VI. Geração de Código

Até a fase de análise semântica, o processo de compilação está orientado pela *linguagem-fonte*. Na frase de geração de código, precisamos considerar *a linguagem-alvo* do compilador. Quando o compilador gera código-objeto, a linguagem-alvo é o *Assembly* de uma máquina específica. Desta forma, o processo de geração de código implica em converter os comandos da linguagem-fonte em uma série de comandos Assembly.

1. A Máquina Alvo

Como o gerador de código é especifico para uma máquina, é importante conhecermos sua organização bem como a arquitetura do seu conjunto de instruções.

1.1. A Família 80x86/x64

A máquina para a qual escrevemos código-objeto, LCX, é um subconjunto da arquitetura x64 (extensão do 80x86 para tratamento de palavras de 64 bits). O programa-alvo em Assembly será montado e linkeditado com o NASM (Netwide Assembler). O código poderá ser executado em ambiente Linux ou MacOS, em processadores compatíveis com o Intel.

1.2. Registradores endereçáveis

O LCX possui 4 registradores de 64 bits, podendo cada um ser alternativamente usado como registradores de 32, 16 ou 8 bits:

64	32	16	8
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL

Possui ainda 2 registradores de 64 bits: RDI e RSI que podem ser usados como apontadores e 4 registradores de 128 bits para armazenar números em formato de ponto flutuante IEEE 754: XMM0, XMM1, XMM2 e XMM3. Utilizaremos apenas a parte baixa desses registradores (ponto flutuante de 32 bits)

1.3. Organização de Memória

A memória principal endereçável do LCX é de 1 Mbyte, endereçada por bytes, onde ficam armazenados o código, os dados e a pilha. Os primeiros 64 Kbytes são reservados para temporários (detalhes mais adiante). A posição inicial do espaço de endereçamento é determinada pelo sistema operacional, a partir do qual temos um valor de deslocamento que se inicia por 0. A pilha ocupa a parte final do espaço:

Endereço	Conteúdo
0	Temporários
64K	Dados

. . .

1M-1	Pilha

Um endereço de memória é um número sem sinal de 64 bits.

1.4. Tipos de Dados

a) **Caractere: ocupa uma posição de memória, variando de 0 a 255.**
b) **Inteiro: ocupa 4 bytes, variando de -2^{31} a $2^{31}-1$.**
c) **Real: ocupa 4 bytes, com precisão de 6 casas decimais compartilhadas entre a parte inteira e a fracionária.**
d) **Lógico: inteiro de 32 bits que assume os valores 0 (falso) ou 1 (verdadeiro).**
e) **Vetor de inteiros: ocupa 4n bytes, onde n é o número de elementos do vetor, com índice inicial 0.**
f) **Vetor de caracteres: ocupa n bytes, onde n é o número de elementos do vetor, com índice inicial 0.**
g) **Vetor de reais: ocupa 4n bytes, onde n é o número de elementos do vetor, com índice inicial 0.**
h) **String: é um vetor de caracteres com no máximo 255 posições úteis e finalizado pelo caractere de valor 0h.**
i) **Apontador: inteiro sem sinal de 64 bits que armazena um endereço de memória relativo ao início do espaço de armazenamento.**

1.5. Modos de Endereçamento

a) Registrador: Indica o nome do registrador onde o dado será buscado ou armazenado.

Ex: `mov eax, ebx`

Indica que o conteúdo do registrador ebx será copiado para o registrador eax.

b) Imediato: Permite que a origem seja uma constante.

Ex: `mov eax, 10`

Indica que o valor 10 será copiado para o registrador eax.

Formatos de imediatos possíveis:

- **Inteiro em base decimal. Ex: 1234**
- **Inteiro em base hexadecimal. Ex: 0xC0DE ou C0DEh (letras minúsculas ou maiúsculas)**
- **Caractere. Ex: 'A'**
- **Real. Ex: -12.34**

c) Endereçamento Direto (Deslocamento): Indica a posição de memória na qual será buscado ou armazenado um dado. Utilizaremos este modo para endereçar posições dentro do espaço de endereçamento "M" solicitado ao sistema operacional.

Ex: `mov eax, [qword M+1]`

Copia o inteiro que está na segunda posição do espaço de endereçamento M para o registrador eax.

d) Endereçamento Indexado: Semelhante ao endereçamento direto, mas neste caso o endereço é dado pelo conteúdo de um registrador.

Ex: `mov al, [rdi]`

Copia o byte, cujo endereço é o valor armazenado em rdi, para o registrador al. O registrador de índice deve endereçar apenas dentro do espaço válido, portanto seu valor deve ser M + deslocamento válido.

1.6. O Conjunto de Instruções

O Assembly da máquina LCX possui as seguintes instruções:

Inst.	Formatos	Descrição	Exemplo

add	add rD,rO
	add rD,imed
		rD←rD+rO
		rD←rD+imed
			add eax,ebx
			add eax,1
			add al,bl

addss	addss rD,rO	rD←rD+rO (float)	addss xmm0,xmm1
cvtsi2ss	cvtsi2ss rD,rO	rD←(float)rO (int 64 bits)	cvtsi2ss xmm0,rax
cvtss2si	cvtss2si rD,rO	rD←(int 64 bits)rO	cvtss2si rax,xmm0
cdq	cdq	edx:eax←(64 bits)eax	cdq
cdqe	cdqe	rax←(64 bits)eax	cdqe
cmp	cmp r1,r2 cmp r1,imed	comp. r1 e r2(32) comp. r1 e imed.	cmp eax,ebx cmp eax,0
comiss	comiss r1,r2	comp. r1 e r2(float)	comiss xmm0,xmm1
divss	divss rD,rO	rD←rD/rO (float)	divss xmm0,xmm1
idiv	idiv reg	edx:eax←edx:eax/reg(32) eax (quoc), edx(resto)	idiv ebx
imul	imul reg	edx:eax←eax*reg (32)	imul ebx
int	int imed	Executa interrupção	int 21h
ja	ja dest	PC←dest se r1>r2 (float)	ja R1
jae	jae dest	PC←dest se r1≥r2 (float)	jae R1
jb	jb dest	PC←dest se r1<r2 (float)	jb R1
jbe	jbe dest	PC←dest se r1≤r2 (float)	jbe R1
jg	jg dest	PC←dest se r1>r2 (int)	jg R1
jge	jge dest	PC←dest se r1≥r2 (int)	jge R1
jl	jl dest	PC←dest se r1<r2 (int)	jl R1
jle	jle dest	PC←dest se r1≤r2 (int)	jle R1
je	je dest	PC←dest se r1=r2 (int ou float)	je R1
jmp	jmp dest	PC←dest	jmp R1
jne	jne dest	PC←dest se r1≠r2 (int ou float)	jne R1

mov	mov rD,rO
	mov reg,imed
	mov reg,[qword M+d]
	mov [qword M+d],reg
	mov r1,[r2]
	mov [r1],r2
		rD←rO (int)
		reg←imed (int)
		reg←[M+d] (int)
		[M+d] ←reg (int)
		reg1←[reg2] (int)
		[reg1] ←reg2 (int)
			mov eax,ebx
			mov eax,0
			mov eax,[qword M+1]
			mov [qword M+1],eax
			mov eax,[rdi]
			mov [rdi],eax

movss	movss rD,rO
	movss reg,[qword M+d]
	movss [qword M+d],reg
	movss r1,[r2]
	movss [r1],r2
		rD←rO (float)
		reg←[M+d] (float)
		[M+d] ←reg (float)
		reg1←[reg2] (float)
		[reg1] ←reg2 (float)
			movss xmm0,xmm1
			movss xmm0,[qword M+1]
			movss [qword M+1],xmm0
			movss xmm0,[rdi]
			movss [rdi],xmm0

mulss	mulss rD,rO	rD←rD*rO (float)	mulss xmm0,xmm1
neg	neg reg	reg← - reg (int)	neg eax
pop	pop reg	reg←pilha (int 15 ou 64)	pop dx
push	push reg	pilha←reg (int 16 ou 64)	push dx
roundss	roundss rD,rO,flag	rD← trunc(rO)	roundss xmm0,xmm1, 0b0011
sub	sub rD,rO sub rD,imed	rD←rD-rO rD←rD-imed	sub eax,ebx sub eax,1
subss	subss rD,rO	rD←rD-rO (float)	subss xmm0,xmm1
syscall	syscall	Executa interrupção	syscall

2. Netwide Assembler (NASM)

O NASM é um ambiente de montagem e linkedição de programas escritos em Assembly para a família x64 e compatíveis.

Comandos em um programa assembly têm a forma básica:

Rótulo: Mnemônico operandos ;comentário

Ex: R1: mov eax,0 ; zera registrador eax

Além dos comandos, o programa tem diretivas para estruturação das seções de dados e código. Os templates que utilizaremos para a geração de código são dados a seguir:

Linux:

```
section .data       ; Sessão de dados
M:                  ; Rótulo para demarcar o
                   ; início da sessão de dados
resb 10000h         ; Reserva de temporários

    ; ***Definições de variáveis e constantes

section .text       ; Sessão de código
global _start  ; Ponto inicial do programa

_start:             ; Início do programa
    ; ***Comandos

; Halt
mov rax, 60  ; Chamada de saída
mov rdi, 0   ; Código de saida sem erros
syscall  ; Chama o kernel
```

MacOS:

```
global start  ; Ponto inicial do programa
section .data       ; Sessão de dados
M:                  ; Rótulo para demarcar o
                   ; início da sessão de dados
resb 10000h         ; Reserva de temporários

    ; ***Definições de variáveis e constantes

section .text       ; Sessão de código
start:             ; Início do programa
    ; ***Comandos

; Halt
mov rax, 0x2000001 ; Chamada de saída
mov rdi, 0   ; Código de saida sem erros
syscall  ; Chama o kernel
```

Vários blocos de uma mesma seção podem aparecer durante o programa; eles serão concatenados pelo montador.

3. Geração de Código para Declarações

Variáveis

Deve-se reservar uma área da memória de dados para cada variável, a partir da posição 10000h. O tamanho desta área é determinado pelo tipo de variável. O mnemônico do comando indica quantos bytes são reservados por unidade solicitada: resb (1), resd (4), resq (8). Ex:

```
Var
Caractere letra;
String nome;
Inteiro matricula;
Real media;
Apontador inteiro prox;
Vetor inteiro notas[10];
```

Código gerado:

```
section .data
M:
    resb 10000h             ;temporários
    resb 1                  ;Car. em 10000h
    resb 100h               ;String em 10001h
    resd 1                  ;Int em 10101h
    resd 1                  ;Float em 10105h
    resq 1                  ;Apontador em 10109h
    resd 10                 ;Vet 10 int em 10111h
```

O endereço da área deve ser guardado na tabela de símbolos para futuras referências. A próxima posição de memória disponível deve ser atualizada para as próximas declarações. Um contador de dados (variável global do compilador) deve manter o endereço da próxima posição disponível, sendo incrementado a cada reserva.

Constantes

Além da reserva de memória e do registro do endereço inicial, é preciso armazenar o valor da constante na memória de dados a ela reservada. O tipo da constante deve ser determinado pelo analisador léxico. O mnemônico do comando indica quantos bytes são reservados por unidade solicitada: db (1), dd (4). Para strings, deve ser acrescentado o byte de fim de string. Ex:

```
Const A='a';
Const B=-1;
Const C="Compiladores";
Const D=1.5;
```

Código Gerado:

```
section .data
M:
    resb 10000h              ;temporários
    db 'a' 1                 ;Car. em 10000h
    dd -1                   ;Inteiro em 10001h
    db "Compiladores",0      ;String em 10005h
    dd 1.5                   ;Float em 10012h
```

Tempo de compilação

Lex	*Token*	*Classe*	*Tipo*	*Tamanho*	*End*
"A"	ID	const	caractere	1	10000h
"B"	ID	const	inteiro	4	10001h
"C"	ID	const	string	13	10005h
"D"	ID	const	real	4	10012h

4. Geração de Código para Expressões

Temporários

Resultados parciais da avaliação de expressões são colocados na área de dados temporários. A primeira posição disponível é M+0. Deve ser declarada uma função NovoTemp que retorna o endereço da próxima posição disponível e incrementa o contador de temporários do tamanho da memória reservada. Sempre que um comando chama uma expressão, o próximo temporário disponível pode retornar à posição inicial da área.

O tamanho de um temporário é determinado pelo tipo da sub-expressão cujo valor será armazenado nele.

Fatores

F → const *{ se const é string ou real então*
declarar constante na área de dados. Ex:

```
section .data
    db const.lex,0
    dd 1.5
section .text
```

F.end := contador dados
Atualizar contador dados
F.tipo e F.tam vêm do R.Lex
senão
F.end:=NovoTemp
mov reg, imed
mov [M+F.end], reg }

F → "("Exp")" *{ F.end := Exp.end }*

F → não Fι *{ F.end := NovoTemp }*
{ mov reg, [qword M+Fι.end] }
{ neg reg }
{ add reg,1 }
{ mov [qword M+F.end], reg}

F → id ***{ F.end := id.end; F.tipo := id.tipo; F.tam := id.tam }***

Acesso a elementos de um vetor

F → id "[" Exp "]"

- **Gere um novo temporário para F.end**
- **Carregue o conteúdo que está no endereço Exp.end para um registrador de 64 bits, reg1**
- **Multiplique o valor em reg1 pelo número de bytes que o tipo do vetor ocupa, se for maior que 8 bits.**
- **Some id.end a reg1;**
- **Reg1 agora tem o endereço do elemento. Carregue o conteúdo da posição [reg1] para reg2**
- **Transfira o conteúdo de reg2 para o novo temporário.**

Termos

T → F_1 ① { (* | / | % | E ②) F_2 ③ }*

① ***{ T.end := F_1.end } { T.tipo := F_1.tipo }***
② ***{ guardar o tipo do operador (*, /, % ou E) }***
③ ***{ carregar o conteúdo de T.end no reg1 (`mov,movss`) }***
{ carregar o conteúdo de F_2.end no reg2}
{ converter tipos e expandir sinal se necessário (cdq,cvtsi2ss,cdqe)}
{ conforme operador e tipo, gerar instrução (`imul,idiv,mulss,divss`) entre reg1 e reg2) }
{ T.end := NovoTemp}
{ guardar resultado em T.end (`mov,movss`) }

Expressões simples

ExpS → [- ①] T_1 ② { (+ | - | OU ③) T_2 ④ }*

① *{ verificar a necessidade de negação de T_1 }*
② *{ negar o valor de T_1 se for o caso (NovoTemp, `mov, neg, mov ou movss,mulss,movss...`) }*
{ ExpS.end := T_1.end }
{ ExpS.tipo := T_1.tipo }
③ *{ guardar operadores }*
④ **regras semelhantes ao ③ de Termos. Verifique a simulação do operador OU com instruções aritméticas.**

Rótulos

Deve ser declarada uma função NovoRot que retorna o valor do próximo rótulo, começando por Rot1, e incrementando o contador de rótulos que é uma variável global. Os rótulos serão usados nas instruções de salto.

Expressão

Exp → $ExpS_1$ ① [R ② $ExpS_2$ ③]

① *{ Exp.end := $ExpS_1$.end }*
{ Exp.tipo := $ExpS_1$.tipo }
② *{ guardar relacional }*
③ *{ carregar conteúdo de Exp.end em reg1 e $ExpS_2$.end em reg2 (converter tipos se necessário) }*
{ comparar reg1 e reg2: `cmp,comiss` }
{ RotVerdadeiro:=NovoRot }
{ gerar instrução Jxx RotVerdadeiro, onde Jxx será `je` (=), `jne` (<>), `jl/jb` (<), `jg/ja` (>), `jge/jae` (>=), `jle/jbe` (<=) }
{ `mov eax, 0` }
{ RotFim := NovoRot }
{ `jmp` RotFim }
{ RotVerdadeiro: }
{ `mov eax, 1` }
{ RotFim: }
{ Exp.end:=NovoTemp }
{ Exp.tipo:=TIPOLÓGICO }
{ `mov [qword M+`Exp.end`], eax` }

Para strings, comparações têm que ser feitas com loops em ASSEMBLY !

5. Geração de Código para Comandos

Atribuição

C → id = Exp; ***{ carregar conteúdo de Exp.end }***
{ armazenar o resultado em id.end }

Obs: strings devem ser movidas caractere a caractere

Repetição

C → Repita para id=Exp
{ Código para atribuição }
{ RotInicio:=NovoRot }
{ RotFim:=NovoRot }
{ RotInicio: }
até Exp ***{ carregar conteúdo de Exp.end }***
{comparar com o conteúdo de id, se id>Exp desvia para RotFim }
[passo const]
Comando
{ incrementa id de const ou 1 por default; desvia para RotInicio }
{ RotFim: }

C → Repita enquanto
{ RotInicio:=NovoRot }
{ RotFim:=NovoRot }
{ RotInicio: }
Exp ***{ carregar conteúdo de Exp.end }***
{ se exp é falsa, desvia para RotFim }

Comando
{ desvia para RotInicio }
{ RotFim: }

Teste

C → Se
{ RotFalso:=NovoRot }
{ RotFim:=NovoRot }
Exp *{ carregar conteúdo de Exp.end }*
{ se exp é falsa, desvia para RotFalso }
Comando;
Senão
{ desvia para RotFim }
{ RotFalso: }
Comando;
{ RotFim: }

C → Se
{ RotFalso:=NovoRot }
Exp *{ carregar conteúdo de Exp.end }*
{ se exp é falsa, desvia para RotFalso }
Comando;
{ RotFalso: }

Entrada

A entrada do teclado é sempre do tipo string. Deve-se criar um buffer para a entrada (temporário de 256 bytes) e utilizar a instrução de interrupção da forma:

Linux:

```
mov     rsi, M+buffer.end
mov     rdx, 100h  ;tamanho do buffer
mov     rax, 0 ;chamada para leitura
mov     rdi, 0 ;leitura do teclado
syscall
```

MacOS:

```
mov     rsi, M+buffer.end
mov     rdx, 100h  ;tamanho do buffer
mov     rax, 0x2000003   ; Chamada de leitura
mov     rdi, 1 ; Teclado
syscall
```

Após a interrupção, RAX conterá o número de caracteres lidos, incluindo a quebra de linha. Para o tipo string, devemos transferi-lo para a variável, trocando a quebra de linha (0Ah) por 0 (fim de string). Caso o dado seja do tipo inteiro, antes de ser armazenado, deve ser convertido. Rot0 a Rot3 devem ser substituídos por novos rótulos. O resultado estará em eax:

Linux/MacOS:

```
mov eax, 0                  ;acumulador
mov ebx, 0                  ;caractere
mov ecx, 10                 ;base 10
mov dx, 1                   ;sinal
mov rsi, M+buffer.end       ;end. buffer
mov bl, [rsi]               ;carrega caractere
cmp bl, '-'                 ;sinal - ?
jne Rot0                    ;se dif -, salta
mov dx, -1                  ;senão, armazena -
add rsi, 1                  ;inc. ponteiro string
mov bl, [rsi]               ;carrega caractere

Rot0:
push dx                     ;empilha sinal
mov  edx, 0                 ;reg. multiplicação

Rot1:
cmp  bl, 0Ah                ;verifica fim string
je Rot2                     ;salta se fim string
```

```
imul ecx                    ;mult. eax por 10
sub bl, '0'                 ;converte caractere
add eax, ebx                ;soma valor caractere
add rsi, 1                  ;incrementa base
mov bl, [rsi]               ;carrega caractere
jmp Rot1                    ;loop

Rot2:
pop cx                      ;desempilha sinal
cmp cx, 0
jg Rot3
neg eax                     ;mult. sinal

Rot3:
```

Caso o dado seja do tipo real, antes de ser armazenado na variável correspondente, deve ser convertido. Rot0-Rot3 devem ser substituídos por novos rótulos. O resultado estará em xmm0:

Linux/MacOS:

```
mov rax, 0                  ;acumul. parte int.
subss xmm0,xmm0             ;acumul. parte frac.
mov rbx, 0                  ;caractere
mov rcx, 10                 ;base 10
cvtsi2ss xmm3,rcx           ;base 10
movss xmm2,xmm3             ;potência de 10
mov rdx, 1                  ;sinal
mov rsi, M+buffer.end       ;end. buffer
mov bl, [rsi]               ;carrega caractere
cmp bl, '-'                 ;sinal - ?
jne Rot0                    ;se dif -, salta
mov rdx, -1                 ;senão, armazena -
add rsi, 1                  ;inc. ponteiro string
mov bl, [rsi]               ;carrega caractere

Rot0:
```

```
push rdx                    ;empilha sinal
mov  rdx, 0                 ;reg. multiplicação

Rot1:
cmp  bl, 0Ah                ;verifica fim string
je Rot2                     ;salta se fim string
cmp  bl, '.'                ;senão verifica ponto
je Rot3                     ;salta se ponto
imul ecx                    ;mult. eax por 10
sub bl, '0'                 ;converte caractere
add eax, ebx                ;soma valor caractere
add rsi, 1                  ;incrementa base
mov bl, [rsi]               ;carrega caractere
jmp Rot1                    ;loop

Rot3:
       ;calcula parte fracionária em xmm0

add rsi, 1                  ;inc. ponteiro string
mov bl, [rsi]               ;carrega caractere
cmp  bl, 0Ah                ;*verifica fim string
je Rot2                     ;salta se fim string
sub bl, '0'                 ;converte caractere
cvtsi2ss xmm1,rbx           ;conv real
divss xmm1,xmm2             ;transf. casa decimal
addss xmm0,xmm1             ;soma acumul.
mulss xmm2,xmm3             ;atualiza potência
jmp Rot3                    ;loop

Rot2:
cvtsi2ss xmm1,rax           ;conv parte inteira
addss xmm0,xmm1             ;soma parte frac.
pop rcx                     ;desempilha sinal
cvtsi2ss xmm1,rcx           ;conv sinal
mulss xmm0,xmm1             ;mult. sinal
```

Saída

A saída para o vídeo é sempre do tipo string.

Linux:

```
mov     rsi, M+string.end    ;ou buffer.end
mov     rdx, string.tam      ;ou buffer.tam
mov     rax, 1 ;chamada para saída
mov     rdi, 1 ;saída para tela
syscall
```

MacOS:

```
mov     rsi, M+string.end    ;ou buffer.end
mov     rdx, string.tam      ;ou buffer.tam
mov     rax, 0x2000004    ;chamada para saída
mov     rdi, 1            ;saída para tela
syscall
```

Onde o tamanho do string não conta com o fim do string.

Se o dado a ser impresso for inteiro, deverá ser convertido e armazenado em um buffer temporário, com endereço buffer.end. O código de conversão de inteiros é dado a seguir. Rótulos devem ser substituídos por novos rótulos:

Linux/MacOS:

```
mov eax, [qword M+Exp.end];inteiro a ser
    ;convertido
mov rsi, M+buffer.end  ;end. string ou temp.
mov rcx, 0             ;contador pilha
mov rdi, 0             ;tam. string convertido
cmp eax, 0             ;verifica sinal
jge Rot0               ;salta se número positivo
mov bl, '-'            ;senão, escreve sinal -
mov [rsi], bl
add rsi, 1             ;incrementa índice
add rdi, 1             ;incrementa tamanho
```

```
neg eax                 ;toma módulo do número

Rot0:
mov ebx, 10             ;divisor

Rot1:
add rcx, 1              ;incrementa contador
cdq                     ;estende edx:eax p/ div.
idiv ebx                ;divide edx;eax por ebx
push dx                 ;empilha valor do resto
cmp eax, 0              ;verifica se quoc. é 0
jne Rot1                ;se não é 0, continua

add rdi,rcx             ;atualiza tam. string

;agora, desemp. os valores e escreve o string

Rot2:
pop dx                  ;desempilha valor
add dl, '0'             ;transforma em caractere
mov [rsi], dl           ;escreve caractere
add rsi, 1              ;incrementa base
sub rcx, 1              ;decrementa contador
cmp rcx, 0              ;verifica pilha vazia
jne Rot2                ;se não pilha vazia, loop

;executa interrupção de saída
```

Se o dado a ser impresso for real, deverá ser convertido e armazenado em um buffer temporário, com endereço buffer.end. O código de conversão de reais é dado a seguir. Rótulos devem ser substituídos por novos rótulos:

Linux/MacOS:

```
mov xmm0, [qword M+Exp.end]   ;real a ser
    ;convertido
mov rsi, M+buffer.end  ;end. temporário
mov rcx, 0             ;contador pilha
mov rdi, 6             ;precisao 6 casas compart
mov rbx, 10            ;divisor
cvtsi2ss xmm2, rbx     ;divisor real
subss xmm1, xmm1       ;zera registrador
comiss xmm0, xmm1      ;verifica sinal
jae Rot0               ;salta se número positivo
mov dl, '-'            ;senão, escreve sinal -
mov [rsi], dl
mov rdx, -1            ;Carrega -1 em RDX
cvtsi2ss xmm1, rdx     ;Converte para real
mulss xmm0, xmm1       ;Toma módulo
add rsi, 1             ;incrementa índice

Rot0:
roundss xmm1, xmm0, 0b0011 ;parte inteira xmm1
subss xmm0, xmm1           ;parte frac xmm0
cvtss2si rax, xmm1         ;convertido para int

;converte parte inteira que está em rax

Rot1:
add rcx, 1             ;incrementa contador
cdq                    ;estende edx:eax p/ div.
idiv ebx               ;divide edx;eax por ebx
push dx                ;empilha valor do resto
cmp eax, 0             ;verifica se quoc. é 0
jne Rot1               ;se não é 0, continua
```

```
sub rdi, rcx            ;decrementa precisao

;agora, desemp valores e escreve parte int

Rot2:
pop dx                  ;desempilha valor
add dl, '0'             ;transforma em caractere
mov [rsi], dl           ;escreve caractere
add rsi, 1              ;incrementa base
sub rcx, 1              ;decrementa contador
cmp rcx, 0              ;verifica pilha vazia
jne Rot2                ;se não pilha vazia, loop

mov dl, '.'             ;escreve ponto decimal
mov [rsi], dl
add rsi, 1              ;incrementa base

;converte parte fracionaria que está em xmm0

Rot3:
cmp rdi, 0              ;verifica precisao
jle Rot4                ;terminou precisao ?
mulss xmm0,xmm2         ;desloca para esquerda
roundss xmm1,xmm0,0b0011 ;parte inteira xmm1
subss xmm0,xmm1         ;atualiza xmm0
cvtss2si rdx, xmm1      ;convertido para int
add dl, '0'             ;transforma em caractere
mov [rsi], dl           ;escreve caractere
add rsi, 1              ;incrementa base
sub rdi, 1              ;decrementa precisao
jmp Rot3

; impressão

Rot4:
mov dl, 0               ;fim string, opcional
mov [rsi], dl           ;escreve caractere
```

```
mov rdx, rsi            ;calc tam str convertido
mov rbx, M+buffer.end
sub rdx, rbx            ;tam=rsi-M-buffer.end
mov rsi, M+buffer.end   ; endereço do buffer

;executa interrupção de saída. rsi e rdx já
foram calculados então usar só as instruções
para a chamada do kernel.
```

Para o comando *writeln*, anexar a quebra de linha ao final do string, antes de imprimi-lo.