

Strings and Arrays

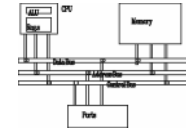
Computer Organization and Assembly Languages

Yung-Yu Chuang

2006/12/18

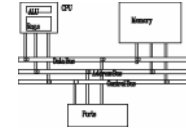
with slides by Kip Irvine

Announcements



- Voting next week.
- Assignment #5 will be due on 12/27 (Wed) midnight.
- Final project will be announced next week.
- SIMD next lecture.

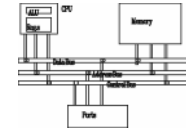
Overview



- 90% of time spent on 10% of code. They are often loops manipulating on strings (essentially 1D arrays) and arrays.
- Intel's optimized string primitive instructions
- Some typical string procedures
- Two-dimensional arrays
- Searching and sorting integer arrays

SIMD next time, which is even more powerful

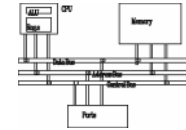
String primitive instructions



- Move string data: **MOVSB, MOVSW, MOVSD**
- Compare strings: **CMPSB, CMPSW, CMPSD**
- Scan string: **SCASB, SCASW, SCASD**
- Store string data: **STOSB, STOSW, STOSD**
- Load ACC from string: **LODSB, LODSW, LODSD**

- Only use memory operands
- Use **ESI**, **EDI** or both to address memory

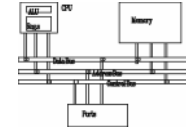
MOVSB, MOVSW, MOVSD



- The **MOVSB**, **MOVSW**, and **MOVSD** instructions copy data from the memory location pointed to by **ESI** to the memory location pointed to by **EDI**.

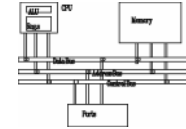
```
.data
source DWORD 0FFFFFFFFh
target DWORD ?
.code
mov esi,OFFSET source
mov edi,OFFSET target
movsd
```

MOVSB, MOVSW, MOVSD



- **ESI** and **EDI** are automatically incremented or decremented:
 - **MOVSB** increments/decrements by 1
 - **MOVSW** increments/decrements by 2
 - **MOVSD** increments/decrements by 4

Direction flag

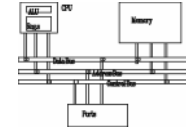


- The direction flag controls the incrementing or decrementing of **ESI** and **EDI**.
 - **DF** = clear (0): increment **ESI** and **EDI**
 - **DF** = set (1): decrement **ESI** and **EDI**

The direction flag can be explicitly changed using the **CLD** and **STD** instructions:

CLD	; clear Direction flag
STD	; set Direction flag

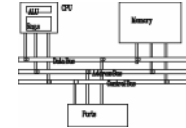
Using a repeat prefix



- **REP** (a repeat prefix) can be inserted just before **MOVSB**, **MOVSW**, or **MOVSD**.
- **ECX** controls the number of repetitions
- Example: copy 20 doublewords from source to target

```
.data
source DWORD 20 DUP(?)
target DWORD 20 DUP(?)
.code
cld                ; direction = forward
mov ecx,LENGTHOF source ; set REP counter
mov esi,OFFSET source
mov edi,OFFSET target
rep movsd          ; REP checks ECX>0 first
```

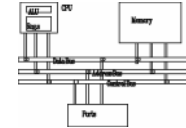
Using a repeat prefix



REP	Repeat while ECX>0
REPZ, REPE	Repeat while Zero=1 and ECX>0
REPNZ, REPNE	Repeat while Zero=0 and ECX>0

- Conditions are checked first before repeating the instruction

Your turn . . .

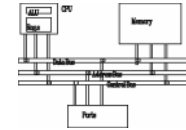


- Use **MOVSD** to delete the first element of the following doubleword array. All subsequent array values must be moved one position forward toward the beginning of the array:

array DWORD 1,1,2,3,4,5,6,7,8,9,10

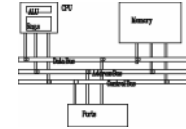
```
.data
array DWORD 1,1,2,3,4,5,6,7,8,9,10
.code
cld
mov ecx,(LENGTHOF array) - 1
mov esi,OFFSET array+4
mov edi,OFFSET array
rep movsd
```

CMPSB, CMPSW, CMPSD



- The **CMPSB**, **CMPSW**, and **CMPSD** instructions each compare a memory operand pointed to by **ESI** to a memory operand pointed to by **EDI**.
 - **CMPSB** compares bytes
 - **CMPSW** compares words
 - **CMPSD** compares doublewords
- Repeat prefix (**REP**) is often used

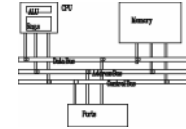
Comparing a pair of doublewords



If source > target, the code jumps to label L1;
otherwise, it jumps to label L2

```
.data  
source DWORD 1234h  
target DWORD 5678h  
  
.code  
mov esi,OFFSET source  
mov edi,OFFSET target  
cmpsd          ; compare doublewords  
ja L1          ; jump if source > target  
jmp L2          ; jump if source <= target
```

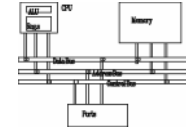
Comparing arrays



Use a **REPE** (repeat while equal) prefix to compare corresponding elements of two arrays.

```
.data
source DWORD COUNT DUP(?)
target DWORD COUNT DUP(?)
.code
mov ecx,COUNT          ; repetition count
mov esi,OFFSET source
mov edi,OFFSET target
cld                    ; direction = forward
repe cmpsd              ; repeat while equal
```

Example: comparing two strings

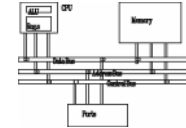


This program compares two strings of equal lengths (source and destination). It displays a message indicating whether the lexical value of the source string is less than the destination string.

```
.data  
source BYTE "MARTIN  "  
dest    BYTE "MARTINEZ"  
str1 BYTE "Source is smaller",0dh,0ah,0  
str2 BYTE "Source is not smaller",0dh,0ah,0
```

*Note that we add spaces after **source** because the following procedure only works for equal-length strings.*

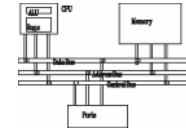
Example: comparing two strings



```
.code
main PROC
    cld                      ; direction = forward
    mov esi,OFFSET source
    mov edi,OFFSET dest
    mov ecx,LENGTHOF source
    repe cmpsb
    jb source_smaller
    mov edx,OFFSET str2 ;source is not smaller
    jmp done
source_smaller:
    mov edx,OFFSET str1 ;source is smaller
done:
    call WriteString
    exit
main ENDP
END main
```

- The **SCASB**, **SCASW**, and **SCASD** instructions compare a value in **AL/AX/EAX** to a byte, word, or doubleword, respectively, addressed by **EDI**.
- Useful types of searches:
 - Search for a specific element in a long string or array.
 - Search for the first element that does not match a given value.

SCASB example

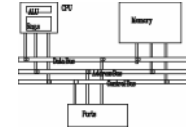


Search for the letter 'F' in a string named str:

```
.data
str BYTE "ABCDEFGH",0
.code
mov edi,OFFSET str
mov al,'F'           ; search for 'F'
mov ecx,LENGTHOF str
cld
repne scasb          ; repeat while not equal
jnz quit
dec edi              ; EDI points to 'F'
```

JNZ tests whether the loop exits on EXC=0 or Z=0

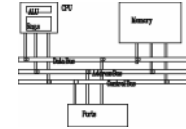
STOSB, STOSW, STOSD



- The **STOSB**, **STOSW**, and **STOSD** instructions store the contents of **AL/AX/EAX**, respectively, in memory at the offset pointed to by **EDI**.
- Example: fill an array with 0FFh (memset)

```
.data
Count = 100
str BYTE Count DUP(?)
.code
mov al,0FFh           ; value to be stored
mov edi,OFFSET str    ; ES:DI points to target
mov ecx,Count         ; character count
cld                   ; direction = forward
rep stosb             ; fill with contents of AL
```

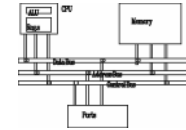
LODSB, LODSW, LODSD



- The **LODSB**, **LODSW**, and **LODSD** instructions load a byte or word from memory at **ESI** into **AL/AX/EAX**, respectively.
- Rarely used with **REP**
- **LODSB** can be used to replace code

```
mov al,[esi]
inc esi
```

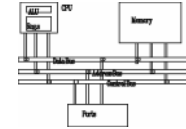
Example



- convert each decimal byte of an array into an its ASCII code.

```
.data
array 1,2,3,4,5,6,7,8,9
dest 9 DUP(?)
.code
    mov esi,OFFSET array
    mov edi,OFFSET dest
    mov ecx,LENGTHOF array
    cld
L1:
    lodsb
    or al,30h
    stosb
    loop L1
```

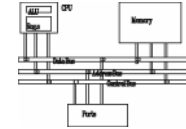
Array multiplication example



Multiply each element of a doubleword array by a constant value.

```
.data
array DWORD 1,2,3,4,5,6,7,8,9,10
multiplier DWORD 10
.code
    cld                ; direction = up
    mov esi,OFFSET array ; source index
    mov edi,esi        ; destination index
    mov ecx,LENGTHOF array ; loop counter
L1: lodsd              ; copy [ESI] into EAX
    mul multiplier      ; multiply by a value
    stosd               ; store EAX at [EDI]
    loop L1
```

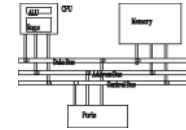
Selected string procedures



The following string procedures may be found in the Irvine32 and Irvine16 libraries:

- **Str_length**
- **Str_copy**
- **Str_compare**
- **Str_ucase**
- **Str_trim**

Str_compare procedure

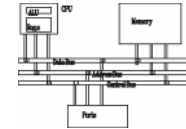


- Compares *string1* to *string2*, setting the Carry and Zero flags accordingly
- Prototype:

```
Str_compare PROTO,  
    string1:PTR BYTE, ; pointer to string  
    string2:PTR BYTE ; pointer to string
```

relation	carry	zero	Branch if true
str1<str2	1	0	JB
str1==str2	0	1	JE
str1>str2	0	0	JA

Str_compare source code

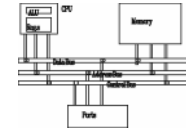


```
Str_compare PROC USES eax edx esi edi,  
    string1:PTR BYTE, string2:PTR BYTE  
    mov     esi,string1  
    mov     edi,string2  
L1:  mov     al,[esi]  
    mov     dl,[edi]  
    cmp     al,0          ; end of string1?  
    jne     L2            ; no  
    cmp     dl,0          ; yes: end of string2?  
    jne     L2            ; no  
    jmp     L3            ; yes, exit with ZF = 1  
L2:  inc     esi          ; point to next  
    inc     edi  
    cmp     al,dl         ; chars equal?  
    je      L1            ; yes: continue loop  
L3:  ret
```

Str_compare ENDP

CMPSB is not used here

Str_length procedure



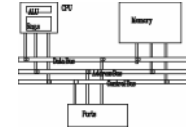
- Calculates the length of a null-terminated string and returns the length in the **EAX** register.
- Prototype:

```
Str_length PROTO,  
    pString:PTR BYTE ; pointer to string
```

Example:

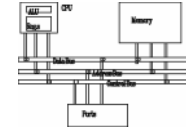
```
.data  
myString BYTE "abcdefg",0  
.code  
    INVOKE Str_length, ADDR myString  
; EAX = 7
```

Str_length source code



```
Str_length PROC USES edi,  
    pString:PTR BYTE      ; pointer to string  
    mov edi,pString  
    mov eax,0              ; character count  
L1:  
    cmp byte ptr [edi],0  ; end of string?  
    je  L2                ; yes: quit  
    inc edi                ; no: point to next  
    inc eax                ; add 1 to count  
    jmp L1  
L2: ret  
Str_length ENDP
```

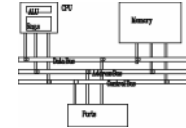
Str_copy Procedure



- Copies a null-terminated string from a source location to a target location.
- Prototype:

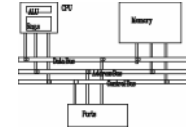
```
Str_copy PROTO,  
    source:PTR BYTE, ; pointer to string  
    target:PTR BYTE ; pointer to string
```

Str_copy Source Code



```
Str_copy PROC USES eax ecx esi edi,  
    source:PTR BYTE,      ; source string  
    target:PTR BYTE       ; target string  
  
    INVOKE Str_length,source ; EAX = length  
    mov ecx,eax             ; REP count  
    inc ecx                 ; add 1 for null byte  
    mov esi,source  
    mov edi,target  
    cld                     ; direction = up  
    rep movsb               ; copy the string  
    ret  
Str_copy ENDP
```

Str_trim Procedure



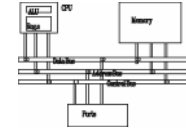
- removes all occurrences of a selected trailing character from a null-terminated string.
- Prototype:

```
Str_trim PROTO,  
    pString:PTR BYTE, ; points to string  
    char:BYTE          ; char to remove
```

Example:

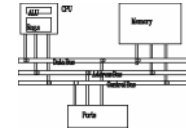
```
.data  
myString BYTE "Hello###",0  
.code  
    INVOKE Str_trim, ADDR myString, '#'  
; myString = "Hello"
```

Str_trim Procedure



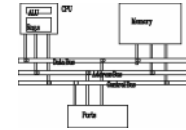
- **Str_trim** checks a number of possible cases (shown here with # as the trailing character):
 - The string is empty.
 - The string contains other characters followed by one or more trailing characters, as in "Hello##".
 - The string contains only one character, the trailing character, as in "#".
 - The string contains no trailing character, as in "Hello" or "H".
 - The string contains one or more trailing characters followed by one or more nontrailing characters, as in "#H" or "####Hello".

Str_trim source code



```
Str_trim PROC USES eax ecx edi,
    pString:PTR BYTE,      ; points to string
    char:BYTE              ; char to remove
    mov    edi,pString
    INVOKE Str_length,edi   ; returns length in EAX
    cmp    eax,0           ; zero-length string?
    je     L2              ; yes: exit
    mov    ecx,eax         ; no: counter = string length
    dec    eax
    add    edi,eax         ; EDI points to last char
    mov    al,char         ; char to trim
    std    edi             ; direction = reverse
    repe   scasb           ; skip past trim character
    jne    L1              ; removed first character?
    dec    edi             ; adjust EDI: ZF=1 && ECX=0
L1:mov    BYTE PTR [edi+2],0 ; insert null byte
L2:ret
Str_trim ENDP
```

Str_ucase procedure



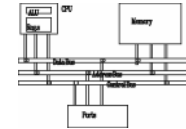
- converts a string to all uppercase characters. It returns no value.
- Prototype:

```
Str_ucase PROTO,  
    pString:PTR BYTE    ; pointer to string
```

Example:

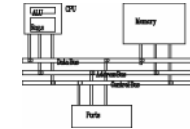
```
.data  
myString BYTE "Hello",0  
.code  
    INVOKE Str_ucase, ADDR myString
```

Str_ucase source code



```
Str_ucase PROC USES eax esi,  
    pString:PTR BYTE  
    mov esi,pString  
L1:mov al,[esi]           ; get char  
    cmp al,0              ; end of string?  
    je  L3                ; yes: quit  
    cmp al,'a'             ; below "a"?  
    jb  L2  
    cmp al,'z'             ; above "z"?  
    ja  L2  
    and BYTE PTR [esi],11011111b ;conversion  
L2:inc esi                ; next char  
    jmp L1  
L3:ret  
Str_ucase ENDP
```

Two-dimensional arrays



logically

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0

row-major

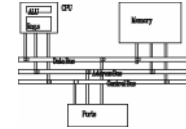
10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

column-major

10	60	B0	20	70	C0	30	80	D0	40	90	E0	50	A0	F0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- IA32 has two operand types which are suited to array applications: base-index operands and base-index displacement

Base-index operand

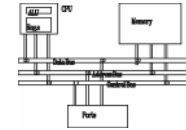


- A base-index operand adds the values of two registers (called base and index), producing an effective address. Any two 32-bit general-purpose registers may be used.

[base + index]

- Base-index operands are great for accessing arrays of structures. (A structure groups together data under a single name.)

Two-dimensional table example



```
table BYTE 10h, 20h, 30h, 40h, 50h
RowSize = ($-table)
        BYTE 60h, 70h, 80h, 90h, 0A0h
        BYTE 0B0h, 0C0h, 0D0h, 0E0h, 0F0h
```

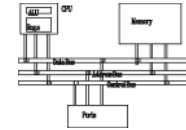
logically

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0

physically

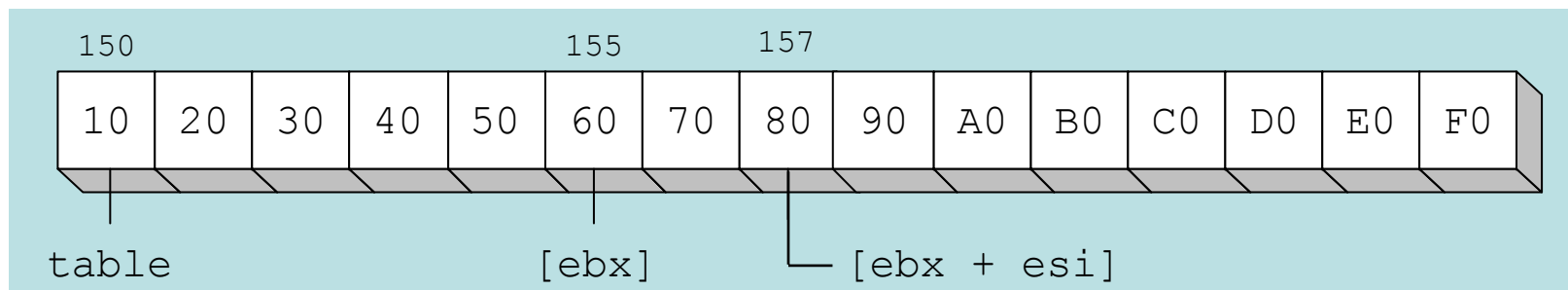
10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Two-dimensional table example

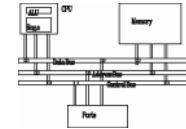


The following code loads the table element stored in row 1, column 2:

```
row_index = 1
column_index = 2
mov ebx, OFFSET table
add ebx, RowSize * row_index
mov esi, column_index
mov al,[ebx + esi]
```



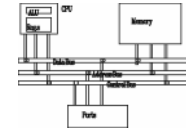
Scale factors



```
table WORD 10h, 20h, 30h, 40h, 50h
RowSize = ($-table)
        WORD 60h, 70h, 80h, 90h, 0A0h
        WORD 0B0h, 0C0h, 0D0h, 0E0h, 0F0h

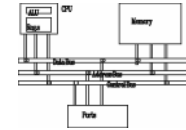
.code
row_index = 1
column_index = 2
mov ebx, OFFSET table
add ebx, RowSize * row_index
mov esi, column_index
mov al, [ebx + esi*TYPE table]
```

Sum of row example



```
row_sum PROC uses ebx ecx edx esi
; inputs:EBX=byte array offset,EAX=row index
;          ECX=row size in bytes
; return: EAX=row sum
    mul ecx          ; row index * row size
    add ebx, eax      ; row offset
    mov eax, 0        ; sum = 0
    mov esi, 0        ; column index
L1: movzx edx, BYTE PTR [ebx+esi]
    add  eax, edx
    inc  esi
    loop L1
    ret
row_sum ENDP
```

Structure application



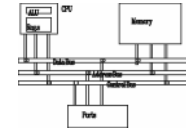
A common application of base-index addressing has to do with addressing arrays of structures (Chapter 10). The following defines a structure named COORD containing X and Y screen coordinates:

```
COORD STRUCT  
    X WORD ? ; offset 00  
    Y WORD ? ; offset 02  
COORD ENDS
```

Then we can define an array of COORD objects:

```
.data  
setOfCoordinates COORD 10 DUP(<>)
```

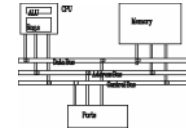
Structure application



The following code loops through the array and displays each Y-coordinate:

```
    mov     ebx,OFFSET setOfCoordinates
    mov     esi,2           ; offset of Y value
    mov     eax,0
L1:
    mov     ax,[ebx+esi]
    call    WriteDec
    add     ebx,SIZEOF COORD
    loop    L1
```

Base-index-displacement operand

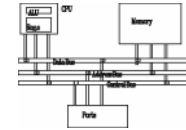


- A base-index-displacement operand adds base and index registers to a constant, producing an effective address. Any two 32-bit general-purpose registers may be used.
- Common formats:

[base + index + displacement]
displacement [base + index]

- **base** and **index** can be any general-purpose 32-bit registers
- **displacement** can be the name of a variable or a constant expression

Two-dimensional table example



The following code loads the table element stored in row 1, column 2:

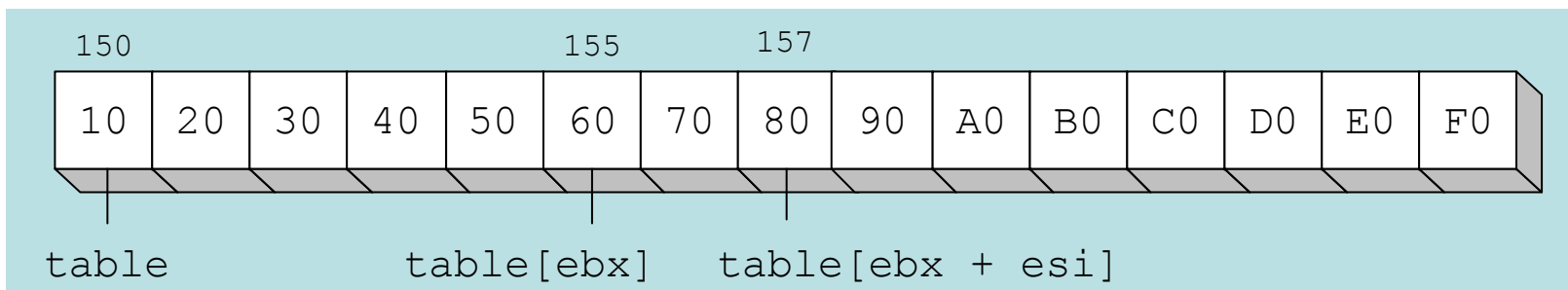
```
row_index = 1
```

```
column_index = 2
```

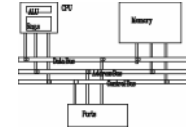
```
mov ebx, RowSize * row_index
```

```
mov esi, column_index
```

```
mov al, table[ebx + esi*TYPE table]
```

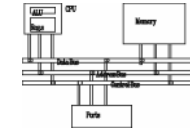


Searching and sorting integer arrays

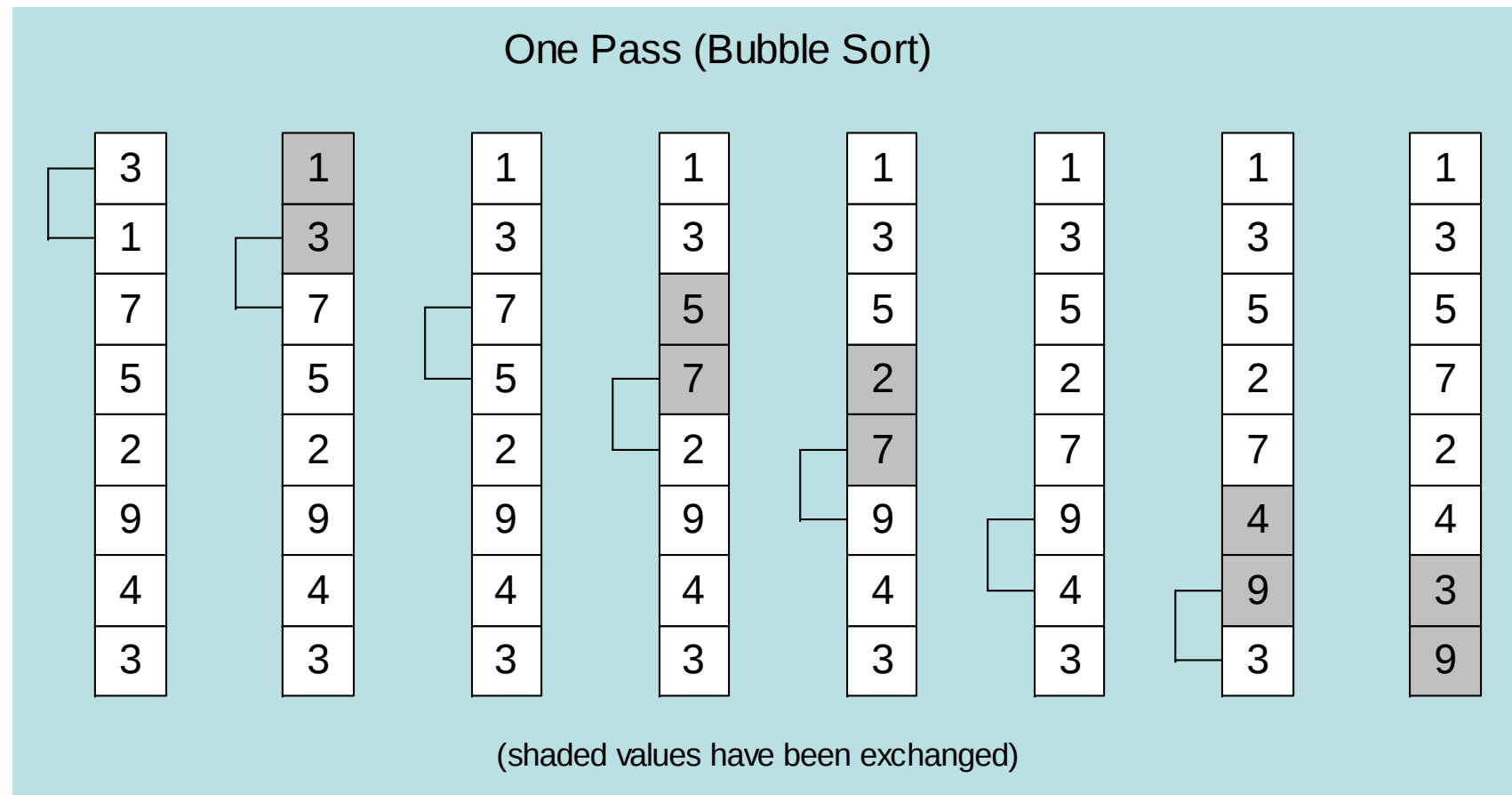


- Bubble Sort
 - A simple sorting algorithm that works well for small arrays
- Binary Search
 - A simple searching algorithm that works well for large arrays of values that have been placed in either ascending or descending order
- Good examples for studying algorithms. Knuth used assembly for his book.
- Good chance to use some of the addressing modes introduced today

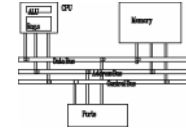
Bubble sort



Each pair of adjacent values is compared, and exchanged if the values are not ordered correctly:



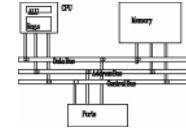
Bubble sort pseudocode



N = array size, cx1 = outer loop counter, cx2 = inner loop counter:

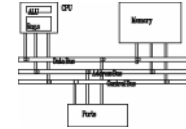
```
cx1 = N - 1
while( cx1 > 0 )
{
    esi = addr(array)
    cx2 = cx1
    while( cx2 > 0 )
    {
        if( array[esi] < array[esi+4] )
            exchange( array[esi], array[esi+4] )
        add esi, 4
        dec cx2
    }
    dec cx1
}
```

Bubble sort implementation



```
BubbleSort PROC USES eax ecx esi,
    pArray:PTR DWORD, Count:DWORD
    mov     ecx,Count
    dec     ecx                ; decrement count by 1
L1: push   ecx                ; save outer loop count
    mov     esi,pArray        ; point to first value
L2: mov     eax,[esi]          ; get array value
    cmp     [esi+4],eax        ; compare a pair
    jge     L3                ; if [esi]<=[esi+4],skip
    xchg    eax,[esi+4]        ; else exchange the pair
    mov     [esi],eax
L3: add     esi,4              ; move both pointers forward
    loop    L2                ; inner loop
    pop     ecx                ; retrieve outer loop count
    loop    L1                ; else repeat outer loop
L4: ret
BubbleSort ENDP
```

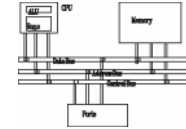
Binary search



- Searching algorithm, well-suited to large ordered data sets
- Divide and conquer strategy
- Classified as an $O(\log n)$ algorithm:
 - As the number of array elements increases by a factor of n , the average search time increases by a factor of $\log n$.

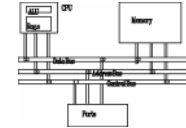
Array Size (n)	Maximum Number of Comparisons: $(\log_2 n) + 1$
64	7
1,024	11
65,536	17
1,048,576	21
4,294,967,296	33

Binary search pseudocode



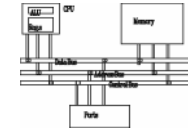
```
int BinSearch( int values[], int count
               const int searchVal, )
{
    int first = 0;
    int last = count - 1;
    while( first <= last )
    {
        int mid = (last + first) / 2;
        if( values[mid] < searchVal )
            first = mid + 1;
        else if( values[mid] > searchVal )
            last = mid - 1;
        else
            return mid; // success
    }
    return -1;          // not found
}
```

Binary search implementation



```
BinarySearch PROC uses ebx edx esi edi,  
    pArray:PTR DWORD,    ; pointer to array  
    Count:DWORD,         ; array size  
    searchVal:DWORD      ; search value  
  
LOCAL first:DWORD,      ; first position  
    last:DWORD,         ; last position  
    mid:DWORD           ; midpoint  
    mov     first,0      ; first = 0  
    mov     eax,Count    ; last = (count - 1)  
    dec     eax  
    mov     last,eax  
    mov     edi,searchVal ; EDI = searchVal  
    mov     ebx,pArray   ; EBX points to the array  
L1:                                     ; while first <= last  
    mov     eax,first  
    cmp     eax,last  
    jg      L5           ; exit search
```

Binary search implementation



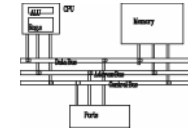
```
; mid = (last + first) / 2
    mov    eax,last
    add    eax,first
    shr    eax,1
    mov    mid,eax

; EDX = values[mid]
    mov    esi,mid
    shl    esi,2           ; scale mid value by 4
    mov    edx,[ebx+esi] ; EDX = values[mid]

; if ( EDX < searchval(EDI) ) first = mid + 1;
    cmp    edx,edi
    jge    L2
    mov    eax,mid         ; first = mid + 1
    inc    eax
    mov    first,eax
    jmp    L4              ; continue the loop
```

base-index
addressing

Binary search implementation



```
;else if( EDX > searchVal(EDI)) last = mid - 1;
L2: cmp    edx,edi                ; (could be removed)
     jle    L3
     mov    eax,mid               ; last = mid - 1
     dec    eax
     mov    last,eax
     jmp    L4                   ; continue the loop

; else return mid
L3: mov    eax,mid               ; value found
     jmp    L9                   ; return (mid)

L4: jmp    L1                   ; continue the loop
L5: mov    eax,-1                ; search failed
L9: ret
BinarySearch ENDP
```