

CS2810 Lecture Notes – Set 1

Chapters 2-3 of text
Instruction Sets/MIPS Architecture

ISA Types

- Generally, a machine's architecture is classified in terms of its instruction set architecture.
 - How many operands are needed 0, 1, 2, 3, ..
 - What is the hardware environment in which we are operating?
 - e.g. how many general purpose registers?
 - What types of instructions are necessary?

Design Principles

- In designing a computer, the ISA has a heavy influence on what the architecture of the machine will look like.
 - Often, the instruction set is defined and then the machine designed
 - Because of legacy code, one must often design the machine so that an older ISA also executes on the machine
 - Sometimes this is done through emulation

Design Principles

- ***Simplicity favors regularity***
- The more regular the environment, the simpler the design
- e.g. having several different types of add's means there is little regularity in the design, and thus the implementation will be less simple

Design Principles

- ***Smaller is faster***
- More hardware may require signals to travel further => slower
- E.g. more complex instruction set, more registers, ...

Assembly Language Programming

- Our goal over the next several slides is to build enough of our assembly language – called MIPS, to write the following simple assembly language program (ALP)
- Write an ALP to ask the user to input two integers and then print out their sum

Assembler vs Compiler

- We have all used compilers
 - Translates a program in a high level language into machine language
- An assembler is similar, only it translates a program in assembly language into machine language
- Assembly language is close to machine language
- Generally a machine has only one assembly language

Question

- Which, compiler or assembler, uses a more abstract language.

MIPS, SPIM, MARS

- MIPS – Name of the processor
 - There are many versions of this processor.
 - It is widely used in embedded applications
 - It is a RISC processor
 - One 32 bit word/machine language instruction
- SPIM – Name of the emulator for MIPS
 - Runs in most any environment
 - What is an emulator?

MIPS, SPIM, MARS

- MARS
 - More recent MIPS Emulator
 - Does everything SPIM does and a little more
 - <http://courses.missouristate.edu/KenVollmar/MARS/>
 - On the class web site is the MIPS Instruction Reference Guide – download it especially if you don't have the text.
 - OR – You can use the help in Mars

MIPS

- We will begin by describing the MIPS architecture and then as we go along we will describe individual instruction types.
- MIPS is a load/store machine
 - Meaning: Arithmetic can only be done in registers, e.g. you cannot directly add two numbers in memory together. You must first load them into registers and then add.

MIPS

- 32 Registers
 - \$0 through \$31
 - Mostly general purpose
 - 32 bits each
 - Can combine pairs 0,1; 2,3; etc. to get 64 bits
- Number of instructions (op-codes) is large, so we will only consider a subset
 - Because we are working with a subset, on the test, only instructions as shown on the MIPS handout will be needed
 - You can include this sheet on your tests

Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$t1	9	temporary (not preserved across call)
\$t2	10	temporary (not preserved across call)
\$t3	11	temporary (not preserved across call)
\$t4	12	temporary (not preserved across call)
\$t5	13	temporary (not preserved across call)
\$t6	14	temporary (not preserved across call)
\$t7	15	temporary (not preserved across call)
\$s0	16	saved temporary (preserved across call)
\$s1	17	saved temporary (preserved across call)
\$s2	18	saved temporary (preserved across call)
\$s3	19	saved temporary (preserved across call)
\$s4	20	saved temporary (preserved across call)
\$s5	21	saved temporary (preserved across call)
\$s6	22	saved temporary (preserved across call)
\$s7	23	saved temporary (preserved across call)
\$t8	24	temporary (not preserved across call)
\$t9	25	temporary (not preserved across call)
\$k0	26	reserved for OS kernel
\$k1	27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)

FIGURE A.6.1 MIPS registers and usage convention.

MARS Help

- When you bring up MARS, on the top toolbar, you will find a Help button
- Select Help and you will see the MARS 3.7 Help window.
- To review the MIPS instructions select the MIPS tab in the top of the window, and the Basic instructions or extended (pseudo) instructions tab below it.

Important Note

- MARS as an emulator is, for the most part, a superset of MIPS
 - Executes the same MIPS instruction set
 - Executes the same pseudo-instruction set and system calls
 - Extends both pseudo and system instructions
- All MIPS emulators, in additions to MIPS instructions allow for pseudo instructions

Pseudo address/instruction

- As a RISC machine, all MIPS instructions take one 32-bit word - regularity
 - NOTE: In MIPS all words must be on a 32-bit boundary, i.e. the absolute address must be divisible by 4
- There is a basic set of MIPS instructions
 - Most emulators extend this set with additional op-codes and/or addressing modes

Pseudo address/instruction

- There is a basic set of MIPS instructions
 - Most emulators extend this set with additional op-codes and/or addressing modes
 - Generally, these extensions are translated into 2 or more basic MIPS instructions
 - Thus a 4 instruction program may translate into more than 4 words

Pseudo address/instruction

- As an emulator, the process of I/O is facilitated with special I/O instructions that are not translated but result in system calls to special routines
 - This is much like what is done in a high level language when I/O is performed.

MIPS

- 32 registers – 32 bits each
- add \$t1, \$t2, \$t3
- addi \$t1, \$t2, 25
 - addi \$t1,\$t2, -23
- lw \$t0, #(\$t1) # only on word boundary
- lw \$t0,address # pseudo instruction
- sw \$t0,#(\$t1)
- Sw \$t0,address

MIPS Machine Language

- All MIPS assembly language instructions are converted to a single 32-bit machine language instruction
- EXCEPT – pseudo instructions which may be converted to multiple 32-bit words.

MIPS Addition

- MIPS addition is one of the simpler of the MIPS instructions
- Addition (integer only for now)
 - MIPS is a 2's complement machine
 - Adds the contents of two registers and stores the result in a third register

MIPS Addition

- Standard add
 - add \$5, \$6, \$3 # $\$5 = \$6 + \$3$
- Add immediate (operand is part of the instruction)
 - addi \$5, \$2, 25 # $\$5 = \$2 + 25$
 - addi \$4, \$2, -23 # $\$4 = \$2 - 23$
 - addi \$6, \$7, 0xff # $\$6 = \$7 + ff_{16}$

MIPS Addition

- MIPS also has what is called an unsigned add

`addu $t0,$t1,$t2`

- The `addu` is like the `add`, except it does not trap on overflow.

MIPS Subtraction

- Standard subtract
 - sub \$5, \$6, \$3 # \$5=\$6 - \$3
- Notes:
 - MIPS is a 2's complement machine and hence negative integers are stored in 2's complement.
 - There is no MIPS subtract immediate because you can do it with addi of a negative.

Overflow

- Both addition and subtraction of integers can generate overflow
 - If the operands are of different signs, overflow cannot occur.
 - If the operands are of the same sign (both + or both -) overflow can occur.
 - Under this condition, overflow has occurred if the sign of the result differs from the sign of the two operands.

Register Addressing

- All of MIPS' 32 registers can be addressed simply with $\$ \#$, where $0 \leq \# \leq 31$
- Also, each of MIPS's 32 registers have a mnemonic to associate with them
 - $\$zero$ denotes register $\$0$
 - $\$t0 - \$t7$ denotes registers $\$8 - \15
 - By convention, these are temporary register, i.e. not preserved across procedure calls
 - Review the MIPS handout for the mnemonic for each of the 32 registers.

C to MIPS

- Assume register contents are as follows
 - \$t0 = contents of location a
 - \$t1 = b
 - \$t2 = c
 - \$t3 = d
 - \$t4 = e
- Write the MIPS code for
 - a=b+c;
 - a=b+c+d;
 - a=(b+c)-(d+e);
 - a=c-10;

C to MIPS

- Write the MIPS code for

`a=b+c;` *add \$t0, \$t1, \$t2*

`a=b+c+d;` *add \$t0, \$t2, \$t3*
 add \$t0, \$t0, \$t3

`a=(b+c)-(d+e);` *add \$t0, \$t1, \$t2*
 add \$t5, \$t3, \$t4
 sub \$t0, \$t0, \$t5

`a=c - 10;` *addi \$t0, \$t2, -10*

Pseudo-Operands

- In MIPS, the space for an immediate operand is 16 bits, i.e. the low order 16 bits of the 32-bit instruction
 - What this means is that the immediate operand must be $-32,768 \leq \text{value} \leq 32,767$
 - You will find in MARS and SPIM, an instruction like
 - `addi $t0,$t1,0x45000`

Works!

Pseudo-Operands

- The way this works is because the assembler recognizes that the value is too big, and replaces this instruction with two others lui (load upper immediate) and addi (add immediate). Thus,

addi \$t0,\$t1,0x45000

Becomes

*lui \$t0,0x4 # load 4 into \$t0 and then
shift left 16 bits, bringing 0's
on the right*

addi \$t0, \$t0, 0x5000

Instructions to Date

- So far we have used or covered
 - Add
 - Sub
 - Lui
 - Register numbers and mnemonics
- For the lui instruction, why did I choose and immediate operand that was hex?

Instructions Covered So Far

- For the lui instruction, why did I choose and immediate operand that was hex?
 - Converting a 32-bit hex to its two 16-bit parts is trivial. If the value is given in decimal
 - Convert the decimal to its hex, and then split it

Referencing Memory

- MIPS memory
 - In the emulator, you have a total of 1MB of memory
 - MIPS memory can be addressed at the byte (8 bits), half-word (16 bits) or word (32-bits) level
 - Memory addresses are 32 bits, unsigned
 - Bytes fall on any 32-bit address
 - Half words are only on even boundaries
 - Words are only on (mod 4) boundaries

Referencing Memory

- Memory referencing instructions

lb, sb load/store byte

lh, sh load/store half-word

Note: lh and sh are used for unicode characters

lw, sw load/store word

Signs for register Loads

- When a byte (8-bits) or a half word (16 bits) is loaded into a register, the machine views it as a number (integer, 2's complement scheme).
- Because of this, the sign bit (bit 7 or 15) is replicated for the remainder of the 32-bit register contents.
- On unsigned loads, a 0 is replicated.
- This is not true for sw, sh, and sb

Addressing Memory

- We have defined the op-codes to access memory, but not how to address memory

lw <register>, <memory location>

sw <register>, <memory location>

<memory location> is of the form

offset(<register>)

lw \$t0, 8(\$t1)

Accessing Memory

- Assume that register \$5 contains the address in memory of the variable k. If we want to store the contents or value of k in register \$t6:

```
lw    $t6, 0($5)
```

What about the word following that of location k?

```
lw    $t6, 4($5)
```

Accessing Memory

- Let's say we have defined a variable FIRST and allocated a memory location for it.
 - How would we load the contents of location or variable FIRST in register \$t1?

```
la <register>, <memory>
```

This is a pseudo-instruction that loads the address of <memory> in <register>

Accessing Memory

```
la $t1,FIRST
```

Note: This does not load the contents of location FIRST into \$t1. It loads the address for FIRST in \$t1

Accessing Memory

- Now, let's say we want to load the contents of memory location k into register $\$t1$.

```
la $t1,k
```

```
lw $t1,0($t1)
```

Accessing Memory

- There is another way to do what we just did. It generates a pseudo-address approach that once again the assembler takes care of

```
lw $t1,k($t1)
```

Putting it all together

- Let's say W , X , Y , and Z are memory or variables already declared (we will show you how to do this later).
- Show how to computer

$$Z = X+Y-Z+W$$

Putting it all together

$$Z = X+Y-Z+W$$

$$Z = X+Y-Z+W$$

```
la $t0,X
lw $t0,0($t0) # $t0=X
la $t1,Y
lw $t1,0($t1) # $t1=Y
add $t0,$t0,$t1 # $t0=X+Y
la $t9,Z
lw $t1,0($t1)
sub $t0,$t0,$t1 # $t0=X+Y-Z
la $t1,W
lw $t1,0($t1)
add $t0,$t0,$t1 # $t0=X+Y-Z+W
sw $t0,0($t9) # Z=X+Y-Z+W
```

OR

$$Z = X+Y-Z+W$$

```
lw $t0,X($0)
lw $t1,Y($0)
add $t0,$t0,$t1
lw $t1,Z($0)
sub $t0,$t0,$t1
lw $t1,W($0)
add $t0,$t0,$t1
sw $t0,W($0)
```

What Next?

- So far we know how to:
 - add and subtract integers
 - Load and store values from a defined memory location
- But we don't know how to
 - Read in integers
 - Allocate memory in which to store those integers

Difference between HL language and Assembly Lang.

- In a high level language when one declares a variable, that variable has a type.
- Based on the declared type, the compiler only allows certain operations to be performed
- In assembly language, a variable name references a byte location, and it says nothing about the type of data stored there

Declaring a “Variable”

```
.word 10, 0x25, -3    # allocates a word of value  
                      # 10, hex 25, and -3 in  
                      # consecutive memory  
                      # locations
```

```
.byte 5, 0xff, 17    # allocates bytes rather than  
                     # words
```

```
.ascii "a test"     # allocates an ASCII string
```

Declaring a “Variable”

```
.asciiiz “a test” # allocates a null terminated  
# ASCII string
```

```
.asciiiz “\t\n” # allocates a null terminated  
# ASCII string of a tab followed  
# by a newline character
```

```
.space 100 # allocates 100 bytes (25 words)  
# of memory, values unknown
```

Declaring a “Variable”

- Actually, all the preceding has done is allocate storage and given it value(s).
- We need to be able to reference those locations, and we do that by giving the storage a label
 - If you want you can call this a variable name (a letter followed by letters & digits), but it is different, as we indicated from a programming language variable name.

Declaring a “Variable”

- Let's say we wanted to allocate a 32-bit location for an integer which we will label a x

```
X: .word 0 # The value 0 is arbitrary, but usually  
# ok.
```

Declaring a “Variable”

Let's say we wanted to allocate a null terminated string “Input an integer: “ and label it INPUT

```
INPUT: .asciiz “Input an Integer: “
```

```
# Notice we don't explicitly put in the null
```

Remember, a null character is a byte with decimal equivalent value of 0

Take a Breadth

- We are getting close to knowing enough MIPS to solve our beginning problem, i.e.

Write an ALP to ask the user to input two integers and then print out their sum

- What we need is a way to do I/O and the basic structure or template of a MIPS ALP

BASIC Structure of MIPS Program

- To make it simpler on the grader, and me, **EVERYONE** will need to adhere to the following basic MIPS program structure

*# assignment # and your name – a **MUST***

.data

the data etc. to be used by your program

.text

main:

Use the label “main” to label the first

instruction of your program

That which is in red is the template

I/O

- MIPS I/O is quite complicated
- We use a simplified version of the real I/O by using emulator supplied system calls

System Calls

- There are numerous system calls
- A specific system call is identified by an integer which is placed in register \$v0.
- Any parameters needed by a particular system call are expected to be in registers \$a0,.....
- The next slide defines the system calls – actually there are more than we will need for this class

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$a0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

FIGURE A.9.1 System services.

Syscall Examples

- Let's say we want to print as an integer the contents of register \$t0

```
addi $v0,$zero,1      # 1 is the code for print int
add $a0,$t0,$zero     # $a0 gets the parameter
syscall
```

Useful Pseudo-Instructions

li \$v0,5

loads a 5 in \$v0

move \$v0,\$t0

copies \$t0 into \$v0

Pseudo Instructions

- What is meant by a pseudo-instruction?
 - An instruction that SPIM understands, but for which there is no MIPS instruction. The SPIM assembler translates a pseudo-instruction into MIPS code

`move $t0,$s1` \Rightarrow `add $t0,$s1,$zero`

`bge $s0, $s1, loop` \Rightarrow `slt $t0, $s0, $s1`
`bne $t0, $zero, loop`

At Last

- Remember, the problem was:
- Write an *ALP* to ask the user to input two integers and then print out their sum

```
#####
```

```
# Example 1
```

```
# This program
```

```
#     asks the user to input two integers
```

```
#     and then prints out their sum
```

```
# Author: Don Cooley
```

```
#####
```

`.data`

`Inmess1: .ascii "Input first integer: "`

`Inmess2: .ascii "Input the second Integer: "`

`Outmess: .ascii "Their sum is "`

`NL: .ascii "\n"`

```
.text
```

```
main:
```

```
# get the first integer
```

```
li $v0,4
```

```
la $a0,Inmess1
```

```
syscall
```

```
li $v0,5
```

```
syscall
```

```
move $t0,$v0    # $t0 = first integer
```

Get second integer

li \$v0,4

la \$a0,Inmess2

syscall

li \$v0,5

syscall

move \$t1,\$v0 #not really needed

add \$t0,\$t0,\$t1 # $t0 = \text{first int} + \text{second int}$

li \$v0,4

la \$a0,Outmess

syscall

li \$v0, 1

move \$a0,\$t0

syscall

li \$v0,4

la \$a0,NL

syscall

li \$v0,10

syscall

More MIPS

sll \$t0,\$t1,3 # shift left logical 3 bits

srl \$t0,\$t1,4

and \$t0,\$t1,\$t2 # all of these are bit-by-bit

andi \$t0,\$t1,5

or \$t0,\$t1,\$t2

ori \$t0,\$t1,0xfc

xor \$t0,\$t1,\$t2

nor \$t0,\$t1,\$t2

MIPS

- Why have the NOR instead of simply having a NOT instruction?
- NOR is 2-operand, and NOT is single, thus the NOR fits better with the AND and OR.

Question

- Can you make a NOR into a NOT?
- `nor $t1,$t1,$zero`

Question

- Give multiple ways to store a 1 in register \$t0
- `addi $t0,$zero,1`
- `nor $t0, $zero,$zero`
- `andi $t0,$t0,1`
- `ori $t0,$zero,1`

So Far

- We now have:
 - Basic program template
 - Add and subtract
 - Basic I/O (system calls)
 - Logical (Boolean) operations
 - Shift's
 - Assembler directives
 - Memory access – Load/Store
 - What do you think is most needed next?

Logical and Loops?

- If-then
- If-then-else
- While
- For
- In order to implement any of these instructions, we must have an ability to make comparisons, and an ability to branch or jump

Comparisons

- `beq $t0,$t1,eql`
 - Branch to the instruction labeled “`eql`” if the contents of register `$t0` are equal to contents of register `$t1`
- `bne $t0,$t1,neq`
 - Branch to instruction labeled “`neq`” if the contents of register `$t0` are not equal to contents of register `$t1`

If-Then-else

- What really has to happen for an if-then-else?

<Perform some kind of test and if false jump around next code>

{This is the true part}

<jump around this next segment>

{This is the false part}

If-Then-else

- Tests are always comparisons, = <>, < <=, etc.
- Jumping from the true code around the false code requires an unconditional branch or jump.
- What happens on a branch/jump
 - Program counter's (PC's) contents are replaced in some way

Branches and Jumps

- On a branch instruction, the low order 16 bits of the instruction contains the offset.
 - $\text{NewPC} = \text{OldPC} + 4 * \text{offset}$
 - The offset is measure in words since instructions have to be on word (32-bit) boundaries

Branches and Jumps

- On a jump instruction, only 6 bits are used for the opcode, and hence 26 bits are available for the distance to jump
 - $\text{NewPC} = \text{OldPC}_{[28-31]} + \text{Jump}_{[2-27 \leftarrow 0-26]} + 0_{[0-1]}$

What we are doing is appending the left-most 4 bits of the current PC, and the jump amount shifted left 2 bits, i.e. $\times 4$, and then 2 0's on the end.

Branches and Jumps

- What is the maximum distance, in bytes that a:
 - Branch can execute
 - Jump can execute
- There is really only one unconditional jump.
 - Its op-code is “J <label>”

Branches and Jumps

- There are several branch instructions.
 - Many of the branch instructions are pseudo-instructions
 - Look in Mars for the pseudo instructions
 - All of the pseudo-branch instructions can be implemented using the regular MIPS branch instructions with some extra calls possibly to other instructions such as lui, etc.
- All of these instructions are defined in MARS ->Help->Basic instructions & Extended

If-Then

- In MIPS, implement

if $A \neq B$ then

 {do something}

{whatever}

```
lw    $t0,A($zero)
lw    $t1,B($zero)
beq   $t0,$t1,done
      {do something}
done: {whatever}
```

If-Then-Else

Implement the following in MIPS and/or pseudo code.

If $A > B$ then

 {do something}

else

 {do something different}

{whatever}

```
lw    $t0,A($zero)
```

```
lw    $t1,B($zero)
```

```
ble   $t0,$t1,else
```

```
{do something}
```

```
b    done
```

```
else: {do something else}
```

```
done: {whatever}
```

*Note: In general, pseudo instructions
(branches) will be relied on heavily*

While

Implement the following:

```
while (A<10)
    {do something}
    {modify A}
{whatever}
```

```
lw $t0,A($zero)
ovr: bge $t0,10,done
    {do something}
    {modify A, i.e. $t0}
b ovr
done:
```

For Loop

- Implement the following

```
for (i=0; i<= MAX; i=i+2)  
    {do something}
```

For Loop

```
li $t0,0  
lw $t1, MAX($zero)  
ovr: bgt $t0,$t1,done  
    {do something}  
    addi $t0, $t0, 2  
    b ovr  
done: ...
```

Jump vs Branches

- The only difference between a jump and a branch is the distance jumped and that “j’s” are unconditional
 - J ABC
 - Beq \$zero,\$zero, ABC
 - Etc.

Branches

- In an instruction like `beq $t0,$t1,ABC`, there are only 16 bits for the jump distance
- 16 bits 2's complement
 - -2^{15} to $+2^{15} - 1$
 - Since instructions can only be on word boundaries, these distances are in words, i.e. the actual number of bytes jumped is $4 \times$ that number

- Let's say the instruction `beq $t0,$t1,ABC` is at memory location 100 and that the instruction labeled ABC is at location 48.
- What will be the value in the low order 16 bits?
 - When the branch is executed, the PC (program counter) has already been updated to the value 104 (next instruction after the `beq`)
 - Thus, the value would be $(48 - 104)/4 = -14$
 - Remember, these 16 bits will be in 2's complement since the value is negative

Jumps – Getting the Most out of 32 bits

- Because the op-code takes 6 bits, for a jump, that makes available only 26 bits.
 - Note: unlike the branch instructions which specify a word offset relative to the PC, the jump specifies a real (absolute) address.
- These 26 bits are expanded as follows:
 - Because instructions are on a word boundary, 2 bits are appended to the end for the word, i.e. two 0's are appended to the right end, i.e. multiply by 4
 - The 4 high order bits currently in the PC are appended on the left of the address to make a total of 32 bits.

Machine Representation

R-Type

- Op 6 bits (op code)
- Rs 5 bits first source operand
- Rt 5 bits second register operand
- Rd 5 bits register destination
 - (notice that register order in the word is not the same as in the assembly code)
- Shamt: Shift amount
- Funct: function code

Machine Representation

I-Type

- Op 6 bits (op code)
- Rs 5 bits first source operand
- Rt 5 bits second register operand
- Value 16 bits immediate value

Machine Representation

\$t1 has the base address of A, and \$s2 has the value of H

$A[300]=H+A[300];$

Lw	\$t0,1200(\$t1)	35	9	8	1200
Add	\$t0,\$s2,\$t0	0	18	8	0 32
Sw	\$t0,1200(\$t1)	43	9	8	1200

- Write a MIPS assembly program to add integers input by the user until the user enters a zero. At that point, the program should output the sum. Each input request by the program and the final sum output should be annotated with appropriate messages. The console output of your program should appear similar to the following example;

ENTER A ZERO TO END

enter a number: 21

enter a number: 42

enter a number: 9

enter a number: 145

enter a number: 0

the sum of this list is: 217

```
.data
value1:    .word 77
msg1:     .asciiiz "ENTER A ZERO TO END\n"
msg2:     .asciiiz "Enter a number: "
msg3:     .asciiiz "The sum of these numbers is: "
```

```
.text
```

```
.globl main
```

```
main:
```

```
    la $a0,msg1    #load address of msg1 in $a0
```

```
    li $v0,4
```

```
    syscall        #system call code to print msg1
```

loop:

la \$a0,msg2

#load address of msg2 in \$a0

li \$v0,4

syscall

#system call code to print msg2

li \$v0, 5

syscall

#system call to read an integer

beq \$v0, \$zero, end

add \$t0, \$t0, \$v0

#add number to total

j loop