IFES – INSTITUTO FEDERAL DO ESPÍRITO SANTO

BRUNNA DALZINI DE OLIVEIRA

LAVÍNIA CORTELETTI

MATHEUS BUTTER DIAS

MAYANNARA TRINDADE CARVALHO

VINICIUS FREITAS ROCHA

ANÁLISE COMPARATIVA ENTRE AS IMPLEMENTAÇÕES DE LINGUAGEM C E ASSEMBLY

SERRA - ES

2018

BRUNNA DALZINI DE OLIVEIRA

LAVÍNIA CORTELETTI

MATHEUS BUTTER DIAS

MAYANNARA TRINDADE CARVALHO

VINICIUS FREITAS ROCHA

ANÁLISE COMPARATIVA ENTRE AS IMPLEMENTAÇÕES DE LINGUAGEM C E ASSEMBLY

Trabalho realizado como requisito para a disciplina de Arquitetura e Organização de Computadores do curso de Bacharelado em Sistemas de Informação do Instituto Federal do Espírito Santo, Campus Serra, ministrada pelo professor Flávio Giraldeli Bianca.

SERRA - ES

2018

SUMÁRIO

1. INTRODUÇÃO

“Quero aprender!” - Carvalho, Mayannara

; sobre a linguagem Assembly

O presente trabalho tem como objetivo demonstrar as diferenças de implementação das linguagens de alto e baixo nível.

Ao citar a linguagem de alto nível estamos dizendo que é a linguagem cuja sintaxe se aproxima mais da linguagem humana e se distancia mais da linguagem de máquina. Sendo assim, as linguagens de alto nível não estão diretamente relacionadas à arquitetura do computador, ela possui um nível de abstração que faz com que você entenda aquele código mais rapidamente, pois não é necessário ter o conhecimento sobre as características do processador, como instruções e registradores. Essas características são abstraídas na linguagem de alto nível.

E ao citar linguagem de baixo nível, estamos nos referindo a uma linguagem que se aproxima mais da máquina. Ela segue características da arquitetura de computador. Uma diferença com as linguagens de alto nível é que a de baixo nível utiliza apenas instruções que serão executadas pelo processador. Você precisa ter um conhecimento mais direto da arquitetura do computador para realizar algo.

Através do estudo de três problemas, que serão desenvolvidos em ambas as linguagens de alto e baixo nível, montaremos uma comparação dos códigos elaborados para suas resoluções. Enquanto que do ponto de vista do usuário os dois programas se comportam da mesma forma, a parte da programação apresenta suas particularidades.

Utilizaremos a linguagem C no repl.it e no Microsoft Visual Studio Community 2017, para os testes de programação de alto nível, e a linguagem Assembly x86 no emu8086, para os de baixo nível.

2. APRESENTAÇÃO DOS PROBLEMAS

Como já citado anteriormente, o presente trabalho conta com três problemas a serem resolvidos em C e Assembly, são eles:

- Problema 1: Ler a hora de início e a hora de fim de um jogo de Xadrez (considere apenas horas inteiras, sem os minutos) e calcule a duração do jogo em horas, sabendo-se que o jogo pode ter a duração de 24 horas e iniciar em um dia e terminar no dia seguinte.

- Problema 2: Declarar um vetor de inteiros com 10 elementos e solicitar ao usuário que digite os elementos, um por um, até completar o vetor. Em seguida, ordenar o vetor (sem fazer cópia do mesmo) usando o algoritmo *Insertion Sort* e, por fim, exibir o vetor ordenado na tela.

- Problema 3: Em Análise Combinatória, definimos a Combinação ***n*** elementos (total de elementos) ***r*** a ***r*** (tamanho do subconjunto), como:

$$C\,{}^{n}_{r} = \binom{n}{r} = \frac{n!}{r! \cdot (n-r)!}$$

  Elaborar um programa que solicita ***n*** e ***r*** ao usuário, calcula C^{n} e exibe o resultado na tela. No entanto, o cálculo do fatorial deve ser feito por um módulo separado. Ou seja, deve-se tomar o número que se deseja calcular o fatorial e passá-lo como parâmetro para um procedimento/função que calculará o fatorial do número. Essa função/procedimento deverá retornar o resultado para o programa principal, que se encarregará usá-lo para calcular a Combinação e exibir o resultado na tela. Além disso, o fatorial máximo em uma arquitetura de 16 bits é 8, pelo fato de 8! (40320) ser o maior fatorial que pode ser representado na base binária com a quantidade de bits disponível.

3. DESENVOLVIMENTO

“Esses são bons.” - Butter, Matheus

; sobre os problemas propostos e seus requisitos

Com a implementação dos exercícios propostos nas duas linguagens ficou notório o quanto muda do ponto de vista do programador a criação dos enunciados dados, e podemos destacar os itens do tópico a seguir.

3.1. DIFERENÇAS NA IMPLEMENTAÇÃO

Para melhor entendimento, alguns dos exemplos podem ser simplesmente modificações de variáveis do trabalho ou representações da mesma na outra linguagem.

Registradores

Levando em consideração o processo de aprendizado que houve, principalmente no caso da linguagem de baixo nível, resolvemos pôr uma breve explicação sobre os registradores, apesar de não mexermos diretamente com eles em C.

Na linguagem C é possível implementar um programa sem ter a mínima ideia sobre o que é e para que serve um registrador, a decisão de qual registrador usar é omitida da visão do programador de alto nível, enquanto que em Assembly é fundamental conhecer quais são e como funcionam.

Os registradores da arquitetura original do Assembly x86 são de 16 bits e estão divididos em dois grupos: os registradores de propósito geral, onde o programador tem autonomia para escolher qual será utilizado e para que fim; E os registradores de propósito especial ou reservados. Nestes, são armazenadas informações específicas para cada operação.

Cada registrador é dividido em 2 partes de 8 bits, uma alta e uma baixa, que são identificadas no Assembly pelos sufixos H (*high*) e L (*low*), respectivamente. Também é possível acessar o conteúdo como um todo, os 16 bits, através do sufixo X.

Os registradores são utilizados para guardar as informações que servirão como parâmetro para as mais diversas operações a serem implementadas. É necessário ter destreza e paciência para definir onde colocar a sua informação e como gerir o programa sem colocar em risco a integridade do conteúdo.

Impressão de dados

Em C, a impressão de dados é realizada com um simples função pré definida *printf*, que vai pegar como parâmetro a string a ser mostrada entre aspas, podendo também converter uma variável por meio de "%" + o tipo da variável (%d valor inteiro, %f valor decimal etc).

Já em Assembly, a mensagem, salva em uma variável ou não, será à moda antiga formada letra a letra por um loop. Também pode ser impressa seguindo o comando LEA DX,string + MOV AH,9 + INT 21H, onde o LEA é responsável por fazer uma cópia do endereço da string para o registrador DX, e a segunda parte chama a função 09h da interrupção 21h para display da string atualmente armazenada no registrador DX (o conjunto de comandos vai imprimir a string até encontrar um '$' nela, interrompendo o processo). Outra forma encontrada, fazendo o uso do procedimento *print_num* presente na biblioteca emu8086.inc, que será melhor explicado mais à frente no trabalho.

Exemplificando:

`LEA DX, msg1`
`MOV AH, 9`
`INT 21h`
	`printf("Hora de inicio: ");`

`MOV AL, BL`
`; BL havia recebido a duração anteriormente`
`CALL print_num`
	`printf("%d", duracao);`

Declaração de variáveis

Em ambas linguagens podemos declarar variáveis de acordo com o tipo e inicializá-las com os valores desejados ou até mesmo indefinidos.

Na linguagem Assembly, podemos declarar variáveis do tipo DB (define byte) ou do tipo DW (define word). As do tipo byte possuem o tamanho de 1 byte, já as do tipo word possuem 2 bytes, podemos defini-las de acordo com a necessidade da variável a ser criada.

Em C podemos iniciar uma variável com o tipo int, float, void, char, entre outros. Ainda sobre C, podemos também utilizar modificadores de tipo,como o *"long"* ou *"short"*, que permitem alterar o tamanho da capacidade da variável, assim como vimos nas variáveis de Assembly.

Exemplificando:

`horaINICIAL  DB  ?`
`combinacao DW ?`
`mensagem DB 'Texto aqui $'`
	`int horaINICIAL;`
	`long int combinacao;`
	`char mensagem[] = "Texto aqui";`

Condicionais

Utilizamos a estrutura CMP para comparar dois elementos em Assembly.

Basicamente, o CMP realiza a subtração do primeiro parâmetro pelo segundo, alterando uma flag de acordo com o resultado. Uma subtração pode ser menor que zero, igual a zero ou maior que zero. E é justamente nesse ponto que a estrutura de *jump* trabalhará.

Existem diversos tipos de *jump*, que podem ser adequados de acordo com a estrutura da condicional a ser implementada. Um comando em C bem parecido ao *jump* do Assembly, é o *goto.* Não utilizamos esse comando na implementação, mas ele poderia entrar na estrutura condicional do *if* e "pular" desse ponto para qualquer outra parte do código, independente das instruções que estivessem a seguir.

Voltando para o Assembly… Não é possível realizar uma comparação entre duas variáveis diretamente, o CMP não possibilita isso (temos que transferir o valor de uma variável para um registrador, por exemplo), caso que é facilmente

implementado em C com a simples estruturação de um *if* com variáveis e operadores.

Exemplificando:

```
MOV AL, horaINICIAL
CMP AL, horaFINAL

JA horaINICIALmaiorFINAL
; código (outras funcionalidades)
horaINICIALmaiorFINAL:
; código
```

```
if (hi > hf) {
                // código
        }
```

Estruturas de repetição

Em Assembly um loop funciona da seguinte forma: o registrador CX é inicializado com o valor desejado e o loop continuará sendo executado enquanto o valor de CX não for zero. Cada vez que o loop é executado o valor de CX é subtraído em uma unidade. Além disso, para que o loop seja realizado, é necessário inserir no código *LOOP nomeDaEstrutura*.

A ideia do funcionamento não é muito diferente da linguagem C. Em C, da mesma forma, um contador é inicializado, uma condição é determinada e um valor para incremento é selecionado.

Exemplificando:

```
mov cx, 10
numVetor:

   ; código

   loop numVetor
```

```
for(f1=0; f1<10; f1++)
        {
                // código
        }
```

Procedimentos

Numa função em C, precisamos simplesmente passar ou não um parâmetro para a função que ela retornará o resultado de acordo com suas instruções.

Em Assembly não passamos parâmetros para a função, os valores das variáveis são os que estão atualmente armazenados nelas, e são eles que serão utilizados pela função. Atentando-se também ao fato de quais variáveis estão sendo usadas na função.

O retorno da função é igual, no sentido de que em ambos os casos retornamos ao local onde o código se encontrava antes.

No entanto, em C é possível retornar a variável que queremos usar diretamente a partir de um ponto na função, o que só é possível indiretamente em Assembly.

Exemplificando:

```
FATORIAL    PROC

      ; código

      ; aqui ocorreriam as modificações
      ; na variável fat, seu último
      ; resultado armazenado é o que
      ; retorna

       RET

FATORIAL    ENDP
```

```
int fatorial (int numero)
{
       // código

       // a variável fat recebe o
       // valor do fatorial de número
       return fat;

       // podemos retornar diretamente
       // a variável "fat" para a main
}
```

3.2. QUANTIDADE DE INSTRUÇÕES

"Deu loop infinito de novo." - Dalzini, Brunna

; sobre um erro frequente em Assembly durante o trabalho

O conjunto de instruções é uma caixa de ferramentas em que cada ferramenta oferece uma utilidade. O conjunto de instruções da linguagem de programação .c oferece ferramentas de manejo facilitadas para operar o computador (linguagem de alto nível), se aproximando assim mais da linguagem humana. Para isso é necessário agrupar um conjunto "grande" de instruções de linguagem de máquina, gerando ferramentas bem completas que podem resolver muitos problemas.

Já a "caixa de ferramentas" do Assembly oferece instruções bem menores e objetivas que chegam mais próximo da máquina (linguagem de baixo nível). Com elas é possível implementar os mesmos problemas que as linguagem de médio e alto nível, porém por serem mais específicas e menores é necessário um conjunto maior

de instruções para chegar ao mesmo ponto, vendo pelo ponto de vista do programador.

Uma analogia interessante é pensar em resolver um quebra cabeça em que é possível usar 5 peças grandes, linguagem de alto nível, ou 50 peças pequenas, linguagem de baixo nível. O resultado final é o mesmo para o usuário: a imagem pronta. Mas se tratando de código de linguagem para o computador, as 50 peças pequenas podem ser arranjadas a fim de que o processo seja executado com maior performance.

Resumindo... As linguagens de alto nível oferecem um encapsulamento “grande” de linguagem de máquina para facilitar o uso do ser humano, então não possuem muita flexibilidade, e a linguagem de baixo nível oferece um encapsulamento “pequeno”, que requer um conhecimento mais específico para a implementação dos comandos, e que se bem utilizados trarão grandes benefícios em performance.

3.3. PROCEDIMENTOS EM ASSEMBLY

As rotinas poderiam simplesmente ser chamadas ao longo do programa ao declararmos a biblioteca emu8086.inc, mas para fins acadêmicos optou-se por não realizar o include da mesma, e sim copiar os procedimentos/macros utilizados nos problemas e analisar o funcionamento de cada um.

PUTC

PUTC é um macro para impressão de caracteres na tela.

Ele recebe um caracter, copia o conteúdo de AX para a pilha de memória, move o caracter para dentro de AL e o 0Eh para AH, printa o AL *(caracter)* e retorna com o valor da pilha para AX. Após a impressão o curso caminha em uma unidade.

PRINT_NUM

PRINT_NUM é um procedimento que imprime um número, o conteúdo do registrador AX, pra ser mais específico.

Ele vai trabalhar imprimindo números positivos através do procedimento PRINT_NUM_UNS, da seguinte forma:

Caso a entrada seja um número negativo, ele imprime o sinal (-), faz o complemento a dois do valor, e chama esse procedimento PRINT_NUM_UNS para a impressão do valor positivo.

Para garantir a integridade dos dados nos registradores, o processo vai utilizar a pilha na memória, escrever o conteúdo dos registradores DX e AX, e resgatá-los ao término.

SCAN_NUM

O procedimento SCAN_NUM recebe um número como entrada e o armazena no registrador CX.

Ele tem um conjunto bem completo de testes, que identificam cada *caracter* digitado pelo usuário. Se o conteúdo for válido, isto é, se o valor de entrada for do tipo numérico, ao apertar a tecla *Enter* o conteúdo será enviado para o registrador CX.

3.4. TEMPO DE IMPLEMENTAÇÃO

“Calma, to tentando entender.” - Corteletti, Lavínia

; sobre a lógica só fazendo sentido antes da execução do código em Assembly

Quanto à linguagem C, quase todo o grupo já tinha algum conhecimento prévio sobre, então a implementação foi bem menos complicada, fora o fato desta linguagem ser mais próxima das outras que normalmente utilizamos para programar.

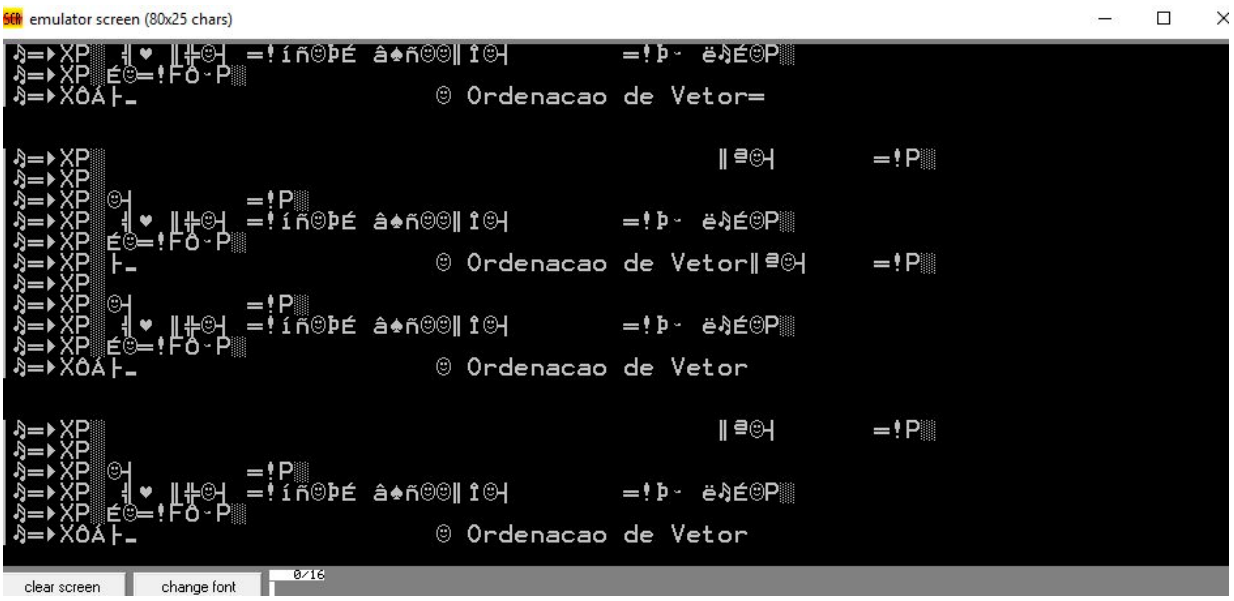
Já para Assembly, não podemos afirmar exatamente o tempo que foi gasto, visto que nenhum componente do grupo teve qualquer contato com a linguagem antes e todos tiveram que aprender do início. O aprendizado ocorreu simultaneamente à execução dos exercícios e ao estudo da linguagem de montagem, com os membros do grupo compartilhando o que entendiam e não entendiam nos seus “teste.asm” individuais.

Houve uma grande resistência para aceitar o funcionamento das instruções e interrupções por ser algo muito diferente à primeira vista, mas depois de um tempo comparando a execução dos programas no emulador com seu código escrito, pudemos assimilar melhor o que estava acontecendo.

Essa resistência somada às diferenças entre as linguagens de alto nível e de baixo nível causaram um grande atraso no início da implementação dos problemas propostos para análise.

Se desconsiderarmos o tempo de primeiro contato com a linguagem, o problema 1 foi o que demandou menos tempo em ambas linguagens, por ser o mais simples dos três.

O problema 2 foi o mais demorado. Em .c pela parte de relembrar como trabalhar na linguagem, e em .asm, porque trabalhar com loop foi além de difícil, com resultados bem inesperados.



Já na descrição da questão 3 temos o aviso do *overflow* que deu um norte para o que seria lidado. Apesar da compreensão rápida no C relembrado, a manipulação de vetores e os loops no Assembly causou um pouco de confusão, e fomos madrugada adentro trabalhando no código.

Abaixo, uma tabela com o tempo aproximado de implementação de cada questão nas linguagens C e Assembly.

	Tempo aproximado de implementação		
Linguagem	**Problema 1**	**Problema 2**	**Problema 3**
C	20 minutos	40 minutos	25 minutos
Assembly	1 dia	2 dias	1 dia

4. CONCLUSÃO

“Se a divisão não der inteiro, o que acontece?” - Freitas, Vinicius

; sobre as diferenças entre C e Assembly

Antes de iniciarmos o trabalho já tínhamos algum receio a respeito das dificuldades que iríamos enfrentar para solucionar os problemas em Assembly, por se tratar de uma linguagem que se aproxima mais da máquina e termos pouca familiaridade com o assunto de linguagens de baixo nível.

Outro problema era o tempo disponível para a tarefa, já que estamos no final do período e sem termos todas as aulas por causa dos jogos da Copa do Mundo.

Mas apesar desses imprevistos, graças às aulas dadas em sala e às pesquisas realizadas por conta própria conseguimos avançar na elaboração do relatório.

Partimos para gerar os códigos em ambas as linguagens.

Começamos com o C e já era possível identificar que desde que você compreenda a lógica de programação, tendo estudado C há muito tempo ou nem estudado, não seria um problema a resolução dos problemas. Isso, claro, nos referindo às semelhanças existentes entre as linguagens de alto nível.

O tempo gasto nessa parte foi o de lembrar os comandos da linguagem e/ou entender as diferenças entre C e Python, linguagem vista até o momento no curso, e a primeira questão foi pensada para dar base às demais.

No momento de desenvolver em Assembly as questões que foram resolvidas em C com certa rapidez, o grupo precisou rever vários conceitos teóricos de arquitetura e estudar praticamente do zero o funcionamento da linguagem, levando muito mais tempo para entender o melhor caminho a seguir e poder terminar os códigos.

Coisas que pareciam simples em alto nível se tornaram extremamente complexas. Para uma simples leitura, tínhamos que entender como um processador de 16 bits visualiza, armazena, recebe dados e faz cálculos. Foi algo bem trabalhoso.

O problema 2 foi, sem dúvidas, o mais trabalhoso dos três. Ele sugou quase todas as energias do grupo para sua elaboração. Foi feita uma primeira parte, então pulamos para o problema 3, e quando voltamos ainda encontramos várias dificuldades para continuar. O loop de ordenação simplesmente não dava certo de jeito nenhum.

Mais pesquisas, mais testes. O grupo voltou para as teorias iniciais procurando entender a linguagem de novas formas. Um esforço colossal para chegar no resultado final.

Depois que aprendemos como eram feitas determinadas rotinas, conseguimos criar ligações com outras linguagens e perceber que no fim das contas a linguagem de baixo nível não é algo de outro mundo, mas com absoluta certeza requer muita atenção e prática. Bem mais do que as de alto nível, que são mais “acessíveis”.

O clássico “Hello World” em tela

#Python

```
print(“Hello World”)
```

//C

```
#include <stdio.h>

int main(){
    printf(“Hello World”);
}
```

;Assembly

```
ORG 100h

LEA DX,msg
MOV AH, 9
INT 21h

ret

msg db ‘Hello World $’
```

Escutamos em sala de aula que esse trabalho forma dois tipos de pessoas, as que fazem o trabalho, mas depois não querem mais continuar os estudos nessa área de baixo nível, e as que escolhem dar prosseguimento nisso.

Não temos certeza em qual nos encaixamos. Sem sombras de dúvida, esse trabalho foi uma grande fonte de aprendizado no referente às diferenças entre as

linguagens de alto e baixo nível, e também uma ótima de forma de pôr em prática o conteúdo teórico trabalhado durante a disciplina.

REFERÊNCIAS BIBLIOGRÁFICAS

Material disponibilizado pelo professor na Sala do Tempo.

8086 assembler tutorial for beginners. Disponível em: <http://jbwyatt.com/253/emu/asm_tutorial_01.html>

Assembly - Arithmetic Instructions. Disponível em: <https://www.tutorialspoint.com/assembly_programming/assembly_arithmetic_instructions.htm>

OLIVEIRA, William. **O que é linguagem de programação de alto/baixo nível?** Disponível em: <https://woliveiras.com.br/posts/o-que-e-linguagem-de-programacao-de-alto-nivel/>

DOS INT 21h - DOS Function Codes. Disponível em: <http://spike.scu.edu.au/~barry/interrupts.html#ah09>