INSTITUTO FEDERAL
Espírito Santo
Campus Serra

INSTITUTO FEDERAL DO ESPÍRITO SANTO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

BRENO AMÂNCIO AFFONSO
JONATHAN CASTRO SILVA
JOSIAS NEVES JARDIM BORBA
PAULO SOUSA SANCHES LOPES
RAFAEL BARROS LEÃO BORGES

RESOLVENDO PROBLEMAS EM C E ASSEMBLY

SERRA
2024

BRENO AMÂNCIO AFFONSO
JONATHAN CASTRO SILVA
JOSIAS NEVES JARDIM BORBA
PAULO SOUSA SANCHES LOPES
RAFAEL BARROS LEÃO BORGES

RESOLVENDO PROBLEMAS EM C E ASSEMBLY

Trabalho apresentado à disciplina de Arquitetura e Organização de Computadores como requisito parcial para obtenção de nota.

Orientador: Prof. Flávio Giraldeli

SERRA
2024

SUMÁRIO

1 INTRODUÇÃO

O presente relatório possui como objetivo esclarecer detalhes do funcionamento de um programa em baixo nível. Para isto, foram utilizadas duas linguagens para a resolução de três problemas: C e Assembly x86 (16 bits).

A linguagem C foi utilizada para demonstrar a resolução usando uma linguagem de alto nível, ou seja, sem que o programador se atente sobre quais instruções serão utilizadas a nível de processador, visto que estas escolhas são feitas pelo compilador. Já a linguagem Assembly x86, na versão de 16 bits, foi utilizada, por meio do emulador emu8086, para a programação direta em baixo nível, sendo de responsabilidade do programador decidir quais instruções serão usadas.

Para o primeiro problema, a tarefa escolhida foi implementar uma calculadora simples, com as seguintes operações: adição, subtração, multiplicação e divisão.

O segundo caso, trata da verificação se um determinado número n (n é maior ou igual a 2) é ou não primo. Como resultado, deve haver a impressão deste resultado e, caso não seja primo, a impressão do conjunto de divisores deste número.

O terceiro problema trata-se do cálculo da soma de uma determinada progressão aritmética (PA). O fluxo de execução baseia-se em: solicitar os três parâmetros necessários (valor do primeiro e último elementos e a quantidade de elementos), passagem destes valores para uma função (em C) ou um procedimento (em Assembly) que efetuará o cálculo, e impressão do resultado.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

Implementar três programas simples, utilizando a linguagem C e Assembly x86 de 16 bits. Bem como realizar, para cada um dos casos, uma análise comparativa entre o uso de linguagens de baixo nível (Assembly x86) e linguagens de alto nível (C), objetivando uma maior compreensão dos fundamentos estudados na disciplina de Arquitetura e Organização de Computadores.

1.1.2 OBJETIVOS ESPECÍFICOS

- Elaborar e resolver um problema simples de condicionais e operações aritméticas em C e Assembly.
- Desenvolver um algoritmo para verificar se um número é primo, em C e Assembly.
- Desenvolver um algoritmo para calcular a soma dos termos de uma progressão aritmética (PA).
- Realizar uma análise comparativa e demonstrar a diferença de complexidade entre linguagens de alto e baixo nível.

2 DESENVOLVIMENTO

2.1 MACROS/PROCEDIMENTOS DO INC\EMU8086.INC

Para a realização das atividades, foi necessário copiar alguns macros ou procedimentos já concluídos, para que o tempo fosse gasto apenas realizando a confecção da solução para o problema apresentado.

Um macro é um pedaço de código que irá substituir a sua chamada, sendo possível passar uma espécie de variável para que uma parte do código seja modificada em cada chamada. O macro copiado foi o PUTC, um macro responsável por imprimir um caractere na tela. Ele recebe um caractere, coloca o seu código ASCII na metade inferior do registrador AX, em AL, e utiliza a interrupção 10h/0Eh, que imprime o caracter em AL na tela e avança o cursor.

Um procedimento é uma espécie de função, é uma parte do código que pode ser chamada pelo programa para realizar alguma tarefa específica. Os procedimentos utilizados foram o SCAN_NUM, PRINT_NUM e PRINT_NUM_UNS, três procedimentos relacionados com a interação com o usuário. Todos utilizam os comandos . e POP para restaurar o valor dos registradores modificados para aqueles que estavam antes do procedimento.

O SCAN_NUM recebe o input do usuário e o coloca no registrador CX. Para isso, utiliza a interrupção 16h/00h para coletar o botão pressionado pelo teclado, e imprime seu código ASCII com a interrupção 10h/0Eh. Após isso, verifica se o caractere foi um sinal negativo, e modifica a variável make_minus para guardar essa informação, logo após, verifica se o "Enter" foi o caractere capturado, caso não tenha sido, verifica se o caractere é um número (vendo se código ASCII do caractere é menor que o código ASCII referente ao "0" e depois se é maior que o código ASCII referente ao "9"), caso não seja um número, imprime um espaço vazio para apagar o caractere impresso e retorna o cursor em um espaço, utilizando o macro PUTC. Em seguida, multiplica o número acumulado por 10 para que a casa da unidade fique livre, e então verifica se ele continua sendo um número de 16 bits, caso não seja, apaga o último caractere impresso e retorna para o número antes de sua multiplicação por 10. Após isso, com a casa da unidade liberada, adiciona o número digitado ao número salvo e novamente verifica se continua possuindo 16 bits, caso não tenha, apaga o último número digitado da tela e do número salvo. Posteriormente, volta para o começo e repete até que o caractere "Enter" seja pressionado, em que verifica se a variável de número negativo está verdadeira, caso esteja, transforma o número em negativo e retorna, caso contrário, apenas retorna.

O PRINT_NUM imprime o número que estiver no registrador AX. Para isso, verifica se o número é zero, caso seja, utiliza o macro PUTC para imprimir "0", caso não seja, verifica se o número é negativo, caso seja, troca seu sinal, imprime um menos na tela "-", e chama o procedimento PRINT_NUM_UNS, caso não seja, apenas chama o procedimento PRINT_NUM_UNS.

O PRINT_NUM_UNS imprime o número que estiver no registrador AX, sua diferença para o PRINT_NUM, é que não se importa se o número for negativo. Para imprimir o número, coloca em CX 1, para fazer uma espécie de *flag* para não imprimir "0" antes do primeiro dígito; coloca em BX o valor de 10.000, pois o maior valor possível em 16 bits é 65.535; e verifica se AX é 0, se for, imprime "0" e finaliza o procedimento. Caso contrário, vê se o divisor, BX, é 0, se for, o número acabou, senão verifica se CX foi mudado para 0, se sim, vai diretamente para a impressão, senão, verifica se

AX é menor que BX, caso seja, quer dizer que o resultado da divisão será 0, então pula o valor atual de BX e o divide por 10, senão vai para a impressão. Para imprimir, primeiramente, diz que o CX recebe 0, pois, para entrar na impressão é necessário que a divisão não dê 0, então coloca 0 em DX para que o resultado da divisão não seja afetado, e então divide o valor de AX por BX. Como o resultado, que fica em AX, sempre estará entre 0 e 9, AH nunca terá valor maior que 0, então apenas soma 30h (48 em decimal), para chegar na região da tabela ASCII onde os números estão, e então imprime esse caractere usando o macro PUTC. Em seguida, pega o resto da divisão, que é o número anterior sem a casa de maior valor, e o coloca em AX, repetindo todo o processo, até que BX seja igual a 0, que significa o fim do número.

2.2 PROBLEMA 1

O problema 1 se trata do mais simples dos 3 resolvidos, uma calculadora para valores inteiros (adição, subtração, multiplicação e divisão), sendo necessário aproximadamente **10 minutos** para sua resolução utilizando a linguagem C.

Entretanto, a resolução em Assembly demorou, aproximadamente, 90 minutos, ou seja, houve um aumento de **9 vezes (800%)** entre o tempo para a resolução em C e em Assembly. Essa diferença ocorre pelo fato do Assembly utilizar diretamente os registradores da CPU, algo que não ocorre na programação em C, ou seja, a carga de trabalho aumenta significativamente para o programador em baixo nível.

Outra discrepância entre as linguagens ocorre nas instruções de conta, em que C utiliza uma forma mais próxima do humano, em que são utilizados símbolos para representar as contas, como “+” para soma ou “*” para multiplicação, tal abstração não ocorre em Assembly, onde é necessário utilizar as instruções para cada conta, com duas tendo um comportamento diferente do comum.

As instruções para adição e subtração são simples, “ADD num1, num2”, traduzindo para C se torna “num1 = num1 + num2”, a subtração segue a mesma lógica. Agora, para a divisão e multiplicação, existe um registrador específico que é o dividendo (DX:AX) que vai receber o resultado da conta, porém não é possível traduzir essa conta diretamente para C, pois dependendo do tamanho de num2 (o

divisor), existem 2 comportamentos, um em que num2 tem tamanho de 1 byte, fazendo o dividendo ser apenas AX, com o quociente indo para AL e o resto para AH. Ou num2 é uma word, que nesse caso são 16 bits, 2 bytes, em que o dividendo será DX:AX, sendo DX a parte alta do número e AX a parte baixa, nela, o quociente vai para AX e o resto fica em DX. O mesmo comportamento ocorre com a multiplicação, porém nela e na soma existe o risco de overflow, algo que é necessário de ser tratado, para que o código não pare, por nenhuma razão aparente para o usuário.

Outro elemento diferente para Assembly são as condicionais, a diferença entre Assembly e C nesse quesito é que em C condicionais existem e em Assembly não. Na realidade, isso não é bem verdade, pois existem formas de se tratar condições em Assembly, porém, são bem diferentes, já que é necessário pensar em qual *flag* servirá o seu propósito, e então utilizar uma instrução que modifique essa *flag* para que, enfim, um JUMP condicional seja utilizado. JUMP é uma instrução que muda o ponteiro de instrução para o local apontado, um “rótulo”.

Já em C, só é necessário fazer uma comparação que dê um resultado de verdadeiro ou falso, sendo o inteiro 1 considerado verdadeiro e o inteiro 0 falso, além disso, existem os operadores lógicos para condicionais em C (como and, && e or, ||), algo que não existe em Assembly.

2.3 PROBLEMA 2

O problema 2 foi o mais demorado para ser concluído, tendo o seu código em C tomando cerca de **15 minutos** para ser resolvido inicialmente, levando mais 5 minutos para ajustes. Ao comparar com o tempo necessário para a finalização completa, tivemos um aumento de **6 vezes (500%)**, indo de 20 minutos em C, para, aproximadamente, **120 minutos** em Assembly x86 de 16 bits, essa diferença torna clara a discrepância de complexidade entre as tarefas.

Outro ponto interessante que também demonstra a diferença de complexidade é a quantidade de linhas necessárias para concluir cada programa. Claro que uma diferença é esperada, afinal, são linguagens diferentes, mas foram necessárias 40 linhas para concluir o código em C, incluindo indentação e

comentários, já em Assembly, foram necessárias 364 linhas, para realizar o mesmo problema.

Além disso a linguagem C oculta vários aspectos que são necessários para o funcionamento do código em Assembly, um exemplo claro são os registradores, elementos que não são utilizados diretamente em um código em C, mas que são essenciais para o funcionamento de um código em Assembly, razões para a disparidade em quantidade de linhas. A principal sendo a utilização das funções de *scan* e de *print* ser diferente, em C é apenas necessário escrever “#include <stdio.h>” no topo do código para que elas estejam disponíveis para uso, algo que não existe em Assembly, onde foi necessário copiar os procedimentos equivalentes para dentro do código para tornar seu uso possível.

Nesse mesmo sentido, em Assembly, existem interrupções que fazem impressão, então, para a impressão de strings, foi apenas necessário utilizar essa uma interrupção, que usa o endereço do primeiro caractere para imprimir a série inteira de caracteres. Essa estratégia, infelizmente, não funciona para números, então foi necessário copiar a função PRINT_NUM, para fazer a impressão de números.

Outro exemplo é a divisão, onde em C, a divisão, por padrão, utiliza números de ponto flutuante, sendo obrigatória sua conversão para inteiro para a resolução do problema, além disso, possui uma instrução que retorna diretamente o módulo resultante da divisão, sendo ele o “%”, diferente de como se faz a divisão em Assembly, em que, por padronização da Intel, o registrador AX é o dividendo e o valor passado para a instrução (DIV ou IDIV) é o divisor (sendo possível ser apenas outro registrador ou um valor guardado na memória).

Outro elemento que a linguagem C possui, mas que é diferente em Assembly são os loops. Em Assembly as instruções LOOP e suas variações existem, elas utilizam o registrador CX para sua verificação de parada. Essa forma de loop é parecida com outra existente em Assembly, que foi a utilizada na solução, os JUMPs, ambas dependem dos “rótulos” para saber para onde pular ou o que repetir. JUMPs foram utilizados, pois o desenvolvedor responsável os achou mais simples de prever.

2.4 PROBLEMA 3

O problema 3, mesmo sendo o que inclui a maior quantidade de elementos diferentes, por ser direto, levou **10 minutos** para ter sua solução na linguagem C. Tempo que ao ser aumentado em **60 vezes (5900%)** se equipara ao tempo necessário para resolvê-lo em Assembly, que foi de **10 horas**. Sendo essas 10 horas dividas em: 5 horas para montar o código do começo, e 5 horas para ajustes e correções de problemas no código. Vale ressaltar que grande parte do tempo gasto foi por confusões na hora de implementar uma ideia que, no nosso modelo mental do código, parecia fazer sentido, porém não fez.

Assim como citado nos problemas anteriores, a implementação em C nesse problema oculta diversos fatores que são cruciais para o funcionamento completo do código. A exemplo de elementos ocultos, temos: o controle dos registradores e pilhas para carregar e gerenciar os dados, procedimentos de escrita e leitura (print e scan), instruções que são fundamentais para o funcionamento do comando executado na linguagem de alto nível, monitoramento de casos com overflow, e, não obstante, a aplicação de operações básicas como soma, multiplicação e divisão envolvendo conhecimento elementar dos registradores utilizados.

O primeiro passo, a fim de produzir o código, foi receber os dados no intuito de passá-los para a função (“procedimento”, em assembly), que posteriormente seria criado pelo grupo. Para isso, em C, assim como nas atividades que antecedem a essa, foi usado a biblioteca <*stdio.h*> que permite o uso de funções prontas como *scanf()* e *printf()* para obter um *input* e produzir um *output*. Em assembly, foi necessário usar os mesmos procedimentos seguidos pelas atividades anteriores, sendo esses: *SCAN_NUM* e *PRINT_NUM*. Com o diferencial de que, para armazenar os valores, foi utilizado o conceito de pilhas.

Esclarecendo um pouco mais, a escolha de se usar pilhas ocorreu devido ao fato de o procedimento em assembly não possuir parâmetros definidos na chamada da função. Para isso, é necessário separar registradores ou criar uma pilha com os parâmetros necessários. Assim, usando as instruções de *PUSH* e *POP*, movimentamos os dados recebidos pelo usuário pela pilha, permitindo assim o acesso deles pelo procedimento. Foi necessário também, durante a coleta dos parâmetros com o *POP*, descobrir que a chamada do procedimento *CALL* também

usa a pilha para armazenar o local de retorno, assim tendo que movê-la para um registrador a fim de ter acesso aos dados.

O segundo passo foi implementar a função usando os parâmetros passados pela pilha. Nessa etapa houve as instruções de *ADD*, *IMUL*, e *IDIV*, onde foi necessário verificar se houve overflow com o *JO* (*Jump if Overflow*) e devolver o resultado da operação no registrador *AX*, e para verificar *overflow* armazenou-se os valores 0 ou 1 no registrador DX (indicando ausência ou presença de *overflow*) a depender do *jump* executado. Para implementar em C, tudo isso foi feito apenas criando uma função com os parâmetros necessários e retornando o resultado da operação.

O terceiro passo consiste basicamente, ao voltar do procedimento, de uma comparação (*CMP)* seguida de JE para direcionar o código para o fluxo normal de imprimir o resultado ou imprimir mensagem de *overflow*. Assim, encerrando-se o código. Em C, isso foi implementado com apenas função de *printf* mais as variáveis que armazenam o resultado da função.

3 CONCLUSÃO

Por fim, o principal quesito a destacar é a extrema diferença entre a programação em alto nível e baixo nível, sendo clara a diferença de tempo para o desenvolvimento de um projeto em cada situação, bem como a quantidade de linhas de código que devem ser escritas, como visto durante o desenvolvimento, um if em C, se traduz em no minimo uma operação compare (CMP) e um jump (JMP), tornando o código em Assembly bem mais extenso. Contudo, a programação em linguagens como o Assembly x86 se prova necessária, visto que, há determinados casos em que o código gerado por um compilador não será a melhor opção com relação, por exemplo, ao uso de memória, mesmo que, de forma geral, estes códigos gerados já sejam suficientemente bem estruturados.

Com relação ao aprendizado do grupo, é uma constante que o conhecimento prático de Assembly, fixou outros conhecimentos já vistos anteriormente, sobretudo com relação à divisão de tarefas dentro da CPU. Outro ponto em comum a todos os integrantes é que, apesar de sabermos a diferença entre linguagens de baixo e alto níveis, na prática, nenhum de nós sabiamos as aplicações de Assembly, ou o que de fato acontece com um código, por exemplo, em C, após ser compilado.