MIPS32 Instruction Set

Computer Architecture

Lectures 3-4

Mechatronics Engineering

Assistant Professor: Isam M. Asaad

References:

- ICS 233 Course, Computer Architecture and Assembly Language, Prof. Muhamed Mudawar, College of Computer Sciences and Engineering, King Fahd University of Petroleum and Minerals.
- EECS2021E (RISC-V Version)- Computer Organization and Architecture - Fall 2019 Lectures, Amir H. Ashouri, York University.

Outline

Part1

- Instruction Set Architecture
- Overview of the MIPS Processor
- R-Type Arithmetic, Logical, and Shift Instructions
- I-Type Format and Immediate Constants
- Jump and Branch Instructions
- Translating If Statements and Boolean Expressions
- Load and Store Instructions
- Translating Loops and Traversing Arrays
- Addressing Modes

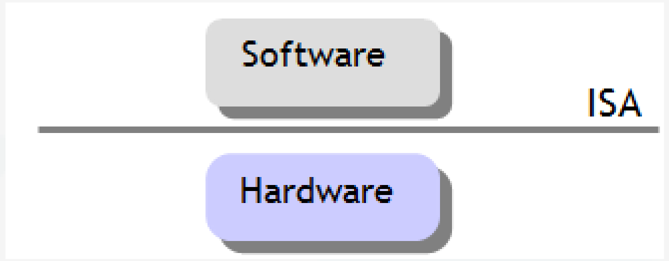
Part2

- Assembly Language Statements
- Assembly Language Program Template
- Defining Data
- Memory Alignment and Byte Ordering
- System Calls
- Procedures
- Parameter Passing and the Runtime Stack

Presentation Outline (Part1)

Instruction Set Architecture (ISA)

- The ISA is an interface between hardware and software
- An ISA includes the following …
  - Instructions and Instruction Formats
    - Data Types, Encodings, and Representations
    - Addressing Modes: to address Instructions and Data
    - Handling Exceptional Conditions (like division by zero)
  - Programmable Storage: Registers and Memory



- Examples

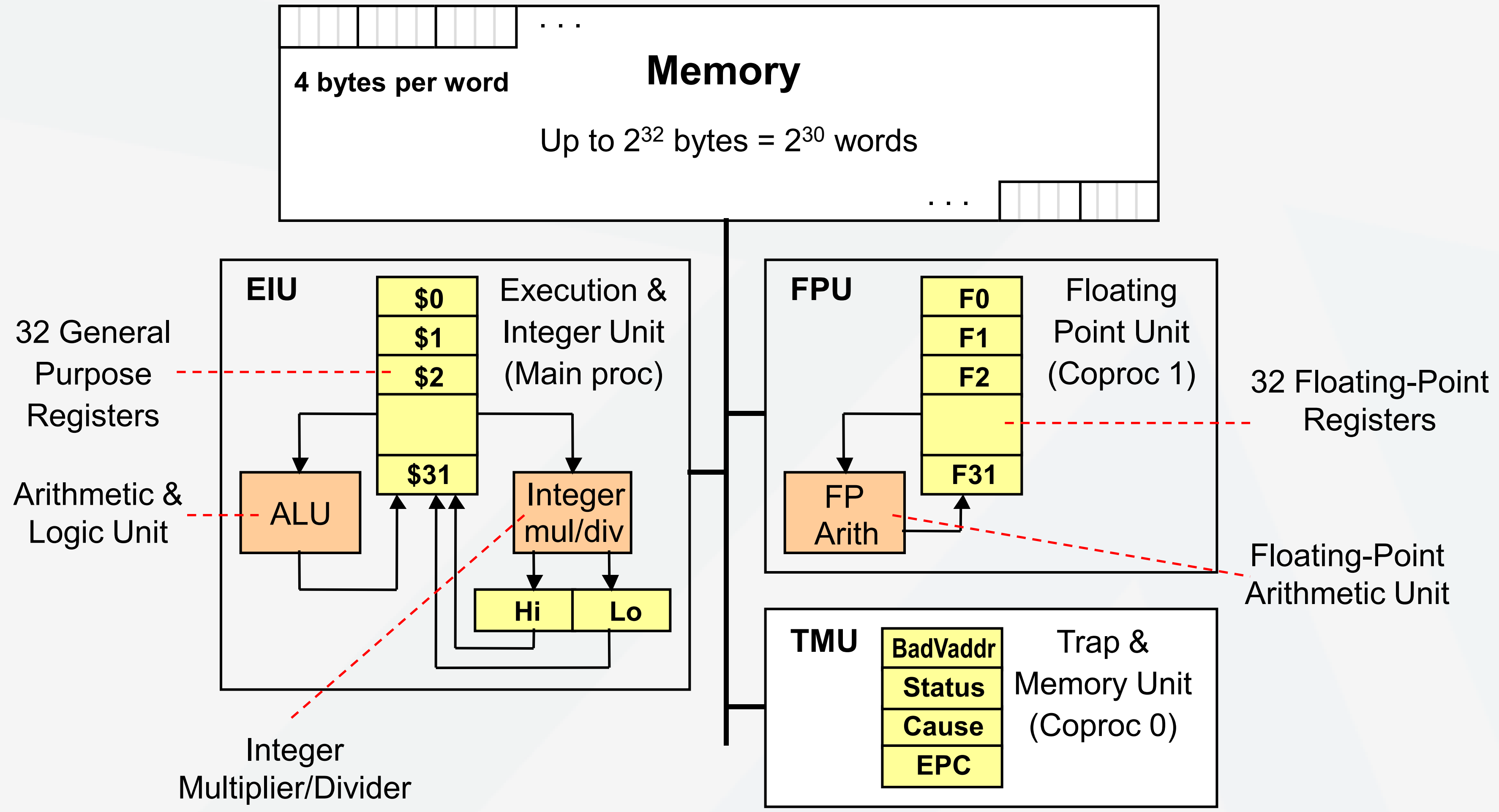
Examples	(Versions)	First Introduced in
– Intel	(8086, 80386, Pentium, ...)	1978
– MIPS	(MIPS I, II, III, IV, V)	1986
– PowerPC	(601, 604, …)	1993

Instructions

- Instructions are the language of the machine
- We will study the MIPS instruction set architecture
  - Known as **Reduced Instruction Set Computer (RISC)**
  - Elegant and relatively simple design
  - Very popular, used in many products
    - Silicon Graphics, ATI, Cisco, Sony, etc.
  - Comes next in sales after Intel IA-32 processors
    - Almost 100 million MIPS processors sold in 2002 (and increasing)
- Alternative design: Intel IA-32
  - Known as **Complex Instruction Set Computer (CISC)**

Next . . .

Overview of the MIPS Architecture


. . .
4 bytes per word
Memory
Up to 2^{32} bytes = 2^{30} words
. . .
EIU
$0
$1
$2
$31
Execution & Integer Unit (Main proc)
32 General Purpose Registers
ALU
Arithmetic & Logic Unit
Integer mul/div
Hi
Lo
Integer Multiplier/Divider
FPU
F0
F1
F2
F31
Floating Point Unit (Coproc 1)
32 Floating-Point Registers
FP Arith
Floating-Point Arithmetic Unit
TMU
BadVaddr
Status
Cause
EPC
Trap & Memory Unit (Coproc 0)

MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - Assembler uses the dollar notation to name registers
    - $0 is register 0, $1 is register 1, …, and $31 is register 31
  - All registers are 32-bit wide in MIPS32
  - Register $0 is always zero
    - Any value written to $0 is discarded
- Software conventions
  - There are many registers (32)
  - Software defines names to all registers
    - To standardize their use in programs
  - Example: $8 - $15 are called $t0 - $t7
    - Used for temporary values

$0 = $zero	$16 = $s0
$1 = $at	$17 = $s1
$2 = $v0	$18 = $s2
$3 = $v1	$19 = $s3
$4 = $a0	$20 = $s4
$5 = $a1	$21 = $s5
$6 = $a2	$22 = $s6
$7 = $a3	$23 = $s7
$8 = $t0	$24 = $t8
$9 = $t1	$25 = $t9
$10 = $t2	$26 = $k0
$11 = $t3	$27 = $k1
$12 = $t4	$28 = $gp
$13 = $t5	$29 = $sp
$14 = $t6	$30 = $fp
$15 = $t7	$31 = $ra

MIPS Register Conventions

- ❖ Assembler can refer to registers by name or by number
  - ✧ It is easier for you to remember registers by name
  - ✧ Assembler converts register name to its corresponding number

Name	Register	Usage
$zero	$0	Always 0 (forced by hardware)
$at	$1	Reserved for assembler use
$v0 – $v1	$2 – $3	Result values of a function
$a0 – $a3	$4 – $7	Arguments of a function
$t0 – $t7	$8 – $15	Temporary Values
$s0 – $s7	$16 – $23	Saved registers (preserved across call)
$t8 – $t9	$24 – $25	More temporaries
$k0 – $k1	$26 – $27	Reserved for OS kernel
$gp	$28	Global pointer (points to global data)
$sp	$29	Stack pointer (points to top of stack)
$fp	$30	Frame pointer (points to stack frame)
$ra	$31	Return address (used by jal for function call)

Instruction Formats

❖ All instructions are 32-bit wide, Three instruction formats:

❖ Register (R-Type)

- ✧ Register-to-register instructions
- ✧ Op: operation code specifies the format of the instruction

Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6

❖ Immediate (I-Type)

- ✧ 16-bit immediate constant is part in the instruction

Op^6	Rs^5	Rt^5	immediate^{16}

❖ Jump (J-Type)

- ✧ Used by jump instructions

Op^6	immediate^{26}

Instruction Categories

- Integer Arithmetic
  - Arithmetic, logical, and shift instructions
- Data Transfer
  - Load and store instructions that access memory
  - Data movement and conversions
- Jump and Branch
  - Flow-control instructions that alter the sequential sequence
- Floating Point Arithmetic
  - Instructions that operate on floating-point registers
- Miscellaneous
  - Instructions that transfer control to/from exception handlers
  - Memory management instructions (e.g. stack)

Next . . .

R-Type Format

Op^6	Rs^5	Rt^5	Rd^5	sa^5	$funct^6$

- **Op**: operation code (opcode)
  - Specifies the operation of the instruction
  - Also specifies the format of the instruction
- **funct**: function code – extends the opcode
  - Up to $2^6 = 64$ functions can be defined for the same opcode
  - MIPS uses opcode 0 to define R-type instructions
- Three Register Operands (common to many instructions)
  - **Rs**, **Rt**: first and second source operands
  - **Rd**: destination operand
  - **sa**: the shift amount used by shift instructions

Integer Add / Subtract Instructions

Instruction	Meaning	R-Type Format					
add $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x20
addu $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x21
sub $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x22
subu $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x23

- add & sub: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- addu & subu: same operation as add & sub
  - However, no arithmetic exception can occur
  - **Overflow is ignored**
- Many programming languages ignore overflow
  - The + operator is translated into **addu**
  - The – operator is translated into **subu**

Addition/Subtraction Example

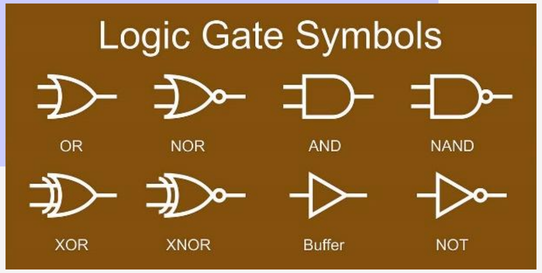
- Consider the translation of: f = (g+h) – (i+j)
- <u>Compiler allocates registers to variables</u>
  - Assume that *f*, *g*, *h*, *i*, and *j* are allocated registers $s0 thru $s4
  - Called the **saved** registers: $s0 = $16, $s1 = $17, …, $s7 = $23
- Translation of: f = (g+h) – (i+j)

```
addu  $t0, $s1, $s2     # $t0 = g + h
addu  $t1, $s3, $s4     # $t1 = i + j
subu  $s0, $t0, $t1     # f = (g+h)–(i+j)
```

  - Temporary results are stored in $t0 = $8 and $t1 = $9
- Translate: `addu $t0,$s1,$s2` to binary code
- Solution:

op	rs = $s1	rt = $s2	rd = $t0	sa	func
000000	10001	10010	01000	00000	100001

Logical Bitwise Operations



- Logical bitwise operations: and, or, xor, nor

x	y	x and y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x or y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0

- AND instruction is used to clear bits: x and 0 = 0
- OR instruction is used to set bits: x or 1 = 1
- XOR instruction is used to toggle bits: x xor 1 = not x
- NOR instruction can be used as a NOT, how?
  - `nor $s1,$s2,$s2` is equivalent to `not $s1,$s2`

Logical Bitwise Instructions

Instruction	Meaning	R-Type Format					
and $s1, $s2, $s3	$s1 = $s2 & $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x24
or $s1, $s2, $s3	$s1 = $s2 | $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x25
xor $s1, $s2, $s3	$s1 = $s2 ^ $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x26
nor $s1, $s2, $s3	$s1 = ~($s2|$s3)	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x27

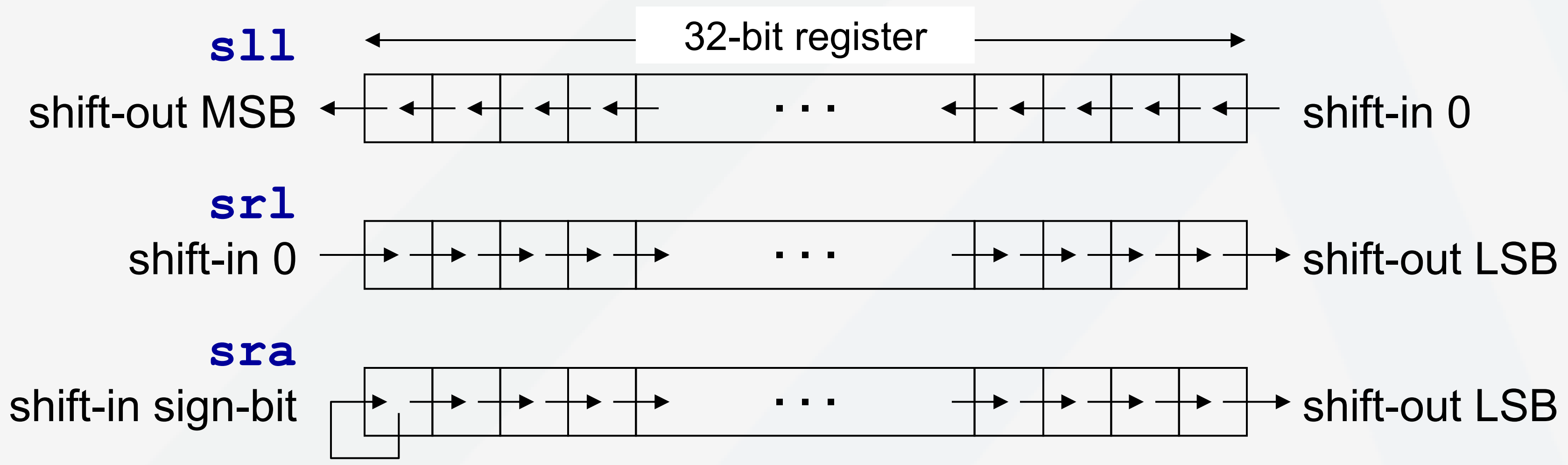
❖ Examples:

Assume `$s1 = 0xabcd1234` and `$s2 = 0xffff0000`

```
and $s0,$s1,$s2      # $s0 = 0xabcd0000
or  $s0,$s1,$s2      # $s0 = 0xffff1234
xor $s0,$s1,$s2      # $s0 = 0x54321234
nor $s0,$s1,$s2      # $s0 = 0x0000edcb
```

Shift Operations

- ❖ Shifting is to move all the bits in a register left or right
- ❖ Shifts by a constant amount: `sll, srl, sra`
  - ✧ `sll/srl` mean shift left/right logical by a constant amount
  - ✧ The 5-bit shift amount field is used by these instructions
  - ✧ `sra` means shift right arithmetic by a constant amount
  - ✧ The sign-bit (rather than 0) is shifted from the left



Shift Instructions

	Instruction	Meaning	R-Type Format					
	sll $s1,$s2,10	$s1 = $s2 << 10	op = 0	rs = 0	rt = $s2	rd = $s1	sa = 10	f = 0
Unsigned	srl $s1,$s2,10	$s1 = $s2>>>10	op = 0	rs = 0	rt = $s2	rd = $s1	sa = 10	f = 2
Signed	sra $s1, $s2, 10	$s1 = $s2 >> 10	op = 0	rs = 0	rt = $s2	rd = $s1	sa = 10	f = 3
	sllv $s1,$s2,$s3	$s1 = $s2 << $s3	op = 0	rs = $s3	rt = $s2	rd = $s1	sa = 0	f = 4
Unsigned	srlv $s1,$s2,$s3	$s1 = $s2>>>$s3	op = 0	rs = $s3	rt = $s2	rd = $s1	sa = 0	f = 6
Signed	srav $s1,$s2,$s3	$s1 = $s2 >> $s3	op = 0	rs = $s3	rt = $s2	rd = $s1	sa = 0	f = 7

- ❖ Shifts by a variable amount: **`sllv`**, **`srlv`**, **`srav`**
  - ✧ Same as **`sll`**, **`srl`**, **`sra`**, but a register is used for shift amount
- ❖ Examples: assume that $s2 = 0xabcd1234, $s3 = 16

```
sll  $s1,$s2,8     $s1 = $s2<<8      $s1 = 0xcd123400
sra  $s1,$s2,4     $s1 = $s2>>4      $s1 = 0xfabcd123
srlv $s1,$s2,$s3   $s1 = $s2>>>$s3   $s1 = 0x0000abcd
```

op=000000	rs=$s3=10011	rt=$s2=10010	rd=$s1=10001	sa=00000	f=000110

Binary Multiplication

- Shift-left (`sll`) instruction can perform multiplication
  - When the multiplier is a power of 2
- You can factor any binary number into powers of 2
  - Example: multiply `$s1` by `36`
    - Factor 36 into (4 + 32) and use distributive property of multiplication
  - `$s2 = $s1*36 = $s1*(4 + 32) = $s1*4 + $s1*32`

```
sll  $t0, $s1, 2      ; $t0 = $s1 * 4
sll  $t1, $s1, 5      ; $t1 = $s1 * 32
addu $s2, $t0, $t1    ; $s2 = $s1 * 36
```

Your Turn . . .

Multiply $s1 by 26, using shift and add instructions

Hint: 26 = 2 + 8 + 16

```
sll   $t0, $s1, 1          ; $t0 = $s1 * 2
sll   $t1, $s1, 3          ; $t1 = $s1 * 8
addu  $s2, $t0, $t1        ; $s2 = $s1 * 10
sll   $t0, $s1, 4          ; $t0 = $s1 * 16
addu  $s2, $s2, $t0        ; $s2 = $s1 * 26
```

Multiply $s1 by 31, Hint: 31 = 32 – 1

```
sll   $s2, $s1, 5          ; $s2 = $s1 * 32
subu  $s2, $s2, $s1        ; $s2 = $s1 * 31
```

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Architecture
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Addressing Modes

I-Type Format

- Constants are used quite frequently in programs
  - The R-type shift instructions have a 5-bit shift amount constant
  - What about other instructions that need a constant?
- I-Type: Instructions with Immediate Operands

Op^6	Rs^5	Rt^5	$immediate^{16}$

- 16-bit immediate constant is stored inside the instruction
  - Rs is the source register number
  - Rt is now the destination register number (for R-type it was Rd)
- Examples of I-Type ALU Instructions:
  - Add immediate: `addi $s1, $s2, 5   # $s1 = $s2 + 5`
  - OR immediate: `ori  $s1, $s2, 5   # $s1 = $s2 | 5`

I-Type ALU Instructions

Instruction	Meaning	I-Type Format			
addi $s1, $s2, 10	$s1 = $s2 + 10	op = 0x8	rs = $s2	rt = $s1	imm^{16} = 10
addiu $s1, $s2, 10	$s1 = $s2 + 10	op = 0x9	rs = $s2	rt = $s1	imm^{16} = 10
andi $s1, $s2, 10	$s1 = $s2 & 10	op = 0xc	rs = $s2	rt = $s1	imm^{16} = 10
ori $s1, $s2, 10	$s1 = $s2 | 10	op = 0xd	rs = $s2	rt = $s1	imm^{16} = 10
xori $s1, $s2, 10	$s1 = $s2 ^ 10	op = 0xe	rs = $s2	rt = $s1	imm^{16} = 10
lui $s1, 10	$s1 = 10 << 16	op = 0xf	0	rt = $s1	imm^{16} = 10

- addi: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- addiu: same operation as addi but overflow is ignored
- Immediate constant for addi and addiu is signed
  - No need for **subi** or **subiu** instructions
- Immediate constant for andi, ori, xori is unsigned

Examples: I-Type ALU Instructions

- Examples: assume A, B, C are allocated $s0, $s1, $s2

```
A = B+5;      translated as     addiu $s0,$s1,5
C = B–1;      translated as     addiu $s2,$s1,-1
```

op=001001	rs=$s1=10001	rt=$s2=10010	imm = -1 = 1111111111111111

```
A = B&0xf;    translated as     andi  $s0,$s1,0xf
C = B|0xf;    translated as     ori   $s2,$s1,0xf
C = 5;        translated as     ori   $s2,$zero,5
A = B;        translated as     ori   $s0,$s1,0
```

Immediate value is the last value in the instruction

- No need for `subi`, because `addi` has signed immediate
- Register 0 (`$zero`) has always the value 0

32-bit Constants

❖ I-Type instructions can have only 16-bit constants

Op^6	Rs^5	Rt^5	immediate^{16}

❖ What if we want to load a 32-bit constant into a register?

❖ Can't have a 32-bit constant in I-Type instructions ☹

✧ We have already fixed the sizes of all instructions to 32 bits

❖ Solution: use two instructions instead of one ☺

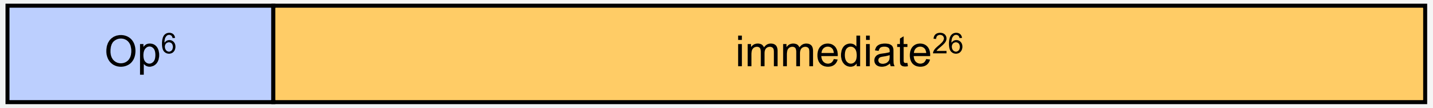
✧ Suppose we want: `$s1=0xAC5165D9` (32-bit constant)

✧ `lui`: **load upper immediate**

Instruction		load upper 16 bits	clear lower 16 bits
`lui $s1,0xAC51`	`$s1=$17`	`0xAC51`	`0x0000`
`ori $s1,$s1,0x65D9`	`$s1=$17`	`0xAC51`	`0x65D9`

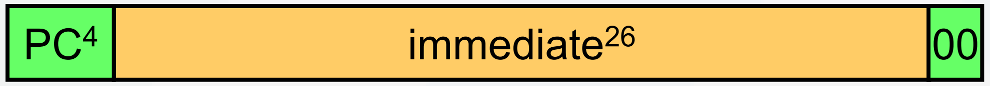
Next . . .

J-Type Format



- J-type format is used for unconditional jump instruction:

```
j    label    # jump  to label
   . . .
label:
```

- 26-bit immediate value is stored in the instruction
  - Immediate constant specifies address of target instruction
- Program Counter (PC) is modified as follows:
  - Next PC = 

least-significant 2 bits are 00
  - Upper 4 most significant bits of PC are unchanged

Conditional Branch Instructions

- MIPS compare and branch instructions:

`beq Rs,Rt,label`	branch to `label` if (`Rs == Rt`)
`bne Rs,Rt,label`	branch to `label` if (`Rs != Rt`)

- MIPS compare to zero & branch instructions

Compare to zero is used frequently and implemented efficiently

`bltz Rs,label`	branch to `label` if (`Rs < 0`)
`bgtz Rs,label`	branch to `label` if (`Rs > 0`)
`blez Rs,label`	branch to `label` if (`Rs <= 0`)
`bgez Rs,label`	branch to `label` if (`Rs >= 0`)

- No need for `beqz` and `bnez` instructions. Why?

Set on Less Than Instructions

- MIPS also provides set on less than instructions

`slt`	`rd,rs,rt`	if (rs < rt) rd = 1 else rd = 0
`sltu`	`rd,rs,rt`	unsigned <
`slti`	`rt,rs,`im^{16}	if (rs < im^{16}) rt = 1 else rt = 0
`sltiu`	`rt,rs,`im^{16}	unsigned <

- Signed / Unsigned Comparisons

  Can produce different results

  Assume `$s0 = 1` and `$s1 = -1 = 0xffffffff`

`slt  $t0,$s0,$s1`	results in	`$t0 = 0`
`sltu $t0,$s0,$s1`	results in	`$t0 = 1`

More on Branch Instructions

- MIPS hardware does NOT provide instructions for …

`blt, bltu`	branch if less than	(signed/unsigned)
`ble, bleu`	branch if less or equal	(signed/unsigned)
`bgt, bgtu`	branch if greater than	(signed/unsigned)
`bge, bgeu`	branch if greater or equal	(signed/unsigned)

Can be achieved with a sequence of 2 instructions

- How to implement: `blt $s0,$s1,label`
- Solution:

```
slt $at,$s0,$s1
bne $at,$zero,label
```

- How to implement: `ble $s2,$s3,label #s2≤s3 ⇔ s3≥s2`
- Solution:

```
slt $at,$s3,$s2 #(lower means smaller and not equal)
beq $at,$zero,label
```

Pseudo-Instructions

- ❖ Introduced by assembler as if they were real instructions
  - ✧ To facilitate assembly language programming

Pseudo-Instructions	Conversion to Real Instructions
`move  $s1, $s2`	`addu  Ss1, $s2, $zero`
`not   $s1, $s2`	`nor   $s1, $s2, $s2`
`li    $s1, 0xabcd`	`ori   $s1, $zero, 0xabcd`

`li    $s1, 0xabcd1234`	`lui   $s1, 0xabcd`
	`ori   $s1, $s1, 0x1234`

`sgt   $s1, $s2, $s3`	`slt   $s1, $s3, $s2`

`blt   $s1, $s2, label`	`slt   $at, $s1, $s2`
	`bne   $at, $zero, label`

- ❖ Assembler reserves \$at = \$1 for its own use
  - ✧ `$at` is called the assembler temporary register

Jump, Branch, and SLT Instructions

Instruction	Meaning	Format			
j label	jump to label	op^6 = 2	imm^{26}		
beq rs, rt, label	branch if (rs == rt)	op^6 = 4	rs^5	rt^5	imm^{16}
bne rs, rt, label	branch if (rs != rt)	op^6 = 5	rs^5	rt^5	imm^{16}
blez rs, label	branch if (rs<=0)	op^6 = 6	rs^5	0	imm^{16}
bgtz rs, label	branch if (rs > 0)	op^6 = 7	rs^5	0	imm^{16}
bltz rs, label	branch if (rs < 0)	op^6 = 1	rs^5	0	imm^{16}

Instruction	Meaning	Format					
slt rd, rs, rt	rd=(rs<rt?1:0)	op^6 = 0	rs^5	rt^5	rd^5	0	0x2a
sltu rd, rs, rt	rd=(rs<rt?1:0)	op^6 = 0	rs^5	rt^5	rd^5	0	0x2b
slti rt, rs, imm^{16}	rt=(rs<imm?1:0)	0xa	rs^5	rt^5	imm^{16}		
sltiu rt, rs, imm^{16}	rt=(rs<imm?1:0)	0xb	rs^5	rt^5	imm^{16}		

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Architecture
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Addressing Modes

Translating an IF Statement

❖ Consider the following IF statement:

```
if (a == b) c = d + e; else c = d – e;
```

Assume that a, b, c, d, e are in $s0, …, $s4 respectively

❖ How to translate the above IF statement?

```
        bne   $s0, $s1, else
        addu  $s2, $s3, $s4
        j     exit
else:   subu  $s2, $s3, $s4
exit:   . . .
```

Compound Expression with AND

❖ If first expression is false, second expression is skipped

```
if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

```
# One Possible Implementation ...
    bgtz   $s1, L1       # first expression
    j      next          # skip if false
L1: bltz   $s2, L2       # second expression
    j      next          # skip if false
L2: addiu  $s3,$s3,1     # both are true
next:
```

Better Implementation for AND

```
if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

The following implementation uses less code

Reverse the relational operator

Allow the program to fall through to the second expression

Number of instructions is reduced from 5 to 3

```
# Better Implementation ...
    blez    $s1, next     # skip if false
    bgez    $s2, next     # skip if false
    addiu   $s3,$s3,1     # both are true
next:
```

Compound Expression with OR

❖ If first expression is true, second expression is skipped

```
if (($s1 > $s2) || ($s2 > $s3)) {$s4 = 1;}
```

❖ Use fall-through to keep the code as short as possible

```
    bgt $s1, $s2, L1     # yes, execute if part
    ble $s2, $s3, next   # no: skip if part
L1: li  $s4, 1           # set $s4 to 1
next:
```

❖ `bgt`, `ble`, and `li` are pseudo-instructions

- ✧ Translated by the assembler to real instructions

Your Turn . . .

- Translate the IF statement to assembly language
- $s1 and $s2 values are unsigned

```
if( $s1 <= $s2 ) {
  $s3 = $s4
}
```

```
    bgtu $s1, $s2, next
    move $s3, $s4
next:
```

- $s3, $s4, and $s5 values are signed

```
if (($s3 <= $s4) &&
      ($s4 >  $s5)) {
  $s3 = $s4 + $s5
}
```

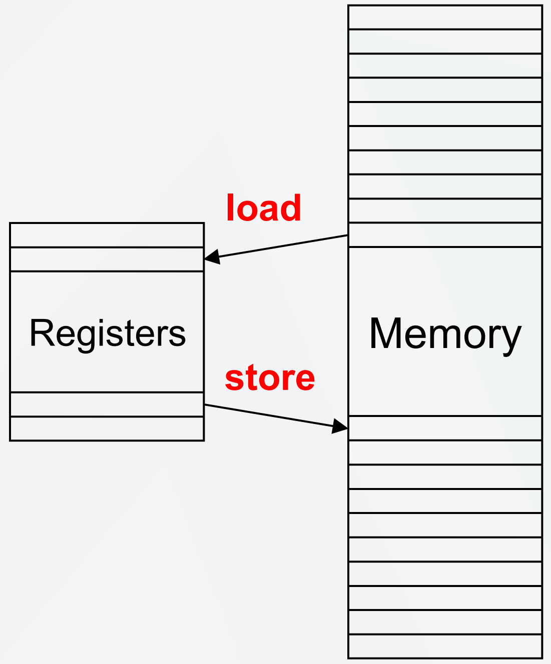
```
    bgt  $s3, $s4, next
    ble  $s4, $s5, next
    add $s3, $s4, $s5
next:
```

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Architecture
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Addressing Modes

Load and Store Instructions

- Instructions that transfer data between memory & registers
- Programs include variables such as arrays and objects
- Such variables are stored in memory
- Load Instruction:
  - Transfers data from memory to a register
- Store Instruction:
  - Transfers data from a register to memory
- Memory address must be specified by load and store



Load and Store Word

❖ Load Word Instruction (Word = 4 bytes in MIPS)

`lw Rt, imm`^{16}`(Rs)   # Rt ← MEMORY[Rs+imm`^{16}`]`

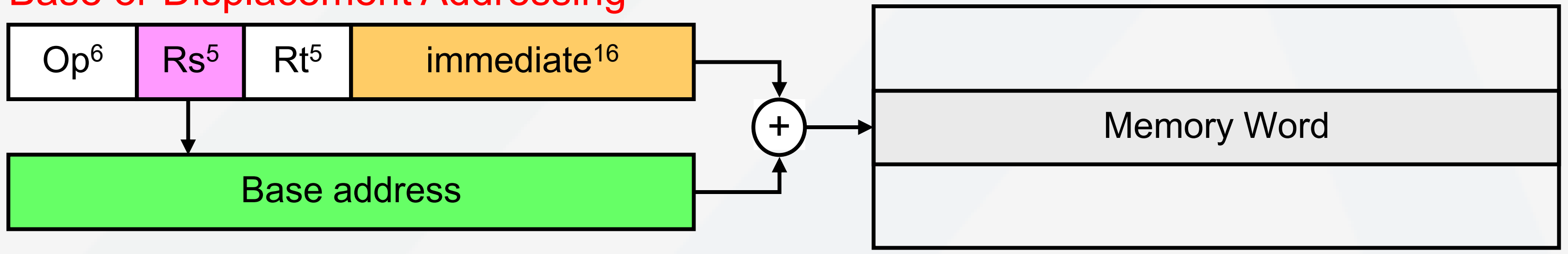
❖ Store Word Instruction

`sw Rt, imm`^{16}`(Rs)   # Rt → MEMORY[Rs+imm`^{16}`]`

❖ Base or Displacement addressing is used

- ✧ Memory Address = Rs (base) + Immediate^{16} (displacement)
- ✧ Immediate^{16} is sign-extended to have a signed displacement

Base or Displacement Addressing

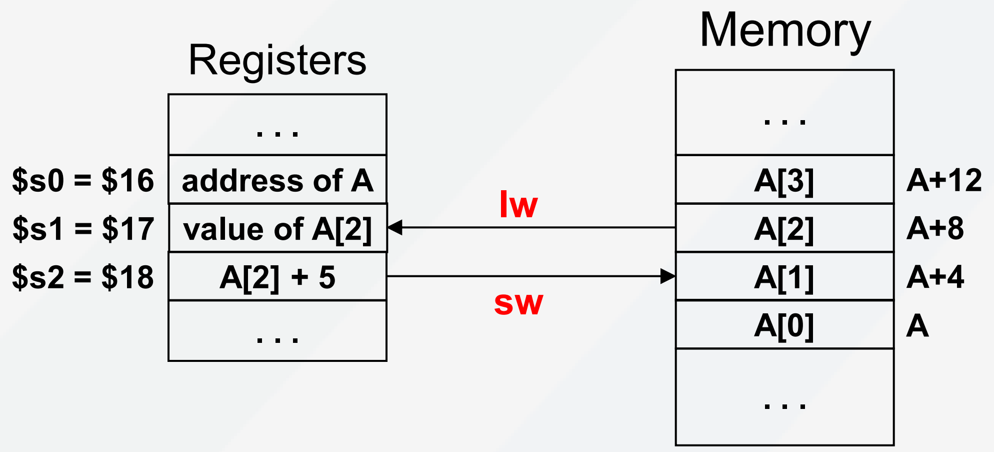


Example on Load & Store

- Translate A[1] = A[2] + 5 (A is an array of words)
  - Assume that address of array A is stored in register $s0

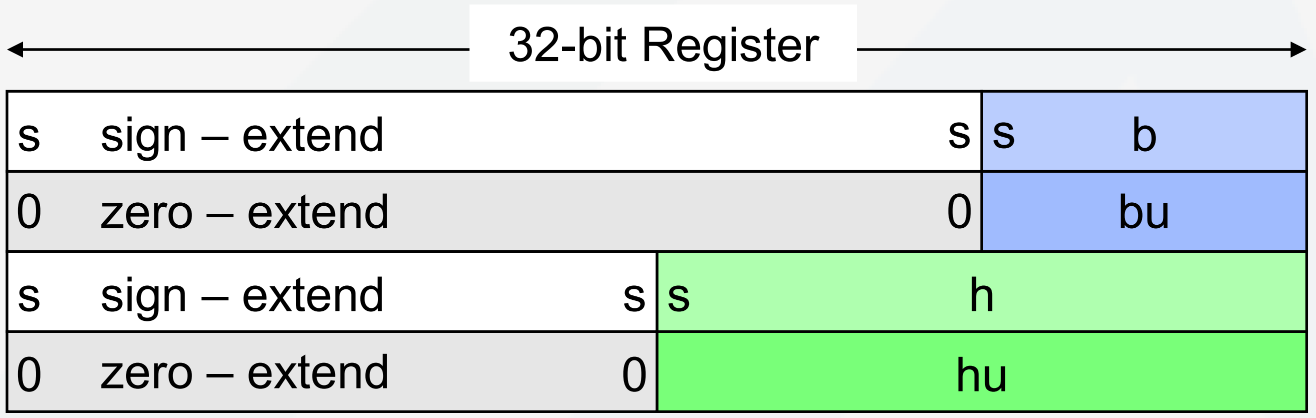
```
lw      $s1, 8($s0)          # $s1 = A[2]
addiu   $s2, $s1, 5          # $s2 = A[2] + 5
sw      $s2, 4($s0)          # A[1] = $s2
```

- Index of *a*[2] and *a*[1] should be multiplied by 4. Why?



Load and Store Byte and Halfword

- ❖ The MIPS processor supports the following data formats:
  - ✧ Byte = 8 bits, Halfword = 16 bits, Word = 32 bits
- ❖ Load & store instructions for bytes and halfwords
  - ✧ lb = load byte, lbu = load byte unsigned, sb = store byte
  - ✧ lh = load half, lhu = load half unsigned, sh = store halfword
- ❖ Load expands a memory data to fit into a 32-bit register
- ❖ Store reduces a 32-bit register to fit in memory



Load and Store Instructions

Instruction	Meaning	I-Type Format			
lb rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x20	rs^5	rt^5	imm^{16}
lh rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x21	rs^5	rt^5	imm^{16}
lw rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x23	rs^5	rt^5	imm^{16}
lbu rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x24	rs^5	rt^5	imm^{16}
lhu rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x25	rs^5	rt^5	imm^{16}
sb rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x28	rs^5	rt^5	imm^{16}
sh rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x29	rs^5	rt^5	imm^{16}
sw rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x2b	rs^5	rt^5	imm^{16}

- Base or Displacement Addressing is used
  - Memory Address = Rs (base) + Immediate^{16} (displacement)
- Two variations on base addressing
  - If Rs = \$zero = 0 then Address = Immediate^{16} (absolute)
  - If Immediate^{16} = 0 then Address = Rs (register indirect)

Next . . .

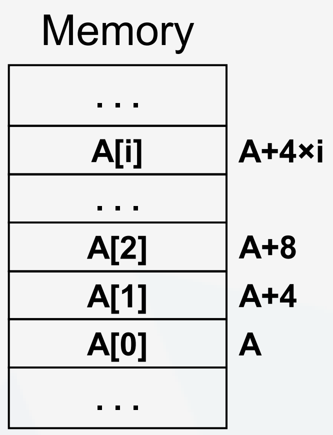
Translating a WHILE Loop

❖ Consider the following WHILE statement:

```
i = 0; while (A[i] != k) i = i+1;
```

Where A is an array of integers (4 bytes per element)

Assume address A, i, k in $s0, $s1, $s2, respectively



❖ How to translate above WHILE statement?

```
        xor    $s1, $s1, $s1     # i = 0
        move   $t0, $s0          # $t0 = address A
loop:   lw     $t1, 0($t0)       # $t1 = A[i]
        beq    $t1, $s2, exit    # exit if (A[i]== k)
        addiu  $s1, $s1, 1       # i = i+1
        sll    $t0, $s1, 2       # $t0 = 4*i
        addu   $t0, $s0, $t0     # $t0 = address A[i]
        j      loop
exit:   . . .
```

Using Pointers to Traverse Arrays

❖ Consider the same WHILE loop:

```
i = 0; while (A[i] != k) i = i+1;
```

Where address of A, i, k are in $s0, $s1, $s2, respectively

❖ We can use a pointer to traverse array A

Pointer is incremented by 4 (faster than indexing)

```
      move   $t0, $s0         # $t0 = $s0 = addr A
      j      cond             # test condition
loop: addiu  $s1, $s1, 1      # i = i+1
      addiu  $t0, $t0, 4      # point to next
cond: lw     $t1, 0($t0)      # $t1 = A[i]
      bne    $t1, $s2, loop   # loop if A[i]!= k
```

❖ Only 4 instructions (rather than 6) in loop body

Copying a String

The following code copies source string to target string

Address of source in $s0 and address of target in $s1

Strings are terminated with a null character (C strings)

```
i = 0;
do {target[i]=source[i]; i++;} while (source[i]!=0);
```

```
    move   $t0, $s0         # $t0 = pointer to source
    move   $t1, $s1         # $t1 = pointer to target
L1: lb     $t2, 0($t0)      # load  byte into $t2
    sb     $t2, 0($t1)      # store byte into target
    addiu  $t0, $t0, 1      # increment source pointer
    addiu  $t1, $t1, 1      # increment target pointer
    bne    $t2, $zero, L1   # loop until NULL char
```

Summing an Integer Array

```
sum = 0;
for (i=0; i<n; i++) sum = sum + A[i];
```

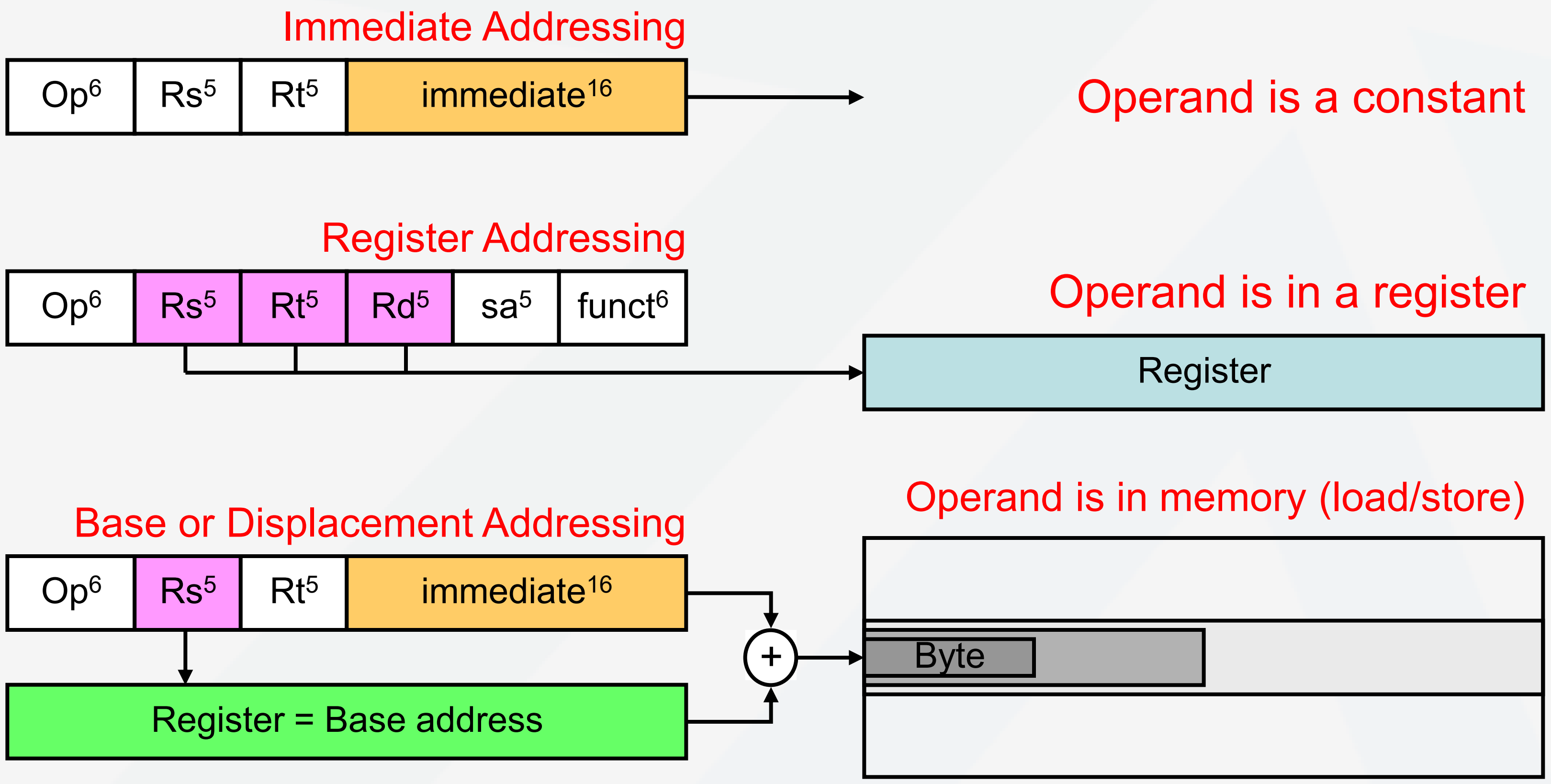
Assume $s0 = array address, $s1 = array length = n (n>0)

```
    move    $t0, $s0          # $t0 = address A[i]
    xor     $t1, $t1, $t1     # $t1 = i = 0
    xor     $s2, $s2, $s2     # $s2 = sum = 0
L1: lw      $t2, 0($t0)       # $t2 = A[i]
    addu    $s2, $s2, $t2     # sum = sum + A[i]
    addiu   $t0, $t0, 4       # point to next A[i]
    addiu   $t1, $t1, 1       # i++
    bne     $t1, $s1, L1      # loop if (i != n)
```

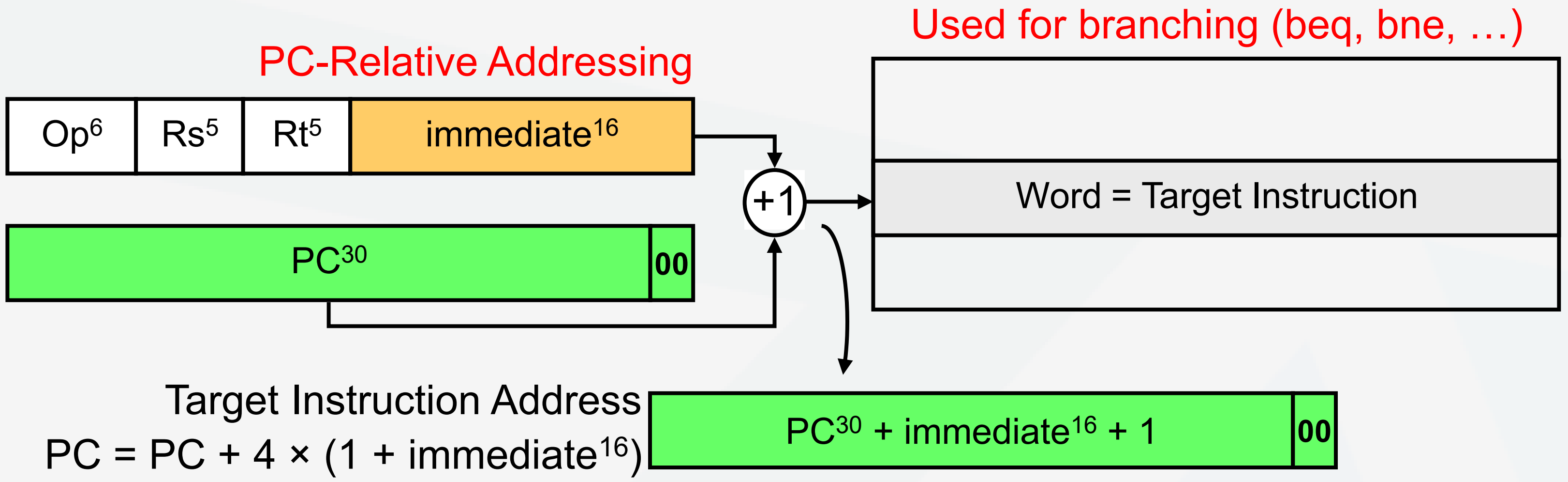
Next . . .

Addressing Modes

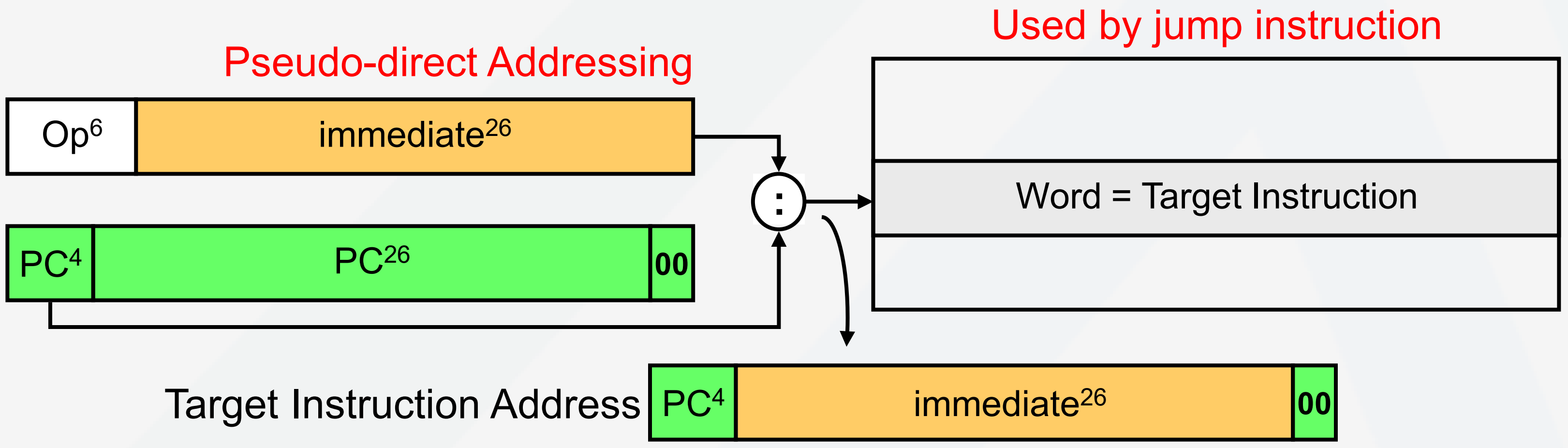
- Where are the **operands**?
- How memory addresses are computed?



Branch / Jump Addressing Modes

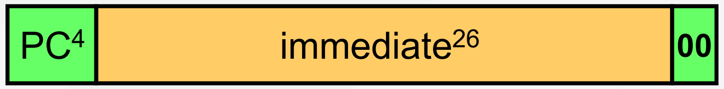


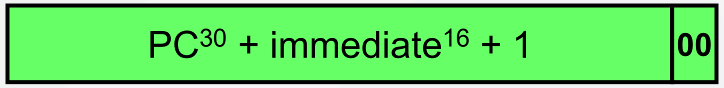
Used for branching (beq, bne, …)
PC-Relative Addressing
Op^6
Rs^5
Rt^5
immediate^{16}
+1
Word = Target Instruction
PC^{30}
00
Target Instruction Address
PC = PC + 4 × (1 + immediate^{16})
PC^{30} + immediate^{16} + 1
00



Jump and Branch Limits

- ❖ Jump Address Boundary = 2^{26} instructions = 256 MB
  - ✧ Text segment cannot exceed 2^{26} instructions or 256 MB
  - ✧ Upper 4 bits of PC are unchanged

Target Instruction Address 

- ❖ Branch Address Boundary
  - ✧ Branch instructions use I-Type format (16-bit immediate constant)
  - ✧ PC-relative addressing: 

    - Target instruction address = PC + 4×(1 + immediate^{16})
    - Count number of instructions to branch from next instruction
    - Positive constant => Forward Branch, Negative => Backward branch
    - At most $\pm 2^{15}$ instructions to branch (most branches are near)

Summary of RISC Design

- All instructions are typically of one size
- Few instruction formats
- Most operations on data are register to register
  - Operands are read from registers
  - Result is stored in a register
- General purpose integer and floating point registers
  - Typically, 32 integer and 32 floating-point registers
- Memory access only via load and store instructions
  - Load and store: bytes, half words, words, and double words
- Few simple addressing modes

Presentation Outline (Part2)

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Assembly Language Statements

- ❖ Three types of statements in assembly language
  - ✧ Typically, one statement should appear on a line

1. Executable Instructions
   - ✧ Generate machine code for the processor to execute at runtime
   - ✧ Instructions tell the processor what to do
2. Pseudo-Instructions and Macros
   - ✧ Translated by the assembler into real instructions
   - ✧ Simplify the programmer task
3. Assembler Directives
   - ✧ Provide information to the assembler while translating a program
   - ✧ Used to define segments, allocate memory variables, etc.
   - ✧ Non-executable: directives are not part of the instruction set

Instructions

- Assembly language instructions have the format:

```
[label:]   mnemonic   [operands]    [#comment]
```

- Label: (optional)
  - Marks the address of a memory location, must have a colon
  - Typically appear in data and text segments
- Mnemonic
  - Identifies the operation (e.g. **add**, **sub**, etc.)
- Operands
  - Specify the data required by the operation
  - Operands can be registers, memory variables, or constants
  - Most instructions have three operands

```
L1:   addiu $t0, $t0, 1        #increment $t0
```

Comments

- ❖ Comments are very important!
  - ✧ Explain the <u>program's</u> <u>purpose</u>
  - ✧ <u>When</u> it was <u>written</u>, <u>revised</u>, and by <u>whom</u>
  - ✧ Explain <u>data</u> <u>used</u> in the program, <u>input</u>, and <u>output</u>
  - ✧ Explain <u>instruction</u> <u>sequences</u> and <u>algorithms</u> used
  - ✧ Comments are also required at the <u>beginning</u> <u>of</u> <u>every</u> <u>procedure</u>
    - ▪ Indicate <u>input</u> <u>parameters</u> and <u>results</u> of a procedure
    - ▪ Describe <u>what</u> the <u>procedure</u> <u>does</u>
- ❖ Single-line comment
  - ✧ Begins with a <u>hash</u> symbol **#** and <u>terminates at end of line</u>

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

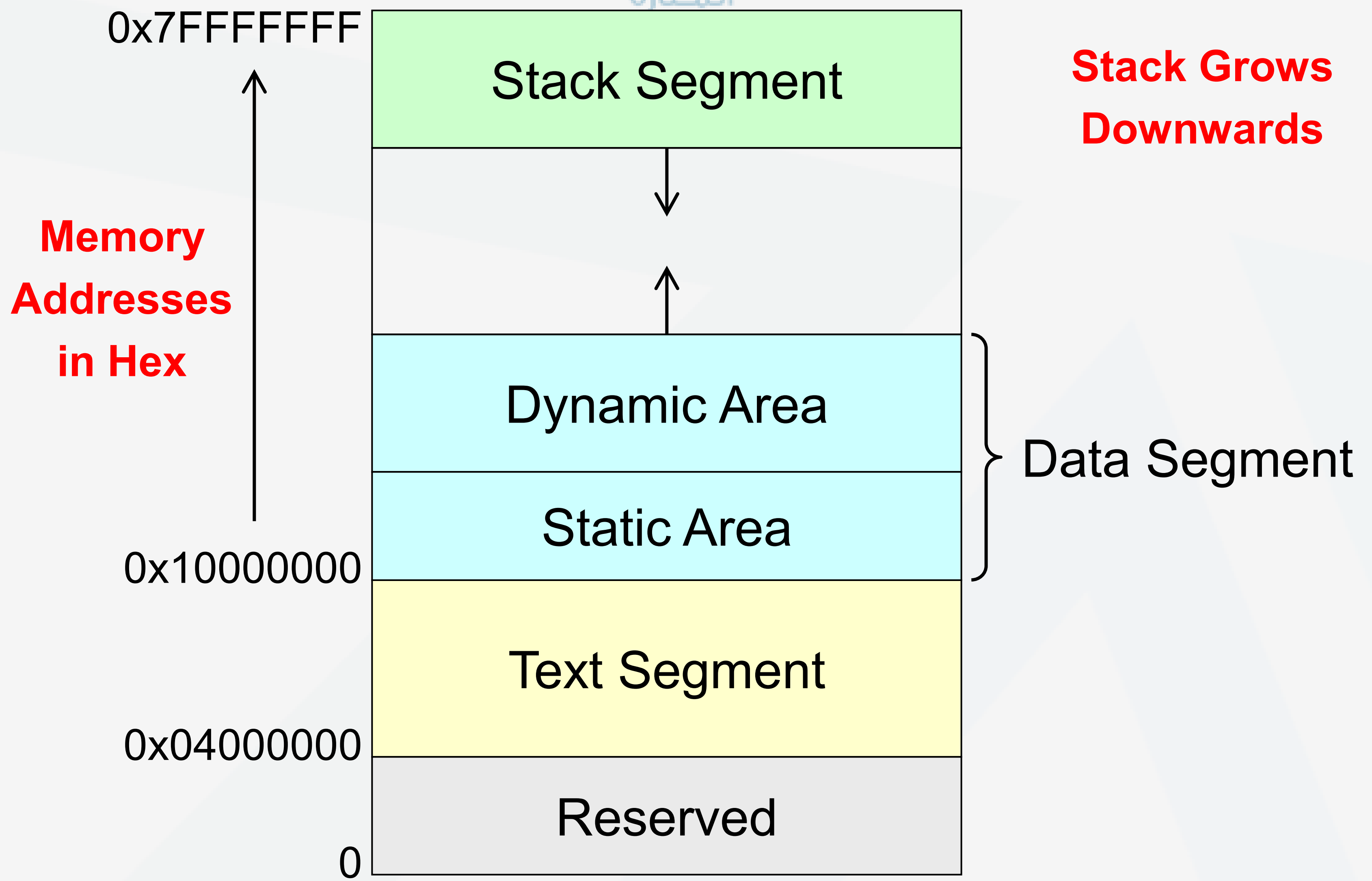
Program Template

```
# Title:                          Filename:
# Author:                         Date:
# Description:
# Input:
# Output:
################# Data segment #####################
.data
 . . .
################# Code segment #####################
.text
.globl main
main:                             # main program entry
 . . .
li $v0, 10                        # Exit program
syscall
```

.DATA, .TEXT, & .GLOBL Directives

- **.DATA** directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.TEXT** directive
  - Defines the code segment of a program containing instructions
- **.GLOBL** directive
  - Declares a symbol as global
  - Global symbols can be referenced from other files
  - We use this directive to declare *main* procedure of a program

Layout of a Program in Memory



Next . . .

Data Definition Statement

- ❖ Sets aside storage in memory for a variable
- ❖ May optionally assign a name (label) to the data
- ❖ Syntax:

[*name:*] *directive* *initializer* [, *initializer*] . . .

```
var1:  .WORD     10
```

- ❖ All initializers become binary data in memory

Data Directives

- **.BYTE** Directive
    - Stores the list of values as 8-bit bytes
- **.HALF** Directive
    - Stores the list as 16-bit values aligned on half-word boundary
- **.WORD** Directive
    - Stores the list as 32-bit values aligned on a word boundary
- **.WORD w:n** Directive
    - Stores the 32-bit value *w* into *n* consecutive words aligned on a word boundary.

Data Directives

- **.FLOAT** Directive
  - Stores the listed values as single-precision floating point (32bit)
- **.DOUBLE** Directive
  - Stores the listed values as double-precision floating point (64bit)

String Directives

- **.ASCII** Directive
  - Allocates a sequence of bytes for an ASCII string
- **.ASCIIZ** Directive
  - Same as **.ASCII** directive, but adds a NULL char at end of string
  - Strings are null-terminated, as in the C programming language
- **.SPACE n** Directive
  - Allocates space of *n* uninitialized bytes in the data segment
- Special characters in strings follow C convention
  - Newline: \n Tab:\t Quote: \”

Examples of Data Definitions

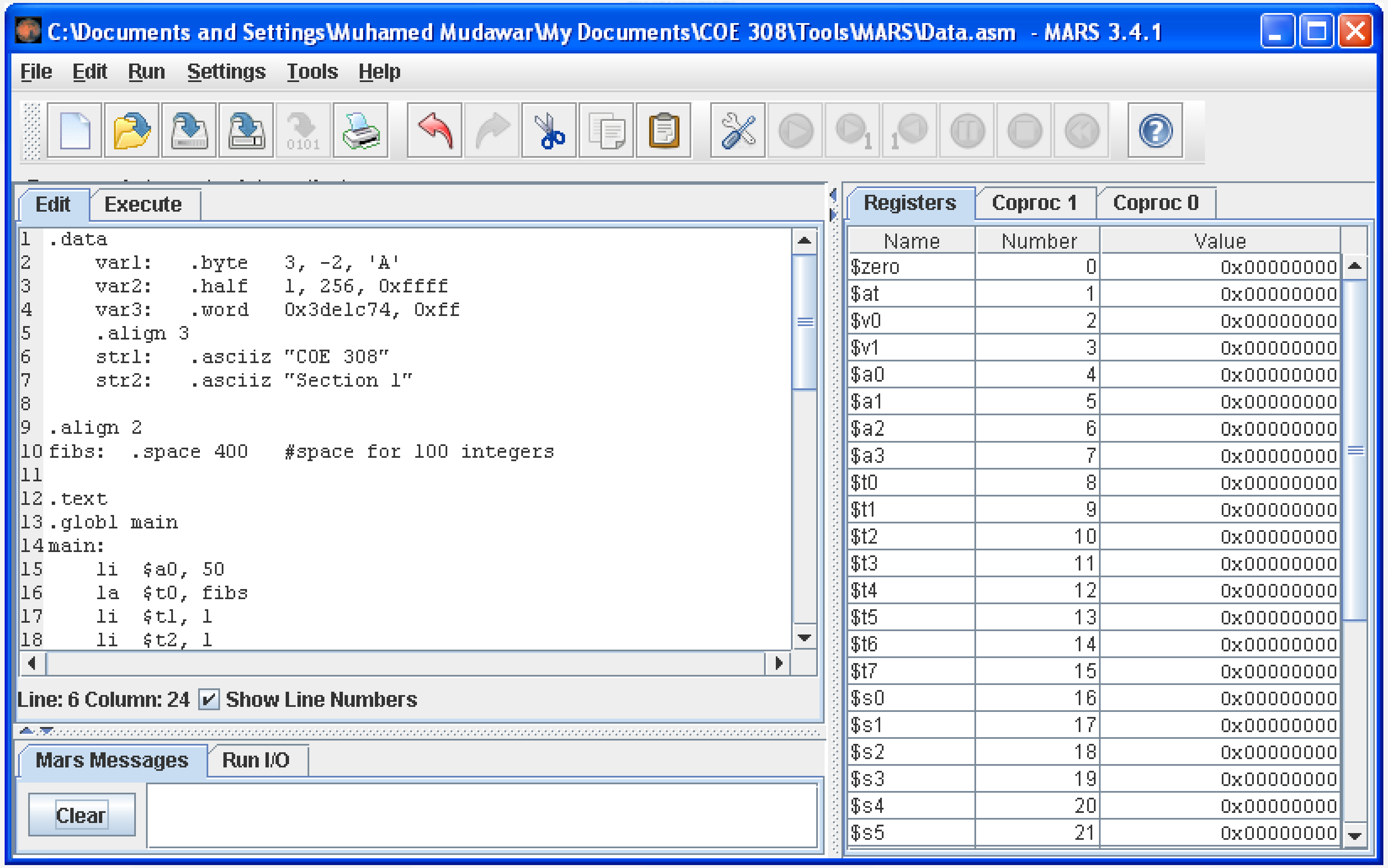
```
.DATA

var1:   .BYTE     'A', 'E', 127, -1, '\n'

var2:   .HALF     -10, 0xffff

var3:   .WORD     0x12345678
var4:   .WORD     0:10

var5:   .FLOAT    12.3, -0.1

var6:   .DOUBLE   1.5e-10

str1:   .ASCII    "A String\n"

str2:   .ASCIIZ   "NULL Terminated String"

array:  .SPACE    100
```

Array of 10 words

If the initial value exceeds the maximum size, an error is reported by assembler

100 bytes (not initialized)

MARS Assembler and Simulator Tool



C:\Documents and Settings\Muhamed Mudawar\My Documents\COE 308\Tools\MARS\Data.asm - MARS 3.4.1
File Edit Run Settings Tools Help
Edit
Execute
Registers
Coproc 1
Coproc 0
Mars Messages
Run I/O
Clear

```
.data
    var1:   .byte   3, -2, 'A'
    var2:   .half   1, 256, 0xffff
    var3:   .word   0x3de1c74, 0xff
    .align 3
    str1:   .asciiz "COE 308"
    str2:   .asciiz "Section 1"

.align 2
fibs:  .space 400    #space for 100 integers

.text
.globl main
main:
    li  $a0, 50
    la  $t0, fibs
    li  $t1, 1
    li  $t2, 1
```

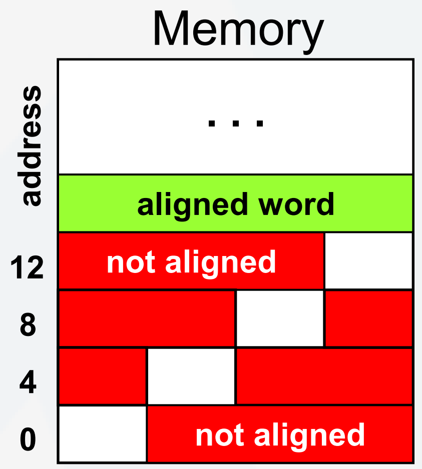
Line: 6 Column: 24 ✓ Show Line Numbers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000

Next . . .

Memory Alignment

- Memory is viewed as an array of bytes with addresses
  - Byte Addressing: address points to a byte in memory
- Words occupy 4 consecutive bytes in memory
  - MIPS instructions and integers occupy 4 bytes
- Alignment: address is a multiple of size
  - Word address should be a multiple of **4**
    - Least significant 2 bits of address should be **00**
  - Halfword address should be a multiple of **2**
- **`.ALIGN n`** directive
  - Aligns the next data definition on a 2^n byte boundary



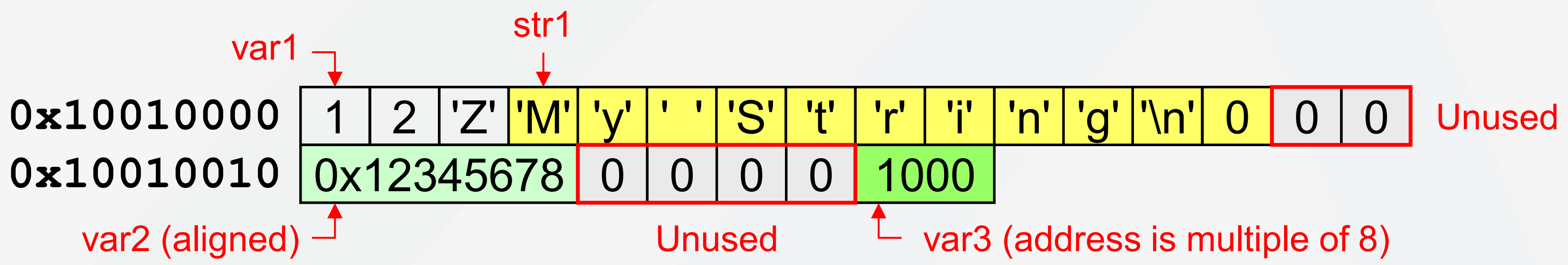
Symbol Table

- Assembler builds a symbol table for labels (variables)
  - Assembler computes the address of each label in data segment
- Example

```
.DATA
var1:   .BYTE   1, 2,'Z'
str1:   .ASCIIZ "My String\n"
var2:   .WORD   0x12345678
.ALIGN  3
var3:   .HALF   1000
```

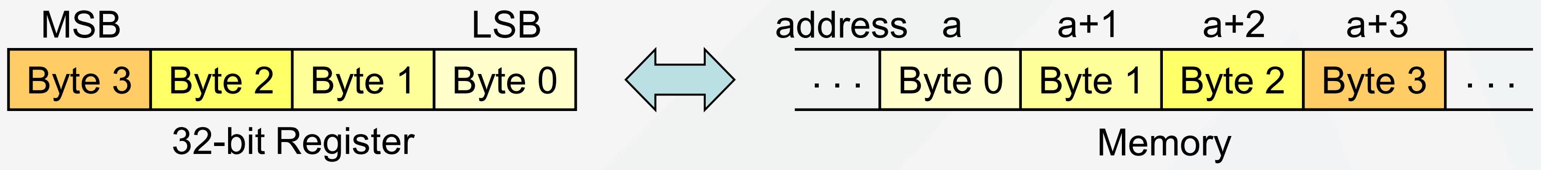
Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018

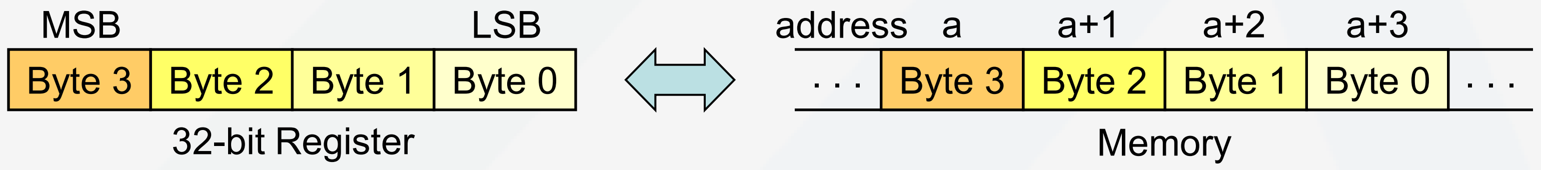


Byte Ordering and Endianness

- ❖ Processors can order bytes within a word in two ways
- ❖ Little Endian Byte Ordering
  - ✧ Memory address = Address of **least significant byte**
  - ✧ Example: Intel IA-32, Alpha



- ❖ Big Endian Byte Ordering
  - ✧ Memory address = Address of **most significant byte**
  - ✧ Example: SPARC, PA-RISC



- ❖ MIPS can operate with both byte orderings

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

System Calls

- Programs do input/output through system calls
- MIPS provides a special `syscall` instruction
  - To obtain services from the operating system
  - Many services are provided in the SPIM and MARS simulators
- Using the `syscall` system services
  - Load the service number in register $v0
  - Load argument values, if any, in registers $a0, $a1, etc.
  - Issue the `syscall` instruction
  - Retrieve return values, if any, from result registers

Syscall Services

Service	$v0	Arguments / Result
Print Integer	1	$a0 = integer value to print
Print Float	2	$f12 = float value to print
Print Double	3	$f12 = double value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	$v0 = integer read
Read Float	6	$f0 = float read
Read Double	7	$f0 = double read
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Exit Program	10	
Print Char	11	$a0 = character to print
Read Char	12	$a0 = character read

Supported by MARS

Reading and Printing an Integer

```
################# Code segment #####################
.text
.globl main
main:                                   # main program entry
  li    $v0, 5                          # Read integer
  syscall                               # $v0 = value read

  move  $a0, $v0                        # $a0 = value to print
  li    $v0, 1                          # Print integer
  syscall

  li    $v0, 10                         # Exit program
  syscall
```

Reading and Printing a String

```
################# Data segment #####################
.data
  str: .space  10          # array of 10 bytes
################# Code segment #####################
.text
.globl main
main:                      # main program entry
  la    $a0, str           # $a0 = address of str
  li    $a1, 10            # $a1 = max string length
  li    $v0, 8             # read string
  syscall
  li    $v0, 4             # Print string str
  syscall
  li    $v0, 10            # Exit program
  syscall
```

Program 1: Sum of Three Integers

```
# Sum of three integers
#
# Objective: Computes the sum of three integers.
#     Input: Requests three numbers.
#    Output: Outputs the sum.
################### Data segment ###################
.data
prompt:   .asciiz     "Please enter three numbers: \n"
sum_msg:  .asciiz     "The sum is: "
################### Code segment ###################
.text
.globl main
main:
      la    $a0,prompt          # display prompt string
      li    $v0,4
      syscall
      li    $v0,5               # read 1st integer into $t0
      syscall
      move  $t0,$v0
```

Sum of Three Integers - Slide 2 of 2

```
li     $v0,5                 # read 2nd integer into $t1
syscall
move   $t1,$v0

li     $v0,5                 # read 3rd integer into $t2
syscall
move   $t2,$v0

addu   $t0,$t0,$t1           # accumulate the sum
addu   $t0,$t0,$t2

la     $a0,sum_msg           # write sum message
li     $v0,4
syscall

move   $a0,$t0               # output sum
li     $v0,1
syscall

li     $v0,10                # exit
syscall
```

Program 2: Case Conversion (Homework)

```
# Objective: Convert lowercase letters to uppercase
#     Input: Requests a character string from the user.
#    Output: Prints the input string in uppercase.
################### Data segment #####################
.data
name_prompt:  .asciiz         "Please type your name:  "
out_msg:      .asciiz         "Your name in capitals is:  "
in_name:      .space 31       # space for input string
################### Code segment #####################
.text
.globl main
main:
      la    $a0,name_prompt   # print prompt string
      li    $v0,4
      syscall
      la    $a0,in_name       # read the input string
      li    $a1,31            # at most 30 chars + 1 null char
      li    $v0,8
      syscall
```

Case Conversion - Slide 2 of 2

```
      la    $a0,out_msg       # write output message
      li    $v0,4
      syscall
      la    $t0,in_name
loop:
      lb    $t1,($t0)
      beqz  $t1,exit_loop     # if NULL, we are done
      blt   $t1,'a',no_change
      bgt   $t1,'z',no_change
      addiu $t1,$t1,-32       # convert to uppercase: 'A'-'a'=-32
      sb    $t1,($t0)
no_change:
      addiu $t0,$t0,1         # increment pointer
      j     loop
exit_loop:
      la    $a0,in_name       # output converted string
      li    $v0,4
      syscall
      li    $v0,10            # exit
      syscall
```

Exercises

Exercise1

```
.data
A: .word 1, 2, 3, 4, 5, 6
B: .word 3, 3, 0, 4, 4, 4
.text
lui $s0, 0x1001
lui $s1, 0x1001
addi $s1, $s1, 24
start:
lw $t0, 0($s0)
lw $t1, 0($s1)
beq $t0, $t1, end
sw $t0, 0($s1)
sw $t1, 0($s0)
addi $s0, $s0, 4
addi $s1, $s1, 4
j start
end:
addi $t1, $t1, 2
or $v0, $t0, $t1
li $v0,10
syscall
```

1- ما هو تمثيل ذاكرة المعطيات قبل تنفيذ الكود

2- ما هو العنوان الذي تبدأ به المصفوفة B

3- ماهي قيمة المسجل $s0 قبل الدخول في الحلقة

4- ماهي قيمة المسجل $s1 قبل الدخول في الحلقة

5- ماهي قيمة المسجل $t0عند المرور الاول في الحلقة

6- ماهي قيمة المسجل $s1 عند المرور الاول بالحلقة

7- ما هو عدد مرات تنفيذ الحلقة

8- ما هو عدد مرات تنفيذ التعليمة (sw $t0 ,0($s1

9- ماهي الصيغة الست عشرية الموافقة للتعليمة (lw $t1,0($s1

11- ماهي الصيغة الست عشرية الموافقة للتعليمة beq $t0, $t1, end

11- ماهي الصيغة الست عشرية الموافقة للتعليمة or $v0, $t0, $t1

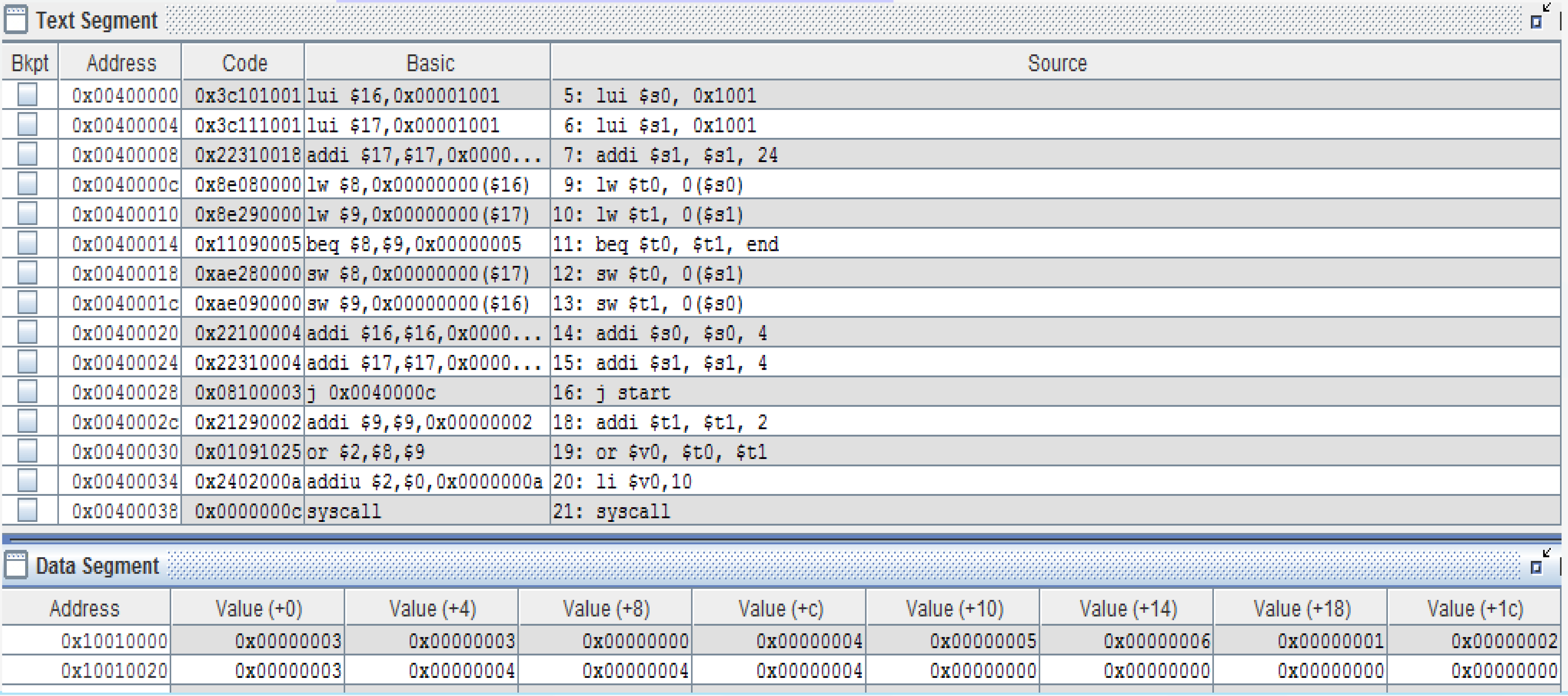
12- ماهي قيم عناصر المصفوفة Aبعد تنفيذ الكود

13- ماهي قيم عناصر المصفوفة Bبعد تنفيذ الكود

14- ما هي قيمة المسجل $v0 بعد تنفيذ الكود

15- ما هو العنوان الذي تدل عليه اللافتة start.

Exercise1



Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c101001	lui $16,0x00001001	5: lui $s0, 0x1001
☐	0x00400004	0x3c111001	lui $17,0x00001001	6: lui $s1, 0x1001
☐	0x00400008	0x22310018	addi $17,$17,0x0000...	7: addi $s1, $s1, 24
☐	0x0040000c	0x8e080000	lw $8,0x00000000($16)	9: lw $t0, 0($s0)
☐	0x00400010	0x8e290000	lw $9,0x00000000($17)	10: lw $t1, 0($s1)
☐	0x00400014	0x11090005	beq $8,$9,0x00000005	11: beq $t0, $t1, end
☐	0x00400018	0xae280000	sw $8,0x00000000($17)	12: sw $t0, 0($s1)
☐	0x0040001c	0xae090000	sw $9,0x00000000($16)	13: sw $t1, 0($s0)
☐	0x00400020	0x22100004	addi $16,$16,0x0000...	14: addi $s0, $s0, 4
☐	0x00400024	0x22310004	addi $17,$17,0x0000...	15: addi $s1, $s1, 4
☐	0x00400028	0x08100003	j 0x0040000c	16: j start
☐	0x0040002c	0x21290002	addi $9,$9,0x00000002	18: addi $t1, $t1, 2
☐	0x00400030	0x01091025	or $2,$8,$9	19: or $v0, $t0, $t1
☐	0x00400034	0x2402000a	addiu $2,$0,0x0000000a	20: li $v0,10
☐	0x00400038	0x0000000c	syscall	21: syscall

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000003	0x00000003	0x00000000	0x00000004	0x00000005	0x00000006	0x00000001	0x00000002
0x10010020	0x00000003	0x00000004	0x00000004	0x00000004	0x00000000	0x00000000	0x00000000	0x00000000

Exercise2

```
.data
A: .word 1, 2, 3, 4, 5, 6
B: .word 3, 3, 0, 4, 4, 4
.text
lui $s0, 0x1001
lui $s1, 0x1001
addi $s1, $s1, 24
start:
lw $t0, 0($s0)
lw $t1, 0($s1)
beq $t0, $t1, end
sw $t0, 0($s1)
sw $t1, 0($s0)
addi $s0, $s0, 4
addi $s1, $s1, 4
j start
end:
addi $t1, $t1, 2
or $v0, $t0, $t1
li $v0,10
syscall
```

1-ما هو تمثيل ذاكرة المعطيات قبل تنفيذ الكود

2- ما هو العنوان الذي تبدأ به المصفوفة B

3- ماهي قيمة المسجل $s0 قبل الدخول في الحلقة

4- ماهي قيمة المسجل $s1قبل الدخول في الحلقة

5- ماهي قيمة المسجل $t0عند المرور الاول في الحلقة

6- ماهي قيمة المسجل $s1عند المرور الاول بالحلقة

7- ما هو عدد مرات تنفيذ الحلقة

8- ما هو عدد مرات تنفيذ التعليمة sw $t0 ,0($s1)

9- ماهي الصيغة الست عشرية الموافقة للتعليمة lw $t1,0($s1)

11- ماهي الصيغة الست عشرية الموافقة للتعليمة beq $t0, $t1, end

11- ماهي الصيغة الست عشرية الموافقة للتعليمة or $v0, $t0, $t1

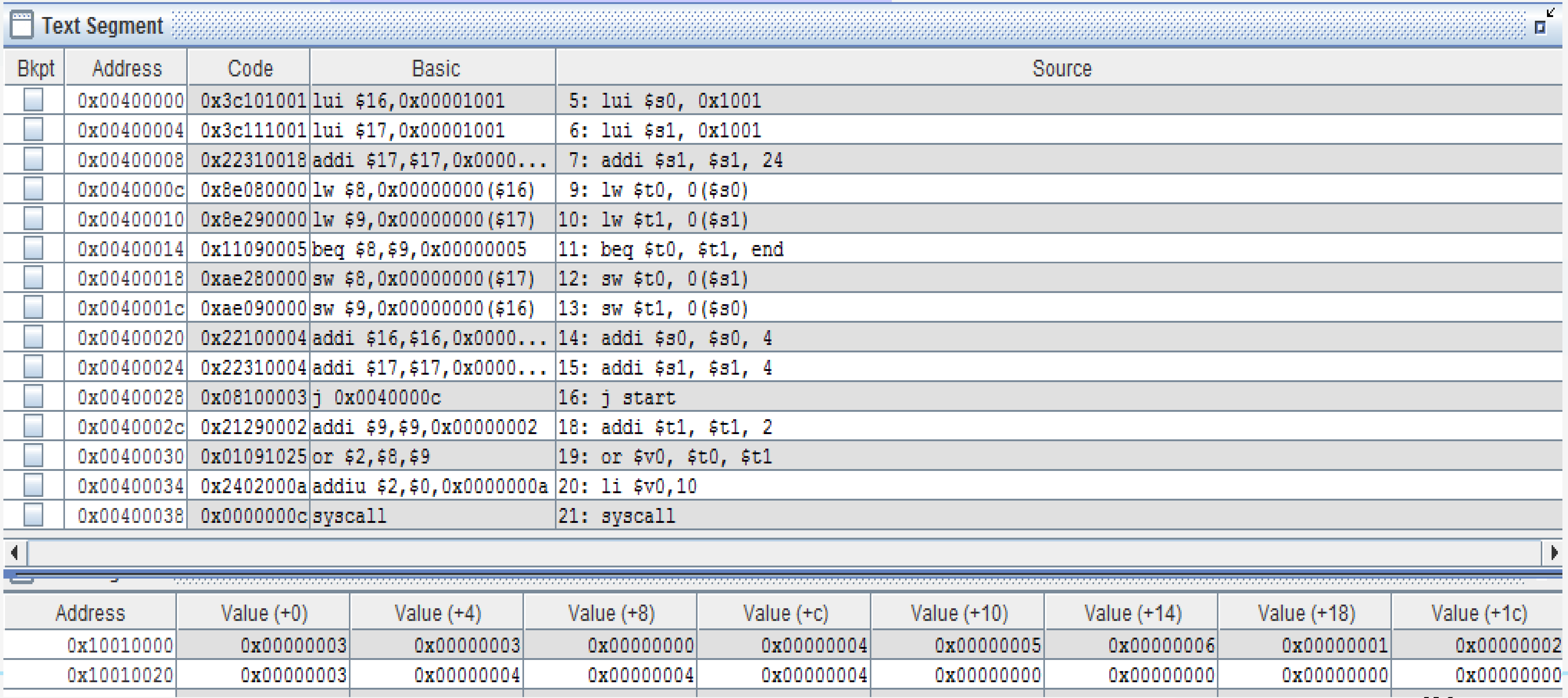
12- ماهي قيم عناصر المصفوفة Aبعد تنفيذ الكود

13- ماهي قيم عناصر المصفوفة Bبعد تنفيذ الكود

14- ما هي قيمة المسجل $v0بعد تنفيذ الكود

15- ما هو العنوان الذي تدل عليه اللافتة start

Exercise2



Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c101001	lui $16,0x00001001	5: lui $s0, 0x1001
☐	0x00400004	0x3c111001	lui $17,0x00001001	6: lui $s1, 0x1001
☐	0x00400008	0x22310018	addi $17,$17,0x0000...	7: addi $s1, $s1, 24
☐	0x0040000c	0x8e080000	lw $8,0x00000000($16)	9: lw $t0, 0($s0)
☐	0x00400010	0x8e290000	lw $9,0x00000000($17)	10: lw $t1, 0($s1)
☐	0x00400014	0x11090005	beq $8,$9,0x00000005	11: beq $t0, $t1, end
☐	0x00400018	0xae280000	sw $8,0x00000000($17)	12: sw $t0, 0($s1)
☐	0x0040001c	0xae090000	sw $9,0x00000000($16)	13: sw $t1, 0($s0)
☐	0x00400020	0x22100004	addi $16,$16,0x0000...	14: addi $s0, $s0, 4
☐	0x00400024	0x22310004	addi $17,$17,0x0000...	15: addi $s1, $s1, 4
☐	0x00400028	0x08100003	j 0x0040000c	16: j start
☐	0x0040002c	0x21290002	addi $9,$9,0x00000002	18: addi $t1, $t1, 2
☐	0x00400030	0x01091025	or $2,$8,$9	19: or $v0, $t0, $t1
☐	0x00400034	0x2402000a	addiu $2,$0,0x0000000a	20: li $v0,10
☐	0x00400038	0x0000000c	syscall	21: syscall

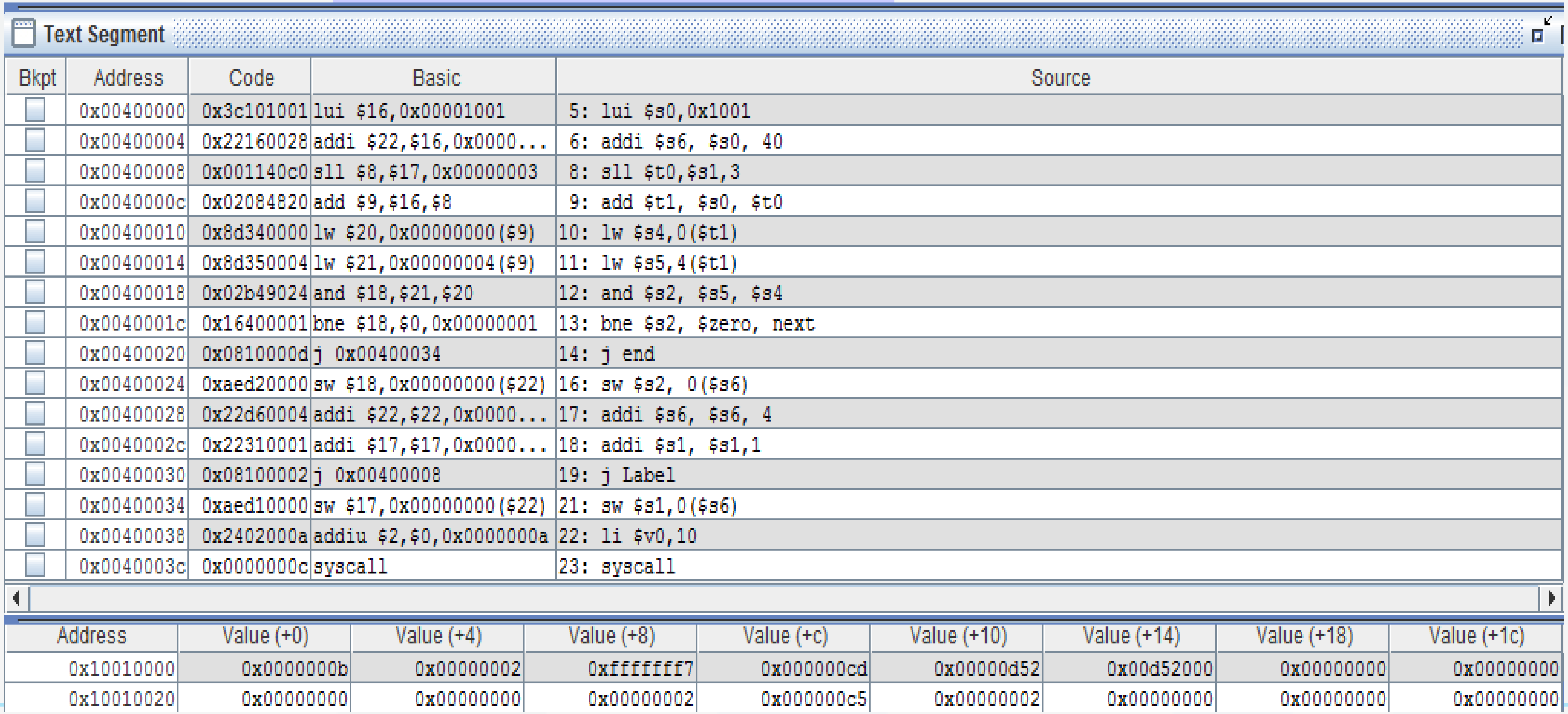
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000003	0x00000003	0x00000000	0x00000004	0x00000005	0x00000006	0x00000001	0x00000002
0x10010020	0x00000003	0x00000004	0x00000004	0x00000004	0x00000000	0x00000000	0x00000000	0x00000000

Exercise3

```
.data
A: .word 11, 2, -9, 0xcd, 0xd52, 0xd52000
B: .word 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
.text
lui $s0,0x1001
addi $s6, $s0, 40
Label:
sll $t0,$s1,3
add $t1, $s0, $t0
lw $s4,0($t1)
lw $s5,4($t1)
and $s2, $s5, $s4
bne $s2, $zero, next
j end
next:
sw $s2, 0($s6)
addi $s6, $s6, 4
addi $s1, $s1,1
j Label
end:
sw $s1,0($s6)
li $v0,10
syscall
```

1 ما هو العنوان الذي تبدأ به المصفوفة B؟
2- ماهي قيمة المسجل $s6قبل الدخول في الحلقة؟
3- ماهي قيمة المسجل $t1بعد انتهاء التنفيذ الأول للحلقة؟
4- ماهي قيمة المسجل $s2بعد انتهاء التنفيذ الأول للحلقة؟
5- ماهي قيمة المسجل $t0بعد انتهاء التنفيذ الثاني للحلقة؟
6- ما هو عدد مرات تنفيذ الحلقة ؟
7- ماهي قيم عناصر المصفوفة Bبعد انتهاء تنفيذ البرنامج؟
8- ماهي القيمة المكافئة للافتة nextالموجودة في التعليمة bne؟
9- بفرض أن عنوان اول تعليمة في الكود0x00851000
10- ، اكتب الصيغة الست عشرية للتعليمة j label
11-اكتب الصيغة الست عشرية للتعليمة sw $s2,0($s6)

Exercise3



Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c101001	lui $16,0x00001001	5: lui $s0,0x1001
☐	0x00400004	0x22160028	addi $22,$16,0x0000...	6: addi $s6, $s0, 40
☐	0x00400008	0x001140c0	sll $8,$17,0x00000003	8: sll $t0,$s1,3
☐	0x0040000c	0x02084820	add $9,$16,$8	9: add $t1, $s0, $t0
☐	0x00400010	0x8d340000	lw $20,0x00000000($9)	10: lw $s4,0($t1)
☐	0x00400014	0x8d350004	lw $21,0x00000004($9)	11: lw $s5,4($t1)
☐	0x00400018	0x02b49024	and $18,$21,$20	12: and $s2, $s5, $s4
☐	0x0040001c	0x16400001	bne $18,$0,0x00000001	13: bne $s2, $zero, next
☐	0x00400020	0x0810000d	j 0x00400034	14: j end
☐	0x00400024	0xaed20000	sw $18,0x00000000($22)	16: sw $s2, 0($s6)
☐	0x00400028	0x22d60004	addi $22,$22,0x0000...	17: addi $s6, $s6, 4
☐	0x0040002c	0x22310001	addi $17,$17,0x0000...	18: addi $s1, $s1,1
☐	0x00400030	0x08100002	j 0x00400008	19: j Label
☐	0x00400034	0xaed10000	sw $17,0x00000000($22)	21: sw $s1,0($s6)
☐	0x00400038	0x2402000a	addiu $2,$0,0x0000000a	22: li $v0,10
☐	0x0040003c	0x0000000c	syscall	23: syscall

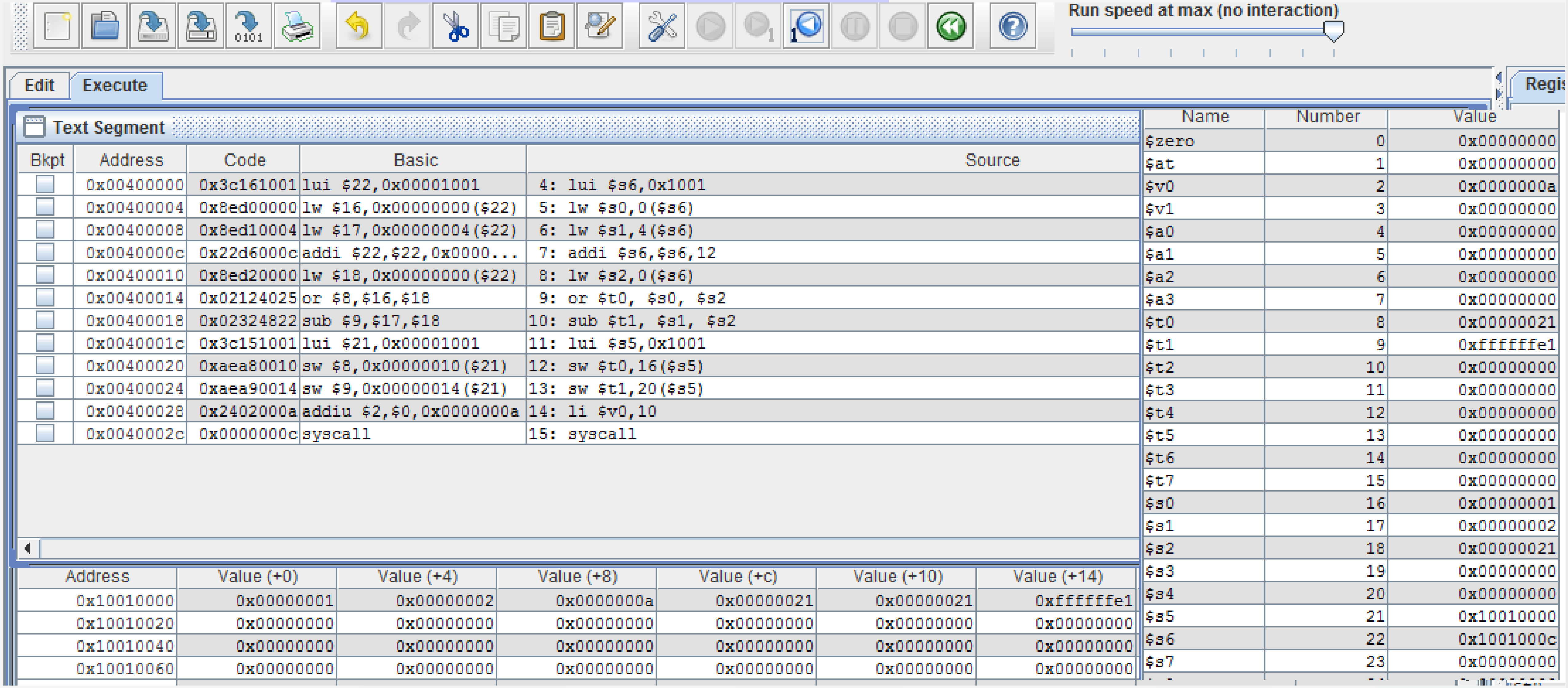
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x0000000b	0x00000002	0xfffffff7	0x000000cd	0x00000d52	0x00d52000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000002	0x000000c5	0x00000002	0x00000000	0x00000000	0x00000000

Exercise4

```
.data
.word 1, 2, 0xa, 33
.text
lui $s6,0x1001
lw $s0,0($s6)
lw $s1,4($s6)
addi $s6,$s6,12
lw $s2,0($s6)
or $t0, $s0, $s2
sub $t1, $s1, $s2
lui $s5,0x1001
sw $t0,16($s5)
sw $t1,20($s5)
li $v0,10
syscall
```

ما هي قيم المسجلات المُستخدمة في البرنامج بعد إنهاء تنفيذ البرنامج؟

Exercise4



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c161001	lui $22,0x00001001	4: lui $s6,0x1001
☐	0x00400004	0x8ed00000	lw $16,0x00000000($22)	5: lw $s0,0($s6)
☐	0x00400008	0x8ed10004	lw $17,0x00000004($22)	6: lw $s1,4($s6)
☐	0x0040000c	0x22d6000c	addi $22,$22,0x0000...	7: addi $s6,$s6,12
☐	0x00400010	0x8ed20000	lw $18,0x00000000($22)	8: lw $s2,0($s6)
☐	0x00400014	0x02124025	or $8,$16,$18	9: or $t0, $s0, $s2
☐	0x00400018	0x02324822	sub $9,$17,$18	10: sub $t1, $s1, $s2
☐	0x0040001c	0x3c151001	lui $21,0x00001001	11: lui $s5,0x1001
☐	0x00400020	0xaea80010	sw $8,0x00000010($21)	12: sw $t0,16($s5)
☐	0x00400024	0xaea90014	sw $9,0x00000014($21)	13: sw $t1,20($s5)
☐	0x00400028	0x2402000a	addiu $2,$0,0x0000000a	14: li $v0,10
☐	0x0040002c	0x0000000c	syscall	15: syscall

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010000	0x00000001	0x00000002	0x0000000a	0x00000021	0x00000021	0xffffffe1
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

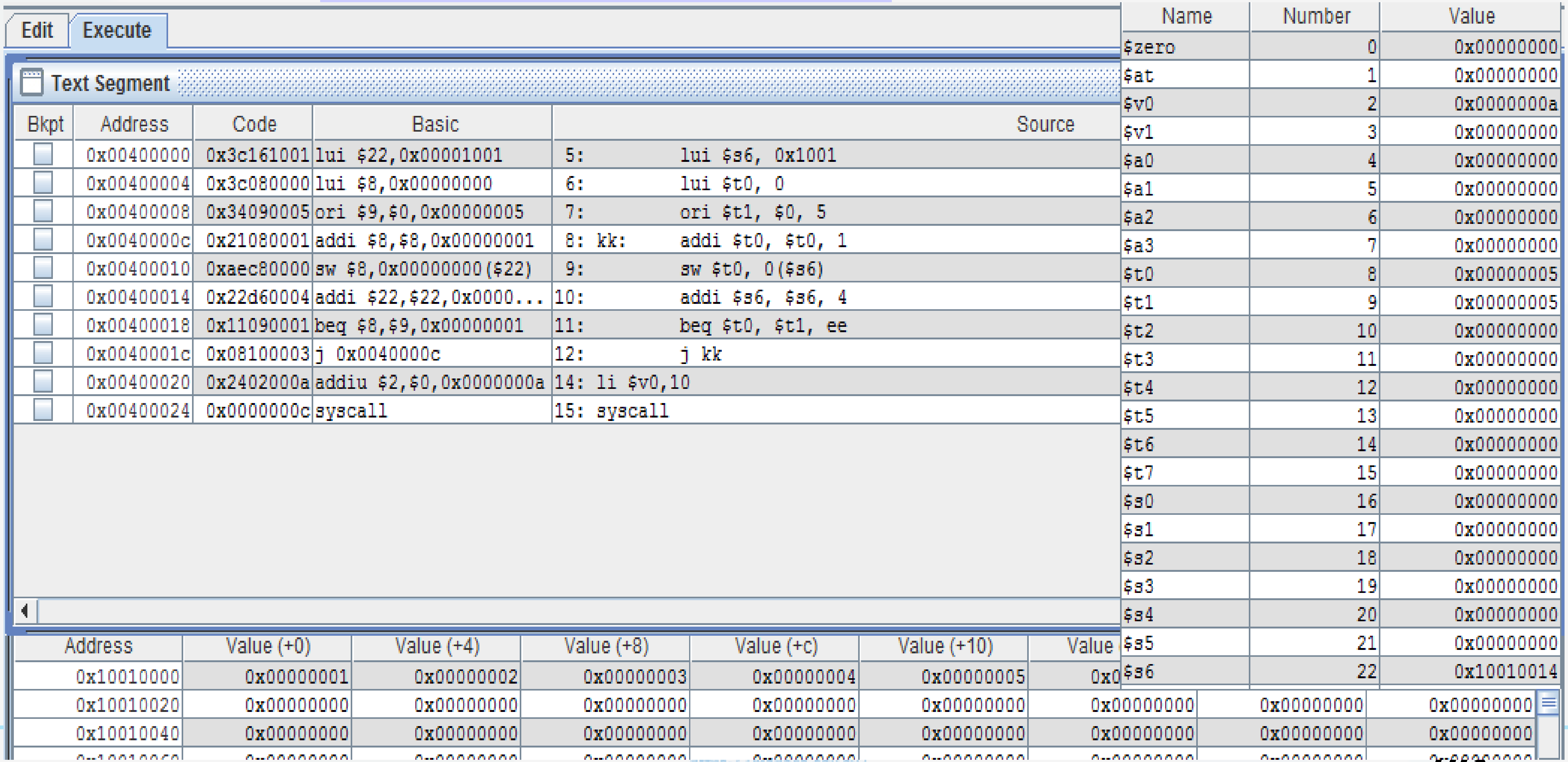
Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000021
$t1	9	0xffffffe1
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000001
$s1	17	0x00000002
$s2	18	0x00000021
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x10010000
$s6	22	0x1001000c
$s7	23	0x00000000

Exercise5

```
.data
.word 0, 0, 0, 0
.word 0, 0 ,0
.text
        lui $s6, 0x1001
        lui $t0, 0
        ori $t1, $0, 5
kk:     addi $t0, $t0, 1
        sw $t0, 0($s6)
        addi $s6, $s6, 4
        beq $t0, $t1, ee
        j kk
ee:
li $v0,10
syscall
```

ما هي قيم المسجلات المُستخدمة في البرنـامج بعد إنـهاء تنفيذ البرنـامج؟

Exercise5



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c161001	lui $22,0x00001001	5: lui $s6, 0x1001
☐	0x00400004	0x3c080000	lui $8,0x00000000	6: lui $t0, 0
☐	0x00400008	0x34090005	ori $9,$0,0x00000005	7: ori $t1, $0, 5
☐	0x0040000c	0x21080001	addi $8,$8,0x00000001	8: kk: addi $t0, $t0, 1
☐	0x00400010	0xaec80000	sw $8,0x00000000($22)	9: sw $t0, 0($s6)
☐	0x00400014	0x22d60004	addi $22,$22,0x0000...	10: addi $s6, $s6, 4
☐	0x00400018	0x11090001	beq $8,$9,0x00000001	11: beq $t0, $t1, ee
☐	0x0040001c	0x08100003	j 0x0040000c	12: j kk
☐	0x00400020	0x2402000a	addiu $2,$0,0x0000000a	14: li $v0,10
☐	0x00400024	0x0000000c	syscall	15: syscall

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value		
0x10010000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x0		
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000005
$t1	9	0x00000005
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x10010014

Exercise6

```
.data
.word 1, 2, 3, 4, 5
.word 0, 0, 0, 0, 0
.text
lui $s6, 0x1001
lui $s5, 0x1001
addi $s5, $s5, 36
ori $t2, $0, 5
tt: lw $t0, 0($s6)
sw $t0, 0($s5)
addi $s6, $s6, 4
addi $s5, $s5, -4
beq $t0, $t2, ee
j tt
ee:
li $v0,10
syscall
```

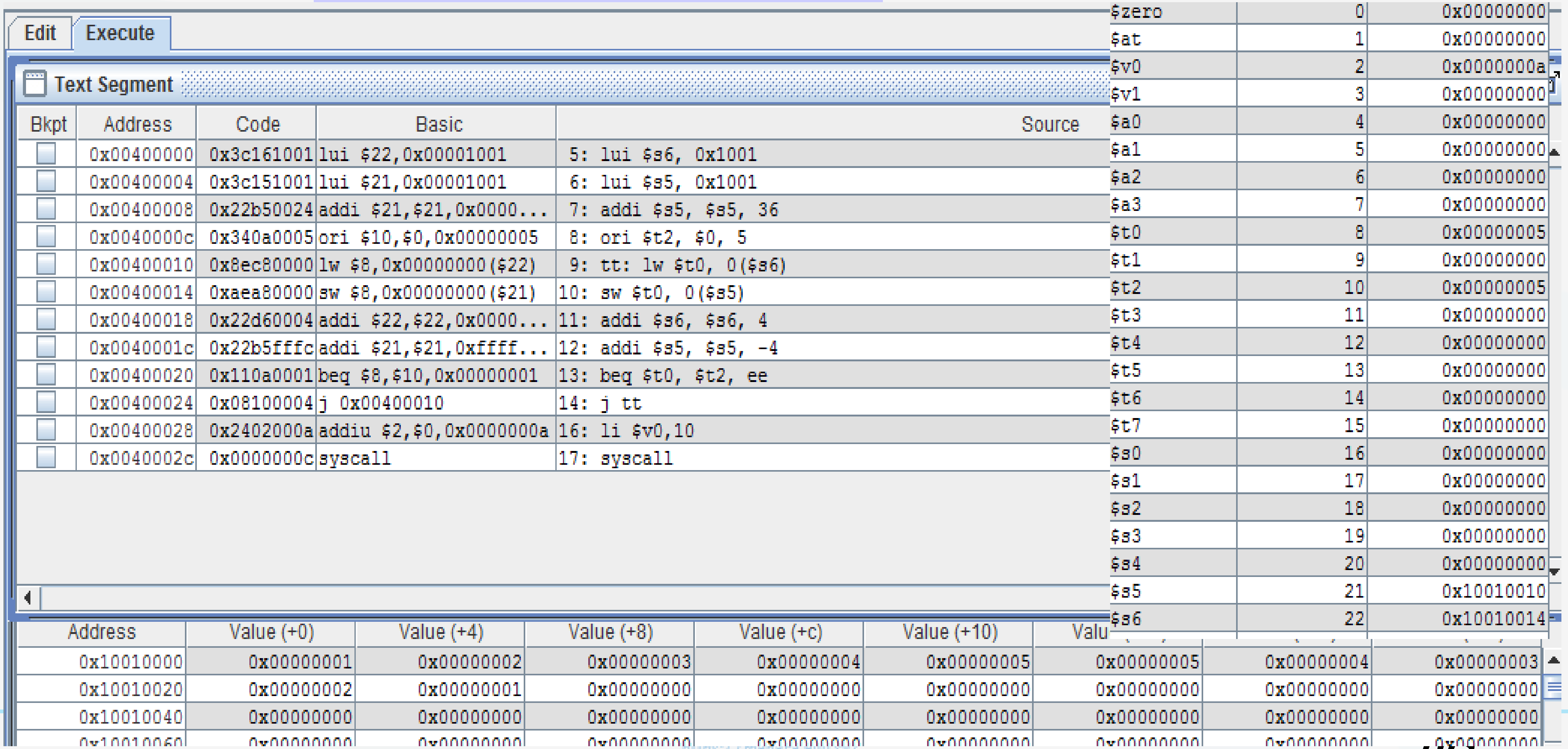
1- ما هي قيمة المسجلات قبل الدخول في الحلقة؟

2- كم مرة يتم تنفيذ الحلقة؟

3- حدد القيمة النهائية للمسجلات بعد إنهاء تنفيذ البرنامج؟

4- ما هي الغاية من هذا المقطع البرمجي؟

Exercise6



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c161001	lui $22,0x00001001	5: lui $s6, 0x1001
☐	0x00400004	0x3c151001	lui $21,0x00001001	6: lui $s5, 0x1001
☐	0x00400008	0x22b50024	addi $21,$21,0x0000...	7: addi $s5, $s5, 36
☐	0x0040000c	0x340a0005	ori $10,$0,0x00000005	8: ori $t2, $0, 5
☐	0x00400010	0x8ec80000	lw $8,0x00000000($22)	9: tt: lw $t0, 0($s6)
☐	0x00400014	0xaea80000	sw $8,0x00000000($21)	10: sw $t0, 0($s5)
☐	0x00400018	0x22d60004	addi $22,$22,0x0000...	11: addi $s6, $s6, 4
☐	0x0040001c	0x22b5fffc	addi $21,$21,0xffff...	12: addi $s5, $s5, -4
☐	0x00400020	0x110a0001	beq $8,$10,0x00000001	13: beq $t0, $t2, ee
☐	0x00400024	0x08100004	j 0x00400010	14: j tt
☐	0x00400028	0x2402000a	addiu $2,$0,0x0000000a	16: li $v0,10
☐	0x0040002c	0x0000000c	syscall	17: syscall

$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000005
$t1	9	0x00000000
$t2	10	0x00000005
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x10010010
$s6	22	0x10010014

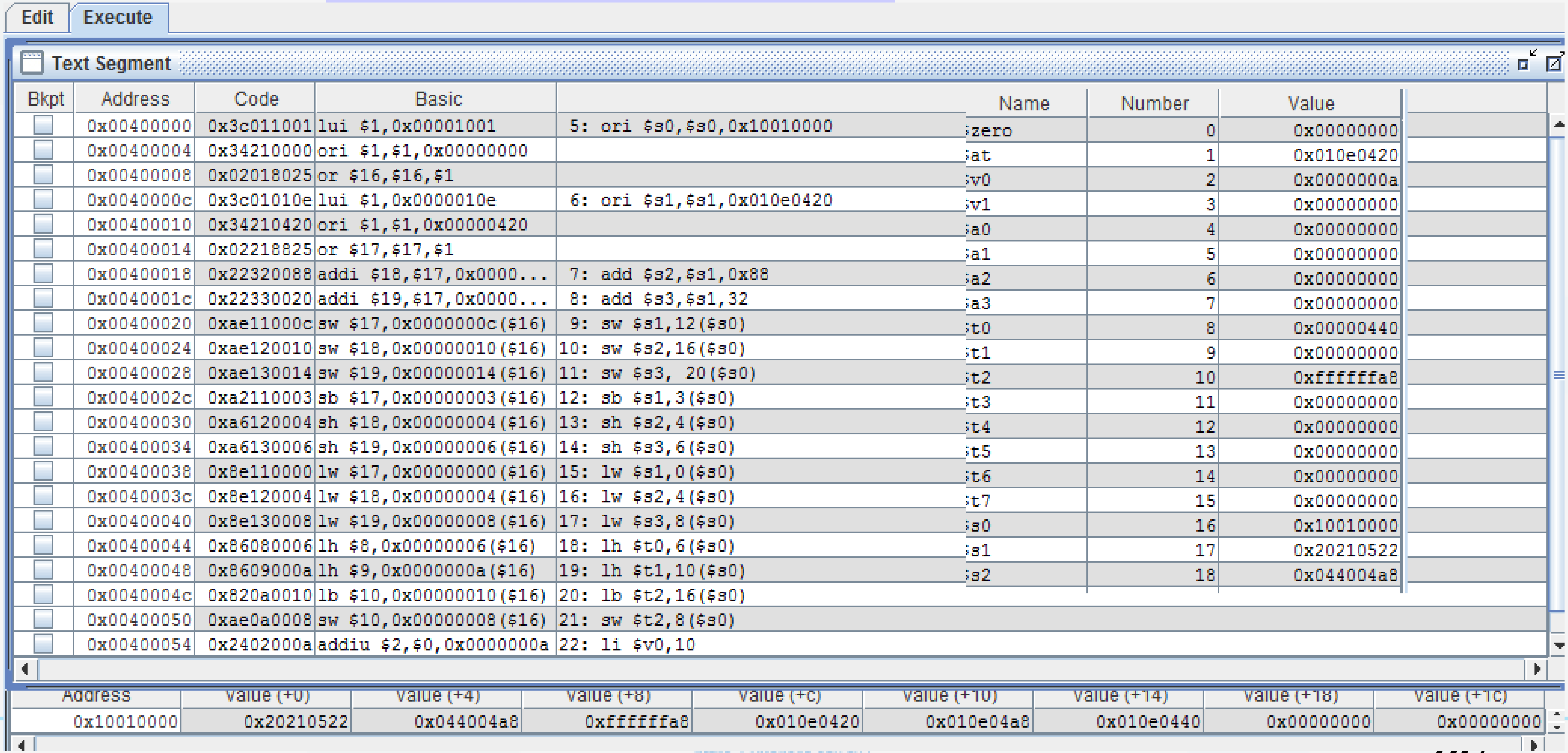
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Valu		
0x10010000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000005	0x00000004	0x00000003
0x10010020	0x00000002	0x00000001	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Exercise7

حدد القيمة النهائية للمسجلات بعد إنهاء تنفيذ البرنامج؟

```
.data
.byte 0x22, 0x5, 33
.space 5
.text
ori $s0,$s0,0x10010000
ori $s1,$s1,0x010e0420
add $s2,$s1,0x88
add $s3,$s1,32
sw $s1,12($s0)
sw $s2,16($s0)
sw $s3, 20($s0)
sb $s1,3($s0)
sh $s2,4($s0)
sh $s3,6($s0)
lw $s1,0($s0)
lw $s2,4($s0)
lw $s3,8($s0)
lh $t0,6($s0)
lh $t1,10($s0)
lb $t2,16($s0)
sw $t2,8($s0)
li $v0,10
syscall
```

Exercise7



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	
☐	0x00400000	0x3c011001	lui $1,0x00001001	5: ori $s0,$s0,0x10010000
☐	0x00400004	0x34210000	ori $1,$1,0x00000000	
☐	0x00400008	0x02018025	or $16,$16,$1	
☐	0x0040000c	0x3c01010e	lui $1,0x0000010e	6: ori $s1,$s1,0x010e0420
☐	0x00400010	0x34210420	ori $1,$1,0x00000420	
☐	0x00400014	0x02218825	or $17,$17,$1	
☐	0x00400018	0x22320088	addi $18,$17,0x0000...	7: add $s2,$s1,0x88
☐	0x0040001c	0x22330020	addi $19,$17,0x0000...	8: add $s3,$s1,32
☐	0x00400020	0xae11000c	sw $17,0x0000000c($16)	9: sw $s1,12($s0)
☐	0x00400024	0xae120010	sw $18,0x00000010($16)	10: sw $s2,16($s0)
☐	0x00400028	0xae130014	sw $19,0x00000014($16)	11: sw $s3, 20($s0)
☐	0x0040002c	0xa2110003	sb $17,0x00000003($16)	12: sb $s1,3($s0)
☐	0x00400030	0xa6120004	sh $18,0x00000004($16)	13: sh $s2,4($s0)
☐	0x00400034	0xa6130006	sh $19,0x00000006($16)	14: sh $s3,6($s0)
☐	0x00400038	0x8e110000	lw $17,0x00000000($16)	15: lw $s1,0($s0)
☐	0x0040003c	0x8e120004	lw $18,0x00000004($16)	16: lw $s2,4($s0)
☐	0x00400040	0x8e130008	lw $19,0x00000008($16)	17: lw $s3,8($s0)
☐	0x00400044	0x86080006	lh $8,0x00000006($16)	18: lh $t0,6($s0)
☐	0x00400048	0x8609000a	lh $9,0x0000000a($16)	19: lh $t1,10($s0)
☐	0x0040004c	0x820a0010	lb $10,0x00000010($16)	20: lb $t2,16($s0)
☐	0x00400050	0xae0a0008	sw $10,0x00000008($16)	21: sw $t2,8($s0)
☐	0x00400054	0x2402000a	addiu $2,$0,0x0000000a	22: li $v0,10

Name	Number	Value
zero	0	0x00000000
at	1	0x010e0420
v0	2	0x0000000a
v1	3	0x00000000
a0	4	0x00000000
a1	5	0x00000000
a2	6	0x00000000
a3	7	0x00000000
t0	8	0x00000440
t1	9	0x00000000
t2	10	0xffffffa8
t3	11	0x00000000
t4	12	0x00000000
t5	13	0x00000000
t6	14	0x00000000
t7	15	0x00000000
s0	16	0x10010000
s1	17	0x20210522
s2	18	0x044004a8

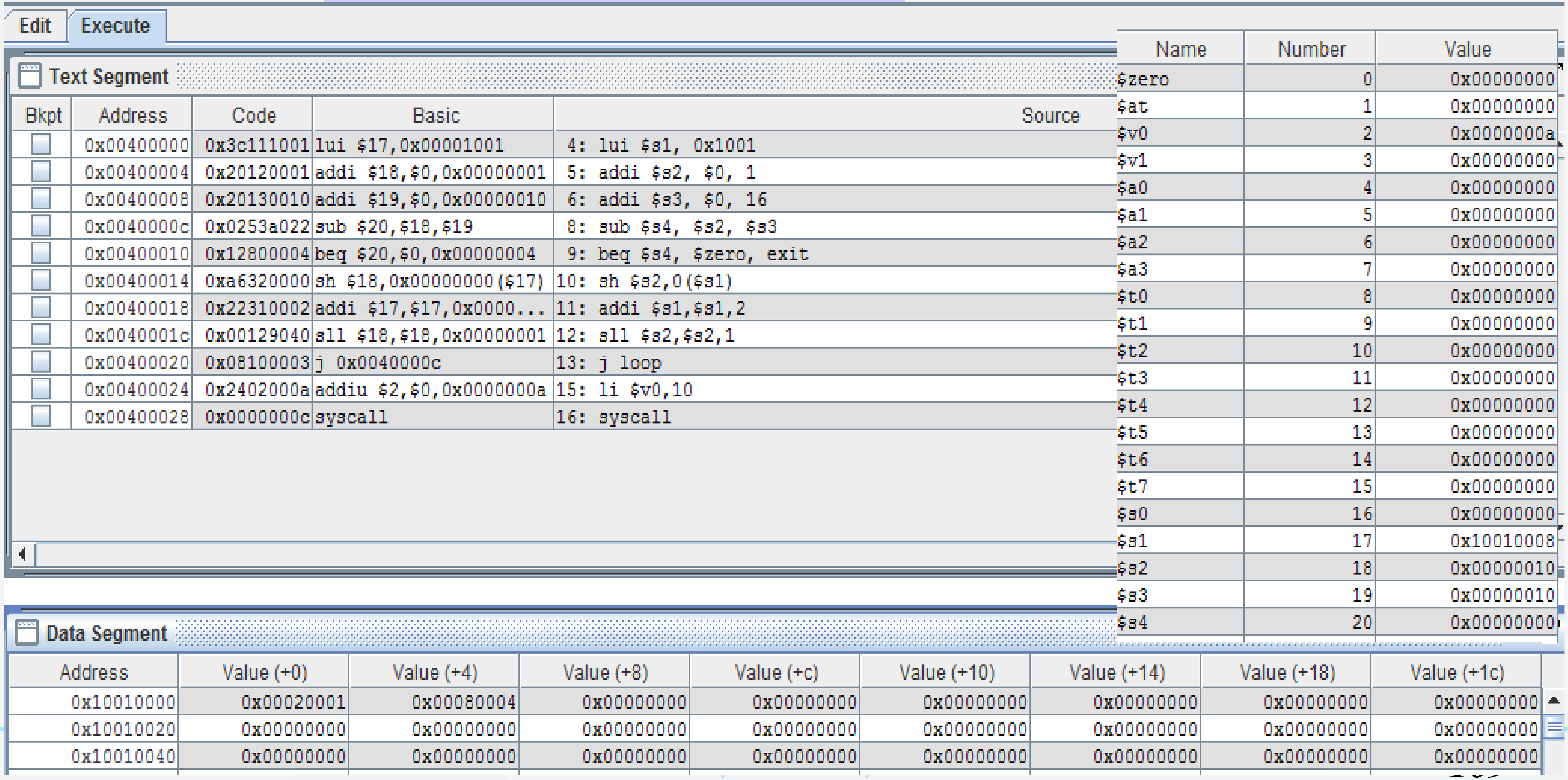
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x20210522	0x044004a8	0xffffffa8	0x010e0420	0x010e04a8	0x010e0440	0x00000000	0x00000000

Exercise8

حدد القيمة النهائية للمسجلات بعد إنهاء تنفيذ البرنامج؟

```
.data
.half 0x1234, 0xddcc, 0xbbaa, 0x9988
.text
lui $s1, 0x1001
addi $s2, $0, 1
addi $s3, $0, 16
loop:
sub $s4, $s2, $s3
beq $s4, $zero, exit
sh $s2,0($s1)
addi $s1,$s1,2
sll $s2,$s2,1
j loop
exit:
li $v0,10
syscall
```

Exercise8



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c111001	lui $17,0x00001001	4: lui $s1, 0x1001
☐	0x00400004	0x20120001	addi $18,$0,0x00000001	5: addi $s2, $0, 1
☐	0x00400008	0x20130010	addi $19,$0,0x00000010	6: addi $s3, $0, 16
☐	0x0040000c	0x0253a022	sub $20,$18,$19	8: sub $s4, $s2, $s3
☐	0x00400010	0x12800004	beq $20,$0,0x00000004	9: beq $s4, $zero, exit
☐	0x00400014	0xa6320000	sh $18,0x00000000($17)	10: sh $s2,0($s1)
☐	0x00400018	0x22310002	addi $17,$17,0x0000...	11: addi $s1,$s1,2
☐	0x0040001c	0x00129040	sll $18,$18,0x00000001	12: sll $s2,$s2,1
☐	0x00400020	0x08100003	j 0x0040000c	13: j loop
☐	0x00400024	0x2402000a	addiu $2,$0,0x0000000a	15: li $v0,10
☐	0x00400028	0x0000000c	syscall	16: syscall

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x10010008
$s2	18	0x00000010
$s3	19	0x00000010
$s4	20	0x00000000

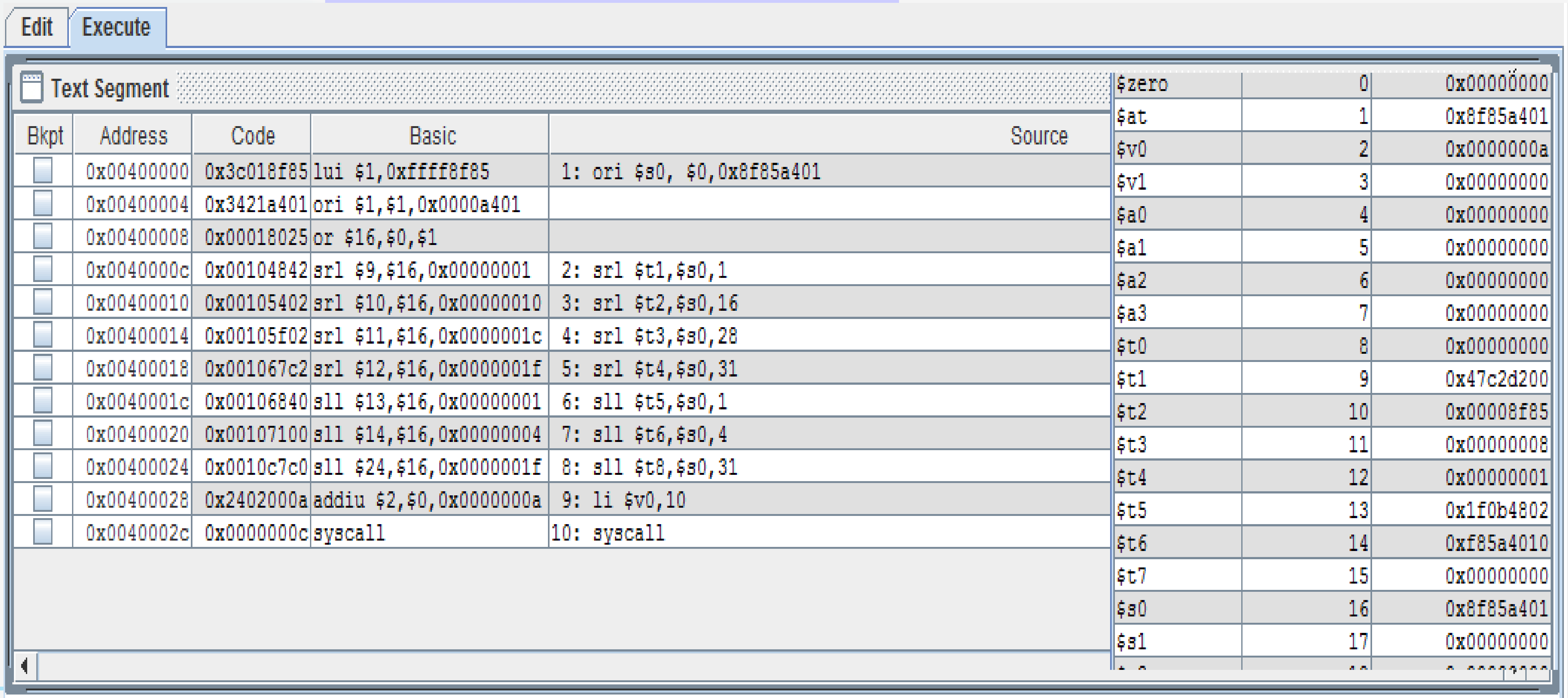
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00020001	0x00080004	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Exercise9

حدد القيمة النهائية للمسجلات بعد إنهاء تنفيذ البرنامج؟

```
ori $s0, $0,0x8f85a401
srl $t1,$s0,1
srl $t2,$s0,16
srl $t3,$s0,28
srl $t4,$s0,31
sll $t5,$s0,1
sll $t6,$s0,4
sll $t8,$s0,31
li $v0,10
syscall
```

Exercise9



Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c018f85	lui $1,0xffff8f85	1: ori $s0, $0,0x8f85a401
☐	0x00400004	0x3421a401	ori $1,$1,0x0000a401	
☐	0x00400008	0x00018025	or $16,$0,$1	
☐	0x0040000c	0x00104842	srl $9,$16,0x00000001	2: srl $t1,$s0,1
☐	0x00400010	0x00105402	srl $10,$16,0x00000010	3: srl $t2,$s0,16
☐	0x00400014	0x00105f02	srl $11,$16,0x0000001c	4: srl $t3,$s0,28
☐	0x00400018	0x001067c2	srl $12,$16,0x0000001f	5: srl $t4,$s0,31
☐	0x0040001c	0x00106840	sll $13,$16,0x00000001	6: sll $t5,$s0,1
☐	0x00400020	0x00107100	sll $14,$16,0x00000004	7: sll $t6,$s0,4
☐	0x00400024	0x0010c7c0	sll $24,$16,0x0000001f	8: sll $t8,$s0,31
☐	0x00400028	0x2402000a	addiu $2,$0,0x0000000a	9: li $v0,10
☐	0x0040002c	0x0000000c	syscall	10: syscall

$zero	0	0x00000000
$at	1	0x8f85a401
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x47c2d200
$t2	10	0x00008f85
$t3	11	0x00000008
$t4	12	0x00000001
$t5	13	0x1f0b4802
$t6	14	0xf85a4010
$t7	15	0x00000000
$s0	16	0x8f85a401
$s1	17	0x00000000

Exercise10

```
lui $s0,0x0000
add $a0,$0,$0
add $a1,$0,$0
add $a2,$0,$0
add $a3,$0,$0
addi $s0,$s0,100
addi $a0,$a0,4
addi $a1,$a1,7
addi $a2,$a2,2
addi $a3,$a3,1
jal example
add $s7,$s7,$s0
j exit
```

```
example:addi $sp,$sp,-4
        sw $s0,0($sp)
        add $t0,$a0,$a1
        add $t1,$a2,$a3
        sub $s0,$t0,$t1
        add $v0,$t0,$t1
        lw $s0,0($sp)
        addi $sp,$sp,4
        jr $ra
exit:
        li $v0,10
        syscall
```

Exercise10

1- حاول معرفة تغيرات المسجلات بعد كل تعليمة.

2- ما القيم التي تم حفظها في المكدس وكم عددها؟

3- ما هو مقدار توسيع ذاكرة الكدسة؟

4- ما العنوان الذي تم حفظه في المسجل $ra قبل الانتقال لتنفيذ البرنامج الفرعي؟

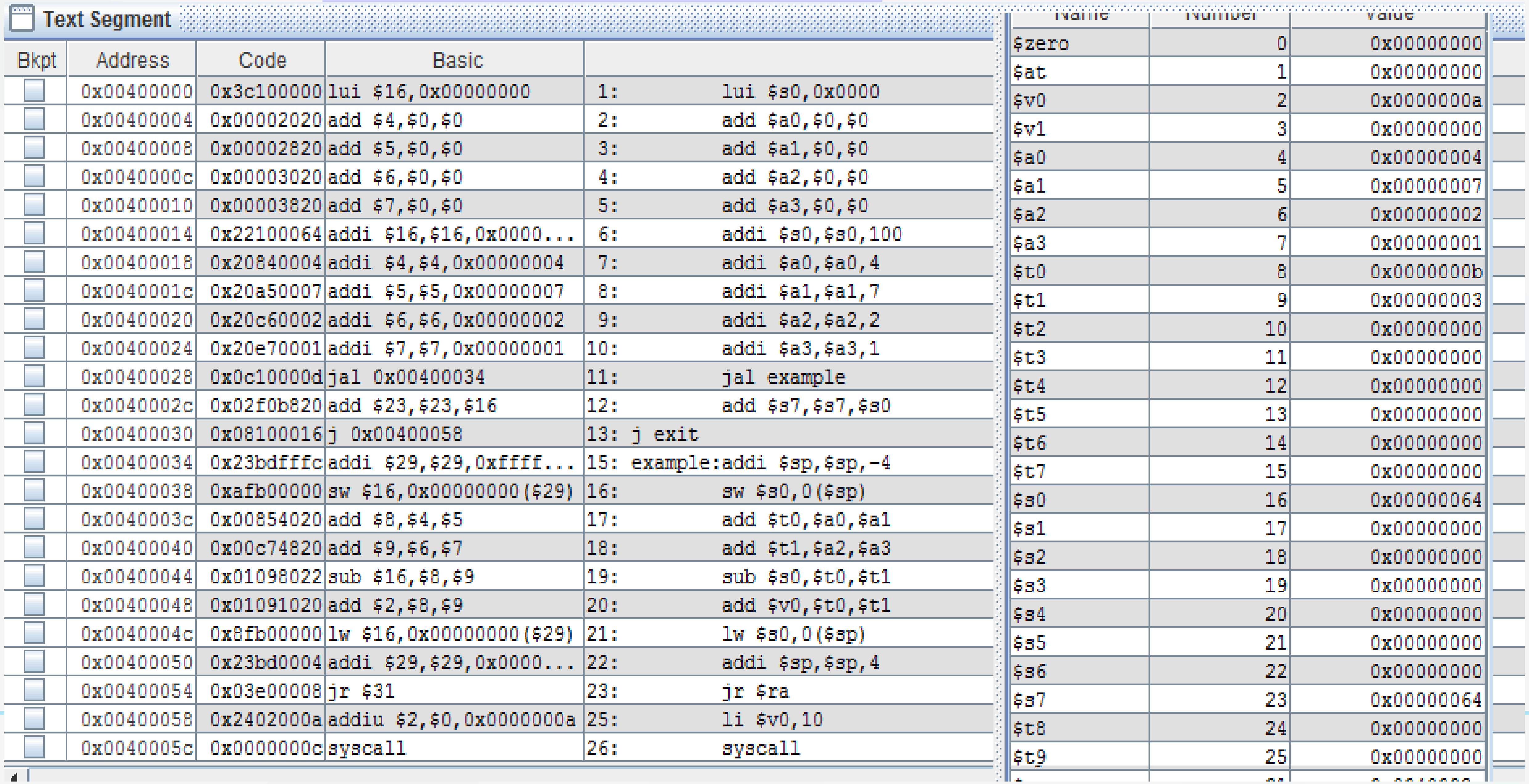
5- تتبع تغيرات المسجل $pc.

6- تتبع تغيرات المسجل $sp.

7- وضح كيف يتم استعادة القيم التي تم حفظها في المكدس.

8- وضح كيف يتم العودة إلى البرنامج الأساس.

Exercise10



Text Segment

Bkpt	Address	Code	Basic	
☐	0x00400000	0x3c100000	lui $16,0x00000000	1: lui $s0,0x0000
☐	0x00400004	0x00002020	add $4,$0,$0	2: add $a0,$0,$0
☐	0x00400008	0x00002820	add $5,$0,$0	3: add $a1,$0,$0
☐	0x0040000c	0x00003020	add $6,$0,$0	4: add $a2,$0,$0
☐	0x00400010	0x00003820	add $7,$0,$0	5: add $a3,$0,$0
☐	0x00400014	0x22100064	addi $16,$16,0x0000...	6: addi $s0,$s0,100
☐	0x00400018	0x20840004	addi $4,$4,0x00000004	7: addi $a0,$a0,4
☐	0x0040001c	0x20a50007	addi $5,$5,0x00000007	8: addi $a1,$a1,7
☐	0x00400020	0x20c60002	addi $6,$6,0x00000002	9: addi $a2,$a2,2
☐	0x00400024	0x20e70001	addi $7,$7,0x00000001	10: addi $a3,$a3,1
☐	0x00400028	0x0c10000d	jal 0x00400034	11: jal example
☐	0x0040002c	0x02f0b820	add $23,$23,$16	12: add $s7,$s7,$s0
☐	0x00400030	0x08100016	j 0x00400058	13: j exit
☐	0x00400034	0x23bdfffc	addi $29,$29,0xffff...	15: example:addi $sp,$sp,-4
☐	0x00400038	0xafb00000	sw $16,0x00000000($29)	16: sw $s0,0($sp)
☐	0x0040003c	0x00854020	add $8,$4,$5	17: add $t0,$a0,$a1
☐	0x00400040	0x00c74820	add $9,$6,$7	18: add $t1,$a2,$a3
☐	0x00400044	0x01098022	sub $16,$8,$9	19: sub $s0,$t0,$t1
☐	0x00400048	0x01091020	add $2,$8,$9	20: add $v0,$t0,$t1
☐	0x0040004c	0x8fb00000	lw $16,0x00000000($29)	21: lw $s0,0($sp)
☐	0x00400050	0x23bd0004	addi $29,$29,0x0000...	22: addi $sp,$sp,4
☐	0x00400054	0x03e00008	jr $31	23: jr $ra
☐	0x00400058	0x2402000a	addiu $2,$0,0x0000000a	25: li $v0,10
☐	0x0040005c	0x0000000c	syscall	26: syscall

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000004
$a1	5	0x00000007
$a2	6	0x00000002
$a3	7	0x00000001
$t0	8	0x0000000b
$t1	9	0x00000003
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000064
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000064
$t8	24	0x00000000
$t9	25	0x00000000