BBM301 Programming Languages
Fall 2020
Assignment 1
Evaluating a New Programming Language
Assembly Language

İbrahim Burak Tanrıkulu, 21827852

November 12, 2020

1 What is Assembly? Why we use this language?

- Every computer has a processor. Processors understand only machine language instructions. However, machine language is too obscure and complex. So, assembly language is designed. 1
- Assembly language is a low-level programming language that invented theoretically in 1947 by Kathleen Booth while working on the ARC2 at Birkbeck, University of London following consultation by Andrew Booth, John von Neumann and Herman Goldstine at Institute for Advanced Study. 2
- Assembly languages helps programmers to write the human-readable code that is almost similar to machine language. Assembly language helps in providing full control of computer tasks. 3
- In the past, many programs and applications were written in assembly language to maximize the machine's performance. Today; C, Java, Python is widely used. Like assembly language, C/C++ can manipulate the bits at the machine level, but it is also portable to different computer platforms. There are C/C++ compilers for almost all computers. 4
- Assembly code is converted into executable machine code by an assembler. The conversion process is referred to as assembly.
- Different processors have different architecture or instructions and Assembly Language mainly consists of mnemonic processor instructions or data, and other statements or instructions. Thus, Assembly languages are hardware dependent; there is a different one for each CPU series. Sometimes, Assembly languages can be different on different operation systems because of system calls.

- Today, the learning of assembly language is still important for programmers. It helps in taking complete control over the system and its resources. By learning assembly language, the programmer is able to write the code to access registers and able to retrieve the memory address of pointers and values. Also Assembly language learning helps in understanding the processor and memory functions.

- I choosed Assembly language in this assignment. Because i am interested in computer architecture and microprocessors. I love to taking control of computer and doing whatever i want. I think Assembly and C language is very functional.

2 Language evaluation criteria

- **Readability & Writability**
  In 1940's programmers were reading and writing machine codes directly. So assembly language provided convenience to programmers at that time. Because Assembly language was more readable and writable than machine code. But today, we are naming Assembly language as "low-level language". Assembly language is less readable and writable than high-level languages like C, Python, Java etc. .
  Simplicity: There are nearly 2000 instructions in this language and instruction names are abbreviations of English words. We must use lots of instructions to do basic operations. Thus, I can easily say that Assembly language is not simple.
  Orthogonality: Assembly language is orthogonal. Because one instruction can do one thing and can use nearly all adressing modes. For example; "add" instruction can take place between "register to register" or "memory to register" or "register to memory".
  Data Types: In Assembly language we define data with sizes (byte or word(2 bytes)). It doesn't matter what we store inside that memory. Also we can define array of bytes or words. All of variables can defined by byte, word, doubleword, quadword or ten bytes.
  Syntax: An assembly program can be divided into three sections: data, bss, text. Statements are entered one statement per line. Each statement follows the following format:
  [label] mnemonic [operands] [;comment]
  The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands or the parameters of the command.
  Abstraction: Assembly language has no support for abstraction but we can use adressing for data abstraction, procedures or macros for process abstraction maybe.
  Expressivity: Assembly Language has flags that we can use in some instructions. JNZ instruction can be used for a loop. Also there are some instructions to do operations easily. For example, "add ecx, 1" (ecx(a register) = ecx + 1) can be done with "inc ecx" .

- **Reliability**
  Reliability changes to assembler and assembler changes to architecture. In low-level assemblers (original assemblers) reliability is almost unavailable. But high-level assemblers are good at this. There is default size-checking and syntax-checking in all assemblers .

- **Cost**
  Training: Assembly language is orthogonal but not simple. There are lots of instruction to learn but every instruction does one thing. Teaching all instructions may be hard but if you think the machine code, assembly is easier then that. Teaching assembly is better than teaching machine code, but worse than teaching high-level programming languages.
  Writing: Assembly language writing is easier than machine code writing but still its hard.
  Compiling: Assembly language is so close to machine code, also this language uses processor's instructions. So compiling is so good. But compiler (assembler) changes according to architecture and it is a problem.
  Executing: Assembler assembles the code to machine code directly and there is no run-time checking. So execution is very fast.
  Implementation: This language runs with any processor. Dont need any expensive hardware.
  Reliability: This language has no support for reliablity or safety. That is up to programmer and this is a big risk.
  Maintaining: Maintaining is so good on this language. Because every instruction does one thing(Orthogonal) and tracing is very easy. You can emulate this language step by step and can easily find mistakes.

- **Other Criteria**
  Portability: Assemblers are change according to computer architecture or operating system. So Assembly language is not portable.
  Generality: Assembly language is low level of many high level programming language. For example; C language firstly compiles a code to Assembly code, then assembler assembles this code to object code. So whatever C language can does are can be done with assembly too. You can nearly everything with Assembly language.
  Well-definedness: Assembly language is well defined. Because this language is orthogonal and uses processor's instructions.

3 Sample Assembly code (MASM32)

```
1  .486                                    ;create 32-bit code
2  .model flat, stdcall                    ;32 bit memory model
3
4  include \masm32\include\masm32rt.inc    ;including libraries
5
6  func macro message                      ;function definitions (macro)
7      mov ecx, 3                          ;loop count
8      my_label:
9          push ecx                        ;push ecx to stack
10         print OFFSET message            ;print macro comes from library
11         pop ecx                         ;pop ecx from stack
12         dec ecx                         ;decreasing count
13         jnz my_label
14     ret
15 endm
16
17 .data                                   ;data segment
18     msg db "Hello, world! ",0           ;our string
19     variable db 12h                     ;variable decleration
20     word_table dw 13, 345, 564, 12      ;array decleration
21     stars db 15 dup('*')                ;15 times *
22
23 .data?                                  ;data(bss) segment
24     num db ?                            ;uninitialized variable
25     my_table dw 10 dup(?)               ;uninitialized array
26
27 .code                                   ;code segment
28 start:
29     mov num, 123                        ;variable assignment
30     mov cx, word_table+2                ;gets 3rd element of word_table
31     lea ebx, word_table                 ;pointer of word_table[0]
32     mov [ebx], cx                       ;word_table[0] = word_table[2]
33     add ebx, 2                          ;ebx = word_table[1] (word = 2 bytes)
34     add [ebx], cx                       ;word_table[1] += word_table[0]
35     func msg                            ;function call with parameter
36     exit                                ;sysexit
37 end start
```

Output: Hello, world! Hello, world! Hello, world!

- **Writing experience**
  Strong points: I feel free to allocate bytes and doing whatever i want. There was not any blocking. Also, debugging was very easy. For example; i was troubling on loop in macro. Then, i discover that print function changes general registers. Thus, i used stack for preserving loop count.
  Weak points: Its hard to understand if you dont know assembly. Also assembly changes with different hardware. I tried to learn nasm assembly first, but then i decided to use masm. Because i think masm is easier and i have Windows PC and masm32 assembler.

4 Identifier definition rules

An identifier consists of a case-sensitive sequence of alphanumeric characters (A-Z, a-z, 0-9) and the following special characters like

- . (period)
- _ (underscore)
- $ (dollar sign)

Identifiers can be up to 31 characters long, and the first character cannot be numeric (0-9).

BNF syntax for identifiers:

```
<identifier> ::= <alphabetic> | <special character> | <identifier><letter>
<letter> ::= . | _ | $ | <alphanumeric>
<alphanumeric> ::= <alphabetic> | <numeric> |
                   <alphanumeric><alphabetic> | <alphanumeric><numeric>
<alphabetic> ::= A | B | ... | Z | a | b | ... | z
<numeric> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Regular Expression for identifiers:

```
( [a-zA-Z] | [._$] ) ( [a-zA-Z] | [._$] | [0-9] )*
```

5 Function definition and call rules (MASM32)

In assembly language, we use both procedures and macros for functions.

BNF syntax for procedure definition:

```
<procedure> ::= <name> proc <arguments>\n <codes> \n ret \n <name> endp
                <name> proc \n <codes> \n ret \n <name> endp
<name> ::= <identifier>
<arguments> ::= <argument> | <arguments> <argument>
<argument> ::= <identifier> : <type>
<type> ::= byte | word | dword | qword | tbyte
<codes> ::= <code> | <codes> \n <code>
```

We call procedures like: **call proc_name [arguments]**

BNF syntax for macro definition:

```
<macro> ::= <name> macro <arguments> \n <codes> \n ret \n endm |
            <name> macro \n <codes> \n ret \n endm
<name> ::= <identifier>
<arguments> ::= <argument> | <arguments> <argument>
<argument> ::= <identifier>
<codes> ::= <code> | <codes> \n <code>
```

We call macros like: **macro_name [arguments]**

What is the difference between procedure and macro? 1 (at next page)

6 References

https://en.wikipedia.org/wiki/Assembly_language
https://www.pcmag.com/encyclopedia/term/assembly-language
https://www.educba.com/what-is-assembly-language/
https://www.tutorialspoint.com/assembly_programming/
http://ftp.linux.org.uk/pub/linux/alpha/alpha/asm3.html
https://pediaa.com/what-is-the-difference-between-macro-and-procedure/

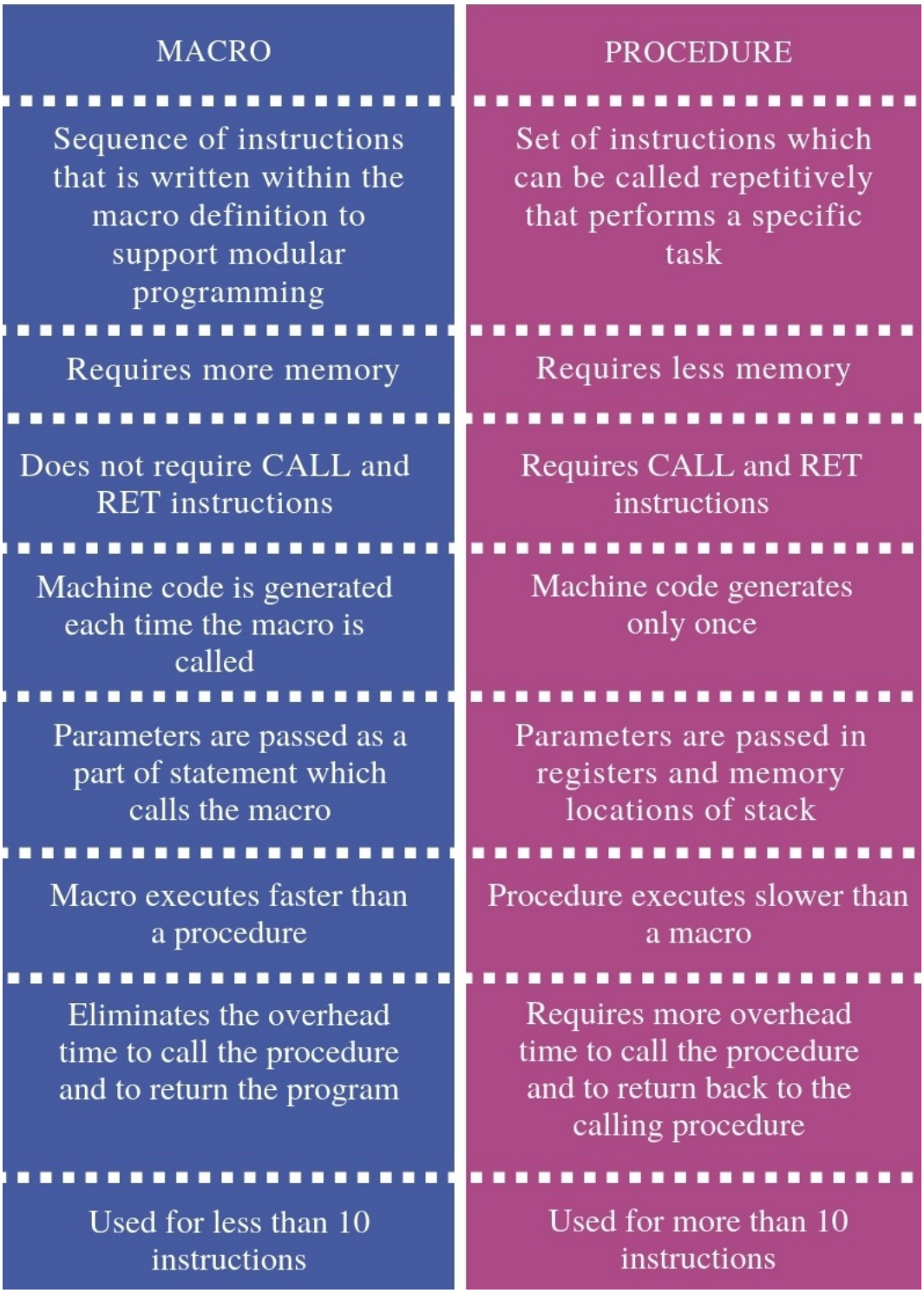

Figure 1: Difference between Macro and Procedure