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

A máquina para a qual escrevemos código-objeto, LCX, é um subconjunto da arquitetura da família 80x86, com instruções que tratam dados de 8 e 16 bits. O programa-alvo em Assembly será montado e linkeditado com o MASM (Microsoft Assembler). O código poderá ser executado em ambiente MSDOS, em máquinas compatíveis com o IBM PC.

1.2. Registradores

O LCX possui 4 registradores de 16 bits, sendo cada um composto de 2 registradores de 8 bits justapostos e endereçáveis:

AX	**AH**	**AL**
BX	**BH**	**BL**
CX	**CH**	**CL**
DX	**DH**	**DL**

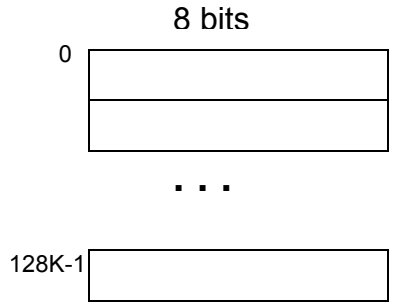
Possui ainda 6 registradores de 16 bits: *DS* armazena o endereço do segmento de dados, *CS* o endereço do segmento de código e *SS* o segmento que contém a pilha. O registrador *SP* contém o endereço do topo da pilha (deslocamento). Os registradores *SI* e *DI* são usados como índices.

DS	**CS**	**SS**	**SP**	**SI**	**DI**

Além disso, possui um registrador específico, *PC (ou IP)*, que armazena o endereço da próxima instrução a ser executada.

1.3. Organização de Memória

A memória principal endereçável do LCX é de 128 Kbytes, endereçada por bytes e dividida em 3 segmentos: código (64K), pilha (16K) e dados (16K para temporários, 32K para variáveis e constantes).



Um endereço de memória é da forma *S:D*, onde *S* é um registrador de segmento e *D* o deslocamento dentro deste segmento.

1.4. Tipos de Dados

a) **Caractere: ocupa uma posição de memória, variando de 0 a 255.**
b) **Inteiro: ocupa 2 bytes, variando de –32768 a 32767.**
c) **Lógico: inteiro que assume os valores 0 (falso) ou 1 (verdadeiro).**
d) **Vetor de inteiros: ocupa 2n bytes, onde n é o número de elementos do vetor, com índice inicial 0.**
e) **Vetor de caracteres: ocupa n bytes, onde n é o número de elementos do vetor, com índice inicial 0.**
f) **String: é um vetor de caracteres que contém o caractere '$' (24h) em alguma posição, indicando o fim do string.**

1.5. Modos de Endereçamento

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

Ex: `mov ax, bx`

Indica que o conteúdo do registrador bx será copiado para o registrador ax.

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

Ex: `mov ax, 10`

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

c) Endereçamento Direto (Deslocamento): Indica a posição de memória na qual será buscado ou armazenado um dado. Utilizaremos este modo para endereçar posições dentro de um determinado segmento, cujo nome é indicado.

Ex: `mov al, DS:[1h]`

Copia o byte que está na segunda posição (1h) do segmento DS, o qual deve ter sido previamente definido.

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

Ex: `mov al, DS:[cx]`

Copia o byte de endereço DS:CX para AL.

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 ax,bx add ax,1
cwd	Cwd	Dx:ax←(32 bits)ax	
cmp	cmp r1,r2 cmp r1,imed	comp. r1 e r2(16) comp. r1 e imed.	cmp ax,bx cmp ax,0
idiv	idiv reg	ax:dx←dx:ax/reg(16) ax (quoc), dx(resto)	idiv bx
imul	imul reg	Dx:ax←ax*reg (16)	imul bx
int	int imed	Executa interrupção	int 21h
jg	jg dest	PC←dest se r1>r2	jg R1
jge	jge dest	PC←dest se r1≥r2	jge R1
jl	jl dest	PC←dest se r1<r2	jl R1
jle	jle dest	PC←dest se r1≤r2	jle R1
je	je dest	PC←dest se r1=r2	je R1
jmp	jmp dest	PC←dest	jmp R1
jne	jne dest	PC←dest se r1≠r2	jne R1

mov	mov rD,rO
	mov reg,imed
	mov reg,S:[end]
	mov S:[end],reg
		rD←rO
		reg←imed
		reg←M[S:end]
		M[S:end] ←reg
			mov ax,bx
			mov ax,0
			mov ax,DS:[1]
			mov DS:[cx],ax

neg	neg reg	reg← - reg	neg ax
pop	pop reg	reg←pilha	pop dx
push	push reg	pilha←reg	push dx
sub	sub rD,rO	rD←rD-rO	sub ax,bx

2. Microsoft Assembler (MASM)

O MASM é um ambiente de montagem e linkedição de programas escritos em Assembly para a família 80x86. Para usar o aplicativo, recomenda-se a leitura do capítulo 8 do livro virtual "The Art of Assembly Programming" de Randall Hyde.

Comandos em um programa assembly têm a forma básica:

Rótulo: Mnemônico operandos ;comentário

Ex: R1: mov ax,0 ; zera acumulador

Além dos comandos, o programa tem diretivas e estruturação de segmentos. Exemplo:

```
sseg SEGMENT STACK          ;início seg. pilha
     byte 4000h DUP(?)      ;dimensiona pilha
sseg ENDS                   ;fim seg. pilha

dseg SEGMENT PUBLIC         ;início seg. dados
     byte 4000h DUP(?)      ;temporários
     ;definições de variáveis e constantes
dseg ENDS                   ;fim seg. dados

cseg SEGMENT PUBLIC         ;início seg. código
     ASSUME CS:cseg, DS:dseg

strt:                       ;início do programa
     ;comandos
cseg ENDS                   ;fim seg. código
END strt                    ;fim programa
```

Vários blocos de um mesmo segmento podem aparecer durante o programa; eles serão concatenados pelo montador. O registrador de segmento de dados deve ser carregado com seu endereço no início do programa:

```
strt:                          ;início do programa
    mov ax, dseg
    mov ds, ax
```

3. Geração de Código para Declarações

Variáveis

Deve-se reservar uma área da memória de dados para cada variável, a partir da posição 4000h. O tamanho desta área é determinado pelo tipo de variável:

```
dseg SEGMENT PUBLIC            ;início seg. dados
    byte 4000h DUP(?)          ;temporários
    byte ?                     ;var. Caract. em 4000h
    byte 100h DUP(?)           ;var. string em 4001h
    sword ?                    ;var. int em 4101h
    sword 10 DUP(?)            ;var. Vet int em 4103h
dseg ENDS                      ;fim seg. dados
```

O endereço da posição inicial 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 declaração.

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. Ex:

```
Var
inteiro    A,B;
caractere  C[256];
caractere  D;
const      E=1;
```

Código Gerado:

```
dseg SEGMENT PUBLIC          ;início seg. dados
    byte 4000h DUP(?)        ;temporários
    sword ?                  ;var. int em 4000h
    sword ?                  ;var. int em 4002h
    byte 256 DUP(?)          ;var. string em 4004h
    byte ?                   ;var. caract em 4104h
    sword 1                  ;const. int. em 4105h
dseg ENDS                    ;fim seg. dados
```

Tempo de compilação

Lex	*Token*	*Classe*	*Tipo*	*Tamanho*	*End*
"A"	ID	var	inteiro	0	4000h
"B"	ID	var	inteiro	0	4002h
"C"	ID	var	caractere	256	4004h
"D"	ID	var	caractere	0	4104h
"E"	ID	const	inteiro	0	4105h

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 DS: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 então*
declarar constante na área de dados:
dseg SEGMENT PUBLIC
byte "const.lex$"
dseg ENDS
F.end := contador dados
Atualizar contador dados
F.tipo e F.tam vêm do R.Lex
senão
F.end:=NovoTemp
mov regA, imed
mov F.end, regA }

F → "(" Exp ")" *{ F.end := Exp.end }*

F → not Fι *{ F.end := NovoTemp }*
{ mov regA, Fι.end }
{ neg regA }
{ add regA,1 }
{ mov F.end, regA}

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 o reg AX**
- **Se o vetor for de inteiros, some o conteúdo de Exp.end em AX, pois cada posição do vetor gasta 2 bytes!**
- **Some id.end a AX;**
- **AX agora tem o endereço do elemento. Carregue o conteúdo do elemento da memória: mov reg, DS:[AX]**
- **Transfira o conteúdo de reg para o novo temporário.**

Termos

As instruções de divisão entre inteiros requererem a expansão do acumulador para 32 bits (instrução `cwd`).

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

① ***{ T.end := F_1.end } { T.tipo := F_1.tipo }***
② ***{ guardar o tipo do operador (*, /, % ou AND) }***
③ ***{ carregar o conteúdo de T.end no regA (`mov`) }***
{carregar o conteúdo de F_2.end no regB}
{ conforme operador, gerar instrução (`imul`,`idiv`) entre regA e regB) }
{ T.end := NovoTemp}
{ guardar resultado de regA em T.end (`mov`) }

Expressões simples

ExpS → [- ①] T_1 ② { (+ | - | OR 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 ...`) }*
{ 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 OR com instruções aritméticas.**

Rótulos

Deve ser declarada uma função NovoRot que retorna o valor do próximo rótulo, começando por R1, 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 AX e $ExpS_2$.end em BX (converter ambos para inteiro, atribuindo 0 ao registrador alto) }*
{ comparar AX e BX: `cmp` }
{ RotVerdadeiro:=NovoRot }
{ gerar instrução Jxx RotVerdadeiro, onde Jxx será `je` (=), `jne` (<>), `jl` (<), `jg` (>), `jge` (>=), `jle` (<=) }
{ `mov AX, 0` }
{ RotFim := NovoRot }
{ `jmp` RotFim }
{ RotVerdadeiro: }
{ `mov AX, 1` }
{ RotFim: }
{ Exp.end:=NovoTemp }
{ Exp.tipo:=TIPOLÓGICO }
{ `mov` Exp.end, `AX` }

Para strings, a única comparação possível é a de igualdade.

5. Geração de Código para Comandos

Estrutura do Programa

S → Decl Bloco ***{ mov ah,4Ch } { int 21h }***

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

Teste

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

C → if
{ 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 do tipo caractere com n+3 posições, onde n é o tamanho do vetor que receberá a entrada ou 255, o que for menor) e utilizar a interrupção 21h – 0Ah. Na primeira posição, deve-se armazenar n ou 255. A segunda posição será preenchida pelo MSDOS com o número de caracteres digitados, após a leitura do string. Os caracteres são armazenados a partir da 3ª. posição. O DOS também acrescenta o caractere de retorno de cursor após o string, sem no entanto contá-lo :

```
mov     dx, buffer.end
mov     al, 0FFh    ;ou tam do vetor
mov     ds:[buffer.end], al
mov     ah, 0Ah
int     21h
```

A quebra de linha pode ser gerada com a interrupção 21h – 02h

```
mov     ah, 02h
mov     dl, 0Dh
int     21h
mov     DL, 0Ah
int     21h
```

Para o tipo string, devemos transferi-lo para a variável, trocando o caractere de retorno pelo '$'. Caso o dado seja do tipo numérico, antes de ser armazenado na variável correspondente, deve ser convertido. O resultado fica no acumulador:

```
    mov     di, buffer.end+2 ;posição do string
    mov     ax, 0            ;acumulador
    mov     cx, 10           ;base decimal
    mov     dx, 1            ;valor sinal +
    mov     bh, 0
    mov     bl, ds:[di]      ;caractere
    cmp     bx, 2Dh          ;verifica sinal
    jne     R0               ;se não negativo
    mov     dx, -1           ;valor sinal -
    add     di, 1            ;incrementa base
    mov     bl, ds:[di]      ;próximo caractere
R0:
    push    dx               ;empilha sinal
    mov     dx, 0            ;reg. multiplicação
R1:
    cmp     bx, 0dh          ;verifica fim string
    je      R2               ;salta se fim string
    imul    cx               ;mult. 10
    add     bx, -48          ;converte caractere
    add     ax, bx           ;soma valor caractere
    add     di, 1            ;incrementa base
    mov     bh, 0
    mov     bl, ds:[di]      ;próximo caractere
    jmp     R1               ;loop
R2:
    pop     cx               ;desempilha sinal
    imul    cx               ;mult. sinal
```

Saída

A saída para o vídeo é sempre do tipo string. A interrupção 21h – 09h exibe um string na tela, cujo endereço está em DS:DX. Expressões numéricas devem ser carregadas para o registrador AX, como inteiros, e convertidas para um string temporário, cujo endereço é carregado em DI:

```
    mov     di, string.end ;end. string temp.

    mov     cx, 0       ;contador
    cmp     ax,0        ;verifica sinal
    jge     R0          ;salta se número positivo
    mov     bl, 2Dh     ;senão, escreve sinal -
    mov     ds:[di], bl
    add     di, 1       ;incrementa índice
    neg     ax          ;toma módulo do número
R0:
    mov     bx, 10      ;divisor
R1:
    add     cx, 1       ;incrementa contador
    mov     dx, 0       ;estende 32bits p/ div.
    idiv    bx          ;divide DXAX por BX
    push    dx          ;empilha valor do resto
    cmp     ax, 0       ;verifica se quoc. é 0
    jne     R1          ;se não é 0, continua
```

```
;agora, desemp. os valores e escreve o string

R2:
    pop     dx          ;desempilha valor
    add     dx, 30h     ;transforma em caractere
    mov     ds:[di],dl  ;escreve caractere
    add     di, 1       ;incrementa base
    add     cx, -1      ;decrementa contador
    cmp     cx, 0       ;verifica pilha vazia
    jne     R2          ;se não pilha vazia, loop

;grava fim de string

    mov     dl, 024h        ;fim de string
    mov     ds:[di], dl     ;grava '$'

;exibe string

    mov     dx, string.end
    mov     ah, 09h
    int     21h
```

Para o comando *writeln*, gerar também a quebra de linha, conforme descrito anteriormente.