◆ Flags Register / Program Status Word

- Holds information about the **state of the processor** (e.g., zero, carry, sign).
- Used to make **decisions** in branching.
- Example: After `SUB`, the **Zero Flag (ZF)** tells if result is zero.

◆ Addressing Modes

- Methods to **access data in memory**.
- Example: `MOV AX, [1234H]` is **direct addressing**.

◆ Program Counter / Instruction Pointer (IP)

- Stores the **address of the next instruction** to execute.
- It increases after each instruction is fetched.

◆ What is a Program

- A **sequence of instructions** that performs a task.
- Written in assembly or high-level language.

◆ Mnemonic

- A **short name** used to represent an instruction.
- Example: `MOV`, `ADD`, `JMP`.

◆ Instruction Groups

1. **Data Movement** – `MOV`, `XCHG`
2. **Arithmetic/Logic** – `ADD`, `AND`
3. **Program Control** – `JMP`, `CALL`

◆ Data Movement

- Moves data between registers or memory.
- Example: `MOV AX, BX`

◆ Arithmetic and Logic Instructions

- Used to **add, subtract**, or perform **bitwise operations**.
- Example: `ADD AX, 5`, `AND AX, BX`

◆ Program Control Instructions

- Used for **branching, looping, calling**.
- Example: `JMP`, `CALL`, `RET`

◆ Special Instructions

- Used for **stack, interrupts, flags**.
- Example: `PUSH`, `POP`, `INT`, `CLC`

◆ Intel IAPX88 Architecture

- 16-bit microprocessor architecture (8088/8086).
- Has **segmented memory, registers, and instruction set**.

◆ Intel Processors – History

- From 4004 → 8086 → 80286 → 80386 → Core i-series.
- Evolved in **power, speed, and instruction set**.

◆ iAPX88 Registers (16-bit)

- Includes **AX, BX, CX, DX** (general) and **SP, BP, SI, DI**.
- Also **segment registers**: CS, DS, ES, SS.

◆ General Instruction Format

- Format: **[opcode] [operand1], [operand2]**
- Example: `MOV AX, 1234H`

◆ Assembler

- Converts assembly code → **machine code**.
- Output: `.OBJ` or `.EXE`

◆ Linker

- Joins multiple object files into **one executable**.
- Resolves **external references**.

◆ Debugger

- Tool to **test and trace** program execution step by step.
- Example: `Turbo Debugger`

◆ AFD (Advanced Full-Screen Debugger)

- A DOS-based tool for **debugging assembly programs**.
- Shows registers, flags, code, memory in full screen.

◆ EX01.ASM – Elaboration

- A sample program (example 1) used to demonstrate **basic instructions** and debugging.

◆ Segmented Memory Model

- Memory divided into **segments**: Code, Data, Stack, Extra.
- Accessed using **segment + offset**.

◆ Physical Address Calculation

- Formula: `Physical = Segment × 10H + Offset`
- Example: `2000:0010 = 20000 + 10 = 20010H`

◆ Paragraph Boundaries

- Every paragraph = **16 bytes**.
- Segment addresses are **aligned to paragraph boundaries**.

◆ Overlapping Segments

- Segments can **share memory areas** if segment values overlap.

◆ Addressing Modes

- How operands are accessed (e.g., immediate, direct, register indirect).

◆ DIRECT ADDRESSING

- Operand is at a **specific memory address**.
- Example: `MOV AX, [1234H]`

◆ Illegal Addressing

- Using unsupported combinations like two memory operands together.
- Example: `MOV [BX], [SI]` ✕

◆ Branching

- Changing the flow of program using **conditions**.
- Example: `JZ`, `JNZ`, `JMP`

◆ Conditional Jumps

- Jump if a **flag condition is true**.
- Example: `JZ`, `JC`, `JNE`

◆ Renamed Conditional Jumps

- Same jumps with **different mnemonics**.
- Example: `JE` = `JZ`, `JNE` = `JNZ`

◆ Unconditional Jumps

- Jump regardless of any condition.
- Example: `JMP label`

◆ Relative Addressing

- Jump to address **relative** to current instruction pointer.
- Used in **short/near jumps**.

◆ Types of Jump

- **Short Jump**: ±127 bytes
- **Near Jump**: same segment

- **Far Jump**: different segment

◆ Bit Manipulation / Multiplication

- Changing individual bits using logic/shifts.
- Example: `AND`, `OR`, `SHL`, `MUL`

◆ Shifting and Rotations

- Move bits left/right, optionally through carry.

◆ SHR (Shift Logical Right)

- Shifts bits to the right, inserts **0** from left.
- Example: `SHR AL, 1`

◆ SAL / SHL (Shift Arithmetic Left)

- Same as logical left shift (fills **0s** at LSB).
- Example: `SAL AL, 1`

◆ SAR (Shift Arithmetic Right)

- Shifts right and **preserves sign bit**.
- Used for signed division.

◆ Shifting

- Moves bits left/right without rotation.
- Can be used for **multiplication/division by 2^n**.

◆ ROR (Rotate Right)

- Rotates bits right, **LSB → MSB**.
- Example: `ROR AL, 1`

◆ ROL (Rotate Left)

- Rotates bits left, **MSB → LSB**.
- Example: `ROL AL, 1`

◆ RCR (Rotate Through Carry Right)

- Rotate bits right **through carry flag**.

◆ RCL (Rotate Through Carry Left)

- Rotate bits left **through carry flag**.

◆ Multiplication in Assembly

- Use `MUL` (unsigned) or `IMUL` (signed).
- Result stored in **AX, or DX:AX**

◆ Extended Shift (Right and Left)

- For multi-word data, combine shift with carry and another register.

◆ Extended Addition

- Add values across multiple words using `ADC` (add with carry).

◆ Extended Subtraction

- Subtract across multiple words using `SBB` (subtract with borrow).

◆ 1. Illegal Instruction

- Memory-to-memory move is not allowed (e.g., `mov [num1], [num2]`).

◆ 2. Memory Access Size

- Sizes of operands must match.
- Instruction like `mov [num1], 5` is ambiguous.
- Assembler cannot determine operand size if no register is used.
- Labels (e.g., `num1`) are just addresses and have no associated size.

◆ 3. Resolving Ambiguity

- Use size specifiers to avoid ambiguity:
  - `mov byte [num1], 5`
  - `mov word [num1], 5`

◆ 4. Memory Access Forms

- **16-bit Move**: `mov ax, [num1]`
- **8-bit Move**: `mov al, [num1]`
- **Illegal Move**:
  - `mov ax, bl` (size mismatch)
  - `mov [num1], [num2]` (memory to memory not allowed)
- **Ambiguous Move**:
  - `mov [num1], 5` (ambiguous without size specifier)

◆ 5. Register Indirect Addressing

- Accessing data placed in **consecutive memory cells** using a register.
- Helps avoid writing multiple instructions for accessing large data sets.
- Register holds the address of memory and can be modified to access the next data.

◆ 6. Registers Used for Addressing

- Four main registers in **iAPX88 architecture** for indirect addressing:
  - `BX` (Base Register)
  - `BP` (Base Pointer)
  - `SI` (Source Index)
  - `DI` (Destination Index)
- Minor differences exist between them (discussed later).

◆ 7. Example Program: Adding 3 Numbers (Program 7)

- Demonstrates register indirect addressing using `BX`.
- Uses `mov bx, num1` and `mov ax, [bx]` to access memory.
- Address changes using `add bx, 2` to move to the next word (since each word = 2 bytes).
- Result stored back in memory.

◆ 8. Explanation of Indirect Addressing

- Square brackets `[]` indicate memory access via register.
- Without brackets, the instruction would behave differently.
- Addressing is **under full program control**, allowing forward/backward navigation in memory.

◆ 9. Loop Control with Indirect Addressing

- Need to repeat operations (e.g., add 100 numbers).
- Requires:
  - **Address modification** (via `add bx, 2`)
  - **Loop control** (via `cx` counter and `jnz` instruction)

◆ 10. Loop Structure

- Example loop for adding 10 numbers:

```
asm
CopyEdit
mov bx, num1
mov cx, 10
l1:
    add ax, [bx]
    add bx, 2
    sub cx, 1
    jnz l1
```

- Uses label `l1` and jump based on Zero Flag (ZF).
- Final result stored using `mov [total], ax`.

◆ 11. Program 8: Adding 10 Numbers

- Full example that:
  - Initializes BX with starting address
  - Uses CX as counter
  - Uses AX as accumulator
  - Writes result to `total`

◆ 12. Explanation of Loop Instructions

- `SUB`: Subtracts 1 from CX, affects ZF
- `JNZ`: Jumps to label if Zero Flag is not set (ZF = 0)
- **Conditional jumps** guide program flow.
- **ZF** is set only by arithmetic/logical operations.

◆ 13. Importance of Conditional Jump

- Provides **decision-making capability** to the processor.
- Equivalent to "if...else" logic.
- Known as **"jump"** in Intel, but also called **"branch"** in other architectures.

◆ 14. Based vs Indexed Addressing

- **Based Addressing**: Using `BX` or `BP`
- **Indexed Addressing**: Using `SI` or `DI`
- Functionality is the same, but terminology differs.

◆ 15. Addressing Modes Overview

- Memory access in iAPX88 uses various addressing modes.
- Memory operands are accessed using one of **seven addressing modes**.
- General form:
  `[base + index + offset]`
  One, two, or all three components may be present.

◆ 16. Jump If Not Zero (JNZ)

- **`jnz`** checks the **Zero Flag (ZF)**.
- Jumps if the last logical or arithmetic operation did **not** result in zero.

◆ 17. Register + Offset Addressing

- A register holds an **index**, not a full address.
- Combined with a constant offset to access array elements dynamically.
- Example: `add ax, [num1 + bx]`

◆ 18. Effective Address (EA) Concept

- The **Effective Address** is the result of adding base, index, and offset.
- EA alone is not enough to access memory — a **segment** is required.

◆ 19. Segment Association

- Memory access needs **segment + offset**.
- Default segment register depends on the register used:

    - `BX`, `SI`, `DI` → **DS**
    - `BP` → **SS**
- **Instruction Pointer (IP)** tied to **CS**, **Stack Pointer (SP)** to **SS**

◆ 20. Segment Override Prefix

- Used to override default segment for one instruction.
- Examples:
    - `mov ax, [cs:bx]` → Override to **CS**
    - `mov ax, [ss:bx]` → Override to **SS**

◆ 21. COM File Memory Model

- In COM files (DOS):
    - **Code, Data, and Stack segments overlap**
    - All segment registers have the same value
    - Simplifies segment handling

◆ 22. Address Wraparound

- **Segment Wraparound**:
    - EA exceeds 16 bits → carry is **dropped**
    - Wraps around to start of the segment
- **Physical Memory Wraparound**:
    - Physical address exceeds 20 bits → wraps to beginning of memory

---

◆ 23. Addressing Modes Summary (7 Modes)

Each one uses `[base + index + offset]` in different combinations:

➤ *1. Offset (Direct) Addressing*

- Example: `mov ax, [1234]`

➤ *2. Register Indirect*

- Uses a register as a pointer
  Example: `mov ax, [bx]`

➤ *3. Register Indirect + Offset*

- Example: `mov ax, [bx + 0100]`

➤ *4. Indexed Addressing*

- Uses index registers (`SI`, `DI`)
  Example: `mov ax, [si]`

➤ *5. Indexed + Offset*

- Example: `mov ax, [si + 300]`

➤ *6. Base + Index*

- Combines base and index registers
  Example: `mov ax, [bx + si]`

➤ *7. Base + Index + Offset*

- Combines all three: base, index, and constant
  Example: `mov ax, [bx + si + 300]`

◆ 24. Illegal Addressing Examples

- **Invalid operand sizes**:
  - `mov al, [bl]` → BL is 8-bit (illegal for address)
- **Invalid register combinations**:
  - `mov ax, [bx - si]` → subtraction not allowed
  - `mov ax, [bx + bp]` → two base registers not allowed
  - `mov ax, [si + di]` → two index registers not allowed

◆ 25. Physical Address Calculation

- Formula:
  Physical Address = (Segment × 16) + Effective Address
- Example:
  - `CS = 0x1000`, `BX = 0x0100`, `SI = 0x0200`, `Offset = 0x0700`
    → Physical Address = `0x10A00`

🔼 26. Bubble Sort Algorithm Basics

- Simple sorting algorithm using repeated comparisons and swaps.
- In each **pass**, adjacent numbers are compared.
- If not in required order, **they are swapped**.
- Repeated until a pass completes **without swaps**.

🔼 27. Ascending vs. Descending Sort

- **Ascending**: smallest to largest.
- **Descending**: largest to smallest.
- Bubble sort logic is adjusted based on the direction of comparison.

🔼 28. Bubble Sort Working Example

- Initial data: `60, 55, 45, 58`
- Through multiple passes, the data becomes sorted:
  - **Pass 1 result**: `55, 45, 58, 60`
  - **Pass 2 result**: `45, 55, 58, 60`
  - **Pass 3 result**: `45, 55, 58, 60` (no swap → done)

🔼 29. Program 12: Bubble Sort in Assembly

- Sorts a list of 10 numbers using bubble sort.
- Uses a **swap flag** in memory to detect if another pass is needed.

🔼 30. Explanation of Bubble Sort Assembly Program

- `swap` flag is reset and checked after each pass.

- Compares `[data + bx]` and `[data + bx + 2]`
- Swapping uses **AX and DX**.
- **JBE** ensures no swap on equal elements, avoiding infinite loops.

🔽 31. Bubble Sort Index Boundaries

- Only compares up to the 9th element (`bx = 18`) to avoid accessing non-existent 11th element.

🔽 32. Effect of Changing JBE to JB or JAE

- `JBE`: allows equal elements to stay in place.
- `JB`: causes unnecessary swaps on equal values.
- `JAE`: changes sort to **descending** order.

🔽 33. Handling Signed Numbers in Sorting

- Sorting logic changes depending on signed/unsigned interpretation.
- Example:

```
assembly
CopyEdit
data: dw 60, 55, 45, 50, -40, -35, 25, 30, 10, 0
```

🔽 34. Impact of JBE vs. JLE in Sorting

- `JBE`: treats numbers as **unsigned**.
  - Negative numbers appear **last**.
- `JLE`: treats numbers as **signed**.
  - Negative numbers appear **first**.

🔽 35. Two's Complement Representation

- Example:
  - `-2` ⟶ `0xFFFE`
  - `2` ⟶ `0x0002`
- Same hex value can represent different numbers based on signed/unsigned interpretation.

🔻 36. Signed vs. Unsigned Comparisons in Assembly

- Example:

```
assembly
CopyEdit
cmp ax, bx
ja label1 ; checks if AX > BX (unsigned)
```

🔻 37. Assembly Comparison Instruction Choice

- Signed jumps: `JL`, `JG`, `JLE`, `JGE`
- Unsigned jumps: `JB`, `JA`, `JBE`, `JAE`
- Correct jump selection ensures **expected sorting behavior**.

🔻 38. Interpretation Responsibility in Assembly

- Unlike high-level languages, **programmer decides** how to interpret data.
- Compiler does this automatically in C/C++ or Java.

🔻 39. Order and Range of Numbers (16-bit)

- **Unsigned**:
  `0 < 1 < 2 < ... < 65535`
- **Signed**:
  `-32768 < -32767 < ... < 0 < ... < 32767`

🔻 40. Extended Multiplication in Assembly

- Performs multiplication of **16-bit multiplier** with a **32-bit multiplicand**.
- Uses **bit-by-bit** processing of the multiplier (16 bits).
- The **multiplicand and result are 32 bits** to prevent overflow on shifts and additions.

🔻 41. Need for 32-bit Storage

- Multiplicand is 32-bit to preserve significant bits during **left shifts**.
- Result also 32-bit to store large values from accumulation.

🔺 42. Extended Addition and Shifting

- `add` and `adc` used to perform **extended 32-bit addition**.
- `shl` and `rcl` used for **extended left shift** of the multiplicand (low and high words).

🔺 43. Program 12: Extended Multiplication (using SHR)

- Loads multiplier in `DX` and checks its bits using `shr`.
- Shifts multiplicand left (32-bit) and adds to result if bit is `1`.
- Repeats for **16 bits** of multiplier.

🔺 44. Memory Map and Debugging

- Multiplicand → `0103–0106`
- Multiplier → `0107–0108`
- Result → `0109–010C`
- Debugger helps track **bitwise changes** in memory.

🔺 45. Final Result in Memory

- Result after full multiplication: `0009EB10` → Decimal **65000**
- Shows correct 32-bit multiplication using extended operations.

🔺 46. Logical Operations in Assembly (8088)

- **Bitwise operations**: `AND`, `OR`, `XOR`, `NOT`
- Affect **individual bits** of a register or memory location.

🔺 47. AND Operation

- Bit is set **only if both bits are 1**.
- Used for **clearing bits** (masking).
- Flags affected: **C O P S Z A**

🔻 48. OR Operation

- Bit is set if **either bit is 1**.
- Used for **setting bits** selectively.
- Flags affected: **C O P S Z A**

🔻 49. XOR Operation

- Bit is set if **bits are opposite**.
- Used for **bit inversion** selectively.
- Flags affected: **C O P S Z A**

🔻 50. NOT Operation

- **Inverts** every bit (1's complement).
- **Non-destructive** and affects **no flags**.

🔻 51. Masking: Selective Bit Clearing (AND)

- Use mask with `1`s at **preserve** positions and `0`s at **clear** positions.
- Example: `and al, 0x0F` → clears upper nibble.

🔻 52. Masking: Selective Bit Setting (OR)

- Use mask with `1`s at **set** positions and `0`s elsewhere.
- Example: `or al, 0xF0` → sets upper nibble.

🔻 53. Masking: Selective Bit Inversion (XOR)

- Use mask with `1`s at **invert** positions and `0`s to preserve.
- Example: `xor al, 0xF0` → inverts upper nibble.

🔻 54. Selective Bit Testing using AND

- Can test individual bits using `and` with a mask.
- Requires `JZ` to check result (zero or non-zero).
- **Destructive**: changes destination.

🔻 55. Selective Bit Testing using TEST Instruction

- `test` is **non-destructive AND**.
- Only **sets flags**, does not change operands.
- Ideal for checking bits without altering data.

🔻 56. Program 13: Extended Multiplication Using TEST

- Replaces `shr` with `test` for checking each bit.
- Uses a **mask in BX**, shifts it left each time to check next bit.
- More elegant and **non-destructive** bit-checking.

🔻 57. Optimization Tip: Ending Loop Without Bit Counter

- Instead of using a counter (CL), can **stop when mask becomes zero**.
- Saves instructions and enhances performance.

🔻 58. Final Output of Program 13

- Same result: `0009EB10` (65000 in decimal).
- Demonstrates correctness of TEST-based multiplication.

🔻 59. Reusability through Subroutines

- Sorting and multiplication are commonly reused in programs.
- Copying code 100 times is **inefficient and unmaintainable**.
- Subroutines allow writing once and calling multiple times.

🔻 60. Problem with JMP for Reuse

- `JMP` causes **permanent diversion** with no return address.
- Can't return to the original location from which it jumped.

🔻 61. Temporary Diversion Analogy

- Jump is like a **highway turn** (permanent).
- Subroutine is like a **roundabout** (returns you to the original place).
- Needed: a **temporary control transfer** mechanism.

🔻 62. CALL and RET Instructions

- `CALL label`: Jumps to subroutine and stores **return address** on the stack.
- `RET`: Returns control to the instruction **after the CALL**.
- Together, they create **temporary flow diversion**.

🔻 63. CALL and RET Independence

- Technically **independent**: `CALL` works without `RET`, and vice versa.
- But **logically used as a pair** in well-structured code.

🔻 64. Parameters in Subroutines

- Different inputs needed each time (e.g., array address, size, sort order).
- Passed via **registers** (e.g., `BX`, `CX`) to the subroutine.

🔻 65. Program 14 – Bubble Sort as Subroutine

- `bubblesort` is labeled subroutine.
- Parameters passed:
  - `BX` → start address of array
  - `CX` → number of elements
- `CALL bubblesort` is followed by `RET` at the end.

🔻 66. Program 14: Addressing and Shifting

- Uses **base + index + offset** addressing (`[BX+SI+2]`).
- `CX` is **decremented and left-shifted** to convert count into byte count.

🔻 67. Debugger Observation of CALL/RET

- `CALL` pushes **return address** onto the stack.
- `RET` pops it and resumes execution at the correct instruction.
- `SP` decrements by 2 on `CALL`, restores on `RET`.

🔻 68. CALL and the Stack

- Stack is **automatically** used to store the return address.
- E.g., CALL at address `014E` → stack stores `0150`, returns after subroutine.

🔻 69. Program 15 – Reusing the Subroutine

- Calls `bubblesort` twice:
  1. On `data` (10 elements)
  2. On `data2` (20 elements)
- Demonstrates **code reusability** without rewriting the sort logic.

🔻 70. Continuous Data Declaration

- `data2` is declared across **two lines**; still treated as a single array.
- No additional label needed due to **contiguous memory placement**.

🔻 71. Multiple Subroutine Calls and Stack Use

- Each `CALL` places the **address of next instruction** on the stack.
- Stack pointer (`SP`) is **decremented and restored** accordingly.

🔻 72. Shortcoming: Register Overwriting

- Subroutine **modifies registers** (`AX`, `CX`, `DX`, `SI`).
- Caller loses any **important data** stored in these registers.
- This becomes unmanageable in large programs.

🔻 73. Need for System Stack for Register Safety

- Future solution: Use the **system stack** to save/restore registers.
- Protects caller's data and makes subroutine usage **safe and reliable**.

🔻 74. Stack – First In Last Out (FILO) Structure

- Stack allows only **one entry and exit point**.
- Last element pushed is the first one popped.
- Visualized as a **test tube with balls**—only the top one can be removed.

🔻 75. Stack Operations – PUSH and POP

- **PUSH** places data on top of the stack (SP is **decremented by 2**).
- **POP** removes the top data (SP is **incremented by 2**).
- Only **word-sized data** (2 bytes) is supported on 8088 stack.

🔻 76. Role of SP and SS Registers

- **SP (Stack Pointer)**: Holds offset to top of the stack.
- **SS (Stack Segment)**: Points to segment of the stack.
- Full address = `SS:SP`.

🔻 77. Decrementing Stack in 8088

- Stack grows **downward** (from higher memory addresses to lower).
- Example: Starts at `2000`, then goes `1FFE`, `1FFC`, etc.

🔻 78. Pushing and Popping Logic

- **PUSH**:
  - `SP ← SP - 2`
  - `[SP] ← Operand`
- **POP**:
  - `Operand ← [SP]`
  - `SP ← SP + 2`

🔻 79. Saving and Restoring Registers using Stack

- Subroutines should **save caller registers** on stack using PUSH.
- Restore them using POP before returning to **prevent data corruption**.

🔻 80. CALL Instruction and the Stack

- `CALL` **pushes the return address (IP)** on the stack.
- RET **pops it back into IP**, resuming execution after the CALL.
- Enables **temporary flow control** like subroutines.

🔻 81. RET Instruction Variants

- `RET`: Pops IP from stack (for **intrasegment** return).
- `RET n`: Pops IP and then **increments SP by n** (used for parameter cleanup).
- `RETF`: For **intersegment** returns, pops both IP and CS.

🔻 82. Example of Stack Usage in CALL/RET

- `SP` initialized at `2000`.
- `CALL` stores return address (e.g., `017B`) → `SP = 1FFE`, value pushed.
- `RET` restores IP and `SP = 2000` again.

🔻 83. Far CALL and RET

- `CALL far`: Pushes **IP and CS**, loads new segment/offset.
- `RETF`: Pops IP, then CS to return control to the calling segment.

🔻 84. PUSH/POP Responsibility of Programmer

- Order of `PUSH` and `POP` must be **reverse** to correctly restore values.
- E.g., `PUSH AX`, `PUSH BX`, then `POP BX`, `POP AX`.

🔻 85. Subroutine Calls with Multiple Layers

- Program 16 uses **two subroutines**:
  - `bubblesort`: Sort logic
  - `swap`: Swaps two elements using `XCHG` and `RET`
- Both are reusable with CALL/RET.

🔻 86. XCHG Instruction for Efficient Swapping

- `XCHG` swaps AX with memory operand (`[bx+si+2]`).
- Shortens code by avoiding use of extra registers.

🔻 87. Flags and Stack Instructions

- **PUSH/POP/CALL/RET** do **not affect flags**.
- `INT` and `IRET` (used for interrupts) also involve stack operations, discussed later.

🔻 88. Stack Usage Summary

Operation	Stack Effect	Description
`PUSH AX`	`SP -= 2` → `[SP] = AX`	Save AX
`POP AX`	`AX = [SP]` → `SP += 2`	Restore AX

Operation	Stack Effect	Description
`CALL`	`SP -= 2, [SP] = IP`	Save return address
`RET`	`IP = [SP], SP += 2`	Resume execution
`CALL far`	Push IP and CS	Jump to another segment
`RETF`	Pop IP, then CS	Return from far call

🔻 89. Problem with Subroutines Destroying Registers

- Subroutines (like `swap`, `bubblesort`) overwrite AX, CX, SI, etc.
- It's hard for the caller to track what registers each subroutine modifies.
- Solution: Use **PUSH** to save and **POP** to restore registers before returning.

🔻 90. Standard Practice: Save Modified Registers in Subroutines

- Use `PUSH` to save used registers at subroutine **entry**.
- Use `POP` to restore them in **reverse order** before `RET`.
- Stack follows **Last In First Out (LIFO)** — order matters.

🔻 91. Program 17 – Safe Subroutines with PUSH and POP

- `swap` subroutine saves and restores **AX**.
- `bubblesort` saves and restores **AX, CX, SI**.
- Ensures that after `CALL`, all caller registers retain original values.

🔻 92. PUSH and POP Recap

Instruction	Operation	Effect on Stack Pointer
`PUSH reg`	Save register value to stack	`SP -= 2`
`POP reg`	Restore from stack to register	`SP += 2`

🔻 93. CALL and RET Behavior

- `CALL`: Pushes **return address** (IP) onto the stack.

- `RET`: Pops it back, **resuming execution**.
- `RETF`: Pops IP and CS (for far returns).

🔻 94. Limitation of Passing Parameters Through Registers

- Max 7 parameters due to limited general-purpose registers.
- Registers get reused in nested subroutine calls.
- **Not scalable or maintainable** for large programs.

🔻 95. Stack as an Alternative for Parameter Passing

- Parameters pushed **before** calling subroutine.
- Stack retains parameters until explicitly removed.
- Parameters live **below** the return address on the stack.

🔻 96. Problem Accessing Parameters on Stack

- `POP` cannot be used directly since **return address is on top** of stack.
- Popping parameters would destroy return flow.
- Need a better way to **peek without removing** stack elements.

🔻 97. Using Base Pointer (BP) to Access Parameters

- **BP defaults to stack segment** like SP.
- Copy `SP → BP`, then use `[BP+offset]` to access parameters.
- This **freezes** the current stack snapshot.

🔻 98. Stack Layout Using BP

BP Offset	Content
`[BP]`	Old BP value (pushed first)
`[BP+2]`	Return address
`[BP+4]`	2nd parameter
`[BP+6]`	1st parameter

🔻 99. Program 18 – Bubble Sort Using Stack Parameters

- Subroutine **saves BP**, then uses `mov bp, sp` to reference stack.
- Accesses array pointer and count using `[bp+6]` and `[bp+4]`.
- Saves/restores **AX, BX, CX, SI, BP** for safety.

🔻 100. `RET n` Instruction

- `RET 4`: Pops return address (2 bytes) + discards **2 parameters (4 bytes)**.
- Efficient cleanup — no separate `POP` needed for parameters.
- Stack returns to state **before parameters were pushed**.

🔻 101. Program 18 Caller Code Explanation

- `push array_address` → via AX
- `push element_count` → via AX
- `call bubblesort`
- Repeats for second array.

🔻 102. Final Stack Flow Summary

- Caller pushes parameters → `CALL` adds return address → subroutine uses BP to access parameters.
- Subroutine restores registers, then `RET n` clears parameters + return address.

🔻 103. Why Use BP Instead of SP

- SP changes dynamically (due to PUSH/POP).
- SP **can't be used in effective address calculations**.
- BP is **static reference point** once initialized (`mov bp, sp`).

🔻 104. Standard Parameter Access Convention in Subroutines

- Start with:

```
asm
CopyEdit
push bp
mov bp, sp
```

- End with:

```
asm
CopyEdit
pop bp
ret n
```

🔻 105. Debugging Tip

- In debuggers:
  - You can verify parameters via `[bp+4]`, `[bp+6]` etc.
  - After `RET 4`, SP returns to value before parameter PUSH.
