III. STRUCTURED ASSEMBLY LANGUAGE PROGRAMMING TECHNIQUES

Structured Data Types

Outline

1. Arrays
2. Strings
3. Structures/Records
4. Sets

Structured Data Types

- Aggregations of atomic or other structured data types.
- Contiguous bytes of data divided according to programmer's concept of data types.

Arrays

- Collection of data of the same type.

- char resb 10 ; array of characters
  num resw 10 ; array of integers

- char times 10 db 65 ; each initialized to ‘A’
  num times 10 dw 0 ; each initialized to 0

Accessing Array Elements

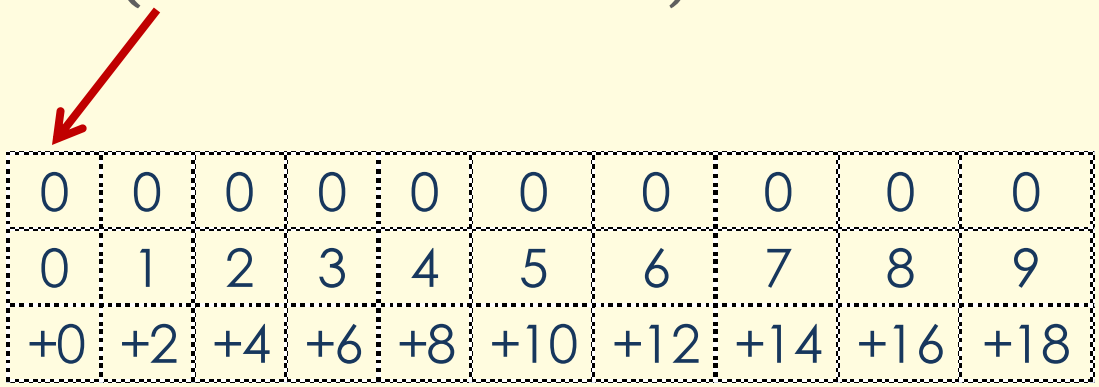
- Each element is of equal *size* (byte, word, ...).
- The variable name is the only name by which we can reference the location of the array in memory.

```
char times 10 db 65
```

- The variable is the **base address** of the array.
- The **i**th element is *i* * *size* far from the base address.

Accessing Array Elements

- array **num** (num times 10 dw 0)



0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9
+0	+2	+4	+6	+8	+10	+12	+14	+16	+18

- We need to know the size of each element.
- The **i**th element is *i* * *size* far from the base address.

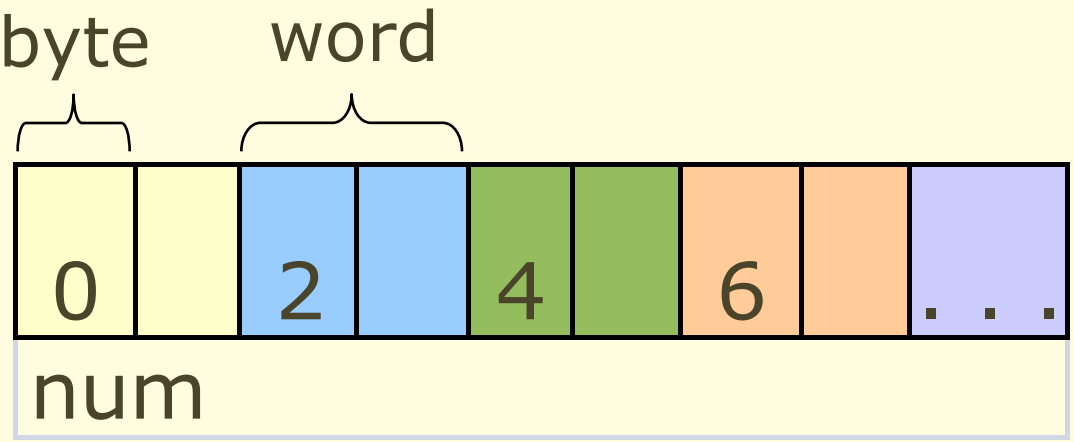
Accessing Array Elements

- array **num** (num times 10 db 0)

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9
+0	+1	+2	+3	+4	+5	+6	+7	+8	+9

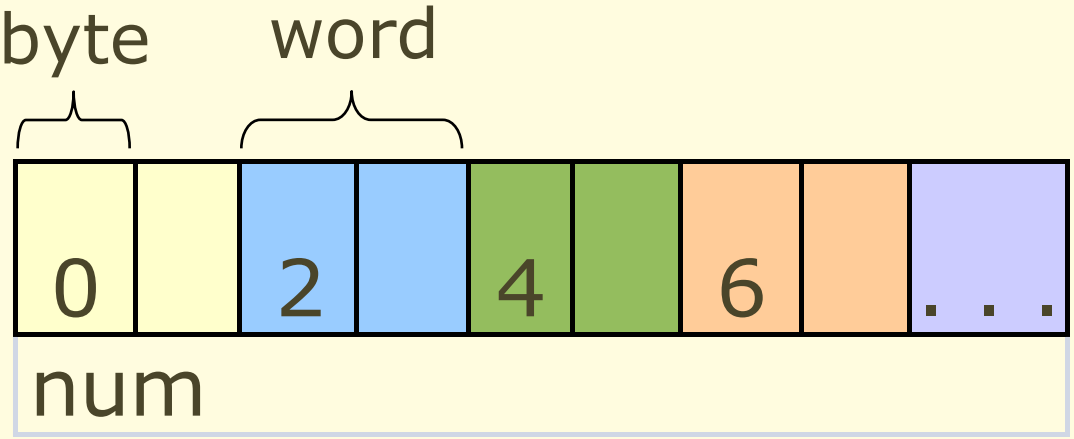
Accessing Array Elements

num resw 10



Accessing Array Elements

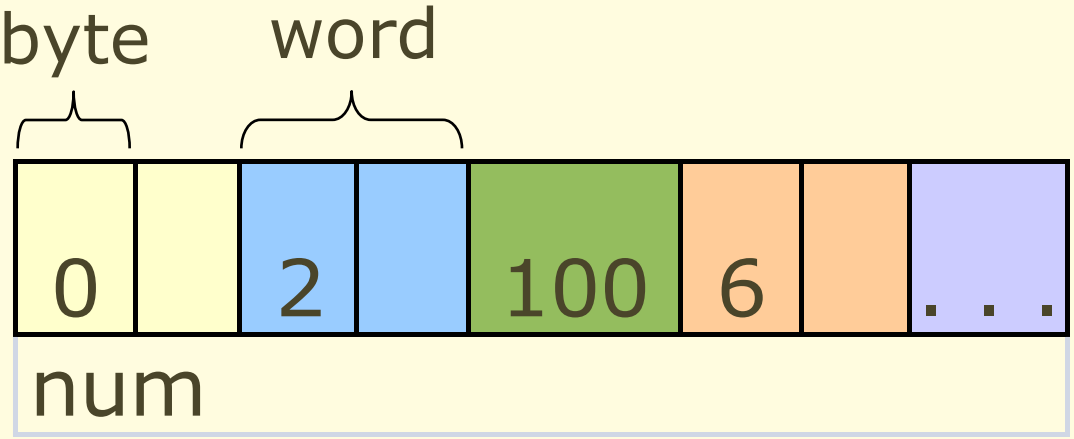
num resw 10



mov word[num+4], 100 (num[2]=100)

Accessing Array Elements

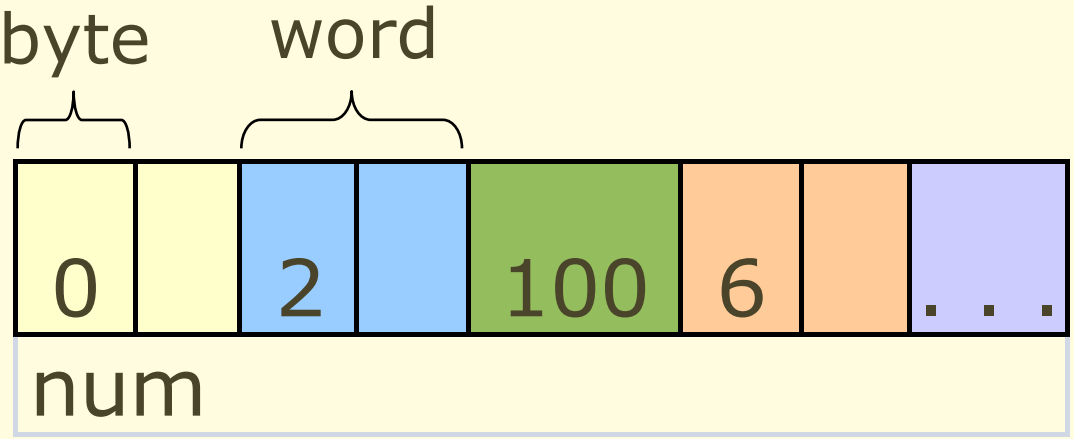
num resw 10



mov word[num+4], 100 (num[2]=100)

Accessing Array Elements

num resw 10

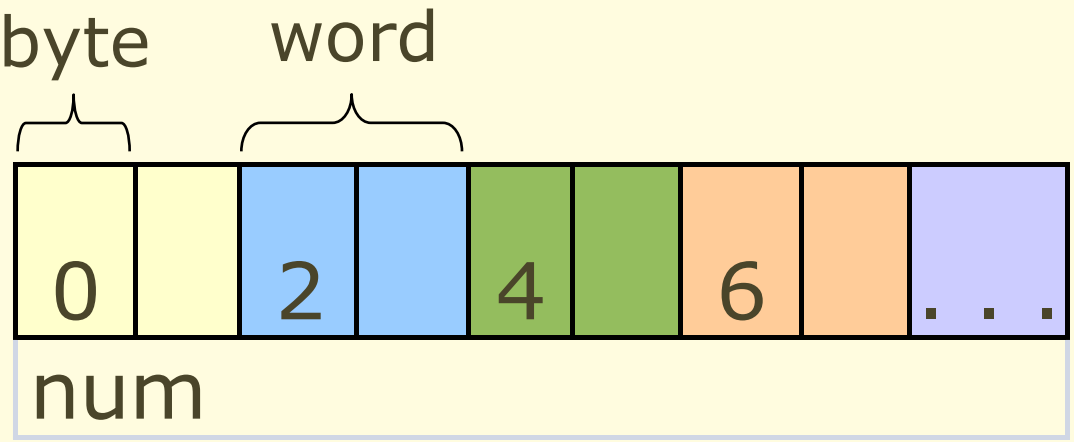


mov word[num+4], 100 (num[2]=100)

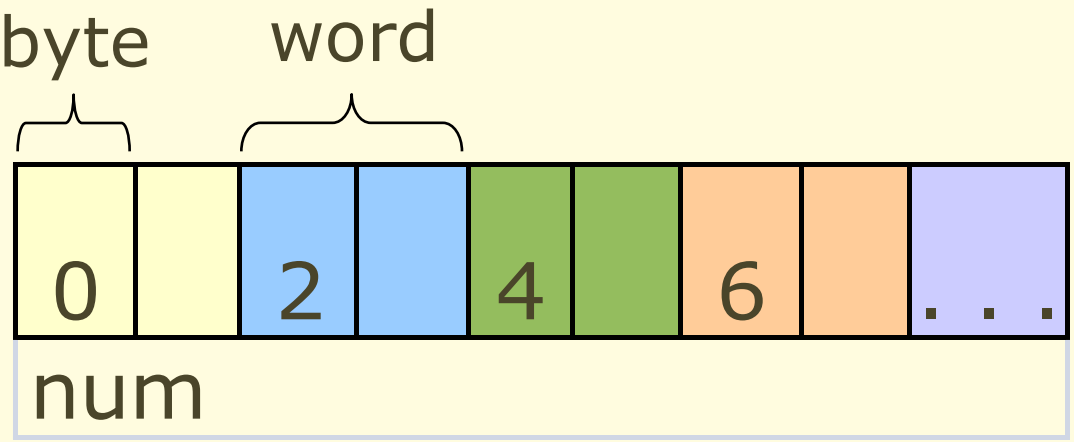
mov word[num+18], 90 (num[9]=90)

Accessing Array Elements

```
size equ 2
num resw 10
i dw 0
```



Accessing Array Elements




```
size equ 2
num resw 10
i dw 0

mov ax, size                    ; compute for i*size
mul word [i]                    ; index starts with 0
mov si, ax                      ; copy offset to index register
mov word[num+si], 100           ; num[i] = 100
```

Accessing Array Elements

```
int i, scores[5];

for (i=0;i<5;i++)
      scanf("%d",&scores[i]);
```

```
i db 0

scores resb 5
```

+0 +1 +2 +3 +4

```
mov esi, 0
for:
      cmp byte[i], 5
      jnl exit
      mov eax, 3
      mov ebx, 0
      lea ecx, [scores+esi]
      mov edx, 2
      int 80h
      inc esi
      inc byte[i]
      jmp for
exit:
```

Accessing Array Elements

```
#define max 10
 int i, scores[max];

 for (i=0;i<max;i++)
        scores[i]=max – i;
```

```
max equ 10
i dw 0
scores resw max
```

```
mov esi, 0
for:
    cmp word[i], max
    jnl exit
    mov ax, max
    sub ax, word[i]
    mov word[scores+esi*2], ax
    inc word[i]
    inc esi
    jmp for
exit:
```

Accessing Array Elements

mov esi, 0	esi =	0	
for:			
cmp word[i], max	i =	0	1
jnl exit			
mov ax, max	ax =	10	10
sub ax, word[i]	ax =	10	9
mov word[scores+esi*2], ax	[scores+esi*2] =	10	9
inc word[i]	i =	1	2
inc esi	esi =	1	2
jmp for			
exit:			

Accessing Array Elements

mov esi, 0		
for:		
cmp word[i], max	i = 1	2
jnl exit		
mov ax, max	ax = 10	10
sub ax, word[i]	ax = 9	8
mov word[scores+esi*2], ax	[scores+esi*2] = 9	8
inc word[i]	i = 2	3
inc esi	esi = 2	3
jmp for		
exit:		

Strings

- Strings are more than just array of characters.
- Strings are treated as **atomic**.
- Strings may have an actual value less than the total number of cells declared.
- char str[10]; // char has 10 cells
  strcpy (str,”Hello”); // char has 5 characters.

String Representation

- string db ’welcome’
  strlen equ $ - string

- string1 db ’this is cool’,0

- string2 resb 50

String Instructions

Mnemonic	Meaning	Operand(s) required
LODS	LOaD String	source
STOS	STOre String	destination
MOVS	MOVe String	source & destination
CMPS	CoMPare Strings	source & destination
SCAS	SCAn String	destination

String Instructions

- Operands use ESI and EDI registers.
- Each string instruction can operate on 8-, 16-, or 32-bit operands.
- As part of execution, string instructions automatically update (increment or decrement) the index register(s) used by them.

String Instructions

- The direction of string processing (forward or backward) is controlled by the direction flag.
- String instructions can accept a repetition prefix to repeatedly execute the operation.
- The three prefixes are divided into two categories: *unconditional or conditional repetition.*

Repetition Prefix

- **rep**
- unconditional repeat prefix which causes the instruction to repeat according to the value in the ECX register

while (ECX ≠ 0)

execute the string instruction;

ECX := ECX–1;

end while

Repetition Prefix

- **repe/repz**
- one of the two conditional repeat prefixes
- Its operation is similar to that of rep except that repetition is also conditional on the zero flag.

Repetition Prefix

- **repe/repz**

  while (ECX ≠ 0)

  execute the string instruction;

  ECX := ECX–1;

  if (ZF = 0) then

  exit loop

  end if

  end while

Repetition Prefix

- **repne/repnz**
- similar to the repe/repz prefix except that the condition tested is ZF = 1

Repetition Prefix

- **repne/repnz**

  while (ECX ≠ 0)

  execute the string instruction;

  ECX := ECX–1;

  if (ZF = 1) then

  exit loop

  end if

  end while

Direction Flag

- The direction of string operations depends on the value of the direction flag.

- If the direction flag is clear (DF = 0), string operations proceed in the forward direction (from head to tail of a string), otherwise, string processing is done in the opposite direction.

Direction Flag

- Two instructions to explicitly manipulate the direction flag:
  - **std** - set direction flag (DF = 1)
  - **cld** - clear direction flag (DF = 0)

String Move Instructions

Move a String (movs)

- movs dest_string, source_string
- movsb
- movsw
- movsd

String Move Instructions

- movsb - move a byte string

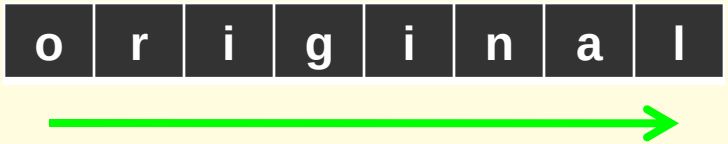
```
ES:EDI := (DS:ESI)          ; copy a byte
if (DF = 0) then            ; forward direction
        ESI := ESI+1
        EDI := EDI+1
else                        ; backward direction
        ESI := ESI–1
        EDI := EDI–1
end if
```

String Move Instructions

```
string1 db 'original'
strlen equ $ - string1

string2 resb 80

mov ECX, strlen
mov ESI, string1
mov EDI, string2
cld                          ; forward direction
rep movsb
```



String Move Instructions

- **Load a String (lods)**
- copies the value from the source string (ESI) to AL (lodsb), AX (lodsw), or EAX (lodsd)
- lodsb - load a byte string

String Move Instructions

- **Load a String (lods)**

```
AL := (DS:ESI)                  ; copy a byte
if (DF = 0) then                ; forward direction
        ESI := ESI+1
else                            ; backward direction
        ESI := ESI–1
end if
```

String Move Instructions

- **Store a String (stos)**
- copies the value in AL (stosb), AX (stosw), or EAX (stosd) to the destination string (EDI)
- stosb - store a byte string

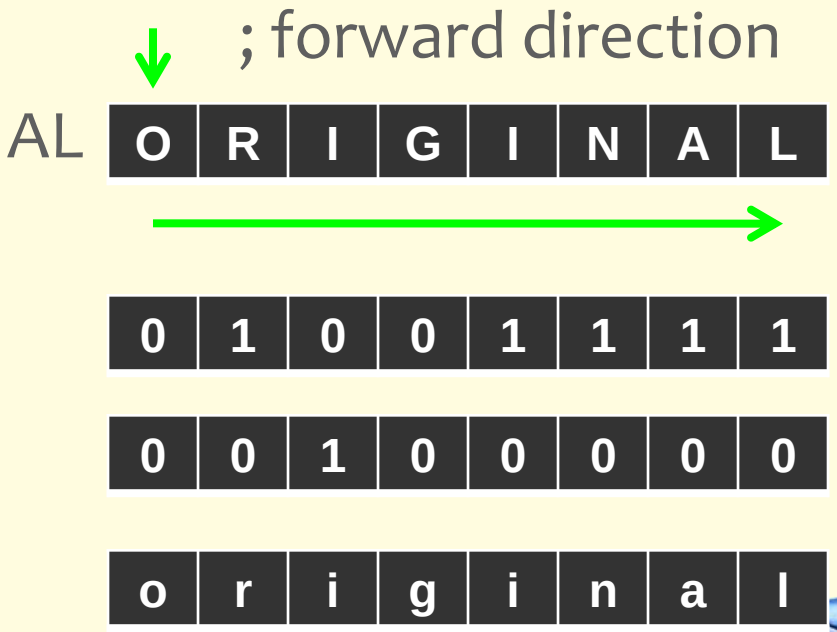
String Move Instructions

- **Store a String (stos)**

```
ES:EDI := AL                    ; copy a byte
if (DF = 0) then                ; forward direction
        EDI := EDI+1
else                            ; backward direction
        EDI := EDI–1
end if
```

String Move Instructions

```
mov ECX, strlen
mov ESI, string1
mov EDI, string2
cld                          ; forward direction
loop1:
    lodsb
    add AL, 32
    stosb
    loop loop1
done:
```



String Compare Instruction

- cmpsb - compare two byte strings
- compare the two bytes at ESI and EDI and set flags

String Compare Instruction

- cmpsb - compare two byte strings

```
if (DF = 0) then                ; forward direction
        ESI := ESI+1
        EDI := EDI+1
else                            ; backward direction
        ESI := ESI–1
        EDI := EDI–1
end if
```

String Compare Instruction

- The cmps instruction compares the two bytes, words, or doublewords at ESI and EDI and sets the flags just like the cmp instruction.
- Like the cmp instruction, cmps performs

  (ESI) – (EDI)

  and sets the flags according to the result.
- The cmps instruction is typically used with the repe/repz or repne/repnz prefix.

String Compare Instruction

```
string1 db 'abcdfghi',0
strLen EQU $ - string1
string2 db 'abcdefgh',0

mov ECX,strLen
mov ESI,string1
mov EDI,string2
cld ; forward direction
repe cmpsb
```

- leaves ESI pointing to g in string1 and EDI to f in string2 Therefore, adding

```
dec ESI
dec EDI
```

- leaves ESI and EDI pointing to the last character that differs

Scanning a String

- The scas instruction is useful in searching for a particular value or character in a string.
- The value should be in AL (scasb), AX (scasw), or EAX (scasd), and EDI should point to the string to be searched.

Scanning a String

- scasb - scan a byte string

  compare AL to the byte at EDI and set flags

  if (DF = 0) then ; forward direction

  EDI := EDI+1

  else ; backward direction

  EDI := EDI–1

  end if

Scanning a String

```
string1 db 'abcdefgh'
strLen EQU $ - string1

mov ECX, strLen
mov EDI, string1
mov AL, 'e'          ; character to be searched
cld                  ; forward direction
repne scasb
dec EDI
```