Chapter 3

Branching

Comparisons and Conditions

Conditional branching in assembly language is a crucial concept that allows programs to make decisions based on certain conditions. These conditions are typically determined by the outcome of comparisons between data. Let's explore the basics of comparisons and conditions in assembly language:

Comparison Operation (CMP):

- The fundamental instruction for all comparisons in assembly language is `CMP`, which stands for **"compare"**.
- `CMP` subtracts the source operand from the destination operand without changing either of them.
- The primary purpose of `CMP` is to update the condition flags based on the comparison result.
- The condition flags, including the **sign flag**, **carry flag**, **zero flag**, and **overflow flag**, are updated to reflect the relationship between the destination and source operands.

Conditional JUMPS:

- After performing a comparison using `CMP`, conditional jumps are used to control program flow based on the condition flags.
- Conditional jumps are instructions like: `JG` (jump if greater), `JL` (jump if less), `JA` (jump if above), and `JB` (jump if below).
- The choice of conditional jump depends on whether you want to compare **signed** or **unsigned** numbers.

Signed vs. Unsigned Numbers:

- In assembly language, signed numbers consider both magnitude and sign, while unsigned numbers focus only on magnitude.

- Signed numbers are represented in two's complement notation, where the sign bit indicates positivity or negativity.
- The interpretation of whether numbers should be treated as signed or unsigned is determined during conditional jumps after comparisons.

Example:

Comparison of -2 and 2:

- In memory, **-2** is represented as `FFFE` (in two's complement) which is equal to **65534**, and **2** is represented as `0002`.
- For unsigned comparison, **65534** is greater than **2** (`JA` would be taken).
- For signed comparison, **-2** is less than **2** (`JL` would be taken).
- For signed numbers **JG** and **JL** will work properly and for unsigned **JA** and **JB** will work properly.

Range of Unsigned vs. Signed:

- Unsigned comparisons treat **0** as the smallest and **65535** as the largest.
- Signed comparisons treat **-32768** as the smallest and **32767** as the largest.
- Negative numbers share memory representation with large unsigned numbers, but their interpretation differs.

Comparison Outcomes:

- The outcomes of comparisons **(CMP)** followed by conditional jumps depend on the interpretation of data as signed or unsigned and the specific conditions being checked.
- Whether a jump is taken or not is determined by the state of condition flags (e.g., zero flag, sign flag) after the comparison.

All meaningful situations both for signed and unsigned numbers that occur after a comparison are detailed in the following table:

DEST = SRC	ZF = 1	When the source is subtracted from the destination and both are equal the result is zero and therefore the zero flag is set. This works for both signed and unsigned numbers.
UDEST < USRC	CF = 1	When an unsigned source is subtracted from an unsigned destination and the destination is smaller, borrow is needed which sets the carry flag.
UDEST ≤ USRC	ZF = 1 OR CF = 1	If the zero flag is set, it means that the source and destination are equal and if the carry flag is set it means a borrow was needed in the subtraction and therefore the destination is smaller.
UDEST ≥ USRC	CF = 0	When an unsigned source is subtracted from an unsigned destination no borrow will be needed either when the operands are equal or when the destination is greater than the source.
UDEST > USRC	ZF = 0 AND CF = 0	The unsigned source and destination are not equal if the zero flag is not set and the destination is not smaller since no borrow was taken. Therefore the destination is greater than the source.
SDEST < SSRC	SF ≠ OF	When a signed source is subtracted from a signed destination and the answer is negative with no overflow than the destination is smaller than the source. If however there is an overflow meaning that the sign has changed unexpectedly, the meanings are reversed and a

		positive number signals that the destination is smaller.
SDEST ≤ SSRC	ZF = 1 OR SF ≠ OF	If the zero flag is set, it means that the source and destination are equal and if the sign and overflow flags differ it means that the destination is smaller as described above.
SDEST ≥ SSRC	SF = OF	When a signed source is subtracted from a signed destination and the answer is positive with no overflow than the destination is greater than the source. When an overflow is there signaling that sign has changed unexpectedly, we interpret a negative answer as the signal that the destination is greater.
SDEST > SSRC	ZF = 0 AND SF = OF	If the zero flag is not set, it means that the signed operands are not equal and if the sign and overflow match in addition to this it means that the destination is greater than the source.

Conditional Jumps

Conditional jumps in assembly language are essential for controlling program flow based on specific conditions. These jumps allow programs to make decisions and execute different code paths accordingly. Here, we'll discuss four types of conditional jumps:

1. Jumps Based on Specific Flag Values:

Mnemonic	Description	Flags / Registers
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

2. Jumps Based on Equality Between Operands or CX:

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0

3. Jumps Based on Comparisons of Unsigned Operands:

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* ≥ *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* ≤ *rightOp*)
JNA	Jump if not above (same as JBE)

4. Jumps Based on Comparisons of Signed Operands:

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* ≥ *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* ≤ *rightOp*)
JNG	Jump if not greater (same as JLE)

Example of Conditional Jump:

```
[org 0x0100]
 mov bx, 0               ; initialize array index to zero
 mov ax, 0               ; initialize sum to zero
l1: add ax, [num1+bx]    ; add number to ax
 add bx, 2               ; advance bx to next index
 cmp bx, 20              ; are we beyond the last index
 jne l1                  ; if not add next number
```

```
 mov [total], ax         ; write back sum in memory
 mov ax, 0x4c00          ; terminate program
 int 0x21

num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50
total: dw 0
```

These conditional jumps, along with the CMP instruction that sets the flags based on the relationship between the destination and source operands, allow assembly language programs to execute specific code paths based on various conditions. Choosing the appropriate jump instruction depends on the specific requirements of the program and the interpretation of data as signed or unsigned.

Un-conditional Jumps

An unconditional jump in assembly language is an instruction that causes the program's control flow to transfer to a different location in the code without any condition or test. Unconditional jumps are used to create loops, implement branching, and control the program's execution flow.

Instruction Format:

- In assembly language, unconditional jumps are typically represented by instructions like `JMP` (short for **"jump"**).
- The `JMP` instruction transfers control to a specified label or memory address unconditionally.

```
JMP target_label
```

In this example, the `JMP` instruction unconditionally transfers control to the code located at the `target_label`.

Example:

```
[org 0x0100]
 jmp start              ; unconditionally jump over data
num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50
total: dw 0
```

```
start:
 mov bx, 0              ; initialize array index to zero
 mov ax, 0              ; initialize sum to zero
l1: add ax, [num1+bx]   ; add number to ax
 add bx, 2              ; advance bx to next index
 cmp bx, 20             ; are we beyond the last index
 jne l1                 ; if not add next number
 mov [total], ax        ; write back sum in memory
 mov ax, 0x4c00         ; terminate program
 int 0x21
```

Relative Addressing

Relative addressing in assembly language allows you to specify a jump or branch to a location based on its position relative to the current instruction. It's like giving directions by saying, **"Go forward 10 steps"** rather than providing an exact address.

Here's a simple explanation of relative addressing:

1. **JMP Instruction:** In assembly language, when you use a `JMP` (jump) instruction, you can specify a target location using relative addressing.
2. **Operand Value:** Instead of providing the exact memory address, you provide a value that represents how many steps forward or backward you want to go from your current position.
3. **Example:** Let's say you have a `JMP` instruction like `JMP 0016`. This means "jump 16 steps forward from where you are right now."
4. **Mechanism:** The assembler and processor handle the calculations. When you execute this instruction, the processor adds the specified value (16 in this case) to the current instruction pointer (IP), effectively moving the program counter to the desired target location.
5. **Simplified Usage:** In your code, you use labels with `JMP` instructions, and the assembler takes care of converting those labels into the correct relative values. This makes it easier for programmers to work with addresses without needing to calculate exact addresses manually.

Types of Jumps

In assembly language programming, there are three types of jumps: near, short, and far. These jumps differ in terms of the size of the instruction and the range of memory they can access.

1. Near Jump:

- A near jump is used when the relative address stored with the instruction is in **16 bits**, allowing it to access anywhere within the **current segment**.
- It operates within the same code segment and uses a 16-bit relative address.
- Near jumps can jump anywhere within the current segment. If you add a large number to the relative address, it will wrap around to the lower part.
- Negative numbers are treated as large numbers and behave accordingly using wraparound behavior.

2. Short Jump:

- A short jump uses a single byte (**8-bits**) to store the offset, like `75F2` with the opcode `75` and operand `F2`.
- It is also known as a signed jump because the offset is treated as a signed byte.
- For short jumps, if the byte is negative, the complement is negated from the instruction pointer (IP); otherwise, the byte is added.
- Short jumps can go **+127** bytes ahead in code or **-128** bytes backward and no further. This limitation arises from the use of a signed byte.

3. Far Jump:

- A far jump is absolute and not position relative. It requires both a segment and an offset to be specified.
- Far jumps are used when you need to jump from one code segment to another, and near or short jumps cannot achieve this.
- A far jump instruction is represented by **five byte**s: two bytes for the segment and two bytes for the offset, making a total of five bytes.
- When a far jump is executed, it loads the code segment (CS) register with the segment part and the instruction pointer (IP) register with the offset part, allowing

execution to continue from that location in physical memory.

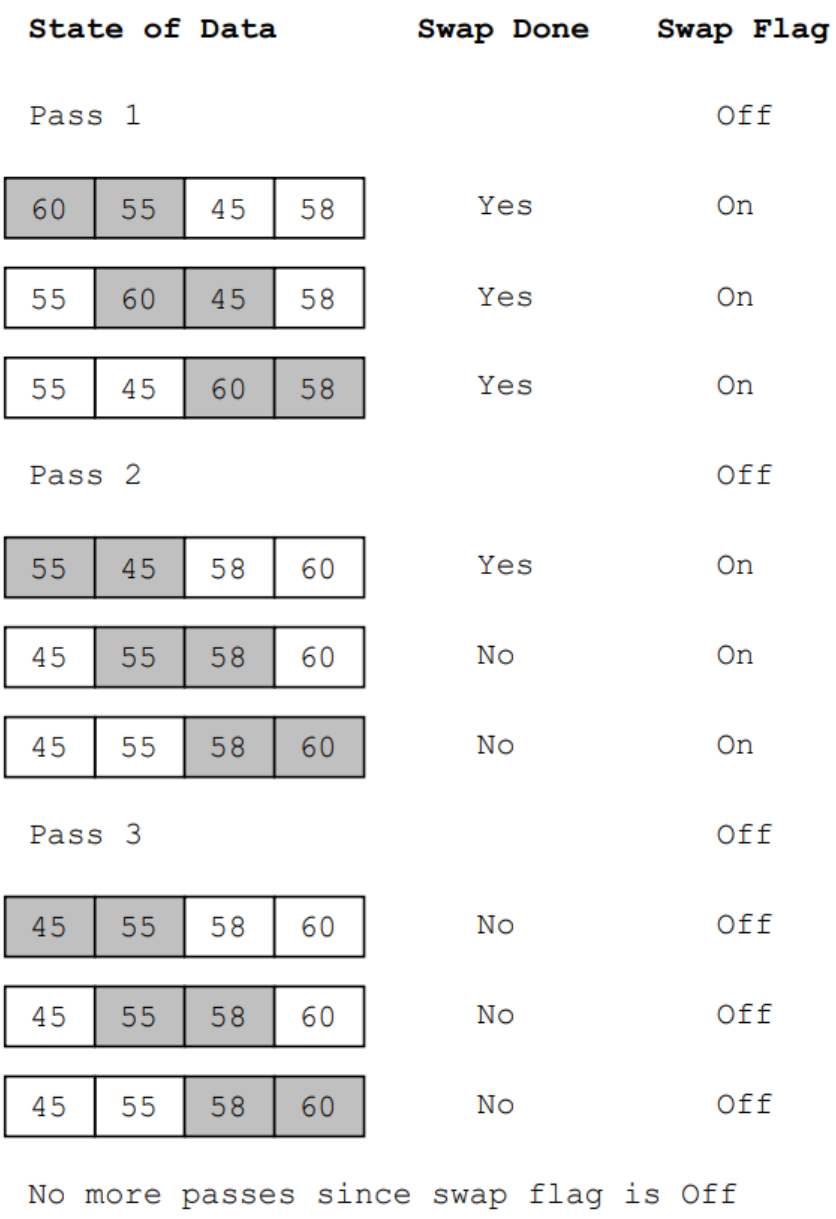
- The far jump capability is essential for achieving control within different code segments.
- Three instructions in assembly language that have a far form are `JMP`, `CALL`, and `RET`, and they are primarily used for program control and inter-segment jumps.

Sorting Example in Assembly

Sorting in assembly language involves arranging a list of numbers in either ascending or descending order. One common sorting algorithm is the bubble sort, which we'll discuss here.

Bubble Sort Algorithm:

- Bubble sort compares consecutive numbers in the list.
- If the numbers are not in the desired order, they are swapped.
- The process repeats for each pair of numbers until no more swaps are needed.
- A complete iteration through the list is called a pass.
- The algorithm continues to perform passes until no swaps occur in a pass.



Example Code:

```
[org 0x0100]
jmp start
data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
swap: db 0

start:
mov bx, 0               ; Initialize array index to zero
```

```
mov byte [swap], 0      ; Reset swap flag to no swaps

loop1:
mov ax, [data+bx]       ; Load number in ax
cmp ax, [data+bx+2]     ; Compare with next number
jbe noswap              ; No swap if already in order

mov dx, [data+bx+2]     ; Load second element in dx
mov [data+bx+2], ax     ; Store first number in the second
mov [data+bx], dx       ; Store second number in the first
mov byte [swap], 1      ; Flag that a swap has been done

noswap:
add bx, 2               ; Advance bx to the next index
cmp bx, 18              ; Are we at the last index?
jne loop1               ; If not, compare the next two

cmp byte [swap], 1      ; Check if a swap has been done
je start                ; If yes, make another pass

mov ax, 0x4c00          ; Terminate program
int 0x21
```

In this code, the bubble sort algorithm is applied to sort a list of numbers stored in the `data` array. The `swap` flag is used to track whether any swaps occurred during a pass. The algorithm continues making passes until no swaps are needed, indicating that the list is sorted.

Additionally, the code demonstrates how the choice of conditional jump instruction (`jbe` in this case) can affect the sorting order. Depending on whether the code uses signed or unsigned interpretations, different sorting results can be achieved. In this example, the code sorts the data in ascending order.

It's important to choose the appropriate conditional jump instruction based on whether you want to sort the data as signed or unsigned numbers, as this can significantly impact the sorting outcome.