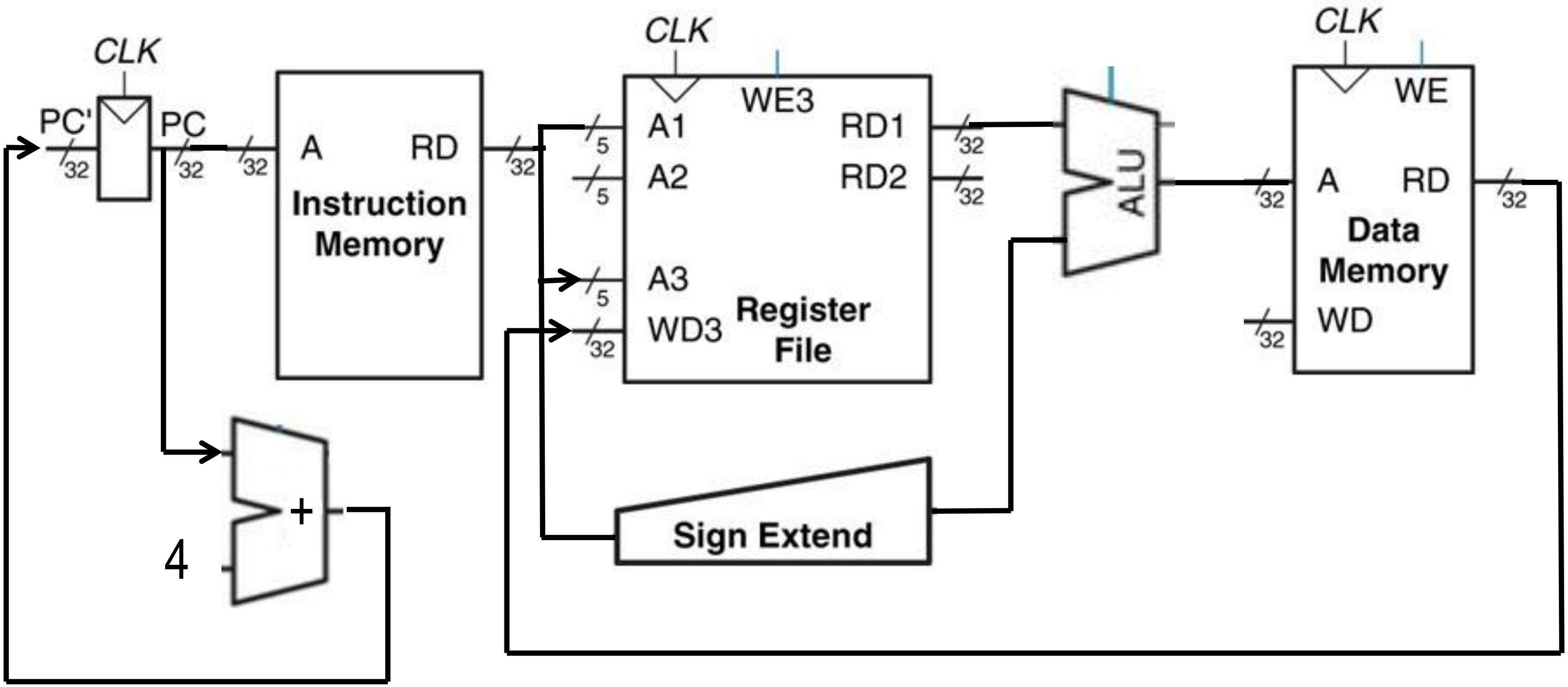
MIPS ASSEMBLY

MIPS Assembly Programming

The Language of the Electronic Computer

32-bit RegisterFile + ALU

MIPS

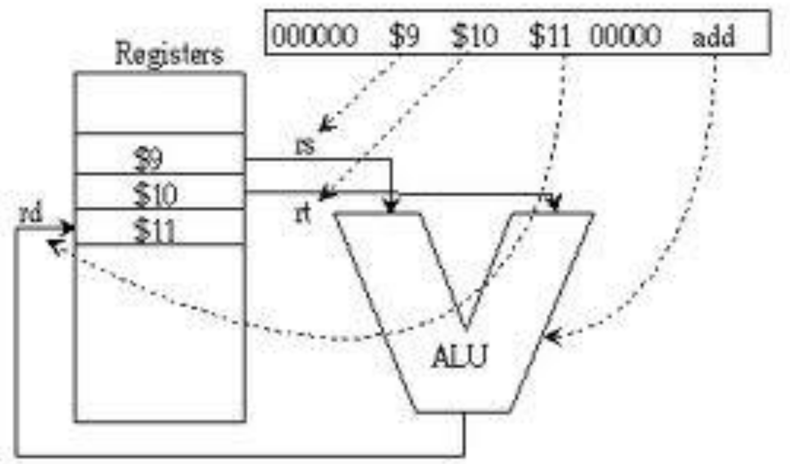


Why Assembly?

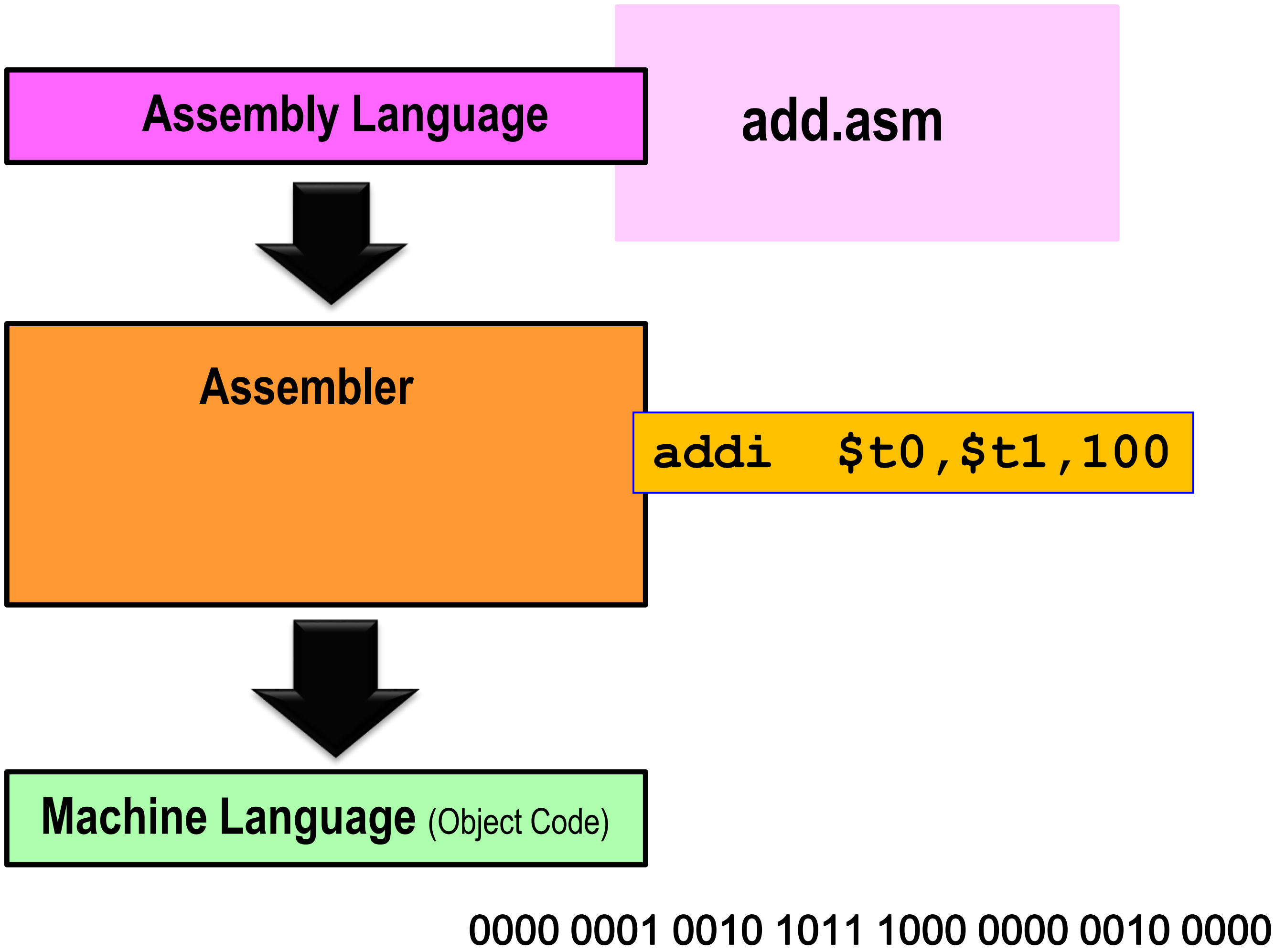
- **Assembly is widely used in industry:**
  - Embedded systems
  - Real time systems
  - Low level and direct access to hardware
- **Assembly is widely used not in industry:**
  - Cracking software protections: patching, patch-loaders and emulators
  - Hacking into computer systems: buffer under/overflows, worms and Trojans.

Assembly-Machine Language

- Each assembly language is specific to a particular computer architecture
- Each computer architecture has its own machine language.



Assembly-Machine Language



MIPS Architecture

- MIPS is a register-to-register, or load/store, architecture
- The destination and sources must all be registers
- Special instructions are needed to access the main memory.

MIPS: Register File

MIPS processors have 32-registers, each of which holds a 32-bit value

- Register addresses are 5-bits (2^5 = 32-bits) long
- The data inputs and outputs are 32-bits wide.

MIPS register names convention

- MIPS register names begin with a dollar sign → **`$`**

1. By number:

   `$0`, **`$1`**, …, **`$31`**

2. By a letter and a number:

   `$a0-$a2` …

MIPS registers

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194304
hi		0
lo		0

Registers | Coproc 1 | Coproc 0

MIPS registers

Name	Number	Use	Preserved across a call?
`$zero`	0	The constant value 0	N.A.
`$at`	1	Assembler temporary	No
`$v0–$v1`	2–3	Values for function results and expression evaluation	No
`$a0–$a3`	4–7	Arguments	No
`$t0–$t7`	8–15	Temporaries	No
`$s0–$s7`	16–23	Saved temporaries	Yes
`$t8–$t9`	24–25	Temporaries	No
`$k0–$k1`	26–27	Reserved for OS kernel	No
`$gp`	28	Global pointer	Yes
`$sp`	29	Stack pointer	Yes
`$fp`	30	Frame pointer	Yes
`$ra`	31	Return address	Yes

Figure 1.4 MIPS registers and usage conventions. In addition to the 32 general-purpose registers (R0–R31), MIPS has 32 floating-point registers (F0–F31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number.

Our first assembly demo program

1.asm

```
# Folder L1/1.asm

		.text
		.globl main
main:
		li	$t0, 2
		li	$t1, 3
		add	$t5, $t1, $t0
		li	$v0, 10
		syscall
```

System Calls: 10

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194304
hi		0
lo		0

Our first assembly demo program

Comments

Assembly directives

```
# Folder L1/1.asm

        .text               #  Informs the assembler that instructions follow
        .globl main         #  Declare as global the label main
main:                       #  Execution starts at main:
        li      $t0, 2      #  $t0 = 2
        li      $t1, 3      #  $t1 = 3
        add     $t5, $t1, $t0   #  $t5 = $t1 + $t0
        li      $v0, 10     #  System call for exit (Load code 10)
        syscall             #  Call operating system to perform operation (exit)
```

Label

Opcode

Operand

`li` is a pseudo-instruction; will talk about it in the next lecture

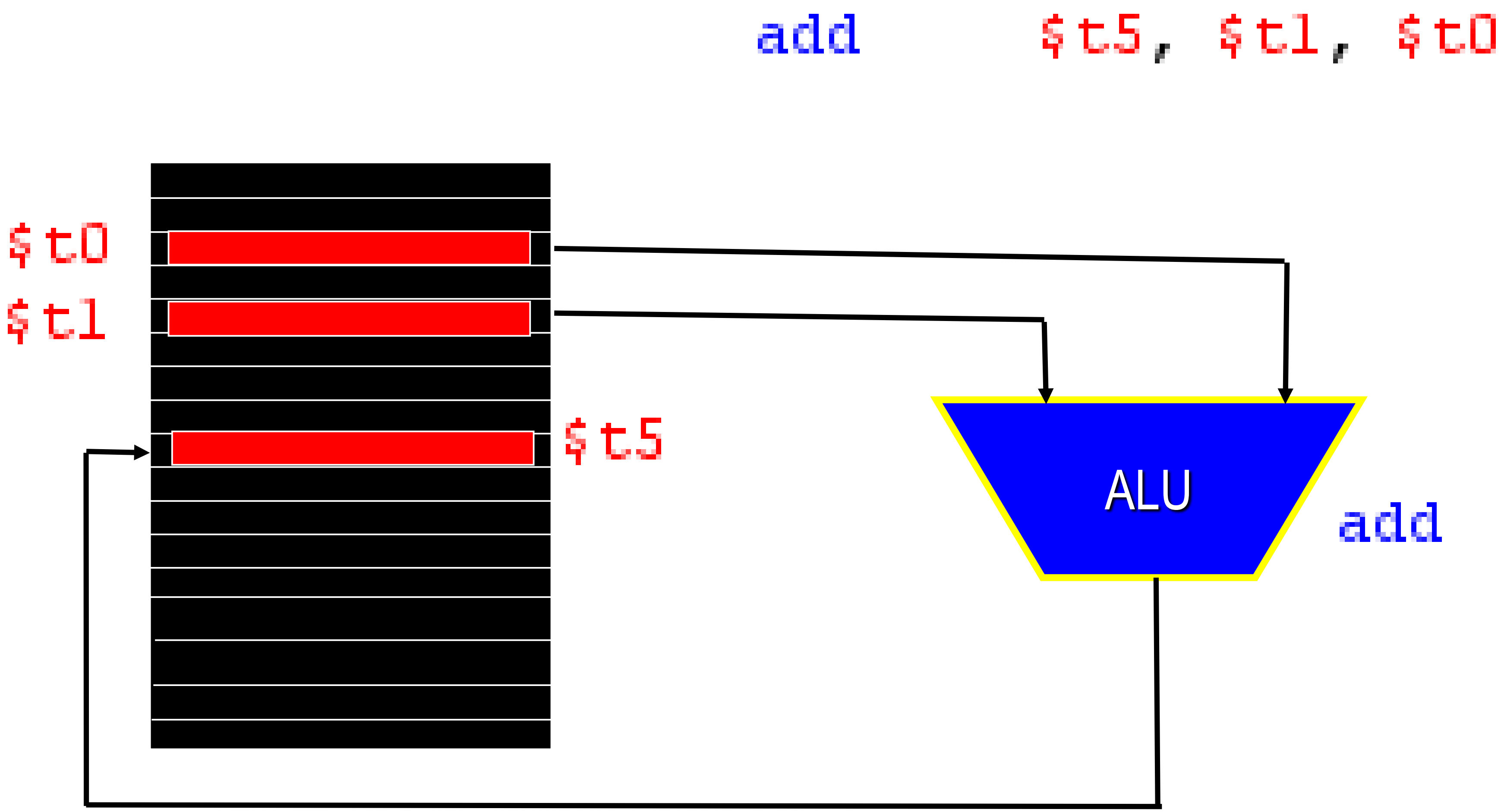
Assemble … GO

$t0	8	2
$t1	9	3
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	5
$t6	14	0
$t7	15	0

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	2
$t1	9	3
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	5
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194324
hi		0
lo		0

[32] 32-bit RegisterFile + ALU



Assembler COMMENTS and LABELS

- Comments: Text following a '#' (sharp) to the end of the line is ignored
- Labels: Are symbols that represent memory addresses
  - Labels take on the values of the address where they are declared
  - Labels declarations appear at the beginning of a line, and are terminated by a colon.

Instructions are divided into three kinds of format

- (**R**, **I** and **J** format)

  - **Register** arithmetic instructions (**R**-format)
  - Memory **Immediate** load and store (**I**-format)
  - Branching and **Jump** instructions (**J**-format).

MIPS instruction format

Basic instruction formats

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15		
J	opcode	address				
	31 26	25				

Floating-point instruction formats

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15		

Figure 1.6 MIPS64 instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as DADDU, DSUBU, and so on. The I format is for data transfers, branches, and immediate instructions, such as LD, SD, BEQZ, and DADDIs. The J format is for jumps, the FR format for floating-point operations, and the FI format for floating-point branches.

MIPS instruction format

Format	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Comments
R	op	rs	rt	rd	shamt	funct	Arithmetic
I	op	rs	rt	address/immediate			Transfer, branch,immediate
J	op	target address					Jump

- **op:** basic operation of instruction
- **funct:** variant of instruction
- **rs:** first register source operand
- **rt:** second register source operand
- **rd:** register destination operand
- **shamt:** shift amount

R-Format example

```
add $t0, $s1, $s2
```

add $t0, $s1, $s2 < R-Format

000000	00111	01000	00110	00000	100000_2
0	7	8	6	0	32_{10}
6-bits	5-bits(t0)	5-bits (s1)	5-bits (s2)	5-bits	6-bits

- **rs** = 8_{10}: first source operand is: **$s1**
- **rt** = 6_{10}: second source operand is: **$s2**
- **rd** = 7_{10}: register destination is: **$t0**
- **funct** = 32 (**add**)

MIPS Instruction set

- Arithmetic, Logic, and Shifting Instructions
- Conditional Branch Instructions
- Load and Store Instructions
- Function Call Instructions

Arithmetic

```
add $t5, $t1, $t0
```

Our first assembly demo program

```
# Folder L1/1.asm

            .text
            .globl main
main:
            li          $t0, 2
            li          $t1, 3
            add         $t5, $t1, $t0
            li          $v0, 10
            syscall
```

li

add

li

syscall

.text and .globl directives

- **.text** directive
  - Defines the section of a program containing instructions
- **.globl main**
  - Declares main as global
- **main:** label that represents a memory address.

```
            .text
            .globl main
main:
            li      $t0, 2
            li      $t1, 3
            add     $t5, $t1, $t0
            li      $v0, 10
            syscall
```

A Breakdown of Segment and Linker Directives

Name	Parameters	Description
`.data`	*addr*	The following items are to be assembled into the data segment. By default, begin at the next available address in the data segment. If the optional argument *addr* is present, then begin at *addr*.
`.text`	*addr*	The following items are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument *addr* is present, then begin at *addr*. In SPIM, the only items that can be assembled into the text segment are instructions and words (via the `.word` directive).
`.kdata`	*addr*	The kernel data segment. Like the data segment, but used by the Operating System.
`.ktext`	*addr*	The kernel text segment. Like the text segment, but used by the Operating System.
`.extern`	*sym size*	Declare as global the label *sym*, and declare that it is *size* bytes in length (this information can be used by the assembler).
`.globl`	*sym*	Declare as global the label *sym*.

li *des*,*const* # load the constant const into *des*

```
li      $t0, 2
li      $t1, 3
```

	Op	Operands	Description
○	`la`	*des, addr*	Load the address of a label.
	`lb(u)`	*des, addr*	Load the byte at *addr* into *des*.
	`lh(u)`	*des, addr*	Load the halfword at *addr* into *des*.
○	`li`	*des, const*	Load the constant *const* into *des*.
	`lui`	*des, const*	Load the constant *const* into the upper halfword of *des*, and set the lower halfword of *des* to 0.
	`lw`	*des, addr*	Load the word at *addr* into *des*.
	`lwl`	*des, addr*	
	`lwr`	*des, addr*	
○	`ulh(u)`	*des, addr*	Load the halfword starting at the (possibly unaligned) address *addr* into *des*.
○	`ulw`	*des, addr*	Load the word starting at the (possibly unaligned) address *addr* into *des*.

`addu` ← new instruction

`add` instruction

`add/addu` instructions

- `add;` signed addition
- `addu;` **u**nsigned addition.

4.4.1 Arithmetic Instructions

	Op	Operands	Description
∘	`abs`	*des, src1*	*des* gets the absolute value of *src1*.
	`add(u)`	*des, src1, src2*	*des* gets *src1* + *src2*.
	`and`	*des, src1, src2*	*des* gets the bitwise and of *src1* and *src2*.
	`div(u)`	*src1, reg2*	Divide *src1* by *reg2*, leaving the quotient in register `lo` and the remainder in register `hi`.
∘	`div(u)`	*des, src1, src2*	*des* gets *src1* / *src2*.
∘	`mul`	*des, src1, src2*	*des* gets *src1* × *src2*.
∘	`mulo`	*des, src1, src2*	*des* gets *src1* × *src2*, with overflow.
	`mult(u)`	*src1, reg2*	Multiply *src1* and *reg2*, leaving the low-order word in register `lo` and the high-order word in register `hi`.
∘	`neg(u)`	*des, src1*	*des* gets the negative of *src1*.
	`nor`	*des, src1, src2*	*des* gets the bitwise logical **nor** of *src1* and *src2*.
∘	`not`	*des, src1*	*des* gets the bitwise logical negation of *src1*.
	`or`	*des, src1, src2*	*des* gets the bitwise logical **or** of *src1* and *src2*.
∘	`rem(u)`	*des, src1, src2*	*des* gets the remainder of dividing *src1* by *src2*.
∘	`rol`	*des, src1, src2*	*des* gets the result of rotating left the contents of *src1* by *src2* bits.
∘	`ror`	*des, src1, src2*	*des* gets the result of rotating right the contents of *src1* by *src2* bits.
	`sll`	*des, src1, src2*	*des* gets *src1* shifted left by *src2* bits.
	`sra`	*des, src1, src2*	Right shift arithmetic.
	`srl`	*des, src1, src2*	Right shift logical.
	`sub(u)`	*des, src1, src2*	*des* gets *src1* - *src2*.
	`xor`	*des, src1, src2*	*des* gets the bitwise exclusive **or** of *src1* and *src2*.

MIPS: Three-address instructions

- MIPS uses three-address instructions for data manipulation
- Each ALU instruction contains a destination and two sources
- For example, an addition instruction (a = b + c) has the form.

```
add $t5,$t1,$t0
```

1 2 3

add signed addition `add $t5, $t1, $t0`

- Performs the Binary Addition algorithm on two 32-bits;
  - Signed Binary
- **`add`** *des, src1, src2* # *des* gets *src1* + *src2*
- **`add $t5,$t1,$t0`** **`# $t5: 4 = 1 + 3`**
- Three registers (**`$t5,$t1,$t0`**) are involved
- Overflow **trap** is possible

```
   1001 1001
+  0111 0001
 ------------
 10001 1010
```

- A **trap** is an interruption in the normal machine cycle.

addu; **u**nsigned addition

- Performs the Binary Addition Algorithm on two 32-bits ;
  - Unsigned Binary
  - Two's Complement
- The destination register can be the same as one of the source registers
- **`add(u)`** *des, src1, src2* # *des* gets *src1* + *src2*
- **`add(u) $t0, $t0, $t1`** # *$t0* = *$t0* + *$t1*
- **`addu`** … ignores overflow trap.

http://programmedlessons.org/AssemblyTutorial/index.html

`add` … `addu`

- The `add` instruction it is used in the cases that overflow is an important factor. Otherwise we use the `addu` instruction

  – For signed numbers, use `add`

  – For unsigned numbers, use `addu`

… and the `addi` instruction …

`addi` ← new instruction

`addi` (`add i`mmediate)> `addi $r1, $r2,4`

MIPS32; `addi` … add immediate			
`addi`	`$r1`	`$r2`	`4`

`addi` (`add` `i`mmediate)> `addi $r1, $r2,4`

MIPS32; `addi` … add immediate			
`addi`	`$r1`	`$r2`	`4`
001000	00001	00010	0000000000000100

`addi` (`add i`mmediate)> `addi $r1, $r2,4`

MIPS32; `addi` … add immediate			
`addi`	`$r1`	`$r2`	`4`
001000	00001	00010	0000000000000100
Op Code	rt	rs	Immediate value 4

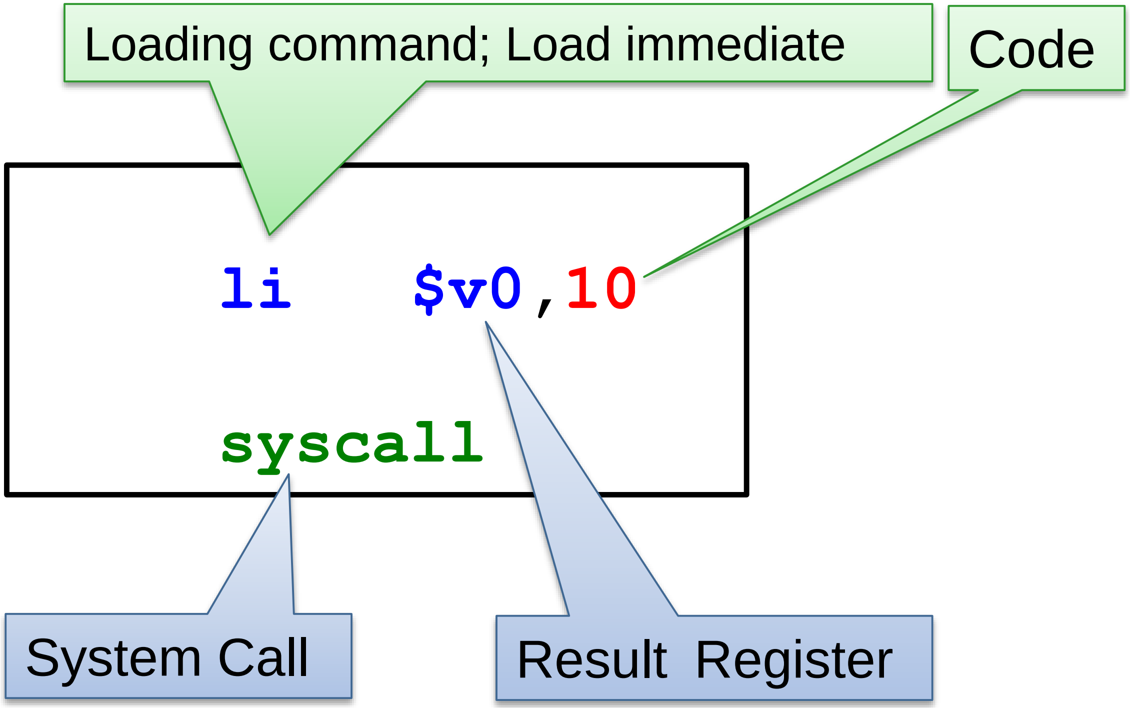
`li` (**l**oad **i**mmediate)

Another immediate instruction

`li` (`l`oad `i`mmediate)

- `li $v0,10` # load immediate `$v0` = 10
- `Syscall` # instruction; action depends on Code loaded in the register: `$v0`

Load 10 into $v0; … terminate



- A system call starts off by loading a specific code into the Result Register
- Then, the **syscall** instruction is called.The final result depends on the code loaded into the Result register
- The above example is the exit **syscall**, since loading the code "10" and calling the "**syscall**" instruction terminates the program.

addi

2.asm*

```
# Folder L1/2.asm

        .text
        .globl main
main:
        li      $t0, 2
        addi    $t5, $t0, 3
        li      $v0, 10
        syscall

```

addi

2.asm*

```
# Folder L1/2.asm

        .text                           #  Informs the assembler that instructions follow
        .globl main                     #  Declare as global the label main
main:                                   #  Execution starts at main:
        li      $t0, 2                  #  $t0 = 2
        addi    $t5, $t0, 3             #  $t5 = $t0 + 3
        li      $v0, 10                 #  System call for exit (Load code 10)
        syscall                         #  Call operating system to perform operation (exit)

```

Assemble … GO

```
# Folder L1/2.asm

            .text
            .globl main
main:
            li      $t0, 2
            addi    $t5, $t0, 3
            li      $v0, 10
            syscall

```

$t0	8	2
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	5
$t6	14	0
$t7	15	0

addi …add **i**mmediate unsigned

- **addi** *des*, *src1*, *const* # *des* *gets* *src1* + *const*
- **addi $t0, $t0, 1** # ***$t0*** = ***$t0*** + **1**

(Overflow trap)

- Sign Extension (done)

`addiu` ← new instruction

Addiu ← add immediate unsigned

- **`addiu`** *des, src1, const* # *des gets src1 + const*
- **`addiu $t0, $t0, 1`** # *$t0* = *$t0* + *1*

(No overflow trap)

Add three numbers …

- **1 + 3 + 4**
- To add the numbers, just use the instruction: **`add`**

In Class

`5 minutes`

Program/Solution

3.asm*

```
# Folder L1/3.asm
# Andrew De Stefano and Chris Crockett, 2012
# This program adds three numbers.

        .text
        .globl main
main:
        li      $t0,1           # $t0 = 1
        li      $t1,3           # $t1 = 3
        li      $t2,4           # $t2 = 4
        add     $t6,$t1,$t0     # $t6 = 4
        add     $t7,$t6,$t2     # $t7 = 8
        li      $v0,10          # System call for exit (Load code 10)
        syscall                 # Call operating system to perform operation (exit)
```

Assemble … GO

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	1
$t1	9	3
$t2	10	4
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	4
$t7	15	8

"More …"

System **Call**s

System Calls (`syscall`)

- MIPS programs can make system calls by placing parameters in specified registers, depending on the call, and executing a code instruction.
- Returned results are made available in other specified registers, also depending on the call.

Syscalls

Service	Code	Arguments	Result
print_int	1	$a0	*none*
print_float	2	$f12	*none*
print_double	3	$f12	*none*
print_string	4	$a0	*none*
read_int	5	*none*	$v0
read_float	6	*none*	$f0
read_double	7	*none*	$f0
read_string	8	$a0 (address), $a1 (length)	*none*
sbrk	9	$a0 (length)	$v0
exit	10	*none*	*none*

PRINT

READ

- **"Service"** explains the function of the **syscall** code
- **"Code"** is the number to be loaded
- **"Arguments"** states the arguments used and where specifically they'd be located
- **"Result"** explains the output

How to use `syscall` system services

1. Load the service number in register `$v0`

2. Load argument values, if any, in registers: `$a0, $a1, $a2)`

3. Issue the `syscall` instruction

4. Retrieve return values, if any, from the result registers.

`la` ← new instruction

`l`oad the `a`ddress

`Syscall → 4`

Print a string of text

Hello world ….

System Calls: 4,10

```
# folder:
        .text                  # The following are to be assembled in to text segment
start:  la      $a0, msg       # Load the address of the message text
        li      $v0, 4         # Load the syscall (4) code for printing the string of text
        syscall
        li      $v0, 10        # Load the syscall (10) code for exiting
        syscall

        .data                  # Informs the assembler that data needed within instructions follows
msg:    .asciiz "hello world! "
```

Assembly directive

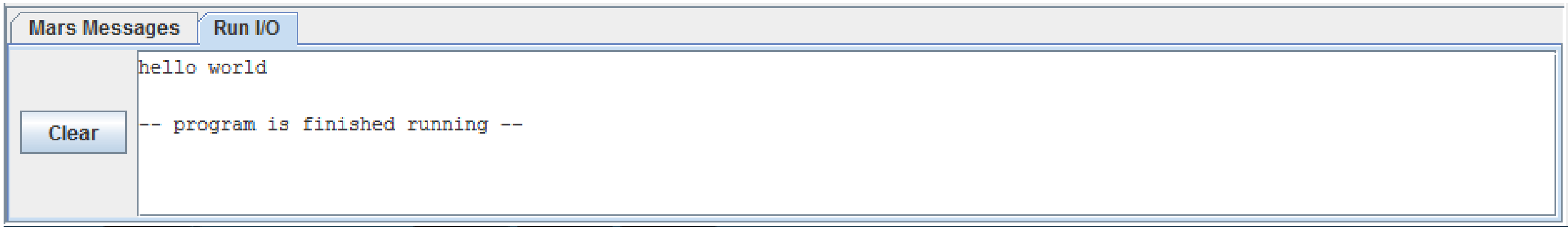
Assembly directive

Assembly directive

label

`la` is a pseudo-instruction; will talk about it in the next lecture

Assemble … GO


Mars Messages
Run I/O
Clear
hello world
-- program is finished running --

Data Directives

Name	Parameters	Description
`.data`	*addr*	The following items are to be assembled into the data segment. By default, begin at the next available address in the data segment. If the optional argument *addr* is present, then begin at *addr*.
`.text`	*addr*	The following items are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument *addr* is present, then begin at *addr*. In SPIM, the only items that can be assembled into the text segment are instructions and words (via the `.word` directive).
`.kdata`	*addr*	The kernel data segment. Like the data segment, but used by the Operating System.
`.ktext`	*addr*	The kernel text segment. Like the text segment, but used by the Operating System.
`.extern`	*sym size*	Declare as global the label *sym*, and declare that it is *size* bytes in length (this information can be used by the assembler).
`.globl`	*sym*	Declare as global the label *sym*.

Data Directives

Name	Parameters	Description
`.align`	*n*	Align the next item on the next 2^n-byte boundary. `.align 0` turns off automatic alignment.
`.ascii`	*str*	Assemble the given string in memory. Do not null-terminate.
`.asciiz`	*str*	Assemble the given string in memory. Do null-terminate.
`.byte`	*byte1 · · · byteN*	Assemble the given bytes (8-bit integers).
`.half`	*half1 · · · halfN*	Assemble the given halfwords (16-bit integers).
`.space`	*size*	Allocate *n* bytes of space in the current segment. In SPIM, this is only permitted in the data segment.
`.word`	*word1 · · · wordN*	Assemble the given words (32-bit integers).

la (load address)

- `la $a0, msg`   # load address of string to be printed into `$a0`

```
        .text                    # The following are to be assembled in to text segment
start:  la       $a0, msg        # Load the address of the message text
        li       $v0, 4          # Load the syscall (4) code for printing the string of text
        syscall
        li       $v0, 10         # Load the syscall (10) code for exiting
        syscall

        .data                    # Informs the assembler that data needed within instructions follows
msg:    .asciiz  "hello world!  "
```

Syscall → 1

Print-out Integer

Prints "out" the result

System Calls: 4,1,10

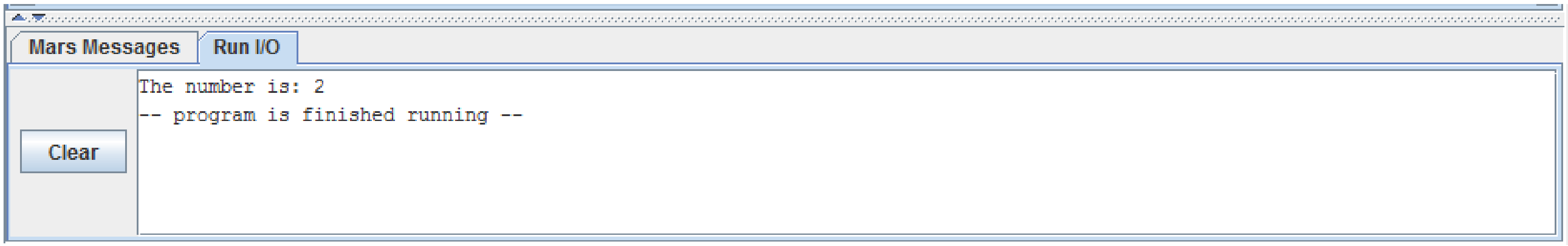
```
# Folder L1\4.asm
# Prints-"out"

        .text
        .globl main
main:
        la      $a0,  output
        li      $v0,4
        syscall

        li      $t0,2
        move    $a0,$t0   # The contents of $t0 are to be copied into register $a0
        li      $v0,1
        syscall

        li      $v0,10
        syscall

        .data
output:
        .asciiz "The number is: "
```

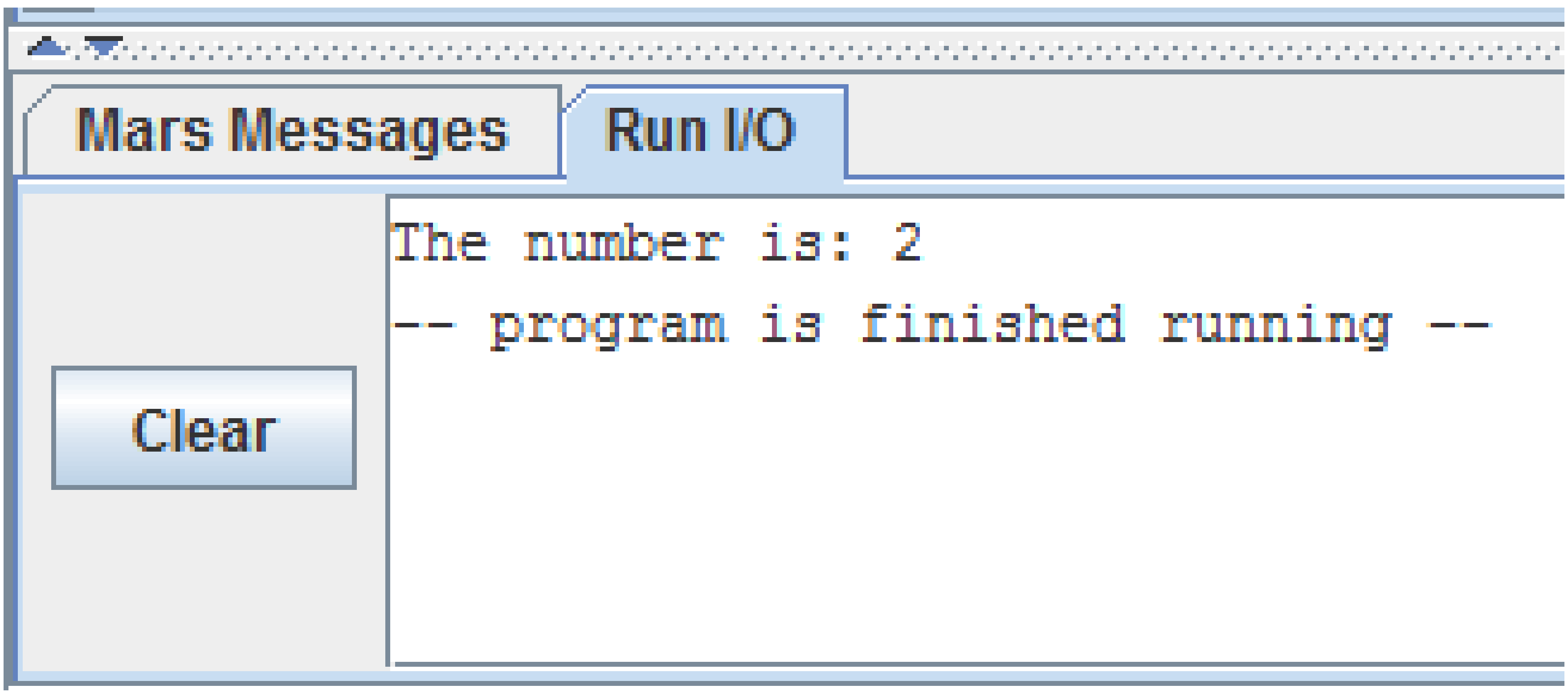
Prints “out” the result

```
# Folder L1\4.asm
# Prints-"out"

        .text
        .globl main
main:
        la      $a0,  output            # load address of string to be printed into $a0
        li      $v0,4                   # System call for printing string (code = 4)
        syscall                         # Call operating system to perform operation (Print string)

        li      $t0,2                   # $t0 = 2
        move    $a0,$t0                 # The contents of $t0 are to be copied into register $a0
        li      $v0,1                   # System call for printing integer (code = 1)
        syscall                         # Call operating system to perform operation (Print integer)

        li      $v0,10                  # System call for exit (code = 10)
        syscall                         # Call operating system to perform operation exit

        .data                           # Directive; Informs the assembler that data needed within instructions follows
output:                                 # Label (output)
        .asciiz "The number is: "       # Declaration for string variable (directive makes string null terminated)
```

move is a pseudo-instruction; will talk about it in the next lecture

Assemble … GO; **The number is: 2**





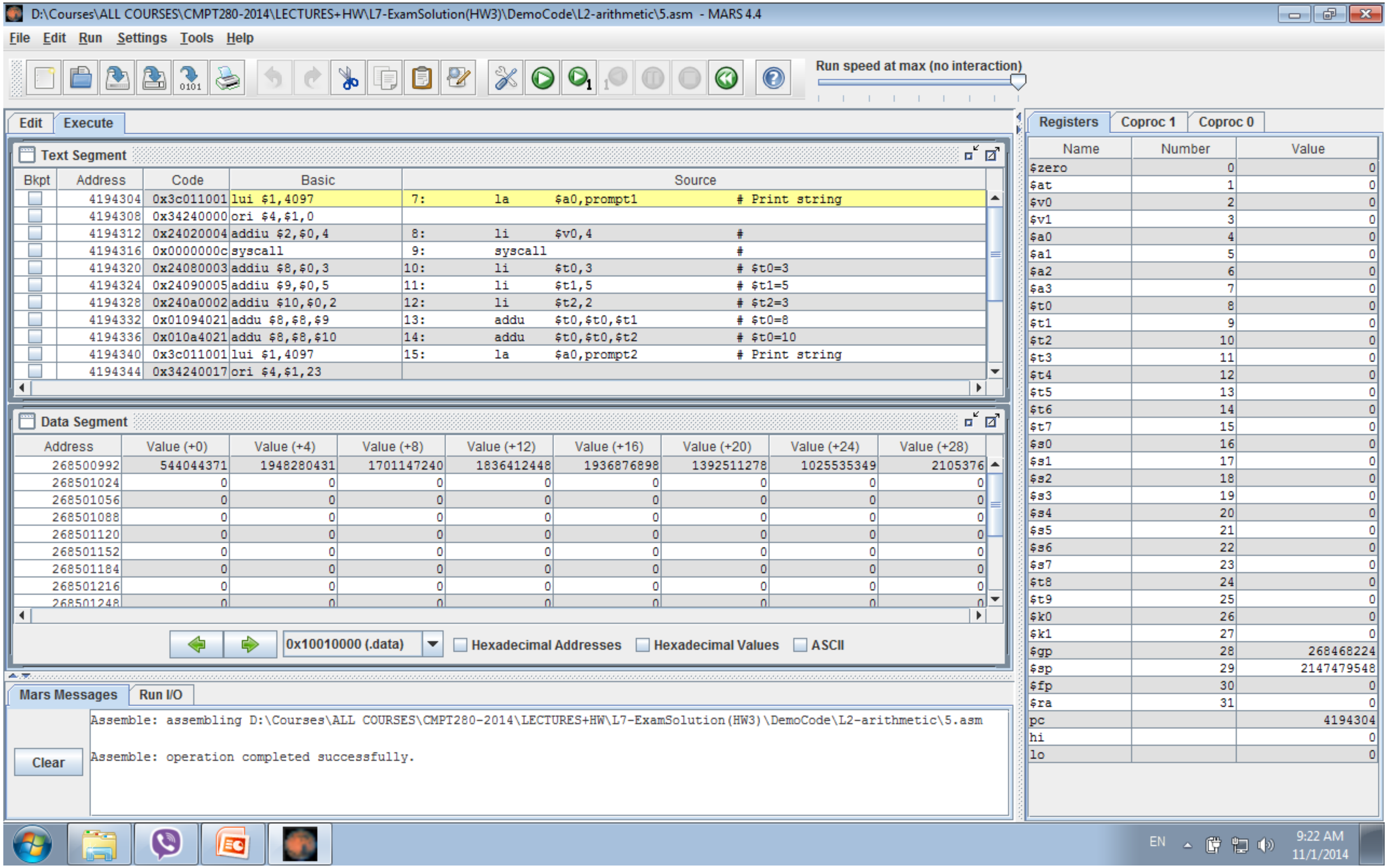
Example

5.asm

```
# Folder: L1/5.asm
# Andrew De Stefano and Chris Crockett, 2012

        .text
        .globl main
main:
        la      $a0,prompt1         # Print string
        li      $v0,4               #
        syscall                     #
        li      $t0,3               # $t0=3
        li      $t1,5               # $t1=5
        li      $t2,2               # $t2=2
        addu    $t0,$t0,$t1         # $t0=
        addu    $t0,$t0,$t2         # $t0=
        la      $a0,prompt2         # Print string
        li      $v0,4               #
        syscall                     #
        move    $a0,$t0             #
        li      $v0,1               #
        syscall                     #
        li      $v0,10
        syscall

        .data
prompt1:
        .asciiz "Sum of three numbers.  "
prompt2:
        .asciiz "Sum =   "
```

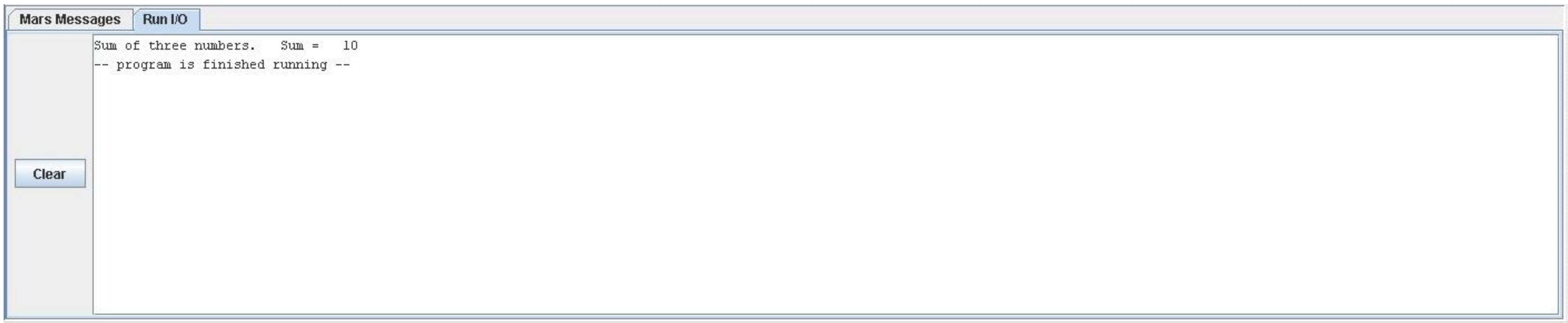
System Calls: 4,4,1,10

Assemble …



D:\Courses\ALL COURSES\CMPT280-2014\LECTURES+HW\L7-ExamSolution(HW3)\DemoCode\L2-arithmetic\5.asm - MARS 4.4
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Text Segment
Bkpt Address Code Basic Source
4194304 0x3c011001 lui $1,4097 7: la $a0,prompt1 # Print string
4194308 0x34240000 ori $4,$1,0
4194312 0x24020004 addiu $2,$0,4 8: li $v0,4 #
4194316 0x0000000c syscall 9: syscall #
4194320 0x24080003 addiu $8,$0,3 10: li $t0,3 # $t0=3
4194324 0x24090005 addiu $9,$0,5 11: li $t1,5 # $t1=5
4194328 0x240a0002 addiu $10,$0,2 12: li $t2,2 # $t2=3
4194332 0x01094021 addu $8,$8,$9 13: addu $t0,$t0,$t1 # $t0=8
4194336 0x010a4021 addu $8,$8,$10 14: addu $t0,$t0,$t2 # $t0=10
4194340 0x3c011001 lui $1,4097 15: la $a0,prompt2 # Print string
4194344 0x34240017 ori $4,$1,23
Data Segment
Address Value (+0) Value (+4) Value (+8) Value (+12) Value (+16) Value (+20) Value (+24) Value (+28)
268500992 544044371 1948280431 1701147240 1836412448 1936876898 1392511278 1025535349 2105376
0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII
Registers Coproc 1 Coproc 0
Name Number Value
$gp 28 268468224
$sp 29 2147479548
pc 4194304
Mars Messages Run I/O
Clear
Assemble: assembling D:\Courses\ALL COURSES\CMPT280-2014\LECTURES+HW\L7-ExamSolution(HW3)\DemoCode\L2-arithmetic\5.asm
Assemble: operation completed successfully.
9:22 AM
11/1/2014

GO …. (result)




```
Sum of three numbers.   Sum =   10
-- program is finished running --
```

`Syscall` → `5`

Read Integer from the Command line

Read and Print integer from command line

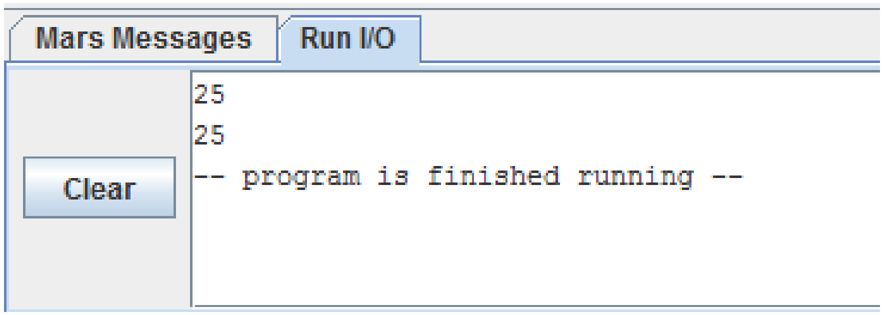
System Calls: 5,1,10

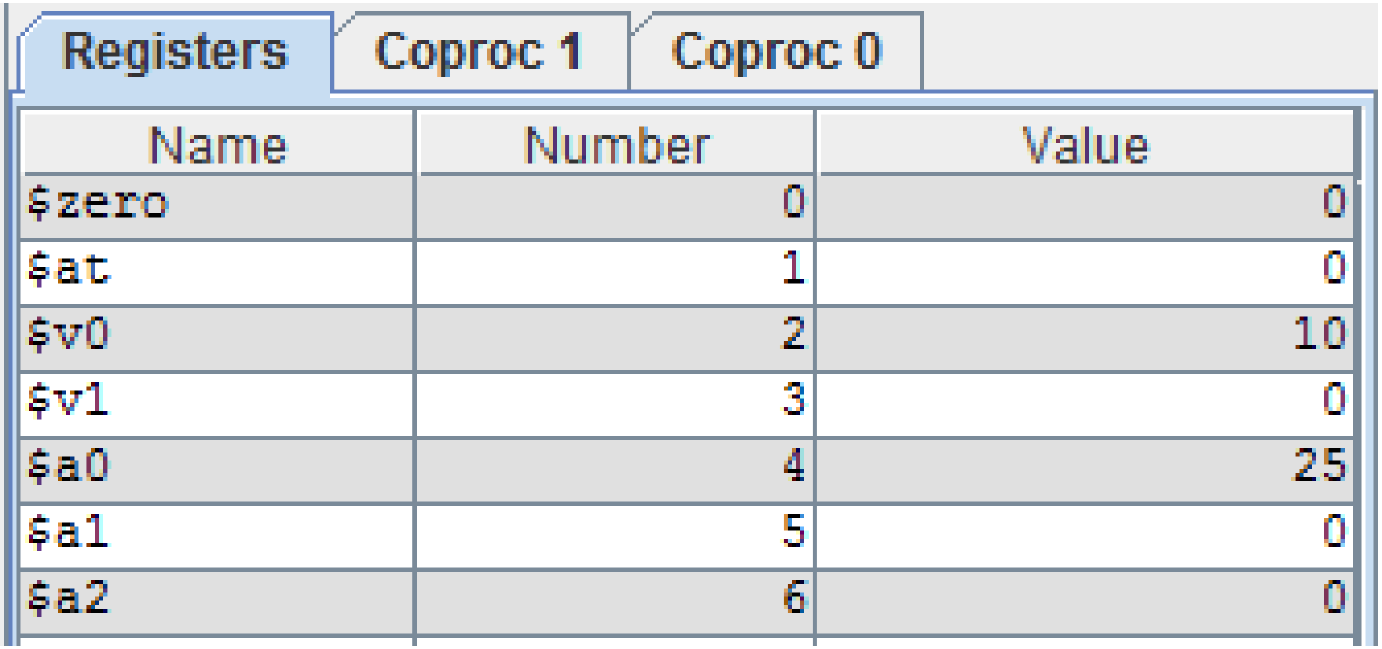
```
# Folder: L1/6.asm
# Read and Print integer f


        .text
        .globl main
main:
        li      $v0, 5
        syscall

        move    $a0, $v0
        li      $v0, 1
        syscall

        li      $v0, 10
        syscall
```

Read and Print integer from command line

```
# Folder: L1/6.asm
# Read and Print integer from command line


        .text
        .globl main
main:
        li      $v0, 5          # syscall for reading integer from the command line (code = 5)
        syscall

        move    $a0, $v0        # Move integer from $vo to $a0
        li      $v0, 1          # syscall for printing integer to command line (code = 1)
        syscall

        li      $v0, 10         # Call operating system to perform operation exit
        syscall
```

Assemble … GO;


Mars Messages
Run I/O
Clear
25
25
-- program is finished running --


Registers
Coproc 1
Coproc 0
Name
Number
Value
$zero 0 0
$at 1 0
$v0 2 10
$v1 3 0
$a0 4 25
$a1 5 0
$a2 6 0

System Services (`syscall`) in MIPS

To print an integer to the screen:

Set `$v0` to `1`

syscall

System Services (`syscall`) in MIPS

To print an integer to the screen: Set **$v0** to **1** syscall
To print a string to the screen: Set **$v0** **to** **4** syscall

System Services (`syscall`) in MIPS

To print an integer to the screen: Set `$v0` to `1` syscall
To print a string to the screen: Set `$v0 to 4` syscall
To read an integer from the keyboard: Set `$v0 to 5` syscall

System Services (`syscall`) in MIPS

To print an integer to the screen: Set `$v0` to `1` syscall
To print a string to the screen: Set `$v0 to 4` syscall
To read an integer from the keyboard: Set `$v0 to 5` syscall
To exit: Set `$v0 to 10` syscall