Campus de Cascavel

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Curso de Bacharelado em Ciência da Computação

CALCULADORA ASSEMBLY

Relatório

Matheus Bueno

Vitor Mayorca Camargo

CASCAVEL

2024

1 FUNÇÃO PRINCIPAL

1.1 Section .data

```
section .data
    msg1 db "Digite uma equação: ", 0 ;Chamada pro input do usuario

    input_scan1 db "%f %c %f", 0        ;Usado no scanf
    input_scan_tam equ $- input_scan1

    msDebugOP   db "SAIDA: num1: %f op: %c num2: %f result: %f", 10, 0
    msDebugERRO db "Erro: num1: %f op: %c num2: %f result: %f", 10, 0
    const dd 1.0                        ;Debug

    exeCorreta  db "%lf %c %lf = %lf", 10, 0                        ;Usado no fprintf
    exeIncorreta db "%lf %c %lf = funcionalidade não disponível", 10, 0  ;Usado no fprintf

    nomeArquivo db "output.txt", 0
    modoArquivo db "a", 0
    ptrArquivo  dq 0                    ;Ponteiro para o arquivo

    fopenErro   db "Falha na abertura do arquivo! Verifique as permissoes de execucao.", 10, 0

    pos_inf dd 0x7F800000               ;Infinito positvo. Usado para detectar overflow
```

Na seção .data foram definidas constantes, strings, e um ponteiro para um arquivo.

Algumas das strings possuem “%lf” e “%c” no meio delas. Isso acontece pois elas são strings de controle, usadas nas funções printf(), scanf() e fprintf(). A constante ptrArquivo irá apontar para o endereço do arquivo aberto durante a execução do programa, e a constante “const” é utilizada numa única operação movss dentro da função de exponenciação. Por fim, a constante pos_inf é utilizada como meio de comparação para detectar o overflow nas operações escalares.

1.2 Section .bss

```
section .bss
    num1 resd 1
    op resd 1
    num2 resd 1
```

Foram definidas apenas 3 variáveis globais: num1, op e num2. Num1 será o primeiro número que o usuário inserir, que operará sobre num2 seguindo o caracter que estiver em op.

1.3 Funções Externas

```
    extern printf
    extern scanf
    extern fopen
    extern fclose
    extern fprintf

section .text
    global main
```

Foram utilizadas 5 funções importadas da linguagem C: printf(), scanf(), fopen(), fclose() e fprintf(). Também foi utilizada a flag global main para definir a função principal do programa

1.4 Section .text

```
main:
;------------------Stack-frame da funcao main
    push rbp
    mov rbp, rsp

;------------------Printf da requisicao de input do usuario
    xor rax, rax
    mov rdi, msg1
    call printf                ;Movemos a mensagem de boas vindas para rdi, e escrevemos ela na tela

;------------------Scanf do input do usuario
    xor rax, rax
    mov rdi, input_scan1
    mov rsi, num1
    mov rdx, op
    mov rcx, num2
    call scanf                 ;Zeramos rax, pois nao temos nenhum ponto flutuante como parametro
```

Inicialmente, foi feito um stack-frame para a função main(). Depois, usamos as funções printf() e scanf() do C para requisitar e receber os inputs do usuário.

```
;------------------Abrimos o arquivo aonde sera escrito o output
    mov rdi, nomeArquivo
    mov rsi, modoArquivo
    call fopen
    mov [ptrArquivo], rax       ;Ponteiro pro arquivo retorna em rax

    cmp rax, 0
    je falhaArquivo
    mov rax, 0                  ;Zeramos rax, para evitar futuros problemas

;------------------Movemos as variaveis lidas para nossos registradores
    movss xmm0, [num1]
    movss xmm1, [num2]          ;Salva os valores lidos. Precisão simples (32 bits)
    mov dil, [op]               ;salva o operador lido    dil é o primeiro byte do rdi
```

E o arquivo de output foi aberto em modo append com uso do fopen(). A função retorna um ponteiro para o arquivo em rax, que é armazenado em [ptrArquivo]. Por fim, as variáveis inseridas pelo usuário são armazenadas em xmm0, rdi e xmm1, respectivamente pela ordem de inserção.

```
;-------------------Compararamos o operador escolhido pelo usuário, e chamamos a função correspondente
    cmp dil, 'a'
    je callAdd

    cmp dil, 's'
    je callSub

    cmp dil, 'm'
    je callMul

    cmp dil, 'd'
    je callDiv

    cmp dil, 'e'
    je callExp

    mov rax, 1      ;Nenhuma operacao valida detectada. Codigo de erro rax 1
    jmp output
```

E aqui acontece um “switch case”, aonde serão chamadas flags específicas para cada operador. Caso o operador não exista, a mensagem de erro genérica é escrita no arquivo

```
;-------------------Aqui, chamamos as operacoes e comparamos o retorno (rax) em baixo da flag output
callAdd:
    call adicao
    jmp output

callSub:
    call subtracao
    jmp output

callDiv:
    call divisao
    jmp output

callMul:
    call multiplicacao
    jmp output

callExp:
    call exponenciacao
    jmp output
```

Quando as operações são escolhidas, o fluxo do código vêm para cá, aonde as funções das operações são chamadas, e o retorna é comparado sob a flag output.

```
;------------------Comparamos os valores armazenados em rax.  0 - OK   != 0 - erro
output:
    cmp rax, 0
    jg notOK
    jmp OK


;------------------Escrevemos no arquivo os resultados
OK:
    call escrevesolucaoOK
    jmp fim


;------------------Escrevemos no arquivo a operacao + msg de erro
notOK:
    call escrevesolucaoNOTOK
    jmp fim
```

Aqui vemos a flag output. Basicamente, todas as funções retornam em rax o valor de 0 em caso de sucesso, ou valores diferentes em caso de erro. Após definir se houve erro ou sucesso, os valores armazenados em xmm0 e xmm1 são escritos no arquivo.

1.5 Falha e Finalização do Programa

```
;------------------Printamos mensagem de erro, caso o arquivo não consiga ser aberto
falhaArquivo:
    xor rax, rax
    mov rdi, fopenErro
    call printf

;------------------Destack-frame + finalizacao da execucao
fim:
    ;---------------Fechamos o arquivo
    mov rdi, [ptrArquivo]
    call fclose
    mov rsp, rbp
    pop rbp

    mov rax, 60
    mov rdi, 0
    syscall
```

Caso o arquivo não tenha sido aberto, o fluxo de execução vai para falhaArquivo:, aonde é exibida uma mensagem de erro, e ocorre o destack-frame + fim da execução da função main. Mesmo em caso de sucesso a flag fim: é chamada.

1.6 Operações

As funções das operações foram importadas de um arquivo externo, armazenado na mesma pasta, com a ajuda do comando %include “operacoes.asm”.

Elas seguem os nomes especificados no pdf do trabalho:

```
;float adicao()
;float subtracao()
;float multiplicacao()
;float divisao()
;float exponenciacao
;void escrevesolucaoOK
;void escrevesolucaoNOTOK
;char getOperador
```

1.6.1 Adição:

```
;------------------------------ Função de SOMA -------------------------------;
adicao:                 ;Funcao de adicao
    push rbp
    mov  rbp, rsp       ;Stack-Frame

    mov byte [op], "+"        ;Mudamos op para o simbolo do operadr

    movss xmm2, xmm0    ;O processador nao deixou fazer vaddss xmm2, xmm1, xmm0

    addss xmm2, xmm1    ;Soma 2 por 1, armazena em 2
    addeado:

    ucomiss xmm2, [pos_inf] ;Checamos se houve overflow, ao comparar com o infinito
    je overflow
    jp overflow

    mov rax, 0          ;Funcao retorna 0 se deu tudo certo

    jmp saidaOp
;-----------------------------------------------------------------------------;
```

A função de soma é simples: o stack-frame é criado, os operadores são somados e armazenados em xmm2 e, em caso de overflow, a flag *overflow* é chamada, retornando 1 em rax. Em caso de sucesso, rax retorna 0. Já o destack-frame é feito sob a flag saidaOP.

O operador [op], que foi inserido pelo usuário, também é alterado para “+”, e será usado na escrita do arquivo de output.

1.6.2 Subtração:

```
;---------------------------- Função de SUBTRAÇÃO -----------------------------;
subtracao:
    push rbp
    mov  rbp, rsp       ;Stack-Frame

    mov byte [op], "-"       ;Mudamos op para o simbolo do operadr

    movss xmm2, xmm0    ;O processador nao deixou fazer vsubss xmm2, xmm1, xmm0

    subss xmm2, xmm1    ;Subtrai 2 por 1, armazena em 2

    subeado:
    ucomiss xmm2, [pos_inf] ;Checamos se houve overflow, ao comparar com o infinito
    je overflow
    jp overflow

    mov rax, 0          ;Funcao retorna 0 se deu tudo certo

    jmp saidaOp
;------------------------------------------------------------------------------;
```

A subtração é feita da mesma forma que a soma: stack-frame, alteração do [op], operação em si, checagem de overflow e destack-frame em saidaOp.

1.6.3 Multiplicação e Divisão:

Também acontecem da mesma forma que a soma e a subtração. A diferença é que as operações chamadas são mulss e divss, respectivamente.

1.6.4 Exponenciação:

```
;For (rcx = op2; rcx > 0; rcx --);  No loop, vamos ir multiplicando xmm0 por xmm1.
expLoop:
    cmp rcx, 0
    jle exitExpLoop     ;Loop para se rcx chegar a 0

    ;xmm0 - resultado   xmm1 - constante    rcx - numero de vezes a ser multiplica
    call multiplicacao  ;Multiplica xmm1 por xmm0. Armazena em xmm2
    movss xmm0, xmm2    ;Dai botamos o resultado dessa iteração em xmm0

    dec rcx             ;rcx--
    cmp rax, 0
    je expLoop          ;Se não deu overflow, continuamos o loop

    jmp exitExpLoop        ;Se deu overflow, codigo de erro
```

A exponenciação se resume a um laço for, que irá chamar a função de multiplicação sucessivamente.

Antes do laço, é identificado se o expoente é negativo e, se sim, insere-se 1 em rax, e é chamado a saída de erro.

1.6.5 Destack-Frame e Tratamento de Overflow:

```
;------------------------------ Flags auxiliares ------------------------------;
overflowExp:
    mov byte [op], "^"       ;Mudamos op para o simbolo do operadr
overflow:
    mov rax, 1
    jmp saidaOp

saidaOp:    ;Destack-frame genérico para todas as funções
    mov rsp, rbp        ;Desalocamos variáveis locais
    pop rbp             ;Recuoeramos ebp antigo + endereço de retorno
    ret                 ;Retorno
;------------------------------------------------------------------------------;
```

Como dito anteriormente, o destack-frame de todas as funções é feito sob a mesma flag (saidaOP), para economizar linhas e espço de memória. Aqui também podemos ver que, em caso de overflow, o retorno de rax é 1, para indicar erro.

1.7 Escrita no Arquivo

Por fim, o retorno das operações é comparado, e o resultado delas é escrito no arquivo nas funções escrevesolucaoOK e escrevesolucaoNotOK:

```
;------------------------------Função EscreveOK -------------------------------;
escrevesolucaoOK:
    push rbp
    mov  rbp, rsp       ;Stack-Frame

;   ------------------printf de debug
    ;mov rax, 3          ;Vamos passar 3 valores em pf
    ;mov rsi, rdi        ;Aqui tava o operador. Vamos mover ele pra rsi
    ;mov rdi, msDebugOP  ;E no rdi vai ficar a mensagem
    ;call printf

;   ------------------Preparamos o fprintf
    mov rax, 2
    mov rdi, [ptrArquivo]
    mov rsi, exeCorreta

    mov rdx, [op]

    cvtss2sd xmm0, xmm0
    cvtss2sd xmm1, xmm1
    cvtss2sd xmm2, xmm2
    call fprintf

    mov rsp, rbp        ;Desalocamos variáveis locais
    pop rbp             ;Recuoeramos ebp antigo + endereço de retorno
    ret                 ;Retorno
;------------------------------------------------------------------------------;
```

Basicamente, a função recebe o ponteiro para o arquivo aberto no modo append, e a função fprintf() escreve no arquivo os dados já armazenados nos registradores escalares. Depois disso, a execução do código termina.

1.8 Exemplo de Saída:

```
26      2.000000 ^ 0.000000 = 1.000000
27      0.000000 ^ 1.000000 = 0.000000
28      2.000000 ^ 2.000000 = 4.000000
29      3.000000 ^ 3.000000 = 27.000000
30      3.000000 ^ -3.000000 = funcionalidade não disponível
```