SEC204

Computer Architecture and Low Level Programming

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk

Website: https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Date

28/10/2019

School of Computing

(University of Plymouth)

Outline

- x86 Assembly
  - Why use assembly?
  - Basic concepts
  - Different ways of using assembly

Main reasons for using assembly nowadays

- Understand how hardware works
  - This way, we can write more efficient software in terms of execution time, memory size, energy consumption and security
  - Reverse engineering to identify software flaws
- Making compilers, hardware drivers, processors
- Optimization
  - execution time
  - memory size
  - energy consumption

Main reasons for NOT writing assembly nowadays

- □ Development time
- □ Reliability and security
- □ Debugging
- □ Maintainability
- □ Portability

X86, X64 and IA-32

- What is **x86** and what **x64**?
  - **x86** is an Intel CPU architecture that originated with the 16-bit 8086 processor in 1978.
  - Today, the term "**x86**" is used generally to refer to any 32-bit processor compatible with the x86 instruction set
  - **IA-32** (short for "Intel Architecture, 32-bit", sometimes also called i386 is the 32-bit version of the **x86** instruction set architecture
  - **x86-64** or x64 is the general name of a series of 64-bit processors and their associated instruction set architecture. These processors are compatible with **x86.**
- What 32bit mean?
  - 32bit Data/address bus, registers, …

Introduction to x86 Assembly Programming

- There are many different assemblers out there: MASM, NASM, GAS, AS86, TASM, A86, Terse, etc. All use radically different assembly languages.
- There are differences in the way you have to code for Linux, Windows, etc.
- GNU Assembler (GAS)
  - AT&T syntax for writing the assembly language
- Microsoft Macro Assembler (MASM)
- Netwide Assembler (NASM)

Pillars of assembly language

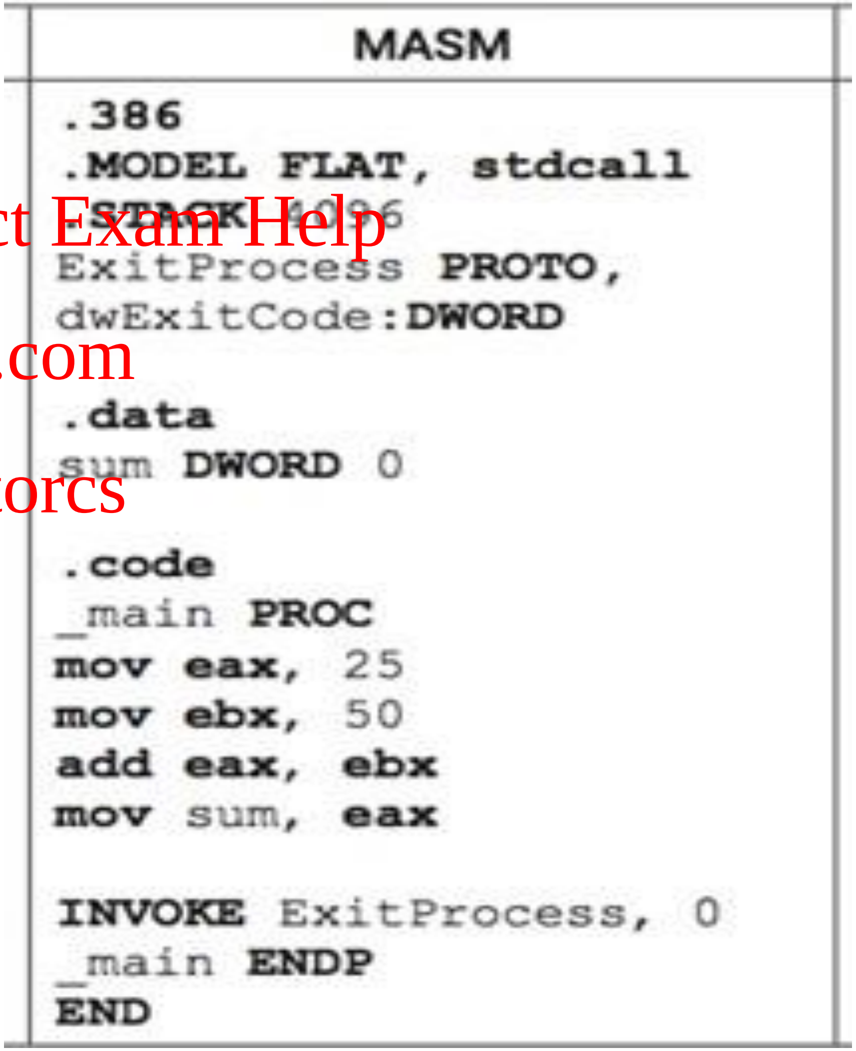
- □ Reserved words
- □ Identifiers
- □ Directives
- □ Sections (or segments)
- □ Instructions

Reserved Words

- Predefined purpose, e.g. mov is a reserved word and an instruction

- These cannot be used in any other way, e.g. for variable names

- **Case-insensitive:** Mov ≡ mov ≡ MOV

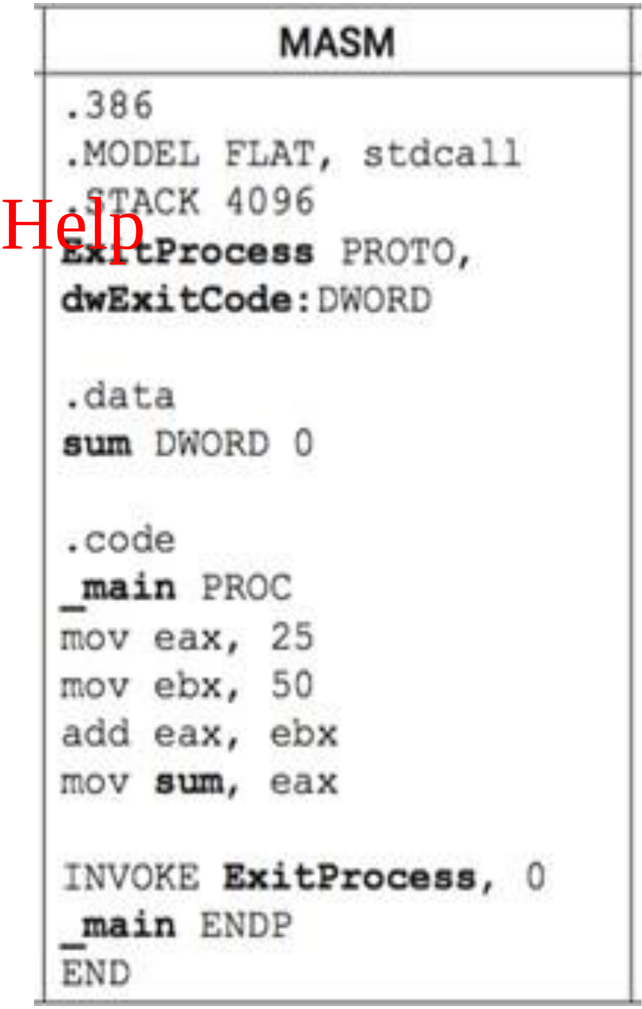


MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
sum DWORD 0

.code
_main PROC
mov eax, 25
mov ebx, 50
add eax, ebx
mov sum, eax

INVOKE ExitProcess, 0
_main ENDP
END
```

Identifiers

- **Programmer defined names given to items** such as variables, constants and procedures
- Length is limited to 247 characters
- Must begin with a letter (A-Z, a-z), underscore, question mark (?), at symbol (@) or dollar symbol ($)
- Please do not use: question mark (?), at symbol (@) or dollar symbol ($)
- Use camelCase for variables, e.g. sumOfProducts
- Use CamelCase for procedures, e.g. ExitProcess
- Use CONSTANT NAME for constants, e.g. GRAVITIONAL ACCELERATION

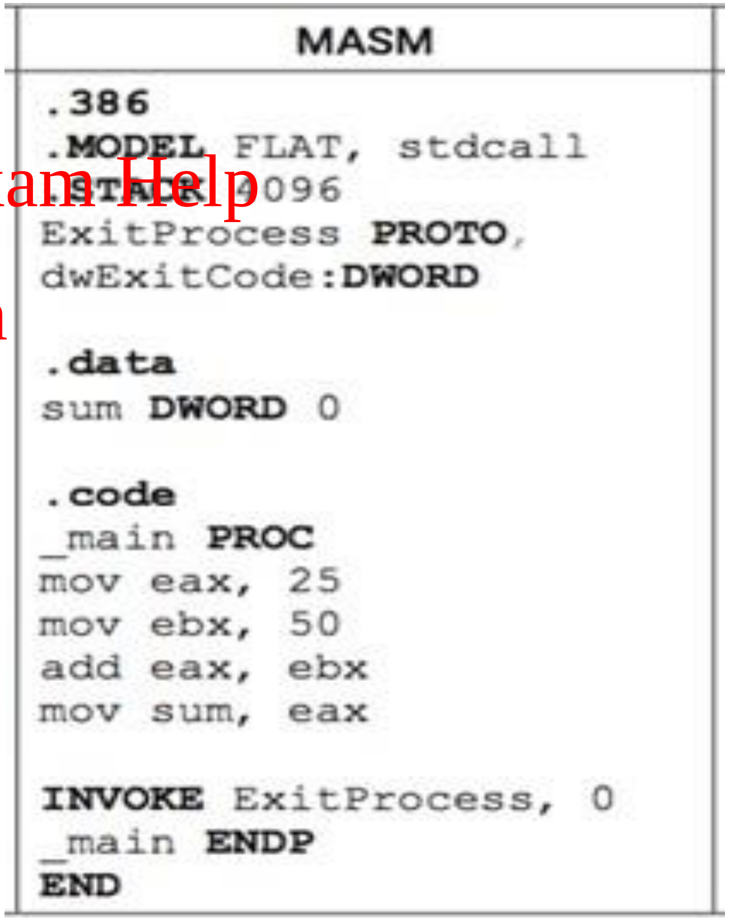


MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
sum DWORD 0

.code
_main PROC
mov eax, 25
mov ebx, 50
add eax, ebx
mov sum, eax

INVOKE ExitProcess, 0
_main ENDP
END
```

Directives

- **Assembler specific commands: direct the assembler to do something**
- Example: ask the assembler to reserve 32-bit memory with literal value 42 in a variable called *answer* with DWORD directive. Code: *answer DWORD 42*
- **Other useful directives:**
  - ***.386*** Enables 80386 processor instructions
  - ***.model*** Sets the memory model. FLAT for 32-bit instructions, and stdcall for assembly instructions
  - ***.stack*** Sets the size of the stack memory segment for the program

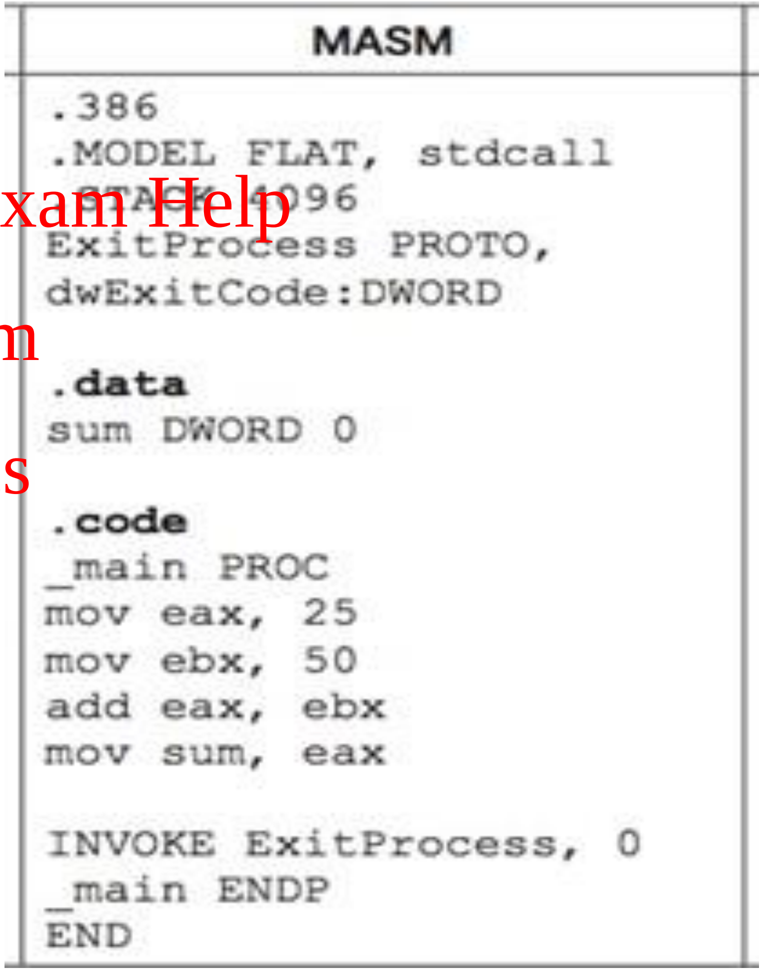


MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
sum DWORD 0

.code
_main PROC
mov eax, 25
mov ebx, 50
add eax, ebx
mov sum, eax

INVOKE ExitProcess, 0
_main ENDP
END
```

Program sections (or segments)

- Special sections pre-defined by the assembler
- Common segments:
  - *.data* uninitialised and initialised variables
  - *.code* executable code and instructions



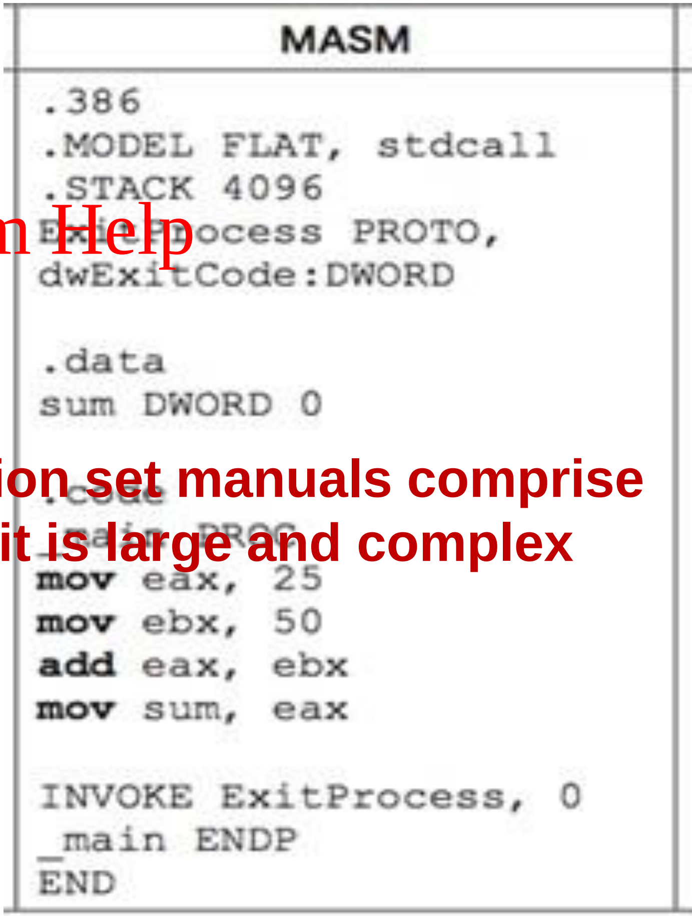
MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
sum DWORD 0

.code
_main PROC
mov eax, 25
mov ebx, 50
add eax, ebx
mov sum, eax

INVOKE ExitProcess, 0
_main ENDP
END
```

Instructions

- **Executable statements in a program**
- **Two basic parts:** *mnemonic and [operands]*
- *Mnemonic* is the instruction name as defined in the architecture's instruction sets
- Some do not require operands, some one or more
- Common code examples:
  - stc no operands sets the carry flag inc eax increment eax by one
  - mov eax, 5 moves literal value 5 to eax register

Intel's x86 instruction set manuals comprise over 2900 pages – it is large and complex

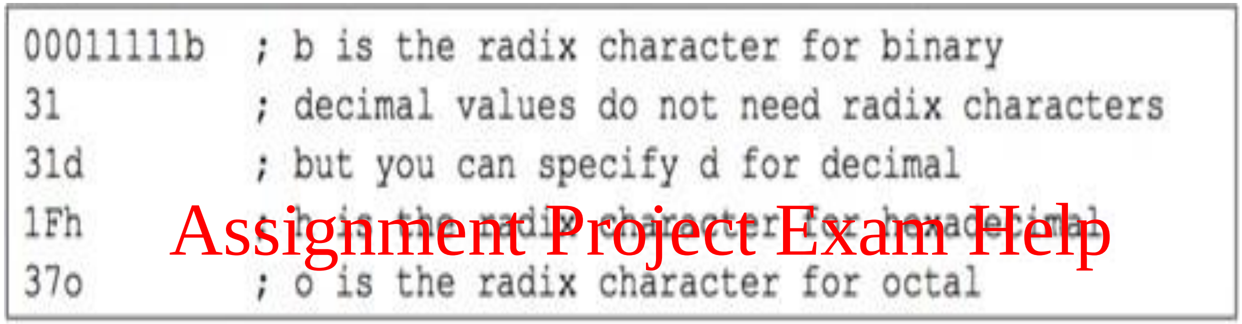


MASM

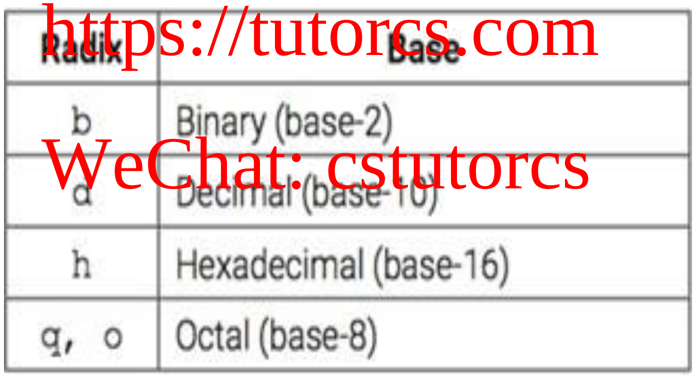
```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
sum DWORD 0

.code
_main PROC
mov eax, 25
mov ebx, 50
add eax, ebx
mov sum, eax

INVOKE ExitProcess, 0
_main ENDP
END
```

Label:	Mnemonic	Operand(s)	;Comment

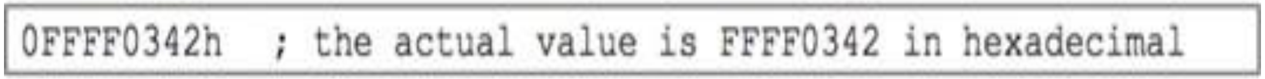
Literals



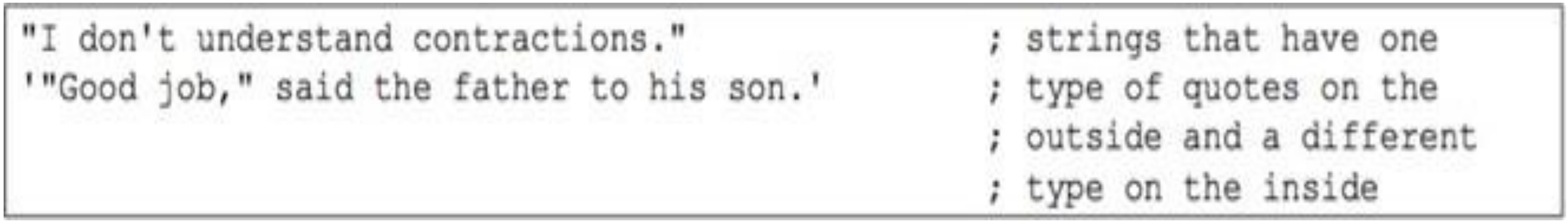
```
00011111b   ; b is the radix character for binary
31          ; decimal values do not need radix characters
31d         ; but you can specify d for decimal
1Fh         ; h is the radix character for hexadecimal
37o         ; o is the radix character for octal
```



Radix	Base
b	Binary (base-2)
d	Decimal (base-10)
h	Hexadecimal (base-16)
q, o	Octal (base-8)

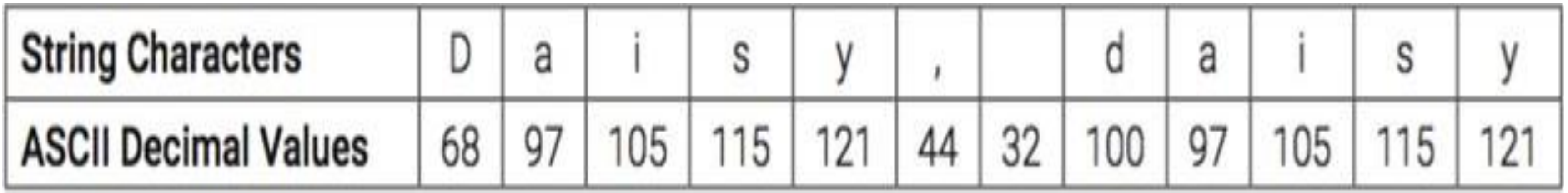


```
0FFFF0342h   ; the actual value is FFFF0342 in hexadecimal
```

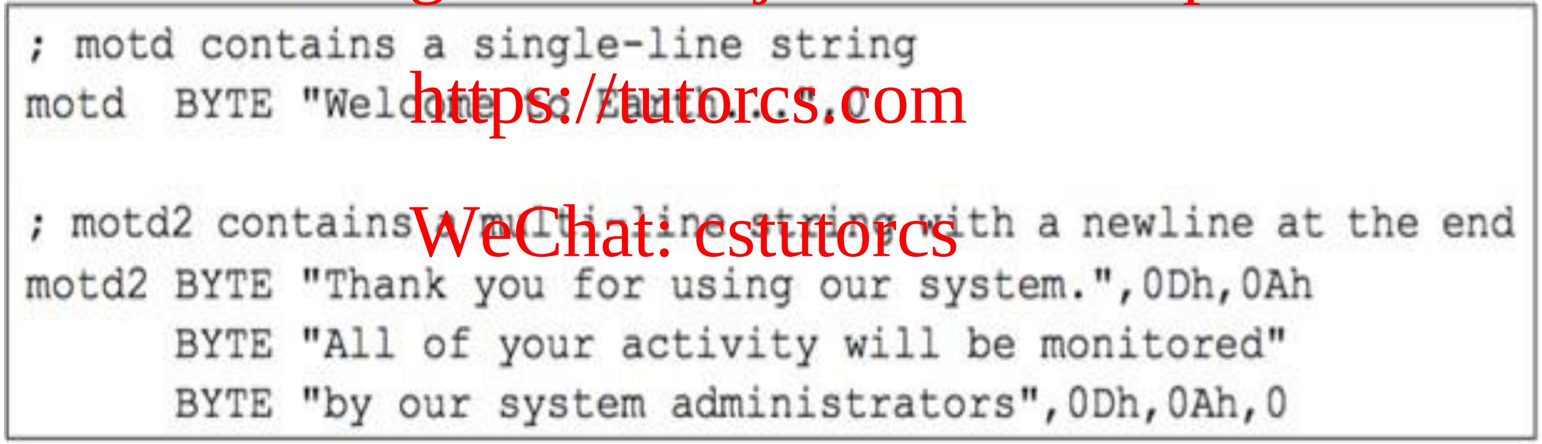


```
"I don't understand contractions."                ; strings that have one
'"Good job," said the father to his son.'          ; type of quotes on the
                                                   ; outside and a different
                                                   ; type on the inside
```

String Literals



String Characters	D	a	i	s	y	,		d	a	i	s	y
ASCII Decimal Values	68	97	105	115	121	44	32	100	97	105	115	121



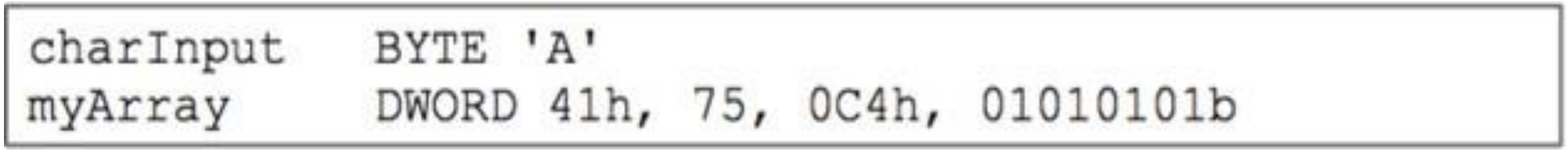
```
; motd contains a single-line string
motd  BYTE "Welcome to Earth...",0

; motd2 contains a multi-line string with a newline at the end
motd2 BYTE "Thank you for using our system.",0Dh,0Ah
      BYTE "All of your activity will be monitored"
      BYTE "by our system administrators",0Dh,0Ah,0
```

- □ Stored as Byte array, each character occupies one byte
- □ Must end with ‘0’
- □ Carriage return: ‘0Dh’
- □ Line-feed: ‘0Ah’

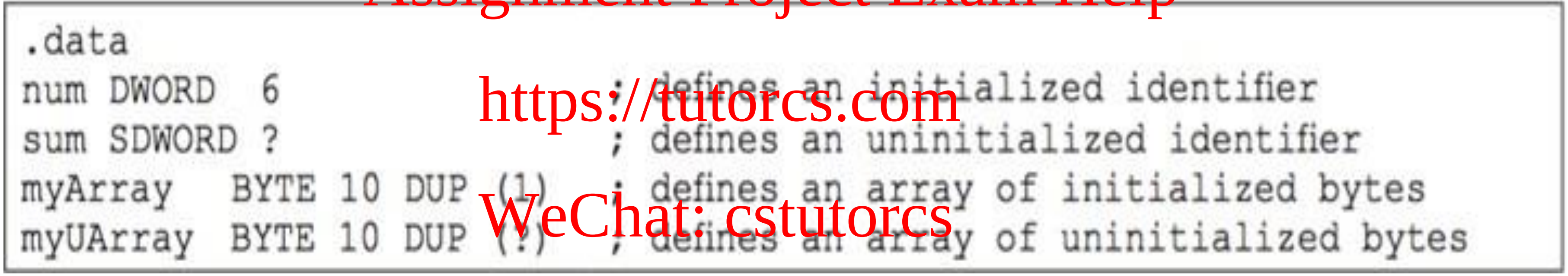
Data Types

- BYTE – 8bit unsigned integer
- SBYTE – 8bit signed integer
- WORD - 16bit unsigned integer
- SWORD - 16bit signed integer
- DWORD - 32bit unsigned integer
- SDWORD - 32bit signed integer
- QWORD – 64bit unsigned integer
- REAL4 – single precision floating point numbers (32bit)
- REAL8 - double precision floating point numbers (64bit)

Variables



```
charInput    BYTE 'A'
myArray      DWORD 41h, 75, 0C4h, 01010101b
```



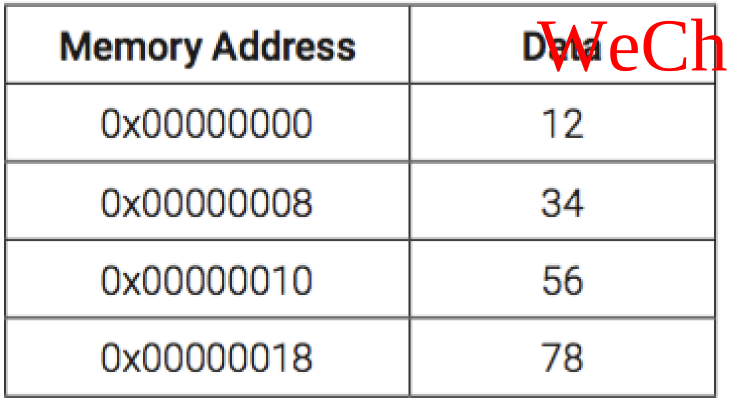
```
.data
num DWORD  6                   ; defines an initialized identifier
sum SDWORD ?                   ; defines an uninitialized identifier
myArray   BYTE 10 DUP (1)      ; defines an array of initialized bytes
myUArray  BYTE 10 DUP (?)      ; defines an array of uninitialized bytes
```

myArray BYTE 10 DUP (1) ; duplicates 1 into the 10-bytes

Storage methods:
Little Endian vs Big Endian

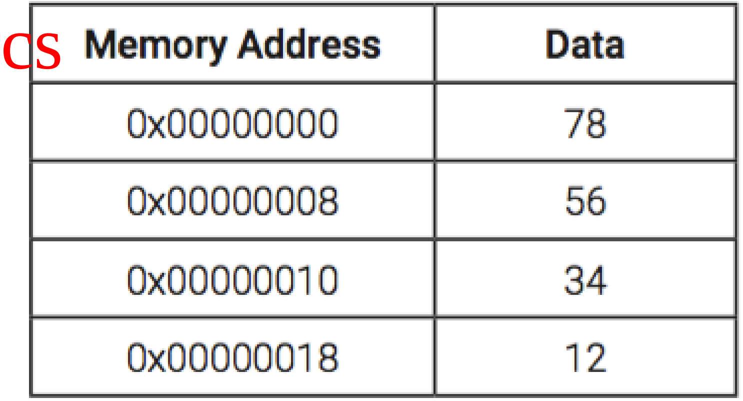
- x86 and x86 64 typically use *Little-Endian*, i.e., **all the bytes are stored in reverse order** (the bits inside a bit are stored normally)
- Store 12345678_{16} in memory

Big-Endian



Memory Address	Data
0x00000000	12
0x00000008	34
0x00000010	56
0x00000018	78

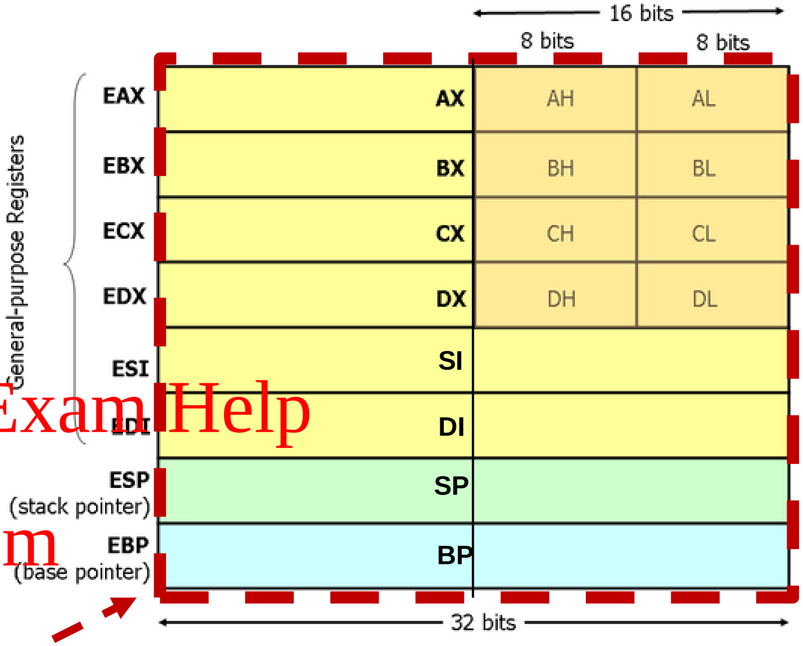
Little-Endian



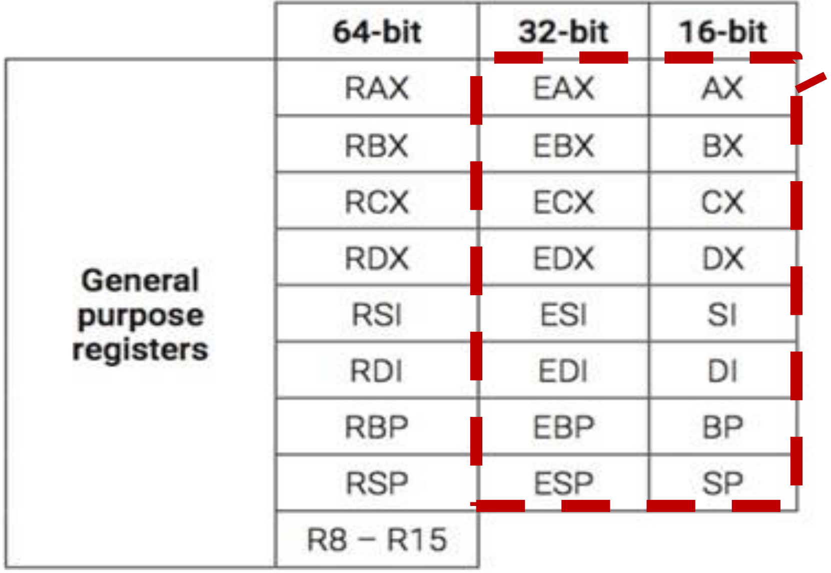
Memory Address	Data
0x00000000	78
0x00000008	56
0x00000010	34
0x00000018	12

Registers (1)

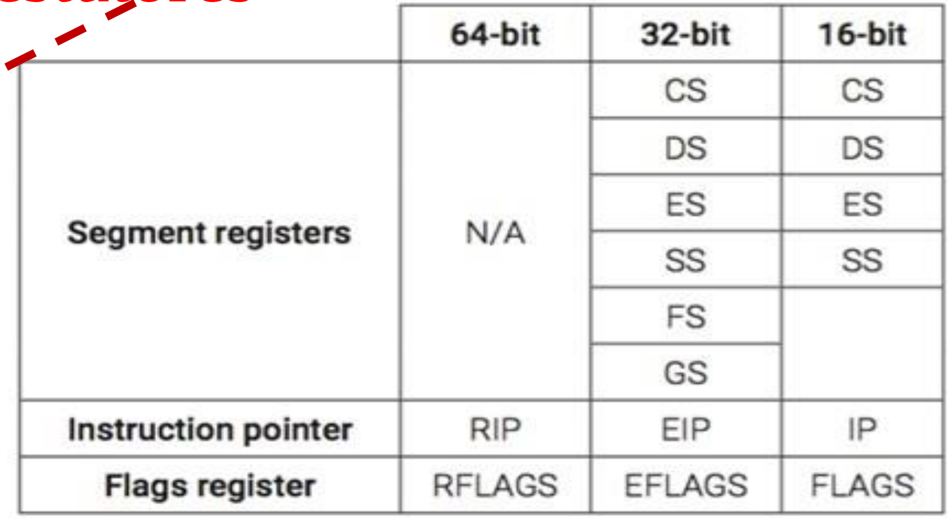
- The lower bytes of some of these registers may be accessed independently as 32, 16 or 8-bit registers
- Older processors use 8bit, 16bit or 32bit registers only – compatibility exists
- There are other registers too… (next slide)



EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
ESP (stack pointer)	SP		
EBP (base pointer)	BP		



	64-bit	32-bit	16-bit
General purpose registers	RAX	EAX	AX
	RBX	EBX	BX
	RCX	ECX	CX
	RDX	EDX	DX
	RSI	ESI	SI
	RDI	EDI	DI
	RBP	EBP	BP
	RSP	ESP	SP
	R8 – R15		



	64-bit	32-bit	16-bit
Segment registers	N/A	CS	CS
		DS	DS
		ES	ES
		SS	SS
		FS	
		GS	
Instruction pointer	RIP	EIP	IP
Flags register	RFLAGS	EFLAGS	FLAGS

Registers (2)

- There are also **eight 80bit floating point registers**
  - ST(0)-ST(7), arranged as a stack
- **Eight 64bit MMX vector registers**
  - Used with MMX instructions (physically they are the same as above)
- **Eight/Sixteen 128/256/512 bit vector registers**
  - 128bit use SSE instructions
  - 256bit use AVX instructions
  - 512bit use AVX2 instructions

Registers (3)

- ***rax/eax***: Default accumulator register.
  - Used for arithmetical operations
  - Function calls place return value.
  - Do not use it for data storage while performing such operations.
- ***rcx/ecx:*** Hold loop counter. Do not overwrite when looping!
- ***rbp/ebp***: Reference data on the stack; more on this later.
- ***rsp/esp***: Used for managing the stack – typically points to the top of the stack.
- ***rsi/esi*** and ***rdi/edi***: Index registers used in string operations.
- ***rip/eip:*** Instruction pointer - shows next instruction to be executed
- ***rflags/eflags***: Status and control registers; cannot be modified directly!

Notations

L A literal value (e.g. 42)

M A memory (variable) operand (e.g. numOfStudents)

R A register (e.g. eax)

□ If you see a number followed by one of these notations, it represents the size of the notation. For instance, L8 means that it is a 8-bit literal value.

□ If multiple notations appear segregated by a slash ('/'), it means that either of these two types may be used. For example, M/R means that either a memory type of a register may be used.

Data movement

- *mov eax, sum ; mov M/R, L/M/R (moving)*
- *xchg eax, sum ; xchg M/R, M/R (swapping)*

- For moving data:

  - Both operands must be the same size.

  - Both operands cannot be memory operands (must use a register as an intermediary).

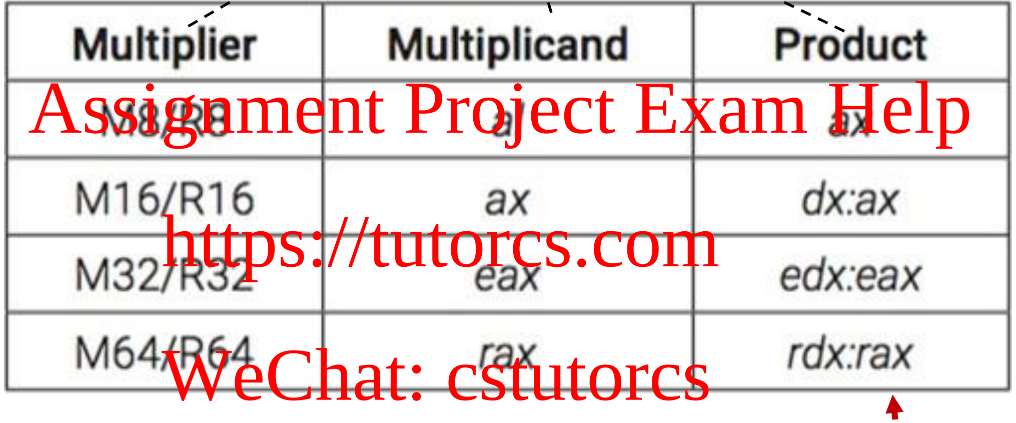
Addition and subtraction

- □ *inc sum ; inc M/R (increment by one)*
- □ *dec sum ; dec M/R (decrement by one)*
- □ *add eax, sum ; add M/R, L/M/R (addition)*
- □ *sub eax, val ; sub M/R, L/M/R (subtraction)*
- □ *neg sum ; neg M/R (negate: 2's complement), this operation is equivalent to subtracting the operand from 0*

- □ In MASM, for addition and subtraction, the second component is added/subtracted from the first component, and the result is stored back into the first component.
- □ In AT&T the exact opposite

MUL (unsigned multiply)

2 x 3 =6

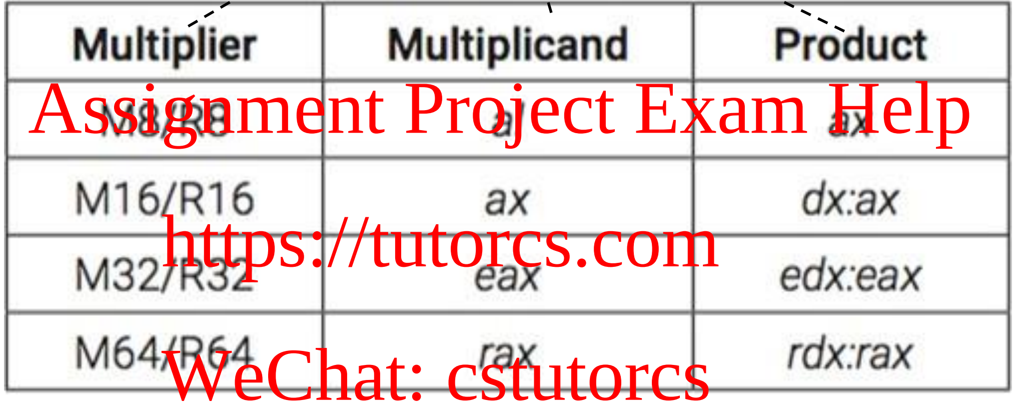


Multiplier	Multiplicand	Product
M8/R8	al	ax
M16/R16	ax	dx:ax
M32/R32	eax	edx:eax
M64/R64	rax	rdx:rax

- □ Multiplication may require more bytes to hold the results. Consider the following 2-bit multiplicand 310 (112) and 2-bit multiplier 310 (112). The product is 910 (10012), and it cannot be contained in 2-bits; it requires 4-bits. At most we require double the size of the multiplier or the multiplicand.
- □ Also, note that the parts of the product are saved in *high:low* format.

MUL - example

2 x 3 =6



Multiplier	Multiplicand	Product
M8/R8	al	ax
M16/R16	ax	dx:ax
M32/R32	eax	edx:eax
M64/R64	rax	rdx:rax

```
.data
var1  WORD 3000h
var2 WORD 100h

.code ; 16bit multiplication
mov ax,var1
mul var2 ; DX:AX = 00300000h, CF=1
```

CF=1 as DX contains non zero data

```
.data
var1 DWORD 3000h
var2 DWORD 100h

.code ; 32bit multiplication
mov eax,var1
mul var2 ; EDX:EAX = 0000000000300000h, CF=0
```

CF=0 as EDX is zero

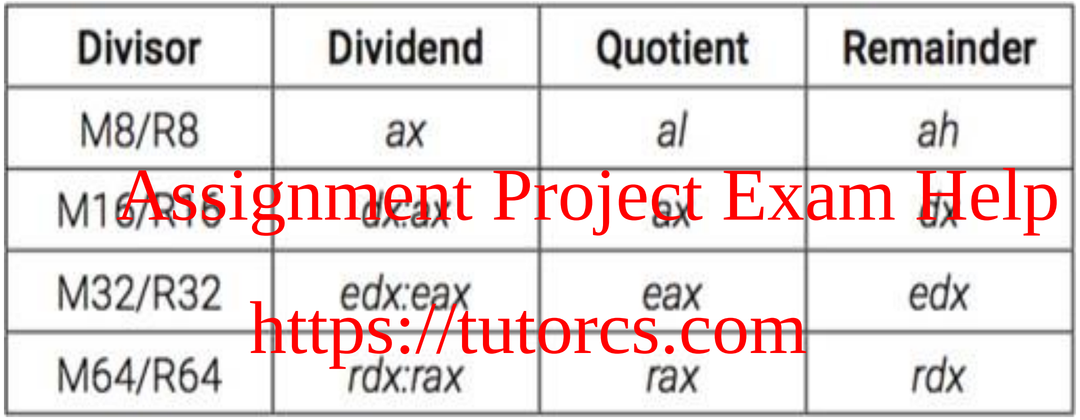
IMUL – signed multiply

- ***imul*** is similar to ***mul***
- However:
  - It preserves the sign of the product by sign-extending it into the upper half of the destination register
  - It sets OF flag to ‘1’ when the less significant register cannot store the result (including its sign)

```
.data
var1 BYTE 48 ; this is decimal
var2 BYTE 4  ; this is decimal

.code ; 8bit multiplication
mov al,var1
mul var2 ; AH:AL = 00C0h, OF=1
```

OF=1 as 8bits are not enough to hold the signed number $C0_{16}$ ($0\ 1100\ 0000_2$). A ‘0’ is needed in AH to hold the sign

DIV (Unsigned Divide)



Divisor	Dividend	Quotient	Remainder
M8/R8	ax	al	ah
M16/R16	dx:ax	ax	dx
M32/R32	edx:eax	eax	edx
M64/R64	rdx:rax	rax	rdx

```
.code ; 16bit division
 mov dx,0h          ; clear dividend, high
 mov ax,8003h   ; dividend, low
 mov cx,100h      ; divisor
 div cx               ; AX = 0080h, DX = 3
```

```
.code ; 32bit division
 mov edx,0 ; clear dividend, high
 mov eax,8003h ; dividend, low
 mov ecx,100h ; divisor
 div ecx ; EAX = 0000 0080h, EDX = 3
```

Different Ways of writing Assembly

- There are 3 ways to write assembly
  - **Use Assembler**
    - It hard and time consuming
    - Best choice regarding performance
  - **Inline assembly (normally in C/C++)**
    - Very good choice regarding performance
    - However, different compilers use different syntax.
  - **Use Instrinsics from C/C++ as it is the most compatible language with assembly**
    - Much easier, no need to know assembly and deal with hardware details
    - Portable
    - Not all assembly instructions supported