5-5-2021

Introduction to Assembly Language Programming: From Soup to Nuts: ARM Edition

Charles W. Kann
Gettysburg College

Introduction to Assembly Language Programming: From Soup to Nuts: ARM Edition

Description

This is an ARM Assembly Language Textbook designed to be used in classes such as Computer Organization, Operating Systems, Compilers, or any other class that needs to provide the students with a overall of Arm Assembly Language. As with all Soup to Nuts books, it is intended to be a resource where each chapter builds on the material from previous chapters, and leads the reader from a rudimentary knowledge of assembly language to a point where they can use it in their studies.

Keywords

ARM, ARM CPU, CPU, Assembly, Assembly Language, Machine Code, ARM Machine Code, Operand2, Recursion, Array Programming, Procedural Programming, Raspberry Pi

Disciplines

Systems Architecture

Introduction to Assembly Language Programming From Soup to Nuts:

ARM Edition

Charles W. Kann

This book is available for free download from:

https://cupola.gettysburg.edu/oer/8

Additional resources for the book are available at:

http://chuckkann.com/ArmAssembly

Last Update: May 1, 2021

Other free 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

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

Kann, Charles W., "Programming for the Web: From Soup to Nuts: Implementing a complete GIS web page using HTML5, CSS, JavaScript, Node.js, MongoDB, and Open Layers." (2018). *Open Textbooks*. 5.
https://cupola.gettysburg.edu/oer/5

Acknowledgments

This textbook would not be what it is today with the help of a number of students who worked on projects for it. I would like to acknowledge them here.

Editing was done by Eugene Krug, who provided a welcome second set of eyes.

The initial idea for the spreadsheet for machine code was done by Raj Chauhan, though I have significantly modified it since.

The Style Guide was done by Aditya Bajaj and Lina Johnson.

The animations to explain how the assembly code instructions work on the CPU were done by David Na and Edem Bamezon

A short answer manual was done by Aditya Bajaj and Lina Johnson.

A longer answer manual was written by Megan McDonough, Rochelle Manongdo, and Daniel Lopez

Two ARM hacking attacks that can be used by faculty to illustrate how to hack ARM executable files were written by Chuck Norris and Luke Craig.

If I have forgotten anyone, please let me know and I will update this list.

Forward

This textbook is a result of my needing to convert a Computer Organization class from MIPS to ARM. Not knowing ARM well enough to proper teach the class, I did what I tend to do, I wrote a book to force myself to learn it.

In the process I could not find any good textbooks on ARM Assembly. Most of the ARM Assembly language books I could find were concerned with topics such as accessing the GPIO ports on a Raspberry Pi. These types of topics are very useful to someone who has a basic knowledge of some assembly language, but they are not textbooks that can be used to teach the topics typically covered using assembly language in a course such as Computer Organization, nor did they cover the material a student generally needs to know when implementing material from another class such as Operating Systems, Compilers, or Computer Security. This textbook was written to help fill that gap.

Another feature that makes this book interesting is that it is the subject of a number of projects from my Computer Architecture class. The Computer Architecture class I teach is a graduate level class, so I believe that students should have to do projects with the class. Coming up with topics is sometimes an issue, and the past several semesters I have given the students the option to create resources to use along with the book. Several have taken up the offer, and there are resources now that include animations, interesting problems, coding style guides, and extra resources with the book. I have found these to be useful, interesting, and well done, and they are included and can be downloaded for free from my web site.

Another anomaly about this textbook is that it is free. I am not an anti-capitalist, but I believe that capitalism, taken to the extreme, is a very bad idea. Having more is the surest way to dissatisfaction I know of. I believe the secrete to having enough is not having more, but needing less. I am retired, and I teach, and my life is wonderful. I have enough, and I want to help those who do not. Nothing makes me happier than to see downloads of my books from community colleges, where often the students are struggling to reach the first rung in a ladder to a better life. I do suggest a $50.00 donation to the scholarship fund of a community college or local food bank at some point in your life if you find the book useful.

Finally, the book was rushed, especially in the last few chapters. The chapter on machine code took much longer than I expected due to the lack of good material for it. There is a spread sheet and illustration with the extra resources that I developed that students seem to find useful, but it still took weeks to write. Chapter 11 is marked as TBD. I hope to get back to all of this someday soon and make it nicer, but for now the textbook is usable, and so I am releasing it. I hope you find it useful, and well worth the price!

Chapter 1 Introduction

Chapter 1.1 What this textbook is about

This book intends to be a first book for students and other readers interested in assembly language; it is intended to take the user from *soup to nuts* in that it will show how to build their first ARM assembly language program and take them all the way until they understand enough about ARM assembly to be able to continue the study of assembly on their own. This makes it an appropriate textbook for either a primary textbook in an ARM assembly language class, as a secondary or supplemental textbook for a class on Computer Organization, or any number of other classes where a basic knowledge of assembly language is useful but not part of the core material, such as a class in Computer Architecture, Operating Systems, Computer Security, Compilers, Reverse Engineering, etc.

This textbook is written so the concepts taught are generally applicable to any assembly language, but since assembly languages are so different, it is important to note that this textbook is specifically about the ARM assembly language. And since assembly language exposes the Central Processing Unit (CPU) architecture, the architecture that will be discussed will be a simplified version of the ARM architecture that this textbook will call the Multi-Stage CPU (MSCPU). Note that all of the ARM assembly operations presented in this book are correct, though just a subset of the entire ARM assembly operations and formats. Likewise, the MSCPU represents a working subset of the ARM CPU and is not intended to implement a complete ARM CPU design. It is an illustrative model of a non-existent CPU, but a CPU that allows the assembly language to be implemented and explained in a working Logisim implemented model for the CPU.

This textbook is designed to be used as part of a Computer Organization class, or as a supplement for a course where students need to know assembly but the teaching of assembly is not intended to be a significant part of the class. For many students and programmers, this will be their only exposure to assembly language and the underlying hardware it represents, so it is important that this textbook not teach just the instructions, but to fill in the gap for these readers between a high level language (HLL) and the computer that will run the program.

So, while assembly language and low level hardware concepts are the main thrust of this textbook, the effect of these concepts and principals in HLL such as Java, Python, and C/C++ will also be an emphasized. It will cover a number of general computer science concepts such as:

1 The difference between values and references (pointers). The book will also highlight the difference between reference types and value types. These concepts are important in all HLL and often not well understood by students studying CS. This lack of understanding possibly comes from both values and references being values in a HLL, muddying the distinction between the two types of access. Assembly language directly accesses the

data on a computer, exposing the difference between a reference and value and requires the user to consider this difference when accessing the data. This makes assembly language the perfect language for teaching these concepts.

2 Program memory types and the difference segments of memory. These include the static or bss (data) segment, text (instruction) segment, heap segment, and stack memory. Proper understanding of these segments affects HLL in many areas including a correct understanding of program scoping and lifetime, program safety (correctness), concurrent execution of a program, among a myriad of other issues that often result in program errors that are difficult to understand and find.

3 Translation of the HLL to assembly language and a basic understanding of how the assembly language is executed in hardware. This allows the programmer to see their programs in the broader context of how a computer works, not simply the abstract model of the HLL in which the program is written.

4 Interfacing between C/C++ and assembly language. This is important to many programmers because most of the programs users will write for the ARM chip will be largely written in a HLL, with small, critical pieces in of the program in assembly. In addition, there are useful tools in C/C++ that will be accessed in this textbook from the earliest chapters. Therefore, it is important to know how to communicate between C/C++ and assembly.

5 Translating assembly code to machine code. The reason this is included is every useful program must be translated into a format that the CPU can understand. The CPU can understand only machine code, which is assembly code translated into a code of 0's and 1's that the CPU can understand. This machine code format will help the user understand how the program is instantiated on the CPU.

6 Programming plans and how they lead to properly structured code. Programming plans, or cognitive schema that are instantiated in structure code blocks in programs, are important for programming in any language. Many students have issues of program structure through much of their academic studies. This book will emphasize how to properly structure code and enforce that structure in assembly language, a language that always is tempting you to break rules to ill effects.

7 The use of the program stack to implement program abstraction and recursion. Stack are a part of most modern computer languages, and to understand their implementation helps programmers to understand their proper usage. Further, recursion is a central component to many algorithms and data structures, and by presenting how recursion is implemented at the assembly level the reader will hopefully gain a larger insight into the principal.

8 How different types of data are stored, with an emphasis on dereferencing operations using an array and struct and implementation of C-type null terminated arrays.

9 The steps needed to produce a program, specifically the compile (or assemble), link, and execute steps. What is done in each step and the output produced will be discussed.

Finally, tools such as `objdump` and `readelf` will be covered to help students understand the structure of an executable file and how it relates to assembly language programs.

The target audience for this textbook is someone who has had at least one semester of programming in a HLL, but who has never programmed in assembly language and wants to know how assembly works and what happens to a program written in a HLL to allow it to run directly on a CPU. As such, it will fill a gap that exists for students who intend to study topics such as Operating System or Computer Architecture.

In addition, but knowing how a CPU works, a programmer can understand how the CPU fits into the overall implementation of a program or larger system architecture. Knowing assembly creates better programmers and designers.

It is also important to point out what this textbook is not. This textbook is not intended to cover all the relevant information for someone who is implementing a program in ARM assembly. For example, it does not cover floating point operations, Thumb or 64-bit ARM assembly, or how to access General Purpose Input/Output (GPIO) ports. While these are important, they do not fit into the statement of purpose for this textbook.

Chapter 1.2 Intended audience

This textbook is appropriate for anyone who has never programmed in assembly language and wants to know how assembly works. Suggested readers could include:

1.1 Students who are taking an ARM assembly language class. This would not be limited to students taking a traditional semester long class at a college or university, but this book could also be given to students who are learning assembly through independent study, a Massively Open Online Classes (MOOC), tutoring, or even as a supplemental text that a student can access for a traditional class.

1.2 Students who need a background in ARM assembly for some other course. The emphasis of the other class would not be on assembly language, but would require a knowledge more than the students currently has.

1.3 Professional programmers who want to increase their marketability by learning ARM assembly.

1.4 Enthusiasts who simply want to learn assembly as they are playing with their ARM Computers, such as a Raspberry Pi.

1.5 Finally, this assembly book could be used as an introductory book on programming. While this would not be the easiest language to learn with, there may be programmers who would like this option, and I think this book would fit that option better than most assembly language textbooks.

This textbook is not intended to be used to become a professional ARM assembly language programmer. It instead seeks to show how the CPU fits into the overall design of a system and how knowing assembly creates a better programmer. It is hoped that the reader will learn

concepts are vital to understanding any HLL, but not always explicit and clear to those learning a HLL. Some of these concepts are important when making decisions about system level design or implementation of a program, and thus will enhance the eventual career of the person who knows them.

Chapter 1.3 What you need and should know

This book is about ARM assembly language, but unlike many other introductory books in a number of ways. First it is not designed to just cover the assembly language, like many books on assembly that use simulators. This textbook was written to cover the entire life cycle of assembly language programming, including creating object and executable files.

This textbook was written to run on a real OS, Raspberry Pi OS (previously called Raspbian) with tools used by real programmers. As such, it requires that the user have a computer that runs the OS on an ARM computer, such as a Raspberry Pi. The book is written for what is believed to be the lowest common denominator processor board, a Raspberry Pi Zero W, which can be currently obtained from any number of retailers for less than $35 US. The Pi Zero W is a minimum configuration that is needed, and any more capable implementation of a board with an ARM CPU should work, including any Pi higher than a zero[1].

The material in this textbook will apply to using ARM assembly using the GNU assembler and linker, both of which are run with the `gcc` command. There are other assemblers and linkers available for ARM assembly, and Raspberry Pi comes with the `as` and `ld` commands that also do assembly and linking. However, the `gcc` command allows input and output to be easily implemented using the `printf` and `scanf` functions and so was chosen for this text.

This textbook also does not cover basic background material that is not germane to the topic of understanding assembly language. The following skills are strongly suggested for anyone reading this book, and there are any number of good textbooks that cover these topics.

1 A working ARM board, as suggested above at least at the level of a Raspberry Pi Zero W. There are many web sites that do an excellent job of explaining how to boot a Raspberry Pi, and the configuration you choose to use is up to the individual user. All that is required for this textbook is that a shell prompt can be opened.

2 A basic understanding of Linux shell commands and the Linux file system. How to use a command prompt is a clear advantage to anyone intending to make a career in nearly any IT field, and studying assembly with this text is a good start at understanding the shell prompt.

3 A knowledge of a Linux editor such as `Vim` or `Emacs`. If for some reason the reader does not know or want to learn `Vim` or `Emacs`, it is possible to use an editor like nano, a GUI editor using X-windows on the ARM computer, or by transferring the files from the PI to

[1]Be aware note that the Pi Zero does not run the most recent ARMv7-A architecture. If you want to use the computer for this class for any project after this semester, you would be well advised to buy a (likely) more expensive development board that is more suitable to longer term use.

a Windows or MAC based computer with a text editor[2]. But a working knowledge of a Unix editor is a skill that most IT professionals should have, so the use of `Vim` or `Emacs` is highly recommended.

4 At least a minimum of one semester of knowledge of programming in some language. The language is not of concern, but the reader should understand branching using if-else logic, looping using for and while looping, and some form of function abstraction.

While it is possible to succeed in this class while learning skills, this textbook will not cover this information, and the reader not conversant in these skills should plan for extra time to learn them. In the long run though, the extra time learning the skills will be time well spent.

Chapter 1.4 What is Assembly Language?

Most readers have probably programmed in a HLL like C/C++ or Java. These are called *compiled languages* because there is a program, called a *compiler*, that takes the information that is contained in a source code file and compiles it together with lots of other information and produces an output that the CPU can understand. For example, in a HLL the data type is remembered for variables, and when an operator is applied to the variable the compiler figures out the correct type of operator (e.g., integer add, floating point add) to use for that variable type.

An assembly language is different in that it is intended to directly control the CPU. When programming in assembly language it important to remember that assembly language does not keep a lot of information about the program as is done in a HLL with a compiler. The programmer must implement the correct type of operator (e.g., `ADD`, `VADD`) that they want to use. An assembler will happily apply a floating point addition (`VADD`) to two integer registers, or vice-versa. It is up to the programmer to remember the data types of the registers and use the correct operation.

An assembly language is different from a HLL in other ways. In a HLL, each statement can correspond to many assembly statements, but each assembly language statement normally corresponds to a single CPU instruction[3]. So, there is a correspondence between assembly language and control of the CPU. This means that each type of CPU needs its own assembly language, and the assembly language are different for each of different CPU.

Assembly languages are designed for different purposes than HLL and thus are used differently. In assembly language an *assembler* takes assembly statements and simply converts them into

[2]This requirement means the editor on the Windows or MAC computer must be a text editor, not an editor that edits text documents (such as Word). In addition, the editor must be able to use Unix newline characters (0xa), not Mac (0xd) or Windows(0xda) characters. Word processing programs, such as Word, are NOT text editors. They will insert all sort of command junk in the file which will cause your compiler to crash badly. Many (if not most) editors that appear to do text editing still insert command characters. There are any number of free text editors for Windows and MAC that can be used, and your instructor should be able to help you choose an appropriate one.

[3]There are some pseudo instructions that can translate to sets of instructions, but these are fixed translations, and all the translated instructions are single CPU instructions.

instructions for the CPU. The instruction must give the correct operation for the given operation types. This gives the programmer a lot of control over the CPU, but it also means that the code must be very detailed, and the user must have some concept of how the CPU works. It also means that the code can be cryptic, and programs can quickly get out of hand. Assembly language programming is really not a good option when a HLL solution can be made to work, but knowledge of assembly language is important for most programmers to understand programs in a HLL. One good rule of thumb is that a programmer should understand one level of abstraction lower than the tools and programs they are working with. A React programmer should understand JavaScript and HTML. An SQL programmer should know a HLL language with loops and if tests. And a programmer in a HLL should know assembly language. That way when problems occur, they are easier to understand and fix.

Chapter 1.5 Format for using this textbook

The rest of this textbook is broken down as follows. Chapter 2 gives an overview of binary representation. Knowing binary numbering systems, including binary whole number, character data, and 2's complement (integer) numbers, is an essential skill in being able to data in a CPU. Understanding how to do arithmetic in 2's complement is vital to being able to read and understand the arithmetic that is going on inside a computer. Being able to do simple multiplication and subtraction using shift operations are skills every programmer should understand. Finally knowing and using binary and shift operations are at the heart of every CPU and many algorithms and used frequently in this textbook. These are the topics of Chapter 2, and a basic knowledge of this material is essential to understanding the rest of the textbook.

Chapter 3 starts the journey of learning ARM Assembly. This chapter starts by creating a HelloWorld program. The assembly language program is written, compiled and linked (using the `make` command), and run from the shell prompt. How to prompt for and print strings and integers is covered. Finally, an introduction to the `gdb` debugger's text user interface (`gdbtui`) is given to show how to visualize and debug running programs.

Chapter 4 builds on the ARM Assembly operations by restricting the operation to a 3-Address format and showing how they would be executed on a hypothetical 3-Address Load/Store CPU. The operations are all valid ARM Assembly operations, but they are purposefully restricted to provide a simplified instruction set that can be built on in Chapter 5.

Chapter 5 builds on the 3-Address CPU to create the MSCPU, which allows the ARM Flexible Operand or Operand2 to be used in the instructions, as well as some other instructions such as the Multiply and Accumulation (MLA) instruction.

Chapter 6 introduces the concept of machine code. The chapter covers how to translates the ARM Assembly instructions from Chapter 3 into machine code using the various machine code

formats needed by the 3-Address CPU. It will also cover how to interpret machine code and how to translate machine code back into 3-Address CPU assembly instructions.

Chapter 7 covers functions and program control flow using the PC. Branching to a function by changing the `pc` to the address of the first instruction is discusses and illustrated. The problem of returning from a function is presented, and the return of the function using the `lr` is shown. Finally the register conventions for ARM are given.

Chapter 8 introduces procedural block structured programming, which is the type of programming most readers are familiar with though they probably do non know it. This style of programming treats blocks of code as statements, and implements a standard structure for branching and looping blocks. How to properly structure a program using procedural block structure is covered in detail, and the ways to avoid *spaghetti code*, or unstructured code, is shown.

Chapter 9 covers recursion, and how to convert from a HLL to assembly for recursion is shown. By implementing recursion in ARM Assembly the concept and purpose of the stack is made more transparent and more clear.

Chapter 10 is about a type of multi-valued variable called an array. How arrays are implement and accessed in ARM Assembly is discussed and illustrated. Also covered is the concept of multi-valued variables and dereference operators, and the relationship between a value and a reference is explained.

What you will learn

In this chapter you will learn:

1 what binary numbers are and how they relate to computer hardware

2 to translate to/from binary, decimal, and hexadecimal

3 binary character data representation in ASCII

4 binary 2's complement format, which is the format used for integers in most computers

5 arithmetic operations for integer numbers

6 binary logic operations

7 the effect of context on data values in a computer

Chapter 2 Review of Binary Arithmetic

One of the major goals of computer science is to use abstraction to insulate the users from how the computer works. For instance, computers can interpret speech and use natural language processing to allow novice users to perform some pretty amazing tasks. Even programming languages are written to enhance the ability of the person writing the code to create and support programs, and a goal of most modern languages and systems is to be hardware agnostic.

Abstraction is a very positive goal, but it hides the fact that some level all computers are just machines. While HLLs abstract and hide the underlying hardware, they must be translated into assembly language to use the hardware. Everything in a computer, even seemingly intelligent operations like understanding speech, is run as a set of mechanical steps on a very small machine called a CPU.

One of the goals of a computer science education is to allow a student to at least metaphorically understand the existence and purpose of these levels; to strip away all of the levels of abstraction and make the workings of the computing machine explicit. Without an understanding of a computer as a machine, even the best programmer, system administrator, support staff, etc., will have significant gaps in what they are able to accomplish. A basic understanding of hardware is important to any computer professional.

Learning assembly language is different than learning a HLL. Assembly language is intended to directly manipulate (or mechanically control) the hardware that a program is run on. It does not rely on the ability to abstract behavior, instead giving the ability to specify exactly how the hardware is to work to the programmer. Therefore, it uses a very different vocabulary than a HLL. That vocabulary is not composed of statements, variables and numbers but of operations, instructions, addresses, and bits.

In assembly it is important to remember that the actual hardware to be used only understands binary values 0 and 1. To begin studying assembly, the reader must understand the basics of binary and how it is used in assembly language programming. The chapter is written to help the reader with the concepts of binary numbers.

Chapter 2.1 Binary Numbers

Chapter 2.1.1 Values for Binary Numbers

Many students will have had a class in CS, Philosophy, or Mathematics covering Logic or Boolean algebra. This class will have used binary values are generally true(T) and false(F) and use special symbols such as "^" for AND and "∨" for OR. This might be fine for mathematics and logic, but is hopelessly inadequate for the engineering task of creating computer machines and languages.

To begin, the physical implementation of a binary value in a CPU's hardware, called a bit, is implemented as a circuit called a *flip-flop* or *latch*. A flip-flop maintains a voltage of either a ground or a positive supply voltage, which is a voltage over ground, called the Voltage at Common Collector (VCC). If the output voltage of the flip-flip is equal to the ground, the flip-flop stores a logical T; when the output voltage of the flip-flop is VCC, the flip-flop stores a logical value of F. While most older Integrated Circuit (IC) components used +5V VCC for true, many modern ICs use +3.3V VCC. There are many different values that are used for VCC in circuits, some with a VCC less than +1V.

But the binary values of true and false really only have meaning when representing logical equations. For computation the binary values of 1 (for true) and 0 (for false) are much more common, as these can be combined easily to create larger values. The most prevalent use of binary data in this textbook will be binary values of 0 and 1.

If a flip-flop maintains a value of 1, it is also called *high*, since the voltage is positive, and *low* if the value is 0. If a switch or gate is used, a high voltage is called *On* or *Open*, and *Off* or *Closed* if the value voltage is low. Thus, while computers work with binary, there are a number of ways we can talk about binary. If the discussion is about memory, the value is *high*, *on*, or *1*. When the purpose is to describe a gate, it is *open/closed*. If there are logical operations, values will be *true/false*. The following table summarizes the binary value naming conventions.

T/F	Number	Switch	Voltage	Gate
F	0	Off	Low	Closed
T	1	On	High	Open

Table 2-1: Various names for binary values

In addition to the various names, engineers are more comfortable with arithmetic operators rather than logical operators. This book will follow the engineering convention that "+" is the OR operator, "*" is the AND operator, and "!" (pronounced *bang*) is the NOT operator.

Some students are uncomfortable with the ambiguity in the names for true and false. They often feel that the way the binary values were presented in their mathematics or philosophy classes (as

true/false) is the "correct" way to represent them. This would be fine if the goal were to create logic equations, equations that resolve to true or false, that are commonly covered in those classes. But this class is about implementing a computer in hardware. There is no correct, or even more correct, way to discuss binary values. How they will be referred to will depend on the way in which the value is being used. Understanding a computer requires the individual to be adaptable to all of these ways of referring to binary values. All of the ways used to represent binary values in Table 2-1 will all be used in this text, though most of the time the binary values of 0 and 1 will be used.

Chapter 2.1.2 Binary Whole Numbers

The numbering system that everyone learned in elementary school is called *decimal* or *base 10* numbers. This numbering system is called decimal because it has 10 digits, [0..9]. Thus quantities up to 9 can be easily referenced in this system by a single number.

Computers use circuits that produce binary output. This output can be either 0 or 1, and so computers use the binary, or base 2, numbering system. In binary, there are only two digits, 0 and 1. So values up to 1 can be easily represented by a single digit. Having only the ability to represent 0 or 1 items is too limiting to be useful. But then so are the 10 values which can be used in the decimal system. The question is how does the decimal handle the problem of numbers greater than 9, and can binary use the same idea?

In decimal when adding 1 plus 9 (1+9) the number 10 is created. The number 10 means that there is 1 group of ten values (the digits 0...9), and 0 one values. The number 11 is 1 group of ten and 1 one. When 99 is reached, we have 100, which is 1 group of hundred, 0 groups of tens, and 0 ones. So the number 1,245 would be 1 group of thousands, 2 groups of hundreds, 4 groups of tens, and 5 ones. This can be easily written using exponential format as:

$$1{,}245 = 1*10^3 + 2*10^2 + 4*10^1 + 5*10^0$$

Base 2 can be handled in the same manner. The number 10_2 (base 2) is 1 group of two and 0 ones, which corresponds to 2_{10} (2 base 10).[4] Counting in base 2 is the same. To count in base 2, the numbers are 0_2, 1_2, 10_2, 11_2, 100_2, 101_2,, 110_2, 111_2, etc. Analogously to the base 10 example above, any base 2 number can be represented in exponential format. Thus, the number 101011_2 can be represented by:

$$1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$$

Base 2 numbers represent value (e.g., quantities, distances) the same as base 10 numbers. There is no such thing as a base 2 number verses a base 10 number. There is a value that can be represented base 2, base 10, or any other base. The value is always the same, it is only the representation that has changed. A number in base 2 can be translated to base 10 using this principal. Consider 101011_2, which is:

$$1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 32 + 8 + 2 + 1 = 43_{10}$$

[4] The old joke is that there are 10 types of people in the world, those who know binary and those who do not.

In order to work with base 2 number, it is necessary to know the value of the powers of 2. The following table gives the powers of 2 for the first 16 numbers (to 2^{15}). It is highly recommended that students memorize at least the first 11 values of this table (to 2^{10}), as these will be used frequently.

n	**2^n**		**n**	**2^n**		**n**	**2^n**		**n**	**2^n**
0	1		4	16		8	256		12	4096
1	2		5	32		9	512		13	8192
2	4		6	64		10	1024		14	16348
3	8		7	126		11	2048		15	32768

Table 2-2: Values of 2^n for n = 0...15

The first 11 (0..10) powers of 2 are the most important because when talking about numbers in base 2 these are the only ones used. This is because the values of 2^n are named when n is a decimal number evenly dividable by 10. For example 2^{10} is 1 Kilo, 2^{20} is 1 Meg, etc. The names for these values of 2^n are given in the following table.

2^{10}	Kilo		2^{30}	Giga		2^{50}	Penta
2^{20}	Mega		2^{40}	Tera		2^{60}	Exa

Table 2-3: Names for values of 2n, n = 10, 20, 30, 40, 50, 60

Using these names and the values of 2^n from 0-9, it is possible to name the most commonly used binary numbers as illustrated below. To find the value of 2^{16}, we would write:

$$2^{16} = 2^{10} * 2^6 = 1K * 64 = 64K$$

Older programmers will recognize this as the limit to the segment size on older PC's which could only address 16 bits. Younger students will recognize the value of 2^{32}, which is:

$$2^{32} = 2^{30} * 2^2 = 1G * 4 = 4G$$

4G was the limit of memory available on more recent PC's with 32 bit addressing, though that limit has been moved with the advent of 64 bit computers.

Chapter 2.2 Translating Binary, Decimal, and Hex Numbers

Chapter 2.3.1 Translating Binary to Decimal

Computers operate in 0's and 1's, therefore when dealing with the internal workings of a computer it is the humans who must adjust to how computers work. However, when the computer produces answers, the humans that use them like to think in decimal. So, it is the job of a programmer to be able to translate between what the computer uses (binary) and what the human end users want to see (decimal). These next sections will deal with how to translate binary to decimal and give 2 ways to translate decimal to binary. The section after that it will explain a useful representation for handling large binary numbers called hexadecimal.

To translate binary to decimal, it is only necessary to remember that each 0 or 1 in a binary number represents the amount of that binary power of 2. For each binary power of 2, you have either 0 or 1 instance of that number. To see this, consider the binary number 1001010_2. This number has $(1 * 2^6) + (0 * 2^5) + (0 * 2^4) + (1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (0 * 2^0) = 64 + 8 + 2 = 74_{10}$. This can be generalized into an easy way to do this conversion. To translate from binary to decimal put the 2^n value of each bit over the bits in the binary number, and add the values which are 1, as in the example below:

```
                  64   32   16    8    4    2    1
| 1001010₂  =     1    0    0    1    0    1    0   = 64 + 8 + 2
```

Chapter 2.3.2 Translating Decimal to Binary using Binary Powers

Two ways to translate decimal number to binary numbers are presented here. The first is easy to explain, but harder to implement. The second is a cleaner algorithm, but why the algorithm works is less intuitive.

The first way to translate a number from decimal to binary is to see if a power of 2 is present in the number. For example, consider the number 433. We know that there is no 2^9 (512) value in 433, but there is one value of 2^8 (or 256). So, in the 9th digit of the base 2 number (note: 9th digit because the numbering is zero based) we would put a 1, and subtract that 256 from the value of 433.

433 - 256 = 177

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	-	-	-	-	-	-	-	-

Next check if there is a 2^7 (128) value in the number. There is, so add that bit to our string, and subtract 128 from the result.

177 - 128 = 49

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	-	-	-	-	-	-	-

Now check for values of 2^6 (64). Since 64 > 49, put a zero in the 2^6 position, and continue.

49 - 0 = 49

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	-	-	-	-	-	-

Continuing this process for 2^5 (32), 2^4(16), 2^3(8), 2^2(4), 2^1(2), and 2^0(1) results in the final answer.

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	0	0	0	1

Thus $433_{10} = 110110001_2$. This result can be checked by converting the base 2 number back to base 10.

Chapter 2.3.3 Translating Decimal to Binary using Division

While conceptually easy to understand, the method to translate decimal numbers to binary numbers in Chapter 1.2.2 is not easy to implement as an algorithm since the starting and stopping conditions are hard to define. There is a way to implement the translation from Base 10 to Base 2 which results in a nicer algorithm.

This second method to convert a decimal number to binary is to do successive divisions by the number 2. This is because if a number is divided and the remainder taken, the remainder is the value of the 2^0 bit. Likewise, if the result of the division in step 1 is divided again by 2 (so essentially dividing by 2*2 or 4), the reminder is the value of the 2^1 bit. This process is continued until the result of the division is 0. The example below shows how this works.

Start with the number 433. 433 divided by 2 is 216 with a remainder of 1. So, in step 1 the result would have the first bit for the power of 2 set to one, as below:

433 / 2 = 216 r 1

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
-	-	-	-	-	-	-	-	1

The number 216 is now divided by 2 to give 108 and the remainder, zero, placed in the second bit.

216 / 2 = 108 r 0

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
-	-	-	-	-	-	-	0	1

The process continues to divide by 2, filling the remainder in each appropriate bit, until at last the result is 0, as below.

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	0	0	0	1

Note that this algorithm gives the same result as the algorithm in Chapter 2.2.2. Why it works will be obvious later in the chapter when multiplication is covered.

Chapter 2.3.4 Converting between binary and hexadecimal

One of the biggest problems with binary is that the numbers rapidly become very hard to read. This is also true in decimal, where there is often a "," inserted between groupings of 10^3. So, for example 1632134 is often written as 1,632,134, which is easier to read.

In binary, something similar is done. Most students are familiar with the term byte, which is 8 bits. But fewer know of a nybble, or 4 bits. 4 bits in binary can represent numbers between 0..15, or 16 values. So, values of 4 bits are collected together and create a base 16 number, called a hexadecimal (or simply hex) number. To do this, 16 digits are needed, and arbitrarily the numbers and letters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F were chosen as the 16 digits. The binary numbers corresponding to these 16 digit hex numbers are given in the table below (note, the normal way to indicate a value is in hex is to write a *0x* before it. So decimal 10 would be *0xA)*.

Binary Number	Hex Digit		Binary Number	Hex Digit		Binary Number	Hex Digit		Binary Number	Hex Digit
0000	0x0		0001	0x1		0010	0x2		0011	0x3

0100	0x4		0101	0x5		0110	0x6		0111	0x7
1000	0x8		1001	0x9		1010	0xA		1011	0xB
1100	0xC		1101	0xD		1110	0xE		1111	0xF

Table 2-4: Binary to Hexadecimal Conversion

The hex numbers can then be arranged in groups of 4 (or 32 bits) to make it easier to translate from a 32 bit computer.

Note that hex numbers are normally only used to represent groupings of 4 binary digits. Regardless of what the underlying binary values represent, hex will be used just to show what the binary digits are. So, in this text all hex values will be unsigned whole numbers.

Most students recognize that a decimal number can be extended by adding a 0 to the left of a decimal number, which does not in any way change that number. For example $00433_{10} = 0433_{10} = 433_{10}$. The same rule applies to binary. So, the binary number $110110001_2 = 000110110001_2$.

But why would anyone want to add extra zeros to the left of a number? Because to print out the hex representation of a binary number, 4 binary digits are needed to do it. The binary number 110110001_2 only has 1 binary digit in the high order byte. So, to convert this number to binary it is necessary to pad it with left zeros, which have no effect on the number. Thus $1\ 10011\ 0001_2 = 0001\ 1011\ 0001_2 = 0x1B1$. Note that even the hex numbers are often padded with zeros, as the hex number 0x1B1 is normally be written 0x01B1, to get groupings of 4 hex numbers (or 32 bits).

It is often the case that specific bits of a 32 bit number need to be set. This is most easily done using a hex number. For instance, if a number is required where all of the bits except the right most (or 1) bit of a number is set, you can write the number in binary as:

$$11111111111111111111111111111110_2$$

A second option is to write the decimal value as: 4294967295_{10}

Finally, the hex value can be written as 0xFFFF FFFE

In almost all cases where specific bits are being set, a hex representation of the number is the easiest to understand and use.

Chapter 2.3 Character Representation

All of the numbers used so far in this text have been binary whole numbers. While everything in a computer is binary, and can be represented as a binary value, binary whole numbers do not

represent the universe of numbering systems that exists in computers. Two representations that will be covered in the next two sections are character data and integer data.

Though computers use binary to represent data, humans usually deal with information as symbolic alphabetic and numeric data. So, to allow computers to handle user readable alpha/numeric data, a system to encode characters as binary numbers was created. That system is called American Standard Code for Information Interchange (ASCII)[5]. In ASCII all characters are represented by a number from 0...127, stored in 8 bits. The ASCII encoding of these characters are shown in the following table.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Table 2-5: ASCII Table

Using this table, it is possible to encode a string such as "Once" in ASCII characters as the hexadecimal number 0x4F6E6365[6] (capital O = 0x4F, n = 0x6E, c = 0x53, e = 0x65).

[5] ASCII is limited to just 127 characters, and is thus too limited for many applications that deal with internationalization using multiple languages and alphabets. Representations, such as Unicode, have been developed to handle these character sets, but are complex and not needed to understand ARM Assembly. So, this text will limit all character representations to ASCII.

[6] By now it is hoped that the reader is convinced that hexadecimal is a generally preferred way to represent data in a computer. The decimal value for this string would be 1,332,634,469 and the binary would be 0100 1111 0110 1110 0110 0011 0010 0101. Having taught for many years, however, I know old habits die hard with many students who will struggle endlessly converting to decimal to avoid learning hex.

Numbers as character data are also represented in ASCII. Note the number 13 is 0xD or 1101_2. However, the value of the character string "13" is 0x3133. When dealing with data, it is important to remember what the data represents. Character numbers are represented using binary values, but are very different from their binary numbers.

Finally, some of the interesting patterns in ASCII should be noted. All numeric characters start with binary digits 0011 0000. Thus 0 is 0x0011 0000, 1 is 00011 0001, etc. To convert a numeric character digit to a number it is only necessary to subtract the character value of 0. For example, '0' - '0' = 0, '1' - '0' = 1, etc. This is the basis for an easy algorithm to convert numeric strings to numbers which will be discussed in the problems.

Also note that all ASCII upper case letters start with the binary string 0100 0001 and are 1 offset for each new character. So *A* is 0100 0001, *B* is 0100 0010, etc. Lower case letters start with the binary string 0110 and are offset by 1 for each new character, so *a* is 0110 0001, *b* is 0110 0010, etc. Therefore, all upper case letters differ from their lower case counterpart by a 1 in the digit 0010. This relationship between lower case and capital letters will be use to illustrate bit wise operations later in this chapter.

Chapter 2.4 Adding Binary Whole Numbers

The final topic before covering how integer values are stored and used in a computer for calculations is how to do addition of binary whole numbers.

When 2 one-bit binary numbers are added, the following results are possible: $0_2+0_2 = 0_2$; $0_2+1_2 = 1_2$; $1_2+0_2 = 1_2$; and $1_2+1_2 = 10_2$. This is just like decimal numbers. For example, 3+4=7, and the result is still one digit. A problem occurs, however, when adding two decimal numbers where the result is greater than the base of the number (for decimal, the base is 10). For example, 9+8. The result cannot be represented in one digit, so a carry digit is created. The result of 9+8 is 7 with a carry of 1. The carry of 1 is considered in the addition for the next digit. This means that when adding two numbers together, adding the digits in the number requires 3 numbers (the two addends and the carry). So, 39 + 28 = 67, where the 10's digit (6) is the result of the two addends (3 and 2) and the carry (1).

The result of $1_2+1_2 = 10_2$ in binary is analogous to the situation in base 10. The addition of 1_2+1_2 is 0_2 with a carry of 1_2 to the next digit.

An illustration of binary addition is shown in the figure below.

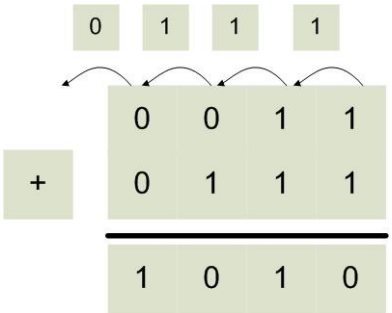


Figure 1: Binary whole number addition

Here the first bit adds $1_2 + 1_2$, which yields a 0_2 in this bit and a carry bit of 1_2. The next bit now has to add $1_2 + 1_2 + 1_2$ (the extra one is the carry bit), which yields a 1_2 for this bit and a carry bit of 1_2. If you follow the arithmetic through, you have 0011_2 (3_{10}) + 0111_2 (7_{10}) = 1010_2 (10_{10}).

Chapter 2.5 Integer Numbers (2's Complement)

Chapter 2.6.1 What is an Integer?

Using only positive whole numbers is too limiting for any valid calculation, and so the concept of a binary negative number is needed. When negative values for the set of whole numbers are included with the set of whole number (which are positive), the resulting set is called *integer* numbers. Integers are non-fractional numbers which have positive and negative values.

When learning mathematics, negative numbers are represented using a sign magnitude format, where a number has a sign (positive or negative) and a magnitude (or value). For example, -3 is 3 units (it's magnitude) away from zero in the negative direction (it's sign). Likewise, +5 is 5 units away from zero in a positive direction. Signed magnitude numbers are used in computers, but not for integer values. For now, just realize that it is excessively complex to do arithmetic using signed magnitude numbers. There is a much simpler way to do things called 2's complement. This text will use the term integer and 2's complement number interchangeably.

Chapter 2.6.2 2's complement operation and 2's complement format

Many students get confused and somehow believe that a 2's complement has something to do with negative numbers, so this section will try to be as explicit here as possible. Realize that if someone asks, "What is a 2's complement?", they are actually asking two very different questions. There is a 2's complement operation which can be used to negate a number (e.g., translate 2 -> -2 or -5 -> 5). There is also a 2's complement representation (or format) of numbers which can be used to represent integers, and those integers can be positive and negative whole numbers.

To reiterate, the 2's complement operation can convert negative numbers to the corresponding positive values, or positive numbers to the corresponding negative values. The 2's complement operation negates the existing number, making positive numbers negative and negative numbers positive.

A 2's complement representation (or format) simply represents number, either positive or negative. If you are ever asked if a 2's complement number is positive or negative, the only appropriate answer is yes, a 2's complement number can be positive or negative.

The following sections will explain how to do a 2's complement operation and how to use 2's complement numbers. Being careful to understand the difference between a 2's complement operation and 2's complement number will be a big help to the reader.

Chapter 2.6.3 The 2's Complement Operation

A 2's complement operation is simply a way to calculate the negation of a binary number. It is important to realize that creating a 2's complement operation (or negation) is not as simple as putting a minus sign in front of the number. A 2's complement operation requires two steps: 1 - Inverting all of the bits in the number; and 2 - Adding 1_2 to the number.

Consider the number 00101100_2. The first step is to reverse all of the bits in the number (which will be achieved with a bit-wise ! operation. Note that the ! operator is a unary operation, it only takes one argument.

$$! (00101100_2) = 11010011_2$$

Note that in the equation above the bits in the number have simply been reversed, with 0's becoming 1's, and 1's becoming 0's. This is also called a 1's complement, though in this text we will never use a 1's complement number.

The second step adds a 1_2 to the number.

$$\begin{array}{rr} & 11010011_2 \\ + & \underline{00000001_2} \\ & 11010100_2 \end{array}$$

Thus, the result of a 2's complement operation on 00101100_2 is 11010100_2, or negative 2's complement value. This process is reversible, as the reader can easily show that applying the 2's complement operation to the value of 11010100_2 is 00101100_2.

While positive numbers will begin with a 0 in the left most position and negative numbers will begin with a 1 in the leftmost position, these are not just sign bits in the same sense as the signed magnitude number, but part of the representation of the number. To understand this difference, consider the case where the positive and negative numbers used above are to be represented in 16 bits, not 8 bits. The first number, which is positive, will extend the sign of the number, which is 0. As we all know, adding 0's to the left of a positive number does not change the number. So, 00101100_2 would become 0000000000101100_2.

However, the negative value cannot extend 0 to the left. If for no other reason, this results in a 0 in the sign bit, and the negative number has been made positive. So, to extend the negative number 11010100_2 to 16 bits requires that the sign bit, in this case 1, be extended. Thus 11010100_2 becomes 1111111111010100_2.

The left most (or high) bit in the number is normally referred to as a sign bit, a convention this text will continue. But it is important to remember it is not a single bit that determines the sign of the number, but a part of the 2's complement representation.

Chapter 2.6.4 The 2's Complement (or Integer) Type

Because the 2's complement operation negates a number, many people believe that a 2's complement number is negative. A better way to think about a 2's complement number is that is a type. A type is an abstraction which has a range of values and a set of operations. For a 2's complement type, the range of values is all positive and negative whole numbers. For operations, it has the normal arithmetic operations such as addition (+), subtraction (-), multiplication (*), and division (/).

A type also needs an internal representation. In mathematics classes, the range of numbers were always abstract, theoretical entities, and the range is assumed to be infinite. But a computer is not an abstract entity, it is a physical implementation of a computing machine. Therefore, all numbers in a computer must have a physical size for their internal representation. For integers, this size is often 8 (byte), 16(short), 32(integer), or 64(long) bits, though larger numbers of bits can be used to store the numbers. Because the left most bit must be a 0 for positive and 1 for negative, using a fixed size also helps to identify easily if the number is positive or negative.

Because the range of the integer values is constrained to the 2^n values (where n is the size of the integer) that can be represented with 2's complement, about half of which are positive and half are negative, roughly 2^{n-1} values of magnitude are possible. However, one value, zero, must be accounted for, so there are 1 less positive numbers than negative numbers. So, while 2^8 is 256, the 2's complement value of an 8-bit number runs from -128... 127.

Finally, as stated in the previous section, just like zeros can be added to the left of a positive number without effecting its value, in 2's complement ones can be added to the left of a negative number without effecting its value. For example:

$$0010_2 = 0000\ 0010_2 = 2_{10}$$

$$1110_2 = 1111\ 1110_2 = -2_{10}$$

Adding leading zeros to a positive number and leading ones to a negative number, is called sign extension of a 2's complement number.

Chapter 2.6 Integer Arithmetic

Chapter 2.7.1 Integer Addition

Binary whole number addition was covered in chapter 1.4. Integer addition is similar to binary whole number addition except that both positive and negative numbers must be considered. For example, consider adding the two positive numbers 0010_2 (2_{10}) + 0011_2 (3_{10}) = 0101_2 (5_{10}).

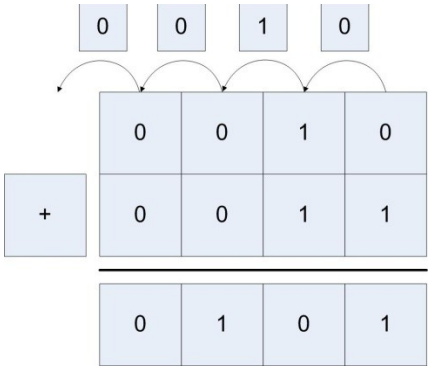


Figure 2: Addition of two positive integers

Addition of mixed positive and negative numbers, and two negative numbers also works in the same manner, as the following two examples show. The first adds 0010_2 (2_{10}) + 1101_2 (-3_{10}) = 1111_2 (-1_{10}), and the second adds 1110_2 (-2_{10}) + 1101_2 (-3_{10}) = 1011_2 (-5_{10}).

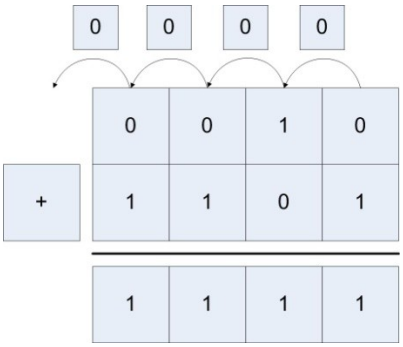


Figure 3: Addition of positive and negative integers

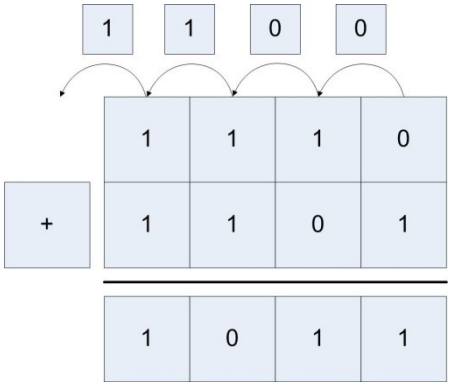


Figure 4: Addition of two negative integers

Because integers have fixed sizes, addition and subtraction can cause a problem known as integer overflow. This happens when the two numbers which are being added are large positive or negative values and the combining of the values results in numbers too big to be stored in the integer.

Chapter 2.7.2 Integer Addition with Overflow

Because integers have fixed sizes, addition and subtraction can cause a problem known as integer overflow. This happens when the two numbers which are being added are large positive or negative values and the combining of the values results in numbers too big to be stored in the integer value.

For example, a 4 bit integer can store values from -8...7. So, when 0100_2 (4_{10}) + 0101_2 (5) = 1001_2 (-7) are added using 4 bit integers the result is too large to store in the integer. When this happens, the number changes sign and gives the wrong answer, as the following figure shows.

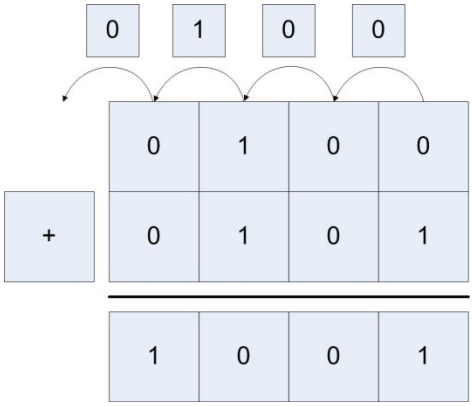


Figure 5: Addition with overflow

Attempting to algorithmically figure out if overflow occur is difficult. First if one number is positive and the other is negative, overflow never occurs. If both numbers are positive or negative, then if the sign of the sum is different than the sign of either of the inputs overflow has occurred.

There is a much easier way to figure out if overflow has occurred. If the carry in bit to the last digit is the same as the carry out bit, then no overflow has occurred. If they are different, then overflow has occurred. In figure 1.3 the carry in and carry out for the last bit are both 0, so there is no overflow. Likewise, in figure 1.4 the carry in and carry out are both 1, so there was no overflow. In figure 1.5 the carry in is 1 and the carry out is 0, so overflow has occurred.

This method also works for addition of negative numbers. Consider adding 1100_2 (-4_{10}) and 1011_2 (-5_{10}) = 0111_2 (7_{10}), shown in figure 1.6. Here the carry in is 0 and the carry out is 1, so once again overflow has occurred.

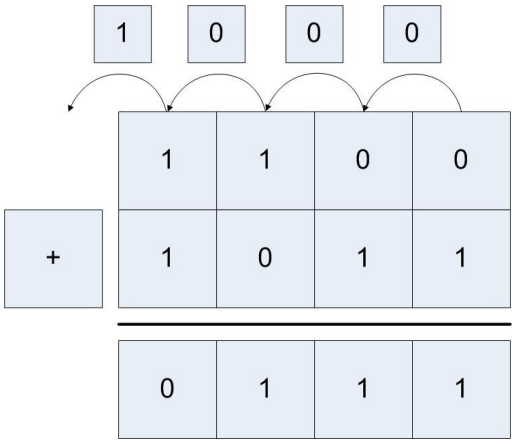


Figure 6: Subtraction with overflow

Chapter 2.7.3 Integer multiplication using bit shift operations

Multiplication and division of data values or variables involves hardware components in the Arithmetic Logic Unit (ALU). In assembly these operations will be provided by the various forms `mul` and `div` operators, and the hardware to implement them is beyond the scope of this book and will not be covered. However, what is of interest in writing assembly is multiplication and division by a constant.

The reason multiplication and division by a constant is covered is that these operations can be provided by bit shift operations, and the bit shift operations are often faster to run than the equivalent `mul` or `div` operations. Therefore, bit shift operations are often used in assembly to do multiplication and division, and therefore it is important for assembly language programmers to understand how this works.

First, consider multiplication of a number by a power of 10 in base 10. In base 10, if a number is multiplied by a power of 10 (10^n, where n is the power of 10), it is sufficient to move the number n places to the right filling in with 0's. For example, 15*1000 (or 15 * 10^3) = 15,000.

This same concept holds in binary. To multiply a binary number (e.g., 15, or 00001111_2) by 2, the number is shifted to the left 1 digit (written as 1111<<1), yielding 00011110_2 or 30. Likewise multiplying 00001111_2 by 8 is done by moving the number 3 spaces to the left (00001111_2<<3), yielding 01111000_2, or 120. So, it is easy to multiply any number represented in base 2 by a power of 2 (for example 2^n) by doing n left bit shifts and back filling with 0's.

Note that this also works for multiplication of negative 2's complement (or integer) numbers. Multiplying 11110001_2 (-15) by 2 is done by moving the bits left 1 space and again appending a 0, yielding 11100010_2 (or -30) (note that in this case 0 is used for positive or negative numbers). Again multiply 11110001_2 (-15) by 8 is done using 3 bit shifts and back filling the number again with zeros, yielding 10001000_2 (-120).

By applying simple arithmetic, it is easy to see how to do multiplication by a constant 10. Multiplication by 10 can be thought of as multiplication by (8+2), so (n*10) = ((n*8)+(n*2)).

$$15*10 = 15 * (8+2) = 15 *8 + 15 * 2 = (00001111_2 << 3) + (00001111_2 << 1) =$$

$$1111000_2 + 11110_2 = 100100110_2 = 150$$

This factoring procedure applies for multiplication by any constant, as any constant can be represented by adding powers of 2. Thus, any constant multiplication can be encoded in assembly as a series of shifts and adds. This is sometimes faster and often easier, than doing the math operations and should be something every assembly language programmer should be familiar with.

This explanation of the constant multiplication trick works in assembly, which begs the question does it also work in a HLL? The answer is yes, but it should never be used. Bit shifts and addition can be done in most programming languages, so constant multiplication can be implemented as bits shifts and addition. But just because it can be done does not mean it should be done. In HLL (C/C++, Java, C#, etc.) this type of code is arcane and difficult to read and understand. In addition, any decent compiler will convert constant multiplication into the correct underlying bit shifts and additions when it is more efficient to do so. And the compiler will make better decisions about when to use this method of multiplication and implement it more effectively and with fewer errors than if a programmer were to do it. So, unless there is some really good reason to do multiplication using bit shifts and addition, it should be avoided in a HLL.

Chapter 2.7.4 Integer division using bit shift operations

Since multiplication can be implemented using bit shift operations, the obvious question is whether or not the same principal applies to division? The answer is that for some useful cases, division using bit shift operations does work. But in general, it is full of problems.

The cases where division using bit shift operations works are when the dividend is positive and the divisor is a power of 2. For example, 00011001_2 (25) divided by 2 would be a 1-bit shift, or 00001100_2 (12). This is achieved as the answer 12.5 is truncated by throwing away the bit which has been shifted out. Likewise, 00 011001_2 (25) divided by 8 is 00000011_2 (3), with truncation again occurring. Note also that in this case the bit that is shifted in is the sign bit, which is necessary to maintain the correct sign of the number.

Bit shifting for division is useful in some algorithms such as a binary search finding parents in a complete binary tree. But again, the compiler will implement division by a constant if bit shifting works, so using bit shifts to do arithmetic should be avoided unless there is a good reason to use it in a HLL.

This leaves two issues. The first is why can this method not be implemented with constants other than the powers of 2. The reason is that division is only distributive in one direction over addition, and in our case, it is the wrong direction. Consider the equation 60/10. It is easy to show that division over addition does not work in this case.

$$60/10 = 60/(8+2) \neq 60/8 + 60/2$$

The second issue is why the dividend must be positive. To see why this is true, consider the following division, -15 / 2. This result in the following:

$$11111001_2 >> 1 = 11111100 = -8$$

Two things about this answer. First in this case the sign bit, 1, must be shifted in to maintain the sign of the integer.

Second, in this case the lowest bit, a 1, is truncated. This means that -7.5 is truncated down to -8. However, many programmers believe that -7.5 should truncate to -7. Whether the correct answer is -7 or -8 is debatable, and different programming languages have implemented as either value (for example, Java implements -15/2 = -7, but Python -15/2 as -8). This same problem occurs with many operations on negative numbers, such a modulus. And while such debates might be fun, and programmers should realize that these issues can occur, it is not the purpose of this book to solve this problem, but to make the reader aware it exists.

Chapter 2.7 Boolean Logical and Bitwise Operators

Chapter 2.7.1 Boolean Operators

Boolean operators are operators which are designed to operate on Boolean or binary data. They take in one or more input values of 0/1[7] and combine those bits to create an output value which is either 0/1. This text will only deal with the most common Boolean operators, the unary operator NOT (or inverse) and the binary operators[8] AND, OR, NAND, NOR, and XOR. These operators are usually characterized by their truth tables, and two truth tables are given below for these operators.

A	NOT
0	1
1	0

Table 1-6: Truth table for NOT operator

Input		Output				
A	**B**	**AND**	**OR**	**NAND**	**NOR**	**XOR**
0	0	0	0	1	1	0
0	1	0	1	1	0	1

[7] Note that the values 0/1 are used here rather than F/T. These operators will be described through the rest of the book using the binary values 0/1, so there is no reason to be inconsistent here.

[8] The term unary operator means having one input. The term binary operator means having two inputs. Be careful reading this sentence, as binary is used in two different contexts. The binary operator AND operates on binary data.

1	0	0	1	1	0	1
1	1	1	1	0	0	0

Table 1-7: Truth table for AND, OR, NAND, NOR, and XOR

Chapter 2.7.2 Logical Boolean Operators

There are two kinds of Boolean operators implemented in many programming languages. They are logical operators and bitwise operators. Logical operators perform Boolean operations to obtain a single value at the end. For example, in Java a programmer might write:

```
if (x != 0)  {
    //do something
}
```

The purpose of an if statement is generally to decide whether or not to enter the statement or code block associated with the if test. The purpose of this statement is just to derive a single logical answer, either true or false. This is the classical example of a binary value being a logical value.

Because logical expressions only care about whether the result is true or false, it is possible to implement expressions that *short-circuit*. In a short-circuit expression the expression is evaluated until a valid answer is determined. Once the answer is determined, any other parts of the expression are skipped, or short-circuited. As we will see later, this is only possible for Boolean operations that are used for logical expressions and not true for general Boolean operations.

To see how this short-circuit behavior can be used, consider the following expression:

```
if ((x != 0) && (y = z / x));
```

Note that a "`;`" is placed after this if test, which means that there is nothing to do if the expression is true. So, what is the purpose of this expression? Note that if x == 0 the first part of the expression is false, so the second part of the expression "`(y = z / x)`" is not executed. This code fragment with the if statement is protecting the division operation from a zero divide.

Note that logical operations are looking to resolve to a single bit and so short circuiting is possible and enforced in many (if not most) HLL's.

The important take away from this is that logical operators are Boolean operations that act only on one binary value and are short circuiting operators.

Chapter 2.7.3 Bitwise Boolean Operators

On the other hand, bit-wise operators are not short circuiting. Consider the following problem. A programmer wants to write a toLower method which will convert an upper case letter to a lower case letter. In chapter 1.3 it was pointed out that the difference between an upper case letter and a lower case letter is that in a lower case letter the bit 0x20 (00100000_2) is 1, whereas

in the upper case letter it is zero. So, to convert from an upper case letter to a lower case letter, it is only necessary to OR the upper case letter with 0x20, turning on this bit. In pseudo code this could be implemented as follows:

```
char toLower(char c) {

    return (c | 0x20)

}
```

In this case the bitwise OR operator, |, needs to operate on every bit in the variables. Therefore, the | operator is not short circuiting, it will process every bit regardless of whether or not a previous bit would cause the operation to fail. It is not seeking a logical result, but to apply the Boolean (not logical) operator to all of the bits.

It is possible to use bitwise operators in place of logical operators, but it is usually incorrect to do so. For example, in the previous if statement, if a bitwise operator had been used, no short circuiting would have occurred and the zero divide could occur.

```
if ((x != 0) & (y / x > 4))
```

Many languages such as C/C++, Java, C#, etc, have both logical (short circuiting) and bitwise operators. In most cases the single character operator is a bit wise operator (e.g., &, |) and the double character operator is the logical operator (e.g., &&, ||).

Now to make things more confusing, in MIPS only bitwise operations are implemented, and they are called logical operators. Yes, this is confusing to new assembly language programmers, and there is no good way to reconcile this, so the user is cautioned to read the material and programs carefully.

Chapter 2.8 Program Context

The final bit of information to take from this chapter is that data in a computer is a series of "1" or "0" bits. In computer memory two bytes containing “01000001” could exist. The question is what does this byte mean? Is the byte an integer number corresponding to decimal 65? Is this an ASCII character, representing the letter "A"? Is it a floating point number, or maybe an address? The answer is you have no idea!

To understand data there has to be a context. HLL always provide the context with the data (for example, the type, as in int a;), so the programmer does not have to worry about it. However, in assembly the only context is the one the programmer maintains, and it is external to the program. Is it possible to convert an integer number from upper case to lower case? Or to add two operations? The answer is yes, anything is possible in assembly, but that does not mean it makes sense to do it.

In assembly language it is important for the programmer to always be aware of what a series of bits in memory represent, or the context of the data. Remember that data without a context is meaningless.

Chapter 2.9 Summary

In this chapter the concept of binary was introduced, as well as ways to represent binary data such as binary whole numbers, integers, and ASCII. Arithmetic and logical operations were defined for binary data. Finally, the chapter introduced the concept of a context, where the context defines the meaning of any binary data.

Chapter 2.10 Exercises

1 What are the following numbers in binary and hexadecimal?

1.a 13_{10}

1.b 15_{10}

1.c 25_{10}

1.d 157_{10}

1.e 325_{10}

1.f 1096_{10}

2 What are the following numbers in decimal?

2.a 10011100_2

2.b $9C_{16}$

2.c $1F_{16}$

2.d $0C_{16}$

2.e $109A_{16}$

3 Give the value of the following numbers in their hexadecimal representation (e.g., 2^{32} = 0x4G)

3.a 2^{16}
3.b 2^{24}
3.c 2^{29}
3.d 2^{34}
3.e 2^{31}

4 Give the 2's complement form for each of the following numbers:

4.a 13

4.b -13
4.c 156
4.d -209

5 Perform the following calculations using 2's complement arithmetic. Show whether there is an overflow condition or not. CHECK YOUR ANSWERS!
5.a 13 + 8 with 5 bits precision
5.b 13 + 8 with 6 bits precision
5.c 13 – 8 with 5 bits precision
5.d 13 – 8 with 6 bits precision
5.e -13 – 8 with 5 bits precision
5.f -13 – 8 with 6 bits precision
5.g 105 – 57 with 8 bits precision

6 Perform the following multiplication operations using binary shift operations. Check your answers.
6.a 5 * 4
6.b 13 * 12
6.c 7 * 10
6.d 15 * 5

7 What is the hex representation of the following numbers (note that they are strings)?
7.a "52"
7.b "-127"

8 Perform the following multiplication and division operations using only bit shifts.
8.a 12 * 4
8.b 8 * 16
8.c 12 * 10
8.d 7 * 15
8.e 12 / 8
8.f 64 / 16

9 Explain the difference between a short circuiting and non short circuiting logical expression. Why do you think these both exist?

10 In your own words, explain why the context of data found in a computer is important. What provides the context for data?

11 Convert the following ASCII characters from upper case to lower case. Do not refer to the ASCII table.
11.a 0x41 (character A)
11.b 0x47 (character G)
11.c 0x57 (character W)

12 Convert the following ASCII characters from lower case to upper case. Do not refer to the ASCII table.

12.a 0x62 (character b)
12.b 0x69 (character i)
12.c 0x6e (character n)

13 Write the functions toUpper and toLower in the HLL of your choice (C/C++, Java, C#, etc.) The toUpper function converts the input parameter string so that all characters are uppercase. The toLower function converts the input parameter string so that all characters are lowercase. Note that the input parameter string to both functions can contain both upper and lower case letters, so you should not attempt to do this using addition.

14 Implement a program in the HLL of your choice that converts an ASCII string of characters to an integer value. You must use the follow pseudocode algorithm:

```
read numberString
  outputNumber = 0
  while (moreCharacters)
     c = getNextCharacter
     num = c - '0';
     outputNumber = outputNumber * 10 + num;
  print outputNumber;
```

15 Write a program in the HLL of your choice to take a string representing a binary number, and write out the number in hex.

16 Write a program in the HLL of your choice to take an integer, and write its value out in hex. You cannot use string formatting characters.

17 Is bit shift by 1 bit the same as division by 2 for negative integer numbers? Why or why not?

18 Can multiplication of two variables (not constants) be implemented using the bit shift operations covered in this chapter? Would you consider using the bit shift operations implementation of multiplication and division for two variables, or would you always use the *mul* or *div* operators in ARM assembly? Defend your choice.

19 Should you use bit shift operations to implement multiplication or division by a constant in a HLL? What about assembly makes it more appropriate to use these operations?

20 What does the Unix strings command do? How does the command attempt to provide a context to data? How might you use it?

What you will learn

In this chapter you will learn:

1 an implementation of a template that that can be used to create the `main` function for an assembly language program using `gcc`

2 how to write comments in ARM assembly for `gcc`

3 the use of the `ldr` assembly instruction to load a memory address and a memory value into a register

4 how to implement an assembly language program to print a "`Hello World`" message

5 how to use the C `printf` and `scanf` functions in a program to input and output data

6 value and reference variables

7 the steps to build a program from shell, illustrating the compiling, linking and execution steps to create a program

8 running a program from shell

9 creating a `makefile` script to compile, link, and execute a program using the `make` command

10 the use `scanf` and `printf` to input and output an integer value, illustrating the difference between a reference and a value

11 learn what the gdbtui is, and how to use it to view a program execution.

Chapter 3 Getting Started with Assembly Language Programming

The first step in learning any new language is being able to create a working template program in that language, then creating a program to read input and produce output. Being able to produce I/O necessary to be able to tell if a program is working, so a properly working I/O program forms the basis for being able to implement for larger and more complex programs. This I/O program is often called a "Hello World" program, and the purpose of this chapter is to create this first program. This will involve the following steps:

1 creating a template file that can be used as a starting point for any program the user will create

2 writing assembly language source files using `printf` and `scanf` for a program to read input and print output for a program

3 using an assembler and linker to translate their source programs into ARM executable programs

4 running the programs from a shell command

5 finally, using the gbdtui to view the program execution and state

Chapter 3.1.1 Template for an assembly language program

When learning a new language there are some programming details that are necessary to allow the program to run, but that cannot be explained to someone first learning the language. These language concepts can only be explained later after the programmer has learned much more about the language. To allow a novice programmer to start programming in the language, these programming details are often given as a format for a program that must simply be copied to create a program. For example, in the Java programming language all methods must exist in a class, and the program must begin in a static function named main that takes an array of strings as an argument. Why this format must be followed is beyond the ability of beginning Java programmers to efficiently comprehend and irrelevant to the material that is covered at that point in the learning process. Therefore, readers are told that they must copy these conventions until they can be explained later.

When programming in assembly language, the same concept applies. There are standard coding requirements for the program that must be included to make the program run. The first step in this chapter is to write an assembly source program with the structure necessary to make an assembly program work. This first program defining a standard format to allow programmers to create a basic program is to be called `template.s`, and it can be copied as a starting point, or template, for all subsequent programs. All of code in the `template.s` program will be explained by the chapter on functions, but for now it should just be copied.

This template source program is shown below. Note that all files containing assembly language source programs should end with the suffix "`.s`". In this program, the comment "`# enter your program here`" is the place you should put the code for your program.

Note, the program lines beginning with a hashtag (`#`) are comment line. There are two different types of comments in assembly. The `#` is used if the entire line is a comment. To comment from the current position on the line to the end of the line, for example to add a comment to a specific instruction, this text will use the characters "`//`". Note that there are other ways to comment to the end of the line in ARM assembly language, so follow your assemblers preference. For `gcc`, it is "`//`", so that is what will be used here.

Second, all programs should contain a header with information about the program. This is standard practice in most introduction to programming classes in every language, and it seems to

be the first good habit that programmers are eager to discard as soon as possible. If you are using this book in a course, it is the author's hope this is the first thing your instructor takes points off for not doing.

Finally note the indenting in the program. ARM assembly allows, but does not require, indentation. Once again, just because the language does not require it is no reason to stop the practice of indenting.

```
#
# Program Name: template.s
# Author: Charles Kann
# Date: 9/19/2020
# Purpose: This program is template that can be used to start ARM assembly
# program using gcc
#

.text
.global main

main:
   # Save return to OS on stack
   SUB sp, sp, #4
   STR lr, [sp, #0]

   # Enter your program here.

   # Return to the OS
   LDR lr, [sp, #0]
   ADD sp, sp, #4
   MOV pc, lr

.data
```

1 template.s

Chapter 3.1.2 Hello World program

The first program that a programmer often writes in a new language is called a "Hello World" program. The purpose of this program is just to:

1 create a program with valid syntax.

2 be able to process output from the program.

3 execute the steps to create a valid program executable.

4 ensure that the executable program file can be run.

To begin, edit the file `helloWorldMain.s`, and enter the following text. You can start by copying the file `template.s` to `helloWorldMain.s` and adding the highlighted code to print the string the "`# Printing the Message`" block below. Be sure to add the `helloWorld` variable in the `.data` section of the program.

```
#
# Program name: helloWorldMain.s
# Author: Charles Kann
# Date: 9/19/2020
# Purpose: This program shows how to print a string using the C function printf
#

.text
.global main

main:
   # Save return to os on stack
   SUB sp, sp, #4
   STR lr, [sp, #0]

   # Printing The Message
   LDR r0, =helloWorld
   BL  printf

   # Return to the OS
   LDR lr, [sp, #0]
   ADD sp, sp, #4
   MOV pc, lr

.data
   # Stores the string to be printed
   helloWorld: .asciz "Hello World\n"
```

2 helloWorldMain.s

Save the file, return to the shell prompt, and then type the following command:

```
gcc HelloWorldMain.s -g -c -o HelloWorldMain.o
```

This line calls the `gcc` command. It will see that the input file to the command has a suffix "`.s`". This "`.s`" suffix indicates to the `gcc` command that the assembler is to be used to process this file.

This `gcc` command has many options, three of which are used here. The `-g` option informs the assembler that debugging information should be produced. Programs in this book will generally use the `-g` option as most programs will have the need to be viewed and debugged. However, the `-g` option causes the compiler to produce large executable files that tend to execute very slowly, so if a program is to be used in a production environment, the -g option is normally omitted.

The `-c` option says that the assembler is to only assemble the code to an object file and not attempt to make an executable file from it. It is used to keep an intermediate file created between the assembly source and the final executable file. This intermediate file is called an *object* file. This object file is often not needed if the goal is only to create an executable file from the assembly file, and in later sections the keeping of the object file will often be omitted.

The final `-o` option informs the compiler to save its output to an output file with the name specified after the option. For this command the file "`helloWorld.o`" is created.

The result of this command is an object file called "`helloWorld.o`". An object file contains the result of translating the assembly code in the file `helloWorld.s` into machine code. Machine code is a translation of assembly instructions into a format that only uses binary values that can be understood by the CPU. However, an object file is not an executable file and cannot be used by the CPU. An object file is an intermediate file between a source code assembly file and an executable file. The machine code in an object file stills needs to be combined with other object files to resolve items that are not defined in the source file. In our case, the `printf` function is not defined in the source code for this program. The `printf` function is called an *external reference* that must be *resolved* before the program can be executed.

The resolution of these external references is achieved by looking in other object files or library files for the definition of the unresolved references. These object and library files must be *linked* to our object file, and when the external reference is found (or *resolved*), the function will be combined with the object from the assembler to create an executable program. The program that creates the executable file is called a *linking loader* (this program can also be called a *linkage editor*, or a *linker,* or a *loader*. We will call the program a linker from here on.) After the linker has found and resolved any references, it writes an executable file that can be run. The linker is run by typing the following command:

```
gcc helloWorldMain.o -g -o helloWorld
```

Because the file name has a ".o" suffix, the `gcc` command knows to run the linker. The final file, `helloWorld`, is a file that can be executed. This executable file can be executed by typing the following command:

```
./helloWorld
```

The running of the program should produce the string "`Hello World`", which tells you that the program is up and running correctly.

Chapter 3.1.3 Notes on the HelloWorld Program

The following are notes on the `helloWorld` program to explain the syntax and semantics of the program.

1 The program is saved in a file `helloWorldMain.s`. Note that this does not match the match the name of the function, which is `main`. It does not match what I have called the program name, which is helloWorld. Languages such as Java require the file name match the name of a public class, and there are restrictions or suggestions as to how to do structure programs and name programs and variables. No such restrictions are generally applied to assembly. An untrained programmer can create a nightmare of names and find strange places to store files, that are often impossible to untangle, even by the programmer themselves. So, be careful to follow any standards in place for naming and file management (such as directory structures) that exist when you create your code. For this book, the programs are stored in directories by the chapter they are found in. This book also has a supplemental style guide which is suggested be used in writing your programs unless your employer or professor chooses different standards.

2 Except for the highlighted lines, all of the code in the `HelloWorld` program was copied from the `Template.s` file. For now, this code is just copied when starting to write all programs and not explained further.

3 Any line that starts with a `#` (hashtag) is a comment line. If the `#` is not the first character on a line, it is not a comment but signifies that the following is a *numeric token*[9]. Note that numeric tokens can be:

3.1 decimal value token specified by `#nnn`, where n is any decimal digit.

3.2 hex value token specified by `#0xnn`, where n is any hexadecimal digit.

3.3 binary value token, specified by `#0bnnnnnnnn`, where n is any binary digit[10].

[9] A numeric token is one or more numbers followed by a blank.
[10] Note that the method of specifying the binary digits is assembler dependent, so check your assembler documentation if #0b does not work.

4 To add a comment after an instruction, use the string "//" [11]. For example, the following is an instruction line containing a comment.

```
bl printf // branch to the printf function
```

5 The `ldr` (Load Register) operation loads a register with a value from memory. In this case the "=" sign means load the *address* of the string "`HelloWorld`" into `r0`.

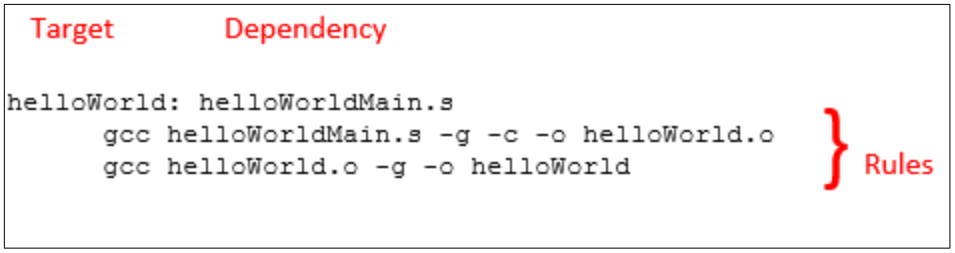
6 The `printf` command looks for the address of the string format to output in `r0`. The register `r0` must contain the address of the format string when calling the function `printf`.

7 The `\n` in the format string is an escape sequence meaning print a new line. A complete list of all escape sequences can be found at **https://en.cppreference.com/w/c/language/escape**.

8 The `bl` (Branch and Link), used in the "`bl printf`" instruction saves the return pointer (where the function is to return), and then branches (or jumps) to the function that is named (`printf`). When the `printf` function completes, it returns to the statement immediately after the function call, as it does in any other language.

Chapter 3.1.4 Using make to Create the Program

The final part of this section is to explain how to create the program more easily. This will be done using a *`makefile`* and the *`make`* command. A `makefile` is a way of automating the steps that need to be done to create some artifact, such as a program. Any task that requires multiple steps to be executed can be accomplished using the `make` command, but its normal use is to create program executable files.



The `makefile` presented here consists of 3 parts:

1 A target, which is the file that is to be created as a result of running this make command.

[11] Some assemblers will use other characters, such as a semicolon, for this, so be careful. gcc can use an @ or //.

2 Dependencies, which are the files or resources that are to be checked for changes that would require the target to be remade. For example, if the source code file is changed, the time stamp on the source code file is more recent than the time stamp on the target. This means that changes have been made to the source file, and these should be reflected in the target. The make command will thus be rerun to incorporate recent changes.

3 Rules are the *recipe* (or the commands to be run and the order to execute them) to recreate the target file. Note that lines containing rules **must be indented with tabs**. The space in front of the gcc commands cannot be blanks, but must be a tab.

In this example the target, or program to be created, is named `helloWorld` and is located in the current directory. It has one dependency, the file `helloWorldMain.s`. If this dependency file is changed, the program `HelloWorld` is remade by using the gcc commands.

To run this `makefile`, type the command *make* at the shell command prompt in the directory where these files reside. The make command will check the modification times of the files `helloWorld` and `helloWorldMain.s`, and run the commands if needed. If everything is currently up-to-date, it will print out "`nothing to make`".

Chapter 3.1 Prompting for an Input String

The next program will show how to read an input string into the program and then print it out as part of a formatted string. The C function `scanf` will be used to read the string. To start, edit a file named `printNameMain.s` and enter the text in the following program.

```
#
# Program Name: printNameMain
# Author: Charles Kann
# Date: 9/19/2020
# Purpose: To read a string using scanf
# Input:
#   - input: Username
# Output:
#   - format: Prints the greeting string

.text
.global main

main:
   # Save return to os on stack
   SUB sp, sp, #4
```

```
    STR lr, [sp, #0]

    # Prompt for an input
    LDR r0, =prompt
    BL  printf

    # Scanf
    LDR r0, =input
    LDR r1, =name
    BL scanf

    # Printing the message
    ldr r0, =format
    ldr r1, =name
    BL  printf

    # Return to the OS
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr

.data
    # Prompt the user to enter their name
    prompt: .asciz "Enter your name: "
    # Format for input (read a string)
    input: .asciz "%s"
    # Format of the program output
    format: .asciz "Hello %s, how are you today? \n"
    # Reserves space in the memory for name
    name: .space 40
```

3 printNameMain.s

To create the program, there is no need to type the commands at the shell prompt since a `makefile` already exists in this directory. Edit the `makefile` and change it to the following:

```
all: helloWorld printName
```

```
helloWorld: helloWorldMain.s
	gcc $@Main.s -g -o $@
	./helloWorld
printName: printNameMain.s
	gcc $@Main.s -g -o $@
	./printName
```

4 makefile for printName

This Makefile has several changes. In this file the first target is "`all`". If the make program is run without specifying a target on the command line, the make will use the first target in the `makefile`, which is often named `all`. This `all` target consists of two other dependencies, `helloWorld` and `printName`, and so these files become targets. Thus, running the `makefile` will run the check to see if either the `helloWorld` or `printName` programs need to be remade. If both are current, it will return the message "`nothing to make`". If either or both of the programs are not current, it will remake the programs so that all the executable files are current.

Note that any of the targets can be explicitly called. For example, *make all* will use the target all, *make helloWorld* will only check and remake only the `helloWorld` program, and *make printName* will check and remake only the `printName` program.

Second, the rules in this makefile specify the program executable is made directly from the source code file without creating an object file. The `gcc` command is still making the object file, but it does not keep it after the gcc command is run. The object file is not something needed to run the program, and so it is not necessary for the program to keep it. Thus, this `gcc` command allows the executable file to be created without having to keep the object file.

Finally, there is a rule for each of the programs that runs the programs. These changes to the `makefile` will remake and test the programs.

Chapter 3.2 Comments on the printName program

The `printName` program changes two things.

1 The first is it uses the `scanf` function to read the values from the user. The `scanf` program uses two registers, `r0` and `r1`, to do this. The first register contains the pointer to (or address of) the string that specifies the value to be read. The value to be read is a string, so the formatting specifier `%s`[12] is used. In this case, the address is loaded into `r0` using the `ldr r0,=input1`[13] instruction. Note that the address of the string, or where the

[12]The formatting characters are the same as those used for the printf statement, and a complete list can be found at http://www.cplusplus.com/reference/cstdio/printf/.

[13]Note that in this textbook a number will often appended to the names of labels that are common words like name, input, format, etc. This is done because when using a label that is common because in the gdb debugger, there

string is in memory, is loaded, not the string itself. The string value is generally too large to load into register, so only the reference, which is a 32-bit address, is loaded.

The second register also contains the address of where to store the result of the `scanf` function. Space for the input string is allocated in the `.data` section of the program using the instruction `name: space 40`, allocating 40 bytes of space to store the user entered name. When the `bl scanf` instruction is executed, the user can enter a name that will be stored in memory at the allocated space associated with `name`.

2 The second change to this program is that the format string for the `printf` function now includes the format specifier `%s`. This format specifier indicates that the `printf` command should look for the address of a string in `r1` and will insert that string into the format string at the position of the `%s` command. It is interesting that the format string passed to the `printf` command is used to determine what other arguments will be used by the `printf` function. In C, these are called *variadic* parameters. Some exercises at the end of this chapter will explore the use of simple variadic parameters.

Before continuing to the next program, the user is advised to understand the difference between a variable (the memory allocated for data value), the reference to a variable (the address where that variable can be found), and the value of the variable (the data that is assigned to that variable, or the value that variable currently contains). When covering variables in introductory programming, variables are often presented as a box that contains a variable. The box has a name and contains a value. Such a metaphor is at best incomplete, and while useful in an introductory class, it becomes untenable when trying to work with references and values.

This distinction of variables and references will become even more apparent in the next example, where the inconsistency of the C `scanf` and `printf` functions when handling data, particularly data other than strings, is more apparent.

Chapter 3.3 Prompting for an Input Integer Number

The next program, `printInt`, shows how to use `scanf` to read an input integer number from a user and then how to output that number back to the user using `printf`. It is contained in the following example program.

```
#
# Program Name:   printIntMain
# Author: Charles Kann
# Date:     9/19/2020
```

appear to be other instances that already use those name, and it makes debugging an issue. The number disambiguate the names.

```
# Purpose: Uses scanf for an integer using data memory
# Input:
#   - input: User entered number
# Output:
#   - format: Prints the number
#

.text
.global main

main:
    # Save return to os on stack
    SUB sp, sp, #4
    STR lr, [sp, #0]

    # Prompt for an input
    LDR r0, =prompt
    BL  printf

    # Scanf
    LDR r0, =input
    LDR r1, =num
    BL scanf

    # Printing the message
    LDR r0, =format
    LDR r1, =num
    LDR r1, [r1, #0]
    BL  printf

    # Return to the OS
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr

.data
    # Allocates space for a word-aligned 4-byte value in the memory
    num: .word 0
    # Prompt the user to enter a number
    prompt: .asciz "Enter A Number\n"
    # Format of the user input, %d means integer number
    input: .asciz "%d"
    # Format to print the entered number
    format: .asciz "Your Number Is %d \n"
```

5 printIntMain

```
all: helloWorld printName printInt

helloWorld: helloWorldMain.s
	gcc $@Main.s -g -o $@
	./helloWorld
printName: printNameMain.s
	gcc $@Main.s -g -o $@
	./printName
printInt:
	gcc $@Main.s -g -o $@
	./printInt
```

6 makefile for printInt

Chapter 3.4 Comments on the PrintInt program

Superficially the programs for `printName` and `printInt` seem to be the same, just that a different type of value is being read and printed. However, these programs differ in a small but significant way. Just as in the `printName` program, the reference to the `num` variable is passed to the `scanf` function, and the memory is updated when the value is read. However, when calling the `printf` function the value of num (not the reference) is stored in `r1`.

```
ldr r1, =num

ldr r1, [r1, #0]
```

In these two lines, first the address of `num` is loaded into `r1`, and next, the value that is at the address `r1+0` is loaded into `r1`. The `printf` function is then called with the value of `num`, not the reference.

Be careful when reading this code fragment. Understanding the difference between a reference and a value is central to understanding much of assembly, and even affects how you can understand many concepts you will encounter in future classes that involve a HLL. References and values are central to much of computer science.

Chapter 3.5 Debugging with gdb

Often a simulator is used to teach assembly. This is often done because the simulators are pared down versions of the real assembly, only presenting the students with what is believed to be the necessary information to understand the concepts the instructor wants to present. The tools that come with these simulators are designed to be easy to use by the students and to present simplified visuals for the students.

This textbook takes a different approach to this problem of how to present material to the students. It uses tools that the reader might encounter in a real world programming environment to create, run, and visualize the code. It requires students to use a real operating system, the Raspberry Pi OS, which is a version of Linux. The reader has to learn to use the bash shell, how the write `make` files to create programs, and that there are intermediate files between the source files and the executable files called object and library files. Later, readers will learn how to decompose and read the object and executable files using commands such as `objdump` and `readelf`.

This section will teach the students the GNU debugger, `gdb`. The `gdb` debugger will be presented with a console oriented interface named `tui` (or `gdbtui`, `gdb`'s Text User Interface). This debugger will be used by the readers to debug program, to help understand the execution of their programs, as well as visualize what is happening with memory and registers during the execution of their program.

Chapter 3.6 Running the gdb commands

To run the `gdb` program, first an executable such file must be created, such as the `printInt` program. This program should be assembler and linked using the `-g` flag, which tells the assembler and linker to include debug information. Note that using the `-g` flag causes the programs to be much larger and run much slower, so this flag should only be used if the created executable file is to be debugged.

Once the program file has been created, it can be run in `gdb`. First, make your console screen include as many columns and rows as possible, as the larger the screen, the more information that can be displayed.. Then use the following command string to start `gdbtui`:

```
gdb printInt -tui
```

The `-tui` option says to use the text user interface, which is an interface based on the Curses library. Because it is text based, this interface does not require the X-windows interface on the Raspberry pi.

The screen should look similar Figure 7.

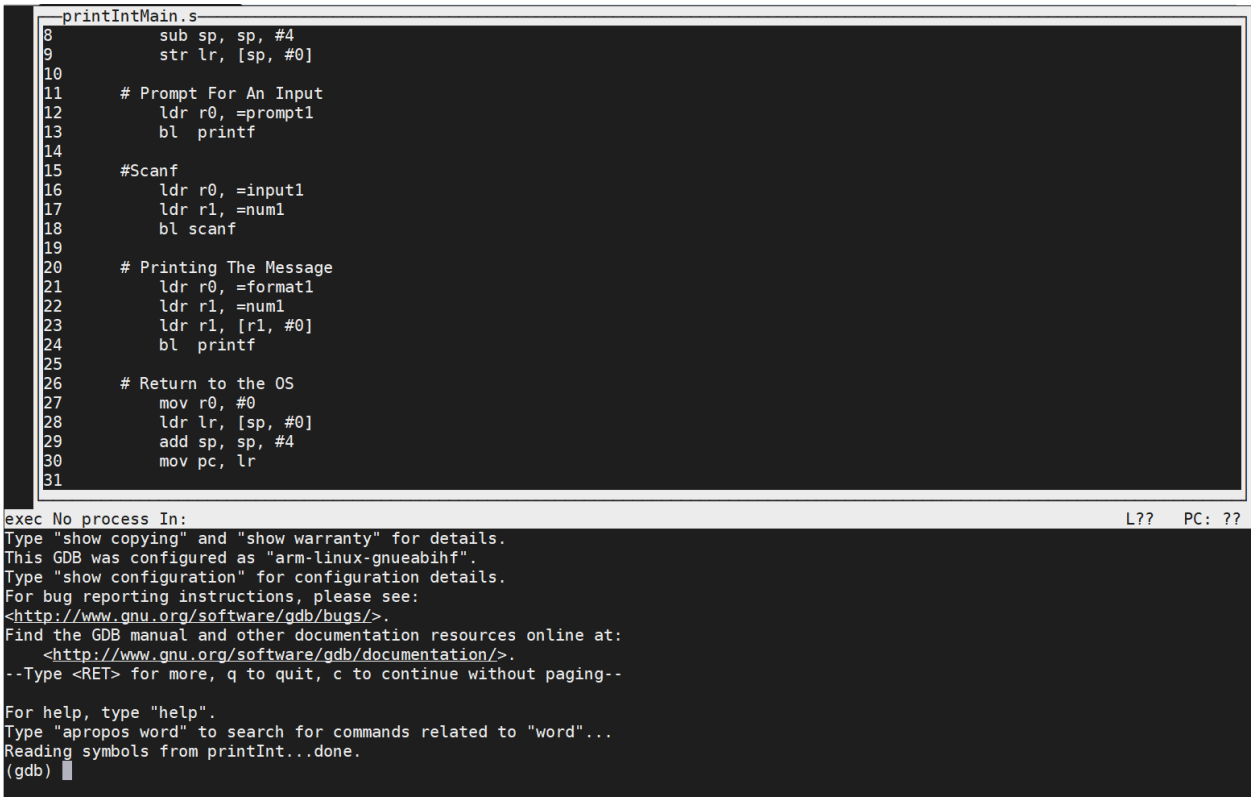


Figure 7: Initial gdb screen

The cursor and input are now focused at the bottom half of the screen, which is the command console. Type the following 3 commands to begin running the program in `gdb`:

```
layout regs

break main

run
```

These 3 commands do the following:

1. the `layout regs` command tells `tui` to show the values of all of the general purpose registers
2. the `break main` command causes the `gdb` program to stop at the first line of code in your program
3. the `run` command causes the `gdb` program to run the program to the break point at the first line of code in your file

Your screen should look similar to Figure 8, showing the registers at the top of the screen and that the program is paused at the breakpoint (`B+>` and highlighted line in the src code).

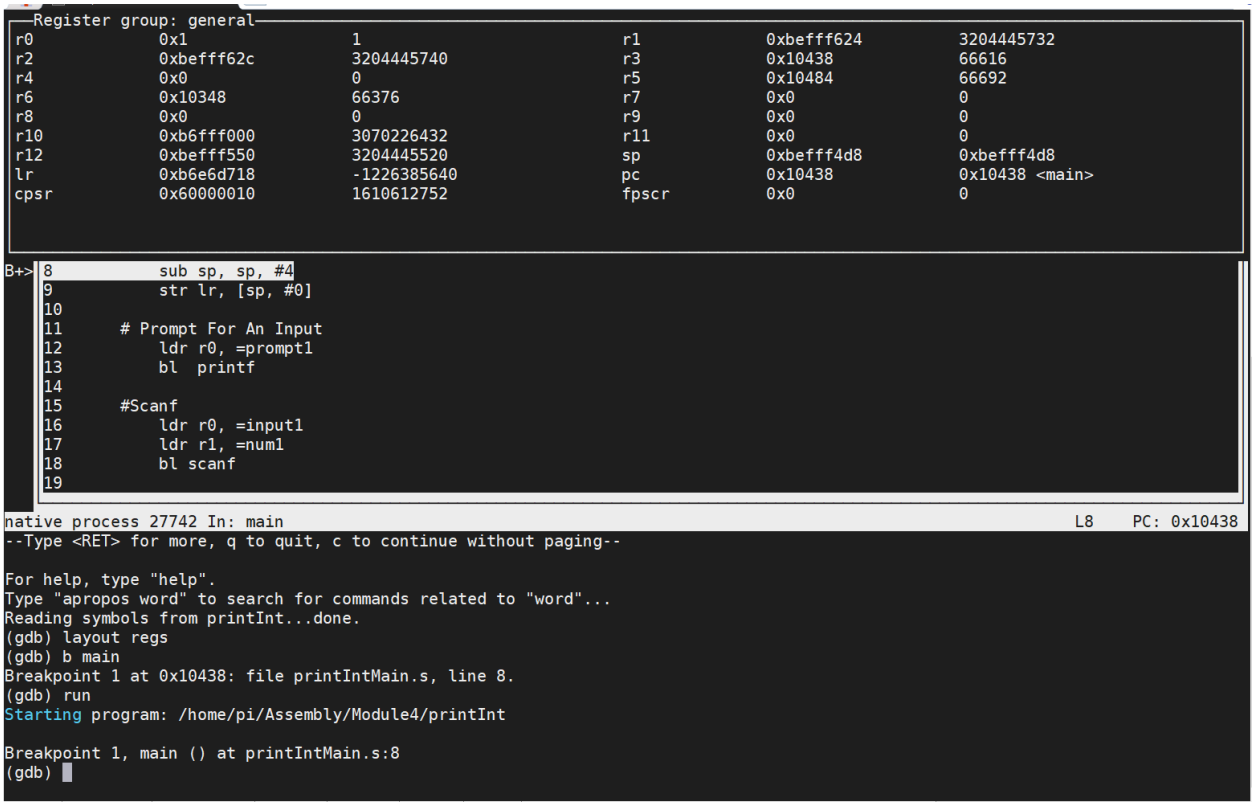


Figure 8: gdb screen with program paused at line 8 in main

The program can be advanced using the `next` command (type `next` or `n` and then `enter`), which will move the program to the next instruction. To repeat the last command that was run, only the `Enter` key needs to be pressed. You can move the program down through the source code a few statements to see this happening.

Now set a break point just before executing the `scanf` function. You can do this by typing `b line#`. In this case, the `bl printf` instruction is on line 18, so you would type `b 18`. Continue running your program by typing `c` (`continue`). When doing this command, the screen becomes messed up. This is a result of printing the string to enter the number. To reset the screen, type `ctrl-l`, (while holding down the control or apple key and typing the l character) which will redraw your screen.

The screen should now appear like Figure 9 with the program now ready to call `scanf`.

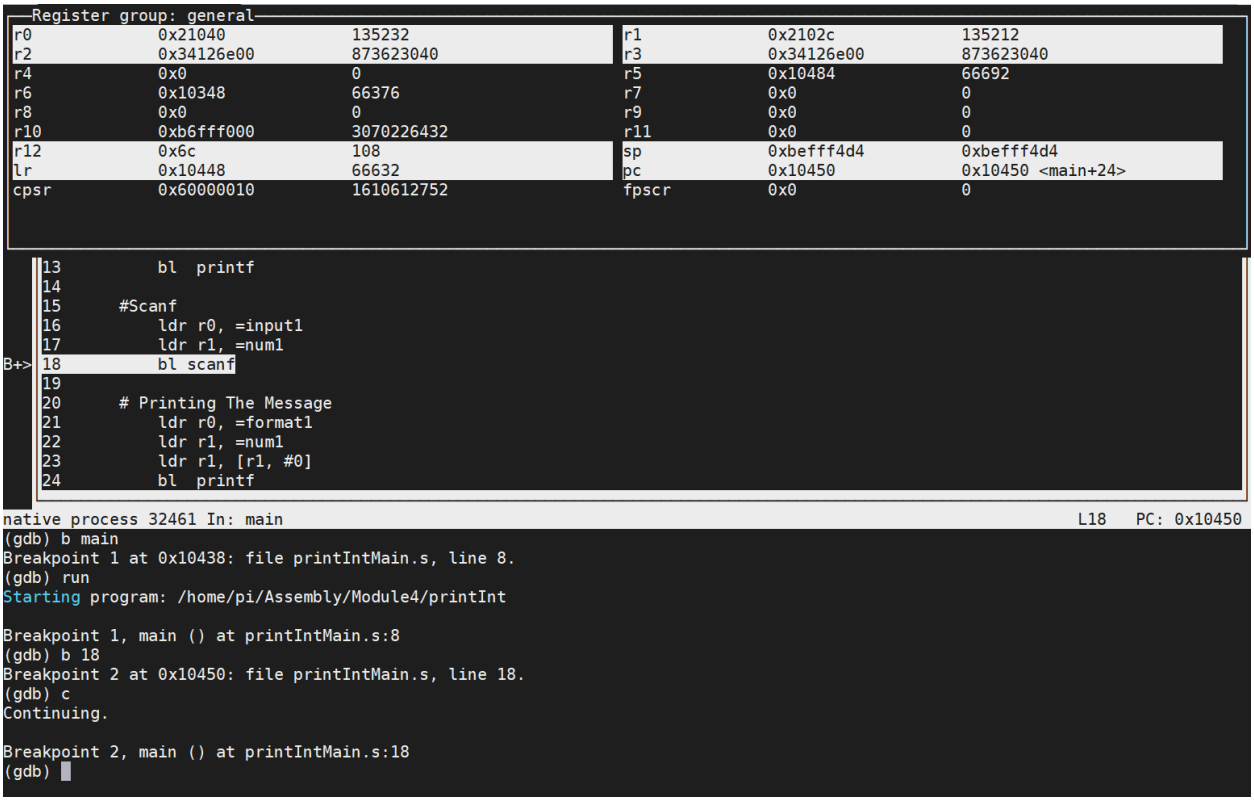


Figure 9: gdb immediately before the call to scanf

Before running `scanf`, examine the values in the program registers and variables. Looking at `r0`, you can see that it has loaded the address of the variable `input` as `0x21040`. We can check that by typing `print &input` in the console to see the addresses match. To see the value at the address of `input`, the `x` (eXamine) command can be used. The command "`x/s 0x21040`" means examine the value at address `0x21040` and to interpret the answer as a string. When this command is run, the string printed out is "`%d`", which is what we expect. This is shown in Figure 10.

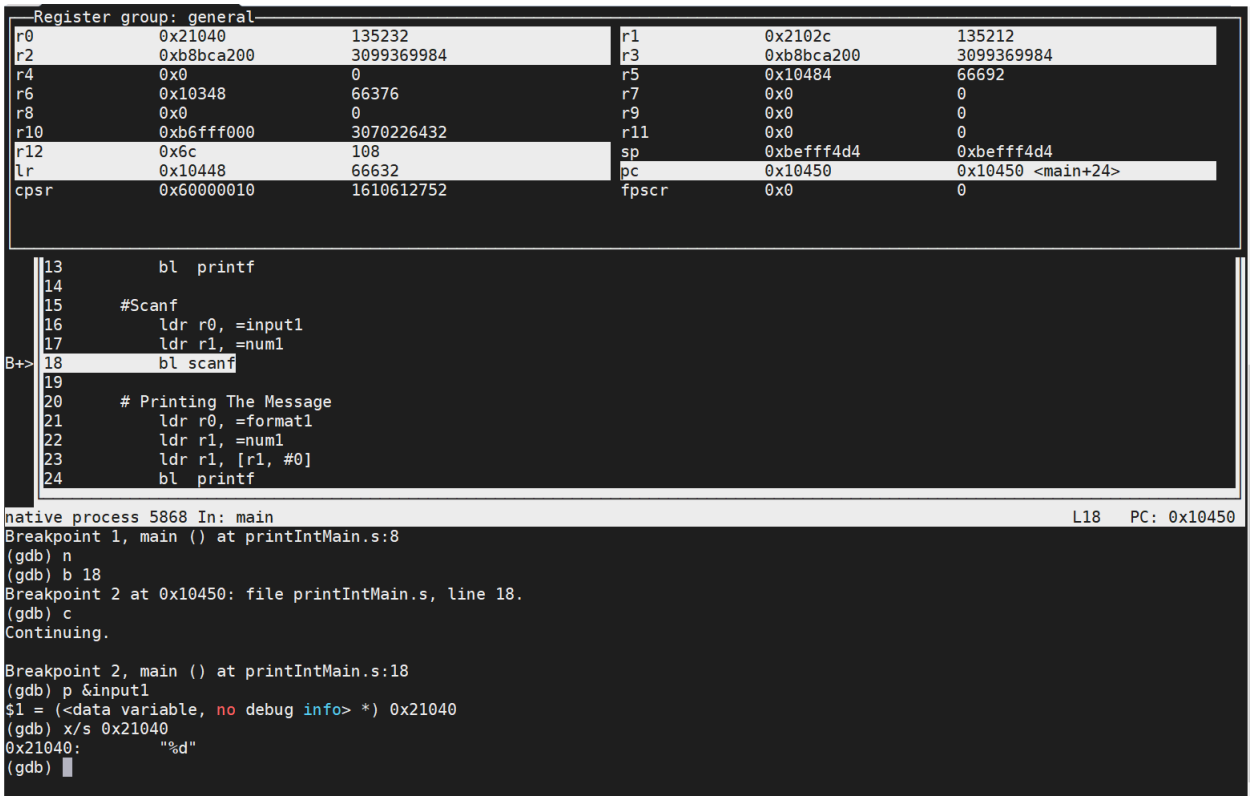


Figure 10: gdb showing addresses and values

These two commands, `print` and `x`, will be very useful in debugging programs. The `print` command will be used to get the address of labels, and `x` will be used for values at those addresses. Using the `print` command to retrieve the address of a label is always done by saying "`print &label`", where `label` is the `label` you want to examine. While the `print` command can print out the value at the `label`, it is not as useful in assembly as in a HLL since there is no type information about the label in assembly, which is available in a HLL. A label is not an address, it is just a name for an address, so a label has no type information. Printing out the value of a label will simply give a hex value that the user will have to interpret. The `x` command is just more useful for printing out values as the format specifier allows a type to be applied to the value at the address. In addition, the `x` command can be used on any address, not just labels. Readers who want to make an equivalence between labels and variables will have a very difficult time until they come to the realization that a label is not a variable.

The format of the `x` command is:

```
x/nfs
```

where `n` is the number of times to repeat the `x` command, `f` is a format specifier, and `s` is the size of the data (in bytes) to be printed. If the string `x/6d4` is specified, the next six four byte signed integers would be printed. This would be useful for an array of 6 integer values.

The formats of the `x` command are outlined in the following table.

Format Symbol	**Meaning**
`x` (default)	Regard the bits of the value as an integer and print the integer in hexadecimal.
`d`	Print as integer in signed decimal.
`u`	Print as integer in unsigned decimal.
`o`	Print as integer in octal.
`t`	Print as integer in binary. The letter `'t'` stands for "two".
`a`	Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located.
`c`	Regard as an integer and print it as a character constant.
`f`	Regard the bits of the value as a floating point number and print using typical floating point syntax.
`s`	Regard the value as a null terminated string.

Finally, to reference a register, use a dollar sign (`$`) suffix. For example, to print out the value of `r0`, use `$r0`. This will also work for the floating point (`s`) and double (`d`) registers.

Chapter 3.7 Conclusions

This chapter introduced the first programs that a programmer needs in any language, programs to make sure that the environment is set up correctly, and that input and output can be accomplished. This chapter is about much more than just getting a first program to work. Many concepts that will be explained in greater detail and used in the rest of this book were covered. For example:

1 using the `ldr` operation

2 accessing the C `printf` and `scanf` functions from assembly

3 value and reference variables

4 the assembler, linker, and run program

5 `makefile` scripts

6 the basics of using `gdbtui`

Readers who found this chapter difficult should take heart. It was filled with new information and new concepts. Assembly language is a very different type of language, and it will require some effort to learn the new way to think.

Chapter 3.8 Problems

1 Implement a program to prompt the user for their name and age, and using 2 calls to `printf`, have the program output the following string on one line:

The user *name* is *age* year old.

Your program should substitute the name and age of in the string with the name and age the user of the program entered. For example, if the user entered "Chuck" and "62", the program should print "The user Chuck is 62 years old."

2 If the references or values to be printed in a `printf` statement fit in a word (32 bits), up to 3 values can be printed using `r1`, `r2`, and `r3` as parameters to the `printf` statement. The `printf` statement only needs to specify that it needs to print 3 values. For example, the following code fragments would print out the integers stored in `r1`, `r2`, and `r3`, producing "`int1 = 5, int2 = 7, and int3 = 9`".

```
    mov r1, #5
    mov r2, #7
    mov r3, #9
    ldr r0, =print3ints
print3ints: .asciz "int1 = %d, int2 = %d, and int3 = %d"
```

Write the format statements to print 2 strings and an int, and 2 ints and a string, and set up the `bl printf` call to properly print out these strings.

3 Implement a program that outputs the following table. The `printf` statements should output the name using `%s` and the number using `%d`, with each name and age stored in at a different label address. The table should be formatted using the tab (`\t`) characters to move to tab settings and newline (`\n`) to put data on a new line.

4 For the following program, print out the address of the num and name variables in `gdb`. Print out the value of each variable also. Submit a screen shot of your program.

5 Input a floating point value using `%f` for `scanf`. Print the value you entered using `printf`. Note that `%f` input for `scanf` reads a float, but the `printf` uses a double value that is stored in `r1` and `r1`. To convert from a float (32 bits) to a double, use the command "`vcvt.F64.F32 D5, S14`". This command converts the value in the float (single precision) register `S14` to the double precision register pair `D5` (registers `S10` and `11`).

What you will learn

In this chapter you will learn:

1 why this chapter covers the ARM instruction set as a 3-address load and store architecture

2 what is an Instruction Set Architecture (ISA)

3 a 3-address architecture

4 how to initialize registers using the `mov` instruction

5 simple arithmetic and logical operations in ARM

6 using shift operations

7 memory access and Von Neumann vs Harvard architectures

8 a 3-address load and store architecture

9 how to read and write memory

10 the different types of memory access in ARM and when to use them

Chapter 4 3-address instruction set

At its core the ARM instruction is a 3-address load and store CPU. Because the assembly language is built to support the underlying architecture, the ARM assembly instruction set is also based on 3-address load and store instructions. A 3-address CPU says that the instructions will have an operator and 3 operands (a destination register and two source values). The source values can be either two registers, or a register and a number. So, instructions in this chapter will be of the format[14]:

```
Operator r1, r2, r3
```

or

[14]In ARM assembly you can use either 2-address of 3-address formats. If a 2-address format is used, the first register is used for both the destination and first source register. Thus the 2-address instruction

```
ADD r1, r2
```

is equivalent to

```
ADD r1, r1, r2
```

The 2 address format can be used in place of 3 address instructions, but will be discouraged in this textbook.

```
Operator r1, r2, #number
```

Many programs can be implemented using only a 3-address instruction subset of the ARM instructions. This chapter will present the ARM instructions as 3-address instructions. Note that this will be done using completely valid ARM instructions, no fake or pseudo instructions will be covered. The instruction set will be developed by restricting the inputs to some operations.

To actually understand the ARM CPU, a model closer to a real ARM CPU will be developed in Chapter 5, the MSCPU. This CPU will be modified to extend the instructions in this chapter. It will also clarify some questions that may arise from having simplified the architecture. For example, when documenting the instructions in this chapter the use of registers will appear arbitrary. When to use `Rn` verses `Rs` will appear to be arbitrary, and `Rm` will sometimes be the second register in the instruction, and other times the third register in the instruction. The order of the registers and immediate values will be inconsistent. The instructions in this chapter will work as documented, but the reasons for the differences will be explained in Chapter 6.

Chapter 4.1 Instruction Set Architecture (ISA)

When implementing a CPU, a virtual representation must first be created to specify a number of architectural decisions that must be made. Obvious things such as:

1 Will the CPU be a Complex Instruction Set CPU (CISC) or a Reduced Instruction Set CPU (RISC)?

2 What are the supported data types, for example will the CPU support ints, shorts, characters, floats, doubles, etc?

3 How big to make the data items, such as integers and addresses. Often this is related to and called the word size for a CPU.

4 The format of data items, such as using two's complement for integer values, IEEE 754 format for floats, and ASCII for characters. The format for all supported data types must be specified.

5 How many registers will exist and how they will be used.

6 How is data transferred between units in the CPU, called the processor *datapath*.

7 How will data be provided to the ALU, or more specifically will the computer be a 0-address (stack), 1-address (accumulator) or 2/3-address (general register) organization.

8 When designing a CPU, memory can be accessed directly by instructions, or the access to memory can be limited to load and store instructions. The ARM CPU falls into the latter category of a load and store architecture.

9 Is there a unified memory, called a Von Neumann or Princeton architecture, or is the memory divided between text and data segments, called a Harvard architecture.

This section will address specifically the last three issues: the type of datapath for the data, how memory is accessed, and the memory organization.

The ARM architecture is a modified version of a *3-address, load and store architecture*. Therefore, the rest of this chapter will be cover a generic version of 3-address load and store architecture that is capable of running ARM instructions.

Chapter 4.2 3-Address CPU

At the most basic level, there are 3 main components to a CPU: the ALU, memory (registers[15]), and a Control Unit (CU). This is illustrated in the following figure showing a very basic and generic 3-Address CPU with 8 registers.

[15]Any program data storage that is part of the CPU is called a register. Any program data storage that is not inside the CPU is called memory. This definition means even on die cache is called memory. If data storage is not part of the CPU, it is not a register.

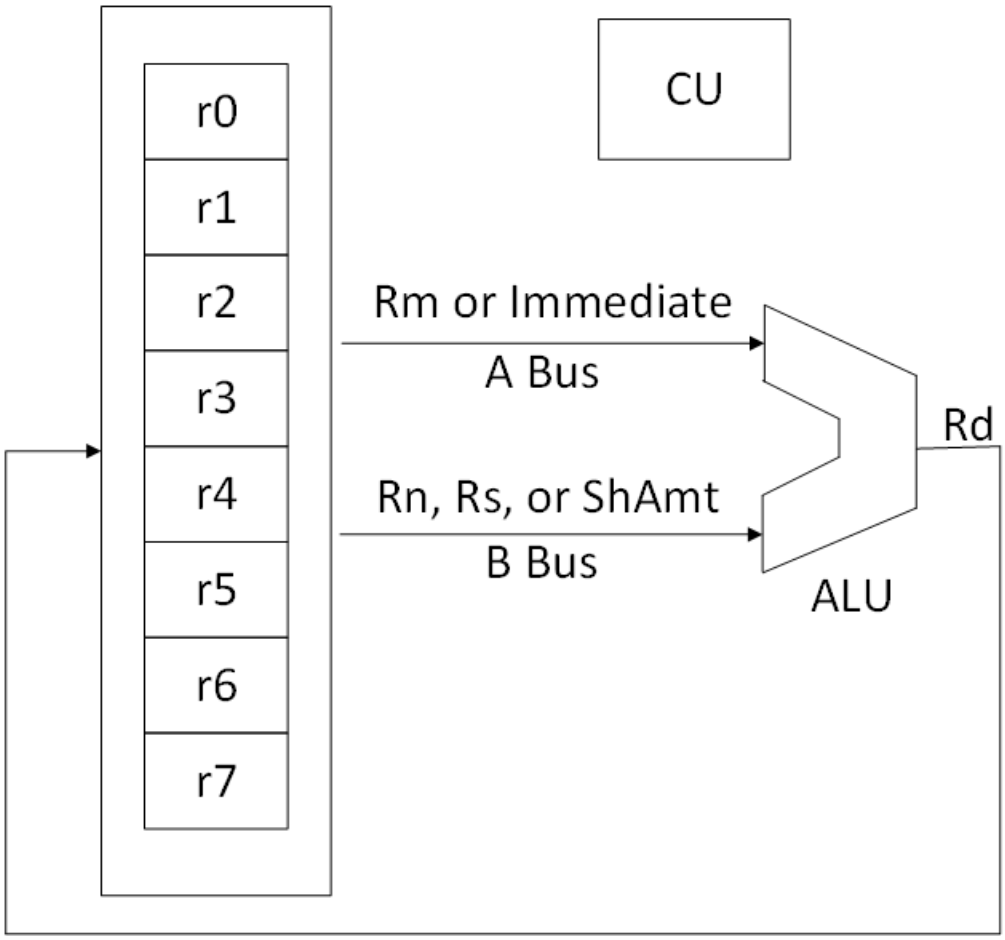


Figure 11: 3-Address CPU

There are a number of details that need to be covered regarding this diagram. First, there are 16 registers accessible in user mode in the ARM architecture. The first 12 will be usable in this chapter, though the first 4 (0-3) are somewhat different from the next 8 (4-12). This difference will be covered in Chapter 7 on functions. The final 3 registers are used for special purposes in ARM and have special names. The register `r13` is called the *stack pointer* (`sp`), register `r14` is called the *link register* (`lr`), and register `r15` is called the `pc`. These registers are special in the ARM CPU and should only be used for their intended purposed. These registers will be covered in chapter 7.

Next, the lines with arrow in the diagram are actually 32 wires, each carrying one bit, and together called a bus. The top set of wires from `Register Bank` to the `ALU` is named the `A Bus`, and the `bottom wire` from the `Register Bank` to the `ALU` is named the `B Bus`.

From the point of view of the assembly programmer, there are two basic formats for the instructions. The first format uses one destination register and two source registers.

```
OP Rd, R1, R2   // meaning is Rd ← R1 OP R2 where OP is the operator
```

The second format uses one destination register, but the source is one register and a numeric value.

```
OP Rd, R1, #num   // meaning is Rd ← R1 OP #num where OP is the operator
```

If this was a book about 3-Address assembly, this would be enough of a definition of the instructions. When writing assembly language for the problems at the end of this chapter, it is complete enough. However, this book is about ARM assembly, and so the correct registers and numeric values must follow the ARM conventions. These conventions will make more sense after covering chapter 5, but for now there are many comments online by ARM assembly programmers about how the register conventions for the instructions seem almost random. For this chapter, these conventions will almost seem random.

The data on the A Bus can either be a value from a register in the register bank, or an immediate value. An immediate value is a number that is specified as part of the assembly instruction. Note that for this 3-address CPU, an immediate value is always an 8-bit signed integer from -128 … 127[16]. In the following example, the immediate value 87 is used on the A Bus.

```
ADD r1, r2, #87
```

If the A Bus gets its value from a register, then register identifier `Rm` will be used to specify the register number. In the following example the value in `r4` is used on the A Bus, so `Rm` would be the 4-bit register identifier `0b0100`. Remember the value placed on the bus is the 32-bit value in register `r4`, not the register number. For example, in the following add instruction where `Rm` is `r4` and register `r4` contains the value `87`, the value passed to the `ALU` is `87`.

```
ADD r1, r2, r4
```

The data on the B Bus is again either a value from a register, or a number called a Shift Amount (`ShAmt`). The `ShAmt` is a 5-bit number from 0 … 31 and again is included in the instruction. For example, the following command shifts `r1` 2 bytes to the left.

```
LSL r1, r1, #2
```

If the value on the B Bus is a register, it can either be the `Rn` or `Rs` register. Most instructions will use `Rn`, but some like multiply or shift, will use `Rs`. The reason for this will be covered in Chapter 5. However, to properly document the instructions in this chapter realize that `Rn` and `Rs` both represent registers whose values are sent on the B Bus. In the following example the `Rd` register is `r1`, `Rn` is `r2`, and `Rm` is `r3`.

```
ADD r1, r2, r3
```

[16]A much larger range of values can be represented in ARM, but will all be 8-bit rotated numbers. For now, it is best to just think of the immediate values as being an 8-bit number. The ARM instruction set guide also says that there is an #imm16 version of MOV that used a 16 bit number, but that was not implemented on my Pi-0 or Pi-4, so it is not included here.

For this chapter, just think of assembly using R_1, R_2, and #num as operands, though the documentation of the operators at the end of the chapter will use the real names to be consistent with the rest of the text and real ARM assembly. And while it might seem like all of this is just useless information, it does matter, and Chapter 6 cannot be understood without it.

The next part of the CPU to look at is the CU. The purpose of the CU is to translate the assembly language into control wires used by the CPU to specify how the CPU executes a given instruction. It is beyond the scope of this textbook and so will not be covered any further in this textbook and will not appear on any future diagrams.

The last detail to cover is the most important. The main purpose of a CPU is do calculations, and those calculations are done in the calculation unit, generally the ALU[17]. Other than branching logic (which will not be covered in this chapter), the purpose of the CPU can be seen as getting the correct data from memory into registers, sending the values of registers to the ALU for execution, and storing the results back to a register, and storing register values back to memory.

There are two types of instructions that are available for this 3-address CPU. The first type of instruction uses two registers values as input to the ALU, with the ALU producing a result value that is saved to a register. This type of instruction looks as follows:

```
ADD r1, r2, r3
```

This statement says to add r2 and r3, storing the value in r1, or r1 ← r2 + r3.

The second type of instruction specifies one input register and one immediate value to a register and the register to save the value to. This type of instruction looks as follows:

```
ADD r1, r2, #3
```

This statement says to add r2 and #3, storing the value in r1, or r1 ← r2 + #3.

Note that in both cases, the add needs 3 operands (or addresses). In the first case the operands are three registers, and in the second case the operands are two registers and an immediate. But there are always 3 operands (or addresses) for every instruction, and this is why it is called a 3-address architecture.

Chapter 4.3 3-Address Instructions

Most basic instructions in the ARM CPU are simple 3-address instructions and can be explained using the CPU in Figure 4.1. This will be done by covering a simple version of the move instruction (mov), followed by addition (ADD) and subtraction (SUB), multiplication (MUL), division (IDIV, UDIV), logical operations (AND, ORR, EOR, BIC, and MVN), and finally shift (LSL, LSR, ASR,

[17]The ARM CPU will have other units that do calculations within the CPU, specifically a *Barrel Shifter* and a *Multiplier*. For the purposes of this chapter, these units do not affect the 3-Address abstraction used here, and so will be made part of the ALU. How these units work will be covered in Chapter 5.

and ROR) operations. Note that only the 3-address format of these instructions will be covered here. Other formats for these instructions will be covered in Section 4.3, or chapter 5.

Chapter 4.3.1 MOV Instruction

To make programs interesting, there must be some data in the registers to operate on. This data comes either from memory, or the instruction that is executing. To use immediate data in the executing instruction, the MOV instruction is used. The MOV instruction allows numbers to be specified and moved into a register, or values in one register to be moved to another register.

There are two basic formats for the mov instruction: a MOV that moves an immediate value into a register and a MOV that moves a value from one register to another register. The first format of this move instruction using an immediate is:

```
MOV Rd, immediate
```

An example of using the immediate format of the mov instruction is:

```
MOV r1, #36
```

This instruction moves the value of decimal 36 into the register r1, or r1 ← 36. Note that the immediate value is an 8-bit number from decimal -128...127. This value can be specified using a decimal value (#36), a hex value (#0x24), or a binary value (#0b00100100).

The second format of the MOV instruction uses 2 registers. This format of the MOV command is:

```
MOV Rd, Rm
```

An example of using this register format of the mov instruction is:

```
MOV r1, r2
```

In this instruction, the value in register r2 is moved into register r1, or r1 ← r2.

Chapter 4.3.2 ADD and SUB instructions

In this section, the add (ADD) and subtract (SUB) operators will be explained. These are 3-address instructions again with two formats: a register format and an immediate format.

The immediate format of these two instructions are:

```
ADD Rd, Rn, immediate

SUB Rd, Rn, immediate
```

An example of using the immediate format of the add and subtract instructions is:

```
ADD r1, r2, #36
SUB r1, r2, #36
```

In these instructions the value of `36` is added to or subtracted from the value in `r2`, and the result stored in `r1`. This corresponds to `r1 ← r2 + 36` in the first instruction and `r1 ← r2 - 36` in the second instruction.

The register format of these two instructions is:

```
ADD Rd, Rn, Rm
SUB Rd, Rn, Rm
```

An example of using the register format of the `add` and `subtract` instructions is:

```
ADD r1, r2, r3
SUB r1, r2, r3
```

For these instructions, the value of `r3` is added to or subtracted from the value in `r2` and the result stored in `r1`. This corresponds to `r1 ← r2 + r3` in the first instruction and `r1 ← r2 - r3` in the second instruction.

Chapter 4.3.3 MUL, SDIV, and UDIV instructions

In this section, the multiply (`MUL`), signed division (`SDIV`), and unsigned division (`UDIV`) operations are explained. The assembly instructions appear to be similar to the `ADD` and `SUB` instructions, but there are a number of differences. First, the `MUL` operation will not have an immediate format. The reason the `mul` instruction does not have an immediate form is that if the program is multiplying or dividing by a constant, there is always a better way to implement the operation using shifts and adds. So, there is no utility to implementing an immediate form of the operations.

The second difference is that the `sp` (`r13`) and `pc` (`r15`) cannot be used in the multiply or divide instructions.

Finally, the inputs to the `MUL` instructions will be `Rm` and `Rs`, not `Rn` and `Rm`.

The register forms of these three instructions are:

```
MUL Rd, Rm, Rs
SDIV Rd, Rn, Rm
UDIV Rd, Rn, Rm
```

An example of using the register format of the `mul, udiv,` and `sdiv` instructions is:

```
MUL r1, r2, r3
```

```
SDIV r1, r2, r3
UDIV r1, r2, r3
```

For these instructions, the value of `r3` is multiplied by or divided into the value in `r2` and the result stored in `r1`. This corresponds to `r1 ← r2 * r3` in the first instruction and `r1 ← r2 / r3` in the second and third instruction.

Chapter 4.3.4 Division on a Raspberry Pi

Because of the amount of circuitry involved, some ARM chips did not implement division in hardware. My current Pi-0 and Pi-4 do not implement the divide instructions. Instead, the function `__aeabi_idiv` is used to do division. If you are using a CPU that does not implement division, use the following code fragment. The `__aeabi_idiv` function does integer division. The input dividend is in `r0` and the divisor is in `r1`. The quotient, which is the return value of the function, is in `r0`.

```
MOV r0, #12
 MOV r1, #3
 BL __aeabi_idiv
 # The quotient 4 is in r0
```

7 Division on a Pi Zero

Chapter 4.3.5 Logical Operations: AND, OR, XOR, and BIC

There are four logical operations in ARM assembly: AND (`AND`), OR (`ORR`), XOR[18] (`EOR`), and Bit Clear (`BIC`). Also, included in this section is the move negative (`MVN`), which can be used to negate the value in a register and thus be considered a NOT operation.

To begin the discussion of these instructions, the definition of a logical operation will be covered. In a HLL, the logical operations all resolve to a single bit indicating if an expression is true or false. To distinguish between these logical operations and operations that are applied to all 32 bits in a register of memory, the term logical operation applies to operations that resolve to a single bit of true/false and bit-wise operations are operations applied to all the bits in a word.

This distinction of logical and bit-wise operations is turned on its head in assembly language. What are called *logical* operations in ARM assembly are bit-wise instructions, e.g., they operate

[18]Pronounced X-OR, not zor, is short for Exclusive Or, hence the instruction mnemonic `EOR` in ARM assembly.

on the 32 bits in the register. The equivalent of a HLL logical expression can be implemented by making the top 31 bits 0 and using only the lowest order bit to maintain the logical value[19]. For this chapter, the logical (32 bit) operations in ARM assembly are covered. The use of logical operations (as used in HLL) will be covered in Chapter 9 on procedural coding, where it will be used for branching.

To show how bit-wise operations can be used, consider a function to convert an ASCII upper case character to a lower case character. In ASCII, an upper case `'A'` is `0x41`, and a lower case `'a'` is `0x61`. The two are different by the bit `0x20`, or the 6th bit of the byte. This difference in the 6th bit is true for all upper and lower case ASCII characters. Thus, an upper case character can be converted to a lower case character by turning on (or *setting*) the 6th bit, which is accomplished by using an `ORR` operation with a *bit mask*[20], where only the 6th bit is set. The following code fragment will result in the 6th bit (bit 5 from the right) being turned on and leaving all the other bits exactly as they are in the original data. This has the effect of converting an upper case character "C" to a lower case character "c".

```
MOV r1, #0x42    // put a character 'B' in r1
MOV r2, #0x20    // bit-mask for upper to lower case
ORR r1, r1, r2   // r1 now contains the character 'b'
```

The meanings of the `AND` (AND) and `ORR` (OR) instructions are what they appear to be: they take two 32 bit registers and AND or OR all the bits against each other in a bit-wise fashion.

The other Boolean operations are `EOR` and `BIC`. The XOR (`EOR`) is not needed for completeness in Boolean logic, but it is a very useful operation because it can be used to simplify Boolean equations. The exercises at the end of this chapter give some interesting uses of the `EOR` operation. The characteristic truth table for the XOR is in the table below.

Input		Output
X	**Y**	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

[19]This is what most HLLs do also. Boolean variables are often 32 bits.
[20]A bit-masking involves manipulating specific bits to turn them on or off. The mask (in this case ORR with 0x20) allows the programmer to pass through all bits in the byte as is, except for the 6th bit, which the operation sets.

The Bit Clear (BIC) instruction performs an AND operation on the complement of the bits in Rn. An example would be the reverse of the upper case to lower case conversion in the previous example using the ORR. In this case to convert a lower case character to an upper case character the 0x20 bit needs to turned off. This can be done in the following code fragment.

```
MOV r1, #0x63    // put a character 'c' in r1
MOV r2, #0x20
BIC r1, r1, r2
```

Finally, the MVN instruction moves the inverse of the bits in one register to another register, sometimes called the *one's complement.*

The immediate forms of these five instructions are:

```
AND Rd, Rn, immediate
ORR Rd, Rn, immediate
EOR Rd, Rn, immediate
BIC Rd, Rn, immediate
MVN Rd, immediate
```

An example of using the immediate format of the AND, ORR, EOR, BIC, and MVN instructions is:

```
AND r1, r2, #0xdf  // convert lower case to upper case
ORR r1, r2, #0x20  // convert upper case to lower case
EOR r1, r2, #0xff  // negate the least significant byte of r2
BIC r1, r2, #0x20
MVN r1, #0x20      //  store the inverted value of 0x20 (0xdf) in r1
```

The register format of these instructions is:

```
AND Rd, Rn, Rm
ORR Rd, Rn, Rm
EOR Rd, Rn, Rm
BIC Rd, Rn, Rm
MVN Rd, Rm
```

An example of using the immediate format of the AND, ORR, EOR, BIC, and MVN instructions is:

```
AND r1, r2, r3
ORR r1, r2, r3
```

```
EOR r1, r2, r3

BIC Rd, r1, r2, r3

MVN Rd, r1, r2
```

Chapter 4.3.6 Shift Operations

Shift operations allow bits in a register to be moved within that register, filling in the bits that were moved with either 0's (logical shift and arithmetic shift of a positive number), 1's (arithmetic shift of a negative number), or the value immediately to the left (right rotate). The shifts operations all have two formats, one using a `ShAmt`[21], which is a 5-bit number between 0 and 31 and a register format[22]. These instructions are documented in the following sections.

LSL instructions

The Logical Shift Left (`LSL`) operation shifts bits from the right to the left in a register. The bits that are moved into the register are given a value of a binary '0', and the bits that are shifted out are simply lost. The formats for the `lsl` operation are:

```
LSL Rd, Rm, #ShAmt

LSL Rd, Rm, Rs
```

In these instructions, the value in `Rm` is shifted by the amount in either the `ShAmt` if a constant is given, or by the amount in the `Rs` register. If the size of the shift using the register is greater than 5 bits (> 31), only the 5 least significant bits are used, effectively producing a value of `n%32`, or the remainder from dividing the size of the shift by 32. An example of using both formats is the following.

```
LSL r1, r2, #2

LSL r1, r2, r3
```

The result of running the first operation is illustrated in the following diagram. In this diagram, all 32 of the bits would be shifted, but only some of the bits are shown with arrows indicating they are shifted. Note that the bits are all moved two places to the left, with the two highest order bits moved to the bit-bucket[23], and the two lowest order bits being set to '0'.

[21]Note that while the ShAmt seems to be equivalent to the immediate value, there are differences between these two values. This will be covered in Chapter 5 when they are shown in the CPU data path.
[22]Note that the shift operations use register Rs, like the multiply operation. This will matter when converting your instructions into machine code, and will be explained in Chapter 5.
[23]A bit-bucket, sometimes call /dev/null in Unix, is just an imaginary place where bits that are thrown away go.

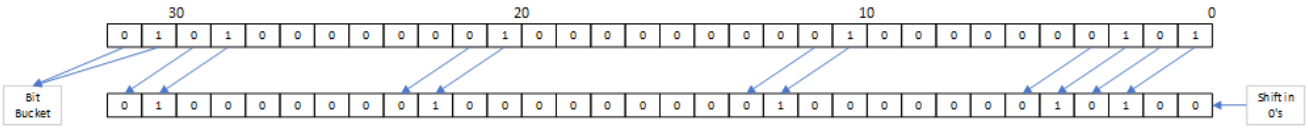


Figure 12: Left Shift Logical

LSR instructions

The Logical Shift Right (`lsr`) is the opposite of the `lsl` operation in that it shifts bits from the left to the right in a register. The bits that are moved into the register are given a value of a binary '0', and the bits that are shifted out are simply lost. The formats for the `LSR` operation are:

```
LSR Rd, Rm, #ShAmt

LSR Rd, Rm, Rs
```

In these instructions, the value in `Rm` is shifted by the amount in either the `ShAmt` if a constant is given, or by the amount in the `Rs` register. If the size of the shift using the register is greater than 5 bits (> 31), only the 5 least significant bits are used, effectively producing a value of `n%32`, or the remainder from dividing the size of the shift by 32. An example of using both formats is the following.

```
LSR r1, r2, #2

LSR r1, r2, r3
```

The result of running the first operation is illustrated in the following diagram.

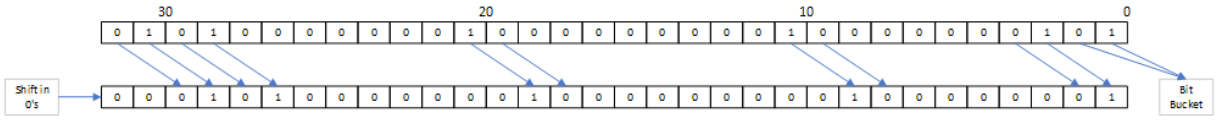


Figure 13: Right Shift Logical

ASR instructions

The Arithmetic Shift Right (`ASR`) is similar to the `LSR` operation in that it shifts bits from the left to the right in a register, and bits that are shifted out are simply lost. However, the `ASR` instruction does not always shift in a binary `'0'` to the bits moved into the register. Instead, the sign bit of the integer number is shifted in these bits. This allows operations such as the division of 2 to be done by the `ASR`. The formats for the `ASR` operation are:

```
ASR Rd, Rm, #ShAmt

ASR Rd, Rm, Rs
```

In these instructions, the value in `Rm` is shifted by the amount in either the `ShAmt` if a constant is given, or by the amount in the `Rs` register. If the size of the shift using the register is greater than 5 bits (> 31), only the 5 least significant bits are used, effectively producing a value of `n%32`, or

the remainder from dividing the size of the shift by 32. An example of using both formats is the following.

```
ASR r1, r2, #2
ASR r1, r2, r3
```

The result of running the first operation is illustrated in the following diagrams. In the first diagram, a positive integer number is shifted right, so the sign bit is '0', and the value shifted in is a binary '0'.

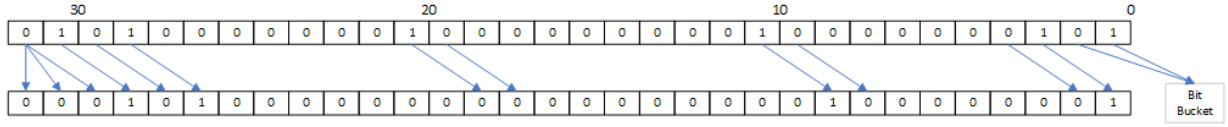


Figure 14: Arithmetic shift for positive value

In the second diagram, a negative integer number is shifted right, so the sign bit is '1', and the value shifted in is a binary '1'.

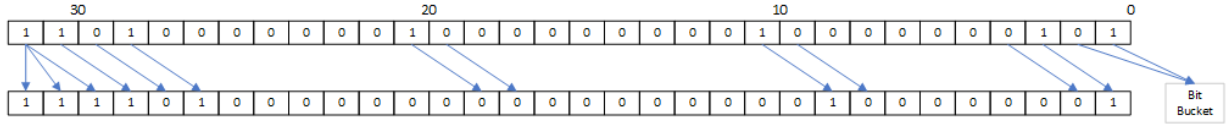


Figure 15: Arithmetic shift for negative value

ROR and RRX instructions

The Rotate Right (`ROR`) and Rotate Right extended (`RRX`) instructions allow the bits in a register to be rotated in that register. To understand this, imagine that the lowest order bit in the register is connected to the highest order bit, as in the following diagram.

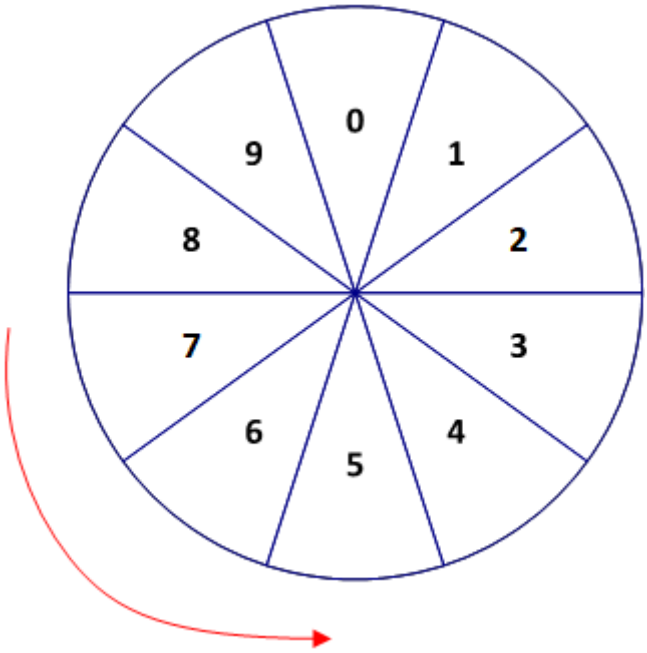


Figure 16: Rotate illustration

In this diagram the value in bit 0 is shifted into the value of bit 9, and the rest of the bits are shifted just as in the `LSR`. An example of this is given in the following diagram or an `ROR` with a 2 bit shift.

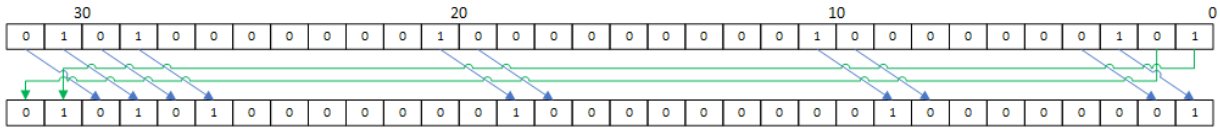


Figure 17: Rotate operation

The `RRX` instruction is similar to the `ROR` instruction, except that the carry-bit is used to store the value that is shifted out, and the current value of the carry-bit is used for the bit to set the highest order bit in the register. This is illustrated in the following diagram.

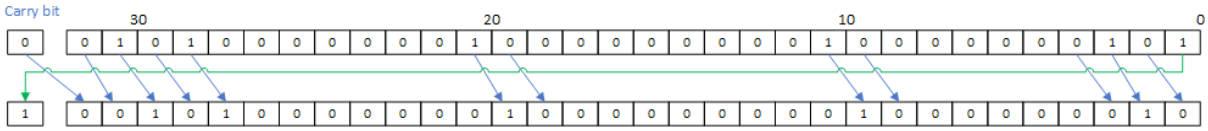


Figure 18: Rotate Extended Operation

Reasons for shift instructions

There are many reasons to do a shift operation. First, shifts make multiplication and division by powers of 2 easier to implement and faster in assembly. For example, the following examples of ARM assembly instructions multiply `r1` by 2, by 8, and divides it by 4.

```
LSL r0, r0, #1

LSL r0, r0, #3

ASR r0, r0, #2
```

Note that while these shifts could be used for multiplication and division in a HLL, they should almost never be used for this purpose, as any decent modern compiler will automatically implement the appropriate operations for the underlying architecture. The resulting code will always more likely be correct and the source program will be easier to understand. Having said that, however, in assembly there is no compiler to provide these operations, so it is correct to use bit shifting for multiplication and division in assembly.

There are many other reasons for using the shift operations. Some algorithms, such as some multiplication and division algorithms, rely on these operations. These operations can also be used to check bits that match hardware or software flags. Most reasons for using these operations deal with low level programming, such as device drivers, and as such many programmers will be able to have a successful career without ever using them. Some algorithms that illustrate the use of these operations will be given in the problems section of Chapter 9.

Chapter 4.4 Load and Store Architecture

Chapter 4.4.1 Load and Store CPU

When designing a CPU, there are two basic ways that the CPU can access memory. The CPU can allow direct access memory as part of any instruction, or only allow memory to be accessed with special instructions called *load and store* instructions. A CPU that allows any instruction to access memory normally has instructions that vary in length and requires the CPU to spend multiple clock cycles decoding the instruction and accessing memory. This is more common with Complex Instruction Set Computers, or CISC architectures, such as the Intel x86 series of CPUs.

ARM is an example of another type of CPU, called a Reduced Instruction Set Computer, or RISC architecture. One of the major design criteria when creating a RISC CPU was that all the instructions would be regular, meaning the instructions would all be the same size and require the instructions use similar amounts of time for decoding and executing the instructions. This regularity of instructions allows the CPU to be run faster and to be optimized using techniques such as pipeline designs that are difficult, if not impossible, on an CISC computer[24]. It also allows the compiler to take advantage of a smaller number of more regular instructions, rather than a large number of generic complex instructions. Being able to compile programs using behavior specific for a program rather than using generic complex instructions allows compilers to better optimize code, making programs faster.

RISC instruction set computers do not allow all instructions to access memory, but have special operations to load and store data to registers. All internal CPU units use registers for input and cannot access memory directly. External data in memory must first be loaded into a register before it can be used.

[24]It can be argued that a CISC CPU can be pipelined, as the Intel X86 architecture is CISC and is also a pipeline architecture. However, the Intel X86 processor has a front-end process that translates the instruction into micro-operations, or uops. Their uops basically represent a RISC architecture and it is this internal architecture that is pipelined.

The CPU architecture illustrated in the following block diagram is the CPU architecture from Figure 11 with the load and store instructions added. This block diagram CPU is now a 3-address load and store architecture.

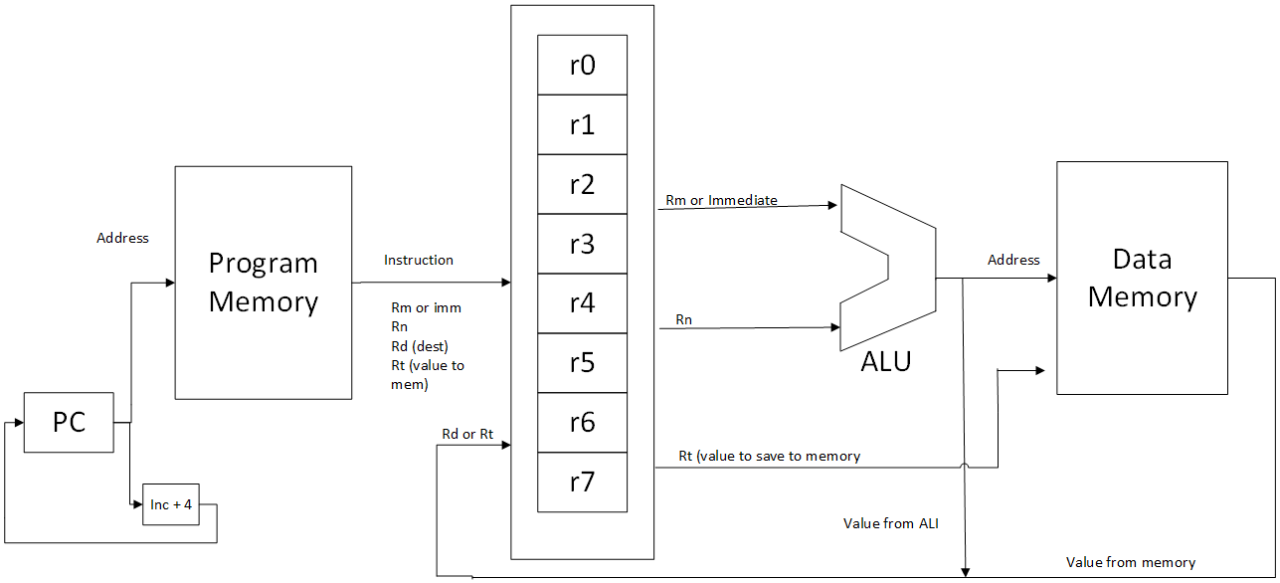


Figure 19: 3-address load and store CPU

Note that this diagram behaves exactly as the CPU in Figure 11 for any 3-address instruction; the changes between this CPU and the 3-address CPU only impact the new load and store commands. How the 3-address CPU maps into this new CPU is shown in the diagram below.

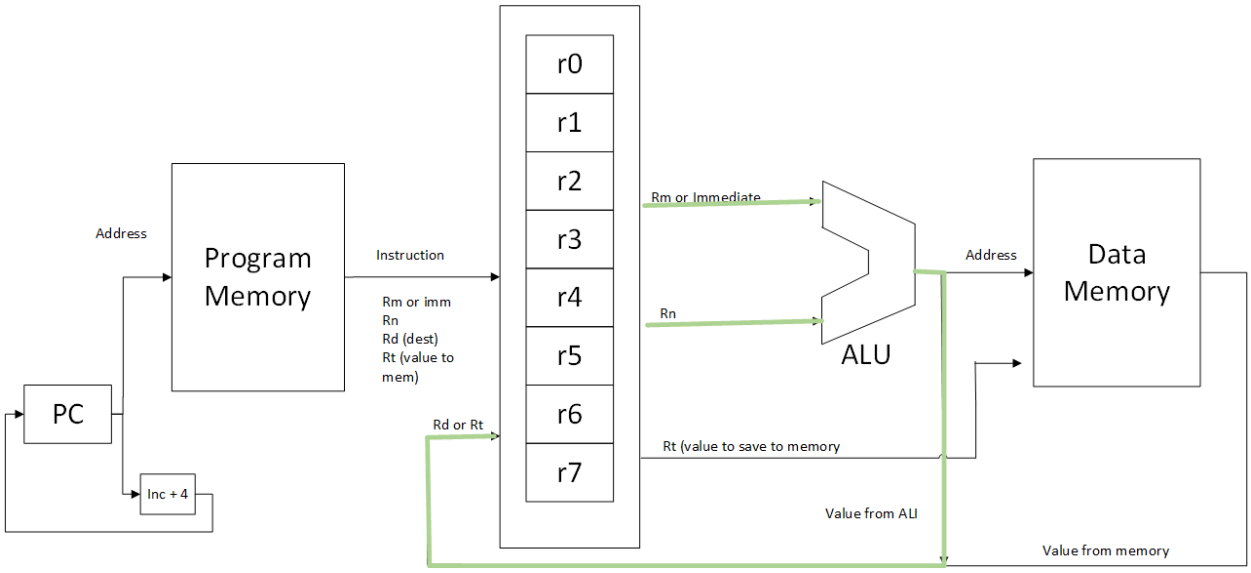


Figure 20: 3-address load and store CPU highlighting 3-address datapath

The changes for all the CPUs in this text will follow this strategy of allowing all previous programs to work, but enhancing the CPU to add new features. If you first identify how the previous CPU is manifested in the new diagram, it will be easier to understand than trying to take into account all of the complexity of the new diagram without a context.

Figure 20 shows that for 3-address instructions, the `ALU` still gets the same values from registers or immediate values, produces an output, and sends the output back to the Register Bank.

The changes in the new CPU are due to adding loading and storing of data from memory to a register. The operation for saving data to a register is illustrated in Figure 21.

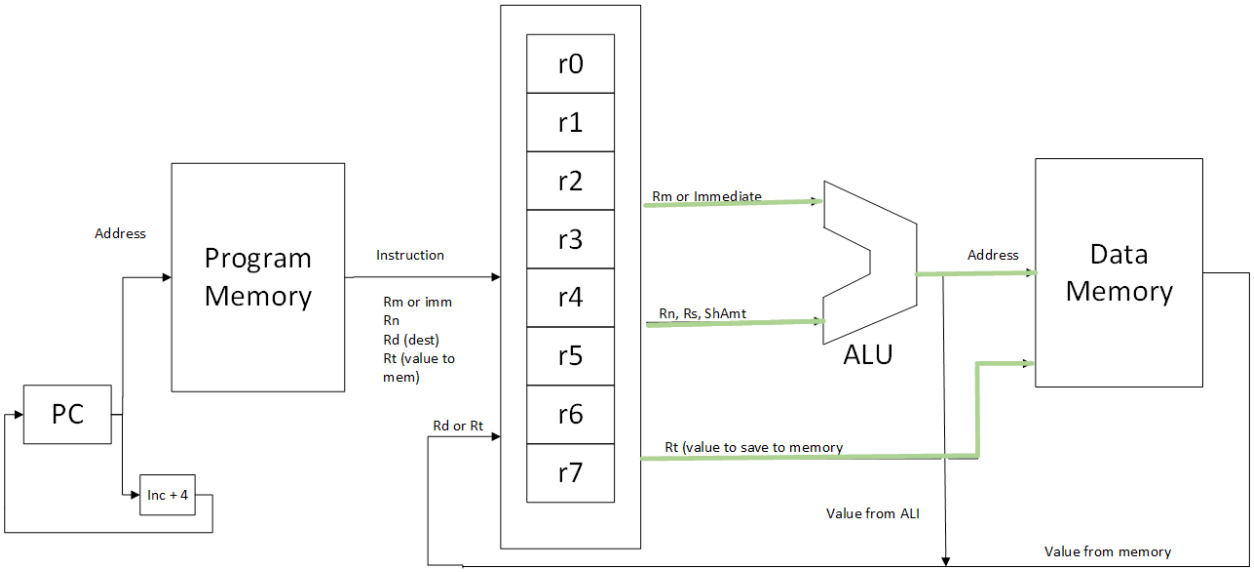


Figure 21: 3-address load and store CPU highlighting store operation

As shown in this diagram, the `Rn` register is added to either the `Rm` register or the `immediate` value to calculate a memory address from which to load the data and the data from the `Rt` register is put on the C bus to write to memory.

Next, figure 22 illustrates the loading data to the memory. As in Figure 21, the address is calculated in the ALU and passed to the data memory. The data memory reads the value at that address and places that value on the bus to send it back to the register bank. Note that this memory value is placed on the same bus as the output form the ALU. At the point where the results of the ALU and the result from a memory read collide a hardware component, called a multiplexer or MUX, will be inserted to choose which value to pass on to the register bank.

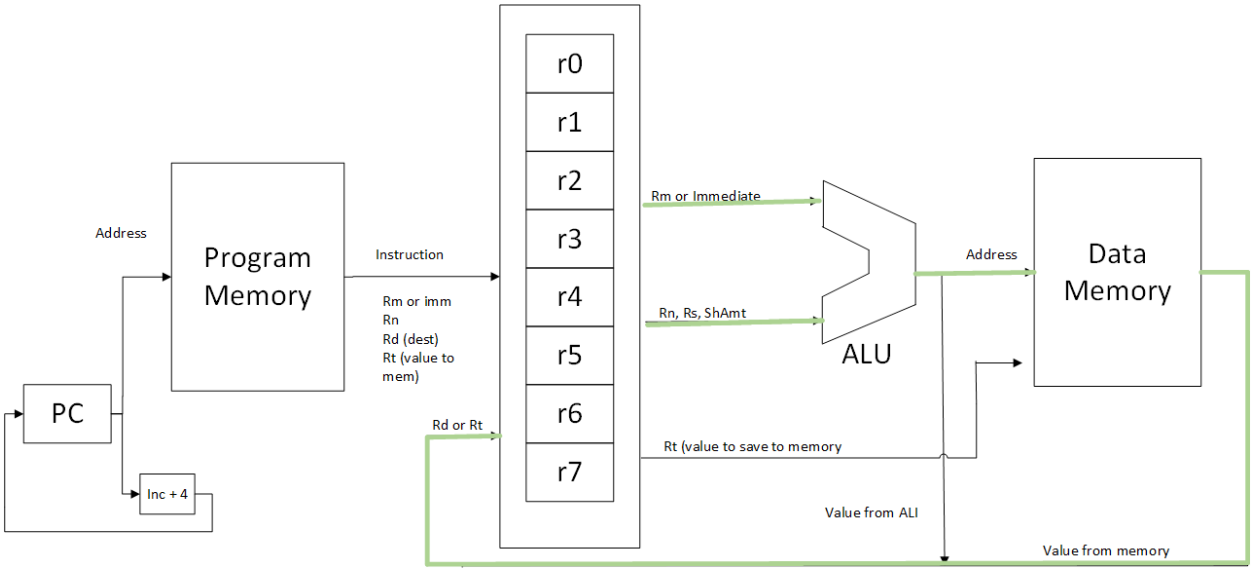


Figure 22: 3-address load and store CPU highlighting store operation

The instruction format for the `LDR` and Store Register (`STR`) instructions can add either an immediate or register value to the `Rn` value to calculate the memory address, so there are two formats for each instruction. The four formats are:

```
LDR Rt, [Rn, immediate]

LDR Rt, [Rn, Rm]

STR Rt, [Rn, immediate]

STR Rt, [Rn, Rm]
```

Examples of using these instructions are:

```
LDR r1, [r2, #4]

LDR r1, [r2, r3]

STR r1, [r2, #4]

STR r1, [r2, r3]
```

One very important detail about the immediate value in these instructions is that it is 12 bits, not the 8 bits that this CPU used to limit the value for immediate values in the Data Operations instructions.

The first `LDR` instruction adds the value representing a base memory address (in this case `r2`) to the immediate value in the address calculation (in this case `#4`) to produce the memory address of the value to load into `r1`. This statement says `r1 ← M[r2 + 4]`.

The second LDR instruction adds the value representing a base memory address (in this case r2) to the register value in the address calculation (in this case r3) to produce the memory address of the value to load into r1. This statement says r1 ← M[r2 + r3].

The first STR instruction adds the value representing a base memory address (in this case r2) to the immediate value in the address calculation (in this case #4) to produce the memory address in which to store the value in r1. This statement says M[r2 + 4] ← r1.

The second STR instruction adds the value representing a base memory address (in this case r2) to the register value in the address calculation (in this case r3) to produce the memory address of the value to load into r1. This statement says M[r2 + r3] ← r1.

Note that there is a new register, Rt, which is r1 in both of the instructions above. Rt is different from other registers because for a LDR instruction it is the register to which the value retrieved from memory is stored and for the STR instruction it specifies the register value to store back to memory. Thus, it is given a separate designation from the other registers.

In addition, in the LDR and STR instructions the ALU no longer produces a value to be stored back to a register, but instead produces the *address* of a *memory location* from which to either read or store a value. For the LDR instruction the memory location is read and the value put on the bus is passed from the memory to registers.

For the STR instruction, the value to write to memory is read from register Rt and sent to the data input port on the Data Memory. The address of where to write the value in memory is produced by the ALU and sent to the data address port on the Data Memory and the value is written into memory at that address.

Chapter 4.4.2 Auto incrementing of the Rt register

The ARM assembly language has a useful feature when executing load and store operations; it allows the Rn register to be automatically updated with the value that was calculated with the memory address used. This is called auto-incrementing the register. The following examples illustrates how to specify auto-incrementing.

No auto-incrementing:

```
LDR r1, [r2, #4]
STR r1, [r2, r3]
```

With no auto-incrementing, the value or r2 is not changed when executing a LDR or STR instruction.

Pre-incrementing

```
LDR r1, [r2, #4]!
STR r1, [r2, r3]!
```

With pre-incrementing, the value or `r2` is changed before calculating the address, and the new address is used when executing a `LDR` or `STR` instruction.

Post-incrementing

```
LDR r1, [r2], #4
STR r1, [r2], r3
```

With post-incrementing, the value or `r2` is changed after calculating the address, and the old address is used when executing a `LDR` or `STR` instruction.

Chapter 4.4.3 Von Neumann vs Harvard Architecture

The next issue is whether or not a single memory exists that contains both text and data (Von Neumann architecture) or whether the memory is split between text and data segments (Harvard architecture). The ARM computer is a Von Neumann architecture and this can have a notable impact on the machine and executable code that is produced. However, the implementation of ARM memory allows the actual access of memory that can *appear* as if the memory is split between text and data. This means the ARM CPU can act like it has a Harvard architecture. This has a large impact on the CPU design, as the 3-address design in Figure 20 divides the memory into a text memory and an address memory. This has little impact in this chapter, but takes on an oversized importance in the actual implementation of the CPU.

Chapter 4.4.4 Addressing modes in ARM assemblye

The previous section on the `LDR` and `STR` commands explained the format of the commands, but not how to use them in ARM assembly. This section will show the different addressing modes, or way to access variables, in ARM assembly. The addressing modes to be covered are immediate, direct, register direct, register indirect, register indirect with offset, indirect, and PC relative addressing.

Immediate Addressing

In immediate addressing the value to be used is included in the instruction itself. An example of immediate addressing is the following `ADD` instruction:

```
ADD r1, r2, #12
```

Note that an immediate value is very different than a constant. An immediate value is not stored in data memory, but is part of the instruction, so there is no need to load the value into a register before using it. A constant is a value stored in memory, like any other variable, except that its value cannot be changed. To use a constant, the value must be loaded into a register, which makes using a constant potentially more expensive to use than an immediate value.

Direct Addressing

With direct addressing, the address of a value is known. An example is when the value is stored at a label. Consider the following code fragment:

```
LDR r1, =num
LDR r1, [r1, #0]
num: .word 5
```

In this code fragment, there is a known address of the value to be loaded. In this example, first the address of `num` (not the value `5`) is loaded into register `r1`. The address in the register is then used to load the value into `r1`.

Register Direct

A value is a register direct if the value is stored in the register. In the following instruction, register direct addressing is used for both of addends.

```
ADD r1, r2, r3
```

Register Indirect

When using register indirect addressing, the address of the variable to use is stored in the register, and the value at that address is loaded into a register to be used later. This can be useful in a number of situations to access data. For example, the following would be an efficient way to access an array of integer data allocated beginning at the address of the label `arr`. In this case `r1` will be incremented to the next element value as each array element value is loaded into `r2`.

```
LDR r1, =arr
LDR r2, [r1], #4
.data
    arr: .word 10
```

Register Indirect with offset

Register indirect with offset addressing is the most efficient and effective way to handle addressing when there is a base address and values that are located at some known offset of that

base. An example is structures or classes and will be used extensively later in this text when discussing the program stack. To see this, consider the following Java class:

```
class A {
    long a;
    int b;
}
```

For an address pointing to the start of the class in `r1`, the value for each variable can be loaded into `r2` using the following instructions:

```
LDR r2, [r1, #0]
LDR r3, [r1, #8]
```

Indirect

Indirect addressing is like register indirect addressing except that the value in memory is actually an address to another value. In fact, the value that is addressed can be an address itself. For example, consider the following code fragment:

```
    LDR r1, =addr_num
    LDR r1, [r1, #0]
    LDR r1, [r1, #0]
.data
    num: .word 5
    addr_num: .word num
```

In this code fragment, the value at the address of label `addr_num` is the address of (reference to) the address of label `num`. To find the actual value, the address chain, or chain of references, must be traversed until the final value is found.

This becomes interesting because it gives an insight into the relationship between references and values. A value is simply the data at the address of a reference. References point to values and what the value is depends on its relationship.

This leads to the definition of two new terms: a *reference type* and a *value type*. A reference type implies that the value at the address is a reference to another value. A value type is a final value in the reference-value chain and is a value that is a program value.

PC relative

This is likely the most important type of addressing in ARM assembly. However, it requires that the concept of the `PC` be covered in some detail in Chapter 8. So, this type of addressing will be covered in detail as part of Chapter 8.

Chapter 4.5 Conclusion

This chapter presented a type of CPU called a 3-Address Load and Store CPU. This CPU was then used to present a subset of the ARM instruction set. This 3-address CPU will be used to build a more complete ARM CPU in the next chapter.

For many purposes, such as an Introduction to Computer Organization course, this 3-address CPU will be sufficient. The rest of this textbook will be written so that concepts can be presented with nothing more than this abstract CPU. This will include translation into machine code, branching, and a simple datapath will be presented for this CPU.

For readers more interested in understanding ARM assembly, it is suggested that they continue to study the ARM instruction set in Chapter 5.

For readers using only a 3-address instruction set, these commands are presented in Appendix 1.

Chapter 4.6 Problems

1 Write the function mod that calculates a modulus (remainder) of a division. This is the % operation in Java and C. For example, 14%3 = 2 (quotient is 4, remainder is 2).

2 Using only multiplication and integer numbers (no floating point numbers allowed) calculate 60% of 9. Use this trick on the following problems to get get

3 Write a program to convert temperature from Celsius to Fahrenheit and from Fahrenheit to Celsius. How close to the actual values are your answers? What changes can you make to improve your program? Rewrite your program to include these changes.

4 Write a program to convert feet to inches and inches to feet (output your answer using the format 5' 3").

5 Write a program to convert kilometers to miles and miles to kilometers.

6 Write a program to accept miles and hours, and calculate an approximate miles/hour.

7 Write a program that loads the character 'A' (upper case A) into a register, and using only logical operations convert the character to 'a' (lower case a).

8 Write a program that inverts the bits in a register using the `eor` (not the `mvn`) instructions.

9 Write a program that swaps two values using a temporary variable.

10 Write a program that swaps two values without using any temporary variables. Note that this program will use the `eor` instruction.

11 Write a program that reads an integer number from a user and calculates the 2's complement of the number, e.g., inverts the number (e.g., positive → negative or negative → positive).

12 Write a program using only shift and add operations that multiplies a number by 8.

13 Write a program using only shift and add operations that multiplies a number by 10.

14 Write a program using only shift and add operations that multiplies a number by 7.

15 A constant value can be more expensive to use than an immediate value. Why? Assume that the value at the address of cons is a constant variable. What is strange about the following statement? Would you ever see such a statement in a program?

```
LDR r1, =cons
STR r1, [r1, #0]
cons: .word 12
```

What you will learn

In this chapter you will learn:

1 using the MSCPU to run ARM instructions

2 why the ARM flexible operand, or operand2, exists

3 the differences between `Rs` and `Rn`

4 how a multi-cycle CPU functions

5 ARM instructions that use operand2

6 pre and post indexed load and store instructions

Chapter 5 A more complete ARM Instruction Set

This chapter continues to build on the previous 3-address abstract CPU using the ARM instructions. The chapter will create an abstract CPU that can run a larger subset of the ARM instruction. This architecture will be implemented to include a data path, implemented in Logisim Evolution, that will be able to run these ARM instructions.

Chapter 5.1 Abstract MSCPU

The following is an abstract view of an ARM-like CPU that implements the flexible operand, or Operand2, of the ARM CPU, and will be called the MSCPU.

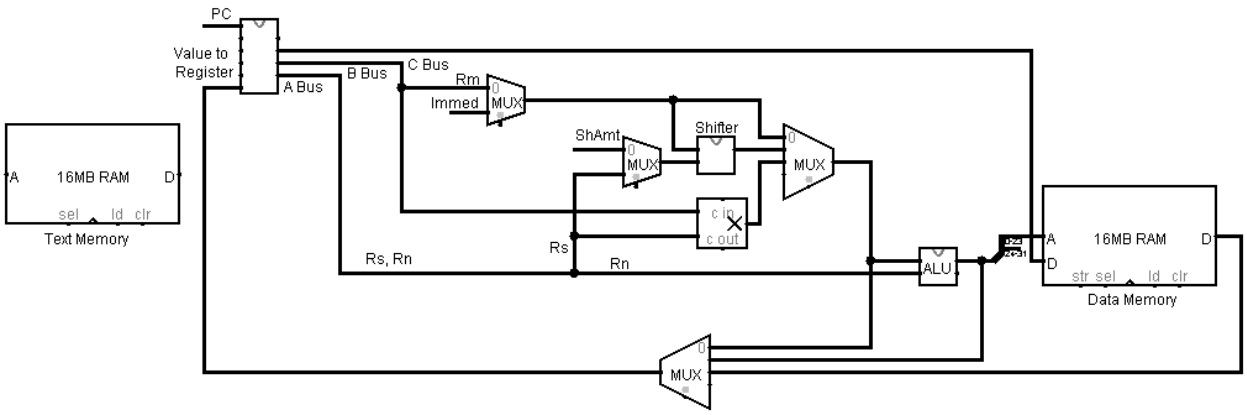


Figure 23: MSCPU

The MSCPU differs from the 3-Address Load and Store CPU in that two new operational units, the multiplier and the barrel shifter, are added to the CPU to do computation before the values are passed to the ALU. These new units allow the CPU to include the Multiply and Accumulate (`mla`) instruction and the Operand2.

This abstract CPU is also developed using Logisim rather than a drawing tool, as the complexity of the drawing is such that it is easier to implement the CPU in a circuit simulation tool, such as Logisim, than to try to draw it abstractly. Using Logisim results in some differences, like including a multiplexer (mux) to select between inputs. The reader can think of a mux as a unit to select which of 2 or more inputs is forwarded to the output. For example, the mux on the `B bus` (the mux with the `Rm` and `immed` inputs at the top center of the diagram) selects if a register, `Rm`, or an `immediate` value is sent forward.

A third difference between the 3-address CPU and the MSCPU is that there is now another bus, the `C bus`. In the MSCPU, the 3 buses have the following usages.

1 The `A bus` runs to three places: the mux to select the bottom input of the barrel shifter; the bottom input to the multiplier; or the bottom input to the ALU. This bus will have either `Rn`, `Rs`, or `ShAmt` on it.

2 The `B bus` runs to a mux that selects between the register value, always `Rm`, and an `immediate` value. The `B bus` then runs to three places: the top input of the barrel shifter; the top input to the multiplier; and a multiplexer that chooses what data to send to the top input of the ALU.

3 The `C bus` is used for the store operation to transfer the data from a register to the data memory.

The final modification is that some of the MSCPU instructions will take 2 clock cycles to run. In the parlance of CPU design, this means the CPU has a Cycles Per Instruction (CPI) of 2. All of the 3-address instructions will still have a CPI=1 as they will use only one of the units (e.g., the multiply unit, the barrel shifter, or the ALU). Only the `MLA` and instructions using the Operand2 will have a CPI=2 as these operations will require 2 cycles, one for the multiplier and shifter, and the other for subsequently using the ALU. This will be explained using the datapath for the instructions in the section below.

Chapter 5.2 Understanding the MSCPU

To understand the MSCPU, explanations and illustrations of a how a number of assembly instructions are implemented will be provided.

3-address ADD operation

The first instruction is a 3-address `ADD` operation using two register values as input. An example is shown below and an explanation will explain how this instruction works in the MSCPU.

```
ADD Rd, Rn, Rm
```

The value of the `Rm` is passed on the `B bus` directly to the ALU, and the value of `Rn` is passed on the `A bus` directly to the ALU. The ALU calculates the result and passes the value back to the Register Bank where it is stored in `Rd`. Note because only one operation unit, the ALU, is used, this operation can be run in one cycle.

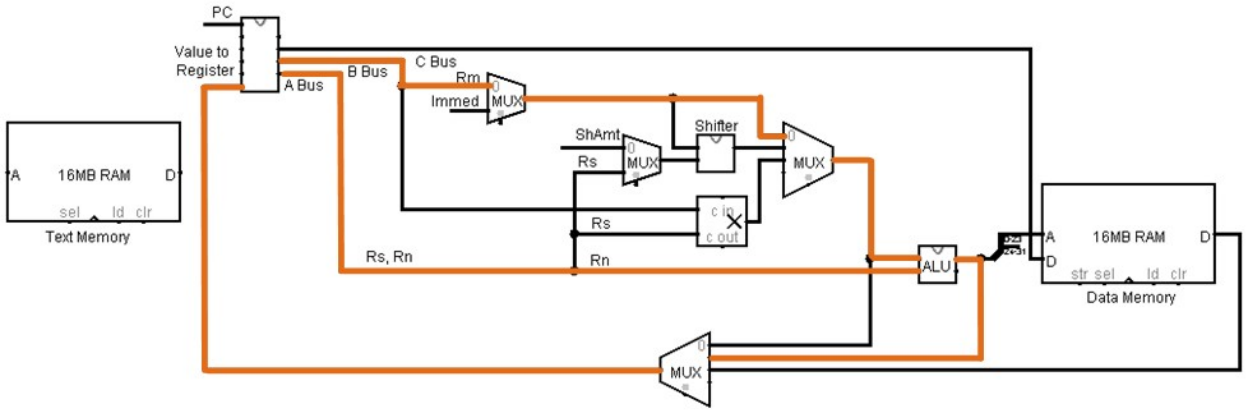


Figure 24: MSCPU ADD operation

`LSL` operation

The `LSL` operation is illustrated in the diagram below.

```
LSL r1, r2, #4
```

In this case the `lsl` operation uses the `ShAmt (4)` as input to the barrel shifter. The value on the `B bus` is the `Rm` register (`r2`). For a shift operation, the value from the shifter does not have to be sent to the ALU and so is passed directly to the mux to be selected and returned to the Register Bank to be stored in `Rd` (`r1`). Note, because only one operation unit, the barrel shifter, is used, this operation can be run in one cycle.

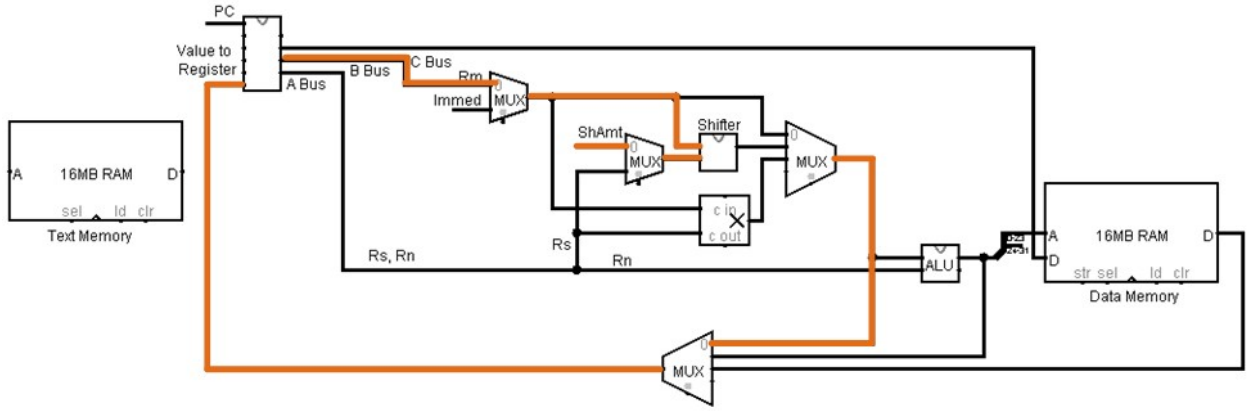


Figure 25: MSCPU LSL operation

`MUL` operation

The `mul` operation is illustrated in the diagram below.

```
mul r3, r1, r2
```

For this instruction, the registers `Rm` and `Rs` are used as input. As with the shift operation, the value from the multiplier does not have to be sent to the ALU, and so it is passed directly to the mux to be selected and returned to the Register Bank to be stored in `Rd`. Note, because only one operation unit, the multiplier, is used, this operation can be run in one cycle.

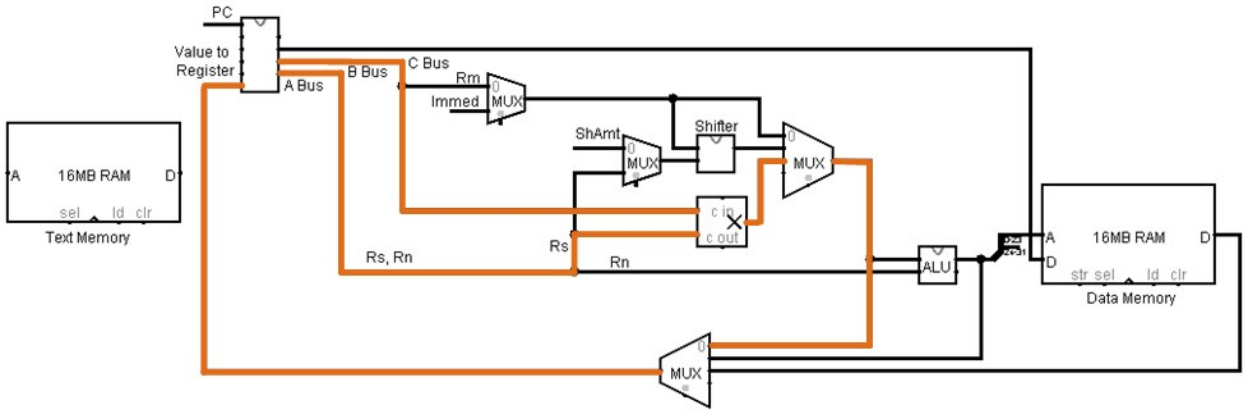


Figure 26: MSCPU MUL operation

Chapter 5.3 Adding the MLA instruction to the MSCPU

The main advantage of the MSCPU architecture is that it allows new and more complex/powerful operations in hardware. Consider a very common case where a program wants to calculate an element address (`ea`) for an element in the array. Knowing the Base Address (`ba`) of the array, the size of each element (`size`), and the index number (`idx`), the following formula will calculate the array address for that element.

```
ea = ba + (size * idx)
```

The MSCPU has put the multiply unit in front of the ALU. Over two cycles it runs the multiplication operation (cycle 1) and addition operation (cycle 2). Thus, in one instruction the CPU can calculate the `ea` using the Multiply and Accumulate (`mla`) instruction. For example, if `ba = 20`, `size = 4`, and the `index = 3`, the address of array element 3 can be calculated by the following code fragment.

```
MOV r1, #20
```

```
MOV r2, #4

MOV r3, #3

MLA r0, r2, r3, r1  // The element address (ea) is in r0
```

This `MLA` instruction tells the CPU to multiply `r2 * r3`, add the result to `r1`, and then store the result in `r0`. This will be done in two cycles. The first cycle multiplies `Rs * Rm` and passes the value on to a multiplexer, which selects the multiply unit result to forward to the ALU.

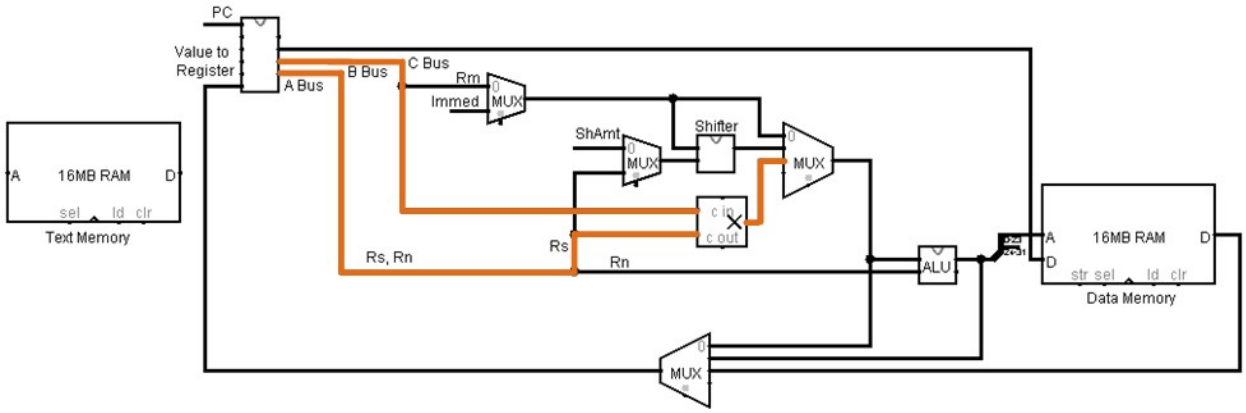


Figure 27: MSCPU MLA operation – Step 1

In the second step, the value register, `Rn`, is placed on the `A bus`. The value of the register `Rn` and the result of the multiplication are passed to the ALU, which adds the values, and passes the result back to the Register Bank to be stored in register `Rd`.

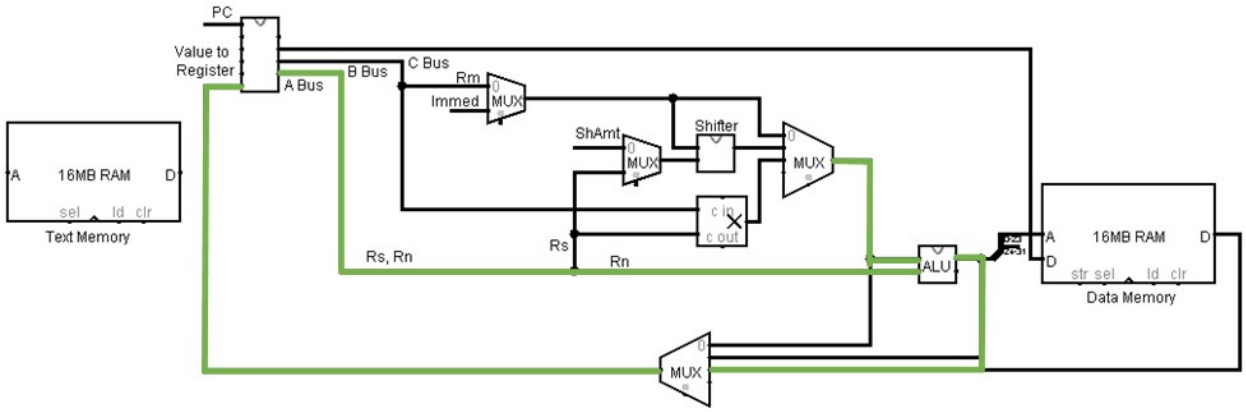


Figure 28: MSCPU MLA operation – Step 2

Since this instruction requires two operation units, the multiplier and the ALU, this instruction requires 2 cycles, or has a CPI=2.

The format for the `MLA` instruction is:

```
MLA Rd, Rm, Rs, Rn
```

An example of using the `mla` instruction is the same as given above.

```
MLA r0, r2, r3, r1
```

Chapter 5.4 Implementing the flexible operand (operand2)

Extending the `mla` example above, it follows that if the multiplication unit can be used in front of the ALU, the barrel shifter can be used for the same purpose. When the barrel shifter is used in this way, the result of the barrel shifting operation is called an operand2.

The 3-address instructions in Chapter 4 are all examples of ARM assembly instructions that use the operand2. They were simplified by restricting the operand2 to use only the format where it was one `immediate` or one `register` value. This is why the 3-address assembly is valid ARM assembly, but just a subset of ARM assembly.

The `operand2` works by first shifting the value in the `Rm` by the value in `Rs`, and passing the shifted value to the `ALU` where the `ALU` operation is performed. This is illustrated in the following diagrams. This is again a two-step instruction. In the first step, the `Rm` register value and the `ShAmt` value are passed to the barrel shifter. The barrel shifter applies the shift from the `ShiftType` and `Rs` to the value in `Rm`.

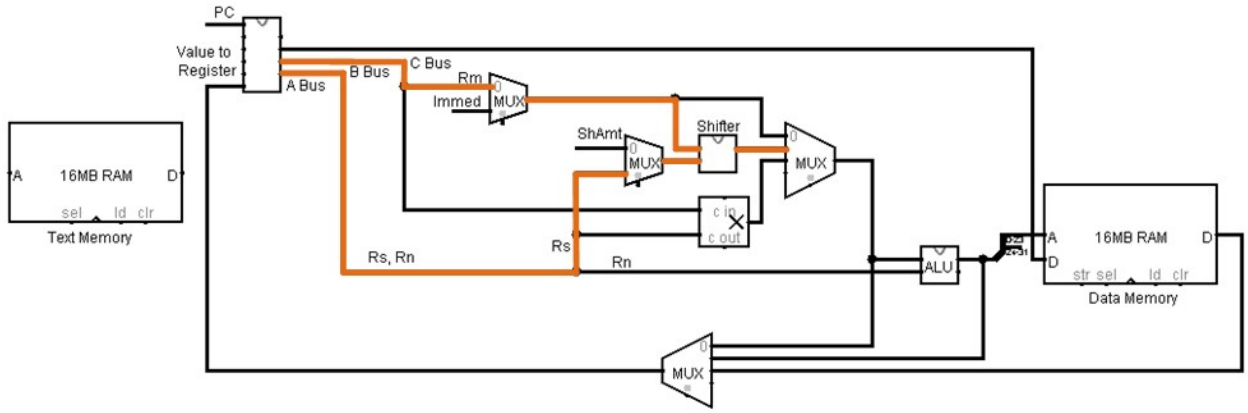


Figure 29: MSCPU Operand2 operation – Step 1

In the next step the value from the barrel shifter and value in the `Rn` register are passed to the ALU where the correct operation is performed.

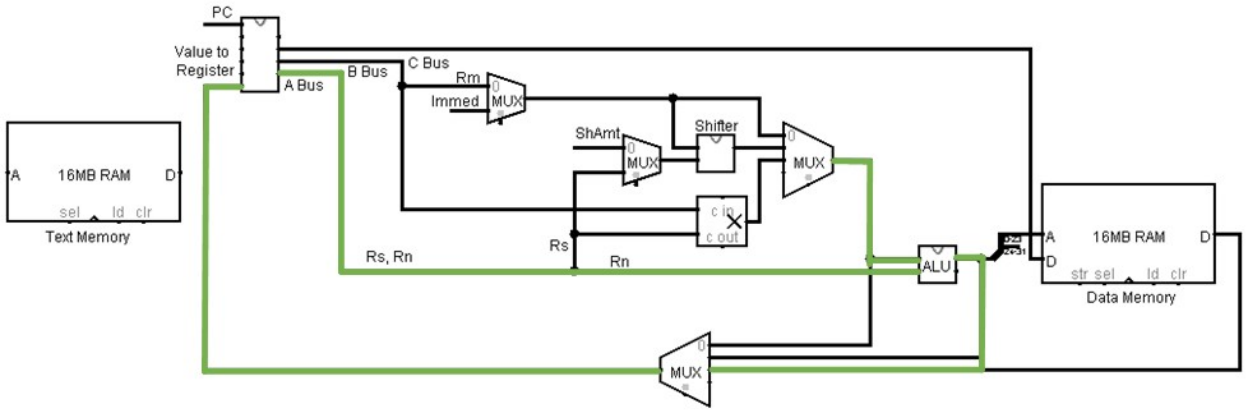


Figure 30: MSCPU Operand2 operation – Step 2

The rest of this section illustrates how the instructions from Chapter 3 will be changed when the `Operand2` is applied to the instructions.

Chapter 5.4.1 Operand2 syntax

The `Operand2` thus has the following format:

```
          <k>   :=  even number less than 32
          <m>   :=  any 32 bits that contains all zeros except for one grouping
                    of less than 8 bits that can contain 1's
          <n>   :=  any number < 32
 <immediate>    :=  # <m> | # <m> , <k>
     <ShAmt>    :=  # <n> | # <n> , <k>
   <ShValue>    :=  <ShAmt | register>
   <shiftOp>    :=  <ShType> # <ShValue>
    <ShType>    :=  lsl | lsr | asr | ror | rrx
  <register>    :=  <Rm> | <Rm> , <shiftOp>
  <operand2>    :=  <immediate> | <register>
```

This definition divides the `operand2` into two large semantic groups, immediate and register types. The next subsection will handle the immediate type of `operand2` and the following section the register type of `operand2`.

Chapter 5.4.2 Operand2 immediate Semantics

The simplest way to understand the immediate value is as an 8-bit number from (0...255) that can be rotated in the by k[25] bits in using the `ror` operation in the 32-bit register. An example of this is representing 256. 256 is too large to be represented in an 8-bit number, but all of the bits are 0 except the 9th bit. Thus 256 can be represented by an 8-bit value of 1 that is rotated right by 24 bits to put it in the 9th bit position. Thus the 1, 0x0000001, is rotated 24 bits right (which is the same as rotating it 8 bits left), and becomes 0x00000010.

Thus, the number 256 can be used as an immediate in ARM assembly. The following instruction is valid:

```
MOV r2, #256
```

Note that 256 cannot be represented in 8 bits, but it can be represented as the value of 1 rotated by 24 bits. This is equivalent of the following `mov` command[26]:

```
MOV r2, #1, 24
```

These two `MOV` instructions are exactly equivalent, as the assembler will attempt to *fix-up* any immediate constant it is given. The fix up tries to represent the value as a group of 8-bits containing 0 and 1 bits, with all the other bits being 0. This group of 8 bits are then rotated until the desired number is achieved.

While there are many numbers that can be represented using an 8-bit rotated value, there are more than cannot. For example, the number 256 requires 9 bits to represent it. This value will not be able to be fixed-up, and an attempt to run the following line of code:

```
MOV r2, #257
```

produces the error message,

```
error: invalid constant (101) after fixup[27]
```

Using the `Operand2` with the `MVN` operation will effectively double the number of constants that can be represented. For example, the number -257 can be specified as follows:

```
MVN r2, #1, 24
```

---

[25]Note k must be even. This is because the number of places that are available in the machine code instruction (covered in Chapter 6) only has 4 bits available for this value. Therefore, to be able to rotate through all 32 bits of the number with only 4 bits, the lowest order bit is dropped. This allows for shifts up to 32 bits, but only for even values.

[26]Note the second value, the amount of the rotate, does not have a hash (#) before it. Using a # before this value results in a syntax error. While this might seem inconsistent and confusing, it is the correct syntax.

[27]Note the value 101 in the error message is the hex value for 257.

The Operand2 can be used with most of the data processing operators with some exceptions, such as MOV, MUL, and all shifts. Thus, constants greater than 256 can be used in the following ADD instruction.

```
ADD r1, r2, #4096
```

To a programmer not familiar with this 8-bit rotation format, the representable constants might seem arbitrary. For example, the ADD instruction works for numbers #256 and #260, but fails for the numbers #257, #258, and #259.

Values that are not valid as a single immediate value can be built using MOV and ORR instructions. For example, to load the ASCII characters A, B, C, and D into r1 can be done with the following series of instructions:

```
MOV r2,     #0x00000064
ORR r2, r2, #0x00006300
ORR r2, r2, #0x00620000
ORR r2, r2, #0x61000000
```

Finally, all of the 3-address operations from chapter 3 (with the exception of multiply and shift instructions) are real ARM instructions that simply include the Operand2. For example, the ADD operation which was defined for an immediate in Chapter 3 as:

```
ADD Rd, Rn, Operand2
```

Chapter 5.4.3 Operand2 Register Semantics

When using a register as the Operand2 value, the value is either just the value of Rm, or the value of Rm shifted in some manner. While the immediate instruction only allowed the Operand2 to be implemented using the ror operation to be applied to an even immediate value, any type of shift (lsl, lsr, asr, ror, rrx) can be specified when using the register semantics. Also, the amount of the shift can be specified either with the ShAmt or a register value. Thus, the following are valid formats of register shifts:

```
MOV r1, r2, #4
MOV r1, r2, lsl #3
MOV r1, r2, asr r3
MOV r1, r2, ror r4
MOV r1, r2, rrx
```

All of the 3 address instructions (again, except for multiply and shifts) from Chapter 4 actually have a format (illustrated using the ADD operation) of:

```
ADD Rd, Rn, Operand2
```

By applying the `Operand2` to these instructions, many different instructions can be produced:

```
ADD r1, r2, r3, LSL 2   // multiply r3 by 4 before adding, useful for
                        // array addressing
ADD r1, r2, r3, ASR #1  // useful for division by 2, for example to find
                        // parent nodes in a Complete Binary Tree
ADD r1, r2, r3, LSR r1  // logically shift the value in r3 by the amount
                        // in r1 before adding it to r3
```

Chapter 5.4.4 Syntax for Load/Store

The Load/Store syntax from Chapter 4 is unchanged when an immediate value is used. However, if the instruction is a register instruction, it will use a *Scaled Register Format*. A Scaled Register Format is similar to a Register Format Operand2, but it is limited in that the shift amount must be a numeric value (the shift amount cannot be a register).

Chapter 5.5 Conclusions

A more complete understanding of the ARM architecture, including the barrel shifter and multiplication unit, makes it easier to understand the assembly language instructions.

Chapter 5.6 Problems

1 On the MSCPU diagram, show the datapath for each of the following instructions:

1. a. `LSL r1, r2, #3`
2. b. `ADD r1, r2, r3, LSL r4`
3. c. `SUB r4, r8, r5, ASL #2`

2 Justify the following 3-address register conventions. For example, why is the Add operation `ADD Rd, Rn, Rm`, but the multiply operation `MUL Rd, Rm, Rs`? (Hint: use the datapath diagrams)

`LSL Rd, Rm, Rs`

`MLA Rd, Rm, Rs, Rn`

3 What is the Immediate Operand2 value for the following decimal numbers? Give the 8 bit value and rotate amount.

a. 198

b. 260

c. 9216

d. 2162688

e. -75

f. -260

4 What is the decimal value for the following Immediate Operand2 values?

a. `MOV #0b1, 22`

b. `MOV #0b1001, 28`

c. `MOV #0b10010001, 30`

d. `MOV #0b1001, 16`

e. `MVN #0b100010`

f. `MVN, #0b1001, 20`

5 Which of the following decimal immediate values are valid as an operand2 value?

a. 24

b. 15

c. 34

d. 27

e. 8

6 For the following `MOV` instructions, convert them to a single immediate value or an immediate with a `ROR`.

7 What is the largest positive even number that can be represented as an ARM immediate? What is the largest positive odd number that can be specified as an ARM immediate?

What you will learn

In this chapter you will learn:

1. the purpose of machine code
2. how to decode a machine code instruction into it parts
3. how to create machine code for use by the CPU
4. the different machine code formats used by the ARM CPU
5. translation of assembly code to machine code
6. translation of machine to assembly code

Chapter 6 Machine Code[28]

Computers cannot directly read assembly code; assembly code must first be converted into a format of 32 bits containing 0's and 1's, called machine code, that is directly usable by the CPU. This machine code is then put on a 32-bit bus where the 0's values in the machine code are set to ground, and the 1's are set to a positive voltage. These wires are then decoded into segments of data that are used by the CPU to actually run the instruction.

In the first section of this chapter, an explanation is provided for how the 32-bit instruction used later in the CPU circuit is decoded. The subsequent section gives the format of the specific instructions with an example of usage for each instruction type. Finally, the last section explains how to decode an instruction in machine code to the corresponding assembly instruction[29].

Chapter 6.1 Decoding a machine code instruction

The process of decoding an instruction consists of taking the 32-bit instruction and breaking it into groups of data that are then passed on to other units in the CPU. These groupings fall into the following basic types:

1. Specific register types (e.g., `Rd`, `Rn`, `Rm`, `Rs`) that contain the register number to use for these registers (e.g., `0b0000` for `r0`, `0b0001` for `r1`).
2. Numeric data, such as the `Immediate` and `ShAmt` values. Note that the numeric values in assembly instructions can be different sizes. For example, the `ShAmt` is a 4-bit number when it represents a rotation in an `Immediate` expression, but is a 5-bit number in a

[28]The instructions in this chapter are based on the document https://documentation-service.arm.com/static/5f8dacc8f86e16515cdb865a. This is a large document that will cover all of the implementation details of ARM machine code.
[29]There are spreadsheets that are available with this distribution that summarize these instructions.

Numeric Shift Instruction. The `Immediate` value can be 4, 8, or 12 bits depending on the type of instruction. For these values, the maximum number of bits (5 bits for the `ShAmt` and 12 bits for the `Immediate`) will be sent to the other CPU units, and those units are responsible for proper processing the data.

3 Control data, bits that indicate to the CPU the type of instruction, will be forwarded to the `CU`, `ALU`, and barrel shifter to properly set up the CPU to handle the instruction. While these units are the heart of the CPU, it is complex, and is a topic separate from assembly language.

In the MSCPU, a *Decoder* unit will be placed between the Text Memory, which retrieves the 32-bit instruction from Text Memory, and the rest of the CPU which will use the parts of the instruction. A black box Decoder unit is shown in Figure 31. The input is the 32-bit instruction, and the outputs are the numeric `Immediate` and `ShAmt` values, the `Rd`, `Rm`, `Rn`, and `Rs` register numbers, and the control information.

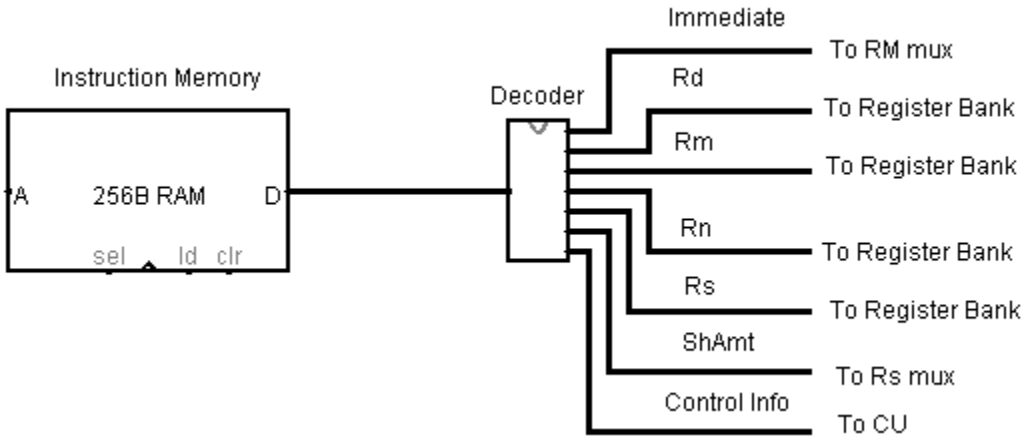


Figure 31: Decoder black box diagram

How the decoding of the instruction is achieved will now be built in stages. The first iteration will use a splitter to break the instruction into 32 separate wires and then to create register number output to forward to the Register Bank.

To start we examine Appendix 3, which gives the machine code formats of all the 3-Address instructions. The Register Instruction gives uses all the registers in a single instruction and is reproduced in Figure 32.

Register

1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code				S	Rn				Rd				Operand2 (Register Format)												
1	1	1	0	0	0	0					0																				

Figure 32: Register Instruction

This instruction shows where each of the register numbers that occur in all instructions, generally called `Rd`, `Rm`, `Rn`, and `Rs`, are located in the instructions. These registers will sometimes be referred to by other names (such as `Rt` for the `ldr` and `str` instructions), and some instructions use other register convention (such as the `mul` instruction that swaps `Rd` and `Rn`). However, these anomalous cases require the CPU to have a knowledge of the instructions being processed to know how to handle them, and since the Decoder unit has no way to know the instruction type, the register identifiers in the register instruction shown above will still be used. The responsibility for determining any anomalous meaning of the registers will be the responsibility of the downstream processing units, in this case the Register Bank. The decoder simply moves 4 bits from a specified position in the instruction to the 4-bit output value. In this case, bits 0-3 are moved to the `Rm` output, bits 8-11 are moved to the `Rs` output, bits 12-15 are moved to the `Rd` output, and bits 16-19 are moved to the `Rn` output. This is shown in the first iteration of the Decoder below.

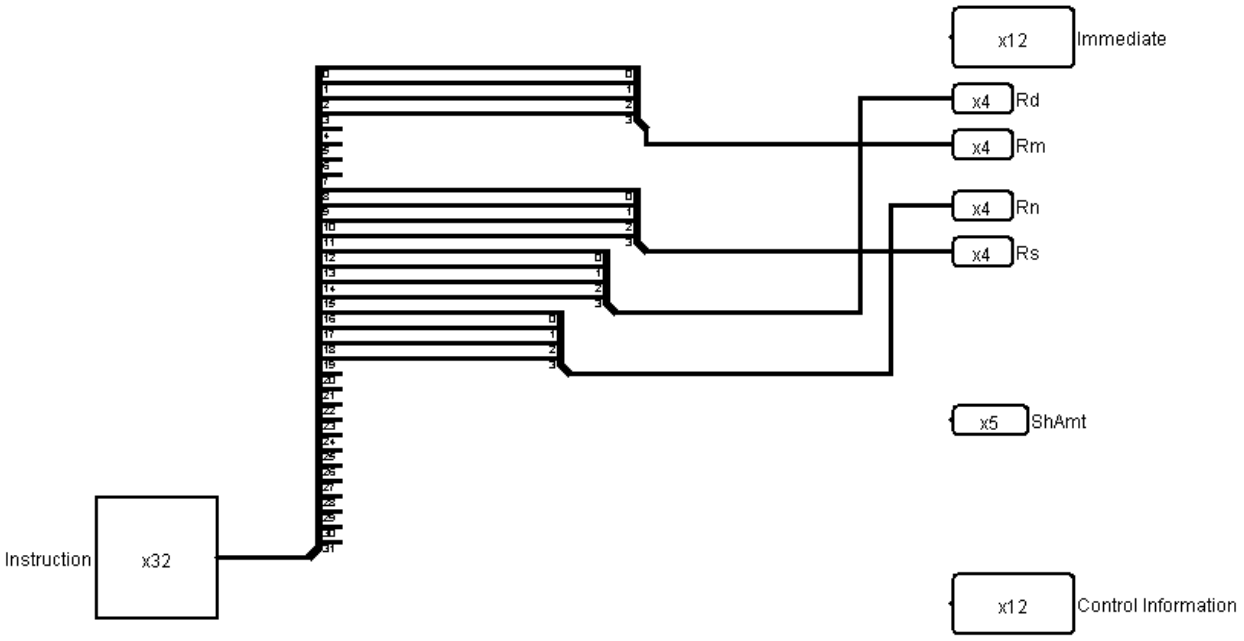
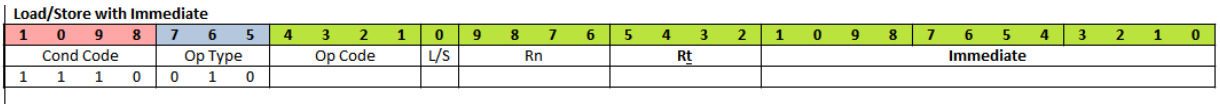


Figure 33: Decoder showing the mapping of instruction to register numbers

The next iteration of the decoder will process the output to the `Immediate` and the `ShAmt`. To see where these numeric values occur in a Register Format, `Operand2` with `ShAmt` and a Load/Store Immediate Instruction are reproduced in Figures 34 and 35. The maximum `ShAmt` is instruction bits 7...11, and the maximum immediate value is instruction bits 0...11.

1	0	9	8	7	6	5	4	3	2	1	0
Register Format Operand2 with ShAmt											
ShAmt					ShiftType		**0**	Rm			

Figure 34: Immediate Instruction



Load/Store with Immediate

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
Cond Code	Op Type	Op Code	L/S	Rn	Rt	Immediate		
1 1 1 0	0 1 0							

Figure 35: Load/Store Immediate Instructions

Once again there are anomalous conditions where the `ShAmt` and `Immediate` value are different sizes depending on the instructions. Again, the largest value is parsed from the input, and the downstream CPU units are responsible for using the values correctly.

Note that both the `ShAmt` and `Immediate` have wires that overlap the wires for the registers. This does not cause any problems as the overlap occurs for different instructions, thus the values for the registers and immediate values are always correct for the instruction being executed. Again, since the Decoder has no way to determine which instruction is being executed, or how many bits to put on the output wires, it will always provide the maximum number of bits to these values, relying on other units to adjust to the proper number of bits used.

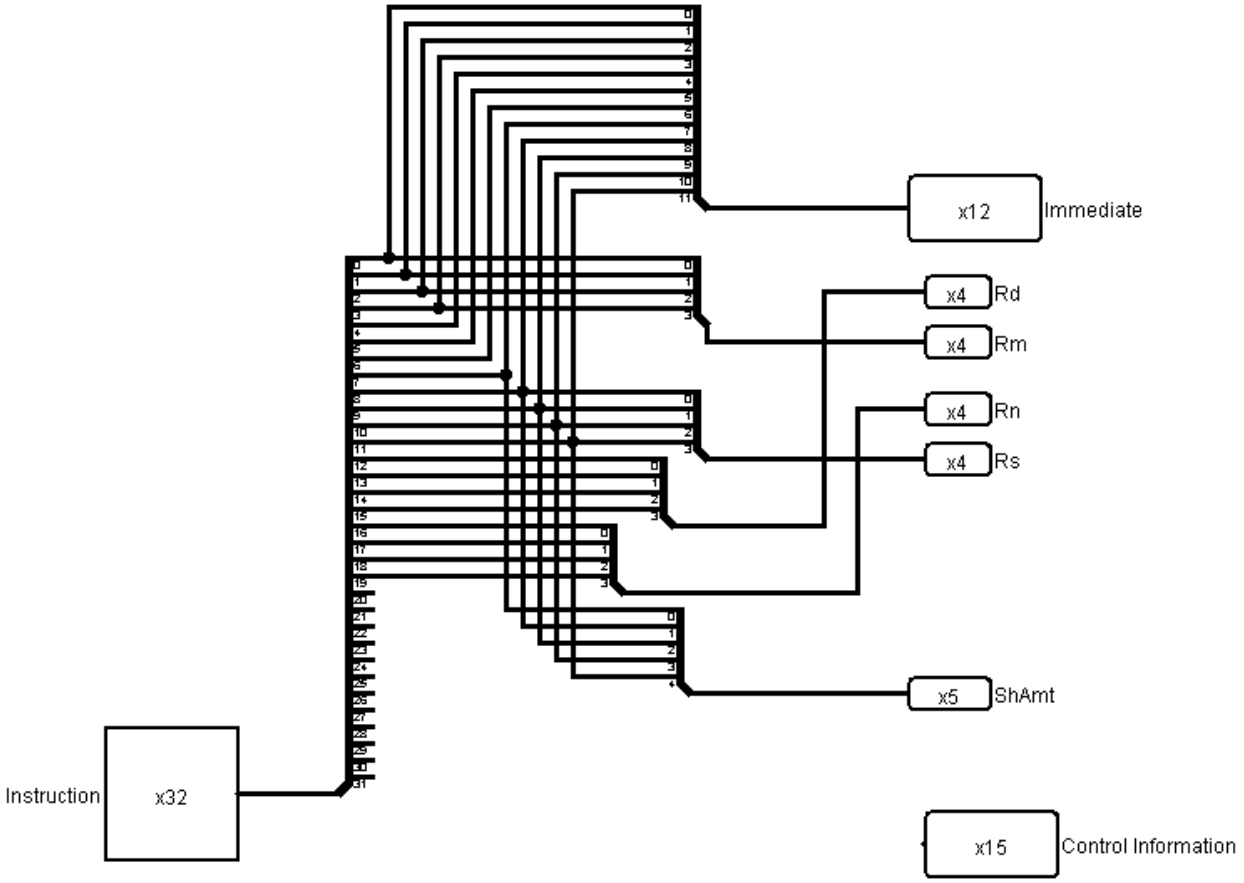


Figure 36: Decoder with added `Immediate` and `ShAmt` values

Finally, the control information needs to be forwarded. For now, all the control information is just grouped together and sent to a single output port. The control information is 15-bits of information, and includes the `CondCode`, `OpType`, `OpCode`, and `ShiftType`. This final iteration of the decoder is presented in the following figure, which represents the final implementation of the Decoder.

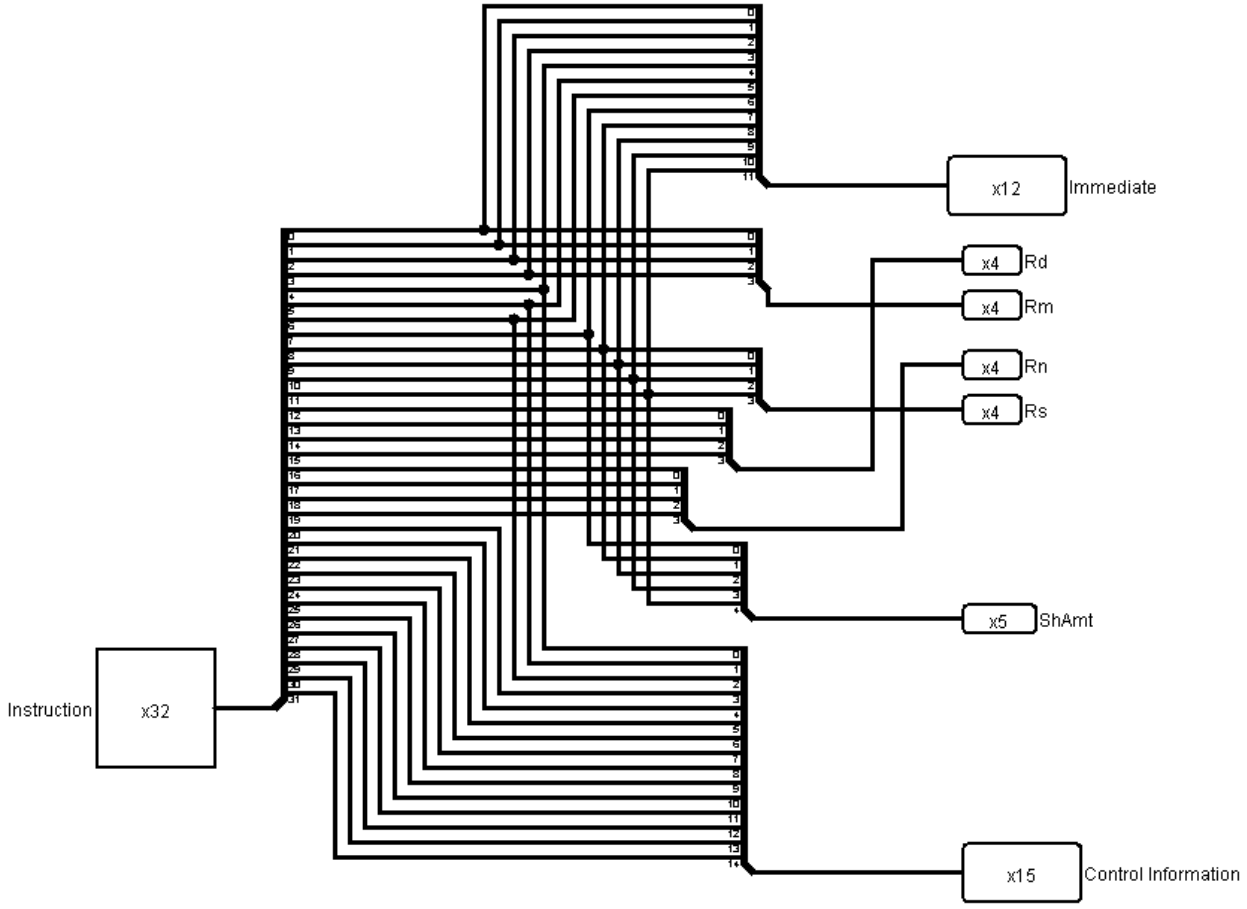


Figure 37: Final Decoder implementation

Chapter 6.2 Machine Code Instruction Formats

This section will present the machine code instruction formats for the operations that were presented in Chapters 4 and 5. These machine code instruction formats can be found in Appendix 3 and are documented in a separate spreadsheet. The format used to present these instructions is encoded as follows:

1 Any column in red should be entered exactly as they appear here. They will be explained later in the text. However, for producing machine code for the current version of the MSCPU, they should all be treated as constants.

2 Fields in blue are control information that are used to define the specific machine code type. These should also be treated as constants for the transaction type.

3 Fields in green are fields of data that must be specified for the machine code type. Some of these fields will contain control information (such as the OpCode for Register and

Immediate Instruction), others will contain number data (the `Immediate` and `ShAmt` values), and finally some will be register numbers.

These formats are broken down into 3 categories: shift and `mov` operations, data processing operations, and load/store operations.

Chapter 6.2.1 Operand2 definition

The Operand2 is often the third operand (or second input operand) in immediate and register instructions and second operand (or the only input operand) in register `MOV` instructions. Therefore, to understand the machine code format of any of the subsequent statements, first the Operand2 must be understood. The definition of Operand2 will be covered using the `MOV` operation.

The Operand2 occupies the least significant 12 bits of the instruction (bits 0...11) for Immediate and Register instructions. There will be 3 formats for the Operand2. There is one format for the Operand2 for Immediate instructions and two formats for the Register instructions.

For an immediate instruction (OpType = "001"), there is one type of Operand2 that is an 8-bit immediate value with a 4-bit amount to rotate the immediate. This text will call this an *Immediate Format Operand2*. The 12-bit format is shown in Figure 38.

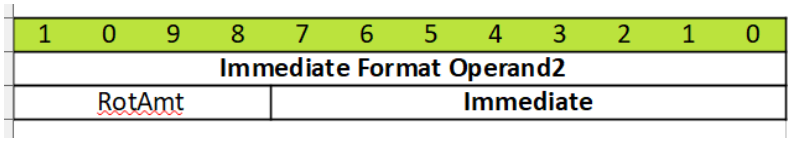


Figure 38: Immediate Format Operand2

If the instruction is a register instruction (OpType="000"), there are two formats for the Operand2. Both of these will use the `Rm` register, and allow the `ShiftType` to be specified. They differ in how they get value of the amount to shift. The *Register Format Operand2 with ShAmt* is specified by a 0 in bit 4 (the fifth bit) of the instruction and contains a 5-bit shift amount value in the `Rm` register. It is shown in Figure 39.

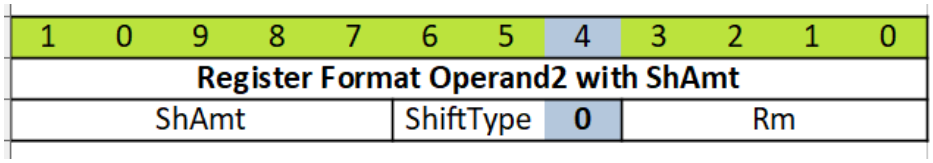


Figure 39: Register Format Operand2 with `ShAmt`

The *Register Format Operand2 with Register* is specified by a 1 in bit 4 (the fifth bit) of the instruction and uses the `Rs` register to specify how much to shift the value in `Rm`. It is shown in Figure 40.

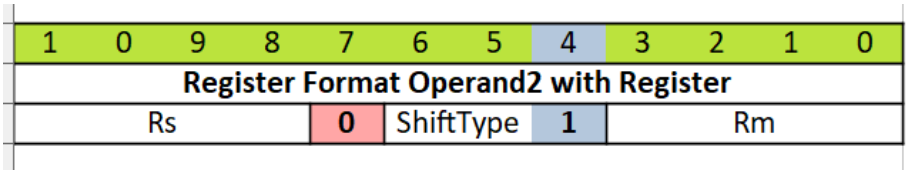


Figure 40: Register Format Operand2 with Register Amount

Note that the ShiftType is needed for these two instructions, and so a table of ShiftType values is given in Table 8. Note that to determine the type of shift, only two bits are needed. The third bit determines if the value to shift is `ShAmt` or a register.

OpType: 000 Special , OpCode: 1110 – Shift Operations				
OpCode	**ShiftType**	**Meaning**	**Inst Type**	
1101	000	MOV	R	bits 4-11 bits also zero, otherwise converted to Shift inst.
1101	00[0/1]	LSL	Shift	0 is ShAmt, 1 is Register
1101	01[0/1]	LSR	Shift	0 is ShAmt, 1 is Register
1101	10[0/1]	ASR	Shift	0 is ShAmt, 1 is Register
1101	11[0/1]	ROR	Shift	0 is ShAmt, 1 is Register
1101	110	RRX	Shift	Shift using R only, but Bit 4 is 0: Bits 7-11 are 0

Table -8: Shift Operations

The use of the Operand2 value is illustrated in the next section using the `MOV` instruction.

Chapter 6.2.2 Operand2 with MOV instruction

The `MOV` instruction is presented first because once the immediate operation is understood, all of the other instructions can be understood as modifications of the `MOV` instruction[30].

The `MOV` instruction is defined by an `OpType = 00[01]` and an `OpCode = 1101`.

The `MOV` instruction has three formats; the first format does not use a Shift operation, and the `Rm` value is passed on unchanged, and is effectively a 3-address instruction[31]; the second passes on a rotated `Immediate` value; and the third is a register operation. The second and third type differ by the least significant bit of the `OpType` and by the type of the `Operand2` they are support.

[30]Note that at a hardware level the `MOV` instruction is different than other data operations in that it bypasses the ALU and returns the value from the barrel shifter directly to the Register Bank. This is why `Rn` is always zero for `MOV` operations, and why it has its own format. Otherwise, it is no different than any other Register or Immediate Instruction.

[31]The 3-address instruction here will only be for a register value. As will be covered in the next section, the 3-address register `MOV` will have meaning for shift operations. The 3-address immediate format is not really interesting by itself.

The first type of `MOV` operation is simply a 3-address instruction format of the `MOV`. An example is the following:

```
MOV Rd, Rs, Rm
```

An example of this instruction is:

```
MOV r1, r2
```

In this instruction, the `Operand2` contains the `Rm` value, but the rest of the bits in `Operand2` (bits 4-11) are zero. As will be seen in the next chapter, this correspond to a `LSL` with a shift of 0 bits, which is a meaningless `LSL` operation.

The machine code format of this `MOV` instruction is:

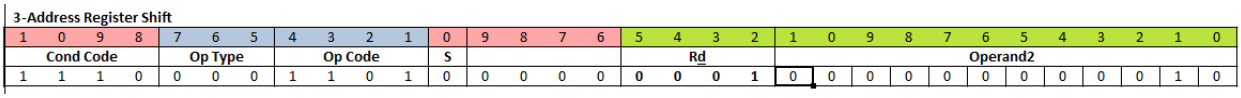


Figure 41: Machine code format for a 3-address `MOV` instruction

This 32-bit value is hard to read, so it is generally expressed in hexadecimal as 0xe1a01002.

The second type of `MOV` is an immediate `MOV`. Remember from Chapter 5.4.2, the immediate `MOV` can have the format:

```
MOV Rd, Immediate, ShAmt (even values only)
```

An example of this format would be:

```
MOV r1, #3, 4
```

The machine code format of the immediate `MOV` instruction is:

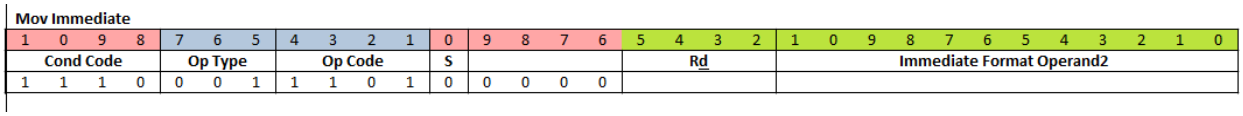


Figure 42: Machine code format for an immediate `MOV` instruction

Since this instruction uses the Immediate Format `Operand2`, the `ShAmt` (amount of the rotation) is `0b0010`[32] in bits 8..11 of the instruction, and the `Immediate` value is `0b00000011` in bits 0...7. Filling in the green boxes in Figure 43 below with `Rd` = `r1` (`0b0001`):

[32]Note that the `ShAmt` looks like it is 2, but remember from Chapter 5 that the rotation amount is an even number, but covers the full range of 32 bits. There is no room for 5 bits, so the least significant bit is dropped, which is why the `ShAmt` in an Operand2 is only even numbers.

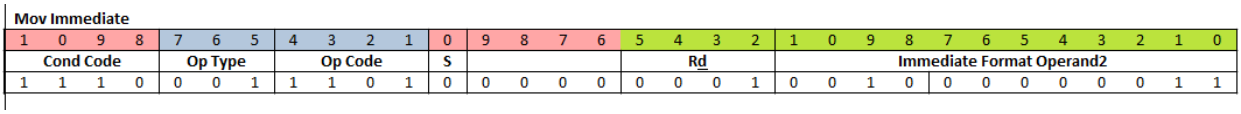


Figure 43: Binary value for instruction `MOV r1, #3, 4`

This 32-bit value expressed in hexadecimal is 0xe3a01203.

The third format of the `MOV` instruction is the register `MOV` instruction. Remember from Chapter 5.4.3 the `MOV` has a register format of:

```
MOV r1, Operand2
```

This corresponds to a machine code format shown in Figure 44:

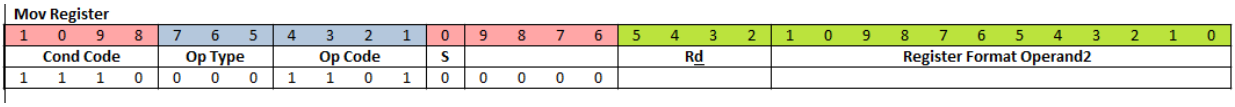


Figure 44: Machine code format for a Register `MOV` Instruction

There are now two possible formats for the register `MOV` instruction. The first is with a Register Format `Operand2` with `ShAmt`. An example is shown in the instruction below:

```
MOV r1, r2, lsl #3[33]
```

Filling in the instruction with the proper values, `Rd = r1 = 0b0001, Rm = r2 = 0b0010, ShAmt = 3 = 0b00011,` and `ShiftType = LSL = 0b00`, this `MOV` instruction would have a binary value given in Figure 45.

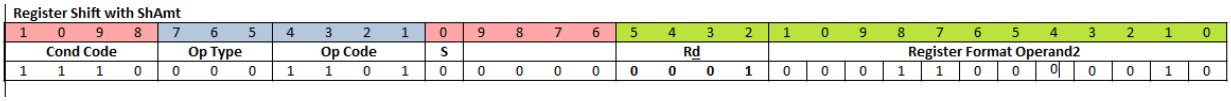


Figure 45: Machine code format for a Register `MOV` Instruction with `ShAmt`

The hexadecimal value of this instruction is 0xe1a01182.

The final possibility for a `MOV` instruction is a Register Format `Operand2` with Register. An example is shown in the instruction below:

```
MOV r1, r2, ASR r3
```

Filling in the instruction with the proper values, `Rd = r1 = 0b0001, Rm = r2 = 0b0010, Rs = r3 = 0b0011,` and `ShiftType = ASR = 0b10`, this `MOV` instruction would have a binary value given in Figure 46.

[33]Note that this `ShAmt` is 5 bits, so odd values are allowed.

Register Shift with Register Value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code					S					Rd				Rs					ShiftType**			Rm			
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	1	0	0	1	0

Figure 46: Machine code format for a Register `MOV` Instruction with Register

The hexadecimal value of this instruction is 0xe1a01352.

To test our logic and see if it is correct, a program is written with these three instructions in it, and then compiled to an object file. The program, which was written in a file called machine.s, is the following:

```
.text
.global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    MOV r1, r3
    MOV r1, #3, 4
    MOV r1, r2, lsl #3
    MOV r1, r2, asr r3


    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr
.data
```

8 Program to check machine code instructions

To compile this program, the command "`gcc machine.s -c -o machine.o`" was run, and the command `objdump` was run on the resulting object file using the command "`objdump machine.o -d`". This produced the output shown on the following screen shot.

```
pi@devpi-0:~/temp $ objdump Machine.o -d

Machine.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e24dd004        sub     sp, sp, #4
   4:   e58de000        str     lr, [sp]
   8:   e1a01003        mov     r1, r3
   c:   e3a01203        mov     r1, #805306368  ; 0x30000000
  10:   e1a01182        lsl     r1, r2, #3
  14:   e1a01352        asr     r1, r2, r3
  18:   e59de000        ldr     lr, [sp]
  1c:   e28dd004        add     sp, sp, #4
  20:   e1a0f00e        mov     pc, lr
pi@devpi-0:~/temp $
```

Figure 47: Output from dumping an object file

The second column in the output from running the `objdump` command gives the hexadecimal value of the object code the command produced. Notice that the output corresponds to the calculated values.

Chapter 6.2.3 Shift operations

There is more to see in the `objdump` output in Figure 47 than just the hex values of the instructions. Note that the register `MOV` instructions were printed out as the shift instructions `LSL` and `ASR`, not the `MOV` instructions we originally inputted. The reason for this is that every shift instruction has a corresponding register `MOV` instruction.

The register instructions:

```
MOV r1, r2, LSL #3

MOV r1, r2, ASR r3
```

are the equivalent in machine code as the following shift equations:

```
LSL r1, r2, #3

ASR r1, r2, r3
```

Thus, there is no need to cover the machine code format for the shift instructions since it is only necessary to covert the shift instructions into register `MOV` instructions, for which the translation to machine code has already been covered.

Chapter 6.2.4 Data operation Instruction Formats

Data operations are the logical and arithmetic operations executed in the ALU. The operations are given in Table 9.

OpType: 00[0/1] – Data Operation			
OpCode	**Meaning**	**Inst Type**	**Action**
0000	AND	I/R	operand 1 AND operand 2
0001	EOR	I/R	operand 1 EOR operand 2
0010	Subtract	I/R	operand 1 - operand 2
0011	RSB	I/R	operand 2 - operand 1
0100	Add	I/R	operand 1 + operand 2
1100	ORR	I/R	operand 1 OR operand 2
1110	MOV	I	For I, operand1 ← operand 2, for R see table below
1110	Shift	R only	See table below

Table -9: Shift Operations

There are two formats for the data operations, an Immediate format, specified by an `OpType = "001"` and a Register format specified by an `OpType = "000"`. These two formats are shown in Figures 48 and 49 below.

Immediate

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Cond Code	Op Type	Op Code	S	Rn	Rd	Immediate format Operand2
1 1 1 0	0 0 1		0			

Figure 48: Immediate Instruction

Register

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Cond Code	Op Type	Op Code	S	Rn	Rd	Register Format Operand2
1 1 1 0	0 0 0		0			

Figure 49: Register Instruction

There are two difference between the `MOV` operation and the other data operations. The first difference is that the `MOV` instruction does not have the `Rn` register that is needed for all the other data operations. The other difference is that an `OpCode` needs to be specified for the Immediate and Register Instructions. These `OpCode` values are specified in the first column in Table 9.

To see how to apply these formats to an instruction, consider the two instructions below. The first is a register ADD instruction.

ADD r1, r2, r3, LSL #5

Using Table 9, the ADD operator has an OpCode = "0b0100", the Operand2 uses a ShiftAmt ="0b00101", from Table 8 the ShiftType = LSL = "0b00", and registers Rd = r1 = "0b0001", Rn = r2, "0b0010", and Rm = r3 = "0b0011". Using these values in the Immediate instruction, the 32-bit representation of this instruction is shown in Figure 50.

ADD r1, r2, r3,

1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code				S	Rn				Rd				Immediate format Operand2											
1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	1	1

Figure 50: Machine Code for `ADD r1, r2, r3, LSL #5`

This 32-bit value in hexadecimal is 0xe0821283.

The next instruction is an immediate `ORR` instruction.

```
ORR r1, r2, #260  // Note #260 = 0x21, 30, or ORR r1, r2, #0x21, 30
```

Using Table 9, the `ORR` operator has an `OpCode = "0b1100"`, the `Operand2` uses an `Immediate value = "0x21"`, the `ShAmt = 30 = "0xffff"`, the `ShiftType` is an implied `RRX`, and registers `Rd = r1 = "0b0001"`, `Rn = r2, "0b0001"`. Using these values in the Immediate instruction, the 32-bit representation of this instruction is shown in Figure 51.

ORR r1, r2, #2

1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code				S	Rn				Rd				Immediate format Operand2											
1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1

Figure 51: Machine Code for `ORR r1, r2, #260`

This 32-bit value in hexadecimal of 0xe3821f21. Assembling these instructions yields these machine code values.

Chapter 6.2.5 Multiply operation

The Multiply (`MUL`) register instruction is a data operation. The register instruction is indicated by an `OpType = "000"`, an `OpCode = "0000"`, and a `multCd = "1001"`. Operations are of the format:

```
MUL Rd, Rm, Rs
```

Note that unlike the other data ops commands, the multiply does not have an immediate format, nor can it use an `Operand2` value.

The lack of an immediate format is likely because the ability to multiply by a constant can be implemented as a series of shift and add operations, and so it was left out of the original ARM architecture.

The inability to use an `Operand2` value is because the multiply unit is run inline with the barrel shifter, so the output of the barrel shifter cannot be used in the `MUL` operation.

The machine code format of the register instruction is shown below. Note, the register `Rd` has been moved, and `Rs` is used in the place of `Rn`.

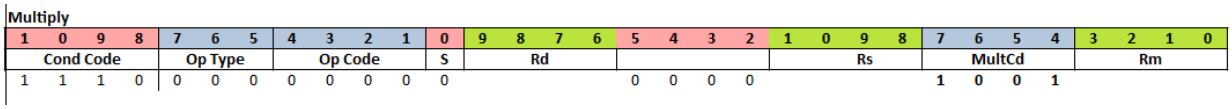


Figure 52: Machine Code for `MUL` operation

An example of this type of instruction is as follows:

```
MUL r1, r2, r3
```

This instruction is represented in the following 32-bit format.

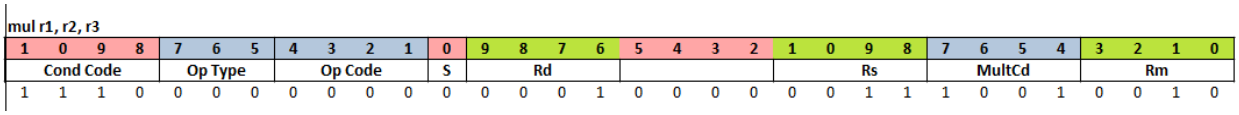


Figure 53: Machine Code `MUL r1, r2, r3`

Once again, this value is written in hexadecimal as `0xe0010392`. This can be checked using the `objdump` command as in the last chapter, and you will find that the `objdump` produces the same output.

Chapter 6.2.6 Load and Store Instructions

The last types of instructions that are covered in this chapter are the load and store instructions[34]. There are two types of load and store instructions, one that calculates the memory address using two register values, and one that calculates the memory address using a register and a 12-bit immediate value. The load and store operations have an `OpType = "010"` for the register format, and an `OpType = "011"` for the immediate format. The format of these two instructions is as follows:

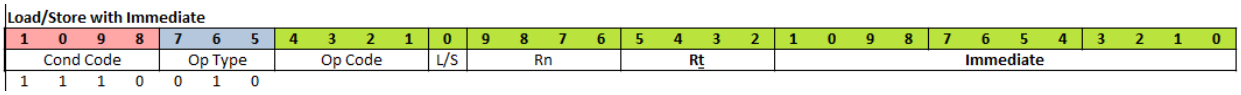


Figure 54: Machine Code format for Immediate operand

[34]Note that for a word or unsigned byte, the OpType for a `LDR`, `LSRB`, `STR`, and `STRB` have values `10[0/1]`, as would be expected. However, as the load and store of a half word, signed byte, and double word are extensions to ARM, they do not follow the expected pattern. The `LDRH`, `LDRSB`, `LDRD`, `STRH`, `STRSB`, and `STRD` all have OpTypes of `000`, just like an immediate instruction. See the problems at the end of the chapter for more information.

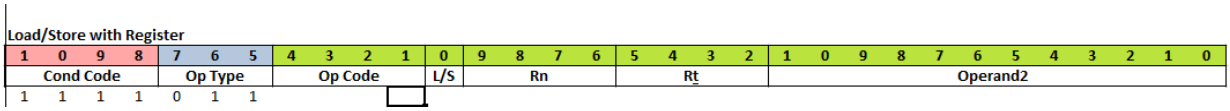


Figure 55: Machine Code format for Register operand

The Load and Store immediate format does not support the rotation immediate of the `Operand2`; instead, the format uses the extra bits to create a larger immediate value, creating a value that does not have the range of the `Operand2` immediate, but can represent all values from 0…4095. This is an important feature because all data items[35] must have an address in the object file, and so the greater accuracy of the immediate is more important than its range. The reader should note that the 12-bit number is a positive whole number. Whether this number is to be added or subtracted from the register is controlled by the `OpCode`, as will be seen later.

The Load and Store register format does not support the Register `Operand2` format, but instead supports a format called a Scaled Register Format. The Scaled Register Input is just a Register `Operand2` that only allows rotation using a `ShAmt`; the ability to shift using a register value is removed.

This format also includes one new field in this instruction, the L/S field. The L/S field specifies if this this a load (`1`) or store (`0`).

Finally, values for the `OpCode` have changed. These changes support the use of Auto incrementing when calculating addresses and are summarized in the table 10[36].

OpType: 01[0/1] : Store and Load Operations
L/S (Load/Store): 0=Load, 1=Store
Immediate values are 12 bits (bits 0-11)

OpCode	L/S	Meaning	Inst Type	Action
1000	0/1	Subtract	I	Immediate negative value only, 0 is Immediate, 1 is R
1100	0/1	Add	R/I	0 is Immediate, 1 is Register
0100	0/1	Post-Indexed	R/I	0 is Immediate, 1 is Register
1101	0/1	Pre-Indexed	R/I	0 is Immediate, 1 is Register

Table -10: Load/Store Operation Codes

Examples of load and store instructions are the following.

```
ldr r1, [r2, #12]

str r1, [r2, r3]
```

[35]This includes items that are stored in the data section. An object file local memory location, called a veneer, will contain the actual address of the data item that is requested.
[36]The individual bits have meaning, but are used inconsistently as the bit meanings have changed in ARM extensions. Since only these 4 operations are allowed for the instructions in this text, use the bits as shown here.

Once again, the correct values can be filled in to the instruction templates above, as shown below.

ldr r1, [r2, #12]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code					L/S	Rn					Rt				Immediate										
1	1	1	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0

Figure 56: Machine Code for `LDR r1, [r2, #12]`

STR r1, [r2, r3, lsl #2]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code					L/S	Rn					Rt				Operand2										
1	1	1	0	0	1	1	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	1

Figure 57: Machine Code `STR r1, [r2, r3, lsl #2]`

When the values from these templates are converted to hexadecimal, the results for the 2 instructions above are 0xe592100c and 0xe7921103. Note that if these instructions are compiled and examined with `objdump`, the results show that the correct calculations have been done.

Chapter 6.3 Decoding Machine Code

For the computer to execute a machine code instruction, it must be able to decode it to set the proper control wires. Therefore, it must be able to understand the meaning of the machine code instructions.

This is also a good skill for any low-level programmer to have. There are many reasons for this. Computer security professionals often need to re-engineer software to find malware, and this requires that they read machine code. Being able to optimize code can require that parts of the machine code be read. These skills will become more valuable as applications such as System on Chip (SoC) or Internet of Things (IoT) become more common. As computers become more complex, and include hardware subsystems such as Single Instruction, Multiple Data (SIMD) or Vector processing, knowing how to properly use these features will be important to even programmers who use only HLL. While there will be compilers that will optimize to these features, a programmer can often make suggestions to the compiler that will allow it to do an even better job at optimization. In order to properly understand how to optimize programs, one must have knowledge of how hardware works; one of the first things to understand in Computer Architecture is how to decode an instruction.

This section explains how to decode an instruction, taking machine code instructions and translating them back to assembly language instructions. It will be done in three steps. The first step is to determine the instruction formats. There are many different instruction formats, and all have different meanings for the fields in the instruction. This first step will derive the instruction format so the fields in the instruction can be read.

The second step is to determine which type operation is being used. For a data operation instruction, it could be an `ADD`, `SB`, `EOR`, etc. For a load/store operation, it could be `ADD`, `SUB`, or pre/post index. For shifts, it could `LSL`, `LSR`, `ASR`, etc. So once the instruction type is determined, the correct table is accessed to choose the correct operation.

Finally, the data fields in the instruction are determined. This means registers and immediate values are retrieved from the instruction.

These steps are all rote or mechanical, and the table lookup and translations of data fields are not explained. However, step 1, determining the instruction format, is somewhat complex and will be explained in the next section. Translation of a machine instruction to assembly is then shown.

Chapter 6.3.1 Determining instruction format

There is a lot that goes into determining the instruction format. First, the `OpType` has to be examined, and based on the `OpType`, different `OpCode` values can specify different types. It can be intimidating to anyone. So, to start, this textbook presents the following flowchart to help the reader determine the instruction type.

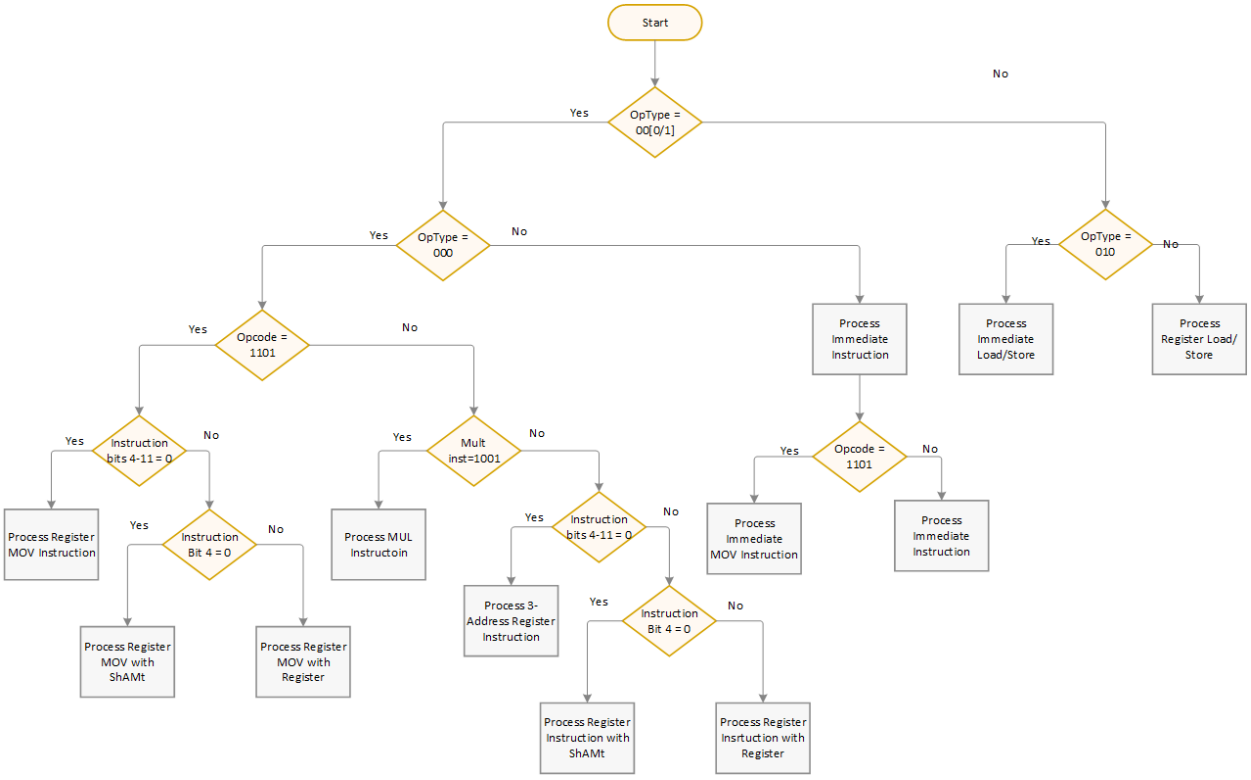
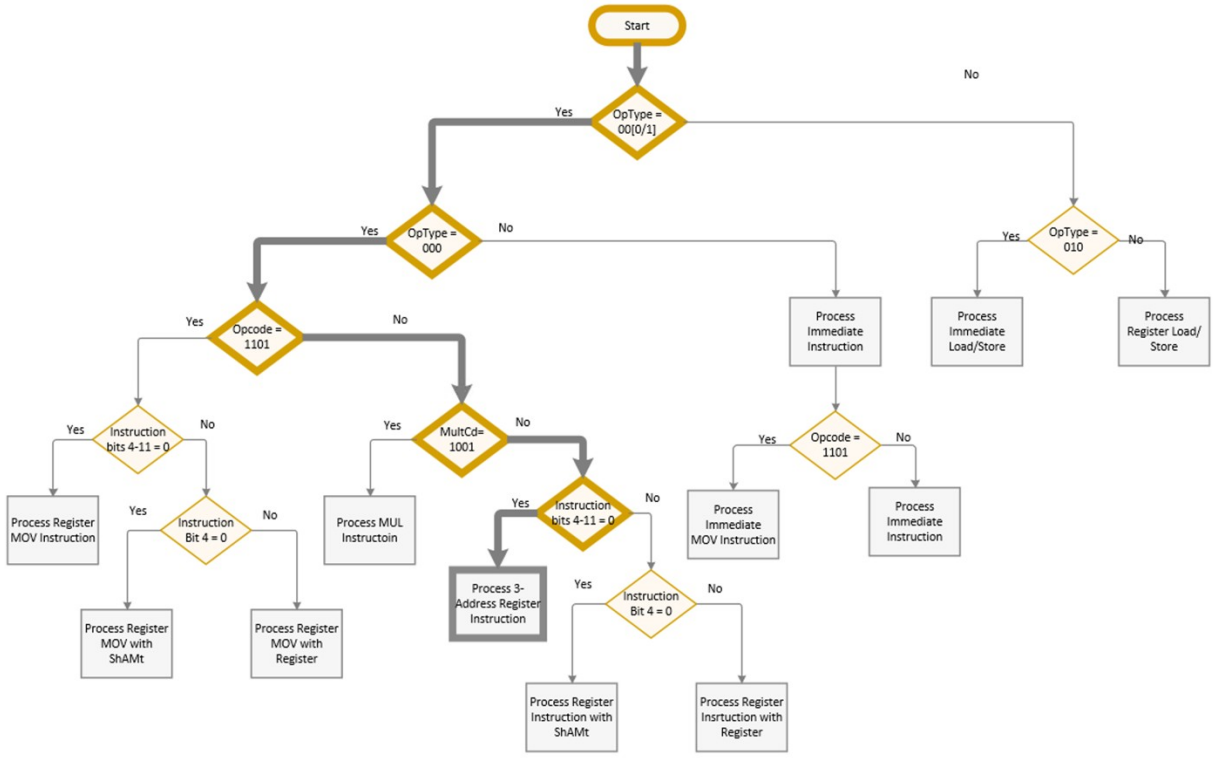


Figure 58: Flow chart to find instruction format

The textbook will now present two examples that utilize the flowchart to determine the instruction format. In both examples, the author will walk the reader through the process of navigating the flowchart with text and with an annotated flowchart.

To use this chart, start with a machine code instruction, such as `0xe0821003`. First, convert the instruction to binary, `0b1110 0000 1000 0010 0001 0000 0000 0011`. Next, break out the `OpType` field, which is `0b000`. This indicates it is a data operation, so take the left leg of the flow chart, and since the lowest order bit is 0 it is a register and check the `OpCode`. The `OpCode` is not `0b1101`, so take the right leg of the flow chart. The `MultCd` is not `1001`, so the instruction must be a register instruction. Since bits 4-11 are 0, the `operand2` is simply `Rm` and a 3-address register instruction will be processed.



The register instruction is the following format:

1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code				S	Rn				Rd				Register Format Operand2												
1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1

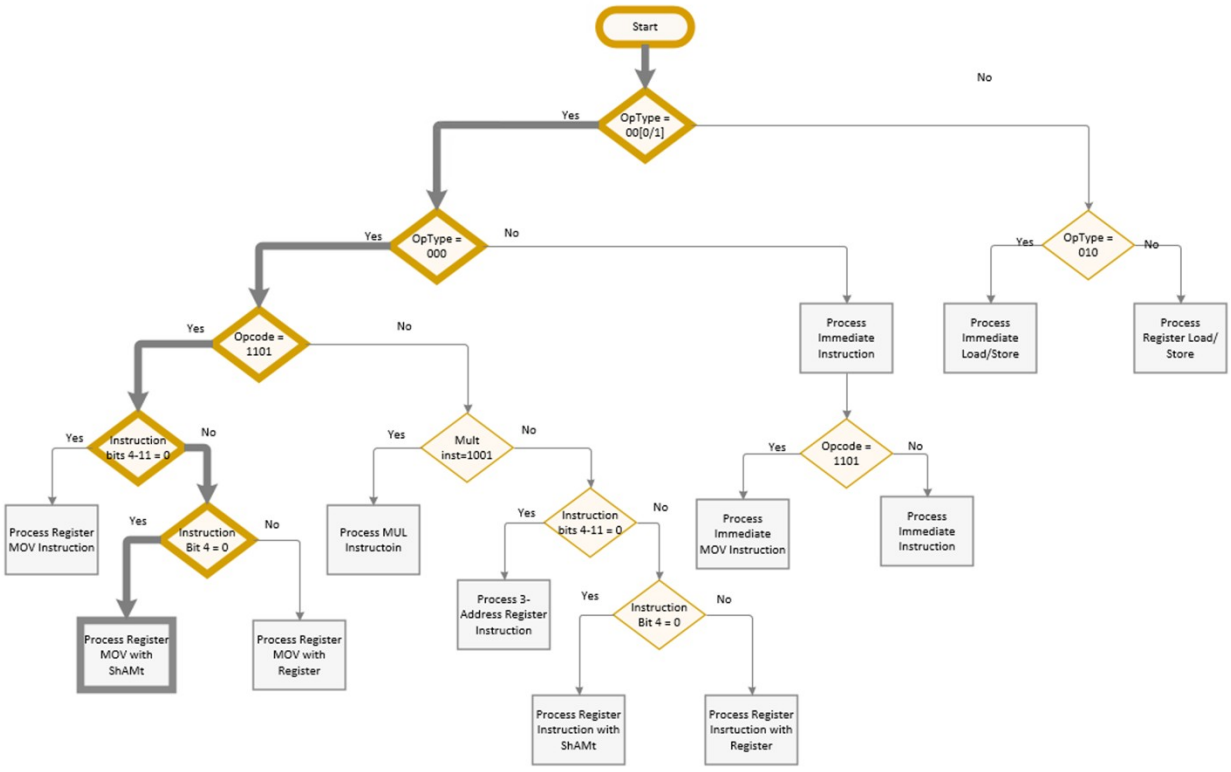
Figure 59: Machine Code format with bits filled in

Filling in the bits, an `OpCode` of `0b0100` is an add instruction, and the registers are `Rn = 2`, `Rd = 1`, and `Rm = 3`. Thus, this corresponds to the assembly instruction:

```
ADD r1, r2, r3
```

To check this, compile this assembly code statement and run `objdump` to see if the original machine code is returned.

Another example is `0xe1a01182`. First, convert the instruction to binary, `0b1110 0001 1010 0000 0001 0001 1000 0010`. Next break out the `OpType` field, which is `0b000`. This indicates it is a data operation, so take the left leg of the flow chart. The least significant bit is a 0, so take the left leg, and check the `OpCode`. The `OpCode` is `0b1101`, so take the left leg of the flow chart. The instruction bits `4-11` are not `0`, so it is a `MOV` operation with an `Operand2` value. Bit `4` is a `0`, so it is a shift with a `ShAmt` value.



The register instruction is the following format:

Register Shift																													
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Cond Code				Op Type			Op Code				S					Rd				Register Format Operand2										
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	**0**	**0**	**0**	**1**	0	0	0	1	1	0	0	0	0	1	1	0

Figure 60: Machine Code format with bits filled in

A `ShiftType` of `0b000` is an LSL instruction, and the registers are `Rm = 2`, `Rd = 1`, and the `ShAmt = 2`. Thus, this corresponds to the assembly instruction:

```
MOV r1, r2, lsl #3
```

Since all register moves have equivalent shift operations, this is also the following:

```
LSL r1, r2, #3
```

Most readers will realize that the computer is not following a flow chart to determine the format for the operation, as a flow chart would require a synchronous circuit. Instead, the computer will simply determine the bits in the original instruction it needs to specify the instruction and compare those bits with the original instruction. Since all the compare operations for the formats can be run in parallel, this is a very fast way to determine the instruction format in the processor. This is shown in the following diagram. For the register `ADD` operation, this bit-mask is `0x00800000`, and for the immediate shift `LSL` it is `0x01a00010`.

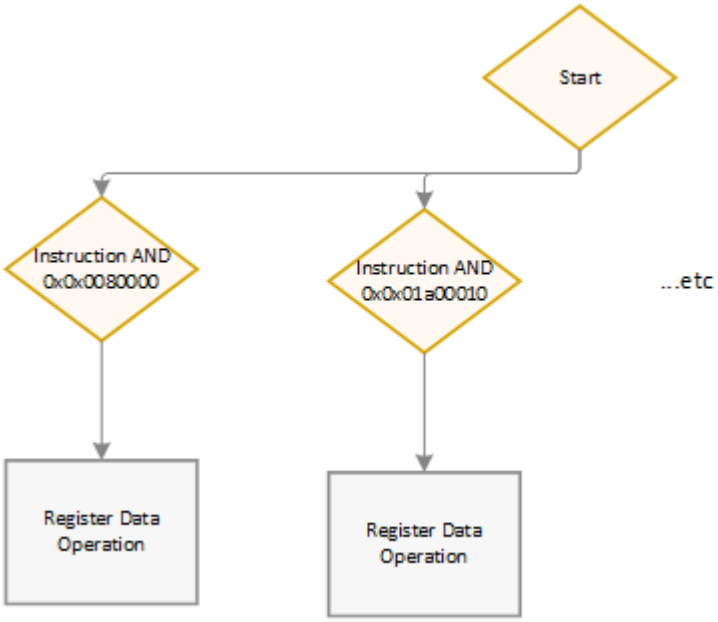


Figure 61: Computer selection of instruction format

Note that while using compare is fast for a computer, it is more confusing for the reader who is likely a person, and the flow chart is probably an easier way to decode the instruction for most readers.

Chapter 6.4 Conclusion

Computers only understand electrical circuits, and those circuits use voltages to represent 0 and 1. Therefore the only way a program can be interpreted by a CPU is if the program is translated into a binary coding that the computer understands. This chapter explained how that binary coding is produced. Through the abstract circuit diagrams, it also gave a hint as to how the CPU uses those bits to execute an instruction.

Chapter 6.5 Problems

1 Specify the bit mask that would be used to convert all 8 of the instruction formats for this chapter. In Chapter 6.3 it said the bit mask to select the register operation format is `0x00800000` and to select the immediate shift operation format is `0x01a00010`. What are the bit masks for the 6 operation formats in this chapter?

2 Translate the following `MOV` instructions into `LSL` instructions or `LSL` instructions to `MOV` instructions.

2.1 `LSL r1, r2, #0`

2.2 `MOV r1, r2, LSL r0`

3 The following shift instruction `LSL r1, #8, 4` is invalid, but it assembles to the machine code instruction `MOV r1, #8`. Why is the original instruction invalid? Try to see if you can figure why the programmer got this wrong? What does this tell you about the `operand2` value?

4 For the following assembly instructions translate them into machine code, with the answers given in hexadecimal format.

5 For the following machine code instructions, translate them into their assembly code.

6 Give the 12-bit values for the following `Operand2` immediate values.

7 Explain why the Multiply Code (bits 4-7 as 1001) can be used to specify a multiply instruction and will never be used in any other register instruction. (Note: you only need to consider the two 1's in the value).

8 Modify the decoder circuit for…

In this chapter you will learn:

1 how a CPU uses the PC register to control the execution flow of statements in a program.

2 the convention used as a mechanism to control the execution of a function, and the return from a function.

3 the purpose of the Link Register (lr)

4 the purpose of a program stack to store information that otherwise might be lost during the execution of the function,

5 how to use the linkage editor (or linker) to create an executable program file.

Chapter 7 Program Control Flow and Functions

At this point most programming textbooks cover procedural programming (branching and loops) before covering functions. This is because in HLLs, branching and looping concern programming logic (how to implement algorithms), and functions are more about abstraction and code organization. This text, however, assumes that this is not the first class in programming for readers, and programming logic and functional abstractions are familiar to the reader. Therefore, looping, branching, and functions are not covered to present logic or abstraction, but to explain program control in a CPU when a program is executing. A secondary goal of this textbook is to present procedural logic and functions in a standard (or canonical) format to simplify how to understand these structures.

In addition, it is the opinion of the author that it is easier to explain program control and organization by covering functions before procedural constructs. There are two reasons for this belief. First, the program control for functions is more straight forward, as functional abstraction can be implemented with simple or no logic. It does not require structures inside of structures, such as looping and if blocks containing other looping and if blocks that are required to implement an algorithm. A properly implemented assembly function should have a single point where it is entered and a single point where it returns. The calling function should have the address of the function and the return point. Any other use of a function will only result in confusion and a strong possibility of an incorrect and probably erroneous[37] program.

The second reason for covering functions first is that the canonical form of functions should always be followed unless there is a very excellent and compelling reason to violate it (for

[37]The term *erroneous*, as used in this textbook, is very specific. A program can be erroneous but still produce the correct result. In that case, the program can be said to produce the correct answer, but the program is not correct. Programs that are erroneous but produce a correct result have misapplied proper principals of programs and misused structures in programs. While they might work, they are fragile and hard to understand. Programmers should be careful to understand the difference between a program that has an error and an erroneous program. This textbook will try to point out correct structuring techniques for programs and strongly encourages correct programming, not just making programs that give correct answers.

instance, operating systems using non-reentrant functions because of time or memory constraints). The reasons for this conical form are easier to explain than the forms for looping and branching statements.

This chapter will proceed as follows. The first section will introduce the Program Counter (`pc`) register. The section will then implement a simple `Increment` function that is called from a `main` function. The use of the `pc` to control the flow of the program will be shown, and the process of calling the `Increment` function from a `main` function and then returning at the end to the `main` function will be illustrated. This is meant to show how program control is accomplished in a program using the `pc`.

The second section of this chapter will explain the issues with the simple program in the first section. It will address these issues, and in the process define the program stack, and why and how to push and pop to the stack.

The final section will explain how to make references to a function available outside of the current file and using this information to create library assembly files that are collections of functions that are used in multiple programs. Makefiles for the programs using these library files are updated to allow the linker to use the functions when creating programs. The concept of Unix library files will be covered for files containing a large number of function objects.

Chapter 7.1 Program Control Flow

This section demonstrates the use of the `pc` register to control the flow of program logic. To start, a simple function called `increment` is created, and this function is called from the `main` function. In the `main` function a variable called `numToInc` is created, and the `increment` function is called using the Branch-and-Link (`bl`) operator. The variable `numToInc` is passed to the function in `r0`, where it is incremented by 1 and returned back in `r0`.

This program is then run in `gdb`, and the value in the `pc` and `lr` registers follow to show how the pc is used to control program flow in this program.

Chapter 7.1.1 main and increment functions

The program that will be used to illustrate program control flow is Program 9. The rest of this section will run this program in `gdb` and demonstrate how the `pc` and `ldr` registers are used in program control flow. To follow along with this discussion, enter this program into a file named incrementMain.s, compile and link the program, and begin to run the program, stopping it at the first line in the `main` method.

```
.global main
main:
    # Save return to os on stack
    SUB sp, sp, #4
    STR lr, [sp, #0]

    # Prompt for and read input
    LDR r0, =promptForNumber
    BL  printf
    LDR r0, =inputNumber
    LDR r1, =numToInc
    BL  scanf

# Increment numToInc by calling increment
    LDR r0, =numToInc
    LDR r0, [r0, #0]
    Bl increment

# Printing the answer
    MOV r1, r0
    LDR r0, =formatOutput
    BL printf

# Return to the OS
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr

.data
    promptForNumber: .asciz "Enter the number you want to increment: \
n"
    formatOutput:  .asciz "\nThe input + 1 is  %d\n"
    inputNumber: .asciz "%d"
    numToInc: .word 0
#end main
.text
#function increment
increment:
    ADD r0, r0, #1
    MOV pc, lr
#end increment
```

9 Increment function

When the program reaches the breakpoint in the `main`, the screen should appear as follows:

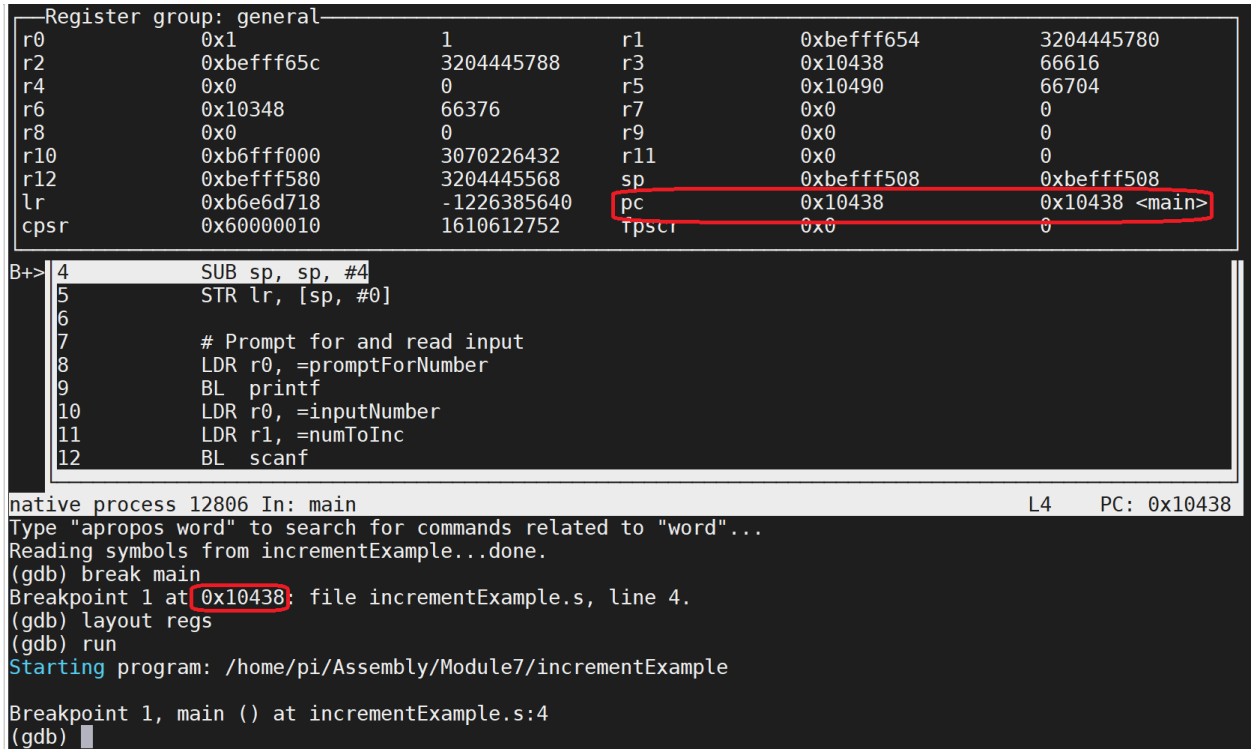


Figure 62: Start of the program to call the `increment` function

In this figure, the column to the left of the source program is the address in memory of that instruction. As this figure shows, the value in the `pc` register is the same as this address, and this shows that the `pc` register contains the current instruction that is being executed. By walking through the program using the `next` command, the next instruction is always 4 bytes away from the current instruction, and the `pc` is increased by 4. This is how the program runs sequential instructions in a program, by increasing the value of the `pc` register by 4 for each instruction.

Now set a break point at line 17, and continue running the program. Enter a value or 5 when the program pauses for input at the `scanf` statement. The program will stop at address `0x1045c`, the `bl` instruction that references the `increment` function. This is shown in the following screen shot.

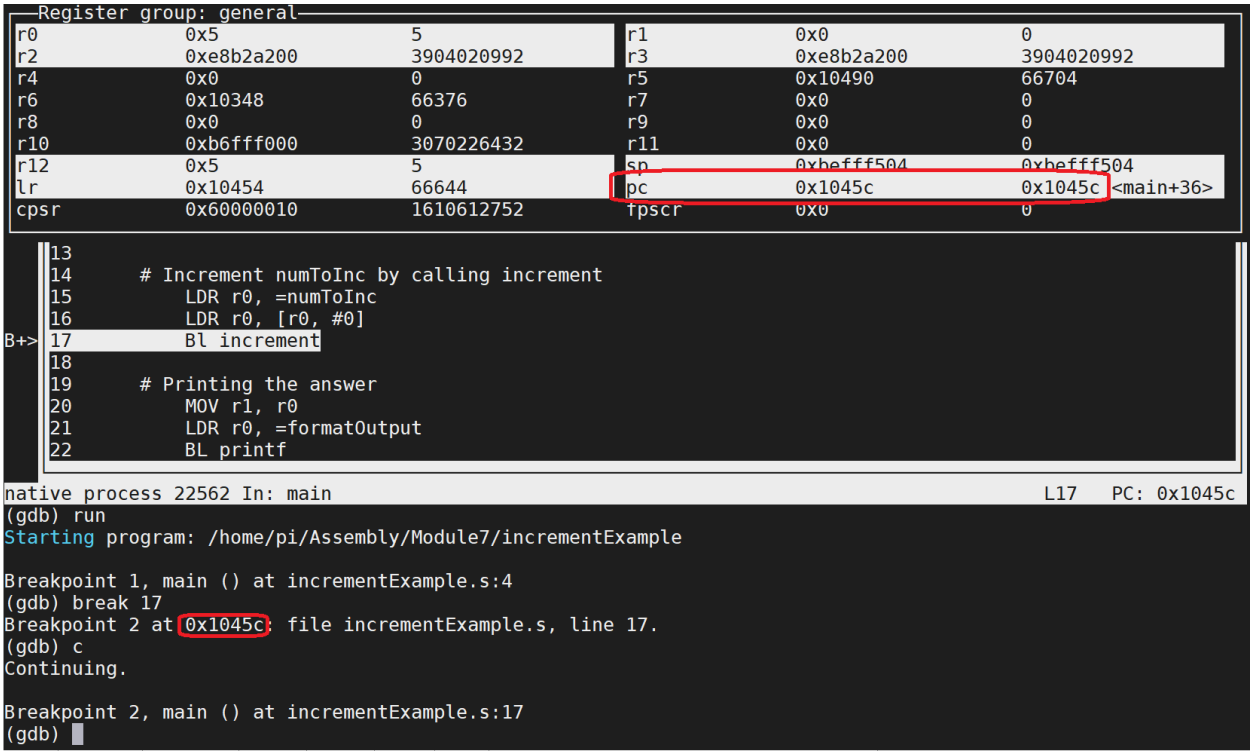


Figure 63: The branch and link command for the `increment` function

The branch instruction is used to cause the program to jump to an address other than the next sequential statement. This branch statement contains the value `0x10478`, which is the address of the start of the `increment` function.

For now, think of the `bl` instruction as a function call. When calling a function, it is expected that when the function completes, it will return to the instruction after call to the function. In this case, that is the "`MOV r1, r0`" statement at address `0x10460`. To allow the function to return to this address, the `lr` register is loaded with this address.

The *`step`* command (not the `next` command) in gdb steps into the method. Using `step`, step to the first instruction in `increment`. Notice the value in the `pc` is `0x10478`, and the value in `lr` is `0x10460`, as expected.

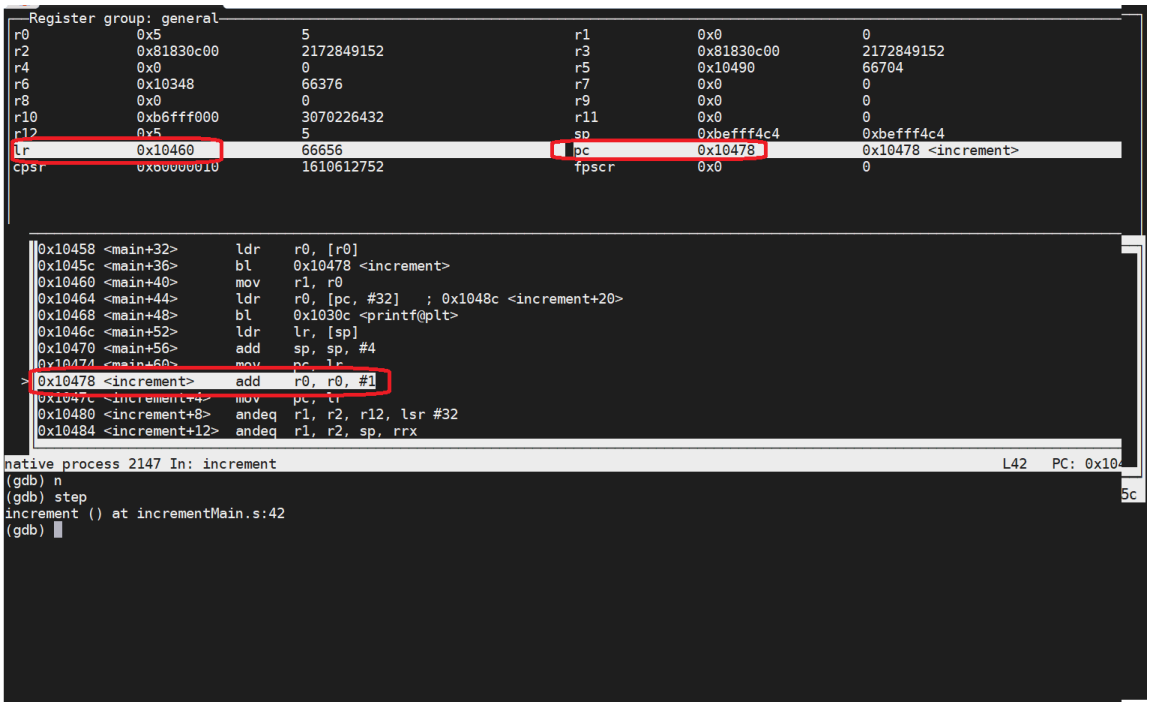


Figure 64: First statement when entering the `increment` function

The second instruction in the `increment` method is a "`MOV pc, lr`", which moves the value in the `lr` (the return point in `main`) into the `pc`, which causes the program to return to the `main`. The screen after running this instruction should look as follows:

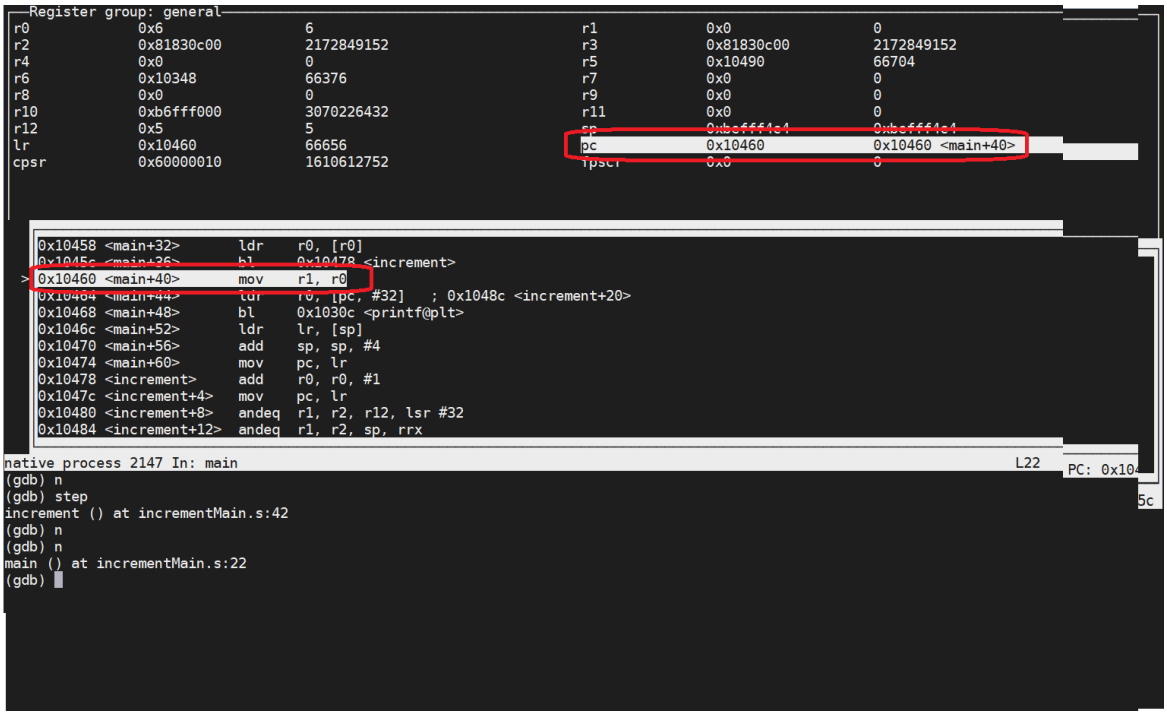


Figure 65: Return from the `increment` function

This shows the return from the function. This example is how the control of flow in a program is combined for a function using a combination of the `pc` and `lr` registers.

The important thing to remember about this section is that the `pc` controls program flow and is possibly the most important register in a computer.

Chapter 7.2 What is a program stack

This section will cover what is the program stack and how to use it with functions. It will first present a problem with the last program. That problem will be solved using a static variable; this is an old solution that is no longer used and the rationale will be discussed. The section will continue by defining a data structure called a stack and show why adding and removing memory from a stack is simple and fast. Finally, the program stack for the `lr` will be shown.

Chapter 7.2.5 Why the increment function is erroneous

The increment function works correctly, so the reader is probably wondering why I call this function erroneous. Remember that the definition of erroneous is not that the program produces an error[38], but a program that is implemented following a bad or non-standard practice that leads to programs that are less well understood, less rigorously implemented, and more error prone.

[38]Program errors are often colloquially called a *bug*. This text will take issue with this term because it makes errors seem somehow less severe and unavoidable. Errors are errors, and most are avoidable, regardless of what users and programmers are led to believe. This text will thus use the more hash term of an error.

This function is erroneous because there is a standard form for functions that should always be followed unless there is a good and well documented reason not to follow the standard[39].

To see what this standard format is, first an example of how a problem can occur will be shown. Consider the case where the `increment` function is much more complex, but still does not call any other function. Note also that now the function passes the value into the function in `r1`, which is a strange behavior. The program will work as shown above. But at some point, the program encounters some complex operation, and the programmer wishes to print out the value of `r1`. They correctly set up the call to `printf`, as shown in the code for the `increment` function in Program 10.

```
#function increment
increment:
    LDR r0, =OutputFormat
    BL printf
    ADD r1, r1, #1
    MOV pc, lr
.data
    OutputFormat: .asciz "r1 = "
#end increment
```

1 **Increment function with `printf`**

If this program is run, it prints out the value of `r1`, but then enters an infinite loop. The `^c` (`ctrl-c`) will stop the program, but the question is, why did it enter an infinite loop?

[39]Note that there will be some "hacker programmer" who will argue that the recommended format shown here requires extra steps, and they can make the program faster by avoiding them. The specific example is chosen to show why that is a bad decision, as not following recommended format will only confuse a programmer seeking to maintain the program. Further, the amount of speed-up from not following the recommended form is negligible, and not worth the increase in risk of causing real program error. Remember these rules:

1 Make a program correct, then make it faster. Once it is correct, you can profile it and using Amdahl's law, address real performance problems, not chimera.
2 It is easier to make a correct program fast than a fast program correct. This is self-explanatory.
3 Fast enough is fast enough. If a program meets any external time constraints, it is fast enough. Adding complexity to make it faster is just being ignorant.

Next run the program in gdbtui, set a break point at the line "`BL printf`" and run the program. When it stops at this line, check the value of the `lr` register, and see that it contains the branch back to the main, as shown in Figure 66.

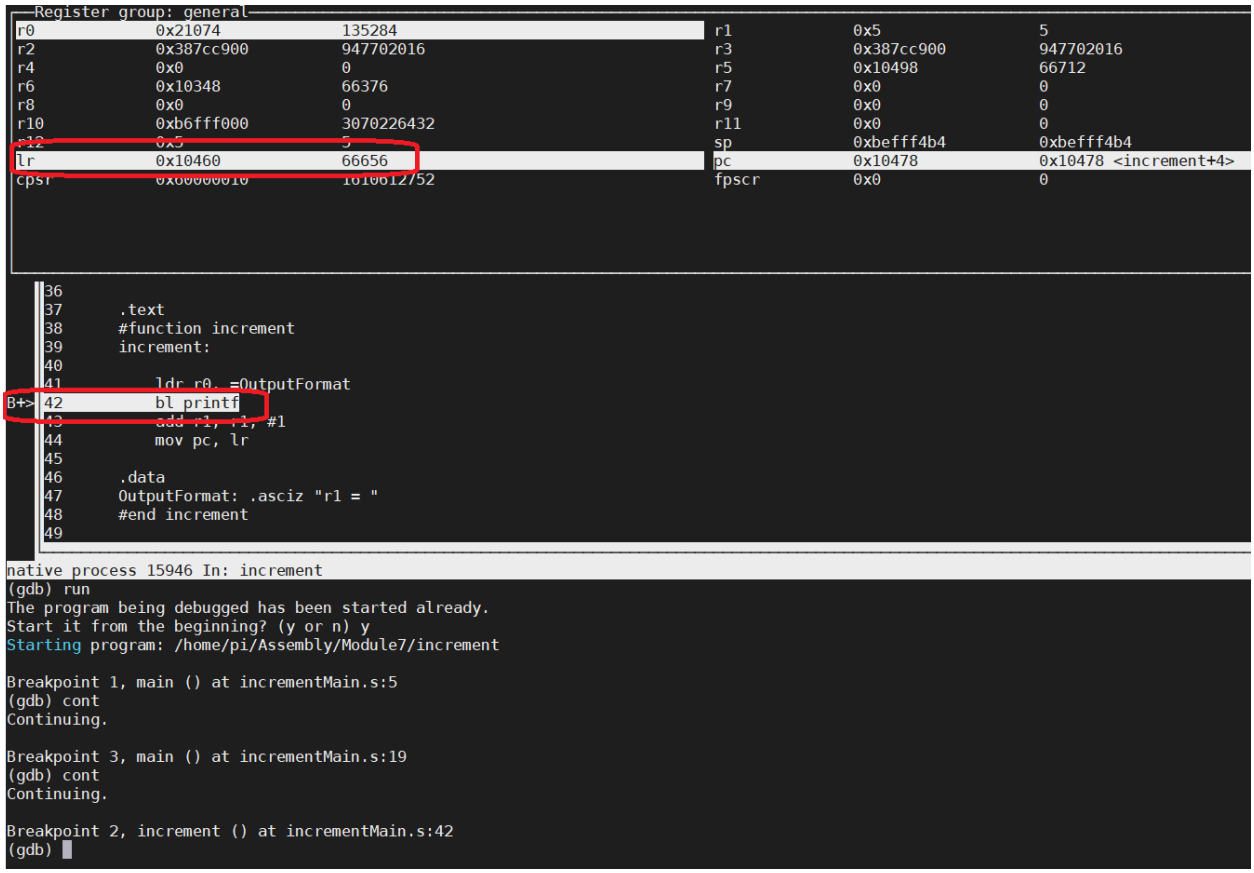


Figure 5: `lr` before branch `printf`

Now type `next`, and notice the value of `lr` after the call to `printf`. The call to the function `printf` has changed it to the current statement, which is the correct place for the function `printf` to return. But this is not where the function `increment` should return, and so when the "`MOV pc, lr`" instruction is executed, it returns to the line "`ADD r1, r1, #1`" in the current function, creating an infinite loop. This is shown in Figure 67.

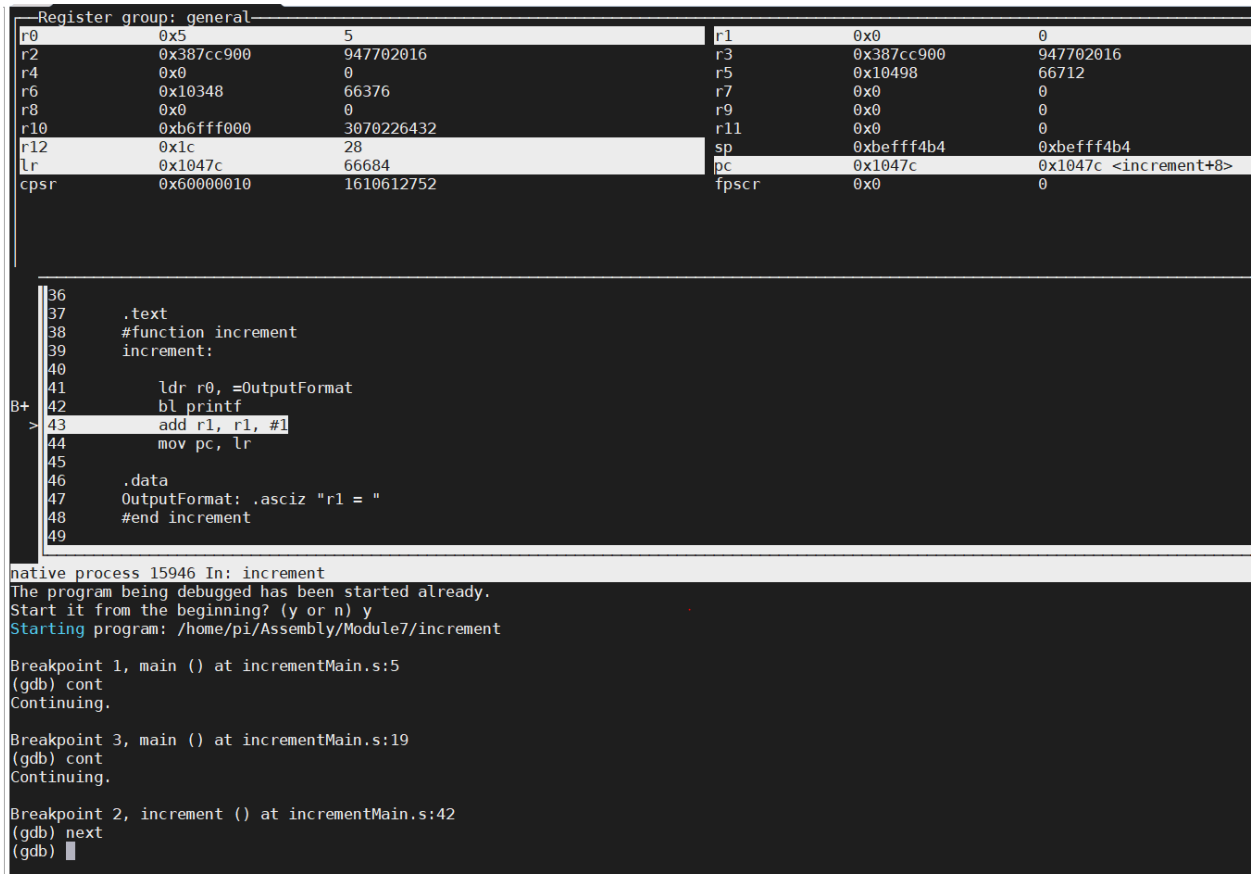


Figure 6: `lr` after branch to `printf`

The `increment` function is a type of function called non-reentrant; it cannot call any other function. Functions that are non-reentrant are often found in the operating system, but non-reentrant functions should be avoided unless there is some very specific use of them.

Chapter 7.2.5 Fixing the problem with a static variable

To fix this problem, some place needs to be found in memory to store the value of the `lr` so it can maintain its original value to be used to return to the calling function. Another register is not a solution, as a function branching to another function branching to another function will overwrite that register. A static value can be created, as shown in Program 11, where the `lr` is stored when entering the function, and then retrieved when exiting the function. This solution was used in early programming language (e.g., FORTRAN versions before FORTRAN 77, COBOL versions before COBOL 2002), but is seldom, if ever, used in current languages. It should be noted that while this solution allows re-entrant function, it does not allow for recursion, as the first call to the function uses the return address variable, and any recursive call to this function will result in the return address being incorrect.

```
#function increment
```

```
increment:
    LDR r3, =ra
    STR lr, [r3, #0]

    LDR r0, =OutputFormat
    BL printf
    ADD r1, r1, #1

    LDR r3, =ra
    LDR pc, [r3, #0]
.data
    ra: word 0
    OutputFormat: .asciz "r1 = "
#end increment
```

2 Saving the lr using a static .data variable

To solve the problem of saving the return address modern programming languages use a *program stack*. A program works by allocating a part of memory for each program or thread that is executing. This stack is used to store several pieces of data, but for now it will be used to store the `lr`.

Chapter 7.2.5 What is a stack

To see how a program stack works, first it is necessary to understand what a stack is. There are two types of memory that can be allocated when a program is running, stack and heap memory. Heap memory is allocated and deleted in non-standard size chunks, and can be held for different periods of time. That makes heap memory management relatively difficult and time consuming.

A stack allocates fixed size chunks of memory that are allocated in a Last In, First-Out (LIFO) format. This allows the stack to be managed relatively easily and quickly.

The most commonly used metaphor for a stack in a computer is trays in a lunch room. When a tray is returned to the stack, it is placed on top of the stack, and each subsequent tray placed on top of the previous tray. When a patron wants a try, they take the first one off of the top of the tray stack. Hence the last tray returned is the first tray used. To better understand this, note that some trays will be heavily used (those on the top), and some, such as the bottom most tray, might never be used.

In a computer a stack is implemented as an array on which there are two operations, push and pop. The push operation places an item on the top of the stack, and so places the item at the next available spot in the array and adds 1 to an array index to the next available spot. A pop operation removes the top most item, so returns the item on top of the stack, and subtracts 1 from the size of the stack.

The following is an example of how a stack works. This example demonstrates the following operations on the stack:

```
push(5)
push(7)
push(2)
print(pop())
push(4)
print(pop())
print(pop())
```

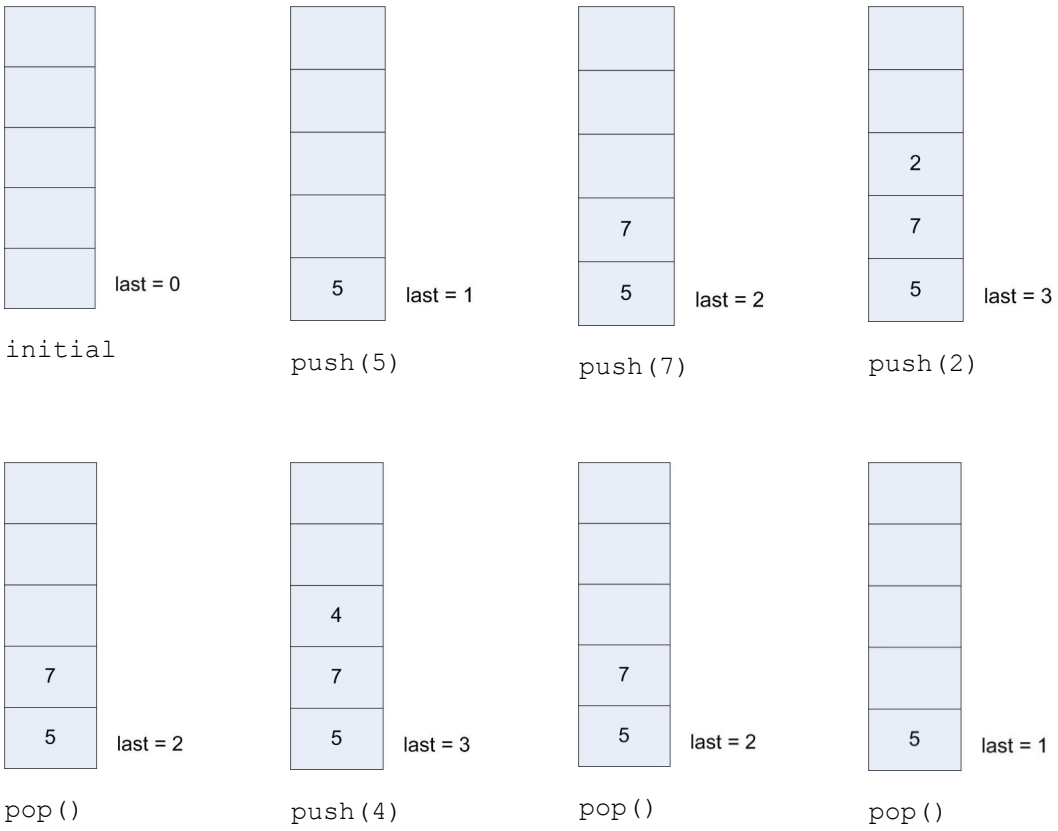


Figure 7: Push/Pop Example 1

The output of the program would be `2, 4, 7`.

This example has a problem though. It seems to imply that every object pushed onto the stack should be the same size. Consider the following example that pushes the words “black” and “cat” onto a stack. In this example, the value pushed on the stack will be different sizes. The

first entry on the stack is the number of spaces that entry requires, and the actual entries follow. This example demonstrates the following operations on the stack:

```
push("Black")
push("Cat")
```

The stack that would be created would be stored in the array of bytes as follows:

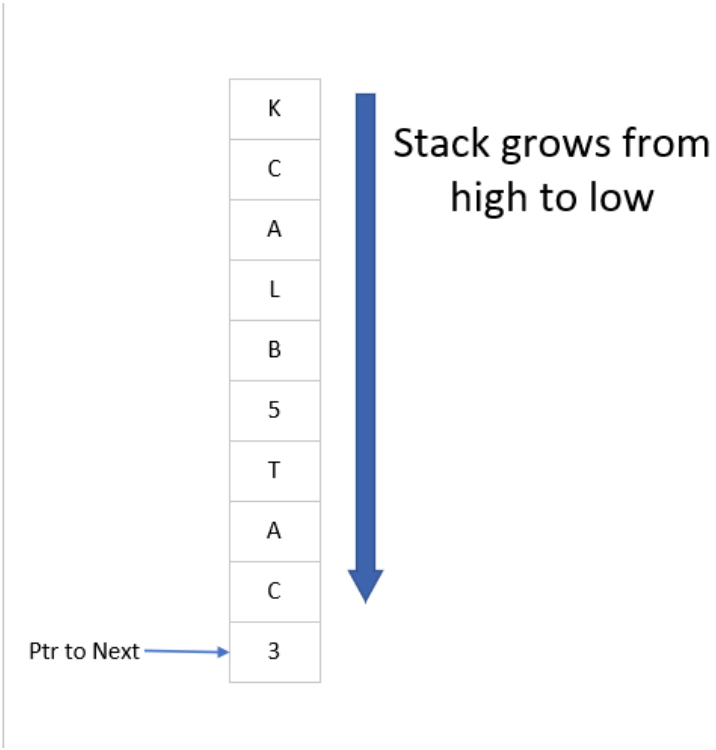


Figure 8: Stack growth

Because the size of the entry (n) is stored with the entry, the entries can be pushed and popped easily by reading the entry at the `Ptr to Next` entry, and adding `n+1` to the pointer. This is how the stack will be implemented in ARM assembly.

Chapter 7.2.5 The program stack

The program stack is an area of memory that by convention all programmers agree to use as stack memory. A pointer to the last entry in the stack, called the *stack pointer* or the `sp` register, is maintained. This points to the beginning of a stack record (or activation record) that maintains data about the function. To create a stack record, the programmer calculates how much space they will need for this stack record. For now, only the `lr` will be stored, so the size will be 4

bytes, the size of a register. This value is *subtracted* from the `sp`, allocating 4 bytes stack record on the stack, and having the `sp` point to that 4 bytes. The `lr` is then stored at that point on the stack. When the function is exited, the 4 bytes are subtracted from the `sp`, returning the memory and causing the `sp` to point to the stack record for the function that called the current function.

```
#function increment
increment:
    #stack push
    SUB sp, 4
    STR lr, [sp, #0]

    LDR r0, =OutputFormat
    BL printf
    ADD r1, r1, #1

    #stack pop
    LDR lr, [sp, #0]
    ADD sp, 4
    MOV pc, lr
.data
    ra: word 0
    OutputFormat: .asciz "r1 = "
#end increment
```

3 Saving the `lr` using the program stack

Because each function uses this same standard format at the start and end of the function, the function `printf` no longer interferes with the `lr` for this function, and the return works correctly.

Observant readers will recognize this is most of the standard code that is placed around the `main` methods of all programs in this textbook.

This is what is meant by a standard format for the functions, that the stack for the function be pushed when the function is entered, and popped when the function is left. It also means that there is *one* entry and *one* exit from any function. You cannot enter a function except at the beginning, and you cannot exit except at the end. Further all code for a function must exist between the push and the pop. This might seem obvious to the reader, but the experience of the author shows that this is far from trivial. The author has seen some pretty egregious violations of this simple rule by experienced programmers, creating situations where the code might work, but is definitely not working as the implementer expected.

Chapter 7.3 Register Conventions

Assembly languages allows a programmer to do anything. If a programmer chooses to add a string pointer to an instruction, it might be meaningless and total nonsense to do so, but assembly language allows it. Likewise, there is no mechanism to enforce protocols that make sure different parts of a program, which can be developed by different programmers, use the same conventions. This can be particularly problematic. If programmers develop functions using different conventions, the results can range from just not making sense, to catastrophic failures of a program, or even every program in a system. To keep this from happening, all assembly language systems have agreed upon standards that all assembly language programs are expected to follow. Any program that does not following these standards is automatically erroneous, and has problems that are just waiting to become bugs.

The standards for writing ARM assembly language are in a document called the Procedure Call Standard for Arm Architecture (AAPCS[40]), with the emphasis on the Application Binary Interface (ABI) section of that document. The conventions and overall structure of a function will follow that standard. Using the standards will result in functions that are easier to implement, understand, and maintain, and functions that are more correct than ones that try to implement functions using adhoc, or worse, no standards.

Chapter 7.4.1 Register Calling Conventions

One issue in implementing functions is register conventions. Register conventions define what registers are valid to pass values into a function and valid for returning values from a function. Register conventions also define the behavior of a register across a function call (e.g., is the register value preserved or not across a function call).

[40]It is called the AAPCS, not the PCSAA, for historical reasons.

To begin to explain these conventions, the following table lists the 16 registers that a user program has access to in ARM assembly, as well as their usage, and whether or not they are preserved across a function call.

Register	**Uses**	**Preserved across function call**
`r0`	Argument and return value	No (Caller preserved)
`r1`	Argument and return value	No (Caller preserved)
`r2`	Argument	No (Caller preserved)
`r3`	Argument	No (Caller preserved)
`r4`	General Purpose	Yes (Callee preserved)
`r5`	General Purpose	Yes (Callee preserved)
`r6`	General Purpose	Yes (Callee preserved)
`r7`	General Purpose	Yes (Callee preserved)
`r8`	General Purpose	Yes (Callee preserved)
`r9`	Normally General Purpose, but in some versions of the AAPCS it is special use	Normally preserved, but see previous column
`r10`	General Purpose	Yes (Callee preserved)
`r11`	General Purpose	Yes (Callee preserved)
`r12`	Scratch register	Yes (Callee preserved)
`r13`	Stack Pointer (`sp`)	Yes (Callee preserved)
`r14`	Link Register (`lr`)	Yes (Callee preserved)
`r15`	Program Counter (`pc`)	No

For this table, the terms Caller and Callee will be defined using the following code fragment.

```
Caller:
    BL Callee
    MOV pc, lr
Callee:
    # do something
    BL Callee
```

As shown in this fragment, the Caller function is any function calling another function. A Callee function is any function called from another function. Note that the Caller and Callee attributes of the relationship between two functions. Functions can be a Caller in relationship to one function, and a Callee in relationship to another function.

The registers in ARM all belong to a specific group or have a specific meaning. Registers `r13`, `r14`, and `r15` are the Stack Pointer (`sp`), Link Register (`lr`) and Program Counter (`pc`), and have been covered in a previous chapter.

Registers `r0-r3` are a group of registers that are used to pass arguments into a function and return values from a function. These registers are not preserved by convention, so they can be used in a function without having to be saved and restored. If they contain values that are needed by the executing function, it is the responsibility of the function to save these values either into preserved registers or stack memory. The standard usage of these registers in this textbook is if the value in the register is meaningful, it should be saved at the start of the function, and that these 4 registers then be used as temporary registers.

For the purposes of this textbook, registers `r4-r12` are group representing general purpose registers. The programmer may safely assume that their values will not be changed as a result of the execution of a Callee function. If a Callee function uses and changes the value in these registers, they must save the original value to the stack when entering the function (in a *push* operation), and restore the original value when leaving the function (in a *pop* operation).

The following program illustrates the use of these registers. In this program, a function called `addValue` is defined which adds a value passed into the function to a value read in from a user, and returns the sum of the argument and the user entered value.

```
main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    MOV r0, #5
    BL addValue
```

```
    MOV r1, r0
    LDR r0, =output
    BL printf

    MOV r0, #0
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr
.data
    output: .asciz "Your answer is %d\n"

.text
# function addValue
addValue:
    SUB sp, sp, #8
    STR lr, [sp,#0]
    STR r4, [sp, #4] // r4 will be used to save r0, so store the
                     // original value to stack
    MOV r4, r0    // Save r0 in r4

    LDR r0, =prompt
    BL printf
    LDR r0, =inputFormat
    LDR r1, =inputNum
    BL scanf
    LDR r1, =inputNum
    LDR r0, [r1, #0]
    ADD r0, r4

    LDR r4, [sp, #4]
    LDR lr, [sp, #0]
    ADD sp, sp, #8
    MOV pc, lr
.data
    inputNum: .word 0
    prompt: .asciz "Enter input number: "
    inputFormat: .asciz "%d"
```

The function `addValue` in this program is passed a value to add in the `r0` register. However, the `r0` register is used by the `scanf` function, and so the value cannot be kept in `r0`. In this program, the original value `r0` (a temporary register) is saved in `r4` (a preserved register). Note that `r4` is a preserved register, so its value is *preserved* or *maintained* across the call to `scanf`. This is in keeping with our convention that when a value in `r0`, `r1`, `r2`, or `r3` is used in the function, the value in that register is saved as soon as the function is entered.

Note that since `r4` is now modified to hold the value originally in `r0`, the `r4` register must be preserved across the call to `addValue`. Thus, `addValue` must save `r4` to the stack when the function is entered, and then retrieve `r4` from the stack when the function returns. This is how the value of a preserved register is maintained across a Callee function and ensures that the value is consistent for the Caller function. A similar process is used in `scanf` if `scanf` uses the `r4` register, which is why the program can be sure the value of `r4` remains unchanged across the call to `scanf`.

Chapter 7.4 Library Files

Normally when writing programs, the entire program is not written in a single file, but instead libraries of functions, objects, and callbacks are created and tied together in a main program. In this textbook only functional abstraction will be considered, and these functions will be gathered together into library files containing multiple similar functions.

This section will be covered by first creating a file `libTypes.s`. First an integer number with an implied decimal point will be explained, and a function to print this variable will be created in this file. Second a function, `inchesToFeet`, will be written in a file named `libConversions.s`, and will be called from a main program. The third subsection will show how to compile and link the pieces of the program together and create a valid executable. The final section will give a brief comment on the Unix `ar`, and the difference between "`.a`" and "`.so`" objects in Unix will be explained.

Chapter 7.4.1 Library file libConversions.s

To start this section the concept of an *implied decimal point integer* number is introduced. This type of number is often called a *decimal number* with a *scale* that is the number of digits to the right of the decimal point. Programmers use decimal numbers when they have a value that has a fixed scale, for instance money which is normally represented as dollars and cents ($3.33) with a scale value of 2. This is useful as it removes the impreciseness of floating point numbers (which might try to print out $3.3333333), which are generally harder to use and require more computer resources to do equivalent operations. These numbers will be used here to allow greater accuracy for the functions in this section, without incurring the added complexity of presenting floating point instructions.

To see how this will work, consider the function `printScaledInteger`, in the file `libTypes.s`. It takes two parameters, the integer value to print in `r0` and the scaling factor as a power of 10 in `r1`. For example, the function would be called with a price of $40.96 as `r0=3795` and `r1 = 100`. The following is the function as it exists in the file `libTypes.s`.

There are a number of features in this file and functions that that reader should be aware of.

1 The name of the file is `libTypes.s`. This is in following with the naming conventions that all library files start with the prefix "`lib`".

2 First, there is a preamble at the top of the file. Any file except for a one-use throw-away file should have a preamble to it (period).

3 The preamble contains the file name. If the file was printed out, there is nothing worse than being unable to find the file again because the name is not known.

4 The function is documented with what it does, and what the input and output parameters are to the function. The input and output are only registers, and so names in assembly are in no way self-documenting. You have to say what the parameters are, or no one will know.

5 The function is named in using a "`.global`" assembly directive. This implies that this function is called from other files and the assembler should make this function available to the linker so it can be resolved at link time. This will be discussed in more detail later when building the program is covered.

6 The registers `r4` and `r5` are preserved registers, and so must be saved when the function is called, and restored when the function returns to the Callee. The is in keeping with the ABI register conventions.

7 The function follows the standard format and pushes the `lr` to the stack when the function is entered and pops the `lr` from the stack when the function exits.

```
#  FileName: libTypes.s
#  Author:   Chuck Kann
#  Date:     1/14/2021
#  Purpose:  Function for use on types
#
#            Types defined and functions:
#            printScaledInt
#
#  Changes:  1/14/2021 - Initial release
.global printScaledInt

#  Function: printScaledInt
#  Purpose:  to print a scaled integer value
#               with the decimal point in the
#               correct place
#
#  Input:    r0 - value to print
#               r1 - scale in
#
#  Output:   r0 - pointer to string that contains
#                    the converted value
.text
PrintScaledInt:
```

```
    # push
    SUB sp, #4
    STR lr, [sp, #0]
    STR r4, [sp, #4]
    STR r5, [sp, #8]
    MOV r4, r0
    MOV r5, r1

    # get whole part and save in r7
    bl __aeabi_idiv // r0/r1, result in r0
    MOV r6, r0
    #get decimal part and save in r7
    MUL r7, r5, r6
    SUB r7, r4, r7

    # print the whole part
    LDR r0, = __PSI_format
    MOV r1, r6
    bl printf
    # print the dot
    LDR r0, = __PSI_dot
    bl printf
    # print the decimal part
    LDR r0, = __PSI_format
    MOV r1, r7
    bl printf

    # pop and return
    LDR lr, [sp, #0]
    ADD sp, #4
    MOV pc, lr

.data
    __PSI_format: .asciz "%d"
    __PSI_dot: .asciz "."
#end printScaledInt
```

4 **Function to print an implied decimal point integer**

This function will be used in the next section where a program that reads feet and inches converts the value to a decimal value of feet.

Chapter 7.4.2 Library file libTypes.s

The next program that will be looked at will take inches as an integer number in `r0` and convert the value to a one decimal place integer representing the number of feet. For example, if 54 inches is passed in, the function will return a value that will be outputted as 1.5 ft.

Note that to conserve space, the functions here will not be fully commented. That does not mean it is okay to not document source programs, just that it is impractical to do so in this textbook.

The first program below is in `libConversions.s` and converts input as feet and inches into a decimal place value of feet.

```
.text

inches2Ft:

    SUB sp, sp, #4

    STR lr, [sp, #0]


    MOV r3, #10

    MUL r0, r0,r3

    MOV r1, #12

    bl __aeabi_idiv

    #answer is returned in r0


    LDR lr, [sp, #0]

    ADD sp, sp, #4

    MOV pc, lr

   1. #END inches2Ft
```

5 Function to convert inches to feet

The following `main` program prompts for a value in inches, coverts it to feet, and prints it.

```
    .text
    .global main

main:
# Save return to os on stack
    MOV sp, sp, #4
    STR lr, [sp, #0]

# Prompt For An Input in inches
    LDR r0, =prompt1
    BL  printf

# Read inches
    LDR r0, =input1
```

```
    SUB sp, sp, #4
    MOV r1, sp
    BL  scanf
    LDR r0, [sp, #0]
    ADD sp, sp, #4

# Convert
    BL inches2Ft
    MOV r4, r0

# Printing The Message
    LDR r0, =format1
    BL  printf
    MOV r0, r4
    MOV r1, #10
    BL printScaledInt
    LDR r0, =newline
    BL printf

# Return to the OS
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr

.data
    prompt1: .asciz "Enter the length in inches you want in feet: \n"
    format1: .asciz "\nThe length in feet is "
    input1:  .asciz "%d"
    newline: .asciz "\n"
    num1: .word 0
```

6 **Program to call `inches2Ft`**

Chapter 7.4.3 Creating the inches2Ft program

Trying to compile and run the `inches2Ft` program directly using the command:

```
gcc inches2FtMain.s -o inches2Ft
```

results in the following error messages:

```
/usr/bin/ld: /tmp/cc5VoT5g.o: in function `main':
(.text+0x28): undefined reference to `inches2Ft'
/usr/bin/ld: (.text+0x40): undefined reference to `printScaledInt'
collect2: error: ld returned 1 exit status
```

The name of the command that had this error is the linker (`/usr/bin/ld`), and the error says that the functions `inches2Ft` and `intScaledInt` could not be found. Thinking about this it is obvious that the problem is that these functions are not in the `inches2FtMain.s` file, but in two library files. These files must be included when linking the file. The following command will do this:

```
gcc inches2FtMain.s libTypes.s libConversions.s -o inches2Ft
```

When library files are changed they are only compiled once to ".o" files, and linking is done with the object files. The following `makefile` assembles the ".s" files to ".o" files, and then uses `gcc` to link all of the ".o" files into a program.

```
all: inches2Ft

MYLIBS = libConversions.o libTypes.o
CC = gcc

inches2Ft: inches2FtMain.o $(MYLIBS)
	$(CC) $@Main.o $(MYLIBS) -g -o $@
	./$@

.s.o:
	$(CC) $(@:.o=.s) -g -c -o $@

clean:
	rm *.o a.out
```

7 **Makefile for `inches2Ft` program**

Chapter 7.5 Problems

1. In the file conversions.s you will implement three functions.

a. The first is k2m, which will convert kilometers to miles by multiplying by 10 and dividing by 16.

b. The second function is mph(int miles, int hours), which will calculate miles per hour by dividing miles (in r0), by hours (in r1), and returning the value in r0.

c. The third function is mph(it kilometers, int hours), and MUST be calculated by converting the kilometers to miles using the function k2m, and then calling mph().

Create a main method in a separate file that will call the kph and mph functions to test them. The function must allow a value to be entered for distance and time in both cases, and print out an answer. You must create good code, including good comments on the methods, and well formatted code.

2. Write the functions CtoF and InchesToFt and add it to the conversions.s file. Write a main program to call it and test it. (40 points) For the InchesToFt,

3. Convert the InchesToFt so it uses implied decimals when converting inches to feet. For example, 14 inches would be 1.16 feet.

4. Create a class Money that you will implement as an integer with two implicit decimal point accuracy. Show it works by reading in Money values from a user, and implemented addition, subtraction, multiplication, and division, and printing valid output as $NNN.NN.

What you will learn.

In this chapter you will learn:

1 why `goto` statements exist in languages

2 how to create logical (or Boolean) variables in assembly

3 the basic control structures used in structured programming, and how to translate them into assembly code; the basic control structures covered are:

3.a `if` statements

3.b `if-else` statements

3.c `if-elseif-else` statements

3.d sentinel control loops
3.e counter control loops

4 calculating branch addresses

Chapter 8 Procedural Programming in Assembly

The structured programming paradigm says that all programs can be built using block structures based on just three (3) types of program control structures. These structures are:

- *sequences,* where programs execute statements in order one after another
- *branches,* where programs to jump to other points in a program
- *loops,* that allow programs to execute a fragment of code multiple times

These three structures are applied to blocks of code. Block structure implies that the block can be treated as a single statement. In practical terms, this means that you can enter the block at the first statement in the block and leave it after the last statement of the block. Effectively, a block is a self-contained unit. While a statement in a block can call a function or method, any branching other than a subroutine call must be to a point inside the currently executing block.

Most modern HLLs are implemented based on block structure with these three program control structures. Most languages include some other structure elements that can be argued are non-structured such as exception handling, `continue`, and `break` statements[41]. The reason for this is

[41]In the opinion of the author, these are structured constructs because they are simply types of branches within a block. But arguing that they should not be included in the language because they are not part of *structured programming* is not helpful, as to write programs without them often requires very convoluted logic. If nothing else, practicality argues for them to be included in a language.

that reasoning about structures, what this text will call *programming plans*, is nearly always more effective than trying to reason about logic flow.

Note that structured programming constructs are not available in assembly language. As was pointed out in Chapter 7 on function execution, the only way to control program execution sequence in assembly language is through the `$pc` register. Therefore, in assembly there are no native structured program constructs. This does not mean that an assembly language programmer should abandon the principals of structured programming. What the lack of language based structured programming constructs means is that the assembler programmer is responsible for writing code which aligns with these principals. Not following structured programming principals in assembly is a sure way to create *spaghetti* code, or code where control is passed uncontrolled around the program, much like the noodles intertwine in a bowl of spaghetti, and following individual strands becomes difficult.

This chapter will introduce pseudo code structure programming control structures similar to those in Java/C/C++/C#. Programmers familiar with those languages should be able to follow the programs with no problems. The text will then show how to translate each control structure from pseudo code into assembly. This text will argue for an approach that treats assembly language programs as a translation of logic that can be implemented in pseudo code, or even a HLL. By understanding the underlying assembly translation, it is hoped that readers who are novice programmers will gain a better understanding and insight into how to structure functions and programs.

All programs in this chapter will be preceded by a pseudo code implementation of the algorithm. Translation from the pseudo code into MIPS assembly will be shown, with the program always representing a translation from the pseudo code. No programs will be developed directly into assembly. The reason for this is though the direct implementation of the programs in assembly allows more flexibility, programmers in assembly language programming often implement these programs in very unstructured fashions which often result in poor programs and can develop into poor understanding and practices.

To reemphasize the point, it has been the experience of the author that although many new assembly language programmers often try to avoid the structured programming paradigm and reason through an assembly language program, the results are seldom satisfactory. The reader is strongly advised to follow the principals outlined in this chapter, and not attempt to develop the programs directly in assembly language.

Program 8.1 Programming Plans

The first issue in this chapter is how to structure program logic. It reminds me of a story where a chess master once played a novice player, and after a few moves, the expert simply did not want to continue the game. The novice was making moves that bothered the expert because the moves made by the novice were so far removed from how the expert reasoned about the game.

A similar feeling emerges when an expert programmer looks at a novice's (or even a poor professional's) program. The programs just feel wrong. It is not just that indenting is off, and the novice programmer has followed some weird, or nonexistent naming standard, it is that nothing is organized correctly.

Most beginning programmers are taught that programs are algorithms where it is the logic that needs to be followed. At some basic level it is true that programs are logic that needs to be understood. However, research has shown that while beginning programmers (and poorly performing professionals) understand programming in terms of algorithmic flow, advanced programmers generally have created cognitive schema for a large number of problems, called *programming plans*[42]. When reading and writing code these programmers will often look first to the plans, perhaps putting in a few lines out code to outline the plan, and only later going back to fill in the details.

To illustrate the difference between a code fragment using a logic-based approach and programming plans, consider a simple fragment to calculate the `sum` of values from `1` to `n`. When using logic, the novice is taught that they should initialize a `sum` to zero, and then create a loop. A loop will start at the `for` statement which initializes a variable `j` to 0, then moves to the next statement that adds `j` to the value of sum, and then at the end of the loop adds `1` to the value of `j` and goes back to the beginning of the loop, as in the following fragment:

```
int sum = 0;
for (int j = 0; j < n; j++)
{
      sum = sum + j
}
```

In the author's experience this way of teaching looping is less than satisfactory. Even for the students who eventually get it, it is abstract and hard to follow, and some people never understand it.

A program plan, on the other hand, does not begin with logic, but with what is to be done, and the structure that applies to it. To calculate a sum, the programmer must start with 0 and add 1 to it for each number up to n. This we have:

```
sum = 0 + 1 + 2 + 3 + 4 + … n
```

[42]Note that programming plans are very different from design patterns. It is surprising how often these two concepts will get confused with each other, when they are plainly different in use and scale.

To do this, we introduce a loop. All loops (counting or sentinel, or any other kind) have the following structure:

```
// initialize
// start iteration
//    check if done
//    do iteration logic
//    get the next item
//    go back and do the next iteration
// place to go when you are done
sum = 0 // initialize
j = 0
while (j <= n) //check if done
{
    sum = sum + j  // iteration logic
    j = j + 1 // get the next item
    // Go back to the beginning
} // Loop ends here
```

This is then called a summation loop, and the programmer refers to it when they need to write larger programs, such as an average program. An average program is simply a summation plan that maintains the number of items processed, and a final statement to calculate the average after the loop. This process of building larger programming is called plan merging.

While there is no research that I know of to say that students learn better when presented logic based programming or programming plans and plan merging, the research is clear that good programmers use programming plans.

But what is obvious to the author is that when doing assembly language programming, if users do not use block structures and procedural language programming constructs, but are left to their own devices, the results are almost uniformly bad. Some structure must be enforced, so block structured programming will be used with programming plans.

Program 8.2 Use of goto statements

Many readers of this text will quickly recognize the main mechanism for program control in assembly, the branch statement, as simply a `goto` statement. These readers have often been told since they started programming that `goto` statements are evil and should never be used. The reasoning behind this rule is seldom explained, and an almost religious adherence has developed to the principal that `goto` statements are always suspect and should never be used. Like most unexamined principles, this simply misses the larger point.

The problem with `goto` statements is that they allow unrestricted branching to any point in a program. Indeed, this type of unrestricted branching leads to many obfuscated programs before structured computing. However, with the advent of structured programming languages, the use of the term *spaghetti code* has even gone out of the normal programmer's vernacular. But it was never the use of `goto` statements that lead to obfuscated programs; it was programmers' penchants for doing the expedient that resulted in unorganized programs. The unrestricted `goto` statement never was the problem, it simply was the mechanism that allowed the programmers to create problems.

In assembly language, the only method of program control is through the `$pc`, and the only way to implement branch statements. The branch statements themselves will not lead to unorganized programs, but the unorganized thoughts of the programmers will. So, this chapter will not teach how to reason about assembly language programs. All programs will be first structured in pseudo code and then translated into assembly language. Readers who follow the methodology presented in this text will never encounter an *unrestricted* `goto`. All branch statements will be explicitly structured, and all branches will be within the code blocks that contain them, just as in structured programming languages. So, the branch statements in this text are not evil, and the idea that somehow simply using a `goto` is wrong needs to be modified in the reader's mind.

Program 8.3 Conditional Execution and the apsr Register

ARM assembly implements conditional execution of assembly language statements, which allows the results of the previous statement to be used to determine whether or not to execute the current statement. This condition can be applied to any operator, but is most useful for the branch (`B` and `BL`) operators. A new operator, the compare (`CMP`) operator, can be used to compare two registers; the branch taken depends on the conditional execution value set for the instruction. For example, to branch to the label `"branchTarget"` if `r0 = r1`, the programmer can use the following code fragment:

```
CMP r0, r1
BEQ branchTarget
```

To branch and link (`BL`) to the function `func1` if the value of `r0 > r1`, the programmer can use the following code fragment:

```
CMP r0, r1
BLGT func1
```

The check for the previous statement can be applied to any statement, not just the `CMP` statement. The following example shows how an `S` can be appended to the `SUB` operation, creating a `SUBS` operation. This operation tells the CPU to store *condition flags* to categorize the result from this instruction. These condition flags are saved in the Application Program Status Register (`apsr`)[43].

[43]There are two Program Status Registers: the Application Program Status Register (`apsr`) and the Current Program

For example, assume that `r4 = r2*(r0-r1)`, but the code only runs if the value of `r0 - r1` is greater than `0`.

```
if ((r0 = r0 - r1) > 0){
      r4 = r2 * r0
}
```

This can be written in ARM assembly as:

```
SUBS r0, r0, r1
MULPL r4, r4, r0
```

In this case if `r0 - r1` is positive (`PL` means plus), the value in `r2` will be multiplied with `r0` and stored in `r4`, otherwise the statement is just ignored.

The `apsr` is a 32-bit register that contains 4 bits that give information about the previous instruction. The 4 bits are condition flags with the following values:

- `N` – The negative flag, which is set to 1 if the result is negative. This is accomplished by assuming the number is a 2's complement value and copying the sign bit from the result of the operation.
- `Z` – The zero flag, which contains 1 to indicate that the result of the instruction is zero.
- `C` – Carry or Unsigned Overflow, which contains 1 to indicate that the result of the operation the 32-bit result register. This bit can be used to implement 64-bit unsigned arithmetic, for example.
- `V` – Signed Overflow, which contains the value of 1 if the result of the operation would overflow a signed 32-bit value. For example, `0x7fffffff` is the largest positive two's complement integer that can be represented in 32 bits, so `0x7fffffff + 0x7fffffff` triggers a signed overflow, but not an unsigned overflow (or carry): the result, `0xfffffffe`, is correct if interpreted as an unsigned quantity, but represents a negative value (`-2`) if interpreted as a signed quantity.

The condition flags set by the `SUBS` instruction are then checked by the next instruction, the `MULPL` instruction. The instruction `MULPL` is a `MUL` instruction with a *condition code* `PL`. The condition code `PL` means only run the statement if the result from the previous command is

---

Status Register (`cpsr`). The `cpsr` holds all the information in the `apsr`, but the `cpsr` holds additional information, and is used in other processor modes (the `apsr` is used if the processor is in User Mode). However only the `apsr` is available in all processor modes. The `apsr` is the more recent name, but many authors will use the `cpsr`. This text will use the `apsr`, but for all uses in this text, the two are interchangeable.

positive (`PL` is short for *plus*). Thus, the pseudo code above is implemented in the two assembly instructions.

Most ARM assembly operations can have a condition code to indicate whether the instruction should be run or not. For example, saying `addgt` means to run the `add` operation if the condition flags indicate the previous instruction resulted in a `gt` condition, and `lslne` runs the `lsl` operation if the previous instruction resulted in a `ne` condition. This leads to ARM having a dauntingly large number of apparent operations because of the combinations of operations and condition codes, but in reality, the number of actual operations and condition codes is much smaller and manageable.

While having the ability of any assembly language statement to set the condition flags, there are 4 ARM assembly instructions that are specifically associated with setting these flags. These 4 instructions are `CMP`, `CMN`, `TST`, and `TEQ`. These operands are summarized in the following table.

Operator	**Meaning**
`CMP`	Compares the two register operands. The operator is effectively a SUBS but the result is not stored.
`CMN`	The operator is effectively an ADDS but does not store the result.
`TST`	The operator is effectively an ANDS but does not store the result. Can be used for bit masking
`TEQ`	The operator is effectively an EORS but does not store the result. Can be used for bit masking.

Table 11: Assembly instructions that set the condition flags

The format for these operators is summarized below. Examples of how to use the CMP operator will be given later in the chapter.

```
CMP Rn, Operand2

CMN Rn, Operand2

TST Rn, Operand2

TEQ Rn, Operand2
```

The condition codes that can be appended to the operators in ARM are summarized in the following table. Note that these condition codes are for the `CMP` operator only. They do not apply to the `CMN`, `TST`, or `TEQ` operators.

Code	**Meaning for CMP operator**	**Flags used**	**Condition**

			Code
EQ	Equal	Z = 1	0000 (0X00)
NE	Not equal	Z = 0	0001 (0X01)
CS or HS	Carry bit set[44], or unsigned higher or same	C = 1	0010 (0X02)
CC or LO	Carry bit clear or unsigned lower	C = 0	0011 (0X03)
MI	Minus or negative	N = 1	0100 (0X04)
PL	Plus or positive	N = 0	0101 (0X05)
VS	V bit set, or signed overflow	V = 1	0110 (0X06)
VC	V bit clear, or no signed overflow	V = 0	0111 (0X07)
HI	Unsigned higher	C = 1 and Z = 0	1000 (0X08)
LS	Unsigned lower or same	C = 0 or Z = 1	1001 (0X09)
GE	Signed greater than or equal	N = V	1010 (0X0a)
LT	Signed less than	N != V	1011 (0X0b)
GT	Signed greater than	Z = 0 and N = V	1100 (0X0c)
LE	Signed less than or equal	N != V or Z = 1	1101 (0X0d)
AL or omitted	Always execute		1110 (0X0e)

Table -12: Condition Codes

These changes to the assembly instructions permit the final parts of the machine code formats to be explained. If the assembly instruction includes an S, the S-bit in the instruction is set to 1. Finally, if the instruction appends a Condition Code, Table 12 above can be used to find the value to set this code to.

Program 8.4 Branching

Chapter 8.4.1 Simple If statements

Now that the handling of conditional logic has been covered, how to use this knowledge to implement branching and looping can be addressed.

This section will begin with a small pseudo code example of an if statement.

[44] A field is *set* if it is equal to 1; it is clear if it is equal to 0.

```
if (r1 > 0)
{
    print("Number is positive")
}
```

In this statement, the value in `r1` is checked to see if it is a positive integer. If it is, the string `"Number is positive"` is printed, otherwise nothing happens. The most important characteristic of this code fragment is the statement that prints the output happens when the condition is positive. This might seem obvious, but taking note of it now will save confusion very soon.

The second thing to notice is that in this code fragment the statement that is executed is a code block contained between the curly brackets (`"{"` and `"}"`). Any code block that is between curly brackets is the equivalent of a statement, which is why this works. However, for the sake of clarity in this discussion all branching will be into a code block.

This simple `if` statement will begin to define the canonical form for an `if` statement. Here the form is simple:

```
CMP r1, r2

B{condition flag for false} End_If_Label

    block of code to enter if condition is true

End_If_Label
```

Note that if the condition tested for is true, the block should be entered. Therefore, the condition flag to check is the condition flag for the false condition. This is obvious if it is thought about, but most programmers instinctively want to branch on the positive condition. Branching on the positive condition actually invalidates structured programming, where blocks are checked and entered, each in turn, if the condition is positive. It leads to branching to solve the immediate problem, and quickly devolves to spaghetti code. That is why almost all students that the author has encountered, when left to their own devices in assembly, reinvent spaghetti code. Disorganized code is the natural orientation, and organized systems are unnatural if not enforced.

The `if` statement in the pseudo code above is now implemented in the following program.

```
    .text
    .global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    MOV r2, #92
```

```
    MOV r1, #0
    CMP r2, r1
    BLE EndIf
        LDR r0, =IsPositive
        BL printf
    EndIf:

    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr

.data
    IsPositive: .asciz "Number is Positive\n"
```

Chapter 8.4.2 Complex logical statements

While the example above showed how to translate a single logical condition, it begs the question of how to translate complex logical conditions. Programmers might think that to translate a condition such as the one that follows requires complex programming logic. Remember that ASCII codes for a number `n` is an integer value such that $30 \leq n \leq 39$, and the integer value of `n`, called `i` here, is `i = n - 30`. This is the result of the following code fragment:

```
if ((n >= 30) && (n <= 39))

    i = n - 30;
```

One of the reasons programs became complex before structured programming became prevalent is that programmers would try to solve this type of complex logical condition by reasoning about the program. This could result in mostly uncommented code that would look very similar to the following program. For readers who recognize this type of program, you are old. For those of you who do not believe programs like this existed, this is actually nice code. It is indented, does not have a hug number of variables in a single global memory, and it works. This would have been uncommon before language that use structured constructs, like C or Java.

```
    .text
        .global main
    main:
        SUB sp, sp, #4
        STR lr, [sp, #0]

        MOV r0, #0x32
        BL converToInt

        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr
```

```
converToInt:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    mov r4, #0x30
    cmp r0, r4
    blt NotANumber

    mov r4, #0x39
    cmp r0, r4
    blt convert
    b NotANumber

IsANumber:
    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr

NotANumber:
    LDR r0, =output
    BL printf

    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr

convert:
    SUB r0, #0x30
    B IsANumber

.data
    output: .asciz "NAN\n"
```

If reasoning about a program is a bad choice to solve logic, how should a programmer proceed? The easy way to solve this problem is to realize that in a HLL, the compiler is going to reduce the complex logical condition into a logical equation that will reduce into a logical (or boolean) variable.

To begin, most HLLs represent a boolean (or logical) variable as a 32-bit value where only the lowest order bit is used. Since only one bit is used, this reduces the equation to a simple logic expression. In face, if the upper 31 bits are assumed to be 0, and only the single lowest order bit is considered, all of the bitwise operations (with the exception of logical NOT) become logical operations. The complex `if` statement above would be translated into the equivalent of the following code fragment:

```
boolean logical = (n >= 30) && (n <= 39));
```

```
if (logical)

    i = n - 30;
```

This code fragment is easily translated into the following assembly language code fragment. Note that in this code fragment `r4` and `r5` will only have a value of 0 or 1, bits 1..31 will always be 0.

```
        .text
        .global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    MOV r0, #0x32
    BL convertToInt

    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr
# End main

convertToInt:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    MOV r4, #0
    MOV r1, #0x30
    CMP r0, r1
    MOVGT r4, #1

    MOV r5, #0
    MOV r1, #0x39
    CMP r0, r1
    MOVLT r5, #1

    AND r4, r4, r5

    MOV r1, #0
    CMP r4, r1
    BEQ Else
        SUB r0, r0, #0x30
        B EndIf

    Else:
        ldr r0, =output
        BL printf

    EndIf:
```

```
        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr

    .data
        output: .asciz "NAN\n"
```

The code for using the logical variable is roughly the same amount of code as the spaghetti code, but I believe most readers will find it much easier to follow, even though it is not documented. This is because the code is logically coherent. It doesn't require a lot of interleaving reasoning with confusing branching, implementing what is effectively unrestricted `goto` statements. As the exercises at the end of the chapter will show, this structure for processing logical statements grows linearly in complexity, whereas the complexity of using program reasoning becomes overwhelming complex quickly.

Chapter 8.4.3 If-Else statements

A more useful version of the `if` statement also allows for the false condition, or an `if-else` statement. If the condition is true, the first block is executed, otherwise the second block is executed. A simple code fragment illustrating this point is shown below.

```
if (($r0 > 0) == 0)
{
   print("Number is positive")
}
else
{
   print("Number is negative")

}
```

This is a modification to the logic in the simple `if` statement. This code will output an answer stating if the value in `r0` is positive or negative. This section builds on the `if` statement to show how to translate an `if-else` statement from pseudo code to assembly language. To translate the `if-else` statement, use the following steps.

1 Implement the conditional part of the statement to create a logical variable that indicates whether to enter the block or branch.

2 Add two labels to the program, one for the `else` and one for the end of the `if` (e.g., an `endIf` label). The branch should be inserted after the evaluation of the logical variable. The negative condition for the branch will be to the `else` label. This allows the positive condition to sequentially flow into the `if` block.

3 At the end of the `if` block, branch around the `else` block by using an unconditional branch statement to the `endIf`. You now have the basic structure of the `if` statement, and your code should like the following assembly code fragment.

```
MOV r1, #0
CMP r0, r1
BLE Else
   # if block
   B EndIf

Else:
   #else block

EndIf:
```

4 Once the structure of the `if-else` statement is in place, you should put the code for the blocks into the program. This completes the `if-else` statement translation. This is the following program.

```
    .text
        .global main

    main:
        SUB sp, sp, #4
        STR lr, [sp, #0]

        MOV r0, #-0x32

        # (if r0 > 0)
        MOV r1, #0
        CMP r0, r1
        BLE Else
            # Code block for if
            LDR r0, =positive
            BL printf
            B EndIf

        Else:
           # Code block for else
           LDR r0, =negative
           BL printf

        EndIf:

        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr
    # End main
```

```
.data
    positive: .asciz "Number is Positive\n"
    negative: .asciz "Number is Negative\n"
```

Chapter 8.4.4 If-ElseIf-Else statements

The final type of branch to be introduced in this text allows the programmer to choose one of several options. It is implemented as an `if-elseif-else` statement. In this statement, the `if` and `elseif` statements will contain a conditional to decide if they will be executed or not. The `else` will be automatically chosen if no condition is true.

To introduce the `if-elseif-else` statement, the following code fragment that translates a number grade into a letter grade is implemented. The following pseudo code fragment shows the logic for this `if-elseif-else` statement.

```
if (grade > 100) || grade < 0)
{
    print("Grade must be between 0..100")
}
elseif (grade >= 90)
{
    print("Grade is A")
}
elseif (grade >= 80)
{
     print("Grade is B")
}
elseif (grade >= 70)
{
     print("Grade is C")
}
elseif (grade >= 60)
{
```

```
        print("Grade is D")
    }
    else{
        print("Grade is F")
    }
```

To translate the `if-elseif-else` statement, once again the overall structure for the statement will be generated, and then the code blocks will be filled in. Readers and programmers are strongly encouraged to implement algorithmic logic in this manner. Students who want to implement the code using some sort of reasoning will find themselves completely overwhelmed and will miss many important algorithmic decisions[45], especially when blocks containing other blocks as nested logic are used later in this chapter.

The steps in the translation of the `if-elseif-else` statement are as follows.

1 Implement the beginning of the statement with a comment, and place a label in the code for each `elseif` condition, and for the final `else` and `EndIf` conditions. At the end of each code block, place a branch to the `end-if` label (once any block is executed, you will exit the entire `if-elseif-else` statement). Your code would look as follows:

```
#if block
    # first if check, invalid input block
    b EndIf
grade_A:
    b EndIf
grade_B:
    b EndIf
grade_C:
    b EndIf
grade_D:
    b EndIf
else:
    b EndIf
End_If:
```

2 Next put the logic conditions in the beginning of each `if` and `elseif` block. In these `if` and `elseif` statements the code will branch to the next label. When this step is completed, you should now have code that looks something like the following (note: the grade is in `r4`):

```
#if block
```

[45]There is good reason to believe that reasoning about programming logic is a fools errand. Miller's theory holds that the average human's short term memory capacity is 7 ± 2 items. It does not take very long before the logic in a program overwhelms a programmer's ~~short term~~short-term memory, and the programmer cannot keep track of all the logic in a given algorithm. This logic must be *chunked* in some fashion to deal with it. Thus, the need for programming plans and other methods to structure a program.

```
    #check 0 <= r4 <= 100
    MOV r1, #0
    MOV r0, #0
    CMP r4, r0
    MOVGE r0, #1

    MOV r2, #0
    MOV r0, #100
    CMP r4, r0
    MOVLE r2, #1

    AND r1, r1, r2
    MOV r2, #1
    CMP r1, r2
    BEQ grade_A   // Grade is valid

    # Code block for Invalid Grade
    B EndIf

grade_A:
    MOV r0, #90
    CMP r4, r0
    BLT grade_B

    # Code block for grade of A
    B EndIf

grade_B:
    MOV r0, #80
    CMP r4, r0
    BLT grade_C

    # Code block for grade of B
    B EndIf

grade_C:
    MOV r0, #70
    CMP r4, r0
    BLT grade_D

    # Code block for grade of C
    B EndIf

grade_D:
    MOV r0, #60
    CMP r4, r0
    BLT Else

    # Code block for grade of D
    B EndIf

Else:
    # Code block for grade of F
```

```
        B EndIf
EndIf:
```

3 The last step is to fill in the code blocks with the appropriate logic. The following program implements this completed `if-elseif-else` statement. This final program, called `CheckGrades.s`, is shown below.

```
    .text
    .global main

    main:
    # Save return to os on stack
        SUB sp, sp, #4
        STR lr, [sp, #0]

        MOV r4, #92

    #if block
        #check 0 <= r4 <= 100
        MOV r1, #0
        MOV r0, #0
        CMP r4, r0
        MOVGE r1, #1

        MOV r2, #0
        MOV r0, #100
        CMP r4, r0
        MOVLE r2, #1

        AND r1, r1, r2
        MOV r2, #1
        CMP r1, r2
        BEQ grade_A   // Grade is valid

        # Code block for Invalid Grade
        LDR r0, =Invalid
        BL printf
        B EndIf

        grade_A:
        MOV r0, #90
        CMP r4, r0
        BLT grade_B

        # Code block for grade of A
        LDR r0, =GradeA
        BL printf
        B EndIf

        grade_B:
        MOV r0, #80
        CMP r4, r0
        BLT grade_C
```

```
        # Code block for grade of B
        LDR r0, =GradeB
        BL printf
        B EndIf

        grade_C:
        MOV r0, #70
        CMP r4, r0
        BLT grade_D

        # Code block for grade of C
        LDR r0, =GradeC
        BL printf
        B EndIf

        grade_D:
        MOV r0, #60
        CMP r4, r0
        BLT Else

        # Code block for grade of D
        LDR r0, =GradeD
        BL printf
        B EndIf

        Else:
        # Code block for grade of F
        LDR r0, =GradeF
        BL printf
        B EndIf

        EndIf:

    # Return to the OS
        ldr lr, [sp, #0]
        add sp, sp, #4
        mov pc, lr

    .data
        GradeA: .asciz "Grade is A\n"
        GradeB: .asciz "Grade is B\n"
        GradeC: .asciz "Grade is C\n"
        GradeD: .asciz "Grade is D\n"
        GradeF: .asciz "Grade is F\n"
        Invalid: .asciz "Grade must be 0 <= grade <= 100\n"
```

Program 8.5 Looping

Loops are central to most algorithms, and hence play an important part in programming. When teaching looping, it seems that most students either tend towards or a taught a view of loops that focuses on how loops work, the reasoning about them. I find this is not really a fruitful approach to the subject, as simply some, if not most, students have a very difficult time recognizing that a loop counter variable takes on a different value each time through a `for` loop. This is why when teaching an introduction to programming class, the author would often emphasize the program plans to be implemented, and allow the students to develop their own model of how this worked.

This book will not try to explain how loops work, but rather introduce loops as a conical structure, just as the structure and not the logic of `if` statements was covered in the previous section. The program plans that then use loops will be implemented by expanding the loop structure, or more appropriately *plan merging*, where different types of plans are combined to create larger program plans.

All looping structures in this textbook will have the following structure:

```
    # initialize first element

startLoopLabel:

        # check if loop is complete, if yes, branch to endLoopLabel


        # loop block


        # get next element

        # branch back to startLoopLabel

    endLoopLabel
```

This structure will be shown by construction to fit the major loop constructs, *sentinel control loops* and *counter control loops*, thus showing that `for` and `while` loops are both adequately addressed with this structure. In addition, by delaying the implementation of the *loop block*, the loop can be addressed as a complete unit, removing the nagging problems where parts of the loop, like branching back to *startLoopLabel*, is forgotten and omitted. Finally, by delaying the implementation of the *loop block*, the degree of complexity when implementing a program can be greatly reduced, making programs easier to implement, maintain, and use. This will be illustrated using a Bubble Sort at the end of the chapter.

The next two sections will show how to implement a sentinel control loop and a counter control loop. The two sections after that will show how to do plan merging by creating a program to query a user to calculate a summation until a sentinel value is entered.

Chapter 8.5.1 Sentinel Control Loop

The definition of a sentinel is a guard, so the concept of a sentinel control loop is a loop with a guard statement that controls whether or not the loop is executed. The major use of sentinel control loops is to process input until some condition (a sentinel value) is met. For example, a sentinel control loop could be used to process user input until the user enters a specific value, for example -1. The following pseudo code fragment uses a `while` statement to implement a sentinel control loop which prompts for an integer and prints that integer back until the user enters a -1.

```
int i = prompt("Enter an integer, or -1 to exit")
while (i != -1)
{
    print("You entered " + i);
    i = prompt("Enter an integer, or -1 to exit");
}
```

The following defines the steps involved in translating a sentinel control loop from pseudo code into assembly.

1 Initialize the loop by setting the sentinel to be checked before entering the loop. For the loop in this program, this requires a number of statements to one prompting and read user input.

2 Create a label for the start of the loop. This is so that at the end of the loop the program control can branch back to the start of the loop.

3 Create a label for the end of the loop. This is so the loop can branch out when the sentinel returns false.

4 Put the check to check the sentinel to see if the loop should be exited.

5 Set the sentinel to be checked as the penultimate statement(s) in the code block for the loop, and then unconditionally branch back to the start of the loop. This completes the loop structure, and you should have code that appears similar to the following:

```
.text
.global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    MOV r0, #-0x32

    # initialize by prompting user, answer in r4
    LDR r0, =prompt
    BL printf
    LDR r0, =input
    LDR r1, =num1
    BL scanf
```

```
        LDR r1, =num1
        LDR r4, [r1, 0]

    StartSentinelLoop:
        MOV r0, #-1
        CMP r4, r1
        BEQ EndSentinelLoop

            # Loop Block

            # Get next value
            LDR r0, =prompt
            BL printf
            LDR r0, =input
            LDR r1, =num1
            BL scanf
            LDR r1, =num1

            B StartSentinelLoop

    EndSentinelLoop:

        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr
    # End main

    .data
        prompt: .asciz "Please enter a number (-1 to end) \n"
        output: .asciz "You entered %d\n"
        input: .asciz "%d"
        num: .word 0
```

6 The structure needed for the sentinel control loop is now in place. The logic to be executed in the code block can be included and any other code that is needed to complete the program. The final result of this program follows.

```
.text
.global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    MOV r0, #-0x32

    # initialize by prompting user, answer in r4
    LDR r0, =prompt
    BL printf
    LDR r0, =input
    LDR r1, =num
    BL scanf
    LDR r1, =num
```

```
    LDR r4, [r1, #0]

StartSentinelLoop:
    MOV r0, #-1
    CMP r4, r0
    BEQ EndSentinelLoop

        # Loop Block
        LDR r0, =output
        MOV r1, r4
        BL printf

        # Get next value
        LDR r0, =prompt
        BL printf
        LDR r0, =input
        LDR r1, =num
        BL scanf
        LDR r1, =num
        LDR r4, [r1, #0]

        B StartSentinelLoop

EndSentinelLoop:

    LDR lr, [sp, #0]
    ADD sp, sp, #4
    MOV pc, lr
# End main

.data
    prompt: .asciz "Please enter a number (-1 to end) \n"
    output: .asciz "You entered %d\n"
    input: .asciz "%d"
    num: .word 0
```

Chapter 8.5.2 Counter control loop

A counter controlled loop is a loop which is intended to be executed some number of times. These are normally associated with a `for` loop in most HLLs. An example is the following pseudo code `for` loop which sums the values from `0` to `(n-1)`.

```
n = prompt("enter the value to calculate the sum up to: ")
total = 0; # Initialize the total variable for sum
for (i = 0; i < n; i++)
{
    total = total + i
}
print("Total = " + total);
```

The `for` statement itself has 3 parts[46]. The first is the initialization that occurs before the loop is executed ("`i=0`"). The second is the condition for continuing to enter the loop ("`i < size`"). The final condition specifies how to get the next value, here the counter is incremented ("`i++`", or add `1` to `i`). These 3 parts of a `for` loop map exactly into the loop structure covered earlier and are translated to this structure as follows.

1 Implement the initialization step to initialize the counter and the ending condition variables.

2 Place a `startLoopLabel` and an `endLoopLabel` in the program.

3 Implement the check to enter the loop block or stop the loop when the condition is met.

4 Add the code at the end of the loop to increment the counter and branch back to the start of the loop.

When you have completed these steps, the basic structure of the counter control loop has been implemented, and your code should look similar to the following:

```
.text

    MOV r0, #0
    MOV r5, #0

    StartCountingLoop:
        CMP r0, r4
        BGE EndCountingLoop

        # Loop Block

        # Get next value
        ADD r0, r0, #1

        B StartCountingLoop
    EndCountingLoop:
```

5 Implement the code block for the `for` statement. Implement any other code necessary to complete the program. The final assembly code for this program should look similar to the following.

```
.text
.global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]
```

[46]In many HLLs like Java or JavaScript, the `for` loop has been repurposed to also implement iterators, which are types of sentinel control loops. This section deals specifically with `for` statements that are used for counter controlled loops.

```
        # Prompt for loop limit, store in r4
        LDR r0, =prompt
        BL printf
        LDR r0, =input
        LDR r1, =num
        BL scanf
        LDR r1, =num
        LDR r4, [r1, #0]

        # initialize the loop,
        # r0 - counter
        # r4 - loop limit
        # r5 - sum

        MOV r0, #0
        MOV r5, #0

        StartCountingLoop:
        CMP r4, r0
        BLE EndCountingLoop

            # Loop Block
            ADD r5, r5, r0

            # Get next value
            ADD r0, r0, #1

            B StartCountingLoop
        EndCountingLoop:

        LDR r0, =output
        MOV r1, r5
        BL printf

        MOV r0, #0
        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr
    # End main

    .data
        prompt: .asciz "Please enter the loop limit to sum \n"
        output: .asciz "The summation from 1 to n is  %d\n"
        input: .asciz "%d"
        num: .word 0
```

Chapter 8.5.3 Nested Code Blocks

It is common in most algorithms to have nested code blocks. A simple example would be a program which calculates the sum of all values from `0` to `n`, where the user enters values for `n`

until a `-1` if entered. In addition, there is a constraint on the input that only positive values of `n` be considered, and any negative values of `n` will produce an error.

This program consists of: a sentinel control loop, to get the user input; an `if` statement, to check that the input is greater than `0`; and a counter control loop. The `if` statement is nested inside of the sentinel control block, and the counter loop is nested inside of the `if-else` statement. Now the importance of being able to structure this program using pseudo code, and to translate the pseudo code into assembly, becomes important.

The pseudo code for this algorithm follows. This pseudo code can be implemented in any HLL if the reader wants to assure themselves that it works, but it is fairly straight forward and should be easy to understand.

```
int n = prompt("Enter a value for the summation n, -1 to stop");
while (n != -1)
{
    if (n < -1)
    {
        print("Negative input is invalid");
    }
    else
    {
        int total = 0
        for (int i = 0; i < n; i++)
        {
             total = total + i;
        }
        print("The summation is " + total);
    }
    n = prompt("Enter a value for the summation n, -1 to stop");
}
```

The program to implement this pseudo code is much larger and more complex. Implementing the program without first producing the pseudo code and translating it to assembly, even for a relatively simple algorithm such as this, is difficult and often yields unfavorable results. Unless the reader has a strong understanding of structured programming, the first step ~~should be to~~for

implementing any program should be to implement it in structured code, either as pseudo code or in a HLL.

Translation of this pseudo code into assembly should following the structure of the program, as is illustrated below.

1 Begin by implementing the outer most block, the sentinel control block. Your code should look similar to the following:

```
.text
.global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    # initialize by prompting user, answer in r4
    LDR r0, =prompt
    BL printf
    LDR r0, =input
    LDR r1, =num
    BL scanf
    LDR r1, =num
    LDR r4, [r1, #0]

    StartSentinelLoop:
        MOV r0, #-1
        CMP r4, r0
        BEQ EndSentinelLoop

        # Loop Block


        # Get next value
        LDR r0, =prompt
        BL printf
        LDR r0, =input
        LDR r1, =num
        BL scanf
        LDR r1, =num
        LDR r4, [r1, #0]

        B StartSentinelLoop

    EndSentinelLoop:

.data
    prompt: .asciz "Please enter the loop limit to sum (-1 to end) \n"
    output: .asciz "The summation from 1 to n is  %d\n"
    input: .asciz "%d"
```

```
num: .word 0
```

2 The code block in the sentinel loop in the above fragment is now replaced by the `if-else` statement to check for valid input. The `if-else` block should just be the structure, and the code blocks should just be comments for now. Do not fill them in with the logic. When completed, your code should look similar to the following:

```
.text
.global main

main:
    SUB sp, sp, #4
    STR lr, [sp, #0]

    # initialize by prompting user, answer in r4
    LDR r0, =prompt
    BL printf
    LDR r0, =input
    LDR r1, =num
    BL scanf
    LDR r1, =num
    LDR r4, [r1, #0]

    StartSentinelLoop:
        MOV r0, #-1
        CMP r4, r0
        BEQ EndSentinelLoop

        # Input Check
        MOV r0, #0
        CMP r4, r0
        BLE ElseInvalid
            # If block

            B EndInputCheck

        ElseInvalid:
            # Else block

        EndInputCheck:


        # Get next value
        LDR r0, =prompt
        BL printf
        LDR r0, =input
        LDR r1, =num
        BL scanf
        LDR r1, =num
        LDR r4, [r1, #0]

        B StartSentinelLoop
```

```
    EndSentinelLoop:

.data
    prompt: .asciz "Please enter the loop limit to sum \n"
    badInput: .asciz "Your input value must be > 0 \n"
    output: .asciz "The summation from 1 to n is  %d\n"
    input: .asciz "%d"
    num: .word 0
```

3 The `if` block in the above code fragment is replaced by the summation loop from the previous program, and the `else` block is replaced by an error message about the value being invalid. This results in the following complete program. While this program is long, it is still fairly easy to follow the logic.

```
    .text
    .global main
    main:
        SUB sp, sp, #4
        STR lr, [sp, #0]

        # initialize by prompting user, answer in r4
        LDR r0, =prompt
        BL printf
        LDR r0, =input
        LDR r1, =num
        BL scanf
        LDR r1, =num
        LDR r4, [r1, #0]

        StartSentinelLoop:
            MOV r0, #-1
            CMP r4, r0
            BEQ EndSentinelLoop

            # Input Check
            MOV r0, #0
            CMP r4, r0
            BLE ElseInvalid

                # summation loop
                # initialize the loop,
                # r0 - counter
                # r4 - loop limit
                # r5 - sum
```

```
                MOV r0, #0
                MOV r5, #0

                StartCountingLoop:
                CMP r4, r0
                BLE EndCountingLoop

                    # Loop Block
                    ADD r5, r5, r0

                    # Get next value
                    ADD r0, r0, #1

                    B StartCountingLoop
                EndCountingLoop:

                LDR r0, =output
                MOV r1, r5
                BL printf

                B EndInputCheck

            ElseInvalid:

                # Print badInput message
                LDR r0, =badInput
                BL printf

            EndInputCheck:


            # Get next value
            LDR r0, =prompt
            BL printf
            LDR r0, =input
            LDR r1, =num
            BL scanf
            LDR r1, =num
            LDR r4, [r1, #0]

            B StartSentinelLoop

        EndSentinelLoop:

        MOV r0, #0
        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr
    # End main

    .data
        prompt: .asciz "Please enter the loop limit to sum \n"
        badInput: .asciz "Your input value must be > 0 \n"
```

```
output: .asciz "The summation from 1 to n is  %d\n"
input: .asciz "%d"
num: .word 0
```

Program 8.6 Machine Code and branching

In order to understand how branch addresses are calculated, it is first necessary to understand how data is stored, or more specifically, how data is aligned in memory when it is stored. When loading data from memory, the data is passed from the memory to a register value as 32 bits. However the 32 bits in memory are not any 32 bits. Remember that the ARM computer is byte addressable, so only groups of 8 bits (or a byte) can be used to specify an address.

There is another constraint on the data being passed from memory to a register, that is that the data passed from memory to a register must be word aligned[47]. For the 32-bit ARM CPU, word alignment means that memory is grouped into collections of 4-byte words, and the transfers occur on words. Thus the memory at addresses 0x0, 0x4, 0x8, 0xc, 0x10, etc., can be transferred from memory to a register. Addresses that are smaller in size than a word (e.g. a byte using ldrb or ldrsb, or half word using ldrh or ldrsh) will still access a byte, but will fixup the value so that it keeps memory correct.

Note that alignment is easy to check since all addresses are byte aligned, the address for a half word align on an even addresses, word align on an addresses divisible by 4, and double word addresses align on an address divisible by 8. Therefore only the last nybble (4 bits) of the address need to be checked to see if the address is aligned. For example, if the last nybble is 0x [0, 2, 4, 6, 8, a, c, or e] or binary 0b...0, it is half-word aligned. If the last nybble is 0x[0, 4, 8, or c] or 0b..00, it is word aligned, and if it is 0x[0,8], or 0b.000, it is double word aligned.

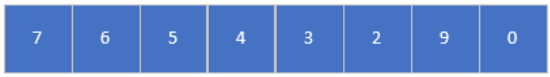
Note that all instructions for the ARM 32 bit architecture are word aligned. This means that the last two bits of all instructions are 0b00. This will have implications in the next section on calculating a branch address.

[47]Current versions of ARM do support unaligned load and store instructions, however they are translated internally to multiple world aligned instructions. There is no way to implement this otherwise because of hardware constraints, such as words crossing page boundaries. If unaligned data is accessed, multiple reads or writes are performed on the individual words. This textbook will not consider unaligned reads or writes, as they can produces unexpected results in certain program environments. They should only be used by advanced programmers who understand the implications of their use.

Chapter 8.6.1 Endianness

Since data storage is being discussed, this is a good place to discuss one of the religious wars that break out now and again in all sciences. In this case, the idea of Big-Endian and Little-Endian[48]. The terms, borrowed them from Jonathan Swift who in *Gulliver's Travels* used them to describe the opposing positions of two factions in the nation of Lilliput. One side broke their boiled eggs at the big end, rebelled against the king, who demanded that his subjects break their eggs at the little end. The story illustrates useless wars about unimportant topics.

The idea of endianness is another such unimportant discussion. First realize that all bits in a byte are numbered in the same order. The order is high order to low order byte, as shown in the following diagram:



This means that the binary string 0b001 1100 = 0x1c = 28_{10}. Note that the byte might not be a number, but could be a character, part of an address, or part of an instruction. The number represented here is just to point out the order of the bits in a bytes.

The question is what ordering to use to present bytes. For example, in a Big-Endian 4 bytes can be represented in such a way as to make the strings make sense, e.g.

This allows the string “ABCDEFGH” to appear as follows when it is viewed in memory .

[48]https://www.ling.upenn.edu/courses/Spring_2003/ling538/Lecnotes/ADfn1.htm#:~:text=He%20borrowed%20them%20from%20Jonathan,eggs%20at%20the%20little%20end.

However, if you want to see the number 2,889,974,002 = 0xAC4180F2, it would look as follows in memory (note that each byte is 2 hex digits, so this number is 4 bytes).

This strange representation for the numeric value can be fixed by using Little-Endian format, or representing bytes as follows:

This gives the correct result for numeric values:

But the string of characters is strange.

So which format is correct, Big-Endian or Little-Endian? This is really a religious argument, where both sides believe firmly they are correct, but neither side is right or wrong. The argument is inane argument because to the computer architecture it simply doesn't matter. It only matters to someone trying to read the memory, and even then the tools to look at the memory can presented as the user wants to see it.

The ARM architecture will support either format, so the choice is really arbitrary. If you work in an environment where most of the computers are Big-Endian, you would probably decide to use Big-Endian. The same with Little-Endian.

The real issue occurs when transferring data between computers. When doing data transfer, most languages provide functions that allow programs to format data into a *network format*. This is an agreed upon standard that allows data to be transferred between computer with different data storage types. Otherwise, the issue of Endianness can, and should, to left to computer bigots and care.

Chapter 8.6.2 Calculating a branch address

The PC contains the address of the instruction to execute, so branching in a CPU implies that the `PC` is changed to a new value. The question is how to calculate the new value to use for the `PC`. To do this, two types of addressing will be introduced, `PC` relative addressing and absolute addressing.

Of the two types of addressing, the concept of absolute addressing is easiest to understand. A absolute address is an absolute address in memory. This is illustrated in the following diagram, where the address of memory of the instruction to execute is 0x35fc. To branch to this address all that would be necessary is that the `PC` be set to this value (e.g. MOV pc, #0x35fc), and the code would begin executing at the new address. Absolute addressing is easy to understand, but is somewhat difficult to implement. When compiling and assembling a program, the absolute addresses of the machine instructions is not yet know. Absolute instruction address at assigned when the program is linked, and thus the calculation of the addresses must be deferred until link time. In addition, the addresses are then fixed, and the code cannot be moved after the absolute address is assigned. This make using absolute addressing problematic, and it is generally only used or functions and a few variables that are created and/or used in a separate file from the one being assembled or compiled.

If absolute addressing is not used, then how are addresses calculate? The answer is that when executing a program, the absolute address of the PC is known. If the distance from the PC to another instruction is also known, the address of the instruction to branch to can be calculated by adding that distance to the PC, and the absolute address of that statement can be calculated. This is known as *PC relative addressing*.

In PC relative addressing, the assembler or compiler can calculate the distance from any statement to any other statement in the same file. Consider the following example. In this example, the "B label_1" statement intends to branch to label_1. The branch instruction passes the distance of the branch statement to label_1 to the CPU, and the CPU calculates and branches to that instruction.

Before showing how to calculate a branch, there are important considerations for calculating branches. First, because the instructions are all 4 bytes big, each instruction is 4 bytes from the previous instruction, so you would think that the each instruction would add 4 to the distance value to branch. However, because the instructions are word aligned, the final two zeros are dropped from the distance value. This means that the distance used in the branch instruction can be obtained by counting the number of instructions between the branch statement and the instruction to be branched to. Note that this **trick** *works*, but is not representative of what is really happening.

Second, when the branch is actually calculated, the value in the PC is the PC of the branch

instruction **plus 8**. So when executing the branch, you should start counting at the instruction two ahead of the current branch instruction when calculating the branch distance. Thus in this example, the PC at the "B label_1" statement is 0x35e8, but the distance used in the branch statement is 3 (0x35e8 + 8 + 3*4 = 0x35fc).

Branching can be in a forward or reverse direction. So the "B label_2" in the previous code fragment branches backward in the program, and the distance for the branch is negative.

The same address calculations are done on the memory data in the program. Note that in this case address of var_1 is ??? from the PC, and var_2 is -4. Why var_2 is -4, even though it occurs after the statement which uses the variable, is left as an exercise at the end of the chapter.

1 Problems

1 If an instruction has both the `S` bit and condition code set in the instruction, what happens if the condition code is not met? For example, what is the result of the following code fragment? Is the branch taken or not? Explain your answer.

```
MOV r1, #5
MOV r2 #12
CMP r1, r2
SUBNES r1, r1, r2
BLT function
```

2 Write a program to prompt a user to enter numbers until a `-1` is entered. As the numbers are entered keep track of the largest, smallest, entries, and total of the values entered. At the end of the program print the value of the largest, smallest, and average for the numbers entered by the user.

3 Write a program to find prime numbers from `3` to `n` in a loop by dividing the number `n` by all numbers from `(2..n/2)` in an inner loop. Using the remainder (`rem`) operation, determine if `n` is divisible by any number. If `n` is divisible, leave the inner loop. If the limit of `(n/2)` is reached and the inner loop has not been exited, the number is prime and you should output the number. So, if the user were to enter `25`, your program would print out "`2, 3, 5, 7, 11, 13, 17, 19, 23`".

4 Write a program to prompt the user for a number, and determine if that number is prime. Your program should print out "`Number` *`n`* `is prime`" if the number is prime, and "`Number` *`n`* `is not prime`" if the number is not prime. The user should be able to enter input a "`-1`" is entered. It should print an error if `0`, `1`, `2` or any negative number other than `-1` is entered.

5 Write a program to allow a user to guess a random number generated by the computer from `1` to *`maximum`* (the user should enter the maximum value to guess). In this program the user will enter the value of *`maximum`*, and the `syscall service 42` will be used to generate a random number from `1` to `maximum`. The user will then enter guesses and the program should print out if the guess is too high or too low until the user guesses the correct number. The program should print out the number of guesses the user took.

6 Write a program to guess a number chosen by the user. In this program a user will choose a secret number from `1..`*`maximum`*. The program will prompt the user for the `maximum` value, which the user will enter. The program will then make a guess as to the value of the secret number, and prompt the user to say if the actual number is higher, lower, or correct. The computer will then guess another number until it guesses the correct secret number. The program should use a binary search to narrow its guesses to select its next guess after each attempt.

Run this program for `maximum` = {`100`, `1,000`, and `10,000`}, and graph the result. What can you say about the number of guesses used by the computer?

7 Prompt the user for a number from `3..100`, and determine the prime factors for that number. For example, `15` has prime factors `3` and `5`. `60` has prime factors `2`, `3`, and `5`. You only have to print out the prime factors, not how many times they occur (for example, in the number `60`, `2` occurs twice).

8 Change the prime factors program in question 10 to print out how many times a prime factor occurs in a number. For example, given the number `60` your program should print out "`2,2,3,5`".

9 Prompt the user for a number `n`, `0<n<100`. Print out the smallest number of coins (quarters, dimes, nickels, and pennies) which will produce `n`. For example, if the user enters "`66`", your program should print out "`2 quarters, 1 dime, 1 nickel, and 1 penny`".

10 Using only `LSL` and `RSL`, implement a program to check if a user input value is even or odd. The program should read a user input integer, and print out "`The number is even`" if the number is even, or "`The number is odd`" if the number is odd.

11 Implement integer division with rounding (not truncation) in MIPS assembly. This can be done by taking the remainder from the division and dividing the original divisor by this number. If the new quotient is greater than or equal to `1`, add `1` to the original quotient. Otherwise, do not change the original quotient.

12 Why is there a signed and unsigned load for a half-word and byte, but not a word?

13 State if the following addresses are valid for the alignment that is required? (Note: just because it might compile on your chip does not mean it is properly aligned).

14 Translate the following B and BL statements into machine code.

15 Show the following hexadecimal values in Big Endian and Little Endian notation.

16 For each of the following addresses, state if it is a valid byte, half-word, word, or double word boundary for the 32=bit arm instruction set we are using. Some may be valid for multiple types of alignment, so check all boxes representing a valid alignment.

What you will learn.
In this chapter you will learn:

1 Register conventions in ARM, and how to use the registers correctly.

2 The standard form for implementing a function.

3 How to store local variables on the stack

4 Recursive programs and implementing them in ARM assembly

Chapter 9 Function Format and Recursion

In Chapter 7 the basic use of functions was covered. However Chapter 7 did not cover the proper format and use of functions. Functions are much like procedural program structures from Chapter 8 in that there are certain recommended conventions that programmers should follow. Programs that follow these forms are nearly always cleaner, easier to implement and maintain, and tend to be more correct than programs that do not.

This chapter will be about how to implement a function using these standard formats. It will rely on the Procedure Call Standard for Arm Architecture (AAPCS[49]) for much of this information. The conventions and overall structure of a function will follow a standard, and implementation of functions to this standard will result in functions that are easier and more correct than ones that try to implement functions using adhoc structures.

To illustrate why standard function formats are better than adhoc methods, the chapter will implement functions using recursion. Recursion is thought by many readers to be hard even in a HLL, and thus to be insurmountable in assembly. However by using standard function formats the readers will be shown how recursion is as easily implemented in assembly as in any HLL, and possibly easier to understand because the mechanism of using the program stack to implement recursion will be transparent. Examples will also be given of how even small deviation from the standard format can lead to erroneous programs.

Chapter 9.1 What is Recursion?

Recursion is a programming construct that solves a problem by breaking it into successively smaller problems, each of the smaller problems being implemented in a function that has the same functionality as the original problem, but is operating on a smaller universe of values. The reducing of the problem continues until some base or ending case is reached. The results of the base case are then returned to each of the functions, which compute results, and these results are then returned to the calling function.

[49]It is called the AAPCS, not the PCSAA, for historical reasons

One of the issues with teaching recursion is that it is not that useful for small problems that can be easily presented to students, and these small problems can be solved using procedural programming more easily and quickly than using recursion, which leads students to believe that recursion is not actually useful. However there are many problems that cannot be solved using procedural programming without the use of a stack, and those problems are almost always more easily solved using recursion rather than procedural programming. These types of problems will be examined in the problems at the end of the chapter. But for now, the principals of recursion will be presented, with the hope that the reader will understand the basic concept of recursion, and be able to apply it to more real world problems later.

The problem that will be presented to illustrate recursion is the summation problem. The problem is to sum all the numbers from 1 to n, and print out the result. So for example, if the user enters 5 into the program, the program will add 0+1+2+3+4+5 = 15. Mathematically this function can be specified as:

$$\sum_{i=0}^{n} i$$

This can be rewritten mathematically as:

```
f(n) = 0 when n == 0
       else f(n) = f(n-1) + n
```

This corresponds to the following Java method:

```
public static int sum(int n) {
    if (n == 0) return 0;
    else return sum(n-1) + n;
}
```

The rest of this section will translate this method into an assembly language program.

To begin creating the assembly function first create the structure for the function. Thus the first step is to define push and pop sections of the function. First for the push, the value of n is passed into the function using r0. Since r0 is needed in the function, convention holds that it must be saved, and so it is moved into r4. Since r4 is a preserved register, its original value is saved to the stack when the function is first entered, and restored when the function returns.

Next the standard block structure of the function must be maintained. Thus the return label is placed just before the pop. All paths that exit this function should use this return label to make sure the stack is correctly popped.

The format for the Summation function can form a template for implementing any recursive function. Initially looks as follows:

```
Summation:
   #push stack.  Save r0 (summation) in r4,
   # so r4 has to be saved by callee convention
   sub sp, sp, #8
   str lr, [sp, #0]
   str r4, [sp, #4]
   mov r4, r0

# pop stack and return
   return:
   ldr lr, [sp, #0]
   ldr r10, [sp, #4]
   add sp, sp, #8
   mov pc, lr
# END Summation
```

The next step in writing the code is filling in the code for the base and recursive conditions. First the base condition is `"if (n ==0) return 0"`. This is translated into the following assembly language code fragment:

```
MOV r1, #0
CMP r0, r4
 BNE recurse
    MOV r0 #0
    B return
```

The code for the recursive condition is `"else return sum(n-1) + n"`. This is translated into the following assembly language code fragment:

```
recurse:
    SUB r0, r4, #1
    BL sum
    ADD r0, r0, r4
    B return
```

Putting these pieces together, the following function is created. A main method is added to complete the program, and a working program to calculate a sum recursively is defined.

```
.text
   .global main

main:
# Save return to os on stack
   sub sp, sp, #4
   str lr, [sp, #0]

   mov r0, #10
   bl Summation
   mov r1, r0
   ldr r0, =output
   bl printf


# Return to the OS
   ldr lr, [sp, #0]
   add sp, sp, #4
   mov pc, lr

.data
   output: .asciz "Summation is %d\n"


.text
# Note: Summation is NOT a global symbol!
# It is a static function
Summation:
   #push stack.  Save r0 (summation) in r10,
   # so r10 has to be saved by callee convention
   sub sp, sp, #8
   str lr, [sp, #0]
```

```
    str r4, [sp, #4]
    mov r4, r0

    # if r0 is 0, return 0
    mov r1, #0
    cmp r1, r4
    beq return    @ return 0 in r0

    add r0, r4, #-1
    bl Summation  @ return value in r0
    add r0, r4, r0 @ return summation in r0
    b return  @ not really needed

# pop stack and return
    return:
    ldr lr, [sp, #0]
    ldr r4, [sp, #4]
    add sp, sp, #8
    mov pc, lr
#END Summation
```

This version of the program is basically equivalent to the Java version of this program that was implemented earlier. There really is nothing mystical or magical about implementing recursion in assembly language if recursion is understood at the HLL and the standards are followed. In fact the assembly language version should be more clear as the details of how the stack is used to implement assembly are fully transparent.

Difficulty with recursion in assembly arises when programmers decide they *know better* how to implement a program, and thus can dispense with the standard format for programming in assembly. These programs become hopelessly complex, and as will be seen in the next section, can result in erroneous programs that might compute the correct answer, but are simply wrong.

Chapter 9.2 An Erroneous implementation of Recursion

The example program in this section is the same summation problem and sum function from the previous section. However now a *very intelligent* programmer notices that the values of r0 and lr has not been modified when the base condition is called. This programmer decides that there is thus no need to branch to the return label in the program, thus saving a few instructions. This programmer now feels good that they have made the program faster, and

see it as a job well done.

```
.text
    .global main

main:
# Save return to os on stack
    sub sp, sp, #4
    str lr, [sp, #0]

    mov r0, #10
    bl Summation
    mov r1, r0
    ldr r0, =output
    bl printf


# Return to the OS
    ldr lr, [sp, #0]
    add sp, sp, #4
    mov pc, lr

.data
    output: .asciz "Summation is %d\n"


.text
# Note: Summation is NOT a global symbol!
# It is a static function
Summation:
    #push stack.  Save r0 (summation) in r10,
    # so r10 has to be saved by callee convention
    sub sp, sp, #8
    str lr, [sp, #0]
    str r4, [sp, #4]
    mov r4, r0

    # if r0 is 0, return 0
    mov r1, #0
    cmp r1, r4
    moveq pc, lr    @ return 0 in r0

    add r0, r4, #-1
    bl Summation  @ return value in r0
```

```
    add r0, r4, r0 @ return summation in r0
    b return  @ not really needed

# pop stack and return
    return:
    ldr lr, [sp, #0]
    ldr r4, [sp, #4]
    add sp, sp, #8
    mov pc, lr
#END Summation
```

There are two problems with this analysis. First, the program looks like it is a few instructions faster, but the amount is speed up is beyond the ability of the computer to measured, and many orders of magnitude smaller than anything on a human time scale. But worse, the programmer unintentionally *added* more executable statements to the program, and does not even realize it.

Second, the program has made the program erroneous. When the program is tested, it produces the correct result, so the programmer believes it is correct. But the program produces this result in a way that the programmer making the changes does not understand, and a subtle incorrect behavior has been added.

To see this incorrect behavior, consider an even more intelligent programmer who realizes the sum from 0...n is the same as 1...n, and so the base condition can be changed from 0 to 1 to save a few clock cycles. This is shown in the following program.

```
.text
    .global main

main:
# Save return to os on stack
    sub sp, sp, #4
    str lr, [sp, #0]

    mov r0, #10
    bl Summation
    mov r1, r0
```

```
    ldr r0, =output
    bl printf


# Return to the OS
    ldr lr, [sp, #0]
    add sp, sp, #4
    mov pc, lr

.data
    output: .asciz "Summation is %d\n"


.text
# Note: Summation is NOT a global symbol!
# It is a static function
Summation:
    #push stack.  Save r0 (summation) in r10,
    # so r10 has to be saved by callee convention
    sub sp, sp, #8
    str lr, [sp, #0]
    str r4, [sp, #4]
    mov r4, r0

    # if r0 is 0, return 0
    mov r1, #1
    cmp r1, r4
    moveq pc, lr    @ return 0 in r0

    add r0, r4, #-1
    bl Summation  @ return value in r0
    add r0, r4, r0 @ return summation in r0
    b return  @ not really needed

# pop stack and return
    return:
    ldr lr, [sp, #0]
    ldr r4, [sp, #4]
    add sp, sp, #8
    mov pc, lr
#END Summation
```

The problem now is that there is an off-by-one bug. The value returned from the summation

program is one too large. So did the new programmer create the off-by-one condition? NO! The condition was created by the first programmer not following the standards for writing recursion. The second programmer simply allowed the presence of the bug to become apparent.

Why? When the first programmer returned without branching to return, that programmer did not pop the stack. This left the stack record with the value n=0 on the stack. Thus when the program returned a 0, the n=0 stack record was unintensionally not popped. The next return was the result of 0+0, or just 0, which is the correct answer, but that answer was achieved incorrectly.

When the second programmer now made the base condition 1, the stack record with n=1 was again incorrectly popped. However now the result is 1+1, which is incorrect. The program modified by the first programmer was always incorrect, but it was not until the modification was introduced that the problem became apparent.

This illustrates an important point for the reader to realize; the bug was introduced in the first modification, not the second modification. The fact that the program *worked* (or rather, produced the correct result) does not change the fact that the modification made the program erroneous (or in this case, simply wrong).

This program also illustrates why programmers often think recursion is difficult to implement in general, and even more so in assembly language. As has been stressed in this textbook from the beginning, programmers are taught or in some fashion come to believe that programs are products of logic, and implementing correct logic produces good programs. When implementing the recursion using a non-standard technique, the results are even more convoluted in assembly, where the possibility of unrestricted branching allows programmers imaginations to run amok. In the long run this produces spaghetti like logic even the person implementing the program loses track of what is actually being done[50].

The truth is good structure produces good (and correct) programs. Implementing logic in a standard format is the path to creating good programs. Starting with a good structure will lead to more simple and correct solutions, and that is true in whatever language the programmer uses.

---

[50]One of my favorite questions from students is "Can you tell me what my program is doing?". This question tells me that the student was so busy implementing something, that they lost track of what they wanted the program to do, and accept that the program must be doing something correct that they can modify to get to the correct answer. A program is always doing what the programmer tells it to do. It is up to the programmer to make sure what it is doing is what is intended.

Chapter 9.3 Problems

1. Comment on the following: “If a program produces a correct result, it is obviously correct”. What does it mean to have a correct program?
2. Present you understanding of the term *erroneous*. Do you think it should be applied to programs?
3. Implement a multiply function using only addition and recursion.
4. Implement a program to reverse the digits in an integer number. For example, for the number 5271, change the integer value to 1725.
5. Implement a recursive program that takes in a number and finds the square of that number through addition. For example if the number 3 is entered, you would add 3+3+3=9. If 4 is entered you would add 4+4+4+4=16. This program must be implemented using recursion to add the numbers together.
6. Write a recursive program to calculate Fibonacci numbers. Use the definition of a Fibonacci number where F(n) = F(n-1) + F(n-2), F(0) = F(1) = 0.
7. Write a recursive program to calculate factorial numbers. Use the definition of factorial as F(n) = n * F(n-1).
8. Combinations, Permutations

In this chapter you will learn:

1 The definition of an array, and how to implement and access array elements using index.

2 What is heap memory, how to allocate it, and how to use it.

3 Null terminated arrays, and array processing using element pointers

4 How to allocate an array in stack memory, on the program stack, or in heap memory, and why arrays are most commonly allocated on heap memory.

5 How to use array addresses to access and print elements in an array.

6 Call by Value, Call by Reference, and Call by Reference Value (or Reference Type)

7 An algorithm to reverse an array, and how to it in ARM assembly.

Chapter 10 Arrays

In HLLs, there is a concept of a multi-valued variables. These multi-valued variables are a single variable that contains other variables. These multi-valued variables include structures such as an array, a structure or a class, or most common data structures such as stacks, queues, heaps, etc.

An array is a generally builtin into a language, and contains multiple values of the same type for primitive values, or extend or implement a base class or interface for objects. The dereference operator is a pair a square brackets, "[]", and for the purposes here the values are referenced by an index specifying a distance from the base of the array[51].

A structure or class is generally built into a language, contains a fixed number of other variables, all of which can have different types. The dereference operator is a ".", and the values are named variables that are contained in that structure or class.

A data structure is generally not built into a language, and contains an indeterminate number of variables. Because the data structures are generally built as programs in a given language, there is a lot of variation on how they behave. The access to the contained variables is generally implemented using function and the "." dereference operator, but this can vary widely depending of the language. For most languages, all the variables contained extend or implement a base class or interface for objects, though some HLL like C/C++ allow the data structure to contain primitive variables. Finally the rules for how to store and access the data

[51]There are exceptions this definition of an array, as languages such as JavaScript or Python implement what appear to be arrays using an associative array (or hash map) where the index can be 0, 1, 2, etc, is a key to a hash table.

vary widely across different types of data structures.

This chapter will only deal with arrays which will be restricted to implementation containing primitive data values and use standard base + offset addressing.

Chapter 11.0 Array definition and access

Because arrays can mean so many different things in HLL, this chapter will give a more precise definition of array that is sufficient to allow its implementation in ARM assembly.

The definition of an array as used in this text is "A multi-valued variable where all the values are the same size, and are contiguous in memory". To understand this definition each part of it will be detailed. First the concept of a multi-valued variable is that a single program variable contains more than one value. For example, the Java statement "`int a[] = new int[10];`" creates a Java variable "`a`" that contains `10 int` vales.

The phrase "all the values are the same size" is used, as opposed to "all the values are of the same type". For primitives these two statements are effectively the same in a HLL. However for the Java statement "Object o[] = new Object(10)", the type of the array is Object, but the values stored can be any type of Object. The statement "o[0] = new Integer()" is valid because the array stores object references (or addresses), which are the same for all types of objects. Thus the array stores objects of different types, but the sizes are consistent.

The term contiguous means touching or connected throughout in an unbroken sequence. This means that the array must be a single allocation in memory, and the values cannot be spread out through memory.

To see how this definition is used, consider the following code fragment:

```
int arr = new int[10]
arr[3] = 5;
```

In this code fragment a single block of 320 bytes are allocated starting at a base address (`baddr`) in memory (for example, `baddr = 0x2350`). The address of an element in the array is calculated by the formula:

```
address = baddr + index * size
```

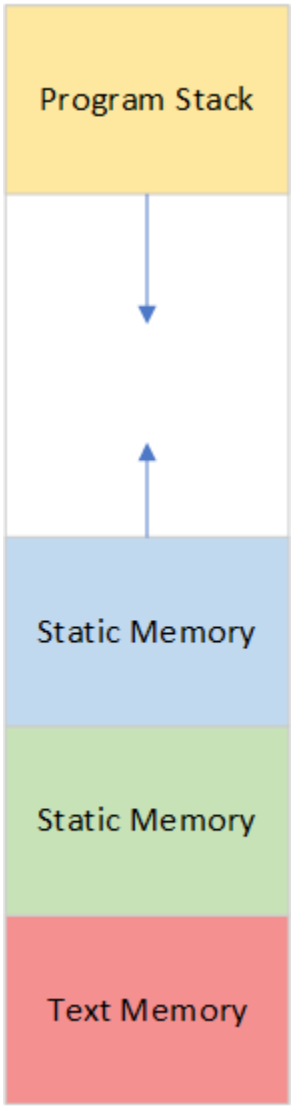
In this formula, the base address would be `0x2350`, the index is 3, and the size of an integer is 4 bytes. Thus the address of the `arr[3]` is `0x2350` + 3 * 4 = `0x235C`, and `M[0x2350] = 5`.

Chapter 12.0 Program Memory

Unless the size of the array is known when the program is compiled, memory for arrays is allocated in heap memory. This means that most arrays are stored in heap memory. In languages such as Java, there is no other option, all arrays are stored in heap memory. To

understand why this is true, the reader needs to know a little about memory management in a program.

From a program point of view, on most computers there are 4 areas of memory: text, where executable programs instructions are stored; static memory[52], where data that is defined when compiling or assembling the program, and thus occurs once for the program is stored; program stack memory, where one set of data (often called *automatic data*) is stored for each function or method invocation, and heap memory. A generic view of a user memory process space is represented in the following diagram.



[52]Static memory can further be broken down into a Block Symbol Storage (BSS) and data segment. The BSS segment contains static data that is not initialized, and thus initialized by default to zero. The data segment contains data that is initialized. But for the purposes here, both are called static data.

In this memory model, the text memory (instructions) and static memory (variables defined at compile of assembly time) are constrained, and thus cannot have their size changed at run time. The stack is dynamic and new memory is allocated at run time, the records on the stack must be a size that is fixed at compile or assemble time to work correctly, so the size of any stack allocated on an array would also have a fixed size that cannot be changed at run time. This leaves only heap memory where variable size arrays (or arrays that the size can be set and changed at run time) can be allocated.

Heap memory is a managed area of memory. Memory is allocated and deallocated as needed. This allocation and deallocation can either be done explicitly using function calls (malloc, calloc, and free) or using language operators such as new (C++, Java) and delete (C++, Java has automatic garbage collection).

To present arrays, the first array example in section 10.3 is a fixed sized array of integer values that is be allocated in static (or data) memory. The program will print the elements in this array. This was chosen as the first example because the array can be (must be) allocated as part of the program, and the values to the array assigned when the program is written.

The second example shows the use of a null terminated character array to allocated on the stack. It will show an example of using a toUpper function to convert this string to all uppercase characters, and then print it out. It will iterate until a null value is found in the array

This section will present arrays using heap access with malloc and free function calls. To allocate an array, malloc is called with a parameter that specifies the number of bytes to allocate, and returns a pointer to a block of memory that is properly aligned for any built-in type, and at least the size requested. When using malloc the memory is not initialized, so the memory should be initialized before being used. If the programmer wants memory initialized to zero, the calloc call should be used.

Chapter 13.0 Processing an Array Using an Index

This first program to use static memory to create of integers, initialize the values in an array, and then create a function, printArrayByIndex, which will walk through the array and print each value on a separate line.

```
      .global printArrayByIndex
      .global main

      .text
      printArrayByIndex:
          #push stack
          SUB sp, sp, #16
          STR lr, [sp, #0]
          STR r4, [sp, #4]
```

```
        STR r5, [sp, #8]
        STR r6, [sp, #12]

        # Save Base Array and Size to preserved registers
        MOV r4, r0
        MOV r5, r1

        # initialize loop for entering data
        # r4 - array base
        # r5 - end loop index
        # r6 - loop index

        MOV r6, #0
        startPrintLoop:
            CMP r6, r5
            BGE endPrintLoop

            LDR r0, =output
            MOV r1, r6
            ADD r2, r4, r6, lsl #2
            LDR r2, [r2, #0]
            BL  printf

            ADD r6,r6, #1
            B startPrintLoop
        endPrintLoop:

        #pop stack
        LDR lr, [sp, #0]
        LDR r4, [sp, #4]
        LDR r5, [sp, #8]
        LDR r6, [sp, #12]
        ADD sp, sp, #16
        MOV pc, lr

    .data
        output: .asciz "The value for element [%d] is %d\n"

    #end printArrayByIndex

    # Main procedure to test printArrayByIndex
    .text
    main:
        #push stack
        SUB sp, sp, #4
        STR lr, [sp, #0]
```

```
        LDR r0, =myArray
        LDR r1, =arrSize
        LDR r1, [r1]
        BL  printArrayByIndex

        #pop stack
        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr

    .data
        myArray: .word 55
                 .word 21
                 .word 78
                 .word 19
        arrSize: .word 4
```

Chapter 11.3.0 Comments on Program

The following list explains how this program works.

1. The following lines create and initialize the array in static (or data) memory. Because the list is in static memory, the values in the list can be changed, but the size of the list cannot be changed. Hence the list is always 4 ints (or 16 bytes) in size.

```
myArray: .word 55
         .word 21
         .word 78
         .word 19
arrSize: .word 4
```

2. The array base and array size are passed into the program using the registers r0 and r1. Because these registers are often changed, as when printf is called, the values need to saved either on the stack or in a preserved register. Here the values are copied to r4 and r5. Since r4 and r5 are now being used, these registers must be saved to the stack in the push and copied back from the stack on the pop.
3. A loop counter variable is created in r6, which means that r6 must be saved on the stack.
4. The loop index in r6 counts the number of iterations from 0...arrSize. Therefore it can be use to calculate the address of the each element in the array. This is accomplished by multiplying the index by 4 (each int is 4 bytes in size) and adding that amount to the base address. This is done in the following formula.

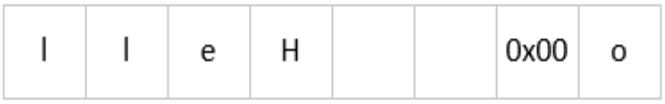
```
ADD r2, r4, r6, lsl #2
```

In this formula, the multiplication by 4 is performed by left logical shift by 2 bits.

5. The value of the array index and element are then printed from r1 and r2.

Chapter 14.0 Processing a Character Array On the Stack

The next example processes an array of characters that are allocated on the stack. In C and many lower level languages, a string is often represented as an array of characters, each character being one byte, that is terminated with a null value (a byte containing 0x00). This is often called a null terminated string. The string "Hello" would be stored in memory, in little endian format, as follows:



The main program allocates 40 bytes on the stack for the array, and calls the printStringByIndex function to print each character in the string.

```
.global printStringByIndex
.global main

.text
printStringByIndex:
    #push stack
    SUB sp, sp, #12
    STR lr, [sp, #0]
    STR r4, [sp, #4]
    STR r5, [sp, #8]

    # Save Base Array to preserved register
    MOV r4, r0

    # initialize loop for entering data
    # r4 - array base
    # r5 - loop index

    MOV r5, #0
    startPrintLoop:
        MOV r0, #0
        LDRB r1, [r4, r5]
        CMP r0, r1
        BEQ endPrintLoop

        LDR r0, =output
        MOV r1, r5
```

```
            ADD r2, r4, r5  // Calculate the array address
            LDRB r2, [r2, #0]
            BL  printf

            ADD r5,r5, #1
            B startPrintLoop
        endPrintLoop:

        #pop stack
        LDR lr, [sp, #0]
        LDR r4, [sp, #4]
        LDR r5, [sp, #8]
        ADD sp, sp, #12
        MOV pc, lr

    .data
        output: .asciz "The value for element [%d] is %c\n"

    #end printStringByIndex

    # Main procedure to test printArrayByIndex
    .text
    main:
        #push stack
        SUB sp, sp, #44
        STR lr, [sp, #0]
        # string is at sp+4 ... s+43

        # load string
        LDR r0, =prompt
        BL printf
        LDR r0, =format
        ADD r1, sp, #4
        BL scanf

        # reload base and call function
        ADD r0, sp, #4
        BL  printStringByIndex

        #pop stack
        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr

    .data
        prompt: .asciz "Enter input string: "
        format: .asciz "%s"
```

Chapter 11.4.0 Comments on program

The first comment on this program is illustrated in the following code that shows how the space for the string is allocated. In these lines it can be seen that the stack pointer is moved to allow enough space for 40 bytes to be saved on the stack for the string. Note that even though this memory is dynamically allocate (e.g. allocated at runtime), the amount of spaces must be specified at compile time to allow the push and pop for the stack to be implemented. Thus like the static array allocation, stack allocations must be specified at compile time and cannot be changed.

```
#push stack
SUB sp, sp, #44
STR lr, [sp, #0]
# string is at sp+4 ... s+43
```

The next two points are illustrated in the loop that processes the string.

```
MOV r5, #0
startPrintLoop:
    MOV r0, #0
    LDRB r1, [r4, r5]
    CMP r0, r1
    BEQ endPrintLoop

    LDR r0, =output
    MOV r1, r5
    ADD r2, r4, r5  // Calculate the array address
    LDRB r2, [r2, #0]
    BL  printf

    ADD r5,r5, #1
    B startPrintLoop
endPrintLoop:
```

The first point is that this loop reads each character as a byte using the LDRB (load byte) instruction. Each byte that is loaded corresponds to one character. Because bytes are being loaded, the address of the item is not multiplied by 4 as was done for word addressing, as shown in the code fragment below:

```
    ADD r2, r4, r5  // Calculate the array address
    LDRB r2, [r2, #0]
```

Finally, this loop does not know how long the string is, but it does know that the string ends with a NULL character. Therefore this string reads each byte (character) in the loop until a NULL

valid is found, and then exits the loop, as shown in the code fragment below:

```
startPrintLoop:
    MOV r0, #0
    LDRB r1, [r4, r5]
    CMP r0, r1
    BEQ endPrintLoop
```

Finally, note that the printf command uses a value for the character to print. This is the same as printing an integer value, but different than printing a string, where a pointer to the string is used.

Chapter 15.0 Processing a (String) Character Array using Pointers

The following example is just like the previous example, but now the array of characters that make up the string are allocated on the heap. Because heap allocation can be set at runtime, the size of the memory that can be used can be changed at runtime.

In the following example, a toUpper function is written that converts a string of characters to all uppercase. Note that to keep the function simple, only character data should be passed to this function, and there is no check for valid input. As a result of running this function, all lowercase characters are converted to uppercase, and uppercase characters are not affected. The string as all uppercase is returned.

```
.global toUpper
.global main
.text

toUpper:
    #push stack
    SUB sp, sp, #12
    STR lr, [sp, #0]
    STR r4, [sp, #4]
    STR r5, [sp, #8]

    # Save Base Array and Size to preserved registers
    MOV r4, r0

    # initialize loop for entering data
    # r4 - element address
    # r5 - constant null
    MOV r5, #0

    startLoop:
        LDRB r1, [r4, #0]
        CMP r1, r5
        BEQ endLoop
```

```
            AND r1, r1, #0xdf
            STRB r1, [r4], #1
            B startLoop
        endLoop:

        #pop stack
        LDR lr, [sp, #0]
        LDR r4, [sp, #4]
        LDR r5, [sp, #8]
        ADD sp, sp, #12
        MOV pc, lr

    .data

    #end printArrayByIndex

    # Main procedure to test printArrayByIndex
    .text
    main:
        #push stack
        SUB sp, sp, #4
        STR lr, [sp, #0]

        # load string
        LDR r0, =prompt
        BL printf
        MOV r0, #40
        BL  malloc
        MOV r5, r0
        MOV r1, r0
        LDR r0, =format
        BL scanf

        MOV r0, r5
        BL toUpper

        # reload base and call function
        LDR r0, =output
        MOV r1, r5
        BL  printf

        #pop stack
        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr

    .data
```

```
prompt: .asciz "Enter input string: "
output: .asciz "\nYour string is %s\n"
format: .asciz "%s"
```

Chapter 11.5.0 Comments on Program

This program illustrates two new concepts. The first is the use of malloc to allocate the memory for the string. The malloc function allocates memory on the heap. Heap memory is managed at runtime, so the amount of memory that is allocated can be specified at runtime. This allows variable size data structures, such as arrays and strings, to be created in programs, and thus most arrays and strings are allocated in heap memory. The malloc function takes in a parameter that is the size in r0, and returns a pointer to memory of that size in r0.

The second new concept is the use of a pointer, and not an index, to walk through the array. This is shown in the following code fragment:

```
MOV r5, #0

 startLoop:
     LDRB r1, [r4, #0]
     CMP r1, r5
     BEQ endLoop
     AND r1, r1, #0xdf
     STRB r1, [r4], #1
     B startLoop
 endLoop:
```

In this code fragment, r4 starts by pointing to the beginning of the string. Each time through the loop the value in r4 is updated to point to the next character in the string using the STRB with post incrementing, e.g. `STRB r1, [r4], #1`. When the character at the pointer address in r4 is the character 0, the end of the string is reached, and the loop halts.

Chapter 16.0 Call By Reference and Call By Reference Variable

Most basic programming classes cover the concept of Call By Value and Call By Reference. The concept is that a when calling a function, a parameter can be copied when the function is called (Call By Value) or the parameter can be a reference to the variable (Call By Reference). When Call By Value is used, the parameter cannot be changed, whereas when a Call By Reference is used, the values in the variables (which is always a multi-valued variable) can be changed. The following simple Java program to reversed an array illustrates this implementation of call by reference, where a multi-valued reference variable, an array, is passed to a function with two

index offsets. The values in the array is then swapped.

```
        .text
        .global main

    SwapByRefType:
        SUB sp, sp, #4
        STR lr, [sp, #0]

        ADD r1, r0, r1, lsl #2
        ADD r2, r0, r2, lsl #2

        LDR r0, [r1, #0]
        LDR r3, [r2, #0]
        STR r0, [r2, #0]
        STR r3, [r1, #0]

    # pop stack and return
        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr

    .data
    #END SwapByRefType

    main:
    # Save return to os on stack
        SUB sp, sp, #16
        STR lr, [sp, #0]
        STR r4, [sp, #4]
        STR r5, [sp, #8]
        STR r6, [sp, #12]

        # Call PrintArray with the ar
        LDR r0, =output
        BL printf
        LDR r0, =ar
        LDR r1, =size
        LDR r1, [r1, #0]
        BL printArrayByIndex
        LDR r0, =newline
        BL printf
```

```
# Reverse Array using SwapByRef
    #initialize loop
    LDR r4, =ar    @ r4 is the base of the array
    LDR r7, =size
    LDR r7, [r7, #0]
    SUB r7, r7, #1 @ -1 for array index
    ASR r5, r7, #1 @ r5 is the loop limit, or or size by 2
    MOV r6, #0     @ r6 is counter

    startMoveLoop:
    # Check end condition
    CMP r6, r5
    BGE endMoveLoop

        MOV r0, r4
        MOV r1, r6
        SUB r2, r7, r6
        BL SwapByRefType

    # next iteration
    ADD r6, r6, #1
    b startMoveLoop
    endMoveLoop:

    # Call PrintArray with the ar
    LDR r0, =output
    BL printf
    LDR r0, =ar
    MOV r1, #5
    BL printArrayByIndex
    LDR r0, =newline
    BL printf

# Return to the OS
    LDR lr, [sp, #0]
    LDR r4, [sp, #4]
    LDR r5, [sp, #8]
    LDR r6, [sp, #12]
    ADD sp, sp, #16
    MOV pc, lr

.data
    output: .asciz "The array is : \n"
    newline: .asciz "\n"
# the variable ar id an array of 5 elments
    size: .word 5
```

```
        ar: .word 15
            .word 1
            .word 27
            .word 9
            .word 16
    #END main
```

In this Java program the array and both indices are copied into parameters before the call to the method, but the array is a reference to other variables. This is not a true Call By Reference, but rather a call using a Reference Variable. In many hll, such as Java, there does not exist a true Call By Reference, as it is hard to ensure program correctness and safety with true Call By Reference[53]. In these languages, all parameters are copied when a function is called and are thus Call By Value. The difference between these two implementation of parameter passing can best be illustrated in assembly, and are implemented in the following two sections.

Chapter 11.6.0 Call by Reference Variable

This first example is the equivalent of the previous Java program. In this program the address of the start of the array is passed to the swap method, along with the two index offset values. The references to the two variables in the array to be swapped are calculated, and the values are swapped. Note that the references to the values to be swapped were calculated in this function.

Chapter 12.6.0 Call by Reference

In this example, only two parameters are passed to the function. These two parameters are the actual references, or addresses, for the variables to be swapped. The swapping of these two variables are visible in the calling program since the actual references, and not copies of the variables, are passed to the function. This is not possible to accomplish in a language such as Java. This is a true implementation of Call By Reference, not a Call By Reference Variable.

```
        .text
        .global main

    SwapByRef:

        SUB sp, sp, #4
        STR lr, [sp, #0]

        LDR r2, [r0, #0]
        LDR r3, [r1, #0]
```

[53]C# is an example of a language that safely implements a true Call By Reference, but in most languages Call By Reference is inherently problematic and is hard to use safely. C# uses the terminology *Call by Reference Type*., but this text will use the more accurate (IMHO) *Call by Reference Variable*.

```
        STR r2, [r1, #0]
        STR r3, [r0, #0]

        LDR lr, [sp, #0]
        ADD sp, sp, #4
        MOV pc, lr

    .data
    #END SwapByRef

    main:
    # Save return to os on stack
        SUB sp, sp, #16
        STR lr, [sp, #0]
        STR r4, [sp, #4]
        STR r5, [sp, #8]
        STR r6, [sp, #12]

        # Call PrintArray with the ar
        LDR r0, =output
        BL printf
        LDR r0, =ar
        LDR r1, =size
        LDR r1, [r1, #0]
        BL printArrayByIndex
        LDR r0, =newline
        BL printf

    # Reverse Array using SwapByRef
        #initialize loop
        LDR r4, =ar    @ r4 is the base of the array
        LDR r7, =size
        LDR r7, [r7, #0]
        SUB r7, r7, #1 @ -1 for array index
        ASR r5, r7, #1 @ r5 is the loop limit, or or size by 2
        MOV r6, #0     @ r6 is counter

        startMoveLoop:
        # Check end condition
        CMP r6, r5
        BGE endMoveLoop

            ADD r0, r4, r6, lsl #2
            SUB r3, r7, r6
            ADD r1, r4, r3, lsl #2
            BL SwapByRef
```

```
        # next iteration
        ADD r6, r6, #1
        b startMoveLoop
        endMoveLoop:

        # Call PrintArray with the ar
        LDR r0, =output
        BL printf
        LDR r0, =ar
        MOV r1, #5
        BL printArrayByIndex
        LDR r0, =newline
        BL printf

    # Return to the OS
        LDR lr, [sp, #0]
        LDR r4, [sp, #4]
        LDR r5, [sp, #8]
        LDR r6, [sp, #12]
        ADD sp, sp, #16
        MOV pc, lr

    .data
        output: .asciz "The array is : \n"
        newline: .asciz "\n"
    # the variable ar id an array of 5 elements
        size: .word 5
        ar: .word 15
            .word 1
            .word 27
            .word 9
            .word 16
    #END main
```

Chapter 17.0 Exercises

1 Change the PrintIntArray subprogram so that it prints the array from the last element to the first element.

2 The following pseudo code converts an input value of a single decimal number from $1 \leq n \geq 15$ into a single hexadecimal digit. Translate this pseudo code into MIPS assembly.

```
main
{
    String a[16]
```

```
    a[0] = "0x0"
    a[1] = "0x1"
    a[2] = "0x2"
    a[3] = "0x3"
    a[4] = "0x4"
    a[5] = "0x5"
    a[6] = "0x6"
    a[7] = "0x7"
    a[8] = "0x8"
    a[9] = "0x9"
    a[10] = "0xa"
    a[11] = "0xb"
    a[12] = "0xc"
    a[13] = "0xd"
    a[14] = "0xe"
    a[15] = "0xf"

    int i = prompt("Enter a number from 0 to 15 ")
    print("your number is " + a[i]
}
```

3 The following method returns a random number from 1 to n, where n is stored in r1.

```
#  Calculate a random number
#  arguments:     r0 - seed  (if seed is 0, get next random value)
#                 r1 - range (from 1 to r1).  If r1 is 0 or negative,
#                range is all ints)
#
Random:
    SUB sp, sp, #8
    # Save return to os on stack
    STR lr, [sp, #0] @ Prompt For An Input
    STR r4, [sp, #4]

#
    MOV r3, #0
    CMP r0, r3
    BNE Reset
      LDR r0, =seed     @ get the seed
      LDR r0, [r0, #0]
    Reset:

    ADD r0, r0, #137    @ get the next seed
    EOR r0, r0, r0, ror #13
    LSR r0, r0, #1      @ make sure it is positive
    MOV r4, r0          @ save the value to r4

# Get the remainder
    MOV r3, #0
    CMP r1, r3
    BLE NoRange
      BL __aeabi_idiv
      MUL r1, r0, r1
```

```
        SUB r4, r4, r1
    NoRange:

# Save the seed to memory
    LDR r0,=seed
    STR r4,[r0, #0]

# Return to the OS
    MOV r0, r4
    LDR lr, [sp, #0]
    LDR r4, [sp, #4]
    ADD sp, sp, #8
    MOV pc, lr

.data
    seed:     .word 25
#end Random
```

a) Create an array of 100 values, and populate it with 100 random numbers.
b) Using the array in part a, find the minimum and maximum value in the array.
c) Calculate the sum and average of all values in the array
d) Implement a Bubble Sort of the array.
e) Find the median value in the array.

4 Create strings of numbers (0+1, Combinations, permutations; using String functions)

5 Reverse a string using procedural programming and recursion.

6 The following pseudo code programs calculates the Fibonacci numbers from 1..n, and stores them in an array. Translate this pseudo code into MIPS assembly, and use the PrintIntArray subprogram to print the results.

```
main
{
    int size = PromptInt("Enter a max Fibonacci number to calc: ")
    int Fibonacci[size]
    Fibonacci[0] = 0
    Fibonacci[1] = 1
    for (int i = 2; i < size; i++)
    {
        Fibonacci[i] = Fibonacci[i-1] + Fibonacci[i-2]
    }
    PrintIntArray(Fibonacci, size)
```

Chapter 2.11 Exceptions

TDB

Glossary