Ambiente de Execução para a linguagem C–

João Pedro Cabral Miranda e Vinicius Viana de Paula
Departamento de Engenharia de Computação e Sistemas Digitais, Escola Politécnica da USP, São Paulo, SP, Brasil

O Ambiente de Execução é um recurso dos compiladores modernos para possibilitar a execução do código do usuário. Essa ferramenta permite que o usuário abstraia diversos problemas de execução do programa. O presente trabalho tem como objetivo desenvolver uma versão desse programa para a linguagem C–. Além de executar o código-objeto, o ambiente criado é capaz de tratar erros de execução, evitar acesso a regiões reservadas de memória (*safety*), interfacear com dispositivos externos, salvar o contexto durante chamadas de funções - com o uso de uma pilha - e tem suporte para um modo de depuração.

***Index Terms*—Ambiente de Execução, C–, *safety*, MVN**

I. Introdução

O Ambiente de Execução é um programa responsável por implementar as abstrações da linguagem e executar o código-objeto do usuário. Ele executa o código compilado linha a linha, resolvendo todos os problemas de execução; como alocação de memória estática e dinâmica, acesso a bibliotecas estáticas e dinâmicas, interfaceamento com o sistema operacional, interação com dispositivos e tratamento de erros de execução[2].

Desse modo, trata-se de uma ferramenta de extrema importância para facilitar e abstrair o processo de execução dos programas, permitindo que o usuário possa se concentrar no desenvolvimento de seus programas, ao invés de se atentar as diversas questões e problemáticas da execução.

II. O Ambiente

Nesta seção serão debatidas de forma geral as principais funcionalidades do ambiente de execução desenvolvido para a linguagem C–, além de motivações que levaram a escolha dessas características e as hipóteses relativas ao funcionamento do compilador e o formato do arquivo gerado por ele.

Neste trabalho, foi solicitada a criação de um ambiente de execução simples capaz de executar o código do usuário linha a linha, além de utilizar a pilha disponível na instrução OS para salvar o contexto de uma função durante uma chamada de sub-rotina e restaurar o contexto durante o retorno de uma sub-rotina.

Contudo, foi utilizada uma abordagem diferente da prevista pelo enunciado. Além de executar o código do usuário linha a linha, o ambiente desenvolvido realiza diversas análises para garantir que não haja nenhum erro de execução ou acesso a regiões reservadas de memória. Ademais, não foi utilizada a pilha proporcionada pela instrução OS: foi desenvolvida uma pilha própria para impedir que o usuário alterasse diretamente o SP (*Stack Pointer*). Também foi adicionada uma funcionalidade de depuração, a partir de uma lista de *break-points* descrita na região de depuração do arquivo. Foram adicionadas sub-rotinas específicas para a implementação das funções *scan* e *print* para realizar a correta interação com a entrada (teclado) e a saída de dados (monitor), com as conversões necessárias (de ASCII para int e vice-versa).Com isso, foi criado um ambiente de execução seguro (*safety*) capaz de permitir a depuração do código do usuário para a linguagem C–.

Em um primeiro momento, será discutida a organização da memória da MVN para a implementação do ambiente de execução. Os endereços 000 a 908 (hexadecimal) são reservados para o código do ambiente, os endereços seguintes são usados para alocar o código-objeto e por fim os demais são utilizados pela pilha, lembrando que o ponteiro da pilha aponta para o primeiro endereço vazio e começa em FFE (hexadecimal).

Mediante as limitações da arquitetura da disciplina (MVN), como suporte para endereçamento de apenas 4KiB de memória, foram criadas hipóteses relativas ao formato do código-objeto de forma a otimizar o uso de memória e permitir que programas maiores - com certas restrições - possam ser executados nesse ambiente. Essas hipóteses foram baseadas no formato do arquivo executável da arquitetura ARM[1], na qual o arquivo é dividido em três partes: cabeçalho (contendo o tamanho da área de texto e da área de dados), texto (instruções a serem executadas) e dados (variáveis utilizadas pelo programa).

Para a presente implementação, o arquivo executável será dividido em 5 partes: cabeçalho (contendo 8 bytes, indicando os tamanhos das demais 4 partes), texto (conjunto de instruções executáveis), dados locais (variáveis locais de funções), dados globais (variáveis globais) e depuração (lista de break-points das instruções). Caso os valores indicados no cabeçalho estejam incorretos, o comportamento do ambiente é imprevisível.

Ressalta-se que para a adição das funcionalidades necessárias para a obtenção e impressão de dados, foi implementada, no ambiente, uma biblioteca dinâmica que exporta as funções *print* e *scan*, e importa os argumentos dessas sub-rotinas, de modo que para os código-objetos que utilizaram estes recursos, será necessário importá-los. Em todos os casos, é necessária a exportação dos argumentos por parte do código-objeto, dado que no arquivo .asm do ambiente há a manipulação destes argumentos nas sub-rotinas associadas a entrada e saída de dados - ainda que estas não sejam utilizadas, é necessária a exportação.

Os argumentos da sub-rotina *PRINT*, que implementa a função *print*, são 2 valores do tipo int (ARG_PT1 e ARG_PT3)

e um símbolo algébrico (ARG_PT2), conforme definição da expressão imprimível pela sub-rotina *PRINT* da linguagem C–. Veja que se o símbolo algébrico for +, -, * ou / a respectiva operação aritmética será realizada com os operandos ARG_PT1 e ARG_PT3 e então o resultado será impresso. Todavia, essa sub-rotina é executada de forma privilegiada; desse modo, caso a operação aritmética gera algum erro, este erro não será tratado. Além disso, caso o ARG_PT2 não seja nenhum desses 4 símbolos, então apenas ARG_PT1 será impresso.

Além disso, o argumento da sub-rotina *SCAN*, que implementa a função *scan*, é o endereço da variável que armazenará o valor lido do teclado. Pressupõe-se que sempre são digitados 4 dígitos entre 0 e 9 ou A e F (valores hexadecimais).

Assim, é pressuposto que o compilador C– é capaz de importar funções e exportar os argumentos, e de colocar tais variáveis, independente de sua existência no código inicial, sendo que para a utilização de *print* ou *scan* os argumentos utilizados são armazenados na área local.

Veja a seguir um exemplo de um arquivo em C– que simplesmente soma os valores de duas variáveis A e B, armazena em uma terceira C.

```
int A;
int B;
int C;
A = 21248;
B = 69;
C = A + B;
```

Seu código-objeto pode ser visto abaixo. Veja que a última instrução se trata de um HM (*Halt Machine*), isto é , o processador é travado e a simulação da MVN acaba, pois o ambiente espera que no código-objeto haja um HM para terminar a execução, caso contrário ela pode nunca acabar. Note que os rótulos TEXTO, LOCAL, GLOBAL e DEBUG são as 4 informações contidas no cabeçalho para definir o tamanho das 4 regiões seguintes do arquivo. Por fim, destaca-se a exportação dos argumentos das funções *print* e *scan*, apesar de não serem utilizados por este código-objeto.

```
          >   TEXTO    ; Exportar HEADER
          >   LOCAL    ;
          >   GLOBAL   ;
          >   DEBUG    ;
          >   ARG_PT1
          >   ARG_PT2
          >   ARG_PT3
          >   ARG_SN
          &   /0000
TEXTO     K   /0008
LOCAL     K   /0006
GLOBAL    K   /0000
DEBUG     K   /0004
                       ; TEXTO
          LD  A
DEBUG2    AD  B        ; AC = A + B
          MM  C
DEBUG1    HM  /0000
                       ; LOCAL
A         K   /5300
B         K   /0045
C         K   /0000
                       ; DEBUG
          LD  DEBUG1
          LD  DEBUG2
                       ; ARGS
ARG_PT1   K   /0000    ; print &
ARG_PT2   K   /0000    ; scan
ARG_PT3   K   /0000
ARG_SN    K   /0000
```

Com isso, é possível reutilizar a região de dados locais entre diferentes funções, dessa forma tornando o uso de memória mais eficiente. Contudo, caso haja chamada de função, toda a região de dados locais será empilhada na pilha do ambiente, enquanto no retorno de função toda a região será desempilhada. Desse modo, o contexto de cada função será restaurado após a execução de suas sub-rotinas.

Os processos de empilhamento e desempilhamento são realizados de forma automática pelo ambiente, com isso não é necessário incrementar instruções no código-objeto para realizar esses procedimentos durante chamadas ou retorno de função. Essa abordagem foi escolhida visto que devido a simplicidade proposta ao compilador C–, acredita-se que ele seria incapaz de acrescentar instruções a área de texto por conta própria. Além disso, essa modelagem permite uma maior segurança (*safety*) ao ambiente, dado que o usuário se torna incapaz de alterar diretamente o valor do SP (*Stack Pointer*).

Vale ressaltar que funções recursivas ou funções aninhadas não são suportadas pelo ambiente. Ademais, espera-se que o compilador acrescente a importação das funções *print* e *scan*, se forem utilizadas, bem como a exportação de seus argumentos, pois o ambiente apresenta uma biblioteca específica para entrada e saída de dados, realizando todo o processo de transformar ASCII para int e vice-versa.

Note que essas transformações, realizadas pelo ambiente, resultaram em um aumento considerável na complexidade do ambiente, pois existem diversos problemas relacionados a esses procedimentos. Por exemplo, a expressão dada no argumento da função *print* pode conter caracteres algébricos (+, -, *, /), além da necessidade de separar cada um dos dígitos para a utilização ou impressão adequada.

O ambiente executa cada instrução da região de texto linha a linha de forma segura (*safety*), ou seja, ele realiza diversas análises sobre cada instrução do código-objeto para garantir que não haja nenhum erro de execução. Quando um erro de execução é detectado uma certa mensagem é impressa no monitor e então a execução encerra com uma instrução HM. A forma como as sub-rotinas *print* e *scan* são executadas não dá o controle ao código-objeto sobre as instruções presentes em tais sub-rotinas, pois o usuário pode apenas chamar essas funções - SC *scan* ou SC *print* - quaisquer outros tipos de interação com essa biblioteca são considerados inválidos e levam ao erro de segmentação que será debatido abaixo.

Os erros detectados pelo ambiente são: *Segmentation Fault*, *Arithmetic Overflow*, *Division by Zero*, *Stack Overflow*, *Stack Underflow* e *Not Allowed*. A tabela I mostra as mensagens que estão associadas a cada um desses erros.

Tabela I
MENSAGENS DE ERRO DO AMBIENTE DE EXECUÇÃO

Mensagem	Erro
AO	*Arithmetic Overflow*
DZ	*Division by Zero*
NA	*Not Allowed*
SF	*Segmentation Fault*
SO	*Stack Overflow*
SU	*Stack Underflow*

O *Segmentation Fault* ocorre quando uma instrução, com exceção de LV e HM, tem como argumento um endereço que não pertence ao espaço alocado para a execução do programa, ou seja, a área de texto, de dados locais e dados globais. Logo, caso o programa tente acessar o cabeçalho do código-objeto, a região de depuração ou os trechos de memória que não foram alocados para o código do usuário, esse erro será disparado.

Contudo há uma exceção para essa regra, que se trata das chamadas das sub-rotinas *SCAN* e *PRINT* da biblioteca do ambiente. Essas sub-rotinas são executadas de forma privilegiada, de modo que possam acessar livremente toda a memória da MVN. Veja que isso não gera falhas de segurança, pois elas pertencem ao ambiente e o único endereço que elas acessam, que é determinado pelo usuário, é o *ARG_SN* que, se por acaso não pertencer ao espaço alocado para a execução do programa, isto é, a área de texto, de dados locais e dados globais, um erro de segmentação será lançado.

O *Arithmetic Overflow* ocorre em três casos distintos: *overflow* nas instruções AD, SB e ML. Para uma instrução de soma, o *overflow* é gerado pela soma de dois números de mesmo sinal que resulta num número com sinal oposto. Já no caso da subtração, o *overflow* acontece quando os dois operandos têm sinais opostos e o resultado tem sinal contrário ao primeiro operando. Por fim na multiplicação, o *overflow* é obtido quando dois números são multiplicados - sendo pelo menos um deles não nulo - e o resultado, quando dividido por um dos dois, não resulta no outro número usado para a multiplicação.

O *Division By Zero* acontece quando o valor do endereço acessado pela instrução DV é zero, ou seja, um erro na execução da instrução 7.

O *Stack Overflow* ocorre quando o ambiente tenta empilhar algo e o valor de SP (*Stack Pointer*) é menor do que o menor endereço disponível para a pilha, ou seja, o ambiente está tentando empilhar na área reservada do código-objeto, enquanto o *Stack Underflow* acontece quando o ambiente tenta desempilhar com o SP valendo FFE, pois neste caso a pilha está vazia.

Por fim, o *Not Allowed* ocorre quando há uma chamada para uma instrução cuja operação é OS, PD ou GD, dado que não existe forma de usar tal comando na linguagem C–. Desse modo, significa que se trata de um uso malicioso dessa instrução. Note que, caso o usuário desejar interagir com dispositivos, ele deve usar as funções *scan* e *print*.

Além disso, o ambiente de execução tem uma funcionalidade de depuração na qual, dada uma lista de *break-points* - determinada pela região de depuração do código-objeto -, a execução do programa é interrompida até que o usuário digite algum caractere. Para isso, foi utilizada a instrução GD /000 com o intuito de interromper a execução momentaneamente. As linhas da região de depuração devem ser preenchidas de modo a apresentarem o formato LD X, onde X é o rótulo da linha que deve ser depurada. Veja o exemplo abaixo que mostra a estrutura dessa região.

```
          &   /0000
          V   ARG_SN
          V   ARG_PT3
          V   ARG_PT2
          V   ARG_PT1
          V   DEBUG
          V   GLOBAL
          V   LOCAL
          V   TEXTO     ; Exportar HEADER
TEXTO     K   /000C
LOCAL     K   /0006
GLOBAL    K   /0000
DEBUG     K   /0004
                        ; TEXTO
          LV  =2
          MM  B         ; B = 2
DEBUG1    LD  A
          DV  B
          MM  C         ; C = A / B
DEBUG2    HM  /0000
                        ; LOCAL
A         K   /000A
B         K   /0000
C         K   /0000
                        ; DEBUG
          LD  DEBUG1
          LD  DEBUG2
                        ; ARGS
ARG_PT1   K   /0000     ; print &
ARG_PT2   K   /0000     ; scan
ARG_PT3   K   /0000
ARG_SN    K   /0000
```

III. DESCRIÇÃO GERAL DA IMPLEMENTAÇÃO

Para o correto funcionamento do ambiente de execução desenvolvido, foi necessária a utilização de múltiplas variáveis que armazenam endereços relevantes à execução - delimitadores das regiões do código, indicando o início de cada uma delas - e de ponteiros para alguns desses endereços específicos, que colaboram nas movimentações efetuadas, a depender do código-objeto. Desse modo, é inicialmente reconhecida a delimitação das distintas regiões da memória - áreas de instruções, de variáveis locais, de variáveis globais, de depuração e a área livre que pode ser utilizada pela pilha -, facilitando assim, a compreensão de qual área está sendo manipulada, e o controle de movimentações nas regiões adequadas - sendo possível identificar os casos de ultrapassagem dos limites da respectiva região manipulada.

A partir do ponteiro do primeiro endereço da área de instruções, podem ser obtidas as instruções, uma a uma, movendo o ponteiro para o endereço posterior a cada execução

completa. Para a existência de um controle da execução das instruções e de uma forma de identificar eventuais erros, cada instrução recebe um tratamento, de acordo com o seu OPCODE. Operações que possuem semelhanças acabam apresentando, também, tratamentos análogos, como é o caso das operações aritméticas, ou até o mesmo tratamento, no caso das operações de movimentação do acumulador para a memória e da memória para o acumulador. As instruções contendo operações LV (*Load Value*) e HM (*Halt Machine*) não apresentam um tratamento particular, sendo diretamente executadas, visto que essas operações não acessam endereços de memória, não alteram a pilha e nem realizam operações aritméticas. Por fim, as operações OS (*Operating System*), GD (*Get Data*) e PD (*Put Data*), são operações não permitidas pelo ambiente de execução. Para o caso das duas últimas, a funcionalidade de *scan* e *print* são implementadas pelo próprio ambiente.

Além disso, os eventuais erros encontrados na etapa de tratamento da operação são apresentados no monitor, havendo distinção entre os erros com o intuito de facilitar a identificação da falha. Assim como já foi citado anteriormente, pode haver ainda casos de *Segmentation Fault*, e para a análise e identificação de tais casos há um tratamento que identifica a região da memória que está sendo acessada e manipulada. Este tratamento ocorre para todas as instruções em que isto pode acontecer.

Por fim, após os devidos tratamentos, caso não seja encontrado nenhum erro na instrução sendo analisada, pode ser efetuada a sua execução, para os casos nos quais não há desvios, nem chamadas ou retornos de sub-rotinas, pois as instruções que apresentam operações associadas a estes casos particulares não são efetivamente executadas, a fim de manter o controle da sequência de instruções subsequentes. Entretanto, tais instruções tem uma finalidade no código-objeto, de modo que é fundamental a ocorrência de manipulações dos próximos endereços, no código-objeto, a serem acessados, bem como o que deve estar armazenado no acumulador, e o armazenamento de um endereço de retorno no caso da chamada de sub-rotinas. Estas situações serão analisadas de forma mais detalhada quando forem abordados os tratamentos de cada operação.

Ademais, é relevante destacar a utilização de sub-rotinas associadas ao empilhamento e desempilhamento, que são fundamentais quando se trata de uma instrução apresentando uma chamada de sub-rotina. Anteriormente ao acesso à sub-rotina, deve ser efetuado o armazenamento do contexto anterior, da rotina que chamou esta sub-rotina, ou seja, todas as variáveis locais presentes devem ser armazenadas, para que, havendo manipulação de quaisquer variáveis na sub-rotina, o contexto da rotina que a chamou não seja afetado quando ocorrer o retorno às suas instruções. Ambos os processos apresentam capacidade de manipular o SP (*Stack Pointer*), que sempre deve apontar para o primeiro endereço vazio após o topo da pilha, permitindo, deste modo, a inserção e remoção da quantidade de variáveis locais que for necessária - dentro do limite do espaço reservado para a pilha.

Destaca-se, também, a necessidade de se conhecer os limites da região da memória destinada à pilha, de modo que a pilha não apresente tamanho maior que o máximo permitido - caso de *Stack Overflow* -, e que o SP não aponte para um endereço inexistente do programa (*Stack Underflow*).

Para a entrada e saída de dados, é utilizada uma área especial do código do ambiente, denominada de biblioteca. A biblioteca contém as sub-rotinas *PRINT* e *SCAN* que podem ser utilizadas para a implementação de obtenção e impressão de dados em um código-objeto. Além dessas duas sub-rotinas, existem aquelas utilizadas para o desenvolvimento adequado das duas primeiras. Vale ressaltar que, antes da chamada/retorno dessas sub-rotinas não há operações com a pilha, pois a sub-rotina *PRINT* não modifica os valores da área de dados locais e a sub-rotina *SCAN* apenas modifica o valor da variável endereçada por ARG_SN.

Finalmente, deve-se pontuar a inserção de mecanismos para que a depuração seja um recurso adicional ao ambiente. Para tal, permite-se a depuração de linhas previamente marcadas e, após o tratamento da operação e anteriormente à execução propriamente dita, é interrompido o processo, de modo que é possível analisar os valores armazenados nos registradores, em especial o acumulador, com o objetivo de verificar se o programa se comporta como esperado. Em sequência, tendo sido finalizado o estudo da condição do programa, retoma-se o processo no momento desejado.

IV. Descrição Detalhada dos Módulos

A. Main

Inicialmente, são armazenados os endereços das sub-rotinas exportadas, *PRINT* e *SCAN*. O programa principal (*MAIN*) também apresenta a atribuição de valores a variáveis associadas à identificação do topo de cada uma das áreas da memória. Além disso, é utilizado um ponteiro para o endereço da instrução atual a ser executada, na área de texto, denominado *PONTEXT*. Dado esse ponteiro, é iniciado um loop para o qual cada instrução é obtida, incrementando-se em 2 o ponteiro a cada instrução. A partir da instrução, é efetuado o seu tratamento, utilizando o módulo *TRATOP*, e em sequência, há o armazenamento do valor no acumulador, garantindo que o comportamento do código-objeto se dará de modo adequado. A finalização da execução se dá a partir de um HM no próprio código-objeto, indicando que era a última instrução, ou em algum caso de erro. A *MAIN* é mostrada a seguir, e é possível observar as atribuições iniciais, bem como o loop apresentando a manipulação do *PONTEXT* e a utilização do seu módulo de tratamento de operações.

```
MAIN    LV  PRINT    ; Armazena o
        MM  ADDR_PT  ; endereço das
        LV  SCAN     ; subrotinas
        MM  ADDR_SN  ; exportadas
        LV  TEXTO
        AD  Cte8     ; TOPTEXT = 8 +
        MM  TOPTEXT  ; & TEXTO;
        AD  TEXTO    ; TOPLOC =
        MM  TOPLOC   ; TOPTEXT + TEXTO;
        AD  LOCAL    ; TOPGLO =
        MM  TOPGLO   ; TOPLOC + LOCAL;
        AD  GLOBAL   ; TOPDEB =
```

```
         MM   TOPDEB   ; TOPGLO + GLOBAL;
         AD   DEBUG    ; TOPLIV =
         MM   TOPLIV   ; TOPDEB + DEBUG
         AD   READ
         MM   PONTLIV
         LD   TOPTEXT  ; PONTEXT =
         AD   READ     ; TOPTEXT + READ
LOOPM    MM   PONTEXT
         MM   RD_INST
RD_INST  K    /0000    ; Obter instrução;
         MM   INSTRU   ; Tratar e executar
         JP   TRATOP   ; operação;
POSTRAT  MM   ACUMU    ; ACUMU = valor do
         LD   PONTEXT  ; AC pós-execução;
         AD   Cte2     ; PONTEXT=PONTEXT + 2
         JP   LOOPM    ; Loop
FIMAIN   HM   FIMAIN   ; Fim da execução
```

B. TRATOP

O módulo *TRATOP* analisa o opcode da instrução e realiza o tratamento necessário, após isso a instrução é executada. Inicialmente, esse módulo obtém o opcode da instrução atual, armazenando na variável *OPCODE*, após isso realiza uma lógica de ”if-else”para determinar qual o tipo de tratamento que a instrução deve receber (caso seu opcode seja 0, 1, 2, 4, 5, 6, 7, 8, 9, A, B, D, E ou F). Caso ela não precise de tratamento, ou seja, seu opcode é 3 ou C, a instrução será diretamente executada. Veja que antes de qualquer instrução ser executada, a sub-rotina *DEPURA* é chamada, para realizar a depuração caso necessário, e então o valor do acumulador é restaurado para o valor obtido após a execução da instrução anterior do usuário. A seguir serão explicadas as subdivisões desse módulo, responsáveis por tratar cada tipo de instrução.

Inicialmente, analisando os módulos de tratamento das operações de desvio (0, 1 e 2), identifica-se a necessidade de verificar se ocorre *Segmentation Fault*. Para isso, é utilizada a sub-rotina *TratADR*. No caso da *Trata0*, é verificado o caso de ser uma linha contendo um *break-point*, por meio da chamada da sub-rotina *DEPURA*, e posteriormente já se altera o valor do ponteiro *PONTEXT*, para a ocorrência do desvio.

Os módulos *Trata1* e *Trata2* se iniciam de modo análogo, porém após o retorno da *DEPURA*, eles carregam o valor armazenado no acumulador, correspondente ao valor do acumulador resultante da execução da instrução anterior do código-objeto, e verificam se a condição de desvio é satisfeita, ou seja, se o valor armazenado no acumulador é zero ou negativo, para as operações 1 e 2, respectivamente. Caso as condições sejam satisfeitas, o ponteiro *PONTEXT* é modificado de modo a atender o desvio, caso contrário a próxima instrução é buscada para execução. O módulo *Trata1* é apresentada no trecho adiante, e pode ser utilizada para compreender os outros dois módulos de tratamento de desvios.

```
Trata1   SC   TratADR  ;
         SC   DEPURA   ; Depuração
         LD   ACUMU
         JZ   PONT1    ; JZ falhou:
         JP   POSTRAT  ; Instr seguinte
PONT1    LD   ADDR     ; PONTEXT = ADDR +
         AD   READ     ; READ (sucesso no JZ)
         JP   LOOPM    ; Próxima iteração
```

O *Trata4* trata as instruções de soma, primeiro chamando a sub-rotina *TratADR* para verificar se há *Segmentation Fault*. Após isso, obtém-se o valor armazenado no endereço apontado pela instrução, por meio da sub-rotina *GETVAR*. Por fim, analisa-se se houve *overflow*, observando se ambos os operandos têm mesmo sinal e se o resultado da soma tem sinal oposto ao dos operandos. O código desse tratamento pode ser visto a seguir.

```
Trata4   SC   TratADR
         SC   GETVAR   ; Obter a variável
         JN   NEG4     ; Variável negativa
         LD   ACUMU    ; ACUMU < 0 e VAR > 0,
         JN   LD_EXEC  ; Sem AO;
         AD   VAR      ; VAR + ACUMU < 0,
         JN   ERRORAO  ; VAR, ACUMU > 0, AO;
         JP   LD_EXEC  ; VAR + ACUMU > 0
NEG4     LD   ACUMU    ; VAR, ACUMU > 0,
         JN   NEG4_2   ; Sem AO;
         JP   LD_EXEC  ; ACUMU > 0 e VAR < 0,
NEG4_2   AD   VAR      ; Sem AO;
         JN   LD_EXEC  ; VAR + ACUMU < 0,
         JP   ERRORAO  ; VAR,ACUMU<0,Sem AO
```

O *Trata5* tem funcionamento análogo ao *Trata4*, com exceção à detecção de *overflow*, na qual os dois operandos devem ter sinais opostos e o resultado da subtração deve ter sinal oposto ao primeiro operando.

Para o caso da multiplicação, é utilizado o módulo *Trata6*. Ele se inicia chamando *TratADR* e, logo em seguida, *GETVAR* para obter a variável introduzida na instrução. Esta variável é escrita em uma posição de armazenamento temporário e, então, é recarregado o valor guardado no acumulador, para que se possa efetuar a multiplicação e verificar a condição de *overflow*, que ocorre quando o sinal do resultado armazenado é diferente do sinal esperado da multiplicação. Para isso, utiliza-se a expressão

$$\frac{AC * VAR}{AC} - VAR,$$

e é verificado se este valor que passa a ser armazenado no acumulador é zero, indicando que não houve *overflow*.

```
Trata6   SC   TratADR
         SC   GETVAR   ; Obter variável
CHECKAC  MM   VAR
         LD   ACUMU
MULTI    MM   VAR2
         JZ   LD_EXEC
         ML   VAR      ; AC = VAR * ACUMU
         DV   VAR2     ;        / ACUMU
         SB   VAR      ; AC = AC - VAR
         JZ   LD_EXEC
ERRORAO  LD   AO       ; arithmetic overflow
         PD   /100
         HM   FIMAIN
```

Para as instruções de divisão utiliza-se o *Trata7*, nele é verificado se há algum acesso de endereço reservado, após

isso obtém-se a variável armazenada no endereço apontado pela instrução. Caso tal variável seja 0, isso leva o ambiente a imprimir DV (*Division By Zero*) no monitor e encerrar a execução, já que dividir por 0 é uma operação aritmética inválida.

As instruções com opcode 8 e 9 recebem o mesmo tratamento (*Trata89*), que consiste em apenas chamar a sub-rotina *TratADR* para verificar se não houve nenhum acesso a alguma região reservada de memória, e então, sabendo-se que o endereço passado é válido, executam a instrução.

As instruções cujas operações estão vinculadas com chamada e retorno de sub-rotinas são tratadas utilizando as sub-rotinas *TrataA* e *TrataB*, respectivamente. Ambas se iniciam chamando, em sequência, as sub-rotinas *TratADR* e *DEPURA*, para que seja garantido que não ocorra *Segmentation Fault*, e seja verificada a possibilidade de ser uma linha de *breakpoint*. Após o retorno da segunda sub-rotina, há uma terceira chamada, da *EMP*, no caso de *TrataA*, e da *DMP*, no caso de *TrataB*. Isto ocorre porque é necessário salvar o contexto antes de uma operação A, e é preciso restaurar o contexto anterior após uma operação B.

Para o *TrataA*, há uma escrita do endereço de retorno - *PONTEXT* + 2 - no cabeçalho da sub-rotina chamada, seguida da modificação do ponteiro *PONTEXT* para obtenção da primeira instrução no interior da sub-rotina. Tal processo é verificado a seguir. Além disso, destaca-se a identificação dos casos de *PRINT* e *SCAN*, que são manipulados de forma distinta.

```
TrataA  SC  GETADDR
        SB  ADDR_PT   ; ADDR = PRINT:
        JZ  SC_PT     ; Exec Privilegiada
        LD  ADDR
        SB  ADDR_SN   ; ADDR = SCAN:
        JZ  SC_SN     ; Exec Privilegiada
        SC  TratADR   ; SegFault
        SC  DEPURA    ; Depuração
        SC  EMP       ; A: Empilhar
        LD  ADDR      ; variáveis locais
↪
        AD  WRITE
        MM  WRTA
        LD  PONTEXT
        AD  Cte2    ; Endereço de retorno
        SB  READ    ; da subrotina:
WRTA    K   /0000   ; PONTEXT + 2
        LD  ADDR
        AD  READ      ; PONTEXT =
        MM  PONTEXT   ; ADDR + READ
        LD  ACUMU
        JP  POSTRAT   ; Fim do tratamento
SC_PT   SC  DEPURA    ; Depuração:
        LD  ADDR      ; SCAN e PRINT:
        AD  WRITE     ; Exec privilegiada
        MM  WRTA2
        LV  POSTRAT   ; Retorno:
WRTA2   K   /0000     ; POSTRAT
        JP  INI_PT    ; Executar print
SC_SN   SC  DEPURA    ; Depuração:
        LD  ADDR      ; SCAN e PRINT:
        AD  WRITE     ; Exec privilegiada
        MM  WRTA3
        LV  POSTRAT   ; Retorno:
WRTA3   K   /0000     ; POSTRAT
        JP  INI_SN    ; Executar scan
```

E para a *TrataB*, o endereço de retorno é lido do cabeçalho da sub-rotina, de modo que o ponteiro *PONTEXT* é modificado com o objetivo de retornar a rotina que fez a chamada, na instrução seguinte àquela que levou a chamada da sub-rotina.

```
TrataB  SC  TratADR   ; Checa SegFault
        SC  DEPURA    ; Depuração
        SC  DMP       ; B: Desempilhar
        LD  ADDR
        AD  READ
        MM  READB
READB   K   /0000     ; PONTEXT =
        AD  READ      ; Endereço de retorno
        MM  PONTEXT   ; da subrotina
        JP  LOOPM     ; Próxima iteração
```

Por fim, há a sub-rotina *TratDEF*, para as operações GD, PD e OS. Como tais operações não são permitidas, este tratamento é simplesmente resumido a uma indicação de erro e encerramento da execução.

C. TratADR

A sub-rotina *TratADR* é responsável por verificar se o endereço acessado pela instrução está dentro da região permitida (área de texto, dados locais e dados globais). Para isso, ela chama a sub-rotina *GETADDR* para obter o endereço acessado pela instrução, e em seguida é verificado se o endereço está contido na região delimitada por *TOPTEXT* (topo da área de texto) e *TOPDEB* (topo de área de depuração). Em caso positivo, a execução continua normalmente, caso contrário imprime-se a mensagem SF no monitor e finaliza-se a execução do programa. O código dessa sub-rotina pode ser visto abaixo.

```
; Sub-rotina TratADR(RESS)
TratADR K   /0000
        SC  GETADDR ; endereço do OI
        SB  TOPTEXT ; ADDR < TOPTEXT
        JN  ERRORSF ; SegFault
        LD  ADDR
        SB  TOPDEB  ; Seguir para a próx
        JN  NEXT    ; parte do tratamento
ERRORSF LD  SF      ; Erro de segmentação
        PD  /100
        HM  FIMAIN
NEXT    RS  TratADR ; Retorno
```

A sub-rotina *GETADDR* é responsável por obter o endereço que será acessado pela instrução a ser executada. Para isso, ela parte do opcode da instrução que está sendo analisada para obter o campo do endereço, usando operações matemáticas. Ela armazena esse endereço na variável *ADDR* e o retorna no acumulador.

A sub-rotina *GETVAR* captura o valor armazenado no endereço contido na variável *ADDR*. Ela acessa esse endereço

e então obtém a variável armazenada nele e guarda em *VAR* e retorna esse valor no acumulador.

D. EMP e DMP

Ademais, para que o comportamento esperado pela execução do código-objeto seja também verificado no caso deste apresentar sub-rotinas, é relevante que o contexto anterior ao acesso ao trecho de uma sub-rotina seja salvo em uma região específica da memória, que pode ser acessada após o retorno à rotina de chamada, com o intuito de restaurar este contexto inicial. Nessa perspectiva, é adotada a pilha como forma de armazenar os dados locais do contexto, logo são utilizadas duas sub-rotinas associadas ao empilhamento e desempilhamento: *EMP* e *DMP*.

A sub-rotina *EMP*, como o nome indica, implementa o que é necessário lidar para que o empilhamento ocorra, desde manipulação do SP (*Stack Pointer*) - para mantê-lo apotando para o primeiro espaço da memória subsequente à última variável armazenada na pilha - até verificação da possível ocorrência de acesso de outra área da memória, a área de depuração, ocasionando um *Stack Overflow*. Esta sub-rotina conta com a utilização de dois ponteiros, *PONTLOC* e *PONTGLO*, que representam, respectivamente, o ponteiro para o endereço da variável local a ser armazenada (inicialmente, aponta para o primeiro endereço da área de dados locais) e o ponteiro para o primeiro endereço da área de variáveis globais.

Assim, é apresentado um loop para o qual o ponteiro *PONTLOC* percorre a área de dados locais por completo - a condição de saída do loop é a equivalência entre *PONTLOC* e *PONTGLO*, indicando que todas as variáveis locais já foram acessadas - e, para cada posição acessada da área especificada, efetua-se uma leitura da variável armazenada que é, então, armazenada na pilha. O SP é manipulado de forma coerente a este armazenamento, e em cada caso, é analisada a possibilidade de acesso a região de depuração, indisponível para a pilha, o que ocasionaria, desse modo, o *Stack Overflow*.

A seguir, é apresentado o loop da sub-rotina *EMP*, sendo possível observar o comportamento descrito anteriormente.

```
LOOPEMP MM   PONTLOC
        SB   PONTGLO ; PONTGLO = PONTLOC:
        JZ   FIMEMP  ; Fim da subrotina
        LD   PONTLOC
        MM   LEITURA ; Leitura da variável
LEITURA K    /0000   ; apontada pelo
        MM   TEMPEMP ; PONTLOC
        LD   SP
        SB   TOPLIV  ; Stack Overflow:
        JN   ERROEMP ; SP > TOPLIV
        LD   SP
        AD   WRITE
        MM   WRT_TOP ; WRT_TOP=WRITE + SP
        LD   TEMPEMP
WRT_TOP K    /0000   ; Mem[SP] = TEMPEMP
        LD   SP
        SB   Cte2    ; Empilhar:
        MM   SP      ; SP = SP - 2
        LD   PONTLOC
        AD   Cte2    ; PONTLOC=PONTLOC + 2
        JP   LOOPEMP ; Recomeçar o LOOP
```

Por sua vez, a sub-rotina *DMP* se inicia de modo análogo à *EMP*, utilizando do ponteiro *PONTLOC* para estar sempre identificando o endereço da área de dados locais que está sendo acessado, e do ponteiro *PTOPLOC*, que aponta para o primeiro endereço da área de dados locais. No caso da *DMP*, o *PONTLOC* é inicialmente manipulado de modo a apontar para o último endereço da área de variáveis locais, e percorre tal área em um loop, até chegar no topo. A condição de saída deste loop é a comparação dos valores dos ponteiros citados, sendo que a repetição é encerrada para o caso de *PONTLOC* ser menor que *PTOPLOC*.

O acesso a cada variável local, com o objetivo de desempilhá-la e restaurá-la em sua posição na memória, é feito manipulando-se o SP, no próprio loop, onde também está presente a operação de escrita na memória da variável. A condição de *Stack Underflow* também é verificada para cada desempilhamento.

O loop da sub-rotina *DMP* está apresentado no trecho subsequente, de modo que é possível identificar a funcionalidade da sub-rotina.

```
LOOPDMP MM   PONTLOC
        SB   PTOPLOC ; PONTLOC < PTOPLOC:
        JN   FIMDMP  ; Fim da subrotina
        LD   SP
        AD   Cte2
        SB   Cte1000
        JZ   ERRODMP
        LD   SP
        AD   Cte2
        MM   SP      ; SP = SP + 2
        AD   READ
        MM   RD_TOP  ; RD_TOP=READ + SP + 2
RD_TOP  K    /0000   ; TEMPDMP =
        MM   TEMPDMP ; Topo da pilha
        LD   PONTLOC
        MM   ESCRITA
        LD   TEMPDMP ; Mem[PONTLOC] =
ESCRITA K    /0000   ; TEMPDMP
        LD   PONTLOC
        SB   Cte2    ; PONTLOC=PONTLOC - 2
        JP   LOOPDMP ; Recomeçar o LOOP
```

E. DEPURA

A sub-rotina *DEPURA* é responsável por analisar se a linha de execução atual, apontada por *PONTEXT*, está associada a um *break-point* e, em caso positivo, por travar a execução do programa com um GD /000. Primeiramente, cria-se o ponteiro para acesso da região de depuração *PONTDEB* e, então, entra-se em um loop denominado *LOOPDB*.

Neste *LOOPDB*, é verificado se o ponteiro *PONTDEB* alcançou o valor de *PONTLIV*, sendo que o caso positivo significa que a linha atual não está presente na lista de depuração e pode ser executada normalmente. Caso o *break-point* da linha apontado por *PONTDEB* seja igual a *PONTEXT*, então o *break* deve ser ativado, ou seja, deve ser executada

uma instrução GD /000 para travar a execução. Contudo, se forem diferentes, o ponteiro é incrementado em 2 e o loop é recomeçado *LOOPDB* até que *PONTDEB* atinja o valor de *PONTLIV* ou *PONTEXT* seja igual a algum *break-point*.

Veja que para cada *PONTEXT* sua igualdade é testada com todos os *break-points*, pois devido as instruções de desvio e chamada ou retorno de sub-rotina, os valores de *PONTEXT* podem diminuir ou aumentar ao longo da execução. A seguir, está apresentado o código dessa sub-rotina.

```
         &    /0700
PONTDEB  K    /0000
                        ; DEPURA
DEPURA   K    /0000
         LD   TOPDEB   ; Conferir se PONTEXT
         AD   READ     ; está na lista de
LOOPDB   MM   PONTDEB  ; depuração;
         SB   PONTLIV  ; Fim da lista:
         JZ   FIMDEPU  ; PONTLIV = PONTDEB
         LD   PONTDEB
         MM   BREAKP   ; BREAKP = PONTDEB
BREAKP   K    /0000    ; BREAK-POINT
         SB   PONTEXT
         JZ   BREAK    ; PONTEXT = *PONTDEB
         LD   PONTDEB
         AD   Cte2     ; PONTDEB=PONTDEB + 2
         JP   LOOPDB   ; Recomeçar loop
BREAK    GD   /000     ; BREAK-POINT
FIMDEPU  RS   DEPURA   ; Fim da depuração
```

F. Biblioteca

Finalmente, serão analisadas as sub-rotinas associadas as funções exportadas pelo ambiente, de modo que o código-objeto possa utilizar as funcionalidades de obtenção e impressão de dados.

Inicialmente, é desenvolvida a sub-rotina *CH2I*, responsável por efetuar a conversão de um caractere *ASCII* para *int*, assim como apresentado a seguir.

```
                        ; Subrotina CH2I
CH2I     K    /0000
         LD   CHAR
         DV   Cte100
         MM   TEMPCH   ; TEMPCH = CHAR/100
         ML   Cte100
         SB   CHAR
         ML   Cte_1    ; CHAR2I = (-1) *
         MM   CHAR2I   ; (TEMPCH*100 - CHAR)
         SC   PEGAINT
         MM   TEMPCH2  ; TEMPCH2 virou int
         LD   TEMPCH
         MM   CHAR2I
         SC   PEGAINT
         MM   TEMPCH   ; TEMPCH virou int
         ML   Cte10
         AD   TEMPCH2  ; INT =
         MM   INT      ; TEMPCH*10 +TEMPCH2
         RS   CH2I
```

A sub-rotina *PEGAINT* está associada a obtenção do inteiro, dado um caractere isolado pela *CH2I*. Seu código é exposto adiante.

```
; Subrotina PEGAINT
PEGAINT  K    /0000
         LD   CHAR2I   ; CHAR2I < 40:
         SB   Cte40    ; CHAR2I está
         JN   DEC      ; entre 0 e 9
         LD   CHAR2I
         SB   Cte37
         MM   CHAR2I   ; CHAR2I agora é int
         RS   PEGAINT
DEC      LD   CHAR2I
         SB   Cte30
         MM   CHAR2I   ; CHAR2I agora é int
         RS   PEGAINT
```

Por sua vez, a sub-rotina *SCAN* se inicia analisando os casos de *Segmentation Fault* do argumento ARG_SN, e posteriormente obtém os quatro bytes da entrada, dois a dois, que são convertidos para inteiro e armazenados de modo adequado, ou seja, em uma única variável int (2 bytes). A *SCAN* é mostrada logo abaixo.

```
                        ; Subrotina SCAN
SCAN     K    /0000
INI_SN   LD   ARG_SN
         SB   TOPTEXT  ; ARG_SN-TOPTEXT<0
         JN   ERRORSF  ; SegFault
         LD   ARG_SN
         SB   TOPDEB
         JN   GETDAT
         JP   ERRORSF
GETDAT   GD   /000     ; Pegar os dois
         MM   CHAR     ; primeiros bytes
 ↪
         SC   CH2I     ; Converter para int
         ML   Cte100   ; Shift de 2 casas
         MM   TEMPSN
         GD   /000     ; Pegar os dois
         MM   CHAR     ; últimos bytes
         SC   CH2I     ; Converter para int
         AD   TEMPSN   ; Obtendo o int
         MM   ARG_SN   ; digitado
         RS   SCAN
```

Por fim, a sub-rotina *PRINT*, dada em sequência, é iniciada verificando qual das operações será efetuada - caso haja uma operação, caso contrário a variável é impressa diretamente. Depois, se a operação é existente, será efetuada, e o resultado será modificado de modo a atender os requisitos da impressão (convertido em hexadecimal), utilizando uma sub-rotina auxiliar, *IMPINT*, responsável por separar cada um dos quatro dígitos do número a ser impresso, e adequá-los à notação hexadecimal.

```
; Subrotina PRINT
PRINT    K    /0000
INI_PT   LD   ARG_PT2 ; Verifica se é
         SB   Cte2A   ; multiplicação
         JZ   ML_PT
```

```
        LD   ARG_PT2 ; Verifica se é
        SB   Cte2B   ; adição
        JZ   AD_PT
        LD   ARG_PT2 ; Verifica se é
        SB   Cte2D   ; subtração
        JZ   SB_PT
        LD   ARG_PT2 ; Verifica se é
        SB   Cte2F   ; Divisão
        JZ   DV_PT
        LD   ARG_PT1 ; Caso contrário
WR_INT  MM   INT     ; é só uma var;
        SC   IMPINT  ; Imprime int
        RS   PRINT
ML_PT   LD   ARG_PT1 ; Efetua a
        ML   ARG_PT3 ; operação
        JP   WR_INT
AD_PT   LD   ARG_PT1 ; Efetua a
        AD   ARG_PT3 ; operação
        JP   WR_INT
SB_PT   LD   ARG_PT1 ; Efetua a
        SB   ARG_PT3 ; operação
        JP   WR_INT
DV_PT   LD   ARG_PT1 ; Efetua a
        DV   ARG_PT3 ; operação
        JP   WR_INT
```

V. Testes e Verificação

Para a verificação do comportamento adequado do ambiente de execução desenvolvido, foi essencial a construção de um conjunto de códigos-objeto, seguindo a estrutura esperada para todos os trechos de código, de modo que, na execução das instruções, todas as operações tivessem seu funcionamento testado, incluindo os casos em que as operações podem falhar.

Nesse sentido, deve-se inicialmente considerar quais as operações que acessam endereços de memória, alteram a pilha ou realizam operações aritméticas, dado que tais operações devem ter o seu correto funcionamento verificado e, em alguns casos, é necessário analisar a forma como o ambiente lida com os possíveis erros que surgem nas instruções envolvendo tais operações. Assim, consideraremos as instruções envolvendo as operações de desvio - JP, JZ e JN -, de aritmética - AD, SB, ML e DV -, de movimentação entre memória e acumulador - LD e MM -, e de chamada e retorno de sub-rotinas - SC e RS. Tal consideração é feita porque as demais instruções são diretamente executadas, por não apresentarem riscos de causar problemas durante a execução - com exceção da instrução OS, que não é suportada pelo ambiente.

Ademais, vale destacar a possibilidade de ocorrência de *Segmentation Fault*, de modo que se deve atentar à região da memória que está sendo manipulada. A existência de tal problema ocorre no acesso indevido da memória, por uma instrução envolvendo uma das operações acima. Para verificar a captura de erros associados a este acesso inadequado, basta apresentar no código-objeto, deste modo, instruções envolvendo endereços de regiões diferentes das áreas de texto, de variáveis locais, e de variáveis globais.

Para o caso das instruções envolvendo desvios, o tratamento se baseia na manipulação do endereço da instrução a ser executada, por parte do ambiente, de modo que há uma mudança de endereço de acordo com a instrução a ser obtida no desvio. O exemplo a seguir contém casos de desvio (JZ e JP), e o seu correto funcionamento ao utilizá-lo como código-objeto indica que as operações de desvio estão sendo tratadas de forma adequada pelo ambiente de execução. Ressalta-se que a operação JN também foi testada, em outros códigos-objeto.

```
        &VVVVVVVVV TEXTO    ; Exportar HEADER
                   LOCAL    ;
                   GLOBAL   ;
                   DEBUG    ;
                   ARG_PT1
                   ARG_PT2
                   ARG_PT3
                   ARG_SN
                   /0000
TEXTO   K    /0022
LOCAL   K    /0008
GLOBAL  K    /0004
DEBUG   K    /0000
                         ; TEXTO
        LV   =3
        MM   D           ; D = 3
        SC   SUM
        MM   C           ; C = A + B
        LD   D
        SB   P          ; Desvio se a condição
        JZ   FIRSTIF    ; do if for satisfeita
        LD   O
        MM   C           ; C = O
        JP   END
FIRSTIF LD   P          ; if(D - P == 0)
        MM   C           ;  C = P
        HM   /0000
SUM     K    /0000       ; Sub-rotina SUM
        LD   A
        AD   B           ; AC = A + B
        RS   SUM
                         ; LOCAL
A       K    /5300
B       K    /0045
C       K    /0000
D       K    /0000
                         ; GLOBAL
P       K    /5050
O       K    /4F4F
                         ; ARGS
ARG_PT1 K    /0000       ; print &
ARG_PT2 K    /0000       ; scan
ARG_PT3 K    /0000
ARG_SN  K    /0000
```

No exemplo anterior, verifica-se que se trata de um caso de comparação do valor de entrada, armazenado em D, com o valor da variável P: se as duas variáveis apresentam o mesmo valor, será impresso no monitor o valor armazenado em ambas, caso contrário, será impresso o que estiver em uma outra variável, O.

Em um segundo momento, considerando o caso das operações aritméticas, sabe-se que elas serão executadas em caso de não ocorrer overflow, no acontecimento da soma, da subtração e da multiplicação, nem divisão por zero. Deste modo, para o teste da divisão, basta utilizar dois exemplos de dados, sendo que em um dos casos, efetua-se uma divisão por zero, garantindo que esteja sendo testada a execução adequada da operação, e o seu possível erro. No caso das demais operações aritméticas, considera-se os casos de serem executadas e de estarem presentes dados para os quais a operação resulta em overflow. Os dois exemplos subsequentes mostram um mesmo código, com a diferença nos valores armazenados nas variáveis, de modo que, para o segundo caso, ocorre overflow. A exemplificação é dada para a multiplicação, mas a análise ocorre de forma análoga para a soma e a divisão.

```
            >   TEXTO   ; Exportar HEADER
            >   LOCAL   ;
            >   GLOBAL  ;
            >   DEBUG   ;
            >   ARG_PT1
            >   ARG_PT2
            >   ARG_PT3
            >   ARG_SN
            &   /0000
TEXTO       K   /000E
LOCAL       K   /0006
GLOBAL      K   /0000
DEBUG       K   /0000
                        ; TEXTO
            SC  MUL
            MM  C       ; C = A * B
            HM  /0000
MUL         K   /0000   ; Sub-rotina MUL
            LD  A
            ML  B       ; AC = A * B
            RS  MUL
                        ; LOCAL
A           K   /0300
B           K   /0010
C           K   /0000
                        ; ARGS
ARG_PT1     K   /0000   ; print &
ARG_PT2     K   /0000   ; scan
ARG_PT3     K   /0000
ARG_SN      K   /0000
```

Modificando os valores armazenados em A e B, de modo a obter overflow:

```
            >   TEXTO   ; Exportar HEADER
            >   LOCAL   ;
            >   GLOBAL  ;
            >   DEBUG   ;
            >   ARG_PT1
            >   ARG_PT2
            >   ARG_PT3
            >   ARG_SN
            &   /0000
TEXTO       K   /000E
LOCAL       K   /0006
GLOBAL      K   /0000
DEBUG       K   /0000
                        ; TEXTO
            SC  MUL
            MM  C       ; C = A * B
            HM  /0000
MUL         K   /0000   ; Sub-rotina MUL
            LD  A
            ML  B       ; AC = A * B
            RS  MUL
                        ; LOCAL
A           K   /0A00
B           K   /0A00
C           K   /0000
                        ; ARGS
ARG_PT1     K   /0000   ; print &
ARG_PT2     K   /0000   ; scan
ARG_PT3     K   /0000
ARG_SN      K   /0000
```

A seguir, analisando os casos de movimentações do acumulador para a memória e vice-versa, o tratamento é idêntico para ambas as instruções (LD e MM), e envolve apenas a análise dos casos *Segmentation Fault*, como explicado anteriormente. Exemplos de instruções envolvendo ambas as operações já foram mostrados anteriormente, de modo que identifica-se a possibilidade de utilizar as duas operações.

Por fim, analisando o caso das operações de chamada e retorno de sub-rotinas (SC e RS), verifica-se também nos exemplos anteriores a sua utilização, nos casos em que a execução está de acordo com o esperado pelo ambiente. Os eventuais erros também são de segmentação.

Como explicitado anteriormente, as operações OS, GD e PD não são suportadas, então as suas eventuais utilizações em um código-objeto resultam na mensagem de erro *NA* (*Not Allowed*). Para o caso da impressão e obtenção de dados, ainda há uma forma de utilizar estas funcionalidades, ou seja, a partir da importação da função desejada.

A seguir é apresentado um exemplo para a utilização do *PRINT*, sendo passados os argumentos ARG_PT1 e ARG_PT3 (números) e o operando desejado, em AGR_PT2.

```
            >   TEXTO   ; Exportar HEADER
            >   LOCAL   ;
            >   GLOBAL  ;
            >   DEBUG   ;
            >   ARG_PT1
            >   ARG_PT2
            >   ARG_PT3
            >   ARG_SN
            <   PRINT
            &   /0000
TEXTO       K   /0004
LOCAL       K   /0006
GLOBAL      K   /0000
DEBUG       K   /0000
                        ; TEXTO
            SC  PRINT   ; print(A + B)
            HM  /0000
```

```
                    ; LOCAL
ARG_PT1 K   /06A9
ARG_PT2 K   /002B
ARG_PT3 K   /3F18
                    ; ARGS scan
ARG_SN  K   /0000
```

Por fim, é apresentado um exemplo de uso da sub-rotina *SCAN*, adiante, na qual o valor lido do teclado é escrito na variável A.

```
        V   TEXTO   ; Exportar HEADER
        V   LOCAL   ;
        V   GLOBAL  ;
        V   DEBUG   ;
        V   ARG_PT1
        V   ARG_PT2
        V   ARG_PT3
        V   ARG_SN
        V   SCAN
        &   /0000
TEXTO   K   /0008
LOCAL   K   /0004
GLOBAL  K   /0000
DEBUG   K   /0000
                    ; TEXTO
        LV  A
        MM  ARG_SN
        SC  SCAN    ; scan(B)
        HM  /0000
                    ; LOCAL
A       K   /0000
ARG_SN  K   /0000
                    ; ARGS print
ARG_PT1 K   /0000
ARG_PT2 K   /0000
ARG_PT3 K   /0000
```

VI. Considerações Finais

A. Perguntas

A partir das seções anteriores, pode-se levantar alguns questionamentos acerca do ambiente desenvolvido para a linguagem C–. Primeiramente, sobre o uso de memória, nota-se que o espaço de memória reservado para o ambiente de execução excede, em muito, o espaço de memória que de fato é utilizado pelo ambiente. Essa foi uma escolha de projeto para facilitar a depuração do código do ambiente, contudo em uma versão final - usada por um usuário - haveria a remoção de todas as pseudo-instruções & (com exceção do & /000), assim otimizando o uso de memória de 908 bytes para 406 bytes - ambos os valores estão em hexadecimal. Além disso, a implementação das diversas funcionalidades do ambiente implica num maior gasto de memória. Dessa forma, essa abordagem gera um alto gasto de memória que poderia ser utilizado em outras regiões (código-objeto ou pilha). Por outro lado, vale ressaltar que o reaproveitamento da área de dados locais aumenta a eficiência do uso da memória da MVN.

Acerca da eficiência, torna-se muito claro que o código desenvolvido é extremamente ineficiente, tendo em vista que o número de instruções próprias do ambiente que são executadas é muito maior que o número de instruções do código-objeto que são executadas. Essa abordagem foi escolhida para garantir todas as funcionalidades presentes no ambiente, como a segurança (*safety*) e a depuração.

A maior dificuldade da implementação do ambiente se trata da lógica necessária para executar indiretamente as instruções de desvio e de chamada/retorno de sub-rotina, pois caso elas fossem executadas diretamente o ambiente perderia o controle da execução, visto que os próximos endereços a serem executados pertencem ao código-objeto e não ao ambiente. Além disso, essa perda de controle pode causar uma falha de segurança no sistema, dado que as verificações de segurança não serão mais feitas e o código do usuário será executado diretamente sem qualquer tratativa.

Ademais, ressalta-se as limitações do código compilado da linguagem C– para o correto funcionamento desse ambiente. O código deve ser particionado nas 5 regiões descritas em II : cabeçalho, texto, dados locais, dados globais e depuração, sendo que se os valores determinados no cabeçalho estiverem incorretos o comportamento do programa é imprevisível. Ademais, se a sintaxe da região de depuração (LD X, onde X é um rótulo) não for respeitada, tal linha X não será depurada. Além disso, espera-se que o compilador do C– consiga importar as funções *PRINT* e *SCAN* do próprio ambiente de execução, dado que há diversas minúcias a serem consideradas nestas funções, como transformar ASCII para int. Dessa forma, acredita-se que o compilador não pode gerar um código-objeto contendo instruções com operações PD e GD, visto que a utilização de tais operações no código-objeto não teria o efeito adequado, sendo necessário um tratamento adequado da entrada e saída de dados, o que é feito pelo ambiente. Também é pressuposto que a última instrução executada do código-objeto seja um HM, caso o ambiente não execute um HM ao detectar um erro, e que não sejam utilizadas funções recursivas ou aninhadas.

Veja que, além das limitações do compilador do C–, há diversos problemas causados pela limitação da arquitetura da MVN. Por exemplo, o *hardware* não é capaz de gerar flag de *overflow* ou suportar instruções com imediato. Desse modo, há casos - como tratamento de instruções aritméticas e determinação do opcode em *TRATOP* - em que são necessários várias ciclos de operação do processador para realizar atividades que em outras arquiteturas, como ARM [1], são realizadas em apenas 1 ciclo.

B. Entrega

Além deste relatório, foram entregues diversos arquivos .asm e .cmm. O ambiente.asm representa todo o ambiente de execução debatido nesse artigo, enquanto os demais .asm e .cmm são arquivos de teste que estão subdivididos em pastas, de acordo com o tipo de teste realizado. Note que os .cmm são arquivos escritos em C–, cuja tradução em *assembly* está contido no arquivo de mesmo nome, mas no formato .asm.

Ademais, note que nem todos os arquivos .asm tem uma versão em .cmm, pois alguns .asm não foram criados com base em um arquivo .cmm específico, e sim, com o objetivo

de testar alguma funcionalidade do sistema ou de testar casos extremos para encontrar erros na implementação. Vale ressaltar que a pasta *codigos_incompativeis* contém arquivos em linguagem de montagem que são incompatíveis com o ambiente, e que são utilizados para observar o funcionamento do sistema nesses casos.

Para gerar o ambiente.mvn com um determinado programa de teste *teste.asm* utilize os seguintes comandos no mvn-cli:

```
mvn-cli assemble -i ambiente.asm >
↪  ambiente.int
mvn-cli assemble -i teste.asm > teste.int
mvn-cli link -i ambiente.int -i teste.int
↪  --complete > ambiente.lig
mvn-cli relocate -i ambiente.lig --base 0
↪  > ambiente.mvn
```

Após iniciar a MVN, escolha como endereço inicial de execução (do comando r) como 0208, com isso o ambiente e o código-objeto do programa de teste serão executados corretamente.

VII. Conclusão

Neste trabalho, foi desenvolvido um ambiente de execução que abstraísse os detalhes relativos a execução do código-objeto para o usuário. Desse modo, permitindo que o programador possa se concentrar no desenvolvimento de sua aplicação, ao invés de se atentar a questões de alocação de memória, interfaceamento com dispositivos externos e tratamento de erros.

Para isso, o ambiente criado é capaz de executar o código-objeto linha a linha, analisando a ocorrência de possíveis erros de execução. Além disso, ele suporta operações com pilha - para instruções SC e RS - e disponibiliza funções para interfaceamento com dispositivos externos e contém um modo de depuração.

Contudo, todas essas funcionalidades levam a um código extremamente ineficiente, dado que para executar uma instrução do usuário é necessário que o ambiente realize diversas análises e tratativas.

Referências

[1] Patterson, David A., and John L. Hennessy. Computer organization and design ARM edition: the hardware software interface. Morgan kaufmann, 2016.

[2] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, techniques, and tools. Vol. 2. Reading: Addison-wesley, 2007.