Problem Set – 3 Solutions

CS 230, Spring 2023

Questions

1. Consider the following MIPS program:

```
1.  fun:
2.      bne $a1,$zero,lbl
3.      li $v0,1
4.      jr $ra
5.  lbl:
6.      addi $sp,$sp,-4
7.      sw $ra,0($sp)
8.      addi $a1,$a1,-1
9.      jal fun
10.     mul $v0,$v0,$a0
11.     lw $ra,0($sp)
12.     addi $sp,4
13.     jr $ra
14. main:
15.     li $t0,2
16.     li $t1,4
17.     move $a0,$t0
18.     move $a1,$t1
19.     jal fun
```

If the program execution begins from main and the initial value of $sp is 0xf064. Then list all the possible values of $sp at:

a. line number 3.
b. line number 8.
c. line number 13.

Tracing the sequence of execution:

Code line	Action	Register Values after Action	Call Frame
`15. li $t0,2`	Set register $t0 = 2	`t0 = 2` `sp = 0xf064`	main
`16. li $t1,4`	Set register $t1 = 4	`t0 = 2, t1 = 4` `sp = 0xf064`	main

`17. move $a0,$t0`	Set register $a0 = $t0	`t0 = 2, t1 = 4`
		`a0 = 2`
		`sp = 0xf064`
			main

`18.  move $a1,$t1`	Set register $a1 = $t1	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 4`
		`sp = 0xf064`
			main

`19.  jal fun`	Jump and link to the label fun. Moves execution to line 2.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 4`
		`sp = 0xf064`
			main$_{19}$->fun

`2.   bne $a1,$zero,lbl`	If $a1 is not zero, then goto line 6 (lbl). Since $a1 is 4, execution goes to line 6.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 4`
		`sp = 0xf064`
			main$_{19}$->fun

`6.   addi $sp,$sp,-4`	Add -4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 4`
		`sp = 0xf060`
			main$_{19}$->fun

`7.   sw $ra,0($sp)`	Store the value of register $ra in memory address stored in $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 4`
		`sp = 0xf060`
			main$_{19}$->fun

`8.   addi $a1,$a1,-1`	Add -1 to register $a1	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 3`
		`sp = 0xf060`
			main$_{19}$->fun

`9.   jal fun`	Jump and link to the label fun. Moves execution to line 2.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 3`
		`sp = 0xf060`
			main$_{19}$->fun$_{9}$->fun

`2.   bne $a1,$zero,lbl`	If $a1 is not zero, then goto line 6 (lbl). Since $a1 is 3, execution goes to line 6.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 3`
		`sp = 0xf060`
			main$_{19}$->fun$_{9}$->fun

`6.   addi $sp,$sp,-4`	Add -4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 3`
		`sp = 0xf05c`
			main$_{19}$->fun$_{9}$->fun

`7.   sw $ra,0($sp)`	Store the value of register $ra in memory address stored in $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 3`
		`sp = 0xf05c`
			main$_{19}$->fun$_{9}$->fun

`8.   addi $a1,$a1,-1`	Add -1 to register $a1	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 2`
		`sp = 0xf05c`
			main$_{19}$->fun$_{9}$->fun

`9.   jal fun`	Jump and link to the label fun. Moves execution to line 2.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 2`
		`sp = 0xf05c`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun

`2.   bne $a1,$zero,lbl`	If $a1 is not zero, then goto line 6 (lbl). Since $a1 is 2, execution goes to line 6.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 2`
		`sp = 0xf05c`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun

`6.   addi $sp,$sp,-4`	Add -4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 2`
		`sp = 0xf058`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun

`7.   sw $ra,0($sp)`	Store the value of register $ra in memory address stored in $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 2`
		`sp = 0xf058`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun

`8.   addi $a1,$a1,-1`	Add -1 to register $a1	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 1`
		`sp = 0xf058`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun

`9.   jal fun`	Jump and link to the label fun. Moves execution to line 2.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 1`
		`sp = 0xf058`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun$_{9}$->fun

`2.   bne $a1,$zero,lbl`	If $a1 is not zero, then goto line 6 (lbl). Since $a1 is 1, execution goes to line 6.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 1`
		`sp = 0xf058`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun$_{9}$->fun

`6.   addi $sp,$sp,-4`	Add -4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 1`
		`sp = 0xf054`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun$_{9}$->fun

`7.   sw $ra,0($sp)`	Store the value of register $ra in memory address stored in $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 1`
		`sp = 0xf054`
			main$_{19}$->fun$_{9}$->fun$_{9}$
			->fun$_{9}$->fun

`8.   addi $a1,$a1,-1`	Add -1 to register $a1	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`sp = 0xf054`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$->fun

`9.   jal fun`	Jump and link to the label fun. Moves execution to line 2.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`sp = 0xf054`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$-> fun$_9$->fun

`2.   bne $a1,$zero,lbl`	If $a1 is not zero, then goto line 6 (lbl). Since $a1 is 0, branch is not taken. Next instruction in sequence is executed.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`sp = 0xf054`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$-> fun$_9$->fun

`3.   li $v0,1`	Set register $v0 = 1	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 1`
		`sp = 0xf054`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$-> fun$_9$->fun

`4.   jr $ra`	Return execution to previous call frame. Moves execution to line 10.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 1`
		`sp = 0xf054`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$->fun

`10.  mul $v0,$v0,$a0`	Multiply $v0 by $a0 and store in $v0	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 2`
		`sp = 0xf054`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$->fun

`11.  lw $ra,0($sp)`	Restore return address of previous call frame from stack	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 2`
		`sp = 0xf054`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$->fun

`12.  addi $sp,4`	Add 4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 2`
		`sp = 0xf058`
			main$_{19}$->fun$_9$->fun$_9$
			->fun$_9$->fun

`13.  jr $ra`	Return execution to previous call frame. Moves execution to line 10.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 2`
		`sp = 0xf058`
			main$_{19}$->fun$_9$->fun$_9$
			->fun

`10.  mul $v0,$v0,$a0`	Multiply $v0 by $a0 and store in $v0	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 4`
		`sp = 0xf058`
			main$_{19}$->fun$_9$->fun$_9$
			->fun

`11.  lw $ra,0($sp)`	Restore return address of previous call frame from stack	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 4`
		`sp = 0xf058`
			main$_{19}$->fun$_9$->fun$_9$
			->fun

`12.  addi $sp,4`	Add 4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 4`
		`sp = 0xf05c`
			main$_{19}$->fun$_9$->fun$_9$
			->fun

`13.  jr $ra`	Return execution to previous call frame. Moves execution to line 10.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 4`
		`sp = 0xf05c`
			main$_{19}$->fun$_9$->fun

`10.  mul $v0,$v0,$a0`	Multiply $v0 by $a0 and store in $v0	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 8`
		`sp = 0xf05c`
			main$_{19}$->fun$_9$->fun

`11.  lw $ra,0($sp)`	Restore return address of previous call frame from stack	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 8`
		`sp = 0xf05c`
			main$_{19}$->fun$_9$->fun

`12.  addi $sp,4`	Add 4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 8`
		`sp = 0xf060`
			main$_{19}$->fun$_9$->fun

`13.  jr $ra`	Return execution to previous call frame. Moves execution to line 10.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 8`
		`sp = 0xf060`
			main$_{19}$->fun

`10.  mul $v0,$v0,$a0`	Multiply $v0 by $a0 and store in $v0	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 16`
		`sp = 0xf060`
			main$_{19}$->fun

`11.  lw $ra,0($sp)`	Restore return address of previous call frame from stack	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 16`
		`sp = 0xf060`
			main$_{19}$->fun

`12.  addi $sp,4`	Add 4 to register $sp.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 16`
		`sp = 0xf064`
			main$_{19}$->fun

`13.  jr $ra`	Return execution to previous call frame. There is no more code line in main, so execution is halted.	`t0 = 2, t1 = 4`
		`a0 = 2, a1 = 0`
		`v0 = 16`
		`sp = 0xf064`
			main

All possible $sp values at a given line can be retrieved by looking at above table.

a. **line number 3:** `0xf054`
b. **line number 8:** `0xf060, 0xf05c, 0xf058, 0xf054`
c. **line number 13**: `0xf058, 0xf05c, 0xf060, 0xf064`

2. Consider the previous question (Q1). For all lines of the program, specify the following:

    a. The type of Instruction.
    b. The addressing mode of the Instruction.

Types of Instructions:

- **R-type:** These are the instructions which use only registers for all operations. Except **sll**, **srl** & **sra** which use 5 bit value for the shift value.
- **I-type:** These instructions include 16 bit immediate values.
- **J-type:** These instructions are used to jump to different locations in the code using 26 bit address.

Type	31 26	25 21	20 16	15 11	10 06	05 00
R-Type	opcode	$rs	$rt	$rd	shamt	funct
I-Type	opcode	$rs	$rt	imm		
J-Type	opcode	address				

Addressing Modes:

- **Immediate:** Use an immediate value encoded within in the instruction.

- **Register:** Use values from registers.
- **Base:** Base memory address is retrieved from a register. Final memory address is calculated by adding offset to the base address. The final address is used to access memory.
- **PC-relative:** An offset is added to the program counter value to find the target address.
- **Pseudodirect:** Part of the program counter and an specified address is concatenated to calculate the target address.

Code line	**Instruction Type**	**Addressing Mode**
`2. bne $a1,$zero,lbl`	I-type (lbl is replaced by a 16-bit immediate value by assembler)	PC-relative
`3. li $v0,1`	I-type	Immediate
`4. jr $ra`	R-type (return address is stored within a register, while J-type instructions need a 26-bit immediate pseudo-absolute address)	Register
`6. addi $sp,$sp,-4`	I-type	Immediate
`7. sw $ra,0($sp)`	I-type (Offset is an immediate value)	Base
`8. addi $a1,$a1,-1`	I-type	Immediate
`9. jal fun`	J-type	Pseudodirect (target address is calculated by concatenating upper 4 bits of (PC + 4) and the specified address.
`10. mul $v0,$v0,$a0`	R-type	Register
`11. lw $ra,0($sp)`	I-type	Base
`12. addi $sp,4`	I-type	Immediate
`13. jr $ra`	R-type	Register
`15. li $t0,2`	I-type	Immediate
`16. li $t1,4`	I-type	Immediate
`17. move $a0,$t0`	R-type	Register
`18. move $a1,$t1`	R-type	Register
`19. jal fun`	J-type	Pseudodirect

3. Suppose there is a system which has 32 bit instructions and 32 general purpose registers. Answer the following questions:
   a. What is the number of bits required for the registers?
   b. Is it possible to have various operations consisting of 20 three-address instructions, 22 two- address instructions and 10 one-address instructions?

a. There are 32 registers. We need $\lceil log_2(32) \rceil$ = **5 bits** to label each register.

b.

Three Address Instructions
There are 3 5-bit registers and 20 operations. Therefore, we have $\mathbf{20 * 2^{15}}$ combinations.

Two Address Instructions
There are 2 5-bit registers and 22 operations. Therefore, we have $\mathbf{22 * 2^{10}}$ combinations.

One Address Instructions
There is 1 5-bit register and 10 operations. Therefore, we have $\mathbf{10 * 2^{5}}$ combinations.

Total number of combinations $= (\mathbf{20 * 2^{15}}) + (\mathbf{22 * 2^{10}}) + (\mathbf{10 * 2^{5}})$
$= \mathbf{10599 * 2^{6}}$

Total bits needed to represent all combinations = $\lceil log_2(\mathbf{10599 * 2^{6}}) \rceil = 20$

Since we have 32 bits instructions, it is possible to encode all operations.

4. Consider the following code segment:

```
int a, b, c, d, e, f, g, h, i, j, k, l, m, n, o;
a = b * c;
d = a - e;
f = d + g;
h = f * i;
k = h - l;
m = k + n;
o = m * k;
```

   a. If you are using temporary registers of MIPS ISA for the code segment, will there be any spill to the memory?
   b. If you are allowed to change the number of temporary registers, then what is the minimum number of registers required so that there is no spill to the memory for the given code segment?

At a program point a variable is live, if it's value will be used at later point in the program without redefinition.

Code line	**Live Variables**
	-
`int a, b, c, d, e, f, g, h, i, j, k, l, m, n, o;`	

	b, c, e, i, g, n, l
a = b * c;	
	a, e, i, g, n, l
d = a - e;	
	d, i, g, n, l
f = d + g;	
	f, i, n, l
h = f * i;	
	n, h, l
k = h - l;	
	k, n
m = k + n;	
	m, k
o = m * k;	
	-

a. Before line 2, 7 variables are live. We need 7 registers to store value of each variable. We also need an additional register for variable '*a*' to store the value of the expression *b**c*. Therefore, **8 registers** are needed. The compiler can reuse registers, assigned to a variable if it is not live. Since MIPS has 10 temporary registers, there will be no spill to memory.

b. The minimum number of registers required is 8. But MIPS has 10 temporary registers so no change is needed.

5. Consider the following program, in which the execution starts from main:

```
fun3:
    move $v0, $ra
    jr $ra
fun2:
    addi $sp, $sp, -4
    sw $ra, ($sp)
    move $t0, $a0
    jal fun3
    move $t1, $v0
    mul $v0, $t0, $t0
    lw $ra, ($sp)
    addi $sp, $sp, 4
    addi $t1, $t1, 0x4c
    sw $t1, ($sp)
    jr $ra
fun1:
    addi $sp, $sp, -4
    sw $ra, ($sp)
    jal fun2
    addi $v0, $v0, 1
```

```
21.     lw $ra, ($sp)
22.     addi $sp, $sp, 4
23.     jr $ra
24. main:
25.     li $t0, 5
26.     li $s1, 0
27.     move $a0, $t0
28.     jal fun1
29.     move $s1, $v0
30.     addi $t2, $s1, 2
31.     nop
```

What will be the value in the register `$t2` after program execution?

Suppose starting stack pointer address is x.

Code line	Action	Register Values after Action	Stack
`25. li $t0, 5`	Set register $t0 = 5	`t0 = 5`	[]

`26. li $s1, 0`	Set register $s1 = 0	`t0 = 5`
		`s1 = 0`
		`sp = x`
			[]

`27. move $a0, $t0`	Set register $a0 = $t0	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`sp = x`
			[]

`28. jal fun1`	Save address of line 29 instruction in $ra and jump to line 17.	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`ra = Addr.line 29`
		`sp = x`
			[]

`17. addi $sp, $sp, -4`	Decrement stack pointer by 4.	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`ra = Addr.line 29`
		`sp = x-4`
			[]

`18. sw $ra, ($sp)`	Store value of $ra (addr of line 29) in stack.	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`ra = Addr.line 29`
		`sp = x-4`
			[Addr.line.29]

`19. jal fun2`	Save address of line 20 instruction in $ra and jump to line 5.	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`ra = Addr.line 20`
		`sp = x-4`
			[Addr.line.29]

`5. addi $sp, $sp, -4`	Decrement stack pointer by 4.	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`ra = Addr.line 20`
		`sp = x-8`
			[Addr.line.29]

`6. sw $ra, ($sp)`	Store value of $ra (addr of line 20) in stack.	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`ra = Addr.line 20`
		`sp = x-8`
			[Addr.line.29, Addr.line.20]

`7. move $t0, $a0`	Set register $t0 = $a0	`t0 = 5`
		`s1 = 0`
		`a0 = 5`
		`ra = Addr.line 20`
			[Addr.line.29, Addr.line.20]

		sp = x-8	
8. jal fun3	Save address of line 9 instruction in $ra and jump to line 2.	t0 = 5 s1 = 0 a0 = 5 ra = Addr.line 9 sp = x-8	[Addr.line.29, Addr.line.20]
2. move $v0, $ra	Set register $v0 = $ra	t0 = 5 s1 = 0 a0 = 5 v0 = Addr.line 9 ra = Addr.line 9 sp = x-8	[Addr.line.29, Addr.line.20]
3. jr $ra	Jump to the address stored in $ra (line 9)	t0 = 5 s1 = 0 a0 = 5 v0 = Addr.line 9 ra = Addr.line 9 sp = x-8	[Addr.line.29, Addr.line.20]
9. move $t1, $v0	Set register $t1 = $v0	t0 = 5 t1 = Addr.line 9 s1 = 0 a0 = 5 v0 = Addr.line 9 ra = Addr.line 9 sp = x-8	[Addr.line.29, Addr.line.20]
10. mul $v0, $t0, $t0	Set $v0 = $t0 * $t0 (5 * 5)	t0 = 5 t1 = Addr.line 9 s1 = 0 a0 = 5 v0 = 25 ra = Addr.line 9 sp = x-8	[Addr.line.29, Addr.line.20]
11. lw $ra, ($sp)	Load address from stack and store in $ra	t0 = 5 t1 = Addr.line 9 s1 = 0 a0 = 5 v0 = 25 ra = Addr.line 20 sp = x-8	[Addr.line.29, Addr.line.20]
12. addi $sp, $sp, 4	Increment stack pointer by 4.	t0 = 5 t1 = Addr.line 9 s1 = 0 a0 = 5 v0 = 25 ra = Addr.line 20 sp = x-4	[Addr.line.29]
13. addi $t1, $t1, 0x4c	Add 0x4c to $t1. 0x4c is 76 in decimal. Each instruction in mips is of 4 bytes. 76/4 = 19. So, after the addition, $t1 stores the address of 19th instruction from line 9 which is line 30. Note that lines with labels are not considered.	t0 = 5 t1 = Addr.line 30 s1 = 0 a0 = 5 v0 = 25 ra = Addr.line 20 sp = x-4	[Addr.line.29]
14. sw $t1, ($sp)	Store $ta in stack.	t0 = 5 t1 = Addr.line 30 s1 = 0 a0 = 5 v0 = 25 ra = Addr.line 20 sp = x-4	[Addr.line.30]
15. jr $ra	Jump to the address stored in $ra (line 20)	t0 = 5 t1 = Addr.line 30 s1 = 0 a0 = 5 v0 = 25 ra = Addr.line 20 sp = x-4	[Addr.line.30]
20. addi $v0, $v0, 1	Increment $v0 by 1	t0 = 5	[Addr.line.30]

		t1 = Addr.line 30 s1 = 0 a0 = 5 v0 = 26 ra = Addr.line 20 sp = x-4	
`21.  lw $ra, ($sp)`	Load address from stack and store in $ra	t0 = 5 t1 = Addr.line 30 s1 = 0 a0 = 5 v0 = 26 ra = Addr.line 30 sp = x-4	[Addr.line.30]
`22.  addi $sp, $sp, 4`	Increment stack pointer by 4.	t0 = 5 t1 = Addr.line 30 s1 = 0 a0 = 5 v0 = 26 ra = Addr.line 30 sp = x	[]
`23.  jr $ra`	Jump to the address stored in $ra (line 30)	t0 = 5 t1 = Addr.line 30 s1 = 0 a0 = 5 v0 = 26 ra = Addr.line 30 sp = x	[]
`30.  addi $t2, $s1, 2`	Add 2 to $s1 and store in $t2	t0 = 5 t1 = Addr.line 30 t2 = 2 s1 = 0 a0 = 5 v0 = 26 ra = Addr.line 30 sp = x	[]
`31.  nop`	Nada	t0 = 5 t1 = Addr.line 30 **t2 = 2** s1 = 0 a0 = 5 v0 = 26 ra = Addr.line 30 sp = x	[]

The value in **`$t2` is 2**.

6. Consider the following program:

```
li $t0, 0xdecadead
li $t1, 0x00000bed
add $t3, $t0, $t1
sw $t0, 65($t3)
lb $s3, 66($t3)
lb $s4, 64($t3)
```

What will be the value of the register s3 and s4 (with explanations), if the computer is:

a. big-endian
b. little-endian

Figure 1 shows how big-endian and little-endian machines store the value `0xdecadead` in memory word `0xdecaea9a`. After the load byte instruction, `lb $s3, 66($t3)`, and `lb $s4, 64($t3)` `$s3` would contain `0x000000ca` on a big-endian system and `0x000000de` on a little-endian system. Whereas the register `$s4` will contain garbage values.

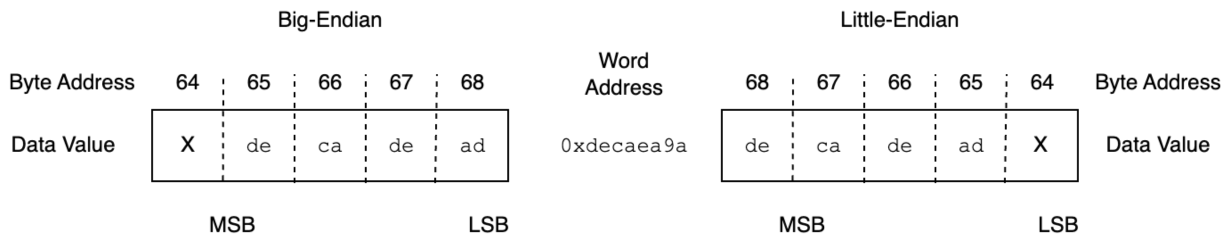

Figure 1