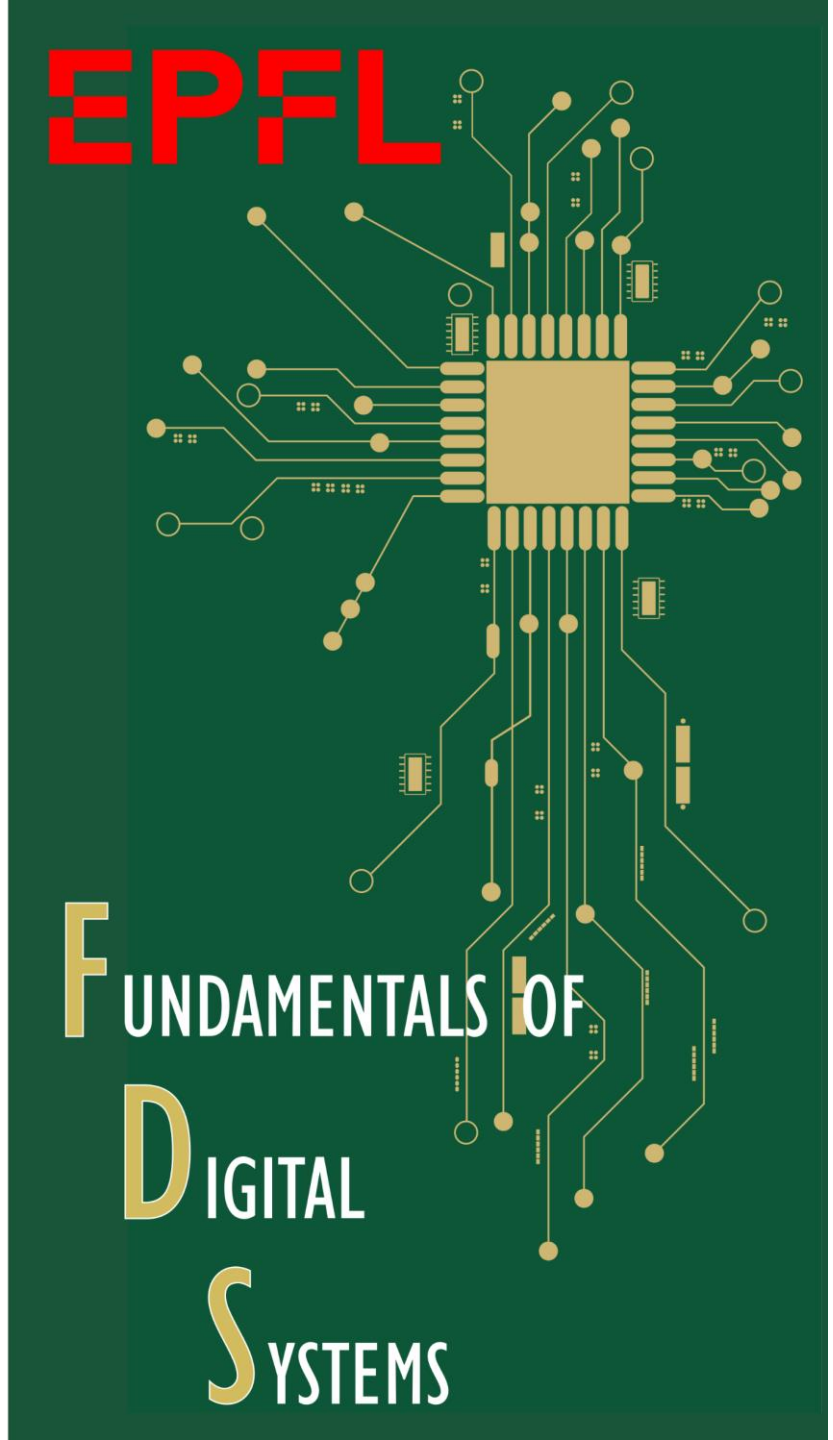


Computer Architecture

RISC-V Assembler Directives

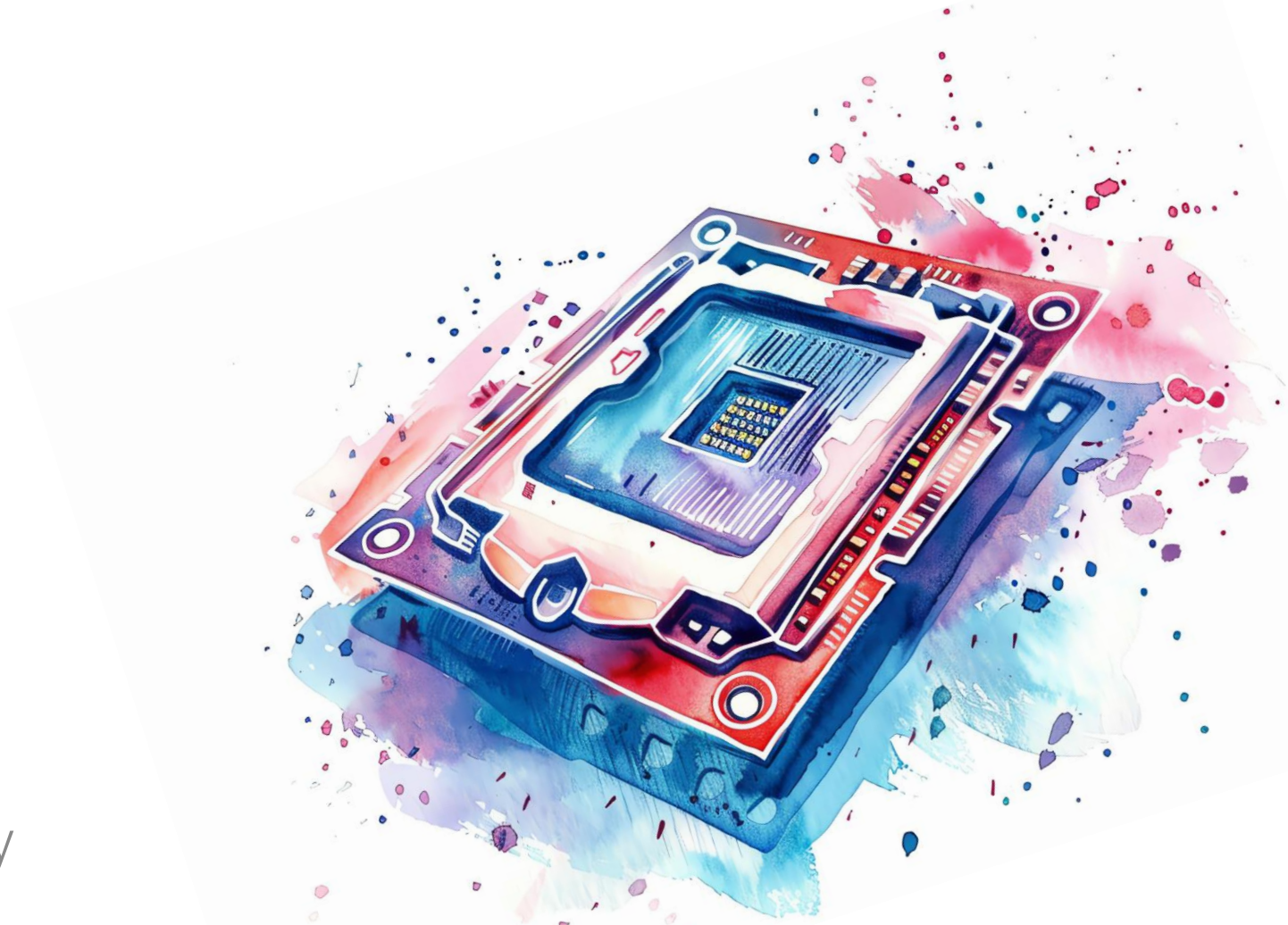
CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025

Previously on FDS

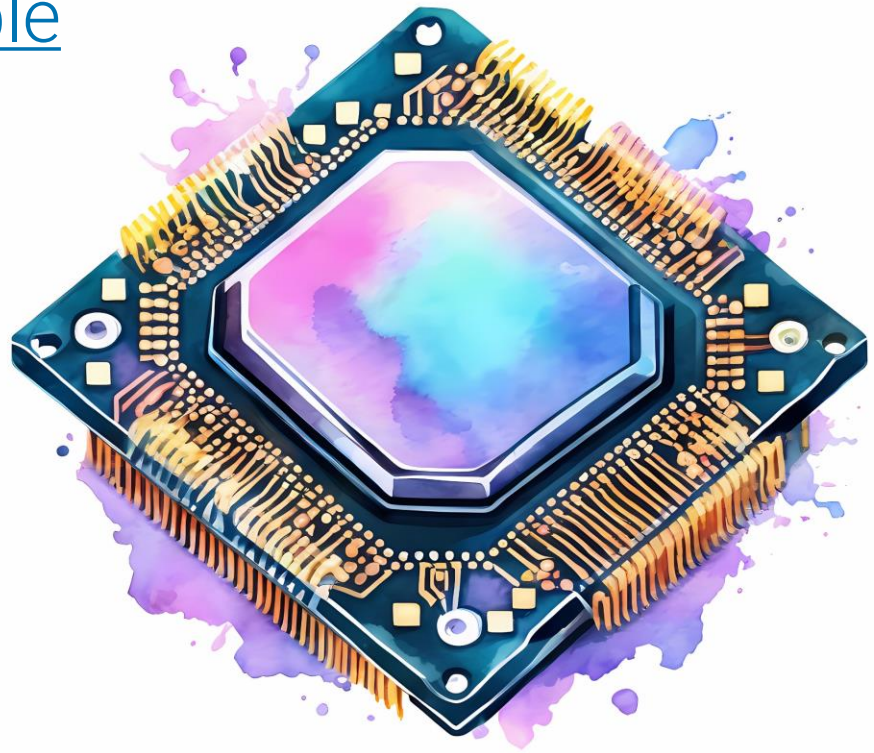
- **Load** (reading) from memory
- **Store** (writing) in memory
- Conditional branches, unconditional jumps



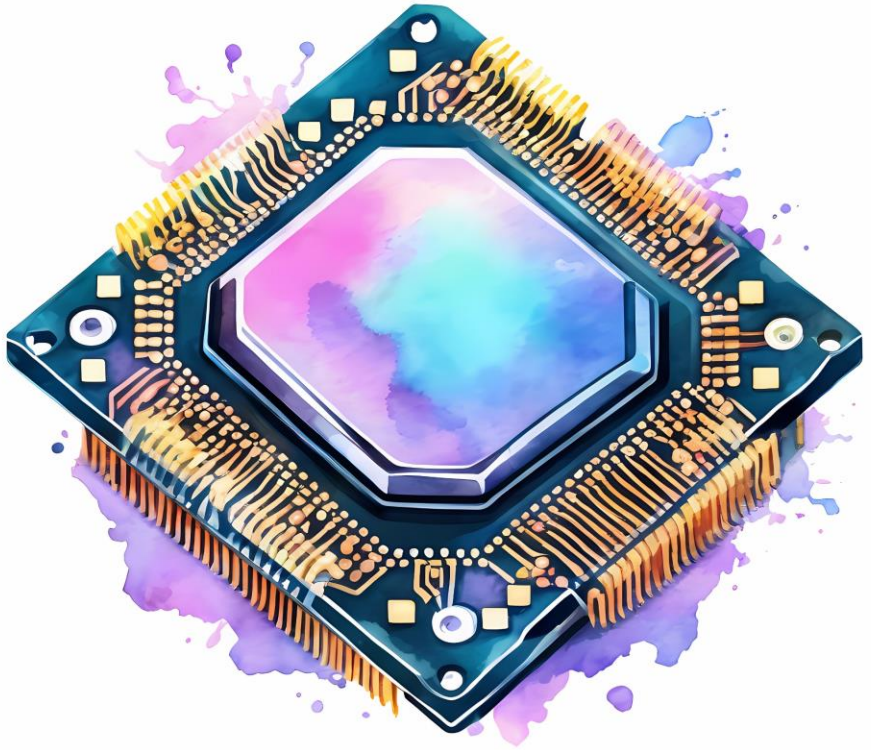
© Supranee / Adobe Stock

Quick Outline

- From Assembly Source Code to Executable
- Assembler Directives
  - Example
- Pseudoinstructions
  - `li`: Load immediate
  - `la`: Load address
- Do-while loop
  - Example
- If-Then-Else
  - Example



Assembler Directives



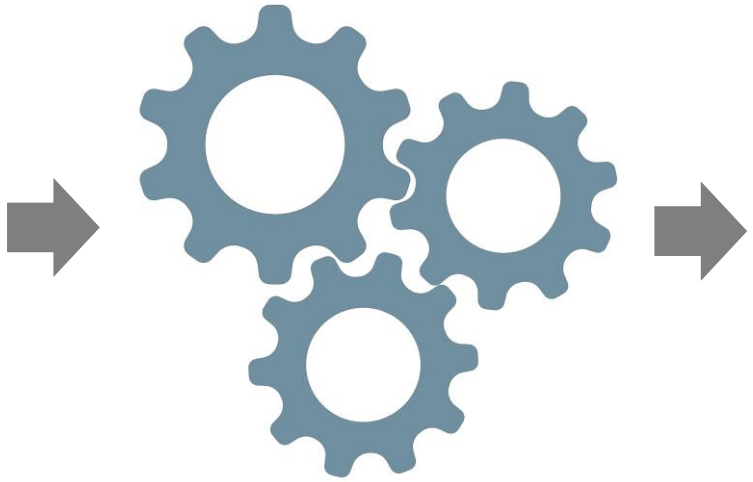
© Anastasi17 / Adobe Stock

From Assembly Source Code To Executable

Compilation, Assembly Process

```
li    t0, 7
li    t1, 1
sll   t2, t0, t1
sll   t3, t2, t1
sll   t3, t3, t1
add   t3, t3, t2
nop
```

Source code in RISC-V assembly



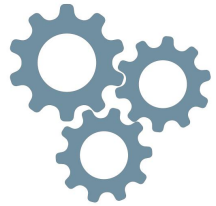
Assembler

*Responsible for the **translation** from source code to the executable in binary; the process is also known as **compilation***

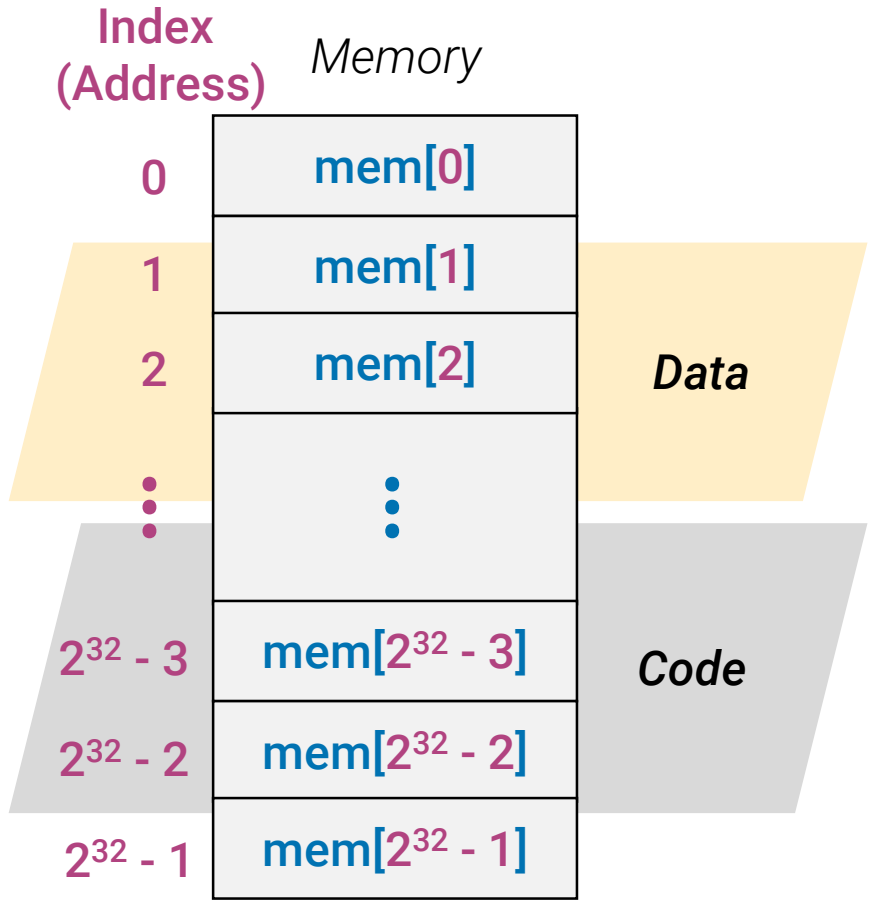
Address	+0	+1	+2	+3
0x00000018	00010011	00000000	00000000	00000000
0x00000014	00110011	00001110	01111110	00000000
0x00000010	00110011	00011110	01101110	00000000
0x0000000C	00110011	10011110	01100011	00000000
0x00000008	10110011	10010011	01100010	00000000
0x00000004	00010011	00000011	00010000	00000000
0x00000000	10010011	00000010	01110000	00000000
----------	--	--	--	--
----------	--	--	--	--

The executable (the RISC-V program) in binary format, in memory

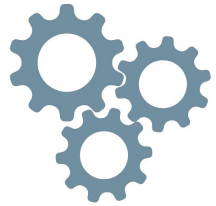
Von-Neuman Architecture



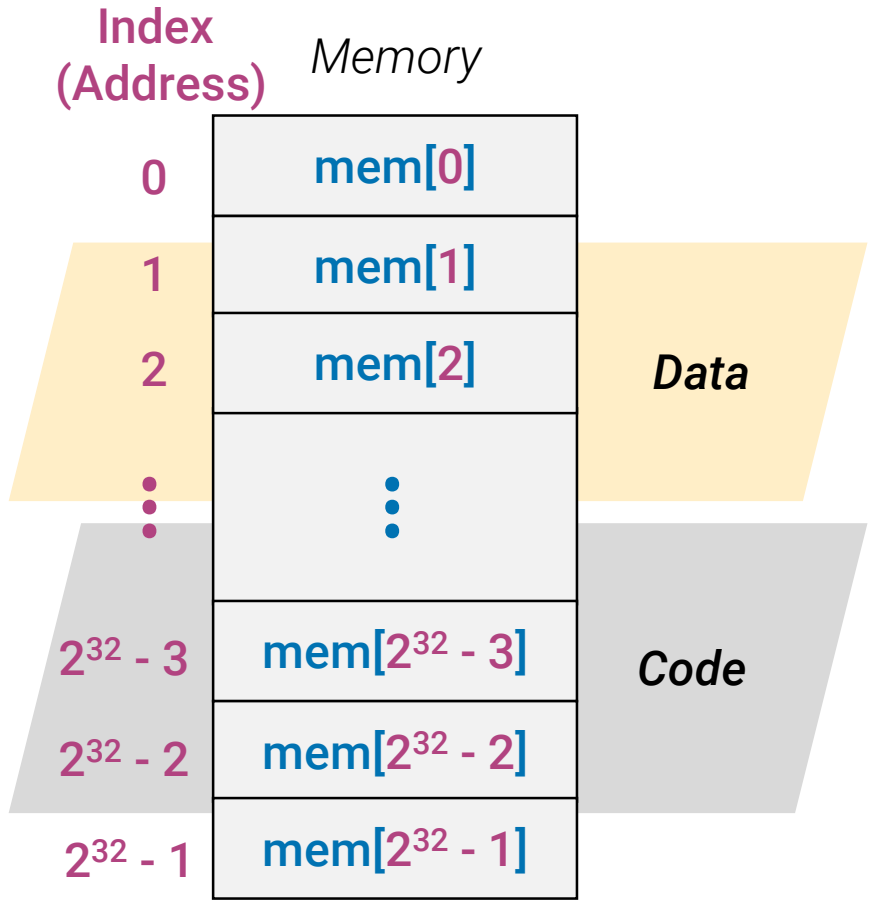
- *Recall:* Unified memory
  - Instructions and data reside in the same memory
- Assembly source code often contains more than one **section**
  1. Code (program instructions)
  2. Data (initialization, results)



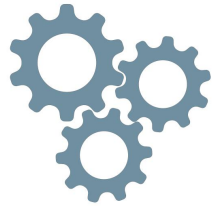
Von-Neuman Architecture



- The compilation process is responsible for creating separate memory regions for **code** and **data**
- When writing a program in assembly, one can mix code and data sections as we like, identifying them with **special labels (assembler directives)**



Assembler Directives…



- … are commands that are part of the assembler syntax, but unrelated to the CPU ISA. They supply data to the program and control the assembly process.

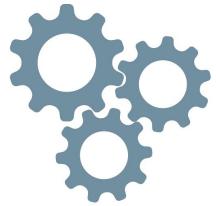
1. Assembler Directives

Directive	Effect
`.text`	Switch to the code segment and place what follows there
`.data`	Switch to the data segment and place what follows there
`.asciiz "Hello, World!"`	Place at current location an ASCII string followed by a null-terminator
`.byte 0xC, 0xA, 0xF`	Place at current location value(s) as byte(s)
`.word 0xCAFE`	Place at current location value(s) as 32-bit word(s)
`.equ coffee, 0xCAFE`	Define a symbol as a constant

Source: CS-173 RV32I Reference Card

Assembler Directives

Sections

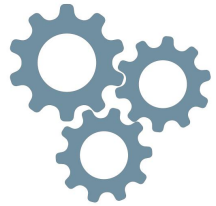


Directive	Effect
`.text`	**Read-only section containing program (executable) code in assembly** • The **.text** sets .text as the current section. • The lines that follow this directive will be assembled into the .text section, which contains executable code. • The .text section is the default section. Therefore, the assembler assembles code into the .text section unless you use, for example, the .data directive to specify data section.

Notice the period

Assembler Directives

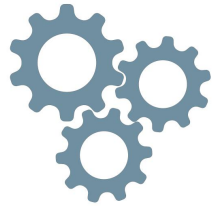
Sections



Directive	Effect
`.data`	**Read-write section containing program variables** • The **.data** directive sets .data as the current section. • The lines that follow will be assembled into the .data section. • The .data section is commonly used to contain arrays of data or preinitialized variables.

Assembler Directives

Initialize Values (Data, Memory)

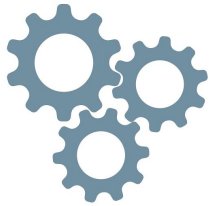


Directive	Effect
`.byte value`$_1$`[,…, value`$_n$`]`	**Initializes one or more successive bytes at the current location**

- A value can be an expression that the assembler evaluates and treats it as an 8-bit signed number, or a character string enclosed in double quotes.
- In the case of a character string, each character in a string is a separate value, and values are stored in consecutive bytes. With little-endian ordering, the first byte occupies the eight least significant bits of a full 32-bit word. The second byte occupies bits eight through 15, while the third byte occupies bits 16 through 23.
- The assembler truncates values greater than eight bits (stores only the least significant eight bits).

Assembler Directives

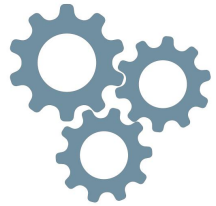
Initialize Values (Data, Memory)



Directive	Effect
`.asciiz`	**Places at the current location an ASCII string followed by a null-terminator (byte 0x00; https://en.wikipedia.org/wiki/Null-terminated_string)** • Character string must be enclosed in double quotes.

Assembler Directives

Initialize Values (Data, Memory), Contd.

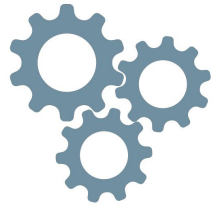


Directive	Effect

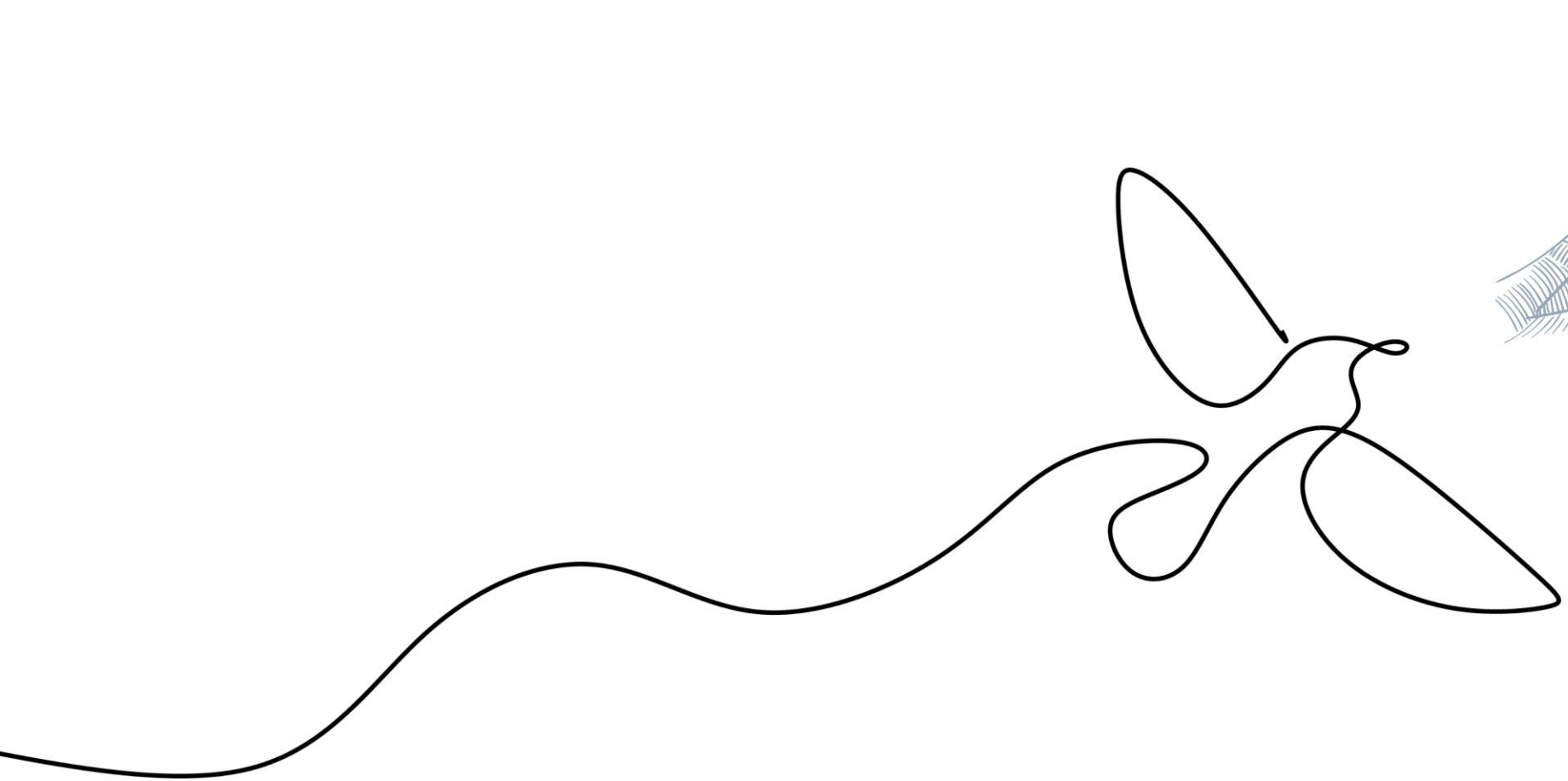
`.word value`$_1$`[,…, value`$_n$`]`
 	
`.uword value`$_1$`[,…, value`$_n$`]`
	Initializes one or more successive words at the current section
	
	• Each value is placed in a 32-bit word by itself and is aligned on a word boundary.
	• A value can be either an expression that the assembler evaluates and treats as a 32-bit signed (.word) or unsigned (.uword) number, or a character string enclosed in double quotes.
	• In the case of a character string, each character in a string represents a separate value and is stored alone in the least significant eight bits of a 32-bit field, which is padded with zeros.

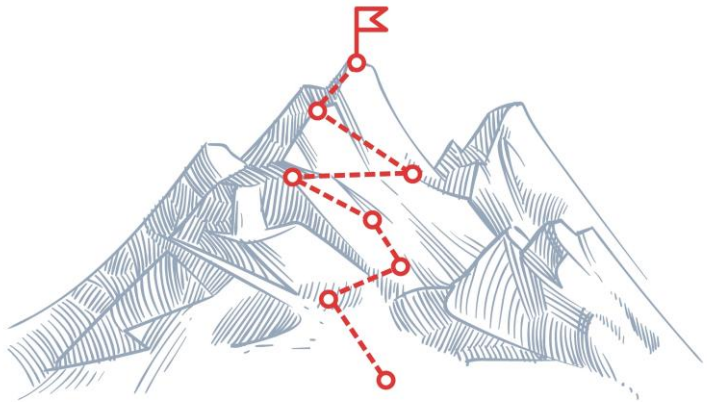
Assembler Directives

Defining Symbols



Directive	Effect
`.equ name, value`	**Symbol definition** • Assigns **value** to a symbol **name.**





Assembler Directives

Example

- **Prepare for** writing an assembly code that reads and analyzes `n` elements of an array of bytes, and saves the result in memory
  - Part 1: Initialization of variables and data
    - Define `n` as a symbol, and set it to $(10)_{10}$
    - Initialize the **array** in memory with 16 randomly chosen hexadecimal numbers: 3F A7 5C 91 DE 02 6B B8 4E 13 C0 8D F4 29 7E 10
    - Initialize the 32-bit **result** in memory to zero
  - Part 2: Initialization of registers
    - Copy `n` to register `t0`
    - Copy the **address** of the **array** to register `t1`
    - Copy the **address** of the **result** to register `t2`

Assembler Directives

Solution, Assembly Code

```
.equ n, 10                                  # constant n = 10
.data                                                          Data
my_array_of_bytes:  .byte   0x3F, 0xA7, 0x5C, 0x91 # array of bytes
                    .byte   0xDE, 0x02, 0x6B, 0xB8 # broken over several
                    .byte   0x4E, 0x13, 0xC0, 0x8D # lines of code
                    .byte   0xF4, 0x29, 0x7E, 0x10 # for readability
my_result:          .word   0
```

- Part 1: Initialization of constants and data
  - Define **n** as a constant, and set it to $(10)_{10}$
  - Initialize the **array of bytes** in memory with 16 randomly chosen hexadecimal numbers: 3F A7 5C 91 DE 02 6B B8 4E 13 C0 8D F4 29 7E 10
  - Initialize the 32-bit **result** in memory to zero

Assembler Directives

Solution, Assembly Code

- Part 2: Initialization of registers
  - Copy `n` to register `t0`
  - Copy the **address** of the **array** to register `t1`
  - Copy the **address** of the **result** to register `t2`

Code

```
.text
li t0, n                          # load immediate, pseudoinstr.
la t1, my_array_of_bytes          # load address, pseudoinstr.
la t2, my_result                  # load address, pseudoinstr.
```

Assembler Directives

Solution, Putting it All Together

EXAMPLES

```
.equ n, 10                                          # constant n = 10
.data                                                                    Data
my_array_of_bytes:  .byte   0x3F, 0xA7, 0x5C, 0x91 # array of bytes
                    .byte   0xDE, 0x02, 0x6B, 0xB8 # broken over several
                    .byte   0x4E, 0x13, 0xC0, 0x8D # lines of code
                    .byte   0xF4, 0x29, 0x7E, 0x10 # for readability
my_result:          .word   0
.text                                                                    Code
li t0, n                                            # load immediate, pseudoinstr.
la t1, my_array_of_bytes                            # load address, pseudoinstr.
la t2, my_result                                    # load address, pseudoinstr.
```

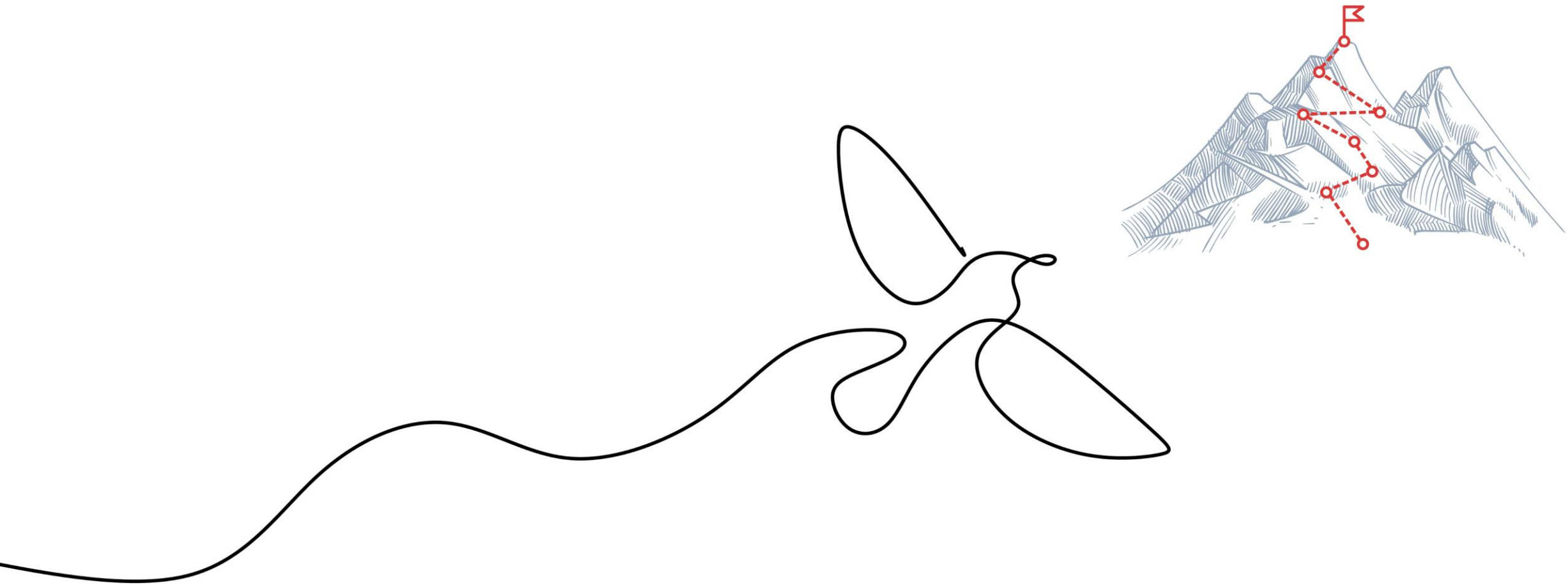
Assembler Directives

Solution, Output from the Venus Simulator

Address	+0	+1	+2	+3
…	…	…	…	…
0x10000018	00	00	00	00
0x10000014	00	00	00	00
0x10000010	00	00	00	00
0x1000000C	F4	29	7E	10
0x10000008	4E	13	C0	8D
0x10000004	DE	02	6B	B8
0x10000000	3F	A7	5C	91

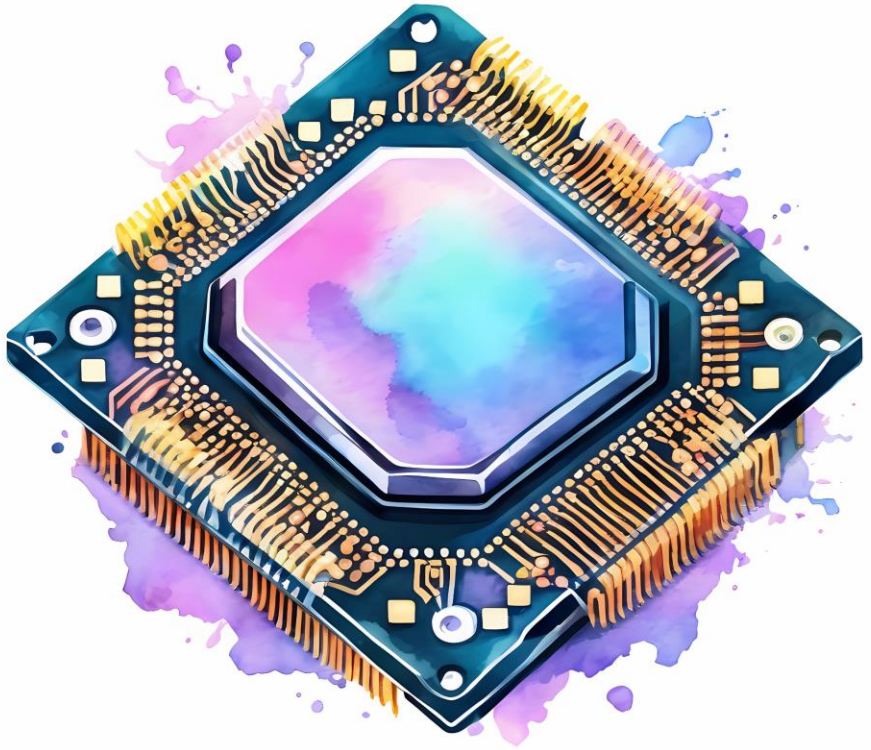
- Two-dimensional memory view in the Venus Simulator (left)
- Data segment starts at address **`0x1000 0000`** in memory

- Registers
  - `x05 (t0) = 0x0000 000A`
  - `x06 (t1) = 0x1000 0000`
  - `x07 (t2) = 0x1000 0010`



Pseudoinstructions

- `li`
- `la`



© Anastasi17 / Adobe Stock

Load Immediate and Load Address

`li, la`

Instruction or Pseudoinstruction		Pseudocode	Type	funct7	funct3	opcode
Move				or Translation		
`nop`		*Nothing*		`addi zero, zero, 0`		
`mv`	`rd, rs1`	rd ← rs1		`addi rd, rs1, 0`		
`li`	`rd, imm`	rd ← imm[†]		*Various translations*		
`la`	`rd, imm`	rd ← address of imm[§]		*Various translations*		
`lui`	`rd, imm`	rd ← imm ≪ 12	U			0x37

Source: CS-173 RV32I Reference Card

- `li` copies the immediate to a register
- `la` copies the address of a label to a register

`li` Pseudoinstruction

Usage

- *Recall:* `li` copies the immediate to a register
- Use `li` when **`imm`** is a constant or a symbol defined with **`.equ`**

Example: `li`

- *Recall:* `li` copies the immediate to a register
- *Recall:* Use `li` when `imm` is a constant or a symbol
- Example:

```
.equ my_constant, 0x12345678  # symbol my_constant = 0x12345678
.text
li t0, my_constant            # t0 = my_constant
li t1, 0x123                  # t1 = 0x123
```

`li` Pseudoinstruction

Various Translations, Partial View

- Small immediate (imm "fits" in 12-bit signed imm space):
  - `li t0, 0x123`
  - Translation: `addi t0, zero, 0x123`

- Medium immediate:
  - `li t0, 0x12345678`
  - Translation: ?

Recall: Integer Register-Immediate Operations

U-type, Instruction LUI

- **LUI: L**oad **u**pper **i**mmediate; used to build 32-bit constants
  - LUI places the 20-bit immediate value in the top 20 bits of the destination register RF[rd], filling the lowest 12 bits with zeros

`li` Pseudoinstruction

Various Translations, Partial View

- Small immediate (imm "fits" in 12-bit signed imm space):
  - `li t0, 0x123`
  - Translation: `addi t0, zero, 0x123`

- Medium immediate:
  - `li t0, 0x12345678`
  - Translation:

```
lui  t0, 0x12345     # t0 = 0x12345000
addi t0, t0, 0x678   # t0 = 0x12345000 + 0x678
```

`li` Translation, Contd.

- **Q:** How is the instruction below translated in RV32I assembly?

```
li t0, 0x12345FF5
```

- *Hint:* Attention, **`addi`** sign-extends the immediate, which would result in performing addition with a negative number in this case
- **A:**

```
lui   t0, 0x12346  # round up upper bits
addi  t0, t0, -11  # correct lower bits
```

What Does *"Fit"* Mean?

- In `li` context, 12-bit **imm** "fits" if the most significant bit is zero
  - Sign-extension keeps the number positive, and so the result of `addi` is as expected
  - Range of immediate **imm** that "fits" is $-2^{10} <$ **imm** $< 2^{10} - 1$

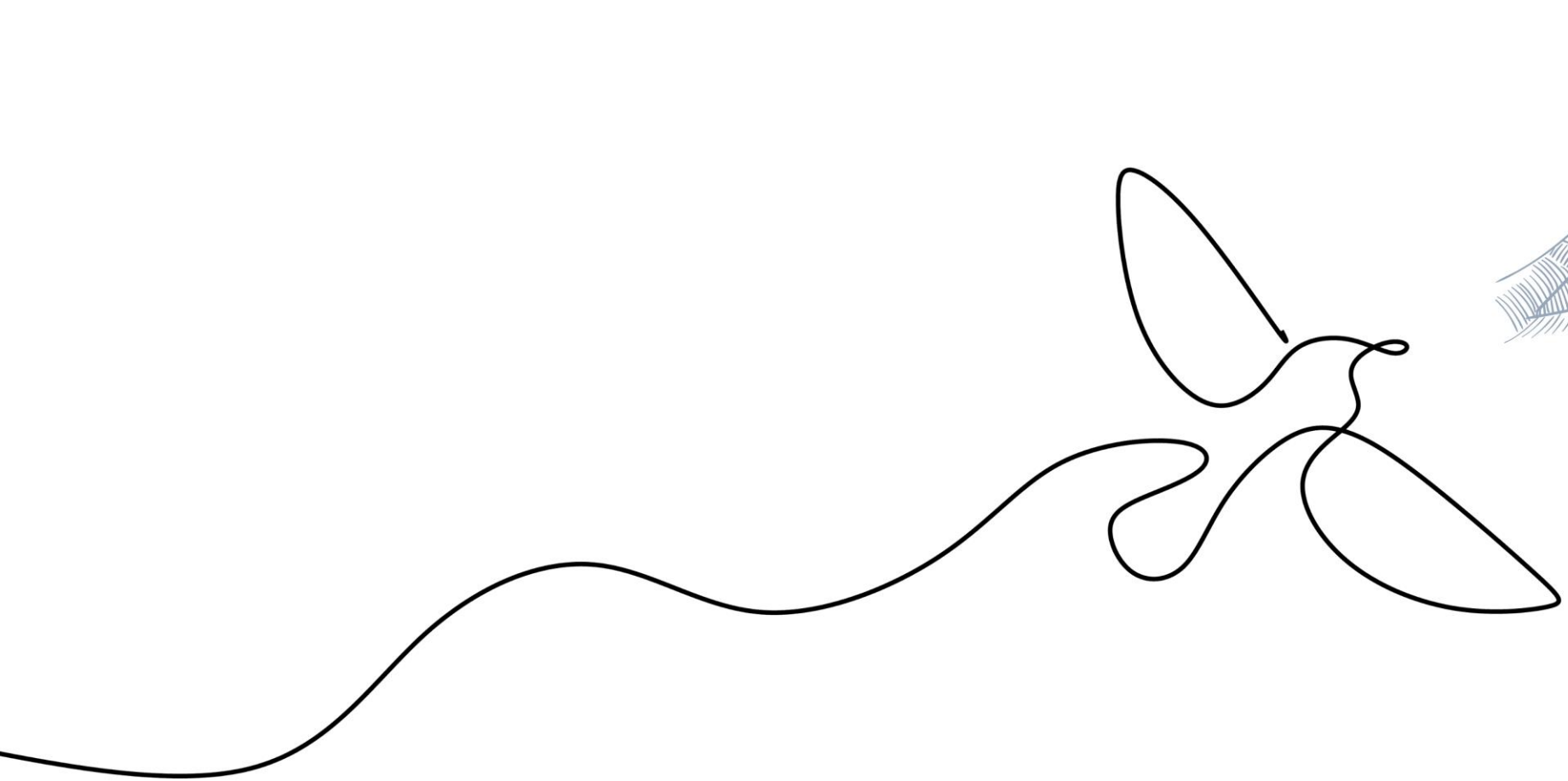
- If the most significant bit of the immediate **imm** is one
  - Sign-extension results in a negative value, and so the result of `addi` is not as expected
  - Solution:

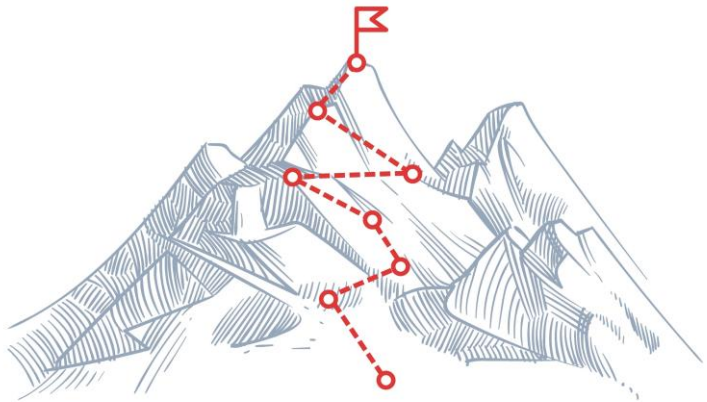
```
# li t0, 0x12345FF5
lui  t0, 0x12346    # round up upper bits
addi t0, t0, -11    # correct lower bits
```

`li` Pseudoinstruction

Various Translations, Complete

- Small immediate(imm "fits" in 12-bit signed imm space):
  - `li t0, 0x123`
  - Translation: `addi t0, zero, 0x123`
- Medium immediate (lower 12 bits "fit" in signed 12-bit signed imm space):
  - `li t0, 0x12345678`
  - Translation:
```
lui  t0, 0x12345     # t0 = 0x12345000
addi t0, t0, 0x678   # t0 = 0x12345000 + 0x678
```
- Large immediate (lower 12 bits do not "fit" in signed 12-bit imm space):
  - `li t0, 0x12345FF5`
  - Translation:
```
lui  t0, 0x12346     # round up upper bits
addi t0, t0, -11     # correct lower bits
```





`la` Pseudoinstruction

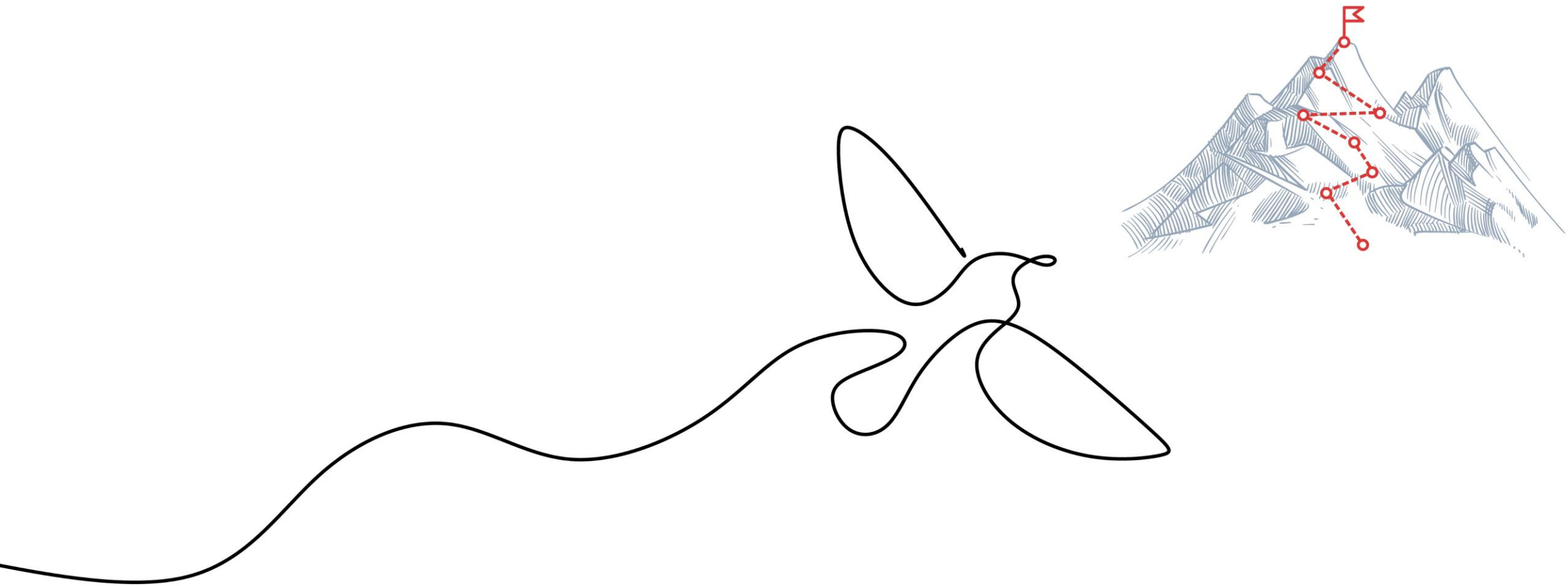
Usage

- *Recall:* `la` copies the address to a register
- Use `la` when `imm` is a label

Example: `la` and `li`

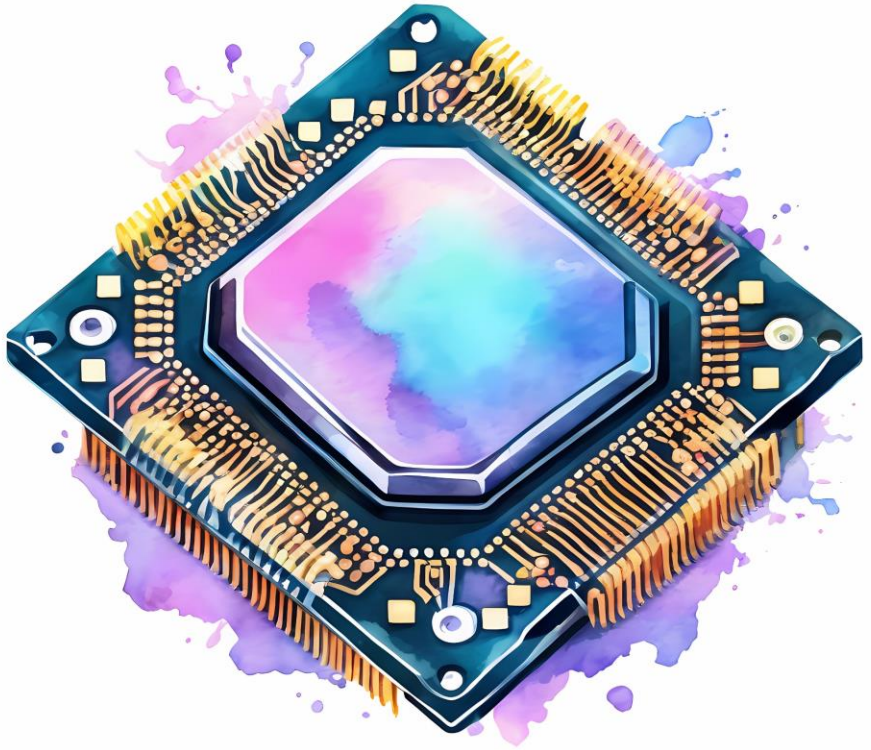
- *Recall:* `la` copies the address to a register
- *Recall:* Use `la` when `imm` is a label

```
.equ my_const, 0x555
.data
my_result:  .word   0x07070707
.text
la t0, my_result          # t0 = address of 0x07070707 in memory
lw t1, 0(t0)              # t1 = mem[0 + t0] = mem[my_result] = 0x07070707
li t2, my_const           # t2 = 0x555, symbol, note the use of pseudoinstr. li
sw t2, 0(t0)              # mem[t0] = t2 = 0x555, overwriting 0x07070707
```



Do-While Loop

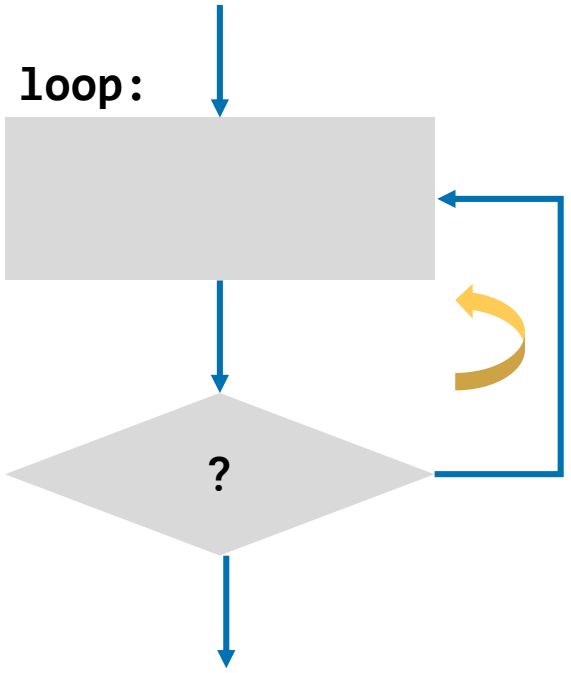
In Assembly



© Anastasi17 / Adobe Stock

Do-While Loop

Methodology




```
# initialization
# loop
do {
  …
  …
  …
} while (?)
```

```
.equ …
.text
  …
  …
loop:
  …
  …
  …
  …
  branch ? loop
```

Sum of an Array of Signed 32-bit Words

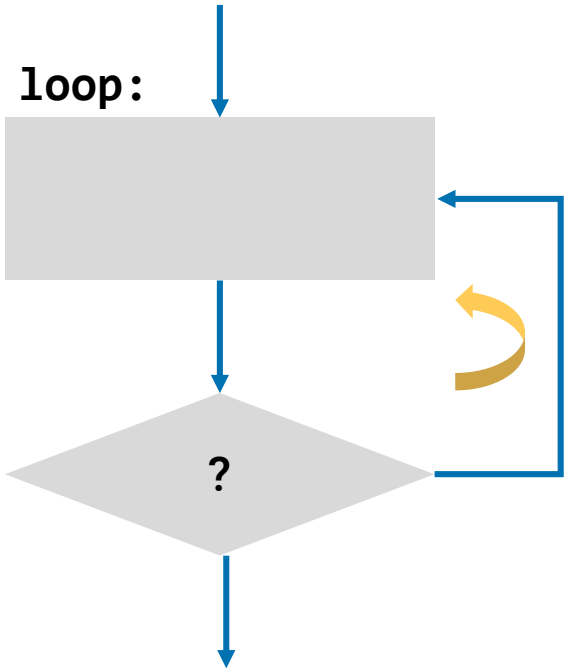
Loops in Assembly

- Write a piece of assembly code to compute the sum of an array of `n` signed 32-bit words in memory
- The array address is in register `t1`
- The array has 99 elements
- The final sum should reside in register `t0`

Ignore the possibility of overflow

Sum of an Array of Signed 32-bit Words

Solution

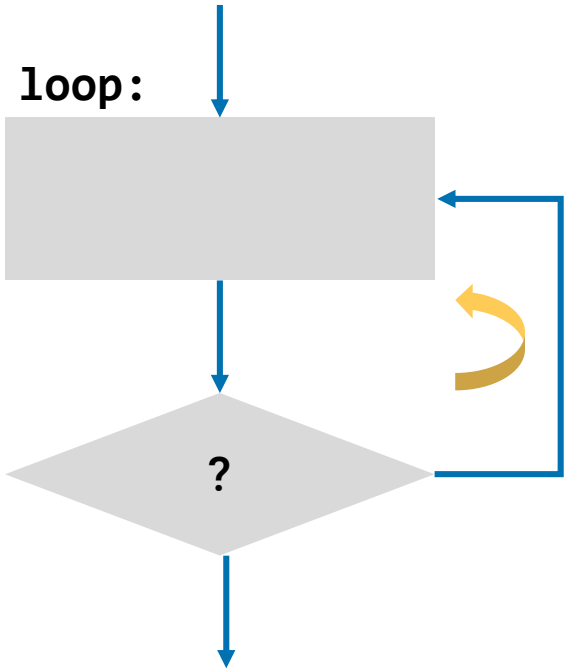



```
n = 99
# t0 = 0
# t2 = n
# loop
do {
  t0 = t0 + mem[t1]
  t1 = t1 + 4
  t2 = t2 - 1
} while (t2 != 0)
```

```
.equ n, 99
.text
  …
  …
loop:
  …
  …
  …
  …
  bne  ? loop
```

Sum of an Array of Signed 32-bit Words

Solution, Contd.



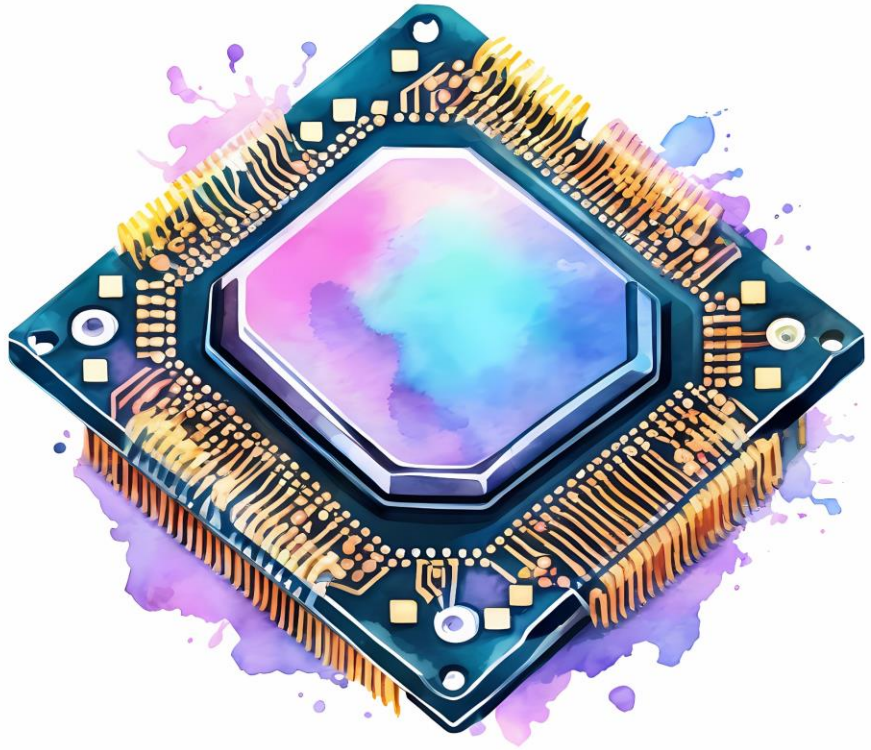

```
n = 99
# t0 = 0
# t2 = n
# loop
do {
  t0 = t0 + mem[t1]
  t1 = t1 + 4
  t2 = t2 - 1
} while (t2 != 0)
```

```
.equ n, 99
.text
  li t0, 0
  li t2, n  # t2 = 99
loop:
  lw   t3, 0(t1) # mem[t1]
  add  t0, t0, t3
  addi t1, t1, 4
  addi t2, t2, -1
  bnez t2, loop
```

Note: bnez rs1, imm ⇔ bne rs1, zero, imm

If-Then-Else

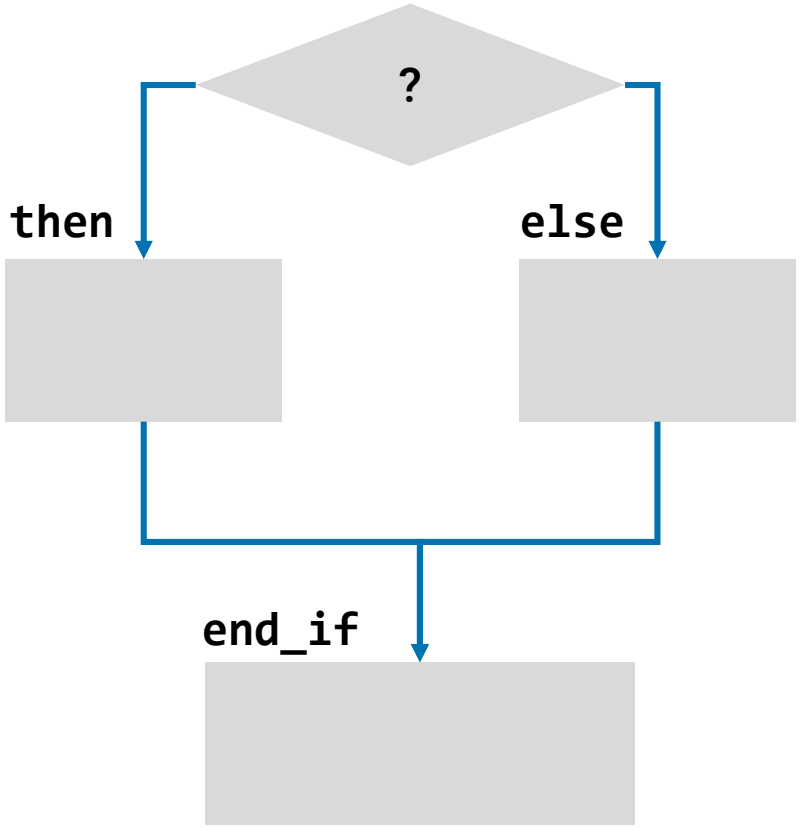
In Assembly



© Anastasi17 / Adobe Stock

If-Then-Else

Methodology



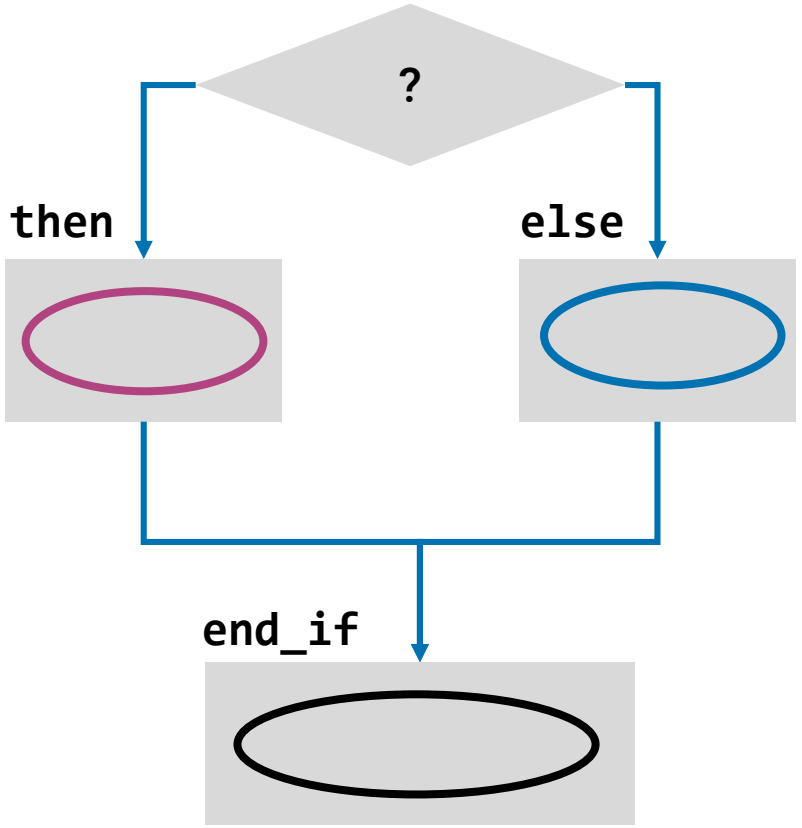

```
if (?) {
  …
} else {
  …
}
```

```
.text
  …
  branch ? then_begin
else_begin:
  …
  j end_if
then_begin:
  …

end_if:
  …
```

If-Then-Else

Methodology, Contd




```
if (?) {
  …
} else {
  …
}
```

```
.text
  …
  branch ? then_begin
else_begin:
  …
  j end_if
then_begin:
  …
end_if:
  …
```

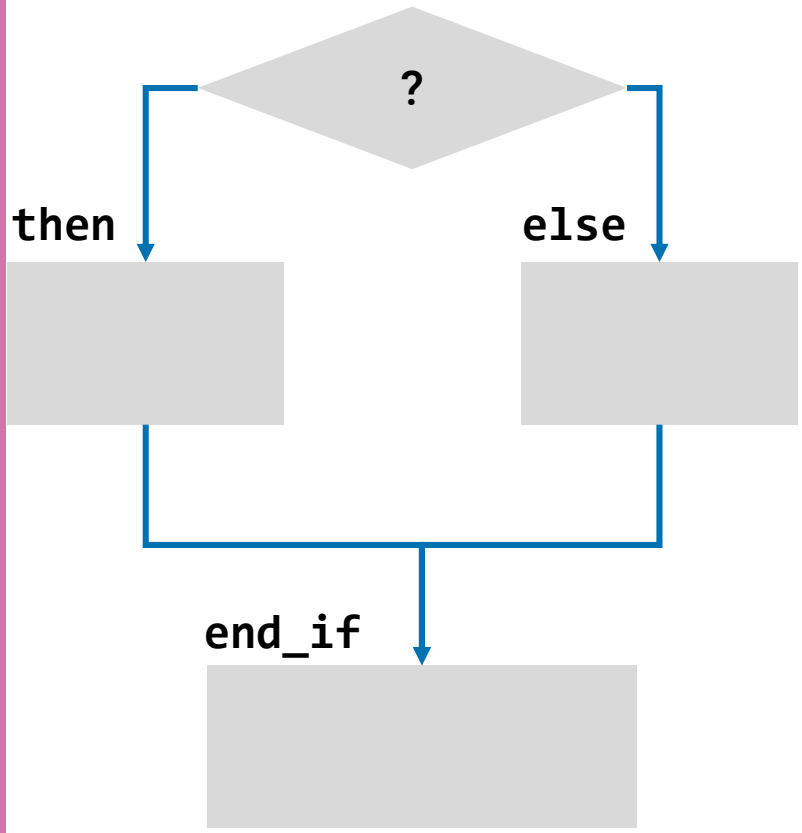
If-Then-Else

Example

- Write a piece of assembly code where
  - If register `t0` is 16, register `t1` gets **in**cremented
  - Otherwise, `t1` gets **de**cremented

If-Then-Else

Solution



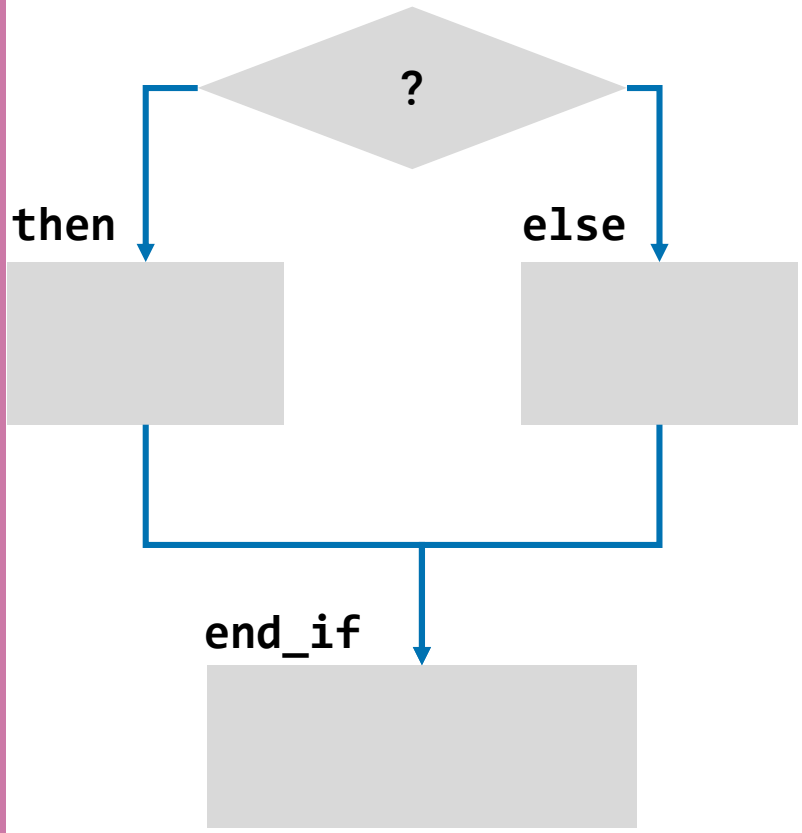

```
if (t0 == 16) {
  t1 = t1 + 1
} else {
  t1 = t1 - 1
}
```

```
.text
  …
  branch ? then_begin
else_begin:
  …
  j end_if
then_begin:
  …

end_if:
  …
```

If-Then-Else

Solution, Contd.



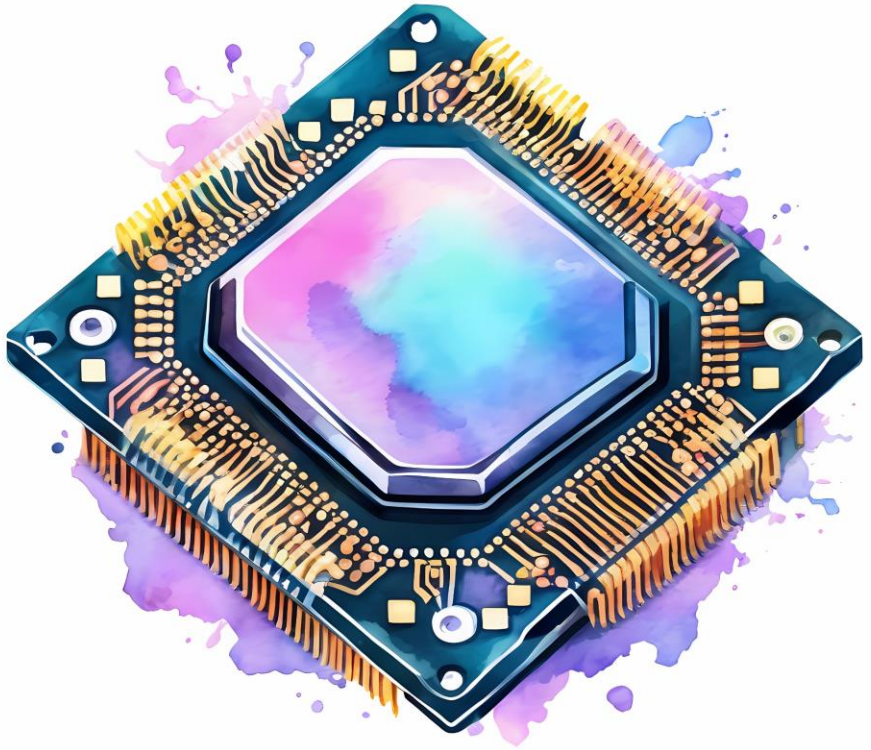

```
if (t0 == 16) {
  t1 = t1 + 1
} else {
  t1 = t1 - 1
}
```

```
.text
  li t2, 16  # t2 = 16
  beq t0, t2, then_begin
else_begin:
  addi t1, t1, -1
  j end_if
then_begin:
  addi t1, t1, 1

end_if:
  …
```

RV32I Reference Card

- Now available on Moodle
- Will be distributed during the final exam



© Anastasi17 / Adobe Stock