KOM206 Laboratory (Dept. of Computer Science, IPB) Module 10 - Apr 29, 2015

Procedure (part 2) and String primitive instructions

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

Objectives

1. able to pass arguments to a procedure using GPRs and stacks
2. able to utilize some string primitive instructions
3. able to utilize command line arguments

1 Theory

1.1 Passing Parameters to a Procedure

1.1.1 Passing parameters using general-purpose registers

Refer to task01 for this. Since the contents of the GPRs are not saved prior to executing a CALL instruction, all six general-purpose registers, namely EAX, EBX, ECX, EDX, ESI, and EDI, can be used to transfer parameters to an invoked procedure. Registers ESP and EBP, however, cannot be used to pass parameters.

The operands are moved to the applicable registers prior to calling the procedure. The procedure can then access the registers directly to perform the specified operation. Since the stack is not used to store the operands before calling the procedure, there is no immediate value added to a (near) return.

1.1.2 Passing parameters using the stack

Refer to task02 for this. The code segment in Listing 1 illustrates using the stack to pass parameters to the called procedure. The calling program passes the augend and addend to a procedure by pushing them onto the stack. The invoked procedure then performs the addition operation and returns the sum to the calling program in the EAX register. Figure 1 depicts the stack.

Listing 1: Passing parameters to a procedure using the stack

```
main:
    push dword [augend]
    push dword [addend]
    call add_proc
    mov [res], eax
    jmp exit

add_proc:
    push ebp ; the EBP register will be used to access the augend and addend
    mov ebp, esp ; by this, EBP can be used to access data on the stack
    mov eax, [ebp+3*dword_size] ; access the augend
    add eax, [ebp+2*dword_size] ; access the addend
    pop ebp
    ; In order to restore the contents of the ESP to its initial value,
    ; an immediate value of eight is added to the ESP upon returning to
    ; the calling procedure.
    ret 2*dword_size ; 2 is for the push augend and addend
```

The calling program pushes the augend and the addend onto the stack, then calls the ADD_PROC procedure. The procedure is a near procedure; therefore, only the EIP register is placed on the stack. The invoked procedure pushes the EBP register onto the stack; the EBP register will be used to access the augend and the addend. Then the value of the ESP register, which points to the location of EBP on the stack, is moved to EBP. The EBP register can now be used to access data on the stack. The augend is then moved to register EAX by the instruction. The augend is located 12 bytes above the address of EBP (ESP); therefore, a value of 12 is added to the contents of the EBP register.

An ADD instruction then adds the contents of register EAX (augend) to the con tents of the stack at location EBP + 8 (addend) and places the sum in register EAX. Then the initial contents of register EBP are popped off the stack and stored in EBP. A near return instruction is then executed, which pops EIP off the stack; register ESP now points to the addend. In order to restore the contents of the ESP register to its initial value, an immediate value of eight is added to the ESP register upon returning to the calling procedure.

1.2 String Primitive Instructions

The x86 instruction set has five groups of instructions for processing arrays of bytes, words, and doublewords, refer to figure 2. Although they are called string primitives, they are not limited to character arrays. In 32-bit mode, each instruction implicitly uses ESI, EDI, or both registers to address memory. References to the accumulator imply the use of AL, AX, or EAX, depending on the instruction data size. String primitives execute efficiently because they automatically repeat and increment array indexes.

String primitive instructions increment or decrement ESI and EDI based on the state of the Direction flag. The Direction flag can be explicitly modified using the CLD and STD instructions as follows.

```
CLD ; clear Direction flag (forward direction)
STD ; set Direction flag (reverse direction)
```

Forgetting to set the Direction flag before a string primitive instruction can be a major headache, since the ESI and EDI registers may not increment or decrement as intended.

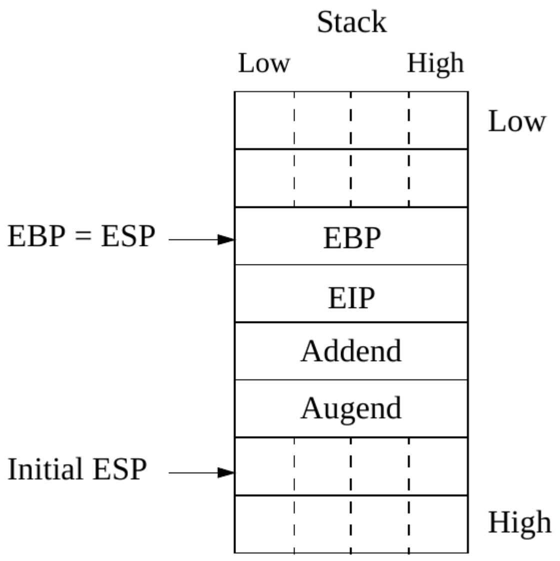

Figure 1: The stack situation for the procedure in listing 1

Instruction	Description
MOVSB, MOVSW, MOVSD	Move string data: Copy data from memory addressed by ESI to memory addressed by EDI.
CMPSB, CMPSW, CMPSD	Compare strings: Compare the contents of two memory locations addressed by ESI and EDI.
SCASB, SCASW, SCASD	Scan string: Compare the accumulator (AL, AX, or EAX) to the contents of memory addressed by EDI.
STOSB, STOSW, STOSD	Store string data: Store the accumulator contents into memory addressed by EDI.
LODSB, LODSW, LODSD	Load accumulator from string: Load memory addressed by ESI into the accumulator.

Figure 2: The string primitive instructions.

REP	Repeat while ECX > 0
REPZ, REPE	Repeat while the Zero flag is set and ECX > 0
REPNZ, REPNE	Repeat while the Zero flag is clear and ECX > 0

Figure 3: The repeat prefixes.

By itself, a string primitive instruction processes only a single mem ory value or pair of values. If you add a repeat prefix (as shown in figure 3), the instruction repeats, using ECX as a counter. The repeat prefix permits you to process an entire array using a single instruction.

In this module, we focus on SCASB, SCASW, and SCASD instructions. They compare a value in AL/AX/EAX to a byte, word, or doubleword, respectively, addressed by EDI. The instructions are useful when looking for a single value in a string or array. Combined with the REPE (or REPZ) prefix, the string or array is scanned while $ECX > 0$ and the value in AL/AX/EAX matches each subsequent value in memory. The REPNE prefix scans until either AL/AX/EAX matches a value in memory or ECX = 0.

1.3 Getting command line arguments

When you start your program from a terminal, Linux gives you information on the stack that you can use even if you do not pass any parameters to your program. Figure 4 shows what the stack looks like when you start your program.

Aguments Block: After the pointer to the path comes pointers to each passed argument, followed by a NULL pointer. This NULL pointer marks the end of passed arguments. *Environment Block:* After the arguments and NULL pointer, Linux will give us pointers to many Environment Variables followed again by a NULL pointer to mark the end of the variables. This list can contain well up to 200 items depending on your system.

In gdb, the command: r[un] [args], begins program execution. If the program normally takes command-line arguments (e.g., foo hi 3), you should specify them here (e.g., run hi 3).

2 Tasks

1. Write a procedure that takes its arguments using general purpose registers
2. Write a procedure that takes its arguments using stacks
3. Write an assembly program that copies a string to another string variable.

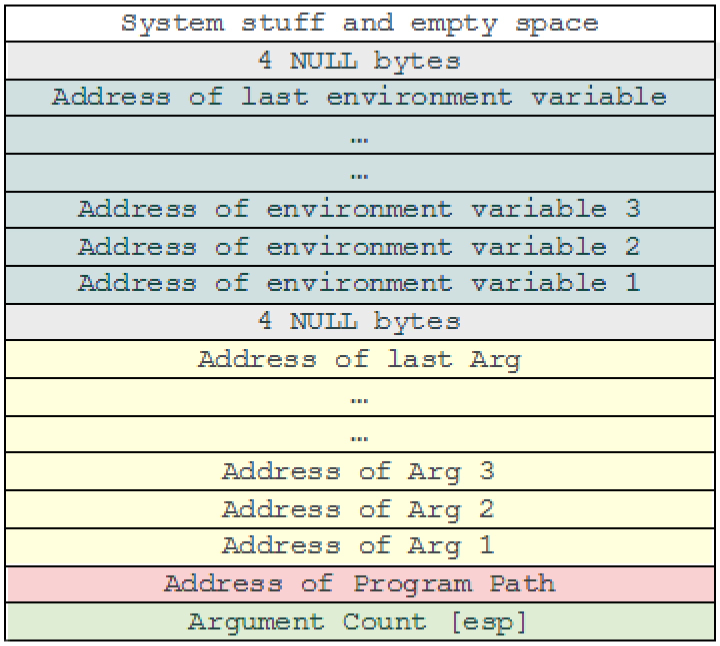
System stuff and empty space
4 NULL bytes
Address of last environment variable
…
…
Address of environment variable 3
Address of environment variable 2
Address of environment variable 1
4 NULL bytes
Address of last Arg
…
…
Address of Arg 3
Address of Arg 2
Address of Arg 1
Address of Program Path
Argument Count [esp]

Figure 4: The stack when a program begins. Courtesy of http://www.dreamincode.net/

4. Write an assembly program that scans for a Matching Character. Concretely, it searches the string in variable named alpha, look-ing for the letter F. If the letter is found, EDI points one position beyond the matching character. If the letter is not found, JNZ exits:

5. Write an assembly program that takes, at most, 3 command-line arguments, print those arguments to stdout. Print an error message if the number of argument is more than 3 or if there is no argument given.

3 Homework

1. When are the differences between stdout and stderr?

4 Miscellany

The content is mainly based on the previous lab module and the following resources: [1], of [2] and [3]. (compiled on 26/04/2015 at 6:43pm)

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.

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