Computer organization

Lab2 Assembly language-MIPS(1)

Data Details

TOPIC

- Data Processing Details
  - storage
  - data transfer
  - address of the target unit in the memory
  - value of Data(relate to the defination and usage)
  - data storage in memory and access
    - Big-endian vs Little-endian
- Practice
  - p1-1,p1-2,p1-3, p2-1,p2-2,p2-3.

Assembly Language based on MIPS

name

- **Data declaration**
  - Data declaration section starts with "**. data**".
  - The declaration means **a piece of memory is required to be allocated**. The declaration usually includes **lable** (name of address on this meomory unit), **size**(optional), and **initial value**(optional).
- **Code definition**
  - Code definition starts with "**.text**", includes **basic instructions, extended instructions, labels of the code**(optional). At the end of the code, "**exit**" system service should be called.
- **Comments:**
  - Comments start from "#" till the end of current line

```
.data
        s1:      .ascii    "Welcome "
        sid:     .space   9
        e1:      .asciiz   " to MIPS World"
.text
main:
        li $v0,8        #to get a string
        la $a0,sid
        li $a1,9
        syscall

        #....

        li $v0,4        #to print a string
        la $a0,s1
        syscall

        li $v0,10       #to exit
        syscall
```

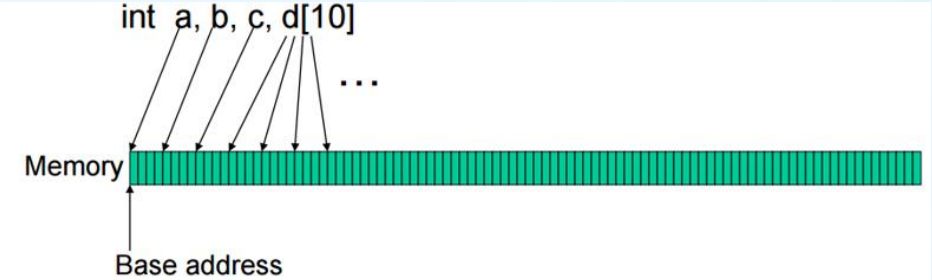
Data storage

- Unit Conversion
  - 1 word = 32bit = 2*half word(2*16bit) = 4* byte(4*8bit)
  - 1 double word = 2 word = 64bit
- Data Storage:
  - Instruction

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - Reg
    - Registers are small storage areas used to store data in the CPU, which are used to temporarily store the data and results involved in the operation.All MIPS **arithmetic** instructions MUST operate on registers.The **size of registers** in MIPS32 is **32 bits.**
  - Memory




```
//in Java， C， Phython
a = b + 1

# in MIPS
lw $t0, b      #from memory to register

addi $t1, $t0, 1

sw $t1, a      #from register to memory
```

MIPS Instruction： Load & Store

- In MIPS
  - Access the data in **memory** could ONLY be invoked by two types of instruction: **load** and **store**.
  - All the **calculation** are based on the data in **Registers**.

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Load (Load to Register)

```
lw          register_destination, RAM_source
                                   # copy word (4 bytes) at
                                   # source_RAM location
                                   # to destination register.
                                   # load word -> lw

lb          register_destination, RAM_source
                                   # copy byte at source RAM
                                   # location to low-order byte of
                                    # destination register,
                                    # and sign -e.g. tend to
                                    # higher-order bytes
                                    # load byte -> lb

 li          register_destination, value
                                    #load immediate value into
                                    #destination register
                                    #load immediate --> li
```

"la" (load address) is a extended (presudo) instruction, which is implemented by two basic instructions: **lui**(load upper immediate), **ori**(bitwise OR immediate).

Labels

Label	Address ▲
mips1.asm	
s1	0x10010000
sid	0x10010008
e1	0x10010010

Basic		
addiu $2, $0, 0x00000008	6:	li $v0, 8
lui $1, 0x00001001	7:	la $a0, sid
ori $4, $1, 0x00000008		

Store (Store to Memory)

```
sw          register_source, RAM_destination
                                  #store word in source register
                                  # into RAM destination

sb          register_source, RAM_destination
                                  #store byte (low-order) in
                                  #source register into RAM
                                  #destination
```

Q: Is there any need to implement the “sa” instruction(store address), why ? If need to implement “sa”, how to do it ?

The Address of the Target Unit in the Memory(1)

- The **"label"**
  - The value of "label" is determined by the Assembler according to the assembly source code.

```
.data
	s1:	.ascii "Welcome "
	sid:	.space 8
	e1:	.asciiz " to MIPS World"
```

Labels

Label	Address ▲
mips1.asm	
s1	0x10010000
sid	0x10010008
e1	0x10010010

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	c l e W	e m o	\0 \0 \0 \0	\0 \0 \0 \0	o t	S P I M	r o W	\0 \0 d l

e.g: la $a0, sid

Basic		
addiu $2, $0, 0x00000008	6:	li $v0, 8
lui $1, 0x00001001	7:	la $a0, sid
ori $4, $1, 0x00000008		

The Address of the Target Unit in the Memory(2)

- The address need to be got from the **Register**(Using the content in register as address).
  - **Load** the word **from the memory unit** whose address is in the register "t0" **to the register** "t2". `lw $t2, ($t0)`
  - **Store** the word **from the register** "t2" **to the memory unit** whose address is in the register "t0". `sw $t2,($t0)`
- **The address need to be caculated by Baseline + offset**(Using the sum of the baseline address and offset as address).
  - Load the word from the memory unit whose **address is the sum of 4 and the value in register "t0"** to the register "t2". `lw $t2, 4($t0)`
  - Store the word in register "t2" to the memory unit whose **address is the sum of -12 and the value in the register "t0"**. `sw $t2,-12($t0)`

Practice 1

Use MIPS to program and realize the following functions on Mars: Using 2 syscall to get the sid which has 8 numbers from input, print out the string:
Welcome XXXXXXXX to MIPS World
(XXXXXXXX is an 8-digit number)

1-1. complete the code on the right hand, move the string " to MIPS World" from the memory unit addressed by "e1" to the memory unit addressed by the sum of 8 and "sid".
1-2. Is there any other way to implement the function
1-3. Which one would get better performance:
1-1 or 1-2 ?

Tips:
1. While get and put string by syscall, the end of string is "\0" which means get a string would add a "\0" at the end of string, print a string would end with "\0"
2. The difference between "ascii" and "asciiz" is that "asciiz" would add "\0" at the end of the string while "ascii" would not.

```
.data
        s1:     .ascii   "Welcome "
        sid:    .space  9
        e1:     .asciiz  " to MIPS World"
.text
main:
        li $v0,8        #to get a string
        la $a0,sid
        li $a1,9
        syscall

#complete code here

        li $v0,4        #to print a string
        la $a0,s1
        syscall


        li $v0,10       #to exit
        syscall
```

The value of Data（1）relate to the defination

```
name:      storage_type    value(s)
```

```
example

var1:      .word   3       # create a single integer:
                           #variable with initial value 3

array1:    .byte   'a','b' # create a 2-element character
                           # array with elements initialized:
                           # to  a  and  b

array2:    .space  40      # allocate 40 consecutive bytes,
                           # with storage uninitialized
                           # could be used as a 40-element
                           # character array, or a
                           # 10-element integer array;
                           # a comment should indicate it.

string1:   .asciiz "Print this.\n"       #declare a string
```

```
.data
        var1:     .word  3
        array1:   .byte 'a', 'b'
```

Labels

Label	Address ▲
mips1.asm	
var1	0x10010000
array1	0x10010004

Data Segment

Address	Value (+0)	Value (+4)
0x10010000	0x00000003	0x00006261

The value of Data （2） relate to the usage(1)

➢ while calculate the data, if the instruction ends with “**u**” means the data are treated as **unsigned** integer, else the data are treated as **signed by defalut**.

```
.include "macro_print_str.asm"
.data
.text
main:
	print_string("\n -1 less than 1 using slt:")
	li $t0,-1
	li $t1,1
	slt $a0,$t0,$t1
	li $v0,1
	syscall

	print_string("\n -1 less than 1 using sltu:")
	sltu $a0,$t0,$t1
	li $v0,1
	syscall
	end
```

TIPS:

1) slt $t1,$t2,$t3

set less than: if $t2 is less than $t3, then set $t1 to 1 else set $t1 to 0

2) sltu $t1,$t2,$t3

set less than unsigned: if $t2 is less than $t3 using unsigned comparision, then set $t1 to 1 else set $t1 to 0

The value of Data （2） relate to the usage(2)

What's the data stored in the $a0 after execute "lb $a0,tdata"?
What are their values when they are treated as unsigned and signed integers respectively

```
.data
	tdata: .byte 0x0F00F0FF
	sx: .asciiz "\n"
.text
main:
	lb $a0,tdata
	li $v0,1
	syscall

	li $v0,36
	syscall

	li $v0,10
	syscall
```

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print integer as unsigned	36	$a0 = integer to print	Displayed as unsigned decimal value.

The value of Data （2） relate to the usage(3)

```
.include "macro_print_str.asm"
.data
	tdata: .byte 0x80
.text
main:
	lb $a0,tdata
	li $v0,1
	syscall

	print_string("\n")
	lb $a0,tdata
	li $v0,36
	syscall

	end
```

```
.include "macro_print_str.asm"
.data
	tdata: .byte 0x80
.text
main:
	lbu $a0,tdata
	li $v0,1
	syscall

	print_string("\n")
	lbu $a0,tdata
	li $v0,36
	syscall

	end
```

Q1: Run the two demos, what's the value stored in the register $a0 after the operation of 'lb' and 'lbu'
Q2: using "-1" as initial value of tdata instead of "0x80", answer Q1 again.

Data-storage and access

- **Byte**
  - data stored in memory are addressed in byte.

- **Starting address**
  - while the address are used in the instruction, it means this is the starting address of a piece of memory which maybe one or more than one memory units.

- While the bitwidth of **data moved** and **data storage location** are **mismatch**
  - while get a byte(8bit) from a register(32bit), the **LSB** part in it is got.
  - while load a byte(8bit) to a register(32bit), the rest 24bits need to be supplemented, it depends on the instruction.
    - **fill with 0** if the load insturction ends with "**u**", such as : "**lbu**", "**lhu**". Is there "lwu" ?
    - **fill with sign bit** if the load insturction ends without "u", such as : "lb", "lh".
    - Is there "sbu" or "shu" while store the data which is one byte("sbu") or a half word("shu") to the memory? why ?

Big-endian vs Little-endian(1)

The CPU's **byte ordering scheme** (or **endian issues**) affects memory organization and defines the relationship between address and byte position of data in memory.

- a **Big-endian** system means byte 0 is always the most-significant (leftmost) byte.
- a **Little-endian** system means byte 0 is always the least-significant (rightmost) byte.

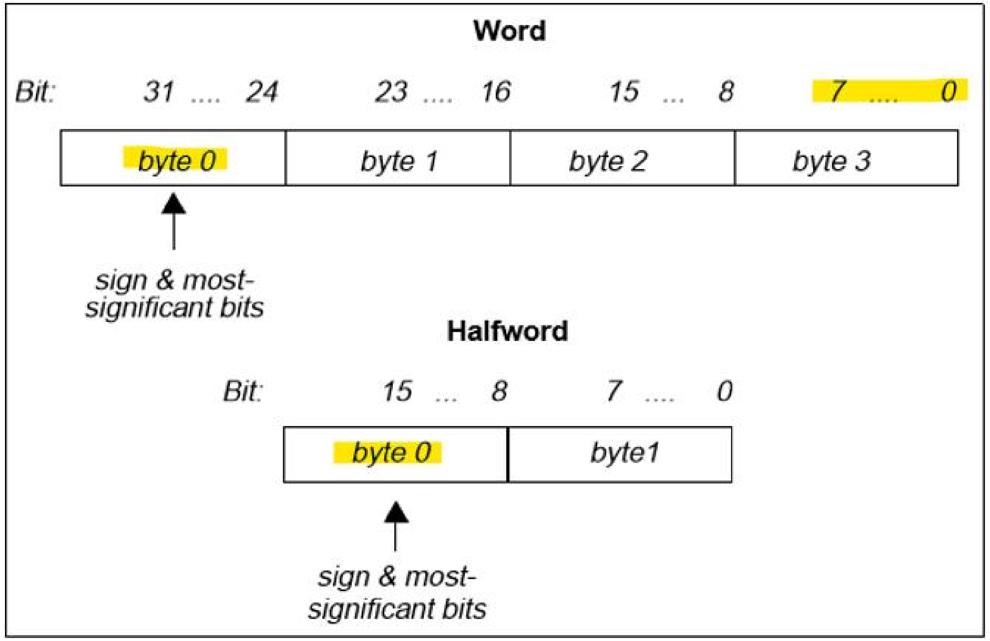


Figure 1-1: Big-endian Byte Ordering

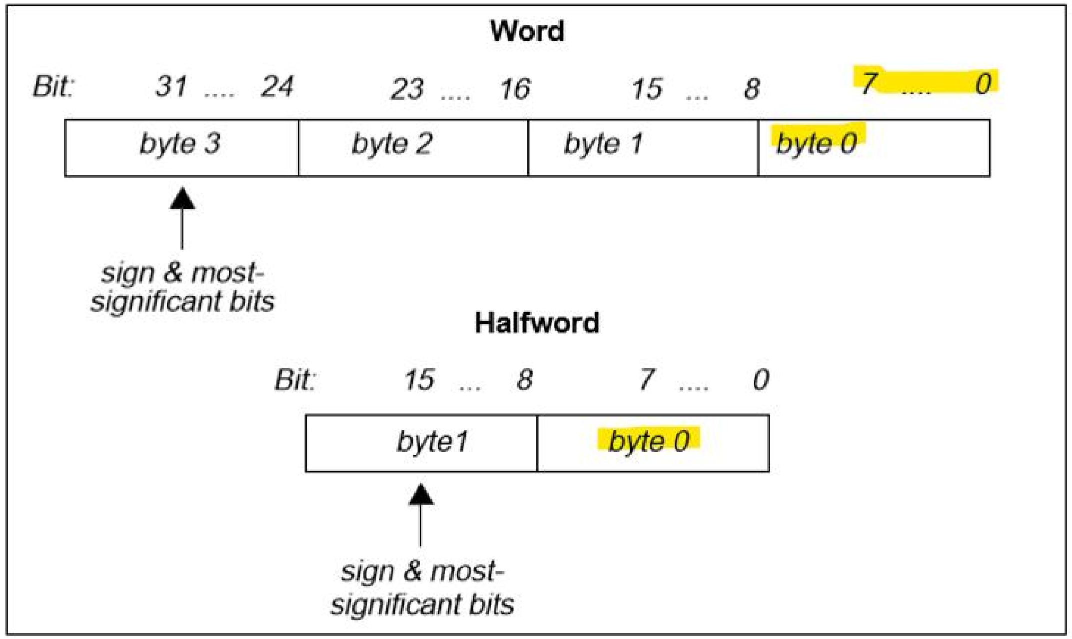


Figure 1-2: Little-endian Byte Ordering

Big-endian vs Little-endian(2)

Run the demo to anwer the question :
Does your simulator work on big-endian or little-endian, explain the reasons.

```
.include "macro_print_str.asm"
.data
	tdata0: .byte   0x11,0x22,0x33,0x44
	tdata:   .word  0x44332211
.text
main:
	lb $a0,tdata
	li $v0,34
	syscall

	end
```

```
.include "macro_print_str.asm"
.data
	tdata0: .byte   0x11,0x22,0x33,0x44
	tdata:   .word  0x44332211
.text
main:
	lh $a0,tdata
	li $v0,34
	syscall

	end
```

print integer in hexadecimal	34	$a0 = integer to print	Displayed value is 8 hexadecimal digits, left-padding with zeroes if necessary.

Big-endian or Little-endian?

```
.include "macro_print_str.asm"
.data
	tdata0: .word   0x00112233, 0x44556677
.text
main:
	la $t0,tdata0
	lb $a0,($t0)
	li $v0,34
	syscall

	la $t0,tdata0
	lb $a0,1($t0)
	syscall

	lb $a0,2($t0)
	syscall

	lb $a0,3($t0)
	syscall

	lw $a0,4($t0)
	syscall

	end
```

Run the demo to anwer the question :

Q1. What's the output of this demo?

A. ***0x***00000033***0x***00000022***0x***00000011***0x***00000000***0x***44556677

*B.****0x***00000000***0x***00000011***0x***00000022***0x***00000033***0x***44556677

*C.****0x***00000044***0x***00000055***0x***00000066***0x***00000077***0x***00112233

*D.****0x***00000077***0x***00000066***0x***00000055***0x***00000044***0x***33221100

Q2. Does your simulator work on big-endian or little-endian, explain the reasons.

print integer in hexadecimal	34	$a0 = integer to print	Displayed value is 8 hexadecimal digits, left-padding with zeroes if necessary.

Practice 2

2-1. The data in a word is 0x12345678, print it in hexdecimal, then exchange the bytes of this word to get the new value 0x78563412 and print updated data in hexdecimal.

2-2. Read a character, judge whether the binary representation of the character’s ascii code is palindrome.

For example, the ascii code of ‘f’ (102 in decimal, 0110_0110 in binary) is a binary palindrome, the ascii code of space(32 in decimal, 0010_0000 in binary) is not.

Tips: You can get more information from Mars' help page.

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Practice 2

2-3. Run the code on the right hand

Answer the questions

1) what’s the value of lable alice?

2) what’s the value of lable tony?

3) what’s the output after execute the syscall on line 23 ?

```
1  .data
2          name:       .space  16          #malloc 16 byte , not initialize   ##### name value : 0x10010000
3          mick:       .ascii "mick\n"      # malloc 4+1 = 5byte = 5 * asciic(byte)
4          alice:      .asciiz "alice\n"   ##### what's the value of alice ?
5          tony:       .asciiz "tony\n"    ##### what's the value of tony ?
6          chen:       .asciiz "chen\n"
7
8  .text
9  main:
10         la $t0,name              #using  name value which is an address, load this address to $t0
11
12         la $t1,mick
13         sw $t1,($t0)             #1,get value of $to, use it as the address of a piece of memory
14         la $t1,alice
15         sw $t1,4($t0)            #baseline : the content of  $t0 , offset :4
16         la $t1,tony
17         sw $t1,8($t0)
18         la $t1,chen
19         sw $t1,12($t0)
20
21         li $v0,4
22         lw $a0,0($t0)
23         syscall                  #what's the output while this syscall is done
24
25         li $v0,10
26         syscall
```

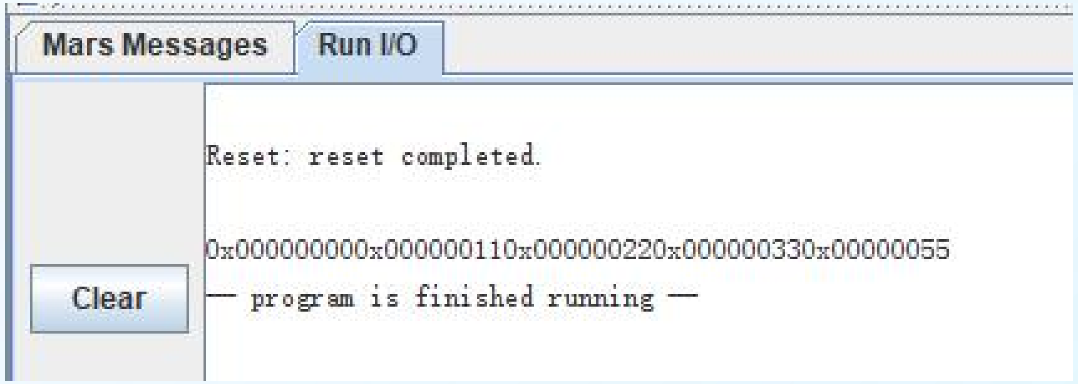
Tips1： macro_print_str.asm

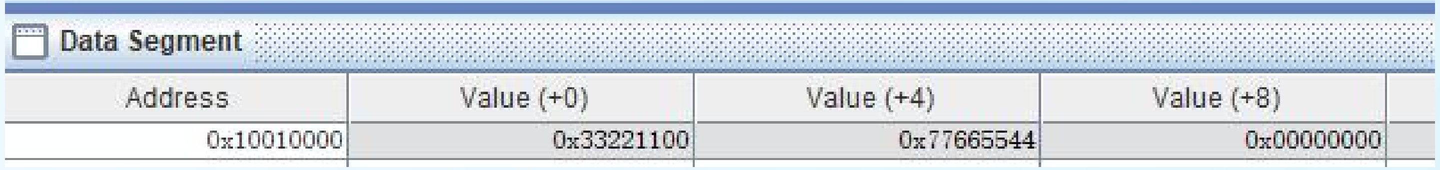
```
.macro print_string(%str)
	.data
		pstr:	.asciiz	%str
	.text
		la $a0,pstr
		li $v0,4
		syscall
.end_macro

.macro end
	li $v0,10
	syscall
.end_macro
```

Get help of defination and usage about macro from Mars‘ help page.

While using the macro, put this file to the same directory as the file which use the macro.

Tips2: the data address in Mars




Data Segment

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x33221100	0x77665544	0x00000000

```
.include "macro_print_str.asm"
.data
	tdata0: .byte
0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77
.text
main:
	la $t0,tdata0
	lb $a0, ($t0)
	li $v0,34
	syscall

	la $t0,tdata0
	lb $a0, 1($t0)
	syscall

	lb $a0, 2($t0)
	syscall

	lb $a0, 3($t0)
	syscall

	lb $a0, 5($t0)
	syscall

	end
```