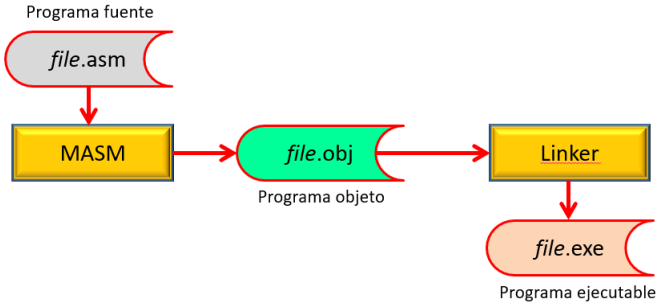
B0:

- MASM: Microsoft Macro Assembler for x86 intel processors (2 versions)



- During assembly, a listing file can also be generated, which contains info about how the asm. File was built (source code, @, offsets, symbols, ML code)
- The linker checks if the program calls another procedure or library functions.
- When running, the loader reads the exe. file into memory and branches the CPU to the program's starting @

BA:

- **DumpRegs**: It displays all the general registers and the flags.

```
.CODE
MOV ECX, 0
CALL  DumpRegs
```

- **DumpMem:** writes a range/block of memory to the console window in hex. To use it, pass in ESI the starting address of the block, in ECX the number of units or elements, and in EBX the unit size (1 : byte, 2 : word, 4 : doubleword)

```
.DATA
      array DWORD 11, 12, 13
.CODE
      MOV  ESI, OFFSET array
      MOV ECX, 3
      MOV EBX, 4
     CALL  DumpMem
```

- **WriteInt:** Writes a 32-bit signed integer to the console window in decimal format with a leading sign and no leading zeros Pass the integer into **EAX.**

```
.DATA
     valInt  SDWORD -317432
.CODE
    MOV  EAX, valInt
    CALL  WriteInt
    CALL Crlf
```

- **WriteHex** behaves the same as above,
- **WriteString:** pass, in EDX register, the string's offset

- **CrlF:** It advances the cursor, in the window console, to the beginning of the next line. It writes down a string containing the ASCII characters 0Dh and 0Ah. 0Dh is the code of CR (Carriage Return, Enter) and 0Ah is the code of LF (Line Feed).
- **ReadInt:** Reads a 32-bit signed integer from the keyboard and returns the value in EAX register. Can be typed with an optional leading plus or minus sign. It sets the Overflow flag and display an error message if the value cannot be represented as a 32-bit signed integer (-2,147,483,648 to + 2,147,483,647)
- **ReadHex:** Same for hex, no error checking for invalid characters.
- **ReadString:** It reads a string from the keyboard, stopping when the user types the ENTER key. Pass the buffer's offset in EDX register. Set ECX to the maximum number of characters that the user can type, plus 1 to save space for the terminating null byte. It returns, in EAX, the count of the number of characters typed by the user.

```
.DATA
      bufferR  BYTE 81 DUP(0)    ; 80 characters plus 0 (terminator)
      charCountR  DWORD ?
.CODE
      MOV  EDX, OFFSET bufferR
      MOV  ECX, 81
      CALL  ReadString
      MOV  charCountR, EAX
```

BB:

- **OFFSET:** returns the distance in bytes, of a label from the beginning of its enclosing DATA segment.
- **TYPE:** returns the size, in bytes, of a single element of a data declaration.
- **LENGTHOF:** counts the number of elements in a single data declaration.
- **SIZEOF:** Returns a value in bytes that is equivalent to multiplying LENGTHOF by TYPE.
- **Symbolic constant:** is not a label, it's an identifier associated with a 32-bit integer expression. Syntax: identifier = expression. They don't reserve storage and only exist during the time while the assembler is assembling a program.

BC: Mapping data values

- **LABEL:** insert a label and give it a size attribute without allocating any storage.

A common use of LABEL is to provide an alternative name and size attribute for the variable declared next in the data segment. In the following example, we declare
a label just before val32 named val16 and give it a WORD attribute:

```
.data
val16 LABEL WORD
val32 DWORD 12345678h
.code
mov ax,val16 ; AX = 5678h
mov dx,[val16+2] ; DX = 1234h
```

val16 is an alias for the same storage location as val32. The LABEL directive itself allocates no Storage.

-**"type" PTR:** overrides the declared size of an operand, allowing the selection of some part of a defined variable. Can also be used to combine elements of a smaller data type.

```
.DATA
myDouble DWORD 12345678h
.CODE
MOV EAX, myDouble       ; EAX =
MOV AX, myDouble        ; error - why? Different sizes
MOV AX, WORD PTR myDouble           ; loads 5678h
MOV WORD PTR myDouble, 4A9Bh        ; saves 4A9Bh
```

BD: Multiplication and Division

- **MUL:** Unsigned multiplies an operand by either AL, AX, EAX or RAX. The product is twice the size of the multiplier, check the CARRY flag for significant bits in the upper half.

Multiplicand	Multiplier	Product
AL	reg/mem8	AH:AL = AL * reg/mem8
AX	reg/mem16	DX:AX = AX * reg/mem16
EAX	reg/mem32	EDX:EAX = EAX * reg/mem32

- **IMUL:** signed integer multiply. Preserves the sign of the product by sign-extending it into the upper half of the destination register. CF and OF are set if the upper half of the product is not a sign extension of the lower half. 2 and 3 operand formats truncate the product. When significant digits are lost OF and CF are set.
  - 1 operand: `IMUL reg/mem(8,16,32 or 64)`
  - 2 operands:

```
IMUL reg16,reg/mem16            IMUL reg32,reg/mem32
IMUL reg16,imm8                 IMUL reg32,imm8
IMUL reg16,imm16                IMUL reg32,imm32
```

  - 3 operands:

```
IMUL reg16,reg/mem16,imm8         IMUL reg32,reg/mem32,imm8
IMUL reg16,reg/mem16,imm16        IMUL reg32,reg/mem32,imm32
```

- **DIV:** Unsigned divide, a single operand is supplied, which is the divisor (denominator).

Dividend	Divisor	Quotient	Reminder
AH:AL	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

All arithmetic status flag values are undefined (?) after executing DIV instructions. If the quotient doesn't fit the destination operand, a divide overflow condition results, causing a CPU interrupt and the current program halts. If the divisor is zero, a division by zero condition results, same outcome.

- **IDIV:** Signed integer division. Same syntax and operands as DIV instruction. Integers must be sign-extended before division takes place. Can use following instructions for sign-extension:
    - CBW (convert byte to word) extends AL into AH
    - CWD (convert word to dword) extends AX into DX
    - CDQ (convert dword to qword) extends EAX into EDX
    - CQO (convert qword to oword) extends RAX into RDX

BE: Transfer of control instructions.

- **JMP:** Causes an unconditional transfer of control (jump) to a destination (label:) inside .CODE. The destination is translated like an offset. `JMP label`

- ***Jcond* instructions:** The conditional transfer only jumps if the condition of a flag or several flags are met. Only ALU instructions can modify flags, the Jcond analyzes the CPU status flags affected by the last executed ALU instruction.

Mnemonic	Description	Flags
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

Jconds based on one flag

BF:

- Jconds are hard to use because the predecessor instructions usually affect several flags at the same time.

- **CMP:** Is used to create conditional logic structures. `CMP leftOp, rightOp` nondestructive implied subtraction. Following combinations permitted:

```
CMP reg, reg          CMP mem, reg          CMP reg, imm
CMP reg, mem          CMP mem, imm
```

`CMP leftOp, rightOp`	Banderas	Jcond
destino == fuente	ZF = 1 y CF = 0	JE
destino <> fuente	not(ZF = 1 y CF = 0)	JNE
destino > fuente	ZF = 0 y CF = 0	JA
destino <=fuente	not(ZF = 0 y CF = 0)	JBE
destino < fuente	ZF = 0 y CF = 1	JB
destino >= fuente	not(ZF = 0 y CF = 1)	JAE

CMP opr sin signo

`CMP leftOp, rightOp`	Banderas	Jcond
destino == fuente	ZF = 1 y CF = 0/1	JE
destino <> fuente	not(ZF = 1 y CF = 0/1)	JNE
destino > fuente	SF == OF	JG
destino <=fuente	not(SF == OF)	JNG
destino < fuente	SF != OF	JL
destino >= fuente	not(SF != OF)	JNL

CMP opr con signo

BG:

- **IF-then implementation:** Involves a conditional jump for the negative of the comparison.

```
    CMP Lfop, Rhop
    JnoCMP outIF
          ;inside if block
outIF:
```

- **If-then-else:** Involves a conditional jump for the negative of the comparison. A JMP is

needed at the end of the block1 to avoid going into the ELSE code.

```
    CMP Lfop, Rhop
    JnoCMP inELSE
          ;inside if block
    JMP outIF
inELSE:
        ;inside else block
outIF:
```

- **While:** Involves a conditional jump for the negative of the comparison. After the block, it requires to jump back to compare again.

```
inWhile: CMP Lfop, Rhop
         JnoCMP outWHILE
                  ;while block
         JMP inWhile
outWHILE:
```

- **DoWhile:** Involves a conditional jump for the same comparison. After the block, it requires to jump back to compare.

```
inDO:       ;do while block
         CMP Lfop, Rhop
         JCMP inDO
```

BH:

- **Bitwise Boolean instructions:** There are no logical boolean assembly instructions, only bitwise (bit to bit).
- **AND:** syntax with destination and source, must be the same size. Can be used for bit masking or intersection of bits. Clears overflow, carry. Modifies Sign, Zero and Parity.
- **OR:** Same rules as AND, useful to set bits without affecting the others, or union of bits.
- **XOR:** Same rules as before. Useful way to invert bits in an operand.
- **NOT:** Returns 1s complement of reg or mem. No flags are affected.
- **TEST:** Performs a non destructive AND bitwise operation between each pair of matching bits in two operands. No operands are modified. Always clears the overflow and carry flags, and modifies the sign, zero and parity flags.

BJ: Macro Procedure

- A MACRO procedure is a named block of Assembly Language instructions. Every time this named block is invoked, Assembler during the preprocessing step expands a copy of the block of instructions into the program code.

```
Macroname MACRO [parameter-1,parameter-2,...]
      ;statement list
ENDM
```

- When you invoke a MACRO, each argument you pass matches a declared parameter.
- Each parameter is replaced by its corresponding argument when the macro is expanded.

BK: Conditional control flow macro directives

Directive	Description
.BREAK	Generates code to terminate a .WHILE or .REPEAT block
.CONTINUE	Generates code to jump to the top of a .WHILE or .REPEAT block
.ELSE	Begins block of statements to execute when the .IF condition is false
.ELSEIF condition	Generates code that tests condition and executes statements that follow, until an .ENDIF directive or another .ELSEIF directive is found
.ENDIF	Terminates a block of statements following an .IF, .ELSE, or .ELSEIF directive
.ENDW	Terminates a block of statements following a .WHILE directive
.IF condition	Generates code that executes the block of statements if condition is true.
.REPEAT	Generates code that repeats execution of the block of statements until condition becomes true
.UNTIL condition	Generates code that repeats the block of statements between .REPEAT and .UNTIL until condition becomes true
.WHILE condition	Generates code that executes the block of statements between .WHILE and .ENDW as long as condition is true

```
.IF eax > ebx && eax > ecx
      mov edx,1
.ELSE
      mov edx,2
.ENDIF
```

Operator	Description
expr1 == *expr2*	Returns true when *expression1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 >= *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 <= *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! *expr*	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 || *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

- **Signed and unsigned comparisons:** If one of the variables being compared is signed, MASM automatically generates a signed jump. When both operands are registers, MASM automatically generates an unsigned jump. Unless, you prefix one of the register operands with the type PTR operator. Then a signed jump is generated.

```
.DATA
result SDWORD ?

.CODE
mov EBX,5
mov EAX,6
.IF SDWORD PTR EAX > EBX
  mov result,1
.ENDIF
```

BL: Stack

- LIFO data structure, values added to the top (push) and removed from the top (pop)
- **Runtime stack:** Stack inside every process, managed by the CPU, using a stack pointer register. ESP (Extended Stack Pointer) always points to the last value added.
- **PUSH:** One operand, accepts reg/mem (16,32,64) or imm (32,64). The operation decrements the stack pointer and copies a value into the location pointed to by the stack pointer ESP. The stack grows downwards.
- **POP:** One operand, accepts reg/mem (16,32,64).
- The stack can be used to save and restore registers when they contain important values.
- **PUSHFD and POPFD:** push and pop the EFLAGS register.
- **PUSHAD:** pushes the 32-bit general-purpose registers on the stack. Order: EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI.
- **POPAD:** pops the same registers in reverse order.
- **PUSHA and POPA:** Do the same for 16-bit registers. Order: AX,CX,DX,BX,SP,BP,SI,DI.

BM: Addressing modes

- Refers to the way in which the CPU accesses the effective values from the operands of an instruction. Two general addressing modes: direct and indirect.
- **Direct addressing:** The operand has the location to place an effective value or the content to get it.
- **Indirect addressing:** The operand has the memory address (offset) of the location to place an effective value or the memory address of the content to get a value. It means the operand has a pointer.
- **Indirect operands [reg] :** reg contains an address. Could be any general-purpose register. ESI and EDI preferred registers. ESI for source, EDI for destination. An indirect operand holds the address of a variable. It can be dereferenced. In protected mode, if the effective address points to an area outside your program's data segment, the CPU executes a General Protection Fault.

```
.DATA
val1 BYTE 10h,20h,30h

.CODE
MOV ESI,OFFSET val1
MOV AL,[ESI]      ; ESI=0000 0000h; AL = 10h
INC ESI
MOV AL,[ESI]      ; ESI=0000 0001h; AL = 20h
INC ESI
MOV AL,[ESI]      ; ESI=0000 0002h; AL = 30h
```