International Conference on New Frontiers of Engineering, Management, Social Science and Education
(IETE) Institution of Electronics and Telecommunication Engineers, O.U. Campus, Hyderabad, India EMSSE-18
16th September 2018, www.conferenceworld.in ISBN: 978-93-87793-40-8

Implementation of high-level programming techniques in Assembly Programming Language using TASM

AlzhanKaipiyev, Maxim Demidenko

BSc (Hons) in Cyber Security, Asia Pacific University, Kuala Lumpur, Malaysia

ABSTRACT

This paper is based on the task, given by Asia Pacific Universityto design algorithms for creating for different shapes using assembly language. The purpose of the paper is to provide deeper understanding of the assembly language and direct communication with the CPU and memory. There could be no direct explanation or formula for creating specific shape. However, it provides full explanation of the high-level languages to assembly workflow and assembly to machine code. Source code snippets and screenshot of the program will be provided. All program written in TASM and TLINK.

Keywords:Assembly Language, Compilation, Low Level Programming, Shape Generation

1. INTRODUCTION

Most programs are written on the high-level languages, where a single statement is transformed into a number of processor assemblies, the machine-language language, and each command is directly interpreted into machine code, which makes it a low-level language. Most often, the assembler language is used to write additions to the operating system or to write programs for direct access to hardware. It is also necessary when optimizing critical blocks in application programs to “speed up their performance”. Since communication with the computer takes place at the machine level, it is necessary to have an idea of how information is stored and processed. For this, electrical elements are used that can only accept two states: "on" and "off." When storing data in storage devices, the sequence of electrical or magnetic charges is also interpreted as the "on" state or "off", which is the content of the recorded information. The following “Fig..1” have shown the levels of programming language. Every language has its own compiler.

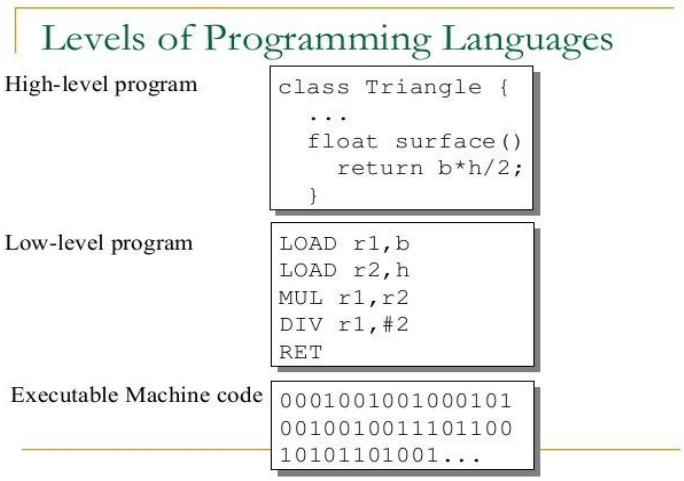

Figure 1. Levels of Programming Language

2.1 ASSEMBLER

"The assembler language is the symbolic programming language that lies closest to the machine language in form and content. The assembler language is useful when:

- You need to control your program closely, down to the byte and even the bit level.
- You must write subroutines for functions that are not provided by other symbolic programming languages, such as COBOL, Fortran, or PL/I" [2].

Assembler - low-level programming language, which is a format for recording machine commands, convenient for human perception. Assembler language commands one to one correspond to the processor commands and, in fact, represent a convenient character form of the record (mnemonic code) of the commands and their arguments. Also, the assembler language provides basic program abstractions: linking program and data parts through labels with symbolic names and directives. Assembler directives allow user to include in the program data blocks (described explicitly or read from the file). Repeat the specified fragment a specified number of times, compile a fragment by condition. Specify the execution address of the fragment, change the values of the labels during the compilation process; use macros with parameters, etc.

2.2 REGISTERS

Register is a certain area of memory inside the processor itself, from 8 to 32 bits long, which is used for intermediate storage of information processed by the processor.

1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 *(bit position)*
9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Main registers				
		AH	AL	**AX** (primary accumulator)
		BH	BL	**BX** (base, accumulator)
		CH	CL	**CX** (counter, accumulator)
		DH	DL	**DX** (accumulator, extended acc.)
Index registers				
0 0 0 0		SI		**S**ource **I**ndex
0 0 0 0		DI		**D**estination **I**ndex
0 0 0 0		BP		**B**ase **P**ointer
0 0 0 0		SP		**S**tack **P**ointer
Program counter				
0 0 0 0		IP		**I**nstruction **P**ointer
Segment registers				
	CS		0 0 0 0	**C**ode **S**egment
	DS		0 0 0 0	**D**ata **S**egment
	ES		0 0 0 0	**E**xtra **S**egment
	SS		0 0 0 0	**S**tack **S**egment
Status register				
	- - - - O D I T S Z - A - P - C			Flags

Registers of general purposeare EAX, EBX, ECX, EDX. They are 32-bit and are divided into two parts, the bottom of which AX, BX, CD, DX - 16-bit, and is divided into two 8-bit registers. So, AX is divided into AH and AL, DX into DH and DL, etc. The letter "H" means the upper case.

So, AH and AL each one byte, AX - 2 bytes (or word - word), EAX - 4 bytes (or dword - double word). These registers are used for operations with data, such as comparison, mathematical operations, or writing data to memory.

General registers are often used in arithmetic functions, processor computations, and output of the result. Therefore, the use of registers as long-term storage of information is not recommended, and it is recommended to use the register for its intended purpose, even though they allow storing any information. For example, the AX register is best suited for intermediate data and calculations, and the CX register is used as a counter in cycles. It is recommended that you first study and understand the register assignments before further deepening into the assembler language. AH in DOS programs is used as a determinant, what service will be used when invoking INT.

Offset registersare EIP, ESP, EBP, ESI, EDI. These registers are 32-bit, the lower half of which is available as IP, SP, BP, SI, DI registers.

EIP is a command pointer and contains an offset (the amount of offset relative to the beginning of the program) to the line of code that will be executed next. That is, the complete address for the next executable code line will be CS: EIP. These registers have shown in "Fig.2".

The ESP register points to the address of the top of the stack (the address where the next variable will be written by the PUSH command). At the same time EBP register serves as the bottom of the stack and stores the lowest point of it. Although stack pointers (the ESP and EBP register) also apply to general purpose registers and can be used in commands, it is strongly recommended that you never involve it to use outside the stack. This is especially important when working in protected mode, when the processor automatically uses the current value of the stack to put values into it, for example, when processing exceptions.

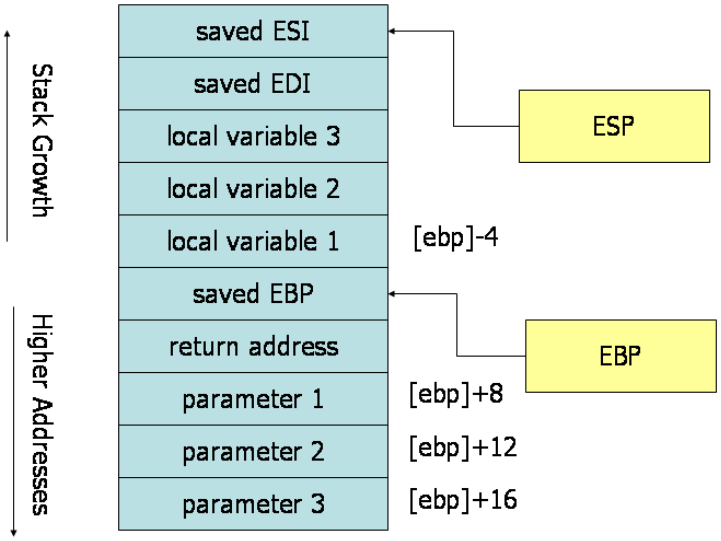

Figure 2. The structure of the stack [1]

The EUR register contains the address from which the information is entered or taken to the stack (or the "depth" of the stack). Parameters of functions have a positive shift relative to EBP, local variables - negative shift, and the full address of this section of memory will be SS: EBP.

The ESI register is the source address and contains the address of the beginning of the information block for the "move block" operation (full DS address: SI), and the EDI register is the destination address in this operation (full ES address: EDI).

2.3 OPERATIONS.

For further understanding it is important to understand how computer executes the code and why assembler is so close to machine code. As an example, consider the MOV operation using 16-bit and 32-bit registers. Such a command as MOV works with multiple registers, therefore with different types of registers and data it will use a unique opcode for understanding the processor of the operation itself. The execution flow of the instruction for x8086 is:Prefixes (optional), Opcode (first byte), Opcode 2 (if needed), Mod, Reg, R/M, Displacement data. [4]

Opcode	Mnemonic	Description
8A /r	MOV r8,r/m8	Move 8-bit operand to the 8-bit register.
8B /r	MOV r16,r/m16	Move 16-bit operand to the 16-bit register.
8B /r	MOV r32,r/m32	Move 32-bit operand to the 32-bit register.

Mod	Displacement
00	If r/m is 110, Displacement (16 bits) is address; otherwise, no displacement
01	Eight-bit displacement, sign-extended to 16 bits
10	16-bit displacement (example: MOV [BX + SI]+ displacement,al)
11	r/m is treated as a second "reg" field

Reg	W = 0	W = 1	double word
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

r/m	Operand address
000	(BX) + (SI) + displacement (0, 1 or 2 bytes long)
001	(BX) + (DI) + displacement (0, 1 or 2 bytes long)
010	(BP) + (SI) + displacement (0, 1 or 2 bytes long)
011	(BP) + (DI) + displacement (0, 1 or 2 bytes long)
100	(SI) + displacement (0, 1 or 2 bytes long)
101	(DI) + displacement (0, 1 or 2 bytes long)

110	(BP) + displacement unless mod = 00 (see mod table)
111	(BX) + displacement (0, 1 or 2 bytes long)

Translate the assembler language into machine code, for the *MOV* command you can use the tables shown above for the command "*MOV CX, 12h*". With *MOV*, there are no prefixes, so you must take the *HEX* from the first table and translate it into *BIN*. 1000 1011= 8Bh = *MOV r16, r / m16*. There is no displacement for this instruction so mod value remains 00. The register used for the example is *CX* and its value in table 101, r / m will be used with value 001 due to the link with mod = 00. Thus, you can output the instruction in the machine code with the knowledge of assembler.

1000 1011 – (MOV 16-bit, 8Bh) 00 - mod 001 – register 110 -r/m

1000 1011 0001 1110 0001 0010 0000 0000 = 8Bh 1Eh 12h 00h

The same principle goes for 32-bit instruction. The register used is *EAX*. The command is "*MOV EAX, 24h*".

1000 1011 - (MOV 32-bit, 8Bh) 00 – mod 000 – register 110 – r/m

1000 1011 0000 0110 0010 0100 0000 0000 = 8Bh 06h 24h 00h

3.1 USING COMPLEX ALGORITHMS IN ASSEMBLER.

After understanding how the processor performs the operations, you can make sure that any information coming into the register, or participating in any operation or process, is not stored in memory, unlike high-level languages. Since long-term storage of information in registers is also not introduced (some information could be corrupted, even in the segments where it was unexpected). Therefore, this problem should be solved by changing the structure of the code written in assembler. The best example is the work of the code optimizer in C [3], compiled with the "GNU GCC"

It should be noted that there would be no work with the stack with push and pop instructions. These commands do not give enough flexibility in working with the x8086 system and it will be much more efficient to use stack pointer (ESP) and base pointer (EBP).

```
1.  pushebp     ; save old base pointer
2.  movebp, esp; new base pointer
3.  subesp, 4   ; allocate one 4-byte variable
```

3.2 BOX SHAPE AND NESTED LOOP TRIANGLES PRINT.

These forms are great for explaining the next concept. When writing code in a high-level language, type C allows the use of variables. Variables can be changed directly during code execution and can change the value as many times as they need. At first look assembler must deal with this, but because of the lack of registers in

assembler it is physically impossible to store the variable in each of them to run some programs. And given the use of registers for their intended purpose, you may need to have a clean register at hand that is not clogged with data. Reviewing disassembled code snippets from C programs, you can run into the notion of volatile data. Variables having this type can be changed during code execution and should not be statically registered before starting. This parameter can be put as in manual, for more accurate memory management, and trust the code optimizer, which will do its job perfectly in most cases.

The code optimizer uses a similar scheme: static variables are sent directly to the register, while volatile ones are stored in the stack, from where they get to the registers only when needed.

```
1.  movdwordptr [ebp+8], 12 ;number of rows
2.  movdwordptr [ebp+12], 24 ;number of columns
3.
4.  mov cx, [ebp+8]
5.
6.  BoxLoop:
7.          mov [ebp+16], cx
8.          mov cx, [ebp+12]
9.
10.         BoxInner:
11.                 mov ah,02h
12.                 mov dl, 206
13.                 int 21h
14.
15.                 loop BoxInner
16.
17.         mov ah,02h
18.         mov dl,10
19.         int 21h
20.         mov cx, [ebp+16]
21.         loop BoxLoop
```

Volatile data, like number of rows and columns, that would be changed during the executions due to work of the loop function and CX register is stored in the stack as the “improvised variable”. However, the static data, like the ASCII value of the printable character is still would be given to the register directly.

3.3 ZIG-ZAG SHAPE PRINT

The implementation of this pattern in C is as follows. This sample code is not thunderous, but it contains enough logic to create problems when writing in assembly language when issuing instructions directly to the processor. To solve this situation, it is suggested to use stack to store more data and create volatile variables.

```
1.  for (int row = 0; row <numRows; ++row){
2.      for (int col = 0; col <numCols; ++col) {
3.          intmodCol = (col % modulusVal);
4.          if (modCol>= numRows){
5.              modCol -= numRows;
6.              modCol = ((numRows - 1) - (modCol + 1));}
7.          modCol = ((numRows - 1) - modCol);
8.          if (modCol == row){
9.              printf("X");}
10.         else{
11.             printf(" ");}}
12.     printf("\n");}
```

Here is an example of the stack variables implementation and their further usage. All the comments are used for readability and understanding on the code, due to lack of naming in assembly itself.All this MOV to the stack could be called as variable declaration, all the comments represent the name of each variable.

```
1.  movdwordptr [ebp+8], 1 ;loop again? (boolean)
2.  movdwordptr [ebp+12], 7 ;number of rows
3.  movdwordptr [ebp+16], 60 ;number of columns
4.  movdwordptr [ebp+20], 0 ;current row
5.  movdwordptr [ebp+24], 0 ;current column
6.
7.  movax, [ebp+12] ;number of rows
8.  sub ax, 1
9.  movbx, 2
10. mulbx
11. mov [ebp+28], ax ;modulus value (spacing)
12.
13. mov dx, 0
14. movax, 0
15. movax, [ebp+24] ;current column
```

```
16.     movbx, [ebp+28] ;modulus value (spacing)
17.     div bx
18.     mov [ebp+32], dx  ;Column Mod Value
19.     cmp dx, [ebp+12] ;number of rows
```

This snippet perfectly shows this type of the workflow. Data from the stack (the improvised variables) goes to the register to complete the calculations. It is important to know in what way register would be used with the specific command. After completing calculations data is stored into the stack again (as the improvised variable).

Also, all the static numbers, that would be the same for any execution are remains in the code and would be given to the register or command directly, there is no point to store this into the stack.

In this way code could implement very complicated algorithms and it would be more understandable to the people. It should be noted that all the "variables" were allocated 4 bytes of stack memory. This was done for better understanding of the code from the side with a clear indentation in the storage of each variable. It is preferable to measure the size of the "variable" manually and set a larger gap only if you are confident of the possible appearance of more information there. In all other cases, it is recommended to use a smaller gap, to save memory and optimize better. Also, do not forget to clean the stack after the end of the algorithm using it, in case there is nothing more to store.

3.4DIAMONDPRINT

There is another example of C code which is implemented and transfer into assembly code. There is enough information to use it. In this case register will be used again.

```
1.      inti, j;
2.      for(i=1; i<=5; i++)      {
3                       for(j=i; j<5; j++){
4.                      printf(" "); }
5.                      for(j=1; j<=(2*i-1); j++){
6.                       printf("*");}
7.      printf("\n"); }
8.      for(i=5; i>=1; i--){
9.                      for(j=i; j<=5; j++){
10.                      printf(" ");}
11.     for(j=2; j<(2*i-1); j++){
12.                     printf("*");}
13.     printf("\n");}
```

Here well seen that stack used as variables in the same way as written previously. In comment is written which values are moved to stack and for which purpose. Assembly is not user-friendly and therefore user should be able to comment everything by himself. Here MOV statementused in the same way as previously.

```
1.  movdwordptr [ebp+8], 10 ; number of row
2.  movdwordptr [ebp+12], 5 ; space
3.  movdwordptr [ebp+16], 1 ; number of character
4.  mov cx,[ebp+8]
5.
6.  DiamondRow:mov [ebp+8], cx
7.  mov cx, [ebp+12]
8.  DiamondSpaces:mov ah,02h
9.  mov dl,32
10. int 21h
11. dec cx
12. cmp cx,0
13. jgeDiamondSpaces
14. mov cx,[ebp+16]
15.
16. Print:mov bx,47
17. addbx,cx
18. mov [ebp+20],bx
19. mov ah,02h
20. mov dl,[ebp+20]
21. int 21h
22. loop print
23. NewLine:mov cx,[ebp+8]
24. mov ah,02h
25. mov dl,10
26. int 21h
27. dec cx
28. cmp cx,0
29. jgeDiamondRow
```

That snippet showing almost the same workflow. However, has a different logic inside it. In that case was used a static declaration where it was declared inside code. Then that variables was used to change CX (which is answering for loop function) and move data inside it.

It should be noted againthat all the "variables" were allocated 4 bytes of stack memory. It is done for optimizing the memory inside the program and allocate the proper space for data which was moved inside it and do not save it inside the register. However, if any user wants to allocate more memory that is possible. However, it is better to avoid it. Simple use command "**LEAVE"** to clear it.

5. CONCLUSION

Due the format of machine commands ("low level") of the assembler language, it is difficult for a person to read andunderstand a program compare with high-level programming languages; the program consists of too "shallow" elements-machine commands, accordingly, programming and debugging become more complicated, theamount of time required increases, the probability of making errors is high. General registers are often used in arithmetic functions, processor computations, and output of the result. Therefore, the use of registers as long-term storage of information is not recommended, and it is recommended to use the register for its intended purpose, even though they allow storing any information. However, this language should not be used on a dailybasis, it is rather used to solve clearly defined problems related to the memory or speed of the program.

6. OUTPUT

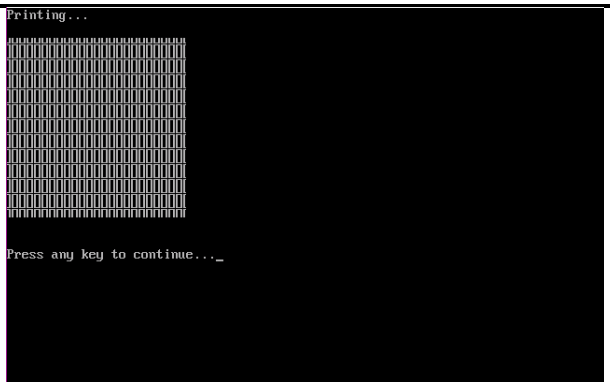

Figure 3. Box shape pattern.

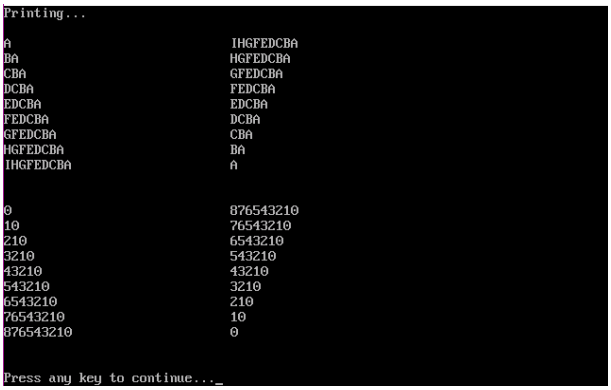

Figure 4. Nested loop triangles pattern.

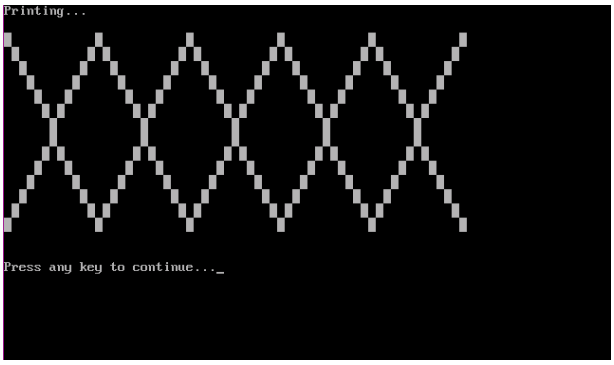

Figure 5. Zig-zag pattern.

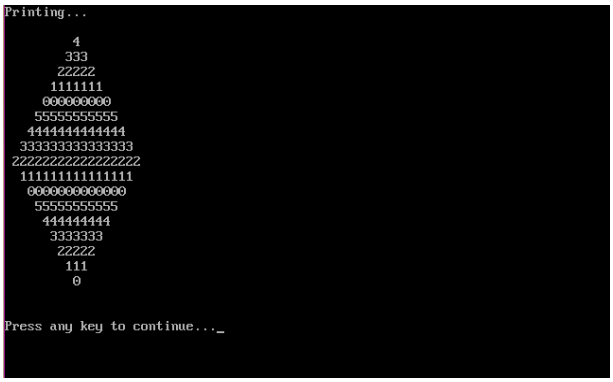

Figure 6. Diamond shape pattern

7. ACKNOWLEDGMENT

The authors would like to share gratitude to Mr Umapathy Eaganathan, Lecturer in Computing, Asia Pacific University, Malaysia for the constant support and motivation which helped to participate in this International Conference and also for journal publication.

REFERENCES

[1] Alan, B. et al., n.d. *x86 Assembly Guide.* [Online] Available at: http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html[Accessed 20 04 2018].

[2] IBM, 2006. *Assembler language (HLASM Language Reference).* [Online] Available at: https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.asma400/asmr102112.htm
[Accessed 10 04 2018].

[3] Joshi, R. U., 2001. *Code Optimization Using the GNU C Compiler.* [Online] Available at: https://linuxgazette.net/issue71/joshi.html [Accessed 24 04 2018].

[4] x86asm.net, n.d. *X86 Opcode and Instruction Reference.* [Online] Available at: http://ref.x86asm.net/coder32.html [Accessed 27 04 2018].