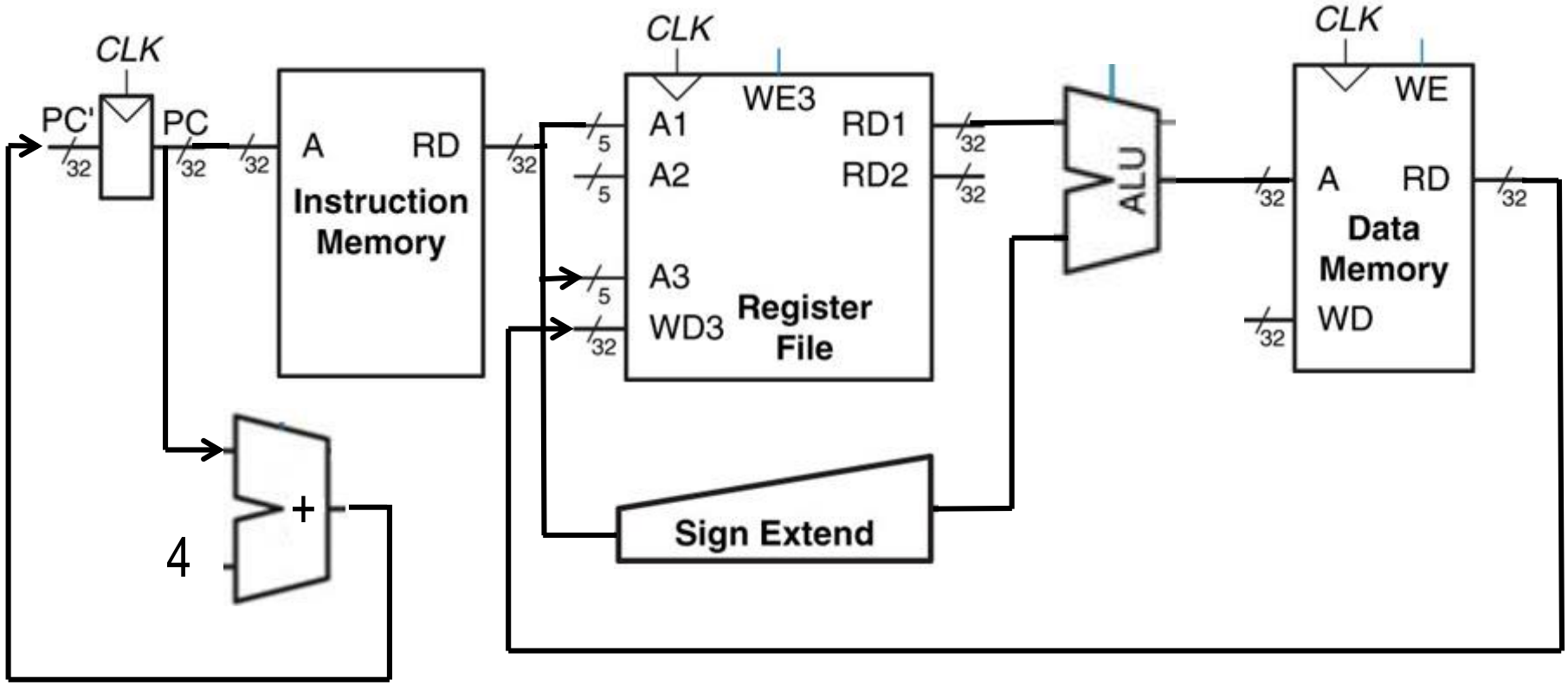
MIPS ASSEMBLY

MIPS Assembly Programming

The Language of a MIPS CPU

32-bit RegisterFile + ALU

MIPS



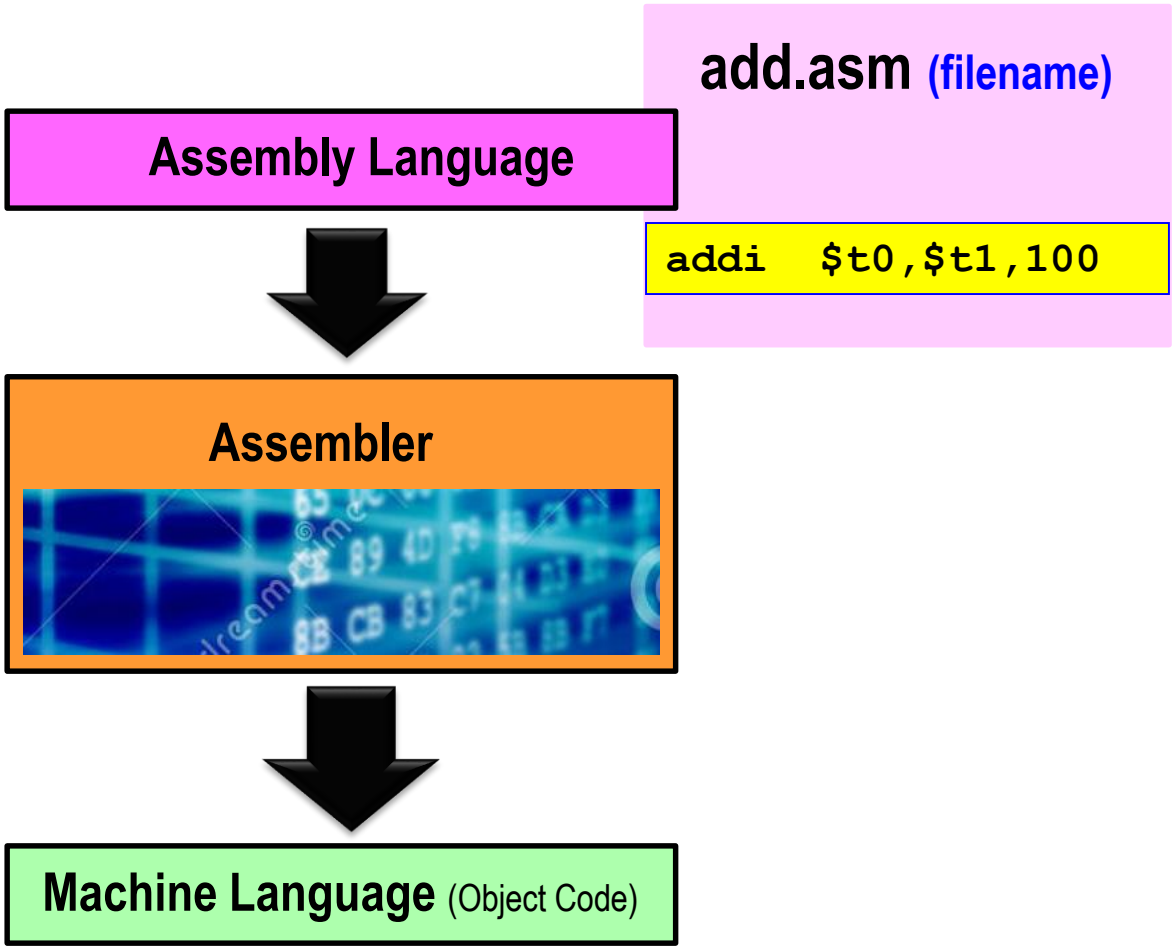
Why Assembly?

- **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 … software reverse engineering
  - Hacking into computer systems: buffer under/overflows, worms and Trojans.

Assembly-Machine Language

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

Assembly & Machine Languages



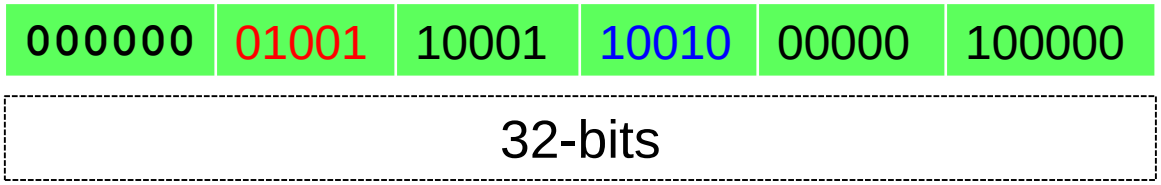
MIPS Architecture

- MIPS architecture is …
  - Register-to-Register
  - Load/Store
- The destination and sources must all be Registers
- To access the main memory, special instructions (Load/Store) are needed.

MIPS: Register File

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

- The data inputs and outputs are 32-bits wide.



MIPS register names convention

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

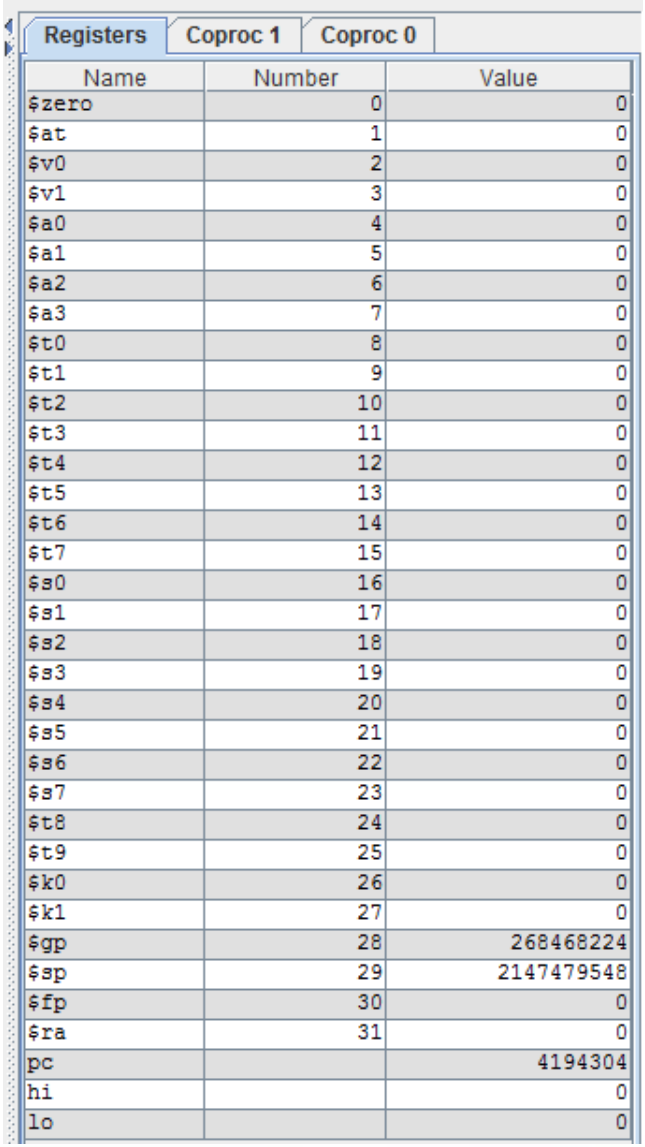
1. By number:

   `$0`, **`$1`**, … ,**`$31`**

2. By a letter and a number:

   `$a0-$a2` …

MIPS registers



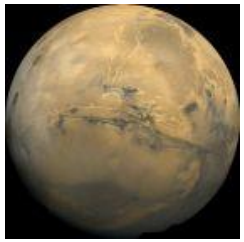
Registers | Coproc 1 | Coproc 0

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

Software http://courses.missouristate.edu/kenvollmar/mars/

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.



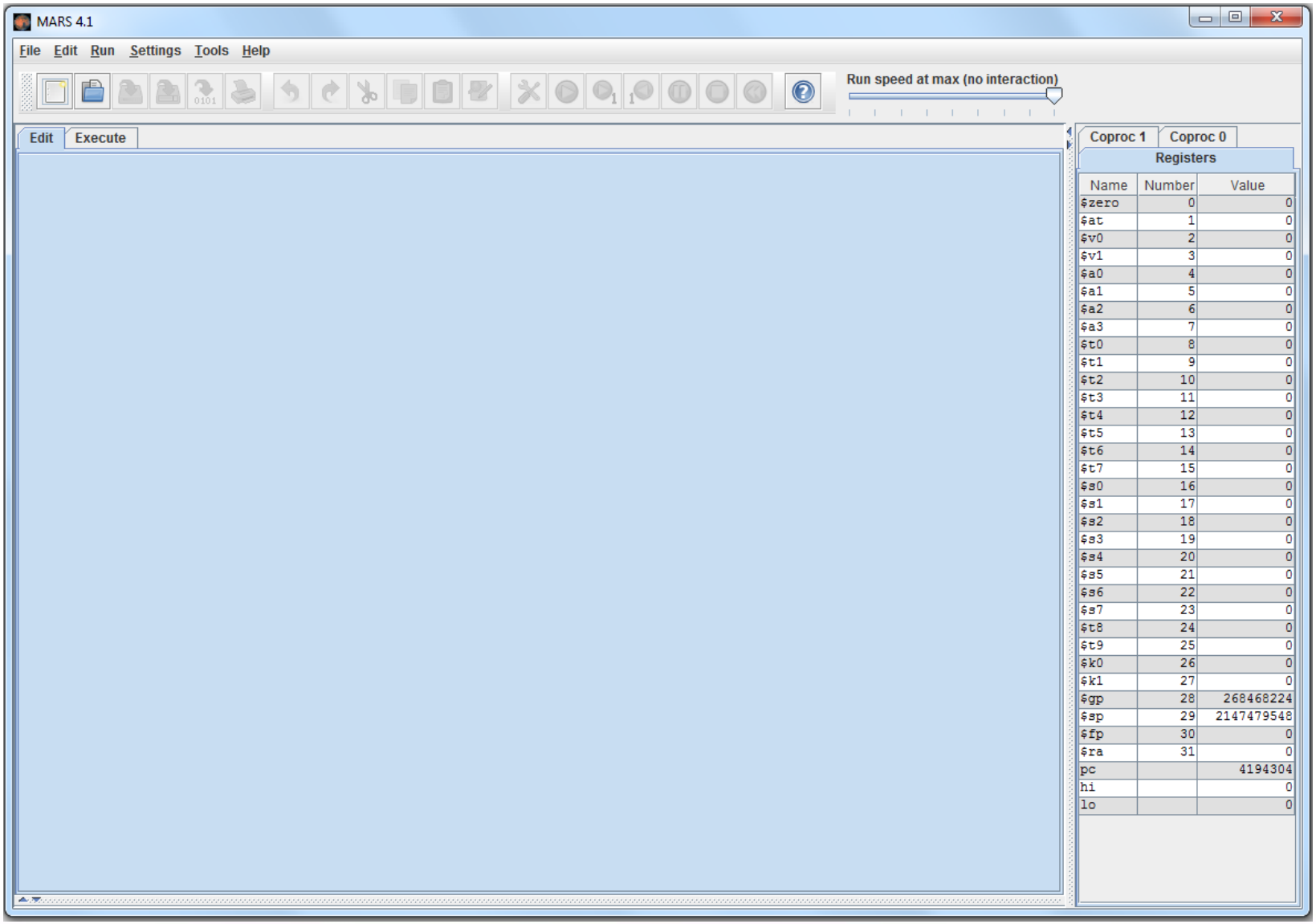
MARS (MIPS Assembler and Runtime Simulator)

An interactive development environment (*IDE) for MIPS Assembly Language Programming*

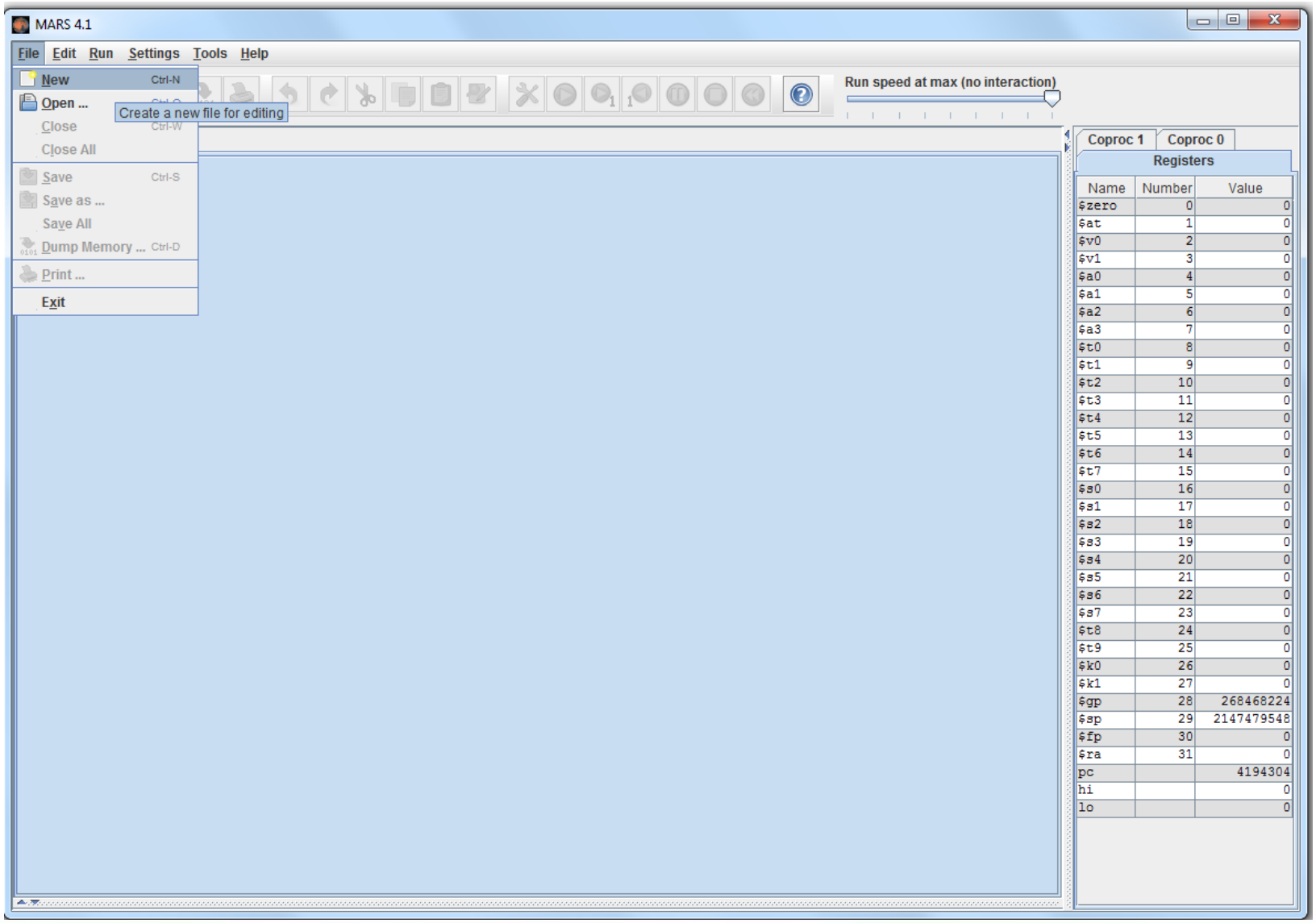
http://courses.missouristate.edu/kenvollmar/mars/

Download MARS

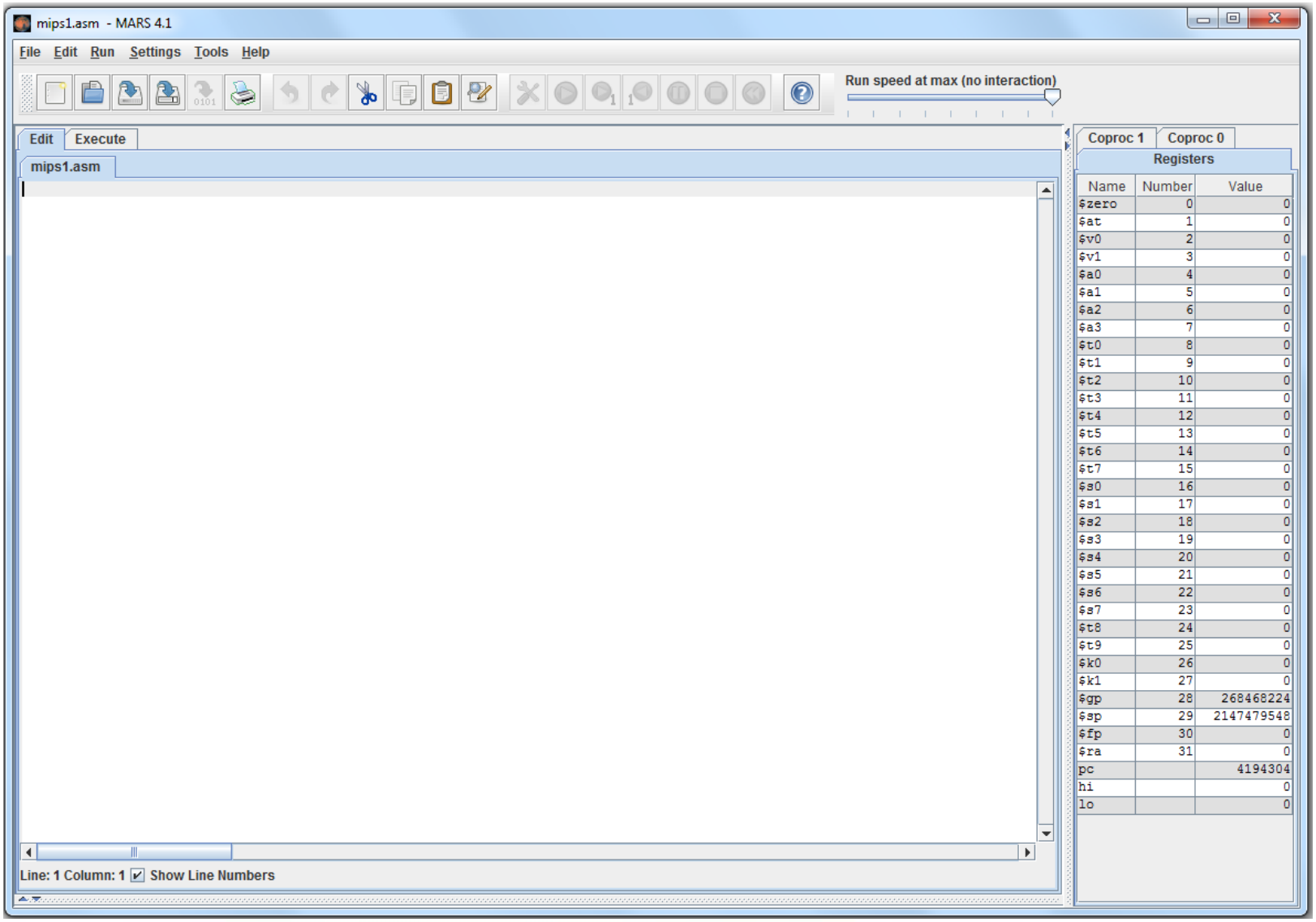
MARS


MARS 4.1
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Coproc 1 Coproc 0
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

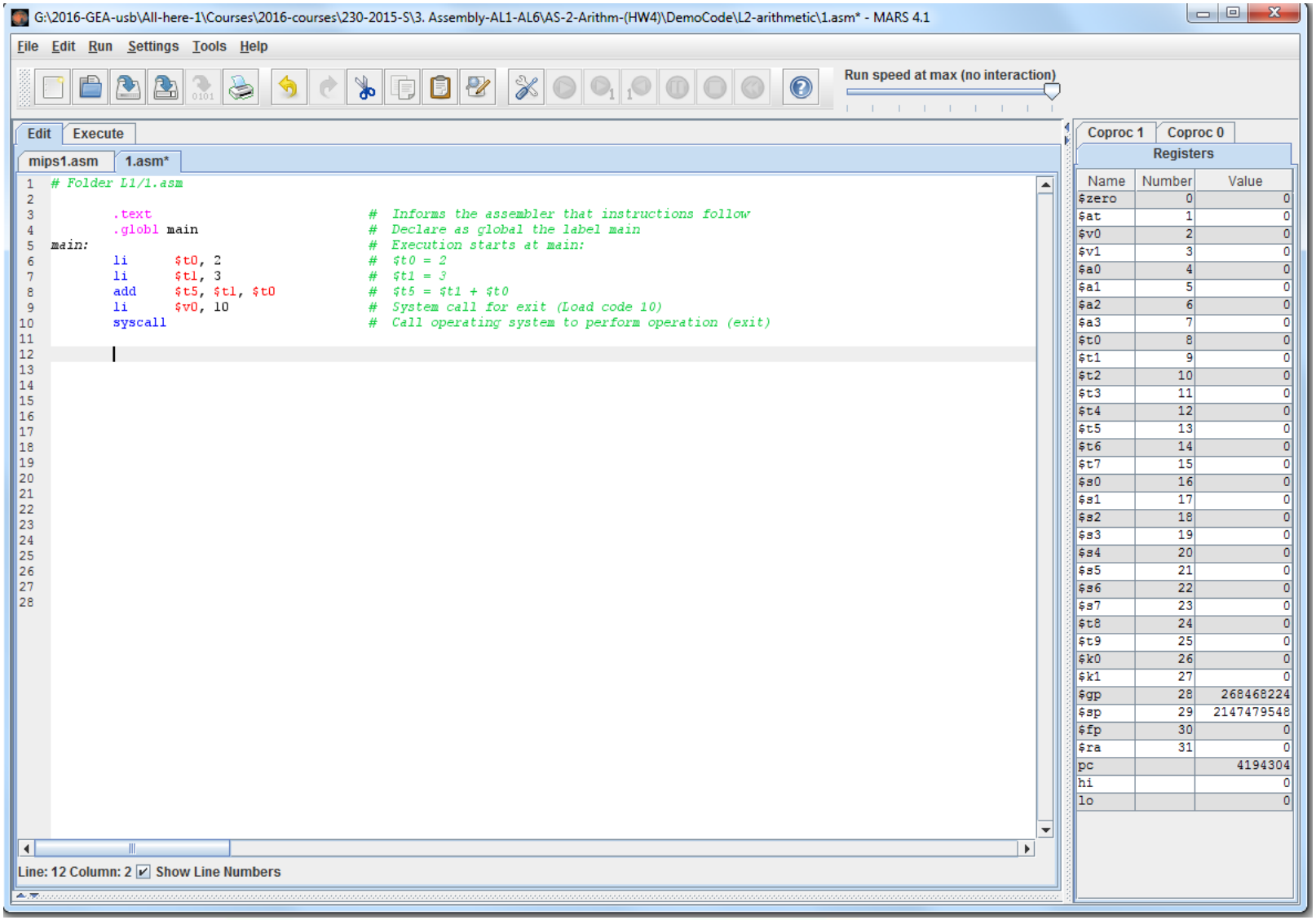
New [file] …


MARS 4.1
File Edit Run Settings Tools Help
New Ctrl-N
Open ...
Close Ctrl-W
Close All
Save Ctrl-S
Save as ...
Save All
Dump Memory ... Ctrl-D
Print ...
Exit
Create a new file for editing
Run speed at max (no interaction)
Coproc 1
Coproc 0
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

Write the code ...

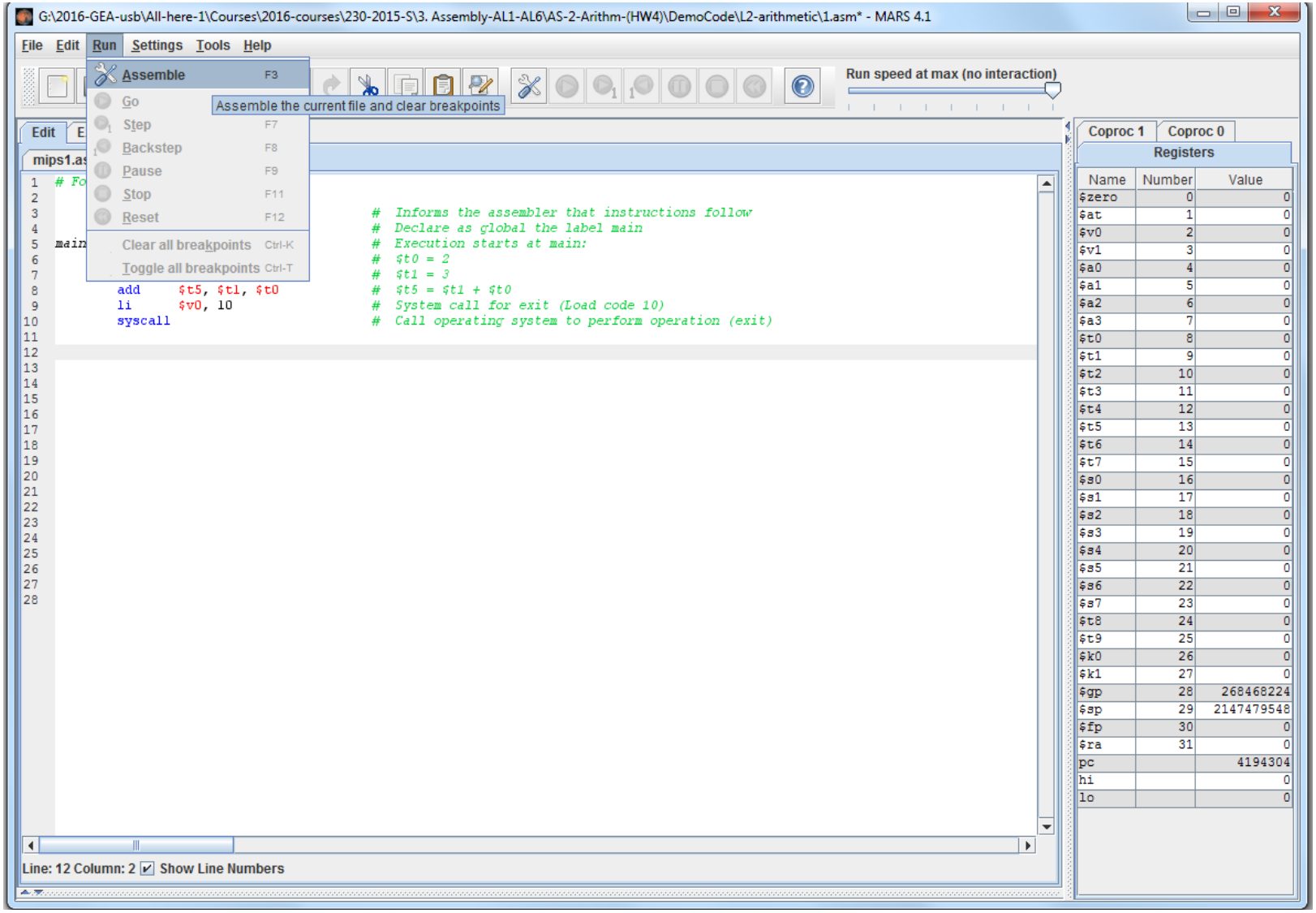


Save As >>> filename.asm

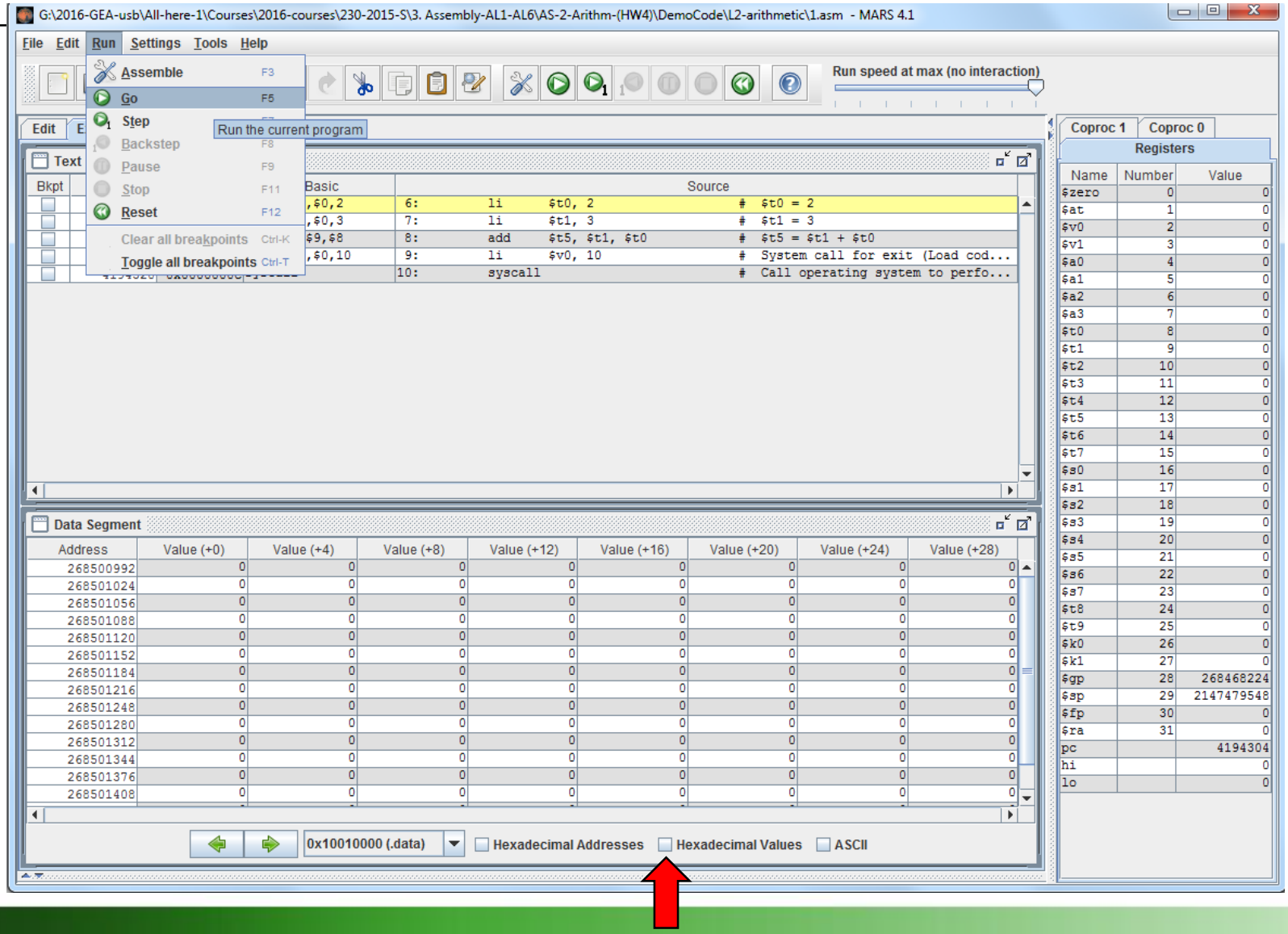


Run the code

Assemble

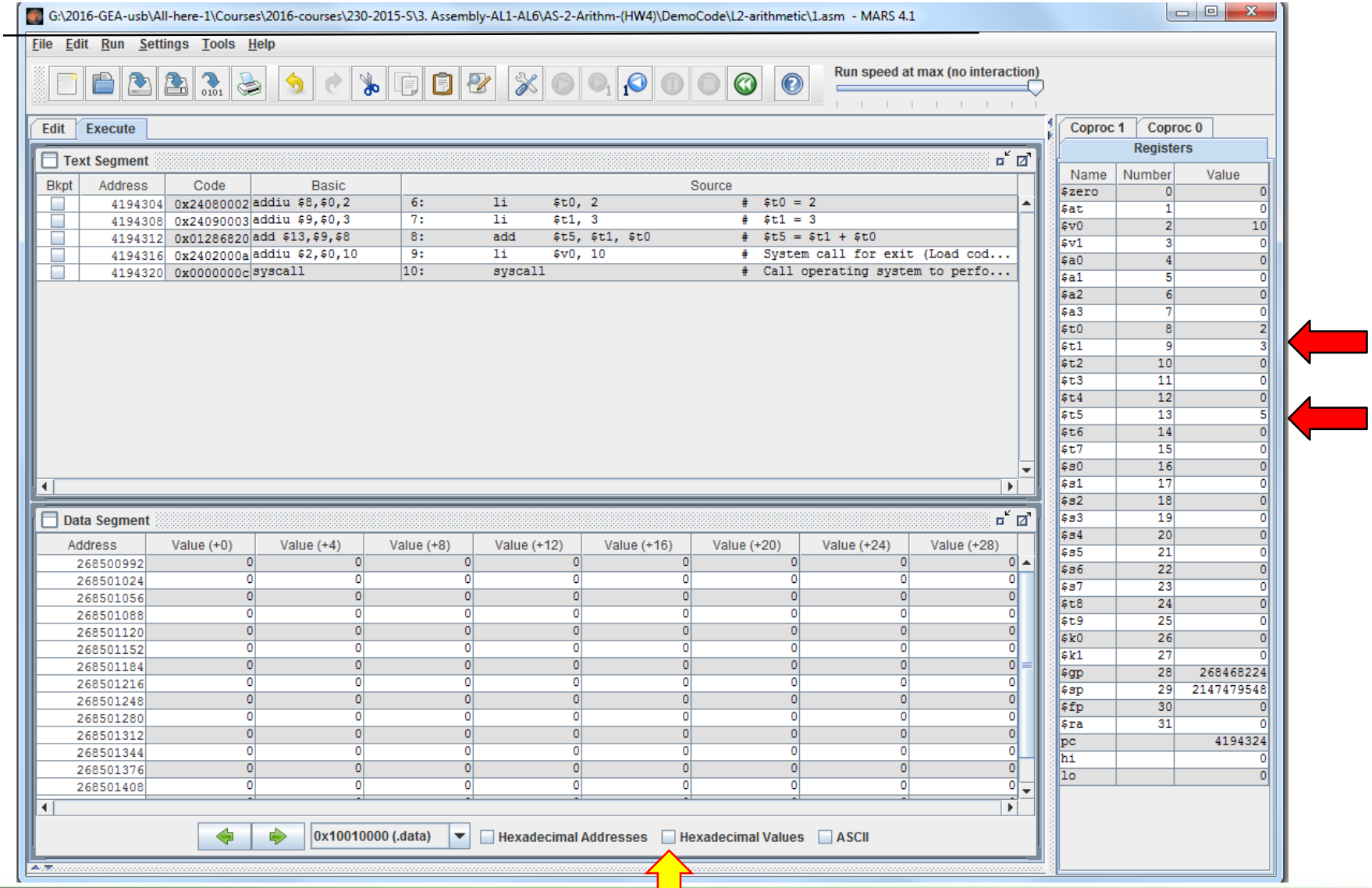

G:\2016-GEA-usb\All-here-1\Courses\2016-courses\230-2015-S\3. Assembly-AL1-AL6\AS-2-Arithm-(HW4)\DemoCode\L2-arithmetic\1.asm* - MARS 4.1
File Edit Run Settings Tools Help
Assemble F3
Go
Step F7
Backstep F8
Pause F9
Stop F11
Reset F12
Clear all breakpoints Ctrl-K
Toggle all breakpoints Ctrl-T
Assemble the current file and clear breakpoints
Run speed at max (no interaction)
Edit
mips1.as
# Informs the assembler that instructions follow
# Declare as global the label main
# Execution starts at main:
# $t0 = 2
# $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)
Line: 12 Column: 2 Show Line Numbers
Coproc 1 Coproc 0
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

Go


G:\2016-GEA-usb\All-here-1\Courses\2016-courses\230-2015-S\3. Assembly-AL1-AL6\AS-2-Arithm-(HW4)\DemoCode\L2-arithmetic\1.asm - MARS 4.1
File Edit Run Settings Tools Help
Assemble F3
Go F5
Step
Backstep F8
Pause F9
Stop F11
Reset F12
Clear all breakpoints Ctrl-K
Toggle all breakpoints Ctrl-T
Run the current program
Run speed at max (no interaction)
Text
Bkpt Basic Source
,$0,2 6: li $t0, 2 # $t0 = 2
,$0,3 7: li $t1, 3 # $t1 = 3
$9,$8 8: add $t5, $t1, $t0 # $t5 = $t1 + $t0
,$0,10 9: li $v0, 10 # System call for exit (Load cod...
10: syscall # Call operating system to perfo...
Data Segment
Address Value (+0) Value (+4) Value (+8) Value (+12) Value (+16) Value (+20) Value (+24) Value (+28)
268500992 0 0 0 0 0 0 0 0
268501024 0 0 0 0 0 0 0 0
268501056 0 0 0 0 0 0 0 0
268501088 0 0 0 0 0 0 0 0
268501120 0 0 0 0 0 0 0 0
268501152 0 0 0 0 0 0 0 0
268501184 0 0 0 0 0 0 0 0
268501216 0 0 0 0 0 0 0 0
268501248 0 0 0 0 0 0 0 0
268501280 0 0 0 0 0 0 0 0
268501312 0 0 0 0 0 0 0 0
268501344 0 0 0 0 0 0 0 0
268501376 0 0 0 0 0 0 0 0
268501408 0 0 0 0 0 0 0 0
0x10010000 (.data)
Hexadecimal Addresses
Hexadecimal Values
ASCII
Coproc 1 Coproc 0
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

The result

The output… uncheck for decimal results



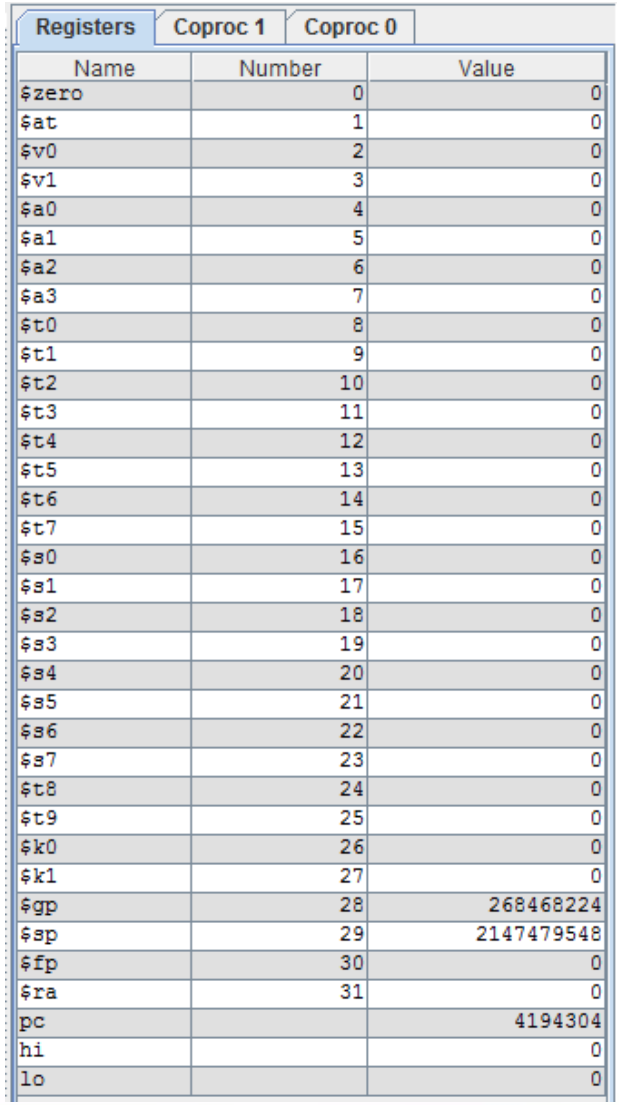
Our first assembly demo program

```
# Folder L1/1.asm

        .text
        .globl main
main:
        li      $t0, 2
        li      $t1, 3

        add     $t5, $t1, $t0

        li      $v0, 10
        syscall
```

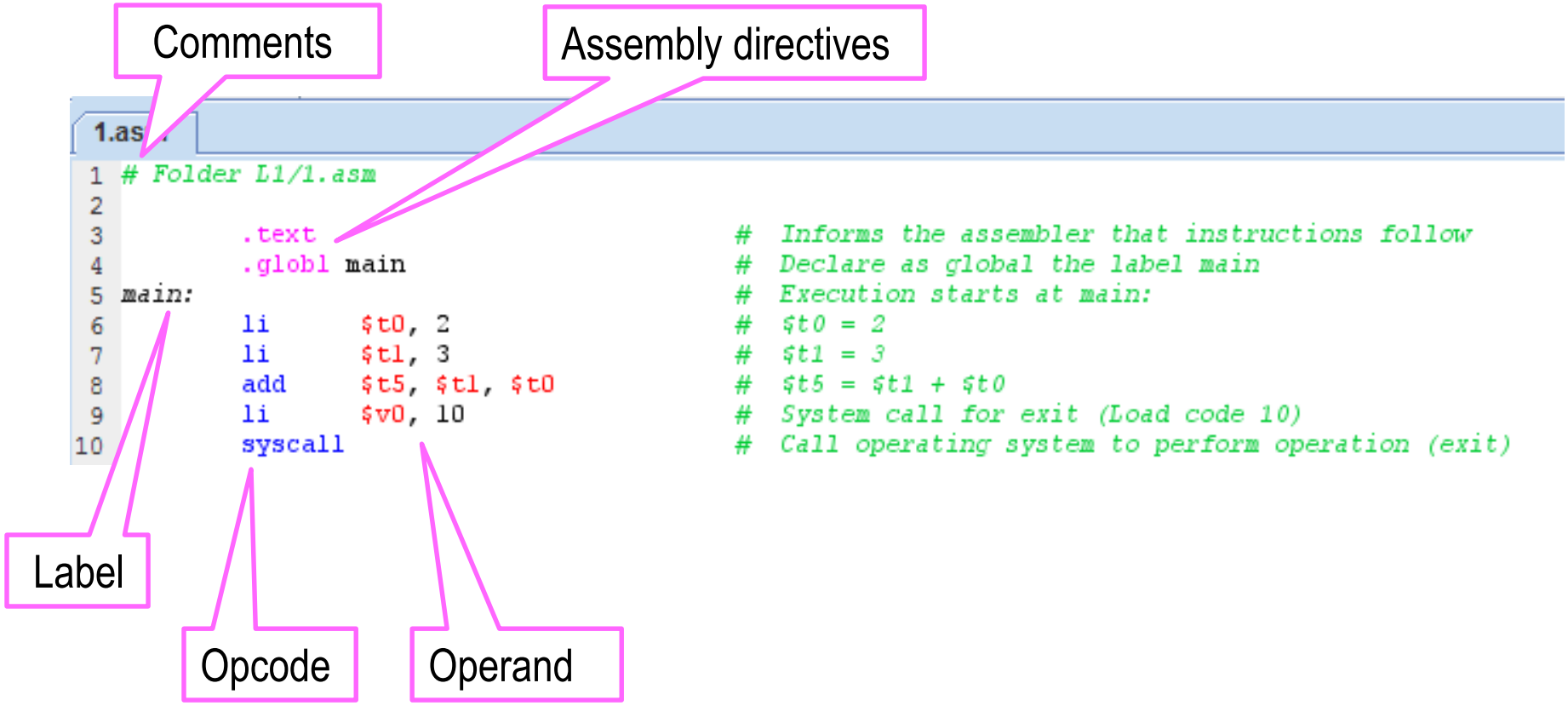


Registers | Coproc 1 | Coproc 0

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

System Call: 10

Our first assembly demo program



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

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 **(:)**

`.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*.

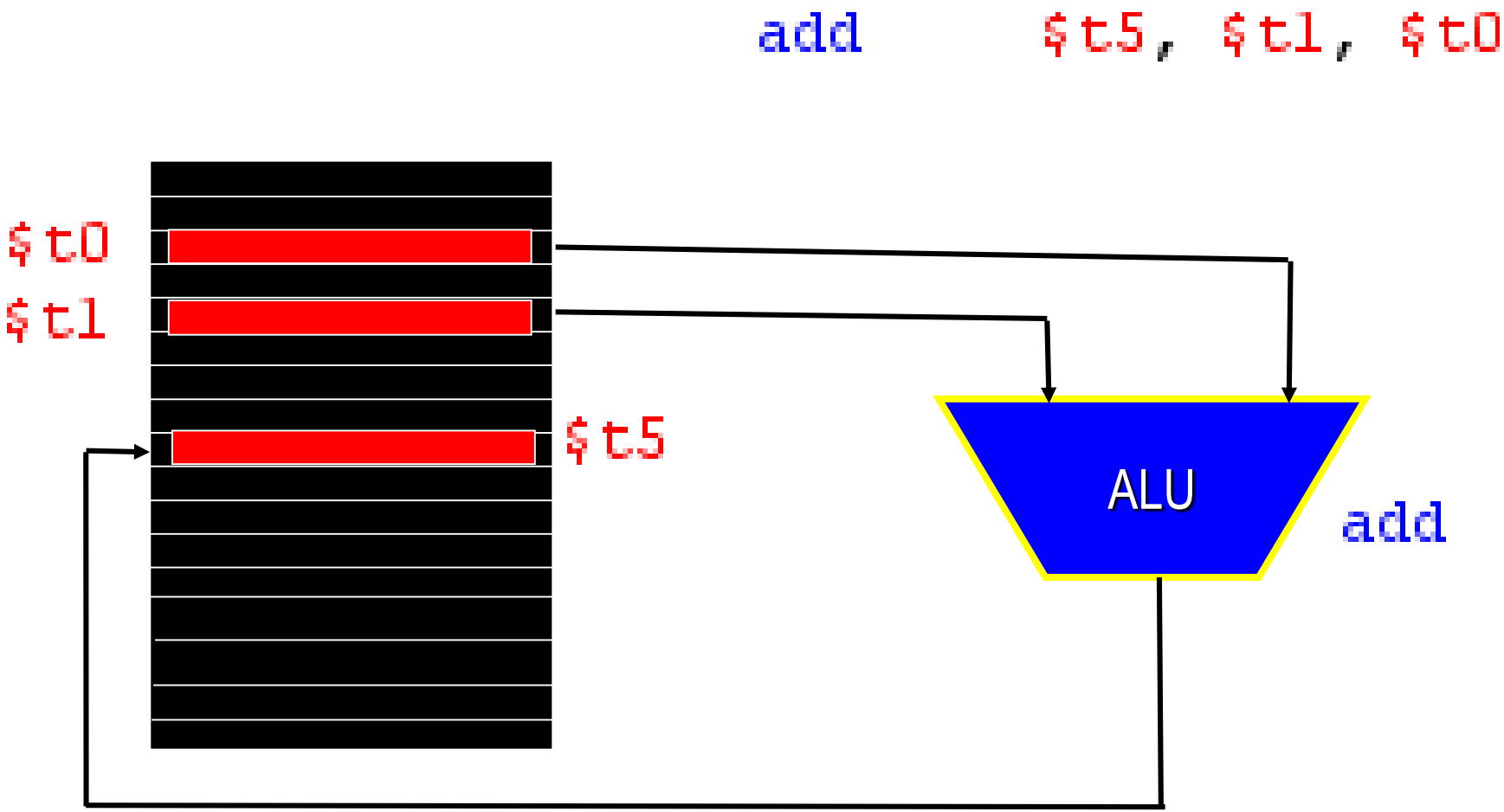
Assemble …and … GO

$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

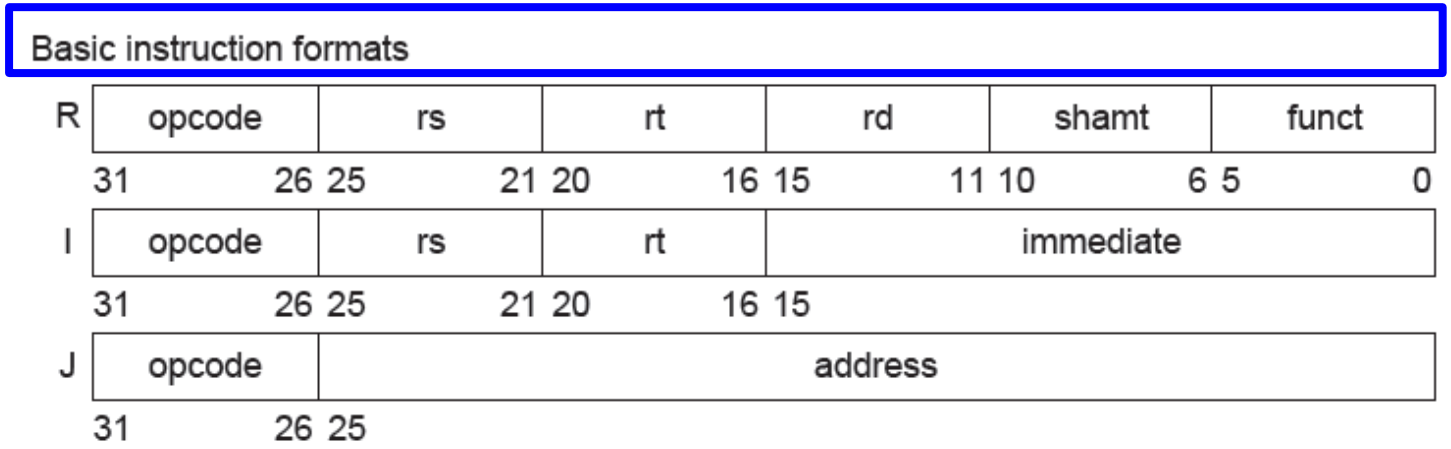


Instructions are divided into three kinds of format

- (**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)



MIPS instruction format (Floating-point)

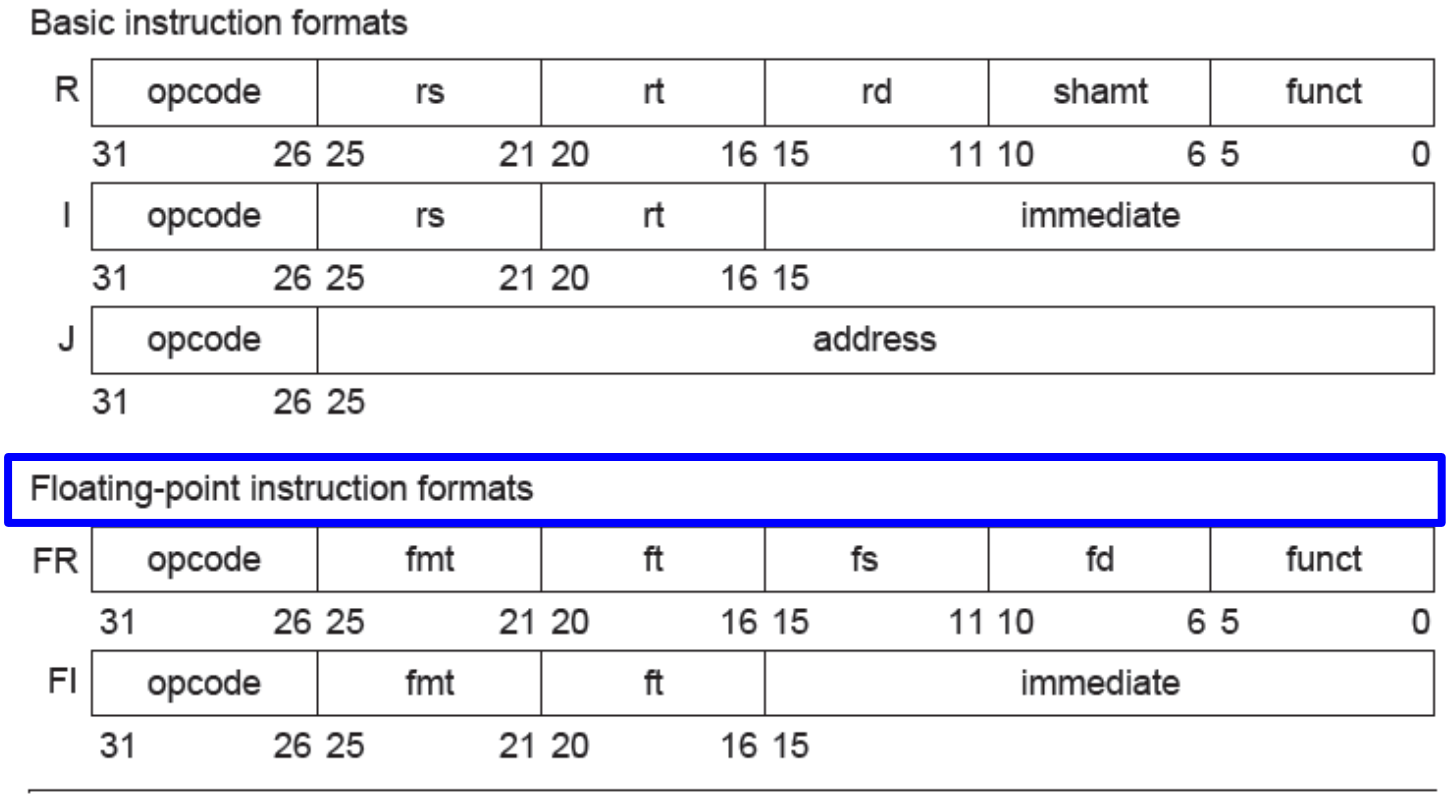


Figure 1.6 MIPS64 instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as DADDU, DSUBU, and so on. The I format is for data transfers, branches, and immediate instructions, such as LD, SD, BEQZ, and DADDIs. The J format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

MIPS Instruction set

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

Example:

Arithmetic

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

Example with **add**

```
 1 # Folder L1/1.asm
 2
 3           .text
 4           .globl main
 5 main:
 6           li      $t0, 2
 7           li      $t1, 3
 8           add     $t5, $t1, $t0
 9           li      $v0, 10
10           syscall
```

li

add

li

syscall

`add;` signed addition

- Performs the Binary Addition algorithm on two 32-bits;
  - Signed Binary

- `add $t5,$t1,$t0` `# [$t5]=[$t1]+[$t0]`
- 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.

More add instructions

More `add` instructions (MIPS)

- **`addu`**
- **`addiu`**

addu ← new instruction

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*.

`addu;` unsigned addition

- Performs the Binary Addition algorithm on two 32-bits ;
  - Unsigned Binary
- The destination register can be the same as one of the source registers
- `add(u) $t0, $t0, $t1` # [$t0] = [$t0] + [$t1]
- `addu` … ignores overflow trap.

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

`add` … `addu`

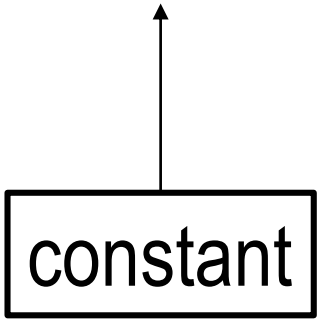
- The **`add`** instruction it is used in the cases that overflow is an important factor. Otherwise we use the **`addu`** instruction

  – For signed numbers, use **`add`**
  – For unsigned numbers, use **`addu`**.

`addiu` ← new instruction

addiu ← add immediate unsigned

- `addiu $t0, $t0, 1` `# [$t0] = [$t0] + 1`



(No overflow trap)

`addi` ← known add instruction

addi (add immediate)

```
addi $t2 $t1 4
```

addi	$rt	$rs	4
001000	01001	01010	0000000000000100
Op Code			Immediate value 4

addi … add-immediate

```
a = b + 4;  (10 is decimal)
```

MIPS Code

```
addi $t2,$t1,4
```

```
a = b - 4;  -4 is decimal)
```

MIPS Code

```
addi $t2,$t1,-4
```

Example with **`addi`**

addi

2.asm*

```
1  # Folder L1/2.asm
2
3          .text
4          .globl main
5  main:
6          li      $t0, 2
7          addi    $t5, $t0, 3
8          li      $v0, 10
9          syscall
10
```

addi

```
        .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

```
 1 # Folder L1/2.asm
 2
 3         .text
 4         .globl main
 5 main:
 6         li      $t0, 2
 7         addi    $t5, $t0, 3
 8         li      $v0, 10
 9         syscall
10
```

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

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

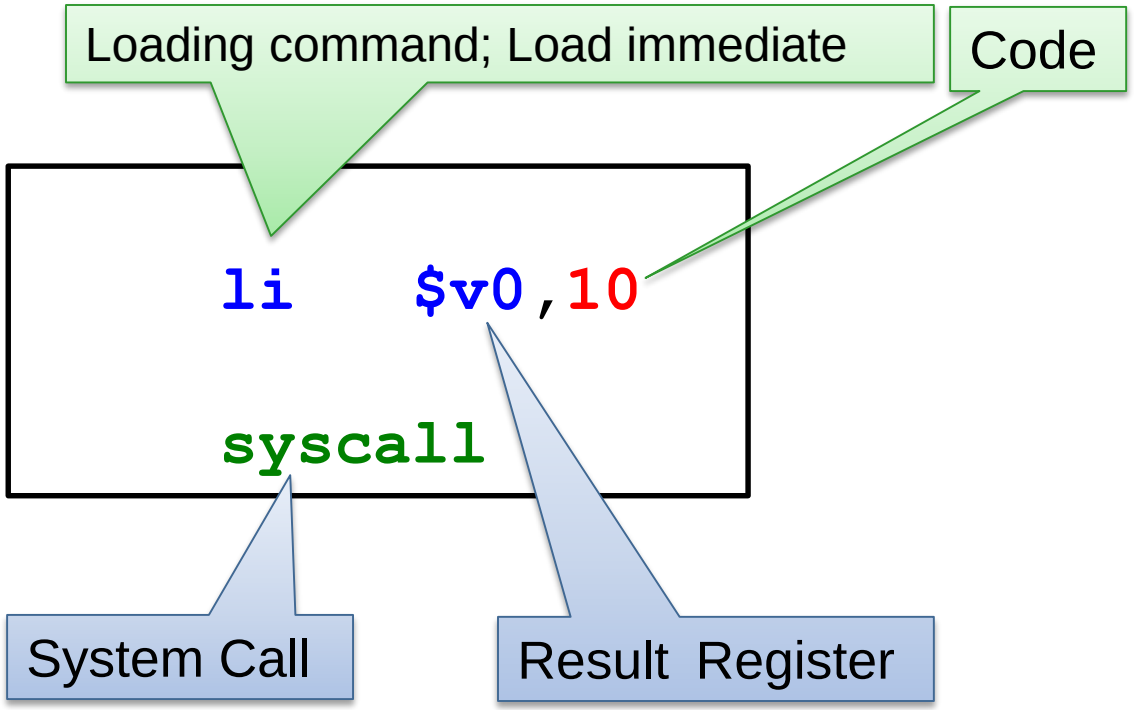
Another immediate instruction

li (load immediate)

- `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.

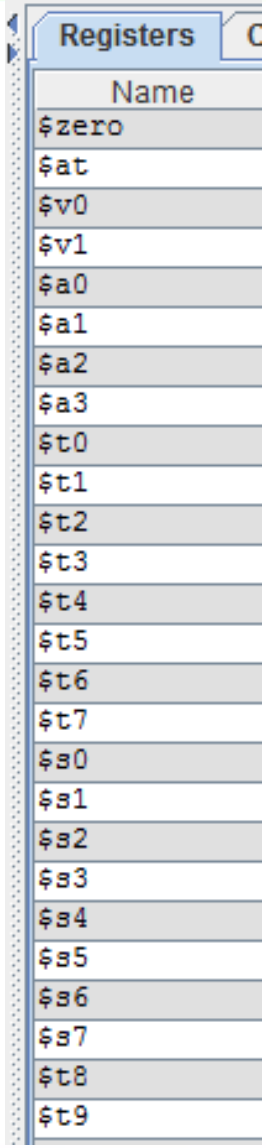
Example-1

Class examples

1. Add three numbers …

- **1 + 3 + 4**
- add the above numbers using only the instructions:

`add` and **`li`**



5 minutes…

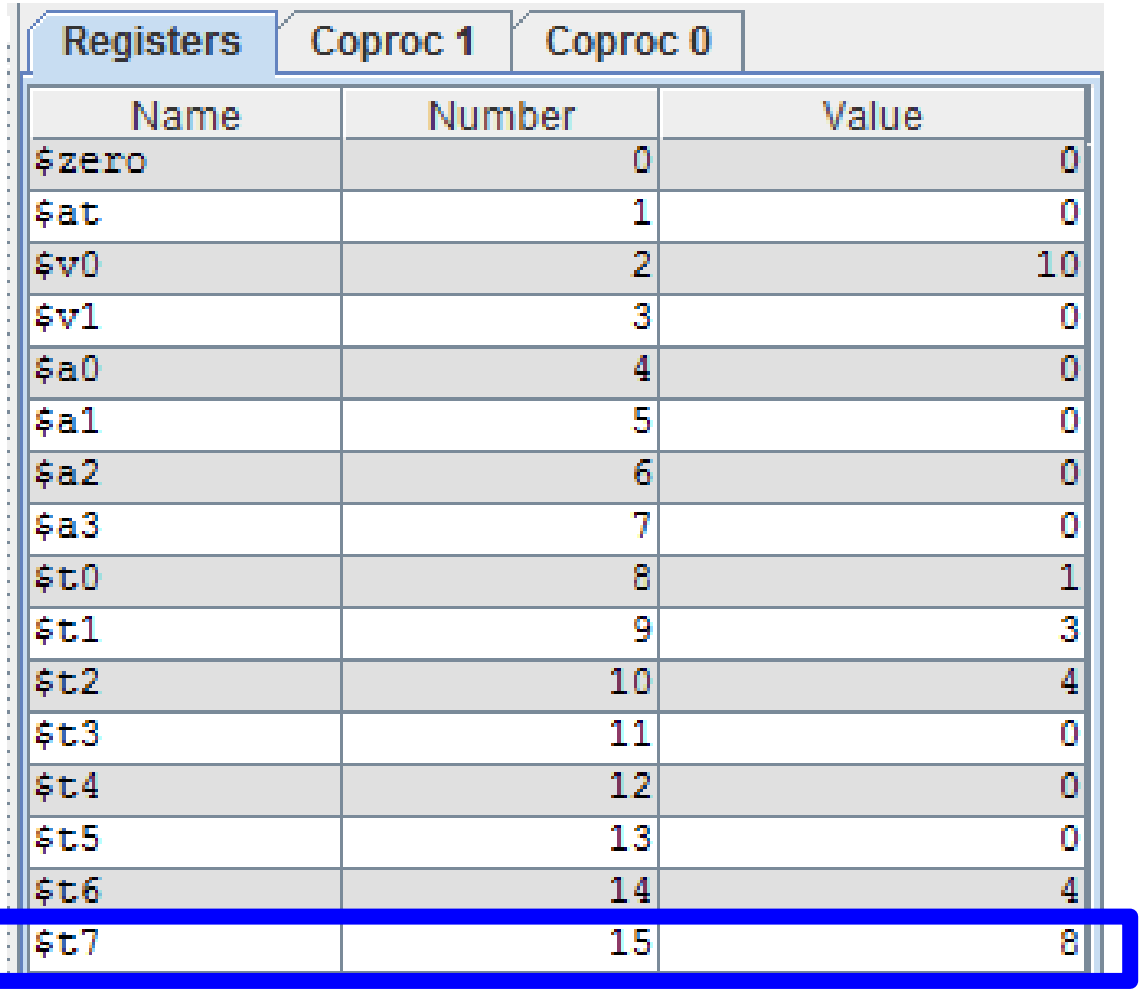
In Class

Program/Solution

```
        .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             # Exit (code 10)
        syscall
```

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

Example-2

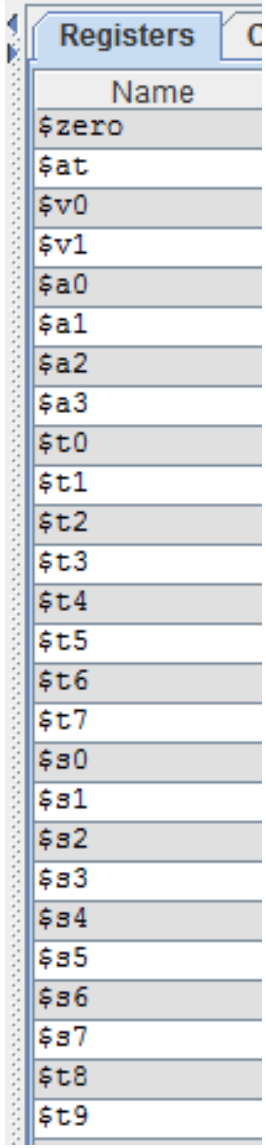
2. New problem: Add three numbers …

- **1 + 3 + 4**
- add the above numbers using only the instructions:

`addi` and **`li`**

5 minutes…

In Class



Solution

```
        .text
        .globl main
main:
        li    $t0, 1
        addi  $t1, $t0, 3
        addi  $t2, $t1, 4

        li    $v0, 10
        syscall
```

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	4
$t2	10	8
$t3	11	0

Example-3

3. New problem: Add three numbers …

- **1 + 3 + 4**
- add the above numbers using only the instructions:

`addi` and **`li`**

… use only one register **`($t0)`**

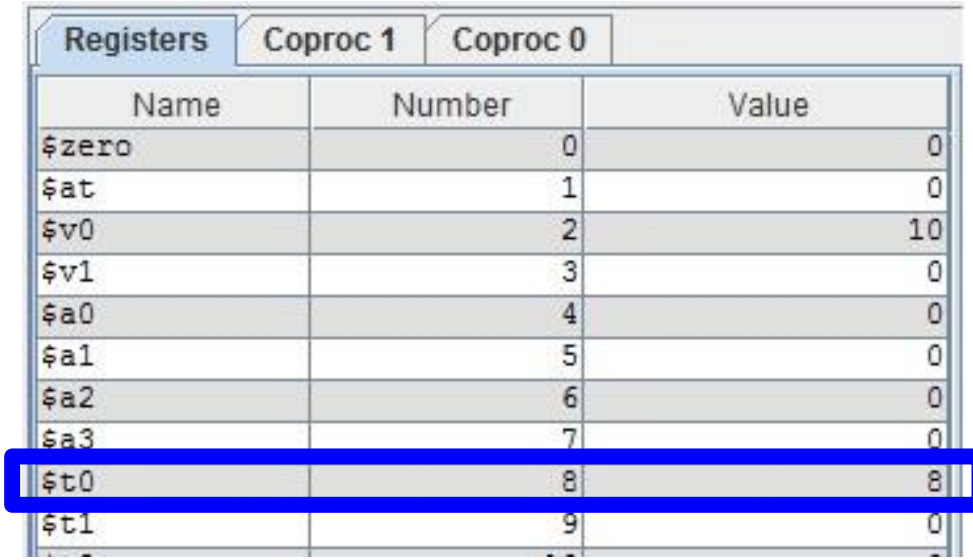
5 minutes…

Add three numbers … using **addi**

```
# Paul Kelusak, S2015

        .text
        .globl main
main:
        li    $t0, 1
        addi  $t0, $t0, 3
        addi  $t0, $t0, 4

        li    $v0, 10
        syscall
```



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	8
$t1	9	0