CSec15233
Malicious Software Analysis

Numbers,

Assemble (Op-coding),

Disassemble

Qasem Abu Al-Haija

Number Systems. Sign, Overflow, and Floating Point

Number Systems

Decimal-to-Hexadecimal:

$420.625_{10} =$

$420.625_{10} = 420_{10} + .625_{10}$

Division	Quotient	Remainder	
420 ÷ 16	26	4	LSB
26 ÷ 16	1	10 (or A)	
1 ÷ 16	0	1	MSB

Multiplication	Product	Carry-out
.625 x 16	10.00	10 (or A)

$420.625_{10} = 1A4.A_{16}$

$4135_{10} = 1027_{16}$

$625.625_{10} = 271.A_{16}$

Number Systems

Binary-Coded Hexadecimal (BCH):

2AC = 0010 1010 1100

1000 0011 1101 . 1110 = 83D.E

Complements

Data are stored in complement form to represent negative numbers

One's complements of 01001100

```
 1111 1111
-0100 1100
 1011 0011
```

Two's complements

```
  1011 0011
+0000 0001
  1011 0100
```

Two's Complement

Decimal	Binary
-128	1000 0000
-127	1000 0001
-126	1000 0010
…	
-2	1111 1110
-1	1111 1111
0	0000 0000
+1	0000 0001
+2	0000 0010
…	
+127	0111 1111

Basic Terminology

bit

byte

nibble

word

kilobyte

megabyte

gigabyte

terabyte

Signed Byte Operands

- D7 (MSB) is the sign, 0 means positive
- D0 to D6 are used for the magnitude of the number
- The range of positive number is [0,127]
- If 2's complement is used for negative numbers, the range is ?
- How do you represent –127 in binary format? How about –128?

Word-Sized Signed Numbers

- D15 is for the sign
- D0-D14 is for the magnitude
- What will be the range?
- Can Pentium 4 specify double word sized signed numbers for operands? What will be the range?

Overflow

- What is an overflow?
- If the result of an operation on signed numbers is too large for the register, the overflow flag (OF) will be raised to notify the programmers
- It is up to the programmer to take care of it
- EXP: 96 +70 0110 0000

  0100 0110

  1010 0110

Overflow in 8-bit operations

- OF is set to 1 if:

(1) There is a carry from D6 to D7 but no carry out of D7 (CF=0)

(2) There is a carry from D7 out (CF=1) but no carry from D6 to D7

- If there is a carry both from D6 to D7 and from D7 out, OF=0
- Why?

Overflow in 16-bit operations

- OF is set to 1 if:

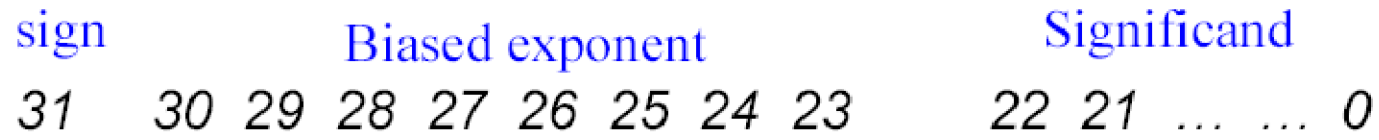
(1) There is a carry from D14 to D15 but no carry out of D15 (CF=0)

(2) There is a carry from D15 out (CF=1) but no carry from D14 to D15

- Examples (page 177, 178)
- The idea is to make use of the OF to handle the overflow problem

IEEE Single-Precision Floating-point Numbers

- 32 bits of data

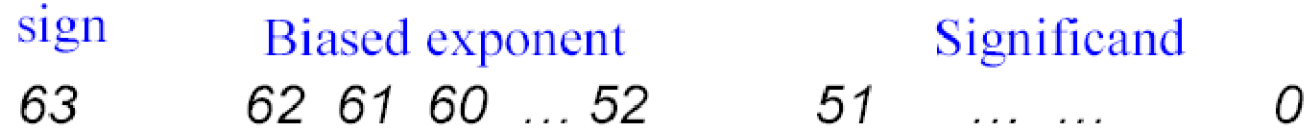


- **Convert from real to single-precision floating-point**

1. From real to binary form
2. Represent the binary number in scientific form **1**.xxxx E yyyy
3. Assign the sign bit
4. The exponent portion yyyy is added to 7F, to obtain the biased exponent, place in bits 23 to 30
5. The significand xxxx is placed in bits 22 to 0

IEEE Double-Precision Floating-point Numbers

- 64 bits of data



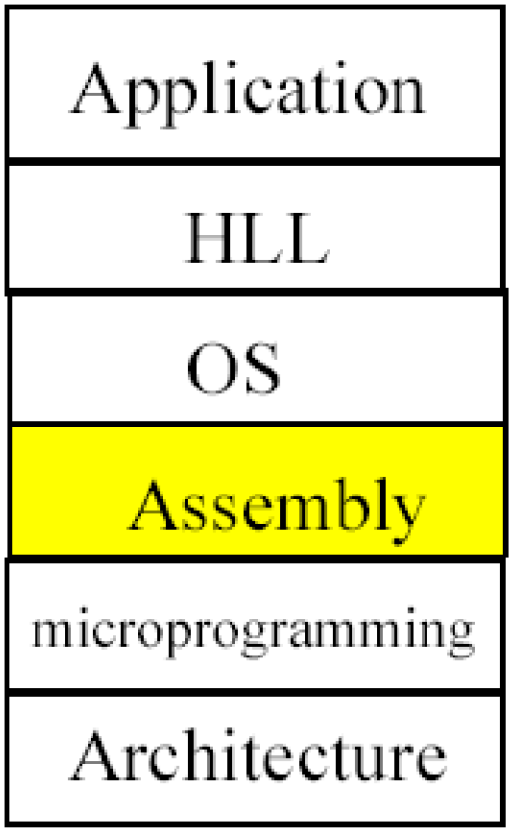
- Convert from real to floating-point

1. From real to binary form
2. Represent the binary number in scientific form **1**.xxxx E yyyy
3. Assign the sign bit
4. The exponent portion yyyy is added to 3FF, to obtain the biased exponent, place in bits 52 to 63
5. The significand xxxx is placed in bits 51 to 0

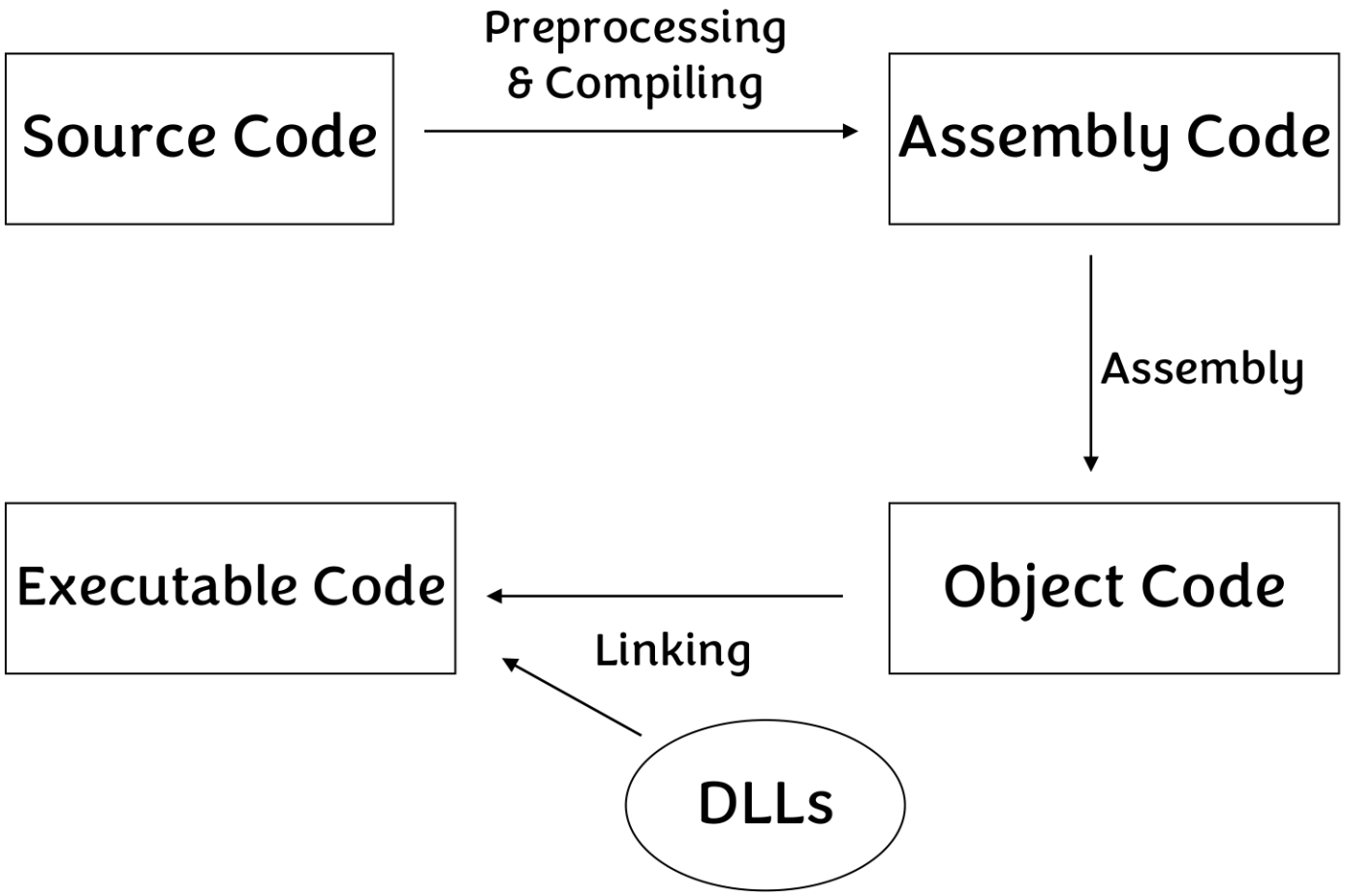
Assemble: Converting Assembly Language Code To Machine Language Code

Assembly Programming

- Machine Language
  - binary
  - hexadecimal
  - machine code or object code
- Assembly Language
  - mnemonics
  - assembler
- High-Level Language
  - Pascal, Basic, C
  - compiler



What Does It Mean to Assemble Code?



Key Benefits of Assembly Language

- There is a one-to-one relationship between the assembly and machine language instructions
- What is found is that a compiled machine code implementation of a program written in high-level language results in inefficient code
  - More machine language instructions than an assembled version of an equivalent handwritten assembly language program
- Two key benefits of assembly language programming
  - It takes up less memory
  - It executes much faster

Languages in terms of applications

- One of the most beneficial uses of assembly language programming is real-time applications.

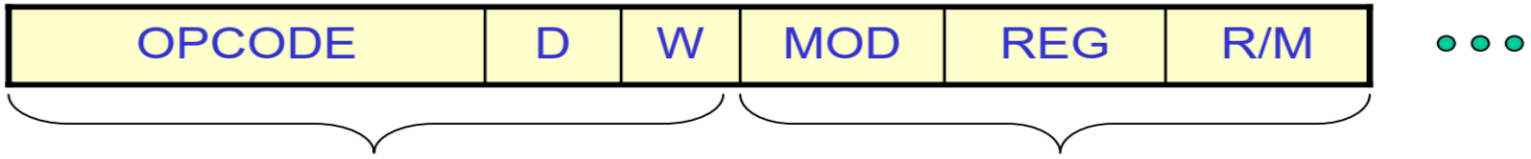
- Real time means the task required by the application must be completed before any other input to the program that will alter its operation can occur.

- For example, the device service routine which controls the operation of the floppy disk drive is a good example that is usually written in assembly language

Languages in terms of applications

- Assembly language is not only good for controlling hardware devices but also for performing pure software operations
  - Searching through a large table of data for a special string of characters
  - Code translation from ASCII to EBCDIC
  - Table sort routines
  - Mathematical routines
- Assembly language: perform real-time operations
- High-level languages: Those operations mostly not critical in time

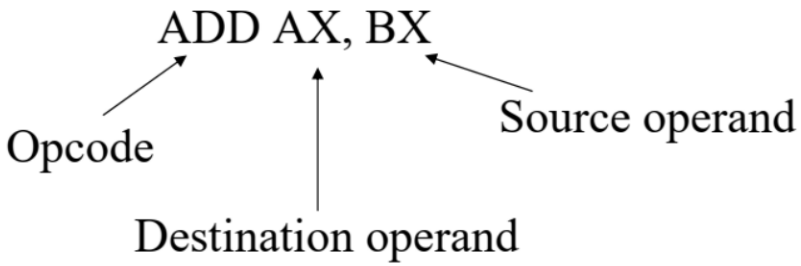
Converting Assembly Language Instructions to Machine Cod



- An instruction can be coded with 1 to 6 bytes
- **Byte 1 contains three kinds of information:**
  - Opcode field (6 bits) specifies the operation such as add, subtract, or move
  - Register Direction Bit (D bit)
    - Tells the register operand in REG field in byte 2 is source or destination operand
      - 1:Data flow to the REG field from R/M
      - 0: Data flow from the REG field to the R/M
  - Data Size Bit (W bit)
    - Specifies whether the operation will be performed on 8-bit or 16-bit data
      - 0: 8 bits
      - 1: 16 bits
- **Byte 2 has two fields:**
  - Mode field (MOD) – 2 bits
  - Register field (REG) - 3 bits
  - Register/memory field (R/M field) – 2 bits

Converting Assembly Language Instructions to Machine Cod

- The sequence of commands used to tell a microcomputer what to do is called a **program**
- Each command in a program is called an **instruction**
- 8086 understands and performs operations for **117 basic instructions**
- The native language of the **IBM PC** is the machine language of 8086/8088
- A program written in machine code is referred to as **machine code**
- In 8086 assembly language, each of the operations is described by alphanumeric symbols instead of just 0s or 1s



Converting Assembly Language Instructions to Machine Cod

- **REG field is used to identify the register for the first operand**

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Converting Assembly Language Instructions to Machine Cod

- 2-bit MOD field and 3-bit R/M field together specify the second operand

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

(a)

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

(b)

Example:

- MOV BL,AL
- Opcode for MOV = 100010
- We'll encode AL so
  - D = 0 (AL source operand)
- W bit = 0 (8-bits)
- MOD = 11 (register mode)
- REG = 000 (code for AL)
- R/M = 011 **(Code for BL)**

OPCODE	D	W	MOD	REG	R/M
100010	0	0	11	000	011

MOV BL,AL => 10001000 11000011 = 88 C3h

ADD AX,[SI] => 00000011 00000100 = 03 04 h

ADD [BX][DI] + 1234h, AX => 00000001 10000001 __ __ h

=> 01 81 34 12 h

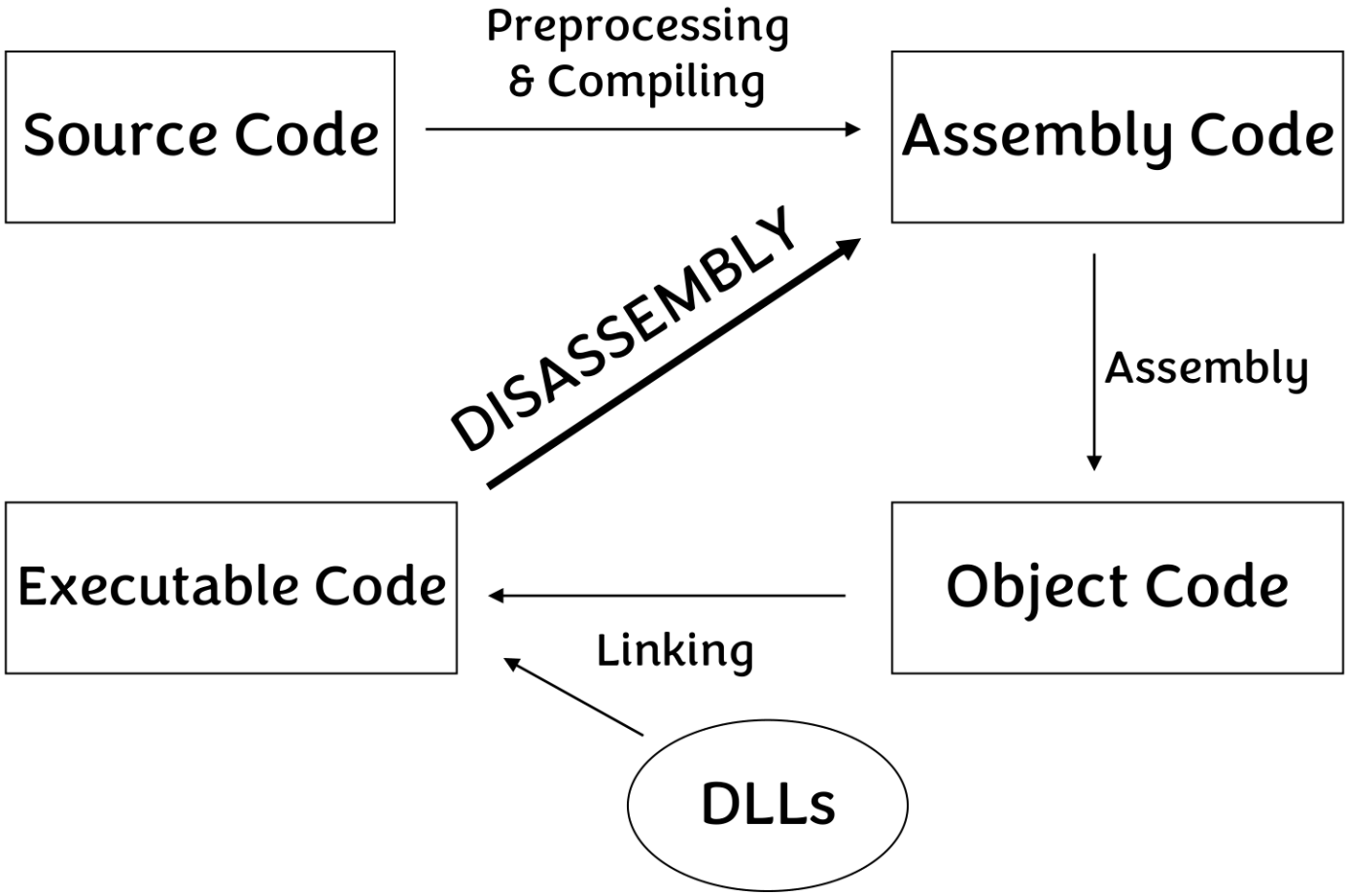
More Examples

ADD 5678H[BX][SI], CX	01 88 78 56 H
SUB DX, AX	29 C2 H
CMP AX, CX	39 C8 H
MOV [0004], AX	A3 04 00 H
PUSH AX	50 H
POP DX	5A H
JZ 20H	74 1E H

The rule for assembling JUMP instructions: Opcode **Disp – 2**

Disassemble: Converting Machine Language Code To Assembly Language Code

What Does It Mean to Disassemble Code?



Why is Disassembly Useful in Malware Analysis?

- It is not always desirable to execute malware: disassembly provides a static analysis.
- Disassembly enables an analyst to investigate all parts of the code, something that is not always possible in dynamic analysis.
- Using a disassembler and a debugger in combination creates synergy.

Disassembly of Machine Codes

- Indeed, there's no real difference between machine language and any other programming language; machine language is just a little harder to read.
  - Understanding it requires patience and the right reference.

- Finding the right reference is a large matter of knowing which CPU architecture the machine language was written for as each type of CPU has its own dialect.
  - It can also be important to know what CPU mode the machine language was written for.
  - Modern x86 CPUs, for instance, can be configured to use 16- or 32-bit operands and addressing by default, and the same sequence of machine language bytes may mean different things depending upon the CPU's state.
  - Matters become even more complex when 64-bit instructions are introduced.

Disassembly of Machine Codes

- Since we're looking at a DOS (i.e., x86 real-mode) executable, a good reference is the Instruction Set Reference (ISR) volume from the Intel Architecture Software Developer's Manual.

- This is a formidable volume, but only a few pages are immediately interesting for our purposes. For instance;
  - Pages 1-2 through 2-6 describe the basic layout of x86 machine language instructions. (Note that since we're dealing with real-mode machine language, we're only interested in 16-bit addressing modes.)
  - Pages A-1 through A-8 give the processor's opcode map. (Note that since we're dealing with such an old program, we can assume that it only uses 8086 integer opcodes; this means that we can ignore all two-byte and escape opcodes in the opcode map.)

8086 Instruction Set Opcodes (1)

Operation	Operands	Opcode
ADC	see ADD	ADD opcode + $10, and xx010xxx (ModR/M byte) for $80-$83
ADD	r/m8, reg8	$00
ADD	r/m16, reg16	$01
ADD	reg8, r/m8	$02
ADD	reg16, r/m16	$03
ADD	AL, imm8	$04
ADD	AX, imm16	$05
ADD	r/m8, imm8	$80 xx000xxx (ModR/M byte)
ADD	r/m16, imm16	$81 xx000xxx (ModR/M byte)
ADD	r/m16, imm8	$83 xx000xxx (ModR/M byte)
AND	see ADD	ADD opcode + $20, and xx100xxx (ModR/M byte) for $80, $81,$83
CALL	32-bit displacement	$9A
CALL	16-bit displacement	$E8
CLD		$FC
CMP	See ADD	ADD opcode + $38, and xx111xxx (ModR/M byte) for $80, $81,$83
CMPSB	ES:[DI]==DS:[SI]	$A6
CMPW	ES:[DI]==DS:[SI]	$A7
DEC	r/m8	$FE, xx001xxx (ModR/M byte)
DEC	r/m16	$FF, xx001xxx (ModR/M byte)
DEC	reg16	$48 + reg16 code
DIV	r/m8	$F6, xx110xxx (ModR/M byte)
DIV	r/m16	$F7, xx110xxx (ModR/M byte)
HLT		$F4
IDIV	r/m8	$F6, xx111xxx (ModR/M byte)
IDIV	r/m16	$F7, xx111xxx (ModR/M byte)
IMUL	r/m8	$F6, xx101xxx (ModR/M byte)
IMUL	r/m16	$F7, xx101xxx (ModR/M byte)

8086 Instruction Set Opcodes (2)

Operation	Operands	Opcode
IN	AL, addr8	$E4
IN	AX, addr8	$E5
IN	AL, port[DX]	$EC
IN	AX, port[DX]	$ED
INC	r/m8	$FE, xx000xxx (ModR/M byte)
INC	r/m16	$FF, xx000xxx (ModR/M byte)
INC	reg16	$40 + reg16 code
IRET	48-bit POP	$CF
JA	8-bit relative	$77
JAE	8-bit relative	$73
JB	8-bit relative	$72
JBE	8-bit relative	$76
JE	8-bit relative	$74
JG	8-bit relative	$7F
JGE	8-bit relative	$7D
JL	8-bit relative	$7C
JLE	8-bit relative	$7E
JMP	32-bit displacement	$EA
JNE	8-bit relative	$75
JZ	8-bit relative	$74
LDS	reg16, mem32	$C4
LES	reg16, mem32	$C5
LODSB	AL = DS:[SI]	$AC
LODSW	AX = DS:[SI]	$AD

8086 Instruction Set Opcodes (3)

Operation	Operands	Opcode
LOOP	8-bit relative	$E2
MOV	r/m8, reg8	$88
MOV	r/m16, reg16	$89
MOV	AL, mem8	$A0
MOV	AX, mem16	$A1
MOV	mem8, AL	$A2
MOV	mem16, AX	$A3
MOV	reg8, imm8	$B0 + reg8 code
MOV	reg16,imm16	$B8 + reg16 code
MOV	r/m8, imm8	$C6, xx000xxx(ModR/M byte)
MOV	r/m16, imm16	$C7, xx000xxx(ModR/M byte)
MOV	r/m16,sreg	$8C, xx0 sreg xxx(ModR/M byte)
MOV	sreg, r/m16	$8E, xx0 sreg xxx(ModR/M byte)
MOVSB	ES:[DI] = DS:[SI]	$A4
MOVSW	ES:[DI] = DS:[SI]	$A5
MUL	r/m8	$F6, xx100xxx (ModR/M byte)
MUL	r/m16	$F7, xx100xxx (ModR/M byte)
NEG	r/m8	$F6, xx011xxx (ModR/M byte)
NEG	r/m16	$F7, xx011xxx (ModR/M byte)
NOT	r/m8	$F6, xx010xxx (ModR/M byte)
NOT	r/m16	$F7, xx010xxx (ModR/M byte)
OR	see ADD	ADD opcode + $08, and xx001xxx (ModR/M byte) for $80, $81,$83

8086 Instruction Set Opcodes (4)

Operation	Operands	Opcode
OUT	addr8, AL	$E6
OUT	addr8, AX	$E7
OUT	port[DX], AL	$EE
OUT	port[DX], AX	$EF
POP	r/m16	$8F
POP	reg16	$58 + reg16 code
POP	sreg	$07 + ES = 0, CS = 8, SS = $10, DS = $18
PUSH	r/m16	$FF, xx110xxx (ModR/M byte)
PUSH	reg16	$50 + reg16 code
PUSH	sreg	$06 + ES = 0, CS = 8, SS = $10, DS = $18
REP		$F3
REPNE		$F2
RET	32-bit POP	$CA
RET	16-bit POP	$C2
SBB	see ADD	ADD opcode + $18, and xx011xxx (ModR/M byte) for $80, $81,$83
SCASB	ES:[DI] == AL	$AE
SCASW	ES:[DI] == AX	$AF
STD		$FD
STOSB	ES:[DI] = AL	$AA
STOSW	ES:[DI] = AX	$AB
SUB	see ADD	ADD opcode + $28, and xx101xxx (ModR/M byte) for $80, $81,$83
XOR	see ADD	ADD opcode + $30, and xx110xxx (ModR/M byte) for $80, $81,$83

8086 Instruction Set Opcodes (5)

addr8 = 8-bit address of I/O port
reg8 = AL = 0, CL = 1, DL = 2, BL = 3, AH =4, CH = 5, DH = 6, BH = 7
reg16 = AX = 0, CX =1, DX =2, BX =3, SP = 4, BP = 5, SI = 6, DI = 7
sreg = ES = 0, CS = 1, SS = 2, DS = 3
mem8 = memory byte (direct addressing only)
mem16 = memory word (direct addressing only)
r/m8 = reg8 or mem8
r/m16 = reg16 or mem16
imm8 = 8 bit immediate
imm16 = 16 bit immediate

Example 1 of Code Disassembly

Assume the first bytes of machine language code are located at offset 01000H. They are:

8C C0 05 10 00 0E 1FA3 04 00 03 06 0C 00 8E C0 …

Disassemble this code to obtain an assembly language code?

Example of Code Disassembly

8C C0	MOV AX ES
05 10 00	ADD AX 0010H
0E	PUSH CS
1F	POP DS
A3 04 00	MOV [0004], AX
03 06 0C 00	ADD AX , [000C]
8E C0 ...	MOV ES AX

Example 2 of Code Disassembly

Assume the first bytes of machine language code are located at offset 01000H. They are:

01 81 56 78 8E C0 8A D8 0E 04 3D 03 06 0C 00 1F 8C C0 E8 00 02

Disassemble this code to obtain an assembly language code.

Example of Code Disassembly

00000H: 01 81 56 78	ADD [BX] [DI] + 7856H, AX
00004H: 8E C0	MOV ES, AX
00006H: 8A D8	MOV BL, AL
00008H: 0E	PUSH CS
00009H: 04 3D	ADD AL, 3DH
0000BH: 74 1F	JE 21H
0000DH: 03 06 0C 00	ADD AX, [000CH]
00011H: 1F	POP DS
00012H: 8C C0	MOV AX, ES

The rule for Disassembling JUMP instructions: Opcode **Disp + 2**

Online Assembler and Disassembler

Provide Assembler and Disassembler tools for different microprocessor architectures

Try it here

https://shell-storm.org/online/Online-Assembler-and-Disassembler/

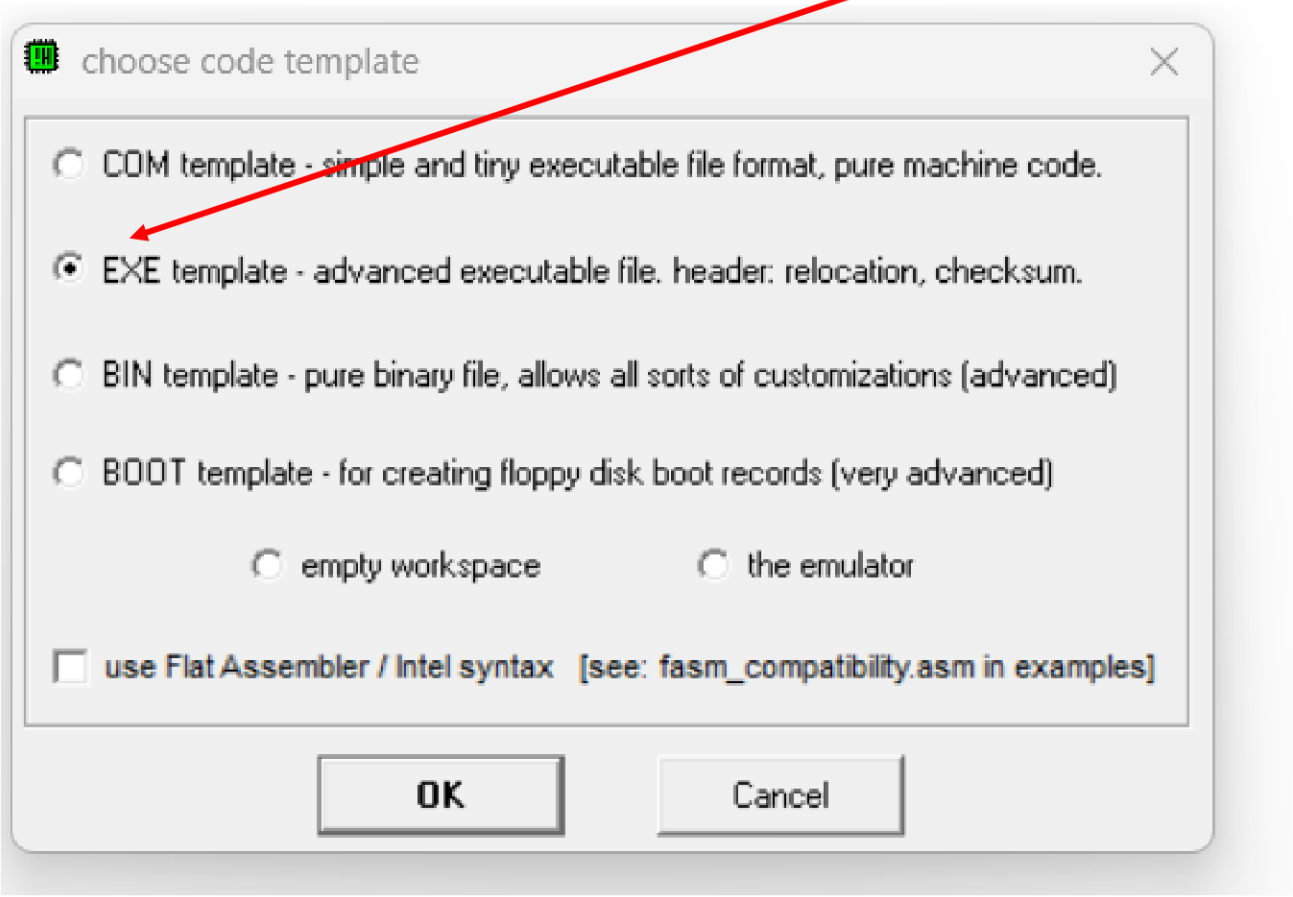
Creating.EXE applications

Using MASM Assembler

Using MASM

- Developed by Microsoft
- Used to translate 8086 assembly language into machine language
- 3 steps:
  - Prepare.ASM file using a text editor
  - Compile your.ASM file using MASM
  - Create. EXE file using LINKer
  - Once you have the. EXE file, debug can be used to test and run the program

Using MASM

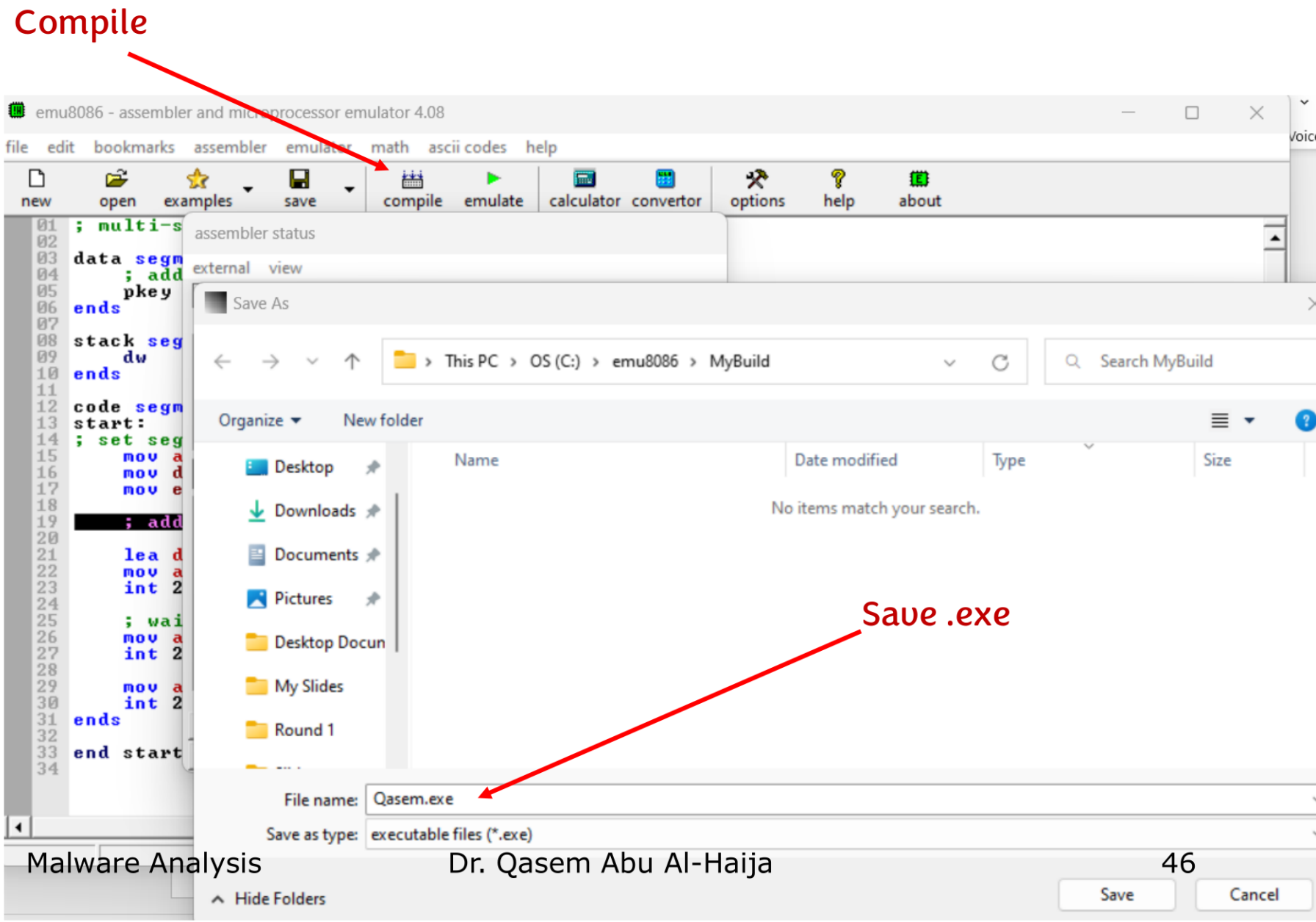

Select this option
choose code template
COM template - simple and tiny executable file format, pure machine code.
EXE template - advanced executable file. header: relocation, checksum.
BIN template - pure binary file, allows all sorts of customizations (advanced)
BOOT template - for creating floppy disk boot records (very advanced)
empty workspace
the emulator
use Flat Assembler / Intel syntax [see: fasm_compatibility.asm in examples]
OK
Cancel

Using MASM

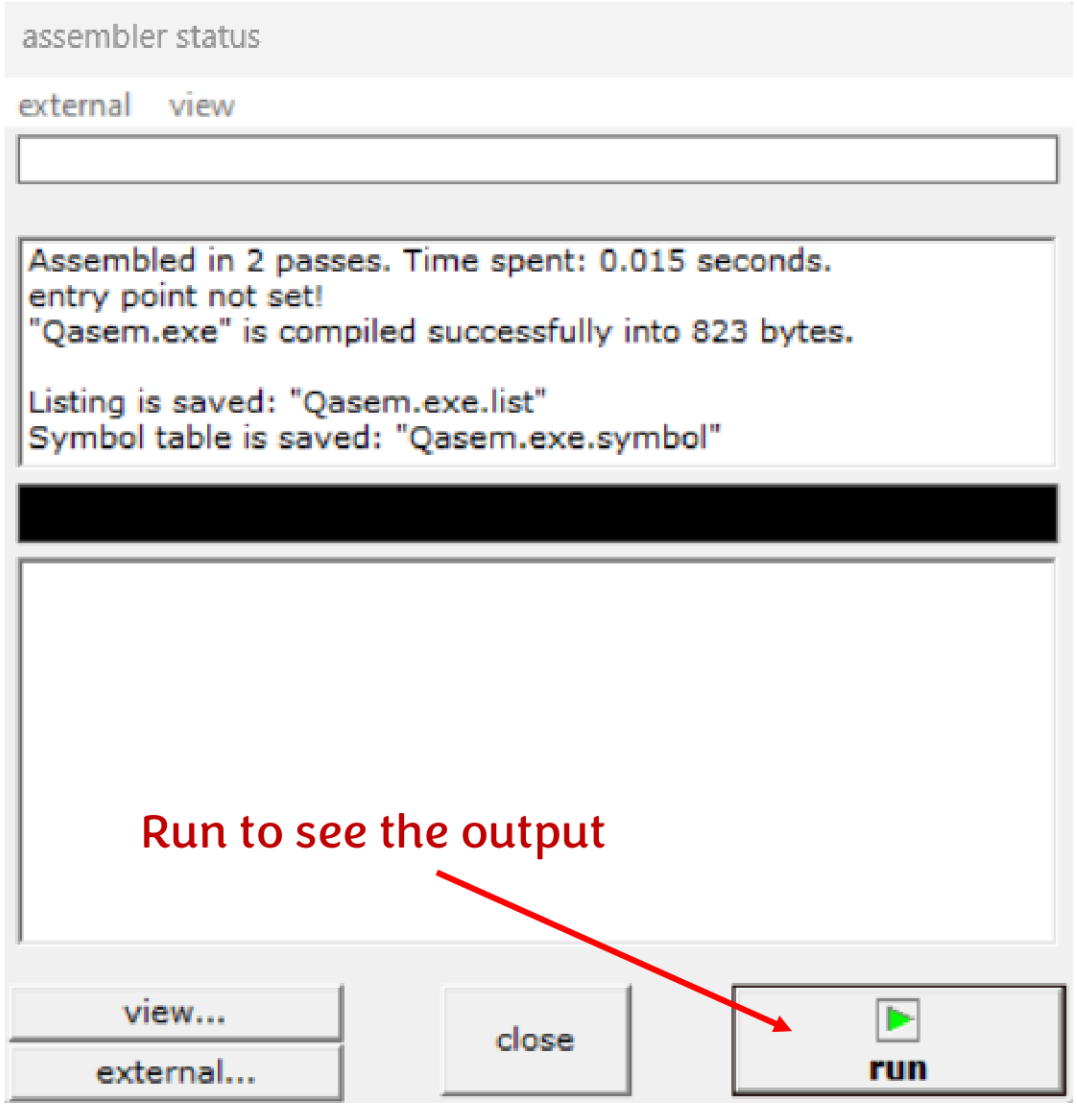
```
; multi-segment executable file template.

data segment
    ; add your data here!
    pkey db "press any key...$"
ends

stack segment
    dw   128  dup(0)
ends

code segment
start:
; set segment registers:
    mov ax, data
    mov ds, ax
    mov es, ax

    ; add your code here

    lea dx, pkey
    mov ah, 9
    int 21h        ; output string at ds:dx

    ; wait for any key....
    mov ah, 1
    int 21h

    mov ax, 4c00h ; exit to operating system.
    int 21h
ends

end start ; set entry point and stop the assembler.
```

Prepare your ASM code

Using MASM


Compile
emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
assembler status
external view
Save As
This PC › OS (C:) › emu8086 › MyBuild
Search MyBuild
Organize New folder
Desktop
Downloads
Documents
Pictures
Desktop Docun
My Slides
Round 1
Name
Date modified
Type
Size
No items match your search.
Save .exe
File name: Qasem.exe
Save as type: executable files (*.exe)
Hide Folders
Save
Cancel

Using MASM


assembler status
external
view
Assembled in 2 passes. Time spent: 0.015 seconds.
entry point not set!
"Qasem.exe" is compiled successfully into 823 bytes.
Listing is saved: "Qasem.exe.list"
Symbol table is saved: "Qasem.exe.symbol"
Run to see the output
view...
external...
close
run

Using MASM

The output screen

Using MASM

The exe file can be reversed:

disassembled using IDA Pro

or

debugged using OlyDbg

Thank you