

Lab Sheet 2 for CS F342 Computer Architecture Semester 1 : 2023-24

Explanation: the basic program in MIPS (initializing variable, take input from user, addition), basic binary analysis of instructions used in MIPS architecture.

Goal : 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: 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 2: 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 memory.

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

4) you can also use move ins to move between registers instead of mem location.

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

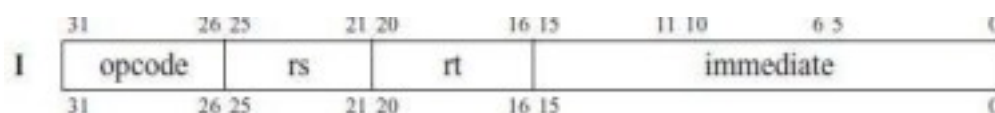
Exercise 3: Disassemble the binary/hex code to MIPS assembly code. Note that pseudoinstructions cannot be identified using this. For your information, a brief discussion of a sample instruction follows below.

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(\$ap) # argc
Cause = 0			[00400004] 27a30004 addiu \$5, \$29, 4 ; 184: addiu \$a1 \$ap 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] 0e100009 jal 0x00400024 [main] ; 188: jal main
LO = 0			[00400018] 00000000 nop ; 189: nop
			[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] 34020003 ori \$2, \$0, 3 ; 10: li \$v0, 3
R5 [a1] = 7ffff1ac			[00400034] 0000000c syscall ; 11: syscall
R6 [a2] = 7ffff1b4			[00400038] 30440000 addi \$8, \$2, 0 ; 12: addi \$t0, \$v0, 0
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 break into 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