

CSE261: Computer Architecture

2. Instruction Set Architecture (1)

Seongil Wi

Class on 9/24: Online Class



- There will be **NO** offline class on September 24
- Instead, a recorded lecture video will be provided
 - This video will be available **ONLY** on September 24

09/17/2024 No Class: Chuseok

09/24/2024

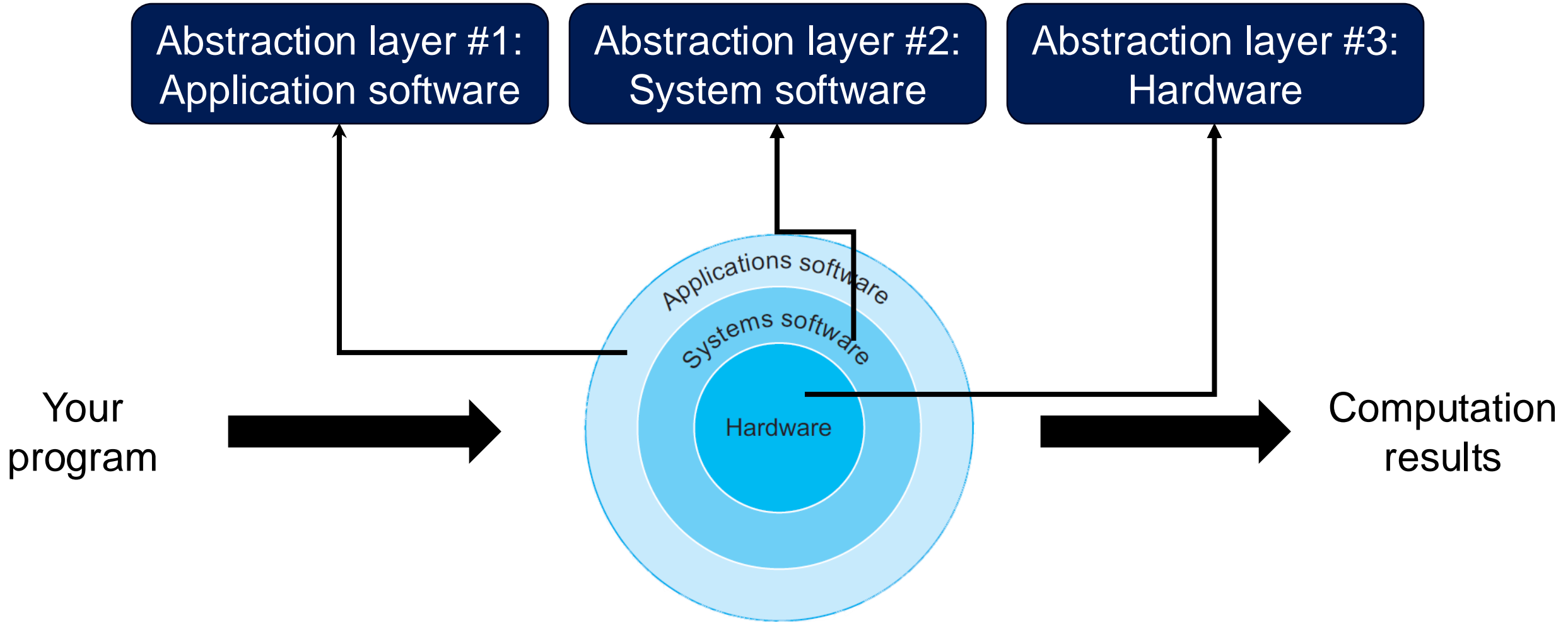
Online class (No offline class)
A recorded lecture video will be
provided

09/26/2024

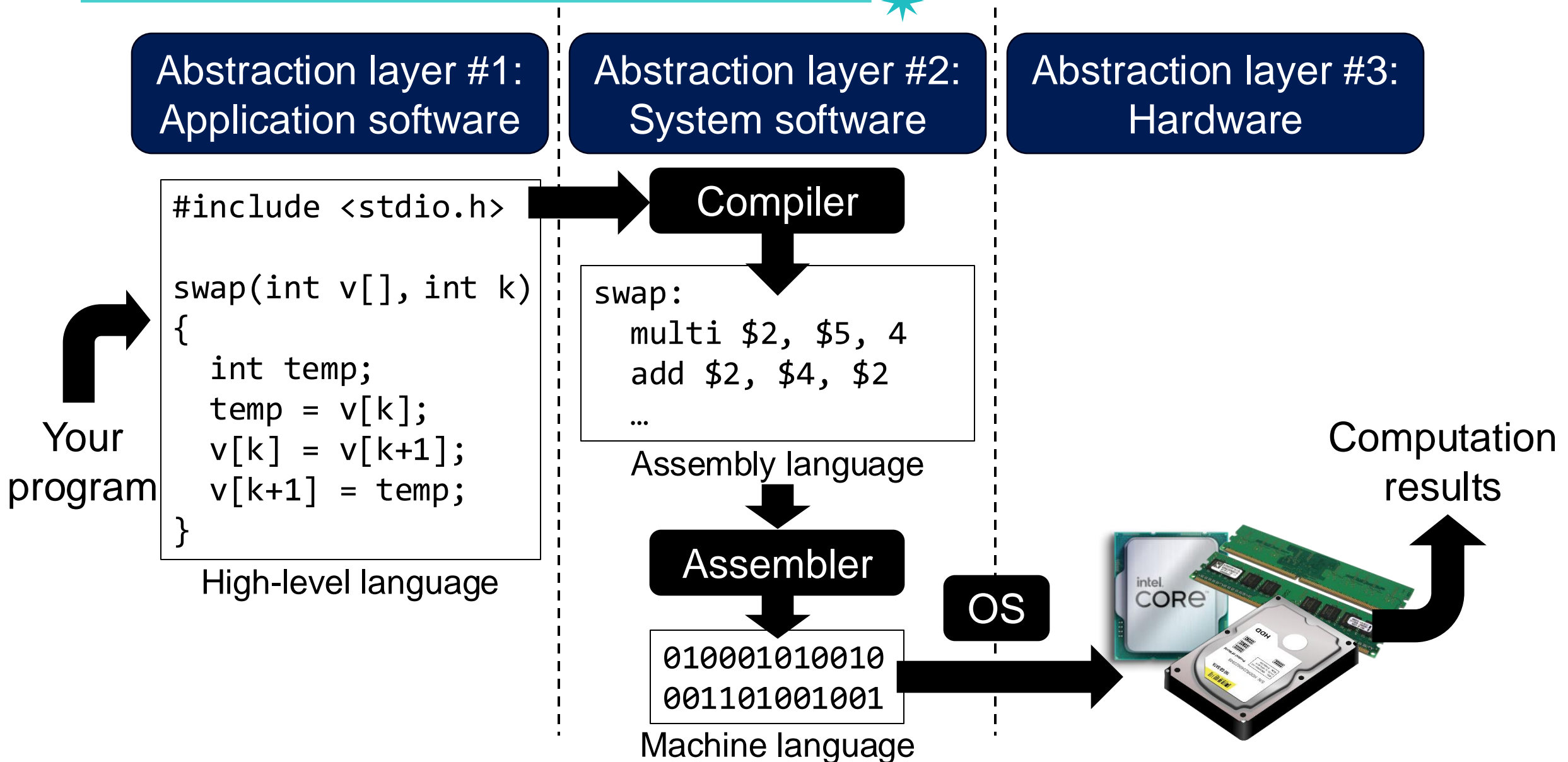
10/01/2024 No Class: Armed Forces Day

10/03/2024 No Class: National Foundation Day

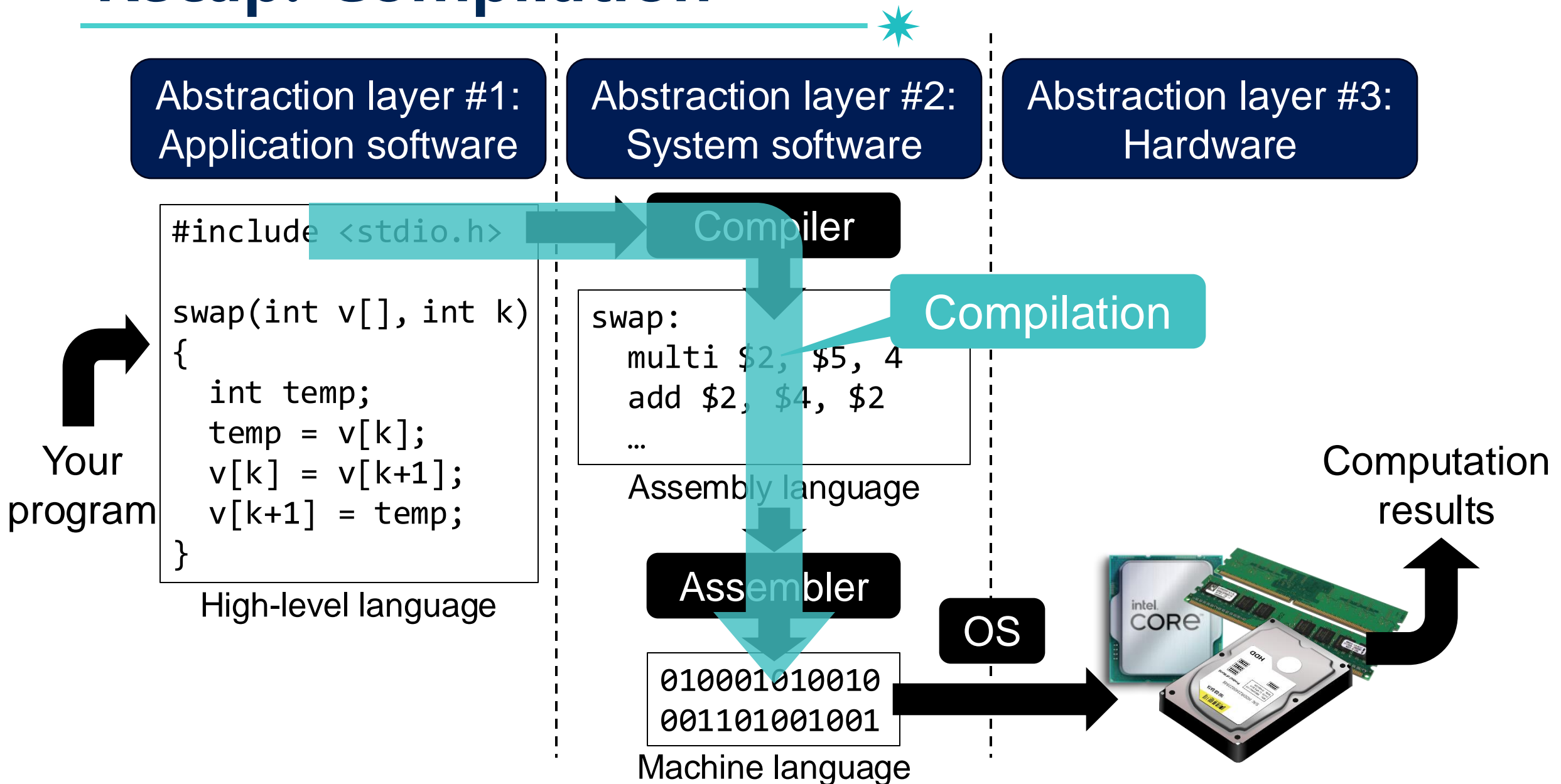
Recap: Computer Abstractions



Recap: Computer Abstractions



Recap: Compilation



Recap: The Hardware/Software Interface

6

Abstraction layer #1:
Application software

Abstraction layer #2:
System software

Abstraction layer #3:
Hardware

Detailed knowledge of
HW is not necessary

Compiler

Instruction Set
Architecture

Interface

Program/System/OS/Compiler
developer

```
010001010010
001101001001
```

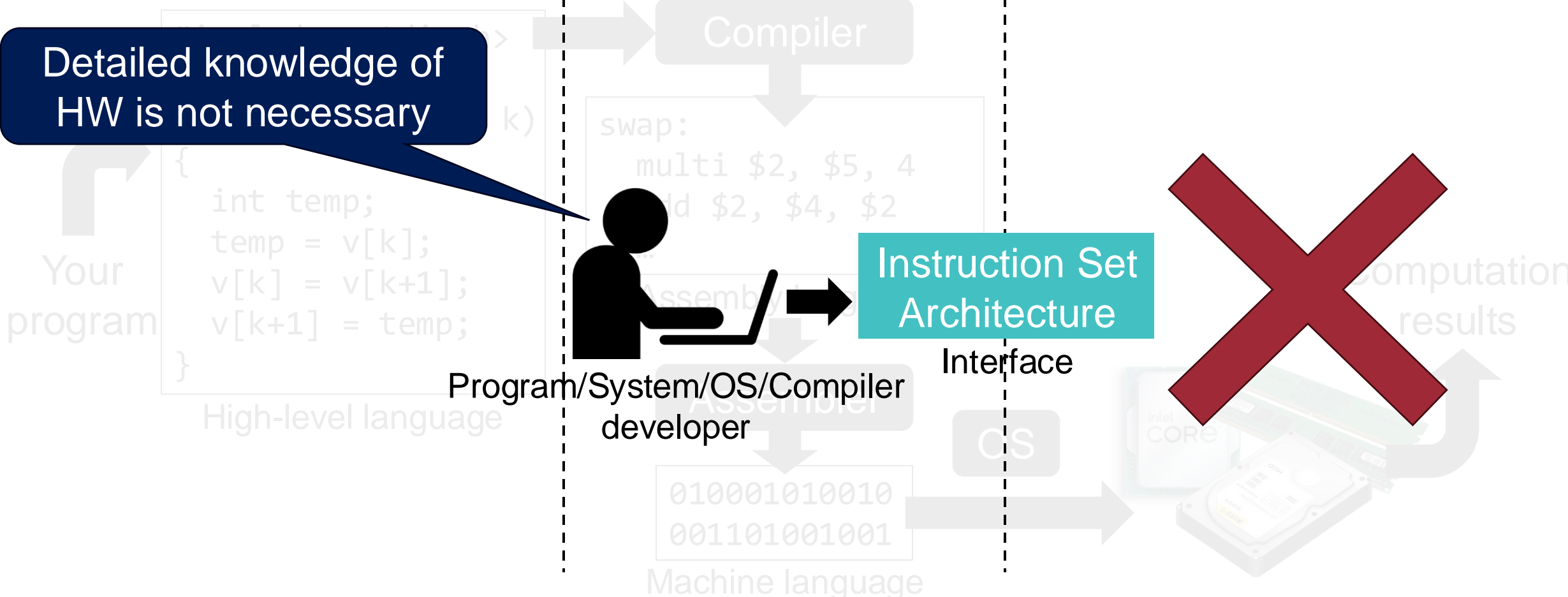
Machine language

CS

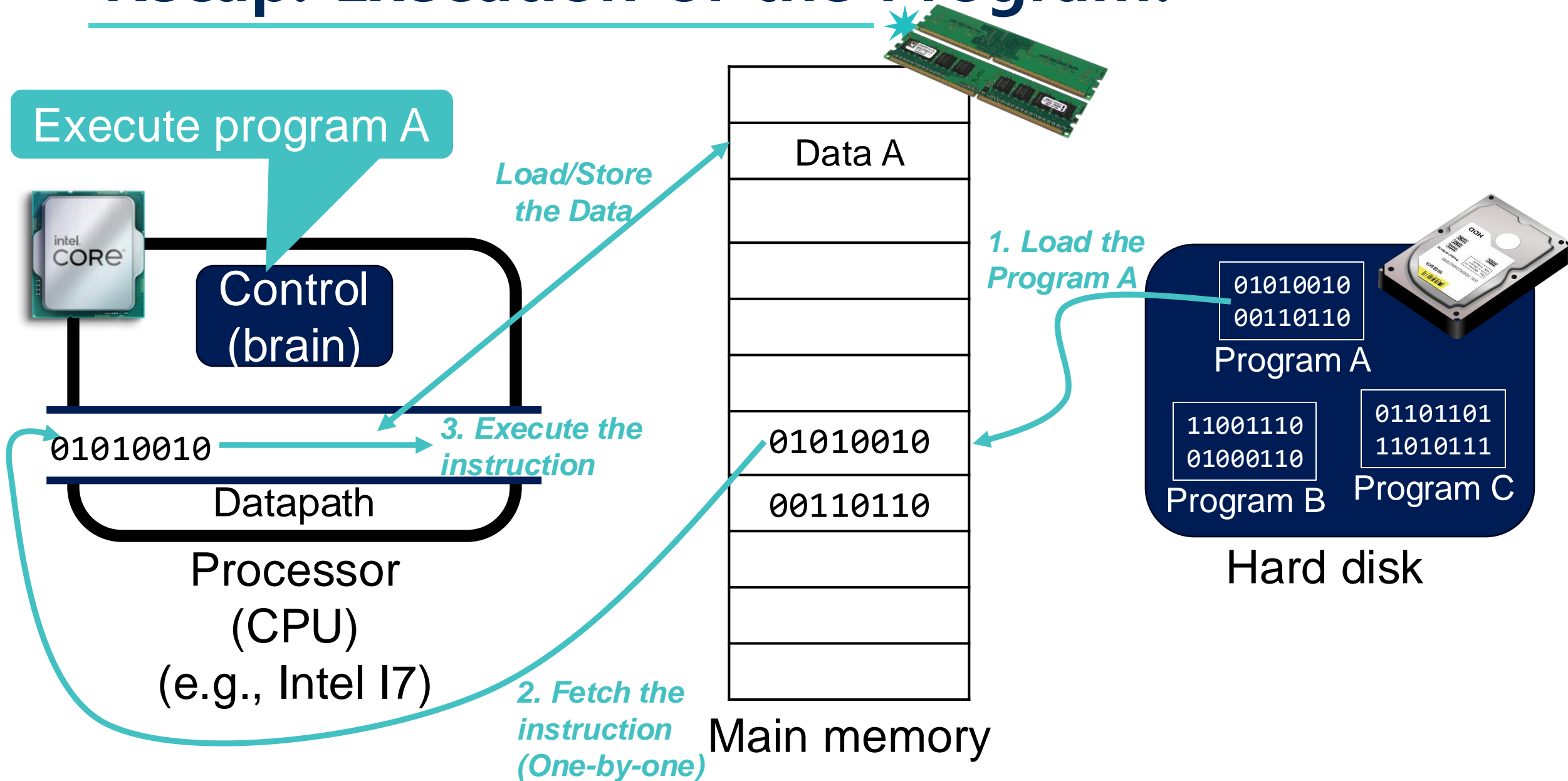
intel
CORE



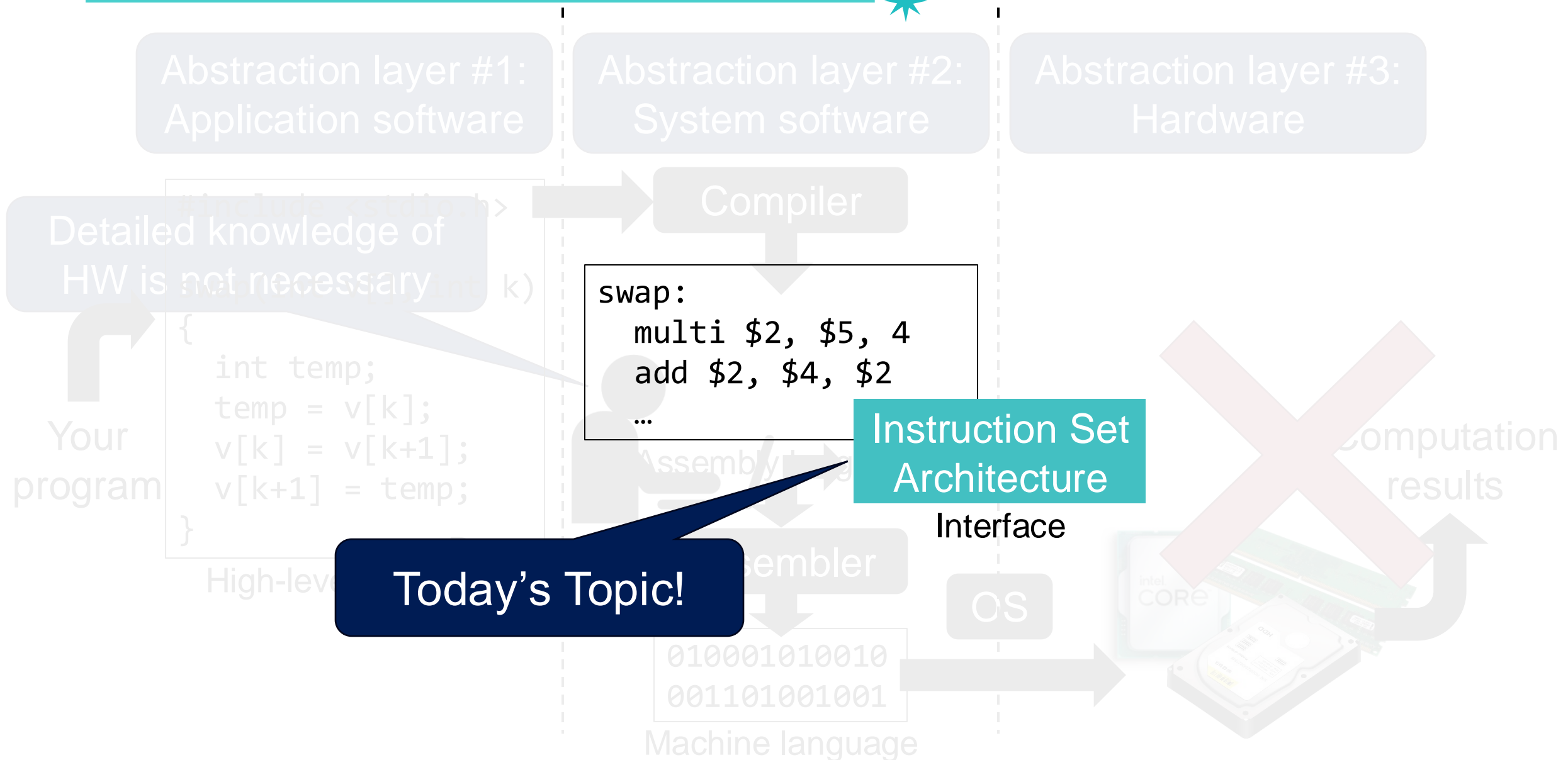
Computation
results



Recap: Execution of the Program!



Today's Topic: Instruction Set Architecture ⁸



Instruction Set Architecture

Instruction Set Architecture (ISA)



- An abstract interface between the hardware and the lowest-level software (called as **Architecture**)
- ISA includes:
 - Instruction set

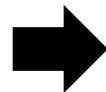
Instruction



- A command that hardware (i.e., CPU) understands
- A group of bits that tells the computer to perform a specific operation

```
slt $t0, $s0, $s1  
add $s2, $s0, $s1  
sub $t2, $s1, $zero  
lw  $t0, 8($s3)
```

Assembly language



Assembler



```
010001010100011  
101010001010100  
010001010101100  
000101011110011
```

Machine language

Instruction

- A command that hardware (i.e., CPU) understands
- A group of bits that tells the computer to perform a specific operation

Mapped to a group of bits

```
slt $t0, $s0, $s1  
add $s2, $s0, $s1  
sub $t2, $s1, $zero  
lw  $t0, 8($s3)
```

Assembly language

Assembler

```
010001010100011  
101010001010100  
010001010101100  
000101011110011
```

Machine language

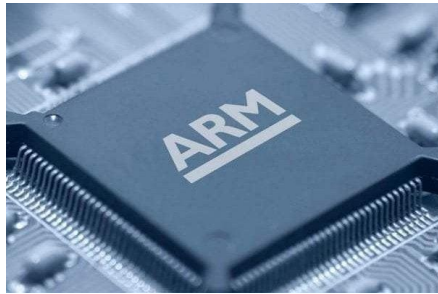
Decodes the bits to understand and perform operations



Instruction Set



- The commands understood by a given architecture



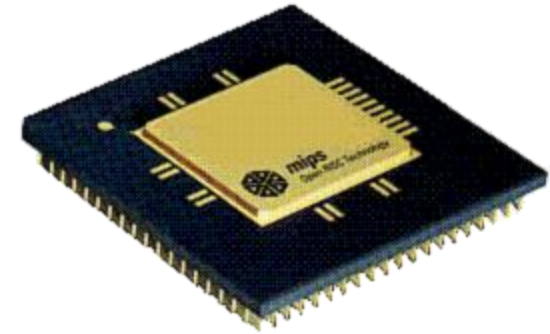
ARM's
instructions

```
pop {r0}  
mov r0, r1  
add r0, r0, r1  
add r0, #16
```



Intel's
Instructions

```
pop eax  
mov eax, ebx  
add eax, ebx  
add eax, 0x10
```



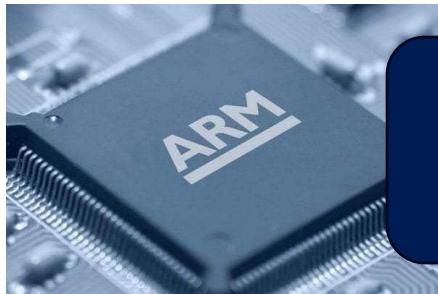
MIPS's
Instructions

```
slt $t0, $s0, $s1  
add $s2, $s0, $s1  
sub $t2, $s1, $zero  
lw $t0, 8($s3)
```

Instruction Set

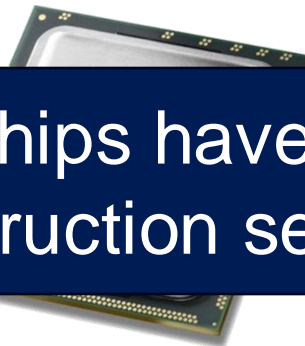


- The commands understood by a given architecture



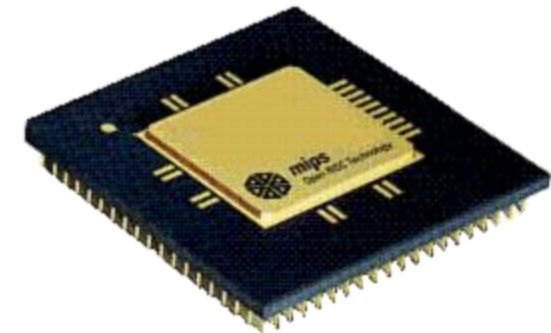
ARM's
instructions

```
pop {r0}  
mov r0, r1  
add r0, r0, r1  
add r0, #16
```



Intel's
Instructions

```
pop eax  
mov eax, ebx  
add eax, ebx  
add eax, 0x10
```



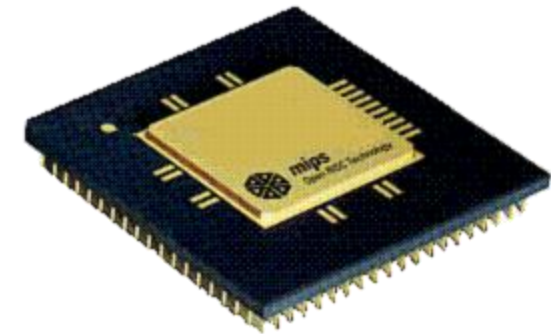
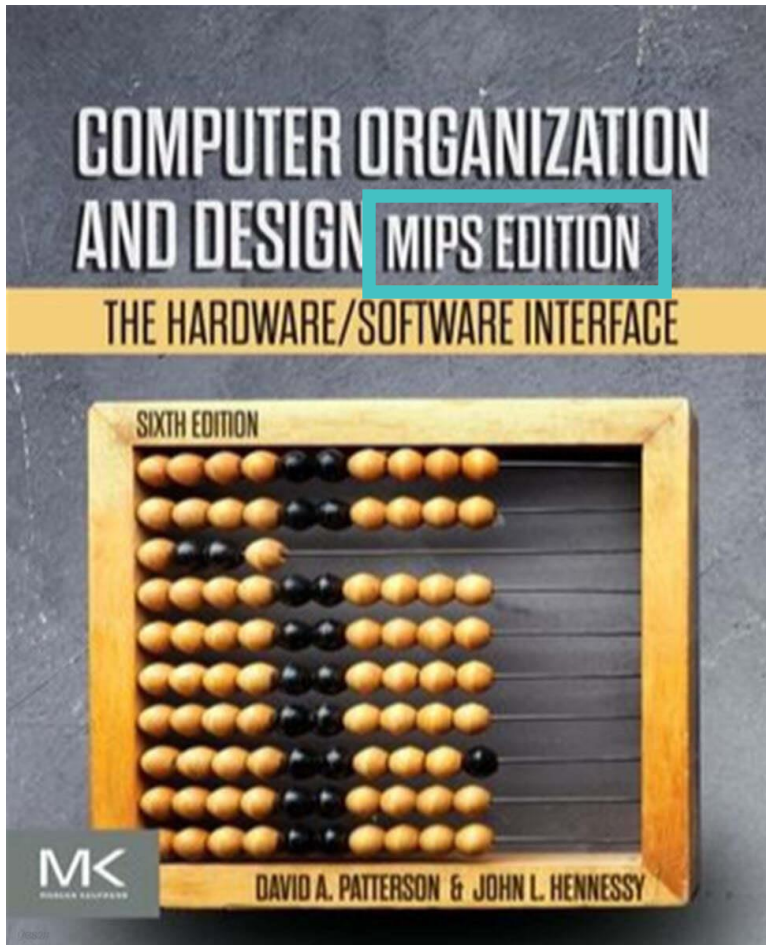
MIPS's
Instructions

```
slt $t0, $s0, $s1  
add $s2, $s0, $s1  
sub $t2, $s1, $zero  
lw $t0, 8($s3)
```

Instruction Set

- The commands understood by a given architecture

We focus on the MIPS instruction set!



MIPS's
Instructions

```
slt $t0, $s0, $s1  
add $s2, $s0, $s1  
sub $t2, $s1, $zero  
lw  $t0, 8($s3)
```

FYI: Program

16



A finite sequence of instructions that performs a specific task

Instruction Set Architecture (ISA)



- An abstract interface between the hardware and the lowest-level software (called as **Architecture**)

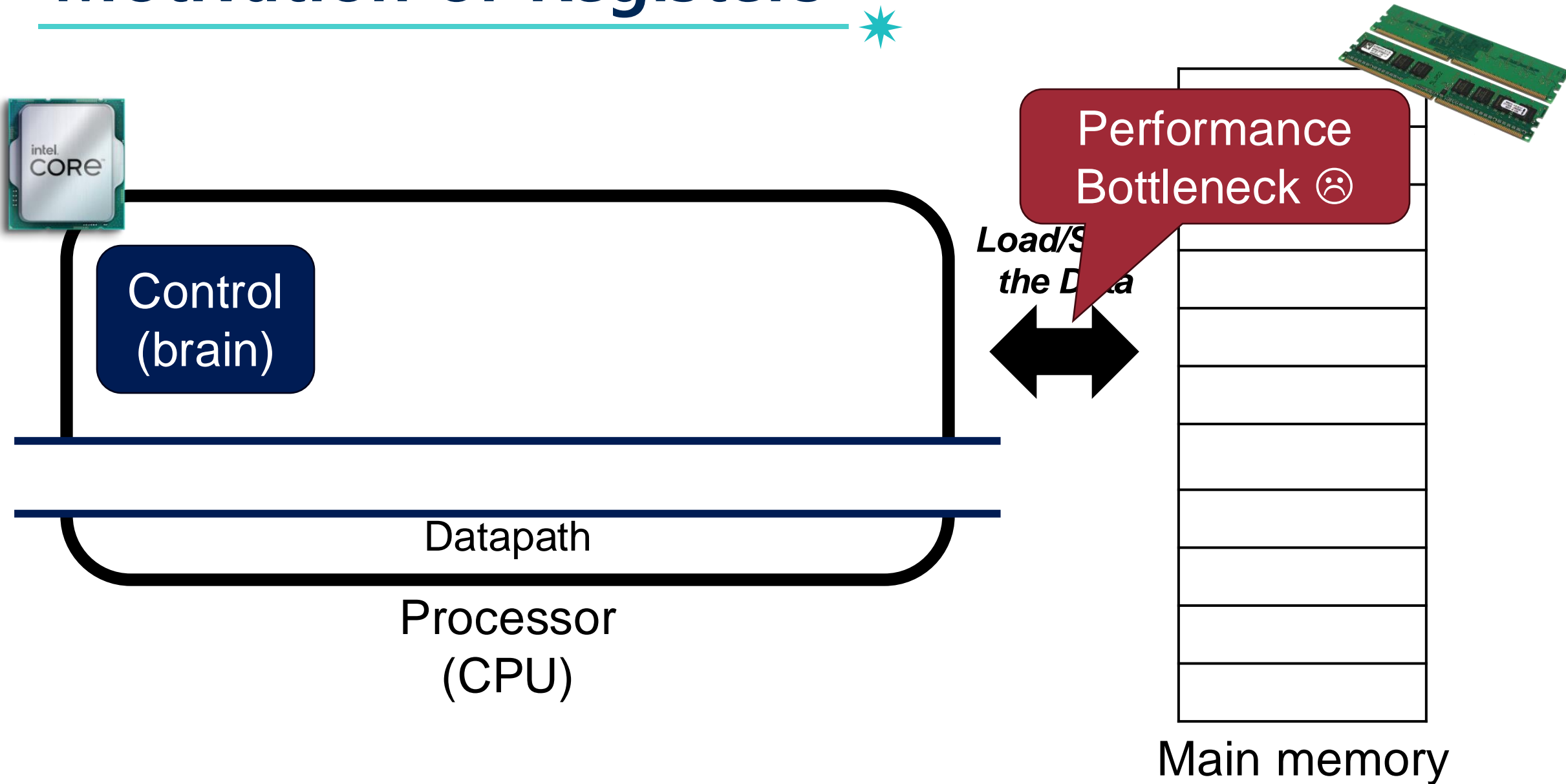
- ISA includes:

- Instruction set
- Registers

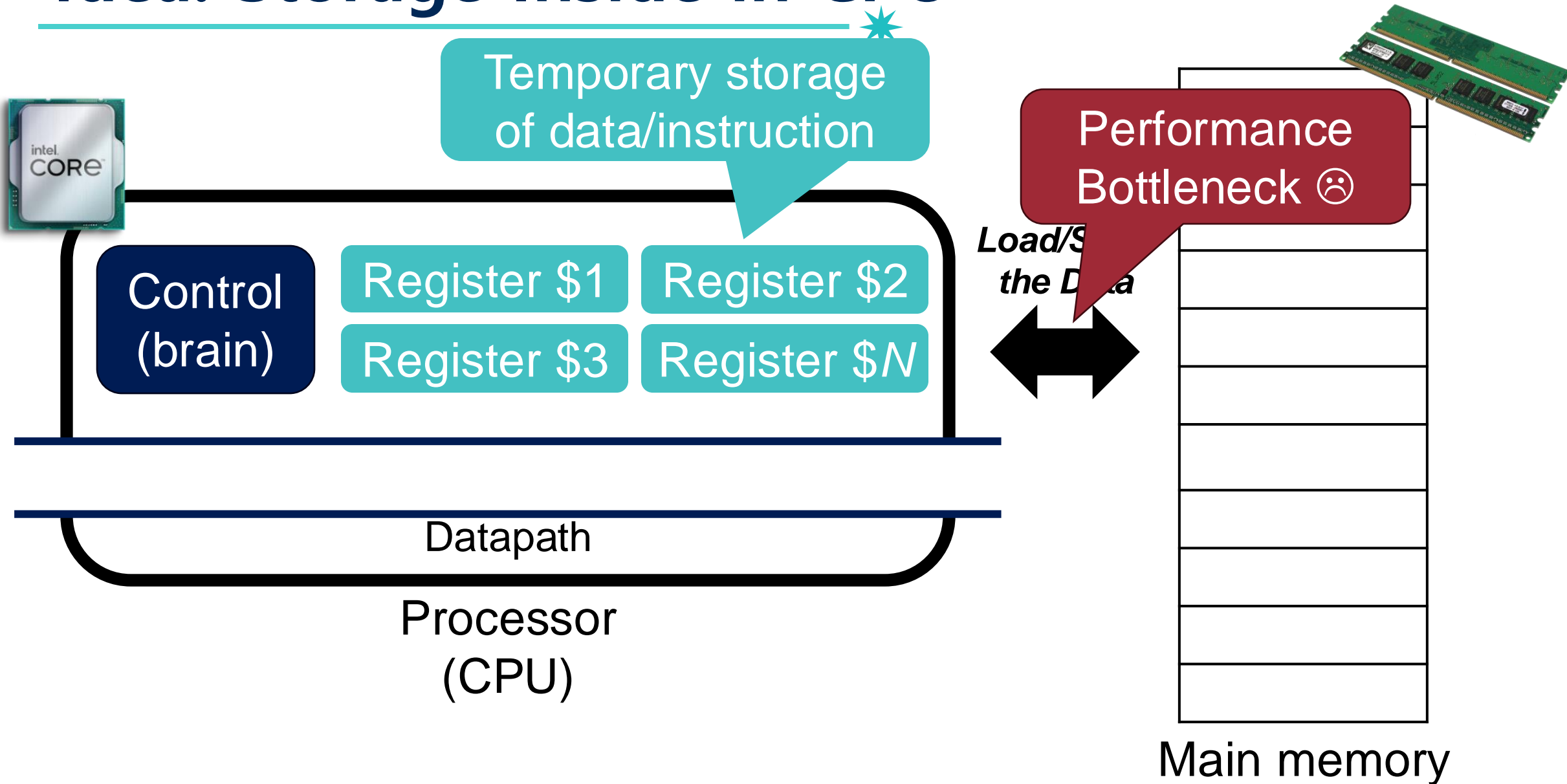
- Size of each register
- Instructions that can use each register

Motivation of Registers

18

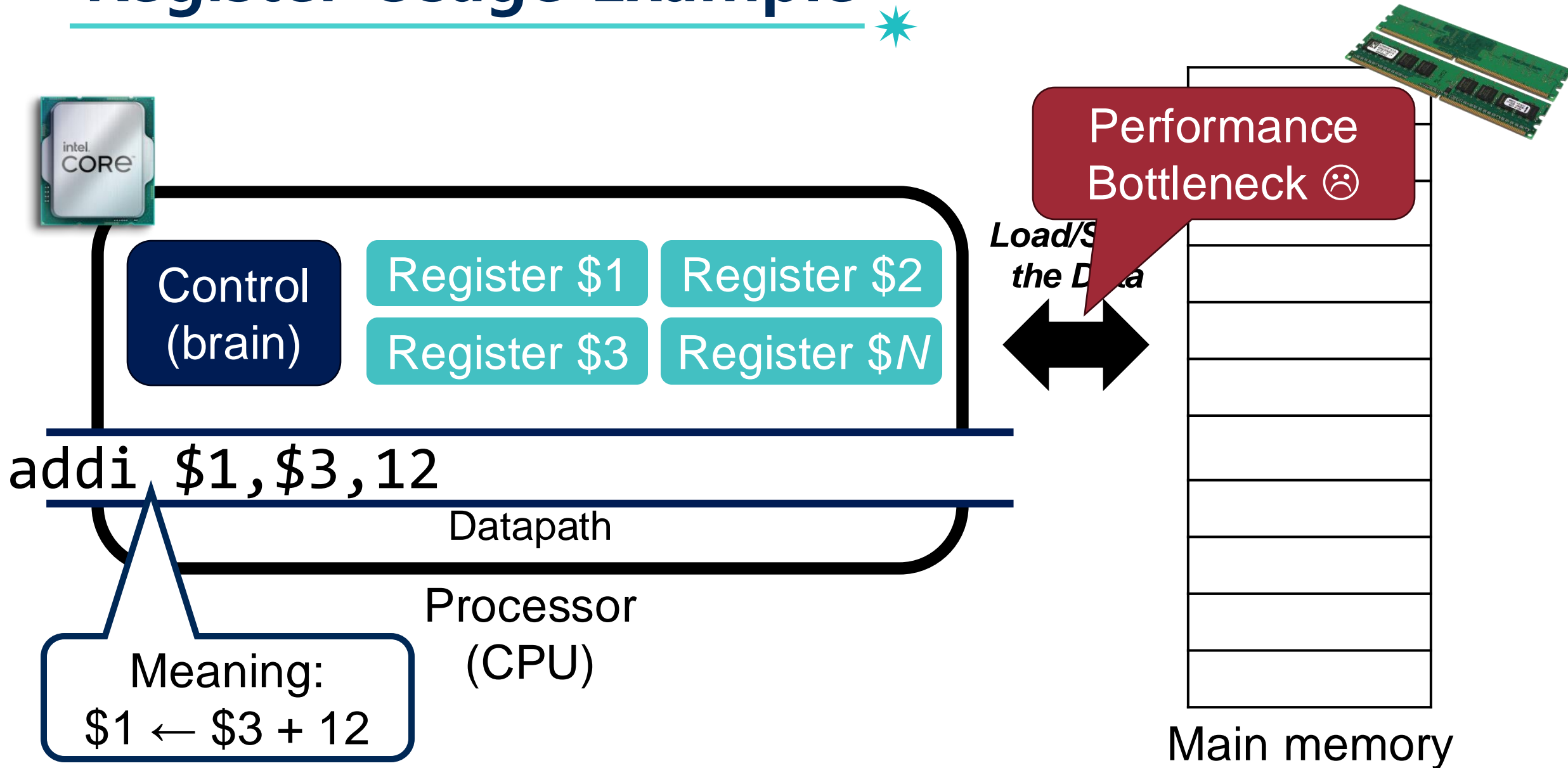


Idea: Storage Inside in CPU

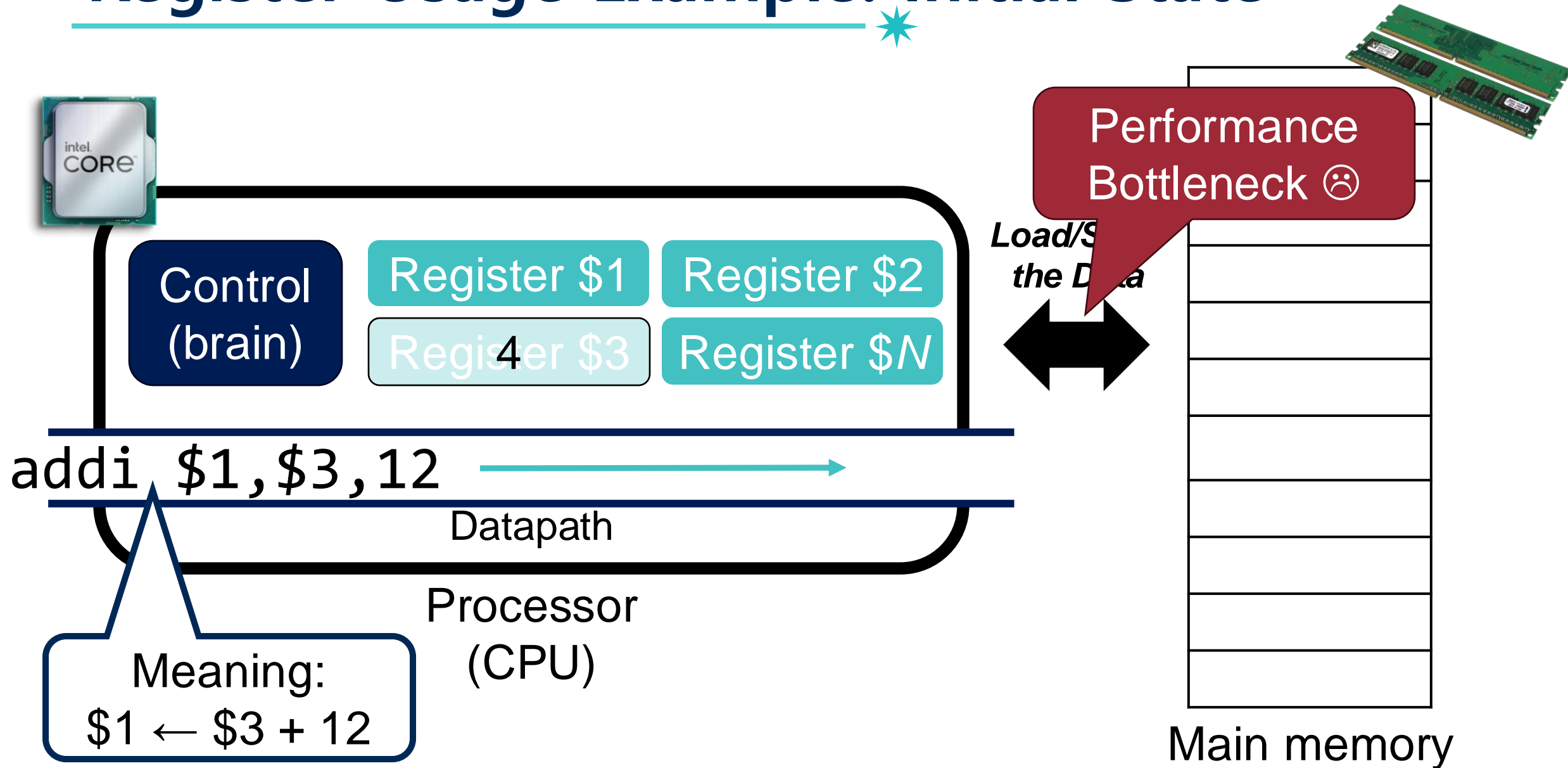


Register Usage Example

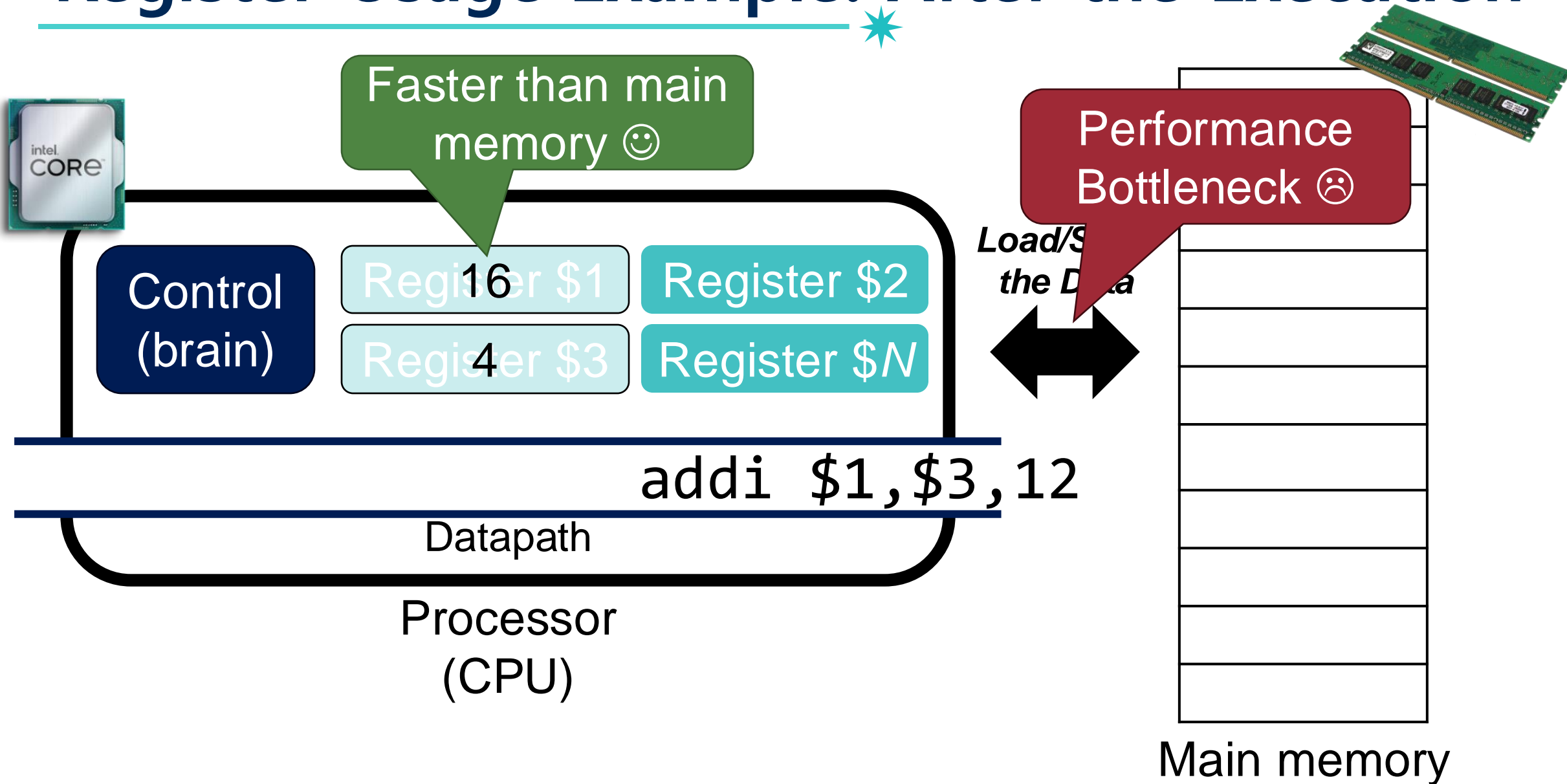
20



Register Usage Example: Initial State



Register Usage Example: After the Execution



Registers



- Reside in CPU
- Temporary storage of data/instruction
- **Faster** than main memory
- Expensive

Special Purpose Registers



- **PC (Program Counter):** holds address of next instruction
- **IR (Instruction Register):** holds the instruction fetched from the memory
- **AC (Accumulator):** holds the result of the computation temporarily

I will provide a detailed explanation of each one as needed

Size of Registers



- MIPS registers are 32-bit
- A “**word**” is the natural unit of data used by as processor
 - Typically, a word size is 32 bits (4 bytes) on a 32-bit machine

Word Size = 32 Bit (in MIPS) *

Word size
= 32 bit

26

Word size
= 32 bit

Control
(brain)

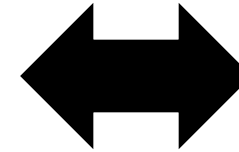
Register A

Register B

Register C

Register N

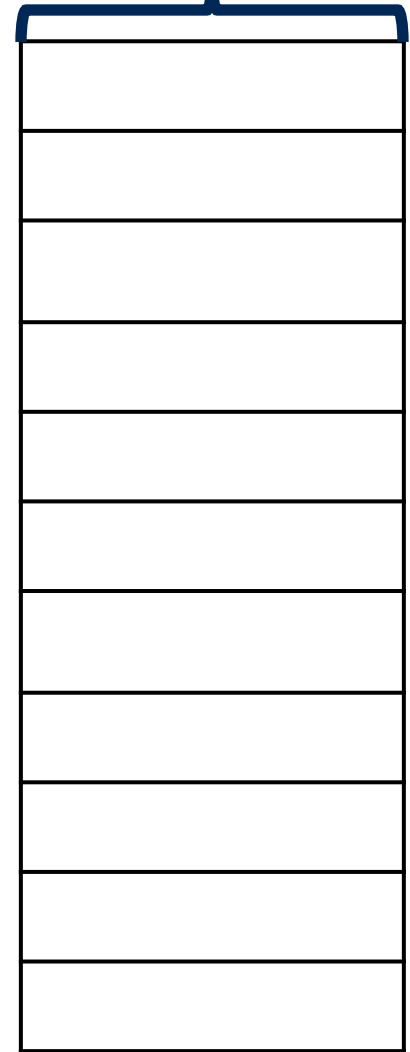
*Load/Store
the Data*



Datapath

Processor
(CPU)

Main memory



System

Control Panel > System and Security > System

Control Panel Home


- Device Manager
- Remote settings
- System protection
- Advanced system settings

View basic information about your computer

Windows edition

Windows 10 Enterprise

© 2018 Microsoft Corporation. All rights reserved.



System

Processor:	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz 3.40 GHz
Installed memory (RAM):	16.0 GB
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

Computer name, domain, and workgroup settings

Computer name:	DESKTOP-8A6BML4	Change settings
Full computer name:	DESKTOP-8A6BML4.hyper.office	
Computer description:		
Domain:	hyper.office	

Windows activation

Windows is activated [Read the Microsoft Software License Terms](#)

Product ID: 00329-10330-00000-AA731 [Change product key](#)

See also

Security and Maintenance

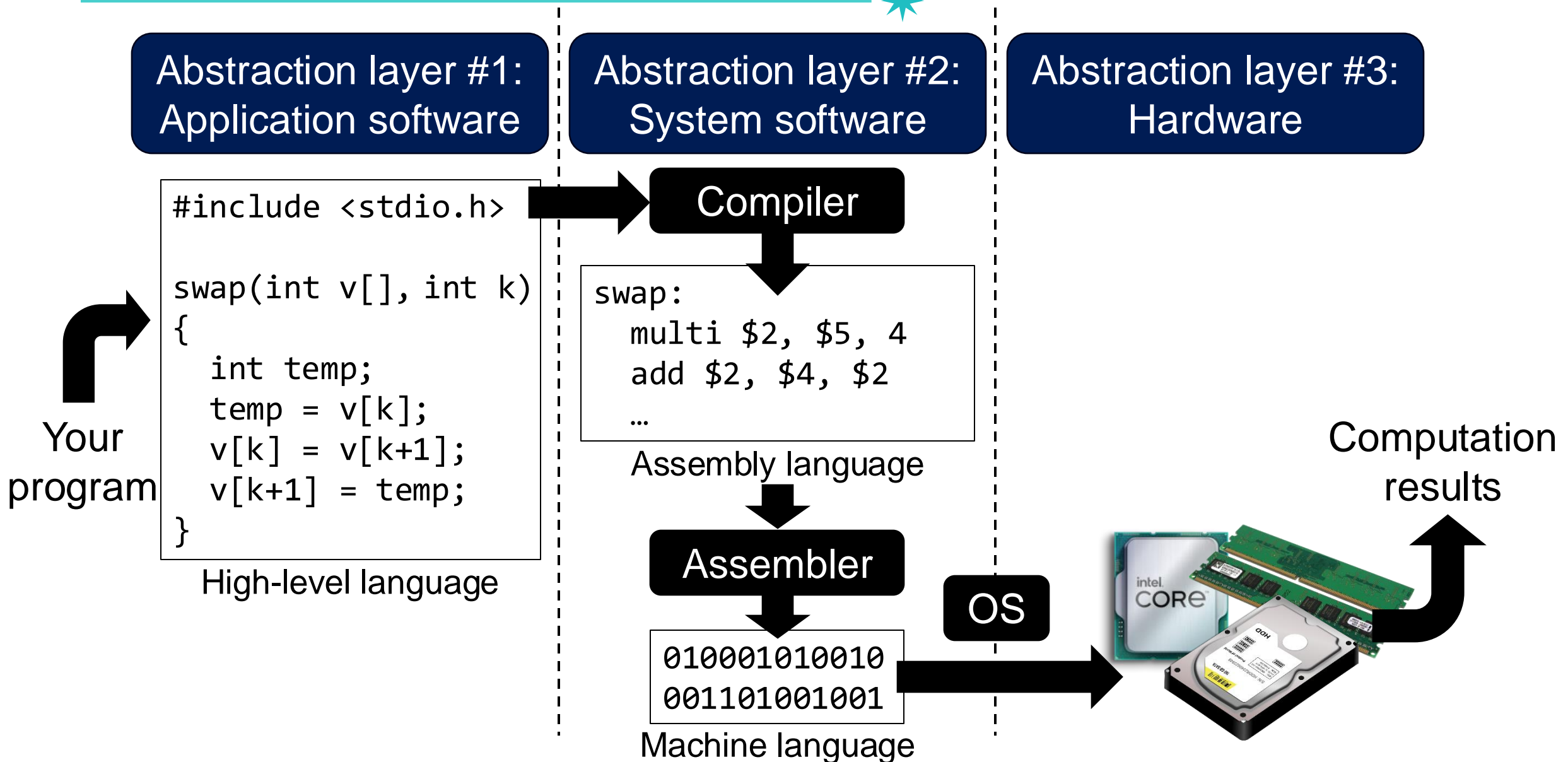
Word size = 64 bit

Instruction Set Architecture (ISA)

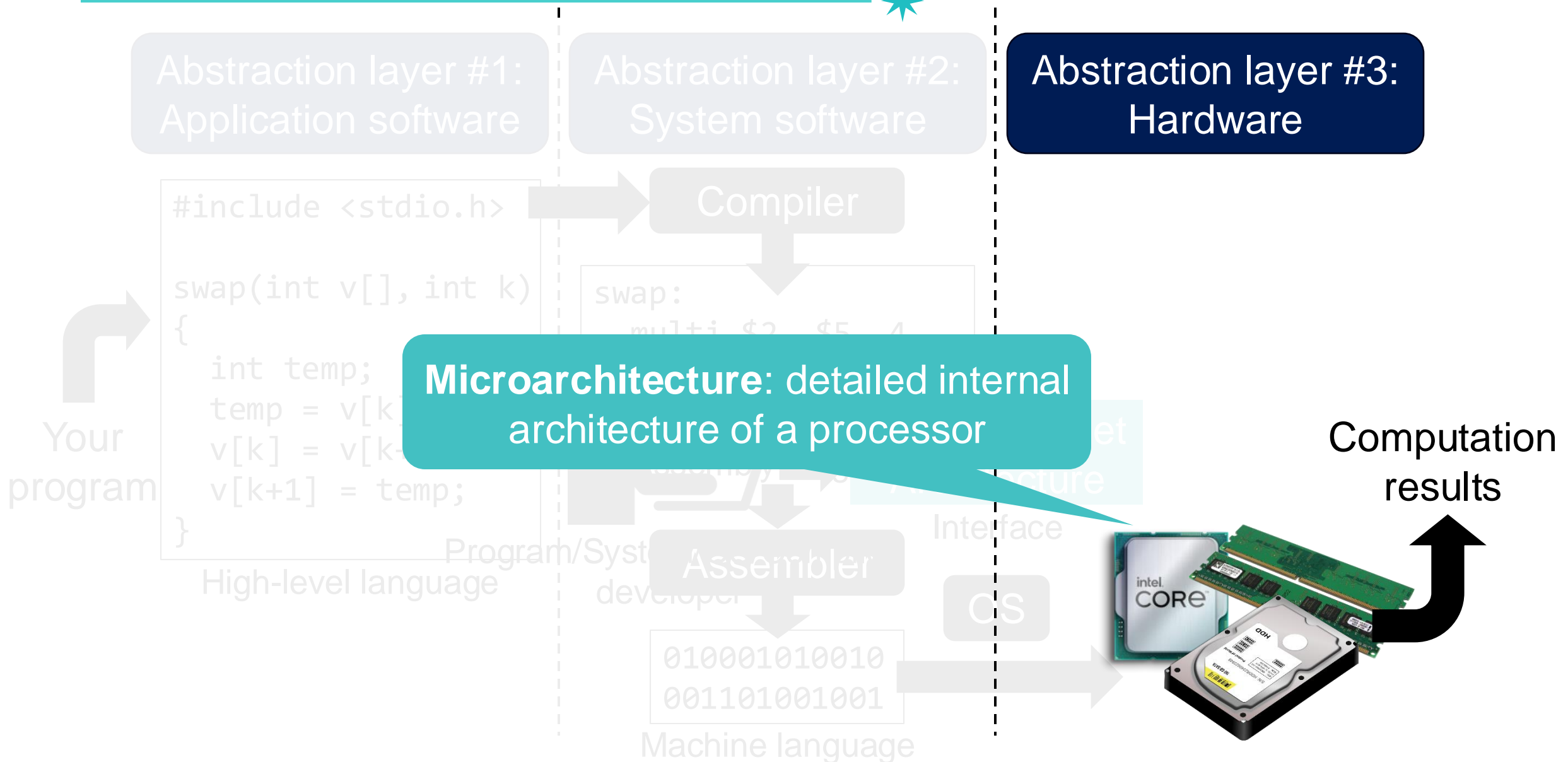


- An abstract interface between the hardware and the lowest-level software (called as **Architecture**)
- ISA includes:
 - Instruction set
 - Registers
 - Operand types
 - Data types (integer, floating points, ...)
 - Addressing modes
 - I/O
- **Programmer's view of processor**
 - But not the details of how it is designed and implemented

Architecture vs. Microarchitecture



Architecture vs. Microarchitecture



Microarchitecture



- Organization of the machine below the ISA
 - Number/location of functional units
 - Pipeline/cache configurations
 - Programmer transparent techniques: prefetching, ...
- Hardware realization
- Logic circuits, VLSI technology, process, ...

Architecture vs. Microarchitecture

Abstraction layer #1:
Application software

Abstraction layer #2:
System software

Abstraction layer #3:
Hardware

Your program

```
#include <stdio.h>

swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

High-level language

Compiler

*Same instruction
set architecture*

```
swap:
    multi $2, $5, 4
    add $2, $4, $2
```

Assembly language

Instruction Set
Architecture

Assembler

```
010001010010
001101001001
```

Machine language



Core i5-7600

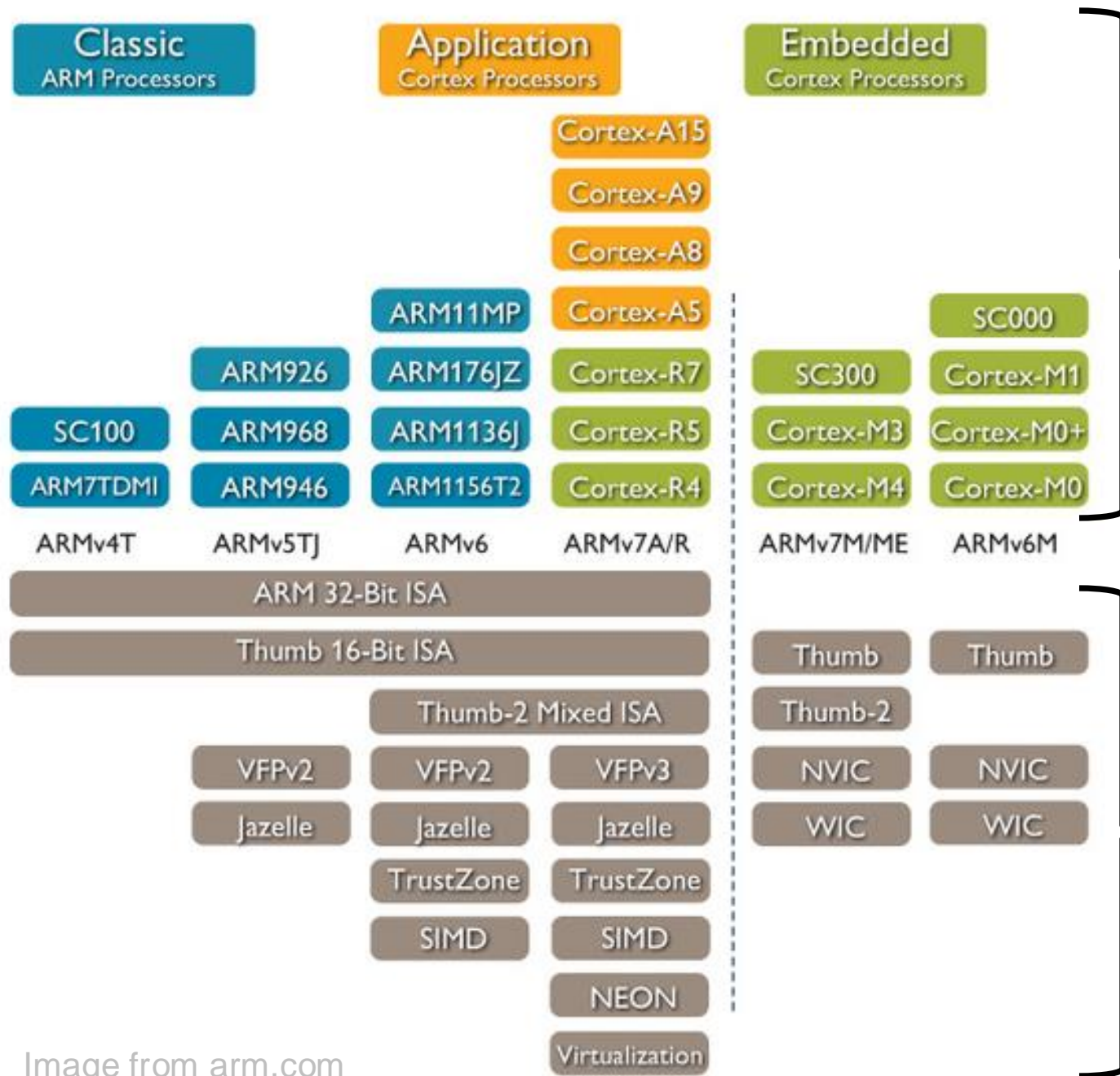


*Different
microarchitecture*



Core i5-7400

Architecture vs. Microarchitecture: ARM



Microarchitecture

Architecture features

Instruction Format

*Later, we will cover MIPS ISA in detail.
For now, let's go over the fundamentals of instructions.*

Instruction Format

35



`addi $1,$3,12`

Instruction Format



Operation

addi

\$1, \$3, 12

Operands

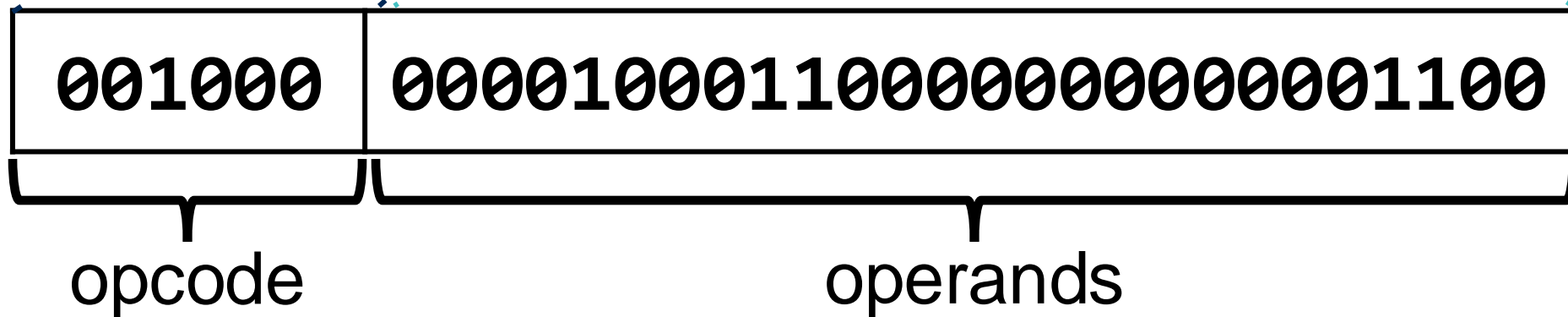
Instruction Format



Operation

Operands

`addi $1,$3,12`



Instruction Length



- **Type #1: Fixed size**

- Every instruction is represented using the same number of bytes
- E.g., MIPS instructions are always 32 bits (4 bytes) long

```
addi $1,$3,12
```

```
j L1
```

- **Type #2: Variable size (8 bits, 16 bits, 32 bits, 64 bits, ...)**



- Different instructions are represented using different numbers of bytes
- E.g., Intel X86, AMD, ...

```
addi $1,$3,12
```

```
j L1
```

Fixed-size vs. Variable-size Instruction

39

	Fixed size	Variable size
Memory management		

addi \$1,\$3,12





j L1

addi \$1,\$3,12

j L1

No waste space

Fixed-size vs. Variable-size Instruction

	Fixed size	Variable size
Memory management		
Decode	 Easy to decode (Always 32 bits => Less hardware required)	 Difficult to decode (Complex hardware required)

```
addi $1,$3,12
```

```
j L1
```





```
addi $1,$3,12
```

```
j L1
```



Fixed-size vs. Variable-size Instruction

41

	Fixed size	Variable size
Memory management		
Decode	 Easy to decode (Always 32 bits => Less hardware required)	 Difficult to decode (Complex hardware required)

The computer architect should decide the length of instruction, opcode, and operand

CPU Design Philosophy: RISC vs. CISC

42

- **Reduced Instruction Set Computer (RISC)**

- Example: MIPS, ARM, PowerPC
- Small and simple instruction set
- Fixed-size instruction format

- **Reduced Instruction Set Computer (CISC)**

- Example: Intel x86, AMD
- A large number of instruction set
- Variable-size instruction format

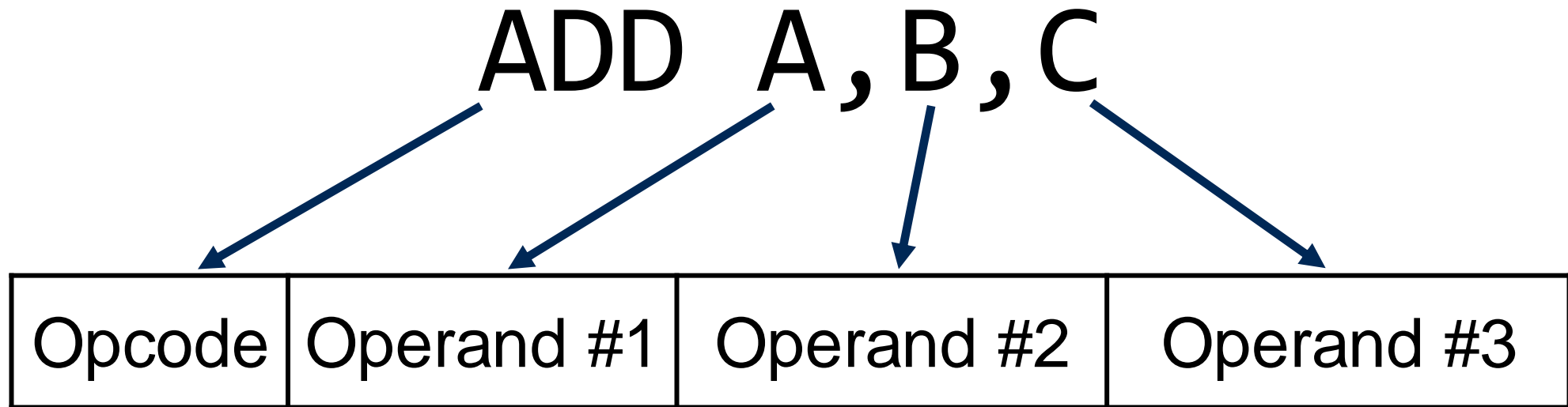
Number of Operands

Number of Operands



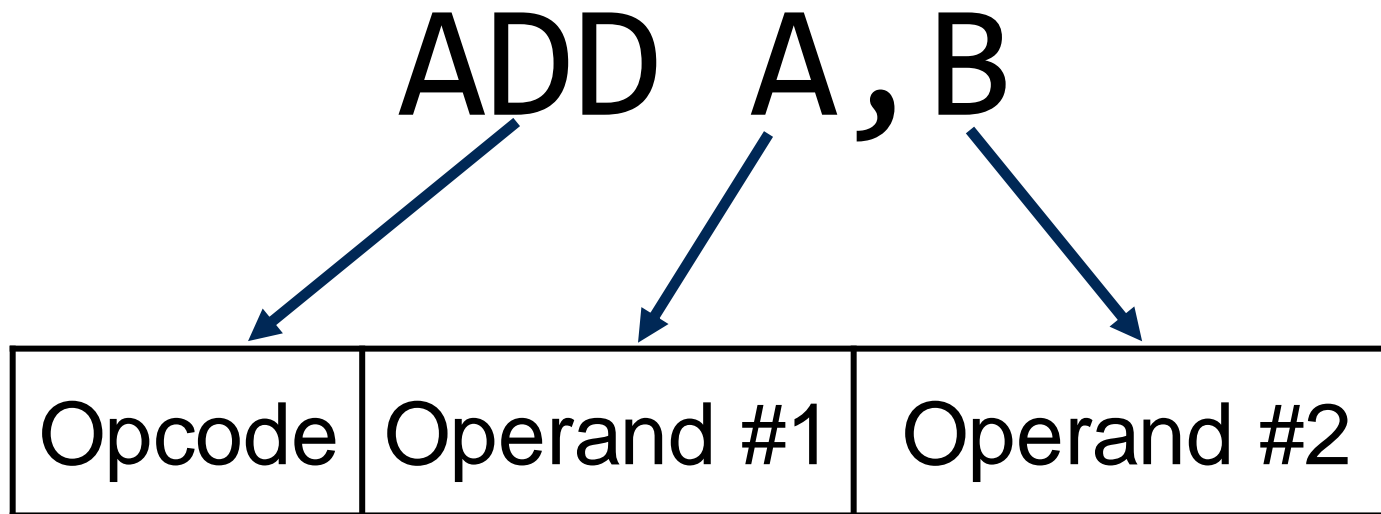
- Three operands
- Two operands
- One operands
- Zero operands

Three Operands Example



Meaning: $A \leftarrow B + C$

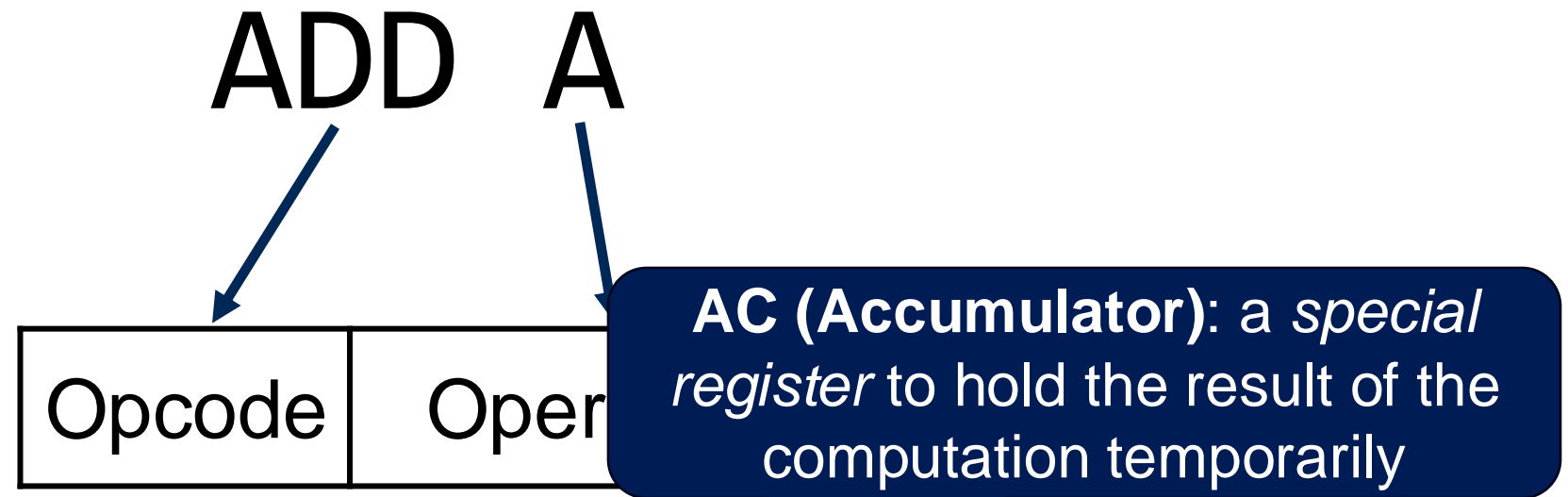
Two Operands Example



Meaning: $A \leftarrow A + B$

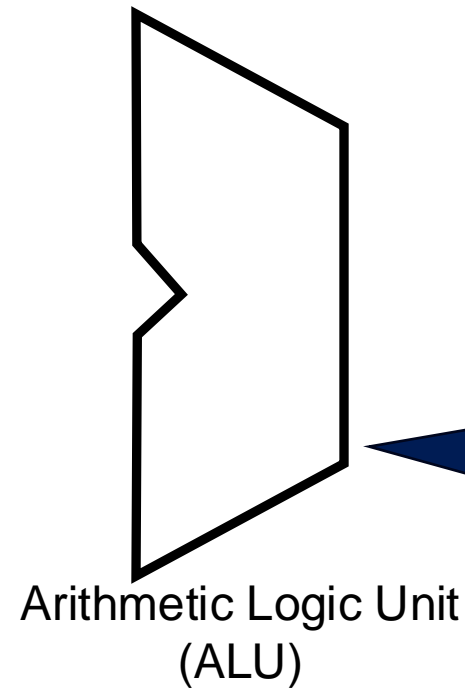
One Operand Example

47



Meaning: $AC \leftarrow AC + A$

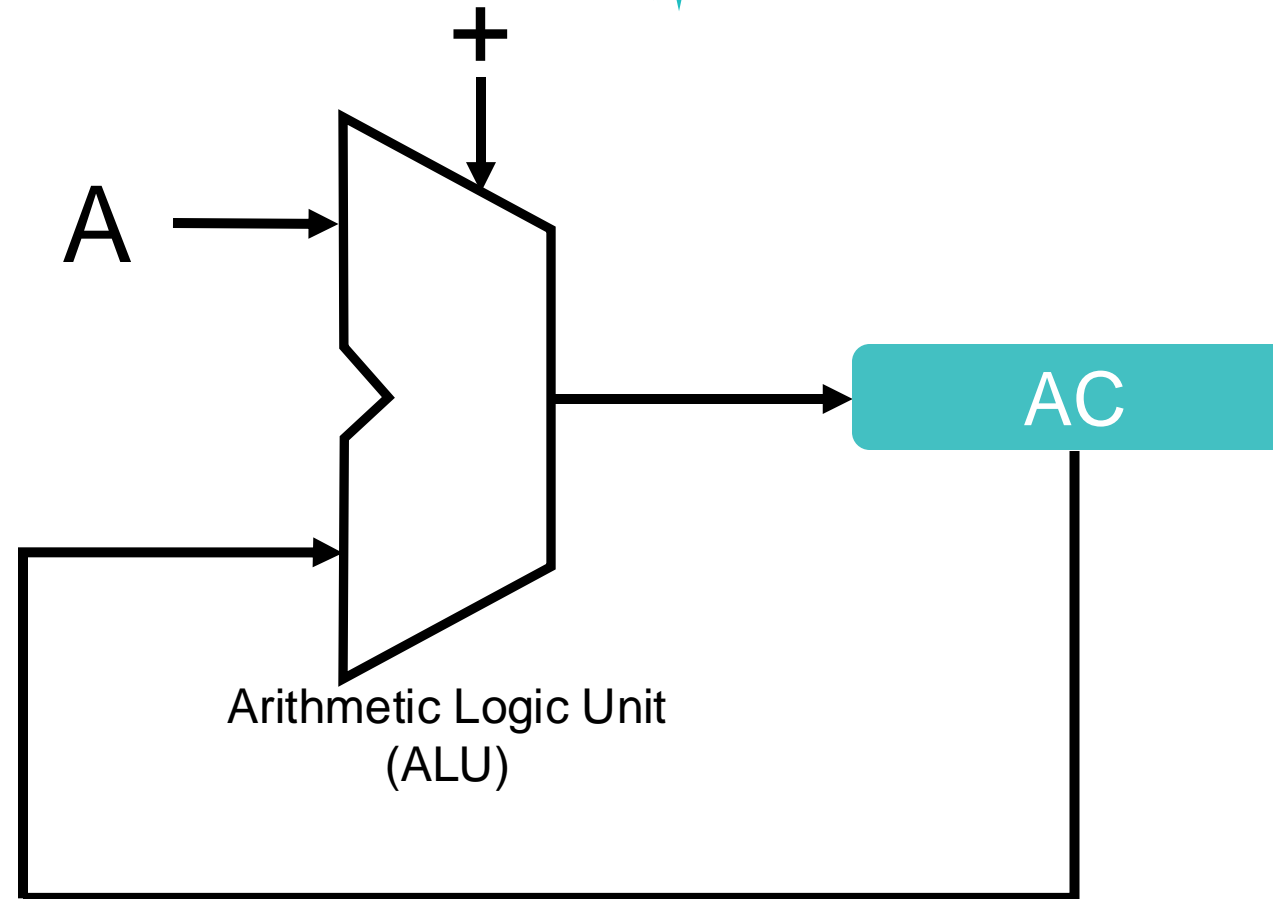
One Operand Example



A logical circuit that performs arithmetic operations

Meaning: $AC \leftarrow AC + A$

One Operand Example



Meaning: $AC \leftarrow AC + A$

Zero Operand Example



Is zero operand instruction possible?

ADD



Opcode

Zero Operand Example: Using Stack

PUSH A

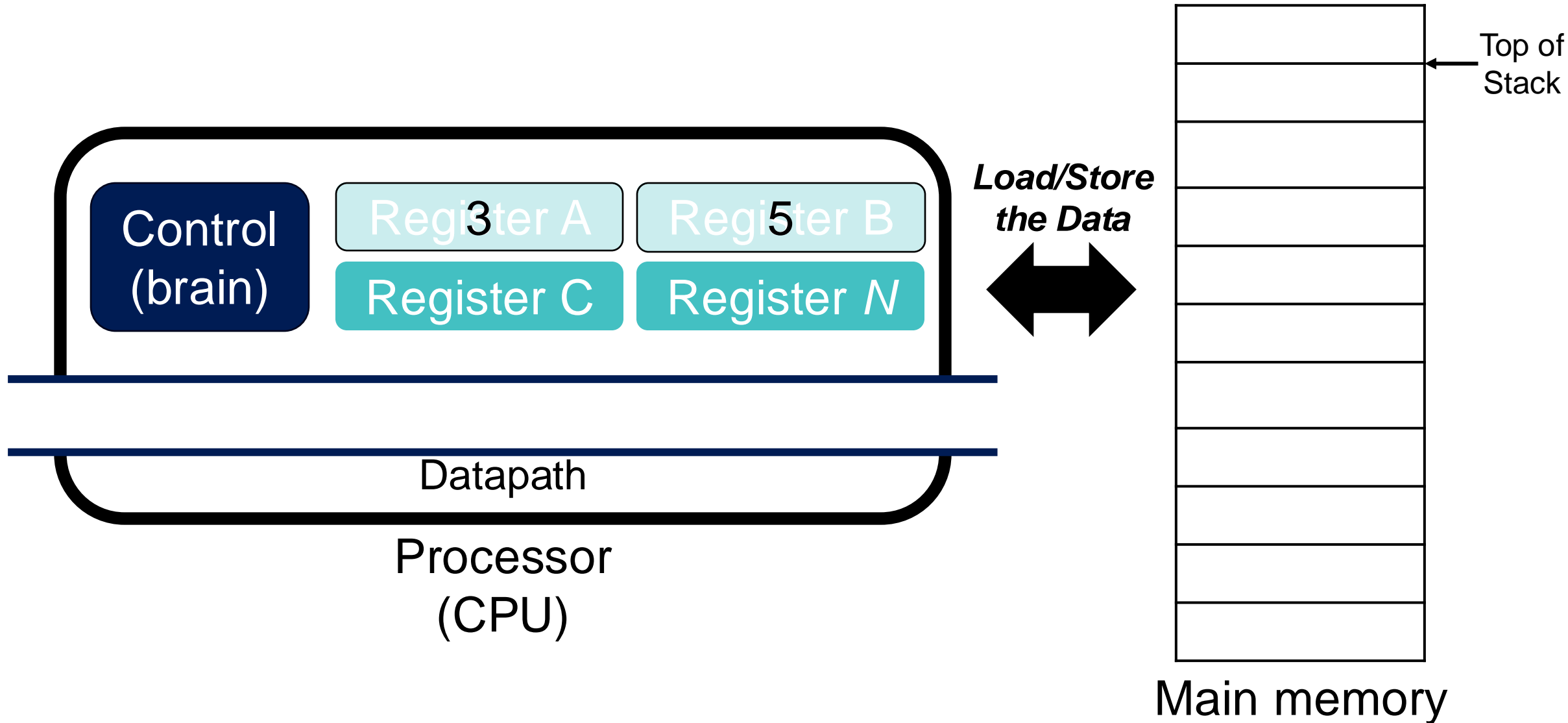
PUSH B

ADD

POP C

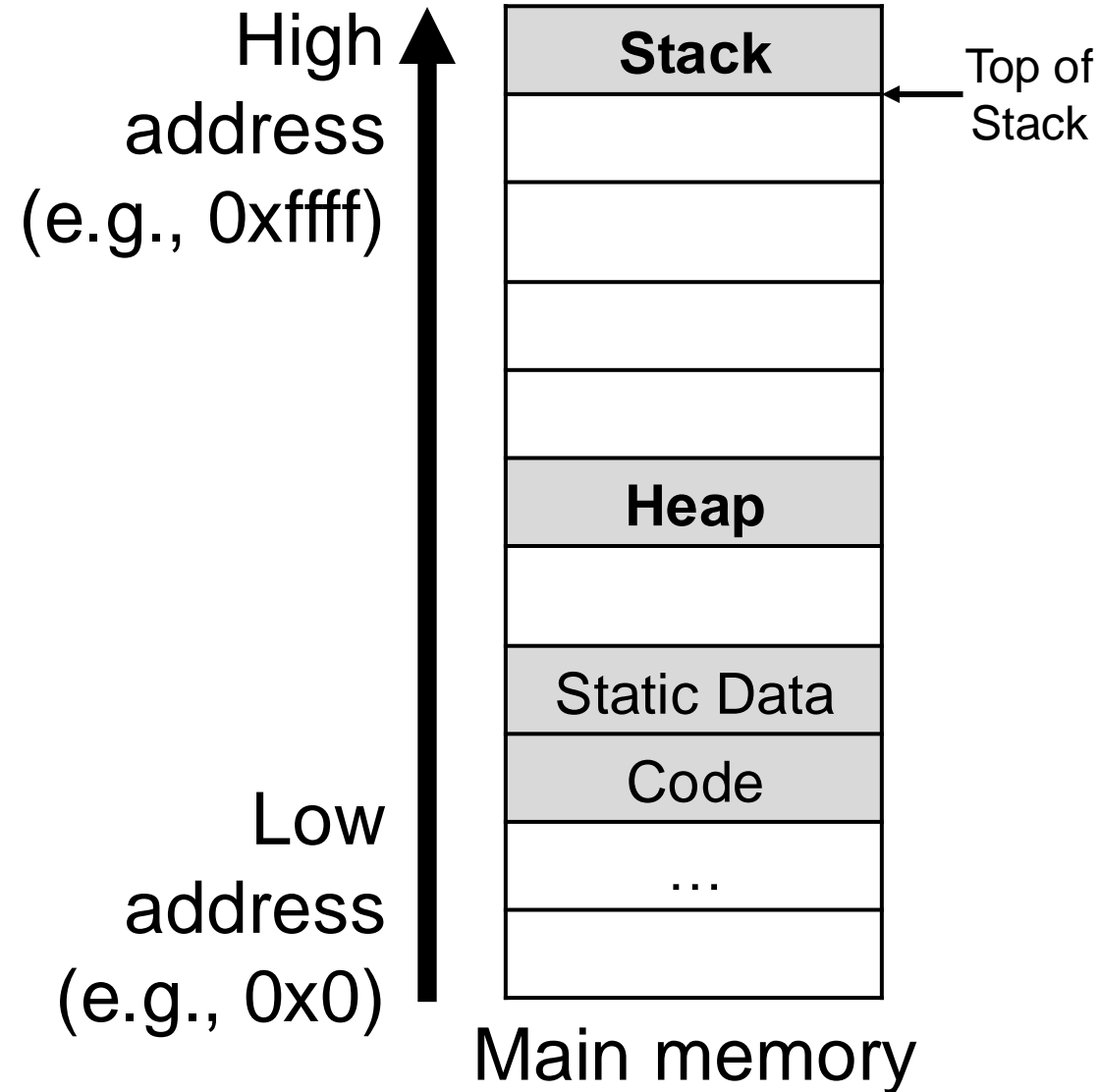
Meaning: $C \leftarrow A + B$

Zero Operand Example: Initial State



FYI: Memory Layout

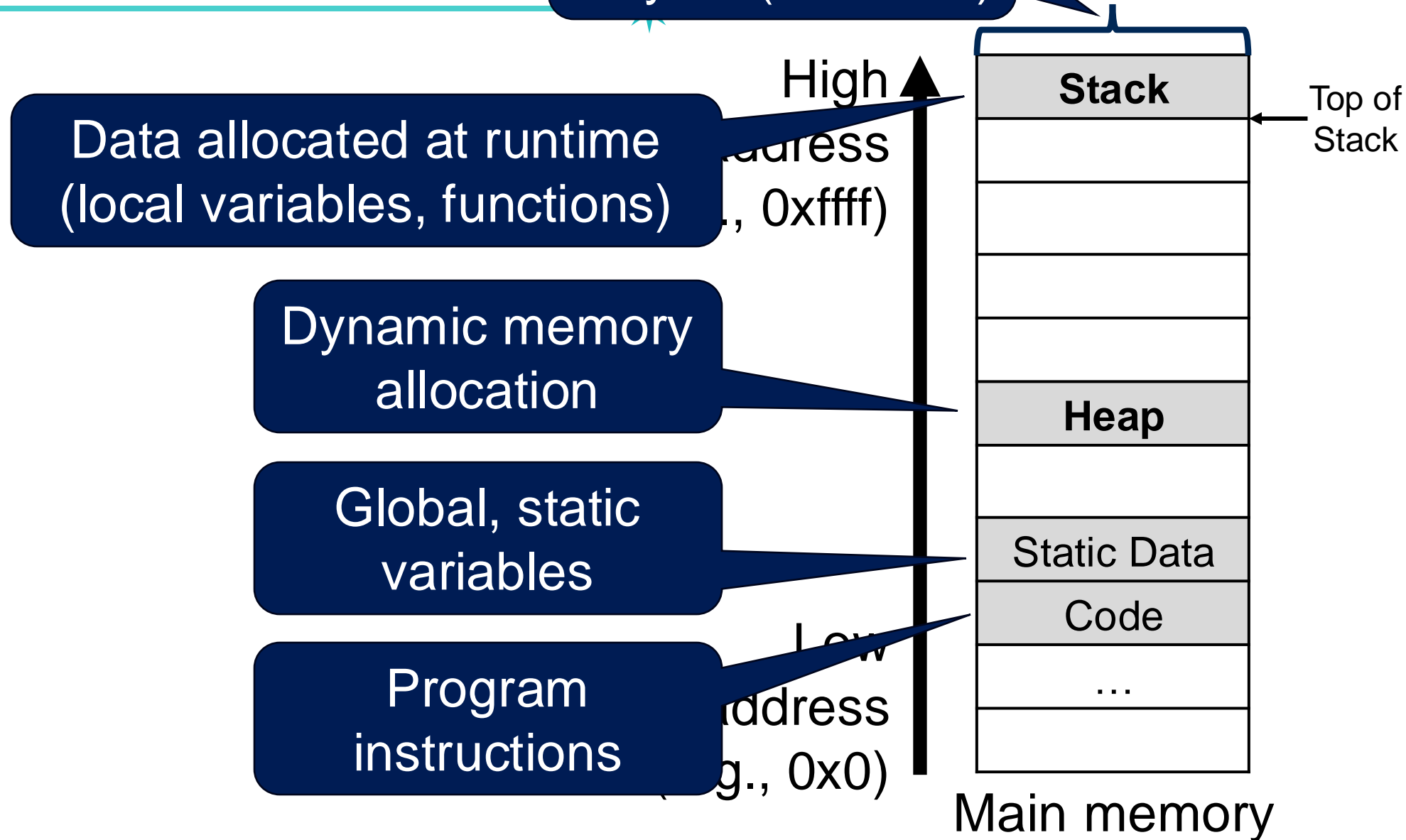
53



FYI: Memory Layout

Word size
4 bytes (= 32 bits)

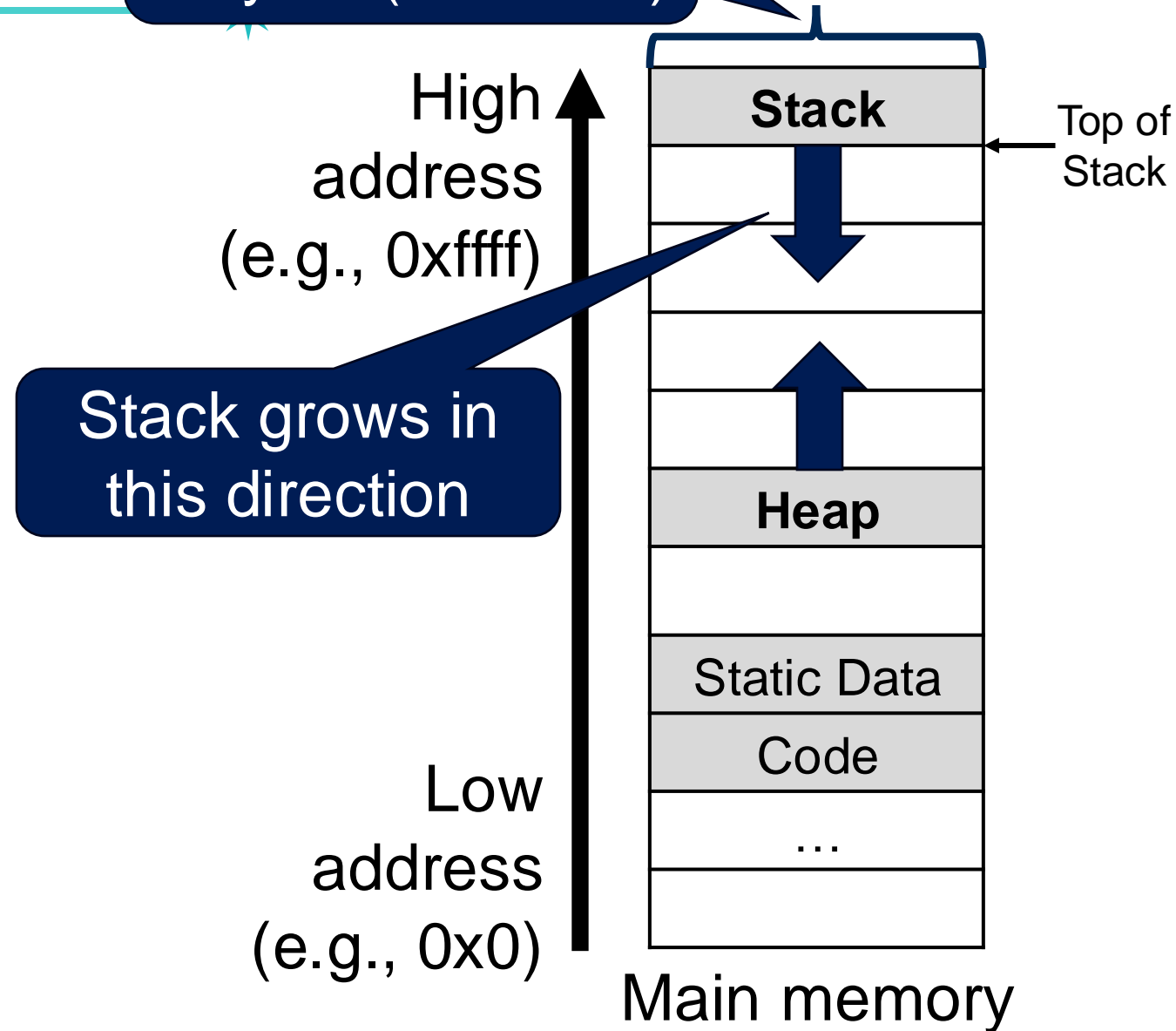
54



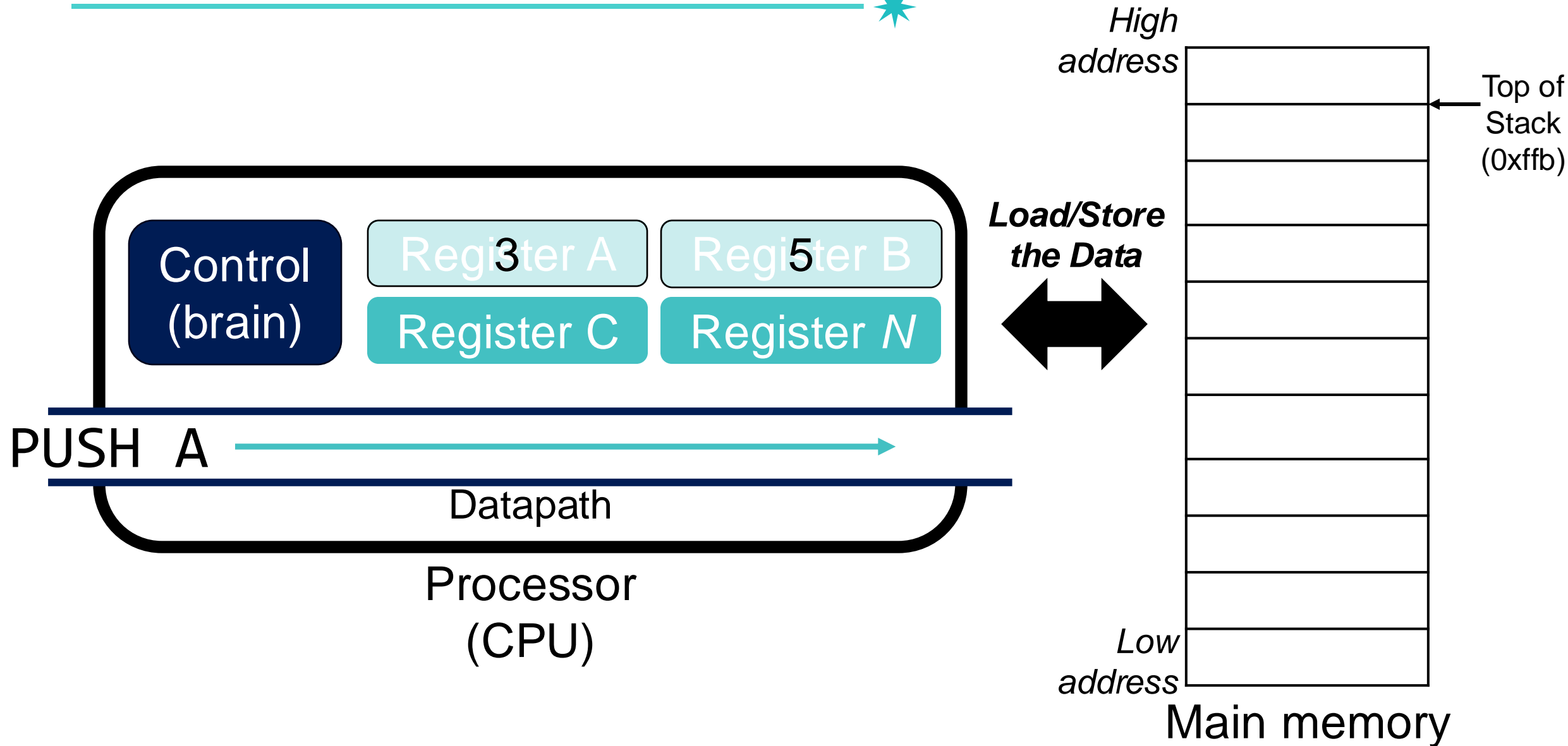
FYI: Memory Layout

Word size
4 bytes (= 32 bits)

55

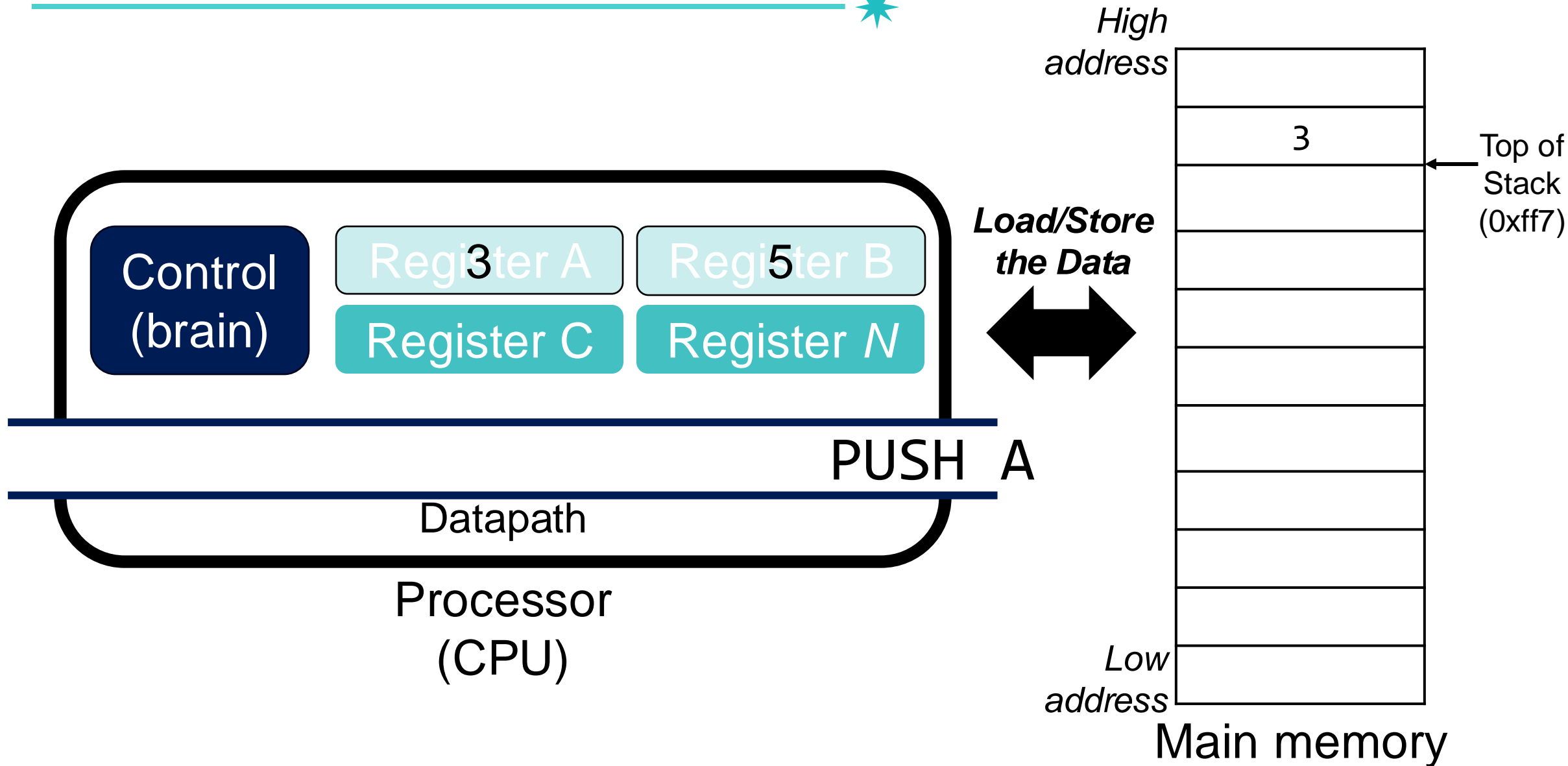


Zero Operand Example: Execution



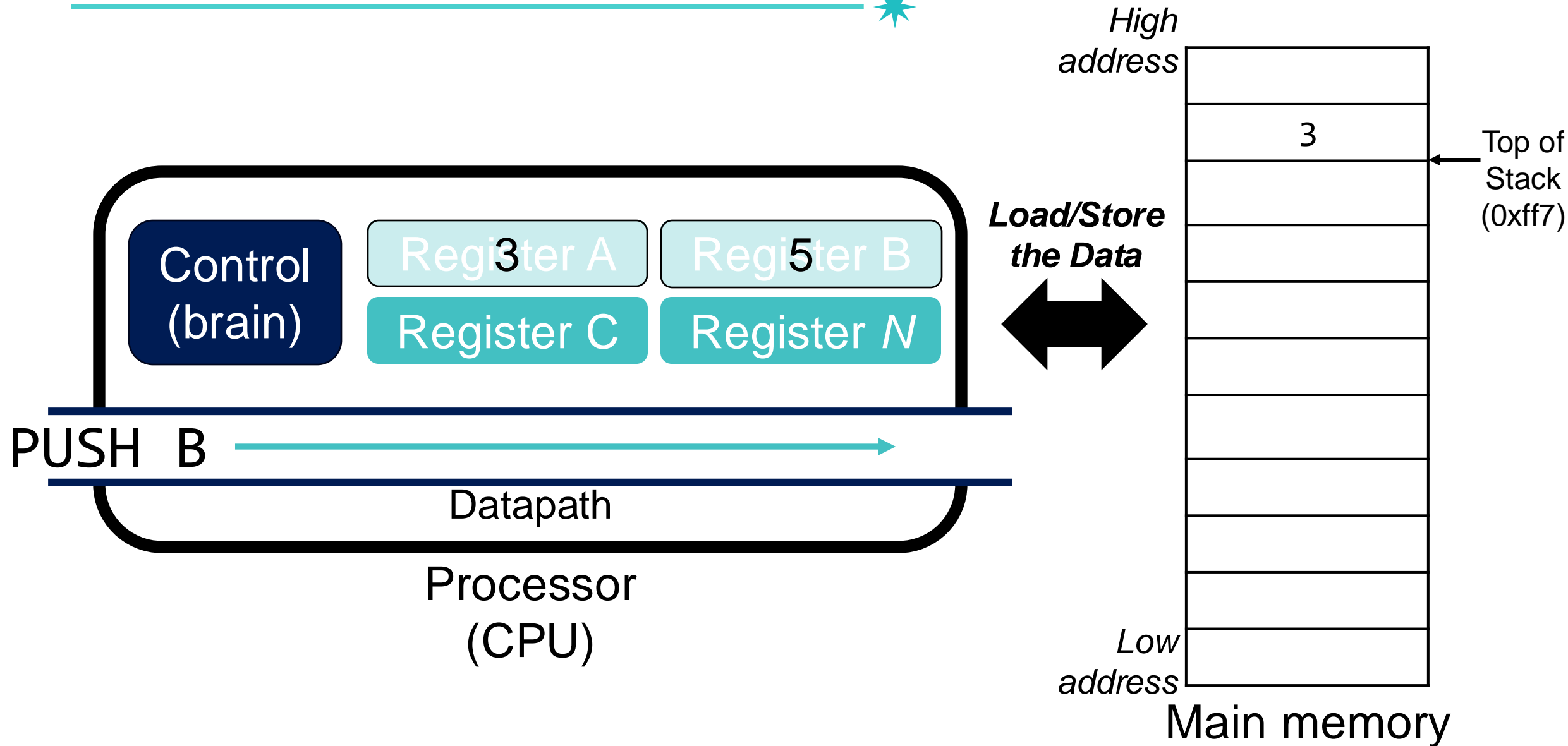
Zero Operand Example: Execution

57



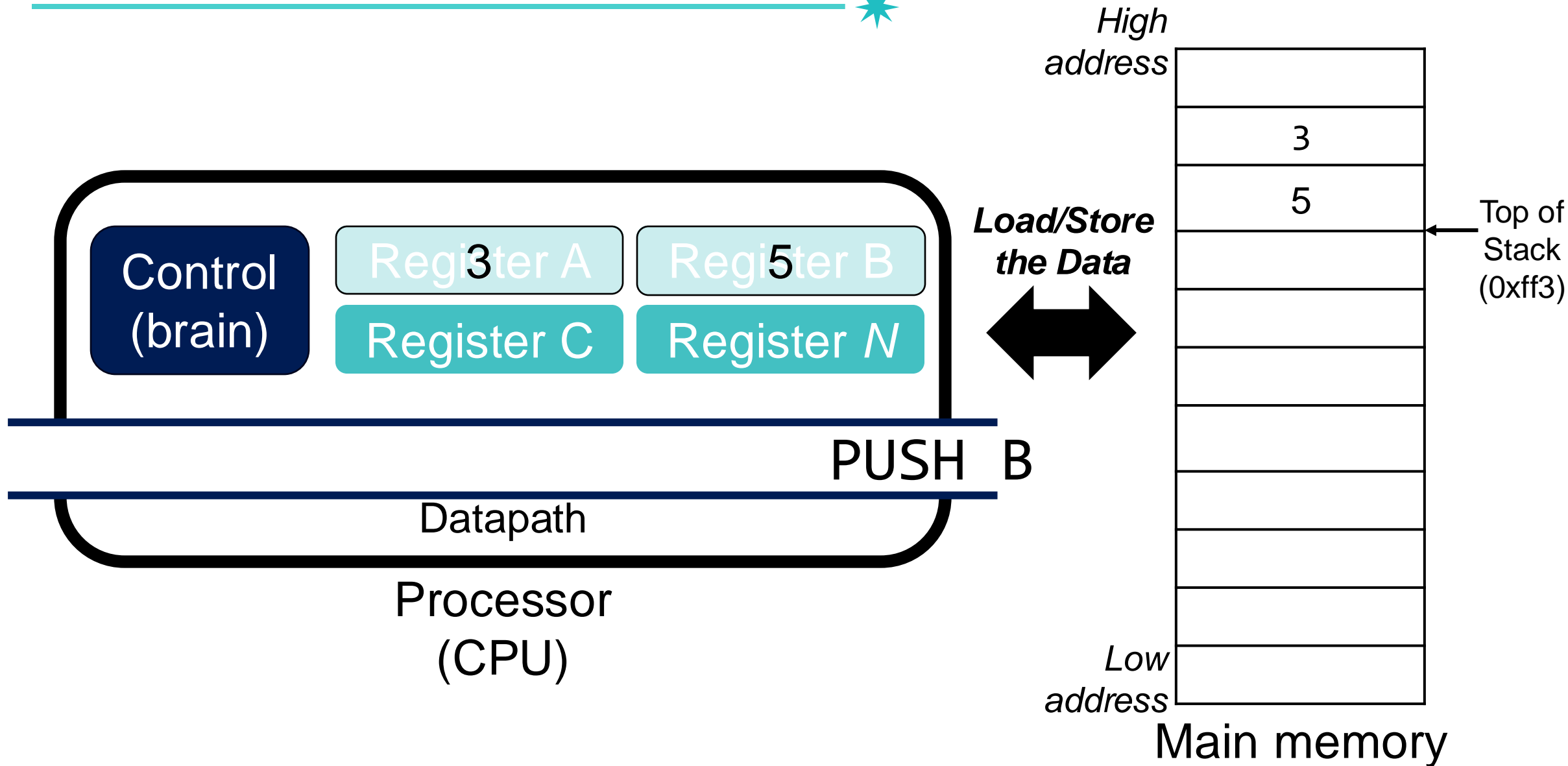
Zero Operand Example: Execution

58

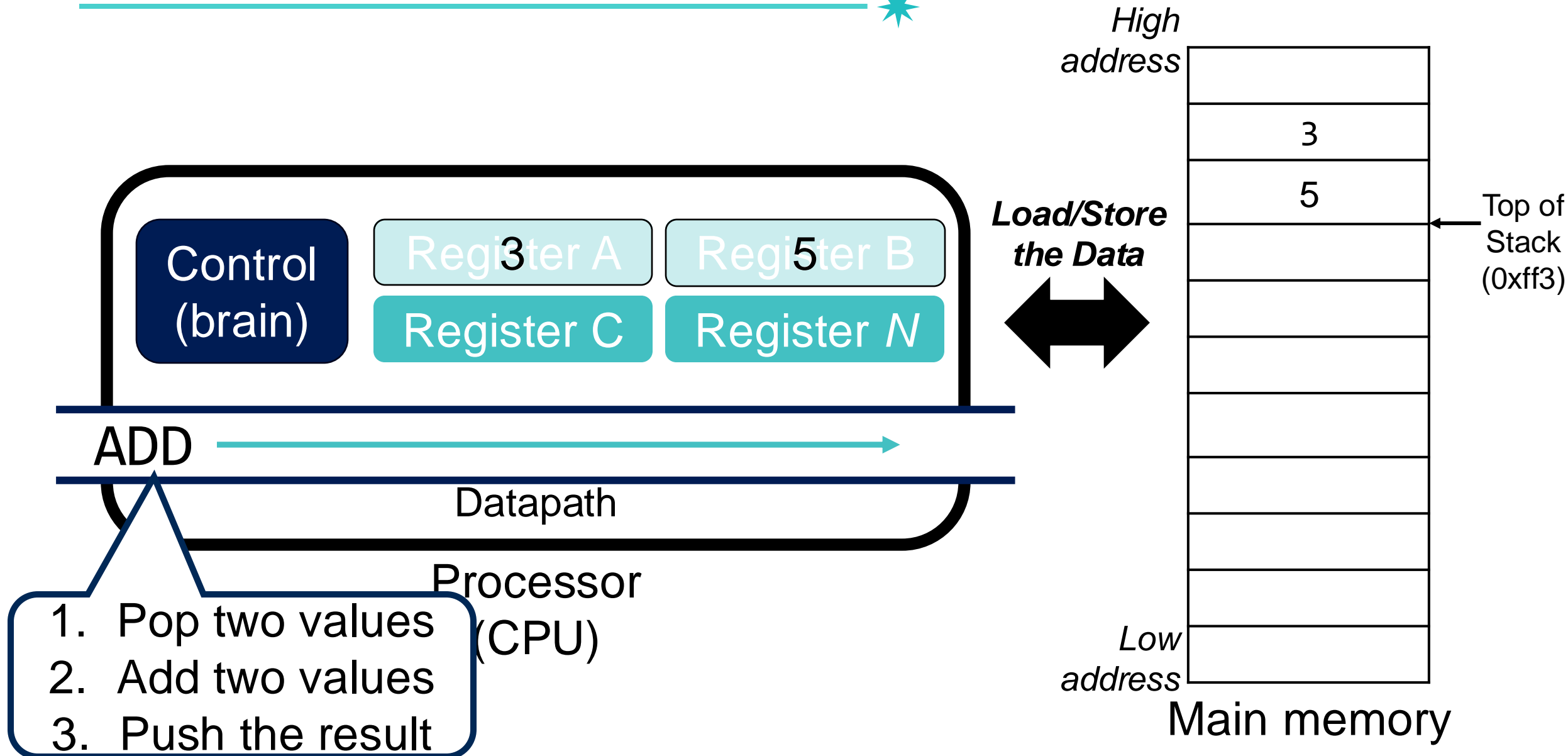


Zero Operand Example: Execution

59

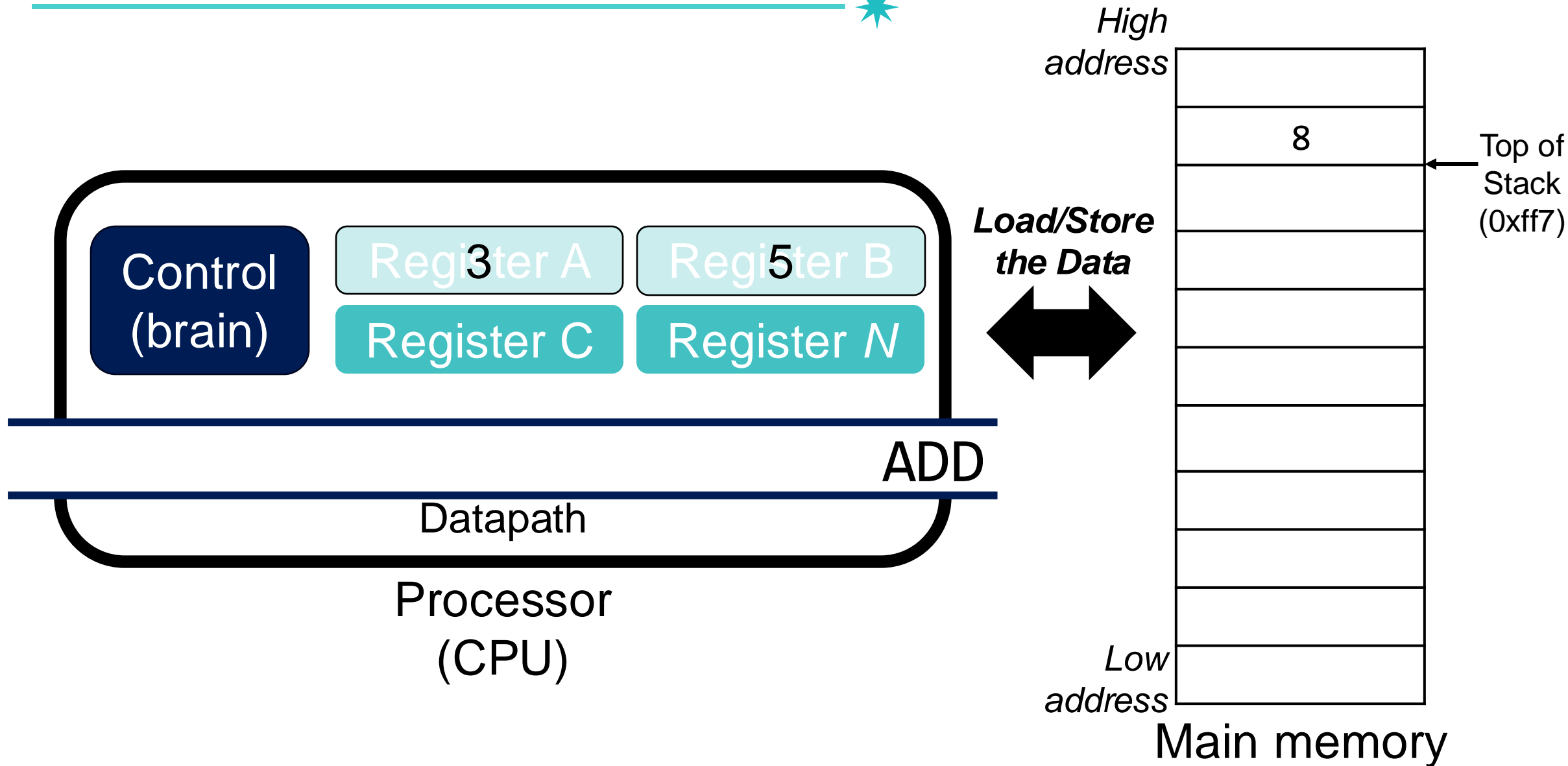


Zero Operand Example: Execution



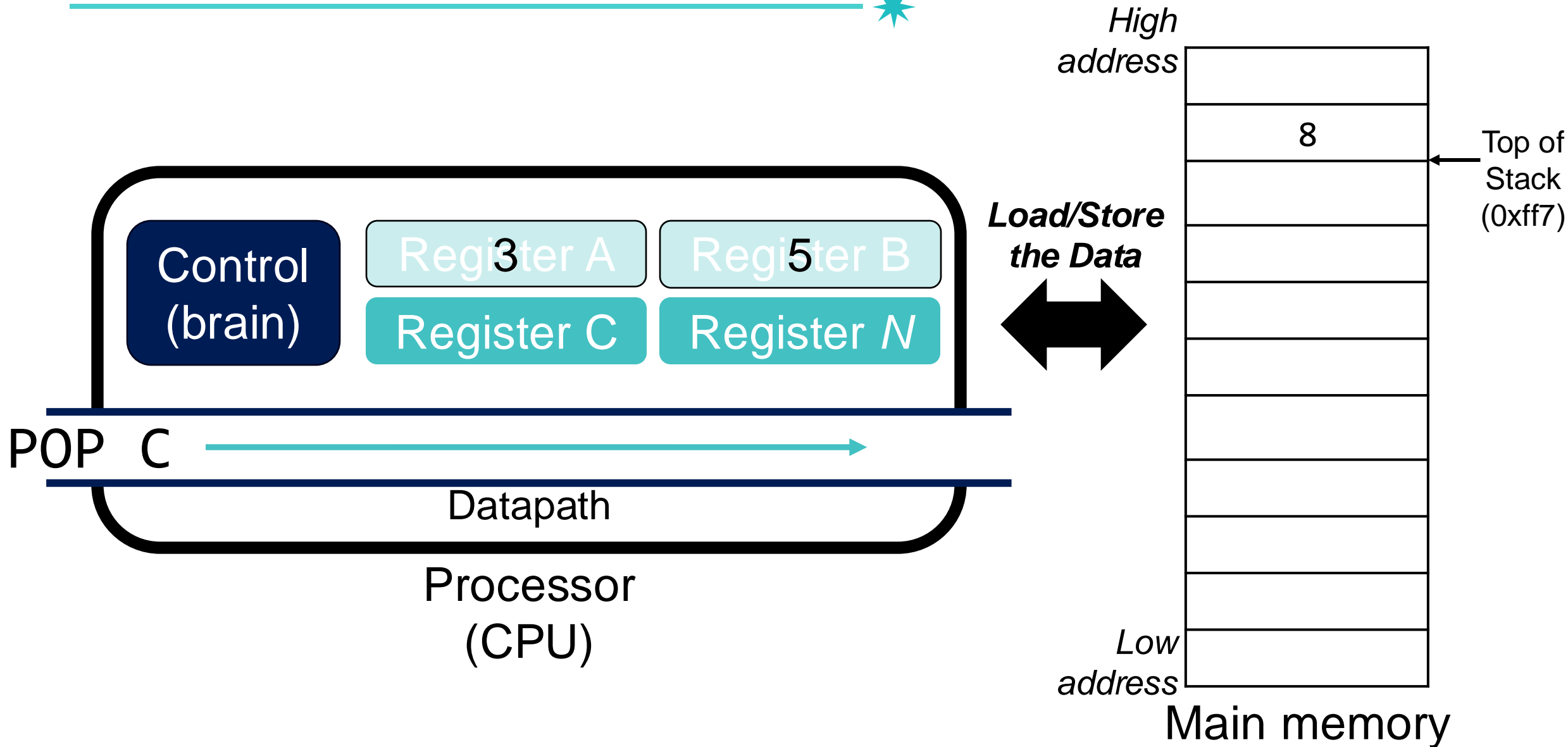
Zero Operand Example: Execution

61

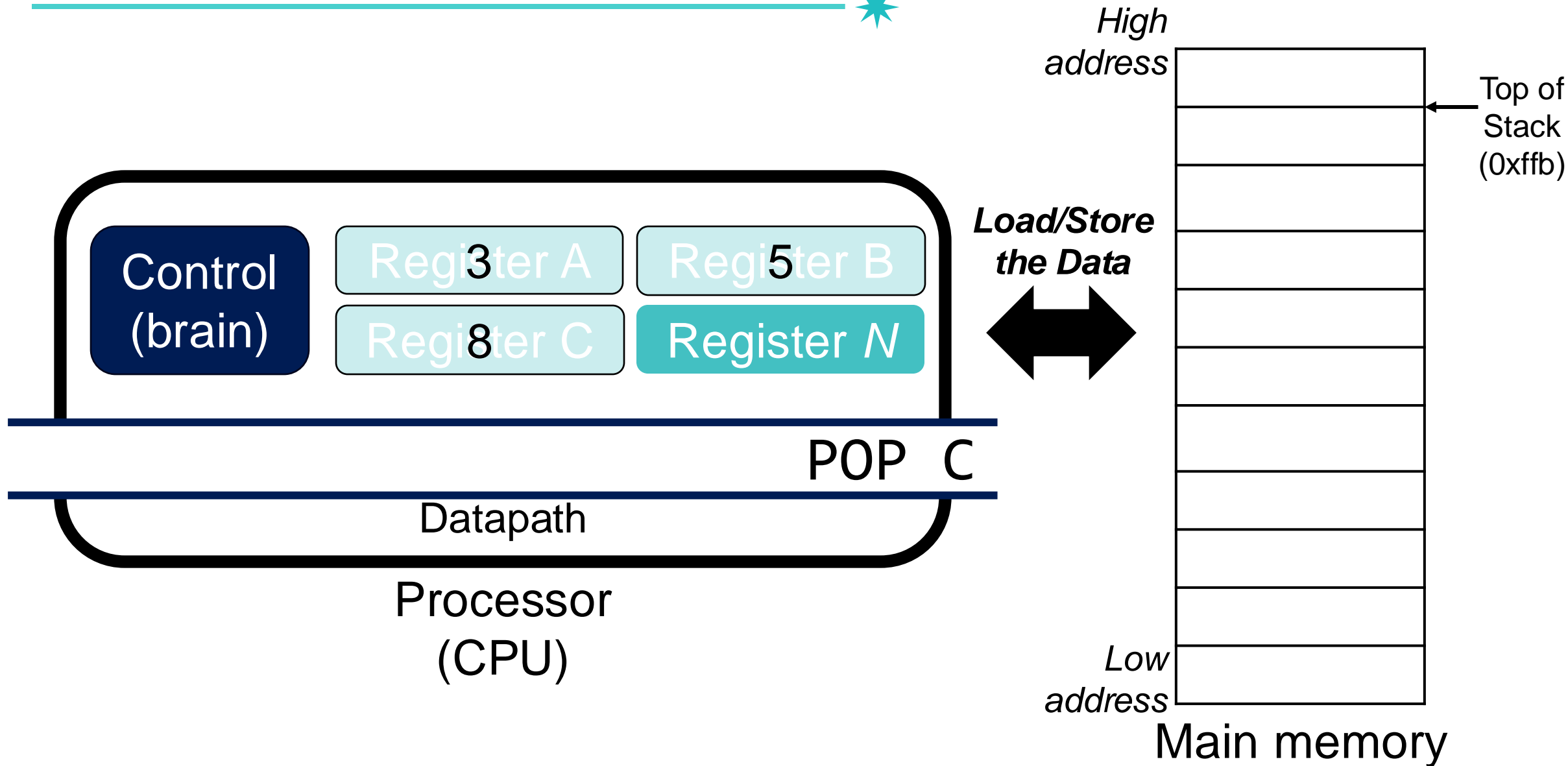


Zero Operand Example: Execution

62



Zero Operand Example: Execution



Zero Operand Example *

PUSH A

PUSH B

ADD

POP C

Meaning: $C \leftarrow A + B$

The Number of Operand Field

Goal: $A \leftarrow B + C$

Three operands:

```
add a, b, c
```

Two operands:

```
mov z, b  
add z, c  
mov a, z
```

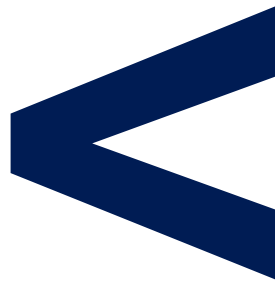
The Number of Operand Field

The number of instructions?

Three operands:

```
add a, b, c
```

of instructions ↓
⇒ Performance ↑




Two operands:

```
mov z, b  
add z, c  
mov a, z
```

Recall: CPU Time

67



$$\text{CPU Time} = \boxed{\frac{\text{Instructions}}{\text{Program}}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$


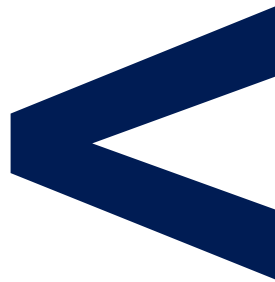
The Number of Operand Field

The number of instructions?

Three operands:

```
add a, b, c
```

of instructions ↓
⇒ Performance ↑



Two operands:

```
mov z, b  
add z, c  
mov a, z
```



Okay, then wouldn't it be good to always design instructions with a lot of operands?

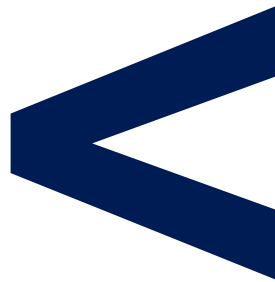
The Number of Operand Field

The number of instructions?

Three operands:

```
add a, b, c
```

Hardware complexity ↑
⇒ Performance ↓



Two operands:

```
mov z, b  
add z, c  
mov a, z
```

Recall: CPU Time



$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \boxed{\frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}} \uparrow$$

Addressing Modes

Addressing Modes



Specify how an operand is interpreted to derive an *effective address*¹⁾

¹⁾ *Effective address*: actual address of the location containing the referenced operand

Types of Addressing Modes



- **Immediate** mode
- **Register (direct)** mode
- **Register indirect** mode
- **Direct** mode
- **Indirect** mode
- **(PC)-relative** mode
- **Base register** mode
- ...

Types of Addressing Modes



- **Immediate** mode
- Register (direct) mode
- Register indirect mode
- Direct mode
- Indirect mode
- (PC)-relative mode
- Base register mode
- ...

Addressing Mode: Immediate Mode



Operand field contains the **actual operand value**

Example: MIPS instruction

`addi $s1, $s2, 17`



Operand value

Types of Addressing Modes



- **Immediate** mode
- Register (direct) mode
- Register indirect mode
- Direct mode
- Indirect mode
- (PC)-relative mode
- Base register mode
- ...

Types of Addressing Modes



- **Immediate** mode
- **Register (direct)** mode
- **Register indirect** mode
- Direct mode
- Indirect mode
- (PC)-relative mode
- Base register mode
- ...

Addressing Mode: Register Mode

Selected register contains the operand

- *Effective address*: selected register
- Operand value: selected register's value

Example: MIPS instruction

add \$s1, \$s2, \$s3

10

Register \$s3

- Effective address: \$s3
- Operand value: 10

Effective address: actual address of the location containing the referenced operand

Addressing Mode: Register Indirect Mode ⁷⁹

Selected register contains the address of operands

- *Effective address*: selected register's value
- Operand value: memory[effective address]

Example: Motorola 68000

MOVE .W (A1), D1

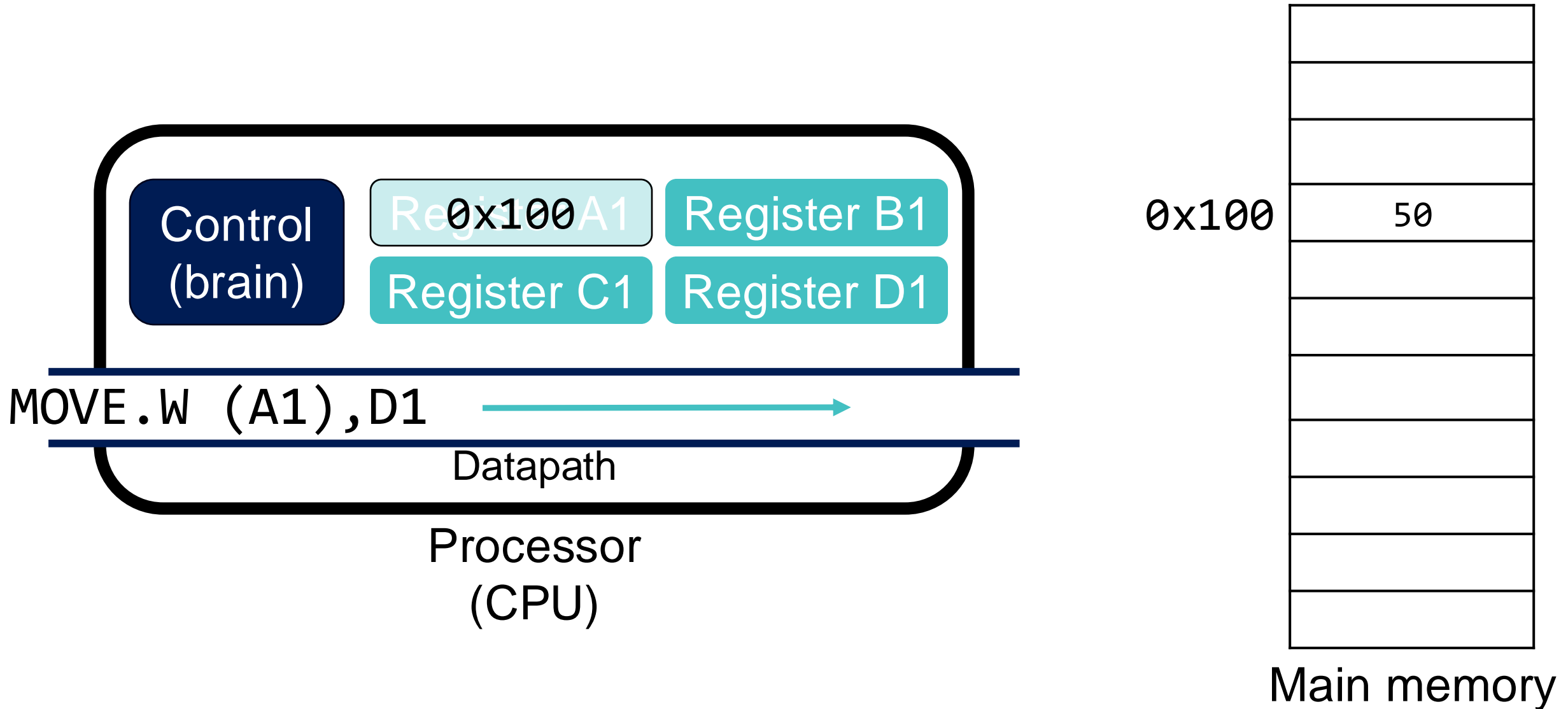
Do not memorize the meaning or syntax of the instructions!
Focus on understanding what the register indirect mode is.

Source

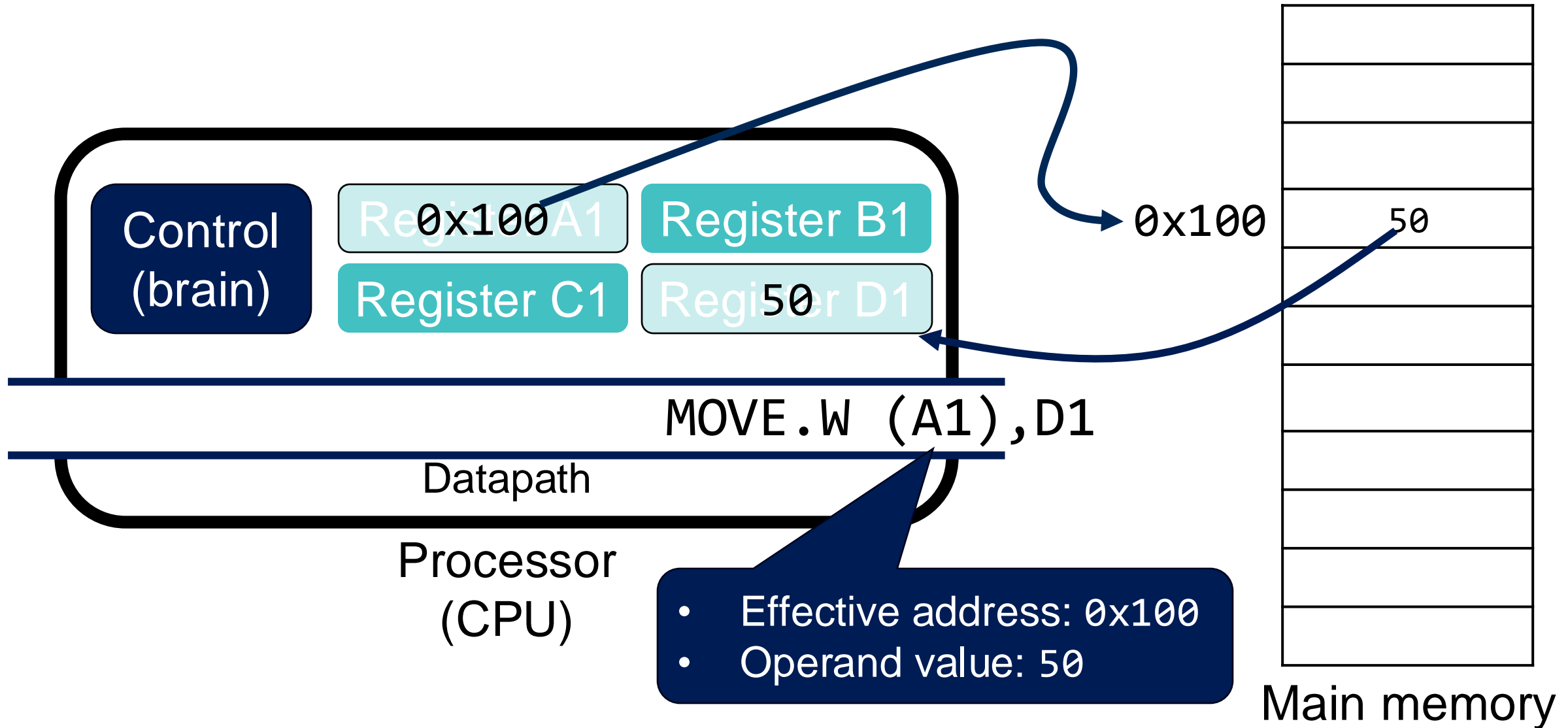
Destination

Effective address: actual address of the location containing the referenced operand

Register Indirect Mode Example: Initial State



Register Indirect Mode Example: After the Exe.



Types of Addressing Modes



- **Immediate** mode
- **Register (direct)** mode
- **Register indirect** mode
- Direct mode
- Indirect mode
- (PC)-relative mode
- Base register mode
- ...

Types of Addressing Modes



- **Immediate** mode
- **Register (direct)** mode
- **Register indirect** mode
- **Direct** mode
- **Indirect** mode
- (PC)-relative mode
- Base register mode
- ...

Addressing Mode: Direct Mode



Effective address is equal to **the address part of the instruction**

- *Effective address*: the address part of instruction
- Operand value: memory[effective address]

Example: Motorola 68000

MOVE.W 0x100, D1

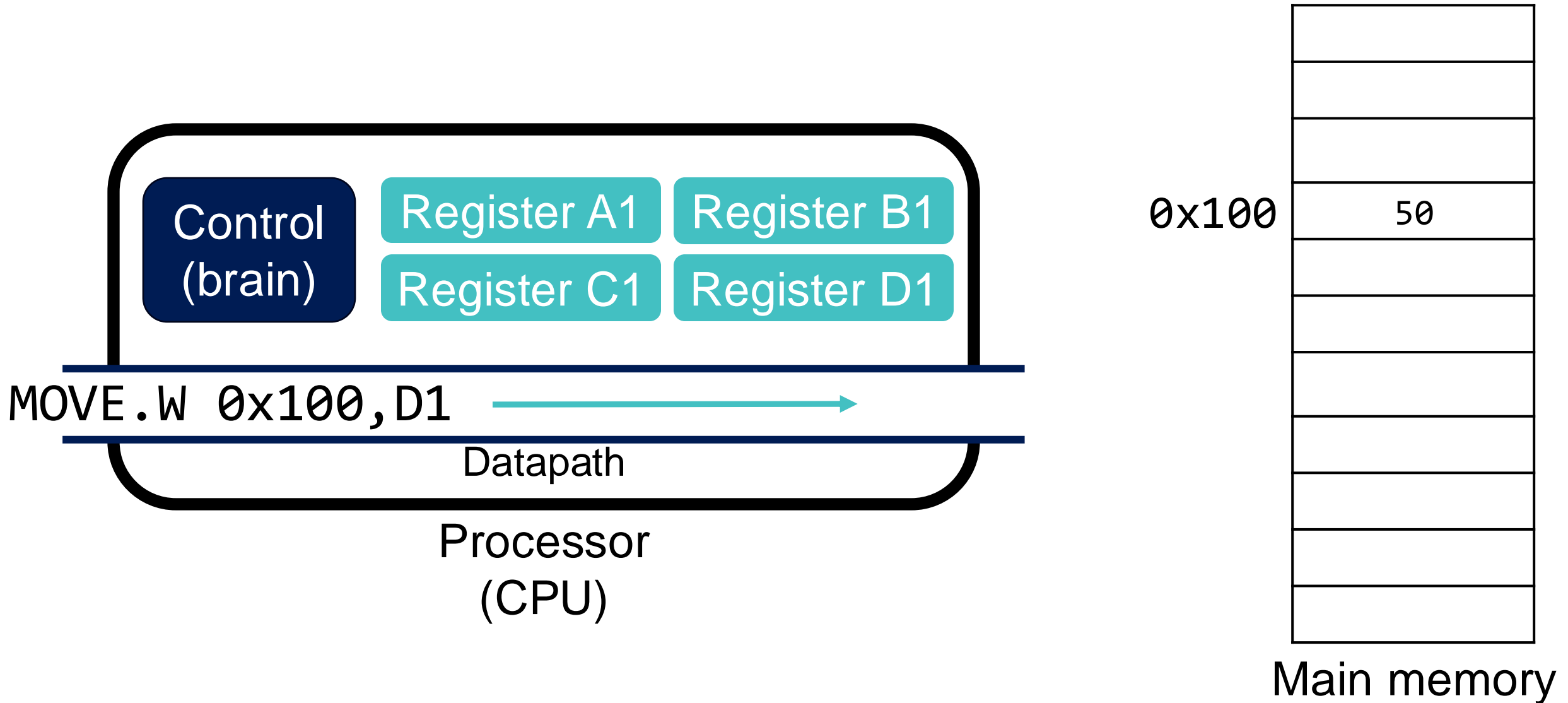


Source



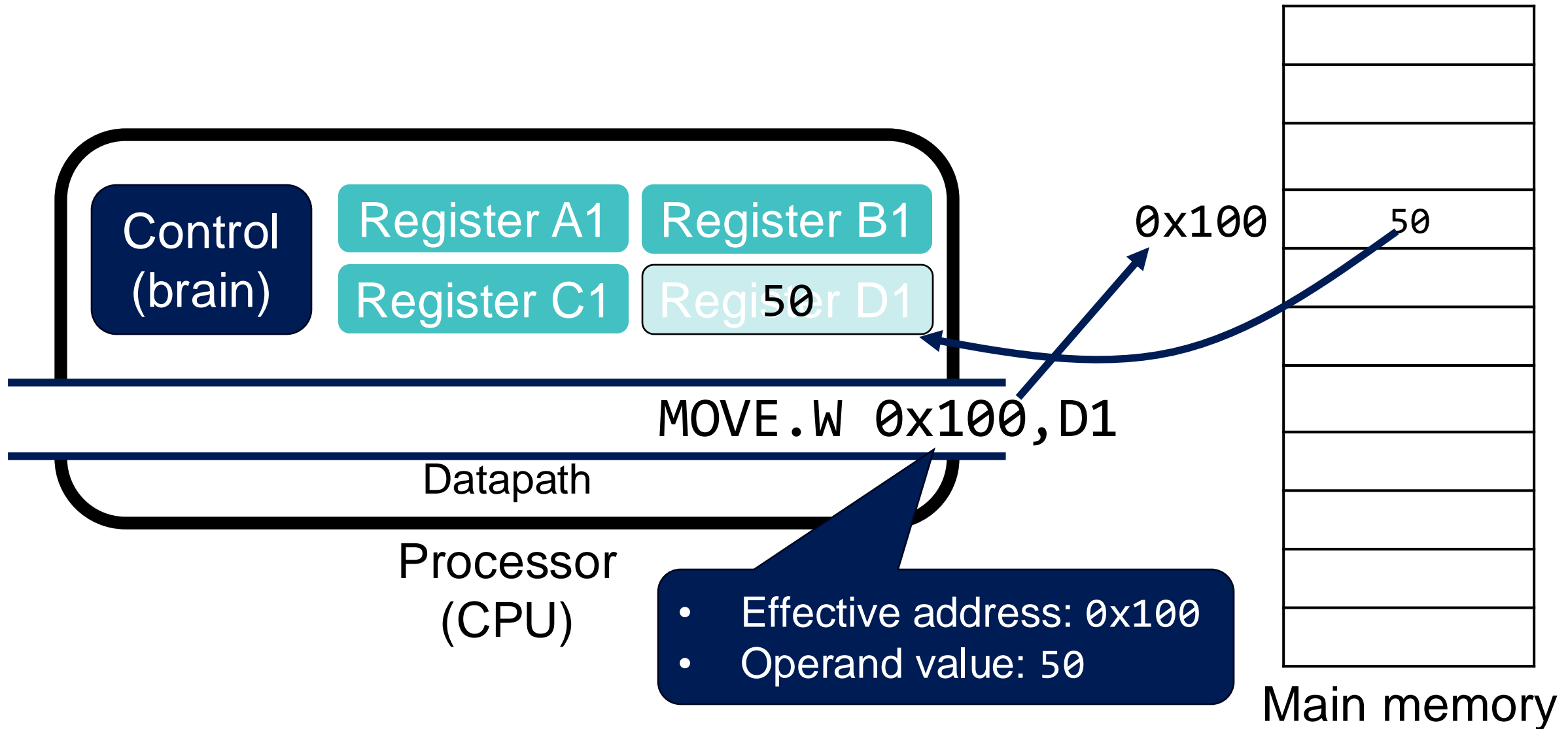
Destination

Direct Mode Example: Initial State



Direct Mode Example: After the Execution

86



Addressing Mode: Indirect Mode



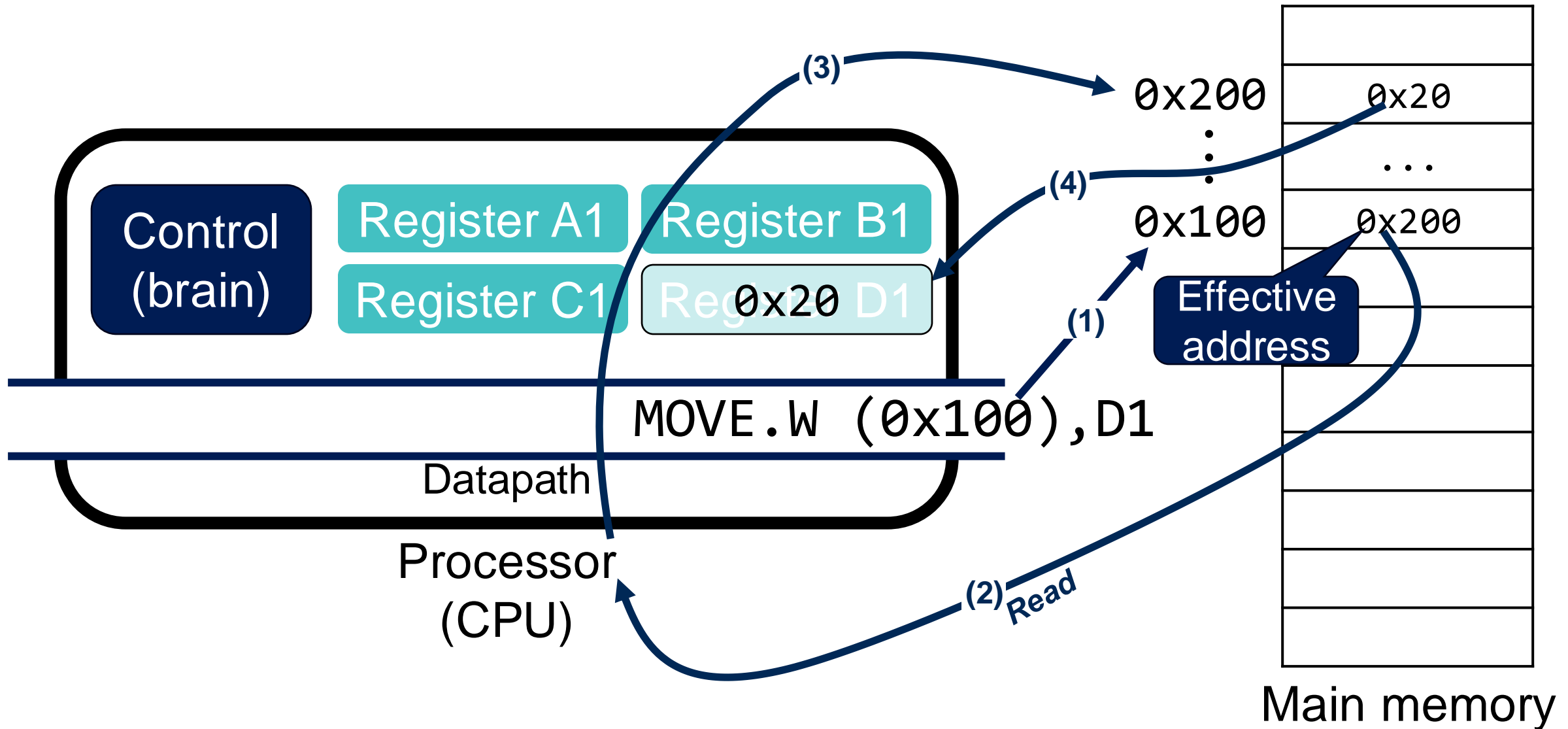
The address field give the address **where the effective address is stored**

- *Effective address*: memory[the address part of instruction]
- Operand value: memory[effective address]

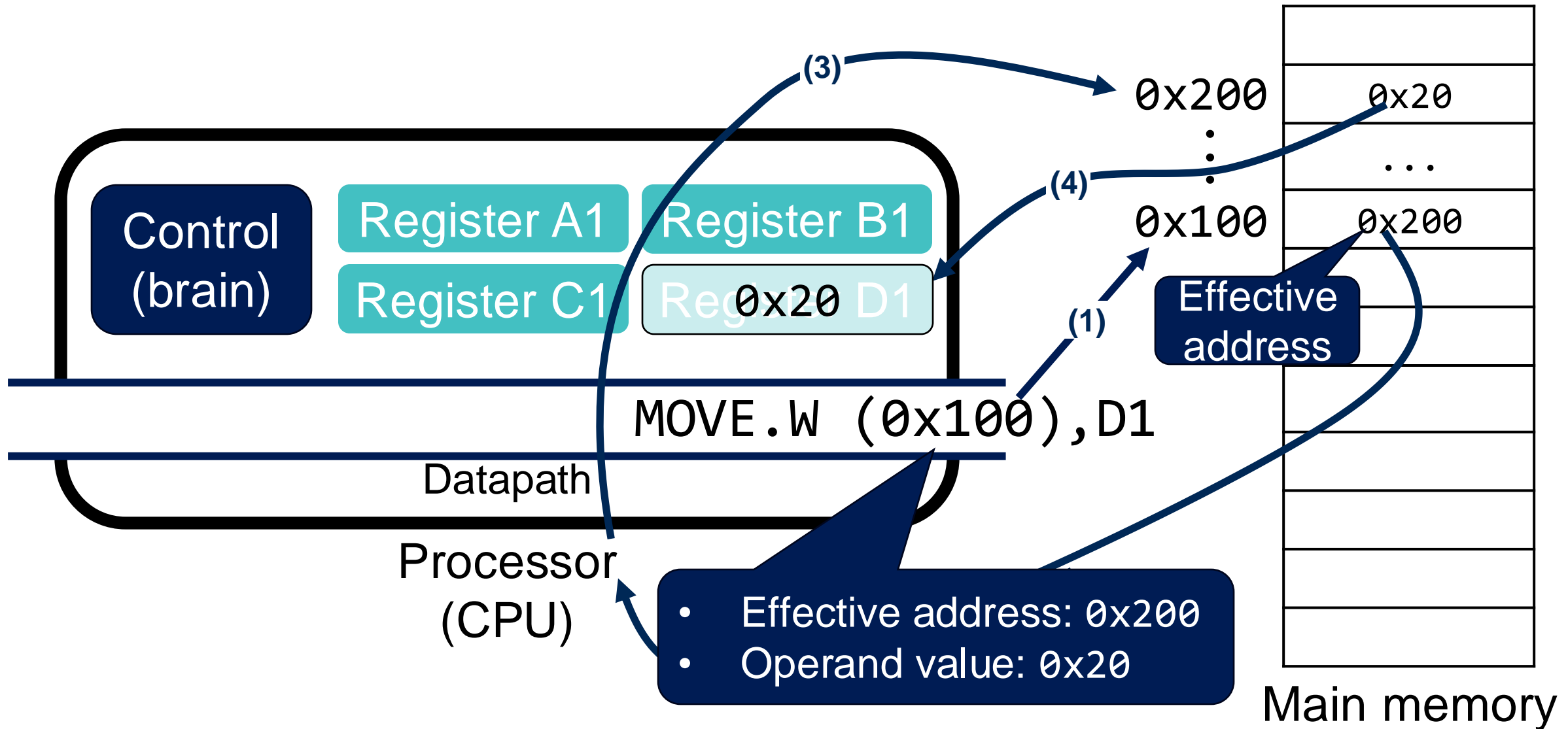
Example: hypothetical machine

MOVE .W (0x100), D1

Indirect Mode Example: After the Execution



Indirect Mode Example: After the Execution



Types of Addressing Modes



- **Immediate** mode
- **Register (direct)** mode
- **Register indirect** mode
- **Direct** mode
- **Indirect** mode
- (PC)-relative mode
- Base register mode
- ...

Types of Addressing Modes



- **Immediate** mode
- **Register (direct)** mode
- **Register indirect** mode
- **Direct** mode
- **Indirect** mode
- **(PC)-relative** mode
- **Base register** mode
- ...

Addressing Mode: PC-Relative Mode

The content of PC is added to the address part of instruction to obtain the *effective address* (branch type instructions)

- *Effective address*: PC + the address part of instruction*4
- Operand value: memory[*effective address*]

Example: MIPS instruction

beq \$t0, \$zero, else

$$\text{Effective address} = \text{Register PC} + \text{Address field value} \times 4$$


Addressing Mode: PC-Relative Mode

The content of PC is added to the address part of instruction to obtain the *effective address* (branch type instructions)

- *Effective address*: PC + the address part of instruction*4
- Operand value: memory[effective address]

Example: MIPS instruction

beq \$t0, \$zero, else

Why x4?

$$\text{Effective address} = \text{Register PC} + \text{Address field value} \times 4$$

200

PC-Relative Mode Example



```
    slt $t0, $s0, $s1
    beq $t0, $zero, else
    add $s2, $s0, $s1
    j  exit
else: sub $s2, $s0, $s1
    ...
exit:
```

PC-Relative Mode Example

set less than

\$t0=1, if \$s0 < \$s1

```
slt $t0, $s0, $s1
```

```
beq $t0, $zero, else
```

```
add $s2, $s0, $s1
```

```
j exit
```

```
else: sub $s2, $s0, $s1
```

```
...
```

```
exit:
```


PC-Relative Mode Example

set less than

\$t0=1, if \$s0 < \$s1

```
slt $t0, $s0, $s1  
beq $t0, $zero, else
```

```
add $s2, $s0, $s1  
exit
```

branch if equal

Go to else if \$t0 == 0

...

exit:

PC-Relative Mode Example

set less than

\$t0=1, if \$s0 < \$s1

```
slt $t0, $s0, $s1  
beq $t0, $zero, else
```

```
add $s2, $s0, $s1  
exit
```

branch if equal

Go to else if \$t0 == 0

if \$s0 < \$s1
a
else:
b

...

exit:

PC-Relative Mode Example

set less than

\$t0=1, if \$s0 < \$s1

```
slt $t0, $s0, $s1
beq $t0, $zero, else
```

```
add $s2, $s0, $s1
exit
```

branch if equal

Go to else if \$t0 == 0

```
if $s0 < $s1
    a
else:
    b
```

exit: ...
Why not single blt (branch less than) instruction?

PC-Relative Mode Example

set less than

\$t0=1, if \$s0 < \$s1

```
slt $t0, $s0, $s1
beq $t0, $zero, else
```

```
add $s2, $s0, $s1
exit
```

branch if equal

Go to else if \$t0 == 0

```
if $s0 < $s1
    a
else:
    b
```

exit: *Why not single blt (branch less than) instruction?*

- Hardware for <, ≥, ... slower than =
- Combining with branch involves more hardware-based work per instruction

Recall: RISC vs. CISC



- **Reduced Instruction Set Computer (RISC)**

- Example: MIPS, ARM, PowerPC
- Small and simple instruction set => **Simple hardware**
- Fixed-size instruction format

Example instruction set:

`slt, beq(=), bne(≠)`

- **Reduced Instruction Set Computer (CISC)**

- Example: Intel x86, AMD
- A large number of instruction set => **Complex hardware**
- Variable-size instruction format

Example instruction set:

`slt, beq(=), bne(≠)`

`bge(≥), bgt(>), ble(≤), blt(<)`

Recall: RISC vs. CISC



- **Reduced Instruction Set Computer (RISC)**

- Example: MIPS, ARM, PowerPC
- Small and simple instruction set => **Simple hardware**
- Fixed-size instruction format

Example instruction set:

`slt, beq(=), bne(≠)`

*We can cover all sets with
the combination of the smallest set*

- **Reduced Instruction Set Computer (CISC)**

- Example: Intel x86, AMD
- A large number of instruction set => **Complex hardware**

More hardware to realize
this instruction set
→ clock cycle period ↑

Example instruction set:

`slt, beq(=), bne(≠)
bge(≥), bgt(>), ble(≤), blt(<)`

PC-Relative Mode Example



```
    slt $t0, $s0, $s1
    beq $t0, $zero, else #if $s0<$s1
    add $s2, $s0, $s1
    j  exit
else: sub $s2, $s0, $s1 #else
    ...
exit:
```

Path #1

Path #2

PC-Relative Mode Example

Assembly language



Machine language

```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0<$s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
...
exit:

```

```

10101010 00110010 01000000 00101010
00010001 00000000 00000000 00000010
...      ...      ...      ...
...      ...      ...      ...
00000100  ...
...

```


PC-Relative Mode Example

Assembly language



Machine language

```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0<$s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
...
exit:

```

```

10101010 00110010 01000000 00101010
00010001 00000000 00000000 00000010
...      ...      ...      ...
...      ...      ...      ...
00000100 ...
.

```

opcode

PC-Relative Mode Example

Assembly language

```
slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0<$s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
...
exit:
```



Machine language

```
10101010 00110010 01000000 00101010
00010001 00000000 00000000 00000010
...      ...      ...      ...
...      ...      ...      ...
00000100 ...
.
```

opcode

There is no specific machine code for labels

PC-Relative Mode Example

Assembly language



Machine language

```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0<$s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
...
exit:

```

```

10101010 00110010 01000000 00101010
00010001 00000000 00000000 00000010

```

...

...

..

...

...

..

Address field
(why 2?)

```

00000100

```

..

.

opcode

PC-Relative Mode Example

Assembly language

```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0<$s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
...
exit:
  
```

2 (relative offset)

Assembler will determine the value of the address field

Machine language

```

10101010 00110010 01000000 00101010
00010001 00000000 00000000 00000010
... ..
... ..
00000100 ..
opcode
  
```

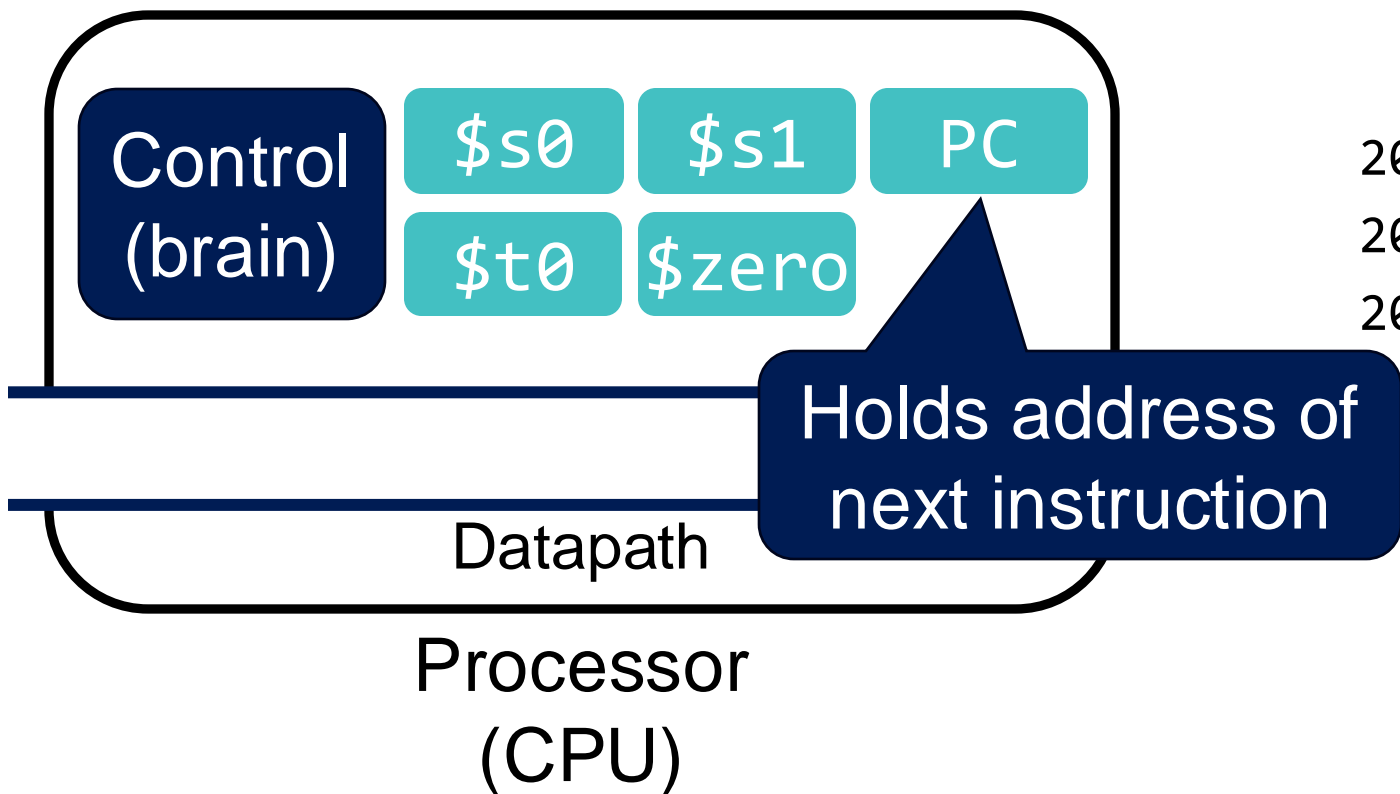
Address field (why 2?)

Now, Let's Execute the Program

Execute The Program



Machine code is loaded into memory (Code section)



	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208	...
212	...
216	00000100 ...
	...
	...

Main memory

```

    slt $t0, $s0, $s1
    beq $t0, $zero, else #if $s0<$s1
    add $s2, $s0, $s1
    j exit
else: sub $s2, $s0, $s1    #else
  
```

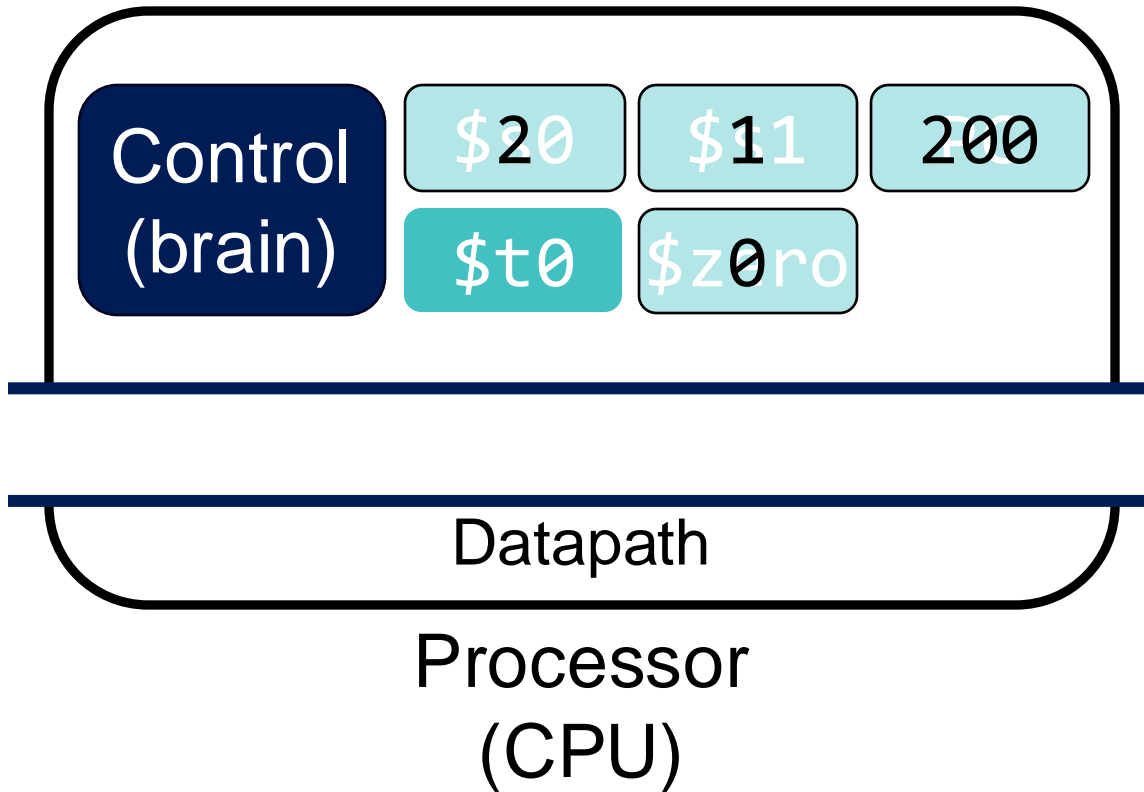
(Recall) Special Purpose Registers

- **PC (Program Counter):** holds address of next instruction
- **IR (Instruction Register):** holds the instruction fetched from the memory
- **AC (Accumulator):** holds the result of the computation temporarily

Execute The Program: Initial State

113

Machine code is loaded into memory (Code section)



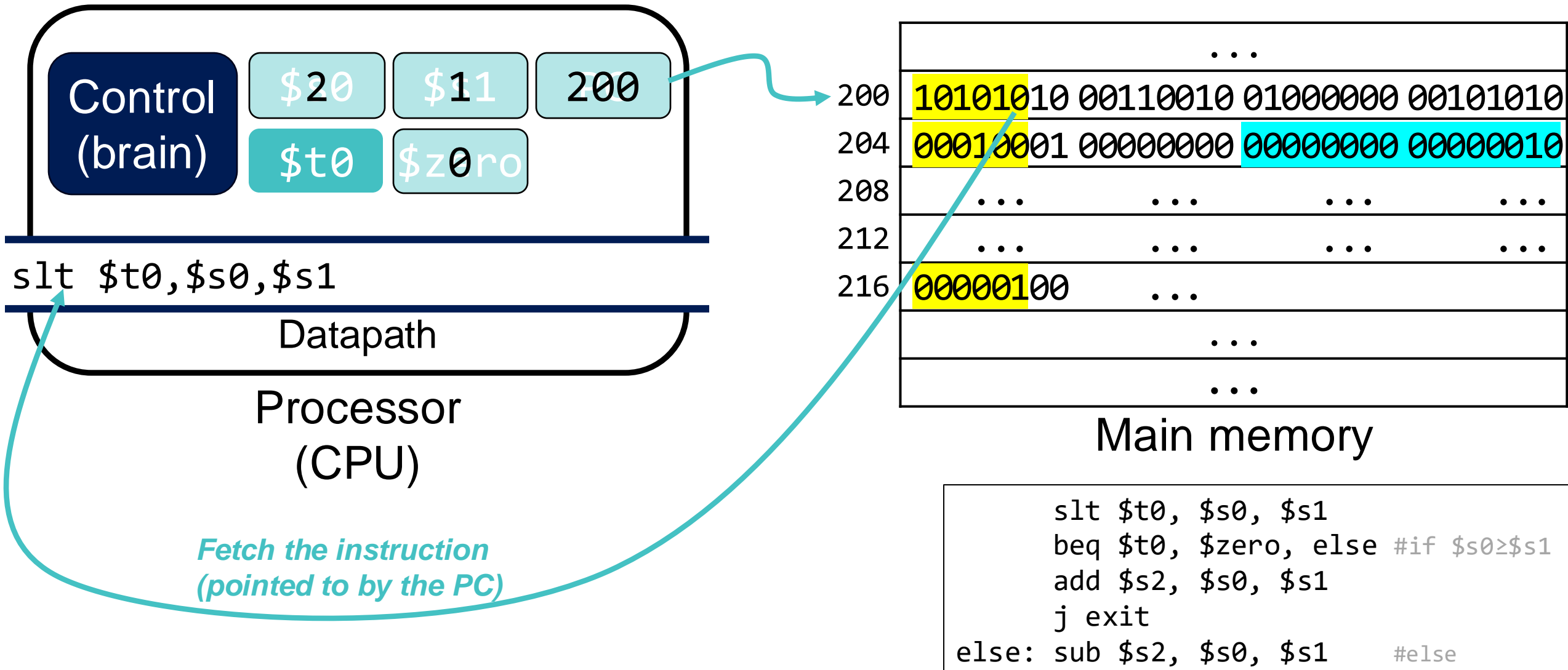
	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208	...
212	...
216	00000100 ...
	...
	...

Main memory

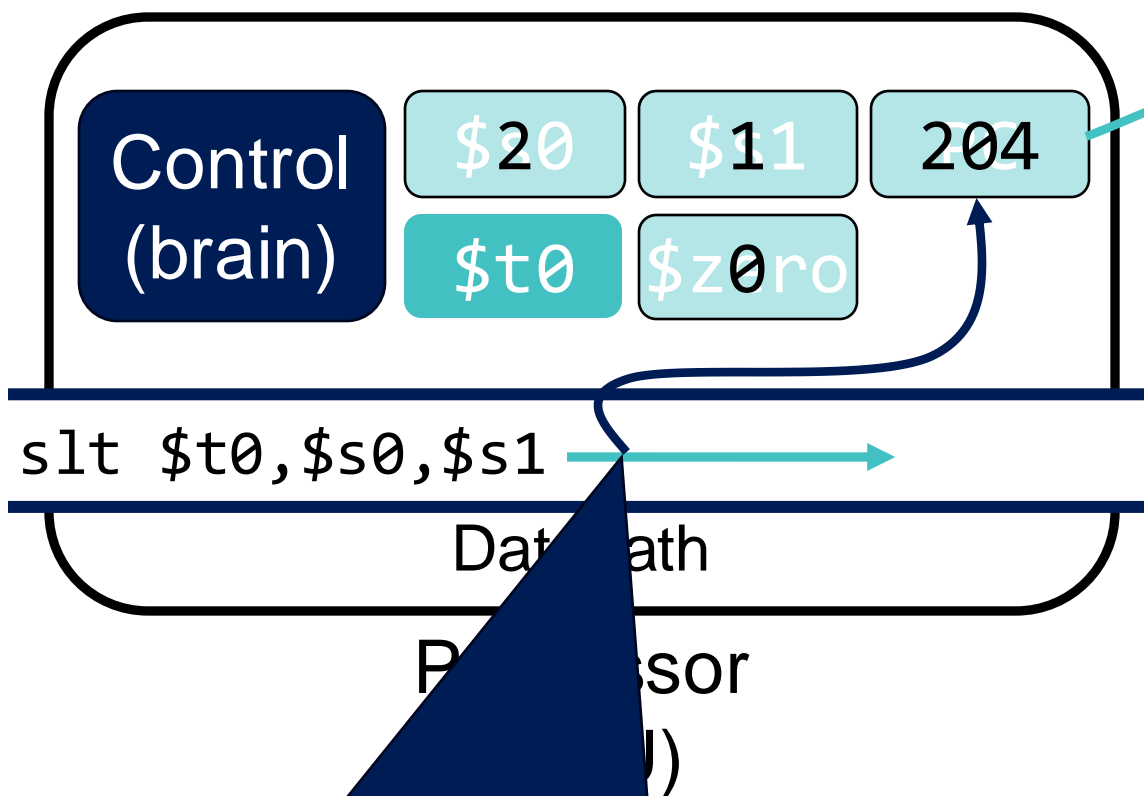
```
slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
```

Execute The Program: 1st Instruction

114



Execute The Program: 1st Instruction



	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208
212
216	00000100 ...
	...
	...

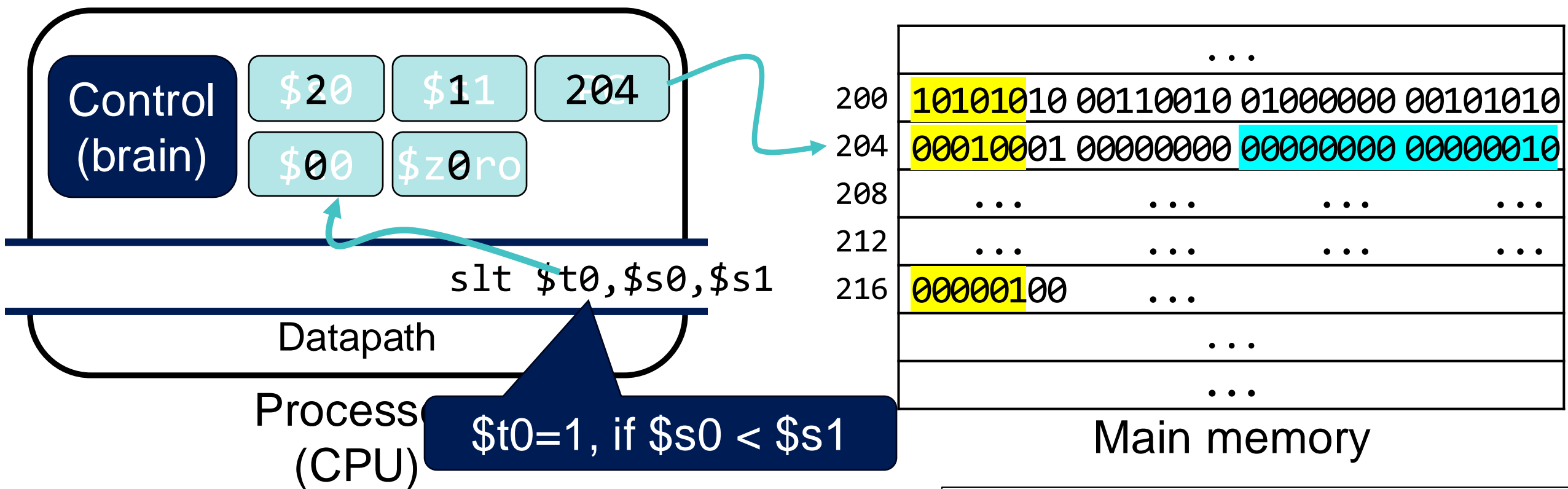
Main memory

In the early stage, the PC is incremented by 4 to point to the next instruction

```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
  
```

Execute The Program: 1st Instruction



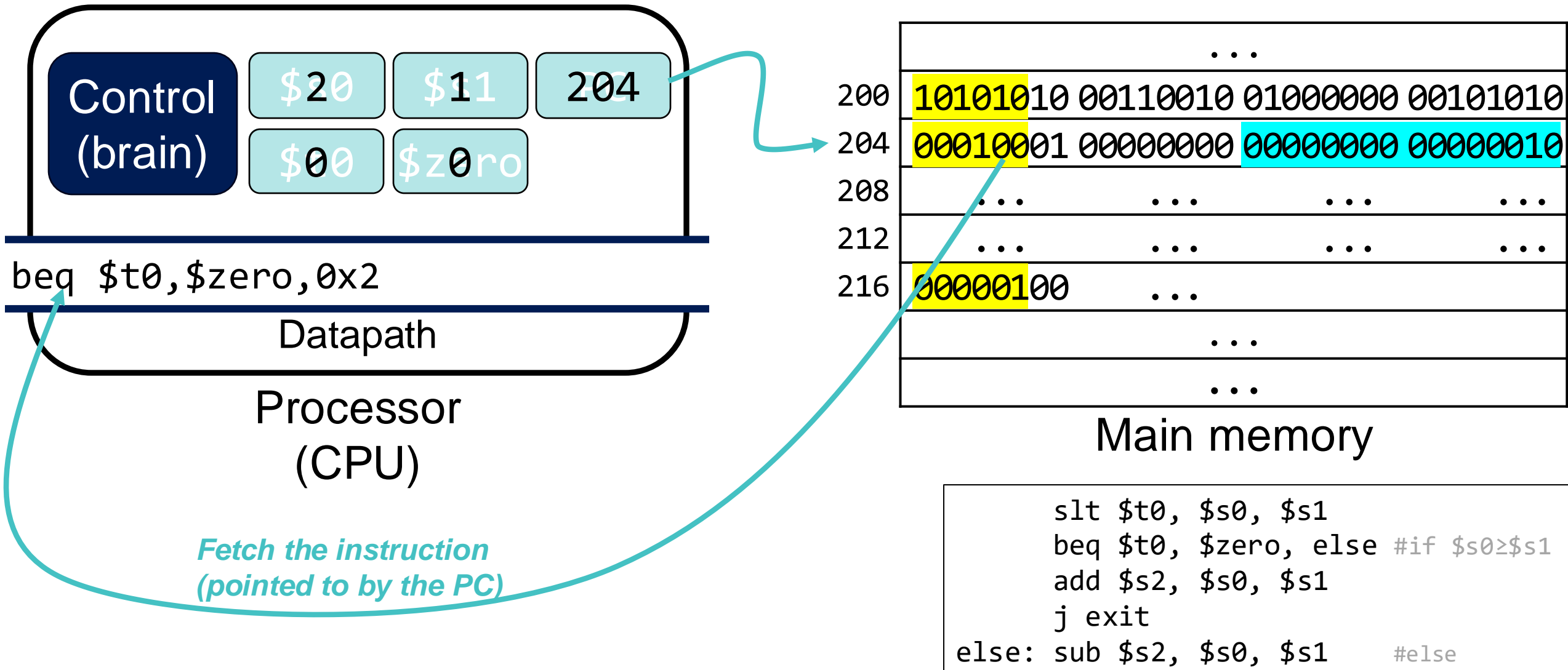
```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1      #else

```

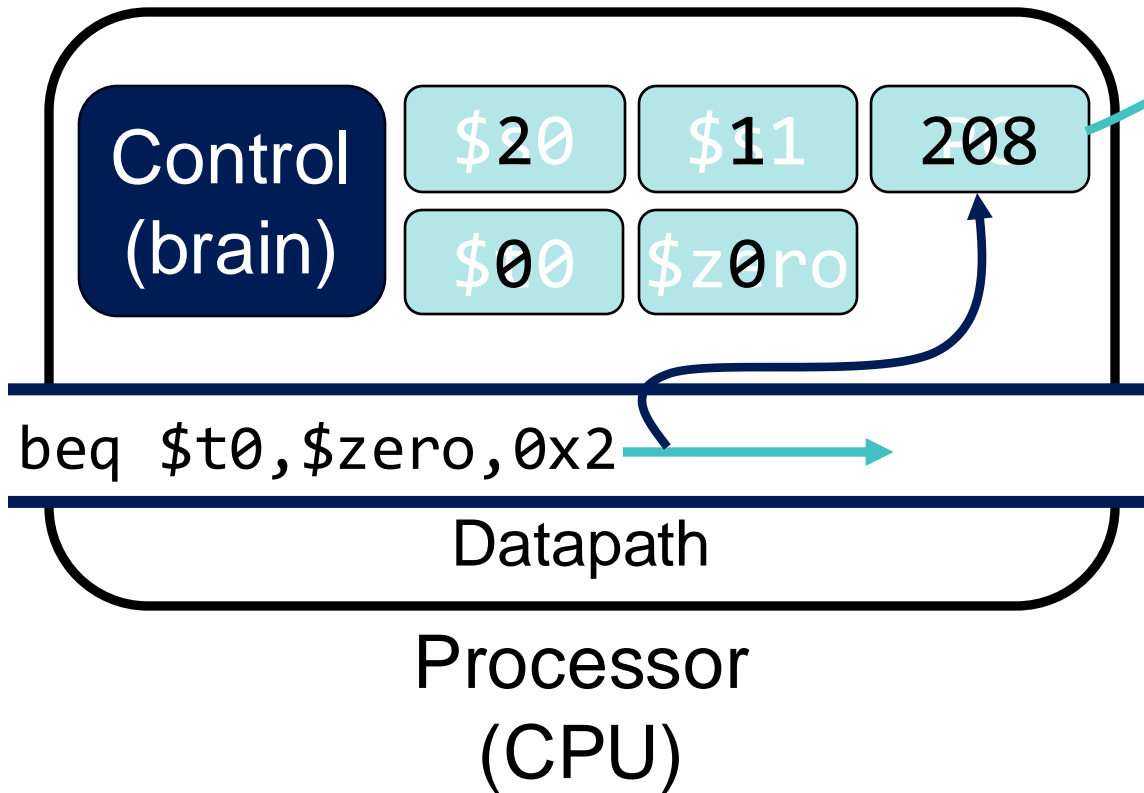
Execute The Program: 2nd Instruction

117



Execute The Program: 2nd Instruction

118

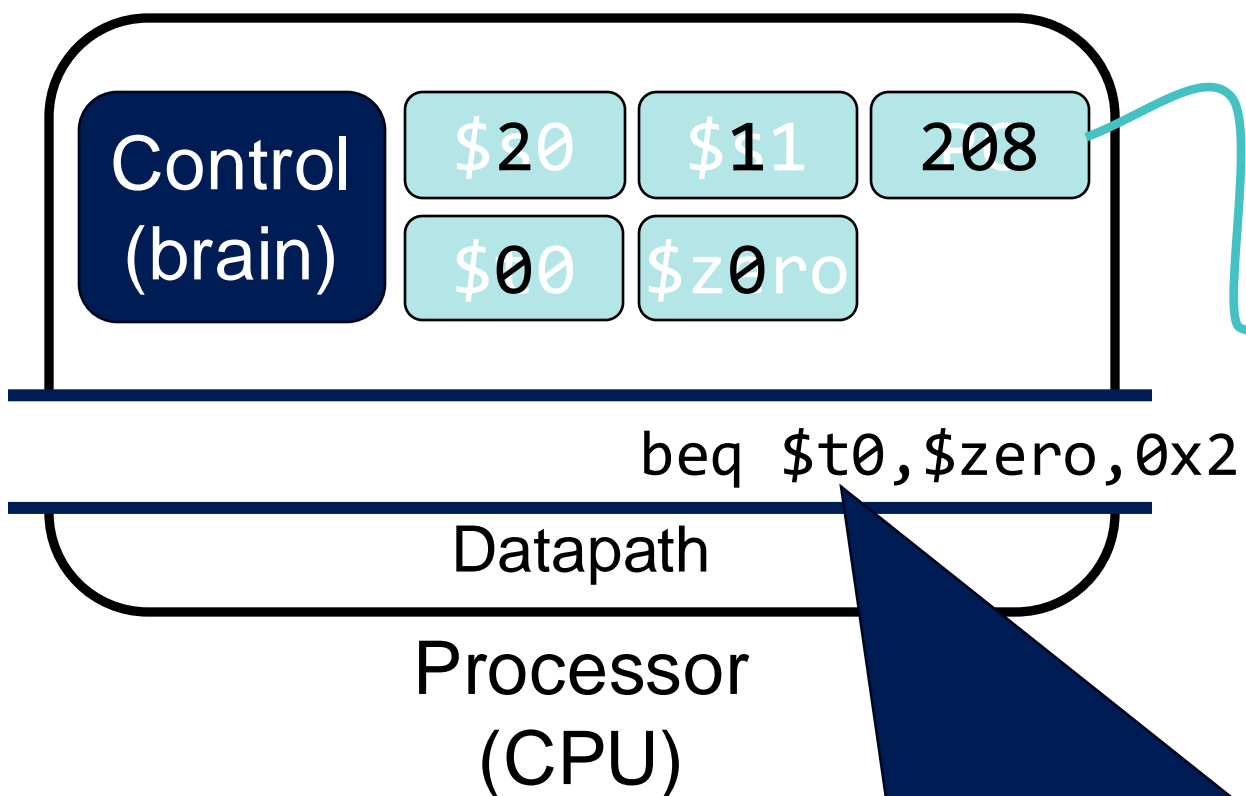


	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208	...
212	...
216	00000100 ...
	...
	...

Main memory

```
slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
```

Execute The Program: 2nd Instruction



	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208	...
212	...
216	00000100 ...
	...
	...

Main memory

$\$t0 == \$zero$

We need to go the else location. Let's update the PC value according to the PC-relative mode

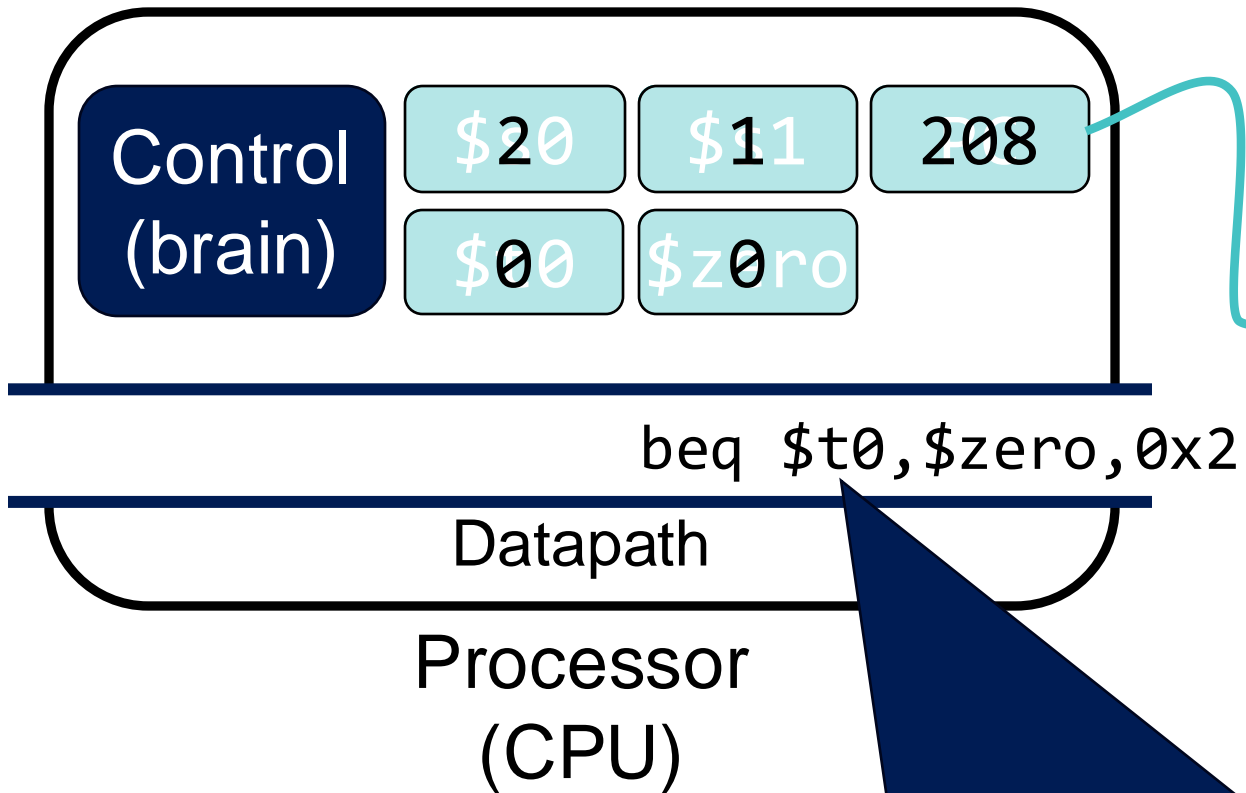
```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1      #else

```

Execute The Program: 2nd Instruction

120



	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208	...
212	...
216	00000100 ...
	...
	...

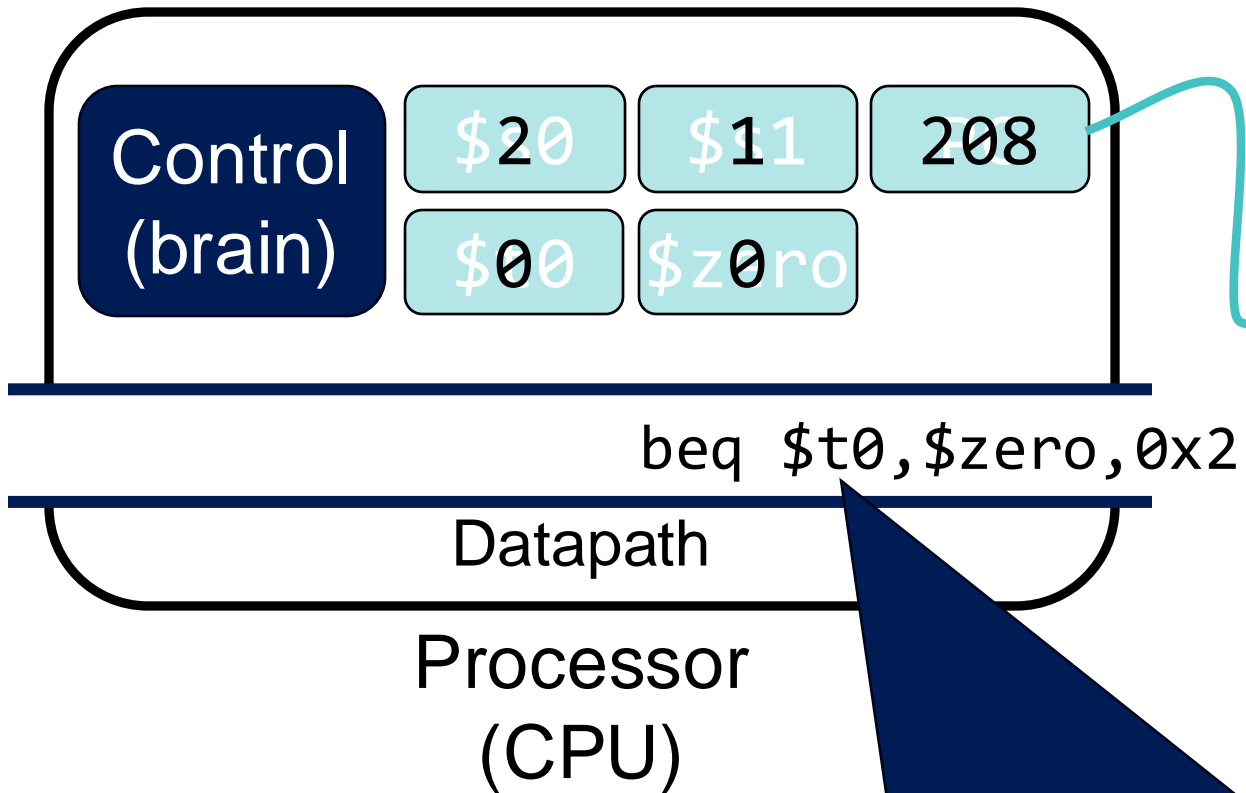
Main memory

PC-relative mode:
PC + address field of instruction*4

```
slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
```

Execute The Program: 2nd Instruction

12



	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208	...
212	...
216	00000100 ...
	...
	...

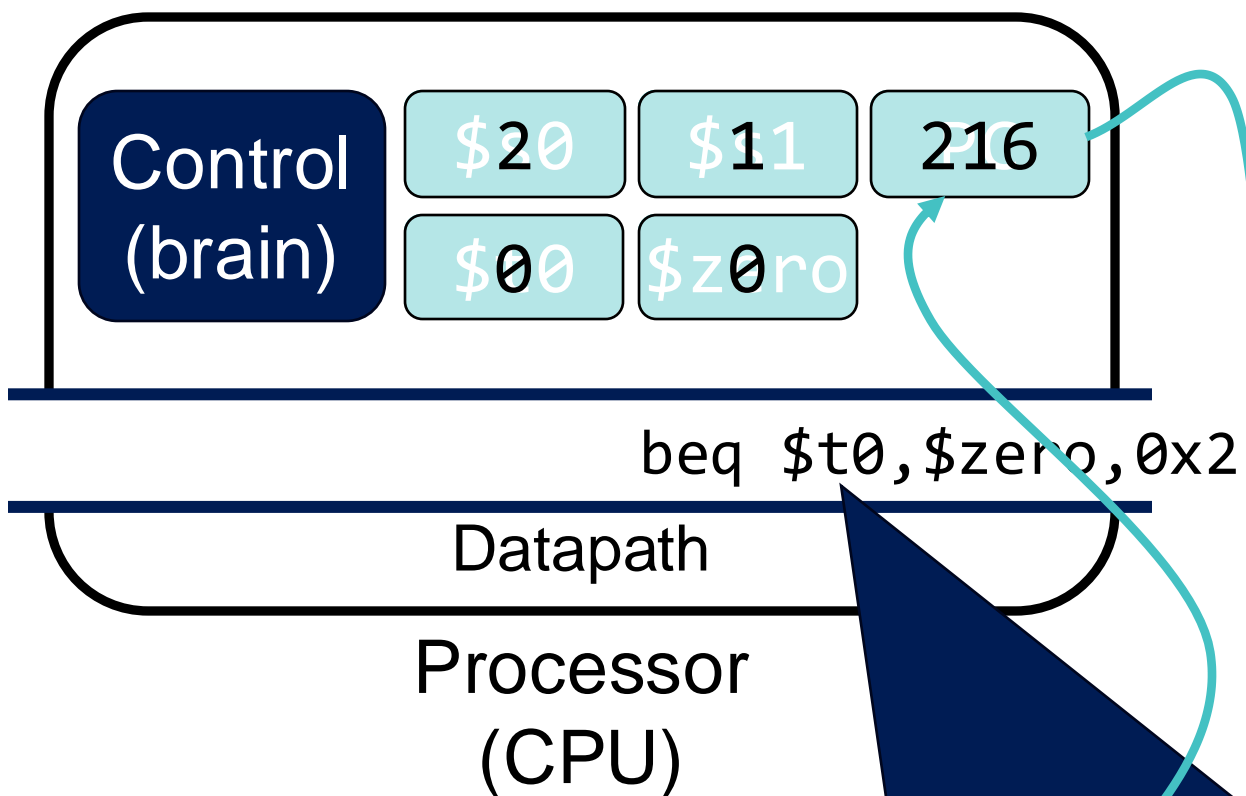
Main memory

PC-relative mode:

$$208 + 0x2 * 4 = 216$$

```
slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1 #else
```

Execute The Program: 2nd Instruction



	...
200	10101010 00110010 01000000 00101010
204	00010001 00000000 00000000 00000010
208
212
216	00000100 ...
	...
	...

Main memory

PC-relative mode:

$$208 + 0x2 * 4 = 216$$

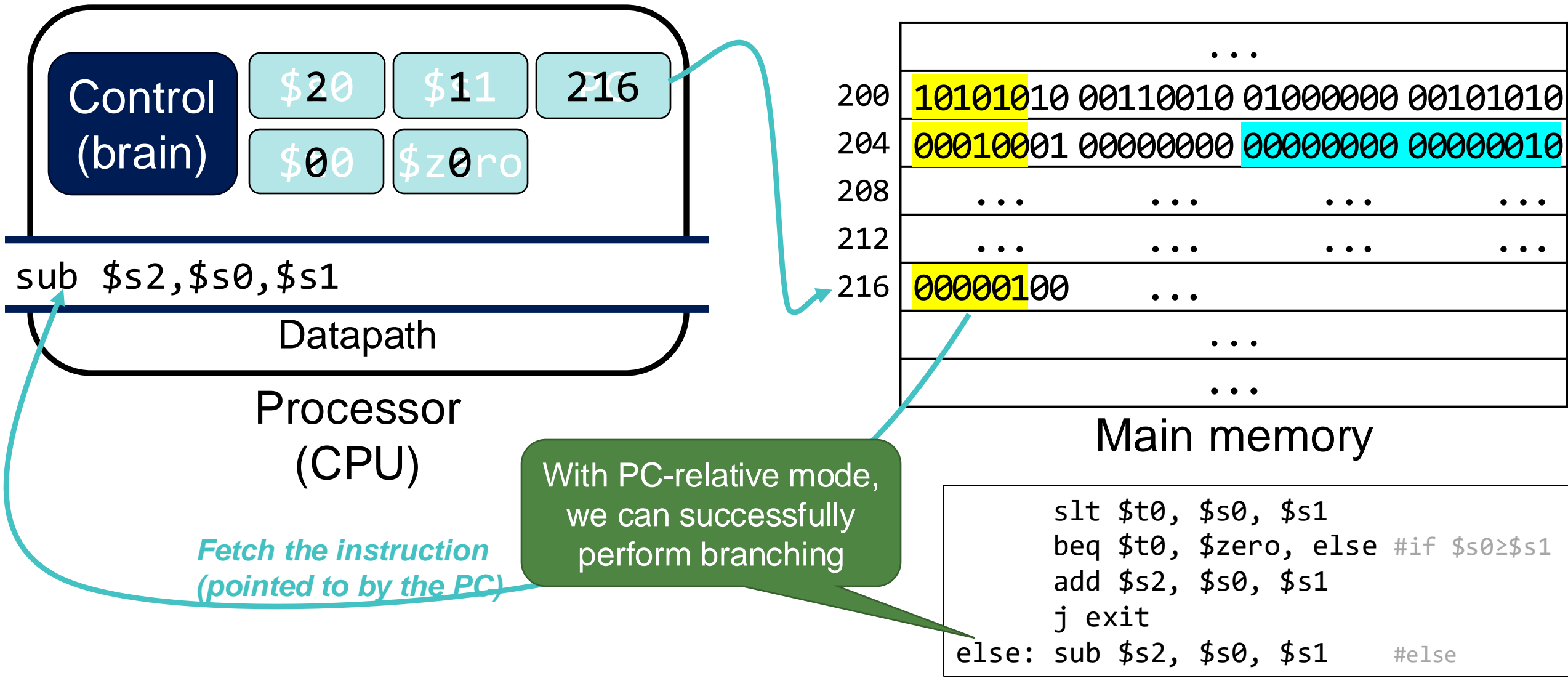
```

slt $t0, $s0, $s1
beq $t0, $zero, else #if $s0 ≥ $s1
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1    #else

```


Execute The Program: 3rd Instruction

123



Addressing Mode: Base Register Mode

124

The content of base register is added to the address part of instruction (offset) to obtain the effective address

- *Effective address*: Base register + offset field of instruction
- Operand value: memory[effective address]

Example: MIPS instruction

`lw $s0, 12($s1)`

Offset

Base register

125



The diagram illustrates a segment of main memory. It consists of a vertical stack of 11 memory cells. The left side of the stack is labeled with addresses, and the right side shows the data stored in each cell. The addresses are 0x100, 0x104, 0x108, and 0x10c, with the top of the stack labeled 'High address' and the bottom labeled 'Low address'. The data values are 0xf2ab, 0xe2, 0x142, and 0x23, respectively, with the bottom of the stack labeled 'Main memory'.

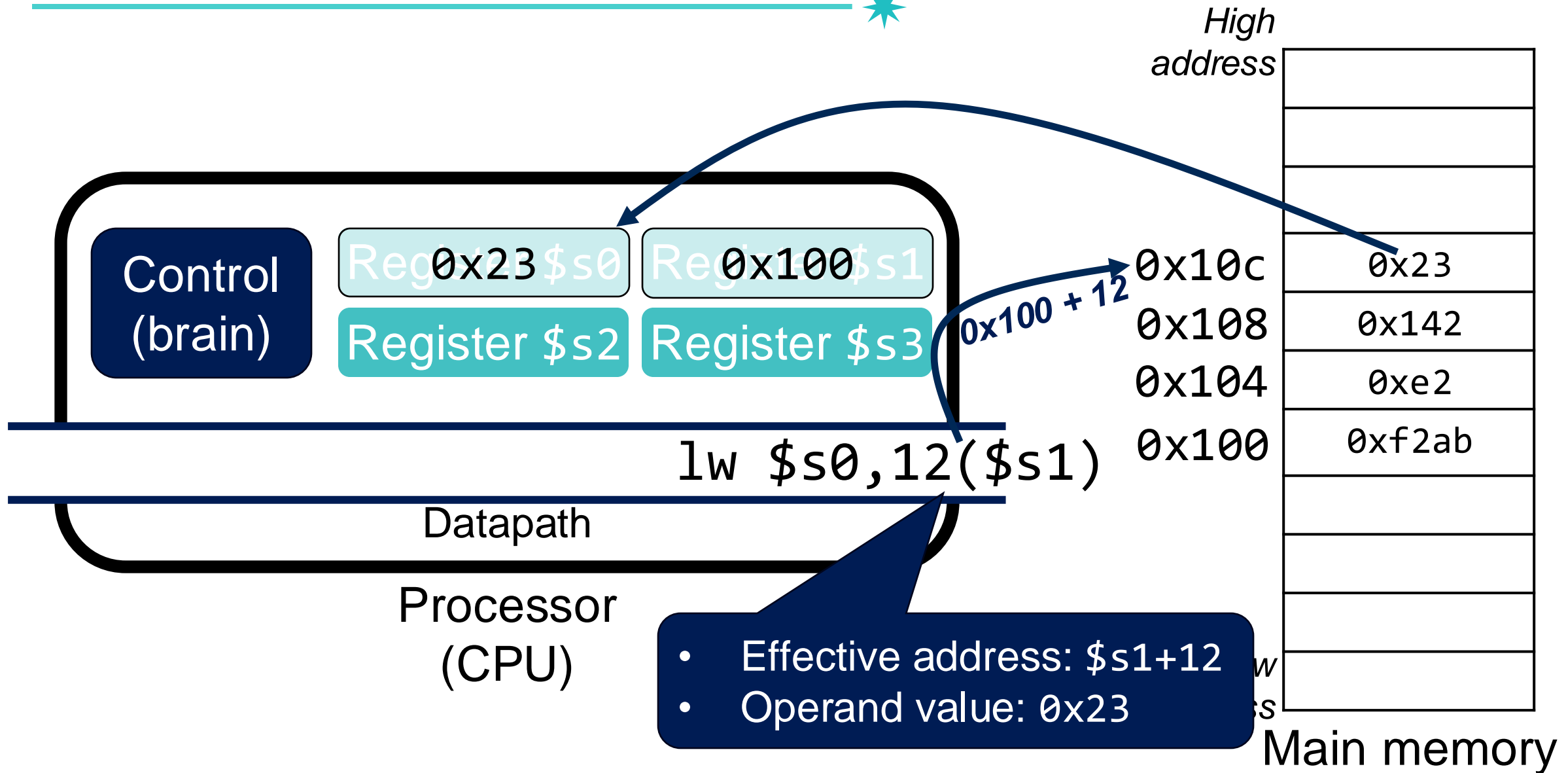
Address	Value
0x10c	0x23
0x108	0x142
0x104	0xe2
0x100	0xf2ab

High address

Low address

Main memory

Base Register Mode Example: After the Exe.



Base Register Mode is Useful to Access Arrays

2

C code

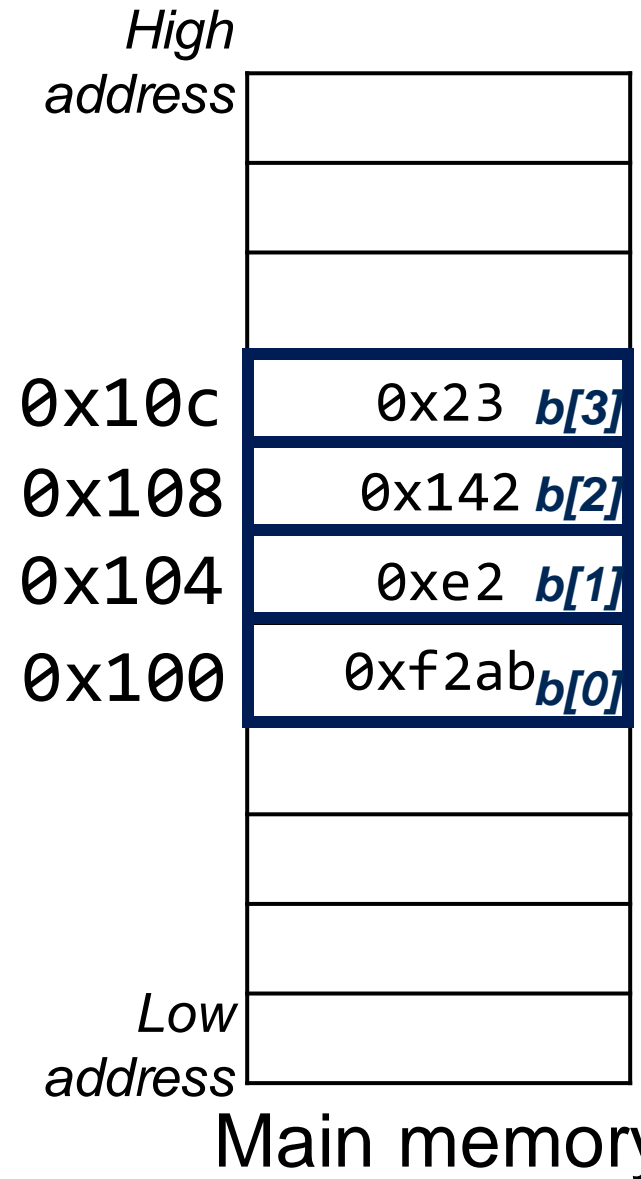
```
int a = b[3]
```

Compiled
MIPS code

```
lw $s0, 12($s1)
```

Index 3 requires
offset of 12 bytes

Base address of
b in \$s1 (0x100)

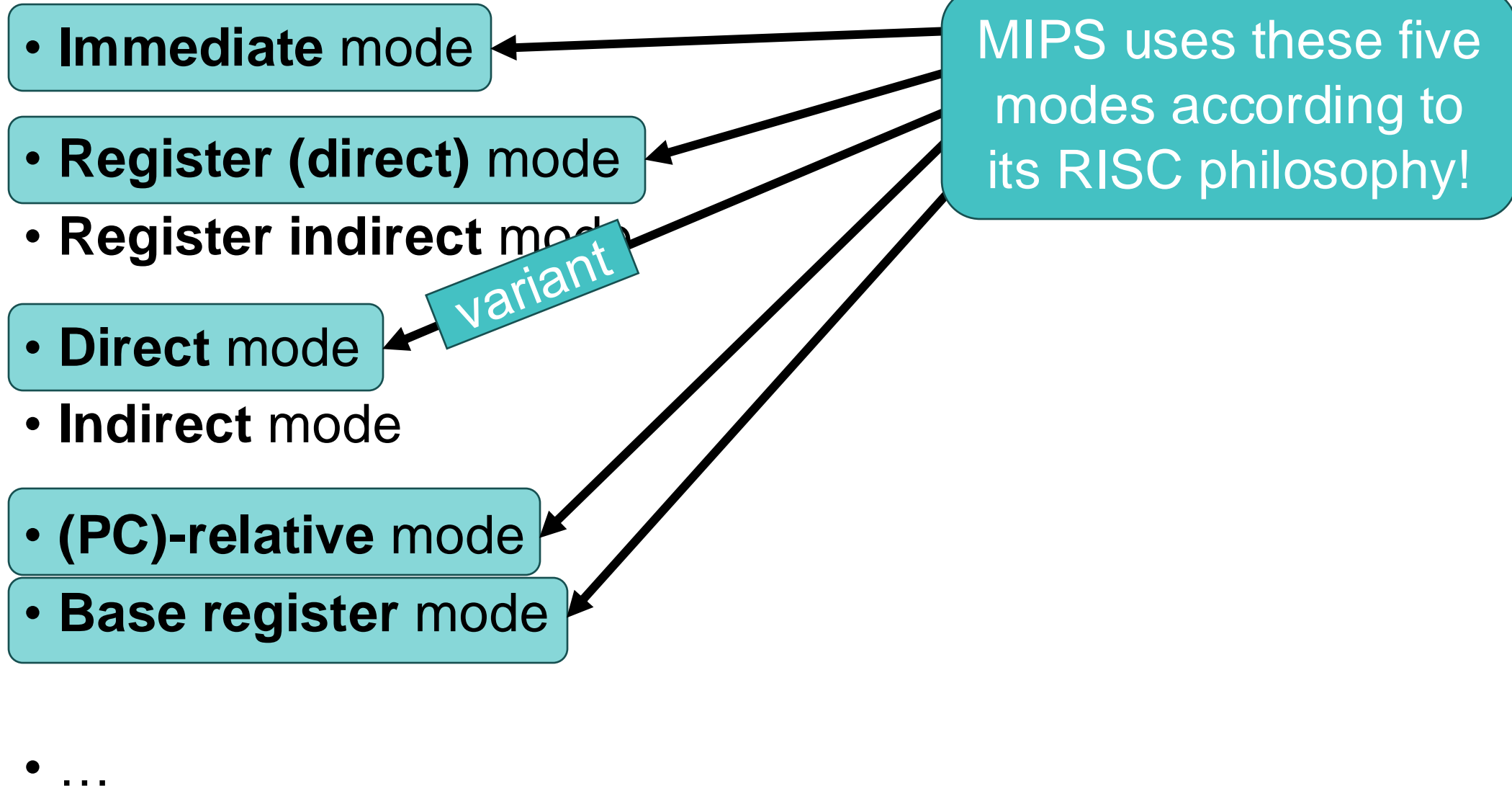


Types of Addressing Modes



- **Immediate** mode
- **Register (direct)** mode
- **Register indirect** mode
- **Direct** mode
- **Indirect** mode
- **(PC)-relative** mode
- **Base register** mode
- ...

Addressing Modes in MIPS



Recall: RISC vs. CISC



- **Reduced Instruction Set Computer (RISC)**
 - Example: MIPS, ARM, PowerPC
 - Small and simple instruction set => **Simple hardware**
 - Fixed-size instruction format
 - Limited addressing mode
- **Reduced Instruction Set Computer (CISC)**
 - Example: Intel x86, AMD
 - A large number of instruction set => **Complex hardware**
 - Variable-size instruction format
 - A large variety of addressing modes

Operation Types

Operation Types



Opcode	Operands
---------------	-----------------

- Arithmetic/Logic instructions (data operations)
 - Modify data values
- Data transfer instructions
 - Copy/move data from one place to another
- Control transfer instructions (program control)
 - Jump or Branch

Operation Types – Arithmetic/Logic



- Integer arithmetic operations
 - Addition, Subtract
 - Multiply, Divide
 - Increment, Decrement
- Logical operations
 - Bitwise AND, Bitwise OR
 - Complement (invert)
- Shift operations
 - Logic shift
 - Arithmetic shift
 - Circular shift
- Floating point arithmetic operations

Operation Types – Data Transfer



- Memory to register
 - E.g., `lw $s0, 12($s1)`
- Register to memory
 - E.g., `sw $s0, 12($s1)`
- Register to register
 - E.g., `add $s0, $s1, $s2`

Operation Types – Control Transfer



- Conditional branch instructions
- Unconditional branch instructions
- Subroutine calls and returns instructions
- Hardware interrupt instructions

Recall: RISC vs. CISC



- **Reduced Instruction Set Computer (RISC)**
 - Example: MIPS, ARM, PowerPC
 - Small and simple instruction set => **Simple hardware**
 - Fixed-size instruction format
 - Limited addressing mode
 - Need less opcode bits
- **Reduced Instruction Set Computer (CISC)**
 - Example: Intel x86, AMD
 - A large number of instruction set => **Complex hardware**
 - Variable-size instruction format
 - A large variety of addressing modes
 - Need more opcode bits and hardware

Signed and Unsigned Numbers

Unsigned Binary Integers

- Given an n-bit number

$$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $2^n - 1$
- Example

0000 0000 0000 0000 0000 0000 0000 1011₂

$$= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$$

- Using 32 bits ($n = 32$)
Range: 0 to +4,294,967,295



*How to represent
negative numbers?*

Approach #1: Sign Magnitude

14

$$001 = 1$$

$$101 = -1$$

Approach #1: Sign Magnitude

$$001 = 1$$

$$101 = -1$$

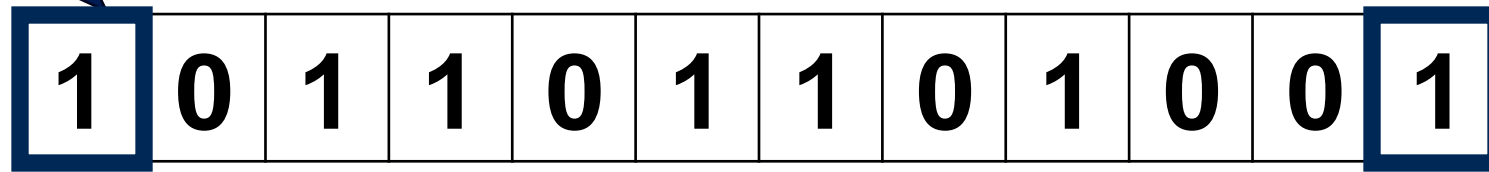
MSB represents the sign

- 0: +
- 1: -

FYI: LSB and MSB



Most Significant Bit (MSB)



Least Significant Bit (LSB)

Approach #1: Sign Magnitude

$$\begin{array}{l} 001 = 1 \\ 101 = -1 \end{array}$$

MSB represents the sign

- 0: +
- 1: -

The rest of the bits
represent the
magnitude



Problems?

Problems of Sign Magnitude

145

$$001 = 1$$


$$+ 110 = -2$$

$$111 \neq -1$$

Inconsistencies in operations

Approach #2: One's Complement




$$001 = 1$$
$$110 = -1$$

To represent a negative number, all the bits of the corresponding positive number are inverted

Approach #2: One's Complement

$$001 = 1$$

$$+ 101 = -2$$

$$110 = -1$$

Provides ease of operations in terms of computing

Approach #2: One's Complement

$$001 = 1$$

$$+ 101 = -2$$

$$110 = -1$$



Problems?

Provides ease of operations in terms of computing

Problems of One's Complement (+ Sign Magnitude)

149

Sign Magnitude

$$000 = +0$$

$$001 = +1$$

$$010 = +2$$

$$011 = +3$$

$$100 = -0$$

$$101 = -1$$

$$110 = -2$$

$$111 = -3$$

One's Complement

$$000 = +0$$

$$001 = +1$$

$$010 = +2$$

$$011 = +3$$

$$100 = -3$$

$$101 = -2$$

$$110 = -1$$

$$111 = -0$$

Problems of One's Complement (+ Sign Magnitude)

15

Sign Magnitude

$$000 = +0$$

$$001 = +1$$

$$010 = +2$$

$$011 = +3$$

$$100 = -0$$

$$101 = -1$$

$$110 = -2$$

$$111 = -3$$

One's Complement

$$000 = +0$$

$$001 = +1$$

$$010 = +2$$

$$011 = +3$$

$$100 = -3$$

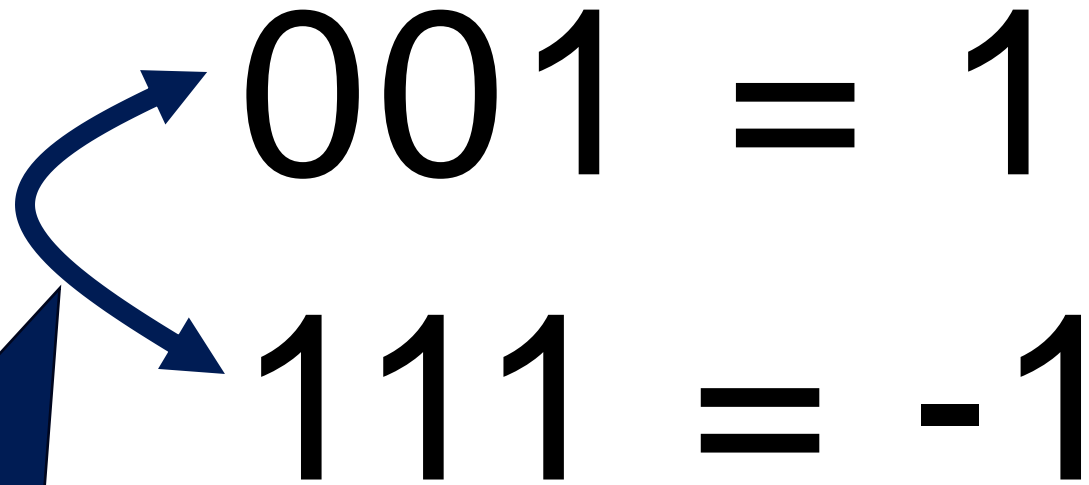
$$101 = -2$$

$$110 = -1$$

$$111 = -0$$

Two representations
for zero

Approach #3: Two's Complement


$$001 = 1$$
$$111 = -1$$

= One's Complement + 1

Approach #3: Two's Complement

Sign Magnitude

000 = +0
001 = +1
010 = +2
011 = +3
100 = -0
101 = -1
110 = -2
111 = -3

One's Complement

000 = +0
001 = +1
010 = +2
011 = +3
100 = -3
101 = -2
110 = -1
111 = -0

Two's Complement

000 = +0
001 = +1
010 = +2
011 = +3
100 = -4
101 = -3
110 = -2
111 = -1

Approach #3: Two's Complement

Sign Magnitude

$$000 = +0$$

$$001 = +1$$

$$010 = +2$$

$$011 = +3$$

Range: -3 ~ +3

$$100 = -0$$

$$101 = -1$$

$$110 = -2$$

$$111 = -3$$

One's Complement

$$000 = +0$$

$$001 = +1$$

$$010 = +2$$

$$011 = +3$$

Range: -3 ~ +3

$$100 = -3$$

$$101 = -2$$

$$110 = -1$$

$$111 = -0$$

Two's Complement

$$000 = +0$$

$$001 = +1$$

$$010 = +2$$

$$011 = +3$$

Range: -4 ~ +3

$$100 = -4$$

$$101 = -3$$

$$110 = -2$$

$$111 = -1$$

Approach #3: Two's Complement

Sign Magnitude

$$000 = +0$$

$$001 = +1$$

One's Complement

$$000 = +0$$

$$001 = +1$$

Two's Complement

$$000 = +0$$

$$001 = +1$$

Provides ease of operations and
a unique representation for zero

$$100 = -0$$

$$101 = -1$$

$$110 = -2$$

$$111 = -3$$

$$100 = -3$$

$$101 = -2$$

$$110 = -1$$

$$111 = -0$$

$$100 = -4$$

$$101 = -3$$

$$110 = -2$$

$$111 = -1$$

2's complement code is the most widely used representation of signed numbers in computer systems 😊

2's-Complement Signed Integers

- Given an n-bit number

$$X = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example

$$\begin{aligned} &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \\ &= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

- Using 32 bits ($n = 32$)
Range: $-2,147,483,648$ to $+2,147,483,647$

2's-Complement: Number Range

- 32-bit signed numbers:

$$0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000_{\text{two}} = 0_{\text{ten}}$$

[illegible]

$$0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 000$$
$$2_{\text{two}} = + 2_{\text{ten}}$$

• • •

$$0111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1110_{\text{two}} = + 2,147,483,646_{\text{ten}}$$

$$0111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111_{\text{two}} = + \ 2,147,483,647_{\text{ten}}$$

[illegible]

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = -2,147,483,647_{\text{ten}}$$

• • •

$$1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1110_{+two} = - \quad 2_{+ten}$$

$$\mathbf{1}_{1111} \quad \mathbf{1}_{1111} \quad \mathbf{1}_{1111} \quad \mathbf{1}_{1111} \quad \mathbf{1}_{1111} \quad \mathbf{1}_{1111} \quad \mathbf{1}_{1111} \quad \mathbf{1}_{1111} +_{\text{two}} = - \quad \mathbf{1}_{\text{ten}}$$

2's-Complement: Number Range

- 32-bit signed numbers:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = +1_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = +2_{\text{ten}}$$

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = +2,147,483,646_{\text{ten}}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = +2,147,483,647_{\text{ten}}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = -2,147,483,648_{\text{ten}}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = -2,147,483,647_{\text{ten}}$$

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = -1_{\text{ten}}$$

maxint

minint

Integer Overflow



Example:

$maxint + 1 = minint$ ($\neq maxint + 1$ in real world)

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2,147,483,647_{ten}$
 $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2,147,483,648_{ten}$

maxint

minint

Zero-Day Alert: Google Chrome Under Active Attack, Exploiting New Vulnerability

📅 Nov 29, 2023 👤 Ravie Lakshmanan

Zero-Day / Web Browser

— Trending News

Google has rolled out security updates to fix seven security issues in its Chrome browser, including a zero-day that has come under active exploitation in the wild.

Tracked as CVE-2023-6345, the high-severity vulnerability has been described as an **integer overflow** bug in Skia, an open source 2D graphics library.



Another Benefit of 2's-Complement: Simplifies Overflow Detection *

16

Basic idea: If ...

Positive Number $+$ Positive Number $=$ Negative Number

Or...

Negative Number $+$ Negative Number $=$ Positive Number

Overflow is occurred!

Another Benefit of 2's-Complement: Simplifies Overflow Detection *

162

From a logic circuit perspective:

$$\begin{array}{rcl} & 010 & = 2 \\ + & 011 & = 3 \\ \hline & 101 & = -3 \quad (\neq 5) \end{array}$$

Another Benefit of 2's-Complement: Simplifies Overflow Detection *

163

From a logic circuit **C_{out}** effective: **C_{in}**

Carry in (C _{in})	Carry out (C _{out})	Overflow occurred
0	0	X
0	1	O
1	0	O
1	1	X

0 1

010 = 2

+ 011 = 3

101 = -3 (≠5)

Another Benefit of 2's-Complement: Simplifies Overflow Detection *

164

From a logic circuit **C_{out}** effective: **C_{in}**

Carry in (C _{in})	Carry out (C _{out})	Overflow occurred
0	0	X
0	1	O
1	0	O
1	1	X

Exception:
Overflow has occurred

0 1

010 = 2

+ 011 = 3

101 = -3 (≠5)

2's-Complement Signed Integers



- 31st-bit (MSB) is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation



- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - +2 = 0000 0000 ... 0010₂
 - 2 = 1111 1111 ... 1101₂ + 1
 - = 1111 1111 ... 1110₂

Sign Extension



- Representing a number using more bits
 - Preserve the numeric value
- **Replicate the sign bit to the left**
 - c.f. unsigned extension: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - 2: 1111 1110 => 1111 1111 1111 1110

Question?