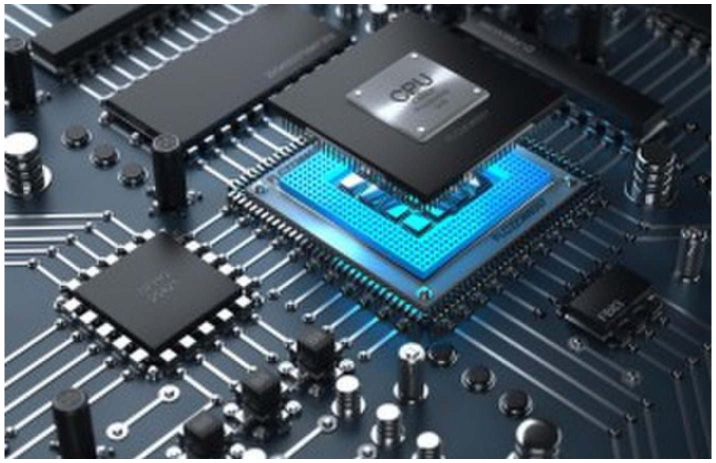


CPU Instructions

CS 154: Computer Architecture

Lecture #4

Winter 2020

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB

Administrative

- Lab 01 – due today!
- Lab 02 – description will be out soon!

Lecture Outline

- Instruction Set Architectures (ISA)

- MIPS instruction formats
- Refresher on some other MIPS instructions

Reference material from CS64 – I'll be going over this a little fast…

Other Factors to CPU Performance: **Power Consumption**

Market trends DEMAND that power consumption of CPUs keep decreasing.

BUT Power and Performance DON'T always go together…

- **Power = Capacitive Load x Voltage^2 x Clock Frequency**
- So:
  - Decreasing Voltage helps to get lower power, but it can make individual logic go slower!
  - Increasing clock frequency helps performance, but increases power!

- It's a dilemma that has contributed to Moore's Law "plateau"

Other Factors to CPU Performance: **Multiple Processors**

- Multicore microprocessors
  - More than one processor per chip

- Requires explicitly parallel programming
  - Compare with instruction level parallelism
  - Hardware executes multiple instructions at once
  - Hidden from the programmer

- Hard to do
  - Programming for performance
  - Load balancing
  - Optimizing communication and synchronization

Pitfalls: **Amdahl's Law**

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

- Your benchmark time is 100, 80 of which comes from a part of the CPU that you want to improve by a factor of ***n***, so:

  $T_{\textbf{improved}} = (80 / \boldsymbol{n}) + 20$

- If you wanted to improve your overall T by a factor of 2 (i.e. drop total from 100 to 50), then you'd need to make **n = 2.7**

  because 50 = (80 / 2.7) + 20 *…ok…*

- Keep that up! Let's go for a factor of **n = 5**, so drop total from 100 to 20:

  i.e. 20 ?= (80 / 5) + 20 *…uh… can't do that…* ☹

Pitfalls: **Idle Power**

- Simply put:
  CPUs will still draw *disproportionate* power when idling.
- Example, even when operating at 10% load,
  the i7 will draw 47% of the power
- Becomes a problem when dealing with large scale implementations, like data centers (Google, Facebook, Amazon, etc…)
- Design challenge: design processors to draw power more proportional to load (requires Physics-level approach, tho…)

Pitfall: MIPS as a Performance Metric

- Note: We're NOT talking about **MIPS the processor type**!!!!

- **MIPS** (millions of instructions per second) is a popular performance metric, *HOWEVER…*

- Doesn't account for
  - Differences in ISAs between computers
    (some ISAs may be more efficient than others)
  - Differences in complexity between instructions (weighted CPIs)

Instruction Set Architecture (ISA)

- The “contract” between software and hardware
  (hence, it’s an abstract model of a computer!)

- Typically described with:
  - programmer-visible states (i.e. registers + memory)
  - the semantics/syntax of the instructions
  - Examples abound in your MIPS Reference Card!

Instruction Set Architecture (ISA)

Many implementations possible for a given ISA

- Most microprocessor families have their own ISA
- Some can be shared across families (b/c they're popular)
  - Example: AMD and Intel processors both run the x86-64 ISA (orig. Intel).
- Some of the same ISAs can be *customized*
  - Many cellphones use the ARM ISA with specific implementations from many different companies including
    Apple, Qualcomm, Samsung, Huawei, etc.
- We'll be using the MIPS ISA in this class.

Classification of ISAs

Intel, AMD (x86)	CISC
ARM, MIPS	RISC
GPUs (AMD, Nvidia)	RISC
Intel/HP (IA-86)	EPIC

- By architectural complexity*
  - **CISC** (complex instruction set computer) and **RISC** (reduced instruction set computer)

** Most popular distinction in commercial CPUs*

- By instruction-level parallelism
  - **VLIW** (very long instruction word) and **EPIC** (explicitly parallel instruction computing)
- By extreme simplification of instructions
  - **MISC** (minimal instruction set computer) and **OISC** (one instruction set computer)

CISC vs RISC:
- *Higher instruction complexity (and CPI)*
- *More transistors*
- *Higher power*
- *Commercial computers vs. embedded computers*

EPIC/VLIW:
- *Less commercial than CISC/RISC*
- *Server/supercomputer use mostly*

MISC/OISC:
- *Little to no parallelism*
- *Mostly in research*

The MIPS ISA

- Developed at Stanford then commercialized by MIPS Technologies, created/led by John Hennessey
  - Stanford CS prof, President (2000-16), author of our textbook…
  - Started multiple important SV companies,
    current Chair of Alphabet, Inc.

- Hennessey and Patterson won the 2017 Turing Award for their work in developing RISC architecture

- MIPS still has a large share of embedded core market
  - Consumer electronics, storage peripherals, cameras, printers, …

Code on MIPS

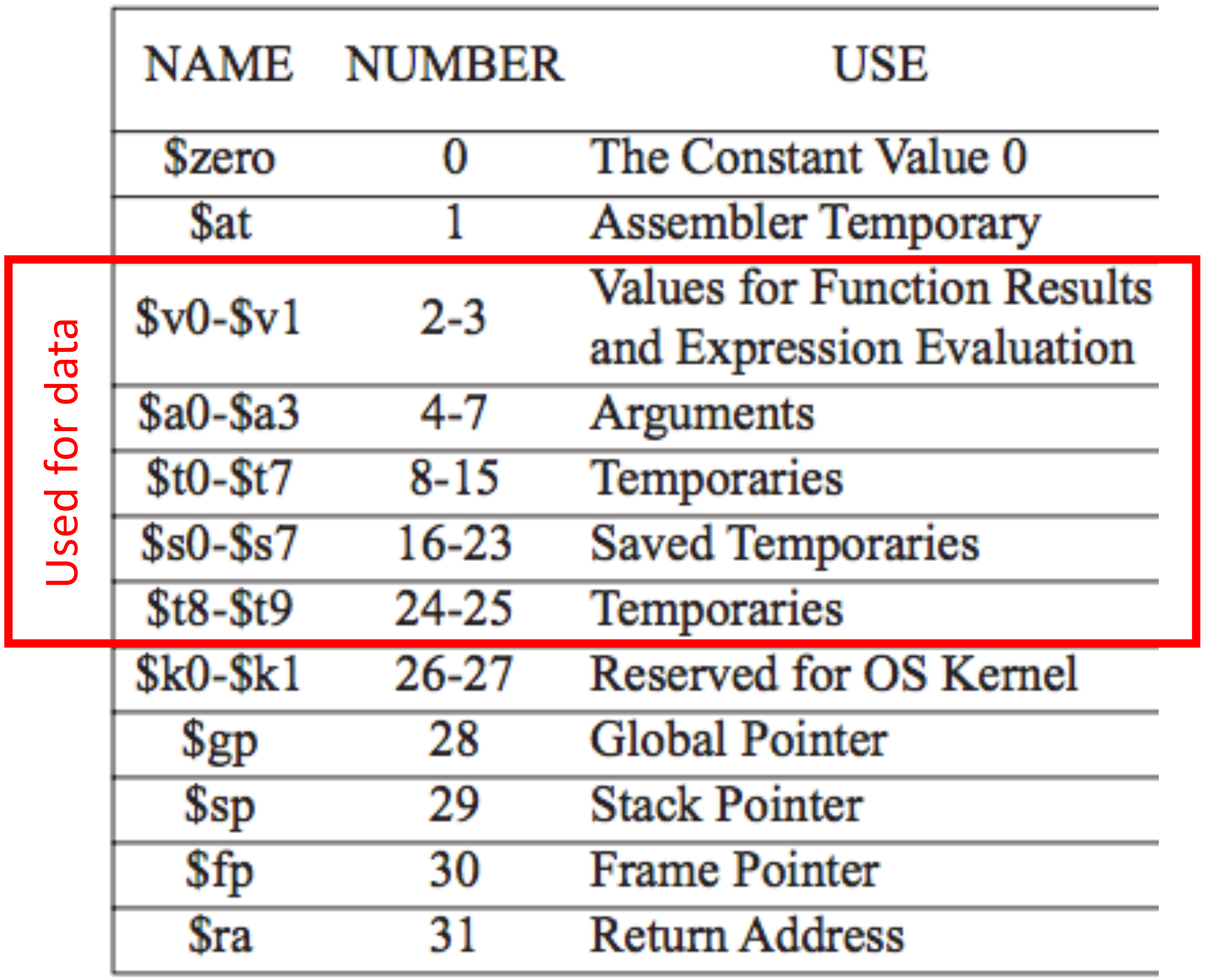
Original	**MIPS**

`x = 5;`
`y = 7;`
`z = x + y;`
	`li $t0, 5`
	`li $t1, 7`
	`add $t3, $t0, $t1`

Available Registers in MIPS

32 registers in all

- Refer to your MIPS Reference Card
- Bring it to class from now on…
- Copy on main webpage



Used for data

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

MIPS Instruction Formats

- Each instruction is represented with **32 bits**

- There are **three** different ***instruction formats***: **R**, **I**, **J**
  - These allow for instructions to take on different roles
  - R-Format is used when it's all about **registers**
  - I-Format is used when you involve **(immediate) numbers**
  - J-Format is used when you do code "**jumping**"
    (i.e. branching)

Instruction Register

?

Registers

```
$t0: ?
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Remember: Memory is addressed in Bytes.

Program Counter

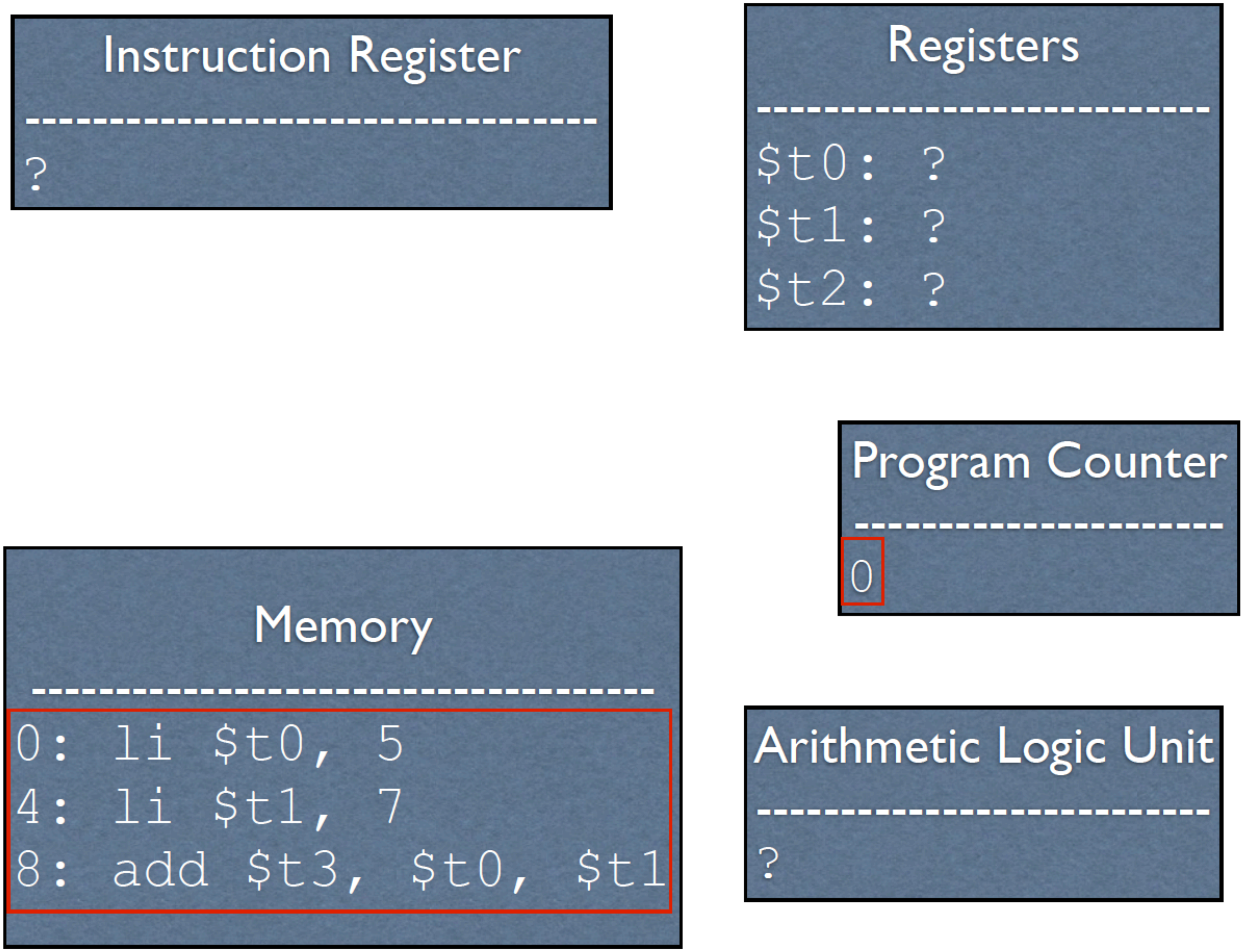
?

Memory

?

Arithmetic Logic Unit

?



```
0:  li $t0, 5
4:  li $t1, 7
8:  add $t3, $t0, $t1
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: ?
$t1: ?
$t2: ?
```

Program Counter

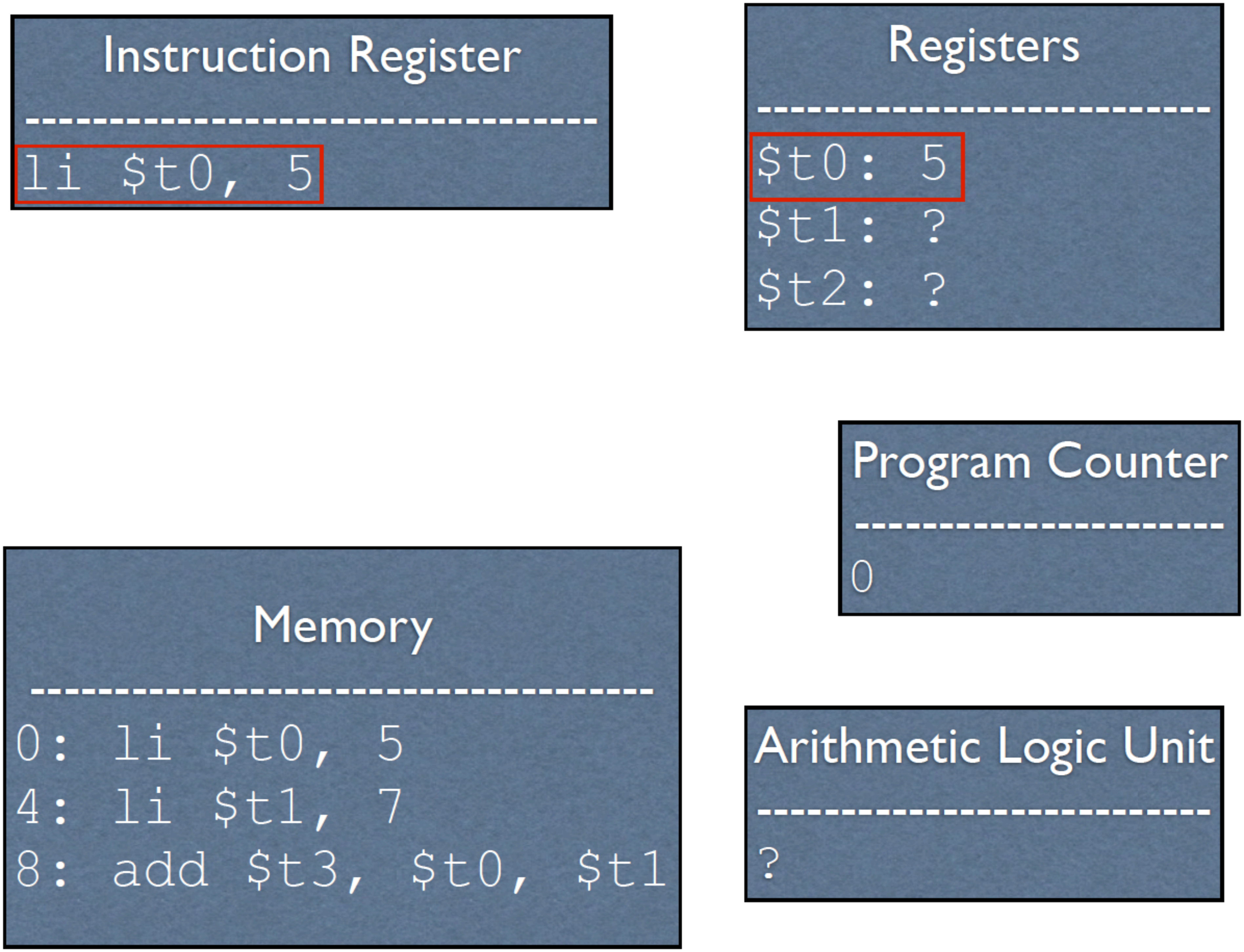
```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```



Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Program Counter

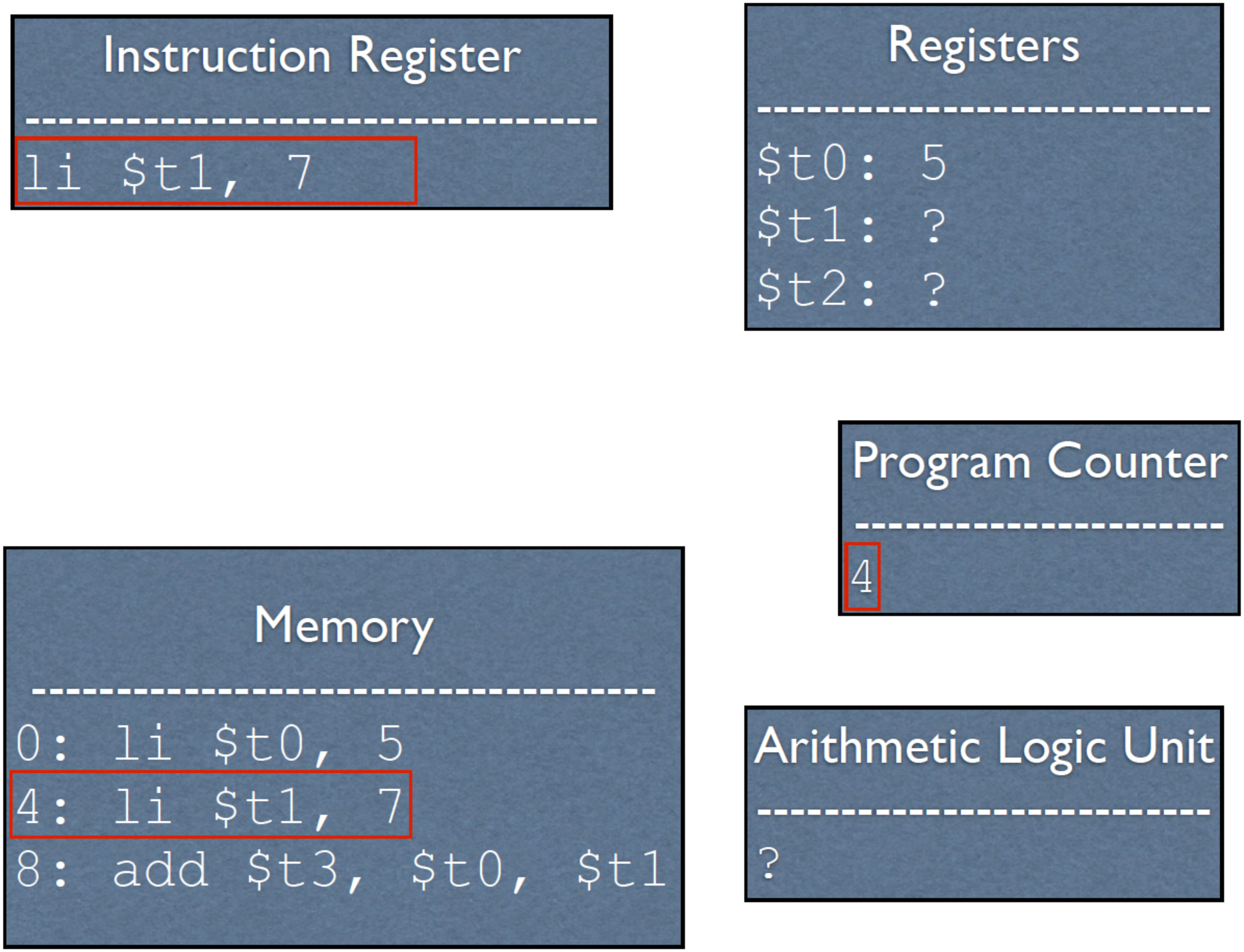
```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
0 + 4 = 4
```



Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Program Counter

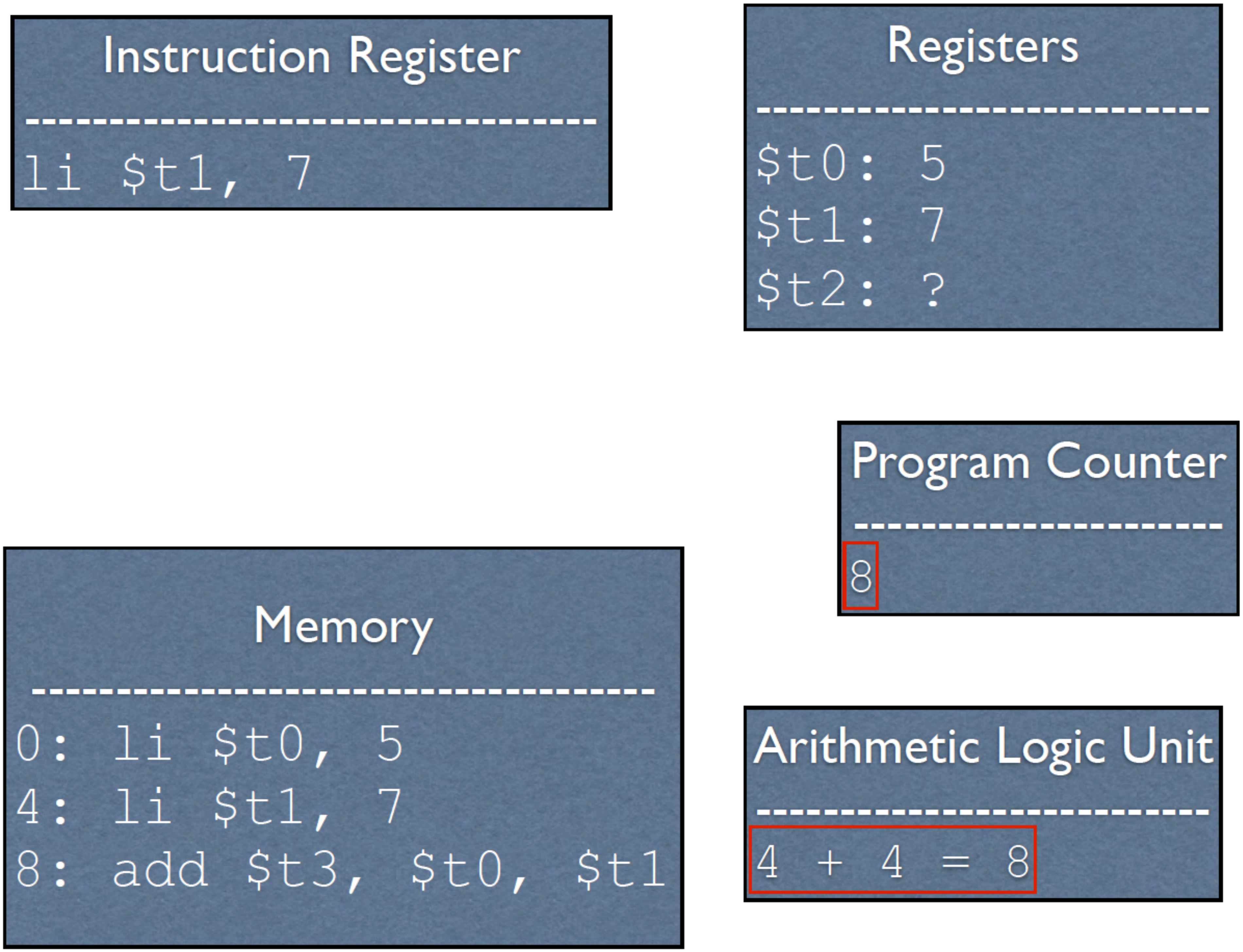
```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```



Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
4 + 4 = 8
```

Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Program Counter

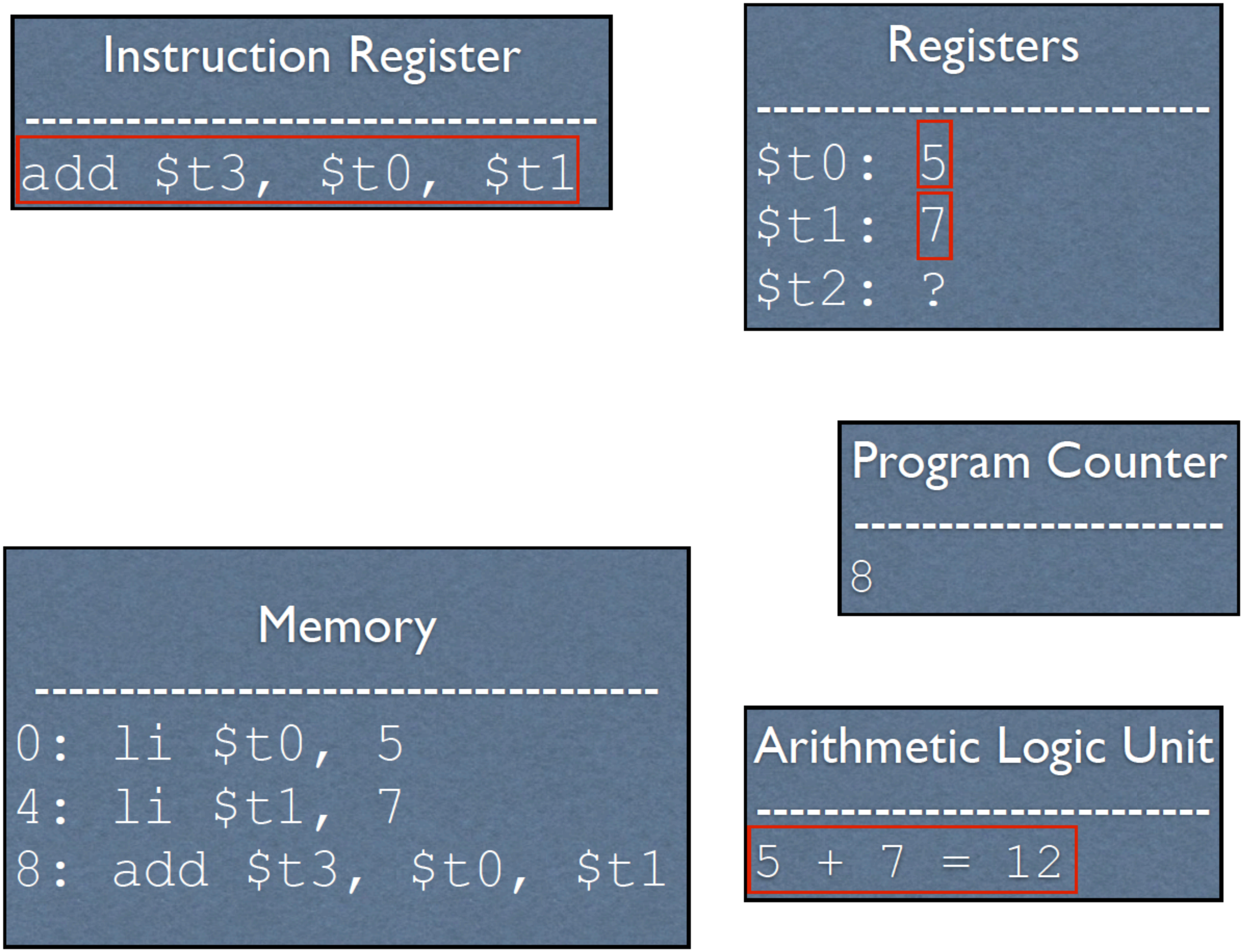
```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```



Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
5 + 7 = 12
```

Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: 12
```

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
5 + 7 = 12
```

Talking to the OS

- We are going to be running on MIPS ***emulator*** called **SPIM**
  - Optionally, through a program called **QtSPIM** (GUI based)
  - *What is an emulator?*
- MIPS features a **`syscall`** instruction, which triggers a ***software interrupt***, or ***exception***
- Outside of an emulator (i.e. in the real world), these instructions **pause the program** and tell the OS to go do something with I/O
- Inside the emulator, it tells the emulator to go *emulate* something with I/O

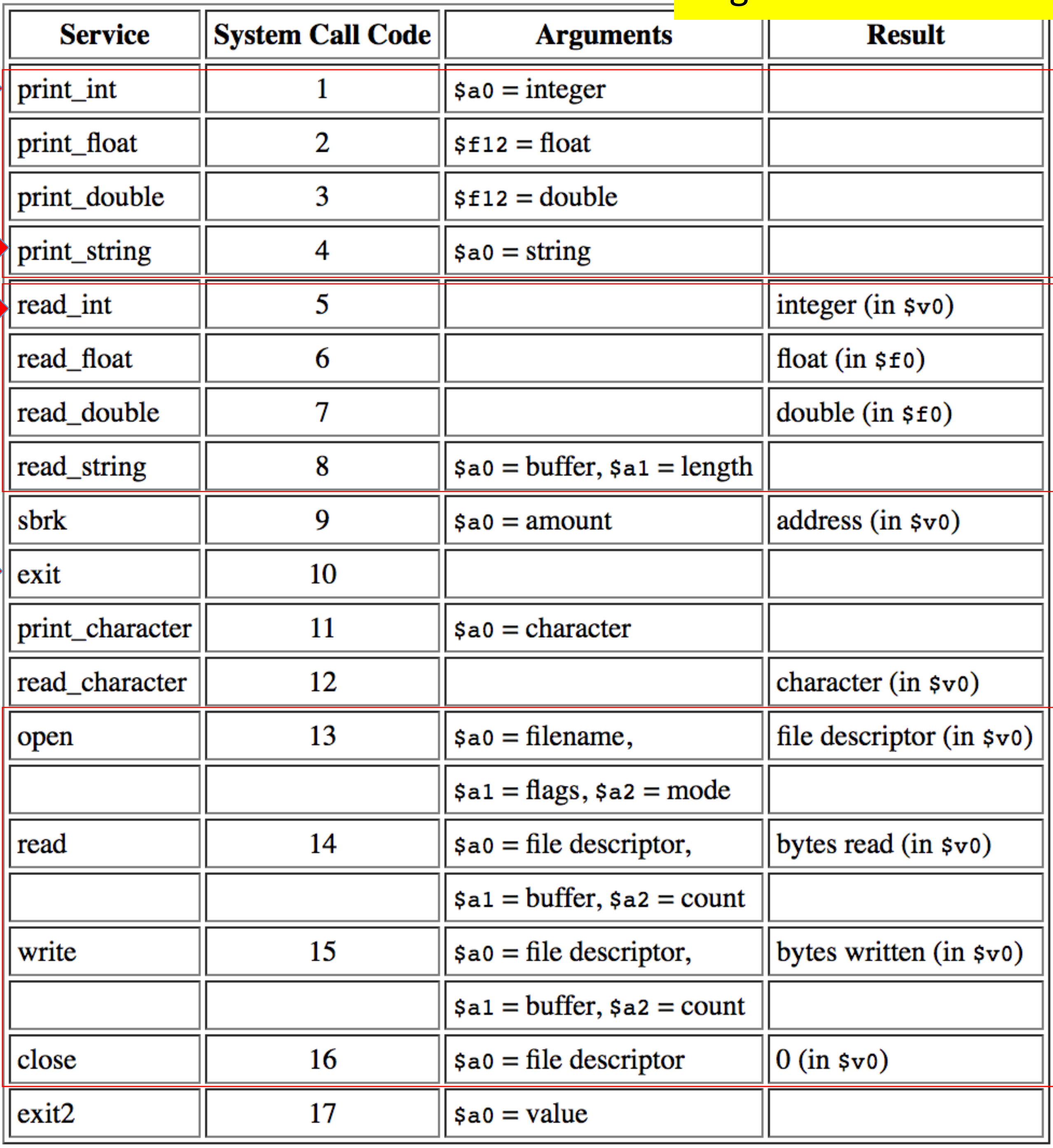
`syscall` (for spim use)

- The OS/emulator has access to the CPU registers
- So we have the OS/emulator's attention, but how does it know what we want?
- We put special values (codes) in the registers to indicate what we want
  - These are codes that can't be used for anything else, so they're understood to be just for `syscall`
  - So… is there a "code book"????

Yes! All CPUs come with manuals.
For us, we have the **MIPS Ref. Card**

MIPS System Services

System call code: placed in **$v0**
Argument: placed in **$a0**



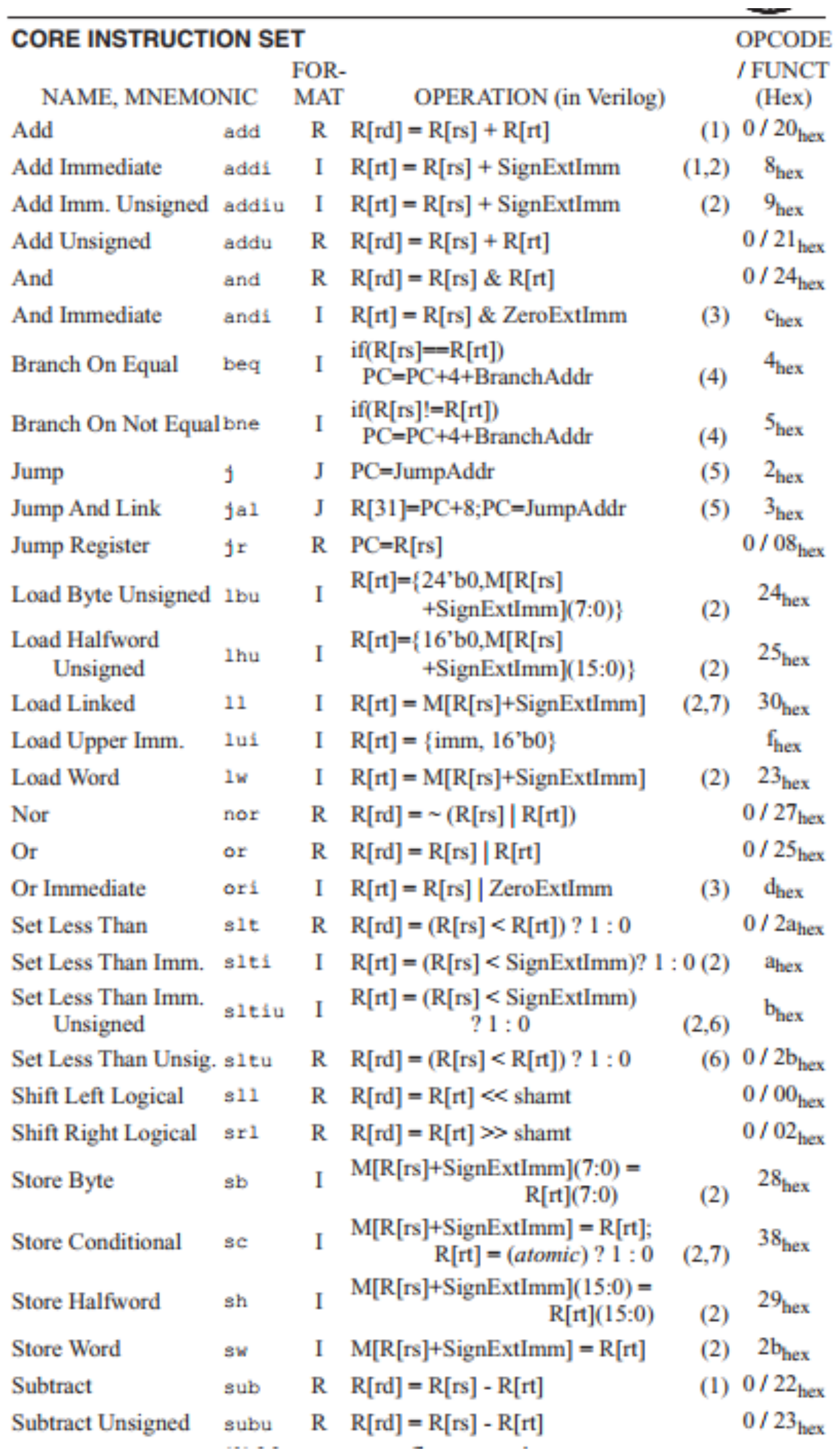
Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	
sbrk	9	`$a0` = amount	address (in `$v0`)
exit	10		
print_character	11	`$a0` = character	
read_character	12		character (in `$v0`)
open	13	`$a0` = filename,	file descriptor (in `$v0`)
		`$a1` = flags, `$a2` = mode	
read	14	`$a0` = file descriptor,	bytes read (in `$v0`)
		`$a1` = buffer, `$a2` = count	
write	15	`$a0` = file descriptor,	bytes written (in `$v0`)
		`$a1` = buffer, `$a2` = count	
close	16	`$a0` = file descriptor	0 (in `$v0`)
exit2	17	`$a0` = value	

stdout

stdin

File I/O

Bring out your MIPS Reference Cards!



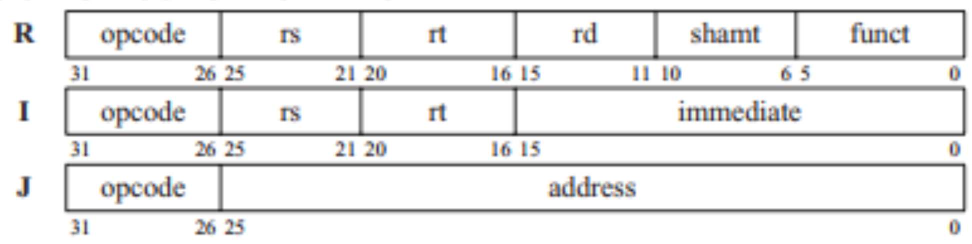
CORE INSTRUCTION SET

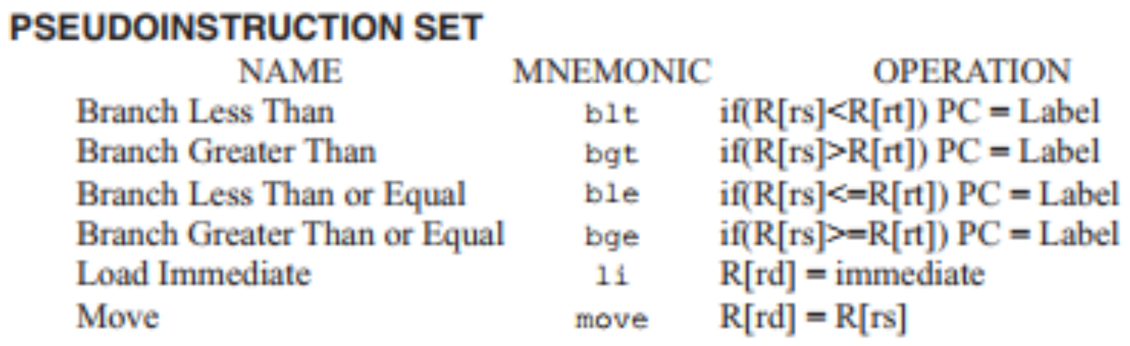
NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

NOTE THE FOLLOWING:

1. Instruction Format Types: **R** vs **I** vs **J**
2. OPCODE/FUNCT (Hex)
3. Instruction formats: Where the actual bits go

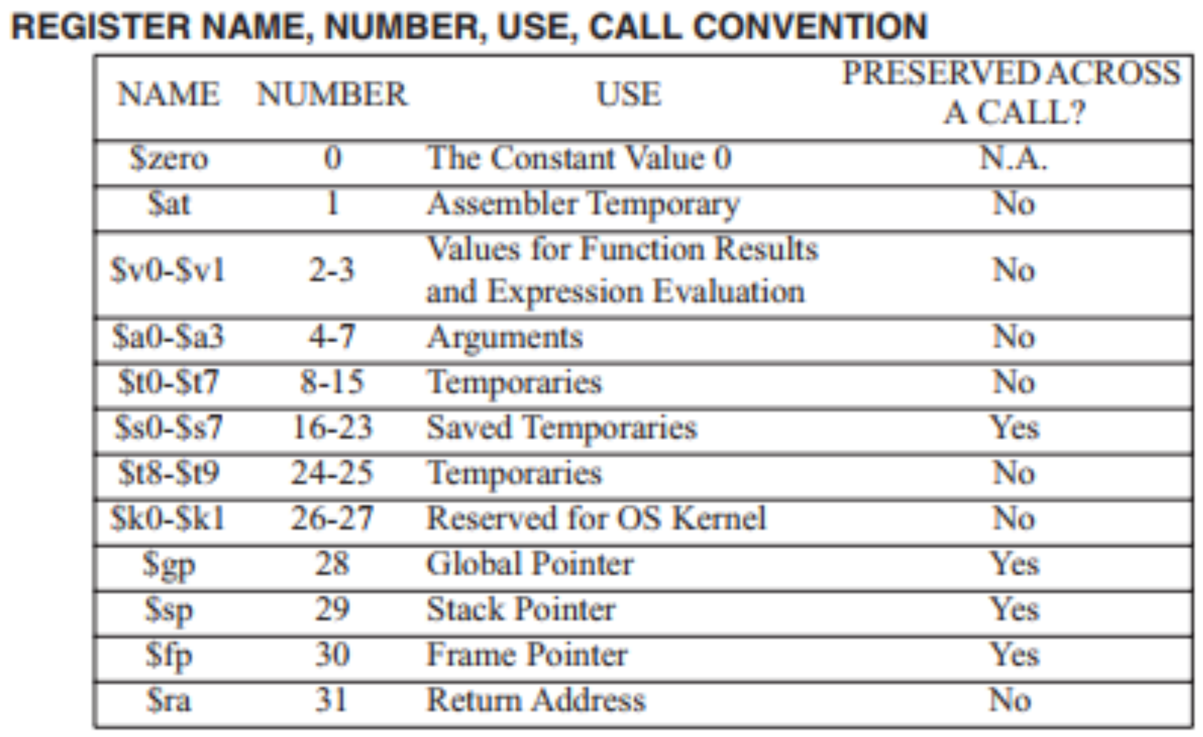
BASIC INSTRUCTION FORMATS





PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	`blt`	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	`bgt`	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	`ble`	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	`bge`	if(R[rs]>=R[rt]) PC = Label
Load Immediate	`li`	R[rd] = immediate
Move	`move`	R[rd] = R[rs]



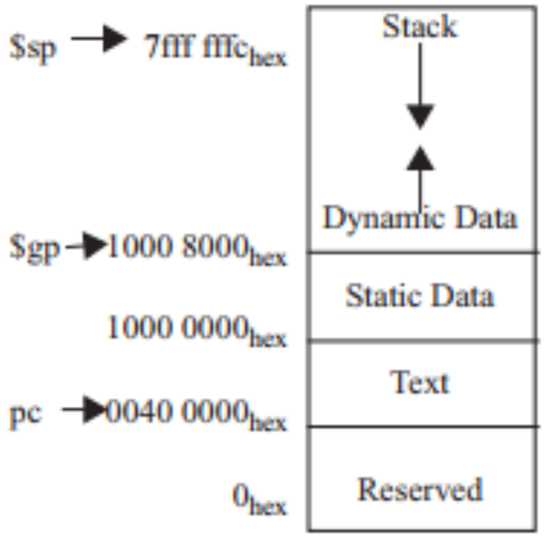
REGISTER NAME, NUMBER, USE, CALL CONVENTION

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	No

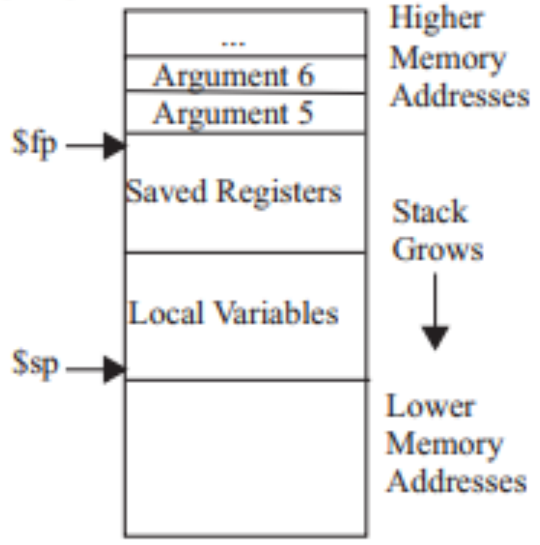
NOTE THE FOLLOWING:

1. Pseudo-Instructions
   - There are more of these, but in this class, you are ONLY allowed to use these + **la**
2. Registers and their numbers
3. Registers and their uses
4. Registers and their calling convention

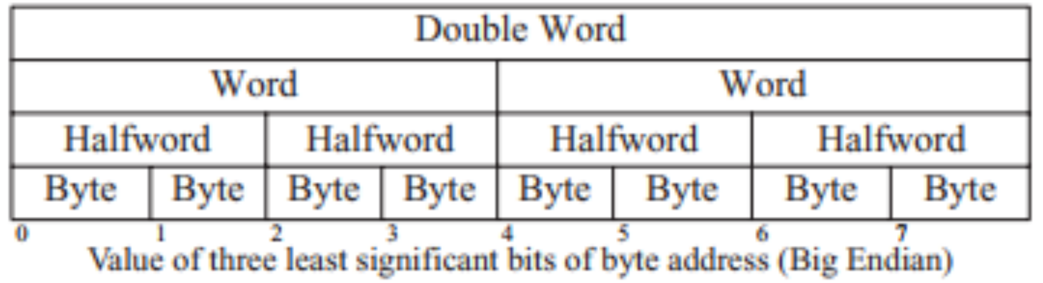
MEMORY ALLOCATION



STACK FRAME



DATA ALIGNMENT



SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	milli-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except µ is used for micro.

NOTE THE FOLLOWING:

1. This is only part of the 2nd page that you need to know

Bring Out Your MIPS Reference Cards!

Look for the following instructions:

- `nor`
- `addi`
- `beq`
- `move`

Tell me everything you can about them, based on what you see on the Ref Card!

Example 1

Syntax for "add"

```
add rd, rs, rt
```

destination, **source1**, **source2**

f = (g + h) – (i + j)

i.e. $s0 = ($s1 + $s2) – ($s3 + $s4)

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Example 2

*f = g * h - i*

$$\text{i.e. } \$s0 = \underbrace{\underbrace{(\$s1 * \$s2)}_{} - \$s3}_{}$$

```
mult $s1, $s2
mflo $t0
# mflo directs where the answer of the
mult should go
sub $s0, $t0, $s3
```

Recap: The **mult** instruction

- To multiply 2 integers together:

```
li $t0, 5              # t0 = 5
li $t1, 6              # t1 = 6
mult $t1, $t0          # multiply t0 * t1
mflo $t2               # t2 = t0 * t1
```

- **`mult`** cannot be used with an ‘immediate’ value
- Then we multiply our multiplier ($t0) with our multiplicand ($t1)
- And we put the result in the destination reg ($t2) using the **`mflo`** instruction

Memory Operations

- Main memory used for composite data
  - e.g.: Arrays, structures, dynamic data
  - In MIPS, use the **.data declaration** to initialize memory values (must be above **.text declaration**)

- Example:

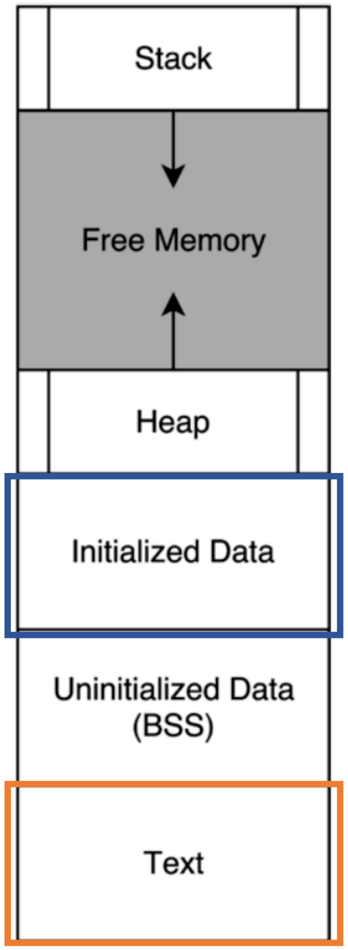
```
.data
var1: .word 42
.text
la $t0, var1        # t0 = &var1
lw $t1, 0($t0)      # t1 = *(&var1) = 42
```

Example
What does this do?

```
.data
name: .asciiz “Lisa speaks ”
rtn: .asciiz “ languages!\n”
age:  .word 7

.text
main:
      li $v0, 4
      la $a0, name    # la = load memory address
      syscall

      la $t2, age
      lw $a0, 0($t2)
      li $v0, 1
      syscall

      li $v0, 4
      la $a0, rtn
      syscall

      li $v0, 10
      syscall
```

What goes in here? →

What goes in here? →



.data Declaration Types *w/ Examples*

```
var1:    .byte 9          # declare a single byte with value 9
var2:    .half 63         # declare a 16-bit half-word w/ val. 63
var3:    .word 9433       # declare a 32-bit word w/ val. 9433
num1:    .float 3.14      # declare 32-bit floating point number
num2:    .double 6.28     # declare 64-bit floating pointer number
str1:    .ascii "Text"    # declare a string of chars
str3:    .asciiz "Text"   # declare a null-terminated string
str2:    .space 5         # reserve 5 bytes of space (useful for arrays)
```

These are now reserved in memory and we can call them up by loading their memory address into the appropriate registers.

YOUR TO-DOs for the Week

- Do your reading for next class (see syllabus)

- Work on Assignment #1 for lab (***lab01***)
  - Meet up in the lab this Friday
  - Do the lab assignment
  - You have to submit it as a **PDF** using ***Gradescope***
  - Due on **Wednesday, 1/15, by 11:59:59 PM**

</LECTURE>