KOM206 Laboratory (Dept. of Computer Science, IPB) Module 2 - Feb 18, 2015

Data Definitions and Stack Operations

Lecturers: ARD, SWJ *Lab tutors: VDE, WUL*

Objectives

1. Able to define data byte per byte (DB), data word per word (DW)
2. Able to utilize the stack memory via its operations

1 Theoretical Review

In a segmented memory model, there are 3 main segments (also called: *sections*): text, data and stack. Each segment is used to contain a specific type of data.

1. **text segment**: This segment defines an area in memory that stores the instruction codes.
2. **data segment**: This segment is used to store the data elements. The data segment starts with the .data segment, which contains initialized data. Above that is the .bss segment; bss is for "block started by symbol". The .bss segment contains data that are statically allocated in a process, but is not stored in the executable file. Instead this data are allocated when the process is loaded into memory. The initial contents of this .bss segment are all 0 bits.
3. **stack segment**: The base address of the stack is contained in the stack segment (SS) register. A general-purpose register (GPR) called the stack pointer (E)SP contains the address of the current top of stack. In the memory, a stack is an array-like LIFO data structure. This segment is mostly used for keeping track of function calls, parameters, local variables, and return addresses.

The arrangement of the various memory segments is as follows. At the lowest address we have the text segment. The text segment does not typically need to grow, so the data segment is placed immediately above the text segment. Above these two segments are the heap and stack segments. The stack is mapped to the highest address of a process. We remark that the aforementioned simple memory layout is not entirely accurate. There are shared object files which can be mapped into a process after the program is loaded which will result in regions in the heap range being used to store instructions and data. This region is also used for mapping shared memory regions into a process.

1.1 Data Definitions

The assembler recognizes a basic set of intrinsic data types, which describe types in terms of their size, whether they are signed, and whether they are integers or reals. Some of the intrinsic data types are BYTE (8-bit unsigned integer), SBYTE (8-bit signed integer), WORD (16-bit unsigned integer), DWORD (32-bit unsigned integer), QWORD (64-bit integer), REAL4 (32-bit IEEE short real), REAL8 (64-bit IEEE long real). The assembler only evaluates the sizes of operands, meaning that we can only assign variables of type DWORD, SDWORD and REAL4 to a 32-bit integer.

A data definition statement sets aside (allocates) storage in the memory for a variable. Listing 1 shows the details of data definitions. We remark the followings:

- Each byte of character is stored as its ASCII value in hexadecimal

- Each decimal value is automatically converted to its 16-bit binary equivalent and stored as a hexadecimal number
- The x86 processors store and retrieve data from memory using little-endian order (low to high).
- Negative numbers are converted to its 2's complement representation
- Short and long floating-point numbers are represented using 32 or 64 bits, respectively

Listing 1: Data definition snippet

```
1  section .data
2  ; for allocating Storage Space for Initialized Data
3  ; [variable-name] data-definition-directive initial-value [,initial-value,...]
4
5  choice          db  'y'      ; db, define byte, allocates 1 byte
6  zero            db   0       ; even if it is zero. one initializer is required
7  random_value    db   ?       ; assigned a random value using  the ? symbol
8
9  number          dw  12345        ; dw, define word, allocates 2 bytes
10 neg_number      dw  -12345
11 w_list          dw  1,2,3,4,5
12
13 real_number1    dd  1.234    ; dd, define doubleword, allocates 4 bytes
14 real_number2    dd -1.2      ; short real
15
16 big_number      dq  123456789    ; dq, define quadword, allocates 8 bytes
17 real_number2    dq  123.456      ;
18 real_number3    dq  3.2E-260     ; long real
19
20 bigger_number   dt  99999999999 ; dt, define ten bytes, allocates 10 bytes
21 real_number9    dt  4.6E+4096    ; extended-precision real
22
23 stars times 9   db  '*'          ; the 'times' directive allows
24                                  ; multiple initializations to the same value.
25
26 ; using multiple initializers
27 ; If multiple initializers are used in the same data definition,
28 ; its label refers only to the offset of the first initializer.
29 list    db  10,20,30,40
30 list2   db  0Ah, 20h, 'A', 22h
31
32 ; Defining Strings of characters
33 ; Enclose strings in single or double quotation marks.
34 ; End a string with a null byte (containing 0); a null-terminated string.
35 ; Each character of a string uses a byte of storage.
36 ; However, strings are an exception to the rule that byte values must
37 ; be separated by commas.
38 greeting1   db  'Good afternoon',0
39 greeting2   db  'Welcome to the Bogor", 0dh,0ah
40            db  'Wish you luck', 0dh,0ah,0
41
42 section .bss
```

```
43 ; for allocating Storage Space for Uninitialized Data
44 ; [variable-name] data-reservation-directive [#units of space to be reserved]
45
46 b       resb    100     ; resb, reserve a byte
47 w       resw    1       ; resw, reserve a word
48 d       resd    10      ; resd, reserve a doubleword
49 q       resq    1       ; resq, reserve a quadword
50 t       rest    1       ; rest, Reserve a Ten Bytes
```

1.2 Stack Operations

1.2.1 PUSH and POP

a PUSH instruction places a data element on top of the stack. Whereas, a POP instruction removed a data element from the top of the stack. Only words or doublewords could be saved into the stack, not a byte.

A stack builds toward lower addresses. When a data item is pushed onto the stack, the (E)SP register is first decremented, then the data item is stored at the new top of stack. When a data item is popped off the stack, it is first stored in the destination address, then the (E)SP register is incremented to point to the new top of stack. The operand size determines the amount that the stack pointer is incremented or decremented. For example, if the operand size is 16 bits, then the SP register is incremented/decremented by 2; if the operand size is 32 bits, then the ESP register is incremented/decremented by 4.

```
1      ; with immediate operands and the eax register
2      mov eax,10h
3      push eax
4      mov ax,01h
5      push eax
6
7      mov eax,77h
8
9      pop eax
10     pop eax
11
12     ; with variables
13     ;push byte [b]; error: invalid size for operand 1
14     push word [a]
15     pop word [c]
```

1.2.2 PUSHAD and POPAD

The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (value before executing PUSHAD), EBP, ESI, and EDI. The POPAD instruction pops the same registers off the stack in reverse order. The corresponding instructions for 16-bit operations are PUSHA and POPA.

```
1      mov eax, 0x12345678
2      mov ecx, 0xabcdef12
3
4      pushad ;push all of the 32-bit general-purpose registers on the stack
5
6      mov eax, 0x0
```

```
7       mov ecx, 0x0
8
9       popad ;pop all of the 32-bit general-purpose registers on the stack
```

1.2.3 PUSHFD and POPFD

The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS. Notice that the MOV instruction cannot be used to copy the flags to a variable, so PUSHFD may be the best way to save the flags. The EFLAGS register contains the status flags (OF, SF, ZF, AP, PF, and CF), the control flag (DF), and the ten system flags (ID, VIP, VIF, AC, VM, RF, NT, IOPL, IF, and TF).

```
1       pushfd ; push flags on stack
2       pop dword [saved_flags] ; copy into a variable
3
4       push dword saved_flags ; push saved flag values
5       popfd ; copy into the flags
```

1.3 Examining memory with the gdb debugger

Here, we review two gdb commands, namely: print (p) and examine (x). Print is a simple command which can print some data values. Examine is strictly for printing data from memory and is quite useful for printing arrays of various types.

The format for the p command is either p expression or p/F expression, where F is a single letter defining the format of data to print. The format choices are d: decimal (default), x: hexadecimal, t: binary, u: unsigned, f: floating point, i: instruction, c: character, s: string, a: address.

The format for examine is x/NFS address where N is a number of items to print (default 1), F is a single letter format as used in the print command and S is the size of each memory location. The size options are b (byte): 1 bytes, h (halfword): 2 bytes, w (word): 4 bytes, g (giant): 8 bytes.

2 Tasks

For each of the following tasks, examine the program run (e.g. registers, variables) via the gdb debugger and the stdout, as well as the compilation messages, the listing file (.lst).

1. For each of these data definition directives: db, dw, dd, dq, dt:
   (a) initialize some variables with signed and unsigned (both small and big) integers and
   (b) initialize some variables with (both small and big) real numbers.
2. Initialize an array of 3 unique 4-byte integers and an array of 10 identic 4-byte integers.
3. Initialize some variables with some null-terminated strings
4. Do some stack operations using PUSH and POP.
5. Do some stack operations using PUSHAD and POPAD.
6. Do some stack operations using PUSHFD and POPFD

3 Homework

1. (True/False): An identifier cannot begin with a numeric digit.
2. (True/False): Local variables in procedures are created on the stack.
3. (True/False): The PUSH instruction cannot have an immediate operand (a constant value or the result of a constant expression)
4. (True/False): The string is stored contiguously.
5. Which data directive creates a 32-bit signed integer variable?
6. Declare an array of byte and initialize it to the first 5 letters of the alphabet.
7. Declare a string variable containing the word TEST repeated 500 times.
8. Which register (in 32-bit mode) manages the stack?
9. When a 32-bit value is pushed on the stack, what happens to ESP?
10. What causes the following warning or error messages and explain the solution:
    - warning: underflow in floating point expression
    - warning: overflow in floating point expression
    - warning: uninitialized space declared in code/data section: zeroing
    - warning: value does not fit in 8 bit field
    - error: invalid floating point constant size

4 Miscellany

The content is mainly based on the previous lab module and the following resources: [1], [2] and [3]. (compiled on 06/02/2015 at 9:07am)

References

[1] R. Seyfarth, *Introduction to 64 Bit Intel Assembly Language Programming for Linux.* CreateSpace, 2012.

[2] K. Irvine, *Assembly Language for X86 Processors.* Pearson Education, Limited, 2011. [Online]. Available: http://books.google.co.id/books?id=0k20RAAACAAJ

[3] J. Cavanagh, *X86 Assembly Language and C Fundamentals.* CRC Press, 2013.