

Structures and Macros

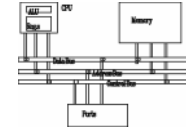
Computer Organization and Assembly Languages

Yung-Yu Chuang

2005/12/08

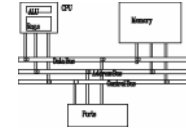
with slides by Kip Irvine

Overview



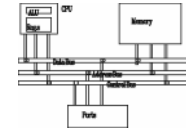
- Structures
- Macros
- Conditional-Assembly Directives
- Defining Repeat Blocks

Structures overview



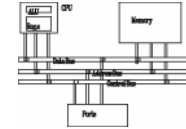
- Defining Structures
- Declaring Structure Variables
- Referencing Structure Variables
- Example: Displaying the System Time
- Nested Structures
- Example: Drunkard's Walk
- Declaring and Using Unions

Structure



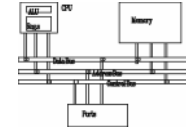
- A template or pattern given to a logically related group of variables.
- field - structure member containing data
- Program access to a structure:
 - entire structure as a complete unit
 - individual fields
- Useful way to pass multiple related arguments to a procedure
 - example: file directory information

Using a structure



- Structures in assembly are essentially the same as structures in C and C++
- Using a structure involves three sequential steps:
 1. Define the structure.
 2. Declare one or more variables of the structure type, called structure variables.
 3. Write runtime instructions that access the structure.

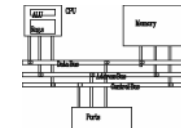
Structure definition syntax



```
name STRUCT  
    field-declarations  
name ENDS
```

- ***field-declarations*** are identical to variable declarations
- The **COORD** structure used by the MS-Windows programming library identifies X and Y screen coordinates

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

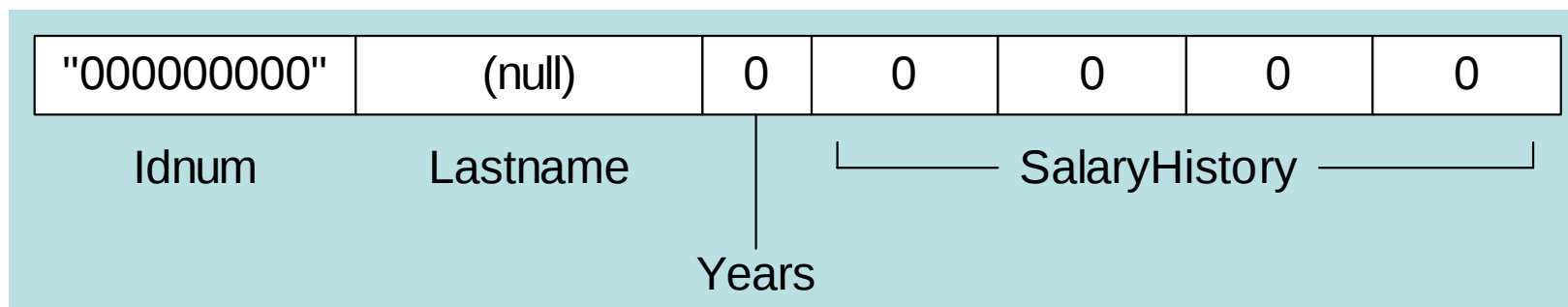


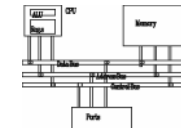
Employee structure

A structure is ideal for combining fields of different types:

```
Employee STRUCT
    IdNum BYTE "0000000000"
    LastName BYTE 30 DUP(0)
    Years WORD 0
    SalaryHistory DWORD 0,0,0,0
Employee ENDS
```

initializers



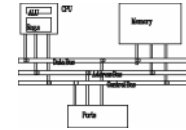


Declaring structure variables

- Structure name is a user-defined type
- Insert replacement initializers between brackets or braces:
 < . . . > or { . . . }
- Empty brackets <> retain the structure's default field initializers (or braces {})
- Examples:

```
.data  
point1 COORD <5,10>  
point2 COORD <>  
worker Employee <>
```

Initializing array fields

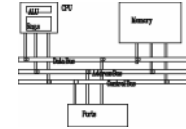


- Use the DUP operator to initialize one or more elements of an array field:

```
person1 Employee <"55522333">  
person2 Employee {"55522333"}  
person3 Employee <, "Jones">  
person4 Employee <,,,2 DUP(20000)>
```

"000000000"	(null)	0	0	0	0	0
Idnum	Lastname	Years	SalaryHistory			

Array of structures



- An array of structure objects can be defined using the DUP operator.
- Initializers can be used

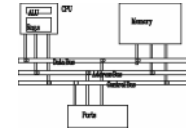
```
NumPoints = 3
```

```
AllPoints COORD NumPoints DUP(<0,0>)
```

```
RD_Dept Employee 20 DUP(<>)
```

```
accounting Employee 10 DUP(<,,,4 DUP(20000) >)
```

Referencing structure variables



```
Employee STRUCT                                ; bytes
    IdNum BYTE "0000000000"                    ; 9
    LastName BYTE 30 DUP(0)                     ; 30
    Years WORD 0                                ; 2
    SalaryHistory DWORD 0,0,0,0                 ; 16
Employee ENDS                                  ; 57
```

.data

worker Employee <>

```
mov eax,TYPE Employee                        ; 57
mov eax,SIZEOF Employee                      ; 57
mov eax,SIZEOF worker                        ; 57
mov eax,TYPE Employee.SalaryHistory          ; 4
mov eax,LENGTHOF Employee.SalaryHistory     ; 4
mov eax,SIZEOF Employee.SalaryHistory        ; 16
```

The diagram illustrates a computer system architecture. At the top left is a box labeled 'CPU' containing two sub-boxes, 'ADD' and 'SUB'. To its right is a box labeled 'Memory'. Below these is a horizontal 'Data Bus'. At the bottom is a box labeled 'Ports'. A 'Control Bus' runs horizontally between the CPU and the Ports. A 'Memory Bus' runs horizontally between the Memory and the Ports. Vertical lines connect the CPU to the Data Bus, the Memory to the Data Bus, and the Ports to the Control Bus and Memory Bus.

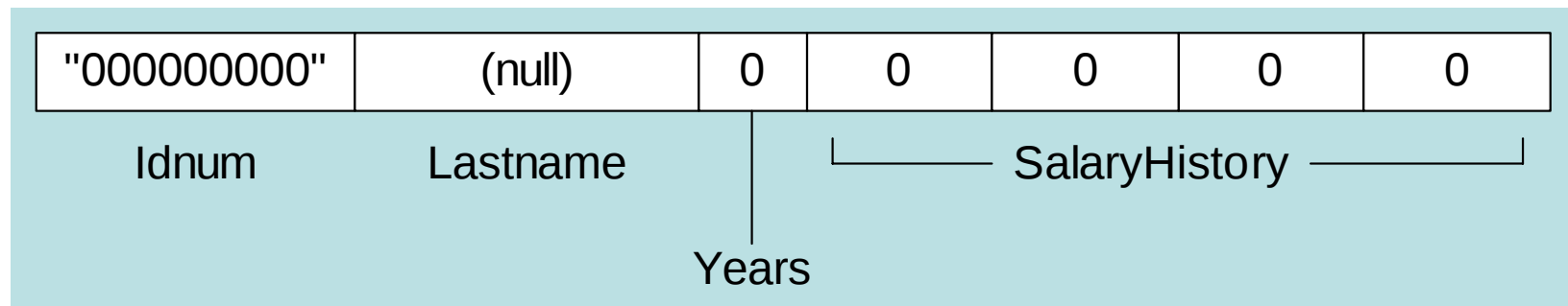
```

mov dx,worker.Years
mov worker.SalaryHistory,20000 ;first salary
mov worker.SalaryHistory+4,30000 ;second salary
mov edx,OFFSET worker.LastName

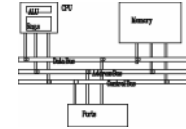
mov esi,OFFSET worker
mov ax,(Employee PTR [esi]).Years

mov ax,[esi].Years ; invalid operand (ambiguous)

```



Looping through an array of points

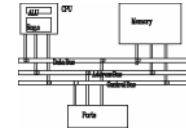


Sets the X and Y coordinates of the **AllPoints** array to sequentially increasing values (1,1), (2,2), ...

```
.data
NumPoints = 3
AllPoints COORD NumPoints DUP(<0,0>)

.code
    mov edi,0           ; array index
    mov ecx,NumPoints   ; loop counter
    mov ax,1            ; starting X, Y values
L1:
    mov (COORD PTR AllPoints[edi]).X,ax
    mov (COORD PTR AllPoints[edi]).Y,ax
    add edi,TYPE COORD
    inc ax
    Loop L1
```

Example: displaying the system time



- Retrieves and displays the system time at a selected screen location using Windows functions.
- How to obtain the system time?

GetLocalTime

The **GetLocalTime** function retrieves the current local date and time.

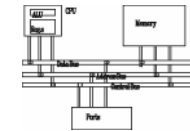
```
void GetLocalTime(  
    LPSYSTEMTIME lpSystemTime  
);
```

Parameters

lpSystemTime

[out] Pointer to a [SYSTEMTIME](#) structure to receive the current local date and time.

SYSTEMTIME structure

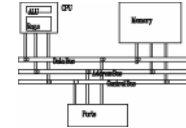


SYSTEMTIME

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME {  
    WORD wYear;  
    WORD wMonth;  
    WORD wDayOfWeek;  
    WORD wDay;  
    WORD wHour;  
    WORD wMinute;  
    WORD wSecond;  
    WORD wMilliseconds;  
} SYSTEMTIME,  
*PSYSTEMTIME;
```

SYSTEMTIME structure in assembly



SYSTEMTIME STRUCT

wYear WORD ?

wMonth WORD ?

wDayOfWeek WORD ?

wDay WORD ?

wHour WORD?

wMinute WORD ?

wSecond WORD ?

wMilliseconds WORD ?

SYSTEMTIME ENDS

- ## SetConsoleCursorPosition

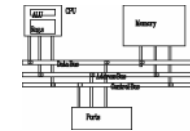
```

BOOL SetConsoleCursorPosition(
    HANDLE hConsoleOutput,
    COORD dwCursorPosition
);

```

```
typedef struct _COORD {
    SHORT X;
    SHORT Y;
} COORD,
*PCOORD;
```

Get standard console output handle



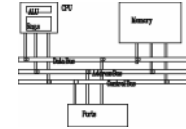
GetStdHandle

The **GetStdHandle** function retrieves a handle for the standard input, standard output, or standard error device.

```
HANDLE GetStdHandle(  
    DWORD nStdHandle  
);
```

Value	Meaning
STD_INPUT_HANDLE (DWORD)-10	Handle to the standard input device. Initially, this is a handle to the console input buffer, CONIN\$.
STD_OUTPUT_HANDLE (DWORD)-11	Handle to the standard output device. Initially, this is a handle to the active console screen buffer, CONOUT\$.
STD_ERROR_HANDLE (DWORD)-12	Handle to the standard error device. Initially, this is a handle to the active console screen buffer, CONOUT\$.

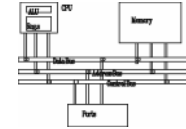
Example: displaying the system time



- Uses a Windows API call to get the standard console output handle. **SetConsoleCursorPosition** positions the cursor. **GetLocalTime** gets the current time of day:

```
.data
sysTime SYSTEMTIME <>
XYPos COORD <10,5>
consoleHandle DWORD ?
.code
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov consoleHandle,eax
INVOKE SetConsoleCursorPosition,
      consoleHandle, XYPos
INVOKE GetLocalTime, ADDR sysTime
```

Example: displaying the system time

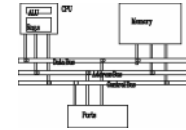


- Display the time using library calls:

```
mov     edx,OFFSET TheTimeIs ; "The time is "  
call    WriteString  
movzx   eax,sysTime.wHour    ; hours  
call    WriteDec  
mov     edx,offset colonStr  ; ":"  
call    WriteString  
movzx   eax,sysTime.wMinute  ; minutes  
call    WriteDec  
mov     edx,offset colonStr  ; ":"  
call    WriteString  
movzx   eax,sysTime.wSecond  ; seconds  
call    WriteDec
```

DEMO!

Nested structures



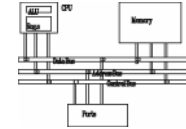
- Define a structure that contains other structures.
- Used nested braces (or brackets) to initialize each **COORD** structure.

```
Rectangle STRUCT
    UpperLeft COORD <>
    LowerRight COORD <>
Rectangle ENDS
```

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS
```

```
.code
rect1 Rectangle { {10,10}, {50,20} }
rect2 Rectangle < <10,10>, <50,20> >
```

Nested structures



- Use the dot (.) qualifier to access nested fields.
- Use indirect addressing to access the overall structure or one of its fields

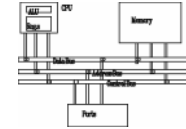
```
mov rect1.UpperLeft.X, 10
mov esi,OFFSET rect1
mov (Rectangle PTR [esi]).UpperLeft.Y, 10

// use the OFFSET operator
mov edi,OFFSET rect2.LowerRight
mov (COORD PTR [edi]).X, 50
mov edi,OFFSET rect2.LowerRight.X
mov WORD PTR [edi], 50
```

The diagram shows a GPU block on the left containing 'API' and 'Image' sub-components. A Memory block is on the right. A central horizontal bus connects them, with labels 'Data Line', 'H.264 Line', and 'Control Line' indicating different data paths. A 'Port' block is connected to the bottom of the bus.

-
- A diagram on a grid showing a path starting from a shaded square, moving up, then right, then up again, ending with an arrow pointing up.

Example: Drunkard's walk



- Uses a nested structure to accumulate path data as the simulation is running
- Uses a multiple branch structure to choose the direction

```
WalkMax = 50
```

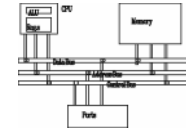
```
DrunkardWalk STRUCT
```

```
    path COORD WalkMax DUP(<0,0>)
```

```
    pathsUsed WORD 0
```

```
DrunkardWalk ENDS
```

Example: Drunkard's walk



```
.data
```

```
aWalk DrunkardWalk <>
```

```
.code
```

```
main PROC
```

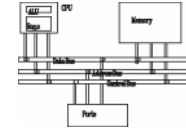
```
    mov esi,offset aWalk
```

```
    call TakeDrunkenWalk
```

```
    exit
```

```
main ENDP
```

Example: Drunkard's walk



TakeDrunkenWalk PROC

LOCAL currX:WORD, currY:WORD

pushad

; Point EDI to the array of COORD objects.

mov edi,esi

add edi,OFFSET DrunkardWalk.path

mov ecx,WalkMax ; loop counter

mov currX,StartX ; current X-location

mov currY,StartY ; current Y-location

Again:

```
; Insert current location in array.  
mov ax,currX  
mov (COORD PTR [edi]).X,ax  
mov ax,currY  
mov (COORD PTR [edi]).Y,ax  
INVOKE DisplayPosition, currX, currY
```

```
mov eax,4 ; choose a direction (0-3)  
call RandomRange
```

```
.IF eax == 0 ; North  
dec currY  
.ELSEIF eax == 1 ; South  
inc currY  
.ELSEIF eax == 2 ; West  
dec currX  
.ELSE ; East (EAX = 3)  
inc currX  
.ENDIF
```

next:

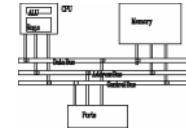
```
add edi,TYPE COORD ; point to next COORD  
loop Again
```

```
finish:
    mov ax,WalkMax ; count the steps taken
    sub ax,cx
    mov (DrunkardWalk PTR [esi]).pathsUsed, ax
    popad
    ret
TakeDrunkenWalk ENDP
```

```
DisplayPosition PROC currX:WORD, currY:WORD
.data
commaStr BYTE ", ",0
.code
    pushad
    movzx eax,currX ; current X position
    call WriteDec
    mov edx,OFFSET commaStr ; ", " string
    call WriteString
    movzx eax,currY ; current Y position
    call WriteDec
    call Crlf
    popad
    ret
DisplayPosition ENDP
```

DEMO!

Improved version



```
DisplayPosition PROC xy:COORD
```

```
.data
```

```
consoleHandle DWORD ?
```

```
.code
```

```
    pushad
```

```
    call Cclrscr
```

```
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
```

```
    mov consoleHandle, eax
```

```
    INVOKE SetConsoleCursorPosition, consoleHandle, xy
```

```
    mov al, '0'
```

```
    call WriteChar
```

```
    mov eax, 100
```

```
    call Delay
```

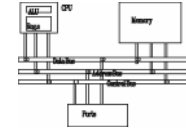
```
    popad
```

```
    ret
```

```
DisplayPosition ENDP
```

DEMO!

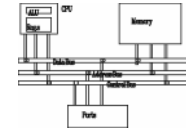
Declaring and using unions



- A union is similar to a structure in that it contains multiple fields
- All of the fields in a union begin at the same offset; size determined by the longest field
 - (differs from a structure)
- Provides alternate ways to access the same data
- Syntax:

```
unionname UNION  
  
union-fields  
  
unionname ENDS
```

Integer union example



The Integer union consumes 4 bytes (equal to the largest field)

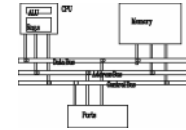
```
Integer UNION  
  D DWORD 0  
  W WORD 0  
  B BYTE 0  
Integer ENDS
```

D, W, and B are often called variant fields.

Integer can be used to define data:

```
.data  
val1 Integer <12345678h>  
val2 Integer <100h>  
val3 Integer <>
```

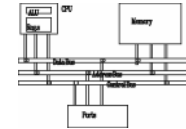
Integer union example



The variant field name is required when accessing the union:

```
mov val3.B, al
mov ax, val3.W
add val3.D, eax
```

Union inside a structure



An Integer union can be enclosed inside a **FileInfo** structure:

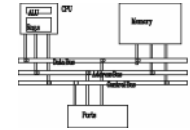
```
Integer UNION
    D DWORD 0
    W WORD 0
    B BYTE 0
Integer ENDS
```

```
FileInfo STRUCT
    FileID Integer <>
    FileName BYTE 64 DUP(?)
FileInfo ENDS
```

```
.data
myFile FileInfo <>
.code
mov myFile.FileID.W, ax
```

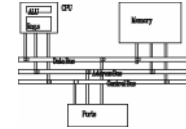
```
FileInfo STRUCT
    UNION FileID
        D DWORD 0
        W WORD 0
        B BYTE 0
    ENDS
    FileName BYTE 64 DUP(?)
FileInfo ENDS
```

Macros



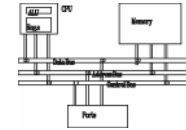
- Introducing Macros
- Defining Macros
- Invoking Macros
- Macro Examples
- Nested Macros
- Example Program: Wrappers

Introducing macros



- A macro is a named block of assembly language statements.
- Once defined, it can be invoked (called) one or more times.
- During the assembler's preprocessing step, each macro call is expanded into a copy of the macro.
- The expanded code is passed to the assembly step, where it is checked for correctness.
- Resulted code is usually faster than real function call, but bigger.

mNewLine macro example



This is how you define and invoke a simple macro.

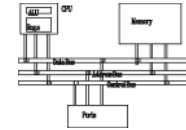
```
mNewLine MACRO                ; define the macro
    call crlf
ENDM

.data

.code
mNewLine                        ; invoke the macro
```

The assembler will substitute "**call crlf**" for "**mNewLine**".

Defining macros



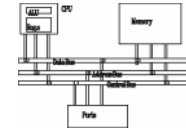
- A macro must be defined before it can be used.
- Parameters are optional. Each parameter follows the rules for identifiers. It is a string that is assigned a value when the macro is invoked.
- Use mMacro to distinguish from functions
- Syntax:

```
macroname MACRO [parameter-1,  
parameter-2, ... ]
```

```
statement-list
```

```
ENDM
```

mPutChar macro



Writes a single character to standard output.

Definition:

```
mPutchar MACRO char:REQ  
    push eax  
    mov al,char  
    call WriteChar  
    pop eax  
ENDM
```

Invocation:

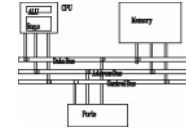
```
.code  
mPutchar 'A'
```

Expansion:

```
1      push eax  
1      mov al,'A'  
1      call WriteChar  
1      pop eax
```

viewed in
the listing
file

Invoking macros

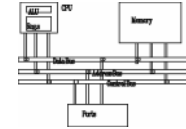


- When you invoke a macro, each argument you pass matches a declared parameter.

macroname arg-1, arg-2, . . .

- Each parameter is replaced by its corresponding argument when the macro is expanded.
- When a macro expands, it generates assembly language source code.
- Arguments are treated as simple text. The number of arguments might not match the number of parameters. Too many, drop and warning; Too few, left with blanks.

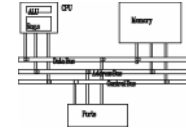
mWriteStr macro



Provides a convenient way to display a string, by passing the string name as an argument.

```
mWriteStr MACRO buffer
    push edx
    mov  edx,OFFSET buffer
    call WriteString
    pop  edx
ENDM
.data
str1 BYTE "Welcome!",0
.code
mWriteStr str1
```

mWriteStr macro



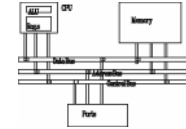
The expanded code shows how the str1 argument replaced the parameter named buffer:

```
mWriteStr MACRO buffer
    push edx
    mov  edx,OFFSET buffer
    call WriteString
    pop  edx
ENDM
```

↓

```
1  push  edx
1  mov   edx,OFFSET str1
1  call  WriteString
1  pop   edx
```

Invalid argument

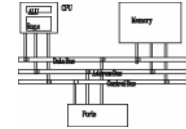


- If you pass an invalid argument, the error is caught when the expanded code is assembled.
- Example:

```
.code  
mPuchar 1234h
```

```
1    push  eax  
1    mov  al,1234h      ; error!  
1    call WriteChar  
1    pop   eax
```

Blank argument

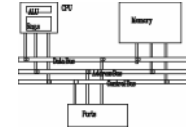


- If you pass a blank argument, the error is also caught when the expanded code is assembled.
- Example:

```
.code  
mPuchar
```

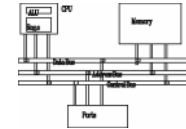
```
1      push  eax  
1      mov  al,  
1      call WriteChar  
1      pop  eax
```

Macro examples



- **mReadStr** - reads string from standard input
- **mGotoXY** - locates the cursor on screen
- **mDumpMem** - dumps a range of memory

mReadStr



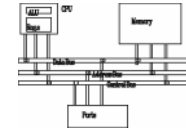
The **mReadStr** macro provides a convenient wrapper around **ReadString** procedure calls.

```
mReadStr MACRO varName
    push ecx
    push edx
    mov edx,OFFSET varName
    mov ecx,(SIZEOF varName) - 1
    call ReadString
    pop edx
    pop ecx
ENDM

.data
firstName BYTE 30 DUP(?)

.code
mReadStr firstName
```

mGotoXY

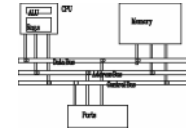


The **mGotoXY** macro sets the console cursor position by calling the **Gotoxy** library procedure.

```
mGotoxy MACRO X:REQ, Y:REQ
    push    edx
    mov     dh,Y
    mov     dl,X
    call    Gotoxy
    pop     edx
ENDM

...
mGotoxy 10, 20
mGotoxy row, col
mGotoxy ch, cl
mGotoxy dh, dl ; conflicts
```

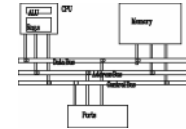
mDumpMem



The **mDumpMem** macro streamlines calls to the link library's **DumpMem** procedure.

```
mDumpMem MACRO address, itemCount, componentSize  
    push ebx  
    push ecx  
    push esi  
    mov esi, address  
    mov ecx, itemCount  
    mov ebx, componentSize  
    call DumpMem  
    pop esi  
    pop ecx  
    pop ebx  
ENDM
```

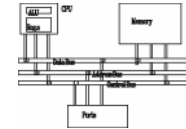
mDumpMem invocation



mDumpMem OFFSET array, 8, 4

mDumpMem OFFSET array, \ ; array offset
LENGTHOF array, \ ; number of units
TYPE array ; size of a unit

mWrite



The **mWrite** macro writes a string literal to standard output. It is a good example of a macro that contains both code and data.

```
mWrite MACRO text
    LOCAL string
    .data                ;; data segment
    string BYTE text,0   ;; define local string
    .code                ;; code segment
    push edx
    mov  edx,OFFSET string
    call writestring
    pop  edx
ENDM
```

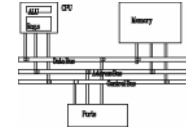
The **LOCAL** directive prevents string from becoming a global label.

```
mWrite "This is the first string"  
mWrite "This is the second string"
```

```
.data  
??0000 BYTE "This is the first string",0  
.code  
push edx  
mov  edx,OFFSET ??0000  
call Writestring  
pop  edx  
.data  
??0001 BYTE "This is the second string",0  
.code  
push edx  
mov  edx,OFFSET ??0001  
call Writestring  
pop  edx
```

Unique labels allow us to call this macro multiple times.

Nested macros



- The **mWriteLn** macro contains a nested macro (a macro invoked by another macro).

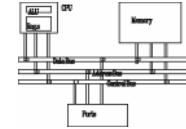
```
mWriteLn MACRO text
    mWrite text
    call Crlf
ENDM
```

```
mWriteLn "My Sample Macro Program"
```

```
2  .data
2  ??0002 BYTE "My Sample Macro Program",0
2  .code
2  push edx
2  mov  edx,OFFSET ??0002
2  call writestring
2  pop  edx
1  call Crlf
```

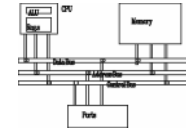
↑ nesting level

Example program: wrappers



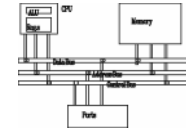
- Demonstrates various macros from this chapter
- Shows how macros can simplify argument passing
- View the [source code](#)

Conditional-assembly directives



- Checking for Missing Arguments
- Default Argument Initializers
- Boolean Expressions
- IF, ELSE, and ENDIF Directives
- IFIDN and IFIDNI Directives
- Special Operators
- Macro Functions

Checking for missing arguments

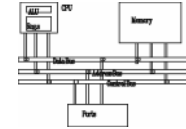


- The **IFB** directive returns true if its argument is blank. For example:

```
IFB <row>          ;; if row is blank,  
    EXITM          ;; exit the macro  
ENDIF
```

```
mWriteStr MACRO buffer  
    push edx  
    mov  edx,OFFSET buffer  
    call WriteString  
    pop  edx  
ENDM
```

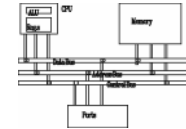
mWriteString example



Display a message during assembly if the string parameter is empty. No code is generated for this macro.

```
mWriteStr MACRO string
    IFB <string>
        ECHO -----
        ECHO * Error: parameter missing in mWriteStr
        ECHO * (no code generated)
        ECHO -----
        EXITM
    ENDIF
    push edx
    mov edx,OFFSET string
    call WriteString
    pop edx
ENDM
```

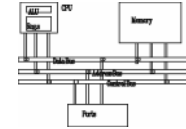
Default argument initializers



- A default argument initializer automatically assigns a value to a parameter when a macro argument is left blank. For example, **mWriteLn** can be invoked either with or without a string argument:

```
mWriteLn MACRO text:=<" ">
    mWrite text
    call CrLf
ENDM
.code
mWriteLn "Line one"
mWriteLn
mWriteLn "Line three"
```

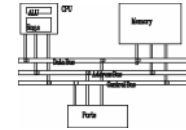
IF, ELSE, and ENDIF directives



A block of statements is assembled if the Boolean expression evaluates to true. An alternate block of statements can be assembled if the expression is false.

```
IF boolean-expression  
    statements  
[ELSE  
    statements]  
ENDIF
```

Boolean expressions

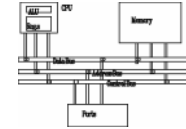


A Boolean expression can be formed using the following operators:

- LT - Less than
- GT - Greater than
- EQ - Equal to
- NE - Not equal to
- LE - Less than or equal to
- GE - Greater than or equal to

Only assembly-time constants may be compared using these operators.

Simple example



The following **IF** directive permits two **MOV** instructions to be assembled if a constant named **RealMode** is equal to 1:

```
IF RealMode EQ 1  
    mov ax,@data  
    mov ds,ax  
ENDIF
```

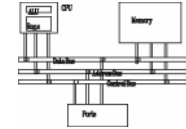
RealMode can be defined in the source code any of the following ways:

RealMode = 1

RealMode EQU 1

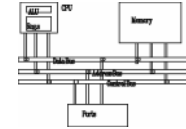
RealMode TEXTEQU 1

mGotoxyConst



```
mGotoxyConst MACRO X:REQ, Y:REQ
    LOCAL ERRS ;; local constant
    ERRS = 0
    IF (X LT 0) OR (X GT 79)
        ECHO Warning: X to mGotoxy is out of range.
        ERRS = 1
    ENDIF
    IF (Y LT 0) OR (Y GT 24)
        ECHO Warning: Y to mGotoxy is out of range.
        ERRS = ERRS + 1
    ENDIF
    IF ERRS GT 0      ;; if errors found,
        EXITM      ;; exit the macro
    ENDIF
    push edx
    mov  dh,Y
    mov  dl,X
    call Gotoxy
    pop  edx
ENDM
```

The **IFIDN** and **IFIDNI** directives

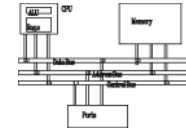


- **IFIDN** compares two symbols and returns true if they are equal (case-sensitive)
- **IFIDNI** also compares two symbols, using a case-insensitive comparison
- Syntax:

```
IFIDNI <symbol>, <symbol>  
    statements  
ENDIF
```

Can be used to prevent the caller of a macro from passing an argument that would conflict with register usage inside the macro.

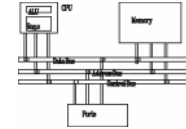
IFIDNI example



Prevents the user from passing **EDX** as the second argument to the **mReadBuf** macro:

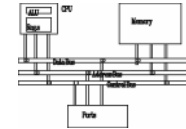
```
mReadBuf MACRO bufferPtr, maxChars
    IFIDNI <maxChars>,<EDX>
        ECHO Warning: Second argument cannot be EDX
        ECHO *****
        EXITM
    ENDIF
    .
    .
ENDM
```

Special operators



- The substitution (&) operator resolves ambiguous references to parameter names within a macro.
- The expansion operator (%) expands text macros or converts constant expressions into their text representations.
- The literal-text operator (<>) groups one or more characters and symbols into a single text literal. It prevents the preprocessor from interpreting members of the list as separate arguments.
- The literal-character operator (!) forces the preprocessor to treat a predefined operator as an ordinary character.

Substitution (&)



Text passed as **regName** is substituted into the literal string definition:

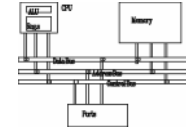
```
ShowRegister MACRO regName
.data
tempStr BYTE " &regName=", 0
...
    mov eax, regName
...
ENDM

ShowRegister EDX      ; invoke the macro
```

Macro expansion:

```
tempStr BYTE " EDX=", 0
```

Expansion (%)



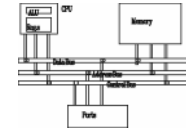
Forces the evaluation of an integer expression. After the expression has been evaluated, its value is passed as a macro argument:

```
mGotoXY %(5 * 10),%(3 + 4)
```

The preprocessor generates the following code:

```
1 push edx
1 mov dl,50
1 mov dh,7
1 call Gotoxy
1 pop edx
```

MUL32



MUL32 MACRO op1, op2, product

IFIDNI <op2>,<EAX>

LINENUM TEXTEQU %(@LINE)

ECHO -----

% ECHO * Error on line LINENUM: EAX cannot be the
ECHO * second argument for the MUL32 macro.

ECHO -----

EXITM

ENDIF

push eax

mov eax,op1

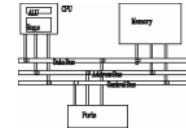
mul op2

mov product,eax

pop eax

ENDM

Literal-text (<>)

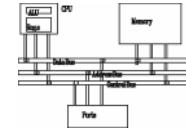


The first macro call passes three arguments. The second call passes a single argument:

```
mWrite "Line three", 0dh, 0ah
```

```
mWrite <"Line three", 0dh, 0ah>
```

Literal-character (!)



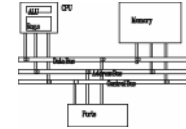
The following declaration prematurely ends the text definition when the first > character is reached.

```
BadYValue TEXTEQU Warning: <Y-coordinate is > 24>
```

The following declaration continues the text definition until the final > character is reached.

```
BadYValue TEXTEQU <Warning: Y-coordinate is !> 24>
```

Macro functions

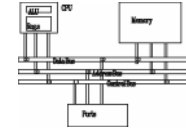


- A macro function returns an integer or string constant
- The value is returned by the **EXITM** directive
- Example: The **IsDefined** macro acts as a wrapper for the **IFDEF** directive.

```
IsDefined MACRO symbol
    IFDEF symbol
        EXITM <-1>                ;; True
    ELSE
        EXITM <0>                  ;; False
    ENDIF
ENDM
```

Notice how the assembler defines True and False.

Macro functions

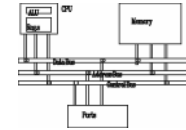


- When calling a macro function, the argument(s) must be enclosed in parentheses

```
IF IsDefined( RealMode )  
    mov ax,@data  
    mov ds,ax  
ENDIF
```

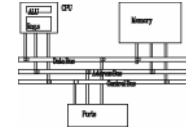
```
IF IsDefined( RealMode )  
    INCLUDE Irvine16.inc  
ELSE  
    INCLUDE Irvine32.inc  
ENDIF
```

Defining repeat blocks



- WHILE Directive
- REPEAT Directive
- FOR Directive
- FORC Directive
- Example: Linked List

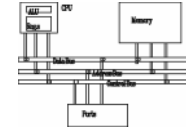
WHILE directive



- The **WHILE** directive repeats a statement block as long as a particular constant expression is true.
- Syntax:

```
WHILE constExpression  
    statements  
ENDM
```

WHILE example

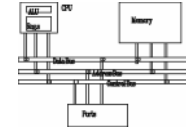


Generates Fibonacci integers between 1 and 1000h at assembly time:

```
.data
val1 = 1
val2 = 1
DWORD val1                ; first two values
DWORD val2
val3 = val1 + val2

WHILE val3 LT 1000h
    DWORD val3
    val1 = val2
    val2 = val3
    val3 = val1 + val2
ENDM
```

REPEAT directive

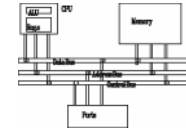


- The **REPEAT** directive repeats a statement block a fixed number of times.
- Syntax:

```
REPEAT constExpression  
  
    statements  
  
ENDM
```

ConstExpression, an unsigned constant integer expression, determines the number of repetitions.

REPEAT example

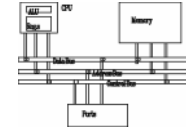


The following code generates 100 integer data definitions in the sequence 10, 20, 30, . . .

```
iVal = 10  
REPEAT 100  
    DWORD iVal  
    iVal = iVal + 10  
ENDM
```

How might we assign a data name to this list of integers?

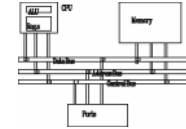
FOR directive



- The **FOR** directive repeats a statement block by iterating over a comma-delimited list of symbols.
- Each symbol in the list causes one iteration of the loop.
- Syntax:

```
FOR parameter, <arg1, arg2, arg3, . . . >  
statements  
ENDM
```

FOR example



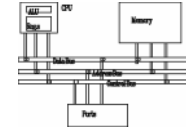
The following Window structure contains frame, title bar, background, and foreground colors. The field definitions are created using a **FOR** directive:

```
Window STRUCT
    FOR color,<frame,titlebar,background,foreground>
        color DWORD ?
    ENDM
Window ENDS
```

Generated code:

```
Window STRUCT
    frame DWORD ?
    titlebar DWORD ?
    background DWORD ?
    foreground DWORD ?
Window ENDS
```

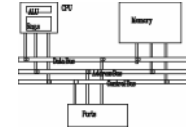
FORC directive



- The **FORC** directive repeats a statement block by iterating over a string of characters. Each character in the string causes one iteration of the loop.
- Syntax:

```
FORC parameter, <string>  
statements  
ENDM
```

FORC example



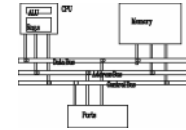
Suppose we need to accumulate seven sets of integer data for an experiment. Their label names are to be Group_A, Group_B, Group_C, and so on. The FORC directive creates the variables:

```
FORC code, <ABCDEFG>  
  Group_&code WORD ?  
ENDM
```

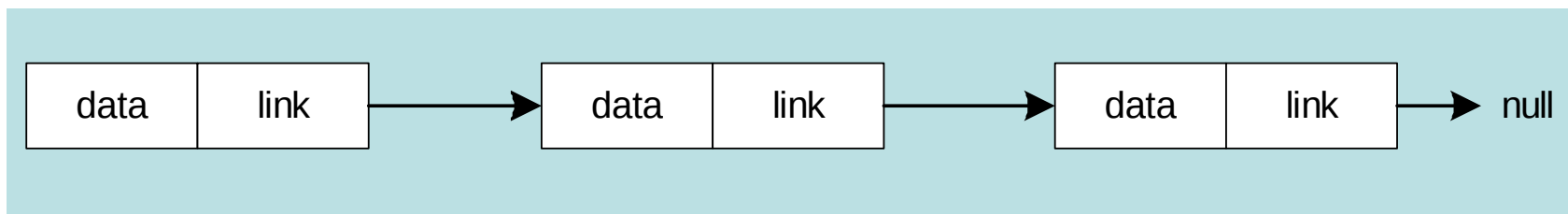
Generated code:

```
Group_A WORD ?  
Group_B WORD ?  
Group_C WORD ?  
Group_D WORD ?  
Group_E WORD ?  
Group_F WORD ?  
Group_G WORD ?
```

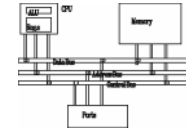
Example: linked List



- We can use the **REPEAT** directive to create a singly linked list at assembly time.
- Each node contains a pointer to the next node.
- A null pointer in the last node marks the end of the list



Linked list



- Each node in the list is defined by a **ListNode** structure:

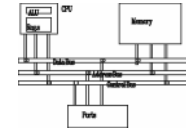
```
ListNode STRUCT
    NodeData DWORD ?    ; the node's data
    NextPtr  DWORD ?    ; pointer to next node
ListNode ENDS
```

```
TotalNodeCount = 15
```

```
NULL = 0
```

```
Counter = 0
```

Linked list

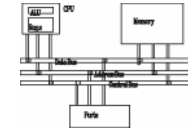


- The **REPEAT** directive generates the nodes.
- Each **ListNode** is initialized with a counter and an address that points 8 bytes beyond the current node's location:

```
.data
LinkedList LABEL PTR ListNode
REPEAT TotalNodeCount
    Counter = Counter + 1
    ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
ENDM
ListNode <0,0>
```

The value of \$ does not change—it remains fixed at the location of the **LinkedList** label.

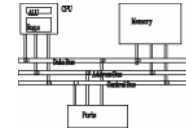
Linked list



The following hexadecimal values in each node show how each **NextPtr** field contains the address of its following node.

offset	contents	
00000000	00000001	
	00000008	← NextPtr
00000008	00000002	
	00000010	←
00000010	00000003	
	00000018	←
00000018	00000004	
	00000020	←
00000020	(etc.)	

Linked list



```
    mov  esi,OFFSET LinkedList
; Display the integers in the NodeData members.
NextNode:
; Check for the tail node.
    mov  eax,(ListNode PTR [esi]).NextPtr
    cmp  eax,NULL
    je   quit

    ; Display the node data.
    mov  eax,(ListNode PTR [esi]).NodeData
    call WriteDec
    call CrLf

    ; Get pointer to next node.
    mov  esi,(ListNode PTR [esi]).NextPtr
    jmp  NextNode
quit:
    exit
```