



# Computer Organization

---

**Lab3**

**MIPS(2)**

**Instructions**



# Topics

- **Types of Instructions**
  - Data transfer
  - Calculation
  - Jump related to the instruction execution
- **How to determine the execution order of instructions**
  - PC register and its updation
- **Practice : p1(1-1,1-2,1-3) ,p2(2-1,2-2,2-3)**
- **Tips: Big-endian vs Little-endian**



# Summary of MIPS32 Instruction Types

- **Function**
  - data **transfer**(load, store)
  - **calculation**(arithmetic, Bitwise operation)
  - **Jump** instructions related to instruction execution
- Relation with **data**
  - **Immediate** is a operand vs all the operands are from registers
  - treat data as **signed** vs treat data as **Unsigned**
- **Coding** of Instruction: R, I, J
- **Basic** instruction vs **Pseudo** instruction
  - lui, ori vs la
  - add \$t0,\$zero,\$t1 vs move \$t0,\$t1



# Common Operations

Description	Op-code	Operand
Add with Overflow	add	destination, src1, src2
Add without Overflow	addu	destination, src1, src2
AND	and	destination, src1, immediate
Divide Signed	div	destination/src1, immediate
Divide Unsigned	divu	
Exclusive-OR	xor	
Multiply	mul	
Multiply with Overflow	mulo	
Multiply with Overflow Unsigned	mulou	
NOT OR	nor	
OR	or	
Set Equal	seq	
Set Greater	sgt	
Set Greater/Equal	sge	
Set Greater/Equal Unsigned	sgeu	
Set Greater Unsigned	sgtu	
Set Less	slt	
Set Less/Equal	sle	
Set Less/Equal Unsigned	sleu	
Set Less Unsigned	sltu	
Set Not Equal	sne	
Subtract with Overflow	sub	
Subtract without Overflow	subu	

Description	Op-code	Operand
Rotate Left	rol	
Rotate Right	ror	
Shift Right Arithmetic	sra	
Shift Left Logical	sll	
Shift Right Logical	srl	
Absolute Value	abs	destination,src1
Negate with Overflow	neg	destination/src1
Negate without Overflow	negu	
NOT	not	
Move	move	destination,src1
Multiply	mult	src1,src2
Multiply Unsigned	multu	



# Signed vs Unsigned (caculation)

*Run the two demos,  
which one will invoke  
the exception  
(arithmetic  
overflow) , why?*

```
.include "macro_print_str.asm"  
.data  
    tdata: .word 0x11111111  
.text  
main:  
    lw $t0,tdata  
    addu $a0,$t0,$t0  
    li $v0,1  
    syscall  
  
    print_string("\n")  
    add $a0,$t0,$t0  
    li $v0,1  
    syscall  
  
end #A
```

```
.include "macro_print_str.asm"  
.data  
    tdata: .word 0x71111111  
.text  
main:  
    lw $t0,tdata  
    addu $a0,$t0,$t0  
    li $v0,1  
    syscall  
  
    print_string("\n")  
    add $a0,$t0,$t0  
    li $v0,1  
    syscall  
  
end #B
```



# Bit-wise Logic Operation(1)

Instruction name	description
<b>and</b> (AND) <i>and dst,src1,src2(im)</i>	Computes the <b>Logical AND</b> of two values. This instruction ANDs (bit-wise) the contents of src1 with the contents of src2, or it can AND the contents of src1 with the immediate value. <b>The immediate value is NOT sign extended.</b> AND puts the result in the destination register.
<b>or</b> (OR) <i>or dst,src1,src2(im)</i>	Computes the <b>Logical OR</b> of two values. This instruction ORs (bit-wise) the contents of src1 with the contents of src2, or it can OR the contents of src1 with the immediate value. <b>The immediate value is NOT sign extended.</b> OR puts the result in the destination register
<b>xor</b> (Exclusive-OR) <i>xor dst,src1,src2(im)</i>	Computes the <b>XOR</b> of two values. This instruction XORs (bit-wise) the contents of src1 with the contents of src2, or it can XOR the contents of src1 with the immediate value. <b>The immediate value is NOT sign extended.</b> Exclusive-OR puts the result in the destination register
<b>not</b> (NOT) <i>not dst,src1</i>	Computes the <b>Logical NOT</b> of a value. This instruction complements (bit-wise) the contents of src1 and puts the result in the destination register.
<b>nor</b> (NOT OR) <i>nor dst,src1,src2</i>	Computes the <b>NOT OR</b> of two values. This instruction combines the contents of src1 with the contents of src2 (or the immediate value). NOT OR complements the result and puts it in the destination register.





# Bit-wise Logic Operation(2)

*Run the demo and answer the question :*

```
.data
    dvalue1: .byte 27
    dvalue2: .byte 4
.text
    lb $t0,dvalue1
    lb $t1,dvalue2

    div $t0,$t1
    mfhi $a0

    li $v0,1
    syscall

    li $v0,10
    syscall
```

```
.data
    dvalue1: .byte 27
    dvalue2: .byte 4
.text
    lb $t0,dvalue1
    lb $t1,dvalue2

    sub $t1,$t1,1
    and $a0,$t0,$t1

    li $v0,1
    syscall

    li $v0,10
    syscall
```

*Q1: Is the output of two demos the same?*

*Q2: If use 5 instead of 4 as the initial value on dvalue2, is the output of two demos the same?*

*Q3: On which situation could use 'and' operation to get the remainder instead of division?*

*Q4: Do the logic operations work quicker than arithmetic operations?*



# Shift Operation

Type	Instruction name	Description	ShiftOperator src1, src2
shift	<b>sll</b> (Shift Left Logical)	Shifts the contents of a register left (toward the sign bit) and inserts zeros at the least-significant bit.	The contents of <b>src1</b> specify the value to shift, and the contents of <b>src2</b> or the immediate value specify the amount to shift.  If src2 (or the immediate value) is greater than 31 or less than 0, src1 shifts by the result of src2 MOD 32.
	<b>sra</b> (Shift Right Arithmetic)	Shifts the contents of a register right (toward the least-significant bit) and inserts the sign bit at the most-significant bit.	
	<b>srl</b> (Shift Right Logical)	Shifts the contents of a register right (toward the least-significant bit) and inserts zeros at the most-significant bit.	
rotate	<b>rol</b> (Rotate Left)	Rotates the contents of a register left (toward the sign bit). This instruction inserts in the least-significant bit any bits that were shifted out of the sign bit.	The contents of <b>src1</b> specify the value to shift, and the contents of <b>src2</b> (or the immediate value) specify the amount to shift. Rotate Left/right puts the result in the destination register.  If src2 (or the immediate value) is greater than 31, src1 shifts by the result of src2 MOD 32.
	<b>ror</b> (Rotate Right)	Rotates the contents of a register right (toward the least-significant bit). This instruction inserts in the sign bit any bits that were shifted out of the least-significant bit.	





# Practice1-1

*Here is a demo to meet the following function: get the integer from input, judge whether the data is odd, if it is odd then print 1, else print 0.*

*1-1-1). Run the demo to see if the function of the code is ok ? if not please find the reason and modify the code to meet the design expectations .*

*1-1-2). Which is(are) basic instruction(s) in the following set: li, move, nor, sra, and, syscall ?*

```
.include "macro_print_str.asm"
```

```
.data
```

```
.text
```

```
main:
```

```
    print_string("please input an integer : ")
```

```
    li $v0,5
```

```
    syscall
```

```
    move    $t0, $v0
```

```
    nor     $t1, $zero, $zero
```

```
    sra    $t2, $t1,    31
```

```
    and    $a0, $t2,    $t0
```

```
    print_string("it is an odd number (0: false,1:true) : ")
```

```
    li $v0,1
```

```
    syscall
```

```
end
```

```
please input an integer : 3
it is an odd number (0: false,1:true) : 1
-- program is finished running --
```



## Practice1-2

*1-2: Calculate the checksum of data. Here are the steps of calculation:*

- 1) The data is grouped in 16bit units*
- 2) The data of each group is accumulated.*
- 3) The accumulation is divided into two parts according to the bit width.  
the first part is the low-address part of subscripts 0 to 15  
the second part is the part of the accumulated sum that exceeds the bit width of 16bit.*
- 4) Then the first part and the second part are accumulated again(if there is no 16bit in second part, fill in 0 on the high-bit bits).*
- 5) The one's complement of accumulated sum is the checksum*

*The data is grouped as: 0x7f00, 0x0001, 0x7f00, 0x0001, 0x0011, 0xcbe0, 0x2ee0, 0x000a, 0x6162,  
calculate its checksum and print it out in hexadecimal.*

*tips: the part2 of the accumulated sum (step3) is 0x0002*



## Practice1-3

*There are 5 shift operations list on page8, practice it's function in MIPS and implement them by Verilog.*

*1-3-1: Write a demo in MIPS to practice 5 shift operations list on page 8 of this slice*

*1-3-2: Implement 5 shift operations list on page 8 of this slice in Verilog, build the testbench to verify its function. It's suggested to build a module with two inputs(src1, src2), five outputs(osll,oslr,osla,orol,oror), the bitwise of all inputs and outputs is 32. the output port "osll" is the output of operation sll src1,src2, and so on.*

*Here are some suggestions for the testcase, the value of src1 could be 0x8000\_0000, 0x0000\_0001, 0xFFFF\_FFFF, 0x0000\_FFFF, the value of src2 could be 0x0010\_0000, 0x0010\_0001, 0x8010\_0000, 0x8010\_0001, 0x0001\_1111, 0x0011\_1111.*

Type	Instruction name
shift	<b>sll</b> (Shift Left Logical)
	<b>sra</b> (Shift Right Arithmetic )
	<b>srl</b> (Shift Right Logical )
rotate	<b>rol</b> (Rotate Left )
	<b>ror</b> (Rotate Right )

# ‘Who’ determine the execution order of instructions

- The CPU takes the value of the **PC** register as the address and fetches the corresponding instruction from the memory.
- **PC** register maintains the address of the instruction currently being executed.
- **After** the current instruction is executed, the value of the **PC** register will be **updated** to determine the next instruction to be executed.

Text Segment				
Offset	Address	Code	Basic	Source
0	0x00400000	0x24020008	addiu \$2,\$0,0x00000008	7: li \$v0,8 #to get a string
4	0x00400004	0x3c011001	lui \$1,0x00001001	8: la \$a0,sid
8	0x00400008	0x34240008	ori \$4,\$1,0x00000008	
c	0x0040000c	0x24050009	addiu \$5,\$0,0x00000009	9: li \$a1,9
10	0x00400010	0x0000000c	syscall	10: syscall
14	0x00400014	0x24020004	addiu \$2,\$0,0x00000004	13: li \$v0,4 #to print a string
18	0x00400018	0x3c011001	lui \$1,0x00001001	14: la \$a0,s1
1c	0x0040001c	0x34240000	ori \$4,\$1,0x00000000	
20	0x00400020	0x0000000c	syscall	15: syscall
24	0x00400024	0x2402000a	addiu \$2,\$0,0x0000000a	16: li \$v0,10 #to exit
28	0x00400028	0x0000000c	syscall	17: syscall

pc		0x00400000
pc		0x00400004
pc		0x00400008
pc		0x0040000c

pc		0x00400028
----	--	------------



# How to update the value of PC register?

- Check if the current instruction is non-jump
  - If the current instruction is non-jump instruction:  $PC = PC + 4$
  - If the current instruction is jump instruction
    - If the current instruction is unconditional jump  $pc = \text{destination address}$
    - If the current instruction is conditional jump
      - If the condition is met:  $PC = \text{destination address}$
      - If the condition is not met:  $PC = PC + 4$





# Conditional Jump

basic instruction	usage
<b>beq \$t0,\$t1,labelx</b>	<i>branch to instruction addressed by the labelx if \$t0 and \$t1 are equal</i>
<b>bne \$t0,\$t1,labelx</b>	<i>branch to instruction addressed by the labelx if \$t0 and \$t1 are NOT equal</i>

pseudo instruction	basic instruction	usage
<b>blt \$t0,\$t1,lable</b>	slt \$1, \$t0, \$t1 bne \$1,\$0, lable	<i># branch to instruction addressed by the label if \$t0 is less than \$t1, data in \$t0 and \$t1 are taken as signed number</i>
<b>ble \$t0,\$t1,lable</b>	slt \$1,\$t1,\$t0 beq \$1,\$0,lable	<i># branch to instruction addressed by the label if \$t0 is less or equal than \$t1, data in \$t0 and \$t1 are taken as signed number</i>
<b>bltu \$t0,\$t1,lable</b>	sltu \$1, \$t0, \$t1 bne \$1,\$0, lable	<i># branch to instruction addressed by the label if \$t0 is less than \$t1, data in \$t0 and \$t1 are taken as <b>unsigned</b> number</i>
<b>bleu \$t0,\$t1,lable</b>	sltu \$1,\$t1,\$t0 beq \$1,\$0,lable	<i># branch to instruction addressed by the label if \$t0 is less or equal than \$t1, data in \$t0 and \$t1 are taken as <b>unsigned</b> number</i>
<b>bgt, bge, bgtu, bgeu.....</b>		





# Branch

*Are the running results of two demos the same ?*

*Modify them without changing the result by using **ble** or **blt** instead*

```
.include "macro_print_str.asm"
.text
    print_string("please input your score (0~100):")
    li $v0,5
    syscall
    move $t0,$v0
case1:
    bge $t0,60,passLable
case2:
    j failLable

passLable:
    print_string("\nPASS (exceed or equal 60) ")
    j caseEnd
failLable:
    print_string("\nFaild(less than 60)")
    j caseEnd
caseEnd:
    end
```

```
.include "macro_print_str.asm"
.text
    print_string("please input your score (0~100):")
    li $v0,5
    syscall
    move $t0,$v0
case1:
    bge $t0,60,passLable
    j case2
case2:
    j failLable

passLable:
    print_string("\nPASS (exceed or equal 60) ")
    j caseEnd
failLable:
    print_string("\nFaild(less than 60)")
    j caseEnd
caseEnd:
    end
```



# Loop

***Compare the operations of loop which calculates the sum from 1 to 10 in java and MIPS.***

***Code in Java:***

```
public class CalculateSum{
    public static void main(String [] args){
        int i = 0;
        int sum = 0;
        for(i=0;i<=10;i++)
            sum = sum + i;
        System.out.print("The sum from 1 to 10 : " + sum );
    }
}
```

***Code in MIPS:***

```
.include "macro_print_str.asm"
.data
    #....
.text
    add $t1,$zero,$zero
    addi $t0,$zero,0
    addi $t7,$zero,10
calcu:
    addi $t0,$t0,1    #i++
    add $t1,$t1,$t0   #sum+=i
    bgt $t7,$t0,calcu #if(t7>t0) t0==t7

    print_string ("The sum from 1 to 10 : ")
    move $a0,$t1
    li $v0,1
    syscall

    end
```



# Demo #1

*The following code is expected to get 10 integers from the input device, and print it as the following sample.  
Will the code get desired result?  
If not, what happened ?*

```
#piece 1/3

.include "macro_print_str.asm"
.data
    arrayx:    .space    10
    str:        .asciiz   "\nthe arrayx is:"
.text
main:
    print_string("please input 10 integers: ")
    add $t0,$zero,$zero
    addi $t1,$zero,10
    la $t2,arrayx
```


```
#piece 2/3
loop_r:
    li $v0,5
    syscall
    sw $v0,($t2)
    addi $t0,$t0,1
    addi $t2,$t2,4
    bne $t0,$t1,loop_r

    la $a0,str
    li $v0,4
    syscall
    addi $t0,$zero,0
    la $t2,arrayx
```

```
#piece 3/3
loop_w:
    lw $a0,($t2)
    li $v0,1
    syscall
    print_string(" ")
    addi $t2,$t2,4
    addi $t0,$t0,1
    bne $t0,$t1,loop_w
end
```

```
please input 10 integers: 0
1
2
3
4
5
6
7
8
9

the arrayx is:0 1 2 3 4 5 6 7 8 9
— program is finished running —
```



*The function of following code is to get 5 integers from input device, and find the min value and max value of them. There are 4 pieces of code, write your code based on them. Can it find the real min and max?*

```
#piece ?/4
.include "macro_print_str.asm"
.data
    min: .word 0
    max: .word 0
.text
    lw $t0,min
    lw $t1,max
    li $t7,5
    li $t6,0
    print_string("please input 5
integer:")
loop:
    li $v0,5
    syscall
    bgt $v0,$t1,get_max
    j get_min
```

```
#piece ?/4
get_max:
    move $t1,$v0
    j get_min
get_min:
    bgt $v0,$t0,judge_times
    move $t0,$v0
    j judge_times
```

```
#piece ?/4
judge_times:
    addi $t6,$t6,1
    bgt $t7,$t6,loop
```

```
#piece ?/4
    print_string("min : ")
    move $a0,$t0
    li $v0,1
    syscall
    print_string("max : ")
    move $a0,$t1
    li $v0,1
    syscall
end
```



## Practice2-1,2-2

1. Answer the questiones on page 15 and16
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	_		





## Practice2-3

Implement the circuit described on page 13 in Verilog.

Build a testbench to verify its function.

NOTES: the width of all the register is 32bits.

The logic described on page 13:

- Check if the current instruction is non-jump
  - If the current instruction is non-jump instruction:  $PC = PC + 4$
  - If the current instruction is jump instruction
    - If the current instruction is unconditional jump  $pc = \text{destination address}$
    - If the current instruction is conditional jump
      - If the condition is met:  $PC = \text{destination address}$
      - If the condition is not met:  $PC = PC + 4$





# Tips : 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.

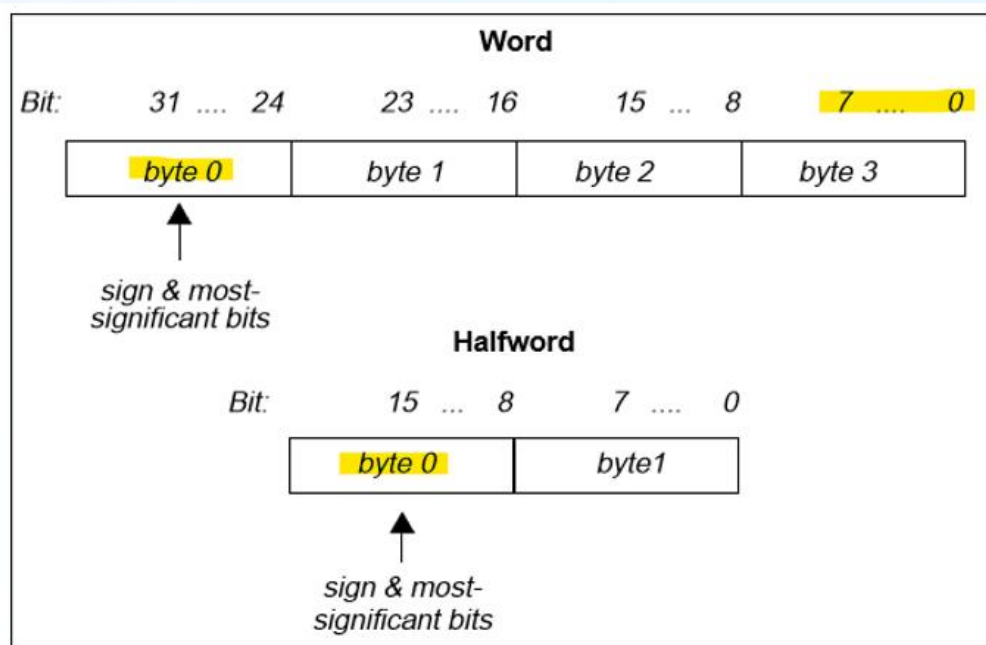


Figure 1-1: Big-endian Byte Ordering

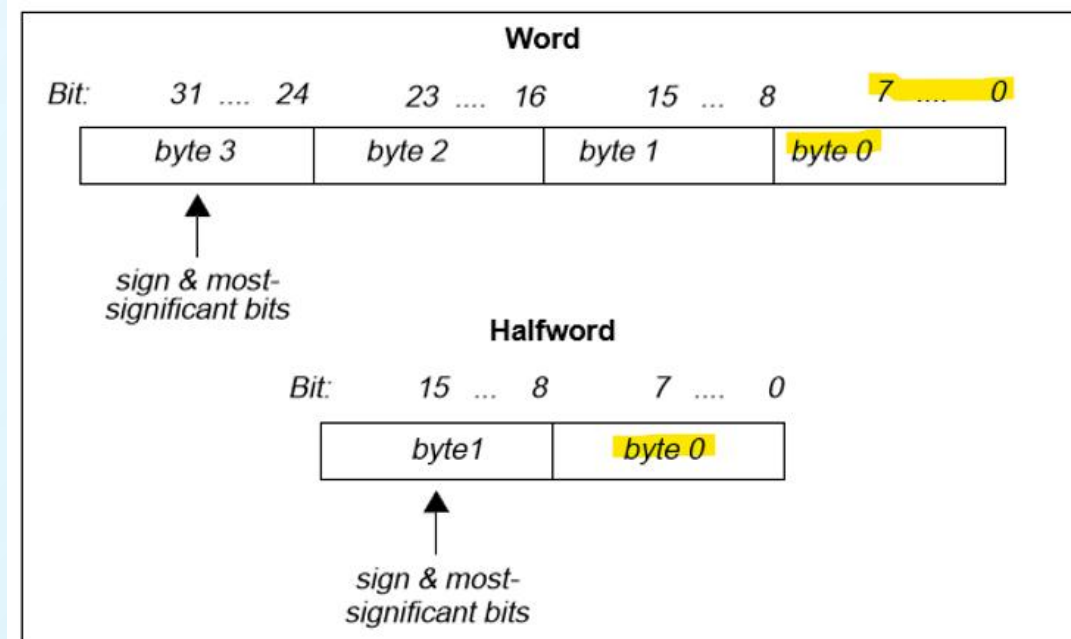


Figure 1-2: Little-endian Byte Ordering



## Tips : Big-endian vs Little-endian(2)

*Run the demo to answer 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.
------------------------------	----	-------------------------	---------------------------------------------------------------------------------



# Tips : 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 answer the question :*

*Q1. What's the output of this demo?*

A. **0x0000000330x000000220x000000110x000000000x44556677**

B. **0x0000000000x000000110x000000220x000000330x44556677**

C. **0x0000000440x000000550x000000660x000000770x00112233**

D. **0x0000000770x000000660x000000550x000000440x33221100**

*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.
------------------------------	----	-------------------------	---------------------------------------------------------------------------------