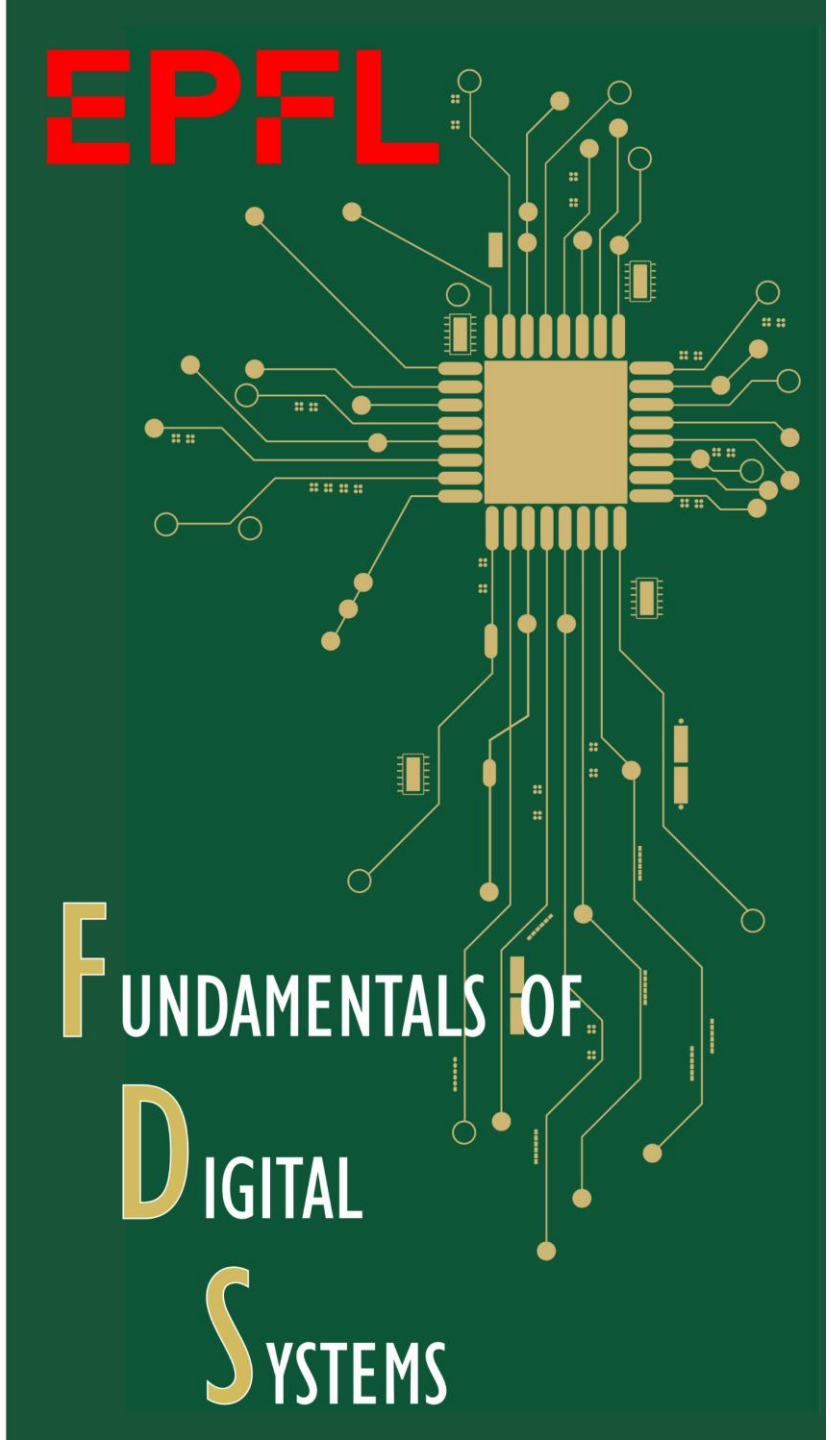


Computer Architecture

RISC-V Memory Access and Control Transfer Instructions

CS-173 Fundamentals of Digital Systems

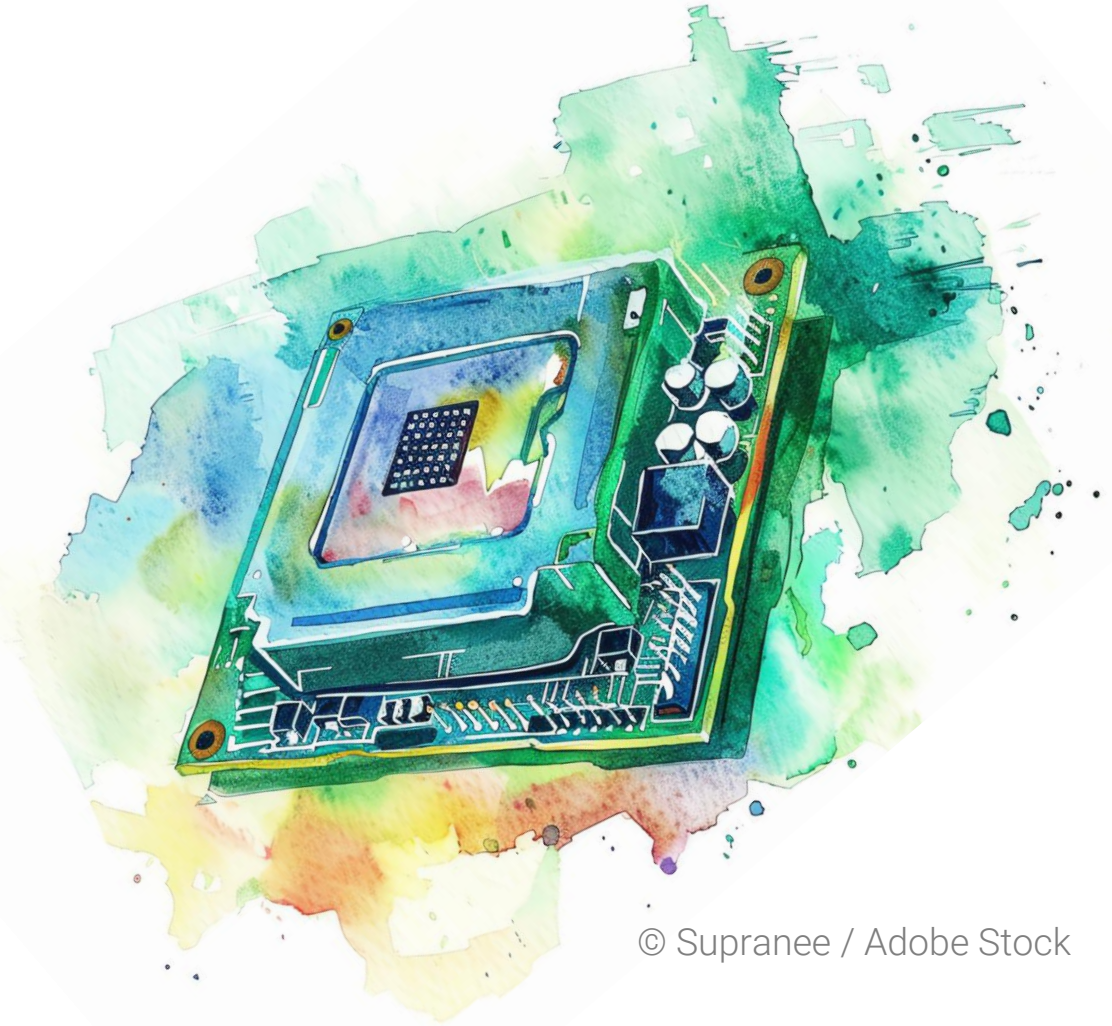
Mirjana Stojilović

Spring 2025

Previously on FDS

RISC-V Instruction Set Architecture

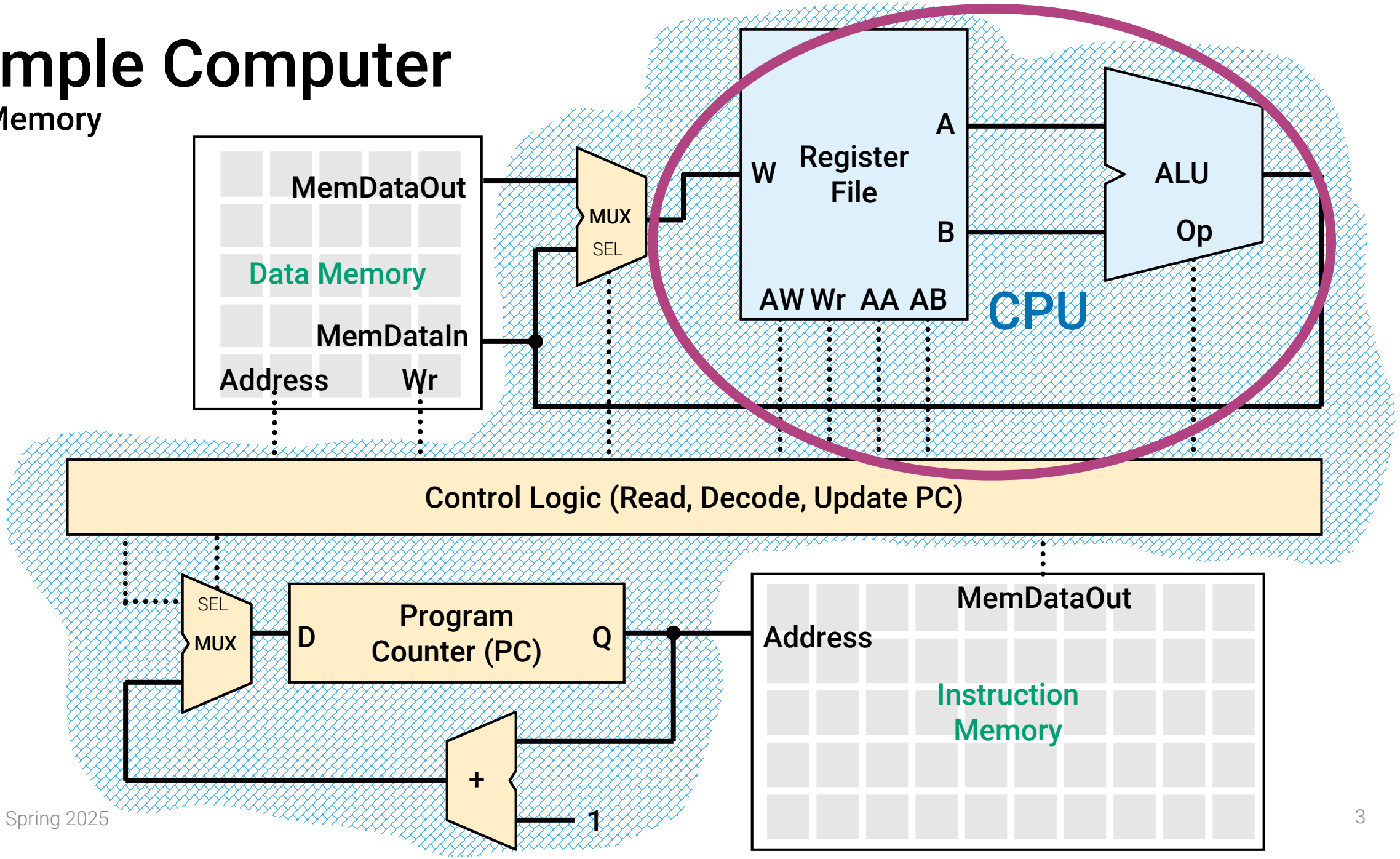
- Registers
- Integer Computational Instructions



© Supranee / Adobe Stock

A Simple Computer

CPU + Memory



RV32I Instruction Set Architecture

Outline

- Registers
- Integer computational instructions
- Instruction formats (**R/I/U**)

Previous lecture

- Memory read (load) and write (store) instructions
- Control transfer instructions
- Instruction formats, contd. (**S/B/J**)

- *Some topics are out of scope for CS-173, to be covered in the Computer Architecture CS-208*
  - *Memory model; Control and status register instructions;*
  - *Environment call and breakpoints; Exceptions, traps, and interrupts;*

RV32I Instruction Set Architecture

Outline

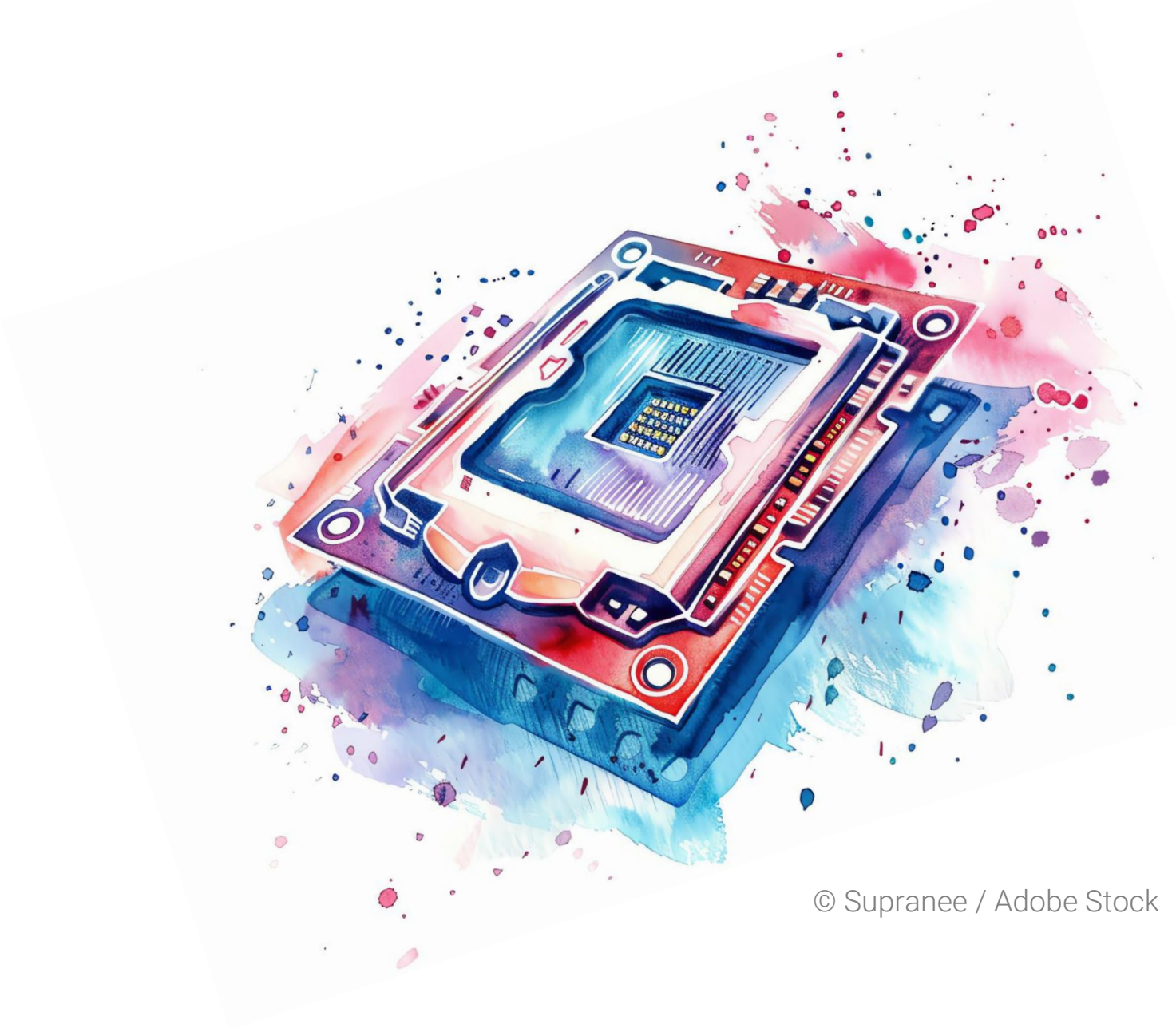
- Registers
- Integer computational instructions
- Instruction formats (**R/I/U**)
- Memory read (load) and write (store) instructions
- Control transfer instructions
- Instruction formats, contd. (**S/B/J**)

This lecture

- *Some topics are out of scope for CS-173, to be covered in the Computer Architecture CS-208*
  - *Memory model; Control and status register instructions;*
  - *Environment call and breakpoints; Exceptions, traps, and interrupts;*

Let’s Talk About

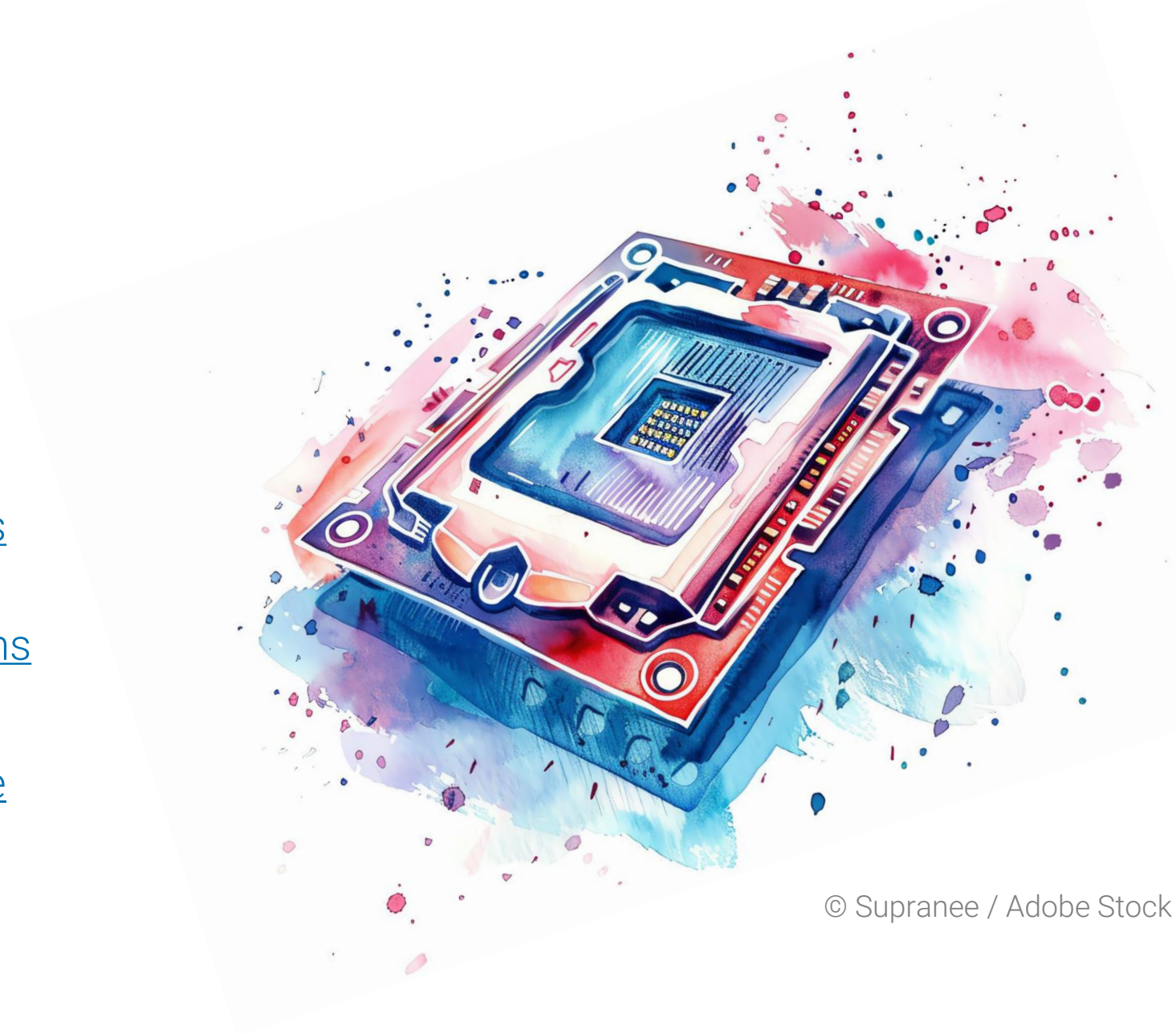
RV32I ISA, Continued…



© Supranee / Adobe Stock

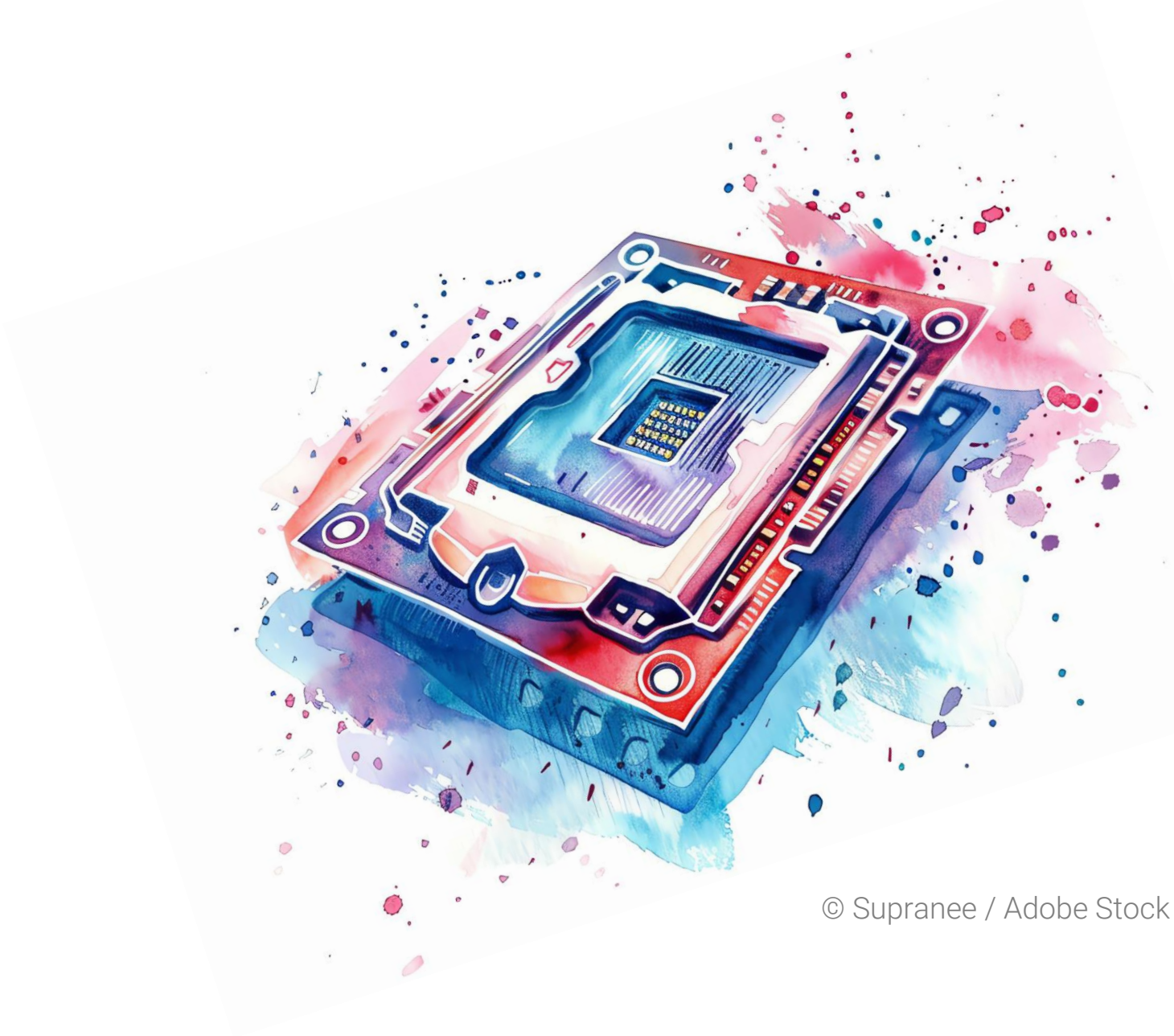
Quick Outline

- Memory
  - General properties
  - Instruction alignment: Example
  - Byte ordering (endianness)
  - Memory read: **load** instructions
    - Example
  - Memory write: **store** instructions
    - Example
- Conditional branches: Example
- Unconditional jumps
- Pseudoinstructions:
  - j and mv



Memory

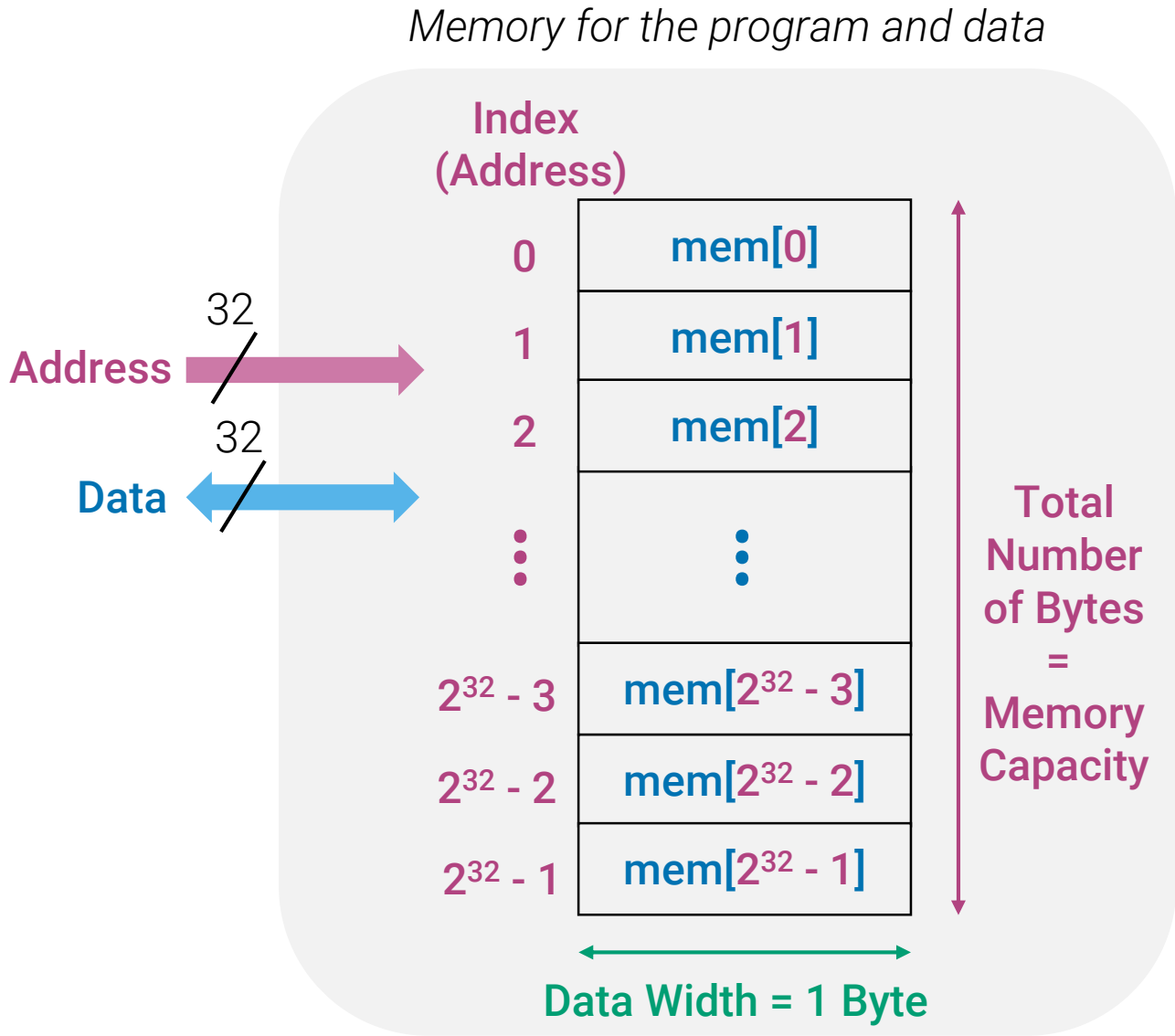
- General Properties
- Byte Order (Endianness)



Memory

RV32I, General Properties

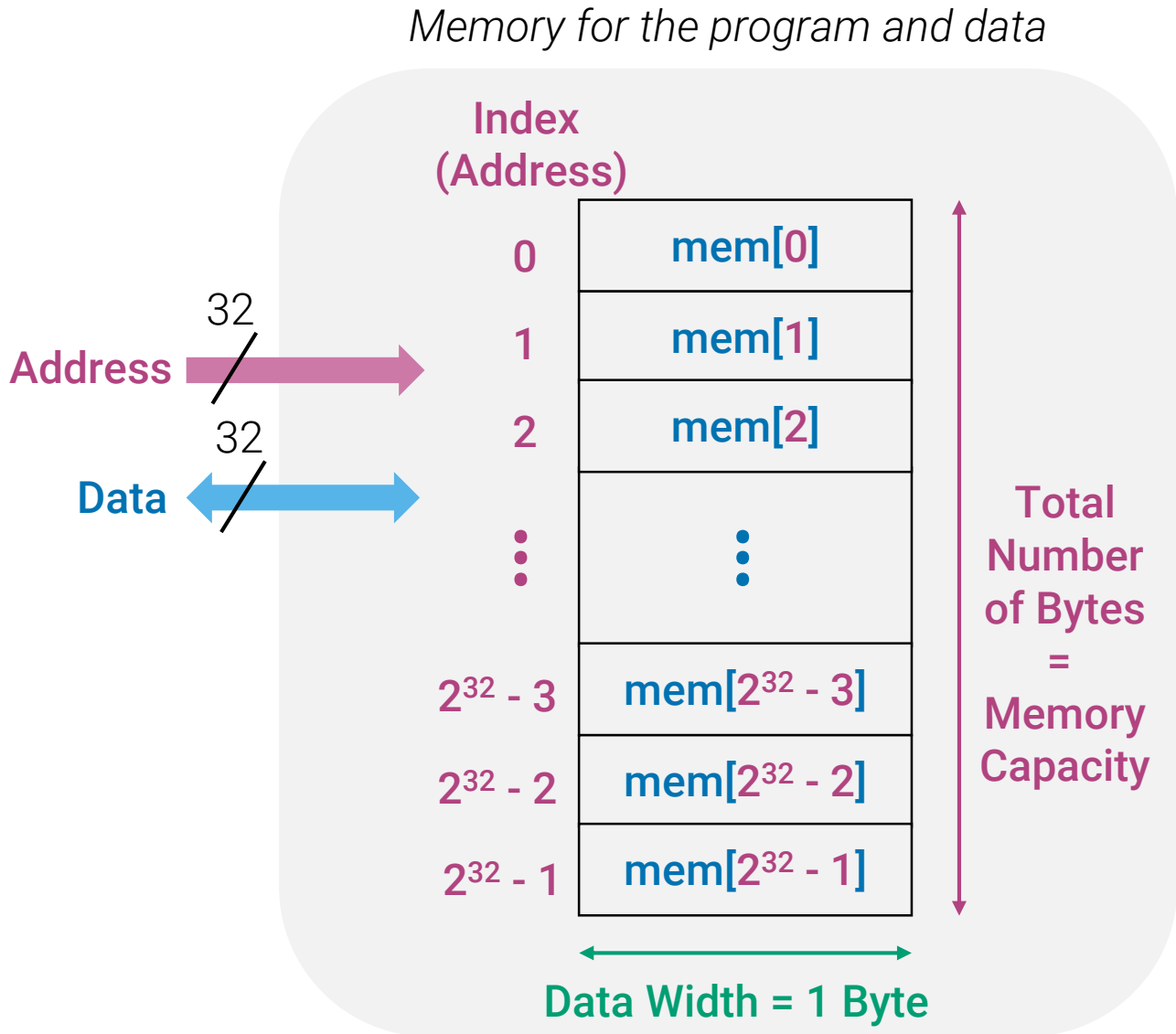
- 32-bit addresses
  - Address range: from 0 to ($2^{32} - 1$)
- Data width is one Byte = 1 **B**
- Memory capacity:
  - NumAddresses × DataWidth = 2^{32} B
- **Von Neuman** architecture
  - Memory is unified and **shared** by the program (instructions) and data



Memory

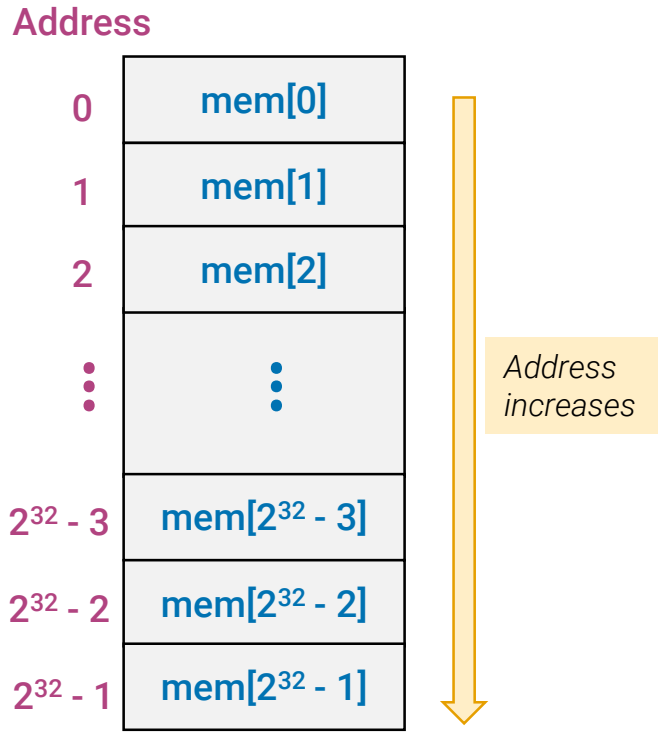
RV32I, Byte-Addressable

- Memory is **byte-addressable**, meaning that one address corresponds to one byte
- Data size terminology
  - **Byte = 8 bits = 1 B**
  - Halfword = 16 bits = 2 B
  - **Word = 32 bits = 4 B**
  - Quadword = 128 bits = 16 B
- RV32I supports reading/writing one byte or an entire word at a time

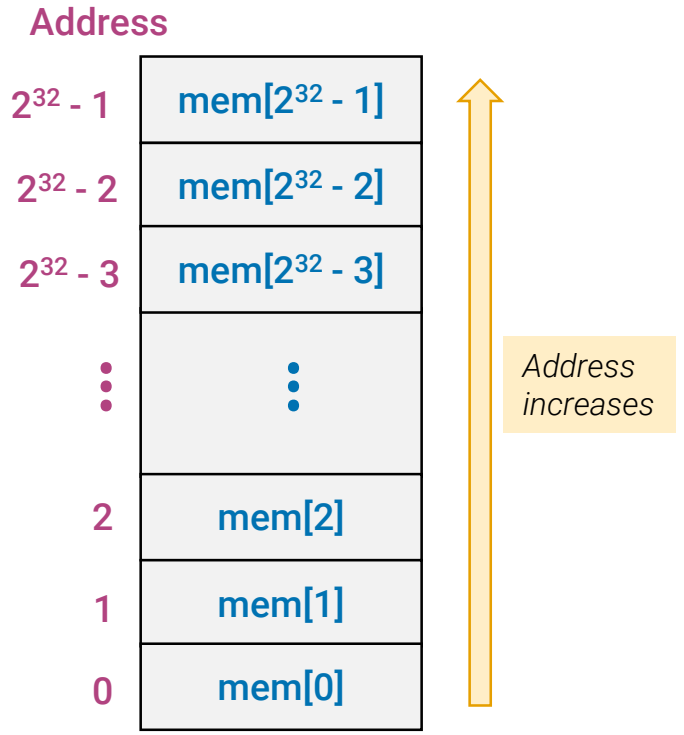


Memory
Visualization

- Interchangeable ways of drawing memory layout you will see in literature
  - **Low** addresses at the **top**
    (addresses growing downward)

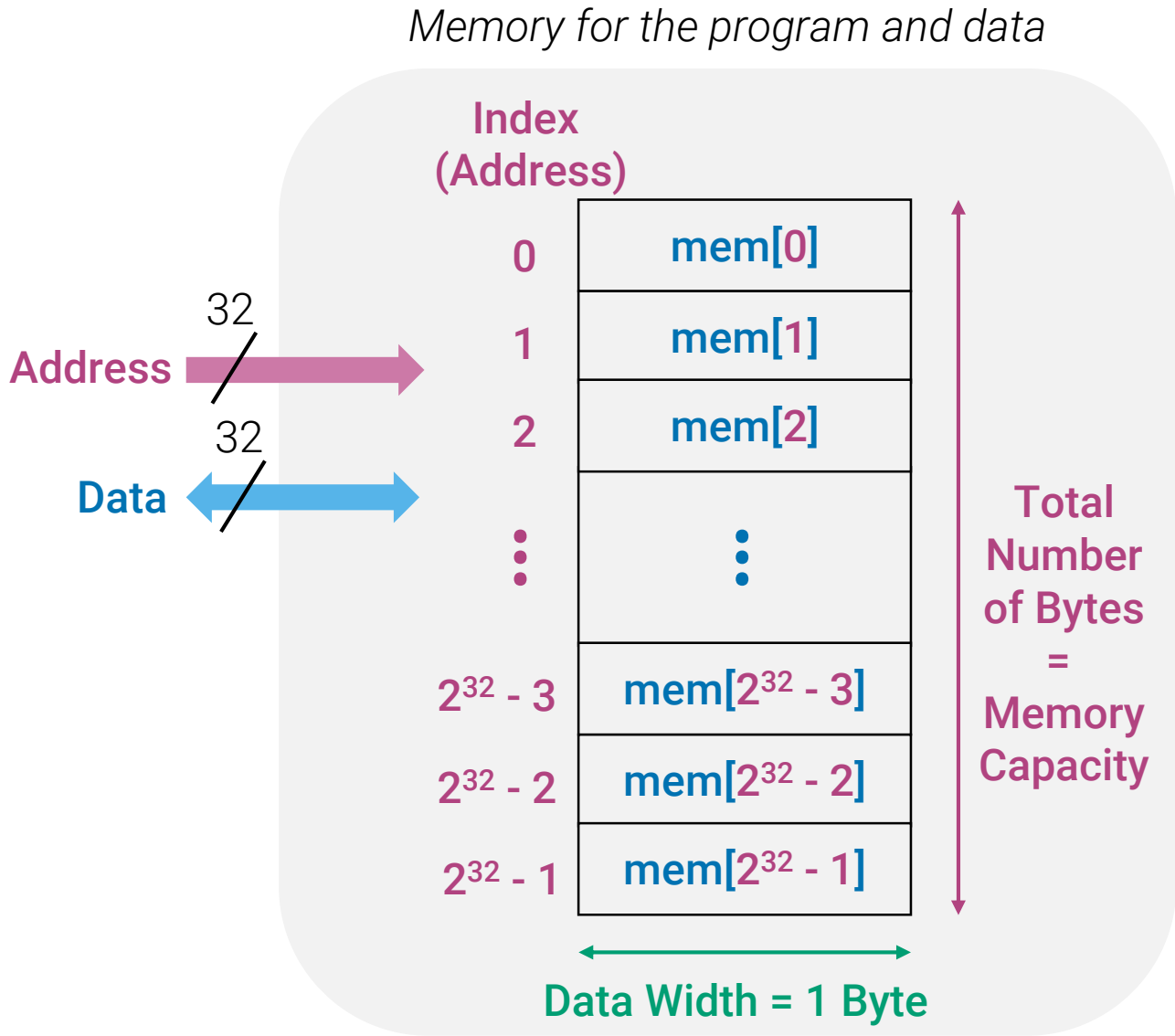


  - **High** addresses at the **top**
    (addresses growing upward)



Memory
Alignment of Instructions

- *Recall:* 32-bit instructions in RV32I
- Instruction occupies one word
  - 1 word = 32 bits = 4 B = 2^2 B
- Memory capacity is 2^{32} B, which corresponds to 2^{30} words
  - N_{words} = Memory Capacity / 1 word = 2^{32} B / 2^2 B = 2^{30}
- Instructions must be naturally aligned on 32-bit boundaries



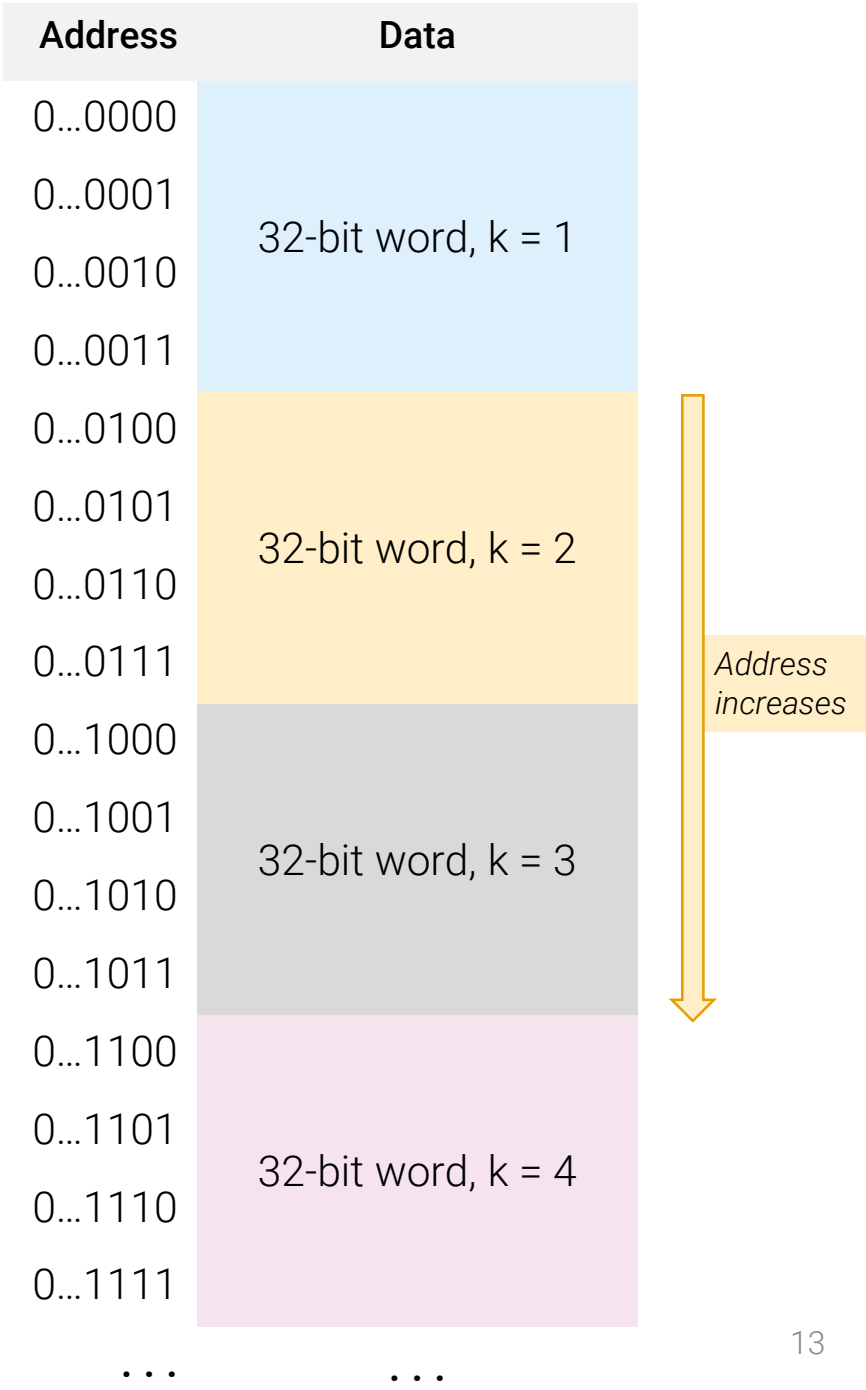
Memory

Alignment of Instructions, Contd.

- *Recall:* Instructions must be naturally **aligned** on 32-bit boundaries
- In other words
  - Occupy four consecutive addresses (each address corresponds to one byte)
  - Valid address ranges: 0-3, 4-7, 8-11, …

  Therefore, instructions span blocks of memory addresses in the format

$$(4k-4, 4k-3, 4k-2, 4k-1)\ ,\ 1 \leq k \leq 2^{30}$$



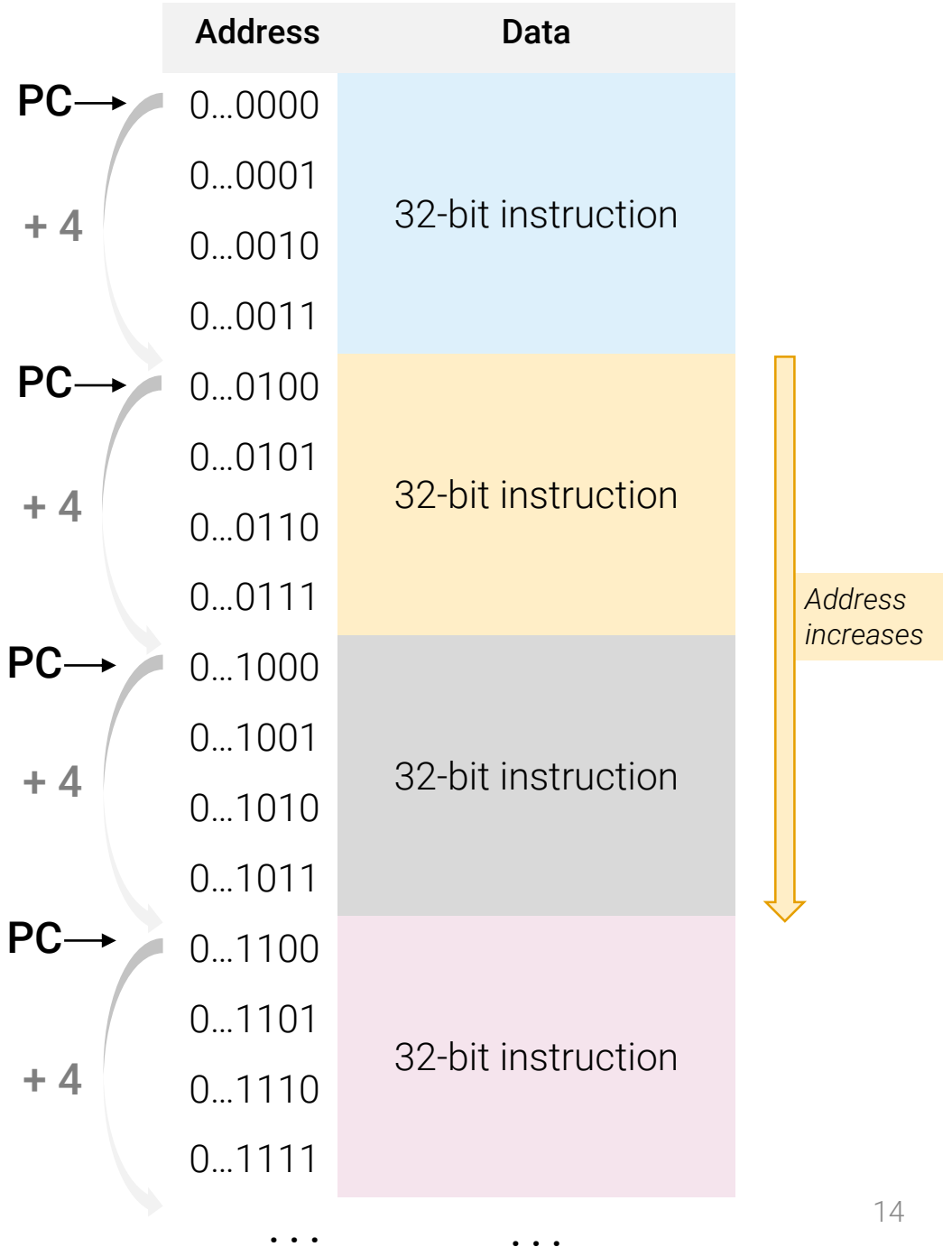
Memory

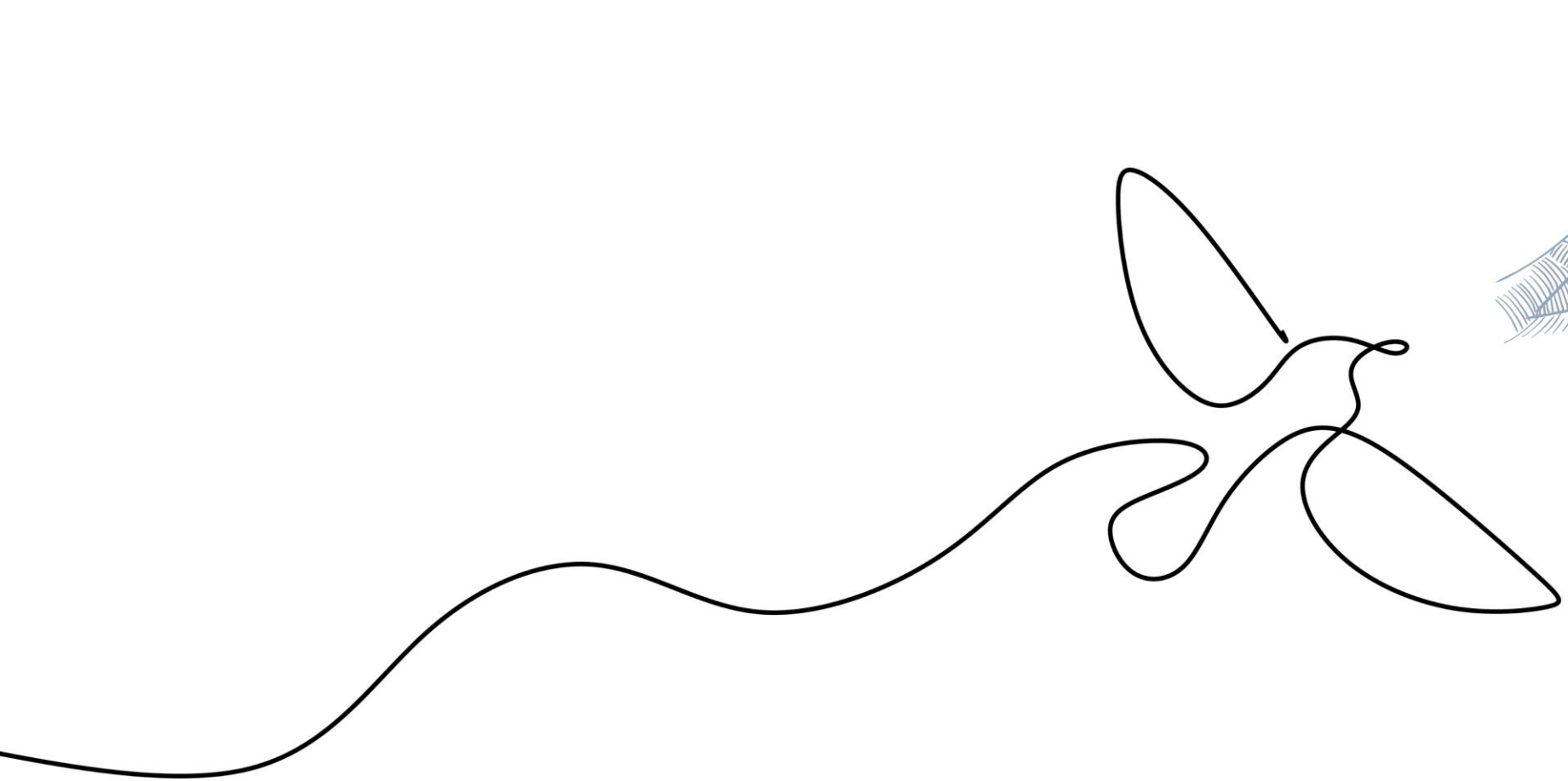
Program Counter Update

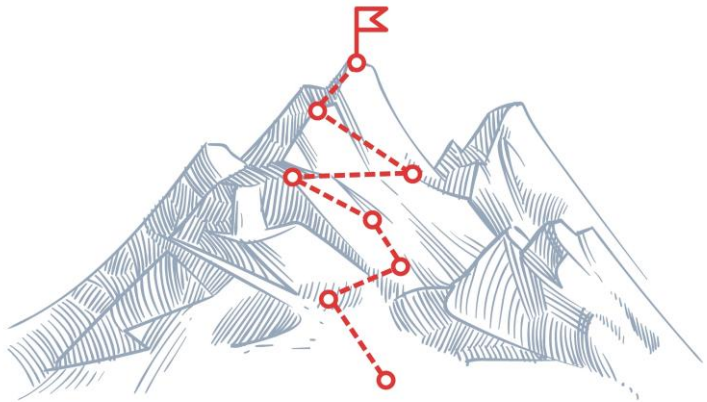
- *Recall:* Instructions must be naturally aligned on 32-bit boundaries
- Program Counter (PC) register keeps **the beginning address of the four bytes** of the program instruction

- Consequently, to prepare for reading the next program instruction from the memory, the value in the PC is increased in steps of four

PC = PC + 4







Recall: Registers

RV32I

- Program variables are commonly kept in `t` and `s` registers
- Registers are in the Register File (**not** in the memory)
- `li` pseudoinstruction
  - `li rd, imm` copies the sign-extended `imm`ediate to the destination register `rd`

Register	Name	Description
`x0`	`zero`	Hard-wired zero
`x1`	`ra`	*Return address
`x2`	`sp`	*Stack pointer
`x3`	`gp`	*Global pointer
`x4`	`tp`	*Thread pointer
`x5-7`	`t0-2`	Temporaries
`x8-9`	`s0-1`	Saved registers
`x10-11`	`a0-1`	*Function arguments/return values
`x12-17`	`a2-7`	*Function arguments
`x18-27`	`s2-11`	Saved registers
`x28-31`	`t3-6`	Temporaries

* *Out of scope for CS-173*

Understanding RISC-V Assembly

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

`li rd, imm`
pseudoinstruction;
copies the sign-extended `imm`ediate to the destination register `rd`

Examine the code at the left and answer the following questions:

- What is the value of `t3` at the end of this program?
- Assuming the program is in memory starting from the address 0x20, at which **memory address** is the instruction below? What is the **range** of memory addresses occupied by this instruction?

```
sll   t3, t3, t1
```

- Rewrite the code to use the **slli** instruction instead of **sll**. Try reducing the code size (the total number of instructions) as much as you can.
- How is the pseudoinstruction **nop** encoded in RV32I?

Understanding RISC-V Assembly

Solution

EXAMPLES

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

`li rd, imm`
pseudoinstruction;
copies the sign-extended **imm**ediate to the destination register **rd**

Q: What is the value of `t3` at the end of this program?

A: The program multiplies the value of register `t0` by 10 and places the result in register `t3`. Therefore, `t3` = 0x46 = $(70)_{10}$.

```
li    t0, 7          # t0 = 7 (input)
li    t1, 1          # t1 = 1 (shift amount)
sll   t2, t0, t1     # t2 = t0 << 1 = t0 * 2 = 14
sll   t3, t2, t1     # t3 = t2 << 1 = t0 * 4 = 28
sll   t3, t3, t1     # t3 = t3 << 1 = t0 * 8 = 56
add   t3, t3, t2     # t3 = t0 * 10 = 70
nop                  # no effect (for debugging)
```

Understanding RISC-V Assembly

Instruction address

EXAMPLES

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

- **Q:** Assuming the program is stored in memory starting from address **0x20**, at which memory address is the instruction below? What is the range of memory addresses occupied by this instruction?

```
sll   t3, t3, t1
```

```
Program start    0x20: li    t0, 7
                 0x24: li    t1, 1
                 0x28: sll   t2, t0, t1
                 0x2C: sll   t3, t2, t1
Starts at 0x30:  0x30: sll   t3, t3, t1      Range: from 0x30 to 0x33
                 0x34: add   t3, t3, t2
                 0x38: nop
```

Understanding RISC-V Assembly

Reducing code size

EXAMPLES

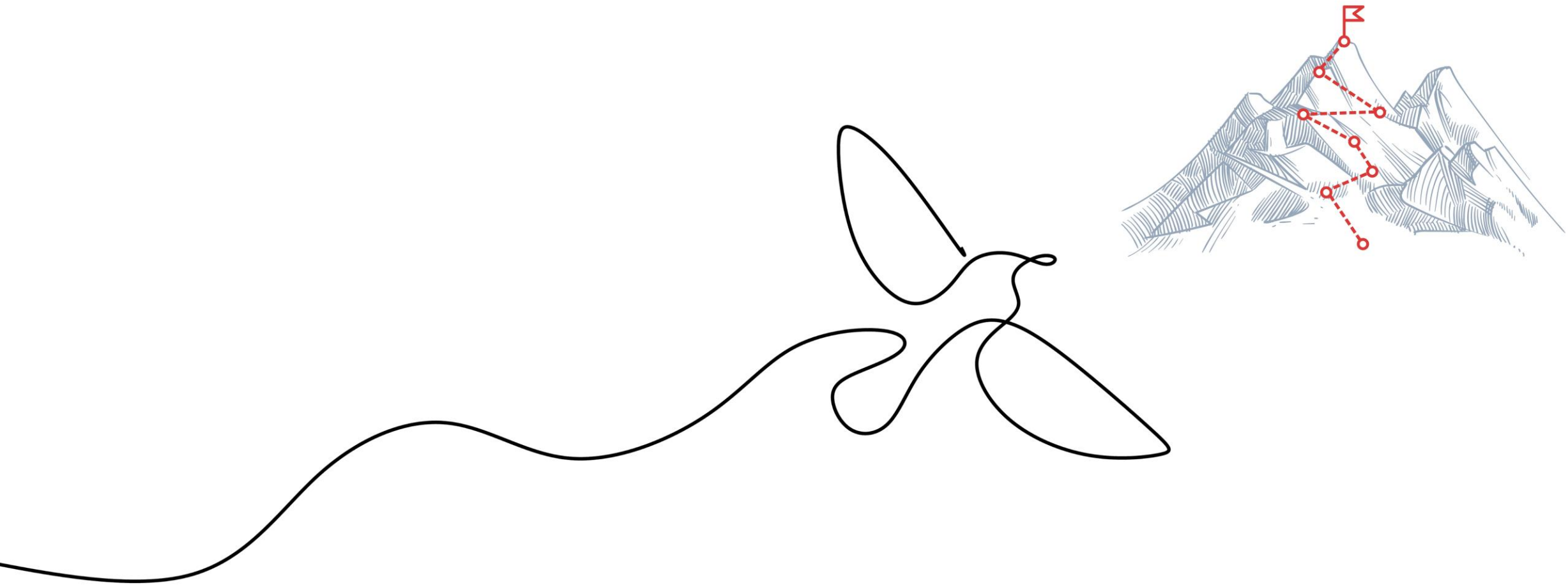
```
0x20: li    t0, 7
0x24: li    t1, 1
0x28: sll   t2, t0, t1
0x2C: sll   t3, t2, t1
0x30: sll   t3, t3, t1
0x34: add   t3, t3, t2
0x38: nop
```

- **Q:** Rewrite the code to use the `slli` instruction instead of `sll`. Try reducing the code size (the total number of instructions) as much as you can.
- **Q:** How is the pseudoinstruction **nop** encoded in RV32I?

- **A:**

```
0x20: li    t0, 7        # t0 = 7
0x24: slli  t1, t0, 1    # t1 = t0 << 1 = t0 * 2
0x28: slli  t2, t0, 3    # t2 = t0 << 3 = t0 * 8
0x2C: add   t3, t1, t2   # t3 = t0 * 10
0x30: nop   # addi x0, x0, 0; can be removed,
            # further reducing the code size in memory
```

By using `slli` (shift left logical by the immediate), we saved two instructions, resulting in a more compact code, occupying eight bytes less in memory.
To further reduce the code size, we can remove **nop** pseudoinstruction.



Byte Ordering

- A 32-bit memory word $W = \{w_{31}, w_{30}, ..., w_1, w_0\}$ has four bytes:

$$B_0 = \{b_7, b_6, ..., b_0\} = \{w_7, w_6, ..., w_1, w_0\}$$ *Small indices, i.e., the **little** end of the 32-bit word*

$$B_1 = \{b_7, b_6, ..., b_0\} = \{w_{15}, w_{14}, ..., w_9, w_8\}$$

$$B_2 = \{b_7, b_6, ..., b_0\} = \{w_{23}, w_{22}, ..., w_{17}, w_{16}\}$$

$$B_3 = \{b_7, b_6, ..., b_0\} = \{w_{31}, w_{30}, ..., w_{25}, w_{24}\}$$ *Big indices, i.e., the **BIG** end of the 32-bit word*

Q: How are these bytes mapped to the word's four consecutive memory addresses $(4k-4, 4k-3, 4k-2, 4k-1)$, $1 \leq k \leq 2^{30}$?

A: Two common orderings exist: **little endian** and **big endian**

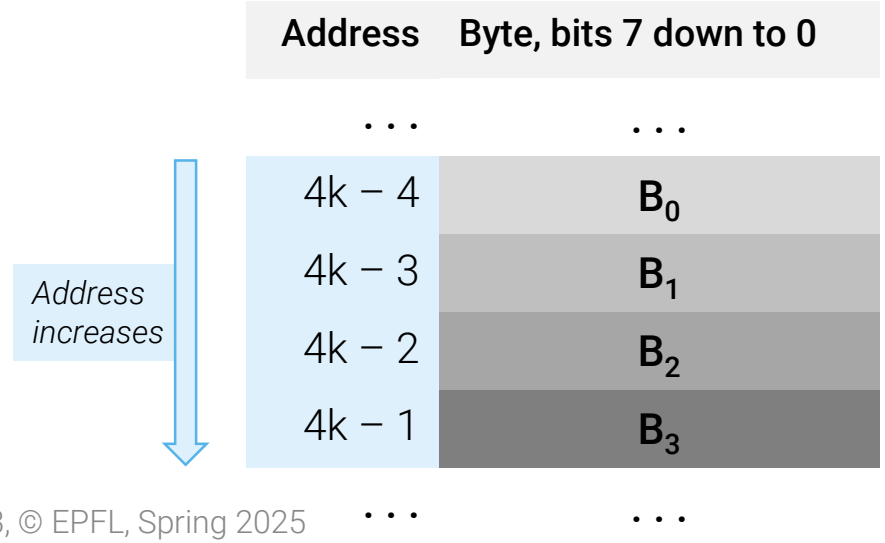
The latest RISC-V ISA specification supports both (20240411). Initially, only little-endian byte ordering was assumed.

Unless specified otherwise, we will assume little-endian byte ordering.

Memory
Byte Ordering

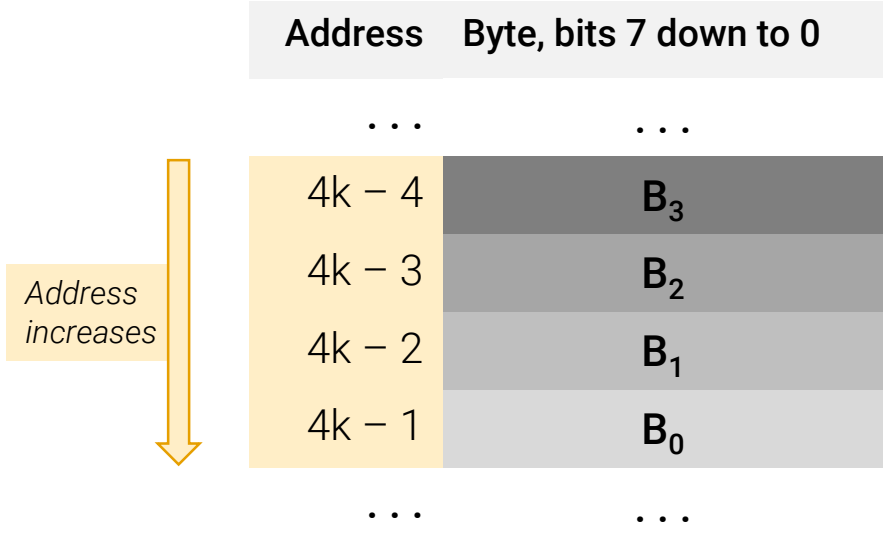
Little-endian

- The least significant bits of the word (byte B_0, the one at the “little” end of the word) is at the lowest memory address



Big-endian

- The most significant bits of the word (byte B_3, the one at the “BIG” end of the word) is at the lowest memory address



Instruction Encoding

Little-Endian Byte Order

EXAMPLES

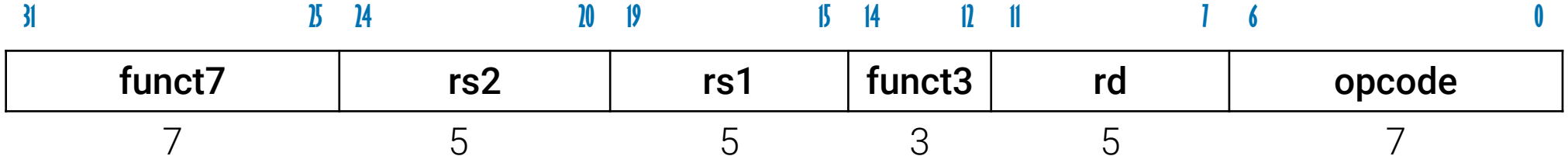
- Consider the instruction `srl t0, t1, t2` at memory address 0x100. Assuming the little-endian byte order, fill in the table below with the corresponding memory contents.

Address	Data
0x100	?
0x101	?
0x102	?
0x103	?

Instruction Encoding

Solution, Little-Endian Byte Order

- `srl t0, t1, t2`
  - *Recall:* `srl` is a register-register operation of R-type format



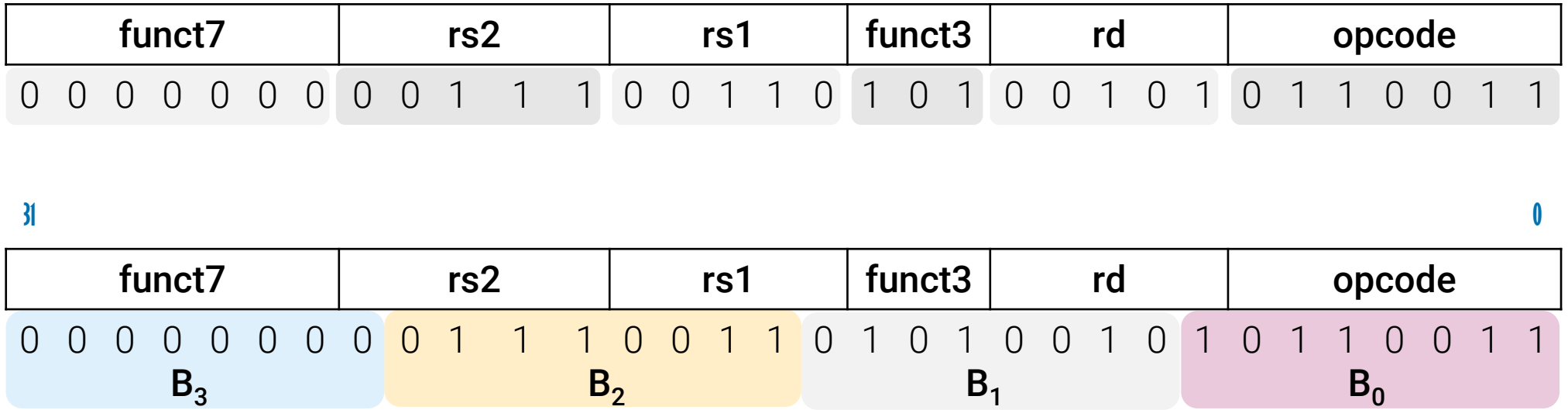
  - funct7 = 0
  - rs2 = t2 = x7 = $(111)_2$
  - rs1 = t1 = x6 = $(110)_2$
  - funct3 = $(101)_2$
  - rd = t0 = x5 = $(101)_2$
  - opcode = $(0110011)_2$

Instruction Encoding

Solution, Little-Endian Byte Order

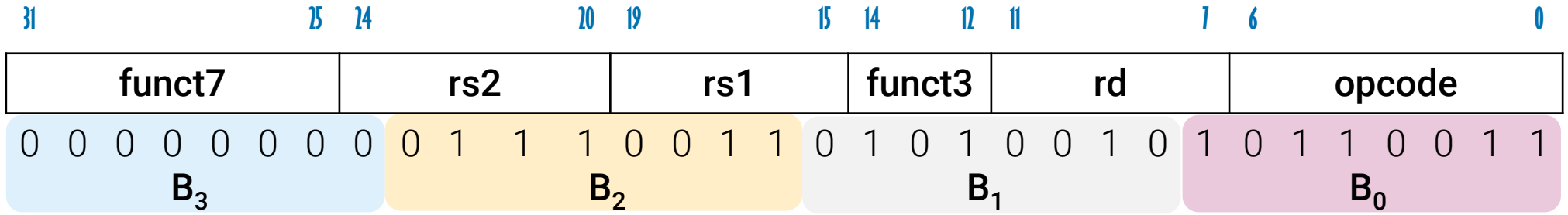
- Placing all bits together at their corresponding indices, we obtain the instruction word

funct7	= 0
rs2	= t2 = x7 = $(111)_2$
rs1	= t1 = x6 = $(110)_2$
funct3	= $(101)_2$
rd	= t0 = x5 = $(101)_2$
opcode	= $(0110011)_2$



Instruction Encoding

Solution, Little-Endian Byte Order



- Therefore, `srl t0,t1,t2` is encoded as **0x007352B3**
- Finally, with the **little** end of the word at the lowest memory address (little-endian byte ordering), the memory contents become:

Address	Data
0x100	0xB3
0x101	0x52
0x102	0x73
0x103	0x00

Note: Check out the online instruction encoder/decoder from UC Davis: Link

Instruction Encoding

Big-Endian Byte Order

EXAMPLES

- Consider the instruction `srl t0, t1, t2` at memory address 0x100. Assuming the big-endian byte order, fill in the table below with the corresponding memory contents.

Address	Data
0x100	?
0x101	?
0x102	?
0x103	?

Instruction Encoding

Solution, Big-Endian Byte Order

EXAMPLES

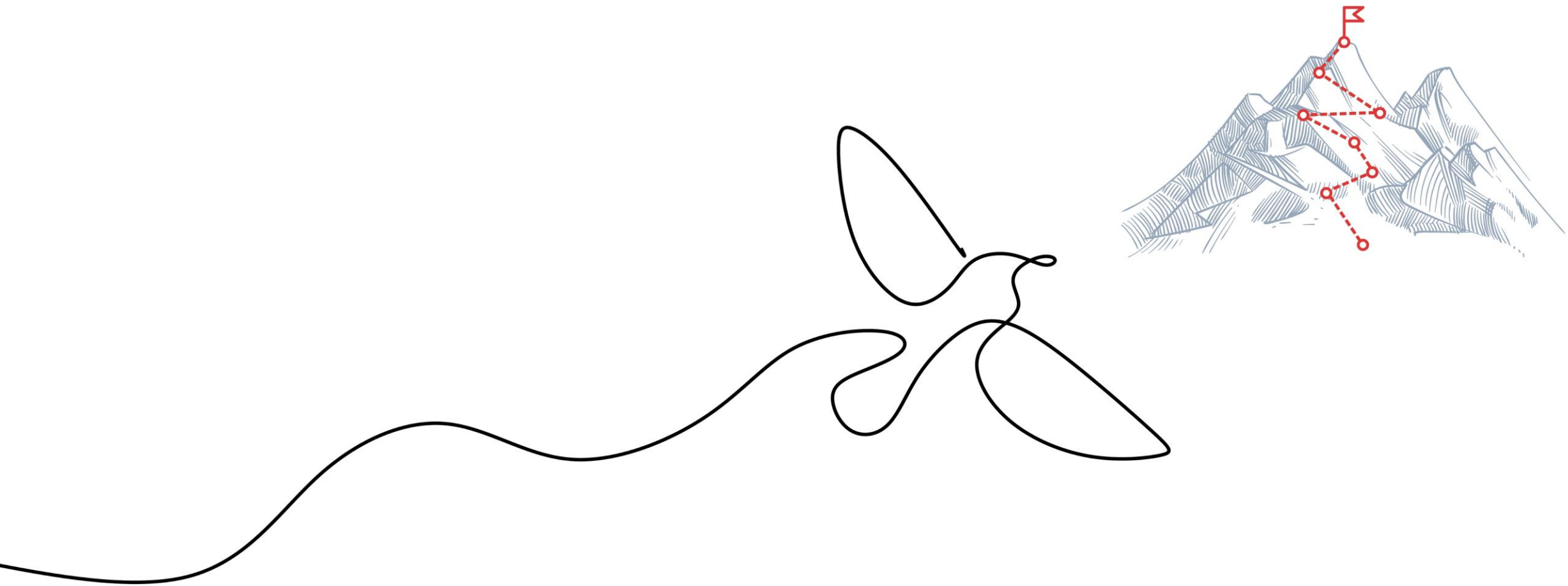
- *Recall:* `srl t0,t1,t2` is encoded as 0x007352B3
- Finally, with the BIG end of the word at the lowest memory address (little-endian byte ordering), the memory contents become:

Big endian

Address	Data
0x100	0x00
0x101	0x73
0x102	0x52
0x103	0xB3

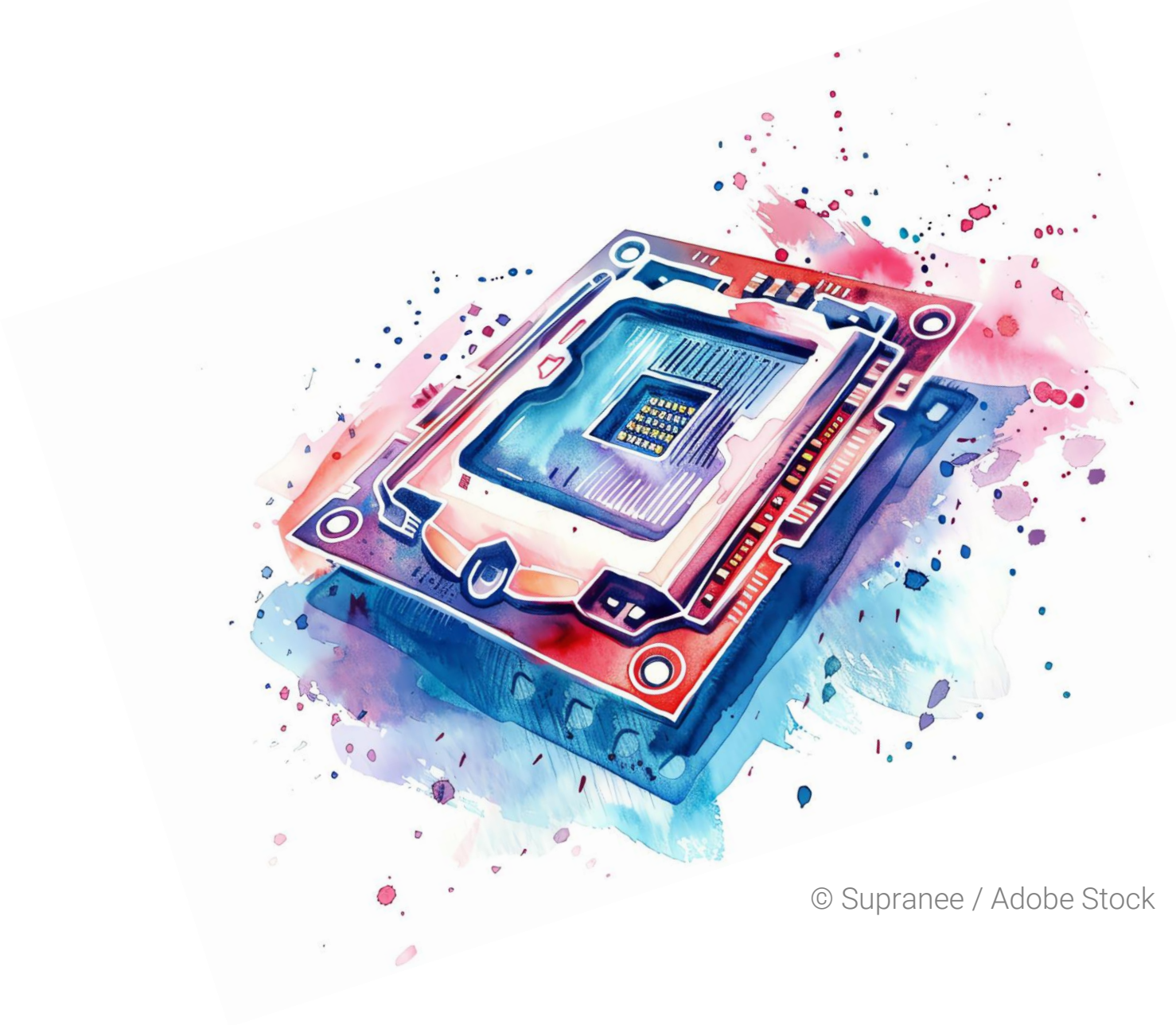
Recall: Little endian

Address	Data
0x100	0xB3
0x101	0x52
0x102	0x73
0x103	0x00



Memory

- Read = **load** from memory
- Write = **store** in memory



© Supranee / Adobe Stock

Memory Read and Write

Load and Store

- Memory **read**/**write** operations are called **load**/**store**
- RV32I provides instructions for **loading** signed and **u**nsigned
  - **b**ytes: `lb`, `lbu`
  - **w**ords: `lw`
- Similarly, it provides instructions for **storing**
  - **b**ytes: `sb`
  - **w**ords: `sw`

Memory Read and Write

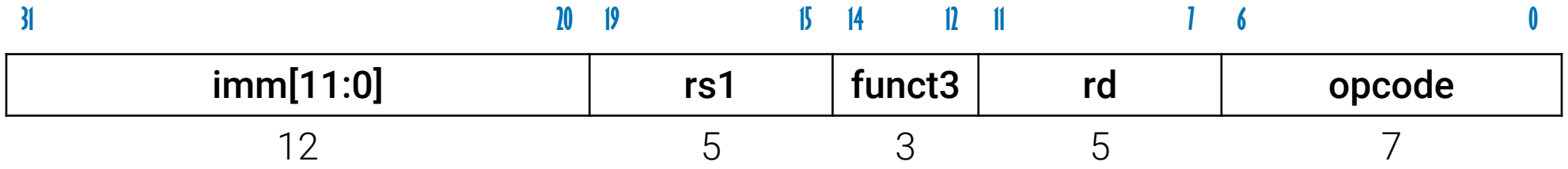
Load and Store, Contd.

- **Signed** bytes read from memory are **sign-extended** to 32 bits and then copied to the destination registers
  - Widening of narrow data allows subsequent integer computation instructions to operate correctly on all 32 bits, even if the natural data types are narrower
- **Unsigned** bytes read from memory, common in text characters and unsigned integers, are **zero-extended** to 32 bits and then copied

- Address and data bus are 32 bits wide; therefore, memory accesses—reading from it or writing into it—operate on 32-bit binary vectors

Load Instructions

I-type Format

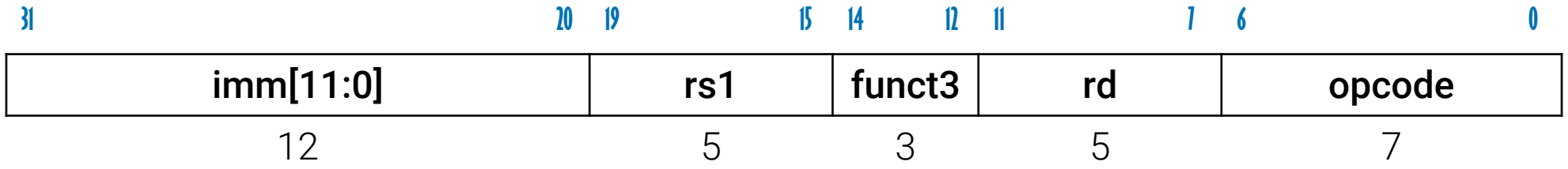


- Load: Copy a value from memory to register **rd** in the register file (RF)
- The memory address is obtained by **adding** **RF[rs1]** and the **sign-extended immediate**
- **rs1:** 5-bit index of the **base** register
- **rd:** 5-bit index of the destination register
- **opcode:** $(0000011)_2$, 7-bit operation code
- **funct3:** 3-bit function code (LB/LBU/LW)
- **imm:** 12-bit immediate, often referred to as the **offset**

*Note: The term **base** comes from it serving as the base to which an offset (positive or negative) is applied to compute the memory address*

Load Instructions

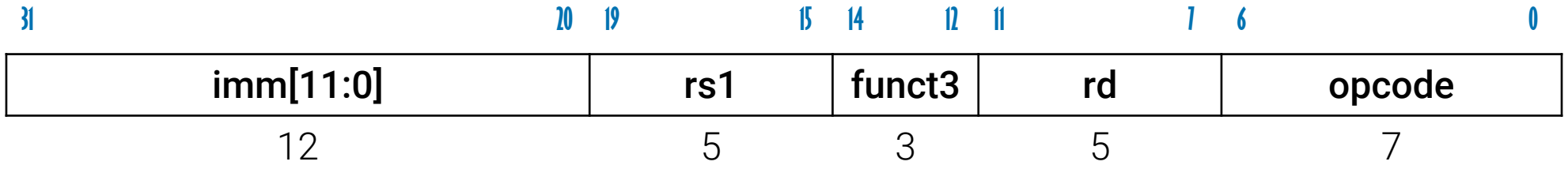
I-type Format



- `lw` instruction copies a 32-bit value from memory into the register RF[rd]
- `lb` copies a sign-extended 8-bit value from memory to the register RF[rd]
  - `lbu` is similar, except that it zero-extends instead

Load Instructions

Usage



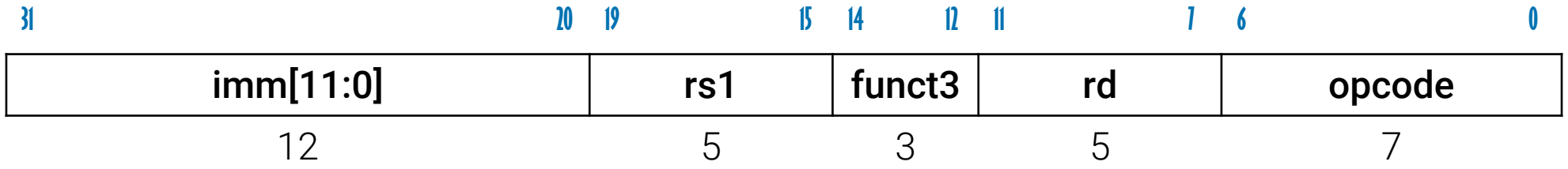
- In assembly (offset = imm):

Instruction	Operation
`lbu rd, offset(rs1)`	RF [rd] = zext (mem [RF [rs1] + sext(offset)] [**7** : 0])
`lb  rd, offset(rs1)`	RF [rd] = sext (mem [RF [rs1] + sext(offset)] [**7** : 0])
`lw  rd, offset(rs1)`	RF [rd] = mem [RF [rs1] + sext(offset)] [**31** : 0]

Note:
*- **sext( )** stands for sign extension*
*- **zext( )** stands for zero extension*

Load Instructions

Encoding

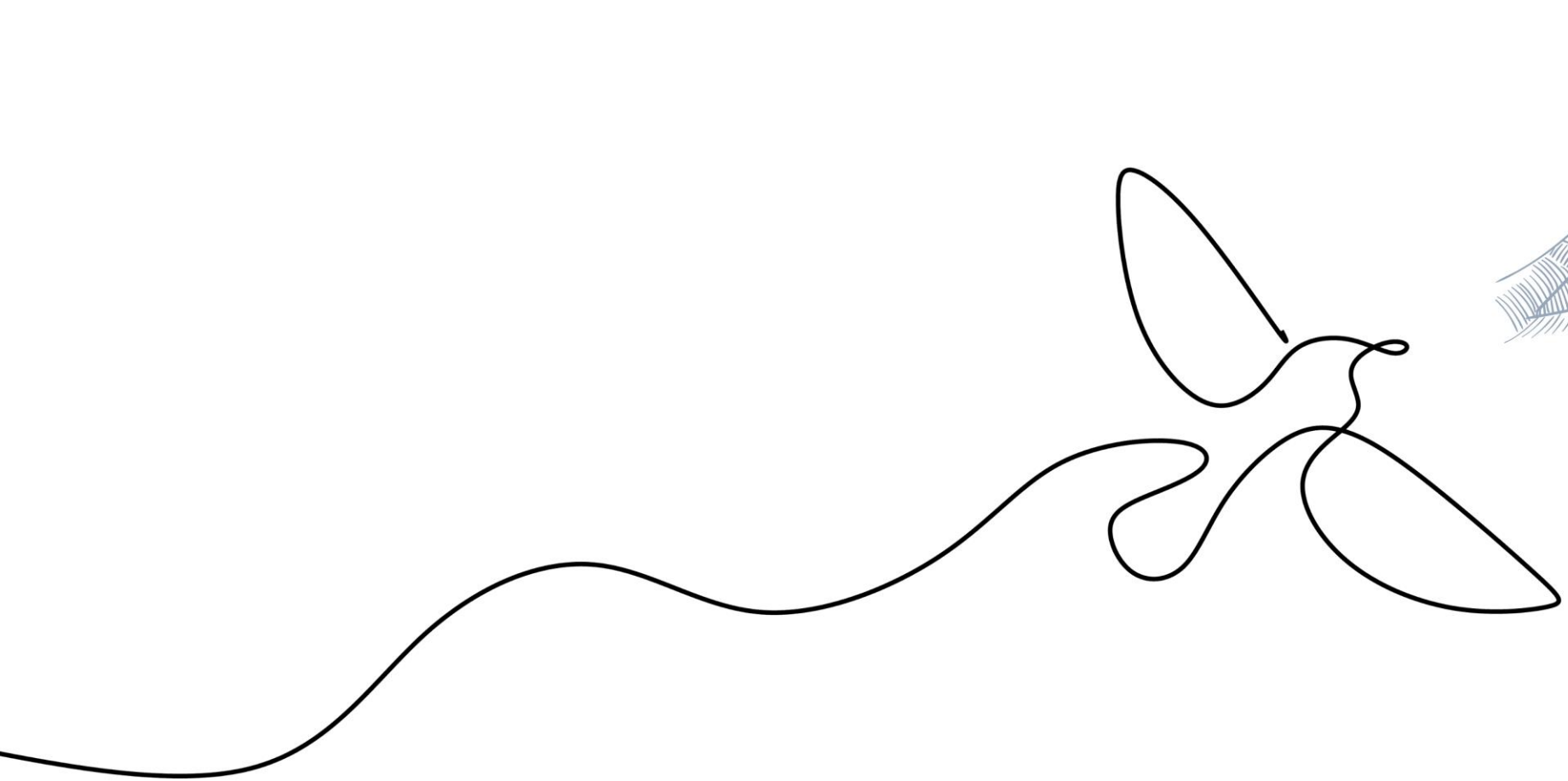


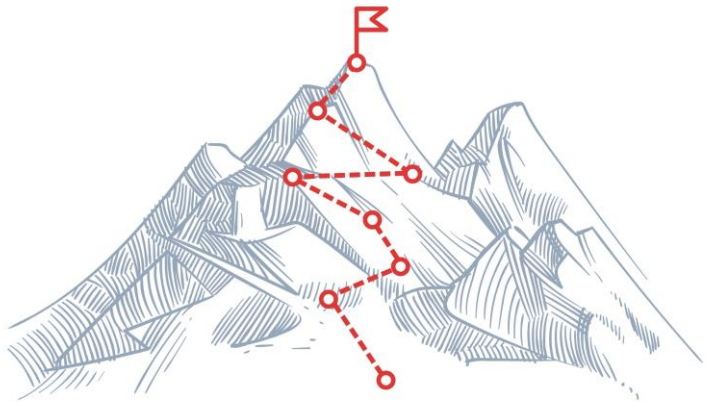
- Encoding

Source: The RISC-V Instruction Set Manual Volume I, ver. 20240411, page 554, Link

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

Note: Load half-word (LH, LHU) is out of scope for CS-173





Memory Reading

- Consider the snippet of memory shown at the right
- Write the instructions to load
  - One byte from memory address 0x200
  - One word from memory address 0x204
- Use `t0` as the destination register
- Little-endian byte ordering

  Unless specified otherwise, we will always assume little-endian byte ordering.

Address	Data
…	…
0x200	0xB3
0x201	0x52
0x202	0x73
0x203	0x00
0x204	0x26
0x205	0x38
0x206	0x3C
0x207	0x11
…	…

Memory Reading

Solution, Load Byte

Instruction	Operation
`lb rd, offset (rs1)`	RF [rd] = sext (mem [RF [rs1] + sext(offset)] [**7** : 0])

- Copy one byte from address **0x200** to register `t0`:

```
lb t0, 0x200(zero)
```

Register	Name	Description
x0	zero	Hard-wired zero

t0 = sext(mem[RF[x0] + 0x200][7:0]) =

= sext(mem[0 + 0x200][7:0]) = sext(mem[0x200][7:0]) =

= sext(0xB3) = sext(1011 0011) =

= 0xFFFF FFB3

Therefore, **t0 = 0xFFFF FFB3.**

Address	Data
...	...
0x200	0xB3
0x201	0x52
0x202	0x73
0x203	0x00
0x204	0x26
0x205	0x38
0x206	0x3C
0x207	0x11
...	...

Memory Reading

Solution, Load Word, Little Endian

Instruction	Operation
`lw rd, offset(rs1)`	RF [rd] = mem [RF [rs1] + sext(offset)] [31 : 0]

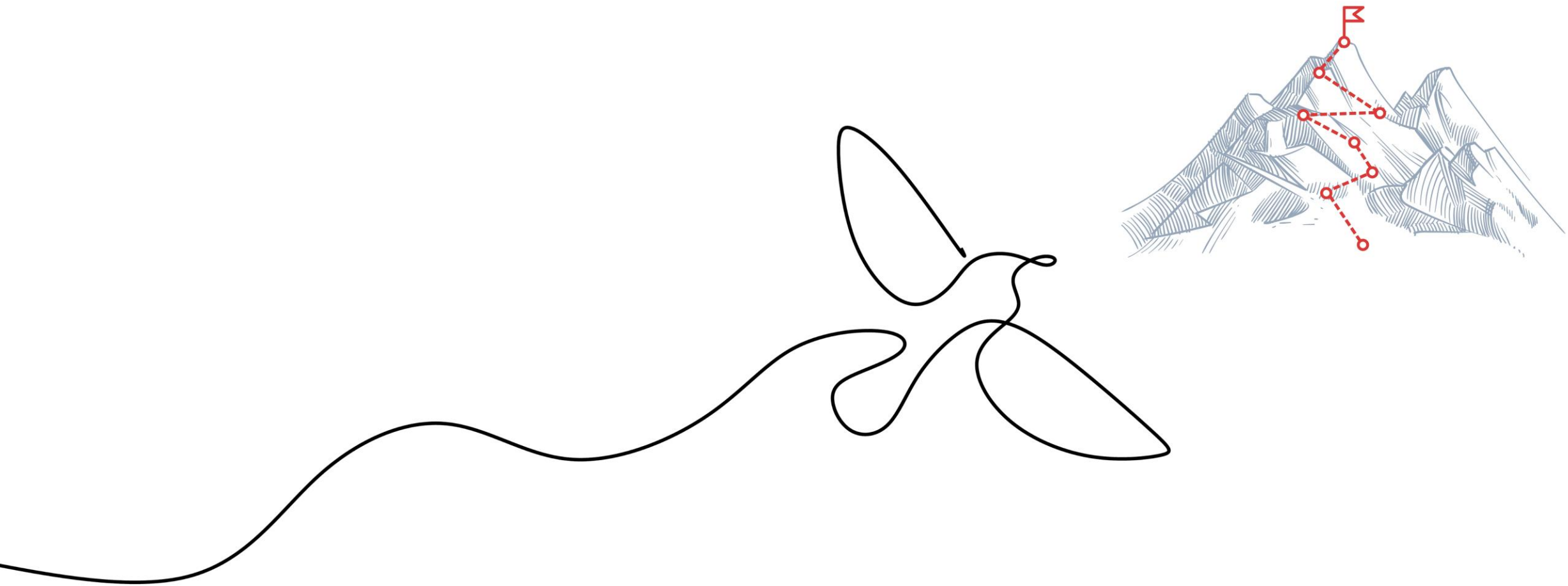
- Copy one word from address **0x204** to register `t0`:

```
lw t0, 0x204(zero)
```

t0 = mem[RF[x0] + 0x204][31:0] =
= mem[0 + 0x204] [31:0]= mem[0x204][31:0]

Conforming to the little-endian byte ordering, **t0 = 0x113C3826**.

Address	Data
...	...
0x200	0xB3
0x201	0x52
0x202	0x73
0x203	0x00
0x204	0x26
0x205	0x38
0x206	0x3C
0x207	0x11
...	...



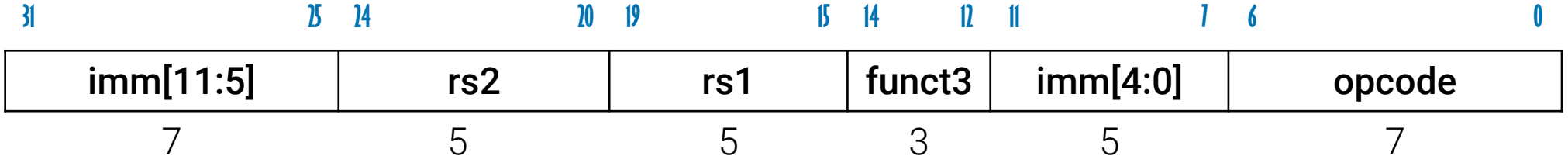
Recall: Memory Read and Write

Load and Store

- Memory **read**/**write** operations are called **load**/**store**
- RV32I provides instructions for **loading** signed and **u**nsigned
  - **b**ytes: `lb`, `lbu`
  - **w**ords: `lw`
- RV32I provides instructions for **storing** data in memory
  - **b**ytes: `sb`
  - **w**ords: `sw`

Store Instructions

S-type Format

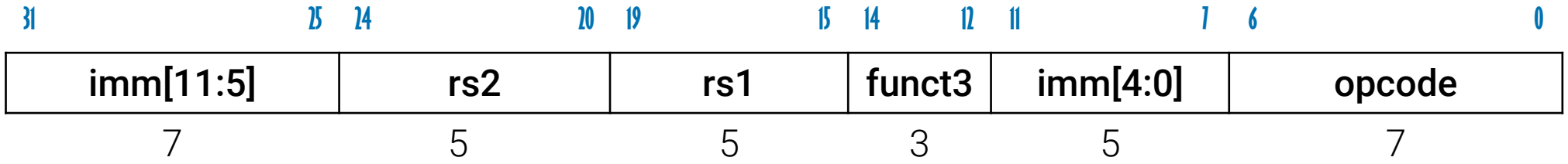


- Store: Copies data from **rs2** from the register file (RF) to the memory
- The memory address is obtained by **adding** **RF[rs1]** and the **sign-extended immediate**
- **rs1:** 5-bit index of the **base** register
- **rs2:** 5-bit index of the register receiving the value
- **opcode:** $(0100011)_2$, 7-bit operation code
- **funct3:** 3-bit function code (SB/SW)
- **imm:** 12-bit immediate, also called **offset**

*Note: The term **base** comes from it serving as the base to which an offset (positive or negative) is applied to compute the memory address*

Store Instructions

Usage



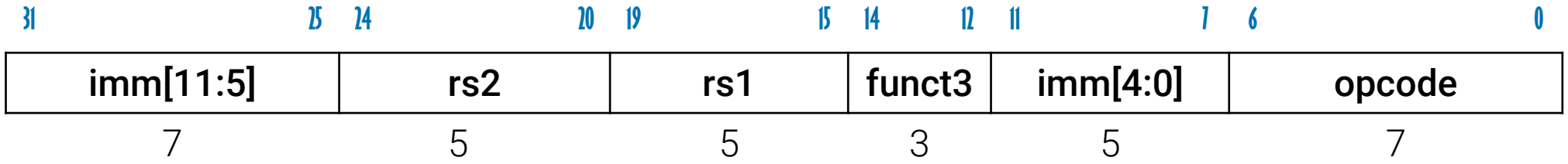
- In assembly (offset = imm)

Instruction	Operation
`sw rs2, offset(rs1)`	mem [RF [rs1] + sext(offset)] [**31** : 0] = RF [rs2]
`sb rs2, offset(rs1)`	mem [RF [rs1] + sext(offset)] = RF [rs2] [**7** : 0]

*Note: **sext( )** stands for sign extension*

Store Instructions

Encoding

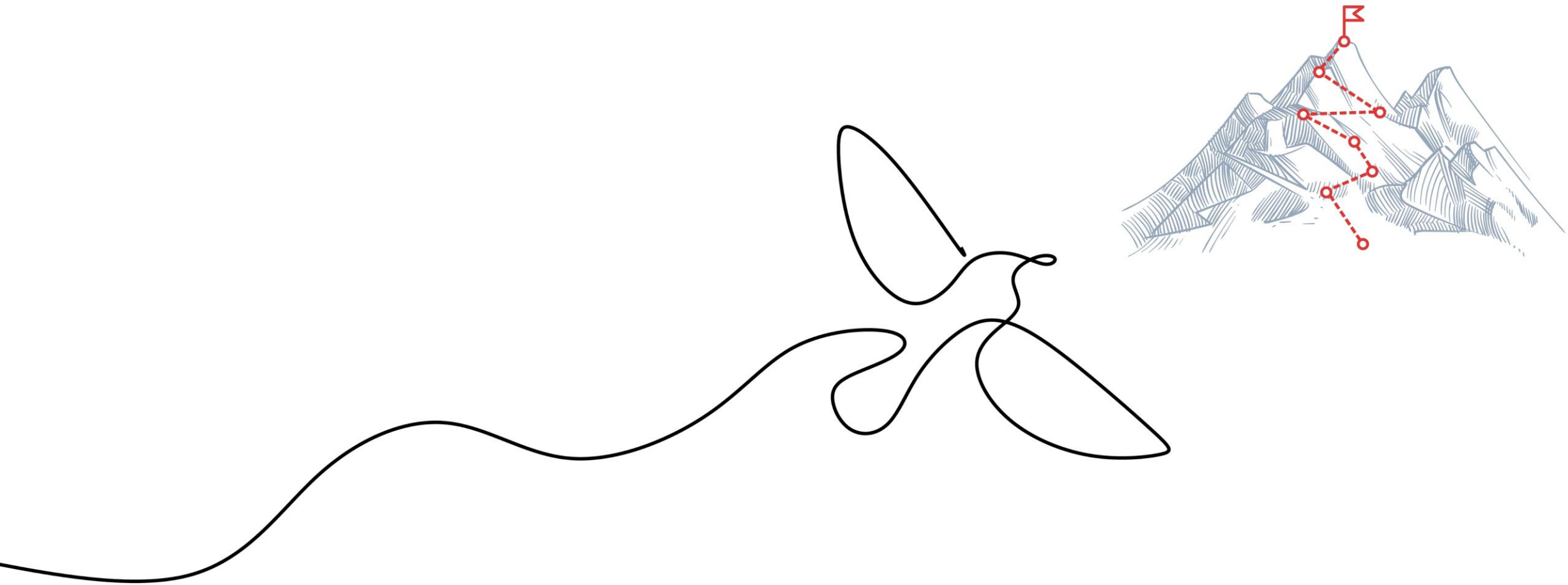


- Encoding

Source: The RISC-V Instruction Set Manual Volume I, ver. 20240411, page 554, Link

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Note: Store half-word (SH) is out of scope for CS-173



Memory Writing

- Consider the snippet of memory shown at the right
- Write the instructions to store (copy to memory)
  - One byte from register `t0` to address 0x201
  - The entire register `t0` to address 0x204
- Assume `t0 = 0x12052025`
- Little-endian byte ordering

  Unless specified otherwise, we will always assume little-endian byte ordering.

Address	Data
…	…
0x200	
0x201	
0x202	
0x203	
0x204	
0x205	
0x206	
0x207	
…	…

Memory Writing

Solution, Store Byte

Instruction	Operation
`sb rs2, offset(rs1)`	mem [RF [rs1] + sext(offset)] = RF [rs2] [**7** : 0]

EXAMPLES

- Copy one byte from register `t0` to address **0x201**;
  `t0 = 0x12052025`
- Solution:

```
sb t0, 0x201(zero)
```

mem[RF[x0] + sext(0x201)] = RF[t0][7:0]

Therefore, **mem[0x201] = 0x25**

Address	Data
...	...
0x200	
0x201	0x25
0x202	
0x203	
0x204	
0x205	
0x206	
0x207	
...	...

Memory Writing

Solution, Store Word, Little Endian

Instruction	Operation
`sw rs2, offset(rs1)`	mem [RF [rs1] + sext(offset)][31:0] = RF [rs2]

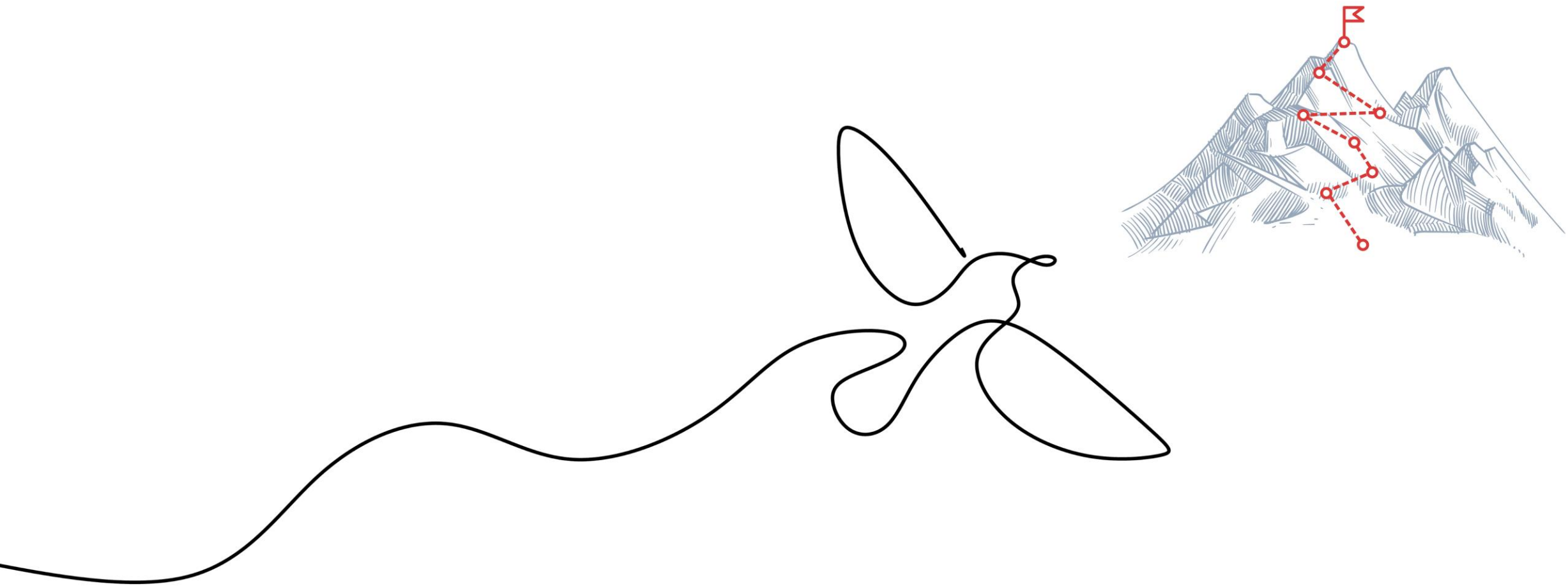
- Copy the entire register `t0` to address **0x204**;
  `t0 = 0x12052025`
- Solution:

```
sw t0, 0x204(zero)
```

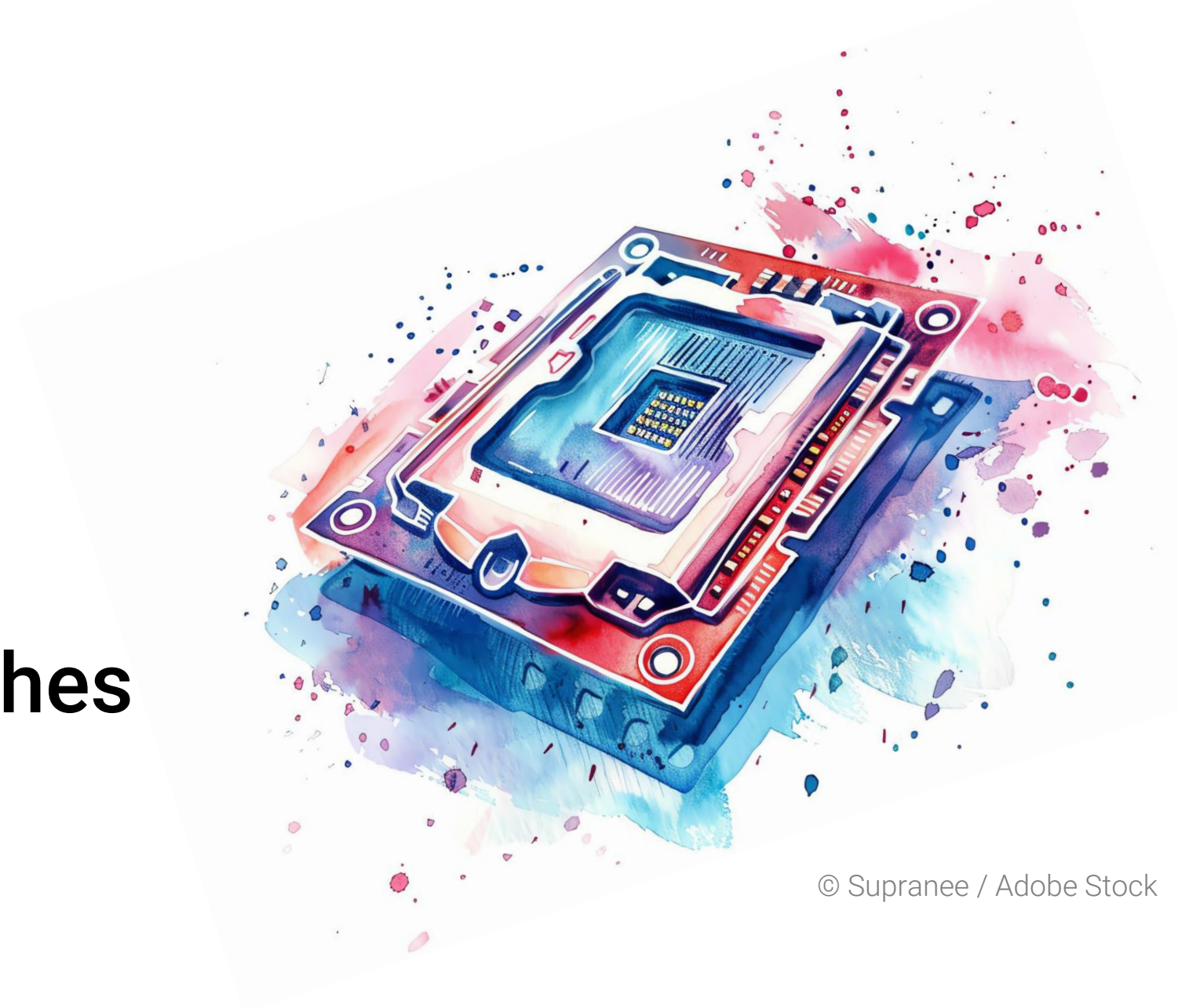
mem[RF[x0] + sext(0x204)][31:0] = RF[t0]

Therefore, **mem[0x204] = 0x12052025**

Address	Data
...	...
0x200	
0x201	
0x202	
0x203	
0x204	0x25
0x205	0x20
0x206	0x05
0x207	0x12
...	...



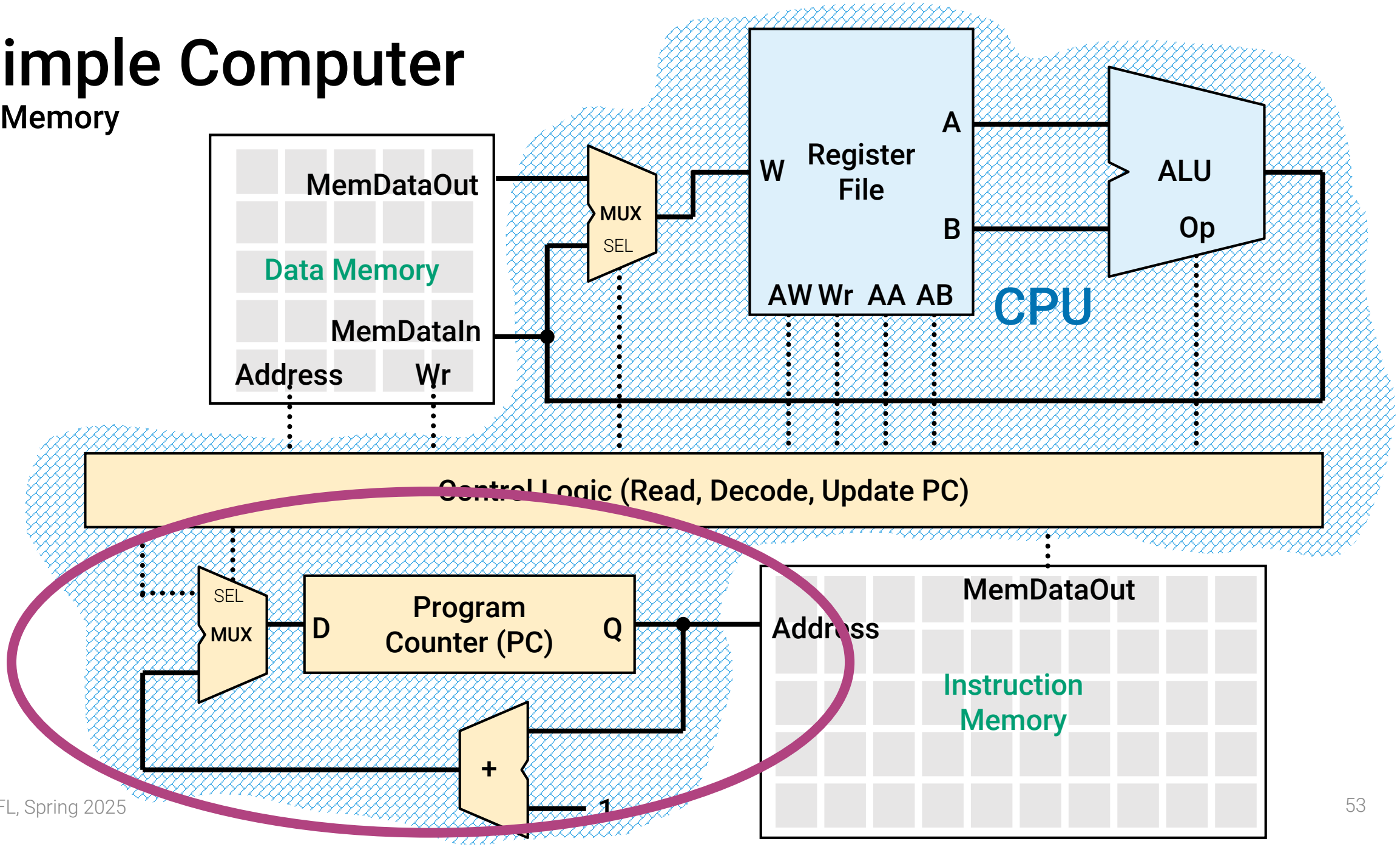
Conditional Branches



© Supranee / Adobe Stock

A Simple Computer

CPU + Memory



Conditional Branches

- RV32I supports comparing two registers and **b**ranching to a specific code line (instruction) if the register contents are
  - **eq**ual: `beq`
  - **n**ot **e**qual: `bne`
  - **g**reater than or **e**qual:
    - `bge` (signed comparison)
    - `bgeu` (**u**nsigned comparison)
  - **l**ess **t**han
    - `blt` (signed comparison)
    - `bltu` (**u**nsigned comparison)
- *Note: Other relationships can be checked by reversing the operands or by using pseudoinstructions*

Conditional Branches

Contd.

- Branches are typically used for `if-else` and loops (`while`, `for`)
- Loops are generally small (< 50 instructions)
- Largest branch distance is limited by the size of the code space in memory
  - Should not branch into the address space for data, variables, external input/output devices, etc.

Conditional Branches

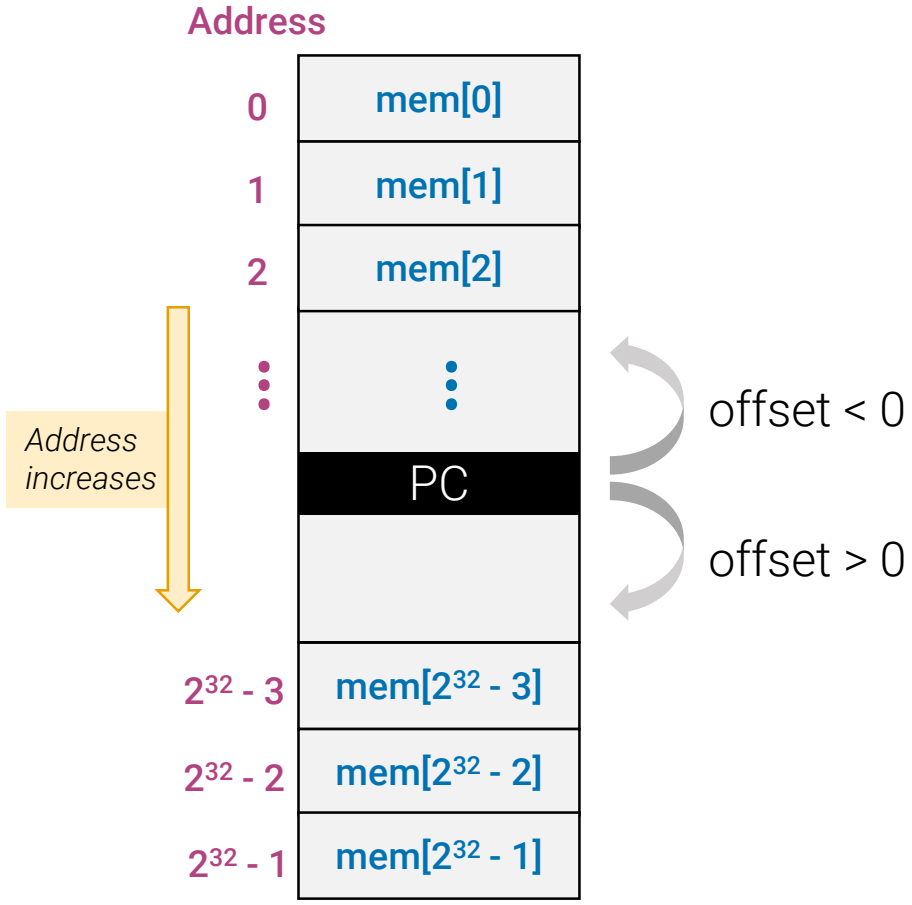
Contd.

- In RISC-V, branches use PC-relative addressing
  - Program Counter (PC) register acts as the base memory address
  - The immediate in two's complement, encoded within branch instruction, serves to compute the offset
  - The offset is added with the PC to find the next PC value

Conditional Branches

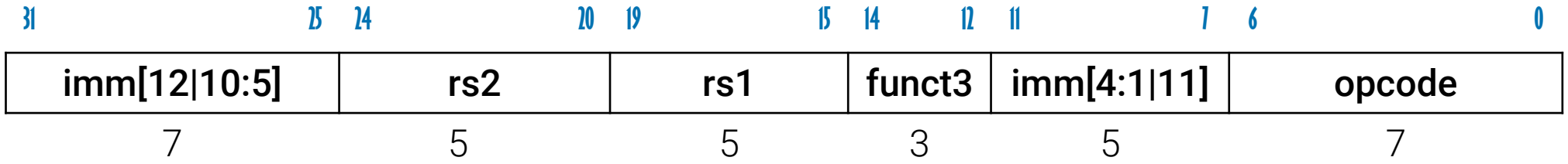
PC-Relative Addressing

- Updated PC, if branch is taken:
  - **PC = PC + offset**
  - **offset = imm << 1 = imm × 2**
  - multiplying the immediate by 2 doubles the branch address range
  - offset is sign-extended, can be positive or negative
- Otherwise:
  - **PC = PC + 4**



Branch Instructions

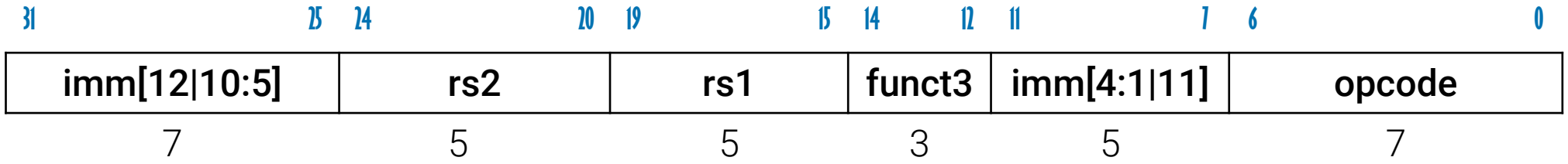
B-type Format



- **rs1:** 5-bit index of the first operand register
- **rs2:** 5-bit index of the second operand register
- **opcode:** $(1100011)_2$
- **funct3:** 3-bit function code for branch condition (BEQ/BNE/BLT/BGE/BLTU/BGEU)
- **imm:** 12-bit immediate. Note that imm[0] is not needed (because of the subsequent multiplication by two which guarantees the least significant bit is zero), which is why it is replaced by imm[11]. This encoding arrangement keeps imm[4:1] and imm[10:5] in their usual place, which simplifies hardware implementation.

Branch Instructions

Usage



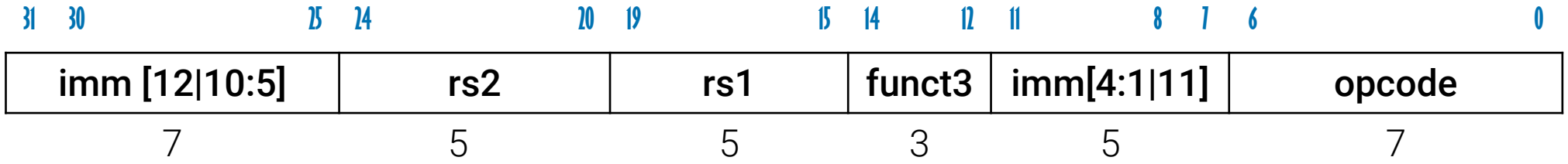
- In assembly (offset = imm × **2**)

Instruction	Operation
`beq rs1, rs2, imm`	if (rs1 == rs2) pc = pc + sext(offset)
`bne rs1, rs2, imm`	if (rs1 $\neq$ rs2) pc = pc + sext(offset)
`blt rs1, rs2, imm`	if (rs1 $<_s$ rs2) pc = pc + sext(offset)
`bge rs1, rs2, imm`	if (rs1 $\geq_s$ rs2) pc = pc + sext(offset)
`bltu rs1, rs2, imm`	if (rs1 $<_u$ rs2) pc = pc + sext(offset)
`bgeu rs1, rs2, imm`	if (rs1 $\geq_u$ rs2) pc = pc + sext(offset)

Note: ***sext( )*** *stands for sign extension;*
$<_s$ *and* $>_s$ *stand for signed comparisons;*
$<_u$ *and* $>_u$ *stand for unsigned comparisons;*
pc *stands for Program Counter;*

Branch Instructions

Encoding



- Encoding

Source: The RISC-V Instruction Set Manual Volume I, ver. 20240411, page 554, Link

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Conditional Branches

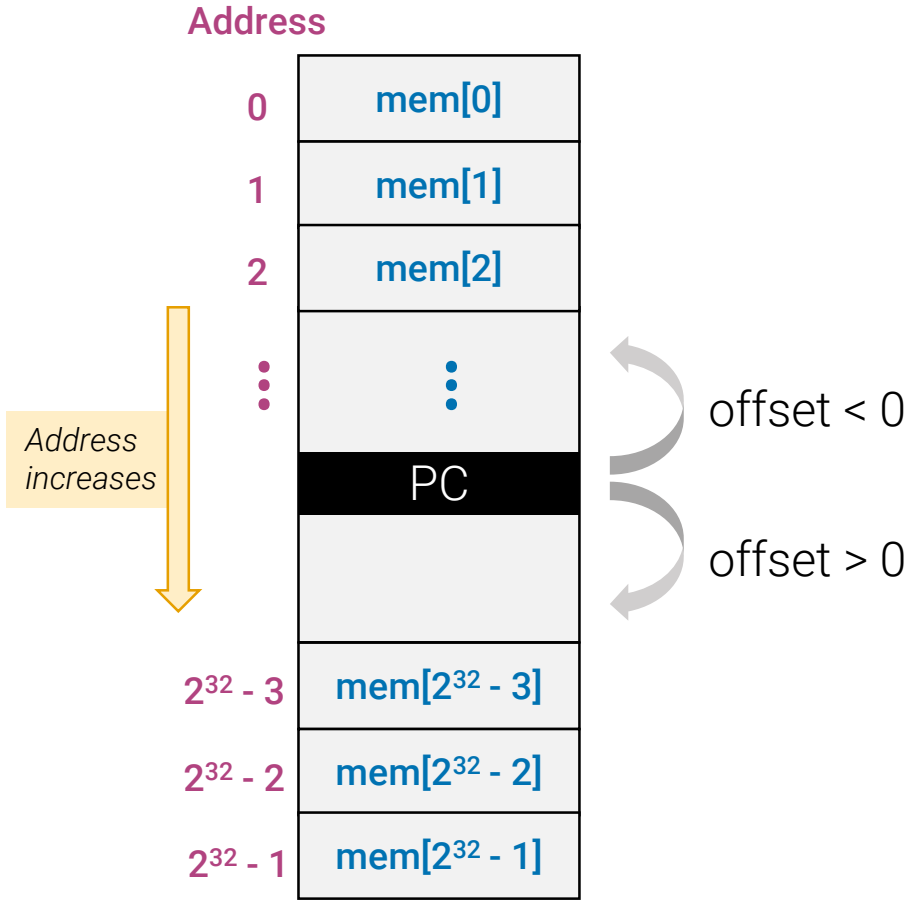
Branch Address Range

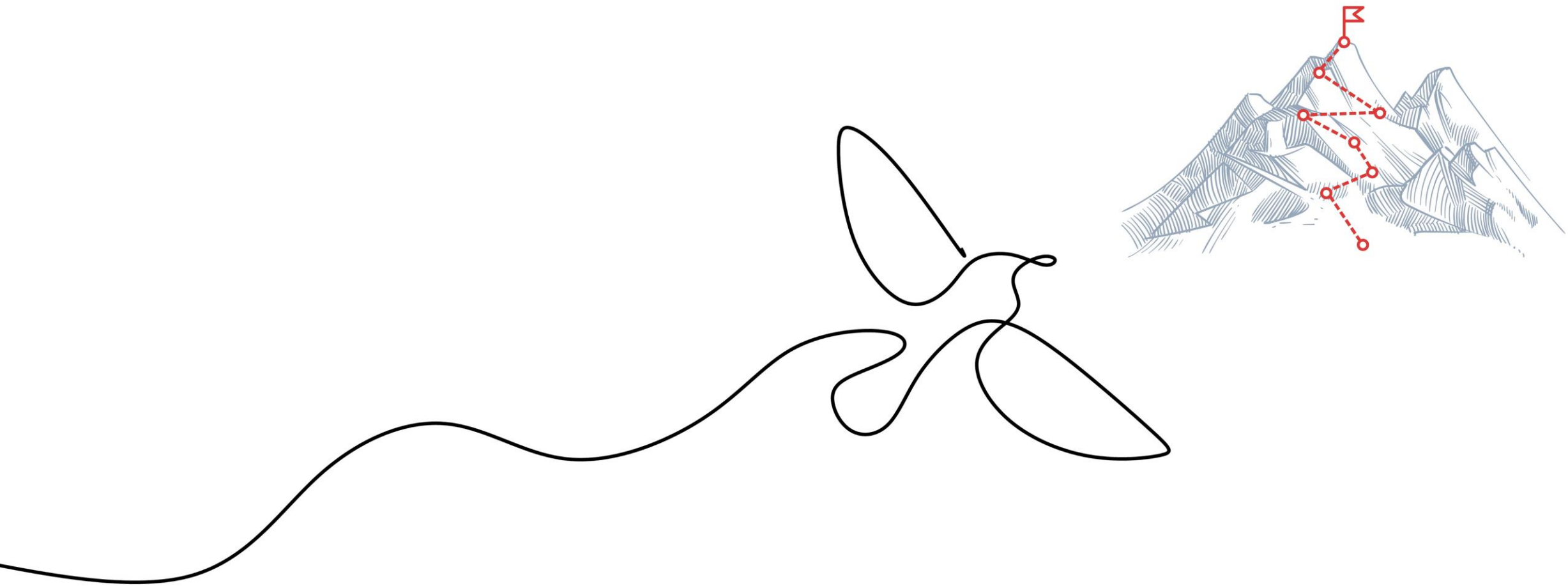
- *Recall:* Updated PC, if branch is taken:
  - **PC = PC + immediate × 2**
- Otherwise:
  - **PC = PC + 4**

Q: What is the branch address range?

A: ±4 kiB

- Offset is on 13 bits (12-bit immediate + one bit thanks to the multiplication by two); hence, the largest absolute offset value is 2^{12}, which amounts to 2^{10} instructions (32-bit words)
- Therefore, the **branch address range** is approx. ±4 kiB on either side of the PC





Find the Max of Three Registers

- Implement in RISC-V assembly the algorithm below, which finds the max of three values x, y, z:

```
max = x;
if (y > max) max = y;
if (z > max) max = z;
```

- Assume x, y, z are 32-bit values stored consecutively in memory, starting from the **address in register t3**
- Use registers t0, t1, t2, and t4 for variables x, y, z, and max
- Copy the value of max to memory at the address immediately following the variable z

Note: For simplicity, you can assume that x, y, and z are already loaded

Address	Data
…	…
from t3 to t3 + 3	x
from t3 + 4 to t3 + 7	y
from t3 + 8 to t3 + 11	z
from t3 + 12 to t3 + 15	max
…	…

mv Pseudoinstruction

- In RV32I, there is a pseudo instruction called **mv**, for elegantly copying contents of one register to another
- `mv rd, rs` is simply an alias for `addi rd, rs, 0`
- We shall use **mv** pseudo instruction in this example

Find the Max of Three Registers

Solution

```
max = x;
if (y > max) max = y;
if (z > max) max = z;

x,  y,  z,  max ->
t0, t1, t2, t4
```

```
        mv   t4, t0                    # 1: Assume max = x initially
        blt  t4, t1, set_max_y         # 2: if max < y, update max = y
check_z:
        blt  t4, t2, set_max_z         # 3: if max < z, update max = z
store_max:
        sw   t4, 12(t3)                # 4: store final max to memory
        beq  zero, zero, program_end   # 5: unconditional branch
set_max_y:
        mv   t4, t1                    # 6: max = y
        beq  zero, zero, check_z       # 7: unconditional branch
set_max_z:
        mv   t4, t2                    # 8: max = z
        beq  zero, zero, store_max     # 9: unconditional branch
program_end:
        nop                            # 10: program end
```

Register	Name	Description
x0	zero	Hard-wired zero

Instruction Execution Sequence

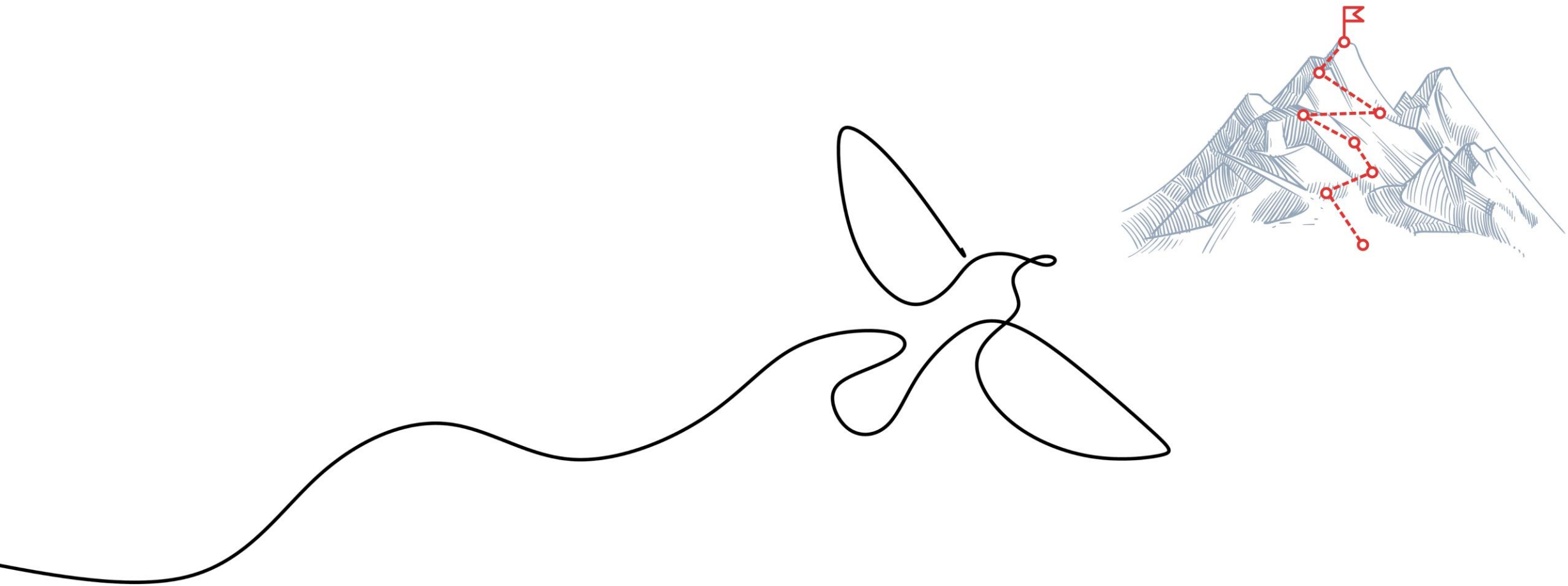
- **Q:** What is the maximum number of instructions this program executes, and under which conditions that happens?

- **A:** Longest code execution happens when all checks need to be performed, i.e., when x < y < z.
- The corresponding number of executed instructions is 10.

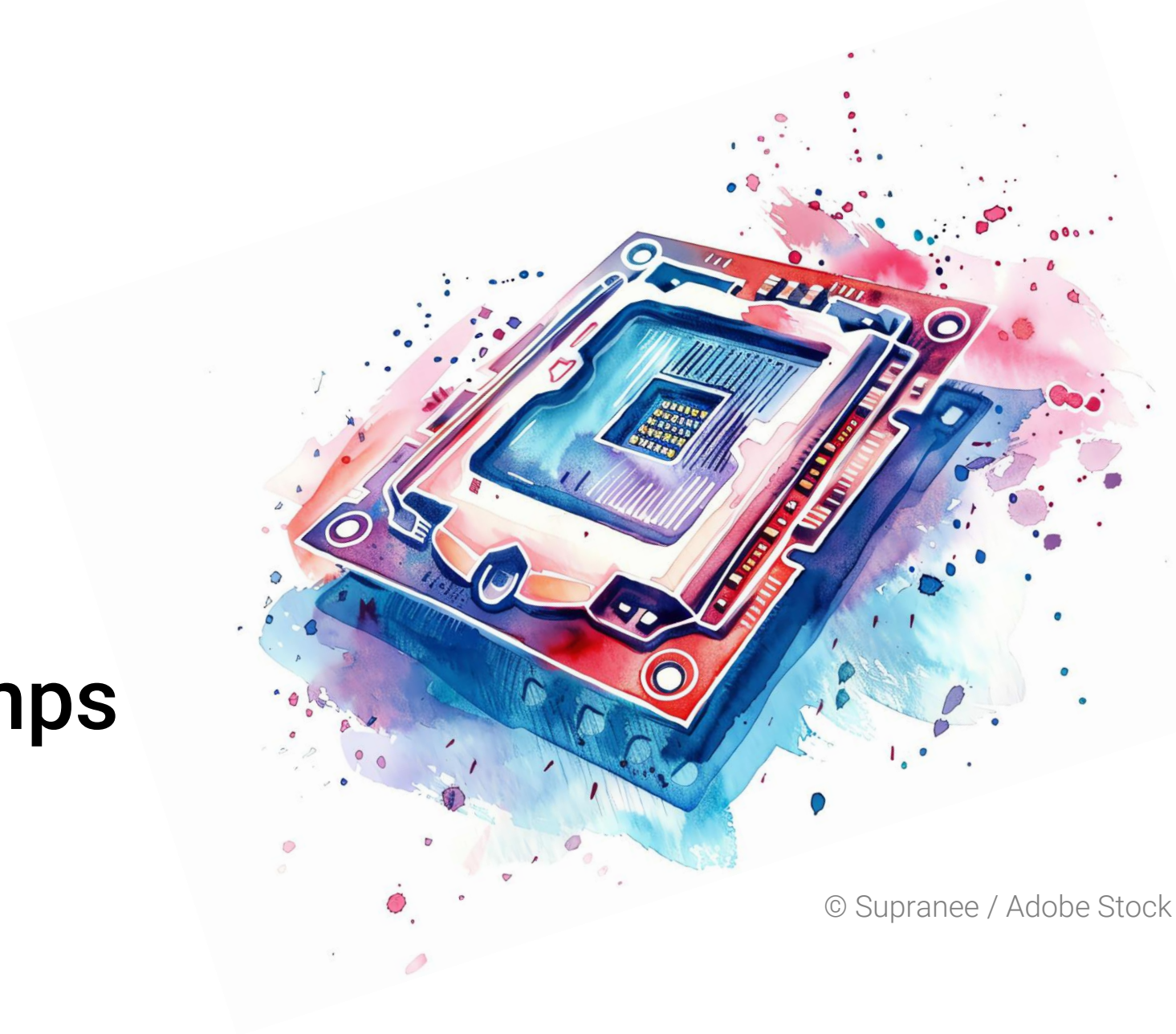
Instruction Execution Sequence

- **Q:** Assuming x > y > z, how many instructions does this program execute?
- **A:** Six instructions (see below)

```
        mv   t4, t0                 # Assume max = x initially
        blt  t4, t1, set_max_y      # if max < y, update max = y
check_z:
        blt  t4, t2, set_max_z      # if max < z, update max = z
store_max:
        sw   t4, 12(t3)
        beq  zero, zero, program_end
program_end:
        nop
```



Unconditional Jumps



© Supranee / Adobe Stock

Jump Instructions

- Jump instructions are used for
  - unconditional branches
  - calling functions
  - returning from functions
- Two supported instructions:
  - **j**ump **a**nd **l**ink: `jal`
  - **j**ump **a**nd **l**ink **r**egister: `jalr`

  Out of scope for CS-173
- **J-type** format

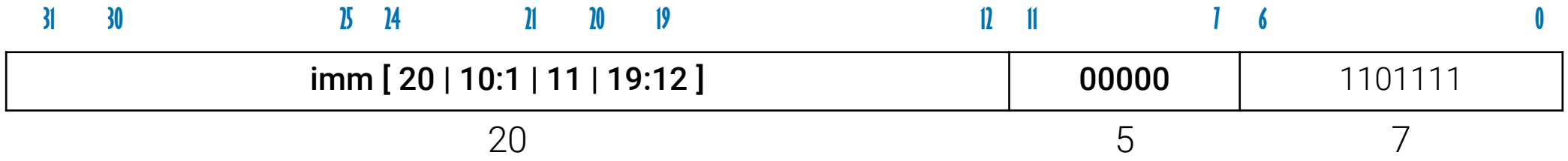
Difference between a jump and branch functionality is subtle and out of scope for CS-173.

J: Unconditional Jump

- Pseudoinstruction to elegantly perform an unconditional jump to a program label. Usage:

Instruction	Operation
`j imm`	pc = pc + sext(offset) offset = imm × 2

- Encoding:



Recall: Find the Max of Three Registers

- Implement in RISC-V assembly the algorithm below, which finds the max of three values x, y, z:

```
max = x;
if (y > max) max = y;
if (z > max) max = z;
```

- Assume x, y, z are 32-bit values stored consecutively in memory, starting from the **address in register t3**
- Use registers t0, t1, t2, and t4 for variables x, y, z, and max
- Copy the value of max to memory at the address immediately following the variable z
- **Use j instead of unconditional branches**

Note: For simplicity, you can assume that x, y, and z are already loaded

Address	Data
…	…
from t3 to t3 + 3	x
from t3 + 4 to t3 + 7	y
from t3 + 8 to t3 + 11	z
from t3 + 12 to t3 + 15	max
…	…

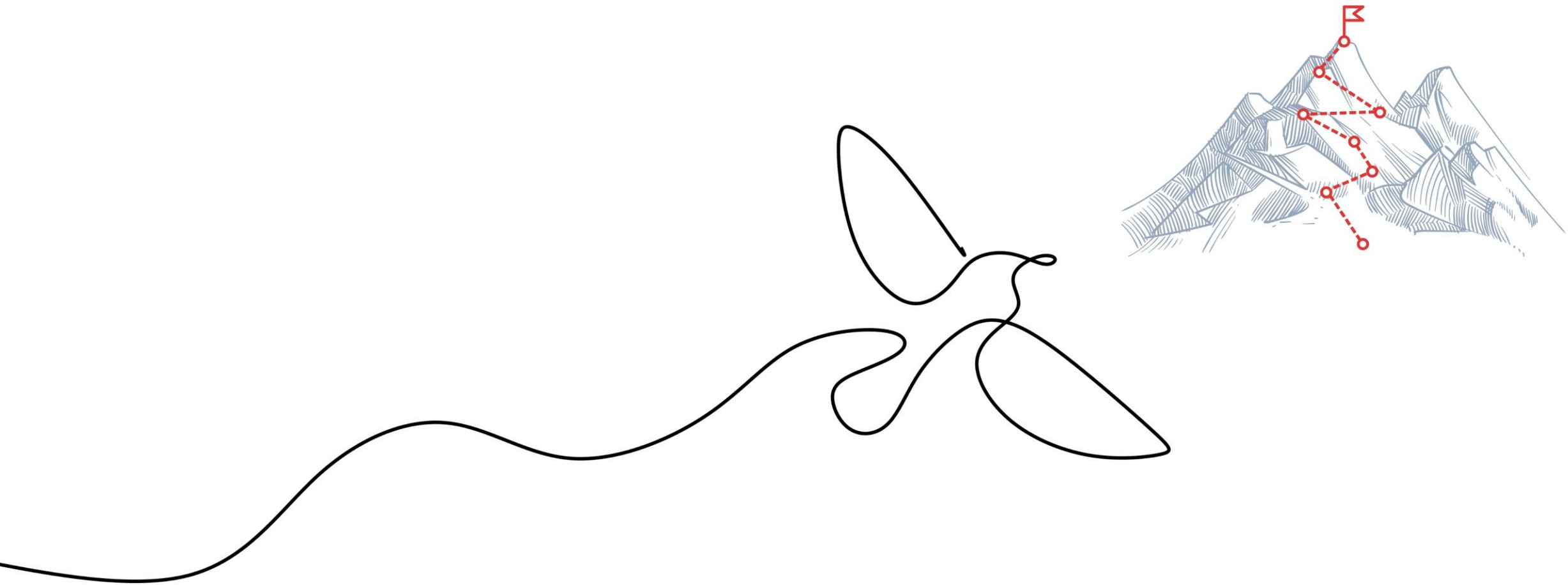
Find the Max of Three Registers

Solution, Use of j helps readability

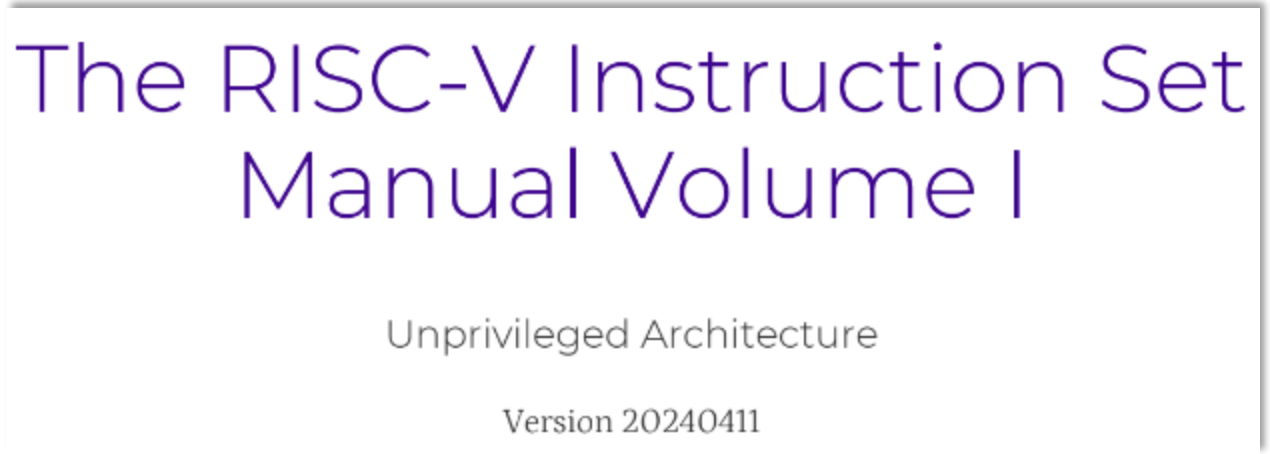
```
max = x;
if (y > max) max = y;
if (z > max) max = z;

x,  y,  z,  max ->
t0, t1, t2, t4
```

```
        mv   t4, t0                 # 1: Assume max = x initially
        blt  t4, t1, set_max_y      # 2: if max < y, update max = y
check_z:
        blt  t4, t2, set_max_z      # 3: if max < z, update max = z
store_max:
        sw   t4, 12(t3)             # 4: store final max to memory
        j    program_end            # 5: unconditional jump
set_max_y:
        mv   t4, t1                 # 6: max = y
        j    check_z                # 7: unconditional jump
set_max_z:
        mv   t4, t2                 # 8: max = z
        j    store_max              # 9: unconditional jump
program_end:
        nop                         # 10: program end
```

EXAMPLES



Literature



Visit online: Link