Chapter 8.1: The Stack

Microprocessors

Stack and LIFO | SS:SP | PUSH / POP | Byte order | Saving registers | Swapping | PUSHF / POPF | Stack safety

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/microprocessors/slides/Chapter_08_01_The_Stack.html

INTRODUCTION

Learning Outcomes

Stack model

Explain how the 8086 stack is addressed through SS:SP and why it grows downward.

PUSH / POP

Know the valid operand types and trace how SP changes with each instruction.

Saving registers

Use the stack in the right order to save, restore and swap register values.

PUSHF / POPF

Save the FLAGS register on the stack and restore flags that an operation has changed.

Stack safety

Recognize and avoid unbalanced PUSH/POP, overflow and wrong-order bugs.

Tracing in emu8086

Read the register and stack windows while single-stepping a program.

The Stack

- The stack is a memory area set aside for **temporary data**.
- `CALL` pushes the procedure's **return address**; `RET` pops it and jumps back to that address.
- `INT` does something similar when it calls an interrupt: the **FLAGS register**, the **code segment (CS)** and the **offset (IP)** are saved on the stack; `IRET` returns from the interrupt.
- `PUSH` : stores a 16-bit value on the stack.
- `POP` : takes a 16-bit value off the stack.

Who uses it?

The programmer (PUSH/POP), the CPU (CALL, INT), the operating system and BIOS interrupt routines.

Unit

The 8086 always writes and reads the stack one **word** (2 bytes) at a time; `PUSH AL` is invalid.

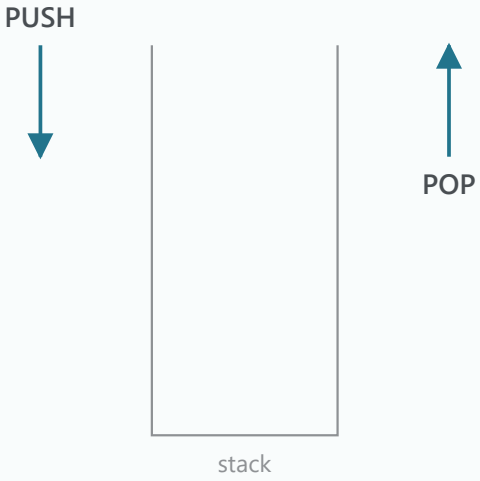
Direction

The stack grows from high addresses toward low ones, i.e. **downward**.

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

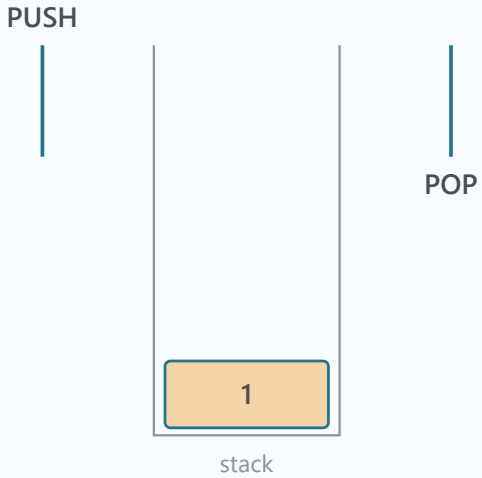


The stack starts empty.

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

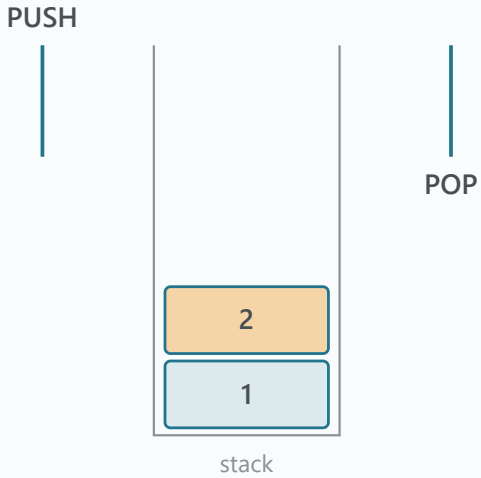


PUSH 1: 1 goes on **top** of the stack.

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

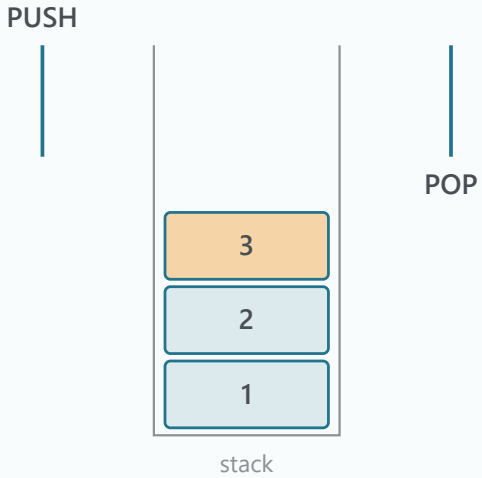


PUSH 2: 2 goes on **top** of the stack.

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

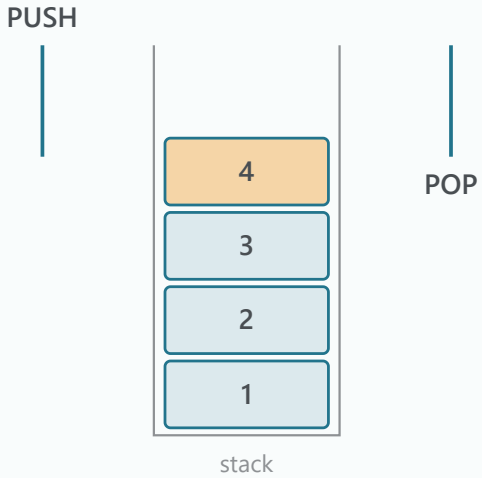


PUSH 3: 3 goes on **top** of the stack.

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

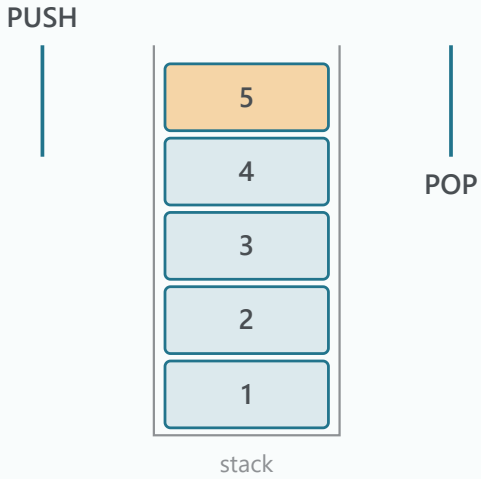


PUSH 4: 4 goes on **top** of the stack.

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

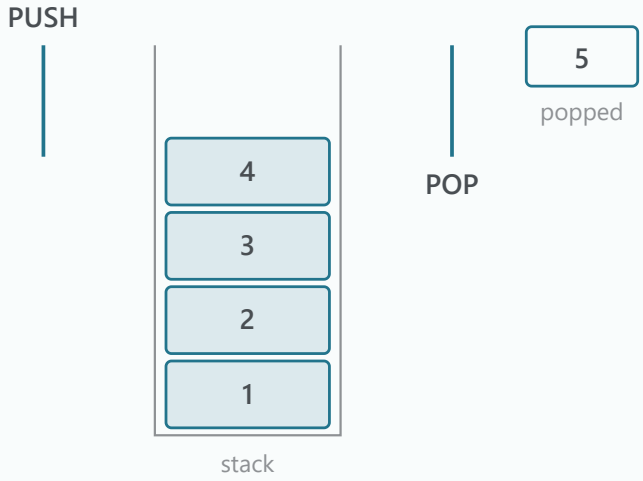


PUSH 5: 5 goes on **top** of the stack.

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

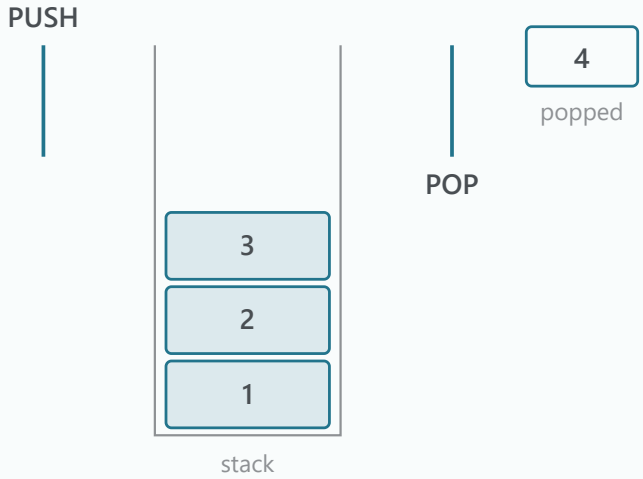


POP → **5** comes off. Output so far: 5

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

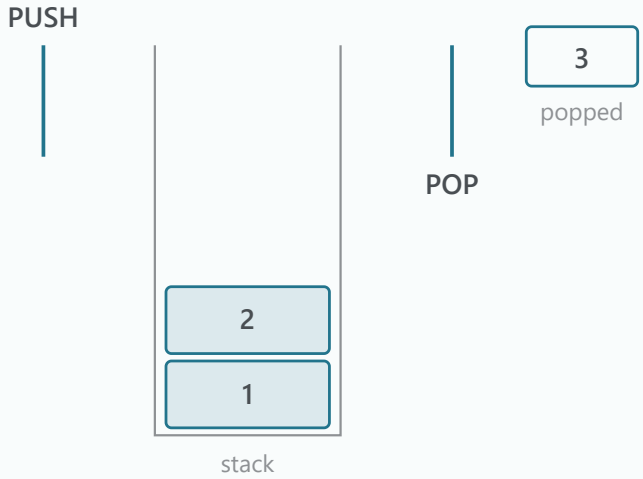


POP → **4** comes off. Output so far: 5, 4

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

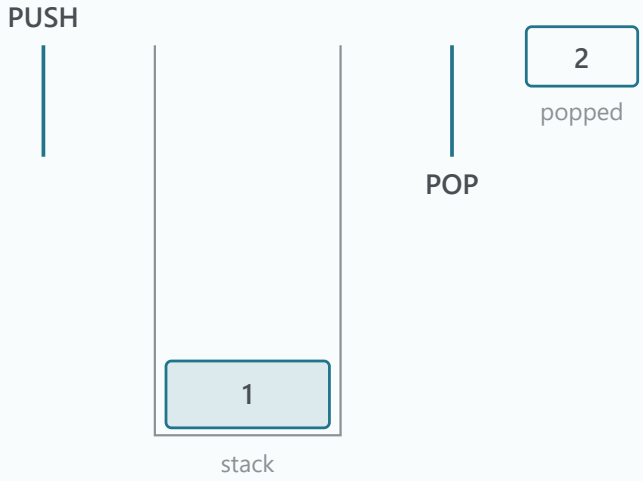


POP → **3** comes off. Output so far: 5, 4, 3

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.

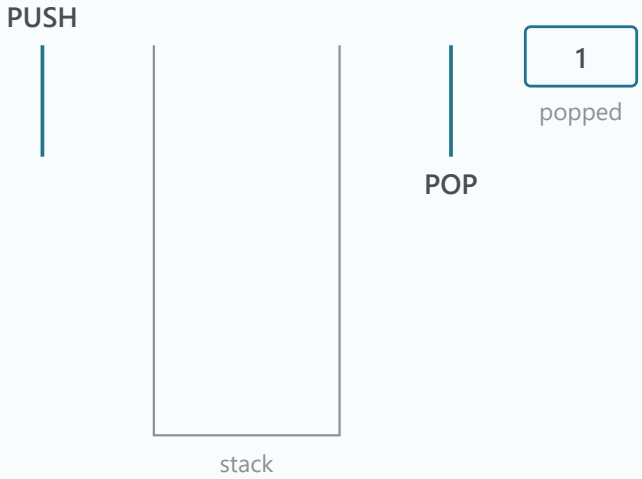


POP → **2** comes off. Output so far: 5, 4, 3, 2

LIFO Order

- LIFO stands for **Last In First Out**.
- The last item pushed is the first item popped.
- If 1, 2, 3, 4, 5 are pushed in that order, the first value popped is **5**, then 4, 3, 2 and finally 1.

Result: the stack returns values in **reverse order**. To restore registers, POP them in the reverse of the PUSH order.



POP → **1** comes off. Output so far: 5, 4, 3, 2, 1

Stack Memory: SS and SP

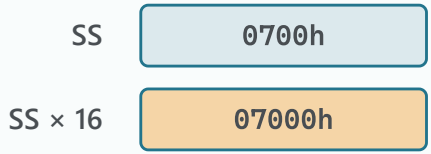
- The stack uses the **SS** (Stack Segment) and **SP** (Stack Pointer) registers.
- SS:SP always points to the word on **top** of the stack (the one pushed last).
- The initial values are usually set by the operating system (loader): for .COM, SS = CS and SP = FFFEh; for .EXE, the values in the header.
- **BP** also uses SS by default, which makes it ideal for accessing data on the stack.



Stack segment SS = **0700h** (in emu8086 all segment registers of a .COM program are 0700h).

Stack Memory: SS and SP

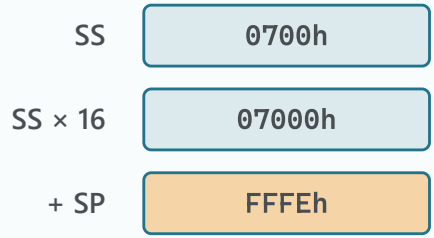
- The stack uses the **SS** (Stack Segment) and **SP** (Stack Pointer) registers.
- SS:SP always points to the word on **top** of the stack (the one pushed last).
- The initial values are usually set by the operating system (loader): for .COM, SS = CS and SP = FFFEh; for .EXE, the values in the header.
- **BP** also uses SS by default, which makes it ideal for accessing data on the stack.



The segment is multiplied by 16 (in hex, append a 0): **07000h**.

Stack Memory: SS and SP

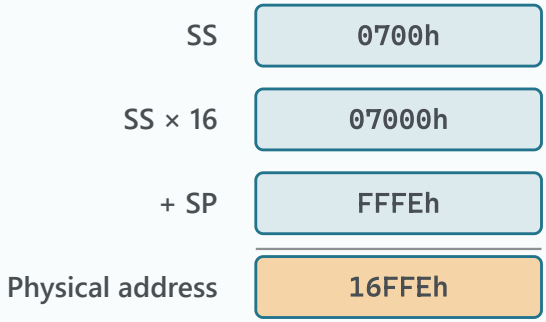
- The stack uses the **SS** (Stack Segment) and **SP** (Stack Pointer) registers.
- SS:SP always points to the word on **top** of the stack (the one pushed last).
- The initial values are usually set by the operating system (loader): for .COM, SS = CS and SP = FFFEh; for .EXE, the values in the header.
- **BP** also uses SS by default, which makes it ideal for accessing data on the stack.



The stack pointer SP = **FFFEh** is added as the offset.

Stack Memory: SS and SP

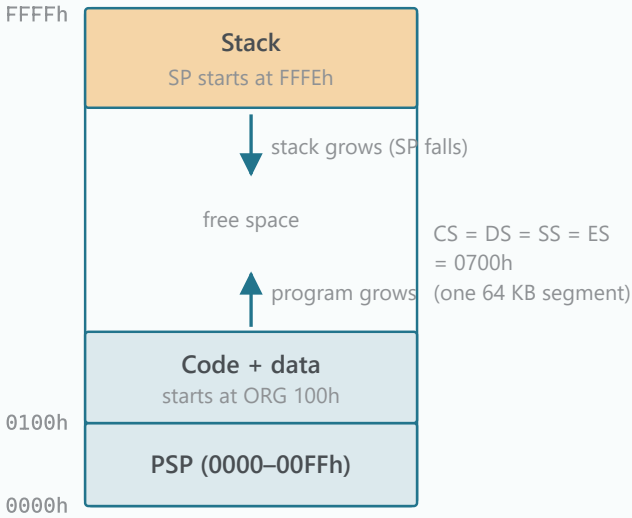
- The stack uses the **SS** (Stack Segment) and **SP** (Stack Pointer) registers.
- SS:SP always points to the word on **top** of the stack (the one pushed last).
- The initial values are usually set by the operating system (loader): for .COM, SS = CS and SP = FFFEh; for .EXE, the values in the header.
- **BP** also uses SS by default, which makes it ideal for accessing data on the stack.



The top of the stack is at physical address **16FFEh** (20 bits).

The Stack Grows Downward

- In a .COM program, code, data and stack share **the same 64 KB segment**.
- The program is loaded upward from 0100h and the stack downward from FFFEh; the free space between serves both.
- Each PUSH **decrements** SP by 2, each POP **increments** it by 2.
- Too many PUSHes can make the stack overwrite code or data: a **stack overflow**.



The PUSH Instruction

Form	Valid operands	Example
`PUSH REG`	AX, BX, CX, DX, DI, SI, BP, SP	`PUSH AX`
`PUSH SREG`	DS, ES, SS, CS	`PUSH DS`
`PUSH memory`	[BX], [BX+SI+7], a 16-bit variable…	`PUSH [BX]`, `PUSH num`
`PUSH immediate`	5, −24, 3Fh, 10001101b…	`PUSH 3Fh`

`PUSH immediate` does **not exist** on the 8086/8088; it came with the 80186. emu8086 accepts it, but for a real 8086 write `MOV AX, 5 + PUSH AX`.

Only **16-bit** operands are allowed: `PUSH AL` and `PUSH byte_var` are errors. Flags have their own instruction: `PUSHF`.

Detail: on the 8086, `PUSH SP` pushes the value of SP *after* the decrement; the 80286 and later push the value before it.

The POP Instruction

Form	Valid operands	Example
`POP REG`	AX, BX, CX, DX, DI, SI, BP, SP	`POP BX`
`POP SREG`	DS, ES, SS (**not CS**)	`POP DS`
`POP memory`	[BX], [BX+SI+7], a 16-bit variable…	`POP result`

Why not CS?

Changing CS alone, while IP stays the same, sends execution to an arbitrary place. CS only changes together with IP: `RETF`, `IRET`, far `JMP/CALL`.

No immediates

`POP 5` is meaningless: the popped value has to be **stored** somewhere.

Flags

PUSH/POP do not change the flags; only `POPF` loads the whole FLAGS register from the stack.

Common idiom: `PUSH CS` + `POP DS` → copies CS into DS (there is no direct MOV between segment registers).

How Do PUSH and POP Work?

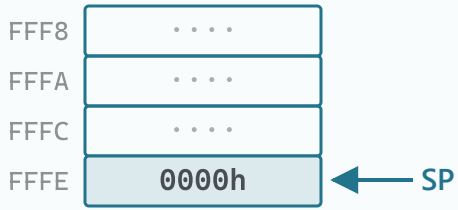
PUSH source

1. SP is decremented by 2.
2. The source value is written to SS:SP.

POP destination

1. The value at SS:SP is written to the destination.
2. SP is incremented by 2.

Order matters: PUSH decrements, then writes; POP reads, then increments. So SP always points to the topmost **occupied** cell.



SS:offset · low address at top

Start: SP = FFFEh, AX = 1234h. (In a .COM program, FFFEh holds the 0000h placed by DOS.)

Step 1 / 5

How Do PUSH and POP Work?

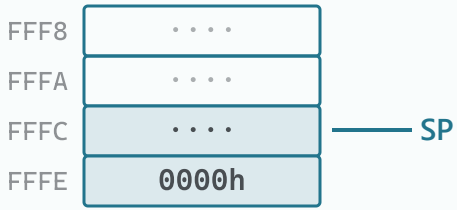
PUSH source

1. SP is decremented by 2.
2. The source value is written to SS:SP.

POP destination

1. The value at SS:SP is written to the destination.
2. SP is incremented by 2.

Order matters: PUSH decrements, then writes; POP reads, then increments. So SP always points to the topmost **occupied** cell.



SS:offset · low address at top

PUSH AX · 1

`PUSH AX` ① SP ← SP − 2 = **FFFCh**. The new top cell is not written yet.

How Do PUSH and POP Work?

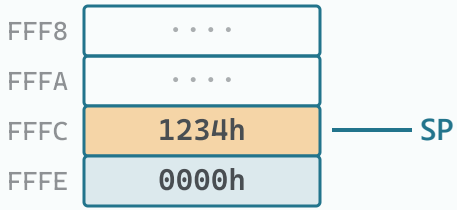
PUSH source

1. SP is decremented by 2.
2. The source value is written to SS:SP.

POP destination

1. The value at SS:SP is written to the destination.
2. SP is incremented by 2.

Order matters: PUSH decrements, then writes; POP reads, then increments. So SP always points to the topmost **occupied** cell.



SS:offset · low address at top

PUSH AX · 2

② The source value is written to SS:SP: [FFFCh] ← **1234h**.

How Do PUSH and POP Work?

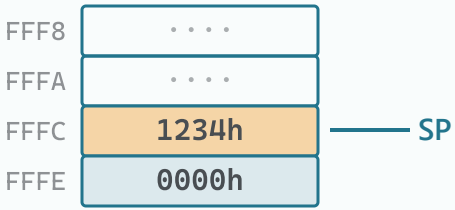
PUSH source

1. SP is decremented by 2.
2. The source value is written to SS:SP.

POP destination

1. The value at SS:SP is written to the destination.
2. SP is incremented by 2.

Order matters: PUSH decrements, then writes; POP reads, then increments. So SP always points to the topmost **occupied** cell.



SS:offset · low address at top

POP BX · 1

`POP BX` ① The value at SS:SP goes to the destination: BX ← **1234h**.

How Do PUSH and POP Work?

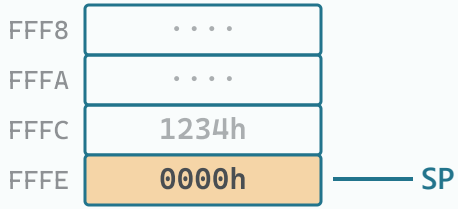
PUSH source

1. SP is decremented by 2.
2. The source value is written to SS:SP.

POP destination

1. The value at SS:SP is written to the destination.
2. SP is incremented by 2.

Order matters: PUSH decrements, then writes; POP reads, then increments. So SP always points to the topmost **occupied** cell.

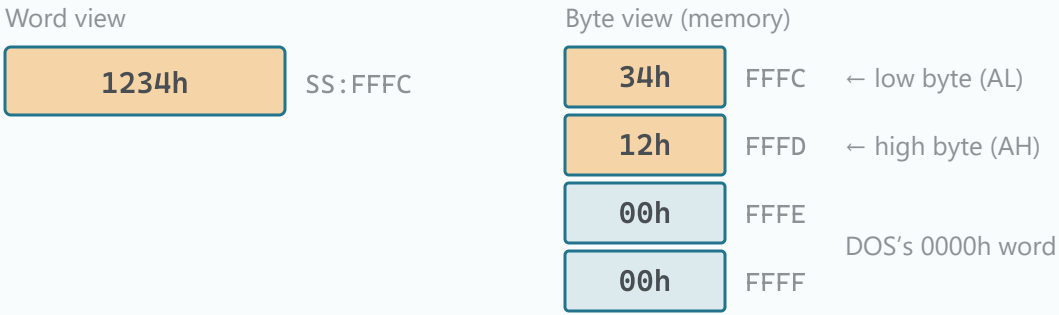


SS:offset · low address at top

POP BX · 2

② SP ← SP + 2 = **FFFEh**. The 1234h at FFFCh is **not erased**; it is just outside the stack now (faded).

Byte Order on the Stack (Little-Endian)



- The 8086 stores a word in memory **low byte first**: after `PUSH AX` (AX = 1234h), [FFFCh] = 34h and [FFFDh] = 12h.
- emu8086's *stack* window shows words; the *memory* window shows bytes. Both present the same data in different forms.
- Keep SP **even** at all times; on the 8086 a word access at an odd address costs an extra bus cycle.

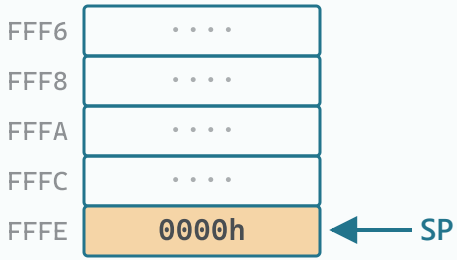
What Changes, What Doesn't?

Instruction	SP	Memory	Destination	Flags
`PUSH source`	−2	[SS:SP] ← source	—	unchanged
`POP dest`	+2	read, not erased	dest ← [SS:SP]	unchanged
`PUSHF`	−2	[SS:SP] ← FLAGS	—	unchanged
`POPF`	+2	read	FLAGS ← [SS:SP]	**all loaded**
`CALL` (near)	−2	[SS:SP] ← return IP	IP ← target	unchanged
`RET` (near)	+2	read	IP ← [SS:SP]	unchanged
`INT n`	−6	FLAGS, CS, IP	CS:IP ← vector	IF = TF = 0
`IRET`	+6	read	IP, CS, FLAGS	**all loaded**

So the result of a comparison (the flags) survives PUSH/POP safely; to save the flags themselves, use PUSHF/POPF.

First Example: `PUSH AX` / `POP BX`

```
1  ORG   100h
2  MOV   AX, 42      ; AX = 42
3  PUSH  AX          ; push AX
4  POP   BX          ; pop into BX
5  RET               ; return to the OS
6  END
```



SS:offset · low address at top

emu8086 .COM start-up: CS = DS = SS = ES = 0700h, IP = 0100h, SP = FFFEh.

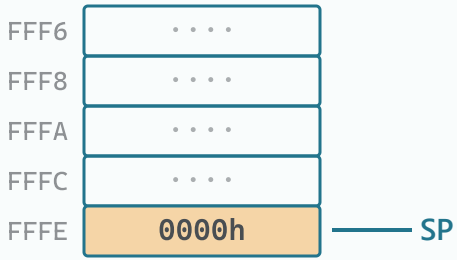
Step 1 / 5

VARIABLES

AX = 0000h BX = 0000h SP = FFFEh IP = 0100h

First Example: `PUSH AX` / `POP BX`

```
1 ORG   100h
▶ MOV   AX, 42      ; AX = 42
3 PUSH  AX          ; push AX
4 POP   BX          ; pop into BX
5 RET               ; return to the OS
6 END
```



SS:offset · low address at top

`MOV AX, 42` : 42 is **decimal** → AX = **002Ah** (not 42h!).

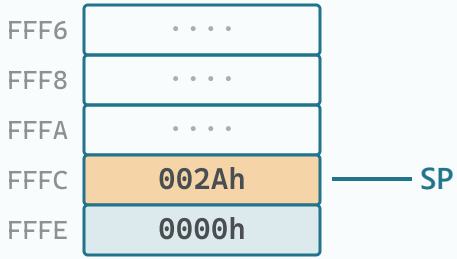
Step 2 / 5

VARIABLES

`AX = 002Ah` `BX = 0000h` `SP = FFFEh` `IP = 0103h`

First Example: `PUSH AX` / `POP BX`

```
1 ORG   100h
2 MOV   AX, 42      ; AX = 42
▶ PUSH  AX          ; push AX
4 POP   BX          ; pop into BX
5 RET               ; return to the OS
6 END
```



SS:offset · low address at top

`PUSH AX` : SP = FFFEh − 2 = **FFFCh**; [FFFCh] ← 002Ah.

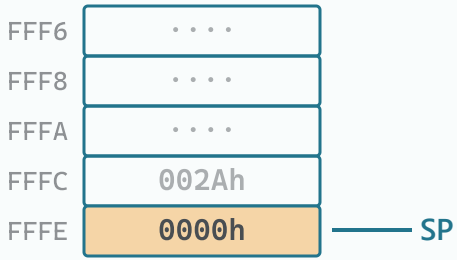
Step 3 / 5

VARIABLES

`AX = 002Ah` `BX = 0000h` `SP = FFFCh` `IP = 0104h`

First Example: `PUSH AX` / `POP BX`

```
1 ORG   100h
2 MOV   AX, 42      ; AX = 42
3 PUSH  AX          ; push AX
▶ POP   BX          ; pop into BX
5 RET               ; return to the OS
6 END
```



SS:offset · low address at top

`POP BX` : BX ← [FFFCh] = 002Ah; SP is back to **FFFEh**. The value was copied from AX to BX through the stack.

Step 4 / 5

VARIABLES

AX = **002Ah** BX = **002Ah** SP = **FFFEh** IP = **0105h**

First Example: `PUSH AX` / `POP BX`

```
1 ORG   100h
2 MOV   AX, 42      ; AX = 42
3 PUSH  AX          ; push AX
4 POP   BX          ; pop into BX
▶ RET               ; return to the OS
6 END
```



SS:offset · low address at top

`RET` : pops the **0000h** on the stack, IP = 0000h → the `INT 20h` in the PSP ends the program.

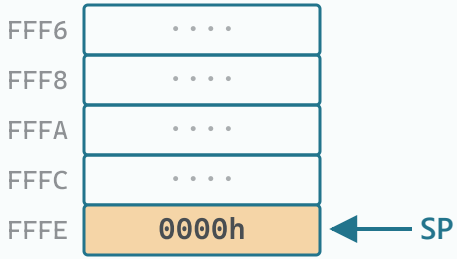
Step 5 / 5

VARIABLES

`AX = 002Ah` `BX = 002Ah` `SP = 0000h` `IP = 0000h`

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1  MOV   AX, 1      ; AX = 1
2  PUSH  AX         ; push 1
3  MOV   AX, 2      ; AX = 2
4  PUSH  AX         ; push 2
5  MOV   AX, 3      ; AX = 3
6  PUSH  AX         ; push 3
7  POP   BX         ; BX ← 3
8  POP   BX         ; BX ← 2
9  POP   BX         ; BX ← 1
```



SS:offset · low address at top

Start: SP = FFFEh.

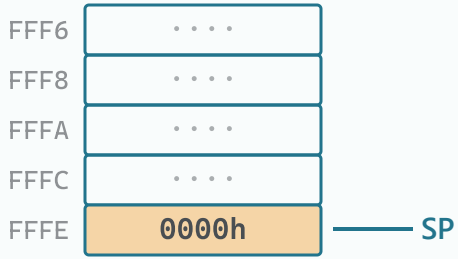
Step 1 / 10

VARIABLES

AX = 0000h BX = 0000h SP = FFFEh

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
▶ MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
3 MOV   AX, 2      ; AX = 2
4 PUSH  AX         ; push 2
5 MOV   AX, 3      ; AX = 3
6 PUSH  AX         ; push 3
7 POP   BX         ; BX ← 3
8 POP   BX         ; BX ← 2
9 POP   BX         ; BX ← 1
```



SS:offset · low address at top

AX = 0001h.

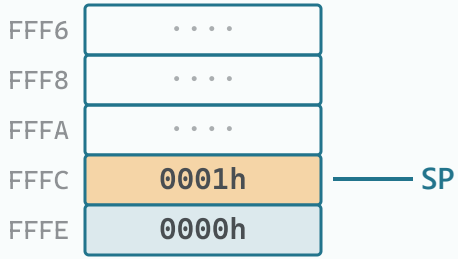
Step 2 / 10

VARIABLES

AX = 0001h BX = 0000h SP = FFFEh

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1  MOV   AX, 1      ; AX = 1
▶  PUSH  AX         ; push 1
3  MOV   AX, 2      ; AX = 2
4  PUSH  AX         ; push 2
5  MOV   AX, 3      ; AX = 3
6  PUSH  AX         ; push 3
7  POP   BX         ; BX ← 3
8  POP   BX         ; BX ← 2
9  POP   BX         ; BX ← 1
```



SS:offset · low address at top

PUSH: SP = FFFCh, [FFFCh] = 0001h.

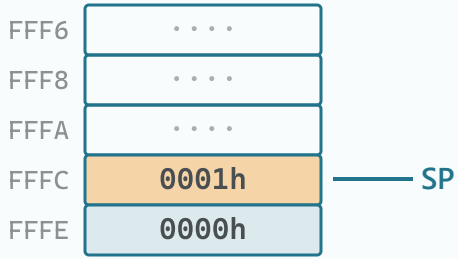
Step 3 / 10

VARIABLES

AX = 0001h BX = 0000h SP = FFFCh

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1 MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
▶ MOV   AX, 2      ; AX = 2
4 PUSH  AX         ; push 2
5 MOV   AX, 3      ; AX = 3
6 PUSH  AX         ; push 3
7 POP   BX         ; BX ← 3
8 POP   BX         ; BX ← 2
9 POP   BX         ; BX ← 1
```



SS:offset · low address at top

AX = 0002h (the old value of AX is safe on the stack).

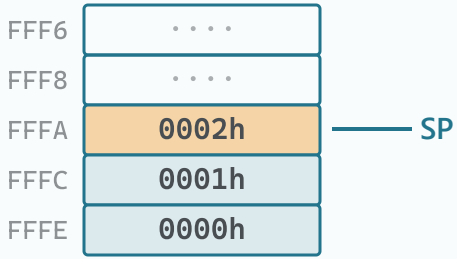
Step 4 / 10

VARIABLES

AX = 0002h BX = 0000h SP = FFFCh

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1 MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
3 MOV   AX, 2      ; AX = 2
▶ PUSH  AX         ; push 2
5 MOV   AX, 3      ; AX = 3
6 PUSH  AX         ; push 3
7 POP   BX         ; BX ← 3
8 POP   BX         ; BX ← 2
9 POP   BX         ; BX ← 1
```



SS:offset · low address at top

PUSH: SP = FFFAh, [FFFAh] = 0002h.

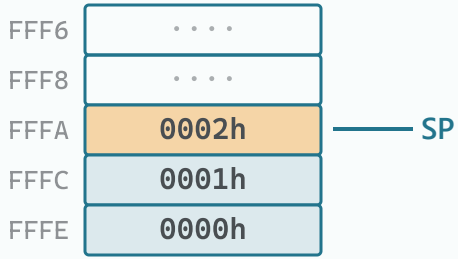
Step 5 / 10

VARIABLES

AX = 0002h BX = 0000h SP = FFFAh

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1 MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
3 MOV   AX, 2      ; AX = 2
4 PUSH  AX         ; push 2
▶ MOV   AX, 3      ; AX = 3
6 PUSH  AX         ; push 3
7 POP   BX         ; BX ← 3
8 POP   BX         ; BX ← 2
9 POP   BX         ; BX ← 1
```



SS:offset · low address at top

AX = 0003h.

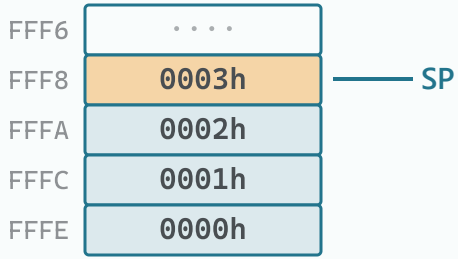
Step 6 / 10

VARIABLES

AX = 0003h BX = 0000h SP = FFFAh

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1 MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
3 MOV   AX, 2      ; AX = 2
4 PUSH  AX         ; push 2
5 MOV   AX, 3      ; AX = 3
▶ PUSH  AX         ; push 3
7 POP   BX         ; BX ← 3
8 POP   BX         ; BX ← 2
9 POP   BX         ; BX ← 1
```



SS:offset · low address at top

PUSH: SP = **FFF8h**; three values on the stack.

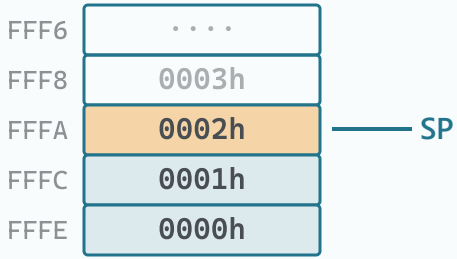
Step 7 / 10

VARIABLES

AX = **0003h** BX = **0000h** SP = **FFF8h**

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1 MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
3 MOV   AX, 2      ; AX = 2
4 PUSH  AX         ; push 2
5 MOV   AX, 3      ; AX = 3
6 PUSH  AX         ; push 3
▶ POP   BX         ; BX ← 3
8 POP   BX         ; BX ← 2
9 POP   BX         ; BX ← 1
```



SS:offset · low address at top

POP BX → **3** (last in, first out). SP = FFFAh.

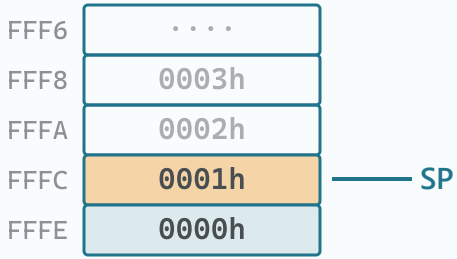
Step 8 / 10

VARIABLES

AX = **0003h** BX = **0003h** SP = **FFFAh**

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1 MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
3 MOV   AX, 2      ; AX = 2
4 PUSH  AX         ; push 2
5 MOV   AX, 3      ; AX = 3
6 PUSH  AX         ; push 3
7 POP   BX         ; BX ← 3
▶ POP   BX         ; BX ← 2
9 POP   BX         ; BX ← 1
```



SS:offset · low address at top

POP BX → **2**. SP = FFFCh.

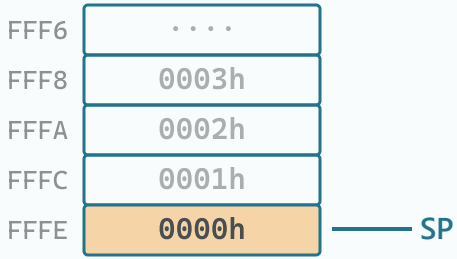
Step 9 / 10

VARIABLES

AX = **0003h** BX = **0002h** SP = **FFFCh**

LIFO Order: 1, 2, 3 In → 3, 2, 1 Out

```
1 MOV   AX, 1      ; AX = 1
2 PUSH  AX         ; push 1
3 MOV   AX, 2      ; AX = 2
4 PUSH  AX         ; push 2
5 MOV   AX, 3      ; AX = 3
6 PUSH  AX         ; push 3
7 POP   BX         ; BX ← 3
8 POP   BX         ; BX ← 2
▶ POP   BX         ; BX ← 1
```



SS:offset · low address at top

POP BX → **1**. SP = FFFEh; the stack is back to its initial state.

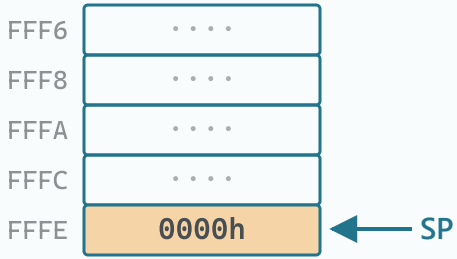
Step 10 / 10

VARIABLES

AX = **0003h** BX = **0001h** SP = **FFFEh**

Saving a Register's Value

```
1  ORG   100h
2  MOV   AX, 1234h
3  PUSH  AX          ; save AX
4  MOV   AX, 5678h   ; change AX
5  POP   AX          ; restore old value
6  RET
7  END
```



SS:offset · low address at top

Goal: borrow AX for another job, then restore its old value.

Step 1 / 6

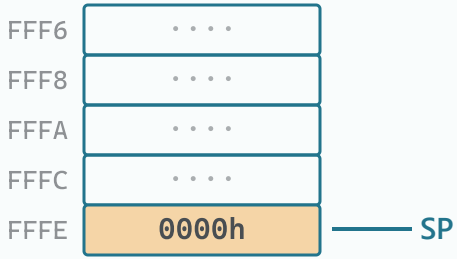
VARIABLES

AX = **0000h** SP = **FFFEh**

Procedures follow exactly this pattern: PUSH the registers they use on entry, POP them on exit.

Saving a Register's Value

```
1 ORG   100h
▶ MOV   AX, 1234h
3 PUSH  AX          ; save AX
4 MOV   AX, 5678h   ; change AX
5 POP   AX          ; restore old value
6 RET
7 END
```



SS:offset · low address at top

AX = 1234h.

Step 2 / 6

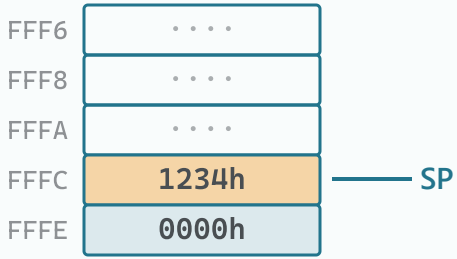
VARIABLES

AX = 1234h SP = FFFEh

Procedures follow exactly this pattern: PUSH the registers they use on entry, POP them on exit.

Saving a Register's Value

```
1 ORG   100h
2 MOV   AX, 1234h
▶ PUSH  AX          ; save AX
4 MOV   AX, 5678h   ; change AX
5 POP   AX          ; restore old value
6 RET
7 END
```



SS:offset · low address at top

AX saved on the stack: [FFFCh] = 1234h.

Step 3 / 6

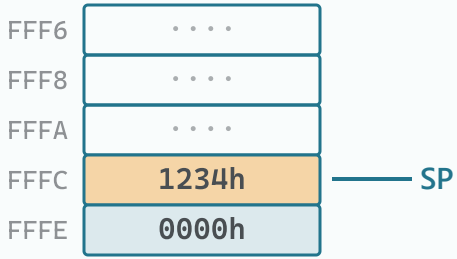
VARIABLES

AX = 1234h SP = FFFCh

Procedures follow exactly this pattern: PUSH the registers they use on entry, POP them on exit.

Saving a Register's Value

```
1 ORG   100h
2 MOV   AX, 1234h
3 PUSH  AX          ; save AX
▶ MOV   AX, 5678h   ; change AX
5 POP   AX          ; restore old value
6 RET
7 END
```



SS:offset · low address at top

AX used for another job: AX = 5678h.

Step 4 / 6

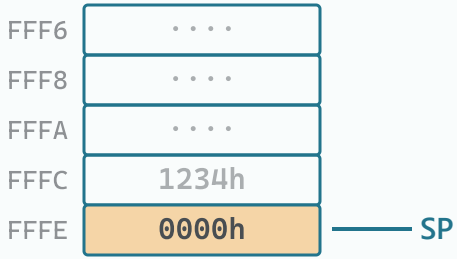
VARIABLES

AX = 5678h SP = FFFCh

Procedures follow exactly this pattern: PUSH the registers they use on entry, POP them on exit.

Saving a Register's Value

```
1 ORG   100h
2 MOV   AX, 1234h
3 PUSH  AX          ; save AX
4 MOV   AX, 5678h   ; change AX
▶ POP   AX          ; restore old value
6 RET
7 END
```



SS:offset · low address at top

`POP AX` : AX gets its old value back → **1234h**.

Step 5 / 6

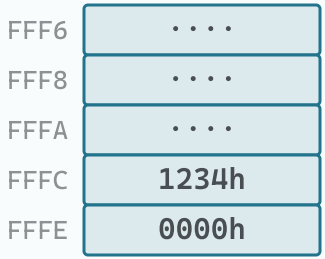
VARIABLES

`AX = 1234h` `SP = FFFEh`

Procedures follow exactly this pattern: PUSH the registers they use on entry, POP them on exit.

Saving a Register's Value

```
1 ORG   100h
2 MOV   AX, 1234h
3 PUSH  AX          ; save AX
4 MOV   AX, 5678h   ; change AX
5 POP   AX          ; restore old value
▶ RET
7 END
```



SS:offset · low address at top

RET: return to 0000h, the program ends.

Step 6 / 6

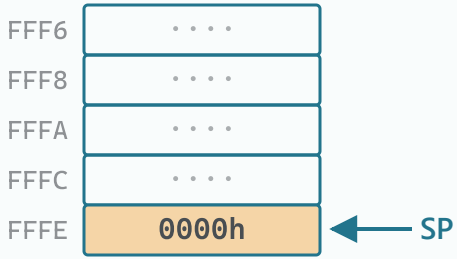
VARIABLES

AX = **1234h** SP = **0000h**

Procedures follow exactly this pattern: PUSH the registers they use on entry, POP them on exit.

Swapping Two Registers with the Stack

```
▶ ORG   100h
2 MOV   AX, 1212h   ; AX ← 1212h
3 MOV   BX, 3434h   ; BX ← 3434h
4 PUSH  AX          ; push AX
5 PUSH  BX          ; push BX
6 POP   AX          ; AX ← old BX
7 POP   BX          ; BX ← old AX
8 RET
9 END
```



SS:offset · low address at top

In emu8086 after loading: CX = 000Bh (program size in bytes: 3+3+1+1+1+1+1 = 11), SP = FFFEh.

Step 1 / 8

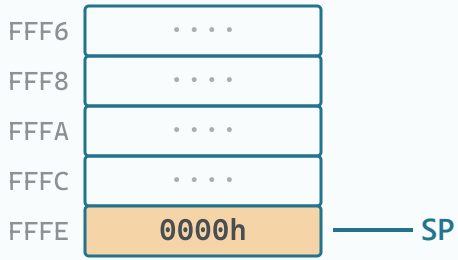
VARIABLES

AX = **0000h** BX = **0000h** CX = **000Bh** SP = **FFFEh**

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Swapping Two Registers with the Stack

```
1  ORG   100h
▶  MOV   AX, 1212h  ; AX ← 1212h
3  MOV   BX, 3434h  ; BX ← 3434h
4  PUSH  AX         ; push AX
5  PUSH  BX         ; push BX
6  POP   AX         ; AX ← old BX
7  POP   BX         ; BX ← old AX
8  RET
9  END
```



SS:offset · low address at top

AX = 1212h.

Step 2 / 8

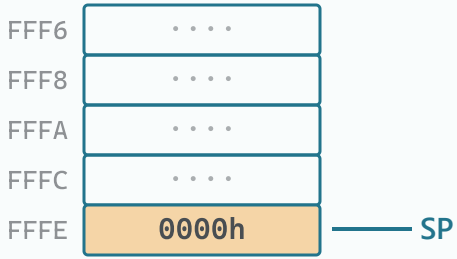
VARIABLES

AX = **1212h** BX = **0000h** CX = **000Bh** SP = **FFFEh**

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Swapping Two Registers with the Stack

```
1 ORG   100h
2 MOV   AX, 1212h  ; AX ← 1212h
▶ MOV   BX, 3434h  ; BX ← 3434h
4 PUSH  AX         ; push AX
5 PUSH  BX         ; push BX
6 POP   AX         ; AX ← old BX
7 POP   BX         ; BX ← old AX
8 RET
9 END
```



SS:offset · low address at top

BX = 3434h.

Step 3 / 8

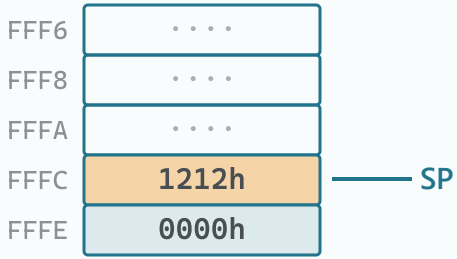
VARIABLES

AX = **1212h** | BX = **3434h** | CX = **000Bh** | SP = **FFFEh**

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Swapping Two Registers with the Stack

```
1 ORG   100h
2 MOV   AX, 1212h   ; AX ← 1212h
3 MOV   BX, 3434h   ; BX ← 3434h
▶ PUSH  AX          ; push AX
5 PUSH  BX          ; push BX
6 POP   AX          ; AX ← old BX
7 POP   BX          ; BX ← old AX
8 RET
9 END
```



SS:offset · low address at top

PUSH AX: [FFFCh] = 1212h, SP = FFFCh.

Step 4 / 8

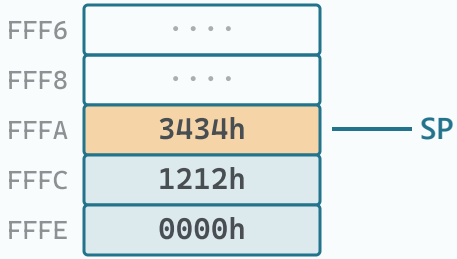
VARIABLES

AX = 1212h BX = 3434h CX = 000Bh SP = FFFCh

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Swapping Two Registers with the Stack

```
1 ORG   100h
2 MOV   AX, 1212h   ; AX ← 1212h
3 MOV   BX, 3434h   ; BX ← 3434h
4 PUSH  AX          ; push AX
▶ PUSH  BX          ; push BX
6 POP   AX          ; AX ← old BX
7 POP   BX          ; BX ← old AX
8 RET
9 END
```



SS:offset · low address at top

PUSH BX: [FFFAh] = 3434h, SP = FFFAh.

Step 5 / 8

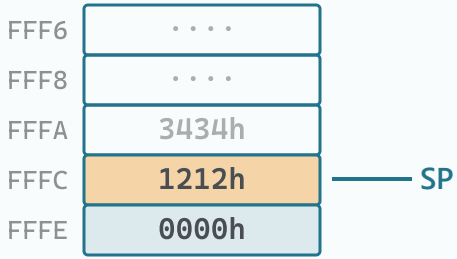
VARIABLES

AX = 1212h BX = 3434h CX = 000Bh SP = FFFAh

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Swapping Two Registers with the Stack

```
1  ORG   100h
2  MOV   AX, 1212h   ; AX ← 1212h
3  MOV   BX, 3434h   ; BX ← 3434h
4  PUSH  AX          ; push AX
5  PUSH  BX          ; push BX
▶  POP   AX          ; AX ← old BX
7  POP   BX          ; BX ← old AX
8  RET
9  END
```



SS:offset · low address at top

POP AX: the top value **3434h** goes to AX.

Step 6 / 8

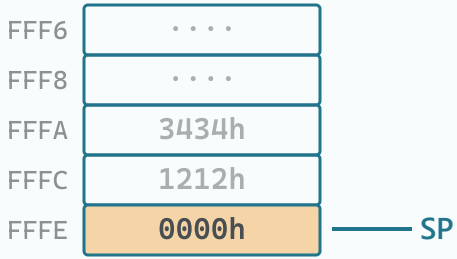
VARIABLES

AX = **3434h** BX = **3434h** CX = **000Bh** SP = **FFFCh**

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Swapping Two Registers with the Stack

```
1  ORG   100h
2  MOV   AX, 1212h   ; AX ← 1212h
3  MOV   BX, 3434h   ; BX ← 3434h
4  PUSH  AX          ; push AX
5  PUSH  BX          ; push BX
6  POP   AX          ; AX ← old BX
▶  POP   BX          ; BX ← old AX
8  RET
9  END
```



SS:offset · low address at top

POP BX: the next value **1212h** goes to BX. Swapped!

Step 7 / 8

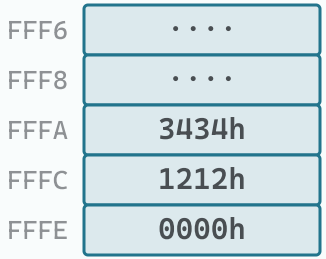
VARIABLES

AX = **3434h** BX = **1212h** CX = **000Bh** SP = **FFFEh**

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Swapping Two Registers with the Stack

```
1 ORG   100h
2 MOV   AX, 1212h  ; AX ← 1212h
3 MOV   BX, 3434h  ; BX ← 3434h
4 PUSH  AX         ; push AX
5 PUSH  BX         ; push BX
6 POP   AX         ; AX ← old BX
7 POP   BX         ; BX ← old AX
▶ RET
9 END
```



SS:offset · low address at top

RET: end of program.

Step 8 / 8

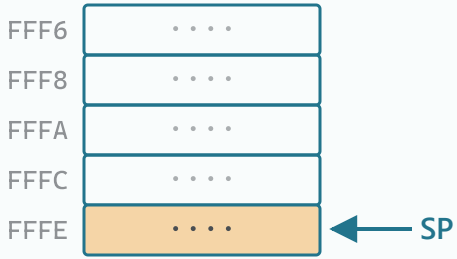
VARIABLES

AX = **3434h** BX = **1212h** CX = **000Bh** SP = **0000h**

Popping in mirror order (BX first) would leave the values where they were. LIFO is the whole trick.

Stack Usage: Three Registers

```
▶ 1 mov ax, 2
  2 mov bx, 4
  3 mov cx, 8
  4 push ax
  5 push bx
  6 push cx
  7 pop  ax
  8 pop  bx
  9 pop  cx
```



SS:offset · low address at top

Loaded as .BIN: CS = SS = 0100h, IP = 0000h, SP = FFFEh. The stack starts empty.

Step 1 / 10

VARIABLES

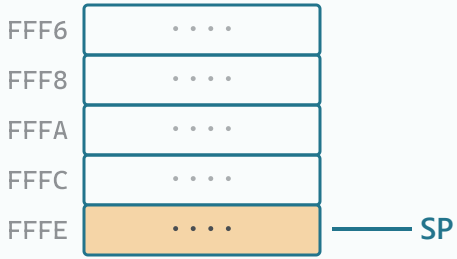
AX = **0000h** BX = **0000h** CX = **0000h**

SP = **FFFEh**

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
▶ mov ax, 2
2 mov bx, 4
3 mov cx, 8
4 push ax
5 push bx
6 push cx
7 pop  ax
8 pop  bx
9 pop  cx
```



SS:offset · low address at top

AX = 0002h.

Step 2 / 10

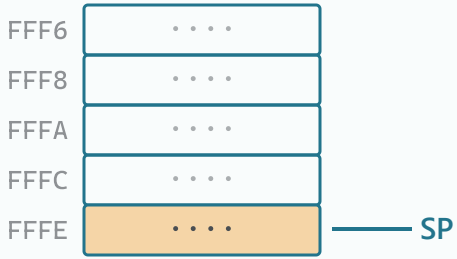
VARIABLES

AX = 0002h | BX = 0000h | CX = 0000h | SP = FFFEh

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
▶ mov bx, 4
3 mov cx, 8
4 push ax
5 push bx
6 push cx
7 pop  ax
8 pop  bx
9 pop  cx
```



SS:offset · low address at top

BX = 0004h.

Step 3 / 10

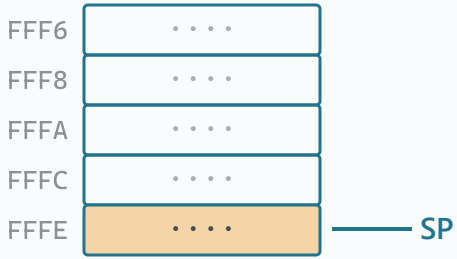
VARIABLES

AX = 0002h BX = 0004h CX = 0000h SP = FFFEh

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
2 mov bx, 4
▶ mov cx, 8
4 push ax
5 push bx
6 push cx
7 pop  ax
8 pop  bx
9 pop  cx
```



SS:offset · low address at top

CX = 0008h.

Step 4 / 10

VARIABLES

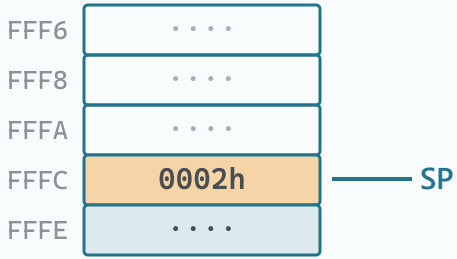
AX = **0002h** BX = **0004h** CX = **0008h**

SP = **FFFEh**

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
2 mov bx, 4
3 mov cx, 8
▶ push ax
5 push bx
6 push cx
7 pop  ax
8 pop  bx
9 pop  cx
```



SS:offset · low address at top

push ax: SP = FFFCh, [FFFCh] = 0002h.

Step 5 / 10

VARIABLES

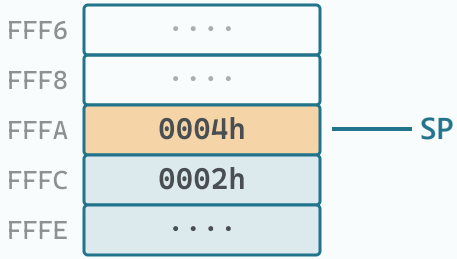
AX = 0002h BX = 0004h CX = 0008h

SP = FFFCh

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
2 mov bx, 4
3 mov cx, 8
4 push ax
▶ push bx
6 push cx
7 pop  ax
8 pop  bx
9 pop  cx
```



SS:offset · low address at top

push bx: SP = FFFAh, [FFFAh] = 0004h.

Step 6 / 10

VARIABLES

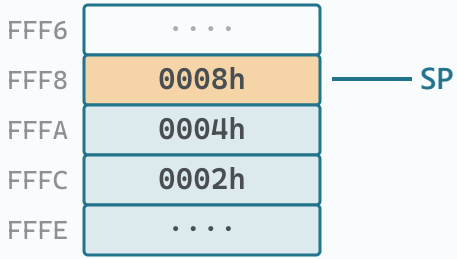
AX = 0002h BX = 0004h CX = 0008h

SP = FFFAh

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
2 mov bx, 4
3 mov cx, 8
4 push ax
5 push bx
▶ push cx
7 pop  ax
8 pop  bx
9 pop  cx
```



SS:offset · low address at top

push cx: SP = **FFF8h**, [FFF8h] = 0008h.

Step 7 / 10

VARIABLES

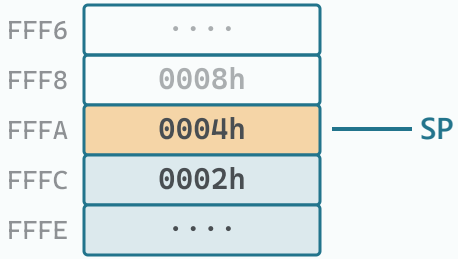
AX = **0002h** BX = **0004h** CX = **0008h**

SP = **FFF8h**

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
2 mov bx, 4
3 mov cx, 8
4 push ax
5 push bx
6 push cx
▶ pop  ax
8 pop  bx
9 pop  cx
```



SS:offset · low address at top

pop ax: AX ← 0008h (the top one).

Step 8 / 10

VARIABLES

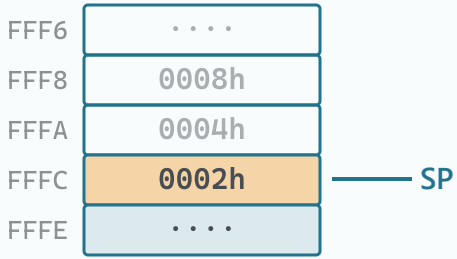
AX = 0008h BX = 0004h CX = 0008h

SP = FFFAh

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
2 mov bx, 4
3 mov cx, 8
4 push ax
5 push bx
6 push cx
7 pop  ax
▶ pop  bx
9 pop  cx
```



SS:offset · low address at top

pop bx: BX ← 0004h (the middle one; unchanged).

Step 9 / 10

VARIABLES

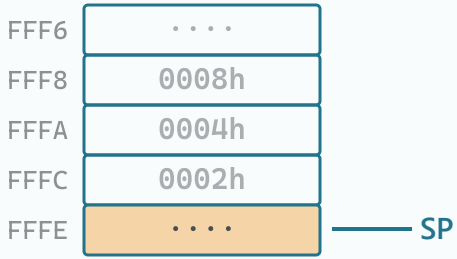
AX = 0008h BX = 0004h CX = 0008h

SP = FFFCh

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Stack Usage: Three Registers

```
1 mov ax, 2
2 mov bx, 4
3 mov cx, 8
4 push ax
5 push bx
6 push cx
7 pop  ax
8 pop  bx
▶ pop  cx
```



SS:offset · low address at top

pop cx: CX ← 0002h. Result: AX and CX have **swapped** values.

Step 10 / 10

VARIABLES

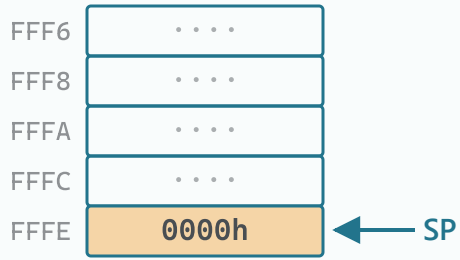
AX = **0008h** BX = **0004h** CX = **0002h**

SP = **FFFEh**

Source: single-stepping in emu8086. The stack window shows the area around SS:SP.

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

msg = 'ABC' is at address 0117h. Goal: print it in reverse order.

Step 1 / 29

VARIABLES

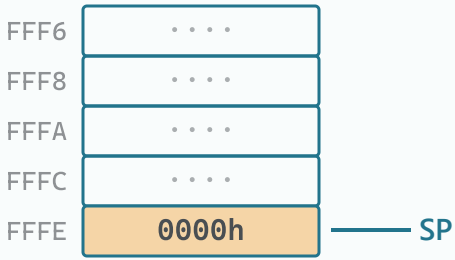
AX = 0000h CX = 0000h DX = 0000h

SI = 0000h SP = FFFEh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
▶  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

SI = **0117h** (address of the msg string).

Step 2 / 29

VARIABLES

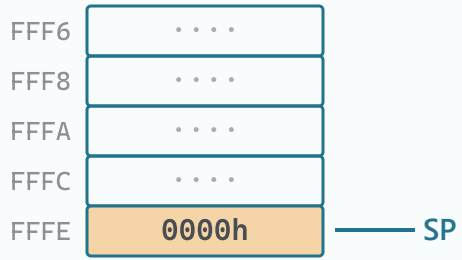
AX = **0000h** CX = **0000h** DX = **0000h**

SI = **0117h** SP = **FFFEh**

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
▶  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

Loop counter CX = 3.

Step 3 / 29

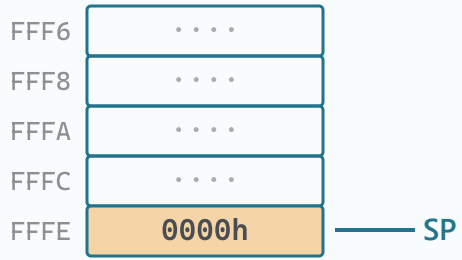
VARIABLES

AX = 0000h | CX = 0003h | DX = 0000h | SI = 0117h | SP = FFFEh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
▶  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

AL ← [SI] = 'A' (41h).

Step 4 / 29

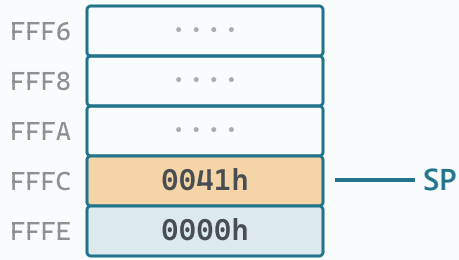
VARIABLES

AX = 0041h | CX = 0003h | DX = 0000h | SI = 0117h | SP = FFFEh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
▶       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

PUSH AX: 'A' goes on the stack as 0041h (AH = 00h).

Step 5 / 29

VARIABLES

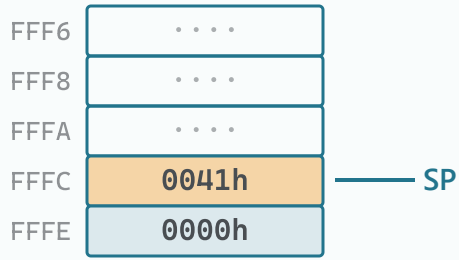
AX = 0041h CX = 0003h DX = 0000h

SI = 0117h SP = FFFCh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
▶       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

SI = 0118h: next character.

Step 6 / 29

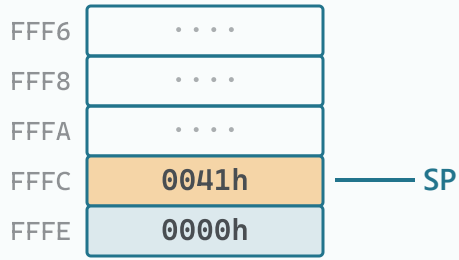
VARIABLES

AX = 0041h | CX = 0003h | DX = 0000h | SI = 0118h | SP = FFFCh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
▶       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

LOOP: CX = 2 → jump back.

Step 7 / 29

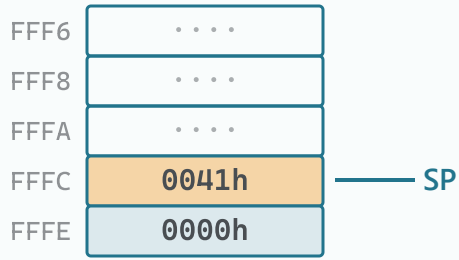
VARIABLES

AX = 0041h | CX = 0002h | DX = 0000h | SI = 0118h | SP = FFFCh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
▶ put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

AL ← [SI] = 'B' (42h).

Step 8 / 29

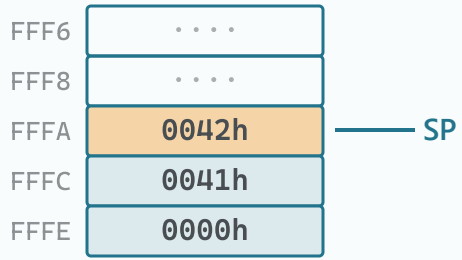
VARIABLES

AX = 0042h CX = 0002h DX = 0000h SI = 0118h SP = FFFCh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
▶       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

PUSH AX: 'B' goes on the stack as 0042h (AH = 00h).

Step 9 / 29

VARIABLES

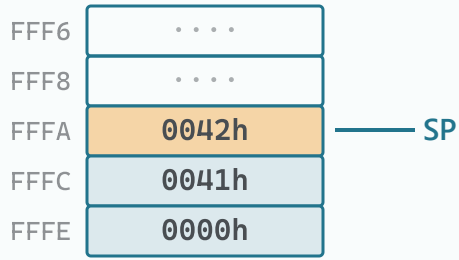
AX = 0042h CX = 0002h DX = 0000h

SI = 0118h SP = FFFAh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
▶       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

SI = 0119h: next character.

Step 10 / 29

VARIABLES

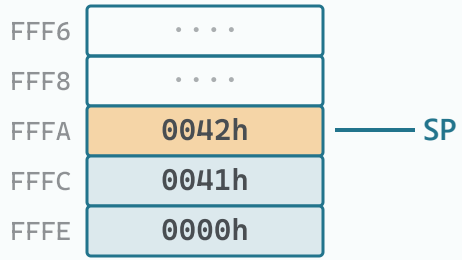
AX = 0042h CX = 0002h DX = 0000h

SI = 0119h SP = FFFAh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
▶       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

LOOP: CX = 1 → jump back.

Step 11 / 29

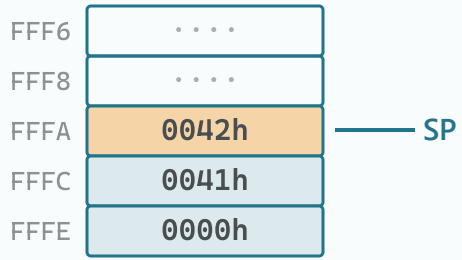
VARIABLES

AX = 0042h | CX = 0001h | DX = 0000h | SI = 0119h | SP = FFFAh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
▶ put: MOV  AL, [SI]
5      PUSH AX        ; save the char
6      INC  SI
7      LOOP put
8  MOV  CX, 3
9  prt: POP  DX       ; DL = char
10     MOV  AH, 2
11     INT  21h       ; print DL
12     LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

AL ← [SI] = 'C' (43h).

Step 12 / 29

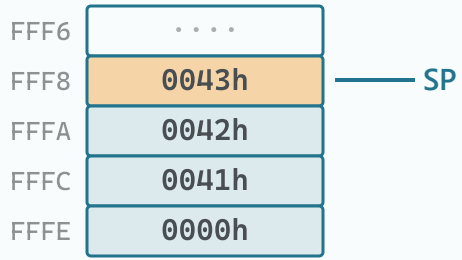
VARIABLES

AX = 0043h | CX = 0001h | DX = 0000h | SI = 0119h | SP = FFFAh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
▶       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

PUSH AX: 'C' goes on the stack as 0043h (AH = 00h).

Step 13 / 29

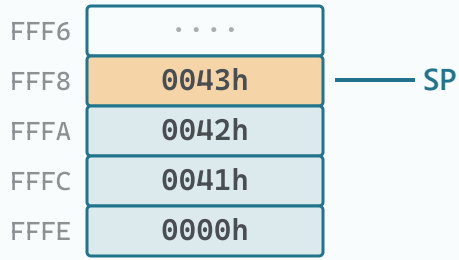
VARIABLES

AX = 0043h CX = 0001h DX = 0000h SI = 0119h SP = FFF8h

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
▶       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

SI = 011Ah: next character.

Step 14 / 29

VARIABLES

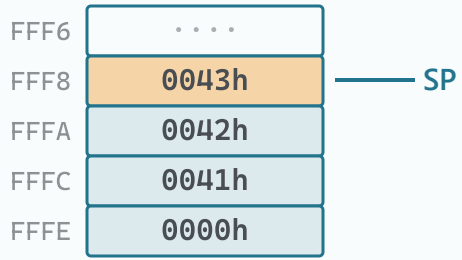
AX = 0043h CX = 0001h DX = 0000h

SI = 011Ah SP = FFF8h

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
▶       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

LOOP: CX = 0 → the loop ends.

Step 15 / 29

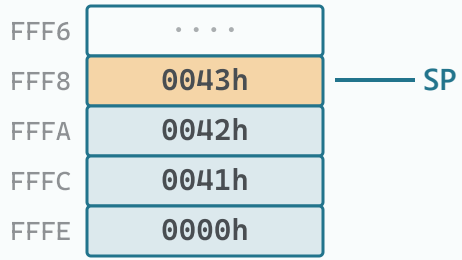
VARIABLES

AX = 0043h | CX = 0000h | DX = 0000h | SI = 011Ah | SP = FFF8h

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
▶ MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

CX = 3 for the second loop.

Step 16 / 29

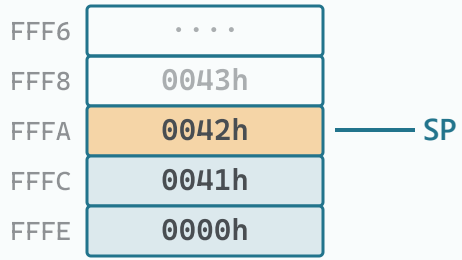
VARIABLES

AX = 0043h | CX = 0003h | DX = 0000h | SI = 011Ah | SP = FFF8h

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
▶  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

POP DX: DX = 0043h → DL = 'C' (last in, first out).

Step 17 / 29

VARIABLES

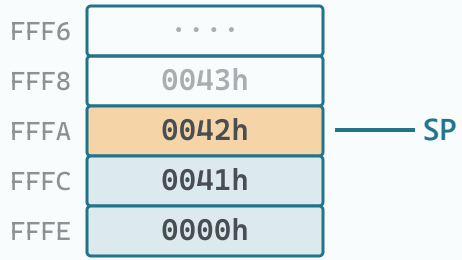
AX = 0043h CX = 0003h DX = 0043h

SI = 011Ah SP = FFFAh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
▶       MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

AH = 02h: DOS character output function.

Step 18 / 29

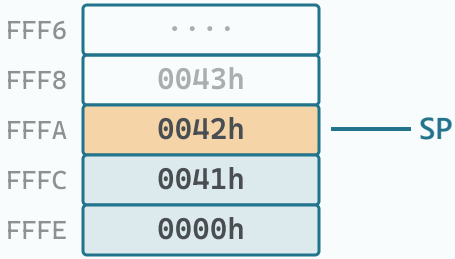
VARIABLES

AX = 0243h CX = 0003h DX = 0043h SI = 011Ah SP = FFFAh

OUTPUT

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
▶       INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

INT 21h: prints **C**; AL = DL = 43h.

Step 19 / 29

VARIABLES

AX = **0243h** CX = **0003h** DX = **0043h**

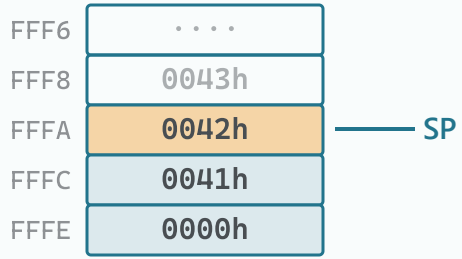
SI = **011Ah** SP = **FFFAh**

OUTPUT

```
C
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
▶       LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

LOOP: CX = 2 → jump back.

Step 20 / 29

VARIABLES

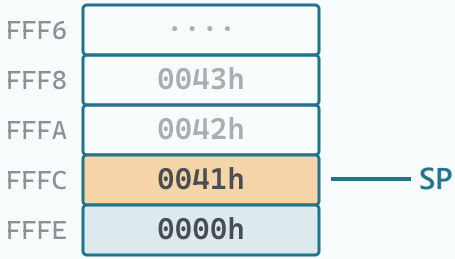
AX = 0243h | CX = 0002h | DX = 0043h | SI = 011Ah | SP = FFFAh

OUTPUT

```
C
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
▶  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

POP DX: DX = 0042h → DL = 'B' (last in, first out).

Step 21 / 29

VARIABLES

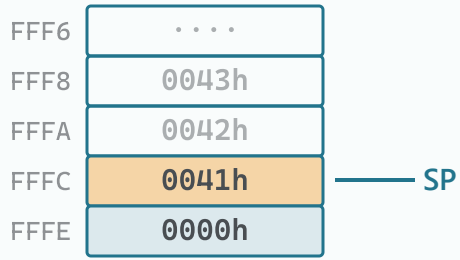
AX = 0243h CX = 0002h DX = 0042h SI = 011Ah SP = FFFCh

OUTPUT

```
C
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
▶       MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

AH = 02h: DOS character output function.

Step 22 / 29

VARIABLES

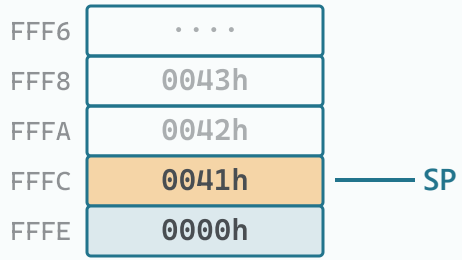
AX = 0243h | CX = 0002h | DX = 0042h | SI = 011Ah | SP = FFFCh

OUTPUT

```
C
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
▶       INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

INT 21h: prints **B**; AL = DL = 42h.

Step 23 / 29

VARIABLES

AX = **0242h** CX = **0002h** DX = **0042h**

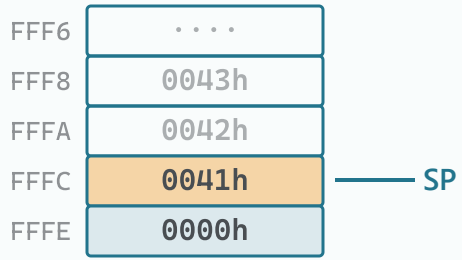
SI = **011Ah** SP = **FFFCh**

OUTPUT

```
CB
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
▶       LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

LOOP: CX = 1 → jump back.

Step 24 / 29

VARIABLES

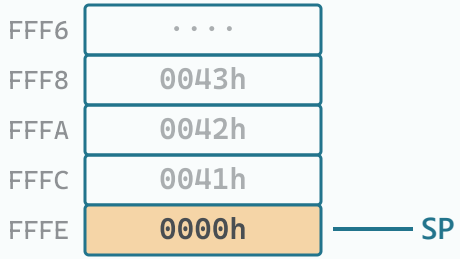
AX = 0242h | CX = 0001h | DX = 0042h | SI = 011Ah | SP = FFFCh

OUTPUT

```
CB
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
▶  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

POP DX: DX = 0041h → DL = 'A' (last in, first out).

Step 25 / 29

VARIABLES

AX = 0242h CX = 0001h DX = 0041h

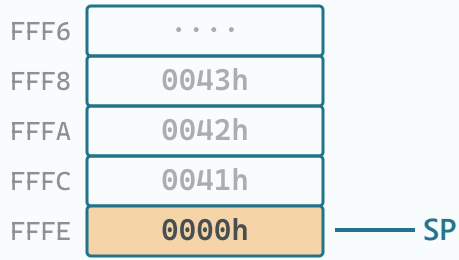
SI = 011Ah SP = FFFEh

OUTPUT

```
CB
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
▶       MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

AH = 02h: DOS character output function.

Step 26 / 29

VARIABLES

AX = **0242h** CX = **0001h** DX = **0041h**

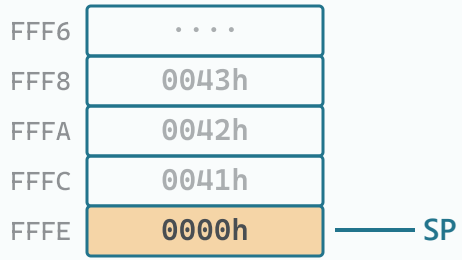
SI = **011Ah** SP = **FFFEh**

OUTPUT

```
CB
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
▶       INT  21h       ; print DL
12      LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

INT 21h: prints **A**; AL = DL = 41h.

Step 27 / 29

VARIABLES

AX = **0241h** CX = **0001h** DX = **0041h**

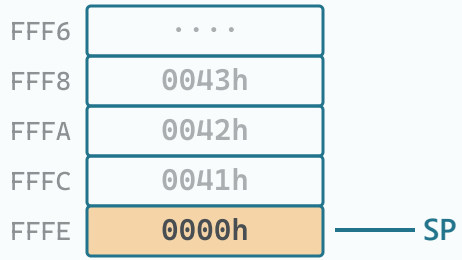
SI = **011Ah** SP = **FFFEh**

OUTPUT

```
CBA
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
▶       LOOP prt
13 RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

LOOP: CX = 0 → the loop ends.

Step 28 / 29

VARIABLES

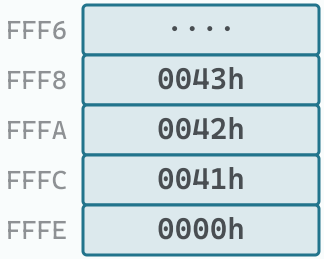
AX = **0241h** | CX = **0000h** | DX = **0041h** | SI = **011Ah** | SP = **FFFEh**

OUTPUT

```
CBA
```

Reversing a String with the Stack

```
1  ORG  100h
2  MOV  SI, OFFSET msg
3  MOV  CX, 3
4  put: MOV  AL, [SI]
5       PUSH AX        ; save the char
6       INC  SI
7       LOOP put
8  MOV  CX, 3
9  prt: POP  DX        ; DL = char
10      MOV  AH, 2
11      INT  21h       ; print DL
12      LOOP prt
▶ RET
14 msg DB 'ABC'      ; 0117h
```



SS:offset · low address at top

The screen shows **CBA**: the stack reversed the string. RET ends the program.

Step 29 / 29

VARIABLES

AX = 0241h CX = 0000h DX = 0041h SI = 011Ah SP = 0000h

OUTPUT

```
CBA
```

EXAMPLES

What Happens to the Registers?

After the code runs, what are AX, BX, CX and SP? (Start: SP = FFFEh)

```
MOV AX, 1111h
MOV BX, 2222h
MOV CX, 3333h
PUSH AX
PUSH BX
PUSH CX
POP AX
POP CX
PUSH AX
POP BX
POP BX
```

Step 1 / 5

EXAMPLES

What Happens to the Registers?

After the code runs, what are AX, BX, CX and SP? (Start: SP = FFFEh)

```
MOV AX, 1111h
MOV BX, 2222h
MOV CX, 3333h
PUSH AX
PUSH BX
PUSH CX
POP AX
POP CX
PUSH AX
POP BX
POP BX
```

After the PUSHes the stack holds (top to bottom): 3333h, 2222h, 1111h; SP = FFF8h.

Step 2 / 5

EXAMPLES

What Happens to the Registers?

After the code runs, what are AX, BX, CX and SP? (Start: SP = FFFEh)

```
MOV AX, 1111h
MOV BX, 2222h
MOV CX, 3333h
PUSH AX
PUSH BX
PUSH CX
POP AX
POP CX
PUSH AX
POP BX
POP BX
```

After the PUSHes the stack holds (top to bottom): 3333h, 2222h, 1111h; SP = FFF8h.

POP AX → AX = **3333h**; POP CX → CX = **2222h**; SP = FFFCh.

Step 3 / 5

EXAMPLES

What Happens to the Registers?

After the code runs, what are AX, BX, CX and SP? (Start: SP = FFFEh)

```
MOV AX, 1111h
MOV BX, 2222h
MOV CX, 3333h
PUSH AX
PUSH BX
PUSH CX
POP AX
POP CX
PUSH AX
POP BX
POP BX
```

After the PUSHes the stack holds (top to bottom): 3333h, 2222h, 1111h; SP = FFF8h.

POP AX → AX = **3333h**; POP CX → CX = **2222h**; SP = FFFCh.

PUSH AX (3333h), POP BX → BX = 3333h; then POP BX → BX = **1111h**.

Step 4 / 5

EXAMPLES

What Happens to the Registers?

After the code runs, what are AX, BX, CX and SP? (Start: SP = FFFEh)

```
MOV AX, 1111h
MOV BX, 2222h
MOV CX, 3333h
PUSH AX
PUSH BX
PUSH CX
POP AX
POP CX
PUSH AX
POP BX
POP BX
```

After the PUSHes the stack holds (top to bottom): 3333h, 2222h, 1111h; SP = FFF8h.

POP AX → AX = **3333h**; POP CX → CX = **2222h**; SP = FFFCh.

PUSH AX (3333h), POP BX → BX = 3333h; then POP BX → BX = **1111h**.

Result: AX = 3333h, BX = 1111h, CX = 2222h, SP = **FFFEh** (4 PUSHes, 4 POPs → balanced).

Step 5 / 5

EXAMPLES

Comparing Ways to Swap

With the stack

```
PUSH AX
PUSH BX
POP  AX
POP  BX
```

4 bytes · 4 memory accesses. Works for any register pair, segment registers included.

With a temp register

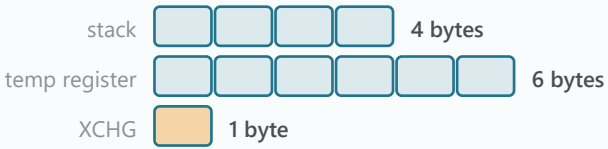
```
MOV CX, AX
MOV AX, BX
MOV BX, CX
```

6 bytes · no memory access, but CX is **clobbered**.

With XCHG

```
XCHG AX, BX
```

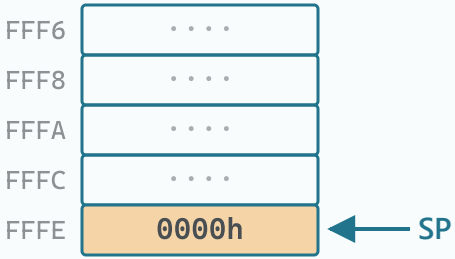
1 byte (93h) · the shortest and fastest way. Cannot be used with segment registers.



The real value of the stack is not swapping but **saving**: preserving registers during a procedure or interrupt, holding intermediate results, copying segment registers (`PUSH DS / POP ES`).

Saving the Flags: PUSHF / POPF

```
1  ORG   100h
2  MOV   AX, 0FFFFh
3  ADD   AX, 1       ; CF=1, ZF=1
4  PUSHF             ; save flags
5  MOV   BX, 5
6  ADD   BX, 3       ; CF=0, ZF=0
7  POPF              ; restore flags
8  RET
```



SS:offset · low address at top

Start: flags clear (IF = 1).

Step 1 / 8

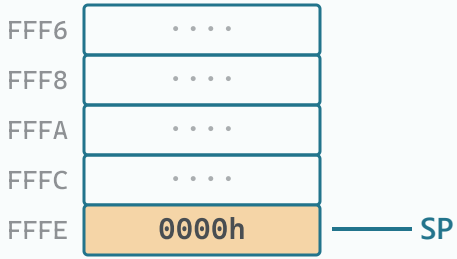
VARIABLES

AX = 0000h BX = 0000h SP = FFFEh CF = 0 ZF = 0

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Saving the Flags: PUSHF / POPF

```
1  ORG   100h
▶  MOV   AX, 0FFFFh
3  ADD   AX, 1       ; CF=1, ZF=1
4  PUSHF             ; save flags
5  MOV   BX, 5
6  ADD   BX, 3       ; CF=0, ZF=0
7  POPF              ; restore flags
8  RET
```



SS:offset · low address at top

AX = FFFFh.

Step 2 / 8

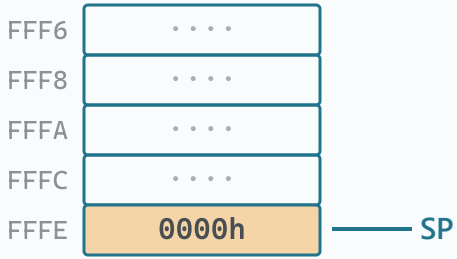
VARIABLES

AX = FFFFh | BX = 0000h | SP = FFFEh | CF = 0 | ZF = 0

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Saving the Flags: PUSHF / POPF

```
1 ORG   100h
2 MOV   AX, 0FFFFh
▶ ADD   AX, 1       ; CF=1, ZF=1
4 PUSHF             ; save flags
5 MOV   BX, 5
6 ADD   BX, 3       ; CF=0, ZF=0
7 POPF              ; restore flags
8 RET
```



SS:offset · low address at top

FFFFh + 1 = 1 0000h → AX = 0000h, **CF = 1, ZF = 1**.

Step 3 / 8

VARIABLES

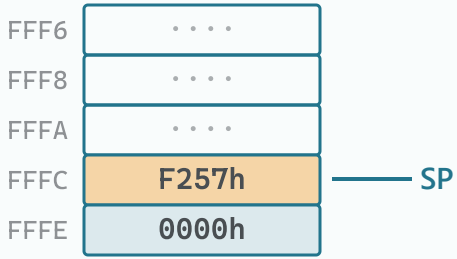
AX = **0000h** BX = **0000h** SP = **FFFEh**

CF = **1** ZF = **1**

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Saving the Flags: PUSHF / POPF

```
1  ORG   100h
2  MOV   AX, 0FFFFh
3  ADD   AX, 1       ; CF=1, ZF=1
▶  PUSHF             ; save flags
5  MOV   BX, 5
6  ADD   BX, 3       ; CF=0, ZF=0
7  POPF              ; restore flags
8  RET
```



SS:offset · low address at top

PUSHF: the FLAGS word (**F257h**: CF, PF, AF, ZF, IF and the 8086's always-1 bits) is pushed.

Step 4 / 8

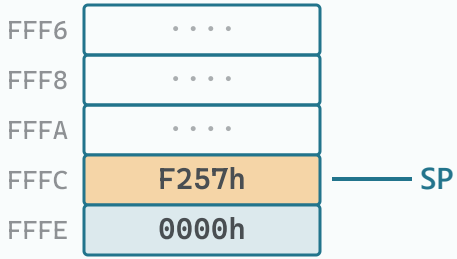
VARIABLES

AX = **0000h** BX = **0000h** SP = **FFFCh** CF = **1** ZF = **1**

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Saving the Flags: PUSHF / POPF

```
1  ORG   100h
2  MOV   AX, 0FFFFh
3  ADD   AX, 1       ; CF=1, ZF=1
4  PUSHF             ; save flags
▶  MOV   BX, 5
6  ADD   BX, 3       ; CF=0, ZF=0
7  POPF              ; restore flags
8  RET
```



BX = 5.

Step 5 / 8

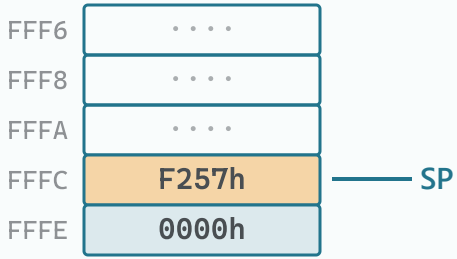
VARIABLES

AX = 0000h | BX = 0005h | SP = FFFCh | CF = 1 | ZF = 1

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Saving the Flags: PUSHF / POPF

```
1  ORG   100h
2  MOV   AX, 0FFFFh
3  ADD   AX, 1       ; CF=1, ZF=1
4  PUSHF             ; save flags
5  MOV   BX, 5
▶  ADD   BX, 3       ; CF=0, ZF=0
7  POPF              ; restore flags
8  RET
```



SS:offset · low address at top

5 + 3 = 8 → CF = 0, ZF = 0. The earlier result flags are lost…

Step 6 / 8

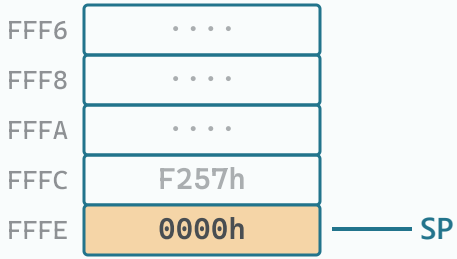
VARIABLES

AX = 0000h BX = 0008h SP = FFFCh CF = 0 ZF = 0

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Saving the Flags: PUSHF / POPF

```
1 ORG   100h
2 MOV   AX, 0FFFFh
3 ADD   AX, 1       ; CF=1, ZF=1
4 PUSHF             ; save flags
5 MOV   BX, 5
6 ADD   BX, 3       ; CF=0, ZF=0
▶ POPF              ; restore flags
8 RET
```



SS:offset · low address at top

...until POPF brings them back: **CF = 1, ZF = 1**.

Step 7 / 8

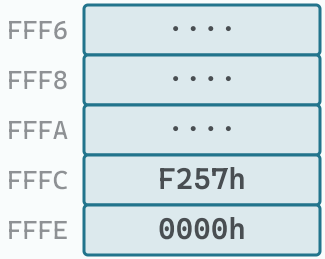
VARIABLES

AX = 0000h BX = 0008h SP = FFFEh CF = 1 ZF = 1

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Saving the Flags: PUSHF / POPF

```
1 ORG   100h
2 MOV   AX, 0FFFFh
3 ADD   AX, 1       ; CF=1, ZF=1
4 PUSHF             ; save flags
5 MOV   BX, 5
6 ADD   BX, 3       ; CF=0, ZF=0
7 POPF              ; restore flags
▶ RET
```



The program ends.

Step 8 / 8

VARIABLES

AX = 0000h BX = 0008h SP = 0000h CF = 1 ZF = 1

Interrupt routines need not save the flags themselves; INT does it automatically. In ordinary code, PUSHF/POPF is needed to preserve a result.

Stack Safety

- PUSH and POP hold temporary data while the program runs.
- Keep the stack intact with **equal numbers of PUSHes and POPs** (on every path, in every branch!).
- If PUSHes and POPs are unbalanced, the stack is corrupted and the program may not return to the operating system.
- `RET` expects a **return address** on top of the stack. In a .COM program DOS pushes **0000h** at start-up: RET goes to CS:0000h (the PSP), where an `INT 20h` ends the program.

Extra PUSH

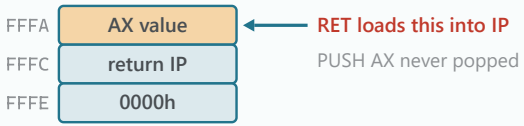
RET takes the last pushed data instead of the return address → a jump to a random address.

Extra POP

The return address is "eaten" by a register; RET returns to the wrong place.

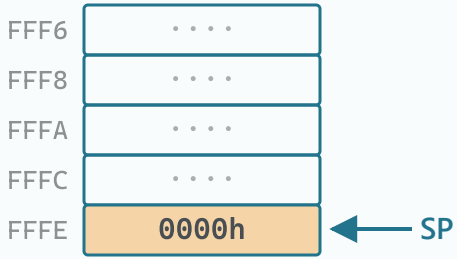
Wrong order

Values get mixed up (as in a swap) and the program silently misbehaves.



Bug Example: Unbalanced PUSH

```
1 ORG   100h
2 MOV   AX, 1234h
3 PUSH  AX          ; no POP!
4 RET               ; IP ← ?
```



SS:offset · low address at top

The program is expected to return to DOS with RET at the end.

Step 1 / 4

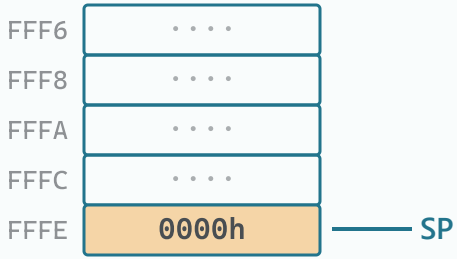
VARIABLES

AX = 0000h SP = FFFEh IP = 0100h

Fix: add POP AX before RET, or don't push at all. Count the PUSHes and POPs of every branch (JMP/Jcc) separately.

Bug Example: Unbalanced PUSH

```
1 ORG   100h
▶ MOV   AX, 1234h
3 PUSH  AX          ; no POP!
4 RET               ; IP ← ?
```



SS:offset · low address at top

AX = 1234h.

Step 2 / 4

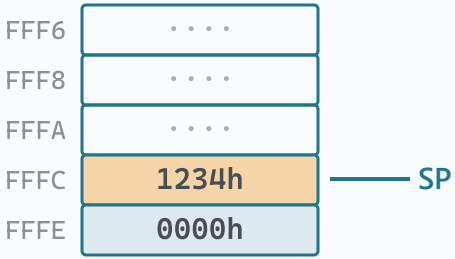
VARIABLES

AX = 1234h SP = FFFEh IP = 0103h

Fix: add POP AX before RET, or don't push at all. Count the PUSHes and POPs of every branch (JMP/Jcc) separately.

Bug Example: Unbalanced PUSH

```
1 ORG   100h
2 MOV   AX, 1234h
▶ PUSH  AX          ; no POP!
4 RET               ; IP ← ?
```



SS:offset · low address at top

AX is pushed but never popped.

Step 3 / 4

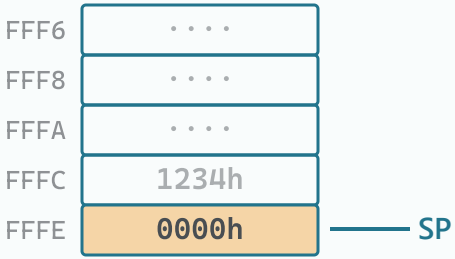
VARIABLES

AX = 1234h SP = FFFCh IP = 0104h

Fix: add POP AX before RET, or don't push at all. Count the PUSHes and POPs of every branch (JMP/Jcc) separately.

Bug Example: Unbalanced PUSH

```
1 ORG   100h
2 MOV   AX, 1234h
3 PUSH  AX          ; no POP!
▶ RET               ; IP ← ?
```



SS:offset · low address at top

`RET` takes the top word as the return address: IP = **1234h**. The CPU starts executing random bytes at 0700:1234h → the program crashes or hangs.

Step 4 / 4

VARIABLES

`AX = 1234h` `SP = FFFEh` `IP = 1234h`

Fix: add POP AX before RET, or don't push at all. Count the PUSHes and POPs of every branch (JMP/Jcc) separately.

STACK SAFETY

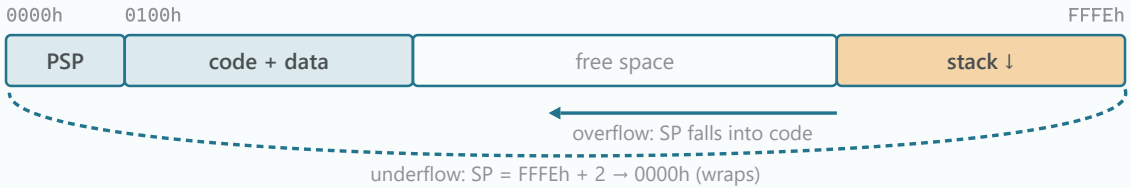
Stack Overflow and Underflow

Overflow

- Runaway recursion, or a PUSH without a POP in a loop.
- As SP falls, the stack moves down and overwrites **code and data** (same segment in .COM).
- The 8086 does not check for this; there is no error, only corruption.

Underflow

- Popping a value that was never pushed.
- POP with SP = FFFEh → SP = **0000h** (wraps around in 16 bits); the next PUSH writes to FFFEh.
- The value read is garbage (or DOS's 0000h).



Rule: size the stack for the deepest call chain plus the most PUSHes. .EXE programs reserve a separate, large enough stack segment with a directive such as `.STACK 100h`.

Good Practice

1. Mirror order

PUSH AX, BX, CX → POP CX, BX, AX. The code reads like a mirror image.

2. Single exit

One exit label instead of several RETs: the POPs are written once.

3. Leave SP alone

Change SP directly only to drop parameters (ADD SP, n) or reserve locals (SUB SP, n).

4. BP frame

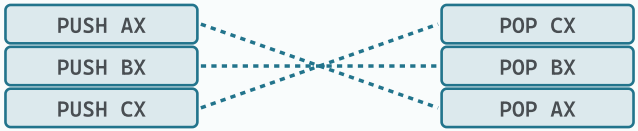
No [SP+n] addressing (SP is not a base register on the 8086); use `MOV BP, SP`.

5. Preserve segments

Routines that change DS/ES should PUSH/POP them too.

6. Check in emu8086

The *stack* button opens the stack window; before RET, check SP is back to its entry value.



mirror image

STACK SAFETY

Tracing the Stack in emu8086

1. Compile and load

compile → emulate. The register panel shows the initial CS, IP, SS, SP (.COM: 0700h, 0100h, 0700h, FFFEh).

2. Single step

Each press of *single step* runs one instruction; changed registers are highlighted. *step back* undoes it.

3. Stack window

The *stack* button at the bottom shows the words around SS:SP; the line SP points to is marked.

4. Memory view

The middle list shows bytes (hex, decimal, ASCII): `B8 12 12` = `MOV AX, 1212h`. Little-endian order is visible here.

After loading, CX holds the program size in bytes (e.g. **000Bh** = 11 bytes in the swap example). This is a convention inherited from DOS DEBUG; programs should not rely on CX.

Lab: The Stack

PUSH AX / POP BX

CODE · 6 bytes

```
1 ORG   100h
2 MOV   AX, 42      ; AX = 42
3 PUSH  AX          ; push AX
4 POP   BX          ; pop in…
5 RET               ; return…
6 END
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	0006	CH	00	CL	06
DX	0000	DH	00	DL	00
SI	0000	DI	0000	BP	0000
SP	FFFE	CS	0700	IP	0100
DS	0700	ES	0700	SS	0700

FLAGS

O	D	I	T	S	Z	A	P	C
0	0	1	0	0	0	0	0	0

CS:IP = 0700:0100 · step 0

MEMORY DS = 0700h

```
0100 B8 2A 00 50 5B C3 00 00
0108 00 00 00 00 00 00 00 00
0110 00 00 00 00 00 00 00 00
0118 00 00 00 00 00 00 00 00
0120 00 00 00 00 00 00 00 00
```

STACK SS:SP

```
FFF6 0000
FFF8 0000
FFFA 0000
FFFC 0000
FFFE 0000 ◀ SP
```

OUTPUT

Program loaded: 6 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (0006h). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary

Concept	Key point
Address	Top of stack = SS:SP; physical address SS × 16 + SP.
Growth	Downward: PUSH SP −= 2, POP SP += 2. The unit is always a word.
PUSH / POP	REG, SREG, memory (no POP CS). PUSH immediate only on 80186+ / emu8086.
LIFO	Values come back in reverse → mirror order to save, same order to swap.
PUSHF / POPF	Saves and restores the FLAGS register; PUSH and POP leave the flags alone.
Safety	Equal numbers of PUSH/POP; in .COM the final RET pops DOS's 0000h.

SUMMARY

Exercises

1. With SS = 2000h and SP = 0100h, what are the physical addresses of the bytes written by `PUSH AX`?
2. What are AX, BX, CX after `MOV AX,1 / MOV BX,2 / MOV CX,3 / PUSH AX / PUSH BX / PUSH CX / POP BX / POP CX / POP AX`?
3. Copy DS into ES using only PUSH/POP.
4. With SP = FFFEh, `PUSH AX / PUSH BX / PUSHF / POP CX` runs. What is SP, and what ends up in CX?
5. Write a PUSH/POP sequence that preserves AX and the flags across an operation.

▶ **Answers**

References

- Intel, *The 8086 Family User's Manual*, 1979 — definitions of PUSH, POP, CALL, RET, INT, IRET.
- B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — Chapters 4 and 6.
- K. R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014 — stack operations and procedures.
- emu8086 help: *8086 Assembler Tutorial, Part 9 (The Stack)*.

The End

Chapter 8.1: The Stack

Previous: Chapter 7: Flow Control · Next: Chapter 8.2: Procedures