CZ1006
CE1006

Assembly Programming

A/P Goh Wooi Boon

Assembly Programming

Introduction to Assembly Language Programming

Learning Objectives (3a)

1. Identify why and when to use assembly language programming.
2. Describe what are addressing modes.

What is an Assembly Program?

- Unlike high-level programming languages, assembly level statements:
  - Are known as **mnemonics**. Each has a one-to-one correspondence with a machine-level binary pattern (**machine code**) that is directly understood by the CPU.
  - Are **hardware-dependent** and address the architecture of processor directly. (e.g. they are CPU register-aware and reference them by name).
  - Are converted to machine code by an **assembler**.

Why Use Assembly Language?

- More efficient codes can be created:
  - Codes with faster execution **speed**.
    e.g. Algorithms for **real-time** signal processing in handheld devices can be **computationally demanding**.
  - More compact program **size**.
    e.g. Low cost embedded devices may have **small memory** capacity but require many functionalities.
  - Exploit **optimized features** of processor's ISA.
    e.g. High-level language compiled codes may not **exploit optimized instructions** and features available in the processor instruction set architecture to produce efficient run-time code.

When to Use Assembly Language?

- Critical parts of the operating system's software.
  Especially parts of system kernel that are constantly being executed (e.g. scheduler, interrupt handlers).
- Input/Output intensive codes.
  Device drivers and "loopy" segments of code that processes streaming data (e.g. video decoders, etc).
- Time-critical codes.
  Code that detect incoming sensor signals and respond rapidly, e.g. Anti-lock brake system (ABS) in cars.

Addressing Modes

- Addressing mode (AM) is concerned with how data is **accessed**, not the way data is processed.
  - The correct AM allows the CPU to identify the actual operand or the address location where operand is stored.
- The VIP processor instruction set architecture supports many different addressing modes.
  - Register direct
  - Absolute address
  - Immediate data
  - Register indirect
  - Register indirect with offset
  - Program counter relative

Ref: 1. Null & Lobur (3rd Ed) section 5.4.2 – Addressing Modes
2. The VIP-1T Technical Reference Guide

Addressing Mode Examples

Addressing Mode	VIP	ARM
Absolute	`MOV R0,[0x100]`	None
Register Direct	`ADD R0,R1`	`ADD r0,r1,r2`
Immediate	`ADD R0,#3`	`ADD r3,r3,#3`
Register Indirect	`MOV R0,[R1]`	`LDR r0,[r1]`
Register Indirect with Offset	`MOV R0,[R2+4]`	`LDR r0,[r1,#4]`
Register Indirect with Index	None	`LDR r0,[r1,r2]`
Implied	`JNE -8`	`BNE LOOP`

Summary

- Codes written well in assembly language can usually execute **faster** and are smaller in **size**.
- Code for **low-level** OS kernels, **I/O** intensive and **time**-critical operations can benefit significantly from assembly-level coding.
- Understanding the **characteristics** and **application** of different **addressing modes** available in a processor's ISA allows programmers to write efficient codes.

Addressing Modes

Register Direct, Absolute Addressing and Immediate Data

Learning Objectives (3b)

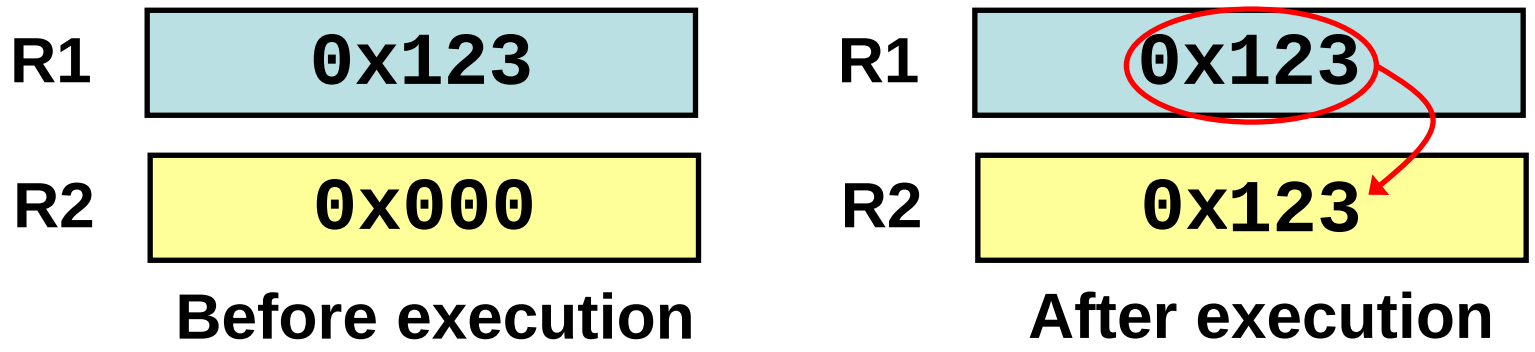
1. Describe what is register direct.
2. Describe what is absolute addressing.
3. Describe what is immediate data and its application.
4. Contrast their execution and instruction length characteristics.
5. Contrast the different between immediate data and absolute addressing

Register Direct, Rn

- Operand is the content of the specified register.
  - A **fast** addressing mode since no memory access is involved during execution.
  - Use where possible but CPUs have **limited** number of general registers.

Instruction Length (word)	Execution Time (cycles)
1	1

- Example: `MOV R2,R1` ← Register Direct

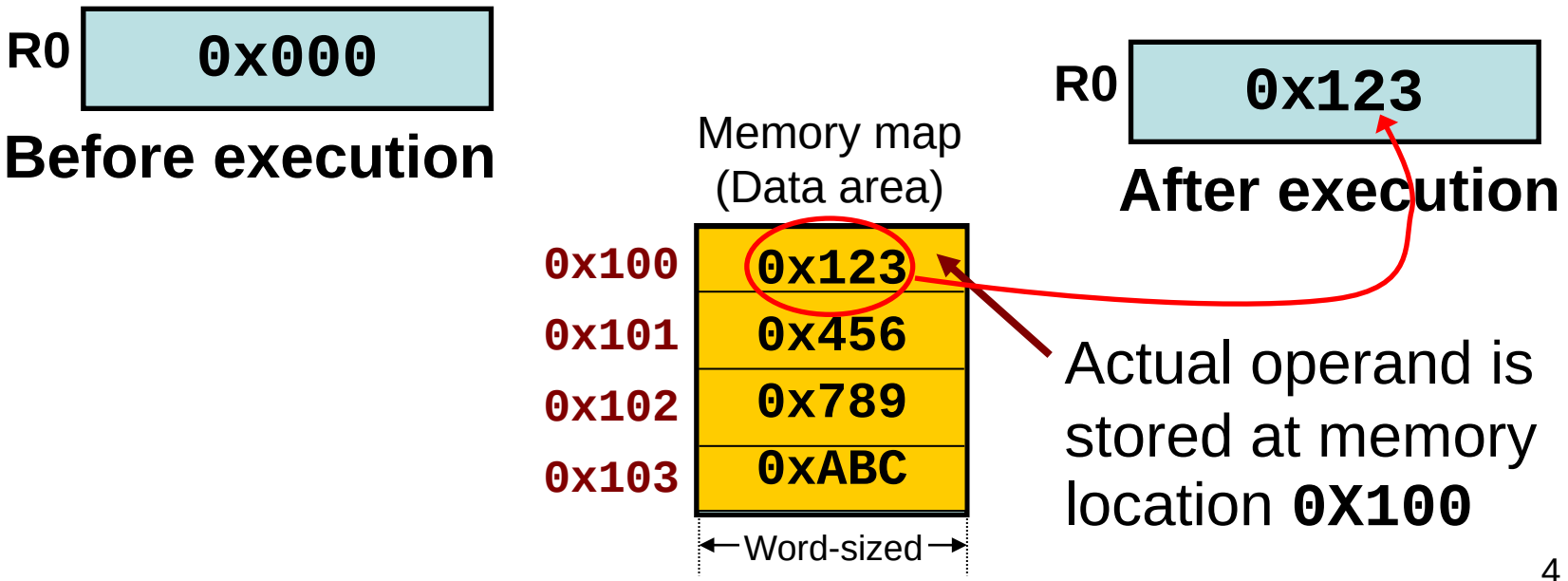


- Note: the VIP processor registers are 12-bit wide.

Absolute Address, [aaa]

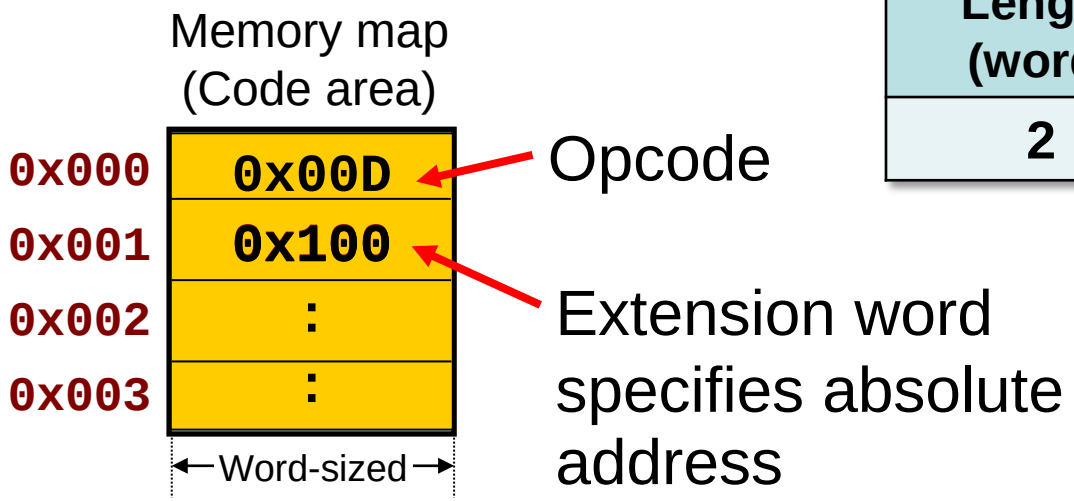
- Operand is stored in memory and is accessed using it **actual** (absolute) **address location**.
  - Also known as **direct** addressing.
  - Used when address of operand (e.g. memory variables) is known at the time of program coding.
  - Example: `MOV R0,[0x100]`

Absolute Addressing
(indicated by [ ] brackets)



Absolute Address (cont)

- The absolute address where the operand is stored is specified as **part of the instruction**.
  - Encoding absolute address as part of the instruction **increases** instruction **length** and execution **time**.
  - Inefficient mode, e.g. a CPU with a 64-bit addressing range requires 8 bytes to specify one absolute address.
  - Example: `MOV R0,[0x100]`



Instruction Length (word)	Execution Time (cycles)
2	3

Program Example

Register Direct & Absolute

- Task1: Copy contents in address **0x100** to **0x101**.

Mem-to-Mem

```
MOV [0x101],[0x100]
```

Instruction Length (words)	Execution Time (cycles)
3	5

Mem-to-Reg-to-Mem

```
MOV R0,[0x100]
MOV [0x101],R0
```

Code Length (words)	Execution Time (cycles)
4	6

- Unlike VIP, some processors do not support memory to memory transfer. In such cases, a register is required to assist in a memory to memory copy operation.

Program Example (Cont)

Register Direct & Absolute

- Task2: Copy contents in address **0x100** to both **0x101** and **0x102**.

Mem-to-Mem

```
MOV [0x101],[0x100]
MOV [0x102],[0x100]
```

Code Length (words)	Execution Time (cycles)
6	10

Mem-to-Reg-to-Mem

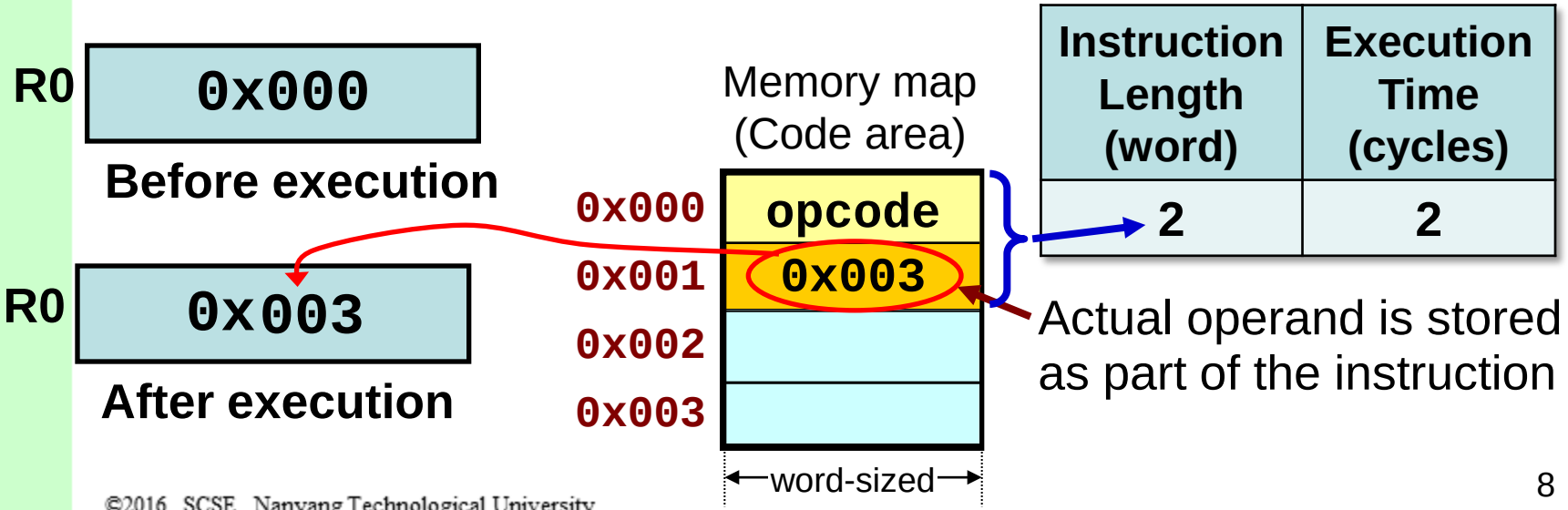
```
MOV R0,[0x100]
MOV [0x101],R0
MOV [0x102],R0
```

Code Length (words)	Execution Time (cycles)
6	9

- Use registers to store data to be copied to multiple memory locations. Your code can be optimized by **reducing access to memory** where possible.

Immediate Data, #n

- **Operand** is directly specified **within** the **instruction** itself.
  - "#" symbol precedes the immediate data.
  - Used for loading **data constants** into registers or memory locations. These values are known at the time of coding (e.g. loading the number of times to do a loop into a counter register).
  - Example: `MOV R0,#3` ← Immediate Data



Program Example

Absolute vs Immediate

- Differences between absolute addressing & immediate data:

```
MOV R0,[0x100]
```

```
MOV R0,#0x100
```

Absolute	Immediate
• Value of **operand** is **not known** at time of coding. • Can be used in both **source** and **destination**. • Execution involves **memory access**. • Typically used to handle **memory variables**.	• Value of operand must be known during coding. • Can only be used as a source operand. • No further memory access during execution. • Typically used to handle **constant values**.

Summary

- Register direct addressing is most **efficient** as its execution involves no access to memory.
- Absolute addressing is used to access memory whose addresses are known during coding.
  - Memory cycle count increases because further memory access occurs during execution.
- Immediate data is used when the operand value (i.e. **constant**) involved is known during coding.
- Absolute & immediate addressing modes requires **extension word(s)** in the instruction encoding.
  - This increases instruction length and execution cycles.

Addressing Modes

Register Indirect Addressing

Learning Objectives (3c)

1. Describe what is register indirect addressing.
2. Describe how register indirect addressing overcomes the limitation of absolute addressing.
3. Describe register indirect with offset and its applications.

Limitation of Absolute Addressing

- The address of the memory variable **must be known** at the time of program coding.
- The address is **not modifiable during run-time**.
- Consider the task of writing 0's into 400 bytes of memory starting at address **`0x100`**.

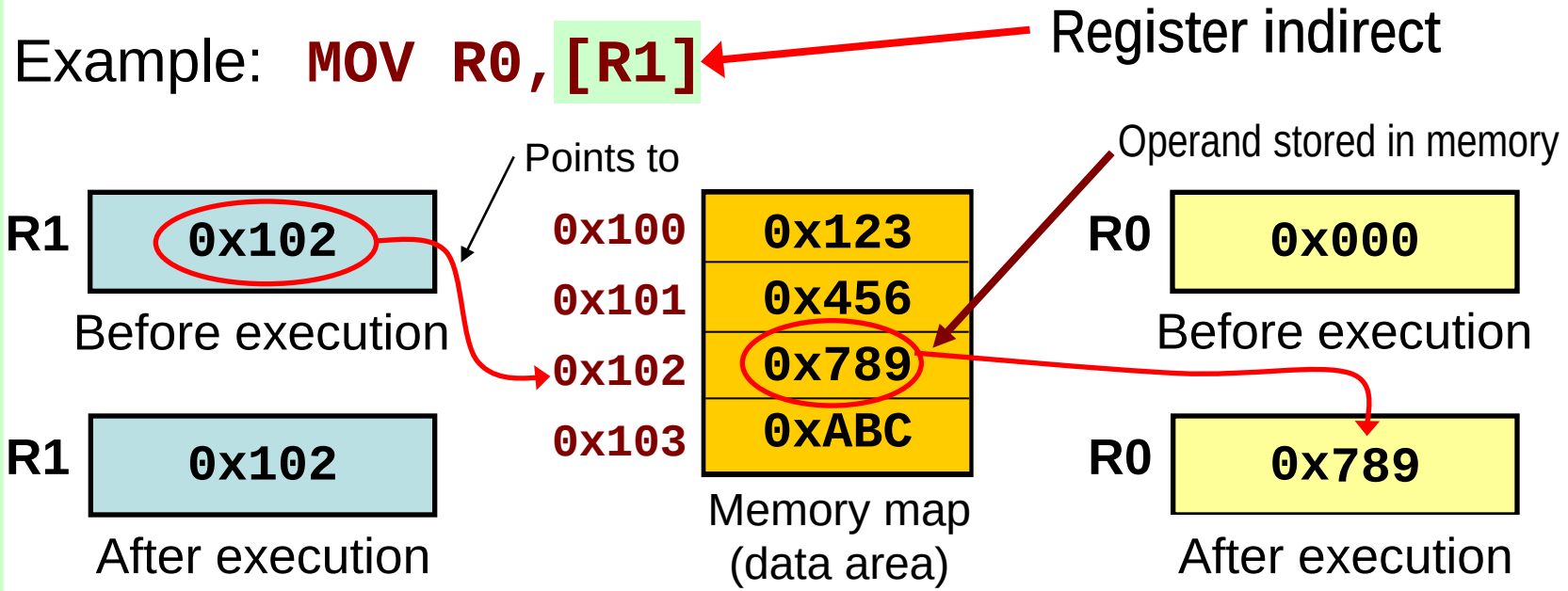
```
MOV R0,#0
MOV [0x100],R0
MOV [0x101],R0
  :
  :   × 400 MOV instructions
  :
```

- The size of the program that uses only absolute addressing mode will be **very long**.

Register Indirect, [Rn]

- Specified register contains the address of the operand and therefore **points** to the operand.
  - In VIP, the indirect register can be any of the two registers (**R0**-**R1**).
  - CPU access operand pointed to by the **contents** in **Rn** during instruction **execution**.
  - Example: `MOV R0,[R1]`

Instruction Length (word)	Execution Time (cycles)
1	2



Program Example

Using Register Indirect

- Consider the task of writing 0's to 400 bytes of memory starting at address **0x100**:

```
MOV R0,#0
MOV [0x100],R0
MOV [0x101],R0
MOV [0x102],R0
MOV [0x103],R0
MOV [0x104],R0
MOV [0x105],R0
MOV [0x106],R0
 :   × 400 MOV instructions
```

1

Using absolute addressing
(earlier example)

```
      MOV   R1,#0x100   3
loop  MOV   [R1],#0
      ADD   R1,#1       2
      loop back 400 times
```

1

Using register indirect

1. Code is much **shorter** than the absolute addressing version.
2. Address of operand can be **computed** during **run-time**.
3. But start address **initialization** is needed.

Program Example (Optimized Version)

Using Register Indirect

- Code optimization is most beneficial within loop constructs as each cycle saved is multiplied by the number of loops.
- Choosing a **shorter instruction** to implement the same functionality is a good way of optimizing your code.

```
        MOV   R1,#0x100
loop    MOV [R1],#0
        ADD   R1,#1
```

loop back 400 times

(earlier version)

```
        MOV   R0,#0
        MOV   R1,#0x100
loop    MOV   [R1],R0
        ADD   R1,#1
```

+2~ ×1

-1~ ×400

loop back 400 times

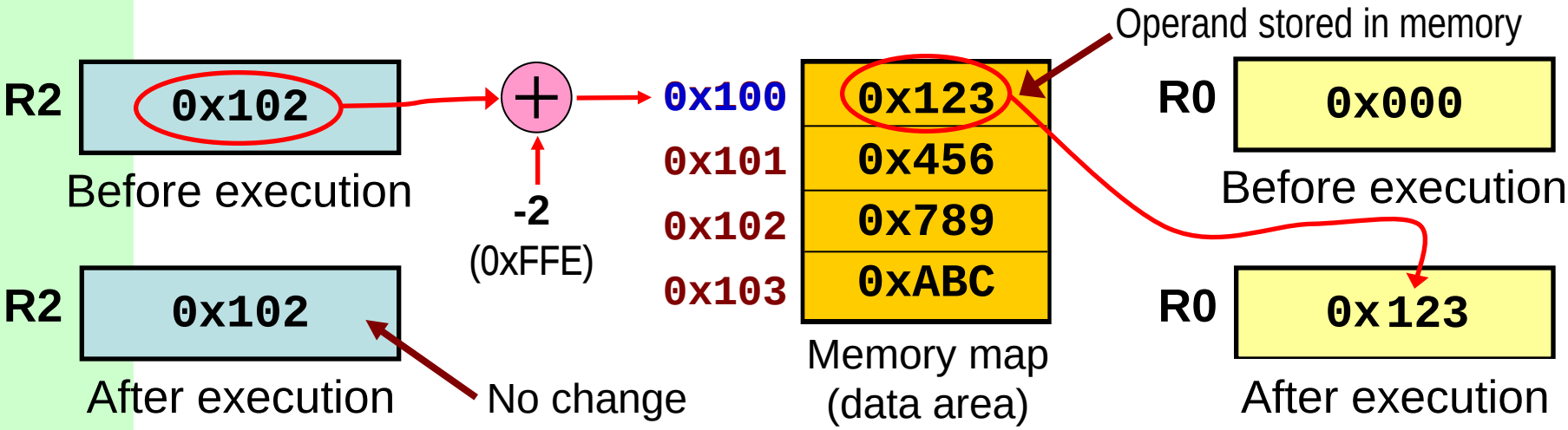
(optimized version)

- 398 clock cycles saved by the optimized code.

Register Indirect with Offset, [Rn+d]

- Effective address of operand obtained by adding an offset to the content of the indirect register.
  - The indirect register with offset is any of the two registers (**R2**-**R3**).
  - The offset is a signed 12-bit value.
  - Content in **Rn** remains unmodified after execution.
  - E.g.: `MOV R0,[R2+0xFFE]` ← Register indirect with offset

Instruction Length (word)	Execution Time (cycles)
2	3



Program Example

Accessing Array Elements

- Use register indirect with offset to access array element whose index is known during coding.

```
main()
{
// assume base address
// of array i is 0x100
int i[5];

i[0]=7;
i[4]=8;
}
```

C program example

Assign first & last elements of array **i** with the values of 7 and 8 respectively.

```
MOV   R2,#0x100      1
MOV   [R2+0],#7      2
MOV   [R2+16],#8     2
```

Using register indirect with offset

1. Initialize base address of array **i** into register R2.
2. Load immediate values of 7 and 8 into i[0] and i[4] using offsets of 0 and 16 of register R2 respectively .
   - Each integer element occupies 4 memory bytes.

Summary

- Register indirect allows the address of memory accessed to be computed during run time.
- VIP ISA supports two variants of register indirect.
  - Register indirect (using registers **R0** and **R1**).
  - Register indirect with offset (using registers **R2** and **R3**)
- Register indirect addressing is useful for accessing contents of arrays.

Addressing Modes

Stacks and PC-related Addressing

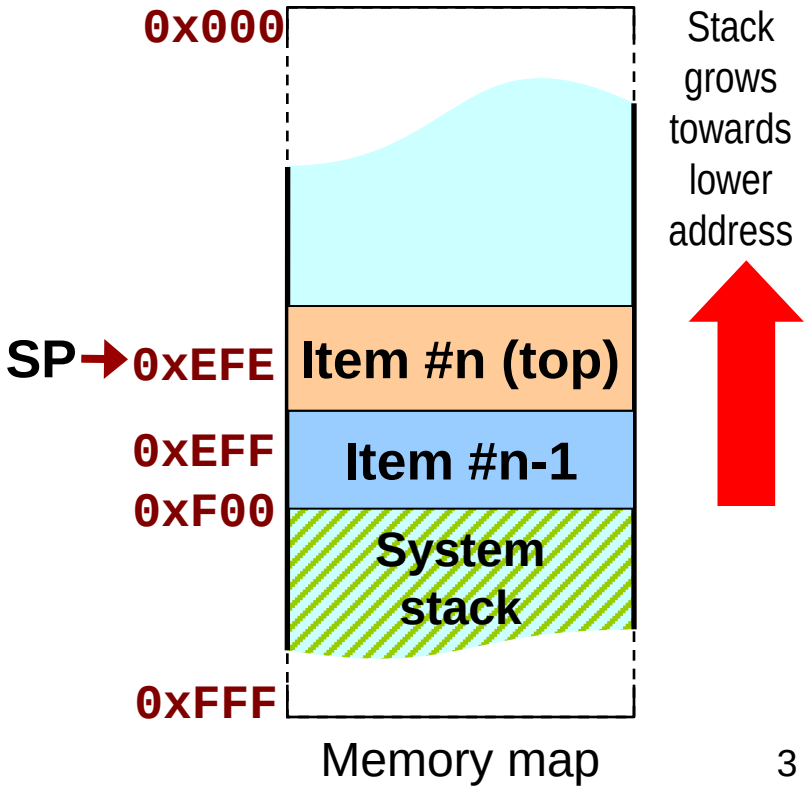
Learning Objectives (3d)

1. List the stack manipulation operations & its implementation.
2. Describe the difference between absolute & relative jump.
3. Describe how relative jump supports position-independent code.
4. Describe how data is accessed using PC relative addressing.

System Stack

- A stack is a first-in, last-out linear data structure that is maintained in the memory's data area.
  - The system stack in the VIP processor is maintained by a dedicated stack pointer (**SP**).
  - Stack grows towards **lower memory** address.
  - The **SP** points to the top item on the system stack.
  - The 3 basic stack operations are **push**, **pop** and **access** items on the stack.

Ref: Null & Lobur (3rd Ed) section A.2.3 – Stacks

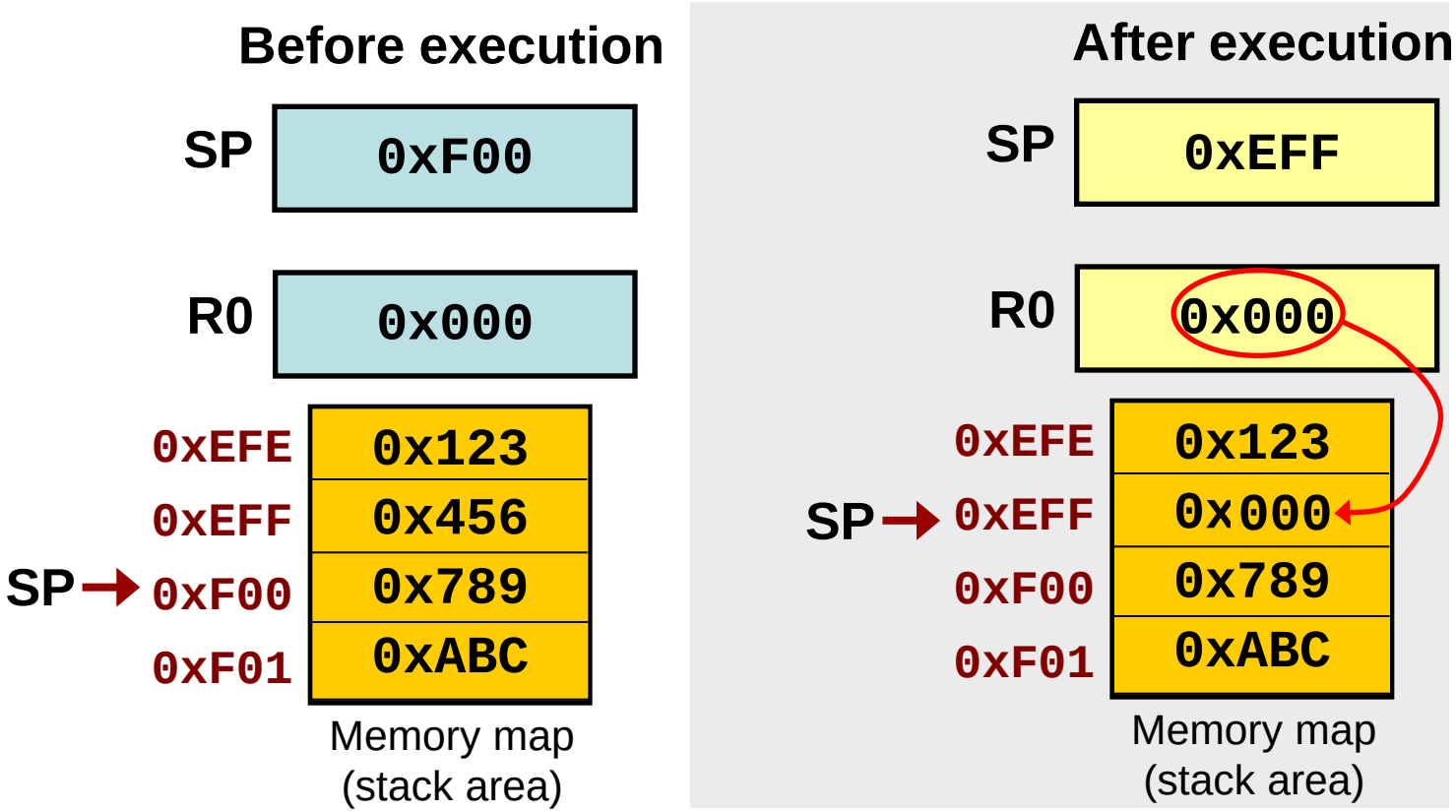


Stack Operations

Push Data to the Stack

- Stack grows as data items are pushed onto the stack using the **PSH** instruction.

Example: **PSH R0**

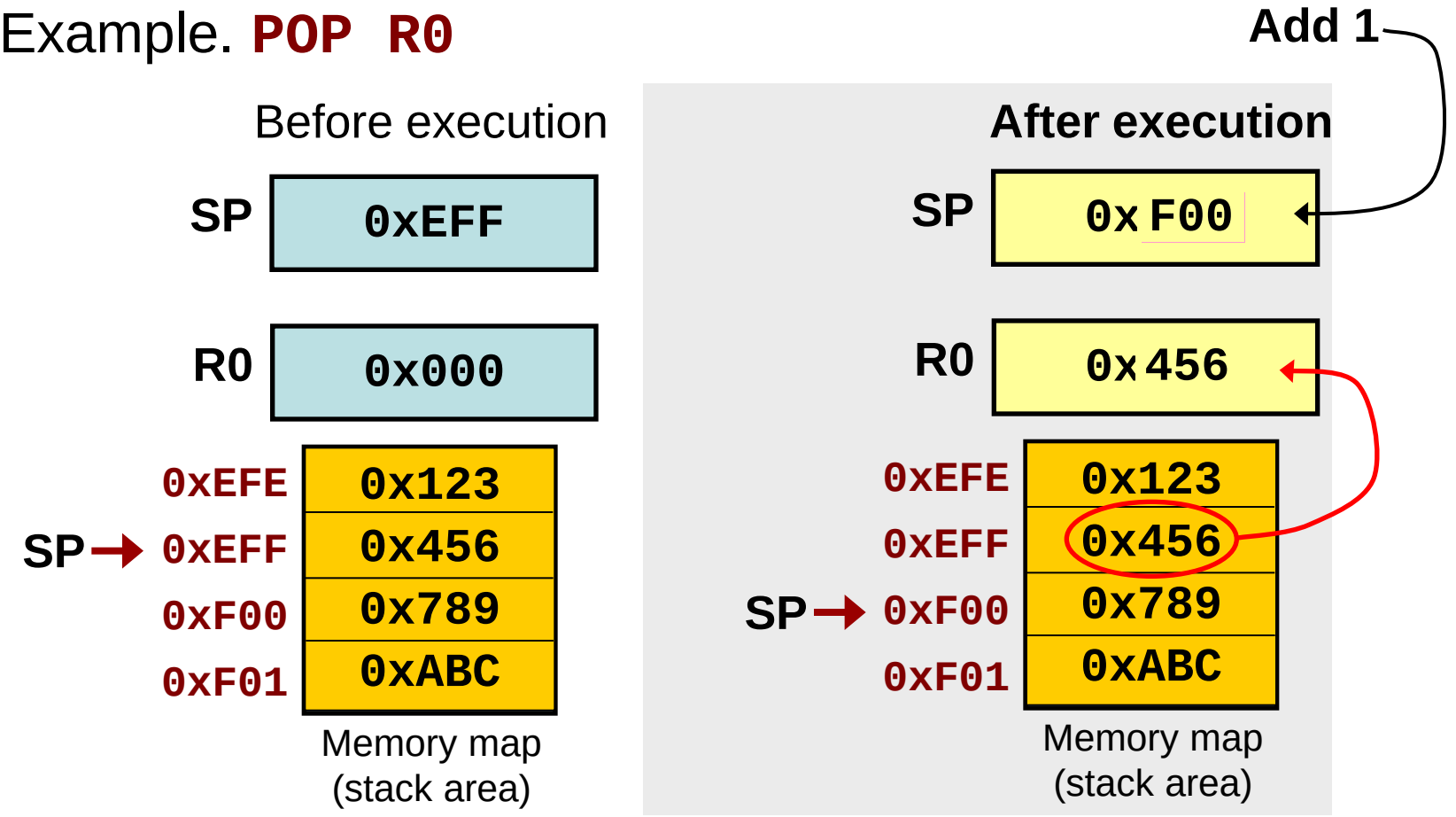


Stack Operations

Pop Data off the Stack

- Stack collapses as data is removed from the stack using the **POP** instruction.

Example. **POP R0**



Program Example

Register Swap using the Stack

- The stack can be viewed as a convenient memory space for temporary storage of data.
- Example: Swap the contents of registers **R0** and **R1**.

```
; Use temporary register R2
MOV  R2,R1
MOV  R1,R0
MOV  R0,R2
```

1 — 3 — 3 Cycles

Using register

```
; Use stack for temp storage
PSH   R1
MOV   R1,R0
POP   R0
```

2 — 2 — 3 — 5 Cycles

Using stack (Example #1)

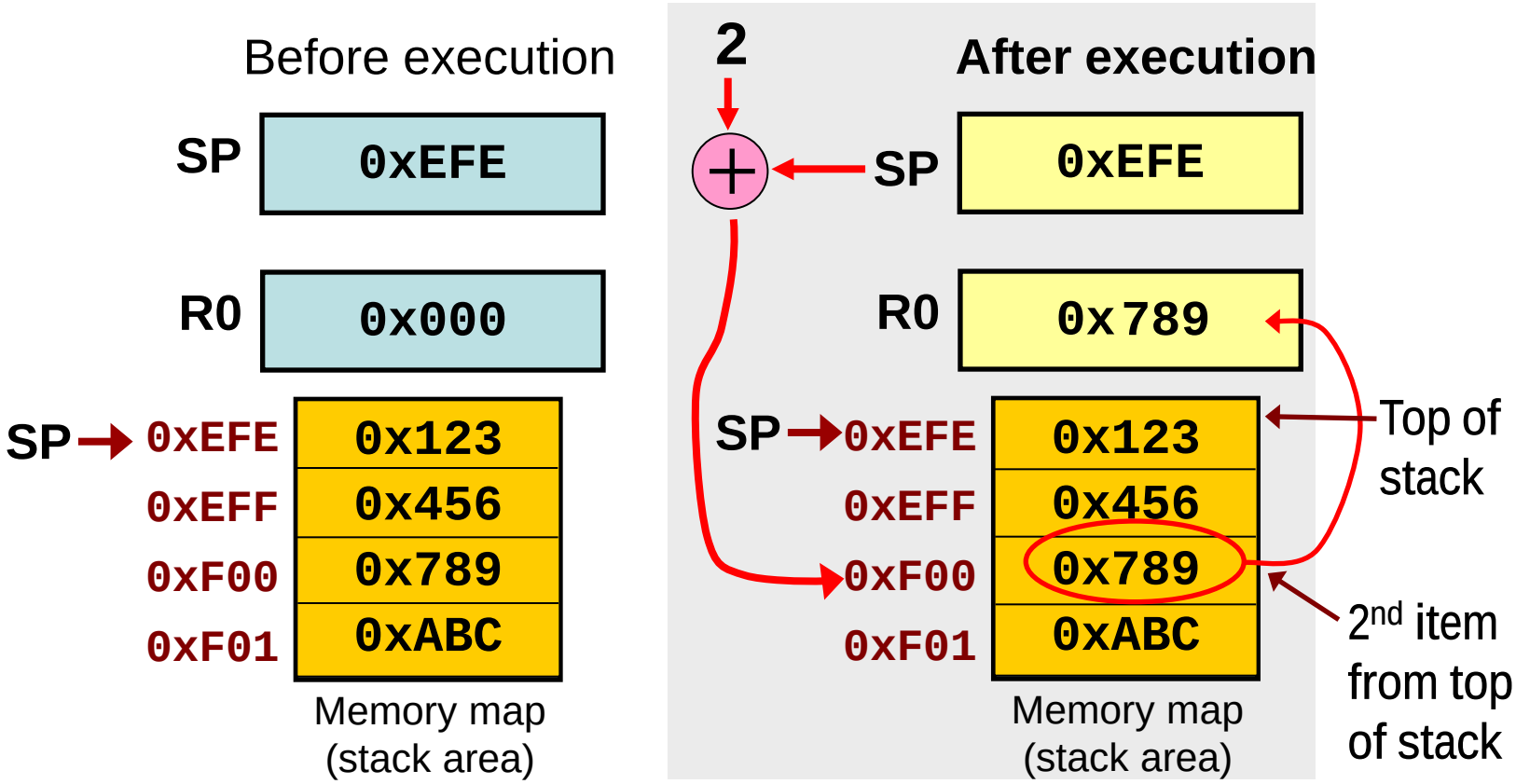
1. Using only registers is not possible if there are no spare registers left.
2. No knowledge of stack address as reference to stack memory is made via SP. But every push needs a pop to avoid stack overflow.
3. Stack operation involve memory access and is slower.

Stack Operations

Accessing Items on the Stack

- Register indirect with **offset** is used to access any items on the stack. **SP** used as reference.

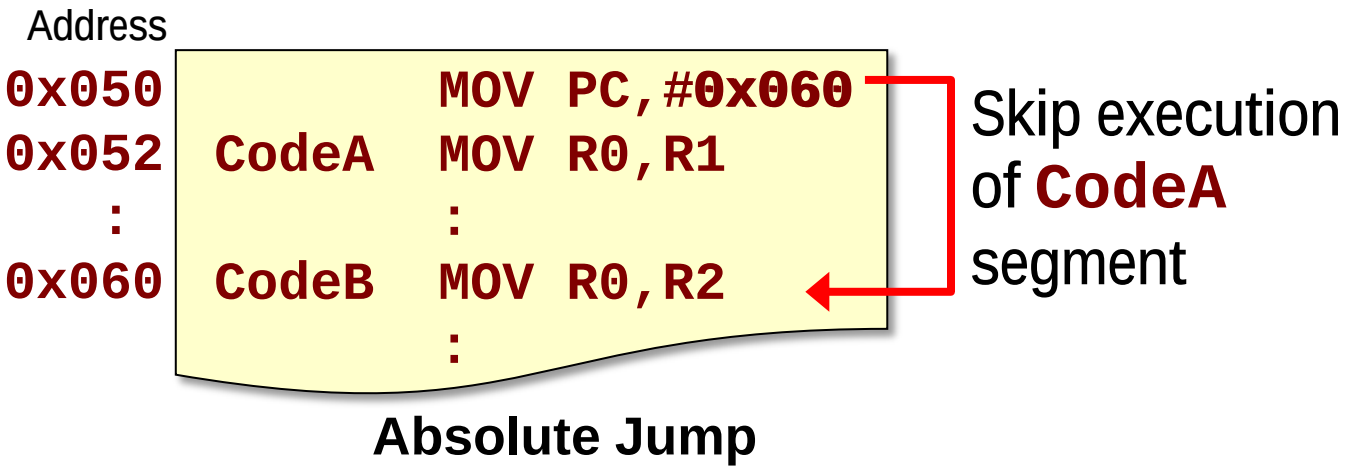
Example `MOV R0,[SP+2]`



Program Counter Related Addressing

Absolute Jump

- A **new address** can be loaded into the PC to alter the sequential order of program execution.
  - An **absolute jump** to a new position in the code can be done by loading the address to jump to into the PC.
  - Example: `MOV PC,#0x060 ;` Jump to CodeB



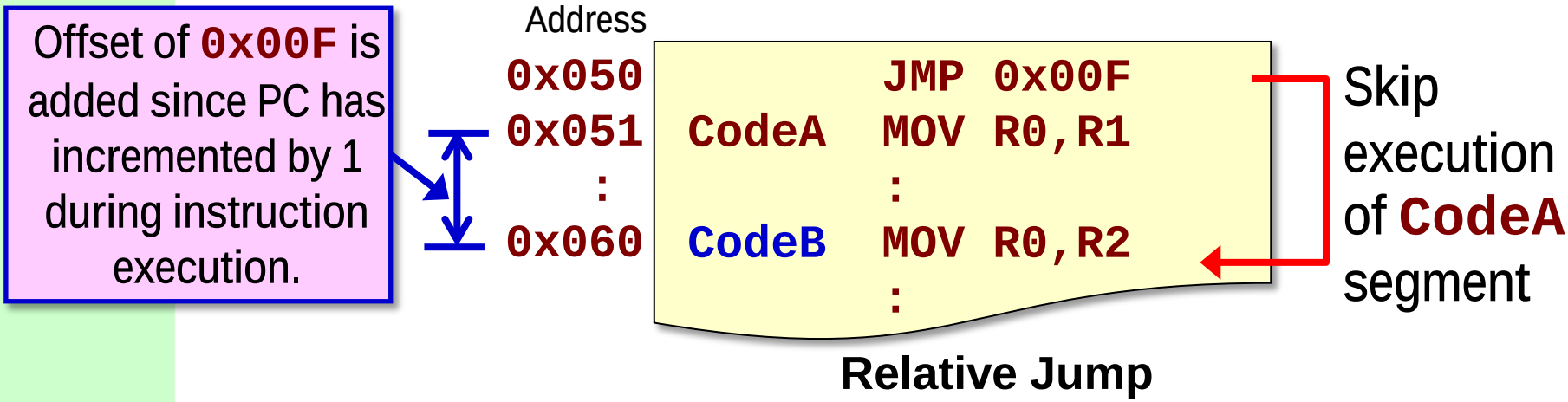
Absolute Jump

  - Absolute jump is not **position-independent**. This code can only execute correctly in this area of code memory.

Program Counter Related Addressing

Relative Jump

- An **offset** can be added to the PC to alter the sequential order of program execution.
  - A **relative jump** can be done using the **JMP** or **BRA** instruction with an appropriate **signed offset** (which is added to one plus the start address of the JMP instruction).
  - Example: `JMP 0x00F ;Jump to CodeB`

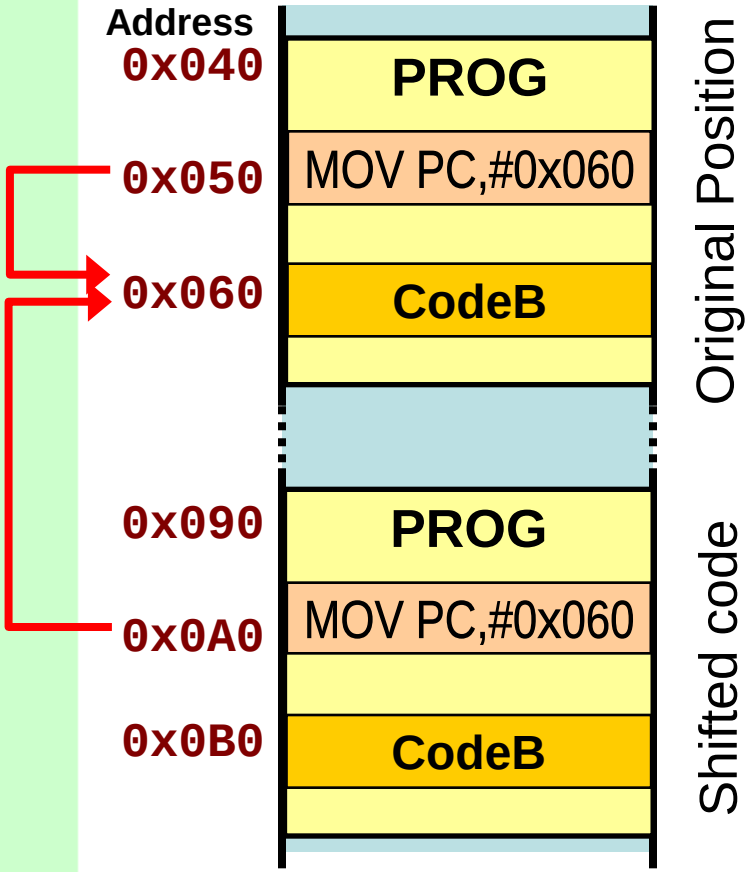


Relative Jump

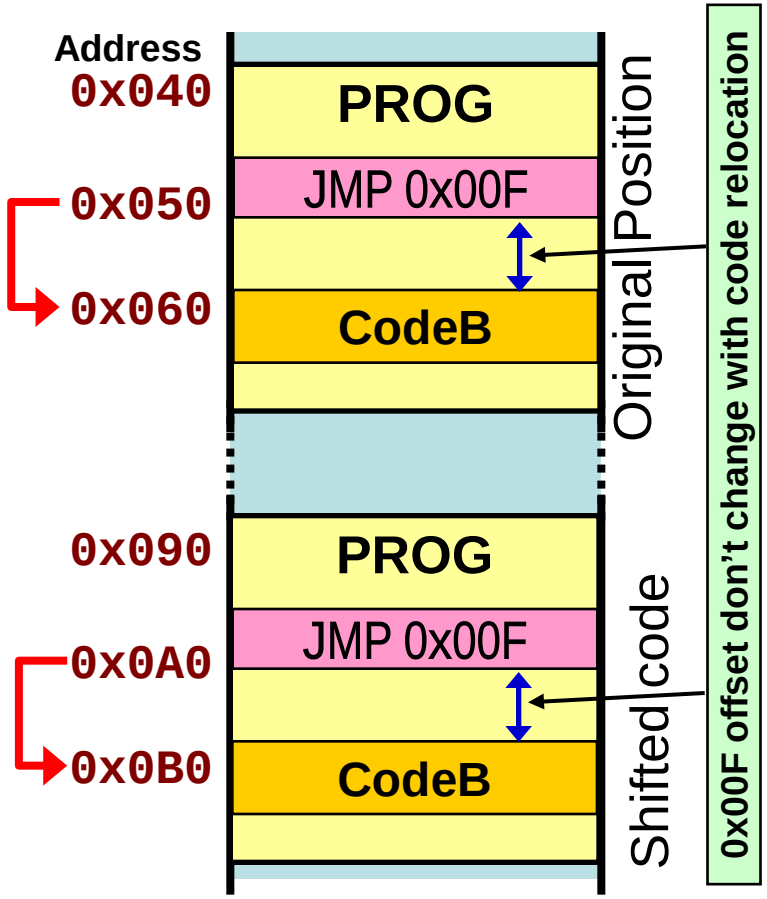
- Relative jump supports **position-independent** code.

Position-Independent Code

- Such programs can be loaded anywhere in memory and still execute correctly (i.e. relocatable).



Does not jump to **CodeB** after **PROG** is shifted to **0x090**



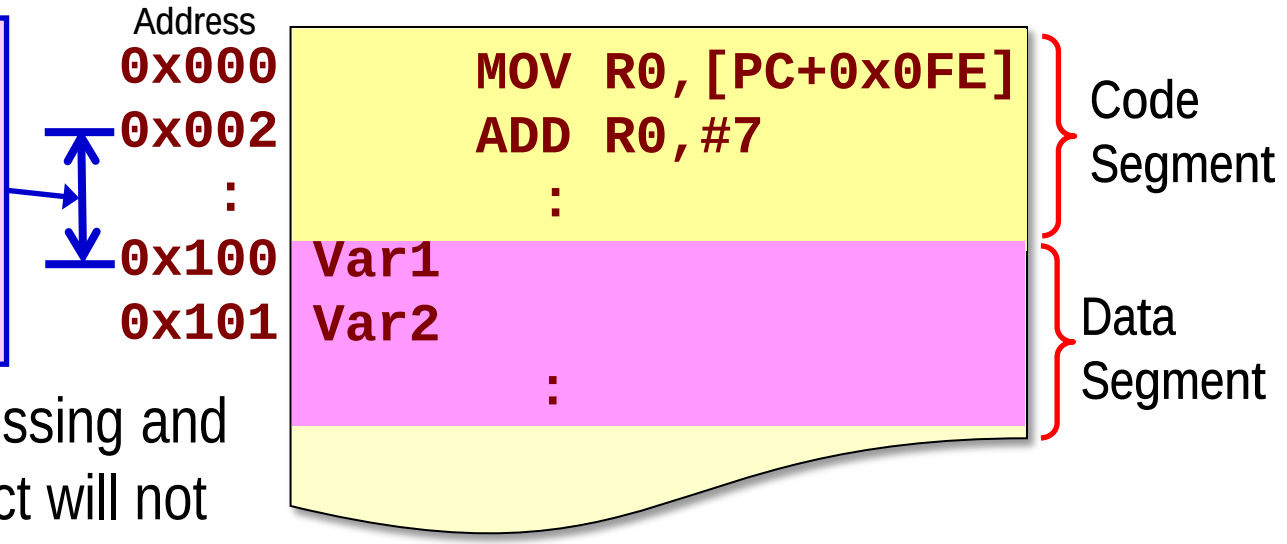
JMP still jump to **CodeB** after **PROG** is shifted to **0x090**

Program Counter Related Addressing

Accessing Data

- Position-independent (P-I) programs also require data to be accessed relative to the PC.
  - **PC relative addressing** is used to access variables in the data segment that is placed either before the start or after the end of a code segment.
  - E.g.: `MOV R0,[PC+0x0FE] ;` Read Var1 in Data Seg in R0

Offset of `0x0FE` is added since PC has incremented by 2 to point to the start of next instruction.



PC-Relative Addressing

- Absolute addressing and Register Indirect will not support P-I code.

Summary

- **PSH** and **POP** can be used to put and remove items to and from the stack respectively.
  - **Register indirect with positive offset** can be used to access items on the stack.
- Non-sequential execution of code can be achieved by modifying the PC contents directly.
  - The JMP instruction does this by **adding** a **signed offset**.
  - Such **relative** jumps create **position-independent** code.
- **PC indirect** with **offset** allows memory data to be accessed in a position-independent manner.