Assemble-Link-Execute cycle

• Step 1: A programmer uses a text editor to create an ASCII text file named the source file.

• Step 2: The assembler reads the source file and produces an object file

    • Object file is a machine-language translation of the program.

• Step 3: The linker reads the object file and checks to see if the program contains any calls to procedures in a link library

• Step 4: The operating system loader utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.

Reserved Words

• Reserved words cannot be used as identifiers

• not case sensitive

• Instruction mnemonics, directives, type attributes, operators, predefined

symbols

Identifiers

• 1-247 characters, including digits

• not case sensitive

Labels

• Data label

    • must be unique

    • example: myArray

• Code label

    • target of jump and loop instructions

    • example: L1:

x86

•The program to be run by the processor is written in memory (In RAM).

•The processor can only understand the numeric representation of the instructions.

•Opcode= operation code, tells the processor what operation should be performed

•Mnemonic a friendly term that describes opcode

•Operand the argument or parameter following the mnemonics

•Instructionsoperations + Operands

Registers

•Basic registers:

- •eax–Accumulator.
- •ebx–Base index
- •ecx–Counter
- •edx–Data register

•Almost every 32-bit register has a 64-bit equivalent.

Segments

•.data identifies the area of the program containing variables

•.code identifies the area of the program containing executable instructions

•.stack 100h identifies the area of the program holding the runtime stack, setting its size

Listing File

•Use it to see how your program is compiled

•Contains

- •source code
- •addresses
- •object code (machine language)
- •segment names
- •symbols (variables, procedures, and constants)

Directives

•A command embedded in the source code that is recognized and acted upon by the assembler.

•Directives can define variables, macros and procedures

•They can assign names to memory segments. NOT CASE SENSITIVE

Data Definition Statement

•A data definition statement sets aside storage in memory for a variable.

[name] directive initializer [,initializer]

Example:

val1 BYTE 10

String

•A string is implemented as an array of characters

•End-of-line character sequence:

- •0Dh = carriage return
- •0Ah = line feed

Example:

•str1 BYTE "Enter your name",0

DUP

•Use DUP to allocate (create space for) an array or string.

Example:

•var1 BYTE 20 DUP(?) ;20 bytes, uninitialized

Defining BYTE and SBYTE Data

•Defines a single byte of storage

Defining WORD and SWORD Data

•Define storage for 16-bit integers

Defining DWORD and SDWORD Data

•Storage definitions for signed and unsigned 32-bit integers

Defining QWORD, TBYTE, Real Data

•Storage definitions for quadwords, tenbyte values, and real numbers

Example:

•quad1 QWORD 1234567812345678h

•val1 TBYTE 10000000000123456789Ah

•rVal1 REAL4-2.1

Big Endian Order

•The most significant byte (the "big end") of the data is placed at the byte with the lowest address.

Example:

•val1 DWORD 12345678h

0000:	12
0001:	34

0002:	56
0003:	78

Little Endian Order

•The least significant byte occurs at the first (lowest) memory address.

Example:

•val1 DWORD 12345678h

0000:	78
0001:	56
0002:	34
0003:	12

Declaring Uninitialized Data

•declare variables with "?"

Example:

•smallArray DWORD 10 DUP(?)

Operand Types

•Immediate operand—uses a numeric or character literal expression

•Register operand—uses a named CPU register

•Memory operand—references a memory location

MOV

•Performs data moves (manipulation)

•Data is copied from source to destination

<table>
<tr><td colspan="3">eax (32 bit)</td></tr>
<tr><td></td><td colspan="2">ax (16 bit)</td></tr>
<tr><td></td><td>ah (8 bit)</td><td>al (8 bit)</td></tr>
</table>

Example:

•mov eax, 8CBh

•mov ecx, edx

•Copy content of edx to ecx

Invalid: mov ecx, dh ;size mismatch

ADD

•ADD destination, source

•$destination \leftarrow destination + source$

Example:

•add eax,edx

- •Adds the contents of eax and edx.
- •Stores the result in eax. ($eax \leftarrow eax + edx$)

Invalid: add 532h, ecx ;cannot be stored in 532h, not a destination

SUB

•SUB destination, source.

•$destination \leftarrow destination - source$

Example:

•sub eax,edx

- •Subtracts edx from eax, and stores the result in eax.

Invalid: sub eax, dl ;bit size difference

INC & DEC

•Unary operators

•mov eax, FFFFFFFEh

•inc eax ;eax = FFFFFFFFh

•Invalid: inc 1C5h

DIV

•Unsigned division, DIV arg

•arg of size 8 bits:

- •$al \leftarrow ax/arg$;Quotient
- •ah $\leftarrow ax\%\ arg$;Remainder

•arg of size 16 bits:

- •$ax \leftarrow dx{:}ax/arg$
- •$dx \leftarrow dx{:}ax\%arg$

•arg of size 32 bits:

- •$eax \leftarrow edx{:}eax/\ arg$

- $edx \leftarrow edx{:}eax \% arg$

- Invalid example: div 5CAh

Example:

- mov ax,0083h ; dividend
- mov bl,2 ; divisor
- div bl ; AL = 41h, AH = 01h

MUL

- $ax \leftarrow al \cdot argument$; If argument is of size 8 bits.
- $dx{:}\ ax \leftarrow ax \cdot argument$; If argument is of size 16 bits.
- $edx{:}\ eax \leftarrow eax \cdot argument$; If argument is of size 32 bits.
- Invalid example: mul2Ah

Example:

- mov al,5h
- mov bl,10h
- mul bl
- AX = 0050h, CF = 0

The flag register

- A 32-bit register inside the x86 processor.
- Has 64 bit extension for long-mode.
- Every bit in this register is “a flag”: It represents True or False.
- Bits values reflect on the result of the last calculation.
- Flags will help us write a code with decisions and branches

Bit Number	Short Name	Description
0	CF	Carry flag
1	1	Reserved
2	PF	Parity flag
3	0	Reserved
4	AF	Auxiliary Carry flag
5	0	Reserved
6	ZF	Zero flag
7	SF	Sign flag
8	TF	Trap flag

9	IF	Interrupt enable flag
10	DF	Direction flag
11	OF	Overflow flag
More bits…..		

Every instruction can have certain effects on some bits of the flags register.

Flag	Description	Example

Zero flag	•The zero flag is set(to 1) whenever the last calculation had the result of zero.
	•It will be cleared(to 0) whenever the last calculation had a nonzero result.
		mov eax,3h
		mov ecx,3h
		sub eax,ecx

Sign flag	•It is a copy of the most significant bit •0 if the result is positive in the two's complement representation. •1 if the result is negative in the two's compliment representation	mov edx,0 dec edx
Carry flag	•The carry flag is set if the addition of two numbers causes a carry out of the most significant bits	mov eax, 0ffffffh add eax, 1

Overflow flag	•Set if the addition of two positive numbers has a negative result.
	•Set if the addition of two negative numbers has positive result.
	•Set if "positive –negative" has a negative result.
	•Set if "negative –positive" has a positive result.
		mov al, 7fh
		mov cl, 1h
		add al, cl

Branching

•Unconditional Transfer: Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address.

•Conditional Transfer: The program branches if a certain condition is true. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

JMP

•JMP destination

•When this is executed the offset of destination is moved into the instruction pointer, causing execution to continue at the new location

Jcond Instruction	Jcond destination
JC	Jump if carry (Carry flag set)
JNC	Jump if not carry (Carry flag clear)
JZ	Jump if zero (Zero flag set)
JNZ	Jump if not zero (Zero flag clear)

Conditional jump	Description
JS/JNS	Jump if sign set/cleared
JC/JNC	Jump if carry set/cleared
JO/JNO	Jump if overflow set/cleared

CMP

•CMP instruction, which is useful for numbers comparison.

•Subtracts: A –B, Changes flags accordingly but

•doesn't change A or B.

•CMP A, B

•0xffffffff >0x00000001 considering unsignednumbers.

•0xffffffff < 0x00000001 considering signed numbers (Two's complement).

Unsigned Comparison

Instruction	Condition being checked
JB (Jump Below)	CF = 1 (left op < right op)
JBE (Jump Below Equal)	CF = 1 or ZF = 1 (left op ≤ right op)
JA (Jump Above)	CF = 0 and ZF = 0 (left op > right op)
JAE (Jump Above Equal)	CF = 0 (left op ≥ right op)

Signed Comparison

Instruction	Condition being checked
JG (Jump Greater)	SF = 0 and ZF = 0 (left op > right op)
JGE (Jump Greater Equal)	SF = OF (left op ≥ right op)
JL (Jump Less)	SF ≠ OF (left op < right op)
JLE (Jump Less Equal)	SF ≠ OF or ZF = 1 (left op ≤ right op)

Jump based on equality

Mnemonic	Description
JE	Jump if equal left op = right op

JNE	Jump if not equal left op ≠ right op
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0
JRCXZ	Jump if RCX – 0 (64 – bit mode