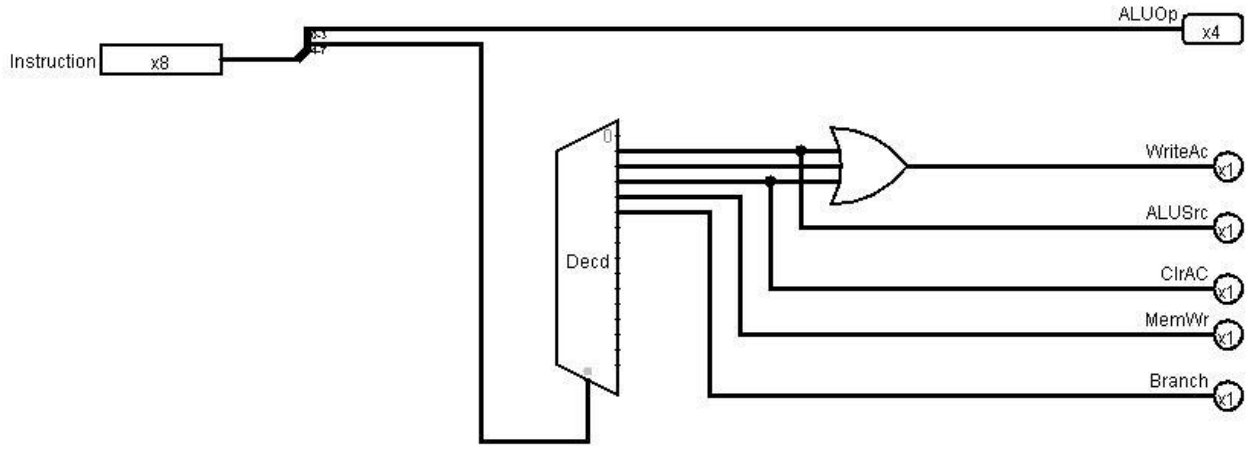


Figure 5-7: Control Unit

[2] An "x" in the table means a *don't care* condition, e.g. the value can be either 0 or 1 as it does not affect the working of the CPU. As a convention, all x values should be coded as 0.
[3] addi, subi, etc.
[4] add, sub, etc.