Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

- Macros
- Macros vs procedures
- Arrays
- Lookup tables
- Hash tables
- Jump tables

Macros

- A macro inserts a block of statements at various points in a program during assembly
- Substitutions made at compile time
  - **Not** a procedure—code is literally dumped into the program with each instantiation
  - Parameter names are substituted
  - Useful for tedious programming tasks

Macros: Generic Format

```
%macro MACRO_NAME   numargs
   Your Code ...
   ...%{1} ...
   ...%{2} ...
   Your Code ...
   JMP %%MyLabel
   Your Code ...
%%MyLabel:
   ...%{N} ...
  Your Code ...
   %endmacro
```

Local Variables in a Macro

- A local label is one that appears in the macro, but is not available outside the macro
- We use the **%%** prefix for defining a local label
  - If the label *MyLabel* in the previous example is not defined as local, the assembler will flag it with errors on the second and subsequent attempts to use the macro because there would be duplicate label names
- Macros can be placed in a separate file
  - use %include directive to include the file with external macro definitions into a program
  - no EXTERN statement is needed to access the macro statements that have been included

Macros

```
; Store into Result the signed result of X / Y
; Calculate Result = X / Y
; (all 16-bit signed integers)
; Modifies Registers AX,DX

%macro DIV16 3       ; Args: Result, X, Y
   MOV   AX, %{2}     ; Load AX with Dividend
   CWD               ; Extend Sign into DX
   IDIV  %{3}         ; Signed Division
   MOV   %{1}, AX     ; Store Quotient
%endmacro
```

Macros

```
; Example: Using the macro in a program
; Variable Section
   varX1      DW     20
   varX2      DW     4
   varR       RESW   1
; Code Section
   DIV16 word[varR], word[varX1], word[varX2]
```

;Will actually generate the following code inline in your program for every instantiation of the DIV16 macro (*You won't actually see this unless you debug the program*).

```
   MOV  AX, word[varX1]
   CWD
   IDIV word[varX2]
   MOV  [varR], AX
```


Macros vs. Procedures

```
Proc_1
       MOV    AX, 0
       MOV    BX, AX
       MOV    CX, 5
       RET

%macro Macro_1    0
        MOV   AX, 0
        MOV   BX, AX
        MOV   CX, 5
%endmacro
```

```
CALL Proc_1
…
CALL Proc_1
…
Macro_1
…
Macro_1
```

Macros vs. Procedures

- In the example the macro and procedure produce the same result
- The procedure definition generates code in your executable
- The macro definition does not produce any code
- Upon encountering *Macro_1* in your code, NASM assembles every statement between the *%macro* and *%endmacro* directives for *Macro_1* and produces that code in the output file
- At run time, the processor executes these instructions without the call/ret overhead because the instructions are themselves inline in your code

Macros vs. Procedures

- Advantage of using macros
  - execution of macro expansion is faster (no call and ret) than the execution of the same code implemented with procedures
- Disadvantages
  - assembler copies the macro code into the program at each macro invocation
  - if the number of macro invocations within the program is large then the program will be much larger than when using procedures

Arrays

- Collection of sequentially addressable equally sized data in memory
- Only first element is labeled
- Index from there given size of each array element

Arrays - declaring

```
;declare and initialize a five element byte array
MyByteArray   DB     00h,0afh,12,83,83h

;declare a 50 element word array
MyWordArray1  RESW   50
MyWordArray2  RESB   100
MyWordArray3  RESD   25

;declare and initialize a 16 element char array
MyCharArray   DB     "ECE390 Rocks!!!", 0
```

Arrays - accessing

```
/* C array example */

int intWordArray[25];

for (int i = 0; i < 25; i++) {
   intWordArray[i] = i * i;
}

/* This code segment declares a 25-element word
   array (50 bytes) and fills each element with
   the square of its index */
```


Arrays - accessing

```
;Assembly array example

intWordArray RESW   25

mov cx, 25
begin_loop:
   mov bx, cx
   imul ax, cx, cx
   dec bx
   shl bx
   mov [intWordArray + bx], ax
loop begin_loop
```

Arrays in 2-dimensions

0	1	2	3	4	5	6	7	8	9	A	B
C	D	E	F	10	11	12	13	14	15	16	17
18	19	1A	1B	1C	1D	1E	1F	20	21	22	23
24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37	38	39	3A	3B
3C	3D	3E	3F	40	41	42	43	44	45	46	47
48	49	4A	4B	4C	4D	4E	4F	50	51	52	53

- This is a 12 col by 7 row 2-D byte array
- It has 12 * 7 = 84 = 54h elements
- Addresses increase as you traverse a row
- Each row begins NUMCOLS addresses after the previous row
- Address of element (row, col) = row*NUMCOLS + col

```
NUMCOLS EQU      12
NUMROWS EQU      7
My2DArray RESB NUMCOLS * NUMROWS
```

```
mov bx, 4
imul bx, NUMCOLS
add bx, 6

mov al, [My2DArray + bx]
  ;[My2DArray + 54] à  AL
```


Lookup tables

- Lookup tables are special purpose arrays commonly used to convert one data form to another
- A lookup table is formed in memory as a list of data that is used to perform conversions

Lookup tables

- Consider movement in a 2-dimensional array by one element to an adjacent element (up, down, left, or right)
  - Array = Matrix of ROWSIZE x COLSIZE elements
  - Pos = Position in array = Row * COLSIZE + Column
  - Dir = Direction (UP=0, RIGHT=1, DOWN=2,LEFT=3)
- Slow and tedious original code compares each possible direction value
- 16 instructions

```
Mbegin
   CMP   byte[DIR], UP
   JE    .MoveUP
   CMP    byte[DIR], RIGHT
   JE    .MoveRT
   CMP    byte[DIR], DN
   JE    .MoveDN
   CMP    byte[DIR], LEFT
   JE    .MoveLT

.MoveUP
   SUB    byte[Pos], COLSIZE
   JMP   .MDone
.MoveRT
   ADD    byte[Pos],1
   JMP   .MDone
.MoveDN
   ADD    byte[Pos],COLSIZE
   JMP   .MDone
.MoveLT
   SUB    byte[Pos],1
   JMP   .MDone
.MDone
```


Lookup tables

- Fast and Compact Table-lookup Code
- Eliminate conditional jumps by defining a table of movements indexed by the Direction
- 4 instructions + 4 word-sized variables

```
Movement
  dw -COLSIZE            ; Movement[0] == Go UP
  dw +1                  ; Movement[1] == Go RIGHT
  dw +COLSIZE            ; Movement[2] == Go DOWN
  dw -1                  ; Movement[3] == Go LEFT
Mbegin
  MOV BX, [DIR]          ; Direction = {0..3}
  SHL BX, 1              ; Index words, not bytes
  MOV AX, [Movement + BX]
  ADD [Pos], AX          ; Position += Movement
MDone:  Finished!
```

Lookup tables

```
DAYS
    PUSH DX
    PUSH SI

    MOV SI, DTAB    ;si points to table
    XOR AH, AH      ;clear AH
    ADD AX, AX      ;double AX
    ADD SI, AX      ;index to right day
    MOV DX, [SI]    ;get string address


    MOV AH, 09h     ;display char magic
    INT 21h

    POP SI
    POP DX
RET
```

```
SUN DB   'Sunday$'
MON DB   'Monday$'
TUE DB   'Tuesday$'
WED DB   'Wednesday$'
THU DB   'Thursday$'
FRI DB   'Friday$'
SAT DB   'Saturday$'

;Lookup table contains addresses of
;day of week strings.  It converts
;0 to "Sunday", 1 to "Monday", etc.

DTAB DW  SUN, MON, TUE,
         WED, THU, FRI, SAT
```

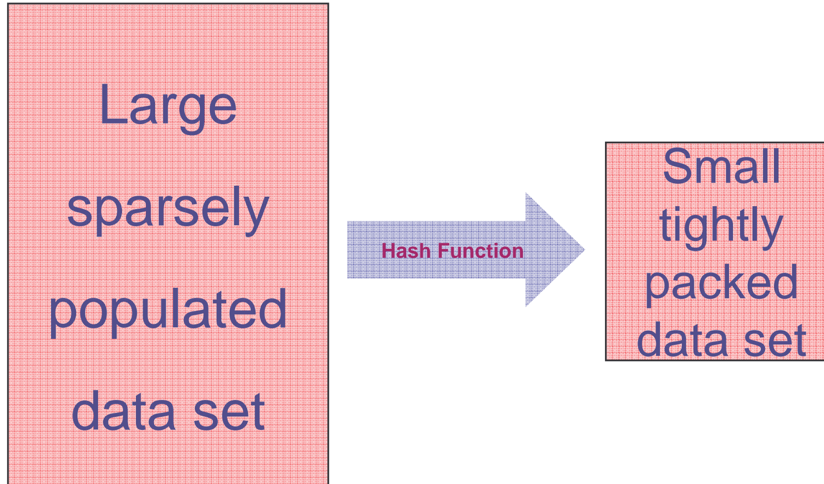
Converts numbers 0 to 6 to days of the week
The number is passed in AL register

Hash functions

- Consider using a lookup table when the input can span a large range of values but is sparsely populated.
- Example:
  - Count incoming packets from each of 100 hosts scattered throughout the Internet using the 32-bit IP source address
  - A Table-Lookup would require 4 billion table entries, most of which would be empty
- This is what we call "A Bad Thing."

Hash functions

- Hash functions re-map large sparsely populated data sets (like the set of all IP addresses) into smaller more manageable data sets.



Hash functions

- A Hash Function, H(x), can be used to re-map the four-byte (W.X.Y.Z) IP address to asmaller (8-bit) value.
- H(W.X.Y.Z)=W xor X xor Y xor Z

H(0.0.0.0) = 0

H(128.174.5.58) = 17

H(224.2.4.88) = 190

H(141.142.44.33) = 14

Hash Table

Index	IP address
0	0.0.0.0
…	
14	141.142.44.33
…	
17	128.174.5.58
…	
190	224.2.4.88
…	

Hash functions

- Collisions can occur when two inputs map to the same output
  - H(128.174.112.1) and H(128.174.1.112) for example
- Collision resolution—Linear Probing
- Use next-available location in the hash table
- Add an extra column in the table for identifiers or tags to distinguish different entries in the table

```
tag = IP_addressTag
index = h(IP_address);
if (ht[index].tag == IP_addressTag) then
  <Done>
else
  <search the hash table until the entry with matching
  Tag is found>
end if
```


Jump tables

- Suppose table entries are pointers to functions
- BX holds a command in the range of 0..N
- Let BX index a function

```
JMP [Jtable+BX]  ; Jump to function
Jtable
          dw     Funct0
          dw     Funct1
          ..
          dw     FunctN

Funct0
Funct1
FunctN
```

Jump tables

```
Jtable
            dw    Funct0
            dw    Funct1
            dw    Funct2
final_result          dw        0
………
  shl    bx
  call   [Jtable+bx]             ; jump to a function
  mov    [final_result], ax      ; a function outputs
                                 ; results in AX
……….
Funct0:
            < set1 of operations >
            ret
Funct1:
            <set2 of operations>
            ret
Funct2:
            <set3 of operations>
            ret
```

- Assume that a result of a computation gives rise to one of a finite number of outcomes
- Depending on the outcome a different function (block of code) must be executed
- Jump table can be used to keep function pointers
- BX indexes a function