COMP1001

Computer Systems

Dr. Vasilios Kelefouras

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

Outline

- Arrays
- Arrays in assembly
- Reading/writing from/to memory
- Memory addressing
- Conditional Branching

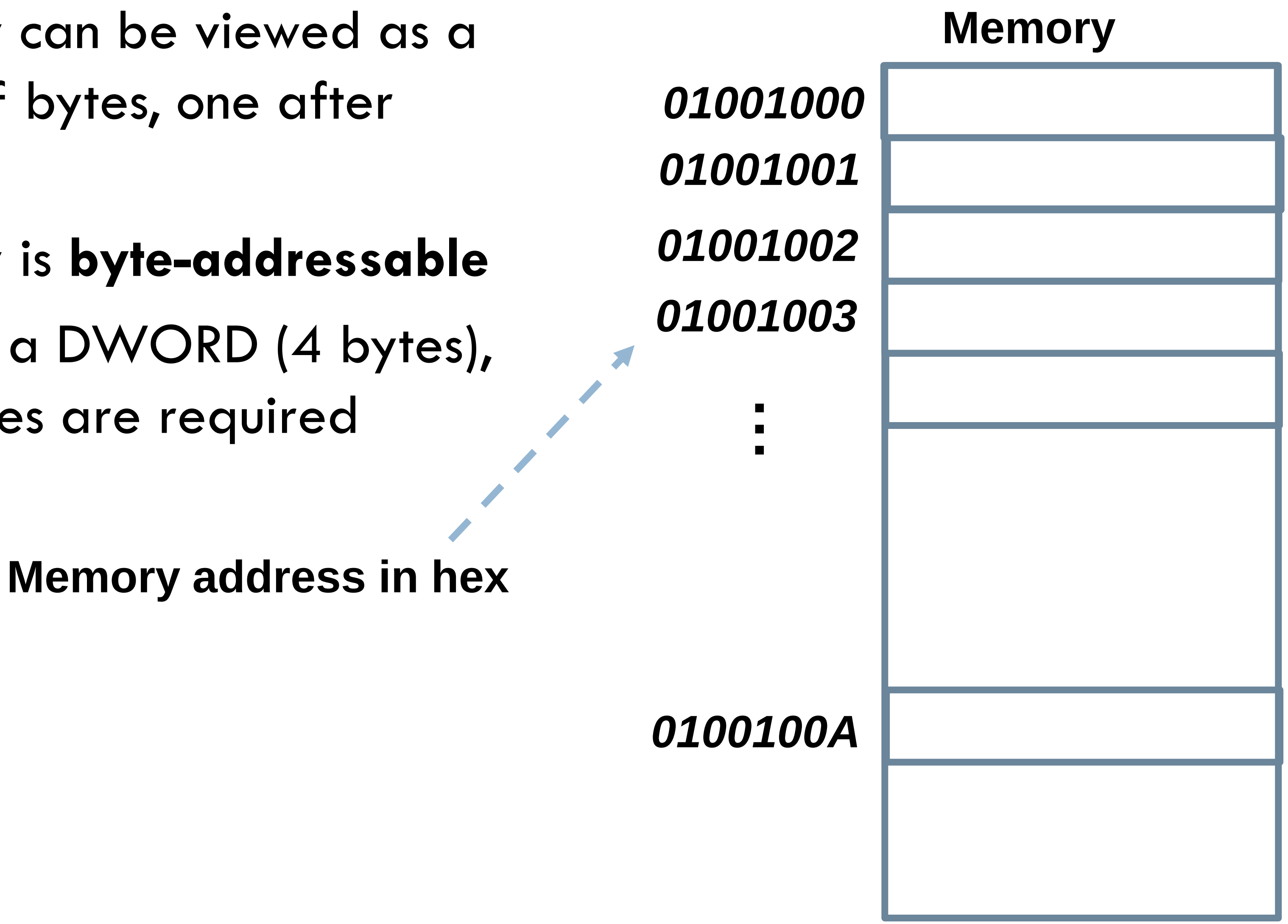
Arrays

- An array is a collection of elements stored in memory one after another (consecutive memory locations)
- All the elements are of the same type, e.g., 4 byte integers
- Below an array of 10 elements is shown.
- In high level programming languages we access an array's element like that:
  - My_Array[0]=45 *; Note that My_Array[0] is the 1st element*
  - My_Array[1]=2
  - My_Array[9]=11

My_Array	45	2	12	98	34	2	7	2	34	11

Main Memory

- Memory can be viewed as a series of bytes, one after another
- Memory is **byte-addressable**
- To store a DWORD (4 bytes), four bytes are required



Arrays in Assembly

- Two arrays are defined below (must be in *.data* section)
  - *arrayA BYTE 2, 4, 6, 8*
  - *arrayB DWORD 0FFFFFh, 0FFFFEh, 0FFFFDh, 0FFFFCh*

arrayA	
Address	**Value**
0x00000000	2
0x00000001	4
0x00000002	6
0x00000003	8

arrayB	
Address	**Value**
0x00000000	FFFFF
0x00000004	FFFFE
0x00000008	FFFFD
0x0000000C	FFFFC

String Literals

- Strings are Byte arrays, each character occupies one byte
- Must end with ‘0’
- An example follows
  - *My_first_string BYTE “Daisy, daisy”,0*

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

Array Attributes

- *arrayA* ***BYTE*** *2, 4, 6, 8*
- *arrayB* ***DWORD*** *0FFFFFh, 0FFFFEh, 0FFFFDh, 0FFFFCh*

arrayA	
Address	**Value**
0x00000000	2
0x00000001	4
0x00000002	6
0x00000003	8

arrayB	
Address	**Value**
0x00000000	FFFFF
0x00000004	FFFFE
0x00000008	FFFFD
0x0000000C	FFFFC

- TYPE ⇒ Data type size
- LENGTH ⇒ Number of elements
- SIZEOF ⇒ Size in bytes

```
MOV eax, TYPE arrayA            ; eax = 1
MOV ebx, LENGTHOF arrayA        ; ebx = 4
MOV ecx, SIZEOF arrayA          ; ecx = 4

MOV eax, TYPE arrayB            ; eax = 4
MOV ebx, LENGTHOF arrayB        ; ebx = 4
MOV ecx, SIZEOF arrayB          ; ecx = 16
```

Array Attributes (2)


```
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

Reading/Writing array elements

- ***mov*** (used to load)
  - *; Assume the following array : arrayA = [2, 4, 6, 8]*

    mov eax, OFFSET arrayA ; loads the memory address of the array

    mov ebx, [eax + TYPE arrayA * 1] ; mov ebx, '[addr]', means load the value at address addr. Here, ebx=4
- ***lea*** (used to load)
  - *lea* eax, arrayA ; load effective address

    mov ebx, [eax + TYPE arrayA * 1] ; ebx = 4
- ***mov*** (used to store)
  - *mov* ecx, 5

    mov [eax + TYPE arrayA * 2], ecx ; now, arrayA = [2, 4, 5, 8]

- Square brackets: can be thought of as "value at address".

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)
- Consider the following integer 0x12345678 in memory

Big-Endian

Memory Address	Data
0x0001	12
0x0002	34
0x0003	56
0x0004	78

Little-Endian

Memory Address	Data
0x0001	78
0x0002	56
0x0003	34
0x0004	12

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.

Addressing Modes

- ***Register Mode***
  - Operands are located in registers
  - Mov eax, ebx
- ***Immediate Mode***
  - A constant integer number (8, 16 or 32 bits)
  - Mov ax, 45h
  - Add ax, 99
- ***Memory Mode*** (it is applied in several different ways)
  - Access to a location in memory is required
  - Slower than the other two
  - *mov eax, OFFSET arrayA*

    *mov ebx, [eax + TYPE arrayA * 1]*

Data movement

- For moving data:
  - Both operands must be of the same size.
  - Both operands cannot be memory operands (must use a register as an intermediary).

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

Addressing Modes – Instruction operand notation

Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

Taken from https://slideplayer.com/slide/6866304/

Unconditional Branching

- Branching means that a program can follow different code paths, even skip instructions, based on implementation.
- Unconditional jump format: ***jmp*** label

```
top:
          mov al, 3
          add al, 5
          jmp bottom
middle:
          add al, 32
bottom:
          add al, 2
```

- In the code above, we skipped the *middle* label and instructions inside it.

Conditional Branching

- Two-step process:
  1. condition checking or comparison of operands
  2. jump based on results
- Condition checking
  - ***cmp*** instruction is used to compare two operands (e.g. equality, less than, greater than, etc.).
  - It does so by subtracting the source operand from the destination, and modifies CPU flags accordingly
  - *cmp al, val ; cmp M/R, L/M/R*
- Many instructions exist for branching based on the results of comparison.

How an if-condition looks like in assembly?


```
COMMENT!
if (val >= 1)
    val = 4
else
    val = 3
!
cmp val, 1   ; compare val with 1
JAE setVal4 ; jump to label if greater
mov val, 3
jmp done
setVal4:
    mov val, 4
done:
    INVOKE ExitProcess, 0 ; call exit function
```

Conditional jump instructions (1)

Sign	Flag	Instruction	Description
Signed and unsigned	OF = 1	JO	Jump if overflow
	OF = 0	JNO	Jump if not overflow
	PF = 1	JP	Jump if parity
	PF = 0	JNP	Jump if not parity
	SF = 1	JS	Jump if sign
	SF = 0	JNS	Jump if not sign
	ZF = 1	JE	Jump if equal
		JZ	Jump if zero
	ZF = 0	JNE	Jump if not equal
		JNZ	Jump if not zero
	CX = 0	JCXZ	Jump if *CX* register is zero
	ECX = 0	JECXZ	Jump if *ECX* register is zero
	RCX = 0	JRCXZ	Jump if *RCX* register is zero

Conditional jump instructions (2)

Signed	SF != OF	JL	Jump if less
		JNGE	Jump if not greater or equal
	SF = OF	JGE	Jump if greater or equal
		JNL	Jump if not less
	ZF = 1 or SF != OF	JLE	Jump if less or equal
		JNG	Jump if not greater
	ZF = 0 and SF = OF	JG	Jump if greater
		JNLE	Jump if not less or equal
Unsigned	CF = 1	JB	Jump if below
		JC	Jump if carry
		JNAE	Jump if not above or equal
	CF = 0	JAE	Jump if above or equal
		JNB	Jump if not below
		JNC	Jump if not carry
	CF = 1 or ZF = 1	JBE	Jump if below or equal
		JNA	Jump if not above
	CF = 0 and ZF = 0	JA	Jump if above
		JNBE	Jump if not below or equal

What is a for loop?

- A **for loop** is perhaps the most common iterative statement
- Let's go through a simple for loop using C language format
- *The following for loop will execute the print(i) function 10 times. Each time the function is being executed, i has a different value : 0,1,2,3…,9*

```
// C like code for loop example
for (i=0; i<10; i++) {
 print(i);
}
```

How a for loop looks like in assembly?


```
// C code
for (i=0; i<100; i++) {
 S1
}

// assembly code
loop:
    S1
    inc i           // increment i
    cmp i, 100      // compare i to 100
    jne loop        // jump if i lower to 100
```

Nested For Loops in Assembly


```
COMMENT!
for(int x = 0; x < 2; x++)
    for (int y = 0; y < 3; y++)
        value++;
!
outer:  ; outer loop label
    mov y, 0 ; set y = 0
    inner:  ; inner loop label
        inc value ; value++
        inc y ; y++
        cmp y, 3 ; compare y with 3
        jne inner ; jump to inner label if less than 3
    inc x ; x++
    cmp x, 2 ; compare x with 2
    jne outer ; jump to outer label if less than 2
INVOKE ExitProcess, 0 ; call exit function
```

Different Ways of writing Assembly

- There are 3 ways to write assembly
  - **Use Assembler**
    - It is 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

Any questions?

Further Reading

- Chapter 1 and Chapter 2 in 'Modern X86 Assembly Language Programming' , available at https://www.pdfdrive.com/download.pdf?id=185772000&h=3dfb070c1742f50b500f07a63a30c86a&u=cache&ext=pdf