Branching

How to interpret the flags (1)?

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.

How to interpret the flags (2)?

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 (1)

unsigned

JC JB JNAE	Jump if carry Jump if below Jump if not above or equal	CF = 1	This jump is taken if the last arithmetic operation generated a carry or required a borrow. After a CMP it is taken if the unsigned destination is smaller than the unsigned source.
JNC JNB JAE	Jump if not carry Jump if not below Jump if above or equal	CF = 0	This jump is taken if the last arithmetic operation did not generated a carry or required a borrow. After a CMP it is taken if the unsigned destination is larger or equal to the unsigned source.
JE JZ	Jump if equal Jump if zero	ZF = 1	This jump is taken if the last arithmetic operation produced a zero in its destination. After a CMP it is taken if both operands were equal.
JNE JNZ	Jump if not equal Jump if not zero	ZF = 0	This jump is taken if the last arithmetic operation did not produce a zero in its destination. After a CMP it is taken if both operands were different.

Conditional Jumps (2)

JC JB JNAE	Jump if carry Jump if below Jump if not above or equal	CF = 1	This jump is taken if the last arithmetic operation generated a carry or required a borrow. After a CMP it is taken if the unsigned destination is smaller than the unsigned source.
JNC JNB JAE	Jump if not carry Jump if not below Jump if above or equal	CF = 0	This jump is taken if the last arithmetic operation did not generated a carry or required a borrow. After a CMP it is taken if the unsigned destination is larger or equal to the unsigned source.
JE JZ	Jump if equal Jump if zero	ZF = 1	This jump is taken if the last arithmetic operation produced a zero in its destination. After a CMP it is taken if both operands were equal.
JNE JNZ	Jump if not equal Jump if not zero	ZF = 0	This jump is taken if the last arithmetic operation did not produce a zero in its destination. After a CMP it is taken if both operands were different.

Conditional Jumps (3)

JA JNBE	Jump if above Jump if not below or equal	ZF = 0 AND CF = 0	This jump is taken after a CMP if the unsigned destination is larger than the unsigned source.
JNA JBE	Jump if not above Jump if below or equal	ZF = 1 OR CF = 1	This jump is taken after a CMP if the unsigned destination is smaller than or equal to the unsigned source.
JL JNGE	Jump if less Jump if not greater or equal	SF ≠ OF	This jump is taken after a CMP if the signed destination is smaller than the signed source.
JNL JGE	Jump if not less Jump if greater or equal	SF = OF	This jump is taken after a CMP if the signed destination is larger than or equal to the signed source.
JG JNLE	Jump if greater Jump if not less or equal	ZF = 0 AND SF = OF	This jump is taken after a CMP if the signed destination is larger than the signed source.
JNG JLE	Jump if not greater Jump if less or equal	ZF = 1 OR SF ≠ OF	This jump is taken after a CMP if the signed destination is smaller than or equal to the signed source.

Signed

Conditional Jumps (4)

JO	Jump if overflow.	OF = 1	This jump is taken if the last arithmetic operation changed the sign unexpectedly.
JNO	Jump if not overflow	OF = 0	This jump is taken if the last arithmetic operation did not change the sign unexpectedly.
JS	Jump if sign	SF = 1	This jump is taken if the last arithmetic operation produced a negative number in its destination.
JNS	Jump if not sign	SF = 0	This jump is taken if the last arithmetic operation produced a positive number in its destination.
JP JPE	Jump if parity Jump if even parity	PF = 1	This jump is taken if the last arithmetic operation produced a number in its destination that has even parity.
JNP JPO	Jump if not parity Jump if odd parity	PF = 0	This jump is taken if the last arithmetic operation produced a number in its destination that has odd parity.
JCXZ	Jump if CX is zero	CX = 0	This jump is taken if the CX register is zero.

Example 3.1.

Example 3.1

```
001 ; a program to add ten numbers without a separate counter
002 [org 0x0100]
003                 mov  bx, 0             ; initialize array index to zero
004                 mov  ax, 0             ; initialize sum to zero
005
006 l1:             add  ax, [num1+bx]     ; add number to ax
007                 add  bx, 2             ; advance bx to next index
008                 cmp  bx, 20            ; are we beyond the last index
009                 jne  l1                ; if not add next number
010
011                 mov  [total], ax       ; write back sum in memory
012
013                 mov  ax, 0x4c00        ; terminate program
014                 int  0x21
015
016 num1:           dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
017 total:          dw   0
```

006	The format of memory access is still base + offset.
008	BX is used as the array index as well as the counter. The offset of 11th number will be 20, so as soon as BX becomes 20 just after the 10th number has been added, the addition is stopped.
009	The jump is displayed as JNZ in the debugger even though we have written JNE in our example. This is because it is a renamed jump with the same opcode as JNZ and the debugger has no way of knowing the mnemonic that we used after looking just at the opcode. Also every code and data reference that we used till now is seen in the opcode as well. However for the jump instruction we see an operand of F2 in the opcode and not 0116. This will be discussed in detail with unconditional jumps. It is actually a short relative jump and the operand is stored in the form of positive or negative offset from this instruction.

Example 3.2.: Unconditional Jump

Example 3.2

```
001  ; a program to add ten numbers without a separate counter
002  [org 0x0100]
003                jmp start                   ; unconditionally jump over data
004
005  num1:         dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
006  total:        dw   0
007
008  start:        mov  bx, 0                  ; initialize array index to zero
009                mov  ax, 0                  ; initialize sum to zero
010
011  l1:           add  ax, [num1+bx]          ; add number to ax
012                add  bx, 2                  ; advance bx to next index
013                cmp  bx, 20                 ; are we beyond the last index
014                jne  l1                     ; if not add next number
015
016                mov  [total], ax            ; write back sum in memory
017
018                mov  ax, 0x4c00             ; terminate program
019                int  0x21
```

003	JMP jumps over the data declarations to the start label and execution resumes from there.

Types of JUMP (1)

3.5. TYPES OF JUMP

The three types of jump, near, short, and far, differ in the size of instruction and the range of memory they can jump to with the smallest short form of two bytes and a range of just 256 bytes to the far form of five bytes and a range covering the whole memory.

Short Jump

EB	Disp

Near Jump

E9	Disp Low	Disp High

Far Jump

EA	IP Low	IP High	CS Low	CS High

Near Jump

When the relative address stored with the instruction is in 16 bits as in the last example the jump is called a near jump. Using a near jump we can jump anywhere within a segment. If we add a large number it will wrap around to the lower part. A negative number actually is a large number and works this way using the wraparound behavior.

Short Jump

If the offset is stored in a single byte as in 75F2 with the opcode 75 and operand F2, the jump is called a short jump. F2 is added to IP as a signed byte. If the byte is negative the complement is negated from IP otherwise the byte is added. Unconditional jumps can be short, near, and far. The far type is yet to be discussed. Conditional jumps can only be short. A short jump can go +127 bytes ahead in code and -128 bytes backwards and no more. This is the limitation of a byte in singed representation.

Far Jump

Far jump is not position relative but is absolute. Both segment and offset must be given to a far jump. The previous two jumps were used to jump within a segment. Sometimes we may need to go from one code segment to another, and near and short jumps cannot take us there. Far jump must be used and a two byte segment and a two byte offset are given to it. It loads CS with the segment part and IP with the offset part. Execution therefore resumes from that location in physical memory. The three instructions that have a far form are JMP, CALL, and RET, are related to program control. Far capability makes intra segment control possible.

Sorting Example

Example 3.3

```
; sorting a list of ten numbers using bubble sort
[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      ; rest 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 second
              mov  [data+bx], dx       ; store second number in first
              mov  byte [swap], 1      ; flag that a swap has been done

noswap:       add  bx, 2               ; advance bx to next index
              cmp  bx, 18              ; are we at last index
              jne  loop1               ; if not compare 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
```

003	The jump instruction is placed to skip over data.
006	The swap flag can be stored in a register but as an example it is stored in memory and also to extend the concept at a later stage.
011-012	One element is read in AX and it is compared with the next element because memory to memory comparisons are not allowed.
013	If the JBE is changed to JB, not only the unnecessary swap on equal will be performed, there will be a major algorithmic flaw due to a logical error as in the case of equal elements the algorithm will never stop. JBE won't swap in the case of equal elements.
015-017	The swap is done using DX and AX registers in such a way that the values are crossed. The code uses the information that one of the elements is already in the AX register.
021	This time BX is compared with 18 instead of 20 even though the number of elements is same. This is because we pick an element and compare it with the next element. When we pick the 9th element we compare it with the next element and this is the last comparison, since if we pick the 10th element we will compare it with the 11th element and there is no 11th element in our case.
024-025	If a swap is done we repeat the whole process for possible more swaps.

State of Data	Swap Done	Swap Flag
Pass 1		Off
60 55 45 58	Yes	On
55 60 45 58	Yes	On
55 45 60 58	Yes	On
Pass 2		Off
55 45 58 60	Yes	On
45 55 58 60	No	On
45 55 58 60	No	On
Pass 3		Off
45 55 58 60	No	Off
45 55 58 60	No	Off
45 55 58 60	No	Off

No more passes since swap flag is Off