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[1]. 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. Memory that exists in the hardware of a computer is quite complex and it is sliced and diced and used by many different sub-systems. This is a result of how hardware 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.
All 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;

[1] 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.

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 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.

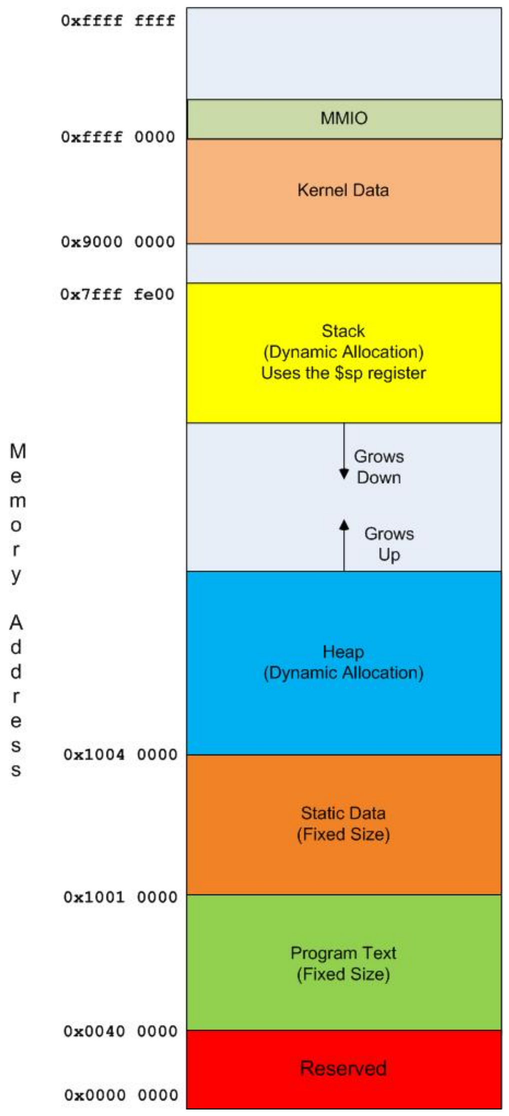


Figure 6-1: MIPS memory configuration

The bytes *in MIPS* are organized in groups of:

1) a single byte (8-bit)
2) a group of 2 bytes (16-bit), called a half-word
3) a group of 4 bytes (32-bit), called a word[2]
4) a group of 8 bytes (64-bit), called a double word.

[2] 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.

These groupings are not random in memory. All groupings start at 0x00000000.

Half-word boundaries:	0x00000000, 0x00000002, 0x00000004, 0x00000008, etc.
Words boundaries:	0x00000000, 0x00000004, 0x00000008, 0x0000000c, etc.
Double word boundaries:	0x00000000, 0x00000008, 0x00000010, 0x00000018, etc.

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 0x00000007, as it is not on a 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[3].

Chapter 6. 2 Static data

Static data[4] is data that is defined when the program is assembled and allocated when the program starts to run. Since 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 0x10010000 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 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 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
```

[3] 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.

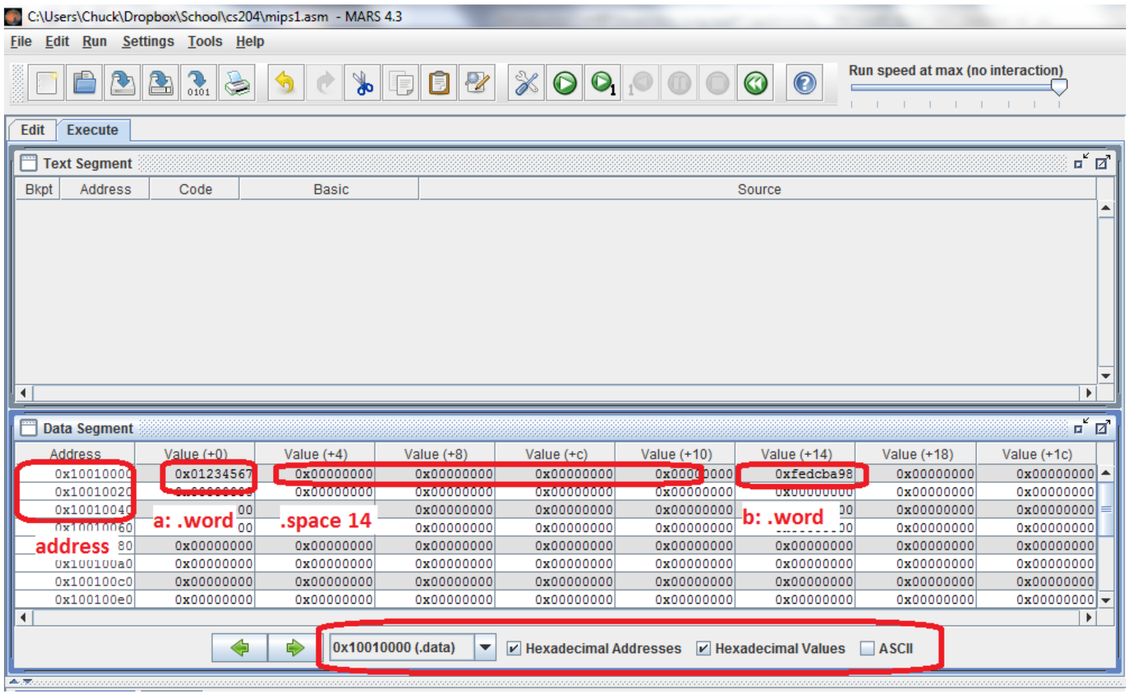
[4] Static is not used here in the same way as it is used in Java or C#. In these languages, static means *classwide*, or one per class. Static variables are associated with the class, not an instance of the class, and static methods cannot access instance variables. The original meaning of static came 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.

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 uses 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
a: .word
.space 14
b: .word
address

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	[illegible]	[illegible]	0x00000000	0x00000000	0x00000000	[illegible]	0x00000000	0x00000000
0x10010060	[illegible]	[illegible]	0x00000000	0x00000000	0x00000000	[illegible]	0x00000000	0x00000000
[illegible]	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 several options for showing the values of the addresses and memory. The first check box, labeled "`Hexadecimal Addresses`", allows memory addresses to be shown in hex or decimal. The checkbox labeled "`Hexadecimal Values`" allows all memory and register values on 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 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 a 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 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 operator. These operators include loading/storing a:

byte (`lb, sb`)
half word (`lh, sh`)
word (`lw, sw`)
double word (`ld, sd`)[5]
In this chapter only words will be considered, so only the `lw` and `sw` will be documented.

[5] 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 shorthand for taking the address of a label and putting it in a register. It does not load memory values into a register.

- `lw` operator, which has several pseudo-operator formats, but only two formats 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]
example:     lw $t1, 0($t0)  <-- copies the value at the address in $t0 into $t1
```

  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 segment
```

- `sw` operator, which has a number of pseudo-operator formats, but only two formats 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 at [R_s + Immediate].

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

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 several 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[6] (or just indirect).

[6] 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.

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 pseudocode 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 pseudocode

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 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 pseudocode (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[7]) 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.

[7] 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.

```
.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"
```

Chapter 6. 5 Exercises

1) Why do "`la label`" instructions always need to be translated into 2 lines of pseudocode? 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? (hint: there is a typo somewhere)

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

    li $v0, 10
    syscall

.data
    .word 8
```