ASSEMBLY LANGUAGE

A Continuation

Assembly Language

THE ARITHMETIC AND LOGIC UNIT (ALU)

a digital logic circuit designed to perform binary integer arithmetic operations, as well as binary logical operations such as "AND", "OR", "NOR", and "Exclusive OR".

depends upon the operation code in the instruction

PROGRAM COUNTER (PC)

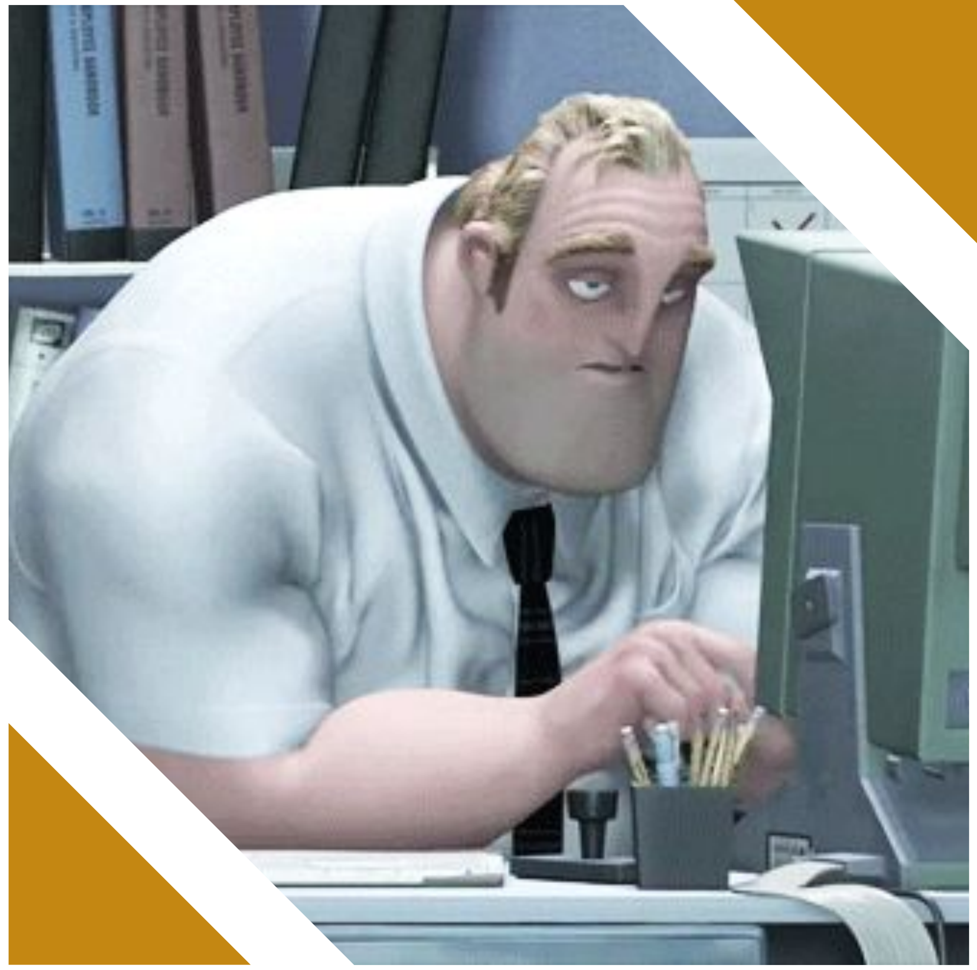
Is it what it really is?

is a register that is initialized by the operating system to the address of the first instruction of the program in memory

After an instruction has been fetched from memory and loaded into the IR, the PC is incremented so that the CPU will have the address of the next sequential instruction for the next instruction fetch.

The name program counter is misleading. But the name has caught on, and there is no way to change to tradition.

better name would be program pointer



MEMORY

Millions Instructions Per Second (MIPS)

a large array of locations where binary information is stored and from which binary information can be fetched, one "word" at a time.

term word refers to a 32-bit quantity. Each location in memory has a 32-bit address.

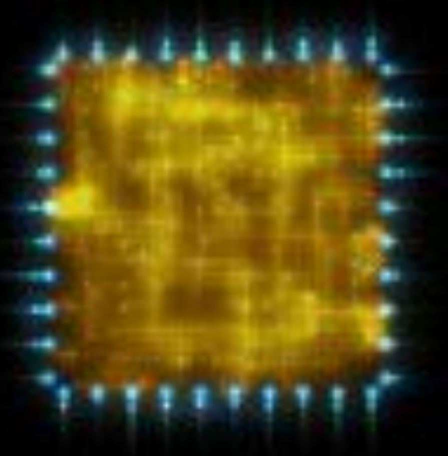
In the MIPS architecture, memory addresses range from 0 to 4,294,967,295 (232 - 1) . The MIPS architecture uses the term half-word to refer to a 16-bit value, and the term byte to refer to an 8-bit value.

In Binary we trust.

THE INSTRUCTION REGISTER (IR)

Of Assembly Language

32-bit register that holds a copy of the most recently fetched instruction.



ruction formats:

gister format

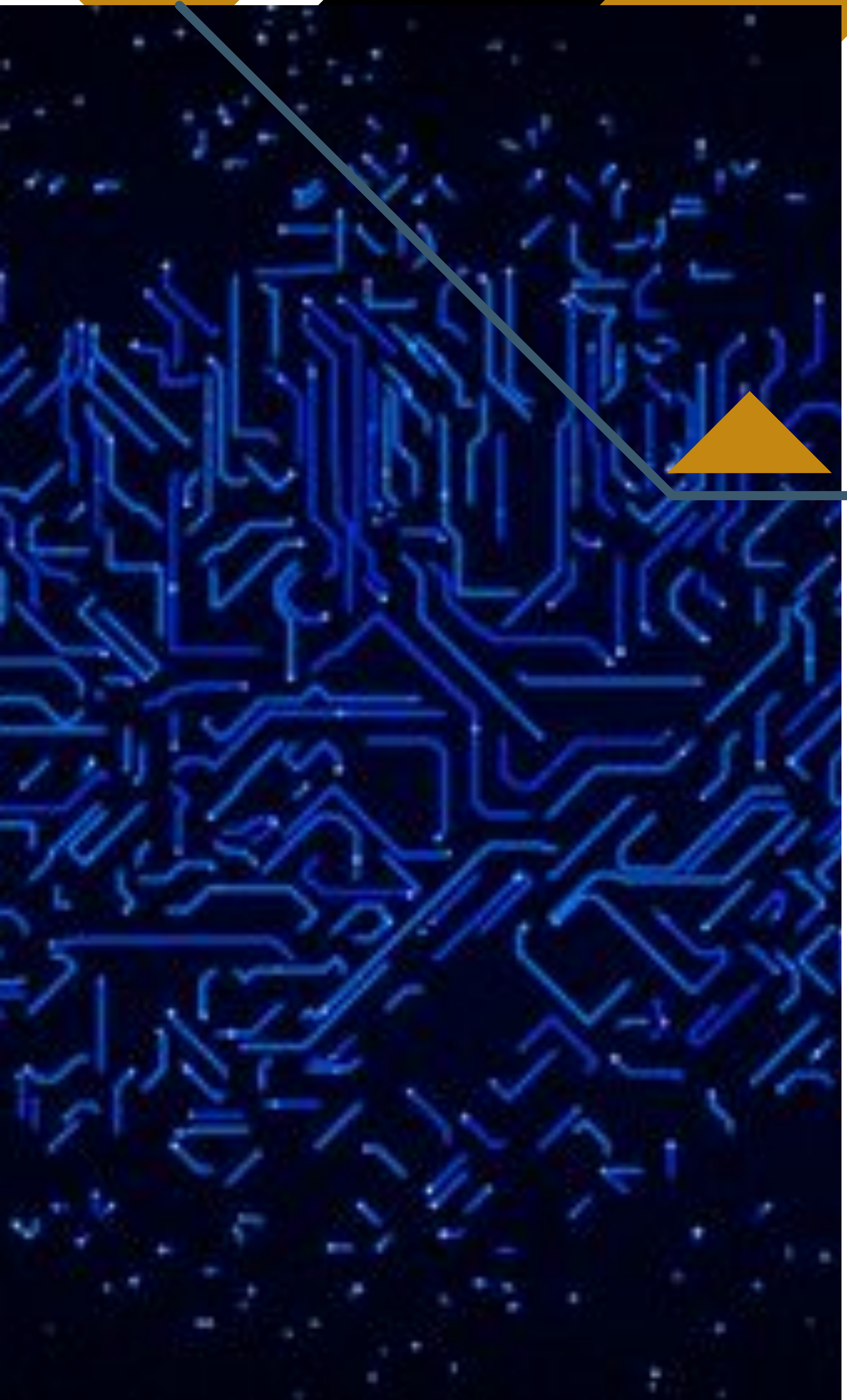
Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	ffffff

Immediate format

Op-Code	Rs	Rt	Immediate
ffffff	sssss	ttttt	iiiiiiiiiiiiiiii

mp Format

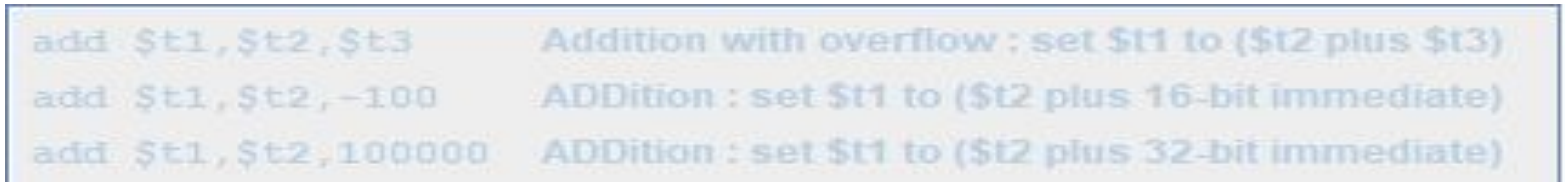
Op-Code	Target
00001f	tttttttttttttttttttttttttt



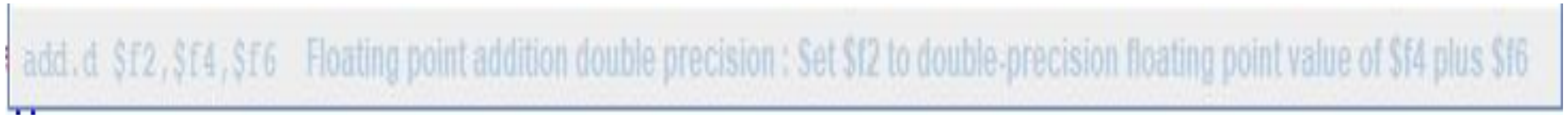
INSTRUCTION SET

MIPS Instruction Set

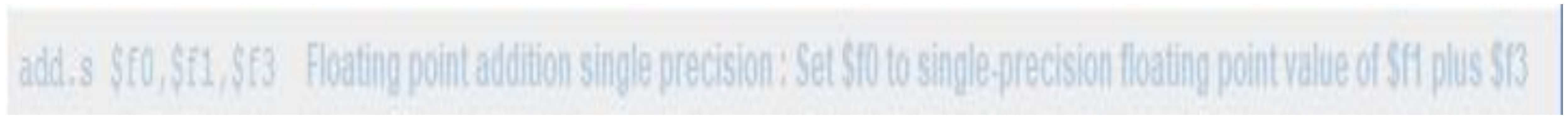
a group of commands for a central processing unit (CPU) in machine language. The term can refer to all possible instructions for a CPU or a subset of instructions to enhance its performance in certain situations.



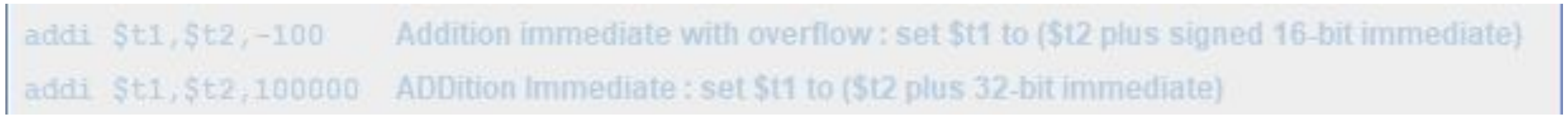
```
add  $t1,$t2,$t3       Addition with overflow : set $t1 to ($t2 plus $t3)
add  $t1,$t2,-100      ADDition : set $t1 to ($t2 plus 16-bit immediate)
add  $t1,$t2,100000    ADDition : set $t1 to ($t2 plus 32-bit immediate)
```



```
add.d $f2,$f4,$f6  Floating point addition double precision : Set $f2 to double-precision floating point value of $f4 plus $f6
```

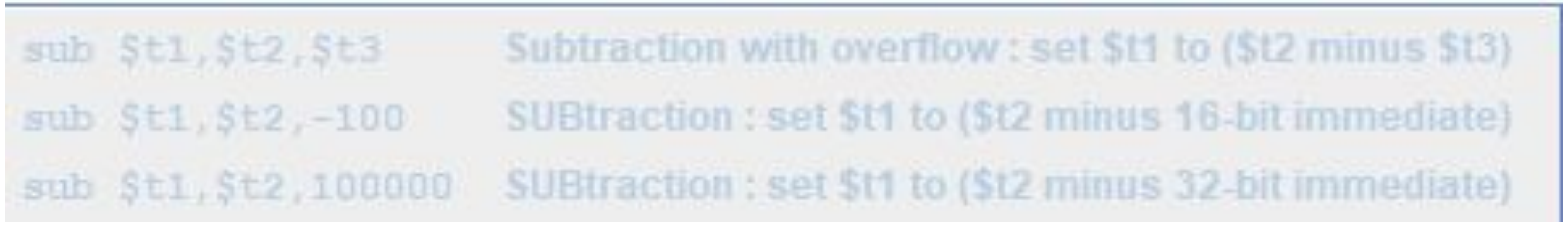


```
add.s $f0,$f1,$f3  Floating point addition single precision : Set $f0 to single-precision floating point value of $f1 plus $f3
```

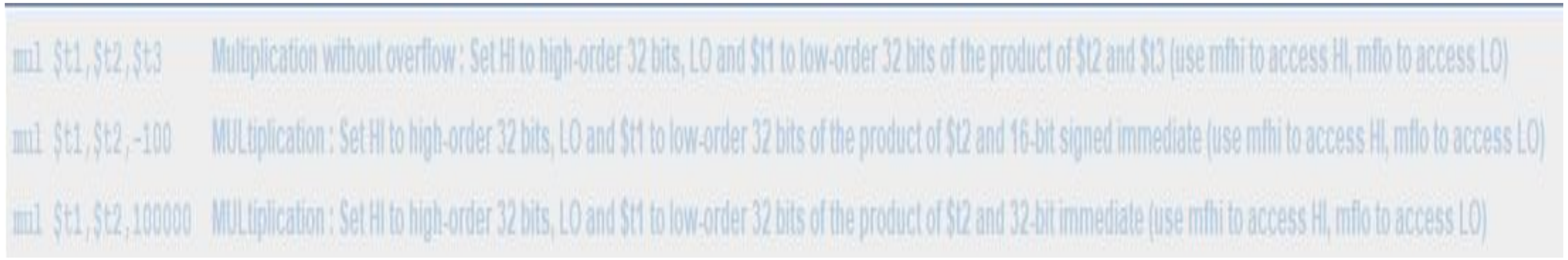


```
addi  $t1,$t2,-100      Addition immediate with overflow : set $t1 to ($t2 plus signed 16-bit immediate)
addi  $t1,$t2,100000    ADDition Immediate : set $t1 to ($t2 plus 32-bit immediate)
```

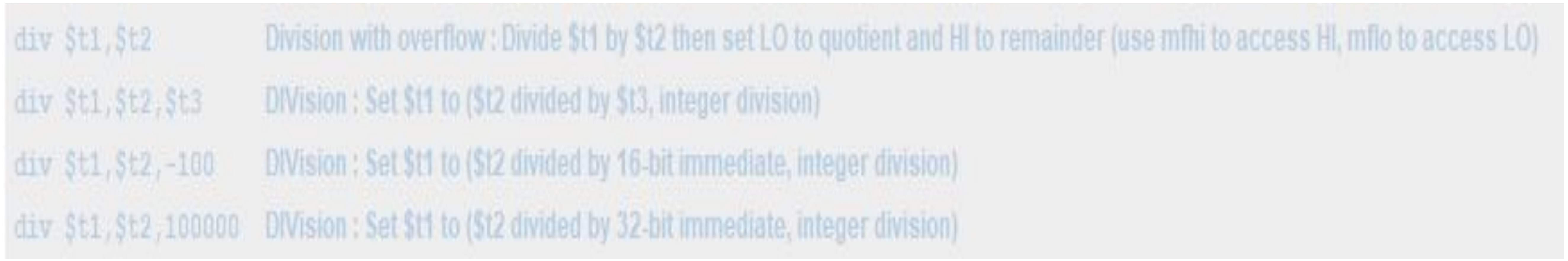
INSTRUCTION SET



sub $t1,$t2,$t3	Subtraction with overflow : set $t1 to ($t2 minus $t3)
sub $t1,$t2,-100	SUBtraction : set $t1 to ($t2 minus 16-bit immediate)
sub $t1,$t2,100000	SUBtraction : set $t1 to ($t2 minus 32-bit immediate)



mul $t1,$t2,$t3	Multiplication without overflow : Set HI to high-order 32 bits, LO and $t1 to low-order 32 bits of the product of $t2 and $t3 (use mfhi to access HI, mflo to access LO)
mul $t1,$t2,-100	MULtiplication : Set HI to high-order 32 bits, LO and $t1 to low-order 32 bits of the product of $t2 and 16-bit signed immediate (use mfhi to access HI, mflo to access LO)
mul $t1,$t2,100000	MULtiplication : Set HI to high-order 32 bits, LO and $t1 to low-order 32 bits of the product of $t2 and 32-bit immediate (use mfhi to access HI, mflo to access LO)



div $t1,$t2	Division with overflow : Divide $t1 by $t2 then set LO to quotient and HI to remainder (use mfhi to access HI, mflo to access LO)
div $t1,$t2,$t3	DIVision : Set $t1 to ($t2 divided by $t3, integer division)
div $t1,$t2,-100	DIVision : Set $t1 to ($t2 divided by 16-bit immediate, integer division)
div $t1,$t2,100000	DIVision : Set $t1 to ($t2 divided by 32-bit immediate, integer division)

Addressing Modes

Load/Store architecture

Systemic approach on every programming languages.

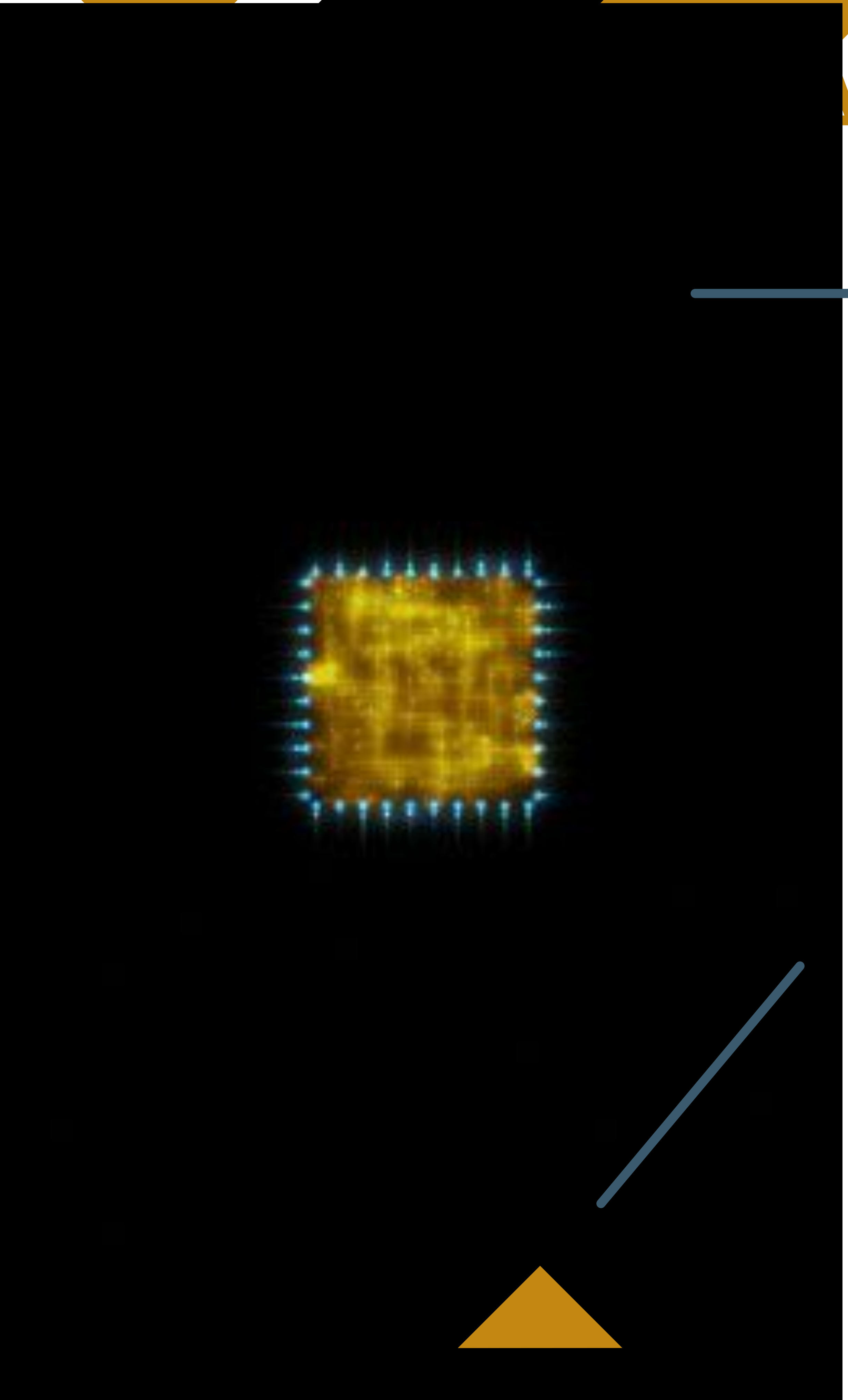
the only instructions that access main memory are the **load and store instructions**. Only one **addressing mode** is implemented in the hardware. The addressing mode is referred to as base address plus displacement. A **load instruction** accesses a value from memory and places a copy of the value found in memory in the register file.

Load Address = la
Get the instruction from the .data
Or memory

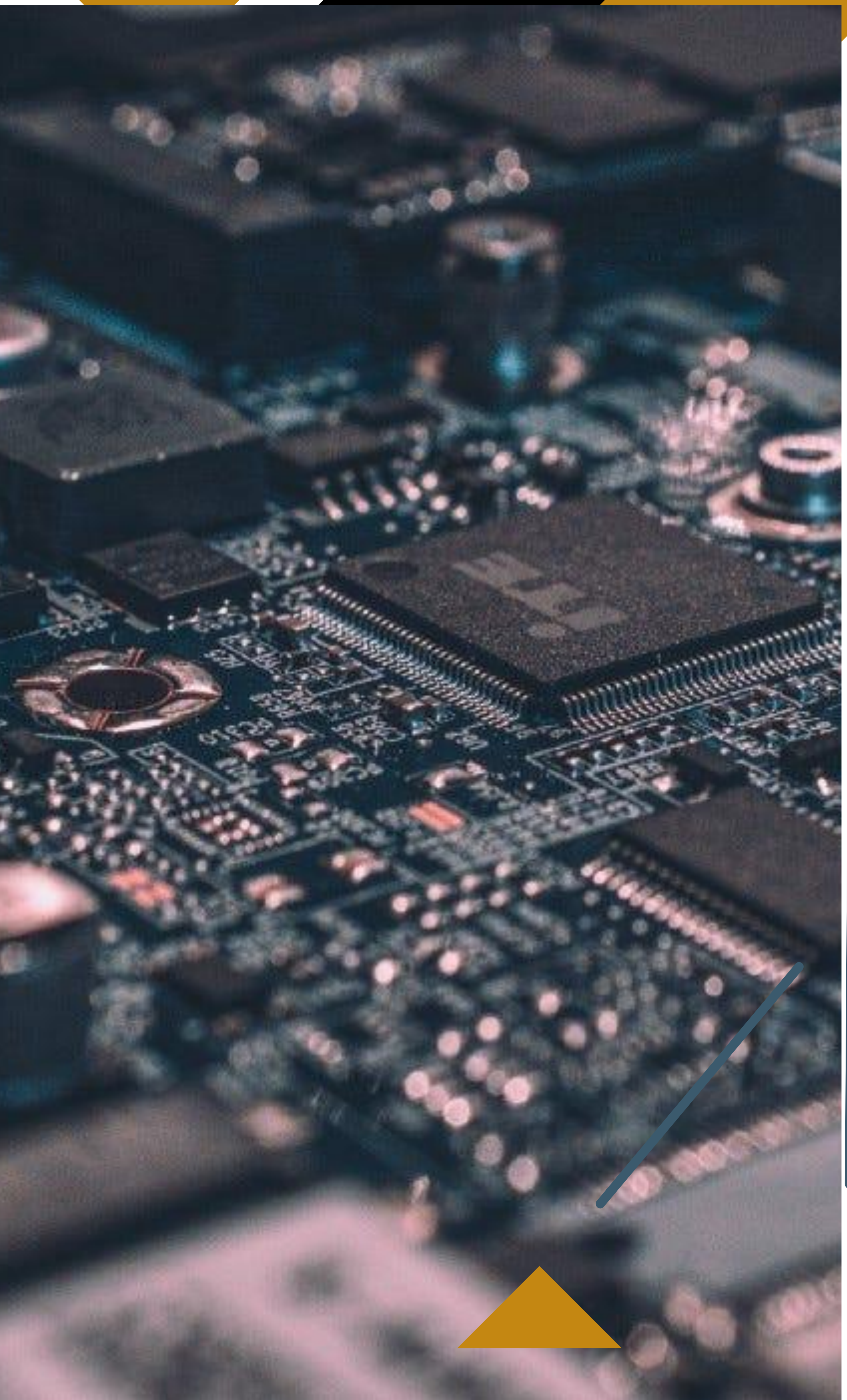
$a0 - $a3,
$s0 - $s7

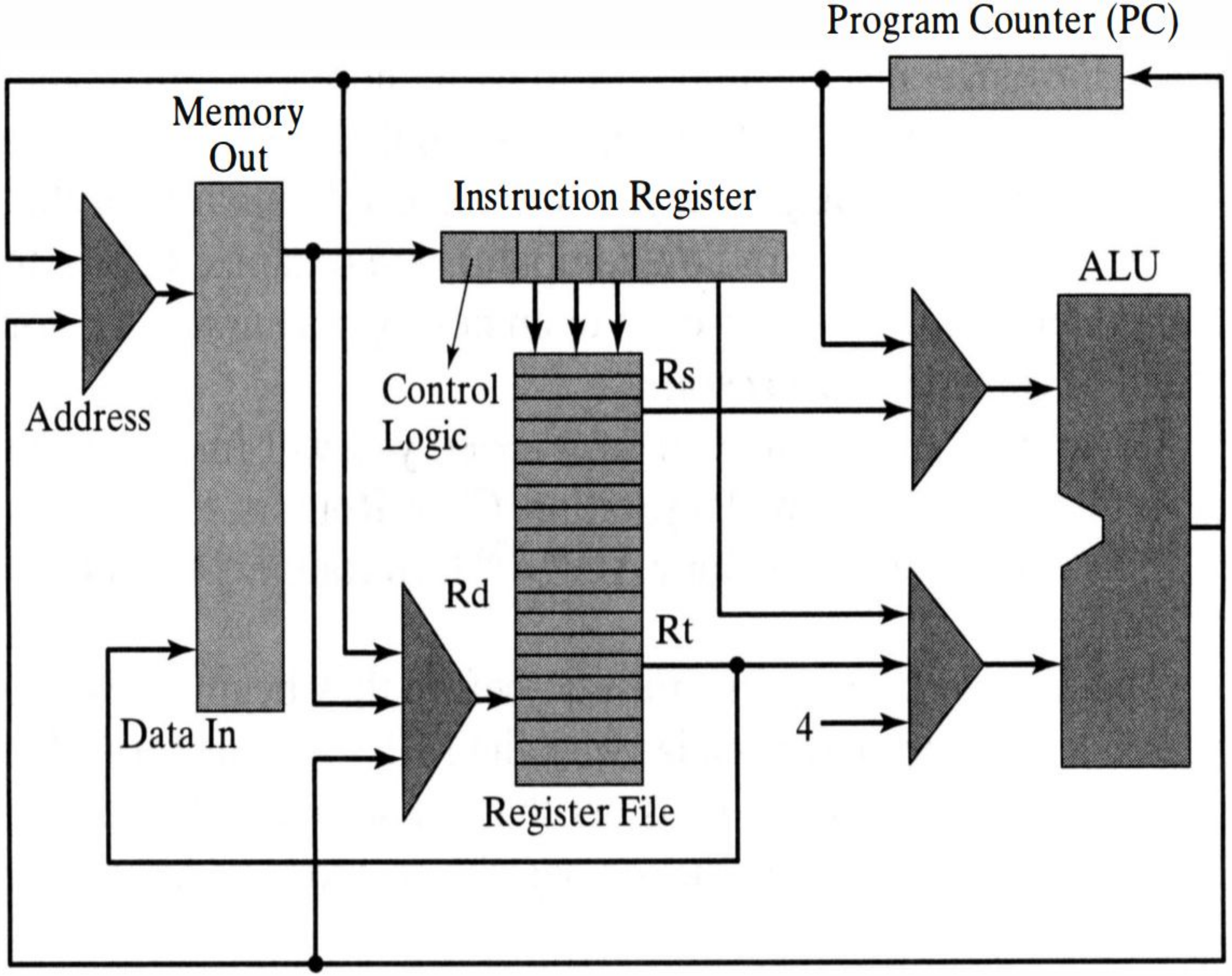
ssembly Programming

How does your data travel?

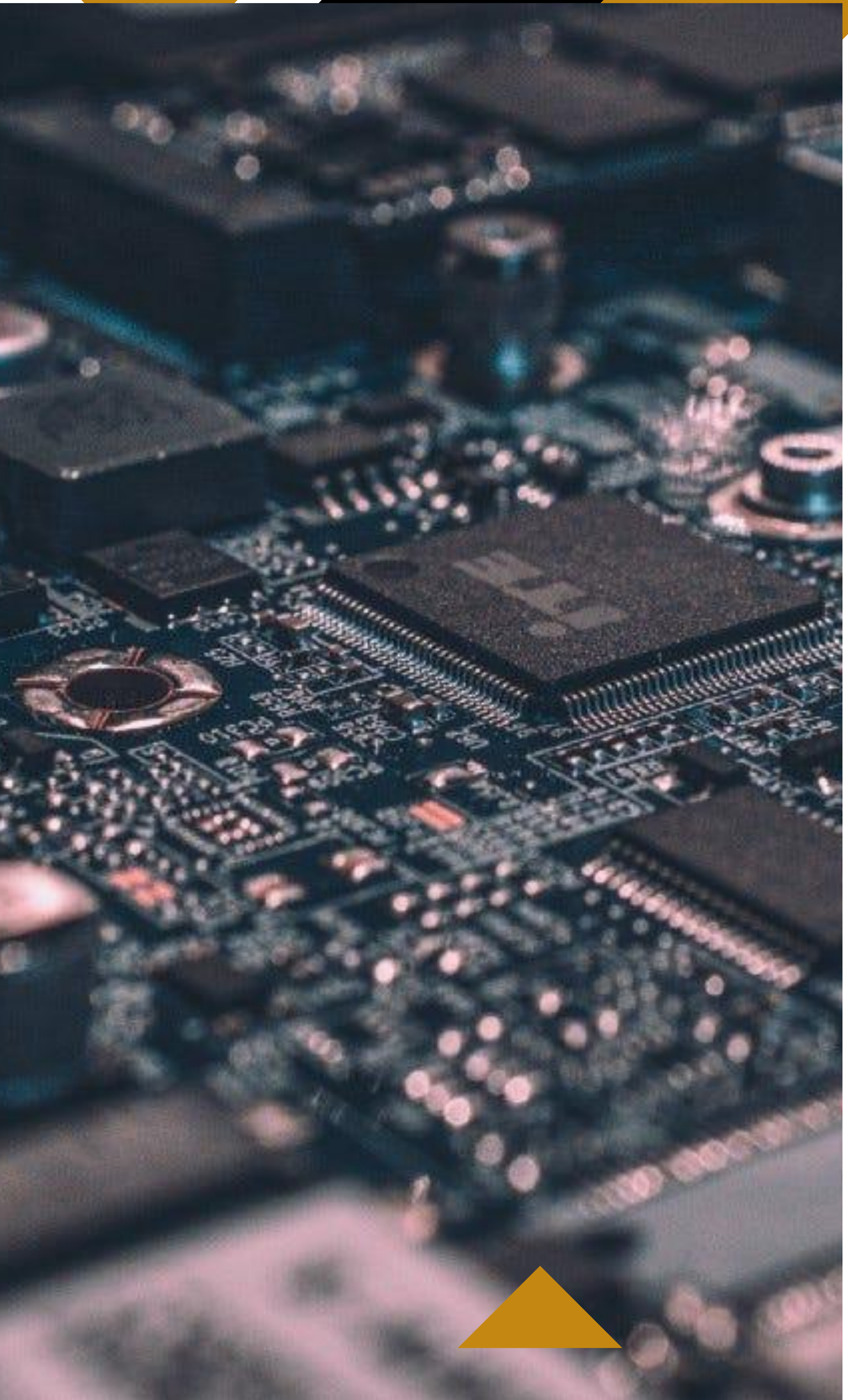


1**. Instruction Fetch Phase**. An instruction is fetched from memory at the location specified by the Pc. The instruction is loaded into the IR. PC is incremented by four.
2. **Operand Fetch Phase**. Two 5-bit codes, Rs and Rt, within the instruction specify which register file locations are accessed to obtain two 32-bit source operands. Decode the Op-Code.
3. **Execute Phase**. The two 32-bit source operands are routed to the ALU inputs where some operation is performed depending upon the Op-Code in the instruction.
4. **Write Back Phase**. The result of the operation is placed into the register file at a location specified by the 5-bit Rd code in the IR. Go to step 1.

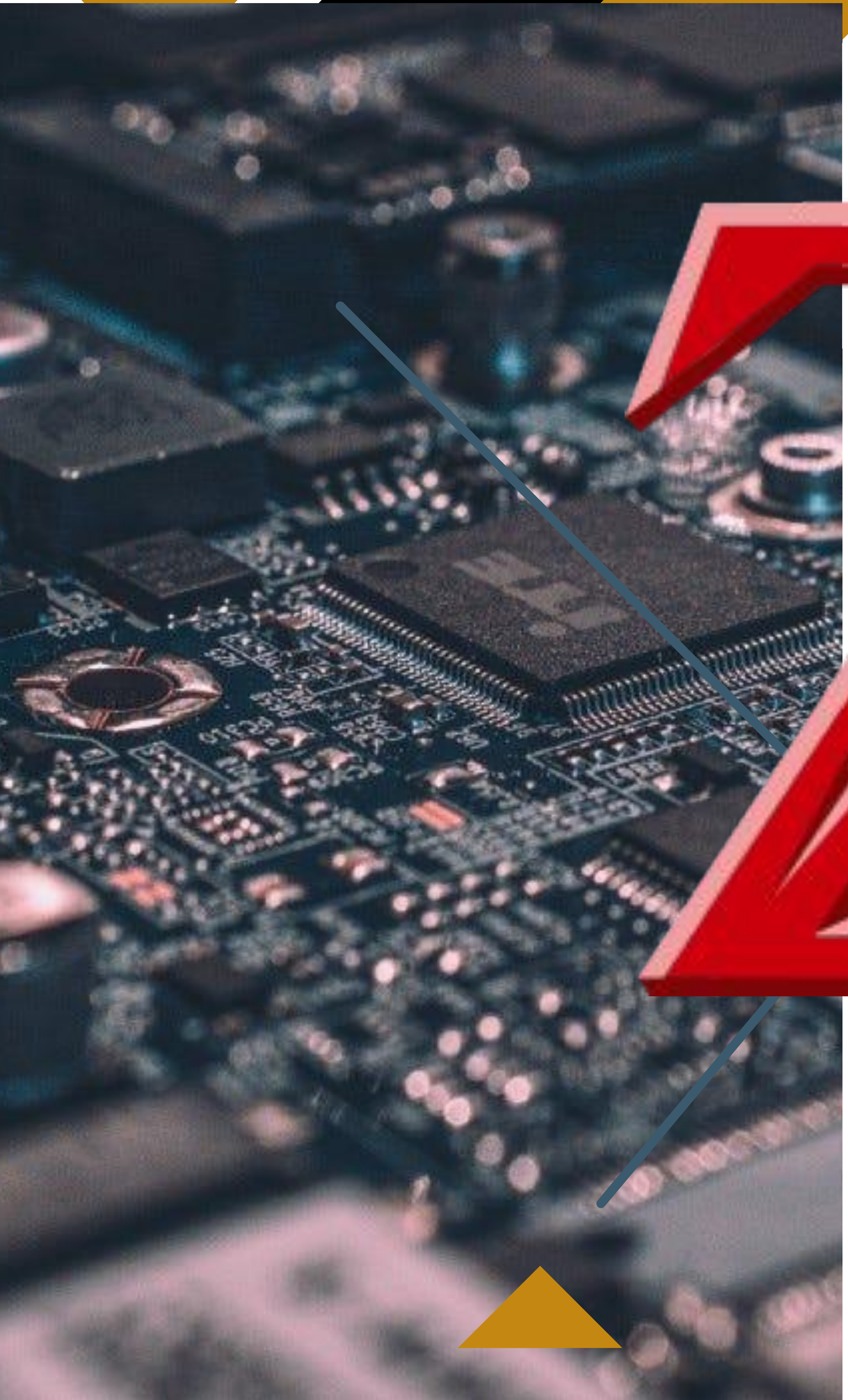




- control unit
- register file
- arithmetic and logic unit (ALU)
- program counter (PC)
- memory
- instruction register (IR)



Games that Assembly Language created



THE LEGEND OF
ZELDA
MICROPROSE

Assembly Programming

Assembly language is low-level code that relies on a strong relationship between the instructions input using the coding language and how a machine interprets the code instructions. Code is converted into executable actions using an assembler that converts input into recognizable instructions for the machine. Though prevalent in the early days of computing, many larger systems use higher-level languages.

Bottom Line