

# Lab Sheets 1 & 2 for CS F342 Computer Architecture

## Semester 1 : 2022-23

**Explanation:** The practical session covers installation and execution of QTSPIM software for MIPS, the basic program in MIPS (initializing variable, take input from user, addition), basic binary analysis of instructions used in MIPS architecture.

### Goal 1: Installing and launching QTSPIM

Open a terminal to run the following commands.

- Check the OS and the processor support on your Linux machine – verify whether it is 32-bit or 64-bit. [Try commands like `uname -p` ; `uname -a` ; `man uname`]
- Download the appropriate QTSPIM binary from Internet (<https://sourceforge.net/projects/spimsimulator/files/> )
- Install using dpkg tool. Do not forget to get “super user privilege” using sudo. `$ sudo dpkg -i <qtspim package>`
- Launch QTSPIM. `$ qtspim`

```
[0040001c] 3402000a ori $t2, $0, 10          ; 181: li $v0 10
[00400020] 0000000c syscall          ; 182: syscall # syscall 10 (exit)

Kernel Text Segment [00000000]..[00010000]
[00000180] 0001d921 addu $t7, $0, $t1          ; 90: move $t1 $at # Save $at
[00000184] 3c019000 lui $t1, -20672          ; 92: sw $v0 $t1 # Not re-entrant and we can't trust
$sp
[00000188] ac220200 sw $t2, 512($t1)          ; 93: sw $a0 $t2 # But we need to use these registers
[0000018c] 3c019000 lui $t1, -20672          ; 95: nfc0 $a0 $t3 # Cause register
[00000190] ac240200 sw $t4, 516($t1)          ; 96: ori $a0 $a0 2 # Extract EnoCode Field
[00000194] 001a6000 mfcc $t6, $t3          ; 97: andi $a0 $a0 0x1f
[00000198] 001a2002 ori $t4, $t6, 2          ; 101: li $v0 4 # syscall 4 (print_str)
[0000019c] 3084001f andi $t4, $t4, 31          ; 102: lw $a0 __m1_
[000001a0] 34020004 ori $t2, $0, 4          ; 103: syscall
[000001a4] 3c049000 lui $t4, -20672 [__m1_]          ; 105: li $v0 1 # syscall 1 (print_int)
[000001a8] 0000000c syscall          ; 106: ori $a0 $a0 2 # Extract EnoCode Field
[000001ac] 34020001 ori $t2, $0, 1          ; 107: andi $a0 $a0 0x1f
[000001b0] 001a2002 ori $t4, $t6, 2          ; 108: syscall
[000001b4] 3084001f andi $t4, $t4, 31          ; 110: li $v0 4 # syscall 4 (print_str)
[000001b8] 0000000c syscall          ; 111: andi $a0 $a0 0x1c
[000001bc] 34020004 ori $t2, $0, 4          ; 112: lw $a0 __except($a0)
[000001c0] 3144003c andi $t6, $t6, 60          ; 113: nop
[000001c4] 3c019000 lui $t1, -20672          ; 114: syscall
[000001c8] 00249021 addu $t1, $t1, $t4          ; 116: sne $a0 0x1f ok_pc # Red PC exception requires
[000001cc] 0c240180 lw $t4, 384($t1)          ;
[000001d0] 00000000 nop          ;
[000001d4] 0000000c syscall          ;
[000001d8] 34010018 ori $t1, $0, 24          ;
special checks

All Rights Reserved.
QTSPIM is distributed under a GNU license.
See the file README for a full copyright notice.
QTSPIM is linked to the Qt library, which is distributed under the GNU Lesser General Public License version 3 and version 2.1.
```

Note: Please refer to the following: (i) HP AppA.pdf (ii) Chapter 2 Hennessey & Patterson Book (iii) **MIPS Reference Data Card**

## ("Green sheet")

**Goal 2:** To get introduced to QTSPIM and implement some code related to - System Calls and User Input. Further, we will do basic integer Add/Sub/And/Or and their immediate flavours (e.g. ori).

Reference for MIPS assembly – refer to the **MIPS Reference Data Card ("Green sheet")** uploaded in CMS. Further, some of the QTSPIM assembly instructions are beyond this data card (e.g. the pseudo instruction **la**).

Additionally use Appendix A (HP\_AppA.pdf) from Patterson and Hennessey "Assemblers, Linkers and the SPIM Simulator" for gaining background knowledge of SPIM.

In this lab we focus on reversing only integer based

instructions (add, or, subi etc.). **Reference for**

### Registers:

0 zero constant 0	16 s0 callee saves
1 at reserved for assembler	...
2 v0 results from callee	23 s7
3 v1 returned to caller	24 t8 temporary (cont'd)
4 a0 arguments to callee	25 t9
5 a1 from caller: caller saves	26 k0 reserved for OS kernel
6 a2	27 k1
7 a3	28 gp pointer to global area
8 t0 temporary	29 sp stack pointer
...	30 fp frame pointer
15 t7	31 ra return Address caller saves

System calls as well as functions (in later part of the semester) should take care of using the registers in proper sequence. Especially take note of V0, V1 [R2, R3 in QTSPIM] and a0-a3 [R4-R7 in QTSPIM] registers.

## Reference for System Calls:

Service	Code (put in \$v0)	Arguments	Result
print_int	1	\$a0=integer	
print_float	2	\$f12=float	
print_double	3	\$f12=double	
print_string	4	\$a0=addr. of string	
read_int	5		int 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	addr in \$v0
exit	10		

## Reference for Data directives:

### **.word w1, ..., wn**

-store n 32-bit quantities in successive memory words

### **.half h1, ..., hn**

-store n 16-bit quantities in successive memory half words

### **.byte b1, ..., bn**

-store n 8-bit quantities in successive memory bytes

### **.ascii str**

-store the string in memory but do not null-terminate it

-strings are represented in double-quotes "str"

-special characters, eg. \n, \t, follow C convention

### **.asciiz str**

-store the string in memory and null-terminate it

### **.float f1, ..., fn**

-store n floating point single precision numbers in successive memory locations

### **.double d1, ..., dn**

-store n floating point double precision numbers in successive memory locations

### **.space n**

-reserves n successive bytes of space

## **Layout of Code in QTSPIM:** Typical code layout

(\* .asm file edited externally) **# objective of the program**

.data **#variable declaration follows this line**

.text **#instructions follow this line**

main: **# the starting block label**

...

xxx

yyy

zzz

.....

li \$v0,10 **#System call- 10 => Exit;**

syscall **# Tells QTSPIM to properly terminate the run**

**#end of program**

## **Class Exercises**

**Exercise 1:** Understanding Pseudo instruction.


Not all instructions used in the lab will directly map to MIPS assembly instructions. Pseudo instructions are instructions not implemented in hardware. E.g. using \$0 or \$r0 we can load constants or move values across registers using add instruction.

**E.g. li \$v0, 10 actually gets implemented by assembler as ori \$v0, \$r0, 10**

In subsequent exercises, identify the pseudo instruction by looking at the actual code used by QTSPIM.

**Exercise 2:** Integer input and output and stepping through the code. Invoking system calls to output (print) strings and and input (read) integers.

**The following code snippet prints the number 10 on console. Modify it to read any number and print it back.**

Hint: To copy it from \$v0 to \$a0, you can use add or addi with 0 or similar options. Edit the code in your editor of choice and then load it in QtSpim. Single Step [  ] through the code and look at the register values as you execute various instructions.

The demo code is given below:

# demo code to print the **integer value 10**

**.data** #variable declaration follow this line

# sample string variable declaration – not used in first exercise.

myMsg: .asciiz "Hello Enter a number." # string declaration #  
          .asciiz directive makes string null terminated

**.text** #instructions follow this line

main:

li \$a0,10

li \$v0,1

syscall

li \$v0,10 #System call - Exit - QTSPIM to properly terminate the  
run syscall

#end of program

## **Lab 2 in continuation...**

**Exercise3:** Modify the above code to output “myMsg” along with the input integer.

Hint: 1) You will use load address MIPS instruction (la \$a0, myMsg)

2) For printing string use (li \$v0 4)

**Exercise 4:** Take 2 integers as input, perform addition and subtraction between them and display the outputs. The result of addition is to be displayed as "The sum is =" and that of

subtraction is to be displayed as "The difference is =". Check if negative integers can be handled.

Hint: To read integer input from user use (li \$v0 5)

2) use store word sw or load word lw instruction to read input variables and store in register

3) Finally perform addition and store in \$a0 and print using (li \$v0 1)

*Observations: List all the pseudoinstructions used in this exercise and discuss.*

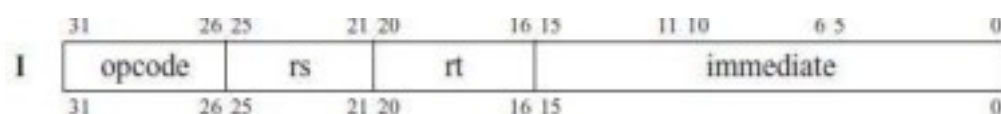
**Exercise 5:** Disassemble the binary/hex code to MIPS assembly code. Note that pseudoinstructions cannot be identified using this. For your information, a brief discussion of a sample instruction follows below.

FP Regs	nt Regs [16]	Data	Text
Int Regs [16]			Text
PC = 0			User Text Segment [00400000]..[00400000]
EPC = 0			[00400000] 8fa40000 lw \$4, 0(\$29) ; 183: lw \$a0 0(\$sp) # argc
Cause = 0			[00400004] 27a50004 addiu \$5, \$29, 4 ; 184: addiu \$a1 \$sp 4 # argv
BadVAddr = 0			[00400008] 24a60004 addiu \$6, \$5, 4 ; 185: addiu \$a2 \$a1 4 # envp
Status = 3000fff10			[0040000c] 00041080 sll \$2, \$4, 2 ; 186: sll \$v0 \$a0 2
			[00400010] 00c23021 addu \$6, \$6, \$2 ; 187: addu \$a2 \$a2 \$v0
HI = 0			[00400014] 0c100009 jal 0x00400024 [main] ; 188: jal main
LO = 0			[00400018] 00000000 ncp ; 189: ncp
			[0040001c] 3402000a ori \$2, \$0, 10 ; 191: li \$v0 10
R0 [r0] = 0			[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)
R1 [a0] = 0			[00400024] 3404000a ori \$4, \$0, 10 ; 6: li \$a0, 10
R2 [v0] = 0			[00400028] 34020001 ori \$2, \$0, 1 ; 7: li \$v0, 1
R3 [v1] = 0			[0040002c] 0000000c syscall ; 8: syscall
R4 [a0] = 1			[00400030] 34020005 ori \$2, \$0, 5 ; 10: li \$v0, 5
R5 [a1] = 7ffff1ac			[00400034] 0000000c syscall ; 11: syscall
R6 [a2] = 7ffff1b4			[00400038] 30480000 addi \$8, \$2, 0 ; 12: addi \$t0, \$v0, 4
R7 [a3] = 0			[0040003c] 3c041001 lui \$4, 4097 [str] ; 14: la \$a0, str
R8 [t0] = 0			[00400040] 34020004 ori \$2, \$0, 4 ; 15: li \$v0, 4
R9 [t1] = 0			[00400044] 0000000c syscall ; 16: syscall
R10 [t2] = 0			[00400048] 21040000 addi \$4, \$8, 0 ; 18: addi \$a0, \$t0, 0
R11 [t3] = 0			[0040004c] 34020001 ori \$2, \$0, 1 ; 19: li \$v0, 1

For code at address 0x0040 0038 – which is having a value of 0x2048 0000 when we look at opcode etc. we get:

Binary representation: 0010 0000 0100 1000 0000 0000  
0000 0000 OpCode value is: 0010 00 (8 decimal)

As per green card, OpCode 8 decimal is for **addi** (type I)



rs value is: 00 010 => 2 decimal- register v0

rt value is: 01 000 => 8 decimal – register t0

immediate value is: 0000 0000 0000 0000 => 0

Hence the instruction is **addi \$t0, \$v0, 0**

In groups, write different assembly instructions and ask your group members to reverse from hex notation.

Also as an exercise reverse the following three values:

1. 22940004

2. 00c23021

3. 34020005



## Lab Sheet 3 for CS F342 Computer Architecture

### Semester 1 – 2022-2023

#### Explanation

**Goals for the Lab:** We build up on Lab Sheet 2 and explore floating point instructions. We also use some new registers (especially those for floating point operations and remainder for integer arithmetic – HI / LO) and convert across numeric data types.

Reference for Floating point Instructions – refer to the **MIPS Reference Data sheet** (“**Green sheet**” - ARITHMETIC CORE INSTRUCTION SET – Page 1, column 2).

Some of the new instructions (and pseudo instructions) are l.s, mov.s, l.d, mov.d, cvt.d.s, cvt.s.d and similarly single precision and double precision flavours of add, sub, neg, mul, div.

#### FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
	31	26 25	21 20	16 15	11 10	6 5 0
FI	opcode	fmt	ft	immediate		
	31	26 25	21 20	16 15		

There are 32 single precision floating point registers / 16 double precision values. In QtSpim they are labelled as FG0, FG1, FG2 .. FG31 for single precision and FP0, FP2, FP4 .. FP30 for double precision.

#### Class Exercises

**Exercise 1:** Scan and Print Float.

```
.data
```

```
val1: .float 0.001
```

```
.text
```

```
main:
```

```
l.s $f12, val1
```



```

li $v0, 2

syscall

li $v0, 6
syscall

mov.s $f12, $f0
li $v0, 2 syscall

li $v0, 10
syscall

```

**Exercise 2:** Use double precision using l.d and mov.d instead of l.s and mov.s. Also use cvt.d.s and cvt.s.d to convert across precession types.

**Exercise 3:** Add/Sub of float: use instructions like add.d and add.s; sub.s, neg.s etc. Use addition along with negation to implement subtraction.

Hint: Read float value, the value will be store in \$f0

move it to any single precision floating registers \$f1 and \$f2

```
add.s $f12, $f1, $f2
```

```
sub.s $f12, $f1, $f2
```

```
neg.s $f12, $f1
```

print the float value using li \$v0, 2 syscall

**Exercise 4:** Using the below example for Integer Multiply / Divide. Extend the example to perform the mul/divide for floating number and also find remainder in division operation ?

```

.data
str0: .asciiz "\nMul:"
str1: .asciiz "\nDiv:"

```

```

w1: .word 300
w2: .word 7

.text

main:

lw $t0, w1
lw $t1, w2

la $a0, str0

li $v0, 4
syscall
mul $a0, $t0, $t1

li $v0,1 # print from $a0 syscall
la $a0, str1
li $v0, 4
syscall

div $a0, $t0, $t1

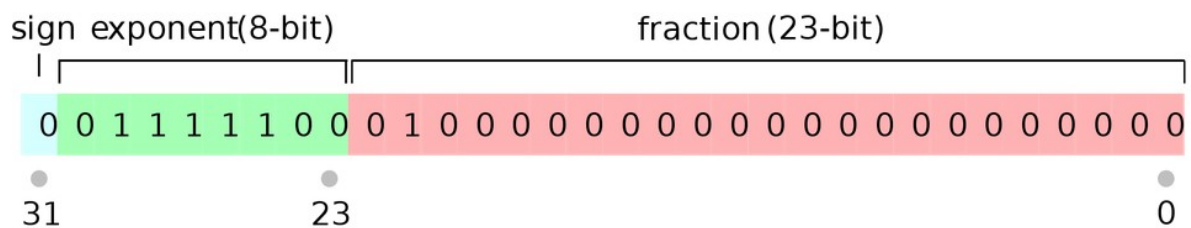
li $v0,1 # print from $a0 syscall
# Modify to also print the modulo / remainder

li $v0,10
syscall

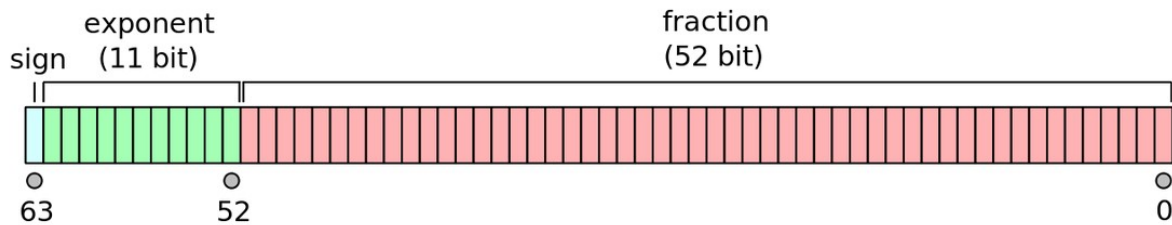
```

## Floating point representation

### IEEE-754 Single Precision



## IEEE-754 Double Precision



**Exercise 5:** Convert Integer to Single precision float and vice versa. Also compute the conversion errors for mathematical operations – e.g. rounding off during divide, precision loss during convert to float and multiple etc. Some of the new instructions are `cvt.s.w`, `cvt.w.s`, `mtc1`/`mtcz`, `mfc1`/`mfcz` ... Note that `cvt.x.w` or `cvt.w.x` (where we convert from / to integers) can only be done with registers in CP1. So if the value is in the main registers you first need to use `mtc1`. Similarly after converting to integer, you need to use `mfc1` to get the values in regular registers. Skeletal algorithm:

- A. Define two integers with relatively large values – but ensure that no overflow in multiplication
- B. multiply and print output – save integer output to a spare register
- C. Convert both integers to single precision float; multiply and print output using float [make sure to use `mtc1` first and then `cvt.s.w` or similar
- D. Convert the output to integer and compare with saved values from step B – first convert to integer in co-processor 1 register and then move the value to `a0` register of main processor.
- E. Convert both integers to double precision float; multiply and print using double precision float
- F. Convert the output to integer and compare with saved values from step B //G. Print difference in output of D and E – (may happen only if large integer values are used)

**Hint:** Take two larger integers such that the multiplication should not create arithmetic overflow i.e., with in the limit  $2^{31}-1$  and greater than  $2^{23}$

**2) Convert both the integer to single precision floating point and double precision floating point**

**3) The multiplication of the two floating point number will give precision error with respect to multiplication of two integer in single precision as single precision contains**

only 23 bits after decimal (mantissa is 23 bit long), where as in double the difference of multiplication of integer and double precision is 0

**Exercise 6:** Explore disassembly for the new instructions.

1. 44880000
2. 46800060
3. 460208c2
4. 3c041001
5. 44042000
6. 0000000c

**CS F342**  
**Computer Architecture**  
Semester 1 – 2022-2023  
Lab Sheet 5

Goals for the Lab: We build up on prior labs and explore i) load/store instructions ii) loops, arrays and string manipulations.

Data Declaration (Recap):

Format for variable name (label) declarations in .data segment:

name-or-label: storage type value(s)

- create storage for variable of specified type with given name and specified value • value(s) usually gives initial value(s).
- for storage type “.space”, gives a number of bytes to be allocated.
- Note: Name (or labels) always followed by colon (:).
- Some Examples
  - var1: .word 3 *#create a single integer variable with initial value 3*
  - list: .word 17 5 92 87 41 30 23 55 -72 36 *#an array of 10 integers*
  - array1: .byte 'a','b' *# 2- element char array- values of a and b (decimal ascii 97, 98)*
  - Array2: .space 40 *# 40 consecutive bytes, **not initialized**; could be used as a 40 element char array, or a 10- element integer array; comment should be used to clarify*

Load / Store Instructions

- RAM access only allowed with load and store instructions
- all other instructions use register operands

LOAD\_EXAMPLES:

**format:**

lw register\_dest, RAM\_source *#copy word (4 bytes) at source RAM location to destination register.*  
lb register\_dest, RAM\_source *#copy byte at source RAM location to low order byte of destination register*

STORE\_EXAMPLES:

**format:**

sw register\_source, RAM\_destination *#store word in src register into RAM dest.*  
sb register\_source, RAM\_destination *#store byte (low order) in src reg into RAM dest.*

sw \$t2, (\$t0) *#store word in register \$t2 into RAM at address contained in \$t0*  
sw \$t2, 12(\$t0) *#store word in register \$t2 into RAM at address (\$t0 12)*

swc1 \$f0, 4(\$t4) *# Mem[ \$t4 + 4 ] = \$f0; Store word(into RAM) from coprocessor 1.*  
sdc1 \$f0, 0(\$t4) *# Mem[ \$t4 + 0 ] = \$f0; Mem[ \$t4 + 4 ] = \$f1 ; Store double(into RAM) from CP 1.*

## Exercise 1: A program to take a string from a user and check whether it is a palindrome or not.

### Hint:

```
lb $t3, 0($t1) # grab the char at lower ptr  
lb $t4, 0($t2) # grab the char at upper ptr  
bne $t3, $t4, not_palin # if different, it's not
```

### Arrays :

Since we have only a small number of registers, it is infeasible to use the registers for long term storage of the array data. Hence, arrays are stored in the Data Segment of a MIPS program. Fundamentally, there are three operations which one can perform on an array:

- Getting the data from an array cell, e.g. `x = list[i]`;
- Storing data into an array cell, e.g. `list[i] = x`;
- Determining the length of an array, i.e. `list.length`.

To access the data in the array requires that we know the address of the data and then use the **load word (lw)** or **store word (sw)** instructions. Words (which is how integers are stored) in MIPS take up 32 bits or 4 bytes. Therefore, if we have a declaration such as:

```
list: .word 3, 0, 1, 2, 6, -2, 4, 7, 3, 7
```

the address that is loaded by the instruction `la $t3, list` is the address of the first '3' in the list. The address of the '0' is 4 greater than that number, and the address of the '6' is 16 greater than that number.

The following snippet of code will place the value of `list[6]` into the `$t4`:

```
la $t3, list # put address of list into $t3  
li $t2, 6 # put the index into $t2  
add $t2, $t2, $t2 # double the index  
add $t2, $t2, $t2 # double the index again (now 4x)  
add $t1, $t2, $t3 # combine the two components of the address  
lw $t4, 0($t1) # get the value from the array cell  
If we wish to assign to the contents of $t4 to list[6] instead, the last line would simply  
be: sw $t4, 0($t1) # store the value into the array cell
```

**Exercise 2: Write a program to search for a character in a given character array.**

**Hint:**

```
beq $s1, $zero, srchdn # check for terminator
seq $t1, $s1, $t0 # compare characters
```

**Exercise 3: Write a program to take string of length 5 as input from user and store its reverse string in different array and then print both the strings. Observe the values in data segment by stepping through the code. Do we need to worry about '\0' termination? Why / why not?**

**Hint: Home assignment**

```
loop:                                # do {
lb $t3, ($t1)                        #     t3 := str[i]
sb $t3, ($t2)                        #     revstr[4-i] := t3
subu $t1, $t1, 0x1                  #     str-- ;
addi $t2, $t2, 0x1                  #     revstr++ ;
subu $t0, $t0, 0x1                  #     i-- ;
bgez $t0, loop                      # } while(i >= 0);
```

**Exercise 4 : Write a program to find the maximum and minimum element in an array.**

**Hint:**

```
sw $v0, ($a0)                        #   arr[i] := x ;
addi $t1, $t1, 0x1                  #   i++ ;
addi $a0, $a0, 0x4                  #   arr++ ;
blt $t1, $t0, input_loop            # } while(i < n);
```

**Exercise 5 : Explore disassembly for the new instructions**

1. 814c0000
2. c08a0000
3. a08a0000
4. e08a0000
5. e48a0000
6. f48a0000
7. 4604103e



**References:**

- [1] Green Sheet and text book appendix.
- [2] <http://tfinley.net/notes/cps104/mips.html>
- [3] <https://www.doc.ic.ac.uk/lab/secondyear/spim/node20.html>
- [4] <https://people.cs.pitt.edu/~childers/CS0447/lectures/SlidesLab92Up.pdf>

# CS F342

## Computer Architecture

Semester 1 – 2022 - 23

### Lab Sheet 6

**Goals for the Lab:** We build up on prior labs and explore basics of functions and recursion.

**Background:**

- Calling a subroutine is between a *caller*, who makes the subroutine call, and the *callee*, which is the subroutine itself.
- The caller passes arguments to the callee by placing the values into the argument registers **\$a0-\$a3**.
- The caller calls **jal** followed by the label of the subroutine. This saves the return address in **\$ra**. The return address is PC + 4, where PC is the address of the jal instruction. If the callee uses a frame pointer, then it usually sets it to the stack pointer. The old frame pointer must be saved on the stack before this happens.
- The callee usually starts by pushing any registers it needs to save on the stack. If the callee calls a helper subroutine, then it must push \$ra on the stack. It may need to push temporary (**\$t0-\$t7**) or saved registers (**\$s0-\$s7**) as well.
- Once the subroutine is complete, the return value is placed in **\$v0-\$v1**. The callee then calls **jr \$ra** to return back to the caller.

**Exercise 1 – With Sample Code:** Study the code given below.

```
.data
    line: .asciiz "\n"
    print_str: .asciiz "value of $s0: "
    .text

    main:

        li $a0,10
        la $a1, print_str
        la $a2, line
        jal increase_the_value #function to increase the value of $s0 by 10 and print

        jal print_value #function to print value stored in $a0

        li $v0,10
        syscall
```

*# cont ... to next page*

increase\_the\_value:

addi \$sp,\$sp,-8 *#4 bytes for each value*

sw \$a0,\$sp *#call by value*

sw \$ra,4(\$sp) *#since we are having nested procedure, which will overwrite the current value of \$ra*

addi \$a0,\$a0,10

jal print\_value *#print\_value is a nested procedure*

lw \$a0,\$sp *#restore the value of \$a0, main function should get old value of \$a0*

lw \$ra,4(\$sp) *#restore the value of \$ra*

addi \$sp,\$sp,8

jr \$ra

print\_value:

addi \$sp,\$sp,-4 *#Since \$a0 will be used to print the string, its original value would be lost*

sw \$a0,\$sp *#saving the original value of \$a0(as received by this procedure)*

*#since we are not calling any other procedure in this procedure value of \$ra wouldnt change, hence no need to store it in stack*

move \$a0,\$a1

li \$v0,4

syscall

lw \$a0,\$sp

li \$v0,1

syscall

move \$a0,\$a2

li \$v0,4

syscall

lw \$a0,\$sp

addi \$sp,\$sp,4

jr \$ra

**Exercise 2:** Write a function to count the number of vowels in a given string and also return the string after removing the vowels and print that string in main function. Call the function twice with two different strings.

Input : String (without space)

Output : Single integer

**Exercise 3:** Write a program that asks if the user wants a triangle or a square. It then asks the user for the size of the object (the number of lines it takes to draw the object). The program then writes a triangle or a square of stars "\*" to the console.

```

*****
*****
*****
*****
*****
*****

```

or

```

*
**
***
****
*****
*****

```

Write a subroutine for each figure. In them, use a subroutine **print\_star\_line** that writes a line of a given number of stars. (that number is passed as an argument to **print\_star\_line** function).

### **Take home assignment:**

Print the pyramid as:

```

      *
     **
    ***
   ****
  *****
 *****
*****

```

**Exercise 4:** Find Factorial of a given integer recursively. Take care of the base case.

**Exercise 5:** Disassemble the following hex instructions.

- 02002009
- 03e00008
- 0c100013

**CS F342**  
**Computer Architecture**  
Semester 1 – 2022 – 23  
**Lab Sheet 7**

### **Goals for the Lab:**

To know exception mechanism in MIPS

Be able to write a simple exception handler for MIPS machine

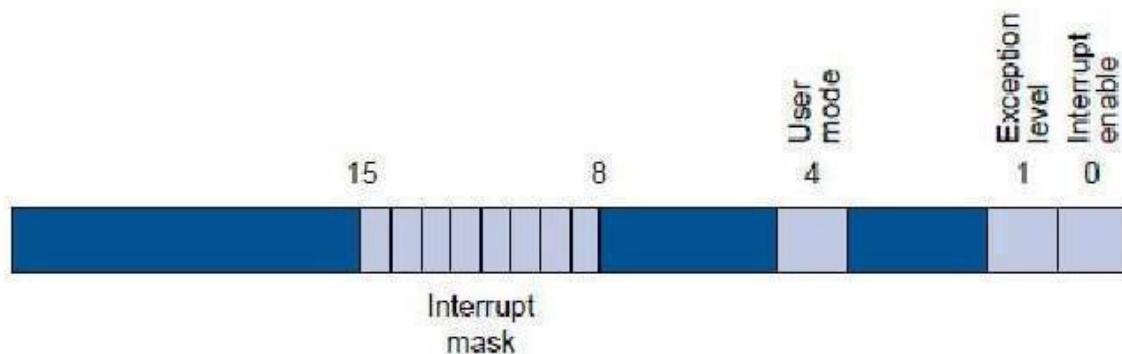
### **Background:**

The relevant registers for the exception handling, in coprocessor 0

<u>Register Name</u>	<u>Usage</u>
BadVAddr	(Bad Virtual Address) Memory address where exception occurred
Status	Interrupt mask, enable bits, and status when exception occurred
Cause	Type of exception and pending interrupt bits
EPC	Address of instruction that caused exception

**Question:** Why do we need EPC? Why can't we use RA?

#### **The Status Register:**

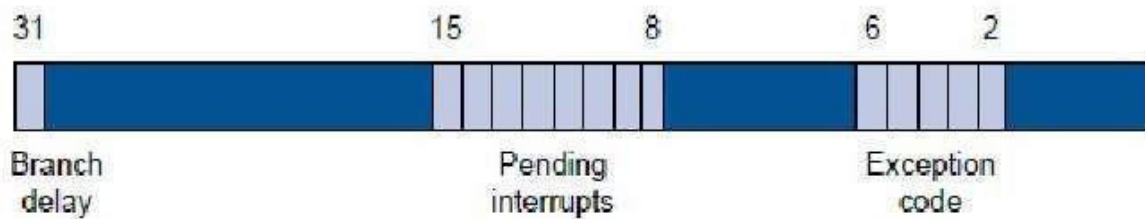


- The user mode bit is 0 if the processor is running in kernel mode and 1 if it is running in user mode.

- The exception level bit is normally 0, but is set to 1 after an exception occurs. When this bit is 1, interrupts are disabled and the EPC is not updated if another exception occurs. This bit prevents an exception handler from being disturbed by an interrupt or exception, but it should be reset when the handler finishes.

If the interrupt enable bit is 1, interrupts are allowed. If it is 0, they are disabled.

### The Cause Register:



The branch delay bit is 1 if the last exception occurred in an instruction executed in the delay slot of a branch.

The interrupt pending bits become 1 when an interrupt is raised at a given hardware or software level.

The exception code register describes the cause of an exception through the following codes:

Code	Name	Description
0	INT	Interrupt
4	ADDRL	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	<code>syscall</code> instruction executed
9	BKPT	<code>break</code> instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic overflow

a. Codes from 1 to 3 are reserved for virtual memory, (TLB exceptions), 11 is used to indicate that a particular coprocessor is missing, and codes above 12 are used for floating point exceptions or are reserved.

## **Exercise 1:**

1. Download the file containing code for causing exceptions at [https://sourceforge.net/p/spimsimulator/code/HEAD/tree/Tests/t\\_t.core.s](https://sourceforge.net/p/spimsimulator/code/HEAD/tree/Tests/t_t.core.s)
2. Load it in QtSpim, ignore any compilation warnings, Click on Run
3. Whenever exception pop-up appears, observe the values of registers used for exception handling and console message (also understand the corresponding piece of code used for causing that exception), then click 'OK'.

**Exercise 2:** Set branch delay bit as '1' by causing an exception in the delay slot of branch.

**Note:** Simulator => Settings => MIPS tab => check 'Enable delayed Branches' then run the code

**Exercise 3:** Study the code below to understand exception handling mechanism in MIPS (Non-evaluative)

*Exceptions and interrupts cause a MIPS processor to jump to a piece of code, at address 80000180hex(in the kernel, not user address space), called an exception handler. This code examines the exception's cause and jumps to an appropriate point in the operating system which then responds to it accordingly.*

*The code in the example below is a simple exception handler, which invokes a routine to print a message at each exception (but not interrupts). This code is similar to the exception handler (exceptions.s) used by the SPIM simulator.*

**.ktext** 0x80000180

*#The exception handler cannot store the old values from \$at,\$a0,\$a1) registers on the stack, as would an #ordinary routine, because the cause of the exception might have been a memory reference that used a #bad value (such as 0) in the stack pointer.*

```
mov $k1, $at      # Save $at register. $k1,$k2 are exception handler register

sw $a0, save0     # Handler is not re-entrant(i.e interrupts are disabled) and can't use
sw $a1, save1     # stack to save $a0, $a1
                  # Don't need to save $k0/$k1
```

*#the exception handler uses the value from the Cause register to test if the exception was caused by an #interrupt (see the preceding table). If so, the exception is ignored. If the exception was not an interrupt, #the #handler calls print\_exc to print a message.*

```
mfc0 $k0, $13     # Move Cause into $k0, $13 is CAUSE register
srl $a0, $k0, 2    # Extract ExcCode field(bits 2-6 in cause register)
```



```

andi $a0, $a0, 0xf
bgtz $a0, done
mov $a0, $k0
mfc0 $a1, $14
jal print_exc

done: mfc0 $k0, $14
addiu $k0, $k0, 4
mtc0 $k0, $14
mtc0 $0, $13
mfc0 $k0, $12
andi $k0, 0xffff # Clear EXL bit which allows subsequent exceptions to change the EPC register

ori $k0, 0x1
mtc0 $k0, $12

lw $a0, save0
lw $a1, save1
mov $at, $k1
eret

.kdata
save0: .word 0
save1: .word 0

```

*# Branch if ExcCode is int (0)*  
*# Move Cause into \$a0*  
*# Move EPC into \$a1*  
*# Print exception error message*  
*# Bump EPC*  
*# Do not re-execute faulting instruction*  
*# EPC*  
*# Clear Cause register*  
*# Fix Status register*  
*# Clear EXL bit which allows subsequent exceptions to change the EPC register*  
  
*# Enable interrupts*  
  
*# Restore registers*  
  
*# exception return, Return to EPC*

**Exercise 4:** Identify the cause of exception from the following values of CAUSE register

1. 0x24
2. 0x30
3. 0x1c

# CS F342

## Computer Architecture

Semester 1 – 2022 – 23

### Lab Sheet 8 & 9

**Goals for the Lab:** Build up on prior labs to further explore functions and also

1. Understand mapping of structures
2. Memory allocation using system calls (syscall 9)
3. Input and output characters (syscall 11, 12)

**Background:** We will be exploring system call 9 (sbrk) for allocating memory. We will also explore when to use temporary registers and when to save register values etc., using examples that may involve more than one return points from the function.

**Exercise 1:** Study the given code for finding factorial of an integer recursively (Also the solution to the Exercise 4 of the previous lab sheet 6)

Input: Single integer  
4  
Output: Single integer  
24

#### Pseudo Code:

```
int factorial(int input)
{
    int output = input;
    if(input > 1)
        output = input *factorial(input-1);
    return output;
}
main()
{
    printf( "Enter a number to find factorial:");
    scanf("%d", &i);
    j = factorial(i);
    printf("The result of factorial for %d is %d\n", i, j);
    exit(0);
}
```

#### MIPS Code:

```
.data
promptMessage: .asciiz "Enter a number to find it's factorial:"
resultMessage: .ascii "\nThe factorial of the given number is:"

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

```

li $v0,5    # get the number from user
syscall
move $a0, $v0
jal findFactorial    #call findFactorial function
move $s0,$v0
li $v0, 4
la $a0, resultMessage
syscall
li $v0, 1    #display the result
move $a0, $s0
syscall
li $v0,10    # exit from main
syscall

findFactorial:
subu $sp,$sp,8    #adjust stack pointer
sw $ra,0($sp)
sw $s0,4($sp)    # since the register s0 will be modified during recursion
                  # a0 is not saved, since its value is not used after return
li $v0,1          # v0 is not saved, since its value is reset before return beq
$a0,0,factDone    #the base case (input = 0) - return 1
move $s0,$a0      #find findfactorial(n-1)
sub $a0,$a0,1
jal findFactorial
mul $v0,$s0,$v0
factDone:
lw $ra,0($sp)
lw $s0,4($sp)
addu $sp,$sp,8
jr $ra

```

### Take home assignment

Write a recursive MIPS assembly program to print the nth number of Fibonacci sequence

Input : Single Integer 6

Output : Single Integer 8

### **Pseudo Code :**

```

int fib(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}

void main()
{
    int n;
    printf("Please enter a non negative integer :");
    scanf("%d",&n);
    ans=fib(n);
    printf("The %dth fibonacci number is %d.",n,ans);
    exit(0);
}

```

**New concept:** To dynamically allocate memory in MIPS use syscall named **sbrk**.

sbrk behaves much more like its namesake (the UNIX sbrk system call) than like malloc– it extends the data segment by the number of bytes requested, and then returns the location of the previous end of the data segment (which is the start of the freshly allocated memory). The problem with sbrk is that it can only be used to allocate memory, never to give it back (release / free).

In this course we may use the term allocate, but keep in mind that its actual implementation is not same as alloc / malloc.

□ To represent structures in MIPS

```
typedef struct node{
int val; //value of this node struct node * left; //pointer to left child
struct node* right; //pointer to right child
} nodeType;
```

MIPS assembly	C equivalent
After Syscall \$v0 points to 12 bytes of free memory (newly allocated) li \$a0,12 <i>//bytes to be allocated</i> li \$v0,9 syscall <i>//now \$v0 holds the address of first byte of 12 bytes of free memory</i> sw \$s0, 0(\$v0) sw \$s1, 4(\$v0) sw \$s2, 8(\$v0)  lw \$s0, 0(\$v0) lw \$s1, 4(\$v0) lw \$s2, 8(\$v0)	a,b,c,ptr are analogous to values of \$s0,\$s1,\$s2,\$v0 respectively.  node* ptr = (node*)malloc(sizeof(node));  # ptr->val = a; // \$s0 has the value # ptr->left = b; // \$s1 has left pointer # ptr->right = c; // \$s2 has right pointer  # a = ptr->val; # b = ptr->left; # c = ptr->right;

**Excercise 1:** Write a MIPS code to dynamically create an array of size N, and then find the sum of the array elements

**Excercise 2:** Write a MIPS code to create Structure to store Name, Roll no, CGPA of Students and display the details of students on console

**Exercise 3:** Complete the code given below to

1. Build an ordered binary tree T containing all the values to be sorted(Integer values)
2. Do an inorder traversal of T, printing out the values of each node.

.data

```
root: .word 0 0 0                # predefining root node as NULL
input: .asciiz "Enter numbers to insert into binary tree (0 to stop): \n"
output: .asciiz "Inorder traversal of the binary search tree: "
```

```

.text
main:
    la $a0, input
    li $v0, 4
    syscall
    li $v0, 5
    syscall                # enter first number
    beqz $v0, end_of_loop1
    la $a0, root           # load root address into $a0 for inserting
values into BST
    sw $v0, ($a0)
loop1:
    li $v0, 5             # enter subsequent numbers
    syscall
    beqz $v0, end_of_loop1 # jump out of current loop if 0 is entered
    jal insert            # call subroutine to insert into BST
    j loop1
end_of_loop1:
    la $a0, output
    li $v0, 4
    syscall
    la $a0, root          # load address of root node for inorder
function, $a0 will always contain address of tree to call inorder traversal
    jal inorder           # call inorder function
    li $v0, 10
    syscall

insert:
    move $t0, $v0
    li $a0, 12
    li $v0, 9
    syscall                # allocate space for 12/4 = 3 integers (one
for value, one for left pointer address, one for right pointer address)
    sw $t0, ($v0)          # store input value into newly created node
    sw $0, 4($v0)          # set left pointer of node to NULL
    sw $0, 8($v0)         # set right pointer of node to NULL
    la $a0, root
    # write code to insert newly created node into the BST

    jr $ra

inorder:
    beqz $a0, end_of_inorder # check if NULL node or not
    # write code to push values onto stack

    # write code to restore values from stack

    # write code to print integer

    # write code to push values onto stack

    # write code to restore values from stack

end_of_inorder:
    jr $ra

```

**Exercise 4:** Modify the above code to incorporate characters instead of integer values.

**Hint:**

- Conditions for branch instructions will change
- Size of the structure will change
- lw, sw will change to lb, sb
- refer syscall 11,12 for printing and reading chars

## CS F342 Computer Architecture

Semester 1 – 2022-23

### Lab Sheet 10 & 11

Goals for the Lab: We build up on prior labs and Exploring sorting techniques using MIPS

#### Exercise 1:

Write a program to implement C bubble sort program given below in MIPS.

##### C program code:

```
int main()
{
    int Sz = 10;
    int List[10] = {17, 5, 92, 87, 41, 10, 23, 55, 72, 36} ;
    int Stop, // $s3: upper limit for pass
    Curr, // $s0: index of current value in comparison
    Next, // $s1: index of successor to current value
    Temp; // $s2: temp storage for swap
    for (Stop = Sz-1; Stop > 0; Stop)
    {
        for (Curr = 0; Curr < Stop; Curr++)
        {
            Next = Curr + 1;
            if ( List[Curr] > List[Next] )
            {
                Temp = List[Curr];
                List[Curr] = List[Next];
                List[Next] = Temp;
            }
        }
    }
    printf("Sorted list in ascending order:\n");
    for (Curr = 0; Curr < Stop; Curr++)
        printf("%d\n", List[Curr]);
}
```

**Hint ::** To convert Curr to offset you can use sll \$t4, \$t2, 2 or similar where \$t2 is Curr, \$t4 is offset from starting address of buffer and shift of 2 implies multiplying by 4.

Partial assembly code: (Highlighted part is complete)

```
.data
list: .word 17, 5, 92, 87, 41, 10, 23, 55, 72, 36
space: .asciiz " "
.text
main:

li $s7, 10 #size of the list(sz)
addi $s3, $s7, -1 # $s3 = Stop = sz-1
```

*#Write the loop, swap code here*



```

exit:    #print the array

la $t0,list
li $t2,0 #as a counter while printing the list

print:

lw $a0,($t0)    #load current word in $a0

li $v0,1
syscall         #print the current word
la $a0,space
li $v0,4
syscall         #print space in b/w words
addi $t0,$t0,4  #point to next word
addi $t2,$t2,1  #counter++
blt $t2,$s7,print

li $v0,10
syscall

```

### **Exercise 2:**

Write a program to implement above program but store floating point numbers instead of integer.

*Hint: Use commands swc1, lwc1, c.le.s, bc1f, bc1t*

Comparison of FP values sets a code in a special register and Branch instructions jump depending on the value of the code:

```

c.le.s $f2, $f4 #if $f2 <= $f4 then code = 1 else code =
0 bc1f label #if code == 0 then jump to label bc1t label #
if code == 1 then jump to label

```

### **Exercise 3:**

Write a program to implement C Insertion sort program given below in MIPS.

**C program code:**

```

int main()
{
    int n = 5;
    int array[5] = { 5, 3, 4, 2, 1 };
    int c = 0;
    int d = 0;
    int t = 0;

    for (c = 1 ; c <= n - 1; c++) {
        d = c;
        while (d > 0 && array[d] < array[d - 1]) {
            t = array[d];

```

```

        array[d] = array[d - 1];
        array[d - 1] = t;
        d--;
    }
}

for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
}

return 0;
}

```

### **Partial assembly code:**

```

.data
array: .word 0 : 1000      # an array of word, for storing values.
size:  .word 5             # actual count of the elements in the array.

sort_prep:
    la    $t0, array       # load array to $t0.
    lw    $t1, size        # load array size to $t1.
    li    $t2, 1           # loop runner, starting from 1.

sort_xloop:
    la    $t0, array       # load array to $t0.
    bge    $t2, $t1, sort_xloop_end # while (t2 < $t1).
    move   $t3, $t2        # copy $t2 to $t3.

sort_iloop:
    la    $t0, array       # load array to $t0.
    mul    $t5, $t3, 4      # multiply $t3 with 4, and store in $t5
    add    $t0, $t0, $t5    # add the array address with $t5, which is the index multiplied with 4.
    ble    $t3, $zero, sort_iloop_end # while (t3 > 0).
    lw     $t7, 0($t0)      # load array[$t3] to $t7.
    lw     $t6, -4($t0)     # load array[$t3 - 1] to $t6.
    bge    $t7, $t6, sort_iloop_end # while (array[$t3] < array[$t3 - 1]).
    lw     $t4, 0($t0)
    sw     $t6, 0($t0)
    sw     $t4, -4($t0)
    subi   $t3, $t3, 1
    j      sort_iloop      # jump back to the beginning of the sort_iloop.

sort_iloop_end:
    addi   $t2, $t2, 1     # increment loop runner by 1.
    j      sort_xloop      # jump back to the beginning of the sort_xloop.
sort_xloop_end
:
    li     $v0, 4          # 4 = print_string syscall.
    la     $a0, sorted_array_string # load sorted_array_string to argument register $a0.
    syscall                               # issue a system call.
    li     $v0, 4          # 4 = print_string syscall.
    la     $a0, line       # load line to argument register $a0.
    syscall                               # issue a system call.
    jal    print           # call print routine.

```

#### Exercise 4:

Observe the sample code given below and Write a program to implement merge sort program in MIPS.

##### partial code:

mergesort:

```
addi    $sp, $sp, -16      # Adjust stack pointer
sw      $ra, 0($sp)        # Store the return address on the stack
sw      $a0, 4($sp)        # Store the array start address on the stack
sw      $a1, 8($sp)        # Store the array end address on the stack

sub      $t0, $a1, $a0     # Calculate the difference between the start and end address (i.e. number of elements * 4)

ble      $t0, 4, mergesortend # If the array only contains a single element, just return

srl      $t0, $t0, 3        # Divide the array size by 8 to half the number of elements (shift right 3 bits)
sll      $t0, $t0, 2        # Multiple that number by 4 to get half of the array size (shift left 2 bits)
add      $a1, $a0, $t0     # Calculate the midpoint address of the array
sw      $a1, 12($sp)       # Store the array midpoint address on the stack

jal      mergesort         # Call recursively on the first half of the array

lw      $a0, 12($sp)       # Load the midpoint address of the array from the stack
lw      $a1, 8($sp)        # Load the end address of the array from the stack

jal      mergesort         # Call recursively on the second half of the array

lw      $a0, 4($sp)        # Load the array start address from the stack
lw      $a1, 12($sp)       # Load the array midpoint address from the stack
lw      $a2, 8($sp)        # Load the array end address from the stack

jal      merge             # Merge the two array halves
```

mergesortend:

```
lw      $ra, 0($sp)        # Load the return address from the stack
addi    $sp, $sp, 16       # Adjust the stack pointer
jr      $ra                # Return
```

#### **Exercise 5:**

Write a MIPS Program to implement Quick sort (Home Work)

### Exercise 6:

Observe the sample code given below and Write a program to implement Binary search program in MIPS.

#### partial code:

```
.data
msg_inputList: .asciiz "Please enter positive numbers in ascending order and a 0 to terminate\n"
msg_searchList: .asciiz "Please enter a number to initSearch for\n"

initSearchList:
    li    $v0, 4          # syscall 4 (print_str)
    la    $a0, msg_searchList # load the search items input message
    syscall                # execute message print

    li    $s2, 0          # set search items counter to 0

searchList:
    li    $v0, 5          # syscall 5 (read_int)
    syscall                # execute int reading
    move   $t1, $v0        # move int to $t1
    blez   $v0, initSearch # start search if 0 was entered

    li    $v0, 9          # syscall 4 (sbrk)
    la    $a0, 4          # 4 bytes allocated for ints
    syscall                # execute memory allocation

    li    $t0, 4          # 4 bytes for an int
    add    $t2, $s4, $s2   # length of the list is counter1 + counter 2
    mul    $t0, $t2, $t0   # length of the input storage address space
    add    $t0, $t0, $s1   # calculate end of address spaces
    move   $s3, $t0        # store end of address space
    sw     $t1, ($t0)      # store input on the heap
    addi   $s2, $s2, 1     # counter++

    j      searchList     # take next input

initSearch:
    move   $t6, $s5        # store end address of input items
    move   $t7, $s3        # store end address of search items

search:
    move   $t5, $s5        # store end address of input items
    beq    $t7, $t6, exit  # if there's nothing to search, exit
```

### Exercise 7:

Observe the sample code given below and Write a program to implement Heap sort program in MIPS.

#### partial code:

```
.text
.globl main
main:

    la $a0, array                # a0 = &array

    la $t0, size lw $a1, 0($t0)
                                # a1 = size(array)

    jal heapsort                # print the array
    move $t0, $a0
    add $t1, $zero, $zero

heapsort:                      # a0 = &array, a1 = size(array)
    addi $sp, $sp, -12
    sw $a1, 0($sp)             # save size
    sw $a2, 4($sp)             # save a2
    sw $ra, 8($sp)             # save return address

heapsort_loop:                # swap(array[0],array[n])
    lw $t0, 0($a0)
    sll $t1, $a2, 2            #t1 = bytes(n)
    add $t1, $t1, $a0
    lw $t2, 0($t1)
    sw $t0, 0($t1)
    sw $t2, 0($a0)

    addi $a2, $a2, -1 jal bubble_down                # n--
                                                    # a0 = &array, a1 = 0, a2 = n

    bnez $a2, heapsort_loop
make_heap:                    # a0 = &array, a1 = size
    addi $sp, $sp, -12
    sw $a1, 0($sp)
    sw $a2, 4($sp)
    sw $ra, 8($sp)

    addi $a2, $a1, -1 # a2 = size - 1

    addi $a1, $a1, -1 srl $a1, $a1, 1                # start_index = size - 1
                                                    # start_index /= 2

    blt $a1, $zero, end_make_heap # if(start_index < 0) return
make_heap_loop:
    jal bubble_down # a0 = &array, a1 = start_index, a2 = size-1 addi
    $a1, $a1, -1
    ble $zero, $a1, make_heap_loop
```

### **Exercise 8:**

Write a program to implement C Selection sort program given below in MIPS.

#### **C program code:**

```
int main() {  
    int arr[10]={6,12,0,18,11,99,55,45,34,2};  
    int n=10;  
    int i, j, position, swap;  
    for (i = 0; i < (n - 1); i++) {  
        position = i;  
        for (j = i + 1; j < n; j++) {  
            if (arr[position] > arr[j])  
                position = j;  
        }  
        if (position != i) {  
            swap = arr[i];  
            arr[i] = arr[position];  
            arr[position] = swap;  
        }  
    }  
    for (i = 0; i < n; i++)  
        printf("%d\t", arr[i]);  
    return 0;  
}
```