CS312 : MIPS Assembly Programming Simulator Manual

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
School of Computing and Electrical Engineering,
IIT Mandi

August 16, 2023

Abstract

To learn the MIPS 32 assembly programming, we will be using the simulator. Simulators are the set of computer programs that run on the host systems; thus, it can easily alter any functionality or programming developed for the target system can be easily altered to meet the new design requirement. One such simulator for MIPS 32 is ***SPIM***. The SPIM simulator read and execute the assembly program written for the MIPS processor. The most recent and current version of SPIM is ***QtSPIM***. The QtSPIM is available for Windows, Ubuntu, and Mac at:

`https://sourceforge.net/projects/spimsimulator/files/`

In this installation manual, the installation steps are focused only on Ubuntu.

QtSPIM Prerequisite Package and Installation Steps

"QtSPIM," aka SPIM using the Qt cross-platform application GUI framework.
Install the "Qt" packages framework that is required for SPIM.

```
sudo apt-get install qt4-dev-tools qt4-doc libqt4-help
```

Download the pre-packaged version of QtSPIM for Linux:

```
wget https://sourceforge.net/projects/spimsimulator/files/qtspim_9.1.20_linux64.deb
```

Install the QtSPIM package

```
sudo dpkg -i qtspim_9.1.20_linux64.deb
```

QtSPIM Building and Usage

To run QtSPIM:

```
qtspim &
```

Once QtSPIM is launched, the main window is appeared as shown below:

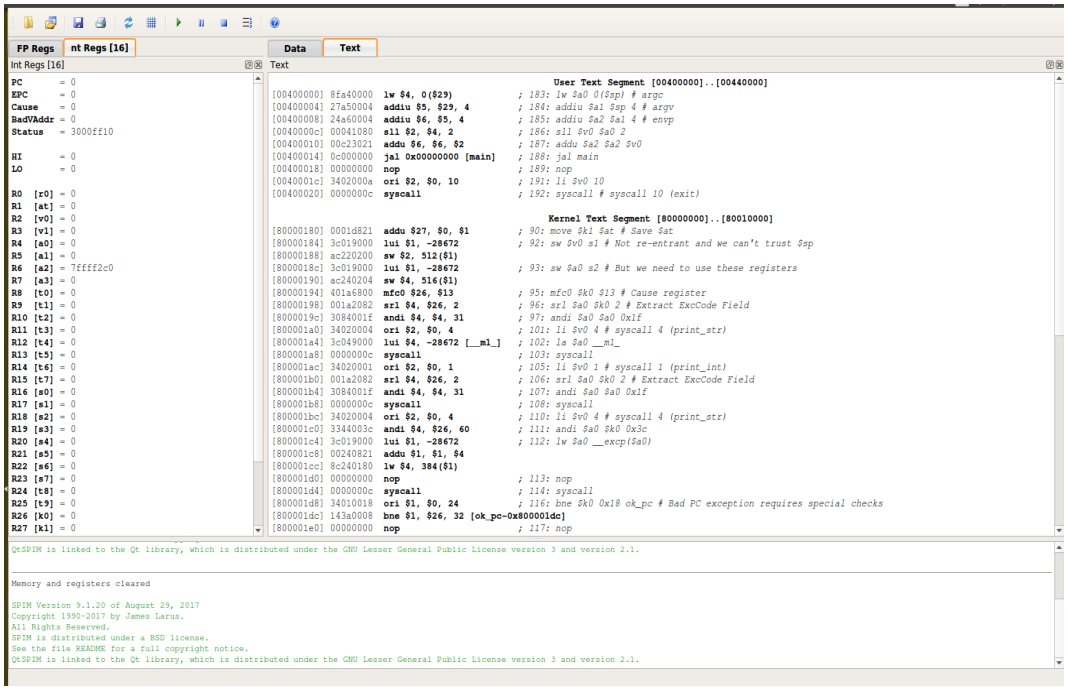

Figure 1: "QtSPIM" Main Screen

The window contains three important panels: Register, Memory, and Message.

Register Panel: The panel shows the total numbers of registers and content of each register. As can be seen, there are two tabs: one for floating-point and one for integer registers. The integer registers comprise a total of 32 registers ($R0$ to $R31$) and the special purpose registers such as Program Counter (PC) and Base Value Address register ($BadVAddr$).

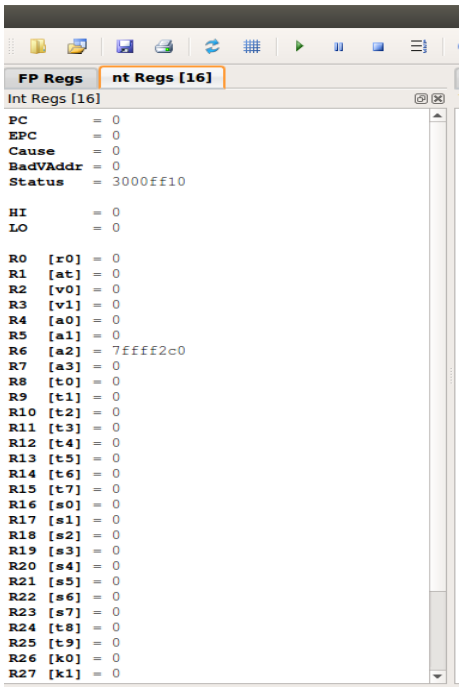

Figure 2: Integer Register Window in Register Panel

Whereas, the floating point registers comprises of total 32 single precision registers ($FG0$ to $FG31$) and 16 double precision registers ($FP0$ to $FP30$) along with some special purpose registers.

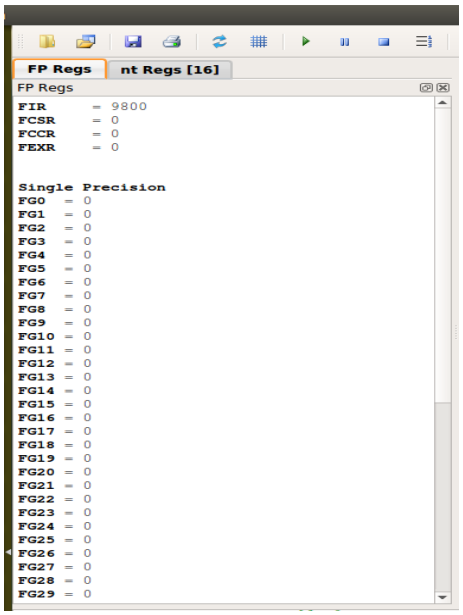

Figure 3: Floating Point Register Window in Register Panel

Memory Panel: The memory panel has two tabs: Data and Text. The text tab presents the content of the memory address space. It has the following information from left to right:

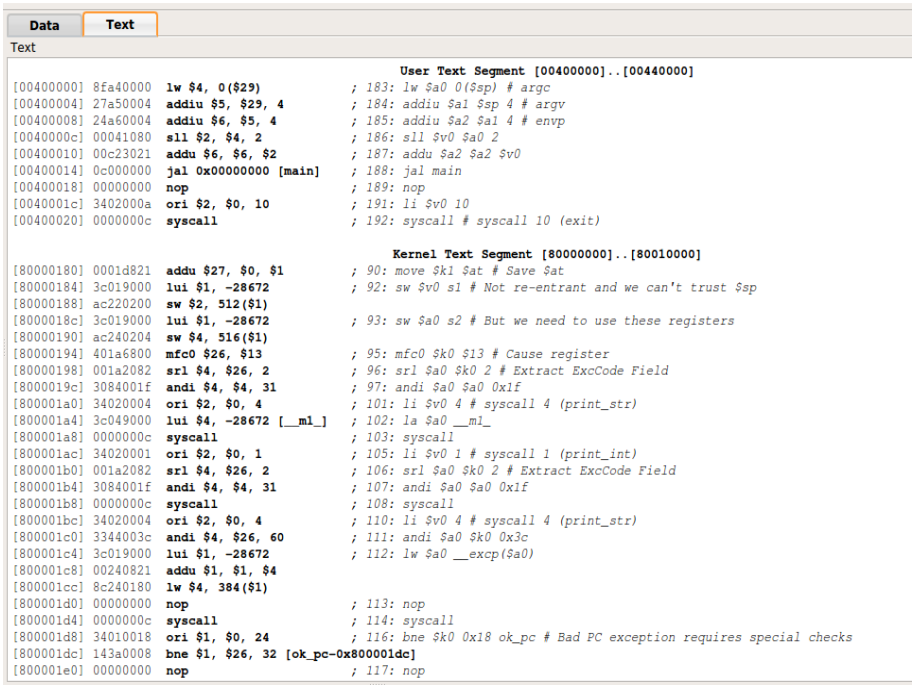

Figure 4: Text tab Window in Memory Panel

- Memory address of an instruction in hexadecimal format (shown in parenthesis).
- The content of that memory address in hexadecimal. In binary form, this is the actual MIPS instruction that the process runs.
- The corresponding "human readable" assembly instruction along with the register number.
- Assembly language program you wrote using symbolic register name and memory register.

The Data tab presents the data memory space. It comprises variables, data array, and stack content.

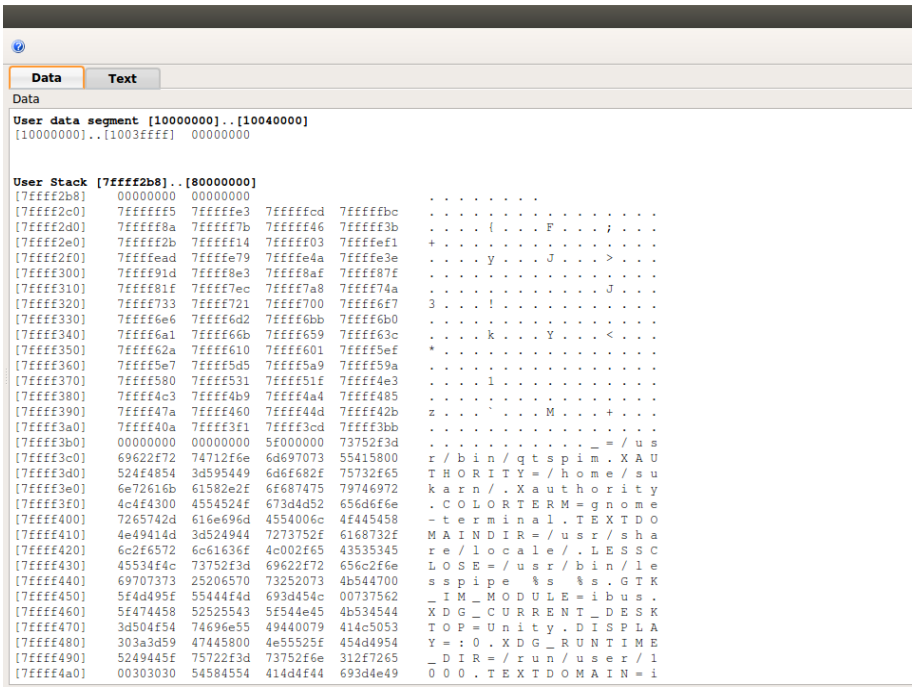

Figure 5: Data tab Window in Memory Panel

Message Panel: The message panel prompt the required message from QtSPIM to the user.

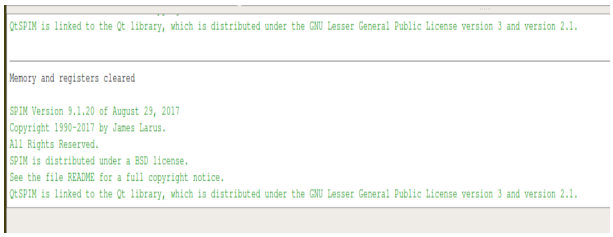

Figure 6: Message / Output Panel

Sample Program Loading and Running Steps

The first sample program loads the two value into two registers and performs addition, and store back the result into the memory location.

```
# A example of sample MIPS instructions
# used to test QtSPIM

# Declare main as a global function
.globl main

# All program code is placed after the
# .text assembler directive
```

```
.text

# The label 'main' represents the starting point
main:
li $t1, 25 # Load immediate value (25)
lw $t2, value # Load the word stored in value (see bottom)
add $t3, $t1, $t2 # Add
sw $t3, Z #Store the answer in Z (declared at the bottom)

# Exit the program by means of a syscall.
# There are many syscalls - pick the desired one
# by placing its code in $v0. The code for exit is "10"
li $v0, 10 # Sets $v0 to "10" to select exit syscall
syscall # Exit

# All memory structures are placed after the
# .data assembler directive
.data

# The .word assembler directive reserves space
# in memory for a single 4-byte word (or multiple 4-byte words)
# and assigns that memory location an initial value
# (or a comma separated list of initial values)
value: .word 10
Z: .word 0
```

In the above program, the *value* and Z are declared separately. The memory location Z store the final output of the addition. The source operands are provided in two forms: immediate and through memory location.

Copy the program and paste into the editor by using the following step: **File -≥ Reinitialize and Load File.** (note that the new Ubuntu "Unity" GUI doesn't show the top-of-screen menu bar until you hover the mouse on it). In this way, the register space is cleared, and the simulator is reset.

Once the program has been loaded into the program memory space, the first instruction is at memory location $0x00400024$. Now the question arises, why is it not started from the memory location $0x0$. This is because the program starts with the function *main*(), but there is some code (like code related to linking and loading) that runs before the *main*().

To simulate your assembly instruction. There are three-step or choices:

- Run the program from beginning to end (using the *play Run/Continue* button or *F5*) to see the output of the program at the Message panel.
- To see the effect of assembly instruction in the program visible state (PC, Register Content and Memory location) step the program line-wise using *123 single step button* or *F10*.
- To run the program until the breakpoints, right-click on any line in the memory panel.

Step through the entire program and figure out how it works. Be make sure you understand how the "QtSPIM" environment works.

More details about the "QtSPIM" can be found at:

- `http://spimsimulator.sourceforge.net/`

Other than the "QtSPIM" simulation environment, following are the list of assembler directives and the registers used in the assembly programming.

Assembler Directive

The assembler directive allows programmer to request the assembler to do some useful work when converting from assembly source code to binary code.

Directive	Result
.word w1,.....,wn	Store n 32 bit values in successive memory words
.half h1,.....,hn	Store n 16 bit values in successive memory words
.byte b1,.....,bn	Store n 8 bit values in successive memory words
.ascii str	Store the ASCII string str in the memory. Strings are in double-quotes i.e. "Computer Science"
.asciiz str	Store the ASCII string str in the memory and null terminate it Strings are in double-quotes i.e. "Computer Science"
.space n	Leave an empty n byte region of memory for later use
.align n	Align the next datum on a 2^n byte of a memory For example, .align 2 aligns the next value on a word memory

Table 1: List of Assembler Directive

Set of Registers

MIPS has 32 general purpose register, that could be used by the programmer based on their requirement. However, the register has been divided into groups and used for different applications. Register has both number (used by the hardware) and name (used by the assembler programmer).

Register Number	Register Name	Description
0	**$zero**	0
2-3	**$v0-$v1**	(values) from expression evaluation and function result
4-7	**$a0-$a3**	(arguments) First four parameter for subroutine
8-15, 24-25	**$t0-$t9**	Temporary Variables
16-23	**$s0-$s7**	Saved values representing final computed result
31	**$ra**	Return address

Table 2: List of Registers and their descriptions

CS214 : MIPS Assembly Programming
Assignment 1: Sequential Construct-I Tutorial
Integer Arithmetic and Logical Instructions

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

August 23, 2023

Sequential Construct

The programming requires a dividing a task, into small unit of work. These unit of work are represented with programming construct that represents part of task. In the sequential construct, the designated task is broken into smaller task one follow by another.

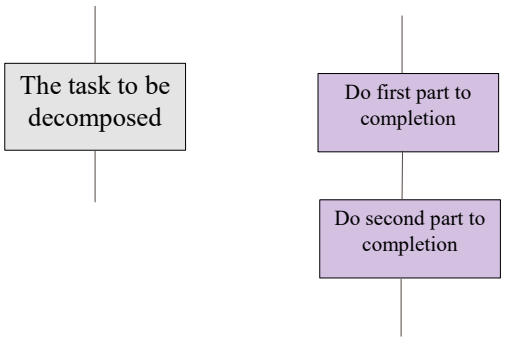

Figure 1: Representational view of Sequential Construct

Arithmetic Instructions

Following are the set of arithmetic instruction that is to be used in this assignment.
In all the list of instruction, $1, $2 and $3 represent the registers for the understanding purposes. In the assignment, you have to use the register name not the corresponding register number. Note that the details and list of the register is already provided in the instruction manual.

Instruction	Example	Meaning	Comments
add	add $1, $2, $3	$1=$2+$3	
subtract	sub $1, $2, $3	$1=$2-$3	
add immediate	addi $1,$2,100	$1=$2+100	"immediate" means a constant number
add unsigned	addu $1,$2,$3	$1=$2+$3	Values are treated as unsigned integers, not two's complement integers
subtract unsigned	subu $1,$2,$3	$1=$2-$3	Values are treated as unsigned integers, not two's complement integers
add immediate unsigned	addiu $1,$2,100	$1=$2+100	Values are treated as unsigned integers, not two's complement integers
Multiply (without overflow)	mul $1,$2,$3	$1=$2*$3	Result is only 32 bit
Multiply	mult $2,$3	$hi,$low=2∗3	Upper 32 bits stored in special register hi Lower 32 bits stored in special register lo
Divide	div $2,$3	$hi,$low=$2/$3	Remainder stored in special register hi Quotient stored in special register lo
Unsigned Divide	divu $2,$3	$hi,$low=$2/$3	$2 and $3 store unsigned values. Remainder stored in special register hi Quotient stored in special register lo

Table 1: Arithmetic Instruction with their details and explanations

Logical Instructions

Following are the set of logical instructions that is to be used in this assignment.

Instruction	Example	Meaning	Comments
and	and $1, $2, $3	$1=$2&$3	Bitwise AND
or	or $1, $2, $3	$1=$2—$3	Bitwise OR
and immediate	andi $1,$2,100	$1=$2&100	Bitwise AND with immediate value
or immediate	ori $1,$2,100	$1=$2—100	Bitwise OR with immediate value
nor	nor $1,$2,$3	$1=$2↓$3	Bitwise NOR
shift left logical	sll $1,$2,10	$1=$2<<10	Shift left by constant number of bits
shift right logical	srl $1,$2,10	$1=$2>>100	Shift right by constant number of bits

Table 2: Logical Instructions with their details and explanations

Data Movement Instructions

Following are the set of data movement instructions that is to be used in this assignment.

Instruction	Example	Meaning	Comments
load word	lw $1, 100($2)	$1=Memory[$2+100]	Copy from memory to register
store word	sw $1, 100($2)	Memory[$2+100]=$1	Copy from register to memory
load upper immediate	lui $1, 100	$1=100x2^16	Load constant into upper 16bits. Lower 16bits are set to zero.
load address	la $1, label	$1=Address of label	Pseudo-instruction (provided by the assembler, not processor!) Load computed address of label (not its contents) into register
load immediate	li $1, 100	$1=100	Loads immediate value into register
move from hi	mfhi $2	$2=hi	Copy from special purpose register hi to general register
move from lo	mflo $2	$2=lo	Copy from special purpose register lo to general register
move	move $1,$2	$1=$2	Copy from register to register

Table 3: List of Data Movement Instruction with their details and explanations

Note: Variation on load and store also exist for smaller data sizes.

1. 16-bit halfword: lh and sh
2. 8-bit byte: lb and sb

System Calls

The SPIM provide a large number of system call. These are the call to Operating System and do not represent MIPS process instruction. These call are either implemented by the OS or standard library.

System calls are used for input, output and to exit the program. These calls are commences with the help of *syscall* function. To use the instruction, the appropriate arguments in registers $v0, $a0-$a1, or $f12 are supplied depending on the specific call required. Following are the list of system calls that are to be required in this assignment.

Service	Operation	Code (in $v0)	Arguements
exit	stop program from running	10	none
exit2	stop program from running and return an integer	17	$a0=result (integer number)

Table 4: List of System Calls with their usage and explanations

CS214 : MIPS Assembly Programming
Tutorial: Sequential Construct-II
Floating Point Instructions
Input Output Instructions
Comparison Instructions

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

August 27, 2023

Sequential Construct

The programming requires a dividing a task, into small unit of work. These unit of work are represented with programming construct that represents part of task. In the sequential construct, the designated task is broken into smaller task one follow by another.

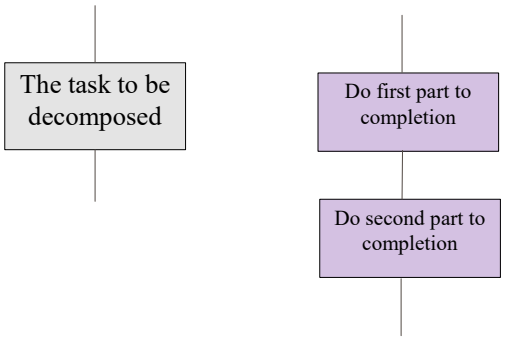

Figure 1: Representational view of Sequential Construct

Floating Point Instructions

MIPS provide several set of floating point instructions:

- Arithmetic
- Data Movement
- Conditional Jump

In this assignment, we have discussed only Arithmetic and Data Movement Instructions. Note that the details and list of the register is already provided in the instruction manual.

Instruction	Example	Meaning	Comments
Arithmetic Instructions			
add	add.s $f0, $f1, $f2	$f0=$f1+$f2	none
subtract	sub.s $f0, $f1, $f2	$f0=$f1-$f2	none
multiply	mul.s $f0, $f1, $f2	$f0=$f1*$f2	none
division	div.s $f0, $f1, $f2	$f0=$f1/$f2	none
absolute	abs.s $f0, $f1	$f0=—$f1—	Absolute Value of floating point number
negative	neg.s $f0, $f1	$f0=-$f1	Negate the floating point number
Data Movement and Conversion Instructions			
load float	l.s $f0, 100($t2)	$f0=Mem[$t2+100]	Copy from Memory to Floating Point Register
store float	s.s $f0, 100($t2)	Mem[$t2+100]=$f0	Copy from Floating Point Register to Memory
load float immediate	li.s $f0, 10.0	$f0=10.0	Load Floating point immediate value into Register
move	move.s $f0, $f1	$f0=$f1	Copy from register to register
convert to integer	cvt.w.s $f2, $f4	$f2=$f4	Convert from single precision FP (f4) to integer (f2)
convert to float	cvt.s.w $f2, $f4	$f2=$f4	Convert from integer (f2) to single precision (f4)

Table 1: Floating Point Instructions with their details and explanations

System Call Related to I/O

System calls are used for input, output and to exit the program. These calls are commences with the help of *syscall* function. To use the instruction, the appropriate arguments in registers $v0, $a0-$a1, or $f12 are supplied depending on the specific call required. The system call will return the result values into the register based on the datatype and operation conducted.

Following are the set of system call that is to be used in this assignment.

Service	Operation	Code (in $v0)	Arguement	Results
print_int	Print integer number (32 bit)	1	$a0=integer to be printed	none
print_float	Print floating-point number (32 bit)	2	$f12=float to be printed	none
print_double	Print floating-point number (64 bit)	3	$f12=float to be printed	none
print_string	Print null-terminated character string	4	$a0=address of string in memory	none
read_int	Read integer number from user	5	none	integer written in $v0
read_float	Read floating-point number from user	6	none	float written in $f0
read_double	Read double floating point number from user	7	none	double written in $f0
read_string	Work the same as standard C library fgets()	8	$a0=memory address of string input buffer $a1=length of string buffer (n)	none
sbrk	Returns the address to a block of memory containing n additional bytes (dynamic memory allocation)	9	$a0=amount	address in $v0
print_char	Print Character	11	$a0=character to be printed	none
read_char	Read Character from user	12	none	char written in $v0

Table 2: List of System Calls with their usage and explanations

Comparison Instructions

Following are the set of comparison instructions that is to be used in this assignment.

Instruction	Example	Meaning	Comments
set on less than	slt $t1, $t2, $t3	if($t2<$t3)$t1=1; else $t1=0	Test if less than. If true, set $t1 to 1. Otherwise set $t1 to 0
set on less than immediate	slti $t1, $t2, 100	if($t2<100)$t1=1; else $t1=0	Test if less than. If true, set $t1 to 1. Otherwise set $t1 to 0
set equal	seq $t1, $t2, $t3	if($t2=$t3)$t1=1; else $t1=0	Test if equal. If true, set $t1 to 1. Otherwise set $t1 to 0
set greater than equal	sge $t1, $t2, $t3	if($t2>=$t3)$t1=1; else $t1=0	Test if greater than equal. If true, set $t1 to 1. Otherwise set $t1 to 0.
set greater than	sgt $t1, $t2, $t3	if($t2>$t3)$t1=1; else $t1=0	Test if greater than. If true, set $t1 to 1. Otherwise set $t1 to 0.
set less than equal	sle $t1, $t2, $t3	if($t2<=$t3)$t1=1; else $t1=0	Test if less than equal. If true, set $t1 to 1. Otherwise set $t1 to 0.
set not equal	sne $t1, $t2, $t3	if($t2! =$t3)$t1=1; else $t1=0	Test if not equal. If true, set $t1 to 1. Otherwise set $t1 to 0.

Table 3: List of Comparison Instructions with their details and explanations

CS214 : MIPS Assembly Programming
Tutorial: Conditional Constructs

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE, IIT Mandi

September 2, 2023

Conditional Construct

The programming requires a dividing a task, into small unit of work. These unit of work are represented with programming construct that represents part of task. The conditional construct is used if the designated task consists of doing one of two subtasks, but not both

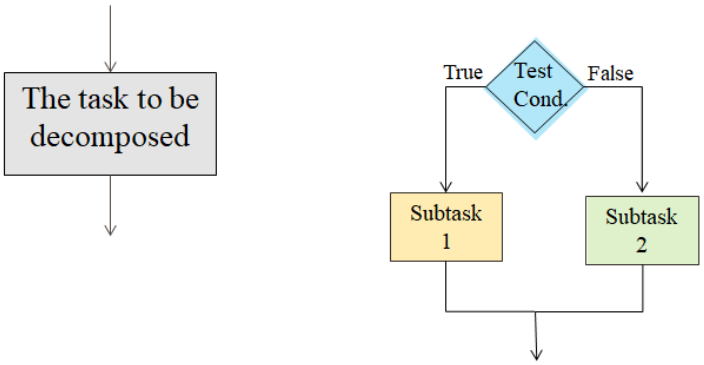

Figure 1: Representational view of Conditional Construct

Conditional Branch Instructions

MIPS provides a large set of Conditional Branch instructions. All conditional branch instructions compare the values in two-register specified within the instruction. If the conditional is evaluated to true, the branch is taken, and the execution flow changes to the new location. Otherwise, the execution flow continues to the next sequential instruction. Note that the details and list of the register is already provided in the instruction manual.

Instruction	Example	Meaning	Comments
Integer Conditional Instructions			
branch on equal	beq $1, $2, 1000	if($1 == $2) go to PC+4+1000	Test if registers are equal
branch on not equal	bne $1, $2, 1000	if($1 != $2) go to PC+4+1000	Test if registers are not equal
branch on greater than	bgt $1, $2, 1000	if($1 >$2) go to PC+4+1000	Test if one register is greater than compared to other
branch on greater than or equal	bge $1, $2, 1000	if($1 >= $2) go to PC+4+1000	Test if one register is greater than or equal to other
branch on less than	blt $1, $2, 1000	if($1 <$2) go to PC+4+1000	Test if one register is less than compared to other
branch on less than or equal	ble $1, $2, 1000	if($1 <= $2) go to PC+4+1000	Test if one register is less than or equal to other
Floating Point Comparison and Conditional Instructions			
Equal Comparison	c.eq.s $f2, $f4	if($f2 == $f4) set code = 1 else code = 0	Test if floating point registers are equal
Less than or Equal to Comparison	c.le.s $f2, $f4	if($f2 <= $f4) set code = 1 else code = 0	Test if one floating point register is less than to equal to another one
Lesst than Comparison	c.lt.s $f2, $f4	if($f2 <$f4) set code = 1 else code = 0	Test if one floating point register is less than to another one
Greater than or Equal to Comparison	c.ge.s $f2, $f4	if($f2 >= $f4) set code = 1 else code = 0	Test if one floating point register is greater than or equal to another one
Greater than Comparison	c.gt.s $f2, $f4	if($f2 >$f4) set code = 1 else code = 0	Test if one floating point register is greater than another one
branch on set code	bclt label	if code == 1 then jump to label	Jump to the label if code is set
branch on reset code	bclf label	if code == 0 then jump to label	Jump to label if code is reset

Table 1: List of Conditional Branch Instructions with their details and explanations

Problem 1: Write a MIPS assembly program that takes two number (can be anything floating point or integer) as an input and print maximum between two of them as follows:

```
32.6 is greater than 25.0
```

Problem 2: Write an assembly program that takes year as an input from the user and check whether the input year is leap year or not. If it is leap year prompt the message

```
Input year is a leap year
```

Otherwise, prompt the message

```
Input year is not a leap year
```

Problem 3: Write an assembly program that determines whether the student is allowed to sit the examination provided his/her attendance is 75%. For the given problem statement, the MIPS assembly program takes the following input: The name of student, Total number of class held and Total class attended by the student. The output format is as follows:

```
Ajay is allowed to sit in the exam.
or
Ajay is not allowed to sit in the exam.
```

Problem 4: Write an MIPS assembly program that takes the marks of the student as an input (in the range of 1-100) and assign the grade. The grading policy are as follows:

```
Grade: A if marks >= 80
Grade: B if 80 < marks >= 60
Grade: C if 60 < marks >= 40
Grade: F otherwise
```

CS214 : MIPS Assembly Programming
Tutorial: Iterative Constructs

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE, IIT Mandi

September 9, 2023

Iterative Construct

The programming requires a dividing a task, into small unit of work. These unit of work are represented with programming construct that represents part of task. The iterative construct is used if the designated task consists of doing a sub-task a number of times, but only as long as some condition is true.

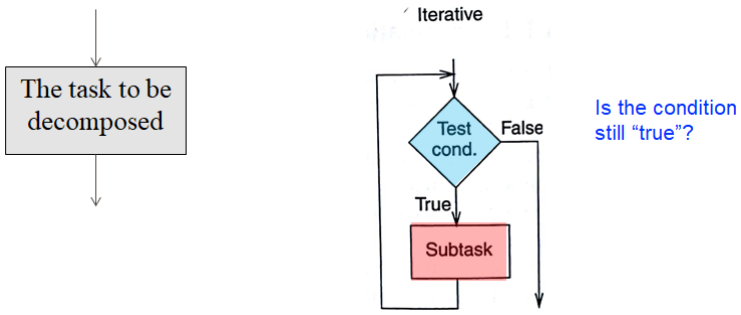

Figure 1: Representational view of Iterative Construct

Iterative Jump Instructions

MIPS provides four iterative jump instructions. All iterative instructions unconditionally jump into a specific address determined within the instruction. It is much easier to use a label for the jump instructions instead of an absolute number. For example, `j loop`, the label `loop` here should be defined somewhere else in the code.

Instruction	Example	Meaning	Comments
jump	`j 2000`	Go to address 2000	Jump to target address
jump register	`jr $2`	Go to address stored in $2	For switch, procedure return
jump and link	`jal 2000`	$ra=PC+4; go to address 2000	Use when making procedure call. This save the return address in $ra.
jump and link register	`jalr $2`	$ra=PC+4; go to address stored in $2	Use when making procedure call and return

Table 1: List of Iterative Jump Instructions with their details and explanations

CS214 : MIPS Assembly Programming
Tutorial 7: Functions

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

October 8, 2023

Functions

Functions are required to utilize the frequently accessed code, make a program more modular and readable and easier for debugging. Execution of a function change the control flow of the program two times: one at the time of calling the function and other at the time of returning from the function.

```
void main()
{
  int y, z;
  y = sum(42, 7);
  z = sum(10, -8);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Figure 1: Function Code Example in C

In the above example, the `main` function invokes the function `sum` twice and the function `sum` return two times, but at the different control point in the `main`. Note that each time the `sum` invoke, the control flow has to remember the appropriate return address.

MIPS uses the jump and link instruction `jal` (format details and example are already given in Assignment-5) to invoke the function

- The `jal` store the return address (which is the address of the next instruction in the control flow of `main`) into the dedicated register `$ra`, before changing the control flow to called function.
- It is the only instruction that can access the value in the program counter. Hence it can easily store the return address `PC + 4` of the caller function in `$ra`.

To transfer control back to caller function, the callee function has to jump to address provided by the `$ra` using the following instruction: `jr $ra`.

Function accept some number arguments and operate upon them and produce return values. For example, in the above code snippet, the values 42 and 7 in the function `sum` are the actual argument and the variables a and b are the formal argument. The function return the sum of a and b as a return value.

MIPS uses the following rules for the function arguments and the return values:

- With MIPS, upto four arguments can be passed by using only argument register $a0 - a3$ before invoking the function with `jal` command.
- A callee function can return upto two values using the register $v0 - v1$, before returning via `jr`.

Note that these above conventions are not enforced by the assembler or hardware, but it will be easy for different programmers to interface with the written code.

A Note about data types of the arguments passed in the function

- MIPS assembly language is untyped, means there is no distinction between integer, float, characters or pointers passed through argument.
- It is the assembly programmer job to type check different variable argument passed in the function. In other words, programmer need to make sure that argument value(s) and return value(s) are consistent to the program.

CS214 : MIPS Assembly Programming
Tutorial: Arrays and Strings

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

September 30, 2023

Arrays

An array is a collection of similar data-type variables. In MIPS assembly programming, accessing an array requires loading the base address into the register.

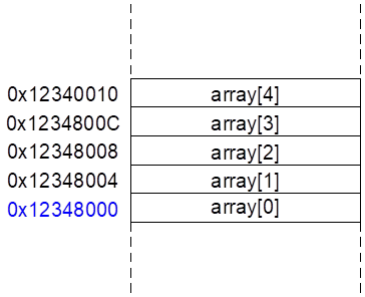

Figure 1: Representational view of array elements

An array can store a large amount of data. Storing these data in the limited number of registers (32) is infeasible. Hence, arrays are stored in the data segment of the program. Essentially, there are three different operations that can perform on the array:

- Receiving the data from the particular index of array into the variable, e.g., `x = list[i];`
- Writing the data from the variable to the particular index of the array, e.g., `list[i] = x;`
- Knowing the length of the array, i.e. `list.length`

To access the data elements in the array, you need to know the array's data address and then use the load word (`lw`) or store word (`sw`) instructions. Recall that the size of the words in the MIPS is 32 bits or 4 bytes. One of the way of declaring an array is

```
array: .word 5, 4, 2, 1, 3, 4, -2, -1, 0, 9
```

The following snippet of the code will place the value of array[4] into the `$s3`:

```
la $s2, array # Assign the address of array into $s2
li $s1, 4 # Place the element index into $s1
add $s1, $s1, $s1  # double the index
add $s1, $s1, $s1   # double the index again (now 4 times)
add $s4, $s1, $s2   # access the index element by adding into the base addresss
lw $s3, 0($s4)  # Assign the value fom the index element of the array to $s3
```

In case if you want to assign the content of `$s3` to array[4], use the following line in place of the load word instruction

```
sw $s3, 0($s4)
```

Some other ways for 1-Dimensional and 2-Dimensional array declaration

1-Dimensional Array

- Different ways to declare string array of 10 characters are as follows:

  ```
  1. my_char_array:    .byte   0:10 (intial value of each element
   in the array: Null \0)
  2. my_char_array:    .space   10
  ```

  The difference between the two declarations is that the first way initializes the memory space before the program begins execution. In contrast, the second way does not initialize memory. It allocates the 10 bytes but does not change their contents before the program begins its execution.

- In case if you need to declare the string array of 10 characters with their initial value as 'A'. There are two different ways:

  ```
  my_char_array:    .byte   'A':13

  Alternatively:

  my_char_array:    .byte   65:13 (usig ASCII Format)
  ```

- Different ways to declare integer array of size 36 are as follows:

  ```
  int_array:  .word   0:36  (intial value of each element: 0)
  int_array:  .word   2:36  (intial value of each element: 2)
  ```

  The other different way to initialize the array element with different values are given at the above code snippet.

2-Dimensional Array

- A declaration of a 2-dimensional array reduces to the allocation of the correct amount of contiguous memory. The base address identifies the first element of the first row within the first column. Consider an example of a string array of 2 by 4. The declaration is as follows:

```
char_2D_Array:    .space   8     # 2 by 4 = 8 bytes of allocated space
```

- An alternative realistic way of declaring an array is given below. Consider an example of an integer array of 5 by 6, where each element of the array is initialized to 9.

```
int_arr:    .word   9:6
.word   9:6
.word   9:6
.word   9:6
.word   9:6
```

CS214: MIPS Assembly Programming
Tutorial 8: Stacks

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

October 28, 2023

Stacks

The stack is a memory area used to save local variables. A certain chunk of main memory is reserved for the stack, called stack space/area in the MIPS machine. The following points need to be considered while working with stack in MIPS:

- The stack expands downwards in terms of memory addresses.
- The stack's top element's memory address is stored in the special purpose register, called Stack Pointer (`sp`).
- MIPS does not provide any `push` and `pop` statement. Instead, it will be explicitly handled by the MIPS assembly programmer.

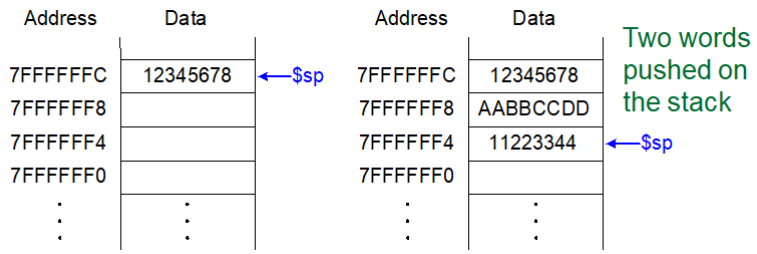

Figure 1: Representational view of Stack

Pushing an Element in the stack In order to place the data or address element in the stack, the following are the two steps that are necessary to follow:

- Progress the stack pointer, `sp` to the down to make space for the newly added element.
- Store the element into the stack.

Following are the two sample example code to push the data elements from the register `$t7` and `$t9` into the stack.

One Way:

```
sub $sp, $sp, 8
sw $t9, 4($sp)
sw $t7, 8($sp)
```

Alternate Way:

```
sw $t9, -4($sp)
sw $t7, -8($sp)
sub $sp, $sp, 8
```

Accessing and Popping Elements With stack pointer (`$sp`), you can access an element in the stack at any position (not just the top one) if and only if you know the position relative to top element `$sp`.

For example, to retrieve the value of `$t7`:

```
lw $a0, 8($sp)
```

With the above command, you can also `pop` or "wipe" the element simply by making the stack pointer upward. For example, to pop the value of `$t9` that was previously added, yielding the stack shown at the bottom:

```
addi $sp, $sp, 8
```

Note that the popped data is still present in memory, but the data past the stack pointer is considered invalid.

CS214 : MIPS Assembly Programming
Tutorial 9: Recursion and Nested Function Calls

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

November 9, 2023

Recursion

As same as the recursive relation exist, for example calculating the factorial of a given number,

```
n! = n * (n-1) * (n-2) ... *2 * 1 = n* (n-1)!
```

The recursive function exist in the programming languages. The recursive functions is a functions that calls itself repeatedly. The recursive function will keep on calling itself, with different parameters, until a terminating condition is met. It's like writing a loop. You can also write a loop to do something over and over again, until some exiting condition is met.

In the MIPS assembly programming, in the case of writing a recursive function, it is the responsibility of the MIPS programmer to save on the stack the contents of all registers relevant to the current invocation of the function before a recursive call is executed. Upon returning from a recursive function call the values saved on the stack must be restored to the relevant registers.

Nested Function Calls As same as recursive function, a similar case happens when you call a function that can call another function.

```
A: ....
   # Put B's args in $a0-$a3
   jal B # $ra = A2
A2: ....


B: ...
   # Put C's args in $a0-$a3
   # erasing B's args !
   jal C
B2: ...
   jr $ra # where does this go?

C: ...
```

```
jr $ra
```

Consider the above example, that have function `A` that calls `B`, which calls `C`.

- As observe in the above code, the arguments for the call to `C` would be placed in `$a0-$a3`, therefore overwriting the original arguments for `B`.
- Similarly, `jal C` overwrites the return address that was saved in `$ra` by the earlier `jal B`.

Register Spilling These cases incurs due to MIPS machine has a limited number of registers for use by all functions, and it's possible that several functions will require the same register for the different purposes. To handle this, we can save important register from being overwritten by a function and restore them after the function completes.

There are two possible ways to save important registers across function calls:

- The caller saves the important register as it knows which registers are important to it.
- The callee knows exactly which register it will use and accordingly it overwrites it.

The caller save the register This can be done by the caller to save the registers that it needs before the function calls. The saved register restore at the later point of time, when the control return back from the callee. However, the problem with this method is sometimes the caller does not know which registers are important for their execution. As a result, it may end up with saving large number of registers.

```
A: li $s0, 2
   li $s1, 3
   li $t0, 1
   li $t1, 2

   # Code pertaining to save the register $s0, $s1, $t0, $t1

   jal B

   # Code pertaining to restore the register $s0, $s1, $t0, $t1

   add $v0, $t0, $t1
   add $v1, $s0, $s1
   jr $ra
```

In the above example, function `A` saves the register `$s0, $s1, $t0 and $t1` before jump to the procedure `B`. However, it may be possible that the procedure `B` may not use these registers.

The callee save the register Another alternative ways is if callee save the value of register before the callee statement starts or before the register is being overwritten. The saved register is restored at the later point of time, when the callee function finishes their execution.

```
B: # Save registers
# $t0 $t1 $s0 $s2

li $t0, 2
li $t1, 7
li $s0, 1
li $s2, 8
...

# Restore registers
# $t0 $t1 $s0 $s2

jr$ra
```

For example, in the above code, function `B` uses register `$t0, $t1, $s0, $s2`. Hence, before using them, the callee procedure save the original values first. Thereafter, restore them before returning. However, as same as the case with the caller, the callee does not know what registers are important to the caller. As a result, it may again end up with saving more number of register.

The caller and callee work together To overcome the scenario that leads to saving more number of registers. MIPS machines uses conventions again to split the registers spilling chores.

- The caller is responsible for handling it **caller saved registers** by saving and restoring them. In other words, the callee may now freely modifying the following set of register, under the assumption that caller already saved them before jumps to callee.

  `$t0-$t9 $a0-$a3 $v0-$v1`

- The callee is responsible for handling it **callee saved register** by saving and restoring them. In particular, the caller now assume that these following set of registers are not altered by the called. Note that the register `$ra` is handled here carefully; as it is saved by a callee who may be caller at some point of time.

  `$s0-$s7 $ra`

Hence, with register spilling, be careful when working with nested functions, which can act as both caller and callee.

Problem 1: Write a MIPS assembly recursive function to find the determinant of a 3 x 3 matrix (it can be integer or floating point array). The address of the array M and the size N are passed to the function on the stack, and the result R is returned on the stack:

Problem 2: Write an efficient MIPS assembly language function that will scan through a string of characters with the objective of locating where all the lower case vowels appear in the string, as well as counting how many total lower case vowels appeared in a string. Vowels are the letters a, e, i, o, u. The address of the string is passed to the function on the stack, and the number of vowels found is returned on the stack. A null character terminates the string. Within this function, you need to call other function that Justifies the relative position within the string where each vowel was found. Notice this will be a nested function call (calling of a function inside the function). Here is an example string:

```
The quick brown fox.

For the above example string the output of your program would be

A Vowel was Found at Relative Position :                  3
A Vowel was Found at Relative Position :                  6
A Vowel was Found at Relative Position :                  7
A Vowel was Found at Relative Position :                 13
A Vowel was Found at Relative Position :                 18
```

```
#GCD
.data
n1: .word 20
n2: .word 15

.text
.globl main
main:
   lw $a0,n1
   lw $a1,n2
   jal GCD

   move $a0,$v0
   li $v0,1
   syscall
   li $v0, 10
   syscall

GCD:
   #GCD(n1, n2)
   # n1 = $a0
   # n2 = $a1

   addi $sp, $sp, -12
   sw $ra, 0($sp)
   sw $s0, 4($sp)
   sw $s1, 8($sp)

   move $s0, $a0
   move $s1, $a1

   li $t1, 0
```

```
    beq $s1, $t1, returnn1

    move $a0, $s1
    div $s0, $s1
    mfhi $a1

    jal GCD

exitGCD:
    lw $ra, 0 ($sp)
    lw $s0, 4 ($sp)
    lw $s1, 8 ($sp)
    addi $sp, $sp, 12
    jr $ra

returnn1:
    move $v0, $s0
    j exitGCD
```

```
#fibonacci
.data
prompt: .asciiz "Enter n: "

.text
.globl main
main:
    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall
    move $a0, $v0
    addi $a0, $a0, -1
        jal     fib
        move    $a0, $v0
        jal     print_int
        jal     print_newline

    li $v0, 10
    syscall

fib:
    addi $sp, $sp, -12
        sw      $ra, 0($sp)
        sw      $s0, 4($sp)
        sw      $s1, 8($sp)

        addi    $s0, $a0, 0
        beq     $0, $s0, done
        addi    $t0, $0, 1
```

```
        beq     $t0, $s0, done

        addi    $a0, $s0, -1
        jal     fib
        addi    $s1, $v0, 0
        addi    $a0, $s0, -2
        jal     fib
        add     $v0, $v0, $s1
        j       finish

done:
    add $v0, $0, $s0
        j       finish

finish:
    lw  $s1, 8($sp)
        lw      $s0, 4($sp)
        lw      $ra, 0($sp)
        addi    $sp, $sp, 12
        jr      $ra

print_int:
        move    $a0, $v0
        li      $v0, 1
        syscall
        jr      $ra

print_newline:
        li      $a0, '\n'
        li      $v0, 11
        syscall
        jr      $ra
```

```
#factorial
.data
    num: .word 16
    ans: .word -1
    prompt: .asciiz "Enter the number:"

.text
main:

    li $v0, 4
    la $a0, prompt
    syscall
    li $v0, 5
    syscall
    sw $v0, num

    lw $a0, num
    jal fac
    sw $s1, ans



    li $v0, 1
    lw $a0, ans
    syscall
    li $v0, 10      #khatam bc
    syscall
fac:
    subu $sp, $sp, 8
    sw $ra, ($sp)
    sw $s0, 4($sp)

    li $s1, 1
```

```
    beq $a0, $zero, done

    move $s0, $a0
    addi $a0, $a0, -1
    jal fac
    mul $s1, $s0, $s1
done:
    lw $ra, ($sp)
    lw $s0, 4($sp)
    addu $sp, $sp, 8
    jr $ra

exit:
    li $v0, 10
    syscall
```

```
.globl main
.data
new: .asciiz"\n"
int_arr1: .word 0:2
        .word 0:2
int_arr2: .word 0:2
        .word 0:2
prompt1: .asciiz "Enter first matrix numbers:"
prompt2: .asciiz "Enter second matrix numbers:"
space: .asciiz " "

.text
main:
li $v0,4
la $a0,prompt1
syscall

li $t9,0

loop1:
bgt $t9,3,prompt
li $v0,5
syscall
move $t1, $v0
sb $t1, int_arr1($t9)
lb $t2 ,int_arr1($t9)
addi $t9,$t9,1
j loop1

prompt:
li $v0,4
la $a0,prompt2
```

```
syscall

li $t9,0

loop2:
bgt $t9,3,redefine
li $v0,5
syscall
move $t1, $v0
sb $t1, int_arr2($t9)
lb $t2 ,int_arr2($t9)
addi $t9,$t9,1
j loop2

redefine:
li $t9,0
li $t8,0
addition:
bgt $t9,3,exit

; bgt $t8,1,newline
lb $t1,int_arr1($t9)
lb $t2,int_arr2($t9)
add $t3,$t1,$t2
li $v0,1
move $a0,$t3
syscall
addi $t9,$t9,1
j addition

newline:
   li $v0,4
```

```
    la $a0,new
    li $t8,0
exit:
li $v0,10
syscall
```

```
.data
input_prompt:   .asciiz "Enter a lowercase string (10 characters): "
output_prompt:  .asciiz "Uppercase string: "
lowercase_str:  .space 11
uppercase_str:  .space 11

.text
.globl main

main:
   li $v0, 4
   la $a0, input_prompt
   syscall

   li $v0, 8
   la $a0, lowercase_str
   li $a1, 11
   syscall

   la $a0, lowercase_str
   la $a1, uppercase_str
   jal convert_to_uppercase

   li $v0, 4
   la $a0, output_prompt
   syscall

   li $v0, 4
   la $a0, uppercase_str
   syscall

   li $v0, 10
```

```
    syscall

convert_to_uppercase:

    loop:
        lb $t0, ($a0)
        beqz $t0, done
        addi $a0, $a0, 1



        li $t1, 'a'
        li $t2, 'z'
        blt $t0, $t1, not_lowercase
        bgt $t0, $t2, not_lowercase



        li $t3, 32
        sub $t0, $t0, $t3




        not_lowercase:
        sb $t0, ($a1)



        addi $a1, $a1, 1
        j loop

    done:

        li $t0, 0
```

```
sb $t0, ($a1)

jr $ra
```