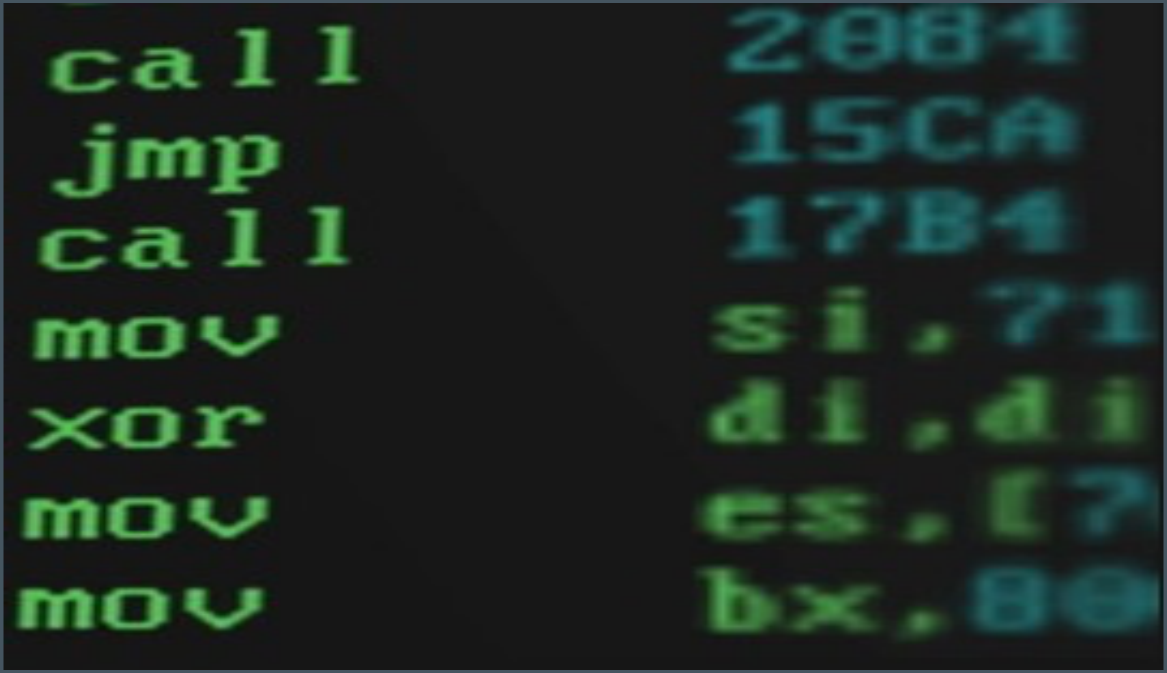
COMP6016 Introduction to Assembly Language Programming



```
call    2084
jmp     15CA
call    17B4
mov     si,71
xor     di,di
mov     es,[7
mov     bx,80
```

MUHAMMAD HILMI KAMARUDIN (MHK)

Module Aims

This module looks at low level programming tools and techniques for the creation, detection and defence against malware.

Students will examine a variety of reverse engineering techniques, at both the systems and network level, and they will be able to utilise them to obtain detailed information on malware. Anti-forensics tools will be considered and also, mechanisms for defeating them.

Lecturers and Location

Lecturers

- Muhammad Hilmi Kamarudin (MHK)

Where and When

- Lectures – Monday 13:00-14:00 OLS
- Practical – Monday 14:00-16:00 OLS

General Course Structure

- Lecture notes will generally be available from the web page
- Other sources of information will be put up there as needed
- This course is 100% coursework – the lectures are there to help you in the practical sessions
- Coursework details will be made available on the module website
- Coursework is subject to standard regulations on mitigating circumstances and academic misconduct

Background knowledge

Essential

- An ability to program in C/C++
- An understanding of IP packets and how they are transmitted
- An understanding of what an OS is

Useful

- Some experience of using an X Windows GUI
- Some experience of systems architecture

OXFORD BROOKES UNIVERSITY

Provisional Lecture Schedule

Week	Topic	Lecturer
1	Module Intro; Introduction to Assembler	MHK
2	Assembler II	MHK
3	Assembler III	MHK
4	Introduction to Malware	MHK
5	Question and Answer Session (Cwk 1 Due)	MHK
6	Command and Control analysis & capture	MHK
7	Reverse Engineering 1 (Static) Coursework 1 submission due	MHK
8	Reverse Engineering 2 (Dynamic)	MHK
9	Anti-reverse Engineering	MHK
10	Question and Answer Session	MHK
11	Revision	MHK
12	Coursework 2 submission and Demos	MHK

Health warning

Please restrict aggressive security assessment techniques to a secured environment

Some of the ideas and techniques described in this module can be used to compromise the security of other systems. To do so where you do not have permission of the system owner is unprofessional and possibly illegal. It will result in University disciplinary action being taken. Legal action may also be taken.

Lecture Learning Outcomes

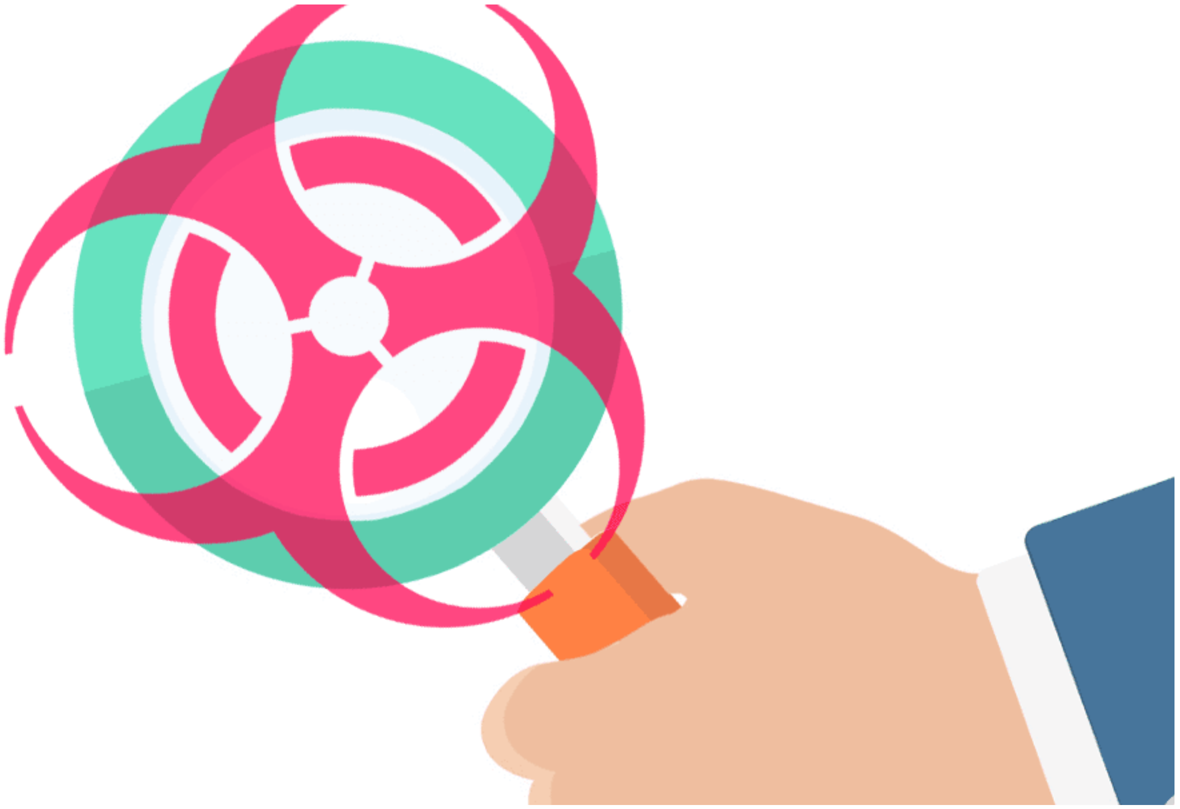
- By the end of today's lecture and practical you should:

- Understand the key components in a machine
- Understand common data representations
- Have an appreciation of the difference between Assembly language programming and higher level programming
- Be able to understand basic assembler language programmes

What is Assembly Language?

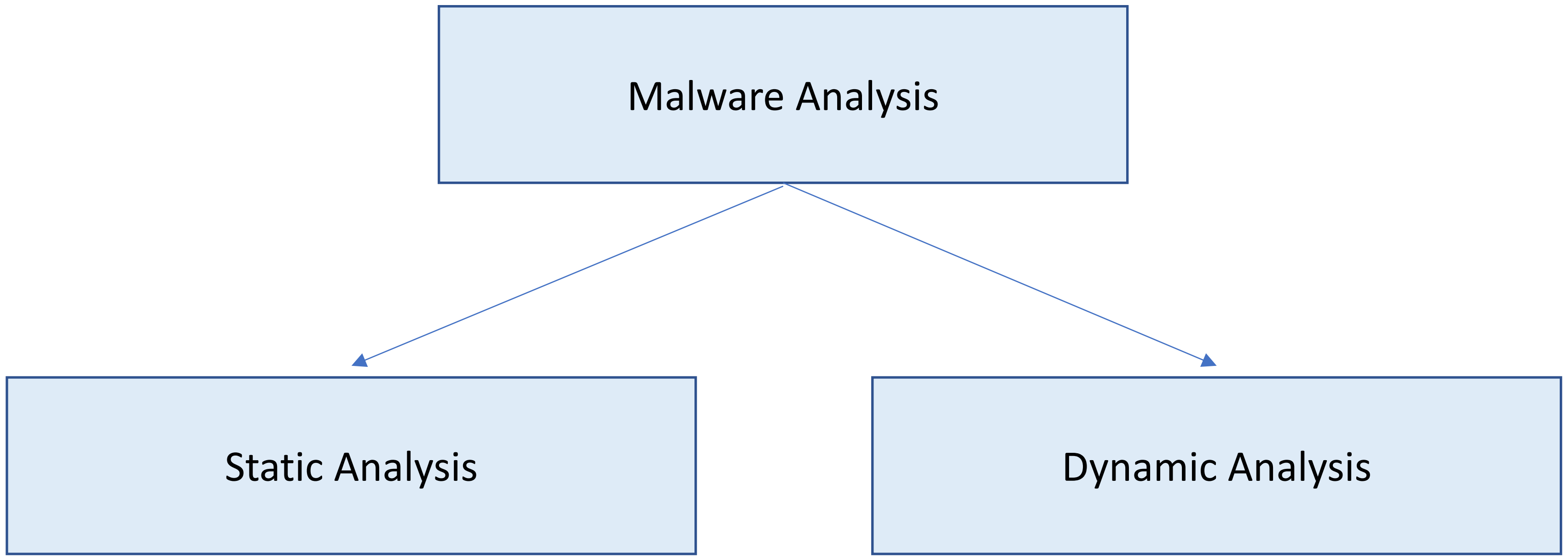
- Each computer has a microprocessor that manages the computer's arithmetical, logical, and control activities

- A processor only understands machine language instructions, which are strings of 1's and 0’s. Machine language is too obscure and complex for using in software development. The low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form



10 mins to figure out why we need to learn and understand assembly language for Malware Analysis?

Types of analysis



Disassembly is a specialized skill that can be daunting to those new to programming. But don’t be discouraged

Abstraction levels

Traditional computer architecture, a computer system can be represented as several *levels of abstraction* that create a way of hiding the implementation details.

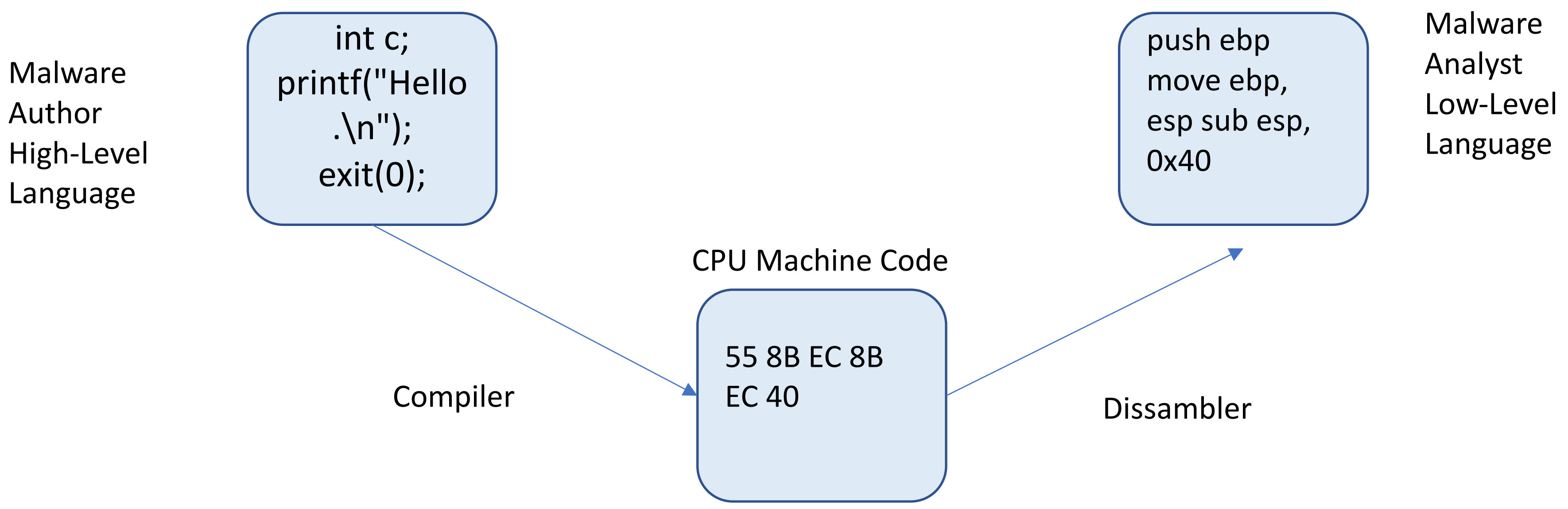


Figure 1: Code level examples

Malware authors create programs at the high-level language level and use a compiler to generate machine code to be run by the CPU.

Abstraction levels cont..

Hardware- consists of electrical circuits that implement complex combinations of logical operators such as XOR, AND, OR, and NOT gates, known as *digital logic*.

Microcode- Known as *firmware*. Microcode operates only on the exact circuitry for which it was designed. It contains microinstructions that translate from the higher machine-code level to provide a way to interface with the hardware.

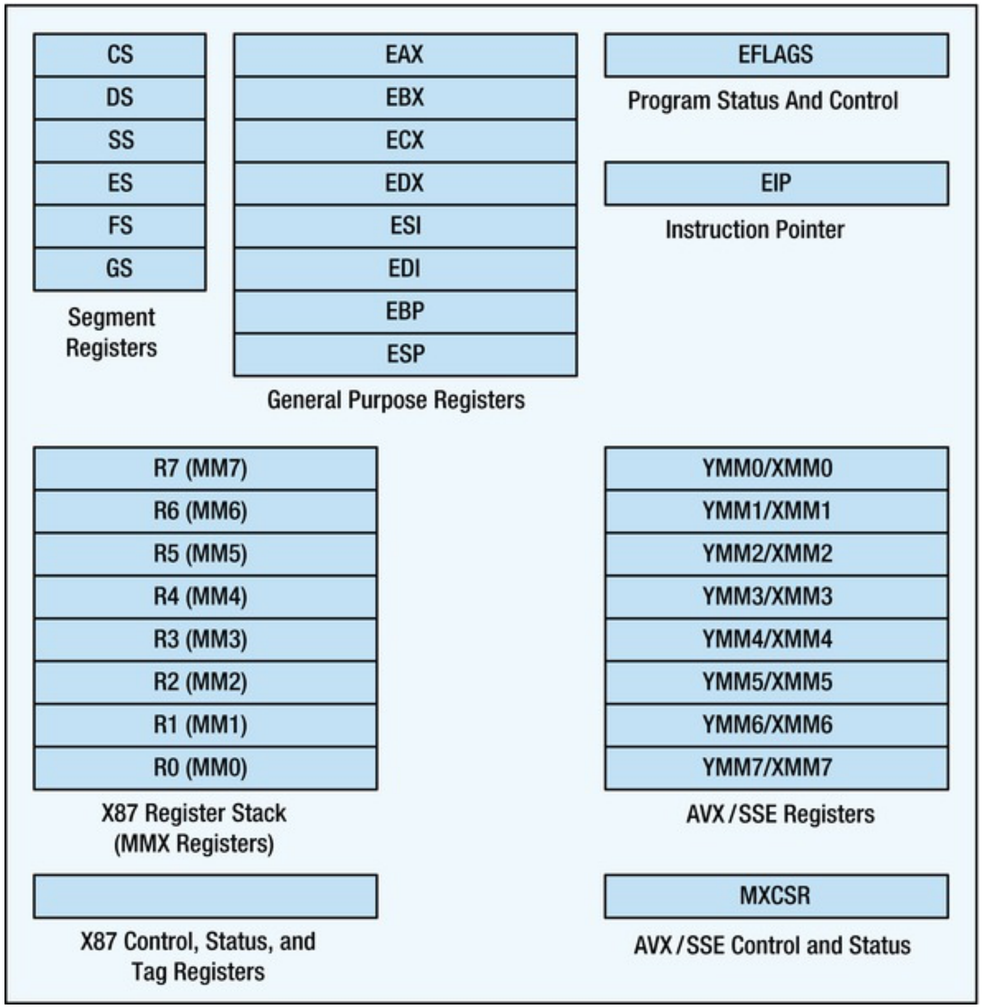
Machine Code- consists of *opcodes*, hexadecimal digits that tell the processor what you want it to do. Machine code is typically implemented with several microcode instructions so that the underlying hardware can execute the code.

Abstraction levels cont..

Low-level languages- A low-level language is a human-readable version of a computer architecture's instruction set. The most common low-level language is assembly language. Malware analysts operate at the low-level languages level because the machine code is too difficult for a human to comprehend.

High-level languages- Most computer programmers operate at the level of high-level languages. High-level languages provide strong abstraction from the machine level and make it easy to use programming logic and flow-control mechanisms.

Internal X86 CPU architecture



From: Kusswurm, D (2014) *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*

Main Memory

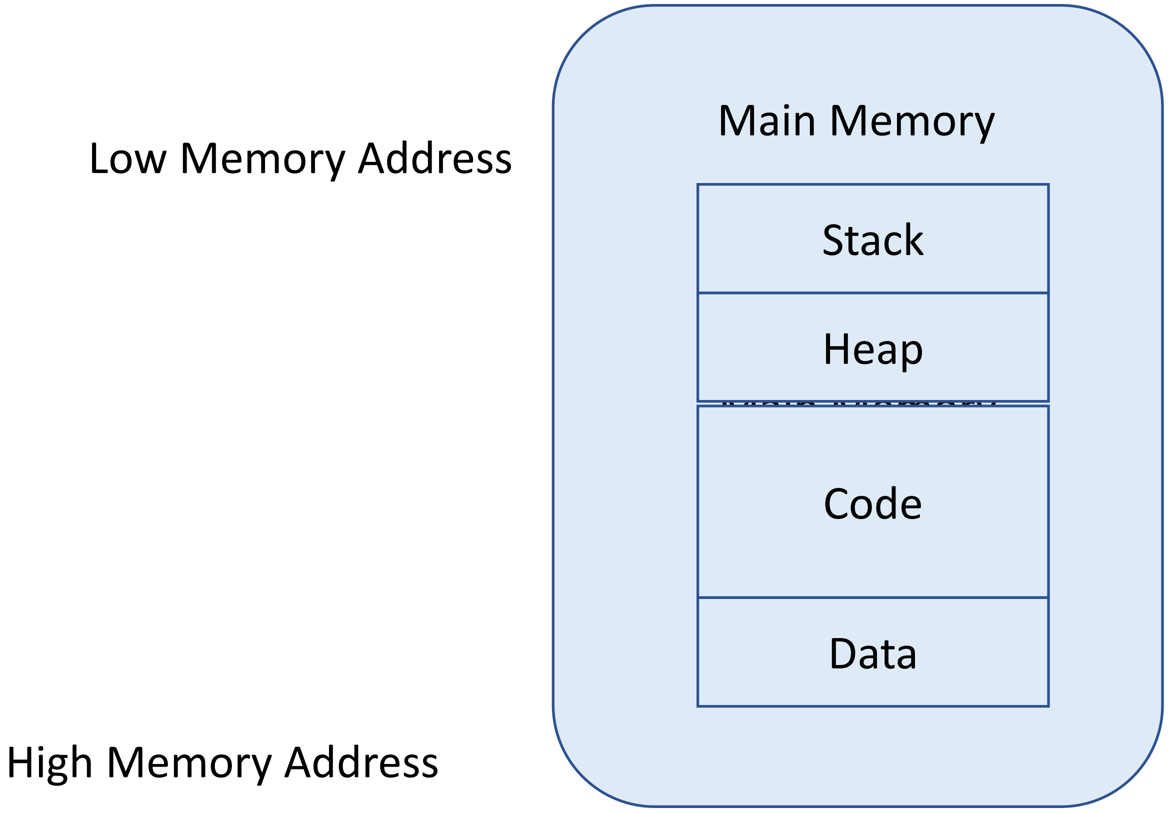


Figure 2: Basic memory layout for a program

Main Memory cont..

Data- This refer to a specific section of memory called the *data section*, which contains values that are put in place when a program is initially loaded.

Code- Code includes the instructions fetched by the CPU to execute the program’s tasks. The code controls what the program does and how the program’s tasks will be orchestrated.

Heap- The heap is used for dynamic memory during program execution, to create (allocate) new values and eliminate (free) values that the program no longer needs.

Stack- The stack is used for local variables and parameters for functions, and to help control program flow. We will cover the stack in depth later in this chapter.

Hexadecimal Number System

Decimal number	Binary representation	Hexadecimal representation
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Decimal to Binary

213 convert into binary=?

Binary to Decimal

11100011=?

Decimal to Hexadecimal

1128 convert into hexa=?

Hexadecimal to Decimal

1128 convert to decimal=?

Example

Binary number 1000 1100 1101 0001 is equivalent to hexadecimal -?

Hexadecimal number FAD8 is equivalent to binary - ?

Registers

- Segment registers – 6 registers that designate blocks of memory for code, data, and stack space. Normally handled by the OS and not accessed directly
- GP registers – 8 of 32 bit registers used for logical, arithmetic, and address calculations. Some can be addressed as multiple 8 bit or 16 bit registers. Convention restricts their use to specific functions (but this can be broken)
- EFLAGS register – status bits that contain the results of operations
- EIP (Instruction Pointer) – the offset of the next instruction to be executed

Registers - EFLAGS

Bit	Abbreviation	Description	Type
0	CF	Carry flag	Status
1		Reserved	
2	PF	Parity flag	Status
3		Reserved	
4	AF	Adjust flag	Status
5		Reserved	
6	ZF	Zero flag	Status
7	SF	Sign flag	Status
8	TF	Trap flag (single step)	Control
9	IF	Interrupt enable flag	Control
10	DF	Direction flag	Control
11	OF	Overflow flag	Status
12-13	IOPL	I/O privilege level (286+ only), always 1 on 8086 and 186	System
14	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System
15		Reserved, always 1 on 8086 and 186, always 0 on later models	
16	RF	Resume flag (386+ only)	System

Registers - EFLAGS

Bit	**Abbreviation**	**Description**	**Type**
17	VM	Virtual 8086 mode flag (386+ only)	System
18	AC	Alignment check (486SX+ only)	System
19	VIF	Virtual interrupt flag (Pentium+)	System
20	VIP	Virtual interrupt pending (Pentium+)	System
21	ID	Able to use CPUID instruction (Pentium+)	System
22-31		Reserved	

Writing Assembly Language Programmes

- Assembly language programming is inherently text based, but an IDE can help with more advanced programming
- Recommended Assembler – NASM (for Linux, Windows and MacOSX)
- Recommended IDE – SASM (Linux)
- Assembler is ***NOT*** self documenting code – make very liberal use of comments
- Assembler is easy to get wrong – do not be afraid to use a debugger

Basic Syntax- Assembly

An assembly program can be divided into three sections:
-The **data** section- is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.
The syntax for declaring data section is: section .data

-The **bss** section - is used for declaring variables. The syntax for declaring bss section is: section .bss

-The **text** section – is used for keeping the actual code. This section must begin with the declaration global main, which tells the kernel where the program execution begins.
The syntax for declaring text section is:

```
section .text
global main
main:
```

Basic Syntax- Assembly

Assembly language statements are entered one statement per line. Each statement follows the following format

```
[label] mnemonic [operands] [;comment]
```

The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands.

```
INC COUNT ; Increment the memory variable COUNT

MOV TOTAL, 48 ; Transfer the value 48 in the memory variable TOTAL

ADD AH, BH ; Add the content of the BH register into the AH register

AND MASK1, 128 ; Perform AND operation on the variable MASK1 and 128

ADD MARKS, 10 ; Add 10 to the variable MARKS

MOV AL, 10 ; Transfer the value 10 to the AL register
```

Anatomy of an assembly language programme

; means comment

This assembles
the code and generates
An object file

The linking stage
turns the object code
into an executable

```
;  hello.asm  a first program for nasm for Linux, Intel, gcc
;
; assemble: : nasm -g -I libasm_io-master/include/libasm_io.inc -f elf64 helloworld.asm
; link:          gcc -g helloworld.o -lasm_io -o helloworld
; run:           ./hello world
; output is:     Hello World

        SECTION .data              ; data section
hello:    db "Hello World!",10     ; the string to print, 10=next line

        SECTION .text              ; code section
        global main                ; make label available to linker
```

SECTION changes
where we are going
to put the code

Anatomy of an assembly language programme

This is the instruction

```
main:                            ; standard  gcc  entry point

        mov     edx,len          ; arg3, length of string to print
        mov     ecx,msg          ; arg2, pointer to string
        mov     ebx,1            ; arg1, where to write, screen
        mov     eax,4            ; write sysout command to int 80 hex
        int     0x80             ; interrupt 80 hex, call kernel

        mov     ebx,0            ; exit code, 0=normal
        mov     eax,1            ; exit command to kernel
        int     0x80             ; interrupt 80 hex, call kernel
```

This is a label.
Labels are optional but can be used to identify and refer to code blocks

These are operands

```
; Hello World
;Assemble: nasm -g -f elf64 -o <filename.o> <filename.asm>
;Link: gcc <filename.o> -no-pie -o <filename>

%include "/home/malware/asm/joey_lib_io_v6_release.asm"
global main

section .data

;a message to print when we enter main
;db stands for define bytes and the 0 at the end is the null terminator,
;so print_string knows when the string ends
hello: db "Hello World!",0
```

```
section .text


main:
		;set up a new stack frame, first save the old stack base pointer by pushing it
		push rbp

		;then slide the base of the stack down to RSP by moving RSP into RBP
		mov rbp, rsp

		;Windows requires a minimum of 32 bytes on the stack before calling any
other functions
		;so this is for compatibility, its perfectly valid on Linux and Mac also
		sub rsp, 32
```

```
        ;move the pointer to our hello message into RDI, RDI is the first parameter
to print string
        mov rdi, QWORD hello
        call print_string_new

        ;prints new line to the console
        call print_nl_new

        ;restore the stack frame of the function that called this function
        ;first add back the amount that we subtracted from RSP
        ;including any additional subtractions we made to RSP after the initial one
(just sum them)
        add     rsp, 32

        ;after we add what we subtracted back to RSP, the value of RBP we pushed
is the only thing left
        ;so we pop it back into RBP to restore the stack base pointer
        pop     rbp


        ret
```

Then, Modify the output to display:

My name is: xxxx
My ID: xxxx
University: xxxx

Instruction Operands

- Operands are used to specify the values that an instruction will act on.
- Most instructions require source operands plus a single destination operand. In most cases these must be specified, but in some cases the operands can be implicit
- In the case of two operand instructions, the first is normally the destination (where the results is going to be stored) and the second is the source (a value to be stored, or the address at which the value can be found)
- There are 3 classes of operands -
- Immediate operands – a constant value or expression that can be immediately evaluated
- Register operands – the value is contained in a GPR
- Memory operands – a memory address to be used to retrieve or store the information. NB either source or destination can be a memory operand, but not both

Instruction Operands

Type	Example	Equivalent C/C++ Statement
Immediate	mov eax,42	eax = 42
	imul ebx,11h	ebx *= 0x11
	xor dl,55h	dl ^= 0x55
	add esi,8	esi += 8
Register	mov eax,ebx	eax = ebx
	inc ecx	ecx += 1
	add ebx,esi	ebx += esi
	mul ebx	edx:eax = eax * ebx
Memory	mov eax,[ebx]	eax = *ebx
	add eax,[val1]	eax += *val1
	or ecx,[ebx+esi]	ecx |= *(ebx + esi)
	sub word ptr [edi],12	*(short*)edi -= 12

From: Kusswurm, D (2014) *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*

Variables & constants

- Assembler does not have conventional variables, instead we allocate memory and optionally initialise it
- Allocating memory is done via labels – the label becomes, in effect, the variable name
- Memory can then be initialised, or left unitialised

db	0x55	; a byte with value 0x55
db	0x55,0x56,0x57	; three bytes in succession
db	'a',0x55	; two byes, character constants are OK
db	'hello',13,10,'$'	; so are string constants
dw	0x1234	; create memory in word sized chunks and initialise ; it to 0x34 0x12
dw	'a'	; 0x61 0x00 (it's just a number)
dw	'ab'	; 0x61 0x62 (character constant)
dw	'abc'	; 0x61 0x62 0x63 0x00 (string)
dd	0x12345678	; allocate memory in double word chunks and set ; it to 0x78 0x56 0x34 0x12

Variables & constants

- Uninitialised memory -

```
buffer:          resb    64              ;
reserve 64 bytes
wordvar:         resw    1               ;
reserve a word
Realarray:        resq    10             ;
array of ten reals
```

- Constants can be defined using the equ directive

```
Maxstudents          equ        100
```

Key instructions – data transfer

- mov – copy data to/from a GPR or memory location to/from a GPR or memory location. (Remember you cannot move from a memory location to a memory location). The mov instruction will not change any flags

```
        mov     ecx,msg             ; copy the value of msg to ecx
        mov     ebx,1                  ; set ebx to the value 1
```

- push – push a GPR, memory location or immediate value onto the stack
- pop – take the top most item off the stack and store it in the specified GPR or memory location

```
       push    eax   ; push the value stored in eax onto the stack
       pop     ebp   ; take the top value from the stack and store
it in ebp
```

- xchg – swap data between two GPRs or a GPR and a memory location

```
       xchg    ax, bx       ; Put AX in BX and BX in AX
       xchg    memory, ax   ; Put "memory" in AX and AX in "memory"
```

Key instructions – binary arithmetic

- add – adds the source and destination operand, result is stored in the destination. Can be used for both signed and unsigned integers and will clear flags
- sub - subtracts the source from destination operand, result is stored in the destination. Can be used for both signed and unsigned integers and will clear flags

```
        add eax, '0'            ; add the ASCII code of '0' to eax
        sub eax, eax            ; zero the eax register
```

- adc – add the source, destination and carry flag state
- sbb – subtract the sum of the source operand and carry flag from the destination operand

```
        adc eax, '0'            ; add the ASCII code of '0' and the
carry flag
                                ; state to eax
        sub eax, 10            ; eax = eax - (10 + carry flag state)
```

Key instructions – binary arithmetic

- imul – perform a signed multiplication between two operands. Destination can be omitted (in which case AL, AX, or EAX will be used) or specified.

```
      imul 10          ; multiply AL by 10 and store the result in AL
```

- mul – perform an unsigned multiplication between the source operand and the AL, AX, or EAX register. Result is stored in AX, DX:AX, or EDX:EAX registers

```
      mul 80000          ; multiply AL by 80000 and store the result in DX:AX
```

- idiv – perform a signed division using AX, DX:AX, or EDX:EAX as the dividend and the source operand as the divisor. Quotient and remainder are saved in register pair AL:AH, AX:DX, or EAX:EDX.

```
      idiv 10          ; divide AX by 10 and store the result in AL:AH
```

- div – perform an unsigned division using AX, DX:AX, or EDX:EAX as the dividend and the source operand as the divisor. Quotient and remainder are saved in register pair AL:AH, AX:DX, or EAX:EDX.

```
      div 10          ; divide AX by 10 and store the result in AX
```

Key instructions – binary arithmetic

- inc – increment the specified operand, doesn’t affect the carry flag
- dec – decrement the specified operand, doesn’t affect the carry flag
- neg – compute the two’s complement value of the operand

Key instructions – data comparison

- cmp – compare the two operands by subtracting the source operand from the destination and then set the status flags. The results of the subtraction are discarded.

```
        cmp ecx, edx            ; compare the values in ecx and edx
```

- cmpxchg – compares the contents of register AL, AX, or EAX with the destination operand and performs an exchange based on the results. (i.e. compare AL/AX/EAX with destination operand, if they are equal, copy the source operand into the destination operand

```
        cmpxchg [foo], edx              ; compare the value at the address
foo
                                        ; with EAX and if they are equal,
put the
                                        ; value of edx into the memory
address
                                        ; foo
```

Key instructions – logical operators

- and, or, xor – calculate the bitwise AND, OR, or XOR respectively of the source and destination operands.
- not – calculate the 1’s compliment of the specified operand, does not affect status flags
- test – calculate the bitwise AND of the source and destination operands and discard the results (essentially set the status flags without changing any values)

IO

- IO is very problematic in assembler, in the early days we could read directly from the keyboard and write directly to the screen.
- Text output will depend on the OS used and whether or not it is 64 or 32 bit
- 32 bit OSes normally work by setting up registers and raising an interupt
- 64 bit OSes normally work by setting up registers and calling a system call
- Input is even more problematic!
- In most cases, we will use an existing library. For this module we recommend libasm_io library at - https://github.com/EriHoss/libasm_io (a copy is on the module website)

IO

This is tells the linker
Where the code starts

```
; ----------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ----------------------------------------------------------------------------------------

        global  _start

        section .text
_start:
        ; write(1, message, 13)
        mov     rax, 1                  ; system call 1 is write
        mov     rdi, 1                  ; file handle 1 is stdout
        mov     rsi, message            ; address of string to output
        mov     rdx, 13                 ; number of bytes
        syscall                         ; invoke operating system to do the write

        ; exit(0)
        mov     eax, 60                 ; system call 60 is exit
        xor     rdi, rdi                ; exit code 0
        syscall                         ; invoke operating system to exit
message:
        db      "Hello, World", 10      ; note the newline at the end
```

This is the 64bit version of EAX. We put the system call number here.

This tells the system to perform the specified system call