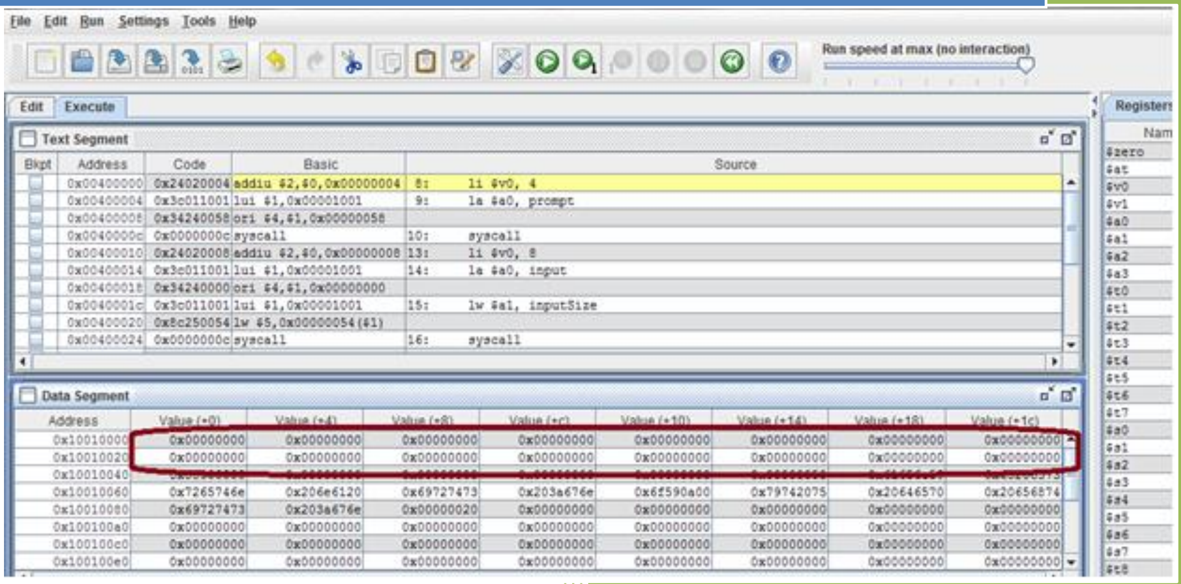
Introduction To MIPS Assembly Language Programming

Charles W. Kann
Gettysburg College

Introduction to MIPS Assembly Language Programming



```
Bkpt  Address     Code        Basic                          Source
      0x00400000  0x24020004  addiu $2,$0,0x00000004     8:      li $v0, 4
      0x00400004  0x3c011001  lui $1,0x00001001          9:      la $a0, prompt
      0x00400008  0x34240058  ori $4,$1,0x00000058
      0x0040000c  0x0000000c  syscall                    10:     syscall
      0x00400010  0x24020008  addiu $2,$0,0x00000008     13:     li $v0, 8
      0x00400014  0x3c011001  lui $1,0x00001001          14:     la $a0, input
      0x00400018  0x34240000  ori $4,$1,0x00000000
      0x0040001c  0x3c011001  lui $1,0x00001001          15:     lw $a1, inputSize
      0x00400020  0x8c250054  lw $5,0x00000054($1)
      0x00400024  0x0000000c  syscall                    16:     syscall
```

```
Address     Value (+0)  Value (+4)  Value (+8)  Value (+c)  Value (+10) Value (+14) Value (+18) Value (+1c)
0x10010000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010020  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010040  [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible]
0x10010060  0x7265746e  0x206e6120  0x69727473  0x203a676e  0x6f590a00  0x79742075  0x20646570  0x20656874
0x10010080  0x69727473  0x203a676e  0x00000020  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100a0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100c0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100e0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
```

This book is, and will always be, a free download. However if you would like to support me to create new content, or simply say "thank you"' for providing this content for free, you can use the site https://www.buymeacoffee.com/CharlesKann and buy me a coffe.

Charles W. Kann

Last Update: Sunday, November 06, 2016

An answer key is currently being written, and is available for the problems in this text. To request a copy of the answer key, write to ckann(at)gettysburg.edu. All requests must come from an account of a recognized educational institution, and the person requesting this material must be listed as an instructor at the institution.

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

Forward

Given the effort of writing a book, the first question an author has to answer is “why bother?” The answer to that question is what frames the book, and what I will describe here.

Why was this book written? First because I do not believe that there is any book currently available which meets the needs I had for a text book. The first is that the can be obtained for a minimal price, or can even be downloaded for free at a number of sites on the web. I am very tired of asking students to pay over $100 per book for classes. I personally have been blessed in so many ways in this world, and I have reached a point in my career where I can take the time to produce this text as a small pay back for all I have been given. I hope that this example will help students who use the book to order their priorities as they go through life, to seek a good outside of personal monetary gain.

I realize that this text could very much use a good editing, unless I can find someone willing to donate the time to do so, I think the basic information in the book is well enough organized to make it useful, and well worth the cost of a free download.

The second reason I wrote this book is that I could not find an assembly programming book that met the need I had in teaching assembly programming. I believe that learning assembly programming is important to every Computer Science student because the principals in assembly affect how high level languages and programs in those languages are implemented. I have purposefully structured the topics in this text to illustrate how concepts such as memory organization (static, heap, and stack) affect variable allocation in high level languages. The chapter on program control is intended to make the student aware of structured programming, which is the basis for control structures in all modern high level languages. Even arrays make more sense in a high level language once one understands how they are implemented, and why. This text is intended to be more than a book about assembly language programming, but to extend assembly language into the principals on which the higher level languages are built.

Finally writing a book is the best way to organize my own thoughts. Much of the material in this text existed for years as a jumble in my own mind. Producing slides and programs for class helped clarify the concepts, but it was when I had to provide a larger organization of the ideas that many of them finally gelled for me. Forcing yourself to explain a concept, particularly in the brutal detail of writing it out, is the best way to organize and learn things.

There are other details about this book that need to be mentioned. Because this book is electronic, it can be released in phases. This text should be looked at in the same way as a beta software release. I know there are mistakes, but I have the ability to correct them and rerelease the text. So comments are welcome.

There is a separate set of appendices which should be available by mid-summer, 2015. I will update this forward with the URL address of those appendices once they are posted. If anyone is in real need of those appendices, I will send them in their current, incomplete, format. I can be contacted at ckann(at)gettysburg.edu.

I will also release an answer guide when it is completed, hopefully in the same mid-summer 2015 time frame. To request an answer guide will require a request from a professor or lecturer at a school, and that the requestor be listed on the department web site for that school. Requests for this document can be made to me at the same address as for the appendices.

I hope the readers find this text useful. I hope it is at least worth the price...

Contents

Table of Figures

Table of Tables

Table of Programs

What you will learn

In this chapter you will learn:

1. binary numbers, and how they relate to computer hardware.
2. to convert to/from binary, decimal, and hexadecimal
3. binary character data representation in ASCII
4. integer numbers, which are represented in binary 2's complement format.
5. arithmetic operations for binary numbers
6. binary logic operations
7. the effect of context on data values in a computer.

Chapter 1 Introduction

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 the program, and a goal of most modern languages is to be hardware agnostic.

Abstraction is a very positive goal, but at some level a computer is just a machine. While High Level Languages (HLL) abstract and hide the underlying hardware, they must be translated into assembly language to use the hardware. One of the goals of a computer science education is to strip away these abstraction and make the workings of the computing machine clear. 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 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 1. 1 Binary Numbers

Chapter 1.1. 1 Values for Binary Numbers

Many students will have had a class covering logic or Boolean algebra, where the 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 Central Processing Unit's (CPU) hardware, called a bit, is implemented as a circuit called a *flip-flop* or *latch.* A flip-flop has a voltage of either 0 volts or a positive voltage (most computers use +5 volts, but many modern computers use +3.3 volts, and any positive voltage is valid). If a flip-flop has a positive voltage it is called *high* or *on* (true), and if it has 0 volts it is *low* or *off* (false). In addition hardware is made up of gates that which can either be *open* (true) or *closed* (false). Finally the goal of a computer is to be able to work with data that a person can understand. These numbers are always large, and hard to represent as a series of true or false values. When the values become large, people work best with numbers, so the binary number 0 is called false, and 1 is called true. Thus while computers work with binary, there are a number of ways we can talk about that 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 is a logical operations values can 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 1-1: Various names for binary values

In addition to the various names, engineers are more comfortable with real operators. This book will follow the convention that "+" is an OR operator, "*" is an AND operator, and "!" (pronounced *bang*) is a 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 classes (as true/false) is the "correct" way to represent them. But keep in mind that 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. They will all be used in this text, though most of the time the binary values of 0 and 1 will be used.

Chapter 1.1. 2 Binary Whole Numbers

The numbering system that everyone learns in school is called *decimal* or *base 10*. 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 switches that can be either on (1) or off(0), 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 1 is added to 9 the number 10 is created. The number 10 means that there is 1 group of ten numbers, and 0 one number. The number 11 is 1 group of 10 and 1 group of one.

When 99 is reached, we have 100, which is 1 group of hundred, 0 tens, and 0 ones. So the number 1,245 would be:

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

Base 2 can be handled in the same manner. The number 10_2 (base 2) is 1 group of two and 0 ones, or just 2_{10} (base 10).[1] 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. Any number in base 2 can be converted to base 10 using this principal. Consider 101011_2, which is:

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

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 1-2: Values of 2^n for n = 0...15

The first 11 powers of 2 are the most important 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 value of 2^n are given in the following table. Using these names and the values of 2^n from 0-9, it is possible to name all of the binary numbers easily 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:

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

4M 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. The names for the various values of 2^n are given in the following table.

2^{10}	Kilo	2^{30}	Giga	2^{50}	Penta

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

2^{20}	Mega	2^{40}	Tera	2^{60}	Exa

Table 1-3: Names for values of 2n, n = 10, 20, 30, 40, 50, 60

Chapter 1. 2 Converting Binary, Decimal, and Hex Numbers

Chapter 1.2. 1 Converting Binary to Decimal

Computers think in 0's and 1's, and when dealing with the internal workings of a computer humans must adjust to the computers mindset. However when the computer produces answers, the humans that use them like to think in decimal. So it is often necessary for programmers to be able to convert between what the computer wants to see (binary), and what the human end users want to see (decimal). These next 3 sections will deal with how to convert binary to decimal, and then give 2 ways to convert decimal to binary. Finally it will give a section on a useful representation for handling large binary numbers called hexadecimal.

To convert 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 convert 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:

$$\begin{array}{rccccccccl} & 64 & 32 & 16 & 8 & 4 & 2 & 1 & \\ 1001010_2 = & 1 & 0 & 0 & 1 & 0 & 1 & 0 & = 64 + 8 + 2 = 74_{10} \end{array}$$

Chapter 1.2. 2 Converting Decimal to Binary using Binary Powers

Two ways to convert 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 convert 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 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 1.2. 3 Converting Decimal to Binary using Division

While conceptually easy to understand, the method to convert decimal numbers to binary numbers in Chapter 1.2.3 is not easy to implement as the starting and stopping conditions are hard to define. There is a way to implement the conversion which results in a nicer algorithm.

The second way 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 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

Chapter 1.2. 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	0x 1	0010	0x 2	0011	0x 3
0100	0x 4	0101	0x 5	0110	0x 6	0111	0x 7
1000	0x 8	1001	0x 9	1010	0x A	1011	0x B
1100	0x C	1101	0x D	1110	0x E	1111	0x F

Table 1-4: 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, I need 4 binary digits 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 in hex. Note that even the hex numbers are often paded 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 where 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 left 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 0xFFFFFFFE

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

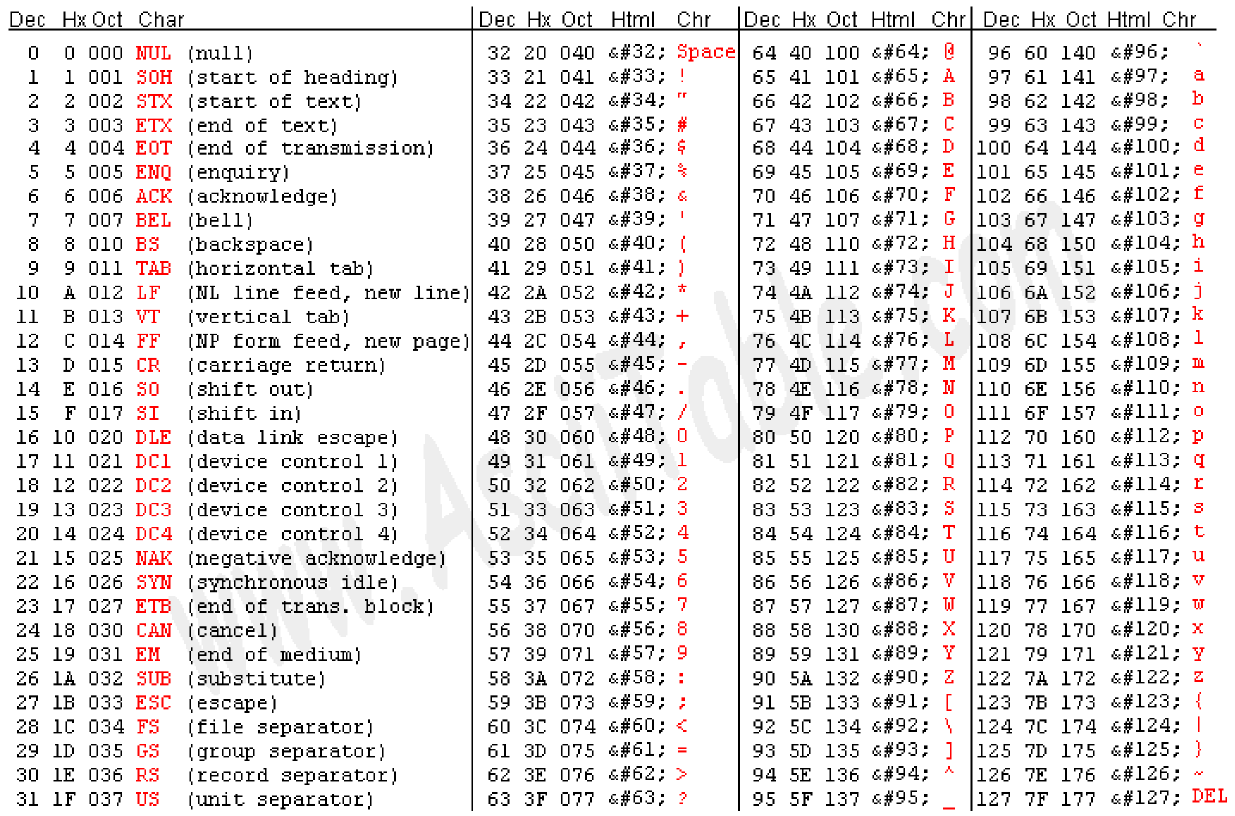
Chapter 1. 3 Character Representation

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 deal 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)[2]. In ASCII all character

[2] 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 MIPS Assembly. So this text will limit all character representations to ASCII.

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

Table 1-5: ASCII Table

Using this table, it is possible to encode a string such as "Once" in ASCII characters as the hexadecimal number 0x4F6E6365[3] (capital O = 0x4F, n = 0x6E, c = 0x53, e = 0x65).

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 digits start with binary digits 0011 0000. Thus 0 is 0x0011 0000, 1 is 00011 0000, 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.

[3] 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 year, however, I know old habits die hard with many students who will struggle endlessly converting to decimal to avoid learning hex.

Also note that all ASCII upper case letters start with the binary string 0010 0000, 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 0100. This relationship between lower case and capital letters will be use to illustrate logical operations later in this chapter.

Chapter 1. 4 Adding Binary Whole Numbers

Before moving on to how integer values are stored and used in a computer for calculations, how to do addition of binary whole numbers needs to be covered.

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 next digit, which is actually adding 3 digits (the two addends, and the carry). So 39 + 28 = 67, where the 10's digit (4) 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, and there is a carry to the next digit of 1_2.

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

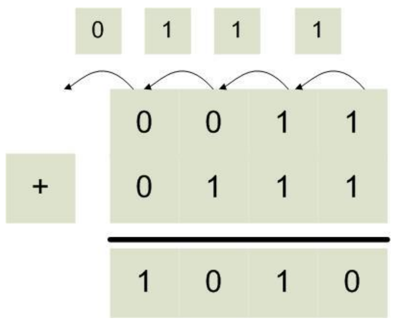


Figure 1-1: Binary whole number addition

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 1. 5 Integer Numbers (2's Complement)

Chapter 1.5. 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 1.5. 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 1.5. 3 The 2's Complement Operation

A 2's complement operation is simply a way to calculate the negation of a 2's complement 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, not two.

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

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.

$$\begin{array}{r} 11010011_2 \\ \underline{00000001_2} \\ 11010100_2. \end{array}$$

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 the 2's complement value of 11010100_2 is 00101100_2. Also note that both the negative and positive values are in 2's complement representation.

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 1.5. 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, numbers were always abstract, theoretical entities, and 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 1. 6 Integer Arithmetic

Chapter 1.6. 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}).

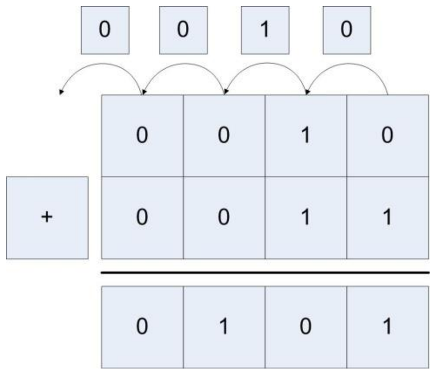


Figure 1-2: Addition of two positive integers

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}).

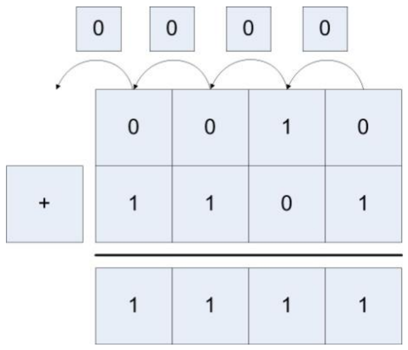


Figure 1-3: Addition of positive and negative integers

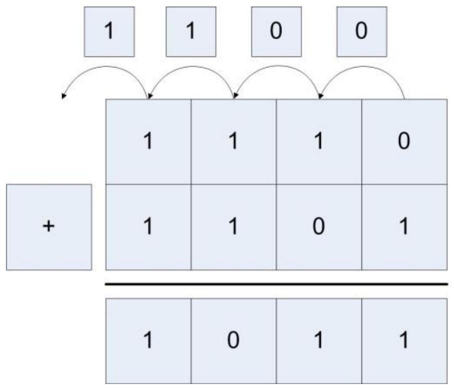


Figure 1-4: Addition of two negative integers

Because integers have fixed sizes, addition and subtraction can cause a problem known as integer overflow. This happens which 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 store in the integer.

Chapter 1.6. 2 Overflow of Integer Addition

Because integers have fixed sizes, addition and subtraction can cause a problem known as integer overflow. This happens which 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 store 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.

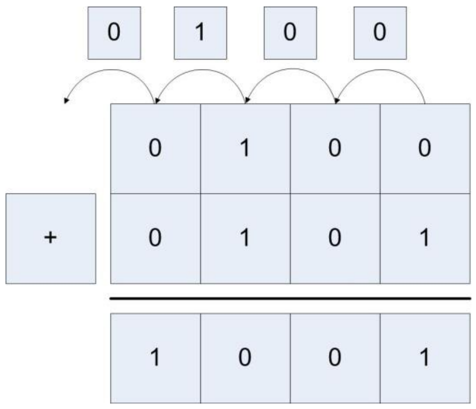


Figure 1-5: Addition with overflow

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.

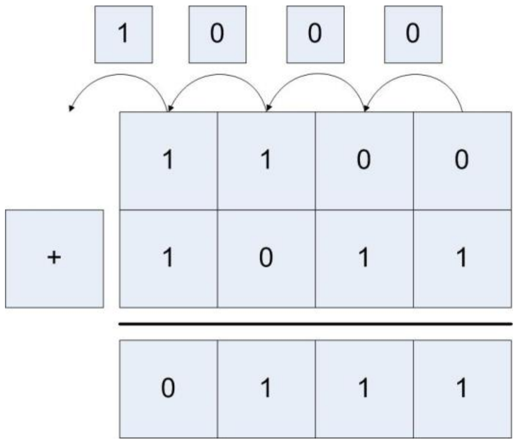


Figure 1-6: Subtraction with overflow

Chapter 1.6. 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` operation. 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 backfilling 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 backfilling 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 and no. Bit shifts and addition can be done in most programming languages, so constant multiplication can be implemented as bits shifts and addition. But 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 1.6. 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). The answer 12.5 is truncated, as this is easily implemented by throwing away the bit which has been shifted out. Likewise,00 011001_2 (25) divided by 2 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 it should be avoided unless there is a strong 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 do more than present the problem.

Chapter 1. 7 Boolean Logical and Bitwise Operators

Chapter 1.7. 1 Boolean Operators

Boolean operators are operators which are designed to operate on a Boolean or binary data. They take in one or more input values of 0/1[4] and combine those bits to create an output value

[4] 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.

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[5] 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

[5] 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.

Input		**Output**				
A	**B**	**AND**	**OR**	**NAND**	**NOR**	**XOR**
0	0	0	0	1	1	0
0	1	0	1	1	0	1
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 1.7. 2 Logical and Bitwise 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) && (y / x > 4))
```

The purpose of this statement is to decide whether or not to enter the statement or code block associated with the if test. What is desired is a single answer, or a single bit, which is either true (1) or false (0). This has two consequences. The first is that in some programming languages (e.g. C/C++) a variable other than a Boolean can be used in the statement, with the consequence that 0 is false, and anything but 0 is true. For example, in the statement `if(x=64)`, the result is true. The equal operator returns a non-zero value, 64, which is true. This has been the cause of many bugs in C/C++, and most modern compilers at least complain about this. This will also be the result of some expressions in assembler, and there will be no compiler to complain about it. So be careful.

The second consequence of the logical Boolean operator is that once a part of it has failed, the entire statement has to fail, and so the rest of the statement will not be executed. This is called short circuiting, and all logical operators are thus short circuiting operators. To see why this is true, consider the if test about. In this if test, if x is 0, then `(x != 0)` is false. Since false and anything is false, there is no need to evaluate the second part of this equation, and so the statement `(y / x > 4)` is not executed. Many programmers will recognize this code, as it is a common pattern for protecting against a zero divide.

The important take away from this is that logical operators are short circuiting 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 upper case letter to letter case, it is only necessary to OR the upper case letter with 0x20. In pseudo code this could be implemented as follows:

```
char toLower(char c) {
    return (c | 0x20)
}
```

In this case the OR operator, |, needs to operator 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 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. &, |, etc.) and the double character operator is the logical operator (&&, ||, etc.).

To make things more confusing, in MIPS only non-short circuiting operators are used, however they are often called logical operators. There is no good way to reconcile this, so the user is cautioned to read the material and programs carefully.

Chapter 1. 8 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 exists. 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 alway 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 represents, or the context of the data. Remember that data without a context is never meaningful.

Chapter 1. 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 1. 10 Exercises

1) What are the following numbers in binary and hexadecimal?
    a. 13_{10}
    b. 15_{10}

   c. 25_{10}
   d. 157_{10}
   e. 325_{10}
   f. 1096_{10}

2) What are the following numbers in decimal?
   a. 10011100_2
   b. $9C_{16}$
   c. $1F_{16}$
   d. $0C_{16}$
   e. $109A_{16}$

3) Give the value of the following numbers (IE. $2^{32} = 4G$)
   a. 2^{16}
   b. 2^{24}
   c. 2^{29}
   d. 2^{34}
   e. 2^{31}

4) Give the 2's complement form for each of the following numbers:
   a. 13
   b. -13
   c. 156
   d. -209

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

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

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

8) Perform the following multiplication and division operations using only bit shifts.
    a. 12 * 4
    b. 8 * 16
    c. 12 * 10
    d. 7 * 15
    e. 12 / 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.
    a. 0x41 (character A)
    b. 0x47 (character G)
    c. 0x57 (character W)

12) Convert the following ASCII characters from lower case to upper case. Do not refer to the ASCII table.
    a. 0x 62 (character b)
    b. 0x69 (character i)
    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 pseducode 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 divide for two variables, or would you always use the *mul* or *div* operators in MIPS 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. to download, install, and run the MARS IDE.
2. what are registers, and how are they used in the CPU.
3. register conventions for MIPS.
4. how memory is configured for MIPS.
5. to assemble and run a program in MARS.
6. the syscall instruction, and how to pass parameters to syscall.
7. what immediate values are in assembly language.
8. assembler directives, operators, and instructions.
9. to input and output integer and string data in MIPS assembly.

Chapter 2 First Programs in MIPS assembly

This chapter will cover first program that is often implemented when writing in a new language, a *Hello World* program. This program is significant in any language because it covers the most fundamental concepts any program can achieve, creating an executing program that can read data in and print results out. Creating an executing program is important because it covers an Integrated Development Environment (IDE) which will allow the programmer to edit the program and to create resultant execution of that program. Being able to input data and output results covers a basic understanding of registers, I/O mechanisms, and provides a mechanism to test algorithms by allowing users to enter data and see if the result is what is expected.

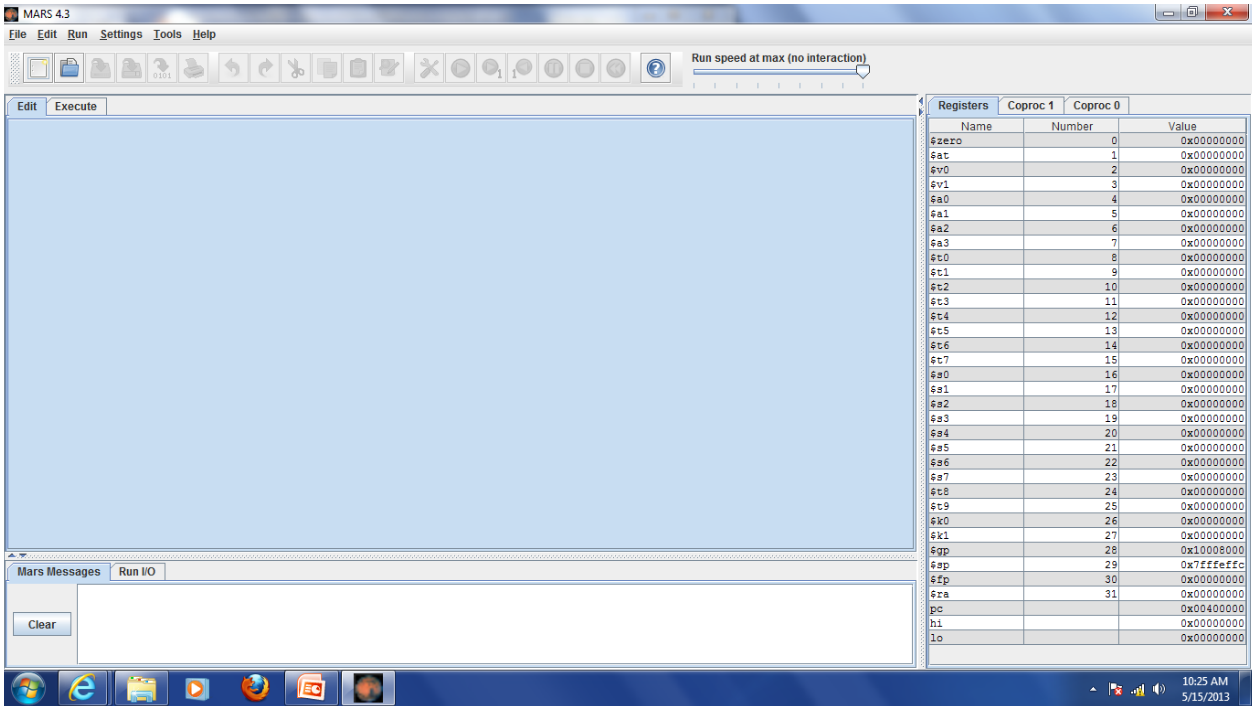
This first program is particularly important because the concepts of statements and variables require a much more in depth knowledge of the language and platform. This chapter is intended to prepare the reader for the rigors of programming MIPS assembly by leading the reader step-by-step into a first working program.

Chapter 2. 1 The MARS IDE

This text will use an IDE called the *MIPS Assembler and Runtime Simulator* (MARS). There are a number of MIPS simulators available, some for educational use, and some for commercial use. However in the opinion of this author, MARS is the easiest to use, and provides the best tools for explaining how MIPS assembly and the MIPS CUP work.

MARS was written by Pete Sanderson and Kenneth Vollmar, and is documented at the site http://courses.missouristate.edu/kenvollmar/mars/index.htm, and should be downloaded from this site. MARS is an executable jar file, so you must have the Java Runtime Environment (JRE) installed to run MARS. A link to Java is on the MARS download page, so you should install Java if it is not on your computer.

Instructions for running MARS are on the download page. When it is started, a page which is similar to the following should come up. This will be the starting point for the material in the rest of the chapter.


MARS 4.3
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0
10:25 AM
5/15/2013

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Figure 2-1: Initial Screen of the MARS IDE

Chapter 2. 2 MIPS and memory

(Note: the next two sections provide background material to assist the reader in understanding the MIPS programs later in the chapter. It might be helpful to first read lightly through this material, then implement the programs. This material is difficult to understand, even for some experienced programmers. It is anticipated that the reader will have to refer to this section throughout the reading of the rest of the book, and quite possibly for future reference in programming in other languages. How memory is implemented and used is a complex and interesting topic, so at least some level of understanding is foundational for the study of Computer Science.) .

It is not unusual for novice programmers to have no concept of memory except as a place to store variables. For a novice this is sufficient, but any real program will require that a programmer have at least a basic knowledge of the types of memory that are used, and the characteristics of each. For example programs that use concurrency are difficult to implement without problems if memory is not understood. Some very powerful design patterns, such as an Immutable Objects, Singletons, or a State Pattern, cannot be understood properly without a knowledge of the characteristics of different types of memory. So every programmer should have at least a basic understanding of how the different types of memory that are used in nearly every computer platform.

One advantage of learning assembly language programming is that it directly exposes many of the types of memory (heap, static data, text, stack and registers) used in a program, and forces the programmer to deal with them. Some memory concepts are more appropriately covered in other courses, such as virtual memory (cache, RAM, and disk) which are generally covered in an OS class.

Chapter 2.2. 1 Types of memory

To a programmer, memory in MIPS is divided into two main categories. The first category, memory that exists in the Central Processing Unit (CPU) itself, is called *register memory* or more commonly simply *registers*. Register memory is very limited and contained in what is often called a *register file* on the CPU. This type of memory will be called *registers* in this text.

The second type of memory is what most novice programmers think of as memory, and is often just called *memory*. Memory for the purposes of this book is where a HLL programmer puts instructions and data. HLL programmers have no access to registers, and so generally have no knowledge of their existence. So from a HLL programmer's point of view, anything stored on a computer is stored in memory.

The non-register memory space of a modern computer, is divided into many different categories, each category having different uses. The different areas of memory studied in detail in this text will be the text, static data, heap and stack sections. Other areas also exist, though this text will not cover them.

Caveats about memory

Students are always complaining that in Computer Science the same terms refer to different things. For example, a binary heap and heap memory are both heaps, but they are completely unrelated terms. As with any study of a complex organization, definitional problems will exist in the study of memory. Therefore it is important to be flexible and understand the contextual meaning or a term and not simply the words. Also keep in mind that external sources of information, like the WWW, may use different terminology, or even the same words with different meanings. So when researching memory, keep the following points in mind.

The first thing to keep in mind is that the view of memory in this text is the programmers view of the memory. The actual implementation of the memory is likely to include virtual memory and several layers of cache. All of this will be hidden from the programmer, so the complexities of the implementation of memory are not considered in this book.

The second thing to keep in mind is that this text will present an older model of memory which is a single threaded process, and does not have virtual program execution (such as the Java Virtual Machine). In reality memory can and does become much more complex than the model given here, but this model is already complex enough, and meets the needs of our assembler programs. So it is a good place to begin understanding memory.

Chapter 2.2. 2 Overview of a MIPS CPU

The following diagram shows a simple design for a 3-Address Load/Store computer, which is applicable to a MIPS computer. This diagram will be used throughout the text to discuss how MIPS assembly is dependent on the computer architecture. To begin this exploration, the components of a CPU and how they interact is explained.

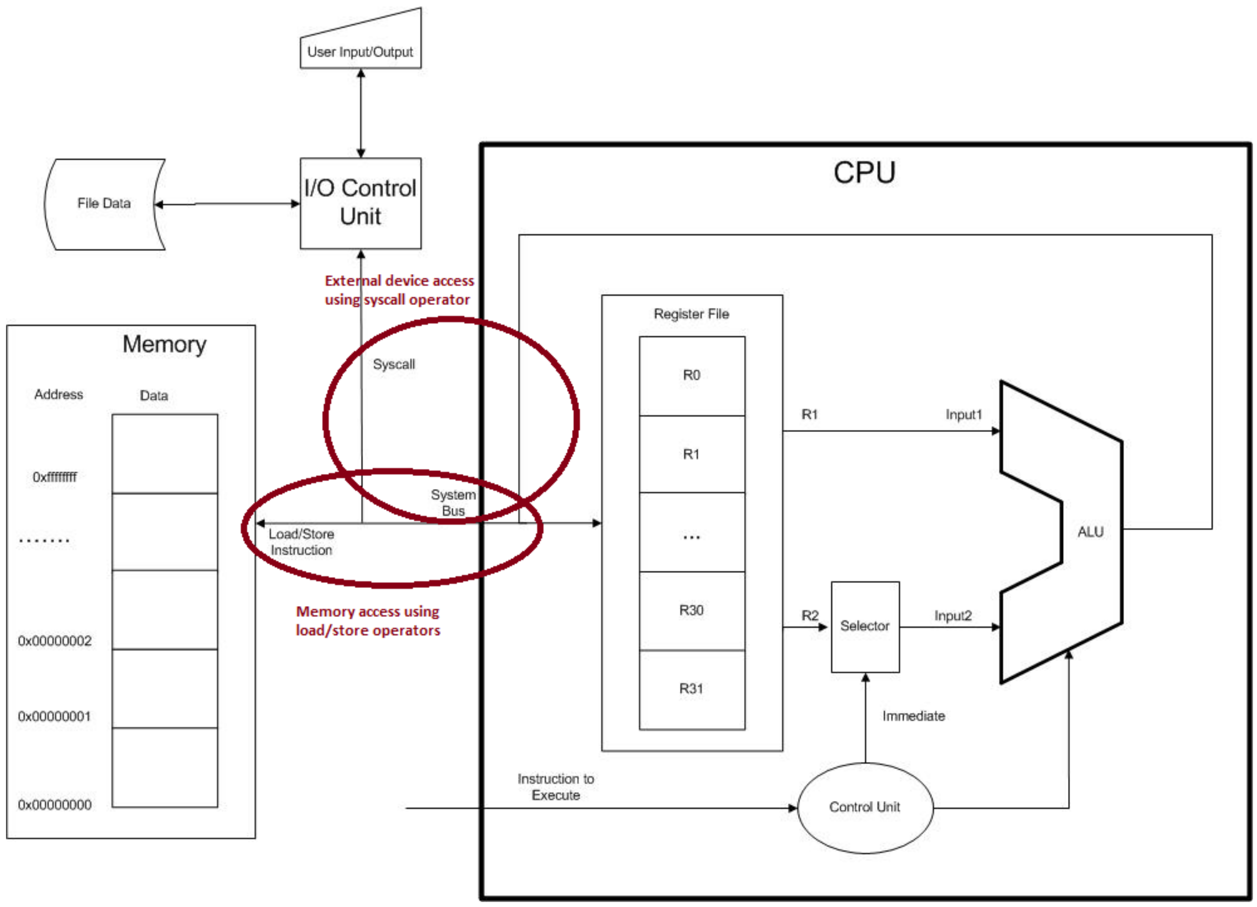


Figure 2-2: 3-address store/load computer architecture

All CPU architectures contain 3 main units. The first is the ALU, which performs all calculations such as addition, multiplication, subtraction, division, bit-shifts, logical operations, etc. Except for instructions which interface to units not on the CPU, such as memory access or interactions with the user of disks, all operations use the ALU. In fact it is reasonable to view basic purpose of the CPU as doing some sort of ALU operation on values from two registers, and storing the result back into a third register.

This interaction of the registers and the CPU helps to explain the purpose of the registers. Registers are a limited amount of memory which exists on the CPU. No data can be operated on in the CPU that is not stored in a register. Data from memory, the user, or disk drives must first be loaded into a register before the CPU can use it. In the MIPS CPU, there are only 32 registers, each of which can be used to store a single 32 bit values. Because the number of these registers is so limited, it is vital that the programmer use them effectively.

In order to use data from memory, the address and data to be read/written is placed on the system bus using a load/store command and transferred to/from the memory to the CPU. The data and address are normally placed on the system bus using a Load Word, lw, or Store Word, sw, operation. The data is then read/written from/to memory to/from a register. To use more than 32 data values in a program, the values must exist in memory, and must be loaded to a register to use.

There is a second way to read/write data to/from a register. If the data to be accessed is on an external device, such as a user terminal or disk drive, the syscall operator is used. The syscall operator allows the CPU to talk to an I/O controller to retrieve/write information to the user, disk drive, etc.

The final part of a CPU is the Control Unit (CU). A CU controls the mechanical settings on the computer so that it can execute the commands. The CU is the focus of a class which is often taught with assembly language, the class being Computer Architecture. This class will not cover the CU in anything but passing detail.

Chapter 2.2. 3 Registers

Registers are a limited number of memory values that exist directly in the CPU. In order to anything useful with data values in memory, they must first be loaded into registers. This will become clearer in each subsequent chapter of this text, but for now it is just important to release that registers are necessary for the CPU to operate on data, and that there are a limited number of them.

Mnemonic	Number	.....	Mnemonic	Number	.....	Mnemonic	Number
$zero	$0		$t3	$11		$s6	$22
$at	$1		$t4	$12		$s7	$23
$v0	$2		$t5	$13		$t8	$24
$v1	$3		$t6	$14		$t9	$25
$a0	$4		$t7	$15		$k0	$26
$a1	$5		$s0	$16		$k1	$27
$a2	$6		$s1	$17		$gp	$28
$a3	$7		$s2	$18		$sp	$29
$t0	$8		$s3	$19		$fp	$30
$t1	$9		$s4	$20		$ra	$31
$t2	$10		$s5	$21			

Table 2-1: Register Conventions

Because the number of registers is very limited, they are carefully allocated and controlled. Certain registers are to be used for certain purposes, and the rules governing the role of the register should be followed. The preceding list is the 32 registers (numbered 0..31) that exist in a MIPS CPU, and their purposes. As with much else in this chapter, the meaning of each of the registers will become clear later in the text.

The conventions for using these registers are outlined below. Note that in some special situations, the registers will take on special meaning, such as with exceptions. These special meanings will be covered when they are needed in the text. Also note that in MARS only the lower case name of the register is valid (for example `$t0` is valid, `$T0` is not).

- `$zero` (`$0`) - a special purpose register which always contains a constant value of 0. It can be read, but cannot be written.
- `$at` (`$1`) - a register reserved for the assembler. If the assembler needs to use a temporary register (e.g. for pseudo instructions), it will use `$at`, so this register is not available for use programmer use.
- `$v0-$v1` (`$2-$3`) –registers are normally used for return values for subprograms. $v0 is also used to input the requested service to syscall.
- `$a0-$a3` (`$4-$7`) - registers are used to pass arguments (or parameters) into subprograms.
- `$t0-$t9` (`$8-$15, $24-$25`) - registers are used to store temporary variables. The values of temporary variables can change when a subprogram is called.
- `$s0-$s8` (`$16-$24`) - registers are used to store saved values. The values of these registers are maintained across subprogram calls.
- `$k0-$k1` (`$26-$27`) - registers are used by the operating system, and are not available for use programmer use.
- `$gp` ($28) - pointer to global memory. Used with heap allocations.
- `$sp` (`$29`) – stack pointer, used to keep track of the beginning of the data for this method in the stack.
- `$fp` (`$30`) – frame pointer, used with the $sp for maintaining information about the stack. This text will not use the $fp for method calls.
- `$ra` (`$31`) – return address: a pointer to the address to use when returning from a subprogram.

Chapter 2.2. 4 Types of memory

MIPS implements a 32-bit flat memory model. This means as far as a programmer is concerned, memory on a MIPS computer starts at address 0x00000000 and extends in sequential, contiguous order to address 0xffffffff. The actual implementation of the memory, which is far from sequential and contiguous, is not of interest to the programmer. The operating system will reliably give the programmer a view of the memory which is flat.

A 32 bit flat memory model says that a program can *address* (or find) 4 Gigabytes (4G) of data. This does not mean that all of that memory is available to the programmer. Some of that memory is used up by the operating system (called *kernel data*), some of it used by the I/O subsystem, etc. But 4G of memory which is addressable.

Figure 4.3 diagrams how the 4G of memory is configured in a MIPS computer. In this chapter only static data and program text memory will be used. Later chapters will cover data such as stack and heap memory. The types of memory used by MIPS are the following:

- Reserved - This is memory which is reserved for the MIPS platform. Memory at these addresses is not useable by a program.
- Program text - (Addresses 0x0040 0000 - 0x1000 00000) This is where the machine code representation of the program is stored. Each instruction is stored as a *word* (32 bits or 4 byte) in this memory. All instructions fall on a *word boundary*, which is a multiple of 4 (0x0040 0000, 0x0040 0004, 0x0040 0080, 0x0040 00B0, etc).
- Static data - (Addresses 0x1001 0000 - 0x1004 0000) This is data which will come from the *data segment* of the program. The size of the elements in this section are assigned when the program is created (assembled and linked), and cannot change during the execution of the program.
- Heap - (Addresses 0x1004 0000 - until stack data is reached, grows upward) Heap is dynamic data which is allocated on an as-needed basis at run time (e.g. with a *new* operator in Java). How this memory is allocated and reclaimed is language specific. Data in heap is always globally available.
- Stack – (Addresses 0x7fff fe00 - until heap data is reached, grows downward) The program stack is dynamic data allocated for subprograms via *push* and *pop* operations. All method local variables are stored here. Because of the nature of the push and pop operations, the size of the stack record to create must be known when the program is assembled.
- Kernel - (Addresses 0x9000 0000 - 0xffff 0000) - Kernel memory is used by the operating system, and so is not accessible to the user.

- MMIO - (Addresses 0xffff 0000 - 0xffff 0010) - Memory Mapped I/O, which is used for any type of external data not in memory, such as monitors, disk drives, consoles, etc.

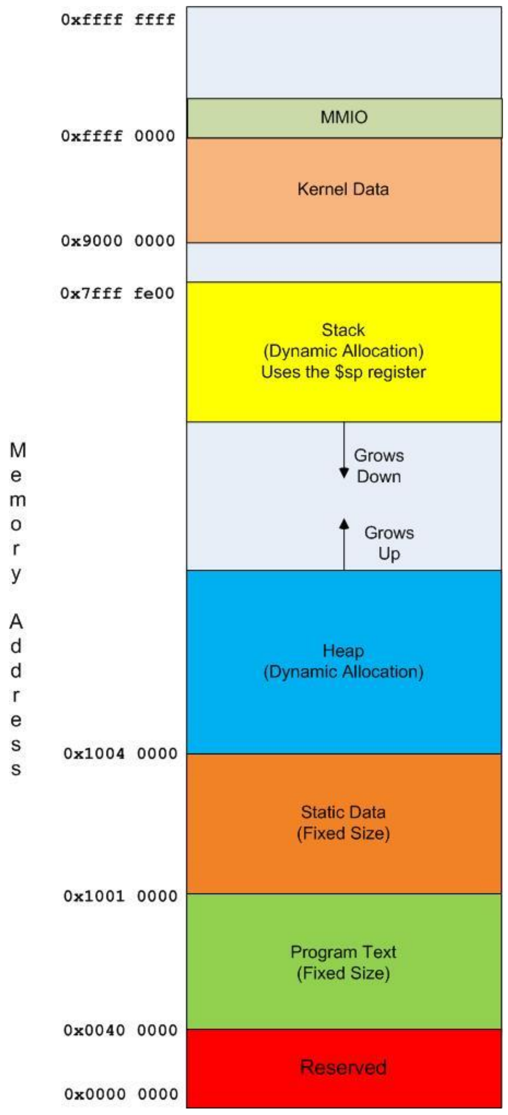


Figure 2-3: MIPS memory configuration

Most readers will probably want to bookmark this section of the text, and refer to it when new memory types or access methods are covered.

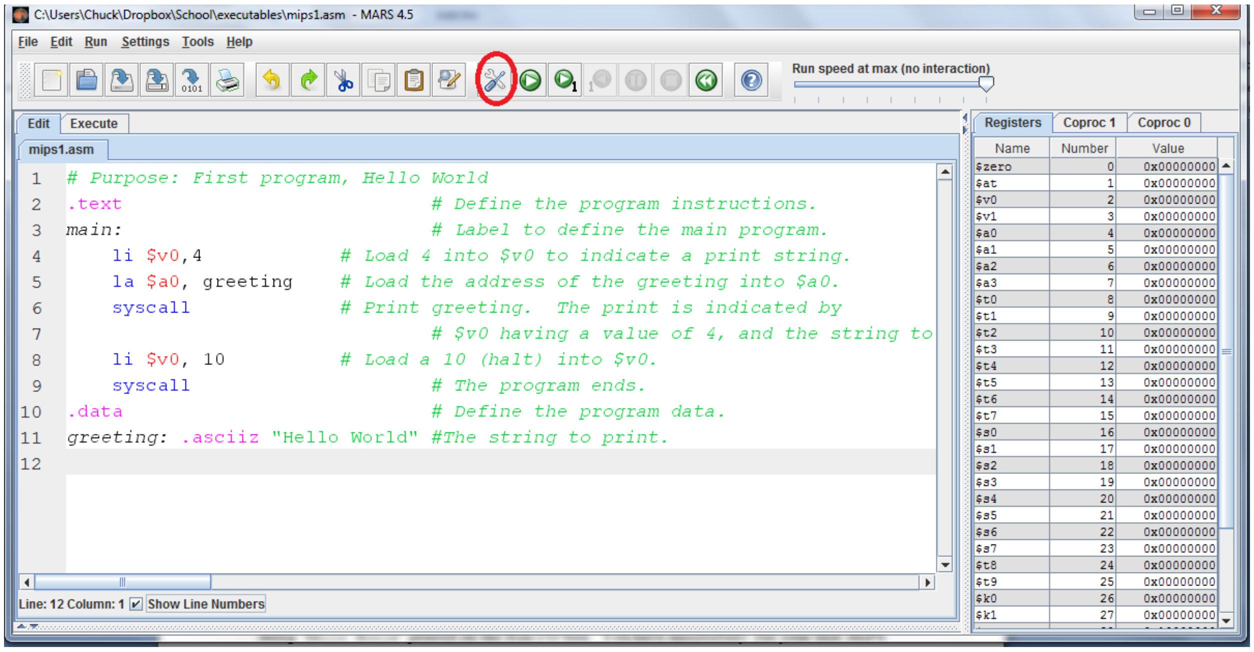
Chapter 2. 3 First program in MIPS assembly

The following is a first MIPS assembly program. It prints out the string "`Hello World`". To run the program, first start the MARS program. Choose the `File->New` menu option, which will open an edit window, and enter the program. How to run the program will be covered in the following the program.

```
# Program File: Program2-1.asm
# Author: Charles Kann
# Purpose: First program, Hello World
.text                   # Define the program instructions.
main:                   # Label to define the main program.
    li $v0,4            # Load 4 into $v0 to indicate a print string.
    la $a0, greeting    # Load the address of the greeting into $a0.
    syscall             # Print greeting.  The print is indicated by
                        # $v0 having a value of 4, and the string to
                        # print is stored at the address in $a0.
    li $v0, 10          # Load a 10 (halt) into $v0.
    syscall             # The program ends.
.data                   # Define the program data.
greeting: .asciiz "Hello World" #The string to print.
```

Program 2-1: Hello World program

Once the program has been entered into the edit window, it needs to be assembled. The option to assemble the program is shown circled in red below.


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
mips1.asm
Registers Coproc 1 Coproc 0
Line: 12 Column: 1
Show Line Numbers

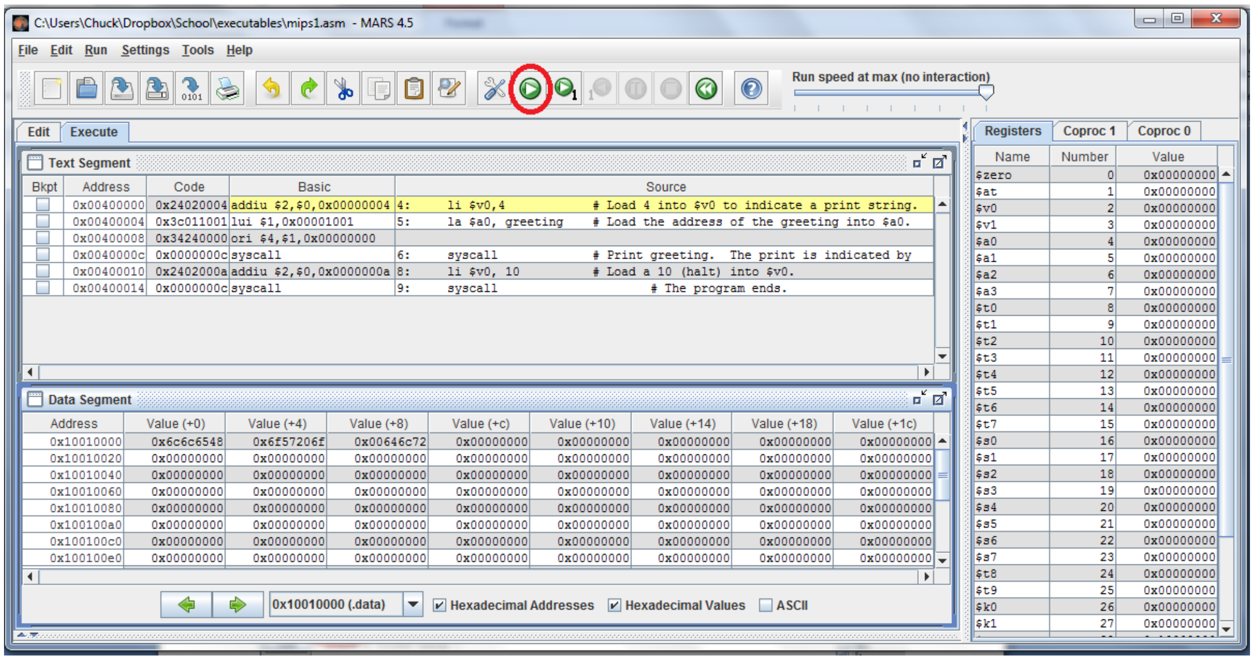
```
# Purpose: First program, Hello World
.text                          # Define the program instructions.
main:                          # Label to define the main program.
    li $v0,4            # Load 4 into $v0 to indicate a print string.
    la $a0, greeting    # Load the address of the greeting into $a0.
    syscall             # Print greeting.  The print is indicated by
                               # $v0 having a value of 4, and the string to
    li $v0, 10          # Load a 10 (halt) into $v0.
    syscall                    # The program ends.
.data                          # Define the program data.
greeting: .asciiz "Hello World" #The string to print.
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000

Figure 2-4: Assembling a program

If you entered the program correctly you should get the edit screen below. If you made any errors, the errors will be displayed in a box at the bottom of the screen.

To run the program, click the green arrow button circled in the figure below. You should get the string "`Hello World`" printed on the Run I/O box. You have successfully run your first MIPS program.


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Registers Coproc 1 Coproc 0

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24020004	addiu $2,$0,0x00000004	4: li $v0,4 # Load 4 into $v0 to indicate a print string.
	0x00400004	0x3c011001	lui $1,0x00001001	5: la $a0, greeting # Load the address of the greeting into $a0.
	0x00400008	0x34240000	ori $4,$1,0x00000000	
	0x0040000c	0x0000000c	syscall	6: syscall # Print greeting. The print is indicated by
	0x00400010	0x2402000a	addiu $2,$0,0x0000000a	8: li $v0, 10 # Load a 10 (halt) into $v0.
	0x00400014	0x0000000c	syscall	9: syscall # The program ends.

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x6c6c6548	0x6f57206f	0x00646c72	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000

Figure 2-5: Running a program

Chapter 2.3. 1 Program 2-1 Commentary

Program 2-1 was used to show how to compile and run a program. This is not an introductory programming class, and most readers probably are less interested in how to make the program work, and more interested in the details of the program. This section will go over a number of those details, and help the reader understand the program.

- MIPS assembler code can be indented, and left white space on a line is ignored. All instructions must be on a single line, The # is means any text from the # to the end of a line is a comment and to be ignored. Strings are denoted by "'s marks around the string.
- Note the comments at the start of the file. These will be called a *file preamble* in this text. At a minimum all program should contain at least these comments. The name of the file should be documented, as unlike some HLL such as Java, the name of the file is no where implied in the program text. It is very easy to lose files in this situation, so the source code should always contain the name of the file. The file preamble should also contain the programmer who created the code, and a short description of why this program was written.
- Assembly language programs are not *compiled*, they are *assembled*. So a program does not consist of statements and blocks of statements as in a HLL, but a number of instructions telling the computing machine how to execute. These instructions are very basic, such as `li $v0, 4` (put the value 4 in `$v0`). Because this is a complete change of perspective from a HLL, it is useful to explicitly make this point, and for the reader to take note of it.

- In MIPS and most assembly languages in general, a "." before an text string means the token (string) that follows it is an as*sembler directive*. In this program the `.text` directive means the instructions that follow are part of a program text (i.e. the program), and to be assembled into a program and stored in the text region of memory. The `.data` directive means that what follows is program data, and to be stored in the static data region of memory. The `.asciiz` directive tells the assembler to interpret the data which follows it as an ASCII string. Typing a "." in the MARS edit window will give you all the MIPS assembler directive.

- In MIPS assembler any text string followed by a ":" is a label. A label is just a marker in the code that can be used in other statements, as the `la $a0 greeting` instruction says to load the text at the label greeting into the `$a0` register. Note that data labels are not equivalent to variables. Labels are just markers in the program, and nothing more. Variables have a type, and there is no typing of any data done in assembler.

- The label `main:` does not need to be included as MARS assumes the program begins at the first line in the assembled program. But it is nice to label the starting point, and generally most runtimes will look for a global symbol name *main* as the place to begin execution. So in this text it will just be included.

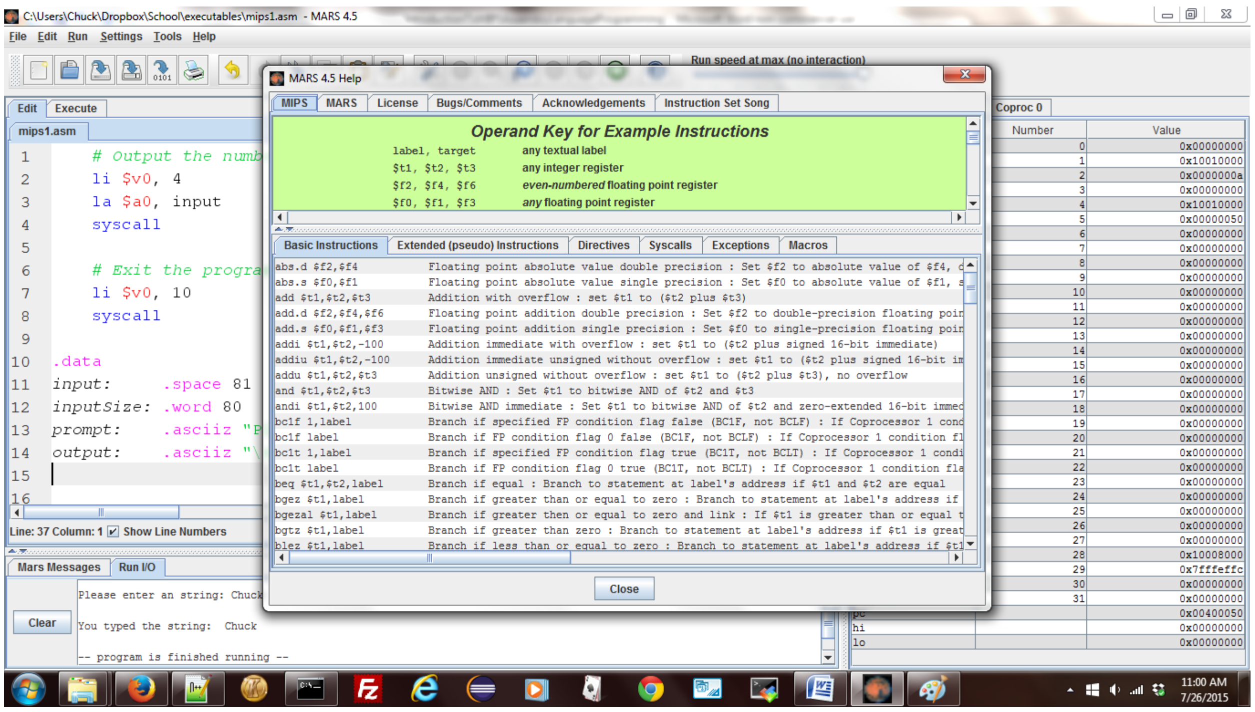
- Any time a constant is included in an instruction, it is called an *immediate* value. The constant must be in the instruction itself, so the value 4 in the instruction `li $a0,4` is an immediate value, but the string "Hello World" is a constant but not an immediate value.

- Only instructions and labels can be defined in a text segment, and only data and labels can be defined in a data segment. You can have multiple data segments and multiple text segments in a program, but the text must be in a text segment and the data in a data segment.

- Operators are text strings like `li`, `la`, and `syscall`. `li` means load the immediate value into the register (e.g. `li $v0,4` means $v0 <- 4). `la` means to load the address at the label into the register. `syscall` is used to request a system service. System services will be covered in more detail later in this chapter. A list of all MIPS operators used in this text can be found in the MIPS Greensheet or by using the help option in MARS.

- Instructions are operators and their arguments. So `li` is an operator; `li $v0,4` is an instruction.

- The syscall operator is used to call system services. System services provide access to the user console, disk drives, and any other external devices. The service to be executed is a number contained in the $v0 register. In this program there are two services that are used. The services are: service 4 prints a string starting at the address of the memory contained in the $a0; and service 10 halts, or exit, the program. A complete list of all syscall be found by using the help menu option (or F1) in MARS. This will bring up the following screen, which gives all the possible options in MARS.


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
mips1.asm
MARS 4.5 Help
MIPS MARS License Bugs/Comments Acknowledgements Instruction Set Song
Coproc 0
Line: 37 Column: 1 Show Line Numbers
Mars Messages Run I/O
Clear
Close
11:00 AM
7/26/2015

```
# Output the numb
li $v0, 4
la $a0, input
syscall

# Exit the progra
li $v0, 10
syscall

.data
input:      .space 81
inputSize:  .word 80
prompt:     .asciiz "P
output:     .asciiz "\
```

Operand Key for Example Instructions

label, target	any textual label
$t1, $t2, $t3	any integer register
$f2, $f4, $f6	*even-numbered* floating point register
$f0, $f1, $f3	*any* floating point register

Basic Instructions | Extended (pseudo) Instructions | Directives | Syscalls | Exceptions | Macros

```
abs.d $f2,$f4         Floating point absolute value double precision : Set $f2 to absolute value of $f4, d
abs.s $f0,$f1         Floating point absolute value single precision : Set $f0 to absolute value of $f1, s
add $t1,$t2,$t3       Addition with overflow : set $t1 to ($t2 plus $t3)
add.d $f2,$f4,$f6     Floating point addition double precision : Set $f2 to double-precision floating poin
add.s $f0,$f1,$f3     Floating point addition single precision : Set $f0 to single-precision floating poin
addi $t1,$t2,-100     Addition immediate with overflow : set $t1 to ($t2 plus signed 16-bit immediate)
addiu $t1,$t2,-100    Addition immediate unsigned without overflow : set $t1 to ($t2 plus signed 16-bit im
addu $t1,$t2,$t3      Addition unsigned without overflow : set $t1 to ($t2 plus $t3), no overflow
and $t1,$t2,$t3       Bitwise AND : Set $t1 to bitwise AND of $t2 and $t3
andi $t1,$t2,100      Bitwise AND immediate : Set $t1 to bitwise AND of $t2 and zero-extended 16-bit immed
bc1f 1,label          Branch if specified FP condition flag false (BC1F, not BCLF) : If Coprocessor 1 cond
bc1f label            Branch if FP condition flag 0 false (BC1F, not BCLF) : If Coprocessor 1 condition fl
bc1t 1,label          Branch if specified FP condition flag true (BC1T, not BCLT) : If Coprocessor 1 condi
bc1t label            Branch if FP condition flag 0 true (BC1T, not BCLT) : If Coprocessor 1 condition fla
beq $t1,$t2,label     Branch if equal : Branch to statement at label's address if $t1 and $t2 are equal
bgez $t1,label        Branch if greater than or equal to zero : Branch to statement at label's address if
bgezal $t1,label      Branch if greater then or equal to zero and link : If $t1 is greater than or equal t
bgtz $t1,label        Branch if greater than zero : Branch to statement at label's address if $t1 is great
blez $t1,label        Branch if less than or equal to zero : Branch to statement at label's address if $t1
```

	Number	Value
	0	0x00000000
	1	0x10010000
	2	0x0000000a
	3	0x00000000
	4	0x10010000
	5	0x00000050
	6	0x00000000
	7	0x00000000
	8	0x00000000
	9	0x00000000
	10	0x00000000
	11	0x00000000
	12	0x00000000
	13	0x00000000
	14	0x00000000
	15	0x00000000
	16	0x00000000
	17	0x00000000
	18	0x00000000
	19	0x00000000
	20	0x00000000
	21	0x00000000
	22	0x00000000
	23	0x00000000
	24	0x00000000
	25	0x00000000
	26	0x00000000
	27	0x00000000
	28	0x10008000
	29	0x7fffeffc
	30	0x00000000
	31	0x00000000
pc		0x00400050
hi		0x00000000
lo		0x00000000

```
Please enter an string: Chuck

You typed the string:  Chuck

-- program is finished running --
```

Chapter 2. 4 Program to prompt and read an integer from a user

The next program that will be studied prompts a user to input a number, then reads that number into a register and prints it back to the user.

```
# Program File: Program2-2.asm
# Author: Charles Kann
# Program to read an integer number from a user, and
# print that number back to the console.
.text
main:
    # Prompt for the integer to enter
    li $v0, 4
    la $a0, prompt
    syscall

    # Read the integer and save it in $s0
    li $v0, 5
    syscall
    move $s0, $v0

    # Output the text
    li $v0, 4
    la $a0, output
    syscall

    # Output the number
    li $v0, 1
    move $a0, $s0
    syscall

    # Exit the program
    li $v0, 10
    syscall

.data
prompt: .asciiz "Please enter an integer: "
output: .asciiz "\nYou typed the number "
```

Program 2-2: Program to read an integer from the user

Chapter 2.4. 1 Program 2-2 Commentary

The following commentary covers new information which is of interest in reading Program 2-2.

- In this program, blocks of code are commented, not each individual statement. This is a better way to comment a program. Each block should be commented as to what it does, and if it is not obvious, how the code works. There should not be a need to comment each line, as a programmer should generally be able to understand the individual instructions.

- A new operator was introduced in this program, the `move` operator. The `move` operator moves the text from one register to another. In this case, the result of the syscall read integer service 5 is moved from register `$v0` to a save register `$s0`.

- Two new syscall services have been introduced. The first is service 5. Service 5 synchronously waits for the user to enter an integer on the console, and when the integer is typed returns the integer in the return register `$v0`. This service checks to see that the value entered is an integer value, and raises an exception if it is not.

- The second new syscall service is service 1. Service 1 prints out the integer value in register `$a0`. Note that with service 4 that string that is at the address in `$a0` (or referenced by `$a0`) is printed. With the service 1 the value in register `$a0` is printed. This difference between a reference and a value is extremely important in all programming languages, and is often a difficult subject to understand even in a HLL.

- In this program, an escape character "`\n`" is used in the string named `output`. This escape character is called the new line character, causes the output from the program to start on the next line. Note that the actually sequence of characters written to the output can vary based on the operation system and environment the program is run on, but the escape character will create the correct output sequence to move to the start of a new line.

Chapter 2. 5 Program to prompt and read a string from a user

The programs to read a number from a user and read a string from a user look very similar, but are conceptually very different. The following program shows reading a string from the user console.

```
# Program File: Program2-3.asm
# Author: Charles Kann
# Program to read a string from a user, and
# print that string back to the console.
.text
```

```
main:
    # Prompt for the string to enter
    li $v0, 4
    la $a0, prompt
    syscall

    # Read the string.
    li $v0, 8
    la $a0, input
    lw $a1, inputSize
    syscall

    # Output the text
    li $v0, 4
    la $a0, output
    syscall

    # Output the number
    li $v0, 4
    la $a0, input
    syscall

    # Exit the program
    li $v0, 10
    syscall

.data
input:      .space 81
inputSize:  .word 80
prompt:     .asciiz "Please enter an string: "
output:     .asciiz "\nYou typed the string:  "
```

Program 2-3: Program to read a string from the user

Chapter 2.5. 1 Program 2-3 Commentary

The following commentary covers new information which is of interest in reading Program 2-3.

- There was two new assembler directives introduced in this program. The first is the `.space` directive. The `.space` directive allocates `n` bytes of memory in the data region of the program, where n=81 in this program. Since the size of a character is 1 byte, this is equivalent to saving 80 characters for data. Why 81 is used will be covered in the discussion of strings later in this section.

- The `.word` directive allocates 4 bytes of space in the data region. The `.word` directive can then be given an integer value, and it will initialize the allocated space to that integer value. Be careful as it is incorrect to think of a the `.word` directive as a declaration for an integer, as this directive simply allocates and initializes 4 bytes of memory, it is not a data type. What is stored in this memory can by any type of data.

- As was discussed earlier in this chapter, the `la` operator loads the address of the label into a register. In HLL this is normally called a reference to the data, and this text will use both of these terms when referring to reference data. This will be shown in the text as follows, which means the value of the label (the memory address) is loaded into a register.

$a0 <= label

- A new operator, `lw`, was introduced in this section. The `lw` operator loads the value contained at the label into the register, so in the preceding program `lw $a1,inputSize` loaded the value `80` into the register $`a1`. Loading of values into a register will be shown in the text as follows, which means the value at the label is loaded into a register.

  $a1 <= M[label]

- In MIPS assembly, a string is a sequence of ASCII characters which are terminated with a null value (a null value is a byte containing 0x00). So for example the string containing "Chuck" would be 0x436875636b00 in ASCII. Thus when handling strings, an extra byte must always be added to include the null terminator. The string "Chuck", which is 5 character, would require 6 bytes to store, or to store this string the following .space directive would be used.

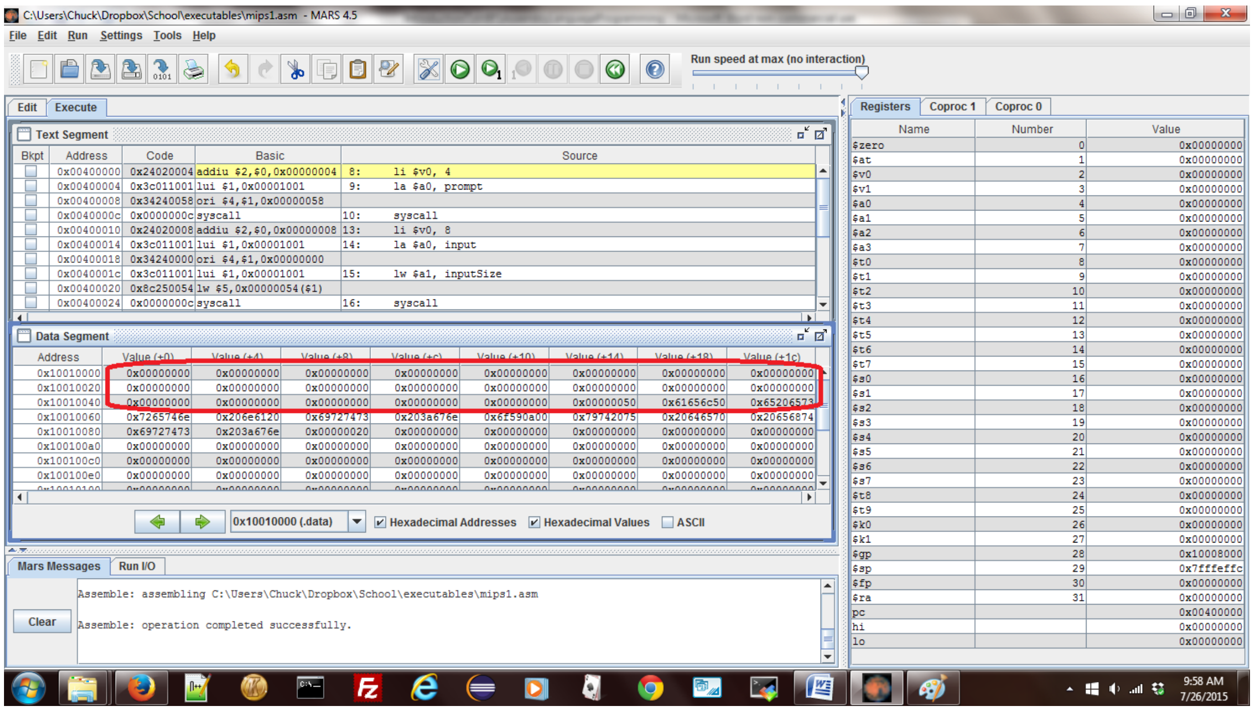
  .space 6

  This is why in the preceding program the string input, which was 80 characters big, required a space of 81. This is also the reason for the assembler directives `.ascii` and `.asciiz`. The `.ascii` directive only allocates the ASCII characters, but the `.asciiz` directive allocates the characters terminated by a null. So the `.asciiz` allocates a string.

- Reading a string from the console is done using the `syscall` service 8. When using `syscall` service 8 to read a string, there are two parameters passed to the service. The first is a reference to the memory to use to store the string (stored in `$a`0), and the second is the maximum size of the string to read (stored in `$a1`). Note that the size is 1 less than the number of characters available to account for the null terminator. If the string the user enters is larger than the maximum size of the string, it is truncated to the maximum size. This is to prevent the program from accessing memory not allocated to the string.

  The parameters passed to the method are the string reference in $a0, and the maximum size of the string in $a1. Note that in the case of the string in $a0, the value for the string is contained in memory, and only the reference is passed to the function. Because the reference is passed, the actual value of the string can be changed in memory in the function. This we will equate to the concept of *pass-by-reference*[6] in a language like Java. In the case of string size, the actual value is contained in $a1. This corresponds to the concept of *pass-by-value* in a language like Java. A Java program to illustrate this is at the end of this chapter. This topic of value and reference types will be covered in much greater details in the chapters on subprograms and arrays.

[6] It would be more exact to call this a *pass-by-reference-value*, as it is not a true *pass-by-reference* as is implemented in a language like C or C#. But this parameter passing mechanism is commonly called *pass-by-reference* in Java, and the difference between the two is beyond what can be explained in assembly at this point.


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Text Segment
Data Segment
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
Clear
9:58 AM
7/26/2015

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24020004	addiu $2,$0,0x00000004	8: li $v0, 4
	0x00400004	0x3c011001	lui $1,0x00001001	9: la $a0, prompt
	0x00400008	0x34240058	ori $4,$1,0x00000058	
	0x0040000c	0x0000000c	syscall	10: syscall
	0x00400010	0x24020008	addiu $2,$0,0x00000008	13: li $v0, 8
	0x00400014	0x3c011001	lui $1,0x00001001	14: la $a0, input
	0x00400018	0x34240000	ori $4,$1,0x00000000	
	0x0040001c	0x3c011001	lui $1,0x00001001	15: lw $a1, inputSize
	0x00400020	0x8c250054	lw $5,0x00000054($1)	
	0x00400024	0x0000000c	syscall	16: syscall

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000050	0x61656c50	0x65206573
0x10010060	0x7265746e	0x206e6120	0x69727473	0x203a676e	0x6f590a00	0x79742075	0x20646570	0x20656874
0x10010080	0x69727473	0x203a676e	0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

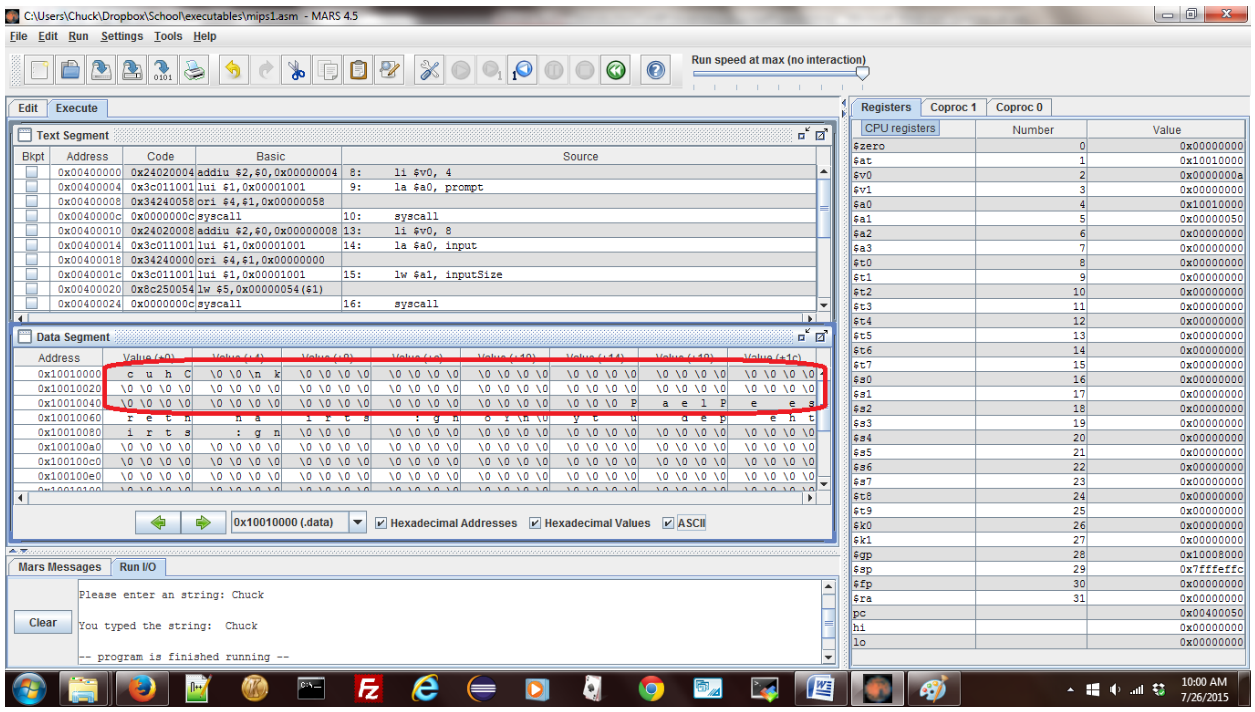
```
Assemble: assembling C:\Users\Chuck\Dropbox\School\executables\mips1.asm

Assemble: operation completed successfully.
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Figure 2-6: Memory before entering a string

- When using syscall service 8, the syscall actually changes the memory in the data region of the program. To understand this, the preceding figure shows the program execution string immediately before the program is run. Note that the memory circled in red is the space which was saved for the input string, and it is all null values.

  Run the program and enter "Chuck" at the prompt for a string. The memory for the input string has been changed to store the value "Chuck", as shown in the circled text in the figure below (be sure to select the ASCII checkbox, or the values will show up in hex).


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.5

Run speed at max (no interaction)

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24020004	addiu $2,$0,0x00000004	8: li $v0, 4
	0x00400004	0x3c011001	lui $1,0x00001001	9: la $a0, prompt
	0x00400008	0x34240058	ori $4,$1,0x00000058	
	0x0040000c	0x0000000c	syscall	10: syscall
	0x00400010	0x24020008	addiu $2,$0,0x00000008	13: li $v0, 8
	0x00400014	0x3c011001	lui $1,0x00001001	14: la $a0, input
	0x00400018	0x34240000	ori $4,$1,0x00000000	
	0x0040001c	0x3c011001	lui $1,0x00001001	15: lw $a1, inputSize
	0x00400020	0x8c250054	lw $5,0x00000054($1)	
	0x00400024	0x0000000c	syscall	16: syscall

Data Segment

```
Address     Value (+0)   Value (+4)   Value (+8)   Value (+c)   Value (+10)  Value (+14)  Value (+18)  Value (+1c)
0x10010000    c  u  h  C \0 \0 \n  k  \0 \0 \0 \0 \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0
0x10010020  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0 \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0
0x10010040  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0 \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0  P   a  e  l  P   e     e  s
0x10010060    r  e  t  n      n  a     i  r  t  s     :  g  n   o  Y \n \0   y  t     u      d  e  p      e  h  t
0x10010080    i  r  t  s      :  g  n  \0 \0 \0     \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0
0x100100a0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0 \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0
0x100100c0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0 \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0
0x100100e0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0 \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0  \0 \0 \0 \0
```

0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecimal Values ✓ ASCII

Run I/O

```
Please enter an string: Chuck

You typed the string:  Chuck

-- program is finished running --
```

CPU registers	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000050
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400050
hi		0x00000000
lo		0x00000000

In his figure there are 8 bytes containing the characters "cuhC \0\0\nk". This is the string "Chuck", plus a new line character which is always returned by service 8, the null terminator and an extra byte of memory which was not used. This shows that the $a0 parameter to service 8 was actually a memory reference, and the service updated the memory directly.

The second thing to note in this figure is that the letters are stored backwards each grouping of 4 bytes, or a memory word. In this example, the string "Chuck\n" was broken into two strings "Chuc" and "k\n". The characters were then reversed, resulting in "cuhC" and "\nk". This is a common format in computer hardware referred to as *little endian*. Little endian means that bytes are stored with the least significant byte in the lowest address, which reverses the 4 bytes in the memory word. Big endian is the reverse, and in a big endian system the string would appear in memory as it was typed. The choice of big endian verses little endian is a decision made by the implementers of the hardware. You as a programmer just have to realize what type of format is used, and adjust how you interpret the characters appropriately.

Note from this figure that the service 8 call always appends a "\n" to the string. There is no mechanism to change this in MARS, and no programmatic way to handle this in our programs. This is an annoyance which we will be stuck with until strings are covered at the end of this text.

Finally see that while the string which is returned has 6 character, "Chuck\n", the other 80 characters in memory have all be set to zero. The space allocated for the string is still 80, but the string size is 6. What determines the string size (the actual number of characters used) is the position of the first zero, or null. Thus strings are referred to as "null terminated". Many HLL, like C and C++[7], use this definition of a string.

Chapter 2. 6 Summary

The purpose of this chapter was to help the reader understand how to create and run simple programs with I/O in MIPS assembly using the MARS IDE. However even a simple "Hello World" program has complexities when written in assembly language, and this chapter attempted to give at least a casual explanation of these. Covered in this chapter were the following concepts:

- How to comment a MIPS program.
- Registers and memory in MIPS computers.
- Assembler directives such as `.text`, `.data`, `.asciiz`, `.space`, and `.word`.
- Labels in MIPS assembly.
- MIPS assembly operators, such as `li`, `la`, `lw`, and `move`.
- System services for interacting with the user console, in particular services 1, 4, 5, and 8.
- The difference between references and values of data.

Chapter 2. 7 Java program for call by value and reference

To better explain the concept of call by value and call by reference, the following Java program is used.

```
public class CallingConventions {
    public static void func(int a, final int b[])
    {
        a = 7;
        b[0] = 7;
    }

    public static void main(String... argv) {
        int a = 5;
        int b[] = {5};

        System.out.println("Before call, a = " + a + " and b[0] = " + b[0]);
        func(a, b);
        System.out.println("After call, a = " + a + " and b[0] = " + b[0]);
    }
}
```

Note that when this program is run, the value of a is not changed by the function, but the value of `b[0]` is changed. What is happening is very much analogous to the previously presented MIPS program which prompted for a string. In this case the variable `a` is a *value type*, e.g. the variable stores the value. In the case of the variable `b`, the values are stored in an array, and those values

[7] C++ uses null terminated strings, but can also define strings as instances of the String class. Instances of the String class are very different from null terminated strings, so do not confuse the two.

cannot be stored in a single data value (or register). So what is stored in `b` is the a reference to the array, or simply the address of `b` in memory. Thus `b` is a reference type.

Note that in both cases when calling the function something is essentially *copied* into a memory location (in our MIPS program registers). But if the value is copied, it cannot be changed. If the reference is copied, while the reference cannot be changed, the value which is referred to can.

This will become very important when discussing subprograms and arrays later in the text.

Chapter 2. 8 Exercises

1) Write a program which prompts the user to enter their favorite type of pie. The program should then print out "`So you like _____ pie`", where the blank line is replaced by the pie type entered. What annoying feature of syscall service 4 makes it impossible at this point to make the output appear on a single line?

2) Using the syscall services, write a program to play a middle "C" for 1 second as a reed instrument using the Musical Instrument Digital Interface (MIDI) services. There are two services which produce the output? What is the difference between them?

3) Write a program to print out a random number from 1..100.

4) Write a program which sleeps for 4 seconds before exiting.

5) Write a program to read and print a floating point number. What is strange about the registers used for this program?

6) Explain the difference between an address and a value for data stored in memory.

7) Write a program to open an input dialog box and read a string value. Write the string back to the user using a message box.

What you will learn

In this chapter you will learn:

1. 3-Address machines.
2. the difference between real (or native) MIPS operators, and pseudo operators.
3. introductory pseudo code.
4. addition and subtraction operators.
5. logical (or bit-wise Boolean) operations in MIPS assembly language.
6. pseudo operators.
7. multiplication and division operators, and how to use them.
8. shifting data in registers, and the different types of shifts in MIPS.

Chapter 3 MIPS arithmetic and Logical Operators

Chapter 3. 1 3-Address machines

The MIPS CPU organization is called a 3-address machine. This is because most operations allow the specification of 3 registers as part of the instruction. To understand this better, consider the following figure, which is similar to figure 2.2 except that now the ALU logic portion of the diagram is emphasized.

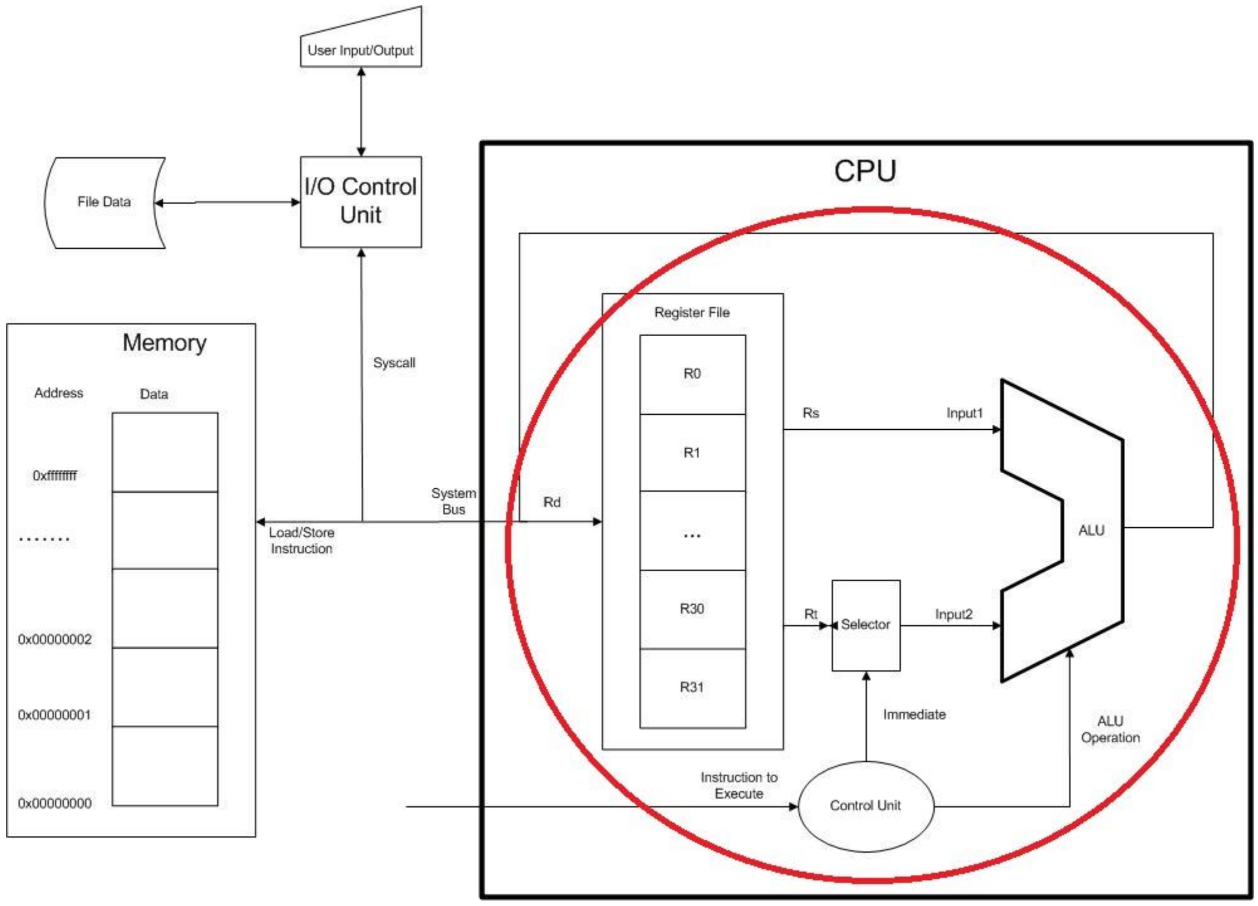


Figure 3-1: MIPS computer architecture

Most MIPS operators take 3 registers as parameters, which is the reason it is called a 3-address machine. They are: the first input to the ALU, always R_s (the first source register); the second input to the ALU, either R_t (the second source register) or an immediate value; and the register to write the result to, R_d (the destination register). All operators, with the exception the shift operators, used in this chapter will follow this format. Even the shift operators will appear to follow this convention, and it will be only when encoding them into machine code in Chapter 4 that they will appear different.

The final thing to note about this diagram is that the ALU takes two inputs. The first is always the R_s register. The second input is determined by the type of operator, of which there will be two types that will be looked at in this chapter. The first type of operator is an Register (R) operator. The register operator will always take the second input to the ALU from the R_t register. The format of all R operators will be as follows:

[operator] R_d, R_s, R_t

This syntax means the following:

R_d <- R_s [operator] R_t

The second type of operator is an Immediate (I) operator. There are a limited number of operations for which an I operator is defined. When they are defined, the operation they perform will be specified by appending an "i" to the end of the operator name, and the format will replace the R_t parameter in the R operator with the I (immediate) value which comes from the instruction. The format of all I operators is:

[operator]i R_t, R_s, Immediate value

This syntax means the following:

R_t <- R_s [operator]i Immediate value

This helps explain the difference between a constant and an immediate value. A constant is a value which exists in memory, and must be loaded into a register before it is used. An immediate value is defined as part of the instruction, and so is loaded as part of the instruction. No load of a value from memory to a register is necessary, making the processing of an immediate value faster than processing a constant.

There are few real (native) operators in MIPS do not follow this 3-address format. The Jump (J) operator is the most obvious, but the branch operators and the store operators are also anomalous in that they drop the R_d operator. These will be covered later and the text, and the reason for the format will be obvious.

Other operations in MIPS will appear to not follow these formats. However these operations are pseudo operations, and do not exist in the real MIPS instruction set. Some of these pseudo operations, such as `move` and `li`, have already been seen in chapter 2. These operators do not require 3 addresses, but they are translated into real operations which do. This will be covered in this chapter.

Chapter 3. 2 Addition in MIPS assembly

Chapter 3.2. 1 Addition operators

There are 4 real addition operators in MIPS assembly. They are:

- `add` operator, which takes the value of the R_s and R_t registers containing integer numbers, adds the numbers, and stores the value back to the R_d register. The format and meaning are:

  format: `add` R_d, R_s, R_t
  meaning: $R_d \leftarrow R_s + R_t$

- `addi` operator, which takes the value of R_s and adds the 16 bit immediate value in the instruction, and stores the result back in R_t. The format and meaning are:

  format: `addi` R_t, R_s, Immediate
  meaning: $R_t \leftarrow R_s$ + Immediate

- `addu` operator, which is the same as the `add` operator, except that the values in the registers are assumed to be unsigned, or whole, binary numbers. There are no negative values, so the values run from $0..2^{32}-1$. The format and the meaning are the same as the `add` operator above:

  format: `addu` R_d, R_s, R_t
  meaning: $R_d \leftarrow R_s + R_t$

- `addiu` operator, which is the same as the `addi` operator, but again the numbers are assumed to be unsigned[8]:

  format: `addiu` R_t, R_s, Immediate
  meaning: $R_t \leftarrow R_s$ + Immediate

In addition to the real operators, there are a number of pseudo add operators, which are:

- `add` using a 16 bit immediate value. This is shorthand for the add operator to implement an addi operator. The same is principal applies for the `addu` if an immediate value is used, and the operator is converted into an `addiu`. The format, meaning, and translation of this instruction is:

  format: `add` R_t, R_s, Immediate
  meaning: $R_t \leftarrow R_s$ + Immediate
  translation: `addi` R_t, R_s, Immediate

[8] Note that the instruction addiu $t1, $t2, -100 is perfectly valid, but as a later example program will show, the results are not what you would expect. The value of -100 is converted into a binary number. When negative numbers are used, the behavior, while defined, is not intuitive. Unsigned numbers means only whole numbers, and when using unsigned numbers it is best to only use unsigned whole numbers.

- `add`, `addi`, `addu`, or `addiu` with a 32 bit immediate value. When considering this operator, it is important to remember that in the real I format instruction the immediate value can only contain 16 bits. So if an immediate instruction contains a number needing more than 16, the number must be loaded in two steps. The first step is to load the upper 16 bits of the number into a register using the Load Upper Immediate (`lui`) operator[9], and then to load the lower 16 bits using the using an Or Immediate (`ori`) operator. The addition is then done using the R instruction add operator. Thus the instruction:

  ```
  addi Rt, Rs, (32 bit) Immediate
  ```

  would be translated to:

  ```
  lui $at[10], (upper 16 bits) Immediate     #load upper 16 bits into $at
  ori $at, $at, (lower 16 bits) Immediate #load lower 16 bits into $at
  add Rt, Rs, $at[11]
  ```

The next section will show a program using add operations which will illustrate all of these operations.

Chapter 3.2. 2 Addition Examples

This section will implement and assemble examples of using the different formats of the add operator. Following the program will be a number of screen shots taken from MARS to provide a detailed discussion of the program.

```
# File:     Program3-1.asm
# Author:   Charles Kann
# Purpose:  To illustrate some addition operators

# illustrate R format add operator
li $t1, 100
li $t2, 50
add $t0, $t1, $t2

# illustrate add with an immediate.  Note that
# an add with a pseudo instruction translated
# into an addi instruction
addi $t0, $t0, 50
add $t0, $t0, 50

# using an unsign number.  Note that the
# result is not what is expected
# for negative numbers.
```

[9] The lui operator loads the upper 16 bits of a register with the high 16 bits in the immediate value.

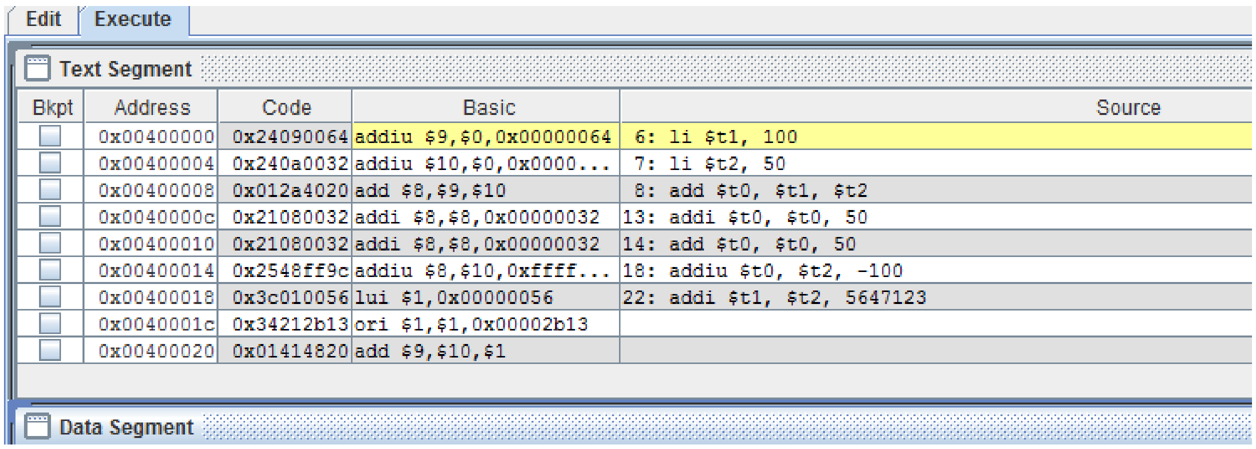
[10] The `$at` is the assembler reserved register. The programmer cannot access it directly, it is reserved for the assembler to use as scratch space, as is done here.

[11] Be careful when using immediate values, as they can be either numbers or bit strings. This can create confusion if you mix hex values with decimal numbers (e.g. 0xffff and -1). So the rule will be given that when using arithmetic (addition, subtraction, multiplication, and division), always use decimal numbers. When using logical operations (and, or, not, xor, etc), always use hex values. The problems with intermingling these will be explored in the problems at the end of the chapter.

```
addiu $t0, $t2, -100

# addition using a 32 immediate. Note that 5647123
# base 10 is 0x562b13
addi $t1, $t2, 5647123
```

Program 3-1: Addition Examples

To begin exploring this program, assemble the program and bring up the execute screen, as illustrated in the follow screen capture. One important detail in this screen capture is the representation of the program found in two columns. The first column is titled *Source*, and this column contains the program exactly as you entered it. The second column is titled *Basic*, and this contains the source code as it is given to the assembler. There are a number of changes between the original source code and the basic code. The basic code has had all of the register mnemonics changed to the register numbers. But the bigger change is that all of the instructions using pseudo operators have been changed into instructions using one or more real operators. For example, in line 14 which has the statement `add $t0, $t0, 50` , the `add` operator has been changed into the `addi` operator, as was discussed in the previous section. The same is true of line 22, where the addi $t1, $t2, 5647123 instruction has been covered to a `lui`, `ori`, and R format `add` instruction.



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
☐	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
☐	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
☐	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
☐	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
☐	0x00400014	0x2548ff9c	addiu $8,$10,0xffff...	18: addiu $t0, $t2, -100
☐	0x00400018	0x3c010056	lui $1,0x00000056	22: addi $t1, $t2, 5647123
☐	0x0040001c	0x34212b13	ori $1,$1,0x00002b13	
☐	0x00400020	0x01414820	add $9,$10,$1	

Data Segment

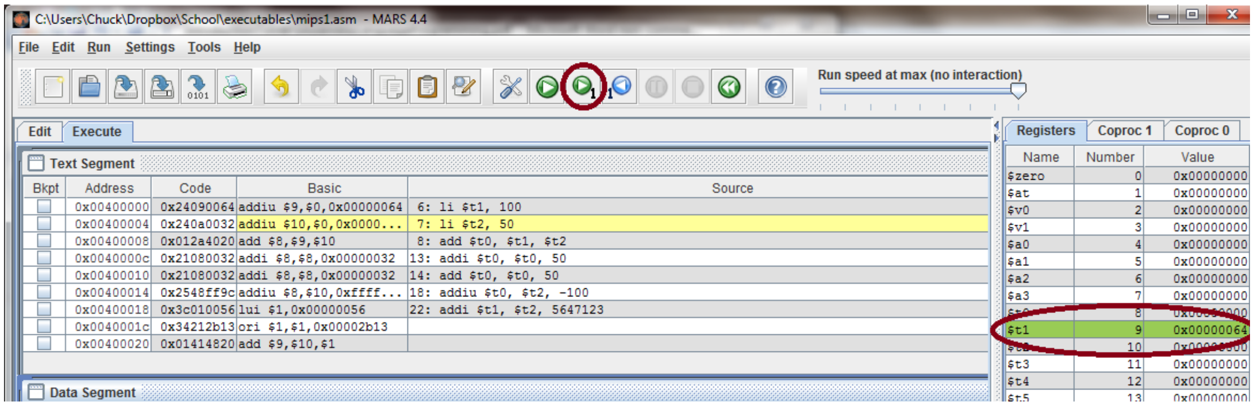
Figure 3-2: Assembled addition example

This output shows that some operators that were used in the previous chapter are actually pseudo operators. The `li` operator was suspect because it only had 2 address parameters, a source and a destination, and indeed it turns out not to be a real operator.

It is often useful to look at the Basic column in MARS to see how your source is actually presented to the assembler.

Next notice that in figure 3-2 the first line of the program is highlighted in yellow. This means that the program is ready to execute at the first line in the program. Clicking on the green arrow as shown in figure 3-3 the program has executed the first line of the program, and is waiting to run at the second line of the program. As a result of running the first line, the register `$t1` (`$9`) has been updated to contain the value 100 (0x64), which is shown in figure 3-3.

Continue running the program, and you will note that the next line to execute will always be highlighted in yellow, and the last register to be changed will be highlighted in green. When line 7 is run, line 8 is yellow and `$t2` (`$10`) contains the value 0x32 (50_{10}) and is highlighted in green. After the addition at line 8, line 13 is highlighted in yellow, and register `$t0` (`$8`) contains 0x96 (or 150_{10}). Continue to step through the program until line 18 highlighted and ready to run. At this point register `$t0` has the value 0xfa (250`10`). Once this statement is executed, the value in $t0 changes from 0xfa changes to 0xffffffce (-50_{10}), not 0x96 (150_{10}) as you might expect.


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.4
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Registers Coproc 1 Coproc 0
Text Segment
Data Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
☐	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
☐	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
☐	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
☐	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
☐	0x00400014	0x2548ff9c	addiu $8,$10,0xffff...	18: addiu $t0, $t2, -100
☐	0x00400018	0x3c010056	lui $1,0x00000056	22: addi $t1, $t2, 5647123
☐	0x0040001c	0x34212b13	ori $1,$1,0x00002b13	
☐	0x00400020	0x01414820	add $9,$10,$1	

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000064
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000

Figure 3-3: Addition Example after running 1 step.

Chapter 3.2. 3 Introduction to pseudo code

Writing a program in assembly language results in very large, complex programs which become hard to write, understand, and debug. HLL were designed to abstract away a large portion of the complexity introduced in writing assembly code. Even though all programs eventually require this level of complexity, a compiler is introduced to translate the relatively simpler structure of a HLL to the more complex assembly language code.

One way to write assembly code would then be to first write the code in a HLL, and perform the translation to assembly in a similar manner to a compiler. However HLL must be formally defined to allow a compiler work properly, and some of the translations implemented by a compiler are less than straight forward.

What is really needed is a language which can be used to specify the salient points of code in a higher level form, but with less formality so that the translation to assembly is relatively straight forward and simple. The language which is usually used for this is called *pseudo code*. As its name implies, pseudo code is not a formal language. Instead it is a very rough set of malleable concepts which can be used to produce an outline an assembly program. The language itself only includes enough detail to allow a programmer to understand what needs to done. How the program is actually implementated is left up to the programmer.

Pseudo code is also useful in that new concepts can be added to the language as needed, so long as the meaning of those constructs is clear to a programmer. This means that the language can be easily changed as the needs of the programmer change.

While there is no one formal definition of pseudo code, this text will give some conventions it will use. Consider the following pseudo code program to read two numbers, add them, and print the result back to the user.

```
main
{
    register int i = input("Please enter the first value to add: ");
    register int j = input("Please enter the second value to add: ");
    register int k = i + j;
    print("The result is " + k);
}
```

This program tells the assembly language programmer to create a program that will contain 3 integer values. The use of the `register` modifier on the `int` declaration tells the programmer that they should use a save register (`s0..s7`) if possible to maintain these values. If the `register` modifier is not used, the programmer has a choice to use memory or registers to store the values. In addition, there will be a `volatile` modifier used later, which will mean that the programmer must use a memory variable. Note that the `register` modifier is only a suggestion. Since the number of save registers is limited to 8, there is a possibility that one will not be available to the programmer, and the variable might need to be written to memory.

The next construct from this pseudo code that will be discussed is the `input` and `print`. These are purposefully made to look like HLL methods to be more easily understood by programmers. But the programmer can implement them in any manner they choose, as macros, as subprograms, or directly in the code. They tell the programmer that for the input, a prompt should be written to the console, and a value read from the user and stored. The print tells the program to write out a string and append the result of the addition.

This simple program will be translated into MIPS assembly in the next section. Note how much more complex the program becomes.

Chapter 3.2. 4 Assembly language addition program

The following assembly language program implements the pseudo code program from the last section. Comments on the code will following in the next section of the text.

```
# File name:      Program3-2.psc
# Author:   Charles Kann
# Purpose:  To illustrate how to translate a pseudo code
#               program into assembly
#
# Pseudo Code
#  main
#  {
#     register int i = input("Please enter the first value to add: ");
#     register int j = input("Please enter the second value to add: ");
#     register int k = i + j;
#     print("The result is " + k);
#  }

.text
.globl main
main:
        # Register conventions
        # i is $s0
        # j is $s1
        # k is $s2
        #  register int i =
        #   input("Please enter the first value to add: ");
        addi $v0, $zero, 4
        la $a0, prompt1
        syscall
        addi $v0, $zero, 5
        syscall
        move $s0, $v0

        # register int j =
        #      input("Please enter the second value to add: ");
        addi $v0, $zero, 4
        la $a0, prompt2
        syscall
        addi $v0, $zero, 5
        syscall
        move $s1, $v0

        # register int k = i + j;
        add  $s2, $s1, $s0

       # print("The result is " + k);
        addi $v0, $zero, 4
        la $a0, result
        syscall
        addi $v0, $zero, 1
        move $a0, $s2
        syscall

        #End the program
        addi $v0, $zero, 10
        syscall
.data
prompt1: .asciiz "Please enter the first value to add: "
prompt2: .asciiz "Please enter the second value to add: "
result:  .asciiz "The result is "
```

Chapter 3.2. 5 Assembly language addition program commentary

Since much of this program uses operators and concepts from previous sections, it does not require as many comments. However the following points are worth noting.

- Preamble comments are still important in any program. However it is a very good idea to include the pseudo code as part of the preamble, to show how the program was developed.
- The `move` operator is a strange operator with only two parameters. This seems strange, and would lead one to believe that it is probably a pseudo operator. You will be asked to show this in the exercises at the end of the chapter.

Chapter 3. 3 Subtraction in MIPS assembly

Subtraction in MIPS assembly is similar to addition with one exception. The `sub`, `subu and subui` behave like the `add`, `addu,` and `addui` operators. The only major difference with subtraction is that the `subi` is not a real instruction. It is implemented as a pseudo instruction, with the value to subtract loaded into the `$at` register, and then the R instruction `sub` operator is used. This is the only difference between addition and subtraction.

- `sub` operator, which takes the value of the R_s and R_t registers containing integer numbers, adds the numbers, and stores the value back to the R_d register. The format and meaning are:

format:	`sub` R_d, R_s, R_t
meaning:	R_d <- R_s - R_t

- `sub` pseudo operator, which takes the value of R_s, subtracts the 16 bit immediate value in the instruction, and stores the result back in R_t. The format, meaning, and translation are:

format:	`subi` R_t, R_s, `Immediate`
meaning:	R_t <- R_s - `Immediate`

translation:	`addi $at, $zero, Immediate`
	`sub` R_t, R_s, `$at`

- `subi` pseudo operator, which takes the value of R_s, subtracts the 16 bit immediate value in the instruction, and stores the result back in R_t. The format, meaning, and translation are:

format:	`subi` R_t, R_s, `Immediate`
meaning:	R_t <- R_s - `Immediate`

translation:	`addi $at, $zero, Immediate`
	`sub` R_t, R_s, `$at`

- `subu` operator, which is the same as the `add` operator, except that the values in the registers are assumed to be unsigned, or whole, binary numbers. There are no negative values, so the values run from 0..2^{32}-1. The format and the meaning are the same as the `add` operator above:

format:	`subu` R_d, R_s, R_t
meaning:	R_d <- R_s + R_t

- `subiu pseudo` operator, which is the same as the `addi` operator, but again the numbers are assumed to be unsigned:

format:	`subiu` R_t, R_s, `Immediate`
meaning:	R_t <- R_s + `Immediate`

translation:	`addi $at, $zero, Immediate`
	`subu` R_t, R_s, `$at`

In addition to the real operators, there are a number of pseudo sub operators, which use 32-bit immediate values. The 32-bit values are handled exactly as with the add instructions, with a sign extension out to 32 bits.

Chapter 3. 4 Multiplication in MIPS assembly

Multiplication and division are more complicated than addition and subtraction, and require the use of two new, special purpose registers, the `hi` and `lo` registers. The `hi` and `lo` registers are not included in the 32 general purpose registers which have been used up to this point, and so are not directly under programmer control. These sections on multiplication and addition will look at the requirements of the multiplication and division operations that make them necessary.

Multiplication is more complicated than addition because the result of a multiplication can require up to twice as many digits as the input values. To see this, consider multiplication in base 10. In base 10, 9x9=81 (2 one digit numbers yield a two digit number), and 99x99=9801 (2 two digit numbers yield a 4 digit number). As this illustrates, the results of a multiplication require up to twice as many digits as in the original numbers being multiplied. This same principal applies in binary. When two 32-bit numbers are multiplied, the result requires a 64-bit space to store the results.

Since multiplication of two 32-bit numbers requires 64-bits, two 32-bit registers are required. All computers require two registers to store the result of a multiplication, though the actual implementation of those two registers is different. It MIPS, the `hi` and `lo` registers are used, with the `hi` register being used to store the 32 bit larger part of the multiplication, and the `lo` register being used to the store the 32 bit smaller part of the multiplication.

In MIPS, all integer values must be 32 bits. So if there is a valid answer, it must be contained in the lower 32 bits of the answer. Thus to implement multiplication in MIPS, the two numbers must be multiplied using the `mult` operator, and the valid result moved from the `lo` register. This is shown in the following code fragment which multiplies the value in `$t1` by the value in `$t2`, and stores the result in `$t0`.

```
mult $t1, $t2
mflo $t0
```

However what happens if the result of the multiplication is too big to be stored in a single 32-bit register? Again consider base 10 arithmetic. 3*2=06, and the larger part of the answer is 0. However 3*6=18, and the larger part of the answer is non-zero. This is true of MIPS multiplication as well. When two positive numbers are multiplied, if the `hi` register contains nothing but 0's then there is no overflow, as the multiplication did not result in any value in the larger part of the result. If the hi register contains any values of 1, then the result of the multiplication did have an overflow, as part of the result is contained in the larger part of the result. This is shown in the two examples, 3*2=06, and 3*6=18, below.

```
          0011               0011
        * 0010             * 0110
     0000 0110          0001 0010
```

So a simple check for overflow when two positive numbers are multiplied to see if the `hi` register is all 0's: if it is all 0's the result did not overflow, otherwise the result did overflow.

This is fine for two positive or two negative number, but what if the input values are mixed? For example, 2*(-3) = -6, and 2*(-8) = -18. To understand what would happen, these problems will be implemented using 4-bit registers.

Remember that 4-bit registers can contain integer values from -8..7. So the multiplication of 2*(-3) and 2*(-6) in 4-bits with an 8-bit result is shown below:

```
          0010              0010
        * 1101            * 1010
     1111 1010         1110 1110
```

In the first example, the high 4-bits are 1111, which is the extension of the sign for -6. This says that the example did not overflow. In the second example, the high 4-bits are 1110. Since all 4 bits are not 1, they cannot be the sign extension of a negative number, and the answer did overflow. So an overly simplistic view might say that if the high order bits are all 0's or all 1's, there is no overflow. While this is a necessary condition to check for overflow, it is not sufficient. To see this, consider the result of 6*(-2).

```
          0010
        * 1010
     1111 0010
```

Once again, the high 4-bits are 1111, so it looks like there is not an overflow. But the difficulty here is that the low 4 bits show a positive number, so 111$\underline{1}$ indicates that the lowest 1 (the one underlined), is really part of the multiplication result, and not an extension of the sign. This result does show overflow. So to show overflow in a the result contained in the `hi` register must match all 0's or all 1's, and must match the high order (sign) bit of the `lo` register.

Now that the fundamentals of integer multiplication have been covered, there are five MIPS multiplication operators which will be looked at. They are:

- `mult` operator, which multiplies the values of R_s and R_t and saves it in the lo and hi registers. The format and meaning of this operator is:

  format: `mult` R_s, R_t
  meaning: `[hi,lo] <-` R_s * R_t

- `mflo` operator, which moves the value from the hi register into the R_d register. The format and meaning of this operator is:

  format: `mflo` R_d
  meaning: R_d `<- lo`

- `mfhi` operator, which moves the value from the hi register into the R_d register. The format and meaning of this operator is:

  format: `mfhi` R_d
  meaning: R_d <- hi

- `mult` operator, which multiples the values in R_s and R_t, and stores them in R_d. The format and meaning of this operator is:

  format: `mult` R_d, R_s, R_t
  meaning: R_d <- R_s * R_t

- `mulo` pseudo operator, which multiples the values in R_s and R_t, and stores them in R_d, checking for overflow. If overflow occurs an exception is raised, and the program is halted with an error. The format and meaning of this operator is:

  format: `mulo` R_d, R_s, R_t
  meaning: R_d <- R_s * R_t

- Note that both the `mult` and `mulo` operators have an immediate pseudo operator implementation. The format, meaning, and translation of the pseudo operators is as follows:

  format: `mult` R_d, R_s, Immediate
  meaning: R_d <- R_s * Immediate
  translation: `addi` \$$R_t$, \$zero, Immediate
  `mult` R_d, R_s, R_t

  format: `mulo` R_d, R_s, Immediate
  meaning: R_d <- R_s * Immediate
  translation: `addi` \$$R_t$, \$zero, Immediate
  `mulo` R_d, R_s, R_t

Chapter 3. 5 Division in MIPS Assembly

Division, like multiplication requires two registers to produce an answer. The reason for this has to do with how the hardware calculates the result, and is harder to explain without considering the hardware used, which is beyond the scope of this textbook. The reader is thus asked to just believe that two registers are needed, and that MIPS will again use the registers `hi` and `lo`.

To understand division, we will again begin with a base 10 example. Remember how division was done when it was introduced to you in elementary school. The result of 17 divided by 5 would be the following:

$$\begin{array}{r|l} & 3\ r2 \\ \hline 5 & 17 \end{array}$$

In this equation, the value 5 is called the divisor, the 17 is the dividend, 3 is the quotient, and 2 is the remainder.

In MIPS, when integer division is done, the `lo` register will contain the quotient, and the `hi` register will contain the remainder. This means that in MIPS integer arithmetic when the quotient is taken from the low register the results will be truncated. If the programmer wants to round the number, this can be implemented using the remainder (see problems at the end of Chapter 5).

Now that the fundamentals of integer division have been covered, there are two MIPS division operators that will be looked at. They are:

- div operator, which has 3 formats. The first format is the only real format of this operator. The operator divides R_s by R_t and stores the result in the [`hi`,`lo`] register pair with the quotient in the `lo` and the remainder in the `hi`. The format and meaning of this operator is:

  format: $\texttt{div}\ R_s,\ R_t$
  meaning: $\texttt{[hi,lo]} \leftarrow R_s\ /\ R_t$

  The second format of the div operator is a pseudo instruction. It is a 3 address format, but is still a pseudo instruction. In addition to executing a division instruction, this pseudo instruction also checks for a zero divide. The specifics of the check will not be covered at this point as it involves `bne` and `break` instructions. The format, meaning, and translation of the pseudo operators is as follows:

  format: $\texttt{div}\ R_d,\ R_s,\ R_t$
  meaning: $\texttt{[if}\ R_t\ \texttt{!= 0]}\ R_d \leftarrow R_s\ /\ R_t$
  $\texttt{else break}$
  translation: $\texttt{bne}\ R_t,\ \texttt{\$zero, 0x00000001}$
  $\texttt{break}$
  $\texttt{div}\ R_s,\ R_t$
  $\texttt{mflo}\ R_d$

  The third format of the div operator is a pseudo instruction. The format, meaning, and translation of the pseudo operators is as follows:

  format: $\texttt{div}\ R_d,\ R_s,\ \texttt{Immediate}$
  meaning: $R_d \leftarrow R_s\ /\ \texttt{Immediate}$
  translation: $\texttt{addi \$}R_t\texttt{, \$zero, Immediate}$
  $\texttt{div}\ R_s,\ R_t$
  $\texttt{mflo}\ R_d$

- `rem` (remainder) operator, which has 2 formats. There are only pseudo formats for this instruction. The first format of the `rem` operator is a pseudo instruction. The format, meaning, and translation of the pseudo operators is as follows:

  format: `div` R_d, R_s, R_t
  meaning: `[if` R_t `!= 0]` R_d `<-` R_s `%` R_t `else break`
  translation:
  ```
  bne Rt, $zero, 0x00000001
  break
  div Rs, Rt
  mfhi Rd
  ```

  The second format of the `rem` operator is also a pseudo instruction. The format, meaning, and translation of the pseudo operators is as follows:

  format: `rem` R_d, R_s, `Immediate`
  meaning: R_d `<-` R_s `/ Immediate`
  translation:
  ```
  addi $Rt, $zero, Immediate
  div Rs, Rt
  mfhi Rd
  ```

Chapter 3.5. 1 Remainder operator, even/odd number checker

Some students question the usefulness of the remainder operator, but a number of interesting algorithms are based on it. For example, this section presents a short program which checks if a number is odd or even. Note that since branching has not yet been covered, this program will print out a 0 if the number is even, and 1 if the number is odd. Because branching has not yet been covered, the program will print out 0 if the number is even, and 1 if it is odd.

The algorithm checks the remainder of a division by 2. If the number is evenly divisible by 2, the remainder will be 0 and the number is even. If the number is not evenly divisible by 2, the remainder is 1 and the number is odd. The pseudo code for this algorithm uses the "%" or modulus operator to obtain the remainder.

```
main
{
    int i = prompt("Enter your number");
    int j = i % 2;
    print("A result of 0 is even, a result of 1 is odd: result = " + j;
}
```

Chapter 3.5. 2 Remainder operator, even/odd number checker

The following is the MIPS implementation of the even/odd checker. To find the remainder the div operator is used to divide by 2 and the remainder retrieved from the `hi` register. If the remainder is 0 the number is even, and 1 if it is odd.

```
# File:      Program3-3.asm
# Author:    Charles W. Kann
# Purpose:   To have a user enter a number,and print 0 if
#            the number is even, 1 if the number is odd
.text
.globl main
```

```
      main:
         # Get input value
         addi $v0, $zero, 4   # Write Prompt
         la $a0, prompt
         syscall
         addi $v0, $zero, 5   # Retrieve input
         syscall
         move $s0, $v0

         # Check if odd or even
         addi $t0, $zero, 2   # Store 2 in $t0
         div $t0, $s0, $t0    # Divide input by 2
         mfhi $s1             # Save remainder in $s1

         # Print output
         addi $v0, $zero, 4   # Print result string
         la $a0, result
         syscall
         addi $v0, $zero, 1   # Print result
         move $a0, $s1
         syscall

         #Exit program
         addi $v0, $zero, 10
         syscall

      .data
      prompt: .asciiz "Enter your number: "
result: .asciiz "A result of 0 is even, 1 is odd: result =  "
```

Program 3-2: Even/odd number checking program

Chapter 3. 6 Solving arithmetic expressions in MIPS assembly

Using the MIPS arithmetic operations covered so far, a program can be created to solve equations. For example the following pseudo code program, where the user is prompted for a value of x and the program prints out the results of the equation $5x^2 + 2x + 3$.

```
main
{
     int x = prompt("Enter a value for x: ");
     int y = 5 * x * x + 2 * x + 3;
     print("The result is: " + y);
}
```

This program is implemented in MIPS below.

```
# File:         Program3-3.asm
# Author:       Charles Kann
# Purpose:      To calculate the result of 5*x^ + 2*x + 3

.text
.globl main
main:
   # Get input value, x
   addi $v0, $zero, 4
```

```
      la $a0, prompt
      syscall
      addi $v0, $zero, 5
      syscall
      move $s0, $v0

      # Calculate the result of 5*x*x + 2* x + 3 and store it in $s1.
      mul $t0, $s0, $s0
      mul $t0, $t0, 5
      mul $t1, $s0, 2
      add $t0, $t0, $t1
      addi $s1, $t0, 3

      # Print output
      addi $v0, $zero, 4      # Print result string
      la $a0, result
      syscall
      addi $v0, $zero, 1      # Print result
      move $a0, $s1
      syscall

      #Exit program
      addi $v0, $zero, 10
      syscall

   .data
   prompt: .asciiz "Enter a value for x: "
   result: .asciiz "The result is: "
```

Program 3-3: Program to calculate 5*x^2 + 2*x + 3

Chapter 3. 7 Division and accuracy of an equation

One thing to always keep in mind when using division with integers in any language (including Java, C/C++, etc) is that the results are truncated. This can lead to errors and different answers depending on the order of evaluation of the terms in the equation. For example, most 5th graders knows that "(10/3) *3 = 10", as the 3's should cancel. However in integer arithmetic the result of "10/3 = 3", and so "(10/3) *3 = 9" (not 10). However if you reverse the order of the operations you will find that "(10*3) / 3 = 10". This is shown in the following program.

```
# File:      Program3-4.asm
# Author:    Charles Kann
# Purpose:   To show the difference in result if
#            ordering of multiplication and division
#            are reversed.

.text
.globl main
main:
    addi $s0, $zero, 10 # Store 10 and 3 in registers $s0 and $s1
    addi $s1, $zero, 3

    div $s2, $s0, $s1   # Write out (10/3) * 3
```

```
        mul $s2, $s2, $s1
        addi $v0, $zero, 4
        la $a0, result1
        syscall
        addi $v0, $zero, 1
        move $a0, $s2
        syscall

        mul $s2, $s0, $s1 # Write out (10*3)/3
        div $s2, $s2, $s1
        addi $v0, $zero, 4
        la $a0, result2
        syscall
        addi $v0, $zero, 1
        move $a0, $s2
        syscall

        addi $v0, $zero, 10 #Exit program
        syscall

    .data
    result1: .asciiz "\n(10/3)*3 = "
    result2: .asciiz "\n(10*3)/3 = "
```

Program 3-4: Program to show order of operations matters

There are times (for example, when calculating a parent node in a Complete Binary Tree, which is covered in most textbooks on Data Structures) that division using truncation is desired. However, in general the simple rule to follow is when there is a mix of operations including division with integer numbers, code the expression to do the multiplication, addition, and subtraction before doing any division. This preserves the greatest accuracy of the result. This principal is true of integer arithmetic in any language, including any HLL.

Chapter 3. 8 Logical operators

Most programmers have seen logical operators in HLL for Boolean data types. As was covered earlier in this text, there are often two form of these operators, the logical (or short-circuiting) operators and the bitwise operators. MIPS only implements the bitwise operators, but they are called *logical* operators. Since all Boolean operations can be implemented with only 3 operations, the AND, OR, and NOT operations, this sections will present these 3 operations. In addition the XOR operation will be included for convenience because it is often easier to state a logical expression using XOR than to do so using AND, OR, and NOT. See the problems at the end of the chapter to see how to implement logical operations using these 4 basic logic operations.

The logic operators covered are:

- `and` operator, which has three formats. The first format is the only real format of this operator. The operator does a bit-wise AND of R_s and R_t and stores the result R_d register. The format and meaning of this operator is:

format:	and R_d, R_s, R_t
meaning:	R_d <- R_s AND R_t

The second format of the `and` operator is a pseudo instruction. In this case the 3rd operator is immediate value, and so this is just a short hand for implementing the `andi` operator. The format, meaning, and translation of the pseudo operators is as follows:

format:	and R_t, R_s, Immediate
meaning:	R_t <- R_s AND Immediate
translation:	andi R_t, R_s, Immediate

The third format of the `and` operator is also a pseudo instruction, and strange in that only logical operators have this format. In this instruction, R_s and R_t are assumed to be the same register, and the 3rd operator is immediate value The format, meaning, and translation of the pseudo operators is as follows:

format:	and R_s, Immediate
meaning:	R_s <- R_s AND Immediate
translation:	andi R_s, R_s, Immediate

- `andi` operator. The operator does a bit-wise AND of R_s and an immediate value, and stores the result Rt register. The format and meaning of this operator is:

format:	andi R_t, R_s, Immediate
meaning:	R_t <- R_s AND Immediate

The shorthand with a single register also applies to the `andi` instruction.

format:	andi R_s, Immediate
meaning:	R_s <- R_s AND Immediate
translation:	andi R_s, R_s, Immediate

- `or` operator, which has three formats. The first format is the only real format of this operator. The operator does a bit-wise OR of R_s and R_t and stores the result R_d register. The format and meaning of this operator is:

format:	or R_d, R_s, R_t
meaning:	R_d <- R_s OR R_t

The second format of the `and` operator is a pseudo instruction. In this case the 3rd operator is immediate value, and so this is just a short hand for implementing the `ori` operator. The format, meaning, and translation of the pseudo operators is as follows:

format:	or R_t, R_s, Immediate
meaning:	R_t <- R_s OR Immediate
translation:	ori R_t, R_s, Immediate

The shorthand with a single register also applies to the `or` instruction.

format:	`or` R_s`, Immediate`
meaning:	R_s `<-` R_s `OR Immediate`
translation:	`andi` R_s`,` R_s`, Immediate`

- `ori` operator. The operator does a bit-wise OR of R_s and an immediate value, and stores the result Rt register. The format and meaning of this operator is:

format:	`ori` R_t`,` R_s`, Immediate`
meaning:	R_t `<-` R_s `OR Immediate`

The shorthand with a single register also applies to the `ori` instruction.

format:	`ori` R_s`, Immediate`
meaning:	R_s `<-` R_s `OR Immediate`
translation:	`ori` R_s`,` R_s`, Immediate`

- `xor` operator, which has three formats. The first format is the only real format of this operator. The operator does a bit-wise OR of R_s and R_t and stores the result R_d register. The format and meaning of this operator is:

format:	`xor` R_d`,` R_s`,` R_t
meaning:	R_d `<-` R_s `XOR` R_t

The second format of the `and` operator is a pseudo instruction. In this case the 3rd operator is immediate value, and so this is just a short hand for implementing the `ori` operator. The format, meaning, and translation of the pseudo operators is as follows:

format:	`xor` R_t`,` R_s`, Immediate`
meaning:	R_t `<-` R_s `XOR Immediate`
translation:	`xori` R_t`,` R_s`, Immediate`

The shorthand with a single register also applies to the or instruction.

format:	`xor` R_s`, Immediate`
meaning:	R_s `<-` R_s `AND Immediate`
translation:	`xori` R_s`,` R_s`, Immediate`

- `ori` operator. The operator does a bit-wise OR of R_s and an immediate value, and stores the result Rt register. The format and meaning of this operator is:

format:	`xori` R_t`,` R_s`, Immediate`
meaning:	R_t `<-` R_s `XOR Immediate`

The shorthand with a single register also applies to the `xori` instruction.

format:	`xori` R_s`, Immediate`
meaning:	R_s `<-` R_s `XOR Immediate`
translation:	`xori` R_s`,` R_s`, Immediate`

- `not` operator. The operator does a bit-wise NOT (bit inversion) of R_s, and stores the result Rt register. The format and meaning of this operator is:

format:	not R_s, R_t
meaning:	R_s <- NOT(R_t)
translation:	nor R_s, R_t, $zero

In addition to the real operators, there are a number of pseudo sub operators, which use 32-bit immediate values. The 32-bit values are handled exactly as with the add instructions, with a sign extension out to 32 bits.

Chapter 3. 9 Using logical operators

Most students have only seen logical operators used as branch conditions in `if`, `while`, and `for` statements, and so they cannot see why these operators are useful. This section will show some uses of these operators.

Chapter 3.9. 1 Storing immediate values in registers

In MARS, the `li` instruction is translated into an "addui R_d, $zero, Immediate" instruction. This is perfectly valid, however when doing addition there always has to be propagation of a carry-bit, and so addition can take a relatively long time (O(logn)..O(n)). However the OR operation is a bit-wise operation and does not have any carry bits, and can be executed in O(1) time. In MARS the OR is not faster than addition, but in some architectures the OR is in fact faster than addition.

If the OR is faster than addition, the `li` instruction can be implemented as "ori R_d, $zero, Immediate" with some advantage. Be aware that an ori is not the same as an addui. This difference is shown in problem 14 at the end of the chapter.

Chapter 3.9. 2 Converting a character from upper case to lower case

Chapter 1 pointed out that in ASCII the difference between upper case and lower case letters is the 0x20 bit. So an upper case letter can be converted to lower case by setting this bit to 1, and an upper case letter can be set to a lower case letter by turning this bit off. One of the exercises in Chapter 1 asked for an implementation of this program in a HLL. Many novice programmers see this problem and realize that convert uppercase letters to lowercase, they can add 0x20. The problem is that if the letter is already uppercase, the result is no longer a letter.

To correctly handle converting from uppercase to lowercase, the letters should be OR'ed with 0x20 (converting from lowercase to uppercase is left as an exercise at the end of the chapter). If this is done, letters which are already lowercase are not changed. This is illustrated in the following program.

```
# File:     Program3-5.asm
# Author:   Charles Kann
# Purpose:  This program shows that adding 0x20 to a character
#           can result in an error when converting case, but
#           or'ing 0x20 always works.
```

```
.text
.globl main
main:
    # Show adding 0x20 only works if the character is upper case.
    ori $v0, $zero, 4
    la $a0, output1
    syscall
    ori $t0, $zero, 0x41# Load the character "Z"
    addi $a0, $t0, 0x20 # Convert to "z" by adding
    ori $v0, $zero, 11  # Print the character
    syscall

    ori $v0, $zero, 4
    la $a0, output2
    syscall
    ori $t0, $zero,0x61 # Load the character "z"
    addi $a0, $t0, 0x20 # Attempt to convert to lower case
    ori $v0, $zero, 11  # Print the character, does not work
    syscall

    # Show or'ing 0x20 works if the character is upper or lower case.
    ori $v0, $zero, 4
    la $a0, output1
    syscall
    ori $t0, $zero, 0x41# Load the character "Z"
    ori $a0, $t0, 0x20 # Convert to "z" by adding
    ori $v0, $zero, 11  # Print the character
    syscall

    ori $v0, $zero, 4
    la $a0, output1
    syscall
    ori $t0, $zero,0x61 # Load the character "z"
    ori $a0, $t0, 0x20 # Attempt to convert to lower case
    ori $v0, $zero, 11  # Print the character, does not work
    syscall

    ori $v0, $zero, 10  # Exit program
    syscall

.data
output1: .asciiz "\nValid conversion: "
output2: .asciiz "\nInvalid conversion, nothing is printed: "
```

Program 3-5: Letter case conversion in assembly

Chapter 3.9. 3 Reversible operations with XOR

Often it is nice to be able to invert the bits in a value in a way that they are easily translated back to the original value. Inverting the bits and restoring them is shown in the program below. One use of this is to swap two values without using a temporary storage space, is given at the end of the chapter.

```
# File:     Program3-6.asm
# Author:   Charles Kann
```

```
#Purpose:   To show the XOR operation is reversible
.text
.globl main
main:
    ori $s0, $zero, 0x01234567 # the hex numbers

    # Write out the XOR'ed value
    la $a0, output1
    li $v0, 4
    syscall
    xori $s0, $s0,  0xffffffff # the results in $t1 will be fedcba98
    move $a0, $s0
    li $v0, 34
    syscall

    # Show the original value has been restored.
    la $a0, output2
    li $v0, 4
    syscall
    xori $s0, $s0,  0xffffffff # the results in $t1 will be fedcba98
    move $a0, $s0
    li $v0, 34
    syscall

    ori $v0, $zero, 10   # Exit program
    syscall
.data
    output1: .asciiz "\nAfter first xor:  "
    output2: .asciiz "\nAfter second xor: "
```

Chapter 3. 10 Shift Operations

The final topic covered in this chapter is the shift operations. Shift allow bits to be moved around inside of a register. There are many reasons to do this, particularly when working with low level programs such as device drivers. The major reason this might be done in a HLL is to multiplication and division, but as was stated earlier, multiplication and division using constants should not be implemented by the programmer, as the compiler will automatically generate the best code for the situation. So these operations are difficult to justify in terms of higher level languages. These operations will simply be presented here with an example of multiplication and division by a constant. The reader will have to trust that these operators are useful in assembly and will be used in future chapters.

There are 2 directions of shifts, a right shift and a left shift. The right shift moves all bits in a register some specified number of spaces to the right, and the left shift moves bits some specified number of spaces to the left.

The bits that are used to fill in the spaces as the shift occurs are determined by the type of the shift. A logic shift uses zeros (0) to replace the spaces.

The arithmetic shift replaces the spaces with the high order (left most) bit. In an integer the high order bit determines the sign of the number, so shifting the high order keeps the correct sign for the number[12]. Note that the sign only matters when doing a right shift, so only the right shift will have an arithmetic shift.

Finally, there is a circular shift (called a rotate in MIPS) that shifts in the bit that was shifted out on the opposite side of the value. For example, when the 4 bit value 0011 is rotated right, a 1 is returned and this 1 is shifted into the right most bit, giving 1001. Likewise, when the 4 bit value 0011 is rotated left, a 0 is returned and this 0 is shifted into the right most bit, giving 0110.

The main use of a rotate command is to allow the programmer to look at each of the 32 bits in a value one at a time, and at the end the register contains the original value. The `rol` and `ror` operators (rotate left/right) are pseudo operators, and a circular shift can be implemented using combinations of left and right logical shifts, and an OR operation. This will be explored in the questions at the end of the chapter.

The following examples show how each of these shifts work.

```
Shift left logical 2 spaces: 0x00000004 -> 0x00000010
Shift right logical 3 spaces: 0x0000001f -> 0x0000003
Shift right arithmetic 3 spaces: 0x0000001f -> 0x00000003
Shift right arithmetic 2 spaces: 0xffffffe1 -> 0xfffffff8
Rotate right 2 spaces: 0xffffffe1 -> 0x7ffffff8
Rotate left 2 space: 0xffffffe1 -> 0xffffff87
```

The program in the following section will implement each of these operations to show how they work.

The following are the shift operations provided in MIPS.

- `sll` (shift left logical) operator. The operator shifts the value in R_t shift amount (shamt) bits to the left, replacing the shifted bits with 0's, and storing the results in R_d. Note that the registers R_d and R_t are used. The numeric value in this instruction is not an immediate value, but a shift amount. Shift values are limited to the range 0..31 in all of the following instructions.

  format: `sll` R_d, R_t, shamt
  meaning: R_d <- R_t << shamt

- `sllv` (shift left logical variable) operator. The operator shifts the value in R_t bits to the left by the number in R_s, replacing the shifted bits with 0's. The value in R_s should be limited to the range 0..31, but the instruction will run with any value.

  format: `sllv` R_d, R_t, R_s
  meaning: R_d <- R_t << R_s

- `srl` (shift right logical) operator. The operator shifts the value in R_t shift amount (shamt) bits to the right, replacing the shifted bits with 0's, and storing the results in R_d.

[12] This is why it is in some sense incorrect to call the high order bit of an integer a "sign bit". To do division the high order (left-most) bit is replicated to the right. If this high order bit was just a sign bit, it would not be replicated, as the one sign bit only specifies the sign.

format: srl R_d, R_t, shamt
meaning: R_d <- R_t >> shamt

- srlv (shift right logical variable) operator. The operator shifts the value in R_t bits to the right by the number in R_s, replacing the shifted bits with 0's. The value in R_s should be limited to the range 0..31, but the instruction will run with any value.

format: srlv R_d, R_t, R_s
meaning: R_d <- R_t >> R_s

- sra (shift right arithmetic) operator. The operator shifts the value in R_t shift amount (shamt) bits to the right, replacing the shifted bits with sign bit for the number, and storing the results in R_d.

format: sra R_d, R_t, shamt
meaning: R_d <- R_t >> shamt

- srav (shift right arithmetic variable) operator. The operator shifts the value in R_t bits to the right by the number in R_s, replacing the shifted bits the sign bit for the number. The value in R_s should be limited to the range 0..31, but the instruction will run with any value.

format: srla R_d, R_t, R_s
meaning: R_d <- R_t >> R_s

- rol (rotate left) pseudo operator. The operator shifts the value in R_t shift amount (shamt) bits to the right, replacing the shifted bits with the bits that were shifted out, and storing the results in R_d.

format: sra R_d, R_t, shamt
meaning: R_d[shamt..0] <- R_t[31..31-shamt+1],
R_d[31..shamt] <- R_t[31-shamt..0],
translation:
```
srl $at, $Rt, shamt
sll $Rd, $Rt, shamt
or $Rd, $Rd,  $at
```

- rolr (rotate fight) pseudo operator. The operator shifts the value in R_t shift amount (shamt) bits to the right, replacing the shifted bits with the bits that were shifted out, and storing the results in R_d.

format: sra R_d, R_t, shamt
meaning: R_d[31-shamt..shamt] <- R_t[31..shamt],
R_d[31..31-shamt+1] <- R_t[shamt-1..0],
translation:
```
srl $at, $Rt, shamt
sll $Rd, $Rt, shamt
or $Rd, $Rd,  $at
```

Chapter 3.10. 1 Program illustrating shift operations

The following program illustrates the shift operations from the previous section.

```
# File:     Program3-6.asm
# Author:   Charles Kann
# Purpose:  To illustrate various shift operations.

.text
.globl main
main:
   #SLL example
   addi $t0, $zero, 4
   sll  $s0, $t0, 2
   addi $v0, $zero, 4
   la $a0, result1
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

   #SRL example
   addi $t0, $zero, 16
   srl  $s0, $t0, 2
   addi $v0, $zero, 4
   la $a0, result2
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

   #SRA example
   addi $t0, $zero, 34
   sra  $s0, $t0, 2
   addi $v0, $zero, 4
   la $a0, result3
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

   #SRA example
   addi $t0, $zero, -34
   sra  $s0, $t0, 2   # sra 2 bits, which is division by 4
   addi $v0, $zero, 4 # Output the result
   la $a0, result4
   syscall
   addi $v0, $zero, 1
   move $a0, $s0
   syscall

   #rol example
   ori $t0, $zero, 0xffffffe1
   ror $s0, $t0, 2
   li $v0, 4
   la $a0, result6
```

```
        syscall
        li $v0, 34
        move $a0, $s0
        syscall

        #rol example
        ori $t0, $zero, 0xffffffe1
        rol $s0, $t0, 2
        li $v0, 4
        la $a0, result6
        syscall
        li $v0, 34
        move $a0, $s0
        syscall

        addi $v0, $zero, 10 # Exit program
        syscall

.data
result1: .asciiz "\nshift left logical 4 by 2 bits is "
result2: .asciiz "\nshift right logical 16 by 2 bits is "
result3: .asciiz "\nshift right arithmetic 34 by 2 bits is "
result4: .asciiz "\nshift right arithmetic -34 by 2 bits is "
result5: .asciiz "\nrotate right 0xffffffe1 by 2 bits is "
result6: .asciiz "\nrotate left 0xffffffe1 by 2 bits is "
```

Program 3-6: Program illustrating shift operations

Chapter 3. 11 Summary

This chapter was concerned with MIPS operators and how to use these operators in instructions. Some simple programs were given to illustrate the operators, but they were very simplistic in that they only used sequences, as branches and loops are not yet covered. In addition, only register memory was used, which greatly limited the types of programs that could be written. These limitations will be dealt with in subsequent chapters.

Chapter 3. 12 Exercises

1) Which of the following operators are pseudo operators? What are the translations of the pseudo operators to real instructions?
```
a) add $t0, $t1, $t2
b) add $t0, $t1, 100
c) addi $t0, $t1, 100
d) subi $t0, $t1, 100
e) mult $t0, $t1
f) mult $t0, $t1, $t2
g) rol $t0, $t1, 3
```

2) Implement a program to do a bitwise complement (NOT) of an integer number entered by the user. You should use the XOR operator to implement the NOT, do not use the NOT operator. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

3) Implement a program to calculate the 2's complement of a number entered by the user. The program should only user the XOR and ADD operators. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

4) Do the following two problems.
   a) Implement a simple program to do a bitwise NAND in MARS. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.
   b) Implement the AND, OR, and NOT operations using only the MIPS `nor` operator. Do the same thing using NAND.

5) Implement a program which multiplies a user input by 10 using only bit shift operations and addition. Check to see if your program is correct by using the `mult` and `mflo` operators. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

6) Implement a program which multiplies a user input by 15 using only bit shift operations and addition. Check to see if your program is correct by using the `mult` and `mflo` operators. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

7) Implement a program which divides a user input by 8 using only bit shift. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

8) Can you implement division by 15 using bit shift operations? Why or why not?

9) Implement an overflow check for multiplication where the two numbers being multiplied are both always positive. Why is this simpler than the generic check implement in the `mulo` operator?

10) Assemble a `mulo` instruction in MARS. Explain the resultant code in the Basic column of the Execute screen.

11) Implement the `rem` operator using only the `div`, `mfhi`, and `mflo` operators.

12) Implement a program to prompt the user for two numbers, the first being any number and the second a prime number. Return to the user a 0 if the second number is a prime factor for the first, or any number if it is not. For example, if the user enter 60 and 5, the program returns 0. If the user enters 62 and 5, the program returns 2.

13) Write programs to evaluate the following expressions. The user should enter the variables, and the program should print back an answer. Prompt the user for all variables in the expression, and print the results in a meaningful manner. **The results should be as accurate as possible.**

```
a) 5x + 3y +z

b) ((5x + 3y + z) / 2) * 3
```

c) $x^3 + 2x^2 + 3x + 4$

d) (4x / 3) * y

14) Enter the following two instructions in MARS, and assemble them. What differences do you notice? Are the results the same? Will one run faster than the other? Explain what you observe.

```
addiu $t0, $zero, 60000
ori   $t0, $zero, 60000
```

15) Write a program to retrieve two numbers from a user, and swap those number using only the XOR operation. You should not use a temporary variable to store the numbers while swapping them. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

16) Using only sll and srl, implement a program to check if a user input value is even or odd. The result should print out 0 if the number is even or 1 if the number is odd. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

17) The ideal gas law allows the calculation of volume of a gas given the pressure(P), amount of the gas (n), and the temperature (T). The equation is:

$$V = nRT / P$$

Since we only have used integer arithmetic, all numbers will be integer values with no decimal points. The constant R is 8.314 and will be specified as (8314/1000). This gives the same result. Implement the idea gas law program where the user is prompted for and enters values for n, T, and P, and V is calculated and printed out. Be careful to implement an accurate version of this program. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

18) Explain the different between a constant value and an immediate value.

19) Correct the following programs.

Program 1

```
.text
main:
    li $v0, 4
    la $a0, result1
    syscall
    li $v0, 1
    li $a0, 4
    syscall

    li $v0, 4
    la $a0, result2
```

```
        syscall
        li $v0, 1
        li $a0, 8
        syscall

        addi $v0, $zero, 10 #Exit program
        syscall
    .data
    result1: .ascii "\nfirst value = "
    result2: .ascii "\nsecond value = "
```

Program 2

```
    .text
    main:
        li $v0, 4
        la $a0, result1
        syscall
        li $v0, 4
        li $a0, 4
        syscall

        li $v0, 4
        la $a0, result2
        syscall
        li $v0, 1
        li $a0, 8
        syscall

        addi $v0, $zero, 10 #Exit program
        syscall
    .data
    result1: .asciiz "\nfirst value = "
    result2: .asciiz "\nsecond value = "
```

Program 3

```
    .text
    main:
        li $v0, 4
        la $a0, result1
        syscall
        li $v0, 4
        li $a0, 4
        syscall

        li $v0, 4
        la $a0, result2
        syscall
        li $v0, 1
        li $a0, 8
        syscall

        addi $v0, $zero, 10 #Exit program
        syscall
    result1: .ascii "\nfirst value = "
    result2: .ascii "\nsecond value = "
```

What you will learn.

In this chapter you will learn:

1. two of the three formats for MIPS instructions.
2. what op codes and functions are for MIPS operations
3. how to format Immediate (I) and Register (R) instructions in machine code.
4. how to represent MIPS instructions in hexadecimal.
5. how to use MARS to check your translations from assembly language to machine code.

Chapter 4 Translating Assembly Language into Machine Code

Everything in a computer is a binary value. Numbers are binary, characters are binary, and even the instructions to run a program are binary. Therefore the assembly language that has been presented cannot be what the computer understands. This assembly code must be converted to binary values. These binary values are called machine code. This chapter will explain how to convert the assembly instructions that have been covered so far into machine code.

Chapter 4. 1 Instruction formats

The MIPS computer is organized in a 3-address format. Generally all instructions will have 3 addresses associated with them. For example, the instruction "`ADD` R_d, R_s, R_t" uses three registers, the first address is the destination of the result, and the second and third are the two inputs to the ALU. The instruction "ADDI R_t, R_s, immediate" also uses three addresses, but in this case the third address is an immediate value.

In MIPS there are only 3 ways to format instructions. They are the R-format (register), the I-format (immediate), and the J-format (jump). This chapter will only cover R-format and I-format instructions. The R-format and I-format are shown below.

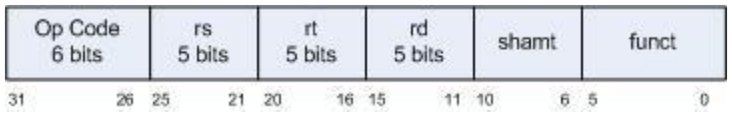


Figure 4-1: R format instruction

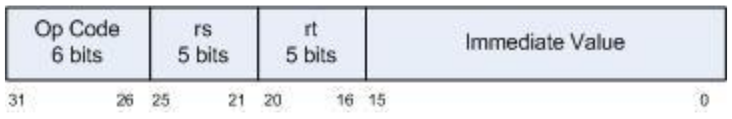


Figure 4-2: I format instruction

The R format instruction is used when the input values to the ALU come from two registers. The I format instruction is used when the input values to the ALU come from one register and an immediate value.

The fields in these instructions are as follows:

- Op-code - a 6 bit code which specifies the operation. These codes can be found for each instruction on the MIPS Green Sheet found at:

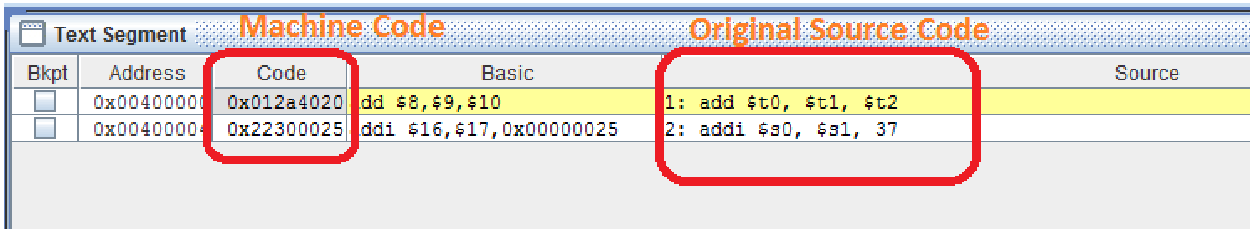
  http://freepdfs.net/mips-green-sheet-pdf/0567a5e7d6d4a3559f2a2139f5cd8d64/.

  Op-codes these are stored as a 2 digit number. The first number is 2 bits wide, and has values from 0-2. The second number is 4 bits wide, and has values from 0x0 to 0xf. So for example the `lw` instruction has an op code of 23, or 100011_2. The `addi` has an op code of 8, or 001000 (if the first digit is not specified, it is zero).

  All R-format operators have an op-code of 0, and are specified as an op-code/function. So the `add` op-code is specified by an op-code/function of 0/20. The operation to be performed by the ALU is contained in the function, which is the last field in the R instruction. Functions, like op-codes, are specified as a 2 bit number and a 4 bit number.

- rs - This is the first register in the instruction, and is always an input to the ALU. Note that rs, rt, and rd are 5 bits wide, which allows the addressing of the 32 general purpose registers.
- rt - This is the second register in the instruction. It is the second input to the ALU when there is an rd specified in the instruction (as in ADD) and the destination register when rd is not specified in the instruction (as in ADDI)
- rd - rd always is the destination register when it is specified.
- shamt - The number of bits to shift if the operation is a shift operation, otherwise it is 0. It is 5 bits wide, which allows for shifting of all 32 bits in the register.
- funct - For an R type instruction, this specifies the operation for the ALU.
- immediate - The immediate value for instructions which use immediate values.

In MARS the machine code which is generated from the assembly code is shown in the Code column, as in figure 4-3. A good way to check if you have translated your code correctly is to see if your translation matches the code in this column.



Bkpt	Address	Code	Basic		Source
☐	0x0040000	0x012a4020	dd $8,$9,$10	1: add $t0, $t1, $t2	
☐	0x0040000	0x22300025	ddi $16,$17,0x00000025	2: addi $s0, $s1, 37	

Figure 4-3: Machine Code example

The rest of the sections of this chapter are intended to help makes sense of this machine code.

Chapter 4. 2 Machine code for the add instruction

This section will translate the following `add` instruction to machine code.

```
add $t0, $t1, $t2
```

The MIPS Greensheet specifies the `add` instruction as an R-format instruction and the op-code/function for the `add` as 0/20. The op-code/function field is made up of two numbers, the first is the op-code, and the second is the function. Note that the function is used only for R format instructions. If the instruction has a function, the number to the left of the "/" is the op-code, and the number to the right of the "/" is the function. If there is only one number with no "/", it is the op-code, and there is no function.

Both the op-code and the function are 6 bits, divided into a 2 bit number and a 4 bit number. So the first number in the op-code and function is 0..3, and the second is 0..f. Both are generally called hex values, so this text will do so as well. So the 6 bits for the op-code translate to 00 0000, and the 6 bits for the function translate to 10 0000. These are placed into the op-code and function fields of the R format instruction shown in figure 4-5 below.

Register R_d is `$t0`. `$t0` is also register `$8`, or 01000, so 01000 is placed in the R_d field.

Register R_s is `$t1`. `$t1` is also register `$9`, or 01001, so 01001 is placed in the R_s field

Register R_t is `$t2`. `$t2` is also register `$10`, or 01010, so 01010 is placed in the R_t field.

The shamt is 00000 as there are no bits being shifted.

The result is the following R-format instruction.

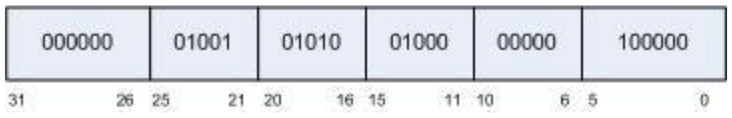


Figure 4-4: Machine code for add $t0, $t1, $t2

Thus the instruction "`add $t0, $t1, $t2`" translate into the bit string

00000001001010100100000000100000

This is as hard to type as it is to read. To make it readable, the bits are divided into groups of 4 bits, and these 4 bit values translated into hex. This results in the following instruction.

```
    0000 0001 0010 1010 0100 0000 0010 0000
0x    0    1    2    a    4    0    2    0    .
```

This can be checked by entering the instruction in MARS, and assembling it to see the resulting machine code. Be careful with the order of the registers when translating into machine code, as the R_d is the last register in machine code.

Chapter 4. 3 Machine code for the sub instruction

This section will translate the following `sub` instruction to machine code.

```
sub $s0, $s1, $s2
```

The MIPS Greensheet specifies the `sub` instruction as an R-format instruction and the op-code/function for the `sub` as 0/22. This means the 6 bits for the op code are 000000 and the 6 bits for the function are 100010.

Register R_d is `$s0` is also register `$16`, or `10000`.

Register R_s is `$s1` is also register `$17`, or `10001`.

Register R_t is `$s2` is also register `$18`, or `10010`.

The shamt is `00000` as there are no bits being shifted.

The result is the following R-format instruction.

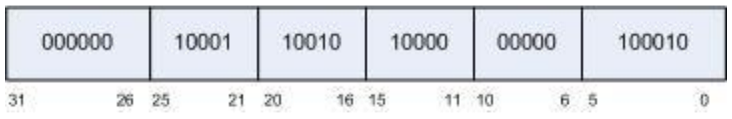


Figure 4-5: Machine code for sub $s0, $s1, $s2

To write this instruction's machine code, the bits are organized in groups of 4, and hex values given. This results in the number 0000 0010 0011 0010 1000 0000 0010 0010_2 or 0x02328022.

Chapter 4. 4 Machine code for the addi instruction

This section will translate the following `addi` instruction to machine code.

```
addi $s2, $t8, 37
```

The MIPS Greensheet specifies the `addi` instruction as an I-format instruction and the op-code/function for the `addi` as 8 (note that there is no function for an I-format instruction). This means the 6 bits for the op code are 001000.

Register R_t is `$s2` is also register `$18`, or `10010`.

Register R_s is `$t8` is also register `$24`, or `11000`.

The immediate value is 37, or `0x0025`.

The Result is the following I-format instruction.

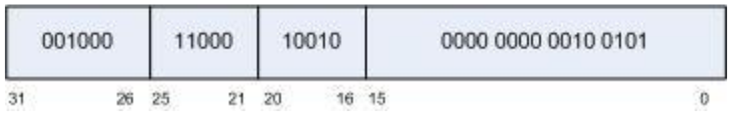


Figure 4-6: Machine code for addi $s2, $t8, 37

To write this instruction's machine code, the bits are organized in groups of 4, and hex values given. This results in the number 0010 0011 0001 0010 0000 0000 0010 0101_2 or 0x23120025.

When translating the I-format instruction, be careful to remember that R_t is destination register.

Chapter 4. 5 Machine code for the sll instruction

This section will translate the following SLL instruction to machine code.

```
sll $t0, $t1, 10
```

The MIPS Greensheet specifies the `sll` instruction as an R-format instruction and the op-code/function for the `sll` as 0/00. This means the 6 bits for the op code are 000000 and the 6 bits for the function are 000000.

Register rd is `$t0` is also register `$8`, or `01000`.

Register rs is not used.

Register rt is `$t1` is also register `$9`, or `01001`.

The shamt is 10, or `0x01010`.

The Result is the following R-format instruction.

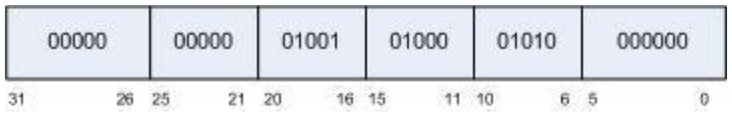


Figure 4-7: Machine code for sll $t0, $t1, 10

To write this instruction's machine code, the bits are are organized in groups of 4, and hex values given. This results in the number 0000 0000 0000 1001 0100 0010 1000 0000_2 or 0x00094280.

Chapter 4. 6 Exercises

1) Translate the following MIPS assembly language program into machine code.

```
.text
.globl main
main:
    ori $t0, $zero, 15
    ori $t1, $zero, 3
    add $t1, $zero $t1
    sub $t2, $t0, $t1
    sra $t2, $t2, 2
    mult $t0, $t1
    mflo $a0
    ori $v0, $zero, 1
    syscall
    addi $v0, $zero, 10
    syscall

.data
```

```
result: .asciiz "15 * 3 is "
```

2) Translate the following MIPS machine code into MIPS assembly language.

```
0x2010000a
0x34110005
0x012ac022
0x00184082
0x030f9024
```

What you will learn.

In this chapter you will learn:

1. how to create a subprogram.
2. the `jal` (call subprogram) and `jra $ra` (return from subprogram) operators.
3. the `.include` MARS assembly directive.
4. How to pass parameters to and retrieve return values from a subprogram.
5. how the Program Counter register (`$pc`) the control program execution sequence.
6. how to structure and comment a file of subprograms.
7. why the subprograms in this chapter cannot call themselves or other subprograms.

Chapter 5 Simple MIPS subprograms[13]

The programs that were implemented in Chapter 3 were very long and required a lot of code to implement simple operations. Many of these operations were common to more than one of the programs, and having to handle the details of these operations each time was distracting, and a possible source of errors. For example, every program needs to use syscall service 10 to exit a program. It would be nice to abstract the method of exiting the program once, and then use this abstraction in every program. One way to abstract data is to use subprograms. Something similar to subprograms exist in every language, and go by the names functions, procedures, or methods.[14]

The concept of a subprogram to create a group of MIPS assembly language statements which can be called (or dispatched) to perform a task. A good example is an abstraction we will call "Exit", which will automatically exit the program for the programmer.

These subprograms are called simple subprogram because they will be allowed to call any other subprograms, and they will not be allowed to modify any save registers. These limits on subprograms simplify the implementation of subprograms tremendously. These limits will be taken away Chapter 8, which will allow the creation of much more powerful, but also much more complex, subprograms.

Chapter 5. 1 Exit Subprogram

The first subprogram to be introduced will be called `Exit`, and will call syscall service 10 to exit the program. Because there will be no need to return to the calling program, this subprogram can be created by creating a label `Exit`, and putting code after that label which sets up the call to syscall and executes it. This is shown in the following program that prompts a user for an integer, prints it out, and exits the program.

[13] These subprograms are simple because they are *non-reentrant*. In simple terms, this means that these subprograms cannot call other subprograms and cannot call themselves. The reason they are non-reentrant will be explained in the chapter. How to make reentrant subprograms requires a concept called a program stack, and will be covered in chapter 8.

[14] All subprograms in this chapter could be implemented using Mars macros. However there is real value in introducing the concept of subprograms at this point in the text, and so macros will not be covered.

```
# File:     Program5-1.asm
# Author:   Charles Kann
# Purpose:  To illustrate implementing and calling a
#           subprogram named Exit.
.text
main:
    # read an input value from the user
    li $v0, 4
    la $a0, prompt
    syscall
    li $v0, 5
    syscall
    move $s0, $v0

    # print the value back to the user
    li $v0, 4
    la $a0, result
    syscall
    li $v0, 1
    move $a0, $s0
    syscall

    # call the Exit subprogram to exit
    jal Exit

.data
    prompt: .asciiz "Please enter an integer: "
    result: .asciiz "\nYou entered: "

# subprogram:     Exit
# author:         Charles Kann
# purpose:        to use syscall service 10 to exit a program
# input:          None
# output:         None
# side effects:   The program is exited
.text
Exit:
    li $v0, 10
    syscall
```

Program 5-1: Implementing and calling Exit subprogram

Chapter 5.1. 1 Commentary on Exit subprogram

1. This program has two `.text` sections. This is because the `Exit` subprogram is not a part of the main, and so exists in a different part of the file. Eventually the subprogram will be put into another file and included when needed. There can be as many `.text` and `.data` statements in assembly language as are needed, and they can be interspersed as they are here. They inform the assembler what type of information follows. When in a `.text` segment, there should be only program text, in a `.data` segment, only data. Later these will also be associated with segments of memory used.

2. The `jal` instruction transfers control of the computer to the address at the label `Exit`. For the present the reader can think of this as *calling* or *executing* a method or function in a HLL.

However the wording used here of transferring control is much more accurate, and this will become clear later in the chapter.

3. The `Exit` subprogram halts the program so there is no return statement when it is complete. This is a special case, and all other subprograms in this text will execute a return using the `jr $ra` instruction. This means that in this case a jump (`j`) operator could have been used to transfer control to the `Exit` subprogram. To be consistent with all other subprogram, this text will always use `jal`.

4. Programmers should always attempt to be consistent with naming conventions. This text will use the conventions of camel casing with the first word beginning with a lower case for variables and labels in a program (e.g. inputLabel), and camel casing with the first word beginning with a capital letter for subprogram (.e.g. InputInteger). There is no industry wide standard for naming conventions as with Java or C#, so the reader is free to choose their own standard. But once chosen, it should be followed.

5. All subprograms should at a minimum contain the information provided in this example. The programmer should state the program name, author, purpose, inputs, outputs, and side effects of running the program. The input and output variables are especially important as they will be registers and not be named as in a HLL. It is very hard after the program is written to remember what these values represent, and not documenting them makes the subprogram very difficult to use and maintain.

Chapter 5. 2 PrintNewLine subprogram

The next subprogram to be implemented will print a new line character. This subprogram will allow the program to stop inserting the control character "\n" into their programs. It will be used here to illustrate how to return from a subprogram.

```
# File:     Program5-2.asm
# Author:   Charles Kann
# Purpose:  To illustrate implementing and calling a
#           subprogram named PrintNewLine.
.text
main:
    # read an input value from the user
    li $v0, 4
    la $a0, prompt
    syscall
    li $v0, 5
    syscall
    move $s0, $v0

    # print the value back to the user
    jal PrintNewLine
    li $v0, 4
    la $a0, result
    syscall
    li $v0, 1
    move $a0, $s0
    syscall
    # call the Exit subprogram to exit
    jal Exit

.data
    prompt: .asciiz "Please enter an integer: "
    result: .asciiz "You entered: "

# subprogram:     PrintNewLine
# author:         Charles Kann
# purpose:        to output a new line to the user console
# input:          None
# output:         None
# side effects:   A new line character is printed to the
#           user's console
.text
PrintNewLine:
    li $v0, 4
    la $a0, __PNL_newline
    syscall
    jr $ra
.data
   __PNL_newline:   .asciiz "\n"

# subprogram:     Exit
# author:         Charles Kann
# purpose:        to use syscall service 10 to exit a program
# input:          None
# output:         None
# side effects:   The program is exited
.text
Exit:
    li $v0, 10
    syscall
```

Program 5-2: Implementing and calling the PrintNewLine subprogram

Chapter 5.2. 1 Commentary on Exit subprogram

1. This program now has three `.text` segments, but the reason for this is becomes obvious when the code is examined. The `PrintNewLine` subprogram requires a value to be stored in the `.data` segment, the newline character. The `.text` segments are needed to inform the assembler that program instructions are again contained in the code. This is emphasized here as it points out a good rule to follow when writing subprograms. Always begin the subprogram with a `.text` statement. If the assembler already thinks it is in a `.text` segment, there is no effect, but the subprogram is protected from the case where the assembler thinks it is assembling a `.data` segment when reaching the subprogram. In real life files can change often, and omitting a simple `.text` of `.data` segment when it should be present can lead to myriads of unnecessary problems.

2. The PrintNewLine subprogram shows how to return from a subprogram using the instruction "`jr $ra`". The next section of this chapter will explain how this actually works. For now, this can be thought of as a *return* statement.

3. A label was needed in the PrintNewLine subprogram to contain the address of the newline variable. In MIPS assembly, you cannot use labels with the same name[15]. This subprogram will eventually become part of a utility package which will be used by a number of programs. Care must be taken to make sure that any label used will conflict with a label in a program. So the convention of putting a double underscore (__) before the variable, a string representing the subprogram it is in (PNL), and another underscore before the variable name (newline) is used. It is hoped that this will solve nearly all name conflicts.

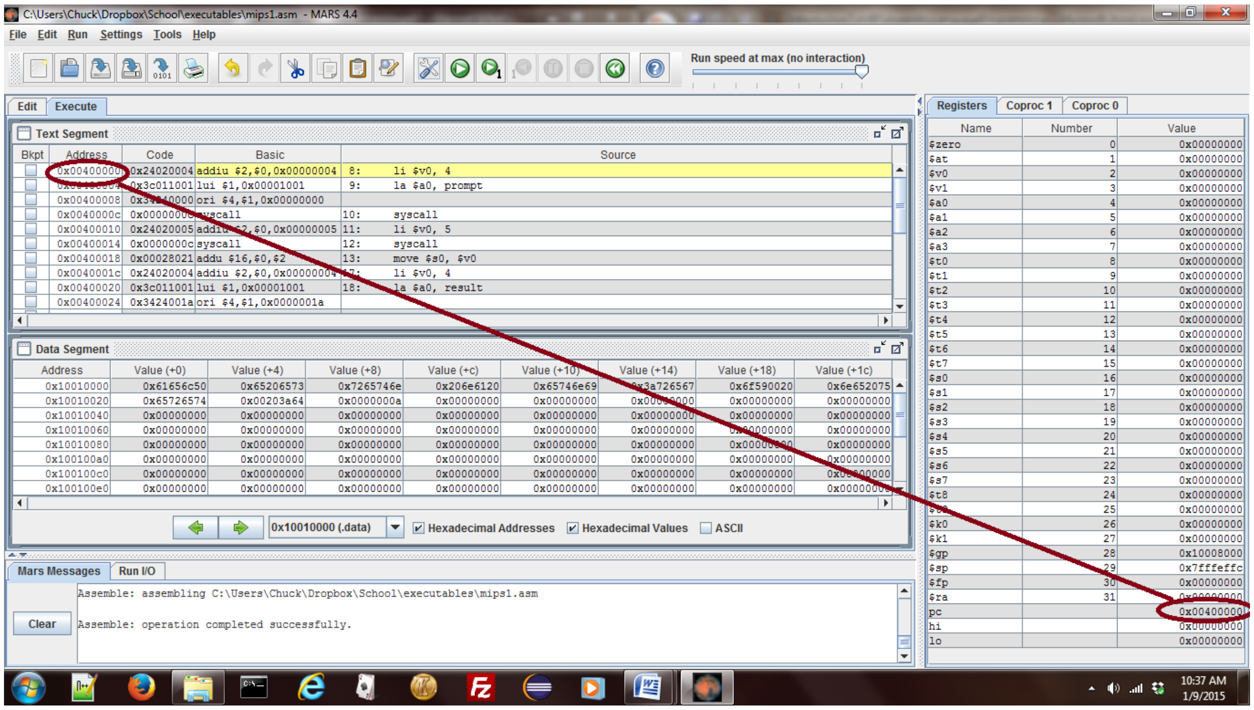
4. The `$a0` and `$v0` registers have been changed, but this is not listed as a side effect of the program. Part of the definition of the registers says that the save registers (`$s0..$s9`) must contain the same value when the subprogram returns as they did when the subprogram was called. For these simple subprogram, that means that these registers cannot be used. All other registers have no guarantee of the value after the subprogram is called. A programmer cannot rely on the values of `$a0` or `$v0` once a program is called. So while changing them is a side effect of calling the function, it does not have to be listed as it is implied in the execution of the method.

Chapter 5. 3 The Program Counter ($pc) register and calling a subprogram

One of the most important registers in the CPU is the `$pc` register. Note that it is not one of the 32 general purpose registers that the programmer can access directly. It is a special register that keeps track of the address in memory of the next instruction to be executed. For example, consider Program5-2. Compile this program and bring up the MARS execution screen, as shown in the Figure5.1. The address column shows the memory address where each instruction is stored. Note that the first line of the program, which is highlighted in yellow because it is the next instruction to execute, is at 0x0040000, and the `$p`c register contains the value 0x0040000. Step to the next instruction, as in Figure 5.2, and you will see that the `$pc` contains the value 00x00400004, which is the address of the next instruction to execute. Thus the `$pc` specifies the next instruction to be executed.

The screen shot in Figure5.3 is at the `jal PrintNewLine` instruction. Now the PC points to 0x0040001c, and the next statement would be 0x00400020. However note that tranlsation of the `jal PrintNewLine` instruction in the basic column has the address 0x00400040 in it. This address is the first line in the `PrintNewLine` subprogram. When the `jal` instruction is executed, the `$pc` register is set to 0x00400040, and the program continues by executing in the `PrintNewLine` subprogram, as shown in Figure5.4. This is how the `jal` operator transfer the control of the computer to run the subprogram.

[15] More accurately, in MIPS you cannot use labels with the same name unless the labels are resolved to addresses so that they do not conflict. If separate assembly and linking are performed, any none global symbols can be used in any file, so long as it is in that file once. This text does not handle separate compilation and linking, so the rule that label names cannot be repeated is correct for all programs in this text.



C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.4

Run speed at max (no interaction)

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24020004	addiu $2,$0,0x00000004	8: li $v0, 4
☐	0x00400004	0x3c011001	lui $1,0x00001001	9: la $a0, prompt
☐	0x00400008	0x34240000	ori $4,$1,0x00000000	
☐	0x0040000c	0x0000000c	syscall	10: syscall
☐	0x00400010	0x24020005	addiu $2,$0,0x00000005	11: li $v0, 5
☐	0x00400014	0x0000000c	syscall	12: syscall
☐	0x00400018	0x00028021	addu $16,$0,$2	13: move $s0, $v0
☐	0x0040001c	0x24020004	addiu $2,$0,0x00000004	17: li $v0, 4
☐	0x00400020	0x3c011001	lui $1,0x00001001	18: la $a0, result
☐	0x00400024	0x3424001a	ori $4,$1,0x0000001a	

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x61656c50	0x65206573	0x7265746e	0x206e6120	0x65746e69	0x3a726567	0x6f590020	0x6e652075
0x10010020	0x65726574	0x00203a64	0x0000000a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Mars Messages

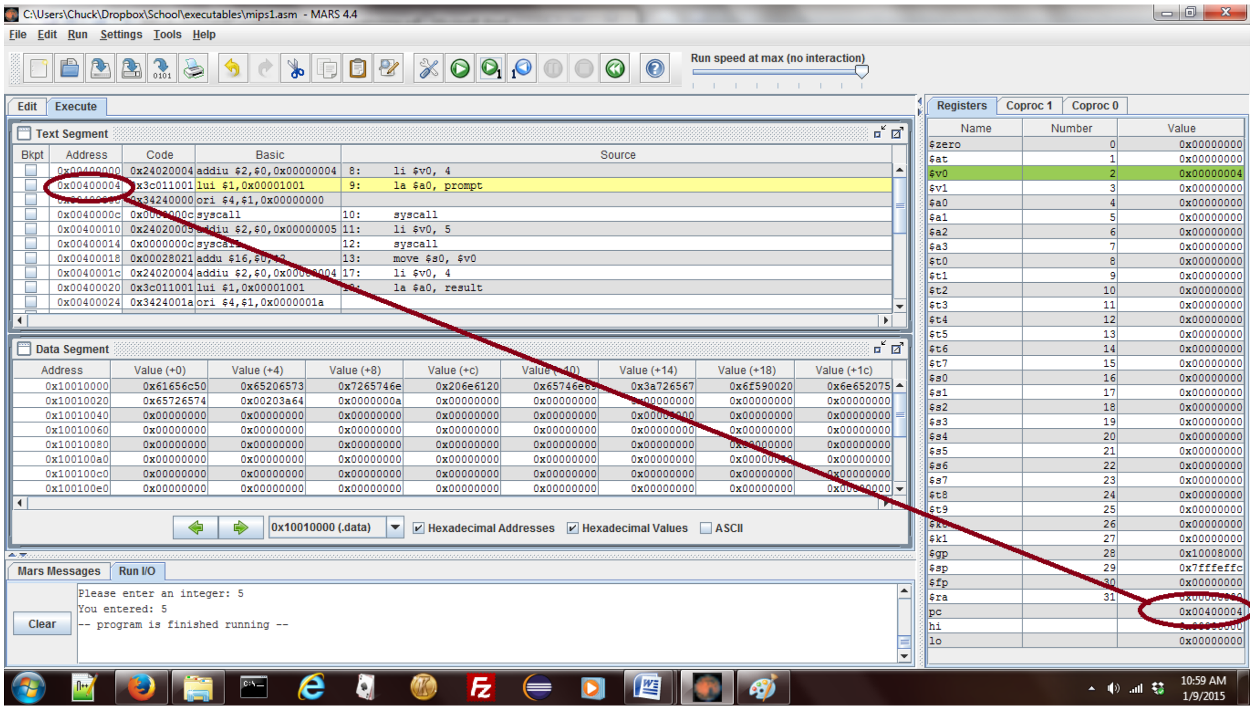
```
Assemble: assembling C:\Users\Chuck\Dropbox\School\executables\mips1.asm

Assemble: operation completed successfully.
```

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

10:37 AM 1/9/2015

Figure 5-1: `$pc` when program execution starts



C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.4

Run speed at max (no interaction)

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24020004	addiu $2,$0,0x00000004	8: li $v0, 4
☐	0x00400004	0x3c011001	lui $1,0x00001001	9: la $a0, prompt
☐	0x00400008	0x34240000	ori $4,$1,0x00000000	
☐	0x0040000c	0x0000000c	syscall	10: syscall
☐	0x00400010	0x24020005	addiu $2,$0,0x00000005	11: li $v0, 5
☐	0x00400014	0x0000000c	syscall	12: syscall
☐	0x00400018	0x00028021	addu $16,$0,$2	13: move $s0, $v0
☐	0x0040001c	0x24020004	addiu $2,$0,0x00000004	17: li $v0, 4
☐	0x00400020	0x3c011001	lui $1,0x00001001	18: la $a0, result
☐	0x00400024	0x3424001a	ori $4,$1,0x0000001a	

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x61656c50	0x65206573	0x7265746e	0x206e6120	0x65746e69	0x3a726567	0x6f590020	0x6e652075
0x10010020	0x65726574	0x00203a64	0x0000000a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Run I/O

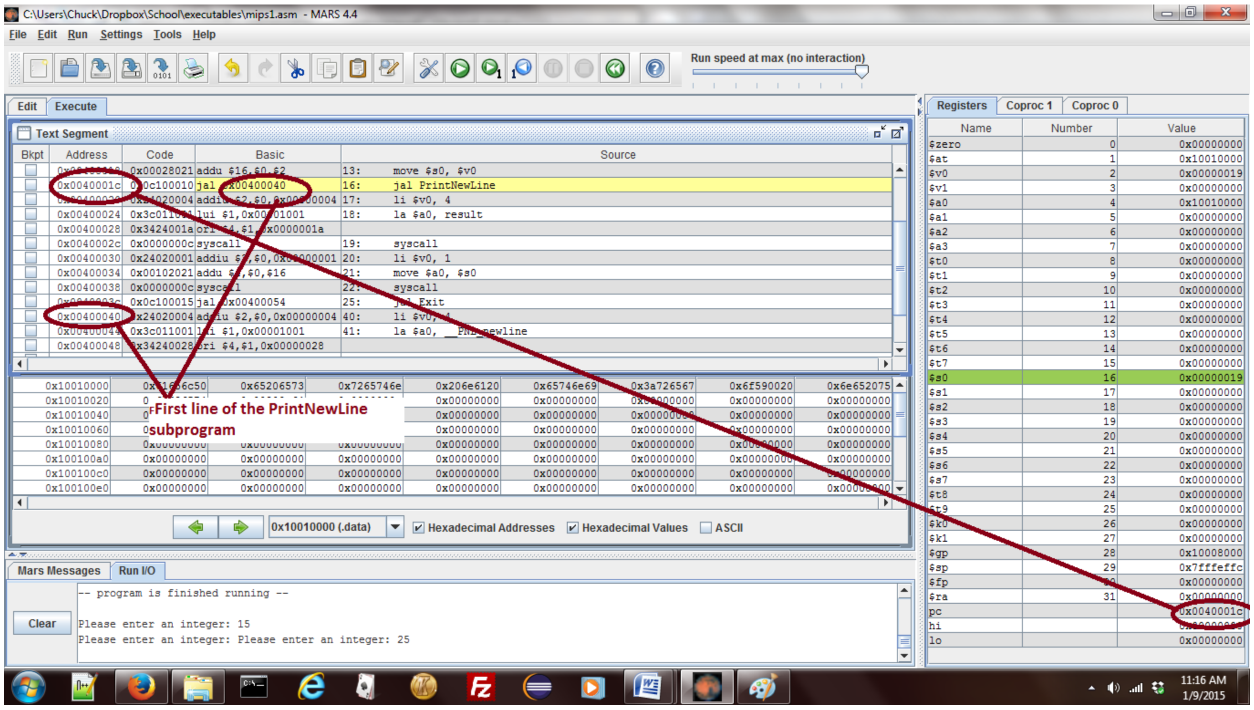
```
Please enter an integer: 5
You entered: 5
-- program is finished running --
```

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000004
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400004
hi		0x00000000
lo		0x00000000

10:59 AM 1/9/2015

Figure 5-2: $pc after the execution of the first instruction


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.4
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Text Segment
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
Clear
First line of the PrintNewLine subprogram
11:16 AM
1/9/2015

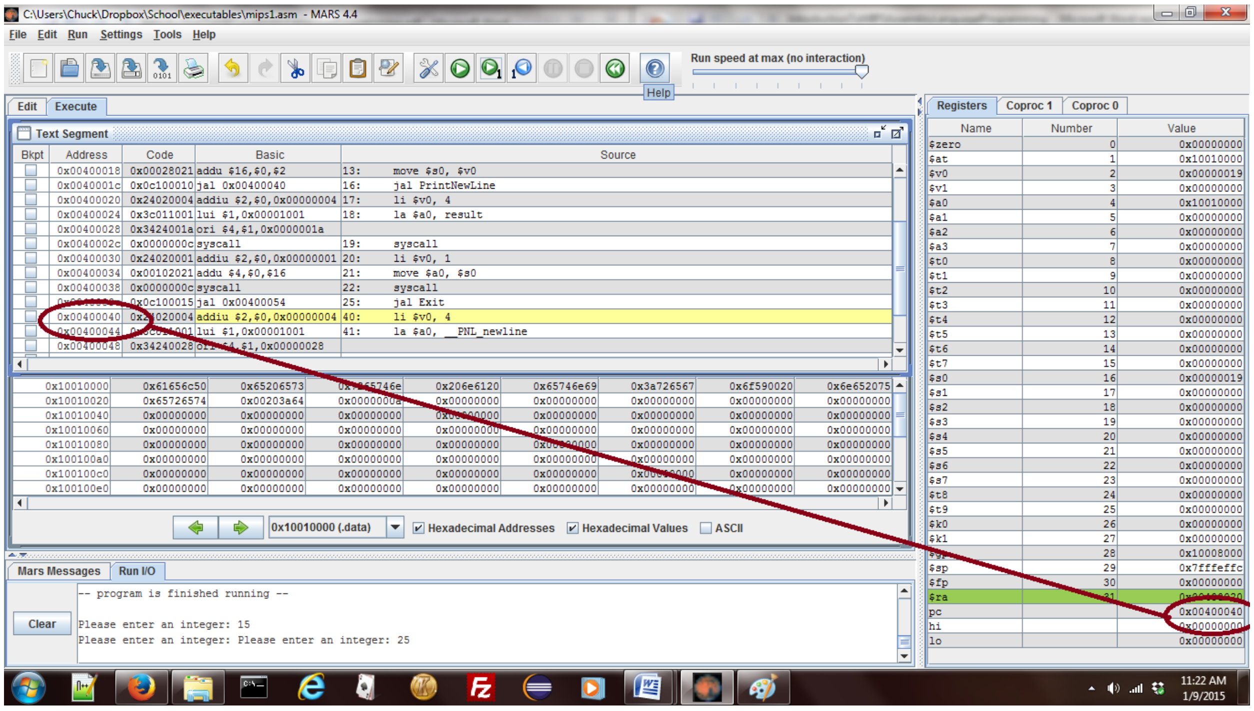
```
Bkpt  Address     Code        Basic                          Source
      0x00400018  0x00028021  addu $16,$0,$2              13:     move $s0, $v0
      0x0040001c  0x0c100010  jal 0x00400040              16:     jal PrintNewLine
      0x00400020  0x24020004  addiu $2,$0,0x00000004      17:     li $v0, 4
      0x00400024  0x3c011001  lui $1,0x00001001           18:     la $a0, result
      0x00400028  0x3424001a  ori $4,$1,0x0000001a
      0x0040002c  0x0000000c  syscall                     19:     syscall
      0x00400030  0x24020001  addiu $2,$0,0x00000001      20:     li $v0, 1
      0x00400034  0x00102021  addu $4,$0,$16              21:     move $a0, $s0
      0x00400038  0x0000000c  syscall                     22:     syscall
      0x0040003c  0x0c100015  jal 0x00400054              25:     jal Exit
      0x00400040  0x24020004  addiu $2,$0,0x00000004      40:     li $v0, 4
      0x00400044  0x3c011001  lui $1,0x00001001           41:     la $a0, __PNL_newline
      0x00400048  0x34240028  ori $4,$1,0x00000028
```

```
0x10010000  0x61656c50  0x65206573  0x7265746e  0x206e6120  0x65746e69  0x3a726567  0x6f590020  0x6e652075
0x10010020  [illegible] [illegible] [illegible] 0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010040  [illegible] [illegible] [illegible] 0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010060  [illegible] [illegible] [illegible] 0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010080  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100a0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100c0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100e0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000

0x10010000 (.data)   [✓] Hexadecimal Addresses   [✓] Hexadecimal Values   [ ] ASCII
```

```
-- program is finished running --

Please enter an integer: 15
Please enter an integer: Please enter an integer: 25
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000019
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000019
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040001c
hi		0x00000000
lo		0x00000000

Figure 5-3: Just before calling PrintNewLine subprogram


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.4
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Help
Edit Execute
Text Segment
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
Clear
11:22 AM
1/9/2015

```
Bkpt  Address     Code        Basic                          Source
      0x00400018  0x00028021  addu $16,$0,$2              13:     move $s0, $v0
      0x0040001c  0x0c100010  jal 0x00400040              16:     jal PrintNewLine
      0x00400020  0x24020004  addiu $2,$0,0x00000004      17:     li $v0, 4
      0x00400024  0x3c011001  lui $1,0x00001001           18:     la $a0, result
      0x00400028  0x3424001a  ori $4,$1,0x0000001a
      0x0040002c  0x0000000c  syscall                     19:     syscall
      0x00400030  0x24020001  addiu $2,$0,0x00000001      20:     li $v0, 1
      0x00400034  0x00102021  addu $4,$0,$16              21:     move $a0, $s0
      0x00400038  0x0000000c  syscall                     22:     syscall
      0x0040003c  0x0c100015  jal 0x00400054              25:     jal Exit
      0x00400040  0x24020004  addiu $2,$0,0x00000004      40:     li $v0, 4
      0x00400044  0x3c011001  lui $1,0x00001001           41:     la $a0, __PNL_newline
      0x00400048  0x34240028  ori $4,$1,0x00000028
```

```
0x10010000  0x61656c50  0x65206573  0x7265746e  0x206e6120  0x65746e69  0x3a726567  0x6f590020  0x6e652075
0x10010020  0x65726574  0x00203a64  0x0000000a  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010040  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010060  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010080  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100a0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100c0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100e0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000

0x10010000 (.data)   [✓] Hexadecimal Addresses   [✓] Hexadecimal Values   [ ] ASCII
```

```
-- program is finished running --

Please enter an integer: 15
Please enter an integer: Please enter an integer: 25
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000019
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000019
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400020
pc		0x00400040
hi		0x00000000
lo		0x00000000

Figure 5-4: Program has transferred control into the PrintNewLine subprogram

Chapter 5. 4 Returning from a subprogram and the `$ra` register

As the previous section has shown, the `$pc` register controls the next statement to be executed in a program. When calling a subprogram the `$pc` is set to the first line in the subprogram, so to return from the subprogram it is simply a matter of putting into the `$pc` the address of the instruction to execute when returning from the subprogram. To find this return address, the mechanics of a subprogram call must be defined.

No matter what HLL the reader has used, the semantics of a subprogram call is always the same. When the call is executed the program transfers control to the subprogram. When the subprogram is finished, control is transferred back to the next statement immediately following the subprogram call in the calling subprogram. This is illustrated in Figure5.5 below.

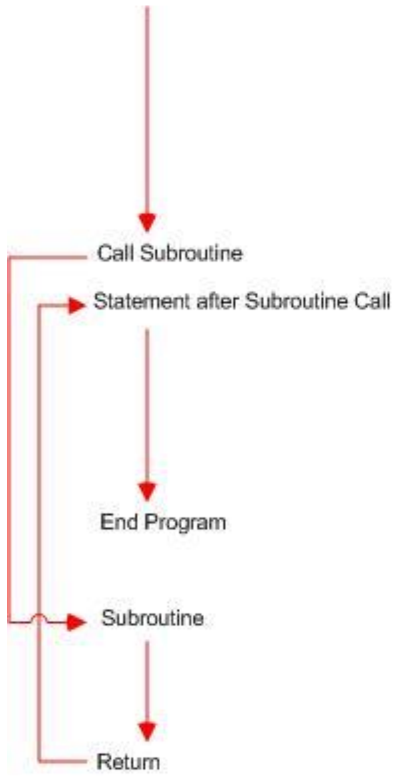
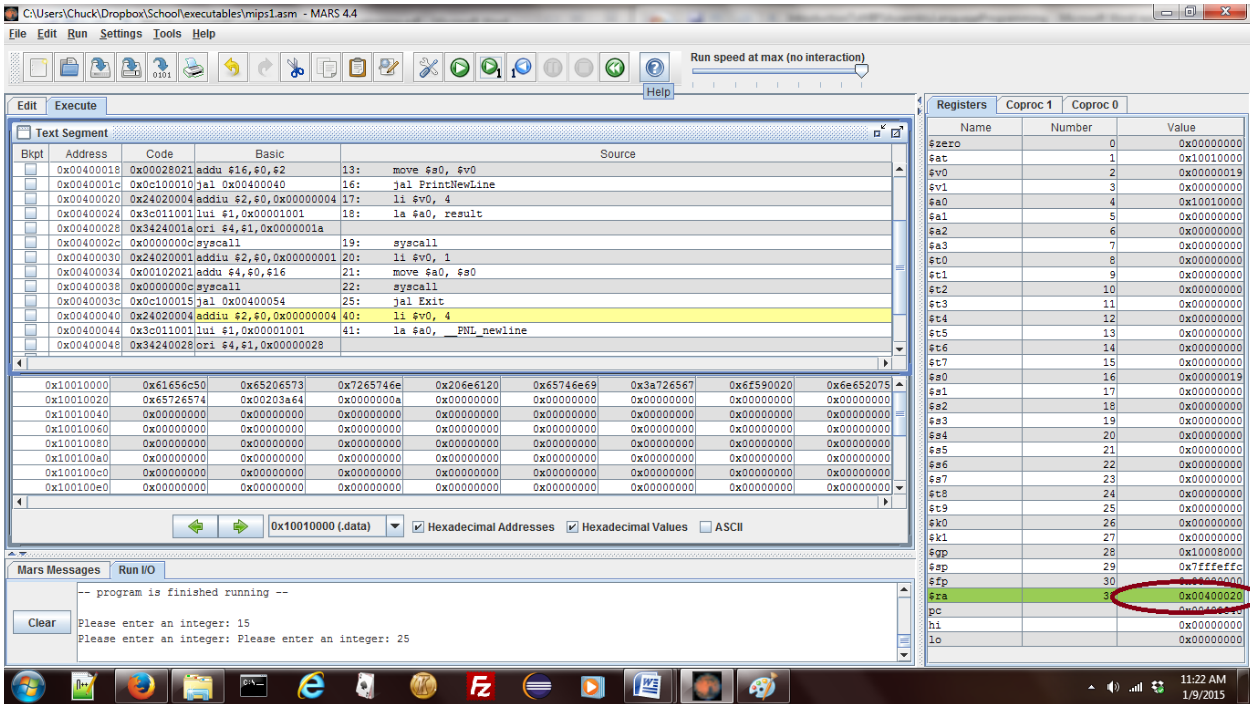


Figure 5-5: Subprogram calling semantics

What this says it that when the subprogram is called the next sequential address of an instruction must be stored, and when the subprogram is complete the control must branch back to that instruction. In the example in the last section, Figure 5.3 showed the subprogram call was at instruction 0x0040001c, and the next sequential instruction would have been 0x00400020. So the value 0x00400020 must be stored somewhere and when the end of the subroutine is reached the program must transfer control back to that statement.

Figure 5.6 is a screen shot when the program is executing at the first line in the PrintNewLine subroutine. This is the same as Figure 5.4, but now the register circled is the `$ra` register. Note that in the `$ra` register is stored the address 0x00400020. This tells us that the jal operator not only sets the `$pc` register to the value of the label representing the first line of the subprogram, it also sets the `$ra` register to point back to the next sequential line, which is the return address used after the subprogram completes.


C:\Users\Chuck\Dropbox\School\executables\mips1.asm - MARS 4.4
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Help
Edit Execute
Text Segment
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
Clear
11:22 AM 1/9/2015

```
Bkpt Address     Code       Basic                      Source
     0x00400018 0x00028021 addu $16,$0,$2             13:     move $s0, $v0
     0x0040001c 0x0c100010 jal 0x00400040             16:     jal PrintNewLine
     0x00400020 0x24020004 addiu $2,$0,0x00000004     17:     li $v0, 4
     0x00400024 0x3c011001 lui $1,0x00001001          18:     la $a0, result
     0x00400028 0x3424001a ori $4,$1,0x0000001a
     0x0040002c 0x0000000c syscall                    19:     syscall
     0x00400030 0x24020001 addiu $2,$0,0x00000001     20:     li $v0, 1
     0x00400034 0x00102021 addu $4,$0,$16             21:     move $a0, $s0
     0x00400038 0x0000000c syscall                    22:     syscall
     0x0040003c 0x0c100015 jal 0x00400054             25:     jal Exit
     0x00400040 0x24020004 addiu $2,$0,0x00000004     40:     li $v0, 4
     0x00400044 0x3c011001 lui $1,0x00001001          41:     la $a0, __PNL_newline
     0x00400048 0x34240028 ori $4,$1,0x00000028
```

```
0x10010000  0x61656c50  0x65206573  0x7265746e  0x206e6120  0x65746e69  0x3a726567  0x6f590020  0x6e652075
0x10010020  0x65726574  0x00203a64  0x0000000a  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010040  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010060  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x10010080  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100a0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100c0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x100100e0  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
```

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

```
-- program is finished running --

Please enter an integer: 15
Please enter an integer: Please enter an integer: 25
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000019
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000019
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400020
pc		0x00400040
hi		0x00000000
lo		0x00000000

Figure 5-6: Saving the `$ra` register

Now the entire life-cycle of calling and returning from a subprogram can be explained. When a subprogram is called, the `jal` operator updates the `$pc` register to point to the address of the label for the subprogram. The `jal` operator also sets the `$ra` register to the next sequential address after the subprogram call. When the "`jr $ra`" instruction is called, the subprogram *returns* by transferring control to the address in the `$ra` register.

This also explains why a subprogram in this chapter cannot call another subprogram. When the first subprogram is called, the value for the return is saved in the `$ra`. When the second subprogram is subsequently called, the value in the `$ra` is changed, and the original value is lost. This problem will be explored more in the exercises at the end of the chapter.

Chapter 5. 5 Input parameter with PrintString subprogram

The next subprogram to be implemented, the `PrintString` subprogram abstracts the ability to print a string. This subprogram will require an input parameter to be passed into the program, the address of the string to print. Remembering the register conventions from chapter 2, the registers `$a0..$`a4 are used to pass input parameters into a program, so the register `$a0` will be used for the input parameter to this subprogram. The subprogram will then load a 4 into `$v0`, invoke `syscall`, and return.

The program from Chapter 5.4 is modified to use the `PrintString` program. The changes to the program are highlighted in yellow. The only two new concepts in this program are the ability to pass parameters into the program, and that the documentation at the beginning of the program should include the input parameter. So no commentary will be provided after this program.

```
# File:         Program5-2.asm
# Author:   Charles Kann
# Purpose:  To illustrate implementing and calling a
#           subprogram named PrintNewLine.
.text
main:
    # read an input value from the user
    la $a0, prompt
    jal PrintString
    li $v0, 5
    syscall
    move $s0, $v0

    # print the value back to the user
    jal PrintNewLine
    la $a0, result
    jal PrintString
    li $v0, 1
    move $a0, $s0
    syscall

    # call the Exit subprogram to exit
    jal Exit

.data
    prompt: .asciiz "Please enter an integer: "
    result: .asciiz "You entered: "

# subprogram:     PrintNewLine
# author:   Charles Kann
# purpose:  to output a new line to the user console
# input:    None
# output:   None
# side effects:   A new line character is printed to the
#           user's console
.text
PrintNewLine:
    li $v0, 4
    la $a0, __PNL_newline
    syscall
    jr $ra
.data
   __PNL_newline:   .asciiz "\n"

# subprogram:     PrintString
# author:   Charles W. Kann
# purpose:  To print a string to the console
# input:    $a0 - The address of the string to print.
# returns:  None
# side effects:   The String is printed to the console.
.text
PrintString:
    addi $v0, $zero, 4
    syscall
    jr $ra

# subprogram:     Exit
# author:   Charles Kann
# purpose:  to use syscall service 10 to exit a program
# input:    None
# output:   None
# side effects:   The program is exited
.text
Exit:
    li $v0, 10
    syscall
```

Program 5-3: Input parameter with the PrintString subprogram

Chapter 5. 6 Multiple input parameters with PrintInt subprogram

When printing an integer value, the program always printed a string first to tell the user what was being printed, and then the integer was printed. This behavior will be abstracted in the following subprogram, `PrintInt`. In this subprogram there are two input parameters: the address of the string to print, which is stored in `$a0`; and the integer value to print, which is stored in `$a1`. Other than passing in two parameters and creating a slightly larger subprogram, this program does not add any new material that needs a commentary section. The changes from the program in Chapter 5.5 are highlighted in yellow.

```
# File:     Program5-2.asm
# Author:   Charles Kann
# Purpose:  To illustrate implementing and calling a
#           subprogram named PrintNewLine.
.text
main:
    # read an input value from the user
    la $a0, prompt
    jal PrintString
    li $v0, 5
    syscall
    move $s0, $v0

    # print the value back to the user
    jal PrintNewLine
    la $a0, result
    move $a1, $s0
    jal PrintInt

    # call the Exit subprogram to exit
    jal Exit
.data
    prompt: .asciiz "Please enter an integer: "
    result: .asciiz "You entered: "
# subprogram:     PrintNewLine
# author:   Charles Kann
# purpose:  to output a new line to the user console
# input:    None
# output:   None
# side effects:   A new line character is printed to the
#            user's console
.text
PrintNewLine:
    li $v0, 4
    la $a0, __PNL_newline
    syscall
    jr $ra
.data
   __PNL_newline:   .asciiz "\n"

# subprogram:     PrintInt
# author:    Charles W. Kann
# purpose:   To print a string to the console
# input:     $a0 - The address of the string to print.
#            $a1 - The value of the int to print
# returns:   None
# side effects:   The String is printed followed by the integer value.
.text
PrintInt:
    # Print string.  The string address is already in $a0
    li $v0, 4
    syscall

    # Print integer.   The integer value is in $a1, and must
    # be first moved to $a0.
    move $a0, $a1
    li $v0, 1
    syscall

    #return
    jr $ra

# subprogram:     PrintString
# author:    Charles W. Kann
# purpose:   To print a string to the console
# input:     $a0 - The address of the string to print.
# returns:   None
# side effects:   The String is printed to the console.
.text
PrintString:
    addi $v0, $zero, 4
    syscall
    jr $ra

# subprogram:     Exit
# author:    Charles Kann
# purpose:   to use syscall service 10 to exit a program
# input:     None
# output:    None
# side effects:   The program is exited
.text
Exit:
    li $v0, 10
    syscall
```

Program 5-4: Multiple input parameters with PrintInt subprogram

Chapter 5. 7 Return values with PromptInt subprogram

The next subprogram in this chapter, PromptInt, shows how to return a value from a subprogram. The registers $a0..$a3 are used to pass values into a subprogram, and the registers $v0..$v1 are used to return values. The program from Chapter 5.6 has been changed to include the PromptInt subprogram, and the changes are highlighted in yellow.

```
# File:     Program5-2.asm
# Author:   Charles Kann
# Purpose:  To illustrate implementing and calling a
#           subprogram named PrintNewLine.
.text
main:
    # read an input value from the user
    la $a0, prompt
    jal PromptInt
    move $s0, $v0

    # print the value back to the user
    jal PrintNewLine
    la $a0, result
    move $a1, $s0
    jal PrintInt

    # call the Exit subprogram to exit
    jal Exit

.data
    prompt: .asciiz "Please enter an integer: "
    result: .asciiz "You entered: "

# subprogram:     PrintNewLine
# author:   Charles Kann
# purpose:  to output a new line to the user console
# input:    None
# output:   None
# side effects:   A new line character is printed to the
#           user's console
.text
PrintNewLine:
    li $v0, 4
    la $a0, __PNL_newline
    syscall
    jr $ra
.data
   __PNL_newline:   .asciiz "\n"

# subprogram:     PrintInt
# author:   Charles W. Kann
# purpose:  To print a string to the console
# input:    $a0 - The address of the string to print.
#           $a1 - The value of the int to print
# returns:  None
# side effects:   The String is printed followed by the integer value.
.text
```

```
PrintInt:
    # Print string.  The string address is already in $a0
    li $v0, 4
    syscall

    # Print integer.   The integer value is in $a1, and must
    # be first moved to $a0.
    move $a0, $a1
    li $v0, 1
    syscall

    #return
    jr $ra

# subprogram:     PromptInt
# author:   Charles W. Kann
# purpose:  To print the user for an integer input, and
#               to return that input value to the caller.
# input:    $a0 - The address of the string to print.
# returns:  $v0 - The value the user entered
# side effects:   The String is printed followed by the integer value.
.text
PromptInt:
    # Print the prompt, which is already in $a0
    li $v0, 4
    syscall

    # Read the integer value.  Note that at the end of the
    # syscall the value is already in $v0, so there is no
    # need to move it anywhere.
    li $v0, 5
    syscall

    #return
    jr $ra

# subprogram:     PrintString
# author:   Charles W. Kann
# purpose:  To print a string to the console
# input:    $a0 - The address of the string to print.
# returns:  None
# side effects:   The String is printed to the console.
.text
PrintString:
    addi $v0, $zero, 4
    syscall
    jr $ra

# subprogram:     Exit
# author:   Charles Kann
# purpose:  to use syscall service 10 to exit a program
# input:    None
# output:   None
# side effects:   The program is exited
.text
Exit:
    li $v0, 10
```

```
          syscall
```

Chapter 5. 8 Create a utils.asm file

The final step in adding the subprograms developed in this chapter is to put them in a separate file so that any program you write can use them. To do this, create a file called *utils.asm*, and copy all of the subprograms (including comments) into this file. Once this is done, this utilities file needs to be correctly documented. The preamble comment (the comment at the start of the file) in the utility file should, at a minimum, contain the following information:

- The name of the file, so that it can be found.
- The purpose of the file, in this case what type of subprograms are defined in it.
- The initial author of the file.
- A copyright statement
- A subprogram index so that a user can quickly find the subprograms in the file, and determine if they are of use to the programmer. Ideally this index should be in alphabetic order.
- A modification history. Every change to this file should be documented here. Even innocuous changes like new comments should be recorded. If currently working programs start to fail for some reason, it makes it much easier to find if this modification history is up to date. Heaven save you from the wrath of a programmer who has spent days trying to track down a bug, only to find that you have made an undocumented change to this file!

The entire file utils.asm is included here to show what it should look like when completed.

```
# File:     utils.asm
# Purpose:  To define utilities which will be used in MIPS programs.
# Author:   Charles Kann
#
# Instructors are granted permission to make copies of this file
# for use by # students in their courses. Title to and ownership
# of all intellectual property rights
# in this file are the exclusive property of
# Charles W. Kann, Gettysburg, Pa.
#
# Subprograms Index:
#     Exit -       Call syscall with a server 10 to exit the program
#     NewLine -    Print a new line character (\n) to the console
#     PrintInt -   Print a string with an integer to the console
#     PrintString -      Print a string to the console
#     PromptInt - Prompt the user to enter an integer, and return
#                 it to the calling program.
#
# Modification History
#     12/27/2014 - Initial release

# subprogram:     PrintNewLine
# author:         Charles Kann
# purpose:        to output a new line to the user console
# input:          None
# output:         None
# side effects:   A new line character is printed to the
#                 user's console
.text
PrintNewLine:
    li $v0, 4
    la $a0, __PNL_newline
    syscall
    jr $ra
.data
   __PNL_newline:   .asciiz "\n"

# subprogram:     PrintInt
# author:         Charles W. Kann
# purpose:        To print a string to the console
# input:          $a0 - The address of the string to print.
#                 $a1 - The value of the int to print
# returns:        None
# side effects:   The String is printed followed by the integer value.
.text
PrintInt:
    # Print string.  The string address is already in $a0
    li $v0, 4
    syscall

    # Print integer.   The integer value is in $a1, and must
    # be first moved to $a0.
    move $a0, $a1
    li $v0, 1
    syscall

    #return
    jr $ra

# subprogram:     PromptInt
# author:         Charles W. Kann
# purpose:        To print the user for an integer input, and
#                 to return that input value to the caller.
# input:          $a0 - The address of the string to print.
# returns:        $v0 - The value the user entered
# side effects:   The String is printed followed by the integer value.
.text
PromptInt:
    # Print the prompt, which is already in $a0
    li $v0, 4
    syscall

    # Read the integer value.  Note that at the end of the
    # syscall the value is already in $v0, so there is no
    # need to move it anywhere.
    move $a0, $a1
    li $v0, 5
    syscall

    #return
    jr $ra

# subprogram:     PrintString
# author:         Charles W. Kann
# purpose:        To print a string to the console
# input:          $a0 - The address of the string to print.
# returns:        None
# side effects:   The String is printed to the console.
.text
PrintString:
    addi $v0, $zero, 4
    syscall
    jr $ra

# subprogram:     Exit
# author:         Charles Kann
# purpose:        to use syscall service 10 to exit a program
# input:          None
# output:         None
# side effects:   The program is exited
.text
Exit:
    li $v0, 10
    syscall
```

Program 5-5: File utils.asm

Chapter 5. 9 Final program to prompt, read, and print an integer

The following program is the final result of this chapter. Make sure that the file utils.asm is in the same directory as the program Program5-9.asm. Bring Program5-9.asm into the edit window in MARS, assemble it, and run it.

```
# File:     Program5-9.asm
# Author:   Charles Kann
# Purpose:  To illustrate implementing and calling a
#           subprogram named PrintNewLine.
.text
main:
    # read an input value from the user
    la $a0, prompt
    jal PromptInt
    move $s0, $v0

    # print the value back to the user
    jal PrintNewLine
    la $a0, result
    move $a1, $s0
    jal PrintInt

    # call the Exit subprogram to exit
    jal Exit
.data
    prompt: .asciiz "Please enter an integer: "
    result: .asciiz "You entered: "
.include "utils.asm"
```

Program 5-6: Final program to prompt for, read, and print an integer

Note that by using subprogram abstraction how much easier this program is to understand than the one at the start of the chapter. It is beginning to look like the HLL code most readers are already familiar with.

Chapter 5. 10 Summary

This chapter covered the basics of subprogram calls, how to implement and call subprograms, and the basics of how to pass basic parameters to the subprogram. It also explained the `$pc` and `$ra` registers, and their role in controlling the execution sequence of a program. How to properly comment subprograms and files was also covered.

Chapter 5. 11 Exercises

1. There are many algorithms presented in this text that would lend themselves to be included as subprograms in the utils.asm file. Implement some or all of the following into the `utils.asm` file, properly documenting them, and include programs to test them.
    a. `NOR` subprogram - take two input parameters, and return the NOR operation on those two parameter.
    b. `NAND`- take two input parameters, and return the NAND operation on those two parameter.
    c. `NOT`- take one input parameters, and return the NOT operation on that parameter.
    d. `Mult4 –` take an input parameter, and return that parameter multiplied by 4 using only shift and add operations.
    e. `Mult10` - take an input parameter, and return that parameter multiplied by 10 using only shift and add operations.
    f. `Swap`- take two input parameters, swap them using only the XOR operation.
    g. `RightCircularShift` - take an input parameter, and return two values. The first is the value that has been shifted 1 bit using a right circular shift, and the second is the value of the bit which has been shifted.
    h. `LeftCircularShift` - take an input parameter, and return two values. The first is the value that has been shifted 1 bit using a left circular shift, and the second is the value of the bit which has been shifted.
    i. `ToUpper` - take a 32 bit input which is 3 characters and a null, or a 3 character string. Convert the 3 characters to upper case if they are lower case, or do nothing if they are already upper case.
    j. `ToLower` - take a 32 bit input which is 3 characters and a null, or a 3 character string. Convert the 3 characters to lower case if they are upper case, or do nothing if they are already lower case.

2. A colleague decided that the PrintInt subprogram should print a new line after the integer has been printed, and has modified the PrintInt to print a new line character as follows.

```
# subprogram:     PrintInt
# author:   Charles W. Kann
# purpose:  To print a string to the console
# input:    $a0 - The address of the string to print.
#           $a1 - The value of the int to print
# returns:  None
# side effects:   The String is printed followed by the integer
value.
.text
PrintInt:
    # Print string.  The string address is already in $a0
    li $v0, 4
    syscall

    # Print integer.   The integer value is in $a1, and must
    # be first moved to $a0.
    move $a0, $a1
    li $v0, 1
    syscall

    # Print a new line character
    jal PrintNewLine

    #return
    jr $ra
```

When the program is run, it never ends and acts like it is stuck in an infinite loop. Help this colleague figure out what is wrong with the program.

a. Explain what is happening in the program
b. Come up with a mechanism which shows that this program is indeed in an infinite loop.
c. Come up with a solution which fixes this problem.

3. Someone has modified the utils.asm file to insert a `PrintTab` subprogram immediately after the `PrintNewLine` subprogram as shown below (changes are highlighted in yellow). The programmer complains that the PrintTab command cannot be called using the "`jal PrintTab`" instruction. What is wrong, and how can this be fixed? Explain all the problems in the code this programmer has written.

```
# subprogram:     PrintNewLine
# author:   Charles Kann
# purpose:  to output a new line to the user console
# input:    None
# output:   None
# side effects:   A new line character is printed to the
#           user's console
.text
PrintNewLine:
    li $v0, 4
    la $a0, __PNL_newline
    syscall
    jr $ra
```

```
.data
   __PNL_newline:   .asciiz "\n"

PrintTab:
    li $v0, 4
    la $a0, tab
    syscall
    jr $ra
.data
    tab: .asciiz "\t"
```

What you will learn.

In this chapter you will learn:

1. What is static and dynamic memory in the MIPS architecture.
2. What is the static data segment of memory.
3. How to allocate data memory in MIPS.
4. How to view the values stored in data memory using MARS.
5. Loading memory using memory addresses.
6. Various methods of loading data from memory into registers, including:
    6.1. Loading data values using labels
    6.2. Loading data values using register offsets
    6.3. Loading data values by modifying addresses through addition
    6.4. Loading data using 32 bit immediate values

Chapter 6 MIPS memory - the data segment

From a MIPS assembly language programmer's point of view, there are 3 main types of memory: static, stack dynamic and heap dynamic[16]. Static memory is the simplest as it is defined when the program is assembled and allocated when the program begins execution. Dynamic memory is allocated while the program is running, and accessed by address offsets. This makes dynamic memory more difficult to access in a program, but much more useful.

This chapter will only cover static memory. In MIPS this will also be called data memory because it will be stored in the `.data` segment of the program. Stack memory will be covered in Chapter 8 when describing general purpose subprograms, and heap memory will be covered in Chapter 9 when describing arrays.

From this point forward in the text, registers will be referred to as registers, and not memory. The term *memory* will always refer to data which is stored outside of the CPU.

Chapter 6. 1 Flat memory model

When dealing with memory, MIPS uses a flat memory model. In reality memory that exists in the hardware of the computer is quite complex, and the memory is sliced and diced, and sent out to many different areas in memory. This is a result of how hardware actually stores data. The actual implementation in hardware of memory uses virtual memory to store programs larger than the actual amount of memory on the computer, and layers of cache to make that memory appear faster.

[16] Note that static and dynamic memory is again an overloaded term in computer science. At a hardware level, dynamic memory is memory implemented using a capacitor and a transistor, and is normally used when a large amount of inexpensive memory is needed. Static memory is implemented using a circuit, and is more expensive. Static memory would be used in registers and possibly cache on a chip. There is absolutely no relationship between hardware level static and dynamic memory, and the way the terms are used in this chapter.

All of these details of how the memory is implemented are hidden by the Operating System (OS), and are not apparent to the programmer. To a MIPS programmer, memory appears to be flat; there is no structure to it. Memory consists of one byte (8 bits) stored after another, and all bytes are equal. The MIPS programmer sees a memory where the bytes are stored as one big array, and the index to the array being the byte address. The memory is addressable to each byte, and thus called *byte addressable*. This was shown in Figure 2.3, which is repeated here for convenience of reference.

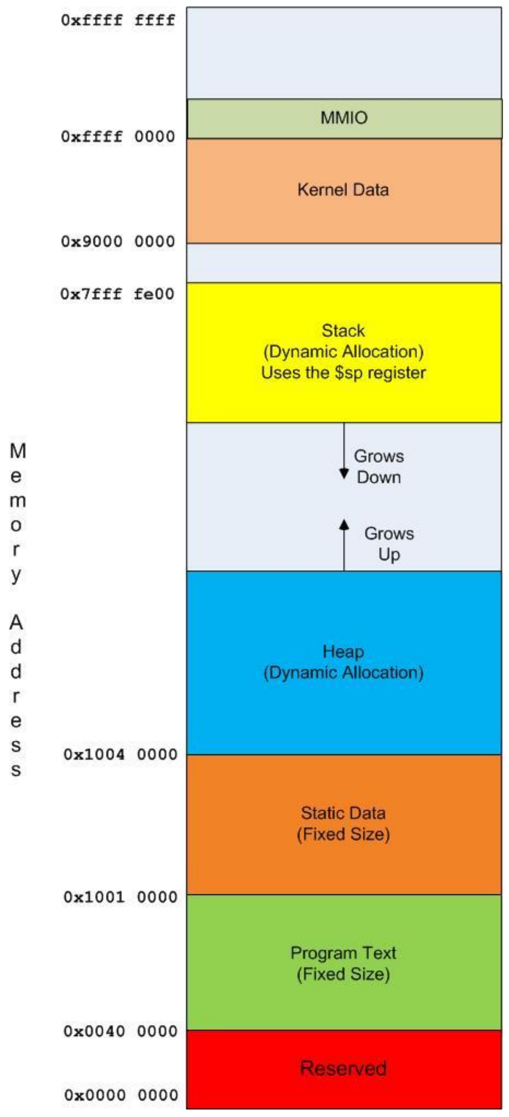


Figure 6-1: MIPS memory configuration

The bytes in MIPS are organized in groups of: 1) a single byte; 2) a group of 2 bytes, called a half-word; 3) a group of 4 bytes, called a word[17]; and 4) a group of 8 bytes, called a double word. These groupings are not random in memory. All groupings start at 0x00000000 and then occur at regular intervals. Thus memory half words would start at addresses 0x00000002, 0x00000004, 0x00000008, 0x0000000a, 0x0000000c, and continue in that manner. Memory words would start at addresses 0x00000000, 0x00000004, 0x00000008, 0x0000000c, 0x00000010, 0x00000014, and likewise continue. Memory double words would start at addresses 0x00000000, 0x00000008, 0x00000010, 0x00000018, and continue. Where the memory groups start is called a *boundary*. You cannot address a group of data except at the boundary for that type. So if you want to load a word of memory, you cannot specify the address 0x15fc8232, as it is not on a word boundary (it is however a byte boundary and a half word boundary). When discussing data, a word of memory is 4 bytes large (32 bits), but it is also located on a word boundary. If 32 bits are not aligned on a word boundary, it is incorrect to refer to it as a word[18].

Chapter 6. 2 Static data

Static data[19] is data that is defined when the program is assembled, and allocated when the program starts to run. Due to the fact that it is allocated once, the size and location of static data is fixed and cannot be changed. If a static array is allocated with 10 members, it cannot be resized to have 20 members. A string of 21 characters can never be more than 21 characters. In addition all variables which will be defined as static must be known before the program is run. These limitations make static data much less useful than stack dynamic and heap dynamic data, which will be covered later.

Static data is defined using the `.data` assembler directive. All memory allocated in the program in a `.data` segment is static data. The static data (or simply data) segment of memory is the portion of memory starting at address `0x10000000` and continuing until address `0x10040000`. The data elements that are defined come into existence when the program is started, and remain until it is completed. So while the data values can change during the course of a program execution, the data size and address does not change.

When the assembler starts to execute, it keeps track of the next address available in the data segment. Initially the value of the next available slot in the data segment is set to `0x10010000`. As space is allocated in the data segment, the next available slot is incremented by the amount of

[17] All machines define their own word size. Older computer used a 16 bit word, and some modern computers use a 64 bit word. Do not let this confuse you. In MIPS all words are 32 bits.

[18] To save space, some languages, such as C Ada, and PLI, allow programmers to use unaligned data structures, where some data might not fall on the correct boundaries. Unaligned structures were used when space was a premium in computers, but these structures are slow to process. Most modern languages do not allow unaligned data, but when dealing with legacy systems programmers might have to deal with it.

[19] Static is not used here in the same way as it is used in Java or C#. In these languages, static means *classwide*, or one per class. Static variables are associated with the class, not an instance of the class, and static methods cannot access instance variables. The original meaning of static came to these languages from C, where static is used in the same way as it is used here in assembler. There is no relationship between the Java or C# use of static, and static memory.

space requested. This allows the assembler to keep track of where to store the next data item. For example, consider the following MIPS code fragment.

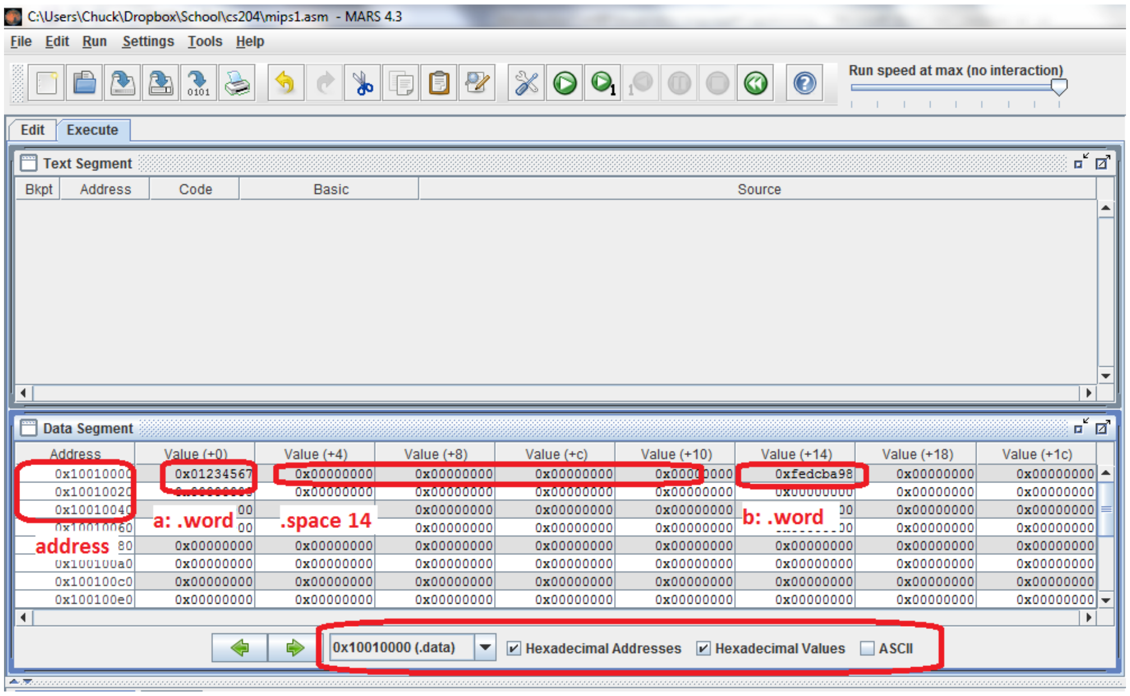
```
.data
a: .word 0x1234567
   .space 14
b: .word 0xFEDCBA98
```

If this is the first `.data` directive found, the address to start placing data is `0x10010000`. A word is 4 bytes of memory, so the label *`a:`* points to a 4 byte allocation of memory at address `0x10010000` and extending to `0x10010003`, and the next free address in the data segment is updated to be `0x10010004`.

Next an area of memory is allocated that using the `.space 14` assembly directive. The `.space` directive sets aside 14 bytes of memory, starting at `0x10010004` and extending to `0x10010011`. There is no label on this part of the data segment, which means that the programmer must access it directly through an address. This is perfectly valid, but is really not that useful, especially when there are multiple `.data` directives used, as it is hard for a programmer to keep track of the next address. Generally there will be a label present for variables in the data segment.

Finally another word of memory is allocated at the label `b:`. This memory could have been placed at `0x10010012` (18_{10}), as this is the next available byte. However specifying that this data item is a word means that it must be placed on a word boundary. Remember that words are more than just 32 bits (4 bytes) in MIPS. Words are successive groups of 4 bytes. Words are thus allocated on addresses which are divisible by 4. If the next available address is not on a word boundary when a word allocation is asked for, the assembler moves to the next word boundary, and the space between is simply lost to the program.

The following MARS screen shows the what the memory looks like after assembling this code fragment.


C:\Users\Chuck\Dropbox\School\cs204\mips1.asm - MARS 4.3

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x01234567	0x00000000	0x00000000	0x00000000	0x00000000	0xfedcba98	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	00	[illegible]	0x00000000	0x00000000	0x00000000	00	0x00000000	0x00000000
0x10010060	00	[illegible]	0x00000000	0x00000000	0x00000000	00	0x00000000	0x00000000
80	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Figure 6-2: Static data memory allocation example

This image shows the contents of the data segment of memory for the previous code fragment. Note the dropdown menu at the bottom of the screen that says the memory being viewed is the "`0x1001000(.data)`" segment. Other portions of memory can be viewed, but for now we will limit the discussions to the data segment.

This image shows there are a number of options for showing the values of the addresses and memory. The first check box, labeled "`Hexadecimal Addresses`", allows memory addresses can be shown in hex or decimal. The checkbox labeled "`Hexadecimal Values`" allows all memory and register values in the entire screen to be displayed in either hex or decimal, and is useful when looking for data values which the programmer knows in decimal. Finally the "`ASCII`" checkbox allows the data in the memory to be displayed as ASCII characters when appropriate. It is useful when looking for ASCII strings in memory.

To read the memory contents in the image, look at the column called address, and 8 columns after it. The column address gives the base address for a grouping of 32 (0x20) byte addresses. So the top row in this table is addresses starting at `0x1001 0000` and extending to `0x1001 001f`, the second row is addresses starting at `0x1001 0020` and extending to `0x1001 003f`, and so on. Each subsequent column is the 4 bytes (or word) offset from the base address. The first column is the `base address + 0` bytes, so it is addresses `0x1001 0000 - 0x1001 0003`, the second column is addresses `0x1001 0004 - 0x1001 0007`, and so on. From this image we can see that the memory at label `a`: stores `0x01234567`, then 14 bytes of uninitialized memory are allocated, the next two bytes of memory are unused and lost, and finally a word at label `b`: which stores `0xfedcba98`. This is what the assembly code had specified.

One last note about the memory. In HLL, memory always had a context. Memory was always a type, such as a string, or an integer, or a float, and so forth. In assembly there is no context maintained with the memory, so it is really just a collection of bits. The context of the memory is the operations that the programmer performs on it. This is very powerful, but it is hard to keep track of and very error prone. One of the biggest skills a programmer can learn by doing assembly is to be careful, and always keep track of the data in the program. In assembly language, there is no type definitions or predefined contexts for variables to keep a programmer from really messing up a program.

Chapter 6. 3 Accessing memory

All memory access in MIPS in done through some form of a load or store opertoar. These operators include loading/storing a: byte (lb, sb); half word (lh, sh); word (lw, sw); or double word (ld, sd)[20]. In this chapter only words of memory will be considered, so only the lw and sw will be documented.

- `lw` operator, which has a number of pseudo operator formats, but only two format will be documented here. The first format is the only real format of this operator. In this format, an address is stored in R_s, and an offset from that address is stored in the Immediate value. The value of memory at [R_s + Immediate] is stored into R_t.

```
format:       lw Rt, Immediate(Rs)
meaning:      Rt <- Memory[Rs + Immediate]
```

  The second format of the `lw` is a pseudo operator that allows the address of a label to be stored in R_s and then the real `lw` operator is called to load the value.

```
format:       lw Rs, Label
meaning:      Rt <- Memory[Label]
translation:  lui Rs, 0x10010000
              lw Rt, offset(Rs)  # offset is displacement of value
                                 # in the data segement
```

- `sw` operator, which has a number of pseudo operator formats, but only two format will be documented here. The first format is the only real format of this operator. In this format, an address is stored in R_s, and an offset from that address is stored in the Immediate value. The value of R_t is stored in memory memory at [R_s + Immediate].

```
format:       lw Rt, Immediate(Rs)
meaning:      Memory[Rs + Immediate] <- Rt
```

  The second format of the `sw` is a pseudo operator that allows the address of a label to be stored in R_s and then the real `lw` operator is called to load the value.

```
format:       lw Rs, Label
meaning:      Memory[label] <- Rt
```

[20] The la operator is not included here because it does not load a value from memory into a register. It is a pseudo operator which is short hand for taking the address of a label, and putting it in a register. It does not load memory values into a register.

translation:
```
lui Rs, 0x10010000
sw Rt, offset(Rs)  # offset is displacement of value
                   # in the data segement
```

Chapter 6. 4 Methods of accessing memory

The addressing mechanisms for the `lw` and `sw` operators shown above are very flexible, and can be used in a number of different ways. These different addressing mechanisms will all prove useful in retrieving memory values. Four methods of addressing data will be shown here, which will be called addressing by label, register direct, register indirect, and register offset, and memory indirect[21] (or just indirect).

Different ways of storing data will lend themselves to different mechanisms to access the data. For example, it will be natural for stack data to use register offset addressing, and natural for array processing to use register direct addressing. Addressing by label will be the one with that initially most readers will be most comfortable with, but will by far be the least useful. It is presented mostly to aid readers comfort level.

To illustrate accessing of memory, the following quadratic calculation program from chapter 2 is used. It implements the equation ax^2+bx+c based on the value of the user input of x, and prints the result. In the examples in the next 3 sections, the constants a=5, b=2, and c=3 will be stored in the data segment, and 3 different memory access methods shown to retrieve them. The pseudo code for this example follows. Note that the use of the volatile keyword tells the programmer that the variable a, b, and c must be stored in memory, and cannot be immediate values. The static keyword tells the program that the memory to be used should be in the data segment.

```
main
{
     static volatile int a = 5;
     static volatile int b = 2;
     static volatile int c = 3;
     int x = prompt("Enter a value for x: ");
     int y = a * x * x + b * x + c;
     print("The result is: " + y);
}
```

Program 6-1: Quadratic program pseudo code

Chapter 6.4. 1 Addressing by label

Sometimes the address of a variable is known, and a label can be defined for its address. This type of data can only exist in the .data segment of the program, which means that this data cannot move or change size. This is often true of program constants, as is the case here. When the variable is stored in the data segment, it can generally be addressed directly using a label.
In the following implementation of the quadratic calculation program the constants `a`, `b`, and `c` will be loaded from memory using the `lw` operator with labels. This is very similar to how

[21] For programmers unfamiliar with languages that include pointer variables, such as C/C++, the concept of indirect addressing is confusing and it is hard to justify why it would be used. Indirect addressing is included here because it is an important topic, but no justification of why it is used will be provided.

programmers are used to accessing variables using their label as a variable name. However be warned that what is being accessed is not the equivalent of a HLL variable or constant, as will become readily apparent throughout the rest of this text.

```
.text
.globl main
main:
   # Get input value and store it in $s0
   la $a0, prompt
   jal PromptInt
   move $s0, $v0

   # Load constants a, b, and c into registers
   lw $t5, a
   lw $t6, b
   lw $t7, c

   # Calculate the result of y=a*x*x + b * x + c and store it.
   mul $t0, $s0, $s0
   mul $t0, $t0, $t5
   mul $t1, $s0, $t6
   add $t0, $t0, $t1
   add $s1, $t0, $t7

   # Store the result from $s1 to y.
   sw $s1, y

   # Print output from memory y
   la $a0, result
   lw $a1, y
   jal PrintInt
   jal PrintNewLine

   #Exit program
   jal Exit

.data
a: .word 5
b: .word 2
c: .word 3
y: .word 0
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "
.include "utils.asm"
```

Program 6-2: Accessing memory variables by label

Chapter 6.4. 2 Register direct access

Register direct access violates the *volatile* keyword in the pseudo code (as did the use of immediate values), but is included here to show the difference between register direct access and register indirect addressing. In register direct access, the values are stored directly in the registers, and so memory is not accessed at all. The following program shows register direct access.

```
.text
.globl main
main:
   # Get input value and store it in $s0
   la $a0, prompt
   jal PromptInt
   move $s0, $v0

   # Load constants a, b, and c into registers
   li $t5, 5
   li $t6, 2
   li $t7, 3

   # Calculate the result of y=a*x*x + b * x + c and store it.
   mul $t0, $s0, $s0
   mul $t0, $t0, $t5
   mul $t1, $s0, $t6
   add $t0, $t0, $t1
   add $s1, $t0, $t7


   # Print output from memory y
   la $a0, result
   move $a1, $s1
   jal PrintInt
   jal PrintNewLine

   #Exit program
   jal Exit

.data
y: .word 0
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "
.include "utils.asm"
```

Program 6-3: Register Direct Access

Chapter 6.4. 3 Register indirect access

Register indirect access differs from register direct access in that the register does not contain the value to use in the calculation, but contains the address in memory of the value to be used. To see this, consider the following example. If the `.data` segment in this program is the first .data segment that the assembler has encountered, the numbering of variables in this segment begins at 0x10010000, so the variable `a` will be at that address. The next allocation that the assembler will find is for the variable `b`. Since the variable `a` took up 4 bytes of memory, the variable `b` will be at memory address 0x10010000 + 0x4 = 0x10010004. Likewise variable `c` will be at memory location 0x10010000 + 0x8 = 0x10010008, and variable `y` will be at 0x10010000 + 0xc = 0x1001000c. This next program illustrates how register indirect addressing works.

```
.text
.globl main
main:
   # Get input value and store it in $s0
   la $a0, prompt
```

```
      jal PromptInt
      move $s0, $v0

      # Load constants a, b, and c into registers
      lui $t0, 0x1001
      lw $t5, 0($t0)
      addi $t0, $t0, 4
      lw $t6, 0($t0)
      addi $t0, $t0, 4
      lw $t7, 0($t0)

      # Calculate the result of y=a*x*x + b * x + c and store it.
      mul $t0, $s0, $s0
      mul $t0, $t0, $t5
      mul $t1, $s0, $t6
      add $t0, $t0, $t1
      add $s1, $t0, $t7


      # Print output from memory y
      la $a0, result
      move $a1, $s1
      jal PrintInt
      jal PrintNewLine

      #Exit program
      jal Exit

   .data
      .word 5
      .word 2
      .word 3
   y: .word 0
   prompt: .asciiz "Enter a value for x: "
   result: .asciiz "The result is: "
   .include "utils.asm"
```

Program 6-4: Register Indirect Access

Chapter 6.4. 4 Register offset access

In the `lw` instruction, the immediate value is a distance from the address in the register to the value to be loaded. In the register indirect access, this immediate was always zero as the register contained the actual address of the memory value to be loaded. In this example, the value will be used to specify how far in memory the value to be loaded is from the address in the register. Again we will make use of the fact that the variable `b` is stored 4 bytes from the variable `a`, and the variable `c` is stored 8 bytes from the variable `a`.

```
   .text
   .globl main
   main:
      # Get input value and store it in $s0
      la $a0, prompt
      jal PromptInt
      move $s0, $v0
```

```
    # Load constants a, b, and c into registers
    lui $t0, 0x1001
    lw $t5, 0($t0)
    lw $t6, 4($t0)
    lw $t7, 8($t0)

    # Calculate the result of y=a*x*x + b * x + c and store it.
    mul $t0, $s0, $s0
    mul $t0, $t0, $t5
    mul $t1, $s0, $t6
    add $t0, $t0, $t1
    add $s1, $t0, $t7


    # Print output from memory y
    la $a0, result
    move $a1, $s1
    jal PrintInt
    jal PrintNewLine

    #Exit program
    jal Exit

.data
    .word 5
    .word 2
    .word 3
y: .word 0
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "
.include "utils.asm"
```

Program 6-5: Register offset access

If a register can contain the address of a variable in memory, by extension it seems reasonable that a memory value can contain a reference to another variable at another spot in memory. This is indeed true, and this is a very common way to access data and structures. These variables are called pointer variables, and exist in the compiled executables for all programming languages. However most modern programming languages prohibit the programmer from accessing these pointer variables directly, mainly because experience with languages that allowed access to these variables has shown that most programmers do not really understand them, or the implications of using them. Accessing these pointer variables is always unsafe, and the improper use of pointers has resulted in many of the worst bugs that many programmers have encountered.

Having warned against the use of pointer variables in a HLL, there is no way to avoid their use in assembly. Assembly programmers must understand how these variables work, and how to safely use them.

The following program shows the use of memory indirect (pointer) variables. The memory at the start of the `.data` segment (0x10010000) contains an address (or reference[22]) to the actual storage location for the constants `a`, `b`, and `c`. These variables are then accessed by loading the address in memory into a register, and using that address to locate the constants.

```
.text
.globl main
main:
   # Get input value and store it in $s0
   la $a0, prompt
   jal PromptInt
   move $s0, $v0

   # Load constants a, b, and c into registers
   lui $t0, 0x1001
   lw $t0, 0($t0)
   lw $t5, 0($t0)
   lw $t6, 4($t0)
   lw $t7, 8($t0)

   # Calculate the result of y=a*x*x + b * x + c and store it.
   mul $t0, $s0, $s0
   mul $t0, $t0, $t5
   mul $t1, $s0, $t6
   add $t0, $t0, $t1
   add $s1, $t0, $t7


   # Print output from memory y
   la $a0, result
   move $a1, $s1
   jal PrintInt
   jal PrintNewLine

   #Exit program
   jal Exit

.data
   .word constants
y: .word 0
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "
constants:
    .word 5
    .word 2
    .word 3
.include "utils.asm"
```

[22] Pointer variables, addresses, and references have largely the same meaning. In MIPS assembly, a reference is a pointer. However there are cases, such as accessing distributed data in a HLL (such as using Java RMI) where a reference could have some different semantics. So the terms pointer variables and addresses are generally called references in HLL, though in most cases they are the same thing.

Chapter 6. 5 Exercises

1) Why do "`la label`" instructions always need to be translated into 2 lines of pseudo code? What about "`lw label`" instructions? Explain the similarities and differences in how they are implemented in MARS.

2) The following table has memory addresses in each row, and columns which represent each of the MIPS boundary types, byte, half word, word, and double word. Put a check mark in the column if the address for that row falls on the boundary type for the column.

Address	**Boundary Type**			
	Byte	**Half**	**Word**	**Double**
0x10010011				
0x10010100				
0x10050108				
0x1005010c				
0x1005010d				
0x1005010e				
0x1005010f				
0x10070104				

3) The following program fails to load the value 8 into $t0. In fact it creates an exception. Why?

```
.text
    lui $t0, 1001
    lw $a0, 0($t0)
    li $v0, 1
    syscall

    li $v0, 10
    syscall
.data
    .word 8
```

4) Translate the following pseudo code into MIPS assembly to show each of the addressing modes covered in this chapter. Note that variables `x` and `y` are static and volatile, so should be stored in data memory. When using register direct access, you do not need to store the variables in memory.

```
main() {
    static volatile int miles =
        prompt("Enter the number of miles driven: ");
    static volatile int gallons =
        prompt("Enter the number of gallons used: ");
    static volatile int mpg = miles / gallons;
    output("Your mpg = " + mpg);
}
```

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:
    a) if statements
    b) if-else statements
    c) if-elseif-else statements
    d) sentinel control loops
    e) counter control loops.
4) How to calculate branch offsets.

Chapter 7 Assembly language program control structures

The structured programming paradigm is built on the concept that all programs can be built using just 3 types of program control structures. These structures are:

- *Sequences* that allow programs to execute statements in order one after another.
- *Branches* that allow programs to jump to other points in a program.
- *Loops* that allow a program to execute a fragment of code multiple times.

Most modern HLLs are implemented based on these 3 program control structures, with some notable extensions such as Java exception handling, continue, and break statements.

But as was pointed out in Chapter 5 on simple subprogram 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 spaghetti noodles intertwine and following individual strands becomes difficult.

This chapter will introduce into pseudo code structure programming control structures similar to those in Java/C/C++/C#. Programmers familiar with those languages should be able follow the programs with no problems. The text will then show how to translate each control structure from pseudo code into assembly.

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 following the translations. 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.

Chapter 7. 1 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 principals, this simply misses the larger point and is incorrect except for limited and defined circumstances.

The problem with goto statements is that they allow unrestricted branching to any point in a program. Indeed this type of unrestricted branching lead 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, resulting 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 most often using branch statements. The branch statements themselves will not lead to unorganized programs, but the unorganized thoughts of the programmers. 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 defined, and all branches will be within blocks of code, just as in structured programming languages. So the branch statements in this text are not evil, and the idea that somehow branching is wrong needs to be modified in the readers mind.

Note that there is also a practical reason why branching is used here. There is no other available mechanism to use to implement program control structures such as branches and loops. So regardless of the readers qualms about this topic, the reality of the situation is that branch statements are the only valid mechanism to implement programs in assembly langauge.

Chapter 7. 2 Simple if statements

This section will deal with simple if statement, e.g. `if` statements that do not have any else conditions. This section will give two examples. The first shows how to translate a pseudo code program with a single logical condition in the `if` statement. The second shows how to handle complex logical conditions.

Chapter 7.2. 1 Simple if statements in pseudo code

This section will begin with a small example of an `if` statement.

```
if (num > 0)
{
    print("Number is positive")
}
```

This program fragment looks simple, but as we have already seen there is a lot of complexity hidden in it. First, the variable num will not be directly useable in the program, and will have to be loaded into a register, and the subprogram for print must be created. But hidden in this program fragment are several conditions that are important in understanding the `if` statement.

1) The statement `(num > 0)` is a statement in which the > operator is returning a logical (boolean) value to be evaluated. It might be easier if the statement was written as follows, which has exactly the same meaning.

    ```
    boolean flag = num > 0;
    if (flag) ...
    ```

    There is no mechanism for putting an expression in a branch statement, so the value of the `boolean` variable will have to be calculated before it will be used in assembly language, just as in this example.

    One peculiarity in assembly (as in C/C++) is that logical values are 0 (false) and anything else (true). This can lead to all sorts of strange consequences, so this behavior will not be allowed, and all boolean values will strictly be 0 (false) and 1 (true).

2) Code blocks are the central organizing unit in this pseudo code. Any code between an open brace "{" and close brace "}" is considered part of a code block. Subprograms consist of a code block, and all control structures must use code blocks to implement the code within a condition.

Chapter 7.2. 2 Simple if statement translated to assembly

The assembly language program for the code fragment above is shown below.

```
.text
    # if (num > 0 )
    lw $t0, num
    sgt $t1, $t0, $zero  # $t1 is the boolean (num > 0)
    beqz $t1, end_if     # note: the code block is entered if
```

```
                             # if logical is true, skipped if false.
                 # {
                 #    print ("Number is positive")
                 la $a0, PositiveNumber
                 jal PrintString
                 # }
             end_if:
             jal Exit
         .data
             num: .word 5
             PositiveNumber: .asciiz "Number is positive"
         .include "utils.asm"
```

Program 7-1: A simple program to show an if statement

In this code fragment comments are inserted to show how the pseudo code is translated into assembly. The following comments help to explain how this program works.

1) The translation of `(num > 0)` takes 2 assembly instructions. The first loads num into `$t1` so that the values can be used. The `sgt $t1, $t0, $zero` instruction loads the `boolean` value into `$t1` so that it can be compared in the `if` test.

2) The if test works by asking the question is the `boolean` value true? If the `boolean` value is true, then the code block is entered (the branch is not taken). If the test is false, branch to the end of the code block, and so the code block is not entered. This might appear backward, as the branch happens if the condition is false, but it is the easiest way to implement the logic (the exercises ask how to implement this if the branch is taken if the condition is true). The reader will quickly grow accustomed to this if used consistently, and doing things in a consistent manner is the best defense against annoying bugs.

3) There are over 20 formats for the branch instruction in MARS. This text will only used 2. If the branch is based on a condition, the `beqz` will always be chosen. The `seq`, `slt`, `sle`, `sgt`, `sge` operators will be used to set the correct condition for the `beqz` operator. The only other branch used will be the unconditional branch instruction, `b`.

4) When implementing a code block, the following will always be used. The final } for the code block will translate into a label. This label will be accessible from the branch statement which is determining whether or not to enter the code block. Thus the simple if code fragment above is translated by:

    a) calculating the `boolean` value to control entering the if statement code block.
    b) entering the code block if the `boolean` is true, or branching around it if it is false.

Chapter 7.2. 3 Simple if statement with complex logical conditions

While the example above showed how to translate a single logical condition, the question of how to translate complex logical conditions is more complex. Programmers might think that to translate a condition such as the one that follows requires complex programming logic.

```
    if ((x > 0 && ((x%2) == 0))  # is x > 0 and even?
```

In fact 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 using programming logic.

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 single equation. So the complex if statement above would be translated into the equivalent of the following code fragment:

```
    boolean flag = ((x > 0) && ((x%2) == 0))
    if (flag)...
```

This code fragment is easily translated into assembly language as follows:

```
    lw $t0, x
    sgt $t1, $t0, $zero
    rem $t2, $t0, 2
    and $t1, $t1, $t2
    beqz $t1, end_if
```

Program 7-2: Assembly logic for ((x > 0) && ((x%2) == 0))

Once again it is left as an exercise for the programmer to convince themselves that this is much easier than implementing the logical condition using logic and various branch statements.

The true power of this method of handling logical conditions becomes apparent as the logical conditions become more complex. Consider the following logical condition:

```
    if ((x > 0) && ((x%2) == 0) && (x < 10))  # is 0 < x < 10 and even?
```

This can be translated into assembly language exactly as it appears in pseudo code.

```
    lw $t0, x
    sgt $t1, $t0, $zero
    li $t5, 10
    slt $t2, $t0, $t5
    rem $t3, $t0, 2
    and $t1, $t1, $t2
    and $t1, $t1, $t3
    beqz $t1, end_if
```

Program 7-3: Assembly language logic for ((x > 0) && ((x%2) == 0) && (x < 10))

Chapter 7. 3 if-else statements

A more useful version of the if statement also allows for the false condition, or a if-else statement. If the condition is true, the first block is executed, otherwise the second block is executed. A simple code fragment that illustrates this point is shown below.

```
  if (($s0 > 0) == 0)
  {
     print("Number is positive")
  }
  else
```

```
{
   print("Number is negative")

}
```

This is a modification to logic in Program 7.1. This code will print if the number is positive or negative. The purpose here is to show how to translate an if-else statement from pseudo code to assembly language. The translation of the if-else statement occurs using the following steps.

1) Implement the conditional part of the statement as in program 7.1.

2) Add two labels to the program, one for the else and one for the end of the if. The `beqz` should be inserted after the evaluation of the condition to branch to the else label. When you have completed step 2, you will have code that looks similar to the following:

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 you code should like the following assembly code fragment.

```
lw $t0, num
sgt $t1, $t0, $zero
beqz $t1, else
#if block
   b end_if
#else block
else:
end_if:
```

Program 7-4: Assembly code fragement for an if-else statement

4) Once the structure of the if-else statement is in place, you should put the code for the block into the structure. This completes the if-else statement translation. This is the following program.

```
.text
    lw $t0, num
    sgt $t1, $t0, $zero
    beqz $t1, else
       #if block
       la $a0, PositiveNumber
       jal PrintString
       b end_if
       #else block
    else:
        la $a0, NegativeNumber
        jal PrintString
    end_if:
    jal Exit
.data
    num: .word -5
    PositiveNumber: .asciiz "Number is positive"
    NegativeNumber: .asciiz "Number is negative"
.include "utils.asm"
```

Program 7-5: if-else program example

Chapter 7. 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. 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 program which 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. Students and programmers are strongly encouraged to implement algorithmic logic in this manner. Students who want to implement the code in a completely forward fashion, where statements are generated as they exist in the program, will find themselves completely overwhelmed and will miss internal and stopping conditions, especially when nest blocks containing nested logic is used late 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 end_if 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 end_if
grade_A:
    b end_if
grade_B:
    b end_if
grade_C:
    b end_if
grade_D:
    b end_if
else:
    b end_if
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 like the following:

```
#if block
    lw $s0, num
    slti $t1, $s0, 0
    sgt $t2, $s0, 100
    or $t1, $t1, $t2
    beqz $t1, grade_A
    #invalid input block
    b end_if
grade_A:
    sge $t1, $s0, 90
    beqz $t1, grade_B
    b end_if
grade_B:
    sge $t1, $s0, 80
    beqz $t1, grade_C
    b end_if
grade_C:
    sge $t1, $s0, 70
    beqz $t1, grade_D
    b end_if
grade_D:
    sge $t1, $s0, 60
    beqz $t1, else
    b end_if
else:
    b end_if
end_if:
```

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.

```
.text
    #if block
        lw $s0, num
        slti $t1, $s0, 0
        sgt $t2, $s0, 100
        or $t1, $t1, $t2
        beqz $t1, grade_A
```

```
            #invalid input block
            la $a0, InvalidInput
            jal PrintString
            b end_if
        grade_A:
            sge $t1, $s0, 90
            beqz $t1, grade_B
            la $a0, OutputA
            jal PrintString
            b end_if
        grade_B:
            sge $t1, $s0, 80
            beqz $t1, grade_C
            la $a0, OutputB
            jal PrintString
            b end_if
        grade_C:
            sge $t1, $s0, 70
            beqz $t1, grade_D
            la $a0, OutputC
            jal PrintString
            b end_if
        grade_D:
            sge $t1, $s0, 60
            beqz $t1, else
            la $a0, OutputD
            jal PrintString
            b end_if
        else:
            la $a0, OutputF
            jal PrintString
            b end_if
        end_if:

        jal Exit
    .data
        num: .word 70
        InvalidInput: .asciiz "Number must be > 0 and < 100"
        OutputA: .asciiz "Grade is A"
        OutputB: .asciiz "Grade is B"
        OutputC: .asciiz "Grade is C"
        OutputD: .asciiz "Grade is D"
        OutputF: .asciiz "Grade is F"

    .include "utils.asm"
```

Program 7-6: Program to implement if-elseif-else statement in assembly

Chapter 7. 5 Loops

There are two basic loop structures found in most introduction to programming books. These two loops structures are sentinel controlled loops and counter controlled loops. These loops are similar to while loops and for loops in most programming languages, so in this text the while loop will be used to implement sentinel control loops, and the for loop to implement counter control loops. How to translate each of these looping structures from pseudo code into assembly language will be covered in the next two sections.

Chapter 7.5. 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) Set the sentinel to be checked before entering the loop.

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 code in place to check the sentinel. If the sentinel check is true, branch to the end of the loop.

5) Set the sentinel to be checked as the last statement in the code block for the loop, and 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
    #set sentinel value (prompt the user for input).
    la $a0, prompt
    jal PromptInt
    move $s0, $v0
    start_loop:
        sne $t1, $s0, -1
        beqz $t1, end_loop
        # code block
        la $a0, prompt
        jal PromptInt
        move $s0, $v0
        b start_loop
    end_loop:
.data
    prompt: .asciiz "Enter an integer, -1 to stop: "
```

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
    #set sentinel value (prompt the user for input).
    la $a0, prompt
    jal PromptInt
    move $s0, $v0
    start_loop:
        sne $t1, $s0, -1
        beqz $t1, end_loop

        # code block
        la $a0, output
        move $a1, $s0
        jal PrintInt

        la $a0, prompt
        jal PromptInt
        move $s0, $v0
        b start_loop
    end_loop:
    jal Exit
.data
    prompt: .asciiz "\nEnter an integer, -1 to stop: "
    output: .asciiz "\nYou entered: "
.include "utils.asm"
```

Program 7-7: Sentinel control loop program

Chapter 7.5. 2 Counter control loop

A counter controlled loop is a loop which is intended to be executed some number of times. Normally this is associated with a for loop in most HLL. The general format is to specify a starting value for a counter, the ending condition (normally when the counter reaches a predetermined value) , and the increment operation on the counter. 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; # Initial the total variable for sum
for (i = 0; i < n; i++)
{
    total = total + i
}
print("Total = " + total);
```

The `for` statement itself has 3 parts. The first is the initialization that occurs before the loop is executed (here it is "`i=0`"). The second is the condition for continuing to enter the loop (here it is "`i < size`"). The final condition specifies how to increment the counter (here it is "`i++`", or add 1 to i). These 3 parts are included in the translation of the structure.

The following defines the steps involved in translating a sentinel control loop from pseudo code into assembly.

1) Implement the initialization step to initialize the counter and the ending condition variables.
2) Create labels for the start and end of the loop.
3) Implement the check to enter the loop block, or stop the loop when the condition is met.
4) Implement the counter increment, 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
    li $s0, 0
    lw $s1, n
    start_loop:
        sle $t1, $s0, $s1
        beqz $t1, end_loop

        # code block

        addi $s0, $s0, 1
        b start_loop
    end_loop:
.data
    n: .word 5
```

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
    la $a0, prompt
    jal PromptInt
    move $s1, $v0
    li $s0, 0
    li $s2, 0 # Initialize the total

    start_loop:
        sle $t1, $s0, $s1
        beqz $t1, end_loop

        # code block
        add $s2, $s2, $s0

        addi $s0, $s0, 1
        b start_loop
    end_loop:

    la $a0, output
    move $a1, $s2
    jal PrintInt

    jal Exit
.data
    prompt: .asciiz "enter the value to calculate the sum up to: "
    output: .asciiz "The final result is: "

.include "utils.asm"
```

Program 7-8: Counter control loop program

Chapter 7. 6 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 important 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);
    }
}
```

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 (see the exercises at the end of the chapter).

However the translation of this pseudo code into assembly is a relatively straight forward process, as will be illustrated here.

1) Begin by implementing the outer most block, the sentinel control block. Your code should look similar to the following:

```
# Sentinel Control Loop
```

```
la $a0, prompt
jal PromptInt
move $s0, $v0
start_outer_loop:
    sne $t1, $s0, -1
    beqz $t1, end_outer_loop

    # code block

    la $a0, prompt
    jal PromptInt
    move $s0, $v0
    b start_outer_loop
end_outer_loop:
.data
    prompt: .asciiz "Enter an integer, -1 to stop: "
```

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. When completed, your code should look similar to the following:

```
# Sentinel Control Loop
la $a0, prompt
jal PromptInt
move $s0, $v0
start_outer_loop:
    sne $t1, $s0, -1
    beqz $t1, end_outer_loop

    # If test for valid input
    slti $t1, $s0, -1
    beqz $t1, else
       #if block
       b end_if
    else:
       #else block
    end_if:

    la $a0, prompt
    jal PromptInt
    move $s0, $v0
    b start_outer_loop
end_outer_loop:
```

3) The if block in the above code fragment is replaced by the error message, and the else block is replaced by the sentinel control loop. This entire code fragment is then placed in the program, resulting in the following complete program.

```
.text
    # Sentinel Control Loop
    la $a0, prompt
    jal PromptInt
    move $s0, $v0
    start_outer_loop:
        sne $t1, $s0, -1
        beqz $t1, end_outer_loop
```

```
                # If test for valid input
              slti $t1, $s0, -1
              beqz $t1, else
                la $a0, error
                jal PrintString
                        b end_if
                else:
                    # summation loop
                    li $s1, 0
                    li $s2, 0 # initialize total

                    start_inner_loop:
                        sle $t1, $s1, $s0
                        beqz $t1, end_inner_loop

                        add $s2, $s2, $s1

                        addi $s1, $s1, 1
                        b start_inner_loop
                    end_inner_loop:
                    la $a0, output
                    move $a1, $s2
                    jal PrintInt

                end_if:

            la $a0, prompt
            jal PromptInt
            move $s0, $v0
            b start_outer_loop
        end_outer_loop:
        jal Exit
    .data
        prompt: .asciiz "\nEnter an integer, -1 to stop: "
        error:  .asciiz "\nValues for n must be > 0"
        output: .asciiz "\nThe total is: "
    .include "utils.asm"
```

Program 7-9: Program illustrating nested blocks

Though somewhat long, this assembly code is straight forward to produce, and relatively easy to follow, particularly if the documentation at the start of the program includes pseudo code for the algorithm that was used.

Chapter 7. 7 A full assembly language program

Now that the basics have been covered, a real assembly language program will be implemented. This program will read numeric grades from a user and calculate an average. The average and corresponding letter grade will be printed to the console.

Before starting the project, it is recommended that the pseudo code be written. This serves two purposes. First it allows the programmer to reason at a higher level of abstraction, and it makes it easier to implement the code because it is a straight translation from pseudo code to assembly. Second, the pseudo code serves as documentation for how the program works. The pseudo code should be included in a comment at the start of the assembly file, and not kept in a separate file so it does not get lost.

Once the pseudo code is written, the assembly code can be implemented. This is done below. Note that this is a program, and not a code fragment that is used to illustrate a MIPS assembly feature. Therefore this has a preamble comment giving information such as Filename, Author, Date, Purpose, Modification History, and the Pseudo Code. Most professors and companies have their own standard for preamble comments, and they should be followed when documenting the code.

Finally realize that it is a myth that assembly code is not readable. If care is taken when writing it and documenting it, it can be just as readable as code in a higher level language. However the code is much more verbose, and the ability to use abstraction is greatly reduced. Just as a high level language is no substitute for assembly when it is needed, assembly is no substitute for a high level language when it is appropriate.

```
# Filename: AverageGrade.asm
# Author:   Charles Kann
# Date:           12/29/2013
# Purpose:  Illustration of program to calculate a student grade
# Modification Log:
#     12/29/2014 - Initial release
#
# Pseudo Code
#global main()
#{
#    // The following variables are to be stored in data segment, and
#    // not simply used from a register.  They must be read each time
#    // they are used, and saved when they are changed.
#    static volatile int numberOfEntries = 0
#    static volatile int total = 0
#
#    // The following variable can be kept in a save register.
#    register int inputGrade  # input grade from the user
#    register int average
#
#    // Sentinel loop to get grades, calculate total.
#    inputGrade = prompt("Enter grade, or -1 when done")
#    while (inputGrade != -1)
#    {
#        numberOfEntries = numberOfEntries + 1
#        total = total + inputGrade
#        inputGrade = prompt("Enter grade, or -1 when done")
#    }
#
#    # Calculate average
#    average = total / numberOfEntries
#
#    // Print average
#    print("Average = " + average)
#
#    //Print grade if average is between 0 and 100, otherwise an error
#    if ((grade >= 0) & (grade <= 100))
#    {
      #        if (grade >= 90)
      #        {
      #            print("Grade is A")
      #        }
      #        if (grade >= 80)
      #        {
      #            print("Grade is B")
      #        }
      #        if (grade >= 70)
      #        {
      #            print("Grade is C")
      #        }
      #        else
      #        {
      #            print("Grade is F")
      #        }
      #    }
      #    else
      #    {
      #        print("The average is invalid")
      #    }
      #}

.text
.globl main
main:
    # Register Conventions:
    #    $s0 - current inputGrade
    #    $s1 - average
    la $a0, prompt
    jal PromptInt
    move $s0, $v0

    BeginInputLoop:
    addi $t0, $zero, -1        # set condition $s0 != -1
    sne $t0, $t0, $s0
    beqz $t0, EndInputLoop     # check condition to end loop

      la $t0, numberOfEntries # increment # of entries
      lw $t1, 0($t0)
      addi $t1, $t1, 1
      sw $t1, 0($t0)

      la $t0, total            # accumulate total
      lw $t1, 0($t0)
      add $t1, $t1, $s0
      sw $t1, 0($t0)

      la $a0, prompt           # prompt for next input
      jal PromptInt
      move $s0, $v0
      b BeginInputLoop
    EndInputLoop:

    la $t0, numberOfEntries   #Calculate Average
    lw $t1, 0($t0)
    la $t0, total
    lw $t2, 0($t0)
    div $s1, $t2, $t1

    la $a0, avgOutput          # Print the average
    move $a1, $s1
    jal PrintInt
    jal PrintNewLine


    sge $t0, $s1, 0            # Set the condition
                               #(average > 0) & (average < 100)
    addi $t1, $zero, 100
    sle $t1, $s1, $t1
    and $t0, $t0, $t1
    beqz $t0, AverageError     # if Not AverageError
        sge $t0, $s1, 90       # PrintGrades
        beqz $t0, NotA
            la $a0, gradeA
            jal PrintString
            b EndPrintGrades
        NotA:
            sge $t0, $s1, 80
            beqz $t0, NotB
            la $a0, gradeB
            jal PrintString
            b EndPrintGrades
        NotB:
            seq $t0, $s1, 70
            beqz $t0, NotC
            la $a0, NotC
            la $a0, gradeC
            jal PrintString
            b EndPrintGrades
        NotC:
            la $a0, gradeF
            jal PrintString
        EndPrintGrades:
        b EndAverageError
    AverageError:                    #else AverageError
        la $a0, invalidAvg
        jal PrintString
    EndAverageError:

    jal Exit

.data
    numberOfEntries: .word 0
    total:           .word 0
    average:         .word
    prompt:          .asciiz "Enter grade, or -1 when done: "
    avgOutput:       .asciiz "The average is "
    gradeA:          .asciiz "The grades is an A"
    gradeB:          .asciiz "The grade is a B"
    gradeC:          .asciiz "The grade is a C"
    gradeF:          .asciiz "The grade is a F"
    invalidAvg:       .asciiz "The average is invalid"
.include "utils.asm"
```

Chapter 7. 8 How to calculate branch amounts in machine code

This chapter has shown how to use the branch statements to implement structure programming logic. However how a branch statement manipulates the `$pc` register to control the execution has yet to be discussed. This section will cover the details of how the branch statement is implemented in machine code.

Chapter 7.8. 1 Instruction Addresses

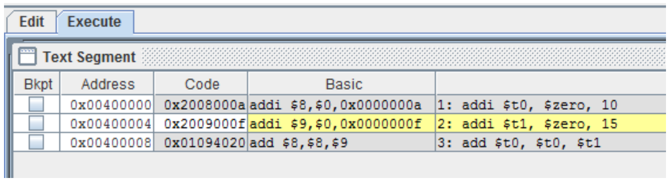
When the memory for the MIPS computer was shown in section 3.2, a segment labeled p*rogram text* (or simply t*ext*) was shown as starting at address 0x00400000. This section of the memory contains the machine code translation of the instructions from the .text segment of your program. Thus the text segment of memory is where all the machine code instructions for the program are stored.

When a program is assembled, the first instruction is inserted at address 0x0040000. The instructions for the program each take 4 bytes, so the assembler keeps an internal counter, and for each instruction it adds 4 to that counter and uses that number for the address of the next instruction. The new instruction is then placed at that address in the memory, and the process is continued to allow each subsequent assembly instruction inserted at the next available word boundary. Thus the first instruction in a program is at address 0x00400000, the second instruction is at address 0x00400004, etc. Note that machine instructions must always start on a word boundary.

A simple example of an assembled program is the following.

```
addi $t0, $zero, 10
addi $t1, $zero, 15
add $t0, $t0, $t1
```

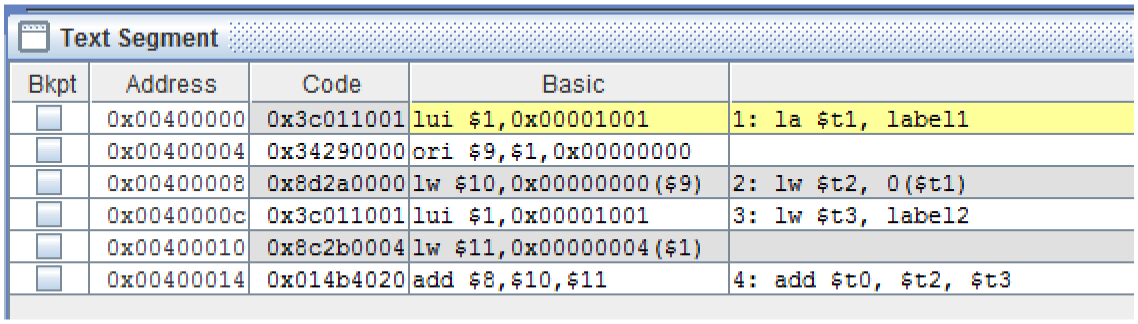
This program is shown below in a MARS screen image. The *Address* column of the grid shows the address of the instruction. In this example, the first instruction is stored at 0x00400000, the second at 0x00400004, and the third at 0x00400008.



Bkpt	Address	Code	Basic	
☐	0x00400000	0x2008000a	addi $8,$0,0x0000000a	1: addi $t0, $zero, 10
☐	0x00400004	0x2009000f	addi $9,$0,0x0000000f	2: addi $t1, $zero, 15
☐	0x00400008	0x01094020	add $8,$8,$9	3: add $t0, $t0, $t1

Figure 7-1: Instruction addresses for a simple program

So if all of the instructions are real instructions, placing the instructions at the correct address is as simple as adding 4 to each previous instruction. The problem is pseudo operators, as one pseudo instruction can map to more than one real instruction. For example, a la pseudo instruction always 2 takes instructions, and thus takes up 8 bytes. Some instructions, such as immediate instructions which can have either 16 bit or 32 bit arguments, can be different lengths depending on the arguments. Thus it is important to be able to translate pseudo operators into real instructions. This is shown in the following example program. Note how the Source column is translated in real instructions in the Basic column. The real instructions are the ones that are numbered in the Source column.



Text Segment

Bkpt	Address	Code	Basic	
☐	0x00400000	0x3c011001	lui $1,0x00001001	1: la $t1, label1
☐	0x00400004	0x34290000	ori $9,$1,0x00000000	
☐	0x00400008	0x8d2a0000	lw $10,0x00000000($9)	2: lw $t2, 0($t1)
☐	0x0040000c	0x3c011001	lui $1,0x00001001	3: lw $t3, label2
☐	0x00400010	0x8c2b0004	lw $11,0x00000004($1)	
☐	0x00400014	0x014b4020	add $8,$10,$11	4: add $t0, $t2, $t3

Figure 7-2: Instruction addresses for program with pseudo operators

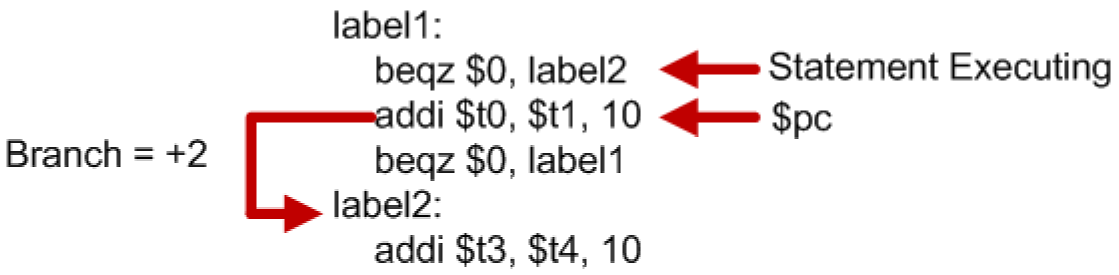
It is important to be able to number the instructions correctly to calculate branch offsets.

Chapter 7.8. 2 Value in the $pc register

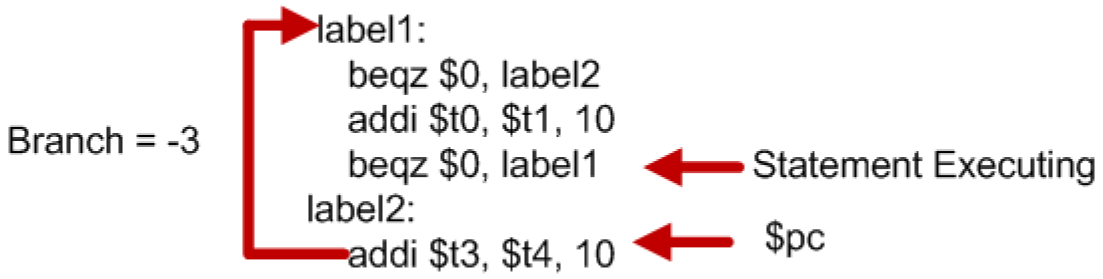
Branches in MIPS assembly work by adding or subtracting the value of the immediate part of the instruction from the `$pc` register. So if a branch is taken, `$pc` is updated by the immediate value, and control of the program continues at the labeled instruction..

Earlier in Chapter 3 when it was explained how the `$pc` register is used to control the flow of the program, it was apparent that at the start of each instruction the `$pc` points to the instruction to execute. So a reader could think that the value to be incremented by the immediate part of the branch is the address of the current instruction. However when an instruction executes, the first thing that is done is the `$pc` is incremented by 4 to point to the next instruction. This makes sense since in the majority of instances the program is processed sequentially. However this means that when a branch is executed the amount which must be added or subtracted will be from the next sequential instruction, not the current instruction.

The following example shows how this works. In the first branch instruction, the branch is to label2. The distance between this instruction and the label consists of 3 real instructions, which is 3 words or 12 bytes, from the current instruction. However since the `$pc` was already incremented to point to the next instruction, the branch will be incremented by 8 bytes, not 12.

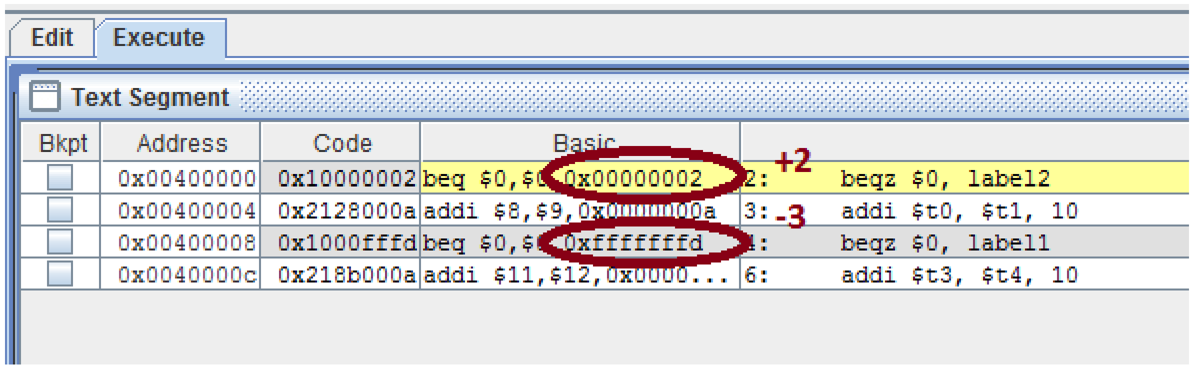


The second branch instruction branches backward to label1. In this case, the distance between the instruction and the label is -2 instructions, which is 2 words or 8 bytes, back from the current instruction. However because the `$pc` is incremented to point to the next instruction, -3 words, or -12 bytes, must be subtracted from the `$pc` in the branch instruction.



```
label1:
    beqz $0, label2
    addi $t0, $t1, 10
    beqz $0, label1
label2:
    addi $t3, $t4, 10
```

The following MARS screen shot shows that this is indeed the branch offsets for each of the branch instructions.



Bkpt	Address	Code	Basic		
☐	0x00400000	0x10000002	beq $0,$0,0x00000002	2:	beqz $0, label2
☐	0x00400004	0x2128000a	addi $8,$9,0x0000000a	3:	addi $t0, $t1, 10
☐	0x00400008	0x1000fffd	beq $0,$0,0xfffffffd	4:	beqz $0, label1
☐	0x0040000c	0x218b000a	addi $11,$12,0x0000...	6:	addi $t3, $t4, 10

Figure 7-3: Branch offset program example

Chapter 7.8. 3 How the word boundary effects branching

Remember that the I format instruction uses a 16 bit immediate value. If this was the end of the story, then branches could be up to 64K bytes from the current `$pc`. In terms of instructions, this means that a branch can access instructions that are -8191..8192 real instructions from the current instruction. This may be sufficient for most cases, but there is a way to allow the size of the branch offset to be increased to 2^{18} bits. Remember that all instructions must fall on a word boundary, so the address will always be divisible by 4. This means that the lowest 2 bits in every address must always be "00". Since we know the lowest two bits must always be "00", there is no reason to keep them, and they are dropped. Thus the branch forward in the previous instruction is 2 (1000_2 >> 2 = 0010_2 , or more simply 8/4 = 2). The branch backward is likewise -3 (110100_2 >> 2 = 11101_2 , or more simply -12/4 = -3).

Be careful to remember that the branch offsets are calculated in bytes, and that the two lowest order 00 bits have been truncated and must be reinserted when the branch address is calculated. The reason this caution is given is that the size of the offset in the branch instruction is the number of real instructions the current `$pc` needs to be incremented/decremented. This is just a happy coincidence. It makes calculating the offsets easier, as all that needs to be done is count the number of real instructions between the `$pc` and the label, but that in no way reflects the true meaning of the offset.

Chapter 7.8. 4 Translating branch instructions to machine code

Now that the method of calculating the branch offsets for the branch instructions has been explained, the following program shows an example of calculating the branch offsets in a program. Note that in this example the trick of dropping the last two bits of the address will be used, so the branch offsets can be used simply by adding/subtracting line numbers. Therefore the text will read "the `$pc` points to line", which is correct, as opposed to "the `$pc` contains the address of line", which would be incorrect.

1) Start with the program as written by the programmer. Note that there are 3 branch statements. Only these 3 branch statements will be translated to machine code. In this case the entire program, including comments, is included so that the reader understands the program. However comments are not kept when a translation to machine code is made, so the subsequent presentations of these programs will drop the comments.

```
# Filename: PrintEven.asm
# Author:   Charles Kann
# Date:           12/29/2013
# Purpose:  Print even numbers from 1 to 10
# Modification Log:
#     12/29/2013 - Initial release
#
# Pseudo Code
#global main()
#{
#    // The following variable can be kept in a save register.
#    register int i
#
#    // Counter loop from 1 to 10
#    for (i = 1; i < 11; i++)
#    {
#        if ((i %2) == 0)
#        {
#            print("Even number: " + i)
#        }
#    }
#}

.text
.globl main
main:
    # Register Conventions:
    #    $s0 - i
    addi $s0, $zero, 1

    BeginForLoop:
    addi $t0, $zero, 11
    slt $t0, $s0, $t0
    beqz $t0, EndForLoop
        addi $t0, $zero, 2
        div $s0,$t0
        mfhi $t0
        seq $t0, $t0, 0
        beqz $t0, Odd
```

```
            la $a0, result
            move $a1, $s0
            jal PrintInt
            jal NewLine

          Odd:
          addi $s0, $s0, 1
          b BeginForLoop
        EndForLoop:

        jal Exit
    .data
        result: .asciiz "Even number: "
    .include "utils.asm"
```

2) The next step is to translate all pseudo instructions in the program into real instructions, and then number each instruction.

Line #	Label	Statement
1		addi $16, $0, 0x00000001
2	BeginForLoop	addi $8, $0, 0x0000000b
3		slt $8, $16, $8
4		beq $8, $0, ????? (label EndForLoop)
5		addi $8, $0, 0x00000002
6		div $16,$8
7		mfhi $8
---		#seq $t0, 4t0, 0 is 4 real instructions
8		addi $1, $0, 0x00000000
9		subu $8, $8, $1
10		ori $1, $0, 0x00000001
11		sltu 48, $8, $1
12		beq $8, $0, ???? (label Odd)
----		# la $a0, result is 2 real instructions
13		lui $1, 0x00001001
14		ori $r, $1, 0x00000000
15		addu $5, 40, $16
16		jal ----- (doesn’t matter at this point)
17		jal ----- (doesn’t matter at this point)
18	Odd	addi $16, $16, 0x00000001
19		beq $0, $0, ???? (label BeginForLoop)
20	EndForLoop	jal ---- (doesn’t matter at this point)

3) Calculate the offsets. The first branch instruction, "`beq $t0, EndForLoop`" is at line 4, so the `$pc` when it is executing would point to line 5. The label is at line 20, so the branch

offset would be 15. The beq instruction is an I type instruction with an op-code of 0x4, so the machine code translation of this instruction 0x1100000f.

The next branch instruction,"`beq $8, $0, Odd`" is at line 12, and the label Odd is at line 18. This means we can subtract 18-13 (as the `$pc` has been updated), and the branch offset is 5. The translation to machine code of this instruction is 0x11000005.

The final branch instruction, "beq $0. $0, BeginForLoop" is at line 19, and the label BeginForLoop is at line 2. This means that we can subtract 2-20, which gives a branch offset of -18. Note that this branch is negative, so -18 must be a negative 2's complement, or 0xffffffee. The translation to machine code of this instruction is 0x0100ffee.

Chapter 7.8. 5 PC relative addressing

The type of addressing done with branch statements is called PC relative addressing. The reason for this name is that all branch addresses are calculated as an offset from the PC. This is contrasted with Jump (J) instructions, which branch to absolute addresses. So while a branch address must be calculated, a jump address is whatever is in the jump instruction. Both implement branches to different parts of the program, so why are there the two different formats?

The first reason is that a J instruction can access the entire .text segment of memory. To access the entire .text segment requires 26 bits to store the address. This leaves no room for registers which need to be compared, as in the I instruction. The branch instructions can do operations like compare registers, but is limited in that the address it contains only has 16 bits. This means that the branch instruction is limited in that it can only access addresses relatively local to the current `$pc`. So the basic difference is that the jump instruction can access any point in the text memory, but cannot be conditional. The branch instruction can be conditional, but cannot access all of the text memory.

PC relative addressing has another advantage. The compiler can generate the code for the branch at compile time, as it does not need to know the absolute addresses of the statements, only how far they are from the current `$pc`. This means that the code can easily be moved (or relocated) in the .text area and still work correctly. In the example of generating machine code for the branch instruction above, note that it really doesn't matter if the code fragment for printing odd/even numbers is at address 0x10010000, 0x10054560, or any other address. The branch is always relative to the $pc, so where the code exists is irrelevant to its correct execution.

However because the J instructions all branch to a fixed address, that address must be defined before the program begins to execute, and the absolute address cannot be changed (the code cannot be relocated).

So the difference between branches and jumps comes down to how they are used. Normally when compiling a program, any program control transfer inside of a file (if statements, loops, etc) is implemented using branch statements. Any program control transfer to a point outside of a file, which means a call to a subprogram, is normally implemented with a jump[23].

Chapter 7. 9 Exercises

1) When using a branch instruction, the lowest two bits in the offset to add to the `$pc` can be dropped.
   a) Why can these two bits be dropped?
   b) What must happen when the branch address is later calculated?
   c) Do you think the lowest two bits can be dropped in the absolute address for a jump instruction? Why or why not?

2) In section 7.8.3 of the textbook, it was said that a branch could access addresses that were -8191…8192 distance from the current `$pc`. However the 2's complement integer has values from -8192..8191. Why the discrepancy between the value of the 2's complement integer and the size of the branch?

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 are 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

[23] Note that to actually implement jumps is complex and requires a program called a *linkage editor*, or more simply a *linker*. How a linker works is beyond the scope of this text.

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) For the following program, translate all the branch statements (b, beqz, bnez, etc) to machine code, making sure you have correctly calculated the relative branch values.

```
.text
.global main
main:
    la $a0, prompt
    jal PromptInt
    move $s0, $v0

    BeginLoop:
        seq $t0, $s0, -1
        bnez $t0, EndLoop

        la $a0, result
        move $a1, $s0
        jal PrintInt

        slti $t0, $s0, 100
        beqz $t0, BigNumber
            la $a0, little
            jal PrintString
            b EndIf

        BigNumber:
            la $a0, big
            jal PrintString
        EndIf:

        la $a0, tryAgain
        jal PrintString

        la $a0 prompt
        jal PromptInt
        move $s0, $v0
        b BeginLoop

    EndLoop:
        jal Exit
.data
    prompt: .asciiz "\nEneter a number (-1 to end)"
    result: .asciiz "\nYou entered "
    big:    .asciiz "\nThat is a big number"
    little: .asciiz "\nThat is a little number"
    tryAgain: .asciiz "\nTry again"
.include "utils.asm"
```

8) Explain why program code that uses PC relative addresses is relocatable, but program code that uses absolute addresses is not relocatable.

9) What is the maximum distance from the PC which can be addressed using a branch statement? Give your answer as an address distance, and as a number of statements.

10) 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).

11) 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".

12) 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".

13) 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.

14) Using only sll and srl, 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.

What you will learn.

In this chapter you will learn:

1) The stack datastructure.
2) The purpose of a program stack, and how to implement it.
3) How to implement reentrant programs.
4) How to store local variables on the stack
5) What recursive subprograms are, and how to implement them.

Chapter 8 Reentrant Subprograms

In Chapter 5 the concept of a subprogram was introduced. At the time the *jal* operand was introduced as a way to call a subprogram, and the *jr $ra* instruction was introduced as the equivalent of a return statement. To implement subprograms which do not call other subprograms, this definition of how to call and return from a subprogram was sufficient. However this limitation on subprograms that they cannot be reentrant is far too restrictive to be of use for in real programs. This chapter will implement an infrastructure for subprogram dispatching that removes the non-reentrant problem from subprograms.[24]

Chapter 8. 1 Stacks

This section will cover the concept of a program stack. To do so, first the data structure of a stack will be implemented.

Chapter 8.1. 1 Stack data structure: definition

Many readers will come into the material in this chapter with either no understanding, or a poor understanding, of a stack data structure. It therefore makes sense to include a section on what a stack is, and how it works.

A stack is a Last-In-First-Out data structure. 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 try 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 a array index to the next available spot. A pop

[24] The subprogram infrastructure given here will have a limitation. Only 4 input parameters to the subprogram, and 2 return values from the subprogram, will be allowed. This limitation is used to simplify the concept of a stack. In many cases in MIPS assembly subprograms are implemented using a Frame Pointer (`$fp`) to keep track of parameters and return values. The following web site gives a good overview of how the `$fp` can be used in subprogram calls:
https://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventionsSummary.pdf

operation removes the top most item, so returns the item on top of stack, and subtracts 1 from the size of the stack.

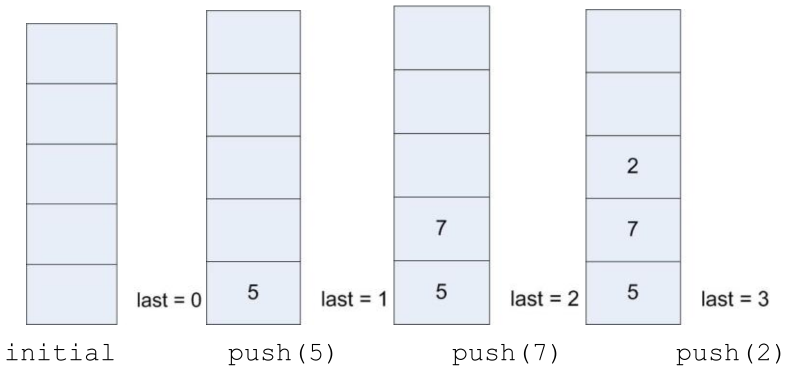
The following is an example of how a stack works. The example begins by creating a data structure for a stack, which is shown below. Note that there is no error checking.

```
class Stack
{
    int SIZE=100;
    int[SIZE] elements;
    int last = 0;
    push(int newElement)
    {
        elements[last] = newElement;
        last = last + 1
    }
    int pop()
    {
        last = last - 1;
        return element[last];
    }
}
```

Program 8-1: Stack class definition

To see how the array works, the following example illustrates the following operations:

```
push(5)
push(7)
push(2)
print(pop())
push(4)
print(pop())
print(pop())
```



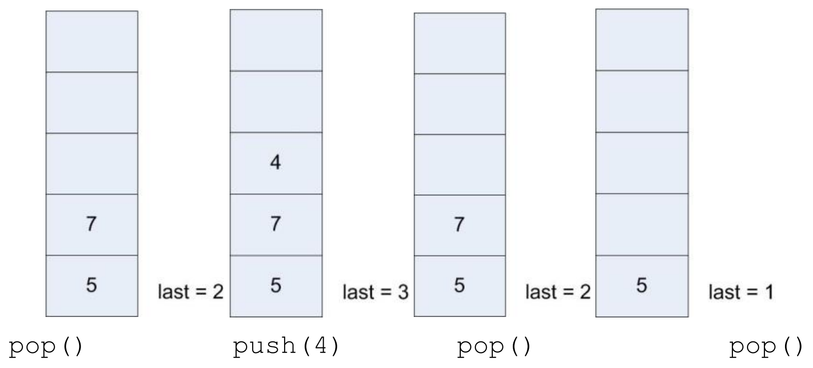


Figure 8-1: Push/Pop Example 1

The output of the program would be `2, 4, 7.`

Chapter 8.1. 2 Another stack implementation

The stack of tray metaphor used in the previous section has the same problem as all metaphors and analogies, that is you cannot reason about the underlying problem using a metaphor or analogy. Of all the logic errors that exist in the world, this is probably the most common, and the most harmful.

For example, there is another stack implementation that is better suited to our needs in describing the program stack. This stack implementation is still a LIFO data structure, but it will grow down in memory and the items on it will not be fixed sized. This is completely alien to our cafeteria tray metaphor, so this text will abandon that metaphor as quickly as it introduced it.

In this implementation, the stack will store groups of characters (strings) of varying sizes. The storage for the stack will use an array of size 7 of `elements`. Each character will be stored in an element. Because the stack grows downward in memory, the first item in the string will be allocated at `elements[size-1]`, and each time a new string is desired, it will be stored in the next lowest available place in memory. Because the strings are not all the same size, the size of each string must be stored in the array with the characters. This means that with each string a number must be stored which gives the size of the string.

In the above paragraph, it says that the strings are not all the same size, but it purposefully does not say that the strings are not all fixed size. The strings must have a known, fixed size when the push operation is executed. This is an important point which will have an impact when discussing the program stack.

The data structure for this new stringStack class is defined below.

```
class stringStack
{
    int SIZE=7;
    int elements[SIZE]; # Note that characters are stored as int
    int last = SIZE-1;
    push(String s) {
        last = (last - s.length())-1;
        elements[last] = s.length();
        int i = last + 1;
```

```
            for (char c in s)
            {
                 elements[i] = c;
                 i = i + 1;
            }
        }

        String pop()
        {
            int i = elements[last];
            int j = last + 1;
            last = last + i;
            for ( ; j < last; j++) {
                s = s + elements[j];
            }
            return s;
        }
    }
```

Program 8-2: String class stack definition

Note that in the StringStack class the size of the string is stored with the string on the stack, and the push and pop operations use the size to determine the amount of memory to be allocated/deallocated. The following illustrates this stack after the strings “ab” and “cde” have been pushed on the stack.

Chapter 8. 2 The program stack

This section will cover the reason why non-reentrant subprograms are a problem. A special type of stack, the program stack, will be introduced to resolve this problem. This stack will allow memory for a subprogram to be allocated when the subprogram is entered, and freed when the program is exited.

The chapter will first show why a stack is needed by showing why subprograms are non-reentrant. It will then show how to solve the problem using a stack. The sections after this one will continue to develop the concept of a program stack, and showing how it can be used to implement programming concepts like recursion.

Chapter 8.2. 1 The non-reentrant subprogram problem

The subprograms presented in Chapter 3 had a limitation that they could not call other subprograms. The problem is illustrated in the following example.

```
.text
.globl main
main:
    jal BadSubprogram

    la $a0, string3
    jal PrintString

    jal Exit
```

```
BadSubprogram:
   la $a0, string1
   jal PrintString

   li $v0, 4
   la $a0, string2
   syscall
   jr $ra

.data
string1: .asciiz "\nIn subprogram BadSubprogram\n"
string2: .asciiz "After call to PrintString\n"
string3: .asciiz "After call to BadSubprogram\n"
.include "utils.asm"
```

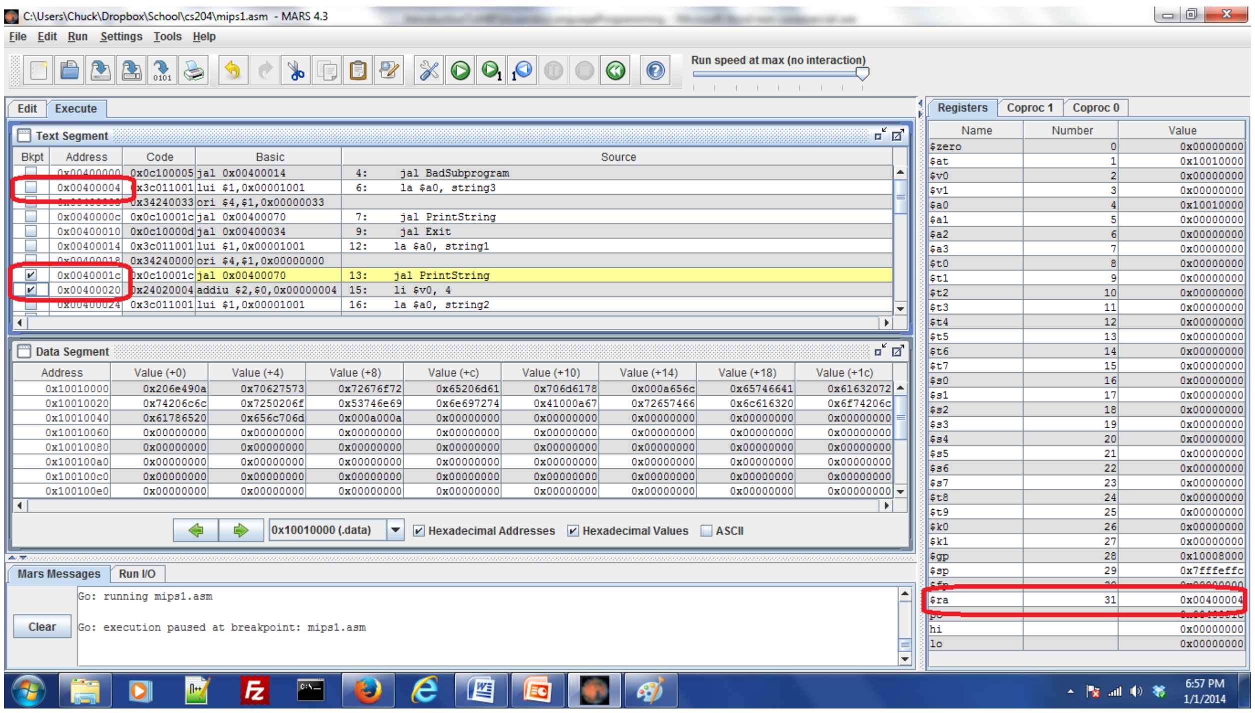
The programmer who wrote this appears to the `jal` and `jr` operators to act like subprogram call and return statements. Therefor the expected output is:

```
In subprogram BadSubprogram
After call to PrintString
After call to example
```

However when this program runs, what appears to be an infinite loop appears, and the output is:

```
In subprogram BadSubprogram
After call to PrintString
After call to PrintString
...
```

In the example subprogram *`BadSubprogram`* above, the subprogram is making a call to another subprogram, `PrintString`. To see what is happening, set 2 break points, one at the "`jal PrintString`" instruction, and one at the following statement “`li $v0, 4`”, and run the program. You should get a MARS screen that looks like the following image.


C:\Users\Chuck\Dropbox\School\cs204\mips1.asm - MARS 4.3

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x0c100005	jal 0x00400014	4: jal BadSubprogram
☐	0x00400004	0x3c011001	lui $1,0x00001001	6: la $a0, string3
☐	[illegible]	0x34240033	ori $4,$1,0x00000033	
☐	0x0040000c	0x0c10001c	jal 0x00400070	7: jal PrintString
☐	0x00400010	0x0c10000d	jal 0x00400034	9: jal Exit
☐	0x00400014	0x3c011001	lui $1,0x00001001	12: la $a0, string1
☐	0x00400018	0x34240000	ori $4,$1,0x00000000	
✓	0x0040001c	0x0c10001c	jal 0x00400070	13: jal PrintString
✓	0x00400020	0x24020004	addiu $2,$0,0x00000004	15: li $v0, 4
☐	0x00400024	0x3c011001	lui $1,0x00001001	16: la $a0, string2

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x206e490a	0x70627573	0x72676f72	0x65206d61	0x706d6178	0x000a656c	0x65746641	0x61632072
0x10010020	0x74206c6c	0x7250206f	0x53746e69	0x6e697274	0x41000a67	0x72657466	0x6c616320	0x6f74206c
0x10010040	0x61786520	0x656c706d	0x000a000a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) [✓ Hexadecimal Addresses] [✓ Hexadecimal Values] ☐ ASCII

Mars Messages

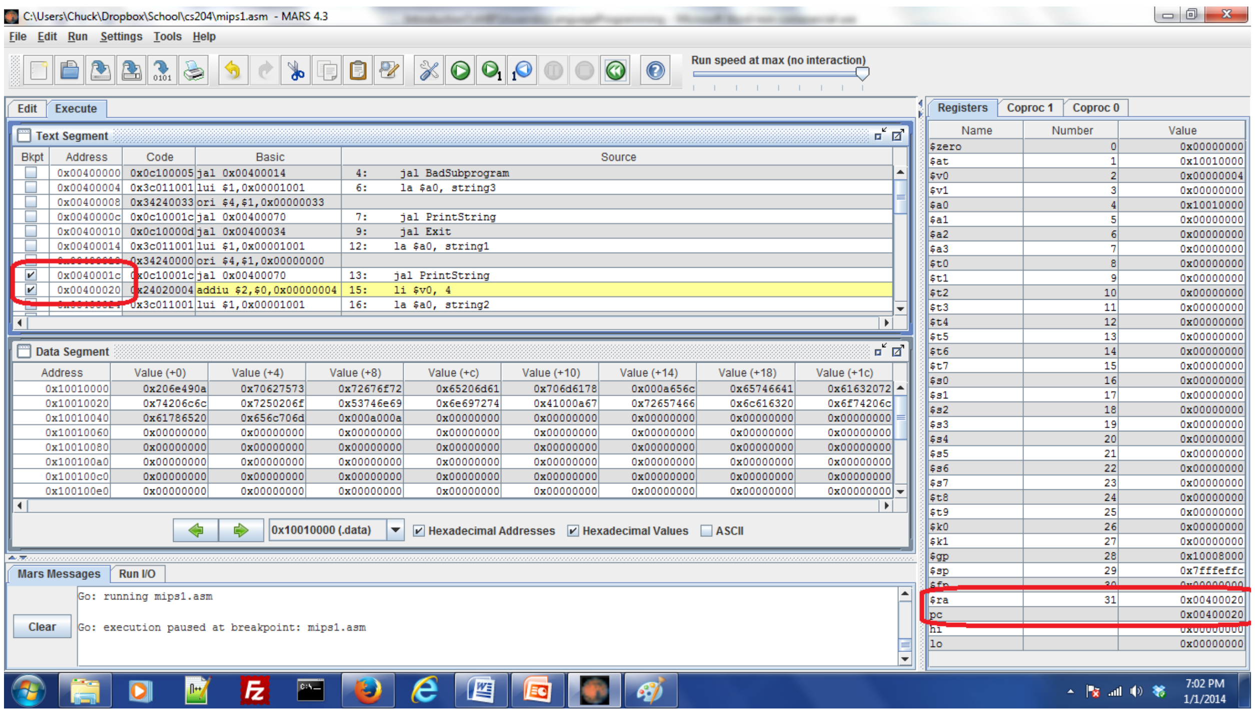
```
Go: running mips1.asm

Go: execution paused at breakpoint: mips1.asm
```

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400004
pc		0x0040001c
hi		0x00000000
lo		0x00000000

6:57 PM 1/1/2014

From this MARS screen image, note that before the call to PrintString the value of the $ra register is address 0x00400004. This is the address which was set when the subprogram BadProgram was called, and is the address the subprogram BadProgram should return to when it completes execution. Now click the green arrow key to continue running the program and it should stop at the statement after the "jal Subprogram" statement, as shown below.


C:\Users\Chuck\Dropbox\School\cs204\mips1.asm - MARS 4.3

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x0c100005	jal 0x00400014	4: jal BadSubprogram
☐	0x00400004	0x3c011001	lui $1,0x00001001	6: la $a0, string3
☐	0x00400008	0x34240033	ori $4,$1,0x00000033	
☐	0x0040000c	0x0c10001c	jal 0x00400070	7: jal PrintString
☐	0x00400010	0x0c10000d	jal 0x00400034	9: jal Exit
☐	0x00400014	0x3c011001	lui $1,0x00001001	12: la $a0, string1
☐	[illegible]	0x34240000	ori $4,$1,0x00000000	
✓	0x0040001c	0x0c10001c	jal 0x00400070	13: jal PrintString
✓	0x00400020	0x24020004	addiu $2,$0,0x00000004	15: li $v0, 4
☐	[illegible]	0x3c011001	lui $1,0x00001001	16: la $a0, string2

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x206e490a	0x70627573	0x72676f72	0x65206d61	0x706d6178	0x000a656c	0x65746641	0x61632072
0x10010020	0x74206c6c	0x7250206f	0x53746e69	0x6e697274	0x41000a67	0x72657466	0x6c616320	0x6f74206c
0x10010040	0x61786520	0x656c706d	0x000a000a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) [✓ Hexadecimal Addresses] [✓ Hexadecimal Values] ☐ ASCII

Mars Messages

```
Go: running mips1.asm

Go: execution paused at breakpoint: mips1.asm
```

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000004
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400020
pc		0x00400020
hi		0x00000000
lo		0x00000000

7:02 PM 1/1/2014

Note that now the `$ra` register points to the current statement address, 0x00400020. What has happened is that the `PrintString` subprogram needed to have a return address, and so when the "`jal PrintString`" instruction was executed, it wrote over the address in the `$ra` register. When this register was overwritten, the subprogram `BadSubprogram` lost its link back to the main subprogram. Now when "`jr $ra`" instruction runs to return to the main, the `$ra` is incorrect, and the program keeps going back to the same spot in the middle `BadSubprogram`. This is in fact an infinite loop, though it was achieved through a strange mechanism. So the `jal` and `jr` operators cannot be thought of as call and return statements. These two operators simply transfer control of the program, and a call and return mechanism is more complicated to implement than these simple operators in assembly.

Chapter 8.2. 2 Making subprograms re-entrant

About the only good thing about the `BadSubprogram` example is that it identifies the problem with the subprogram calling mechanisms is assembly, that the `$ra` needs to be stored when the subprogram is entered and restored just before the program leaves. But the problem with the `$ra` is also a problem with any registers that the program uses, as well as any variables that are defined in the subprogram. Space is needed in memory to store these variables.

The space to store variables and registered which need to be saved for a subprogram is called a stack. When the program begins to run, memory at a high address, in this case 0x7ffffe00, is allocated to store the stack. The stack then grows downward in memory. Generally the area allocated to the stack is sufficient for any properly executing program, though it is common for incorrect programs to reach the limit of the stack memory segment. If a properly running program reaches the limit of the stack memory segment, it can always allocate larger segments of memory.

When a subprogram is entered, it pushes (or allocates) space on the stack for any registers it needs to save, and any local variables it might need to store. When the subprogram is exited, it pops this memory off of the stack, freeing any memory that it might have allocated, and restoring the stack to the state it was in before the subprogram was called. The following program, using the subprogram Good Subprogram, highlights how the `$ra` register while the subprogram is running and then restored just before the it is used to return from the subprogram.

```
.text
.globl main
main:
    jal GoodSubprogram

    la $a0, string3
    jal PrintString

    jal Exit

GoodSubprogram:
   addi $sp, $sp, -4     # save space on the stack (push) for the $ra
   sw $ra, 0($sp)        # save $ra
   la $a0, string1
   jal PrintString
   li $v0, 4
   la $a0, string2
   syscall

   lw $ra, 0($sp)        # restore $ra
   addi $sp, $sp, 4      # return the space on the stack (pop)
   jr $ra

.data
string1: .asciiz "\nIn subprogram GoodExample\n"
string2: .asciiz "After call to PrintString\n"
string3: .asciiz "After call to GoodExample\n"
.include "utils.asm"
```

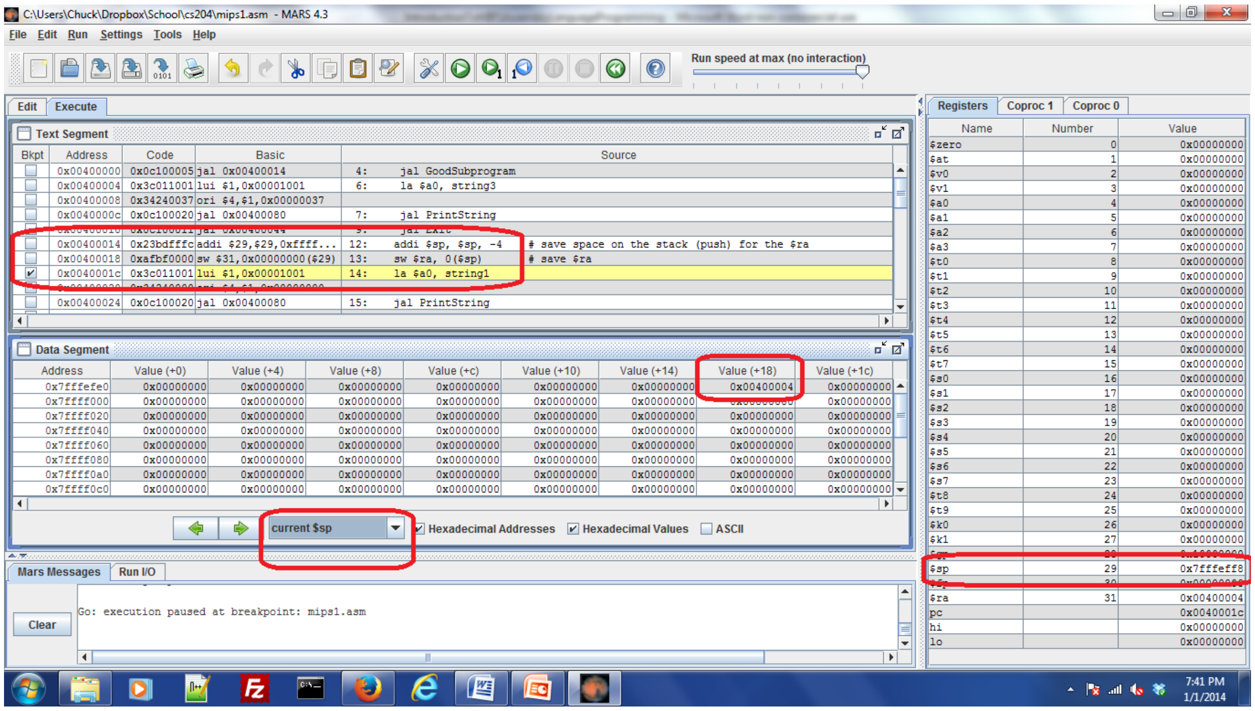
This program will work as expected, and the infinite loop is gone. To show why this program works, the highlighted in lines the program are explained. The first set of highlighted lines are the following.

```
addi $sp, $sp, -4       # save space on the stack (push) for the $ra
sw $ra, 0($sp)          # save $ra
```

This is an example of the type of code that should be placed at the start of a reentrant subprogram. When the subprogram is entered, the stack pointer (`$sp`) register points to the current end of the stack. All subprograms before this one have incremented the `$sp` to allocate space for their automatic variables, so all previous subprograms have stack frames (or activiation records) on the stack for their execution. So all space above the stack pointer is taken, but the space below the $sp is open, and this is where this subprogram allocates its space to place it variables.

Remember that the stack grows downward, which is why 4 is subtracted from `$sp` when the space is allocated. The allocation of 4 bytes is the amount needed to store the `$ra`.

When the GoodSubprogram is run, and execution stopped immediately after the `$ra` is written to the stack, the MARS screen would look as follows.


C:\Users\Chuck\Dropbox\School\cs204\mips1.asm - MARS 4.3
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Text Segment
Data Segment
Registers Coproc 1 Coproc 0
Mars Messages Run I/O
Clear
7:41 PM 1/1/2014

Bkpt	Address	Code	Basic	Source
	0x00400000	0x0c100005	jal 0x00400014	4: jal GoodSubprogram
	0x00400004	0x3c011001	lui $1,0x00001001	6: la $a0, string3
	0x00400008	0x34240037	ori $4,$1,0x00000037	
	0x0040000c	0x0c100020	jal 0x00400080	7: jal PrintString
	0x00400010	0x0c100011	jal 0x00400044	9: jal Exit
	0x00400014	0x23bdfffc	addi $29,$29,0xffff...	12: addi $sp, $sp, -4 # save space on the stack (push) for the $ra
	0x00400018	0xafbf0000	sw $31,0x00000000($29)	13: sw $ra, 0($sp) # save $ra
✓	0x0040001c	0x3c011001	lui $1,0x00001001	14: la $a0, string1
	0x00400020	0x34240000	ori $4,$1,0x00000000	
	0x00400024	0x0c100020	jal 0x00400080	15: jal PrintString

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00400004	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

current $sp [✓ Hexadecimal Addresses] [✓ Hexadecimal Values] ASCII

```
Go: execution paused at breakpoint: mips1.asm
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff8
$fp	30	0x00000000
$ra	31	0x00400004
pc		0x0040001c
hi		0x00000000
lo		0x00000000

Now the select box for the memory to view is not "data", but "current $sp", or stack. Looking at the value of $sp of 0x7ffefff8, it can be seen that the correct return address from GoodSubprogram has been saved.

The second set of highlighted code, shown below, is an example of the type of code which should be placed just before the last statement in a subprogram. In this code the $ra is restored to the value that it contained when the subprogram was called, so the subprogram can return to the correct line in the calling program. The stack frame for this subprogram is then *popped* by adding the amount of memory used back to the stack.

```
    lw $ra, 0($sp)          # restore $ra
    addi $sp, $sp, 4        # return the space on the stack (pop)
```

Note that when using a HLL compiler, the compiler will decide if the overhead of a stack is needed or not, and will handle any the mechanics of including code to handle the program stack. In assembly, this is up to the programmer. If the program does not need to share any data or transfer control outside of the subprogram, the allocation of the stack frame can be avoided.

Chapter 8. 3 Recursion

In computer science, recursion is a mechanism to a way to divide a problem up into successively smaller instances of the same problem until some stopping condition (or base case) is reached. The individual solutions to all of the smaller problems are then gathered together and an overall solution to the problem is obtained. This technique is very powerful for specifying unbounded problems which contain some common description of all parts of the problem. This is common with searching problems, such as trying to find all web pages which are reachable from a given web page. Since the number of connections to any page is unbounded, it is not possible to design a program using looping structures to find all of the pages, and recursion is a natural solution to this problem.

Unfortunately the types of problems that easily lend themselves to recursive solutions are more complex than can be covered in an introductory programming text such as this. Thus the example problems which are presented are more easily solved using other means like iteration. Many programmers studying recursion are left wondering why bother with the recursion, and recursion is seen as hard and not very useful.

Having said about recursion, this chapter will also present recursion using simple problems that can more easily be solved using iteration. Most of the problems at the end of the chapter will also fall into this category. It is difficult to present a true use of recursion without clouding the details of how recursion is implemented, or implementing more complex data structures that would require a complete and separate treatment to explain.

Chapter 8.3. 1 Recursive multiply in a HLL

To implement recursion, both the current state of the solution as well as the path that has lead to this state must be maintained. The current state allows the problem to be further subdivided as the recursion proceeds in a forward manner towards the base case. The path to the current state allows the results to be gathered together back together to achieve the results. This is a perfect situation for a stack.

An example is a recursive definition of a multiply operation. Multiplication can be defined as adding the multiplier (m) to itself the number of times in the multiplicand (n) times. Thus a recursive definition of multiplication is the following:

```
M(m,n) = m (when n = 1)
         else m + M(m, n-1)
```

This is implemented in pseudo code below.

```
subprogram global main()
{
    register int multiplicand
    register int multiplier
    register int answer
    m = prompt("Enter the multiplicand")
    n = prompt("Enter the multiplier")
    answer = Multiply(m, n)
    print("The answer is: " + answer)
}
subprogram int multiply(int m, int n)
{
    if (n == 1)
        return m;
    return m + multiply(m,n-1)
}
```

The following MIPS assembly language program implements the above pseudo code program. Note that at the start of each call to multiply the `$ra` is stored. In all cases but the first call to the subprogram the `$ra` will contain the same address. The stack records storing the `$ra` are really just a way to count how far into the stack the program has gone, so it can return the correct number of times.

There is one other piece of data that has been stored on the program stack and that is the value of `$a1`. As the programmer of this subprogram I know that `$a1` will not be changed in subsequent calls to multiply. However the agreement to return register values unchanged only applies to the save registers (`$s0`..`$s8`). It is perfectly valid for subprograms to change any other register values. So there is no guarantee that the value in `$a0` will not be changed once any subprogram is called. Prudence dictates that therefore it be saved, and the only save place to save it is on the stack.

```
.text
.globl main
main:
    # register conventions
    # $s0 - m
    # $s1 - n
    # $s2 - answer

    la $a0, prompt1            # Get the multiplicand
    jal PromptInt
    move $s0, $v0

    la $a0, prompt2            # Get the multiplier
    jal PromptInt
    move $s1, $v0

    move $a0, $s0
    move $a1, $s1

    jal Multiply         # Do multiplication
    move $s2, $v0

    la $a0, result             #Print the answer
    move $a1, $s2
    jal PrintInt

    jal Exit

Multiply:
    addi $sp, $sp -8           # push the stack
    sw $a0, 4($sp)               #save $a0
    sw $ra, 0($sp)             # Save the $ra

    seq $t0, $a1, $zero        # if (n == 0) return
    addi, $v0, $zero, 0        # set return value
    bnez $t0, Return

    addi $a1, $a1, -1          # set n = n-1
    jal Multiply               # recurse
    lw $a0, 4($sp)             # retrieve m
    add $v0, $a0, $v0          # return m+multiply(m, n-1)

    Return:
    lw $ra, 0($sp)               #pop the stack
    addi $sp, $sp, 8
    jr $ra

.data
prompt1: .asciiz "Enter the multiplicand: "
prompt2: .asciiz "Enter the multiplier: "
result:  .ascii  "The answer is: "
.include "utils.asm"
```

Program 8-3: Recursive multiplication

Chapter 8. 4 Exercises

1) Implement a subprogram which takes 4 numbers in the argument registers `$a0`...`$a3`, and returns the largest value and the average in `$v0` and `$v1` to the calling program. The program must be structured as follows:

```
Subprogram largestAndAverage($a1, $a2, $a3, $a4)
{
    int var0 = $a0, var1 = $a1, var2 = $a2, var3 = $a3;
    $s0 = getLarger($a1, $a2);
    $s0 = getLarger($s0, $a3);
    $v0 = getLarager(s0, $a4); // Largest is in $v0

    $v1 = (var0 + var1 + var2 + var3)/ 4; // Aversge is in $v1
    return;
}

Subprogram getLarger($a0, $a1) {
    $v0 = $a0
    if ($a1 > $a0)
        $v0 = $a1
    return;
}
```

Note the use of the variables `var0...var3`. Because the values of `$a0` and `$a1` (at least) are changed on the call to getLarger, they will not be available when they are needed to calculate the average, and must be stored on the stack. To do this problem correctly, you must calculate the maximum value using the getLarger subprogram shown here, and it must be called before the average is calculated. This implies that at a minimum `$a0` and `$a1` must be stored on the stack, though I would suggest all four be stack variables as shown here.

It is possible to create a solution which does not require the use of the stack variables, for example by simply calculating the average first. Such solutions do not answer the issue of how to handle variables that change using the stack, and are thus incorrect.

2) In the utils.asm file, fix the PrintInt subprogram so that it can call the PrintNewLine subprogram to print a new line character.

3) Implement a subprogram that prompt the user for 3 numbers, finds the median (middle value) of the 3, and returns that value to the calling program.

4) Implement a subprogram that prompts a user to enter values from 0..100 until a sentinel value of -1 is entered. Return the average of the numbers to the calling program.

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 function to calculate the summation of numbers from 1 to n. For example if the user enters 5, your program would add 1+2+3+4+5 and print out the answer 15.

7) Write a recursive program to calculate Fibonacci numbers. Use the definition of a Fibonacci number where F(n) = F(n-1) + F(n-2).

8) Write a recursive program to calculate factorial numbers. Use the definition of factorial as F(n) = n * F(n-1).

What you will learn.

In this chapter you will learn:

1) A third type of memory, heap memory, and how to allocate and use it.
2) The definition of an array, and how to implement and access elements in an array using assembly.
3) 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.
4) How to use array addresses to access and print elements in an array.
5) The Bubble Sort, and how to implement this sort in assembly.

Chapter 9 Arrays

In a HLL, an array is a multi-valued variable: a single array variable can contain many values. Arrays in MIPS assembly will be similar; however, the abstraction of an array is very much constrained in assembly. In MIPS assembly an array is implemented by storing multiple values in contiguous areas of memory, and accessing each value in the array as an offset of the array value. This chapter will cover how to implement and use arrays in MIPS assembly.

Chapter 9. 1 Heap dynamic memory

Before beginning a discussion of arrays, a third type of program data memory is introduced. So far in this text static (data) and stack dynamic memory have been discussed. The third type of memory introduced here is heap dynamic memory.

Arrays can exist in any type of memory, static, stack, or heap. However, when they are stored in static or stack memory, their size is fixed when the program is compiled and cannot change. Most programs that use arrays will need arrays that adjust in size depending on the size of the input data, and this requires heap memory.

Chapter 9.1. 1 What is heap memory

Heap memory was shown in Figure 2.3. As this figure shows, the heap memory segment begins in the program process space immediately after static memory, and grows upward in memory until theoretically it reaches the stack memory segment. In reality most systems limit the amount of heap a process can request to protect against incorrect programs that might be allocating heap inappropriately. Generally, these limits on heap size can be increased if they need to be larger than the default (for example, the Java interpreter can be run with the -Xms or -Xmx parameters to change the heap size).

Heap memory is dynamic, like stack memory, in that it is allocated at run time. However unlike stack memory which is automatically allocated and de-allocated when subprograms are entered and exited, heap memory is allocated based on a request from the programmer, normally using a *new* operator. The new operator allocates the requested amount of memory, and initializes it to a default value (normally zero). Because heap memory is allocated when requested by the user at run time, and the amount of memory can be determined at run time.

Memory in the heap is generally de-allocated when it is no longer needed by the programmer. How this de-allocation is done is dependent on the environment in which the program is being run. For example in C/C++ memory is de-allocated using either the free() function call or the delete operator. Java, C#, and other modern languages use memory managers which automatically free the memory when it is no longer used.

The biggest issue with heap memory is since it is allocated and de-allocated in non-fixed sized pieces, as the memory is de-allocated it leaves numerous holes in the memory that are of various sizes. This makes memory allocation and de-allocation difficult to deal with. Heap memory is much more complicated to manage than static and stack memory, and entire books have been written on heap management.

Because of the complexity of dealing with managing the de-allocation of heap memory, this text will only deal with the allocation of the memory. Any heap memory that is allocated cannot be reused.

Chapter 9.1. 2 Allocating heap memory example – PromptString subprogram

The first example subprogram presented here for using heap memory is a function that allows a programmer to prompt for strings without having to create a blank string variable in the `.data` section of the program. The subprogram first allocates a string variable large enough to hold the string the user is being prompted for, and then uses the syscall service 8 to read a value into that string.

```
.text
main:

    la $a0, prompt1     # Read and print first string
    li $a1, 80
    jal PromptString
    move $a0, $v0
    jal PrintString

    la $a0, prompt2     # Read and print first string
    li $a1, 80
    jal PromptString
    move $a0, $v0
    jal PrintString

    jal Exit

.data
    prompt1: .asciiz "Enter the first string: "
    prompt2: .asciiz "Enter the second string: "

.text
# Subprogram:     PromptString
# Author:   Charles Kann
# Purpose:  To prompt for a string, allocate the string
#           and return the string to the calling subprogram.
# Input:    $a0 - The prompt
#           $a1 - The maximum size of the string
# Output:   $v0 - The address of the user entered string

PromptString:
    addi $sp, $sp, -12  # Push stack
    sw $ra, 0($sp)
    sw $a1, 4($sp)
    sw $s0, 8($sp)

    li $v0, 4           # Print the prompt
    syscall             # in the function, so we know $a0 still has
                        # the pointer to the prompt.

    li $v0, 9           # Allocate memory
    lw $a0, 4($sp)
    syscall
    move $s0, $v0

    move $a0, $s0       # Read the string
    li $v0, 8
    lw $a1, 4($sp)
    syscall

    move $v0, $s0       # Save string address to return

    lw $ra, 0($sp)      # Pop stack
    lw $s0, 8($sp)
    addi $sp, $sp, 12
    jr $ra

.include "utils.asm"
```

Program 9-1: PromptString subprogram showing heap allocation

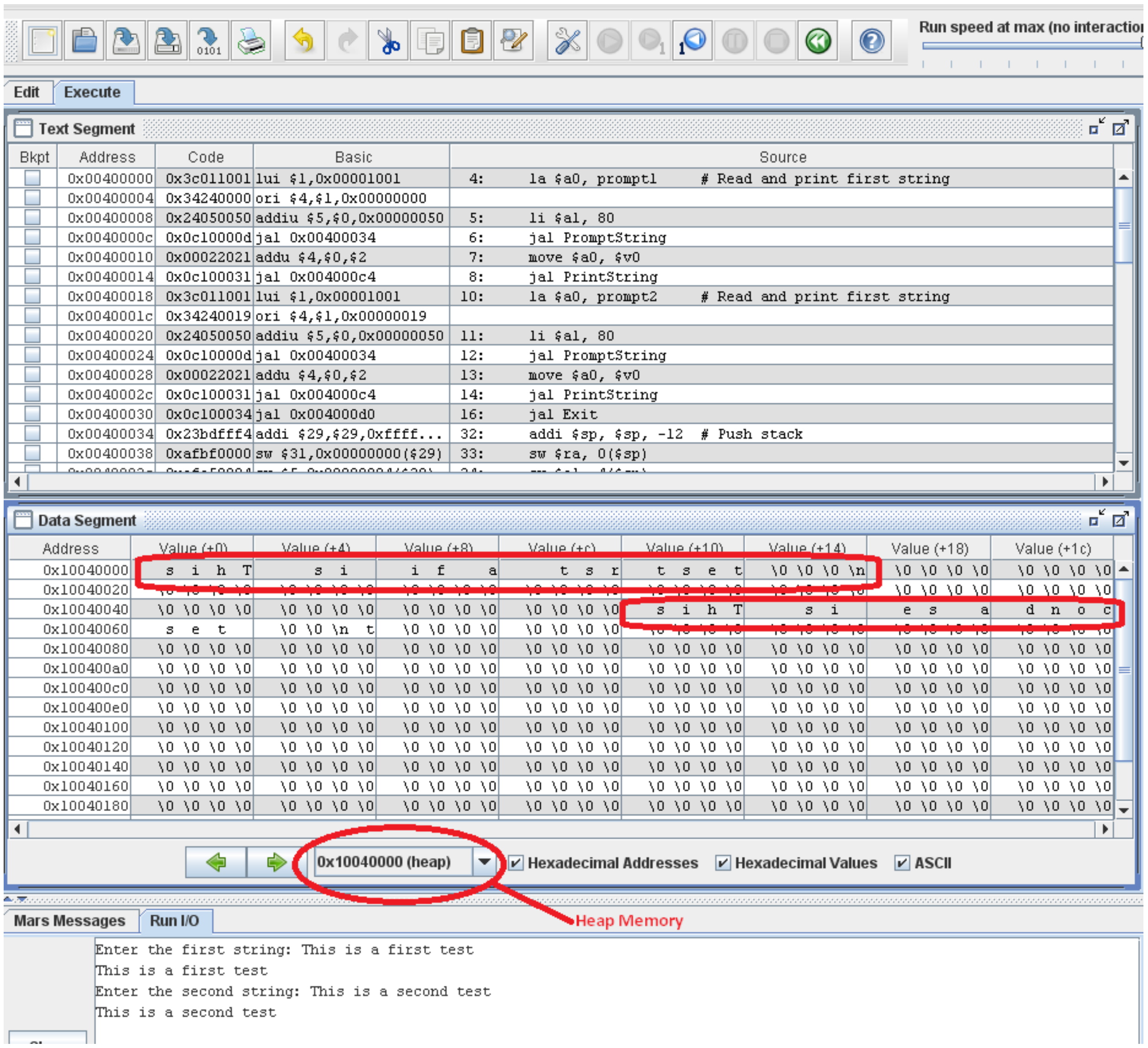
Chapter 9.1. 3 Commentary on PromptString Subprogram

1) To allocate heap memory, the syscall service 9 is used. The address of the memory returned from this heap allocation syscall is in `$v0`. `$v0` is moved to `$a0` to be used in the syscall service 8 to read a string. The address of the memory containing the string is now in `$a0`, and is moved to `$v0` to be returned to the main subprogram.

2) Data which is expect to be unchanged across subprogram calls (including syscall) should always be stored in a save register ($s0 in this example), or on the stack (`$a1` in this example). Do not use any other registers (such as temporary registers like `$t0`) or memory as the values of cannot be guaranteed across subprogram calls.

3) The value of `$s0` is saved when this subprogram is entered, and restored to its original value when the subprogram is left. All save registers must have the same value when leaving a subprogram as when it is entered. Any other registers (`$t0, $a0, $v0`, etc.) can be used with disregard to restoring their original values when the sub program exits.

4) The main subprogram in this example will shows two strings being read. This is to show how the allocated strings exist in heap memory. In the MARS screen shot below, note that the heap memory is now being displayed. In the heap memory the two strings entered ("This is a first test", and "This is a second test") are shown, with each taking up 80 bytes of memory.



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	4: la $a0, prompt1 # Read and print first string
	0x00400004	0x34240000	ori $4,$1,0x00000000	
	0x00400008	0x24050050	addiu $5,$0,0x00000050	5: li $a1, 80
	0x0040000c	0x0c10000d	jal 0x00400034	6: jal PromptString
	0x00400010	0x00022021	addu $4,$0,$2	7: move $a0, $v0
	0x00400014	0x0c100031	jal 0x004000c4	8: jal PrintString
	0x00400018	0x3c011001	lui $1,0x00001001	10: la $a0, prompt2 # Read and print first string
	0x0040001c	0x34240019	ori $4,$1,0x00000019	
	0x00400020	0x24050050	addiu $5,$0,0x00000050	11: li $a1, 80
	0x00400024	0x0c10000d	jal 0x00400034	12: jal PromptString
	0x00400028	0x00022021	addu $4,$0,$2	13: move $a0, $v0
	0x0040002c	0x0c100031	jal 0x004000c4	14: jal PrintString
	0x00400030	0x0c100034	jal 0x004000d0	16: jal Exit
	0x00400034	0x23bdfff4	addi $29,$29,0xffff...	32: addi $sp, $sp, -12 # Push stack
	0x00400038	0xafbf0000	sw $31,0x00000000($29)	33: sw $ra, 0($sp)

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10040000	s i h T	s i	i f a	t s r	t s e t	\0 \0 \0 \n	\0 \0 \0 \0	\0 \0 \0 \0
0x10040020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	s i h T	s i	e s a	d n o c
0x10040060	s e t	\0 \0 \n t	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400e0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040100	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040120	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040140	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040160	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040180	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10040000 (heap) [✓] Hexadecimal Addresses [✓] Hexadecimal Values [✓] ASCII

```
Enter the first string: This is a first test
This is a first test
Enter the second string: This is a second test
This is a second test
```

Figure 9-1: Heap memory example

Chapter 9. 2 Array Definition and creation in Assembly

Most readers of this text will be familiar with the concept of arrays, and using them in a HLL. So this chapter will not cover their use, but how arrays are implemented and elements in the array accessed in assembly. Most HLL go to great pains to hide these details from the programmer, with good reason. When programmers actually deal with the details they often make mistakes that have serious consequences to the correctness of their programs: mistakes that lead to serious correctness problems with their programs, and bugs that can often lead to very difficult locate and fix.

But even though the details of arrays are hidden in most HLL, the details affect how HLL implement array abstractions, and the proper understanding of arrays can help prevent programmers from developing inappropriate metaphors that lead to program issues. Misusing object slicing in C++ or allocating and attempting to use arrays of null objects in Java are issues that can arise if a programmer does not understand true nature of an array.

The following definition of an array will be used in this chapter. **An array is a multivalued variable stored in a contiguous area of memory that contains elements that are all the same size**. Some programmers will find that this definition does not fit the definition of arrays in the HLL language which they use. This is a result of the HLL adding layers of abstraction, such as Perl associative array (which are really hash tables) or Java object arrays or ArrayList. These HLL arrays are always hiding some abstraction, and knowing what an array actually is can help with the understanding of how the HLL is manipulating the array.

The definition of an array becomes apparent when the mechanics of accessing elements in an array is explained. The minimum data needed to define an array consists of a variable which contains the address of the start of the array, the size of each element, and the space to store the elements. For example, an array based at address 0x10010044 and containing 5 32-bit integers is shown in Figure 9-2.

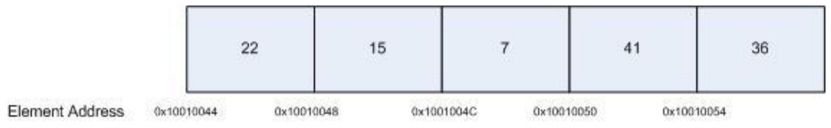


Figure 9-2: Array implementation

To access any element in the array, the element address is calculated by the following formula, and the element valued is loaded from that address.

```
    elemAddress = basePtr + index * size
```

where

- `elemAddress` is the address of (or pointer to) the element to be used.
- `basePtr` is the address of the array variable
- `index` is the index for the element (using 0 based arrays)
- `size` is the size of each element

So to load the element at index 0, the elemAddress is just (0x10010044 + (0 * 4)) = 0x10010044, or the basePtr for the array[25]. Likewise to load element the element at index 2, the elemAddress is (0x10010044 + (2 * 4)) = 0x1001004C.

[25] This calculation of the array address will make it apparent to many readers why arrays in many languages are zero based (the first element is 0), rather than the more intuitive concept of arrays being 1 based (the first element is 1). When thought of in terms of array addressing, the first element in the array is at the base address for the array

Two array examples follow. The first creates an array named grades, which will store 10 elements each 4 bytes big aligned on word boundaries. The second creates an array named id of 10 bytes. Note that no alignment is specified, so the bytes can cross word boundaries.

```
.data.
.align 2
grades: .space 40
id: .space 10
```

To access a grade element in the array grades, grade 0 would be at the basePtr, grade 1 would be at basePtr+4, grade 2 would be at basePtr + 8, etc. The following code fragment shows how grade 2 could be accessed in MIPS assembly code:

```
    addi $t0, 2             # set element number 2
    sll $t0, $t0, 2         # multiply $t0 by 4 (size) to get the offset
    la $t1, basePtr         # $t1 is the base of the array
    add $t0, $t0, $t1       # basePtr + (index * size)
    lw $t2, 0($t0)          # load element 2 into $t2
```

Addressing of arrays is not complicated, but it does require that the programmer keep in mind what is an address verses a value, and to know calculate an array offset.

Chapter 9.2. 1 Allocating arrays in memory

In some languages, such as Java, arrays can only be allocated on the heap. Others, such as C/C++ or C#, allow arrays of some types to be allocated anywhere in memory. In MIPS assembly, arrays can be allocated in any part of memory. However remember that arrays allocated in the static data region or on the heap must be fixed size, with the size fixed at assembly time. Only heap allocated arrays can have their size set at run time.

To allocate an array in static data, a label is defined to give the base address of the array, and enough space for the array elements is allocated. Note also that the array must take into account any alignment consideration (e.g. words must fall on word boundaries). The following code fragment allocates an array of 10 integer words in the data segment.

```
    .data
      .align 2
      array: .space 40
```

To allocate an array on the stack, the `$sp` is adjusted so as to allow space on the stack for the array. In the case of the stack there is no equivalent to the .align 2 assembler directive, so the programmer is responsible for making sure any stack memory is properly aligned. The following code fragment allocates an array of 10 integer words on the stack after the `$ra` register.

```
    addi $sp, $sp, -44
    sw $ra, 0(sp)
    # array begins at 4($sp)
```

(`basePtr + 0`), and so the number of the elements has more to do with how arrays are implemented, than in semantic considerations of what the elements numbers mean.

Finally to allocate an array on the heap, the number of items to allocate is multiplied by the size of each element to obtain the amount of memory to allocate. A subprogram to do this, called AllocateArray, is shown below.

```
# Subprogram:     AllocateArray
# Purpose:    To allocate an array of $a0 items,
#          each of size $a1.
      # Author:   Charles Kann
      # Input:    $a0 - the number of items in the array
      #           $a1 - the size of each item
      # Output:   $v0 - Address of the array allocated

      AllocateArray:
          addi $sp, $sp, -4
          sw $ra, 0($sp)

          mul $a0, $a0, $a1
          li $v0, 9
          syscall

          lw $ra, 0($sp)
          addi $sp, $sp, 4
          jr $ra
```

Program 9-2: AllocateArray subprogram

Chapter 9. 3 Printing an Array

This first program presented here shows how to access arrays by creating a PrintIntArray subprogram that prints the elements in an integer array. Two variables are passed into the subprogram, `$a0` which is the base address of the array, and `$a1`, which is the number of elements to print. The subprogram processes the array in a counter loop, and prints out each element followed by a ",". The pseudo code for this subprogram follows.

```
Subprogram PrintIntArray(array, size)
{
    print("[")
    for (int i = 0; i < size; i++)
    {
        print("," + array[i])
    }
    print("]")
}
```

The following is the subprogram in assembly, along with a test main program to show how to use it.

```
.text
.globl main
main:
    la $a0, array_base
    lw $a1, array_size
    jal PrintIntArray
    jal Exit
```

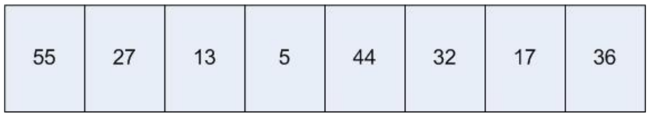
```
.data
    array_size: .word 5
    array_base:
            .word 12
            .word 7
            .word 3
            .word 5
            .word 11

.text
# Subprogram: PrintIntArray
# Purpose: print an array of ints
# inputs: $a0 - the base address of the array
#         $a1 - the size of the array
#
PrintIntArray:
    addi $sp, $sp, -16          # Stack record
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)

    move $s0, $a0               # save the base of the array to $s0

    # initialization for counter loop
    # $s1 is the ending index of the loop
    # $s2 is the loop counter
    move $s1, $a1
    move $s2, $zero

   la $a0 open_bracket          # print open bracket
   jal PrintString

loop:
    # check ending condition
    sge $t0, $s2, $s1
    bnez $t0, end_loop

        sll $t0, $s2, 2         # Multiply the loop counter by
                                # by 4 to get offset (each element
                                # is 4 big).
        add $t0, $t0, $s0       # address of next array element
        lw $a1, 0($t0)          # Next array element
        la $a0, comma
        jal PrintInt            # print the integer from array

        addi $s2, $s2, 1        #increment $s0
        b loop
end_loop:

    li $v0, 4                   # print close bracket
    la $a0, close_bracket
    syscall


    lw $ra, 0($sp)
    lw $s0, 4($sp)
```

```
        lw $s1, 8($sp)
        lw $s2, 12($sp)            # restore stack and return
        addi $sp, $sp, 16
        jr $ra

    .data
        open_bracket:        .asciiz "["
        close_bracket:       .asciiz "]"
        comma:         .asciiz ","
    .include "utils.asm"
```

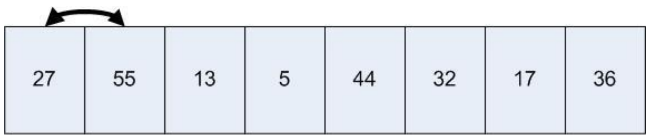
Program 9-3: Printing an array of integers

Chapter 9. 4 Bubble Sort

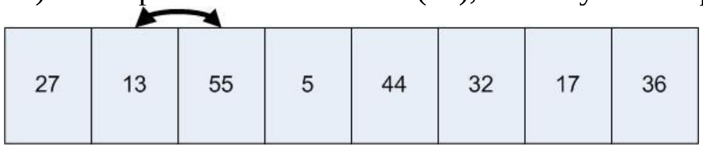
Sorting is the process of arranging data in an ascending or descending order. This example will introduce an algorithm, the Bubble Sort, for sorting integer data in a array. Consider for example the following array containing integer values.



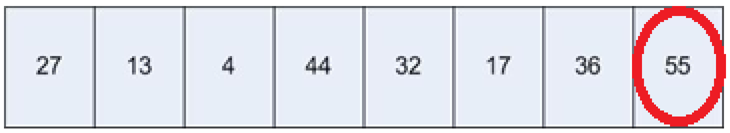
The sort is carried out in two loops. The inner loop passes once through the data comparing elements in the array and swapping them if they are not in the correct order. For example, element 0 (55) is compared to element 1 (27), and they are swapped since 55 > 27.



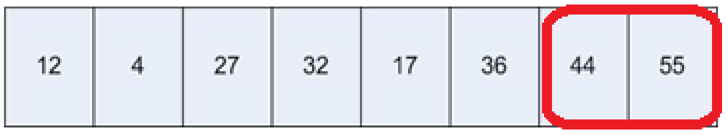
Next element 1 (now 55) is compared with element 2 (13), and they are swapped since 55 > 13.



This process continues until a complete pass has been made through the array. At the end of the inner loop the largest value of the array is at the end of the array, and in its correct position. The array would look as follows.



An outer loop now runs which repeats the inner loop, and the second largest value moves to the correct position, as shown below.



Repeating this outer loop for all elements results in the array being sorted in ascending order.

Pseudo code for this algorithm follws.

```
for (int i = 0; i < size-1; i++)
{
    for (int j = 0; j < ((size-1)-i); j++)
    {
         if (data[j] > data[j+1])
         {
             swap(data, j, j+1)
         }
    }
}

swap(data, i, j)
    int tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
}
```

Chapter 9.3. 1 Bubble Sort in MIPS assembly

The following assembly program implements the Bubble Sort matching the pseudo code algorithm in the previous section.

```
.text
.globl main
main:
    la $a0, array_base
    lw $a1, array_size
    jal PrintIntArray

    la $a0, array_base
    lw $a1, array_size
    jal BubbleSort

    jal PrintNewLine
    la $a0, array_base
    lw $a1, array_size
    jal PrintIntArray

    jal Exit
.data
    array_size: .word 8
    array_base:
            .word 55
            .word 27
            .word 13
            .word 5
            .word 44
            .word 32
            .word 17
            .word 36
.text
# Subproram:      Bubble Sort
# Purpose:        Sort data using a Bubble Sort algorithm
# Input Params:   $a0 - array
#           $a1 - array size
# Register conventions:
#           $s0 - array base
#           $s1 - array size
#           $s2 - outer loop counter
#           $s3 - inner loop counter
BubbleSort:
    addi $sp, $sp -20   # save stack information
    sw $ra, 0($sp)
    sw $s0, 4($sp)      # need to keep and restore save registers
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    sw $s3, 16($sp)

    move $s0, $a0
    move $s1, $a1

    addi $s2, $zero, 0  #outer loop counter
    OuterLoop:
        addi $t1, $s1, -1
        slt $t0, $s2, $t1
        beqz $t0, EndOuterLoop

        addi $s3, $zero, 0 #inner loop counter
        InnerLoop:
            addi $t1, $s1, -1
            sub $t1, $t1, $s2
            slt $t0, $s3, $t1
            beqz $t0, EndInnerLoop

            sll $t4, $s3, 2   # load data[j].  Note offset is 4 bytes
            add $t5, $s0, $t4
            lw $t2, 0($t5)

            addi $t6, $t5, 4  # load data[j+1]
            lw $t3, 0($t6)

            sgt $t0, $t2, $t3
            beqz $t0, NotGreater
                move $a0, $s0
                move $a1, $s3
                addi $t0, $s3, 1
                move $a2, $t0
                jal Swap      # t5 is &data[j], t6 is &data[j=1]

            NotGreater:
            addi $s3, $s3, 1
            b InnerLoop
        EndInnerLoop:
```

```
        addi $s2, $s2, 1
        b OuterLoop
    EndOuterLoop:

    lw $ra, 0($sp)      #restore stack information
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)
    addi $sp, $sp 20
    jr $ra

# Subprogram:           swap
# Purpose:        to swap values in an array of integers
# Input parameters:     $a0 - the array containing elements to swap
#                 $a1 - index of element 1
#                 $a2 - index of elelemnt 2
# Side Effects:         Array is changed to swap element 1 and 2
Swap:
    sll $t0, $a1, 2     # calcualate address of element 1
    add $t0, $a0, $t0
    sll $t1, $a2, 2     # calculate address of element 2
    add $t1, $a0, $t1

    lw $t2, 0($t0)      #swap elements
    lw $t3, 0($t1)
    sw $t2, 0($t1)
    sw $t3, 0($t0)

    jr $ra

# Subprogram: PrintIntArray
# Purpose: print an array of ints
# inputs: $a0 - the base address of the array
#         $a1 - the size of the array
#
PrintIntArray:
    addi $sp, $sp, -16        # Stack record
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)

    move $s0, $a0             # save the base of the array to $s0

    # initialization for counter loop
    # $s1 is the ending index of the loop
    # $s2 is the loop counter
    move $s1, $a1
    move $s2, $zero

   la $a0 open_bracket        # print open bracket
   jal PrintString

loop:
    # check ending condition
    sge $t0, $s2, $s1
    bnez $t0, end_loop

        sll $t0, $s2, 2       # Multiply the loop counter by
                              # by 4 to get offset (each element
                              # is 4 big).
        add $t0, $t0, $s0     # address of next array element
        lw $a1, 0($t0)        # Next array element
        la $a0, comma
        jal PrintInt          # print the integer from array

        addi $s2, $s2, 1      #increment $s0
        b loop
end_loop:

    li $v0, 4                 # print close bracket
    la $a0, close_bracket
    syscall


    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)           # restore stack and return
    addi $sp, $sp, 16
    jr $ra

.data
    open_bracket:       .asciiz "["
    close_bracket:      .asciiz "]"
    comma:        .asciiz ","
.include "utils.asm"
```

Program 9-4: Bubble Sort

Chapter 9. 5 Summary

In this chapter an array was defined as a multivalued variable stored in a contiguous area of memory that contains elements that are all the same size. The chapter then showed why each point in this definition is important, and how this definition can be used to implement an array and access array elements. The implementation and access to the array was shown in a number of programs, such as printing the array and sorting the array.

In terms of why this understanding of the true nature of an array is important, most HLL implement extensions to the basic array type, and a programmer who does not understand these extensions in the language is likely to have situations arise where bugs are encountered that are poorly understood. Even concepts as simple as Object arrays in Java are strange because the initial value of all elements is set to null. This is often confusing to new students until it is realized that in Java the size of an Object is unknown until it is allocated, so the only thing which can be used as the actual element in the Object array is the reference.

Chapter 9. 6 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 ≤ n ≥ 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 AllocateArray subprogram is incorrect in that the allocation can fall on any boundary. This is a problem if the array is of elements that must fall on a specific boundary. For example, if the array is contains ints, the array allocation must fall on full word boundary.

   a) Using the PromptString and AllocateArray subprograms, show how this problem can occur.

   b) Change the AllocateArray program to always do allocations on a double word boundary.

4) 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)
}
```

5) Change the sort in Program 9.2 to use a Selection Sort instead of a Bubble Sort.

6) Implement Program 9.2 so that the user is prompted for the maximum size of the array, and then fill the array with random numbers. Sort the array using any sort you choose. This will require the array be allocated in heap memory. Print out the array.

7) Implement a Binary Search algorithm, and using the results from Exercise 5, show how long the Binary Search takes (on average) for arrays of size 10, 100, and 1000. (You do not have to print out the values in the array).