LINGUAGEM DE MÁQUINA, LINGUAGEM DE MONTAGEM E VINCULAÇÃO

FERNANDO MÁRIO DE OLIVEIRA FILHO

9.7.2016

Sumário

1. Bits e bytes . . . 1
2. O computador MAC216 . . . 3
3. Instruções e programas . . . 4
4. Comunicação memória–registradores . . . 5
5. Operações aritméticas e bit-a-bit . . . 6
6. Constantes imediatas . . . 7
7. Instruções de desvio . . . 7
8. Interrupções . . . 8
9. O que ficou de fora . . . 9
10. Organização da memória . . . 9
11. Hello world e cálculo do fatorial . . . 11
12. Exemplos simples de entrada e saída . . . 12
13. Sub-rotinas . . . 13
14. Um simulador . . . 15
15. Linguagem de montagem . . . 16
16. Sub-rotinas e convenção de chamada . . . 18
17. Vinculação . . . 20
18. O algoritmo de Euclides com sub-rotinas . . . 21
19. Crivo de Eratóstenes e linha de comando . . . 22
20. Tipos de dados compostos . . . 25
21. Gerenciamento dinâmico de memória: o sistema de irmãos . . . 26
22. Algoritmos de reserva e liberação . . . 27
23. Decisões de implementação . . . 28
24. Implementação do algoritmo de reserva . . . 29
25. Inicialização . . . 31
26. Árvores de busca binária . . . 32
27. Notas históricas . . . 35
28. Referências . . . 35

Estas notas utilizam um computador fictício, chamado `MAC216`, para desenvolver os principais conceitos acerca de linguagem de máquina, linguagem de montagem e vinculação. O computador `MAC216`, bem como sua linguagem de máquina e a linguagem de montagem `MACAL` são descritos em detalhes. Programas que ilustram diversas técnicas de programação diferentes, como sub-rotinas, são apresentados.

O computador `MAC216` é uma versão simplificada do computador fictício `MMIX` [1], usado por D.E. Knuth em sua série de livros *The Art of Computer Programming* [2]. Muitos dos conceitos encontrados adiante (e, temo, também muito da forma de apresentação) foram emprestados da descrição do `MMIX` feita por Knuth.

§1. Bits e bytes

O computador `MAC216` trabalha com padrões de 0s e 1s, chamados de dígitos binários ou *bits*. Na verdade, `MAC216` geralmente opera em 64 bits de uma só vez, sendo portanto um computador de 64 bits como a maior parte dos computadores modernos.

Um possível padrão de 64 bits é

1011001000000111001010100101011011010001001000001110110001001011.

Podemos descrever esse mesmo padrão mais concisamente usando a *notação hexadecimal*. Para tanto, agrupamos os bits em grupos de quatro e usamos um dígito hexadecimal para representar cada grupo:

$$
\begin{array}{llll}
\texttt{0} = 0000, & \texttt{4} = 0100, & \texttt{8} = 1000, & \texttt{c} = 1100, \\
\texttt{1} = 0001, & \texttt{5} = 0101, & \texttt{9} = 1001, & \texttt{d} = 1101, \\
\texttt{2} = 0010, & \texttt{6} = 0110, & \texttt{a} = 1010, & \texttt{e} = 1110, \\
\texttt{3} = 0011, & \texttt{7} = 0111, & \texttt{b} = 1011, & \texttt{f} = 1111.
\end{array}
$$

Dígitos hexadecimais sempre serão escritos com uma fonte diferente, como acima, e números hexadecimais serão sempre prefixados por #. Por exemplo, o padrão de 64 bits acima fica

#`b2072a56d120ec4b`.

Uma seqüência de 8 bits é chamada de *byte*. Um byte é a menor unidade de memória endereçável para o computador `MAC216`. Bytes podem ser usados para representar caracteres como letras, números, e sinais de pontuação. Uma forma de fazê-lo é através do *American Standard Code for Information Interchange* (ASCII), que é um código de 7 bits que associa caracteres e valores de controle a cada um dos bytes de #`00` a #`7f`. Uma extensão do código ASCII, conhecida como Latin-1 ou ISO 8859-1, também associa caracteres aos bytes de #`80` a #`ff`; nessa extensão é possível representar letras com acentos, etc.

Seguindo Knuth, usaremos os termos *wyde* para representar uma quantidade de 16 bits, *tetrabyte* (ou "tetra") para representar uma quantidade de 32 bits e *octabyte* (ou "octa") para representar uma quantidade de 64 bits.

Padrões de bits podem ser usados para representar números positivos na base 2. Por exemplo, o número 13 é representado em binário por 1101, já que

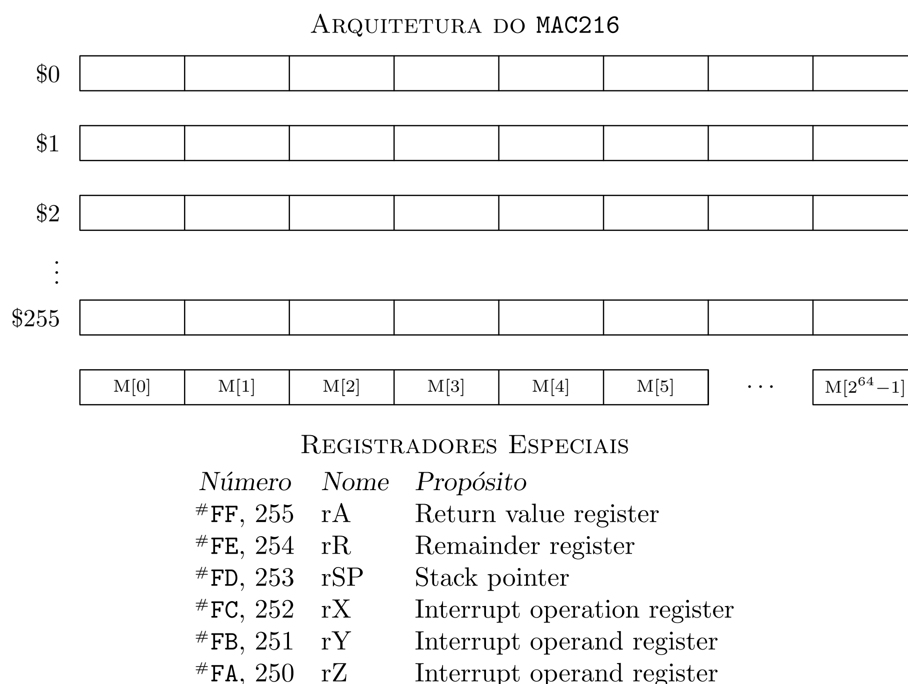
$$13 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^1 + 1 \times 2^0.$$

Note que, como de costume, o bit *mais significativo* da representação, ou seja, aquele que multiplica a maior potência de 2, está escrito à esquerda.

Um byte, wyde, tetra ou octa x pode ser considerado como uma quantidade sem sinal (unsigned), caso em que representa um número não-negativo denotado por $\mathrm{u}(x)$ como descrito acima. Bytes, wydes, tetras e octas sem sinal são capazes de representar diferentes intervalos de números, como mostra a tabela abaixo.

um	pode representar os números
byte	0...255
wyde	0...65.535
tetra	0...4.294.967.295
octa	0...18.446.744.073.709.551.615

Números negativos podem ser representados usando-se a *notação de complemento de dois*. Nesta notação, o bit mais significativo representa o sinal do número. Um byte, wyde, tetra ou octa também pode ser considerado como uma quantidade com sinal (signed), ou seja, como a representação de um número inteiro na notação de complemento de dois. Para determinar o número inteiro $\mathrm{s}(x)$ representado por um byte x com sinal, procedemos da seguinte forma: se o bit mais significativo de x for 0, então $\mathrm{s}(x) = \mathrm{u}(x)$; caso contrário, $\mathrm{s}(x) = \mathrm{u}(x) - 2^8$. Procedemos de maneira análoga se x for um wyde, tetra ou octa com sinal, trocando 8 por 16, 32 ou 64, respectivamente. Assim, -1 é representado pelo byte com sinal #`ff`, bem como pelo wyde com sinal #`ffff`, pelo tetra com sinal #`ffffffff` e pelo octa com sinal #`ffffffffffffffff`.



Registradores Especiais

Número	*Nome*	*Propósito*
#`FF`, 255	rA	Return value register
#`FE`, 254	rR	Remainder register
#`FD`, 253	rSP	Stack pointer
#`FC`, 252	rX	Interrupt operation register
#`FB`, 251	rY	Interrupt operand register
#`FA`, 250	rZ	Interrupt operand register

Figura 1. Arquitetura do `MAC216` e seus registradores especiais. Note que cada registrador é composto de 8 bytes e que há 2^{64} bytes de memória. Os últimos seis registradores são especiais e têm nomes próprios; o uso de cada um é explicado no restante do texto, na medida do necessário. Os registradores especiais rX, rY e rZ são comumente utilizados para guardar valores temporários. Os três registradores reservados não estão indicados acima.

Bytes, wydes, tetras e octas com sinal podem representar diferentes intervalos de números, como mostra a tabela abaixo.

um	pode representar os números
byte	$-128 \ldots 127$
wyde	$-32.768 \ldots 32.767$
tetra	$-2.147.483.648 \ldots 2.147.483.647$
octa	$-9.223.372.036.854.775.808 \ldots 9.223.372.036.854.775.807$

Exercício

1. Descreva algoritmos para somar, subtrair, e calcular o oposto de números inteiros representados na notação de complemento de dois.

§2. O computador `MAC216`

Do ponto de vista do programador, o computador `MAC216` tem 2^{64} células de *memória*, cada uma capaz de conter um byte de informação, e 2^8 *registradores* de 64 bits cada, seis dos quais são especiais. Além disso, há três registradores reservados. A informação é transferida da memória para os registradores, transformada, e transferida de volta dos registradores para a memória.

As células de memória são denotadas por M[0], ..., M[$2^{64} - 1$]. Em geral, se x é um octabyte, então M[x] é um byte de memória. Os registradores são denotados por $0, ..., $255 e alguns deles têm nomes especiais; estes servem para diferentes propósitos, como veremos mais adiante (Figura 1). Finalmente, os três registradores reservados são denotados por @, rI e rIB.

Os 2^{64} bytes de memória são divididos em 2^{63} wydes denotados por $M_2[0] = M_2[1] = M[0]M[1]$, $M_2[2] = M_2[3] = M[2]M[3]$, ... Cada um desses wydes consiste de dois bytes $M[2k]M[2k+1]$ consecutivos e é denotado por $M_2[2k]$ ou por $M_2[2k+1]$, indistintamente.

O primeiro byte de $M_2[2k]$ é o mais significativo, ou seja,

$$u(M_2[2k]) = u(M[2k]) \times 2^8 + u(M[2k+1]).$$

Em termos técnicos, isso quer dizer que o computador `MAC216` segue a convenção *big-endian*, ou seja, o byte mais significativo de um número ocupa a posição de memória de menor índice. Essa é a mesma notação que usamos para escrever números em notação decimal, binária ou hexadecimal: o dígito mais à esquerda, ou seja, o primeiro na ordem de leitura, é o mais significativo. Em contrapartida, a maior parte dos computadores modernos utiliza a convenção *little-endian*: o primeiro byte na memória corresponde ao dígito menos significativo. Essa diferença entre o `MAC216` e computadores modernos importa apenas quando escrevemos um simulador.

Da mesma forma que dividimos a memória em 2^{63} wydes, também a dividimos em 2^{62} tetrabytes

$$M_4[4k] = M_4[4k+1] = \cdots = M_4[4k+3] = M[4k]M[4k+1]\cdots M[4k+3]$$

e 2^{61} octabytes

$$M_8[8k] = M_8[8k+1] = \cdots = M_8[8k+7] = M[8k]M[8k+1]\cdots M[8k+7].$$

Em geral, se x é um octabyte, então $M_2[x]$, $M_4[x]$ e $M_8[x]$ são o wyde, tetra ou octa que contém o byte na posição x. Escrevemos também $M_1[x] = M[x]$.

§3. Instruções e programas

Um programa para o `MAC216` fica guardado na memória assim como os dados sobre os quais opera, ou seja, a memória do `MAC216` contém não só dados mas também instruções.

Uma *instrução* é um tetrabyte cujos bytes são denotados por OP, X, Y e Z, do mais significativo para o menos significativo. O byte OP é o *código de operação* (*operation code* ou *opcode*); os bytes X, Y e Z especificam *operandos*. Por exemplo, o tetrabyte `#2001ff03`, com OP = `#20`, X = `#01`, Y = `#ff` e Z = `#03`, representa a instrução "coloque em $1 o resultado da soma de $255 e $3". Cada instrução tem uma representação simbólica, consistente com a linguagem de montagem que será descrita na §15; a representação simbólica da instrução `#20`, por exemplo, é `ADD`. Cada um dos bytes X, Y e Z também tem uma representação simbólica; por exemplo, a instrução `#2001ff03` é representada por '`ADD $1,$255,$3`'. Em sua forma geral, a instrução `ADD` é escrita '`ADD $X,$Y,$Z`'.

A maior parte das instruções tem três operandos. Algumas instruções tem apenas dois operandos; nesse caso, o primeiro operando é X e o segundo operando é o wyde $YZ = 2^8 \times Y + Z$, ou seja, o byte Z é o menos significativo. Há ainda instruções que possuem apenas um operando, que é a quantidade de três bytes $XYZ = 2^{16} \times X + 2^8 \times Y + Z$, de modo que o byte Z é o menos significativo.

Um programa é executado da seguinte forma. O registrador reservado @, o *instruction pointer*, aponta para a posição da memória que contém a instrução a ser executada a seguir. A instrução é lida, executada, e @ passa a apontar para a próxima instrução, ou, formalmente, $@ \leftarrow @ + 4$, a não ser que a instrução executada seja uma instrução de desvio, que pode mudar o conteúdo do registrador @ de forma arbitrária.

Nas seções seguintes serão descritas as instruções do `MAC216`. Junto a uma descrição informal do que faz uma determinada instrução também será dada uma descrição formal. A instrução '`ADD $X,$Y,$Z`' por exemplo "coloca em $X a soma de $Y

e \$Z" ou, formalmente, $s(\$X) \leftarrow s(\$Y) + s(\$Z)$, o que significa que o registrador X recebe o octa com sinal que é o resultado da soma dos octas com sinal contidos nos registradores Y e Z. Os códigos correspondentes a cada instrução podem ser encontrados na Tabela 2, §9.

§4. Comunicação memória–registradores

Vejamos primeiro as instruções que tratam da movimentação de dados da memória para os registradores e dos registradores para a memória. Todas estas instruções usam os três operandos X, Y e Z. Os operandos Y e Z são sempre usados para calcular um *endereço de memória* A da seguinte forma:

$$A = (u(\$Y) + u(\$Z)) \bmod 2^{64}.$$

Começamos com as instruções que movem dados da memória para os registradores, tratando-os como quantidades com sinal:

- `LDB $X,$Y,$Z` (load byte): $s(\$X) \leftarrow s(M_1[A])$.
- `LDW $X,$Y,$Z` (load wyde): $s(\$X) \leftarrow s(M_2[A])$.
- `LDT $X,$Y,$Z` (load tetra): $s(\$X) \leftarrow s(M_4[A])$.
- `LDO $X,$Y,$Z` (load octa): $s(\$X) \leftarrow s(M_8[A])$.

As instruções acima carregam o byte, wyde, tetra ou octa com sinal que ocupa a posição de memória A no registrador \$X, transformando o byte, wyde, tetra ou octa original num octa com sinal de mesmo valor. Por exemplo, suponha que M[1000] = #`0f` = 15, \$2 = 1000 e \$3 = 0. Então após a instrução '`LDB $1,$2,$3`' temos \$1 = #`000000000000000f`. Se tivéssemos M[1000] = #`ff` = −1, então teríamos \$1 = #`ffffffffffffffff` após a mesma instrução. Assim, quando um byte, wyde ou tetra com sinal é convertido num octa com sinal, seu bit de sinal é "estendido" para todas as posições à esquerda.

A cada instrução de carga com sinal descrita acima corresponde uma instrução de carga sem sinal, que trata as quantidades em questão como quantidades sem sinal:

- `LDBU $X,$Y,$Z` (load byte unsigned): $u(\$X) \leftarrow u(M_1[A])$.
- `LDWU $X,$Y,$Z` (load wyde unsigned): $u(\$X) \leftarrow u(M_2[A])$.
- `LDTU $X,$Y,$Z` (load tetra unsigned): $u(\$X) \leftarrow u(M_4[A])$.
- `LDOU $X,$Y,$Z` (load octa unsigned): $u(\$X) \leftarrow u(M_8[A])$.

As instruções `LDO` e `LDOU` comportam-se da mesma forma, mas há aí uma diferença semântica importante. Bons programadores atentam para essa diferença, de modo a tornar o programa mais legível.

As próximas instruções transferem dados de um registrador para uma posição de memória, tratando todas as quantidades envolvidas como quantidades com sinal:

- `STB $X,$Y,$Z` (store byte): $s(M_1[A]) \leftarrow s(\$X)$.
- `STW $X,$Y,$Z` (store wyde): $s(M_2[A]) \leftarrow s(\$X)$.
- `STT $X,$Y,$Z` (store tetra): $s(M_4[A]) \leftarrow s(\$X)$.
- `STO $X,$Y,$Z` (store octa): $s(M_8[A]) \leftarrow s(\$X)$.

Aqui, ocorre *overflow* se a quantidade guardada no registrador \$X não puder ser representada num byte, wyde ou tetra, a depender da instrução utilizada. Para manter o computador `MAC216` o mais simples possível, overflows são ignorados, embora fosse mais realista ter um registrador especial para indicar se ocorreu overflow durante a execução de uma instrução.

A cada instrução acima corresponde uma instrução que trata todas as quantidades envolvidas como quantidades sem sinal:

- `STBU $X,$Y,$Z` (store byte unsigned): $\mathrm{u}(\mathrm{M}_1[\mathrm{A}]) \leftarrow \mathrm{u}(\$\mathrm{X}) \bmod 2^8$.
- `STWU $X,$Y,$Z` (store wyde unsigned): $\mathrm{u}(\mathrm{M}_2[\mathrm{A}]) \leftarrow \mathrm{u}(\$\mathrm{X}) \bmod 2^{16}$.
- `STTU $X,$Y,$Z` (store tetra unsigned): $\mathrm{u}(\mathrm{M}_4[\mathrm{A}]) \leftarrow \mathrm{u}(\$\mathrm{X}) \bmod 2^{32}$.
- `STOU $X,$Y,$Z` (store octa unsigned): $\mathrm{u}(\mathrm{M}_8[\mathrm{A}]) \leftarrow \mathrm{u}(\$\mathrm{X})$.

Finalmente, na maioria dos programas é útil poder carregar constantes num registrador. Isso pode ser feito usando-se a seguinte instrução, que carrega no registrador $X o wyde sem sinal YZ:

- `SETW $X,YZ` (load constant wyde): $\mathrm{u}(\$\mathrm{X}) \leftarrow \mathrm{YZ}$.

§5. Operações aritméticas e bit-a-bit

Exceto pelas instruções acima e pelas instruções `SAVE` e `UNSAVE` da §9, todas as demais instruções do `MAC216` manipulam dados apenas entre os registradores. Temos por exemplo as quatro operações aritméticas fundamentais:

- `ADD $X,$Y,$Z` (add): $\mathrm{s}(\$\mathrm{X}) \leftarrow \mathrm{s}(\$\mathrm{Y}) + \mathrm{s}(\$\mathrm{Z})$.
- `SUB $X,$Y,$Z` (subtract): $\mathrm{s}(\$\mathrm{X}) \leftarrow \mathrm{s}(\$\mathrm{Y}) - \mathrm{s}(\$\mathrm{Z})$.
- `MUL $X,$Y,$Z` (multiply): $\mathrm{s}(\$\mathrm{X}) \leftarrow \mathrm{s}(\$\mathrm{Y}) \times \mathrm{s}(\$\mathrm{Z})$.
- `DIV $X,$Y,$Z` (divide): $\mathrm{s}(\$\mathrm{X}) \leftarrow \mathrm{s}(\$\mathrm{Y}) / \mathrm{s}(\$\mathrm{Z})$ $[\$\mathrm{Z} \neq 0]$ e $\mathrm{s}(\mathrm{rR}) \leftarrow \mathrm{s}(\$\mathrm{Y}) \bmod \mathrm{s}(\$\mathrm{Z})$.

As instruções de adição, subtração e multiplicação devem estar claras, exceto pela observação de que pode ocorrer overflow. A instrução de divisão necessita de algumas explicações adicionais. Ela segue a definição de divisão e resto do padrão ISO/IEC 9899:1999 para a linguagem C (conhecido como C99). Se $\$\mathrm{Z} \neq 0$, então o quociente da divisão inteira $\mathrm{s}(\$\mathrm{Y}) / \mathrm{s}(\$\mathrm{Z})$ é colocado no registrador $\mathrm{s}(\$\mathrm{X})$. O quociente é a parte inteira do resultado da divisão, ou seja, $10/3 = 3$ e $-10/3 = -3$. O registrador especial rR (*remainder register*) guarda o resto da divisão, de modo a ter-se, após a execução da instrução,

$$\mathrm{s}(\$\mathrm{X}) \times \mathrm{s}(\$\mathrm{Z}) + \mathrm{s}(\mathrm{rR}) = \mathrm{s}(\$\mathrm{Y}).$$

Isso significa, por exemplo, que se $\mathrm{s}(\$\mathrm{Y}) = -10$ e $\mathrm{s}(\$\mathrm{Z}) = 3$, então após a divisão temos $\mathrm{s}(\$\mathrm{X}) = -3$ e $\mathrm{s}(\mathrm{rR}) = -1$. Se $\$\mathrm{Z} = 0$, então $\mathrm{s}(\$\mathrm{X}) \leftarrow 0$ e $\mathrm{s}(\mathrm{rR}) \leftarrow \mathrm{s}(\$\mathrm{Y})$.

Às instruções aritméticas acima correspondem instruções que consideram as quantidades envolvidas como quantidades sem sinal:

- `ADDU $X,$Y,$Z` (add unsigned): $\mathrm{u}(\$\mathrm{X}) \leftarrow (\mathrm{u}(\$\mathrm{Y}) + \mathrm{u}(\$\mathrm{Z})) \bmod 2^{64}$.
- `SUBU $X,$Y,$Z` (subtract unsigned): $\mathrm{u}(\$\mathrm{X}) \leftarrow (\mathrm{u}(\$\mathrm{Y}) - \mathrm{u}(\$\mathrm{Z})) \bmod 2^{64}$.
- `MULU $X,$Y,$Z` (multiply unsigned): $\mathrm{u}(\$\mathrm{X}) \leftarrow (\mathrm{u}(\$\mathrm{Y}) \times \mathrm{u}(\$\mathrm{Z})) \bmod 2^{64}$.
- `DIVU $X,$Y,$Z` (divide unsigned): $\mathrm{u}(\$\mathrm{X}) \leftarrow \mathrm{u}(\$\mathrm{Y}) / \mathrm{u}(\$\mathrm{Z})$ $[\$\mathrm{Z} \neq 0]$ e $\mathrm{u}(\mathrm{rR}) \leftarrow \mathrm{u}(\$\mathrm{Y}) \bmod \mathrm{u}(\$\mathrm{Z})$.

A instrução `DIVU` comporta-se de maneira análoga à instrução `DIV`.

Também temos instruções de shift, com e sem sinal:

- `SL $X,$Y,$Z` (shift left): $\mathrm{s}(\$\mathrm{X}) \leftarrow \mathrm{s}(\$\mathrm{Y}) \times 2^{\mathrm{u}(\mathrm{Z})}$.
- `SLU $X,$Y,$Z` (shift left unsigned): $\mathrm{u}(\$\mathrm{X}) \leftarrow (\mathrm{u}(\$\mathrm{Y}) \times 2^{\mathrm{u}(\mathrm{Z})}) \bmod 2^{64}$.
- `SR $X,$Y,$Z` (shift right): $\mathrm{s}(\$\mathrm{X}) \leftarrow \mathrm{s}(\$\mathrm{Y}) / 2^{\mathrm{u}(\mathrm{Z})}$.
- `SRU $X,$Y,$Z` (shift right unsigned): $\mathrm{u}(\$\mathrm{X}) \leftarrow (\mathrm{u}(\$\mathrm{Y}) / 2^{\mathrm{u}(\mathrm{Z})}) \bmod 2^{64}$.

Nas instruções `NEG` e `NEGU` o byte Y é uma constante sem sinal, não o número de um registrador:

- `NEG $X,Y,$Z` (negate): s($X) $\leftarrow$ Y − s($Z).
- `NEGU $X,Y,$Z` (negate unsigned): u($X) $\leftarrow$ (Y − u($Z)) mod 2^{64}.

Finalmente temos as operações de comparação, com e sem sinal:

- `CMP $X,$Y,$Z` (compare): s($X) recebe −1, 0 ou 1 se s($Y) é menor, igual ou maior (resp.) que s($Z).
- `CMPU $X,$Y,$Z` (compare unsigned): s($X) recebe −1, 0 ou 1 se u($Y) é menor, igual ou maior (resp.) que u($Z).

Operações bit-a-bit (*bitwise operations*) consideram um octabyte x como um vetor v(x) de 64 bits e operam simultaneamente em cada componente desse vetor.

- `AND $X,$Y,$Z` (bitwise and): v($X) $\leftarrow$ v($Y) $\wedge$ v($Z).
- `OR $X,$Y,$Z` (bitwise or): v($X) $\leftarrow$ v($Y) $\vee$ v($Z).
- `XOR $X,$Y,$Z` (bitwise exclusive-or): v($X) $\leftarrow$ v($Y) $\oplus$ v($Z).
- `NXOR $X,$Y,$Z` (bitwise not-exclusive-or): $\bar{\text{v}}$($X) $\leftarrow$ v($Y) $\oplus$ v($Z).

Aqui $\bar{\text{v}}$ denota o *complemento* do vetor v, obtido deste pela troca de 0s por 1s e vice-versa. As operações $\wedge$, $\vee$ e $\oplus$ são aplicadas independentemente a cada bit. Elas são definidas da seguinte forma:

$$\begin{array}{lll} 0 \wedge 0 = 0, & 0 \vee 0 = 0, & 0 \oplus 0 = 0, \\ 0 \wedge 1 = 0, & 0 \vee 1 = 1, & 0 \oplus 1 = 1, \\ 1 \wedge 0 = 0, & 1 \vee 0 = 1, & 1 \oplus 0 = 1, \\ 1 \wedge 1 = 1, & 1 \vee 1 = 1, & 1 \oplus 1 = 0. \end{array}$$

§6. Constantes imediatas

Programas freqüentemente utilizam pequenas constantes numéricas. Por exemplo, é útil poder somar 1 ao valor de um registrador sem ter que carregar o número 1 em outro registrador. Por isso, a cada instrução da forma '`OP $X,$Y,$Z`' que lida com três operandos corresponde uma *versão imediata* na qual o terceiro operando, Z, é tratado como um valor de 8 bits sem sinal e não como o número de um registrador. A versão imediata da instrução '`OP $X,$Y,$Z`' é escrita como '`OPI $X,$Y,Z`' e seu opcode é o opcode da versão não-imediata mais 1.

Para subtrair 15 da quantidade sem sinal guardada no registrador $20 usamos a instrução '`SUBUI $20,$20,15`'. Para copiar o conteúdo do registrador $2 para o registrador $1 podemos usar '`ORI $1,$2,0`'.

É importante ressaltar que uma instrução e sua versão imediata têm opcodes diferentes e são, portanto, instruções diferentes do ponto de vista do processador.

Exercícios

1. Mostre como carregar qualquer octa num dado registrador.
2. Mostre como trocar os valores de dois registradores sem usar um terceiro registrador, usando o menor número possível de instruções.
3. De que maneiras podemos fazer '$0 $\leftarrow$ 0' usando apenas uma instrução do `MAC216`?

§7. Instruções de desvio

Para escrever um programa que faça algo de útil é preciso poder desviar o fluxo das instruções. Uma forma de fazê-lo é através da instrução `JMP`, que causa um desvio incondicional:

- `JMP XYZ` (jump): @ $\leftarrow$ @ + 4 × XYZ.
- `JMPB XYZ` (jump back): @ $\leftarrow$ @ − 4 × XYZ.

A quantidade sem sinal XYZ é formada pelos bytes X, Y e Z como explicado acima (lembre-se de que o byte Z é o menos significativo e de que X é o mais significativo). Assim, é possível desviar o fluxo 16.777.215 instruções para frente ou para trás.

Também há instruções que desviam o fluxo de forma condicional, com base no valor de um dado registrador. São elas:

- `JZ $X,YZ` (jump if zero): $@ \leftarrow @ + 4 \times \mathrm{YZ}$ se $s(\$X) = 0$.
- `JNZ $X,YZ` (jump if nonzero): $@ \leftarrow @ + 4 \times \mathrm{YZ}$ se $s(\$X) \neq 0$.
- `JP $X,YZ` (jump if positive): $@ \leftarrow @ + 4 \times \mathrm{YZ}$ se $s(\$X) > 0$.
- `JN $X,YZ` (jump if negative): $@ \leftarrow @ + 4 \times \mathrm{YZ}$ se $s(\$X) < 0$.
- `JNN $X,YZ` (jump if nonnegative): $@ \leftarrow @ + 4 \times \mathrm{YZ}$ se $s(\$X) \geq 0$.
- `JNP $X,YZ` (jump if nonpositive): $@ \leftarrow @ + 4 \times \mathrm{YZ}$ se $s(\$X) \leq 0$.

A cada uma dessas instruções corresponde uma versão que desvia o fluxo para trás. Por exemplo, temos a instrução '`JZB $X,YZ`', que faz $@ \leftarrow @ - 4 \times \mathrm{YZ}$ se $s(\$X) = 0$. A versão de uma instrução de desvio que desvia o fluxo para trás tem opcode igual ao da instrução que desvia o fluxo para frente mais 1. Note também que os desvios condicionais podem acessar apenas instruções mais próximas: podemos desviar no máximo 65.535 instruções para frente ou para trás.

Finalmente, também é possível desviar para uma posição de memória arbitrária usando a instrução `GO`:

- `GO $X,YZ` (go): $@ \leftarrow u(\$X) + 4 \times \mathrm{YZ}$.
- `GOB $X,YZ` (go back): $@ \leftarrow u(\$X) - 4 \times \mathrm{YZ}$.

Para usar as duas instruções acima, é útil poder carregar num registrador um endereço de memória relativo a uma instrução. Isso pode ser feito com a instrução `GETA`:

- `GETA $X,YZ` (get address): $u(\$X) \leftarrow @ + 4 \times \mathrm{YZ}$.
- `GETAB $X,YZ` (get address back): $u(\$X) \leftarrow @ - 4 \times \mathrm{YZ}$.

Por exemplo, a instrução '`GETA $0,0`' coloca em \$0 o endereço de memória da instrução atual (ela mesma).

§8. Interrupções

Para efetuar operações de entrada e saída, entre outras, um programa deve ser capaz de comunicar-se com o sistema operacional. Para isso serve a instrução `INT`:

- `INT XYZ` (interrupt): transfere o controle para o sistema operacional. O tetrabyte mais significativo do registrador especial rX (o *operation register*) é mudado para XYZ; o tetrabyte menos significativo permanece como antes. O endereço da próxima instrução é guardado no registrador rIB, ou seja, faz-se $\mathrm{rIB} \leftarrow @ + 4$. O fluxo é desviado fazendo-se $@ \leftarrow u(\$\mathrm{rI})$; o registrador rI (*interruption address register*) contém o endereço de memória da rotina do sistema operacional que tratará da interrupção.

Os registradores rY e rZ, bem como o tetrabyte menos significativo de rX, são usados para passar informações para a rotina que trata a interrupção. O registrador rA é usado pela rotina para devolver algum valor ao programa que causou a interrupção. O sistema fornece a seguinte garantia sobre os valores dos registradores: quando o fluxo volta para a instrução após a interrupção, o conteúdo de todos os registradores, exceto talvez por rA, é o mesmo de quando a interrupção foi chamada.

Cada sistema operacional pode oferecer diferentes tipos de interrupção. O simulador apresentado na §14 provê alguns códigos de interrupção:

- `INT 0`: causa o encerramento do programa.
- `INT #80`: executa operações de entrada e saída. Se o tetrabyte menos significativo do registrador rX é 1, então um caractere é lido da entrada padrão e o código correspondente é colocado no registrador rA; se o fim da entrada foi encontrado, então -1 é colocado em rA. Se o tetrabyte menos significativo do registrador rX é 2, então o caractere cujo código se encontra no registrador rY é escrito na saída padrão.
- `INT #DBYYZZ`: escreve na saída padrão o conteúdo dos registradores de números entre Y e Z. Por exemplo, para mostrar o conteúdo dos registradores $10, ..., $20, usa-se a instrução '`INT #DB0A14`'. Este código de interrupção é útil para depuração.
- `INT #ADYYZZ`: escreve na saída padrão o conteúdo da posição de memória cujo endereço se encontra no registrador $Y. Se $Z = 1$, então mostra-se o conteúdo do byte que ocupa a posição; se $Z = 2$, então mostra-se o conteúdo do wyde e assim por diante até $Z = 8$. Por exemplo, se $11 = 1000, então a instrução '`INT #AD0B04`' mostra o conteúdo do tetra $M_4[1000]$.

Há ainda um código de interrupção útil para gerenciamento de memória, que será discutido na §21.

§9. O que ficou de fora

As duas instruções a seguir são úteis para lidar com chamadas de sub-rotinas (veja §16):

- `SAVE $X,$Y,$Z` (save): grava nas posições de memória $X, $X + 8, ... o conteúdo dos registradores de números Y, Y + 1, ..., Z e faz u($X) ← u($X) + 8(Z − Y + 1).
- `REST $X,$Y,$Z` (restore): faz u($X) ← u($X) − 8(Z − Y + 1) e carrega nos registradores de números Y, Y + 1, ..., Z os valores contidos nas posições de memória $X, $X+8, ... Assim, a instrução '`REST $X,$Y,$Z`' desfaz a operação '`SAVE $X,$Y,$Z`'.

A última instrução do `MAC216` é a instrução `NOP` (no operation), que não faz absolutamente nada; os bytes X, Y e Z são ignorados. A Tabela 2 contém todas as instruções descritas acima e seus opcodes.

Um computador mais realista que o `MAC216` teria outras instruções importantes, como por exemplo:

- instruções para manipulação de números de ponto flutuante;
- instruções para comunicação com periféricos;
- instruções para gerenciamento de memória virtual;
- instruções para tirar vantagem de paralelismo, como por exemplo de pipelining; veja por exemplo a instrução `PBN` (probable branch if negative) e similares do `MMIX`.

Fora isso, a máquina `MAC216` é bem parecida com os computadores atuais, embora seja um pouco mais simples e amigável em relação ao programador. As mesmas considerações feitas por Knuth acerca de como a máquina `MMIX` se compara aos computadores modernos também valem para o `MAC216`.

Finalmente, também foi deixada de fora uma discussão do tempo gasto por cada instrução. Nem todas as instruções gastam o mesmo tempo: instruções que acessam a memória são muito mais demoradas do que instruções que operam apenas com registradores. Mesmo instruções aritméticas levam tempos diferentes: somar é mais rápido do que multiplicar, que é mais rápido do que dividir. Instruções bit-a-bit e de shift são mais rápidas que somas.

	#0	#1	#2	#3	#4	#5	#6	#7	
#0x	LDB[I]		LDW[I]		LDT[I]		LDO[I]		#0x
	LDBU[I]		LDWU[I]		LDTU[I]		LDOU[I]		
#1x	STB[I]		STW[I]		STT[I]		STO[I]		#1x
	STBU[I]		STWU[I]		STTU[I]		STOU[I]		
#2x	ADD[I]		SUB[I]		MUL[I]		DIV[I]		#2x
	CMP[I]		SL[I]		SR[I]		NEG[I]		
#3x	ADDU[I]		SUBU[I]		MULU[I]		DIVU[I]		#3x
	CMPU[I]		SLU[I]		SRU[I]		NEGU[I]		
#4x	AND[I]		OR[I]		XOR[I]		NXOR[I]		#4x
	JMP[B]		JZ[B]		JNZ[B]		JP[B]		
#5x	JN[B]		JNN[B]		JNP[B]		GO[B]		#5x
	GETA[B]		SETW	SAVE	REST				
#fx									#fx
							INT	NOP	
	#8	#9	#a	#b	#c	#d	#e	#f	

Tabela 2. Opcodes da máquina `MAC216`. Por exemplo, a instrução `ADD` aparece na metade de cima da linha `#2x`, na coluna `#0`, assim seu opcode é `#20`. A instrução `ADDI`, versão imediata de `ADD`, tem opcode `#21`. Esse padrão é seguido por todas as instruções que têm versões imediatas (marcadas com `[I]` na tabela) e também pelas instruções de desvio e `GETA`, que tem versões que procuram endereços para trás (marcadas com `[B]` na tabela). A instrução `SRUI` é a versão imediata de `SRU`, que ocupa a metade de baixo da linha `#3x` e aparece na coluna `#c`, assim o opcode de `SRUI` é `#3d`.

§10. Organização da memória

O espaço virtual de endereçamento do `MAC216`, composto de 2^{64} bytes, é dividido em duas partes de 2^{63} bytes cada. A primeira parte, composta pelos endereços de `#0000000000000000` a `#7fffffffffffffff`, é chamada de *espaço do usuário* ou *user space*; é nela que que residem o programa do usuário e os dados sobre os quais ele opera. A segunda parte, composta pelos endereços de `#8000000000000000` a `#ffffffffffffffff`, é chamada de *espaço do kernel* ou *kernel space*; é nela que reside o sistema operacional.

O espaço do usuário é ainda dividido em três *segmentos*. O primeiro, de 2^{62} bytes, é chamado de *segmento do heap* ou *heap segment*. Esse trecho de memória pode ser usado pelo programa de qualquer forma desejada.

O segundo segmento, de 2^{61} bytes, é chamado de *segmento da pilha* ou *stack segment* e contém a pilha de execução, usada nas chamadas de sub-rotinas (ver §16). No início de um programa, o registrador rSP contém o endereço inicial do primeiro octa livre do segmento de pilha. Os argumentos passados ao programa pela linha de comando são colocados na pilha (veja §19). O terceiro segmento, também de 2^{61} bytes, é chamado de *segmento do texto* ou *text segment* e contém o programa sendo executado. A primeira instrução do programa começa no primeiro byte desse segmento; o registrador @ contém o endereço desse byte no início do programa.

Observe que os endereços de início e fim de cada um dos espaços e segmentos são *virtuais*. Como usa registradores de 64-bits, o MAC216 pode endereçar 2^{64} bytes de memória, mas dificilmente terá acesso a tanta memória. Assim, os endereços virtuais são mapeados em endereços reais, de forma transparente para o programador.

§11. Hello world e cálculo do fatorial

O primeiro exemplo de programa é o tradicional "Hello world!".

Programa 1. Escreve "Hello world!" na saída padrão.

```
1  #48000005  JMP    5
2  #48656c6c  "Hello␣world!\n"
3  #6f20776f
4  #726c6421
5  #0a000000
6  #5afc0002  SETW   rX,2
7  #59000005  GETAB  $0,5
8  #01fb0000  LDBI   rY,$0,0
9  #4afb0004  JZ     rY,4
10 #fe000080  INT    #80
11 #31000001  ADDUI  $0,$0,1
12 #49000004  JMPB   4
13 #fe000000  INT    0  ▌
```

A descrição acima tem 4 colunas. A primeira contém a numeração das linhas para referência no texto. A segunda coluna contém instruções de máquina. A terceira contém o nome simbólico da operação e a quarta seus operandos.

Quando fornecido ao simulador do MAC216, o programa acima é carregado na memória na ordem em que está dado. A primeira instrução a ser executada é a que está na linha 1, que desvia o fluxo para a instrução da linha 6. Melhor assim, pois as linhas 2–5 não são instruções, mas apenas dados. A saber, os bytes que compõe os tetrabytes das linhas 2–5 são os códigos ASCII dos caracteres da seqüência "Hello␣world!\n", que termina com uma quebra-de-linha, seguidos de um byte 0 que marca o fim da seqüência, já que 0 não representa nenhum caractere (os demais bytes 0 na linha 5 estão lá apenas para preencher o resto do espaço do tetrabyte).

A idéia agora é percorrer cada caractere da seqüência e escrevê-lo na saída padrão, parando ao encontrar um byte 0. O número 2 colocado no registrador rX (linha 6) indicará juntamente com o código de interrupção #80 (linha 10) que o caractere cujo código se encontra em rY deve ser escrito na saída padrão (veja §8). Na linha 7, carregamos o endereço do primeiro byte da seqüência "Hello␣world!\n" no registrador $0. Na linha 8, carregamos em rY o valor do byte que se encontra na posição de memória contida em $0. Se o valor do byte for 0 (linha 9), então pulamos quatro instruções adiante (i.e., pulamos para a linha 13) e o programa é encerrado. Caso contrário executamos uma interrupção com código #80 (linha 10) e o caractere cujo código está em rY é escrito na saída padrão. Daí passamos para o endereço do byte seguinte (linha 11) e pulamos 4 instruções para trás (linha 12), voltando para a linha 8.

Depois de ter entendido o programa acima, deve ser fácil entender o seguinte programa, que calcula 10!.

Programa 2. Guarda 10! no registrador $0.

```
#5a01000a  SETW   $1,10
#5a000001  SETW   $0,1
#4a010004  JZ     $1,4
#34000001  MULU   $0,$0,$1
#33010101  SUBUI  $1,$1,1
```

```
#49000003  JMPB   3
#fedb0000  INT    #DB0000    Mostra 10! na saída padrão.
#fe000000  INT    0  ▌
```

§12. Exemplos simples de entrada e saída

Vejamos agora alguns exemplos simples de programas que lidam com entrada e saída. Nosso primeiro programa lê um número inteiro sem sinal da entrada padrão, supondo que ele esteja escrito na base 10. O número lido é armazenado no registrador \$0. A leitura pára quando um caractere diferente de um dígito decimal é lido, ou quando se chega ao fim do arquivo.

Programa 3. Lê um número escrito na base 10 da entrada padrão e o armazena no registrador \$0. O registrador \$1 é usado para guardar o resultado de comparações.

```
 1  #44000000  XOR    $0,$0,$0   Faz $0 ← 0.
 2  #5afc0001  SETW   rX,1
 3  #fe000080  INT    #80        Lê um caractere.
 4  #2901ff30  CMPI   $1,rA,48   O caractere vem antes do 0?
 5  #50010007  JN     $1,7
 6  #2901ff39  CMPI   $1,rA,57   O caractere vem depois do 9?
 7  #4e010005  JP     $1,5
 8  #33ffff30  SUBUI  rA,rA,48
 9  #3500000a  MULUI  $0,$0,10
10  #300000ff  ADDU   $0,$0,rA
11  #49000008  JMPB   8
12  #fe000000  INT    0  ▌
```

Algo um pouco mais complicado de fazer é escrever um número na saída padrão. A dificuldade está em escrever o número da forma natural, com o dígito mais significativo primeiro. O programa abaixo escreve na saída padrão, na base 10, o número sem sinal que está no registrador \$0. Usamos a seguinte observação: dado $n \geq 0$, o k-ésimo dígito de n, do menos significativo para o mais significativo, é igual a $\lfloor n/10^k \rfloor \bmod 10$. Para usar essa observação, o programa abaixo primeiro calcula a potência p de 10 que tem o mesmo número de dígitos de n, de modo que se $0 \leq n < 10$, então $p = 1$, se $10 \leq n < 100$, então $p = 10$, e assim por diante.

Programa 4. Escreve na saída padrão, na base 10, o número sem sinal contido em \$0. Assim, o programa abaixo escreve 1717 na saída padrão.

```
 1  #5a0006b5  SETW   $0,1717
 2  #5a010001  SETW   $1,1       Calcula p e guarda em $1.
 3  #43020000  ORI    $2,$0,0
 4  #3702020a  DIVUI  $2,$2,10
 5  #4a020004  JZ     $2,4
 6  #3702020a  DIVUI  $2,$2,10
 7  #3501010a  MULUI  $1,$1,10
 8  #49000003  JMPB   3
 9  #5afc0002  SETW   rX,2       Escreve o número.
10  #4a010007  JZ     $1,7
11  #36020001  DIVU   $2,$0,$1
12  #3702020a  DIVUI  $2,$2,10
13  #31fbfe30  ADDUI  rY,rR,48
14  #fe000080  INT    #80
15  #3701010a  DIVUI  $1,$1,10
16  #49000006  JMPB   6
17  #fe000000  INT    0  ▌
```

As linhas 2–8 calculam a potência p de 10 definida acima e as linhas 9–16 escrevem os dígitos do número, do mais significativo para o menos significativo.

Exercícios

1. Escreva um programa que lê caracteres da entrada padrão e os escreve na saída padrão, parando quando o fim do arquivo é encontrado.
2. Combine os programas 2, 3 e 4 num programa que lê um número n da entrada padrão e escreve $n!$ na saída padrão.
3. Modifique o Programa 3 para que leia um número com sinal: se a entrada padrão contém os caracteres `-5` então seu programa deve armazenar no registrador $0 o número -5.
4. Modifique o Programa 4 para que escreva um número com sinal na saída padrão.

§13. Sub-rotinas

Operações como leitura de números da entrada padrão, escrita de números na saída padrão, etc., são em geral efetuadas diversas vezes por um mesmo programa. Se cada vez que quiséssemos ler um número da entrada padrão, por exemplo, precisássemos escrever um código parecido com o do Programa 4, então nossos programas seriam enormes! Sub-rotinas são uma forma de reutilizar código, facilitando o trabalho do programador.

Uma sub-rotina é um trecho de código que executa determinada tarefa (ler um número, escrever um número, etc.). Para *chamar* uma sub-rotina, um programa pula para sua primeira instrução. Quando a sub-rotina acaba, ela deve *retornar*, ou seja, devolver o fluxo para a instrução que segue aquela que a chamou. Para que isso seja possível, é necessário estabelecer uma convenção para chamadas de sub-rotinas. Na §16 descreveremos uma convenção de chamada parecida com a usada pela linguagem C; nesta seção vamos inventar uma convenção mais simples.

Nossa convenção é a seguinte:

- O registrador $10, que será chamado de Ret no restante desta seção, guarda o endereço de retorno. Assim, para devolver o fluxo para quem a chamou, uma sub-rotina executa a instrução '`GO Ret,0`'.
- Se a sub-rotina quer devolver algum valor para quem a chamou, este valor deve ser colocado no registrador rA.
- Qualquer sub-rotina pode alterar o conteúdo dos registradores de qualquer maneira, exceto pela obrigação de devolver o registrador rSP ao estado em que se encontrava no início da chamada. Assim, após uma chamada não sabemos nada sobre o estado de nenhum dos registradores, exceto do registrador rSP.

Por exemplo, a seguinte sub-rotina calcula o fatorial de um número.

Sub-rotina 5. Calcula o fatorial do número em $0 e o coloca em rA.

```
27 #5aff0001 SETW  rA,1
28 #4a000004 JZ    $0,4
29 #34ffff00 MULU  rA,rA,$0
30 #33000001 SUBUI $0,$0,1
31 #49000003 JMPB  3
32 #560a0000 GO    Ret,0  ▌
```

Os números de linha acima não começam do 1 pois a sub-rotina será usada como parte de um programa maior, que lê um número da entrada padrão, calcula seu fatorial e escreve o resultado na saída padrão. As sub-rotinas que compõem esse programa serão apresentadas separadamente; a numeração das linhas indicará como os diferentes trechos devem ser juntados.

Assim, o seguinte programa calcula 10! usando a sub-rotina acima, que foi copiada abaixo nas linhas 6–11:

```
1  #5a00000a  SETW   $0,10
2  #580a0002  GETA   Ret,2
3  #48000002  JMP    3
4  #fedbffff  INT    #DBFFFF    Mostra 10! na saída padrão.
5  #fe000000  INT    0
6  #5aff0001  SETW   rA,1
7  #4a000004  JZ     $0,4
8  #34ffff00  MULU   rA,rA,$0
9  #33000001  SUBUI  $0,$0,1
10 #49000003  JMPB   3
11 #560a0000  GO     Ret,0  ▌
```

Quando a instrução da linha 4 é executada, o registrador rA contém 10!. Note que, antes de chamar a sub-rotina com o desvio da linha 3, o endereço de retorno é guardado no registrador Ret através da instrução da linha 2.

Nosso programa deverá imprimir algum texto na tela, para ser amigável com o usuário. A seguinte sub-rotina escreve uma seqüência de caracteres na saída padrão.

Sub-rotina 6. Escreve na saída padrão a seqüência de caracteres cujo primeiro byte se encontra na posição de memória contida em $0. A escrita acaba quando um byte 0 é encontrado.

```
33 #5afc0002  SETW   rX,2
34 #01fb0000  LDBI   rY,$0,0
35 #4afb0004  JZ     rY,4
36 #fe000080  INT    #80
37 #31000001  ADDUI  $0,$0,1
38 #49000004  JMPB   4
39 #560a0000  GO     Ret,0  ▌
```

Faltam-nos sub-rotinas para ler um número da entrada padrão e para escrever um número na saída padrão. Nós já temos os programas 3 e 4; transformá-los em sub-rotinas é simples.

Sub-rotina 7. Lê da entrada padrão um número escrito na base 10 e o armazena no registrador rA.

```
40 #44000000  XOR  $0,$0,$0
   ⟨Linhas 2–11 do Programa 3⟩
51 #43ff0000  ORI  rA,$0,0    Copia resultado para rA.
52 #560a0000  GO   Ret,0  ▌
```

Sub-rotina 8. Escreve na saída padrão, na base 10, o número sem sinal contido no registrador $0.

```
53 #5a010001  SETW  $1,1
   ⟨Linhas 3–16 do Programa 4⟩
68 #560a0000  GO    Ret,0  ▌
```

Agora temos todas as peças de que precisamos para escrever nosso programa. O código principal chama as sub-rotinas acima para fazer o trabalho.

Programa 9. Lê um número da entrada padrão, calcula seu fatorial e exibe o resultado na saída padrão.

```
1  #48000007  JMP    7
2  #44696769  "Digite␣n:␣"
3  #7465206e
4  #3a200000
```

```
5  #52657370  "Resposta:␣"
6  #6f737461
7  #3a200000
8  #59000006  GETAB  $0,6
9  #580a0002  GETA   Ret,2
10 #48000017  JMP    23         Escreve "Digite␣n:␣".
11 #580a0002  GETA   Ret,2
12 #4800001c  JMP    28         Lê número.
13 #4300ff00  ORI    $0,rA,0
14 #580a0002  GETA   Ret,2
15 #4800000c  JMP    12         Calcula fatorial.
16 #17fffd00  STOUI  rA,rSP,0
17 #5900000c  GETAB  $0,12
18 #580a0002  GETA   Ret,2
19 #4800000e  JMP    14         Escreve "Resposta:␣".
20 #0700fd00  LDOUI  $0,rSP,0
21 #580a0002  GETA   Ret,2
22 #4800001f  JMP    31         Escreve resultado.
23 #5afc0002  SETW   rX,2
24 #5afb000a  SETW   rY,10
25 #fe000080  INT    #80        Escreve quebra-de-linha.
26 #fe000000  INT    0
27 ⟨Sub-rotina 5; calcula o fatorial⟩
33 ⟨Sub-rotina 6; imprime uma seqüência⟩
40 ⟨Sub-rotina 7; lê um número⟩
53 ⟨Sub-rotina 8; imprime um número⟩ ▌
```

Na linha 15 chamamos a sub-rotina que calcula o fatorial. Quando a sub-rotina retorna, a execução passa para a linha 16; nesse momento, o registrador rA contém o fatorial a ser calculado. Em seguida, queremos escrever a seqüência `"Resposta: "` na saída padrão e para fazê-lo chamamos outra sub-rotina. Pela nossa convenção de chamada, não podemos supor que o valor do fatorial, contido em rA antes da chamada, tenha permanecido lá. Por isso guardamos o valor de rA na memória (linha 16) para depois recuperá-lo (linha 20).

Um problema da nossa convenção de chamada é que usamos o registrador Ret para guardar o endereço de retorno. Assim, uma sub-rotina não pode ela mesma chamar outra sub-rotina, ou vai sobrescrever o conteúdo de Ret. A convenção de chamada que veremos na §16 contorna esse problema usando uma pilha para guardar dados locais de uma sub-rotina bem como o endereço de retorno.

Escrever programas mais longos como o acima em linguagem de máquina é tedioso. Traduzir cada instrução para linguagem de máquina é uma tarefa propensa a erros, mas talvez o mais complicado seja calcular o número de passos de cada desvio `JMP` que chama uma sub-rotina. Um *montador* ou *assembler* é um programa que traduz a representação simbólica das instruções, como por exemplo '`ADD $0,$1,$2`', para linguagem de máquina. Além disso, o montador auxilia o programador de outras formas, permitindo por exemplo que o programador associe rótulos a instruções de modo a poder referir-se a elas mais facilmente depois em operações de desvio como `JMP`; calcular o número de passos do desvio passa a ser tarefa do montador.

O processo de juntar trechos de código de diferentes sub-rotinas, que fizemos manualmente na montagem do programa acima, é tarefa do *vinculador* ou *linker*. Um programa maior pode ser composto de diversas sub-rotinas, colocadas em arquivos diferentes. O montador gera o *código-objeto* de cada sub-rotina, gerando uma quase linguagem de máquina, e o vinculador faz a ligação dos diversos pedaços num programa completo. Montadores e vinculadores serão discutidos mais adiante.

§14. Um simulador

`MACSim` é um simulador para o `MAC216` desenvolvido para rodar em computadores com arquitetura little-endian, como a maior parte dos computadores modernos. `MACSim` simula programas na linguagem de máquina do `MAC216` e também faz as vezes de um sistema operacional primitivo, carregando o programa e disponibilizando operações básicas de entrada e saída (veja §8).

O programa em linguagem de máquina é escrito num arquivo texto comum. Cada instrução é composta de 8 dígitos hexadecimais, que fornecem o opcode e os operandos X, Y e Z. Espaços em branco antes de uma instrução são desconsiderados, bem como qualquer coisa que apareça após o último dígito da instrução. Linhas em branco também são desconsideradas.

Assim, o Programa 1, que escreve “Hello world!” na saída padrão, consiste por exemplo do seguinte arquivo de texto:

```
48000005    JMP    3
48656C6C
6F20776F
726C6421
0A000000
5AFC0002    SETW   rX,2
59000005    GETAB  $0,5
01FB0000    LDBI   rY,$0,0
4AFB0004    JZ     rY,4
FE000080    INT    #80
31000001    ADDUI  $0,$0,1
49000004    JMPB   4
FE000000    INT    0
```

Só o que importa para o `MACSim` são os números hexadecimais com 8 dígitos cada que ocorrem no início de cada linha; todo o resto é ignorado. Após cada instrução de máquina é portanto possível escrever qualquer coisa; acima temos a representação simbólica de cada instrução, para fácil compreensão do programa.

Se o arquivo de texto acima se chama ‘`hello.mac`’, então para executar o programa escrevemos

```
> macsim hello.mac
```

na linha de comando. Podemos também passar argumentos de linha de comando ao programa que será simulado, colocando tais argumentos após o nome do arquivo que contém o programa; veremos um exemplo na §19.

A simulação da mémoria do `MAC216` é a parte mais complicada do `MACSim` (não que seja assim tão complicada). O `MACSim` traduz os endereços virtuais de memória do `MAC216` em endereços reais de memória no computador que faz a simulação, de forma transparente. Não é possível, entretanto, acessar todo o espaço de endereçamento virtual do `MAC216`.

§15. Linguagem de montagem

Escrever programas em linguagem de máquina, como fizemos até agora, é tedioso e propenso a erros. A *linguagem de montagem* (*assembly language*) `MACAL` é uma linguagem simbólica simples para a especificação de programas. Um programa escrito em `MACAL` pode ser transformado em código de máquina usando-se o *montador* (*assembler*) `MACAs` e o *vinculador* (*linker*) `MACLk`. Nas seções seguintes, essas etapas serão descritas em detalhes.

Vejamos um primeiro exemplo, uma implementação do algoritmo de Euclides:

Algoritmo 10 (*Algoritmo de Euclides*). Calcula o máximo divisor comum entre dois inteiros $m, n > 0$.

1. $a \leftarrow n \bmod m$.

2. Se $a = 0$, então m divide n e o máximo divisor comum é m.

3. Se não, então faça $n \leftarrow m$ e $m \leftarrow a$ e vá para o passo 1. ▌

O seguinte programa em `MACAL` calcula o máximo divisor comum entre os inteiros sem sinal contidos nos registradores \$0 e \$1, guardando o resultado no registrador \$0.

```
1             m       IS    $0
2             n       IS    $1
3  #36fc0100  euclid  DIVU  rX,n,m
4  #4aff0004          JZ    rR,end
5  #43010000          OR    n,m,0
6  #4300ff00          OR    m,rR,0
7  #49000004          JMP   euclid
8  #fe000000  end     INT   0        $0 = mdc(m,n).  ▌
```

Acima, temos seis colunas. A primeira mostra a numeração das linhas para referência no texto e não faz parte do programa. A segunda coluna traz a instrução de máquina que corresponde à instrução `MACAL` da linha em questão; a instrução de máquina também não faz parte do programa. A terceira traz *rótulos*, que podem estar presentes ou não. A quarta coluna contém um *operador*, a quinta coluna uma *expressão* e a sexta comentários, que no programa são precedidos por um asterisco '`*`', omitido acima.

Cada linha do programa contém uma instrução da `MACAL`; uma instrução tem sempre o seguinte formato:

'`RÓTULO  OPERADOR  EXPRESSÃO`',

sendo o rótulo opcional (a não ser para o pseudo-operador `IS`; veja abaixo).

Com exceção do operador `IS`, todo operador que aparece acima corresponde a uma operação do `MAC216`. O operador `IS` é um *pseudo-operador*, que representa uma operação da linguagem de montagem que não corresponde diretamente a uma operação da linguagem de máquina. O pseudo-operador `IS`, por exemplo, define um *apelido* ou *alias*. Após seu uso na linha 1, por exemplo, o nome '`m`' passa a representar o registrador \$0 no restante do programa; nenhum código de máquina é gerado para as linhas 1 e 2 do programa.

O rótulo '`m`', usado na linha 1, é associado pelo pseudo-operador `IS` ao registrador \$0. Em toda outra ocasião, um rótulo representa o endereço da instrução na qual ocorre. Por exemplo, a instrução da linha 3 tem o rótulo '`euclid`', usado na linha 7 com o operador `JMP`.

Um rótulo é constituído por uma seqüência de letras, números e caracteres underscore e deve começar sempre com uma letra ou um underscore. Maiúsculas e minúsculas são consideradas diferentes. Rótulos devem ser definidos apenas uma vez, ou seja, podem aparecer apenas uma vez na primeira coluna. Rótulos não podem ter o mesmo nome de um operador.

Uma expressão é uma lista de operandos, separados por vírgulas. Cada operando pode ser:

- Um rótulo que pode referir-se a uma instrução ou a um apelido;
- Um registrador, representado por um cifrão seguido de um número entre 0 e 255 (e.g., `$10`), ou o nome de um registrador especial como `rA`;
- Um número positivo ou negativo em notação decimal (e.g., `-10`) ou hexadecimal (e.g., `#FF`);

- Uma seqüência de caracteres envolta por aspas duplas, e.g. `"Hello␣world!"`. Aspas duplas podem ser colocadas dentro de uma seqüência de caracteres quando precedidas pela barra invertida; a quebra-de-linha é representada pela seqüência '`\n`'.

Diferentes operadores requerem operandos diferentes. A expressão fornecida ao operador `DIVU`, por exemplo, pode conter 3 registradores, como na linha 3, ou o último registrador pode ser substituído por um número, caso em que a instrução é montada como a operação `DIVUI`, a versão imediata da operação `DIVU`, do `MAC216`. Isso ocorre na linha 5 do programa acima com o operador `OR`, que no caso corresponde à operação `ORI` do `MAC216`.

O operador `JZ` requer um registrador e um rótulo que especifica a instrução para a qual o fluxo é desviado caso o conteúdo do registrador seja 0. O montador calcula o número de instruções para o desvio e decide se o desvio deve ser para frente (operação `JZ` do `MAC216`) ou para trás (operação `JZB`). Em vez do rótulo, pode ser dado o deslocamento, na forma de um número positivo (desvio para frente) ou negativo (desvio para trás).

Todas as operações do `MAC216` correspondem a operadores da `MACAL`, exceto que versões imediatas ou desvios para trás são deduzidos automaticamente e portanto não estão presentes (i.e., não há operador `ORI` em `MACAL`, por exemplo). Há ainda alguns pseudo-operadores como `IS`; veremos mais sobre eles nas próximas seções.

Exercício

1. Escreva, em linguagem de montagem, o Programa 9 da §13.

§16. Sub-rotinas e convenção de chamada

A convenção de chamada que adotaremos é muito parecida com aquela usada por compiladores para a linguagem C. Uma sub-rotina *recebe* zero ou mais argumentos e, opcionalmente, *devolve* algum valor a quem a chamou. Os argumentos e o valor devolvido são sempre octas.

Para chamar uma sub-rotina, primeiro colocamos os argumentos na pilha, na ordem inversa, de modo que o primeiro argumento fique no topo. Depois guardamos no topo da pilha o endereço de retorno e desviamos para a sub-rotina.

Por exemplo, uma sub-rotina `euclid` que calcula o máximo divisor comum entre dois números recebe números m e n e devolve o máximo divisor comum entre eles. Se $\$0 = m$ e $\$1 = n$, então para chamá-la fazemos algo como

```
STOU  $1,rSP,0
STOU  $0,rSP,8
GETA  rZ,4
STOU  rZ,rSP,16          (1)
ADDU  rSP,rSP,24
JMP   euclid
```

Colocar valores na pilha é uma operação comum. Por isso, a `MACAL` disponibiliza o pseudo-operador `PUSH`. Uma instrução '`PUSH $X`' é montada como duas instruções do `MAC216`:

```
STOUI  $X,rSP,0
ADDUI  rSP,rSP,8
```

Quando duas ou mais instruções `PUSH` são usadas em seguida, o montador utiliza menos instruções de máquina, atualizando rSP apenas uma vez ao final.

Para chamar uma sub-rotina também temos que colocar o endereço de retorno na pilha e fazer um desvio. É isso que faz o pseudo-operador `CALL`. A instrução '`CALL rótulo`' é montada da seguinte forma:

```
GETA    rZ,4
STOUI   rZ,rSP,0
ADDUI   rSP,rSP,8
JMP[B]  k
```

Aqui, k é o desvio calculado pelo montador segundo o rótulo dado; a depender da posição do rótulo, a instrução montada será JMP ou JMPB. Assim, podemos reescrever (1):

```
PUSH  $1
PUSH  $0
CALL  euclid
```

A sub-rotina então faz o seu trabalho e usa o registrador rA para devolver algum valor, caso seja necessário (por isso rA é chamado de "return value register"). Antes de retornar, ela deve remover o endereço de retorno e os argumentos da pilha. A sub-rotina `euclid` faria algo como

```
SUBU  rSP,rSP,24
LDOU  rZ,rSP,16    Pega o endereço de retorno.
GO    rZ,0
```

Esse tipo de limpeza a ser feita pela sub-rotina também é algo comum. Por isso `MACAL` disponibiliza o pseudo-operador `RET`. A instrução '`RET` k', onde $k \geq 0$ é um número inteiro, remove k argumentos e o endereço de retorno da pilha e faz o desvio. Por exemplo, o trecho acima poderia ser escrito simplesmente como '`RET 2`'.

Finalmente, a única garantia que uma sub-rotina precisa dar a quem a chama é a de limpar a pilha antes de retornar, alterando o registrador rSP de forma apropriada. O conteúdo dos demais registradores pode ser mudado arbitrariamente. Por isso é útil usar instruções como `SAVE` e `REST` para guardar o conteúdo de registradores antes de uma chamada e recuperar os valores após o retorno.

Vejamos um primeiro exemplo completo: a sub-rotina `puts`, que recebe o endereço do primeiro caractere de uma seqüência de caracteres terminada em 0 e escreve a seqüência na saída padrão.

```
s      IS    $0
puts   SUBU  s,rSP,16   Carrega endereço do primeiro caractere.
       LDOU  s,s,0
       SETW  rX,2
write  LDB   rY,s,0     Carrega caractere atual.
       JZ    rY,end
       INT   #80
       ADDU  s,s,1      Pula para o próximo caractere.
       JMP   write
end    RET   1  ▌
```

Essa sub-rotina é utilizada pelo seguinte programa que escreve "Hello world!" na saída padrão:

```
1  #48656c6c  hello  STR   "Hello␣World!\n"
2  #6f20576f
3  #726c6421
4  #0a000000
5  #59000004  main   GETA  $0,hello
6  #1f00fd00         PUSH  $0
7  #31fdfd08
8  #58fa0004         CALL  puts
9  #1ffafd00
10 #31fdfd08
11 JMP[B] k
12 #fe000000         INT   0  ▌
```

Atente para o código de máquina gerado para cada instrução da MACAL. Na linha 1, o pseudo-operador STR é usado para montar os códigos ASCII da seqüência de caracteres dada entre aspas, terminando com um byte 0 — e quantos outros forem necessários para completar um tetrabyte. Cabe ainda observar que a instrução a ser montada na linha 11 depende de onde a sub-rotina puts é colocada. Finalmente, o rótulo main usado na linha 5 é especial: ele indica onde o programa deve começar. Decidir o local onde cada sub-rotina deve ficar e fazer o programa começar no lugar certo é o trabalho do vinculador, como veremos na seção seguinte.

§17. Vinculação

No momento, está um tanto confuso como fazemos a ligação de todas as sub-rotinas num programa completo. Fizemos isso manualmente na §13; nesta seção veremos como o vinculador MACLk pode ser usado para fazer esse trabalho por nós.

É sempre uma boa idéia dividir um programa grande em diversos arquivos-fonte. Cada arquivo deve conter algumas poucas sub-rotinas e um deles deve conter a sub-rotina main, a primeira a ser executada.

Cada arquivo é processado separadamente pelo montador MACAs, que gera um arquivo com o *código-objeto* correspondente. O código-objeto é um quase código de máquina; apenas instruções de desvio que fazem referência a rótulos não definidos no mesmo arquivo são deixadas sem montar, mais ou menos como aconteceu no exemplo da seção anterior, quando não sabíamos calcular o desvio na linha 11 por não saber onde estava a sub-rotina puts.

Depois, o vinculador é chamado para ligar diferentes arquivos de código-objeto. Ele coloca os trechos de código de máquina num arquivo executável, resolvendo referências a rótulos deixadas pelo montador. Se algum rótulo não foi definido em nenhum arquivo dado ao vinculador, então o processo falha.

Vejamos como esse processo funciona na prática. A sub-rotina puts pode ser colocada num arquivo chamado 'puts.as':

```
        EXTERN  puts
s       IS      $0
puts    SUBU    s,rSP,16   Carrega endereço do primeiro caractere.
        LDOU    s,s,0
        SETW    rX,2
write   LDB     rY,s,0     Carrega caractere atual.
        JZ      rY,end
        INT     #80
        ADDU    s,s,1      Pula para o próximo caractere.
        JMP     write
end     RET     1  ▌
```

O pseudo-operador EXTERN, que aparece na primeira linha do programa acima, informa ao montador que o rótulo 'puts' deve ser exportado, podendo ser referenciado em outros arquivos. O montador vai incluir essa informação no código-objeto. Sem a instrução 'EXTERN puts', o rótulo 'puts' poderia ser usado apenas de dentro do arquivo 'puts.as'. Rótulos declarados com EXTERN devem ser únicos: arquivos diferentes não podem exportar o mesmo nome.

A rotina principal, que imprime "Hello world!", pode ser colocada num arquivo chamado 'hello.as':

```
        EXTERN  main
hello   STR     "Hello␣World!\n"
main    GETA    $0,hello
        PUSH    $0
        CALL    puts
```

```
        INT     0
```

O rótulo '`main`' deve aparecer em exatamente um dos arquivos que constituem o programa e deve ser exportado usando `EXTERN`. O vinculador vai fazer com que a primeira instrução a ser executada seja aquela da linha com o rótulo '`main`'.

Para gerar o código de máquina executável, precisamos usar o montador para gerar o código-objeto correspondente a ambos os arquivos acima:

```
> macas puts.as
> macas hello.as
```

O primeiro comando faz com que o arquivo '`puts.maco`', contendo o código-objeto para a sub-rotina `puts`, seja gerado. O segundo comando faz com que o arquivo '`hello.maco`' seja gerado. Estes são arquivos de texto, semelhantes a arquivos de entrada para o simulador `MACSim`. É interessante ver e analisar seu conteúdo.

Para gerar o programa executável, usamos o vinculador:

```
> maclk hello.mac puts.maco hello.maco
```

Esse comando cria o arquivo '`hello.mac`', que pode ser dado como entrada para o simulador `MACSim`.

Exercícios

1. Escreva uma sub-rotina `readnum` que lê um número inteiro, na base 10, da entrada padrão e o devolve. Sua sub-rotina deve ignorar espaços em branco e quebras-de-linha antes do primeiro caractere do número.

2. Escreva uma sub-rotina `putnum` que recebe um número inteiro e o escreve, na base 10, na saída padrão.

3. Escreva uma sub-rotina `atoi` que recebe o endereço de uma seqüência de caracteres terminada em 0 e devolve o número inteiro, escrito na base 10, contido na seqüência dada. Sua sub-rotina deve ignorar espaços em branco antes do primeiro caractere do número e deve parar assim que o primeiro caractere não-numérico for encontrado.

4. Escreva uma sub-rotina `printf` que se comporta como a função de mesmo nome da biblioteca padrão da linguagem C. Sua implementação deve aceitar os códigos de formato `%d`, que é substituído por um número, e `%s`, que é substituído por uma seqüência de caracteres. Por exemplo, o trecho de código abaixo escreve `"Nome:␣Jose.␣Nota:␣5"` na saída padrão:

```
msg   STR    "Nome:␣%s.␣Nota:␣%d"
name  STR    "Jose"
      SETW   $0,5
      PUSH   $0
      GETA   $0,name
      PUSH   $0
      GETA   $0,msg
      PUSH   msg
      CALL   printf
```

A função `printf` da biblioteca padrão pode receber um número qualquer de argumentos. Observe que os argumentos da sub-rotina são empilhados do último para o primeiro. Assim, no topo da pilha, logo abaixo do endereço de retorno, estará o endereço da seqüência de caracteres com os códigos de formatação. Dessa forma é possível saber quantos outros argumentos foram empilhados.

§18. O algoritmo de Euclides com sub-rotinas

Vejamos um exemplo completo: um programa que lê dois inteiros m, $n \geq 0$ da entrada padrão e escreve o máximo divisor comum na saída padrão.

Começamos com uma sub-rotina que recebe números m, $n \geq 0$ e devolve o máximo divisor comum:

```
m IS rA; n IS $0; bp IS $1
        EXTERN  euclid
euclid  SUBU    bp,rSP,24   Carrega argumentos.
        LDOU    n,bp,0
        LDOU    m,bp,8
loop    DIVU    rX,n,m      Calcula mdc.
        JZ      rR,end
        OR      n,m,0
        OR      m,rR,0
        JMP     loop
end     RET     2  ▌
```

Observe aqui outra característica da linguagem `MACAL`: podemos colocar diversas instruções numa só linha, separando-as por ponto-e-vírgula.

Nosso programa também usa as sub-rotinas **`readnum`** e **`putnum`** dos exercícios da seção anterior. O programa principal simplesmente chama as sub-rotinas para ler dois números e escrever o máximo divisor comum:

```
      EXTERN  main
main  CALL    readnum
      PUSH    rA
      CALL    readnum
      PUSH    rA
      CALL    euclid
      PUSH    rA
      CALL    putnum
      SETW    rX,2      Quebra-de-linha.
      SETW    rY,10
      INT     #80
      INT     0  ▌
```

§19. Crivo de Eratóstenes e linha de comando

Nesta seção escreveremos o programa '`primes.mac`', que imprime todos os números primos até um dado $n \geq 2$. Uma novidade em relação aos exemplos anteriores é que o número n é dado na linha de comando e não lido da entrada padrão; a invocação

```
> macsim primes.mac 1000
```

imprime todos os números primos até 1000.

Para encontrar todos os primos até um dado número utilizaremos o *crivo de Eratóstenes*:

Algoritmo 11 (*Crivo de Eratóstenes*). Encontra todos os números primos positivos menores ou iguais a um dado $n \geq 2$. O algoritmo mantém um vetor v indexado pelos números $2, \ldots, n$ que marca quais são os primos: ao final do algoritmo temos $v[k] = 0$ se e somente se k é primo. O algoritmo começa por marcar todos os múltiplos de 2 como não-primos, depois todos os múltiplos de 3, depois todos os múltiplos de 5, e assim por diante.

1. Faça $v[k] \leftarrow 0$ para $k = 2, \ldots, n$ e $p \leftarrow 2$.

2. Enquanto $p \leq n$ e $v[p] \neq 0$, faça $p \leftarrow p + 1$.

3. Se $p > n$, o algoritmo termina. Caso contrário, faça $v[kp] \leftarrow 1$ para todo $k \geq 2$ tal que $kp \leq n$; faça $p \leftarrow p + 1$ e vá para o passo 2. ▌

A linha de comando passada quando `MACSim` é invocado é guardada na pilha de execução e o registrador rSP é incrementado para a primeira posição livre da pilha. Quando o programa começa, o octa guardado no endereço rSP $-$ 8 contém o número *argc* de argumentos de linha de comando dados ao programa. Note que, como de costume, o nome do programa é o primeiro argumento, logo sempre temos $argc \geq 1$. As posições de memória rSP $- 8(argc + 1), \ldots,$ rSP $-$ 16 contêm os endereços de memória das seqüências de caracteres, sempre terminas em 0, com cada um dos argumentos do programa.

Por exemplo, se temos a invocação

```
> macsim prog.mac A1 A2 A3
```

então $argc = 4$. A posição de memória rSP $-$ 40 contém o endereço da seqüência `"prog.mac"`, e assim por diante.

Tente escrever o programa '`primes.mac`'. Complicado? Como você o escreveria em C? Uma possibilidade seria a seguinte:

```
int main()
{
  int v[MAX];
  int n = 1000, p, k;
  for (k = 2; k ≤ n; k++) v[k] = 0;
  for (p = 2; p ≤ n; p++) {
    if (v[p]) continue;
    printf("%d\n", p);   /* p é primo */
    for (k = 2 * p; k ≤ n; k += p) v[k] = 1;
  }
  return 0;
}
```

O programa acima usa diversas construções da linguagem C que já estão bem longe da linguagem de montagem, como o comando **for**. Vamos reescrever o programa acima sem usar **for** ou **while**, mas mantendo a ordem das operações quase a mesma:

```
int main()
{
  int v[MAX];
  int n = 1000, p, k;
  p = k = 2;
init_v:
  if (k > n) goto sieve;
  v[k] = 0;
  k++;
  goto init_v;
sieve:
  if (p > n) goto end;
  if (v[p]) goto next_p;
  printf("%d\n", p);
  k = 2 * p;
mark:
  if (k > n) goto next_p;
  v[k] = 1;
  k += p;
  goto mark;
next_p:
  p++;
  goto sieve;
end:
  return 0;
}
```

Transformar o programa acima num programa em linguagem de montagem é agora quase imediato, exceto pelos comandos **if**, que podem requerer a criação de novos rótulos. Por exemplo, o trecho

```
if (a > b) a++;
b++;
```

pode ser representado pelo seguinte trecho em linguagem de montagem:

```
        CMPU  rY,a,b
        JNP   rY,a_leq_b
        ADDU  a,a,1
a_leq_b ADDU  b,b,1
```

O programa abaixo utiliza as sub-rotinas `printf` e `atoi` dos exercícios da §17. O programa usa 1 byte para representar cada entrada no vetor v do algoritmo; a entrada $v[k]$, para $k \geq 2$, fica guardada na posição de memória k, no segmento do heap.

Programa 12. Escreve na saída padrão todos os números primos positivos menores ou iguais a n, sendo n o número passado como primeiro argumento de linha de comando.

```
1  n IS $0; p IS $1; k IS $2;
2            EXTERN  main
3  fmt_str   STR     "%d\n"        Para chamada à printf.
4  main      SUBU    rX,rSP,16     Encontra número n.
5            LDOU    rX,rX,0
6            PUSH    rX
7            CALL    atoi
8            OR      n,rA,0
9            SETW    p,2
10           SETW    k,2
11           XOR     rX,rX,rX
12 init_v    CMPU    rY,k,n        Inicializa v.
13           JP      rY,sieve
14           STBU    rX,k,0
15           ADDU    k,k,1
16           JMP     init_v
17 sieve     CMPU    rY,p,n
18           JP      rY,end
19           LDBU    rY,p,0
20           JNZ     rY,next_p     p não é primo.
21           SAVE    rSP,n,p       p é primo; escreve p na saída.
22           GETA    rX,fmt_str
23           PUSH    p
24           PUSH    rX
25           CALL    printf
26           REST    rSP,n,p
27           SETW    rX,1          Para marcar múltiplos de p.
28           ADDU    k,p,p
29 mark      CMPU    rY,k,n        Marca múltiplos de p.
30           JP      rY,next_p
31           STBU    rX,k,0
32           ADDU    k,k,p
33           JMP     mark
34 next_p    ADDU    p,p,1         Pula para p + 1.
35           JMP     sieve
36 end       INT     0  ▌
```

Note como as linhas 11–36 são uma tradução quase exata do segundo programa em C acima. O que acontece se alguém invocar o programa '`primes.mac`' escrevendo '`macsim primes.mac -1`'?

Exercícios

1. A *seqüência de Collatz* com valor inicial $n_0 \geq 1$, sendo n_0 um número inteiro, é assim definida. O primeiro número da seqüência é n_0 e, dado um número n na seqüência, seu sucessor é calculado da seguinte forma: se n é par, então o sucessor é $n/2$; se n é ímpar, então o sucessor é $3n + 1$.

Se o número 1 ocorre em algum momento numa seqüência, então todo termo seguinte é igual a 1. Conjectura-se que, qualquer que seja o valor inicial, toda seqüência chegue ao número 1 em algum momento. Por exemplo, se $n_0 = 13$, então temos a seqüência

$$13, 40, 20, 10, 5, 16, 8, 4, 2, 1,$$

com 10 termos até o número 1 ser alcançado.

Escreva um programa que calcula o tamanho da maior seqüência de Collatz com valor inicial entre 1 e 10^6.

2. Implemente sub-rotinas `strlen`, `strcmp` e `strcat` que se comportam como as funções de mesmo nome da biblioteca padrão do C.

3. Defina uma palavra como uma seqüência maximal de não-espaços e como um espaço os caracteres com códigos ASCII 9 (tabulação), 10 (quebra-de-linha), 13 (carriage return) e 32 (espaço). Essa definição de palavra será usada nos exercícios seguintes.

Escreva um programa que lê um texto da entrada padrão e imprime o número de caracteres e palavras no texto.

4. Escreva um programa que lê uma linha da entrada padrão, ou seja, lê até o fim do arquivo ou uma quebra-de-linha ser alcançada, e imprime a mesma linha com as palavras em ordem reversa. Por exemplo, se a entrada é "Hello world", então seu programa deve imprimir "world Hello".

5. Escreva um programa que lê n e n números inteiros da entrada padrão e imprime os mesmos números colocados em ordem crescente. Tente usar um algoritmo bom de ordenação, como o mergesort.

6. Escreva um programa que lê um texto da entrada padrão e imprime, em ordem lexicográfica, as palavras do texto, de modo que cada palavra seja impressa apenas uma vez.

§20. Tipos de dados compostos

De agora em diante veremos alguns exemplos mais complicados de programas; se você quiser, pode deixar a leitura desta e das próximas seções para mais perto do fim do curso.

A linguagem C fornece-nos tipos de dados *primitivos*, como **int**, mas também nos permite criar tipos de dados compostos usando o comando **struct**. Por exemplo, uma temperatura pode ser representada pelo tipo

```
typedef struct {
  char unit;
  int T;
} Temp;
```

O campo *unit* contém um caractere que representa a unidade de temperatura, por exemplo '`F`' para Fahrenheit ou '`C`' para Celsius; o campo T guarda a temperatura em si.

Como fica guardada na memória uma variável do tipo **Temp**? Isto depende do tamanho de cada tipo dentro do tipo composto e também de fatores como alinhamento de memória. Por exemplo, uma variável do tipo **char** ocupa 1 byte de memória. Suponha que uma variável do tipo **int** ocupe 4 bytes e que por motivos de alinhamento o endereço de uma variável do tipo **int** seja sempre múltiplo de 4. Assim, se a variável x contém a temperatura -5C, ela fica guardada como:

#`100`	#`43??????`
#`104`	#`fffffffb`

O endereço de x, denotado em C por $\&x$, é #`100`. O campo *unit* começa no endereço #`100` e o campo T começa no endereço #`104`. Note que assim o campo *unit* ocupa 4 bytes na memória, dos quais apenas o primeiro é usado.

Assim, cada campo de uma struct como **Temp** equivale a um deslocamento (*offset*) em relação ao endereço base. Por exemplo, o campo *unit* equivale ao deslocamento 0, já o campo T equivale ao deslocamento 4. Se a é um endereço de memória, usamos a notação $unit(a)$ para representar o conteúdo do campo *unit* da variável do tipo **Temp** cujo endereço base é a.

É simples representar tipos de dados compostos em MACAL: precisamos apenas determinar o deslocamento de cada campo. Por exemplo, uma temperatura ocuparia um octabyte: o primeiro byte do primeiro tetrabyte teria a unidade e o segundo tetrabyte a temperatura em si. O seguinte programa escreve na saída padrão a temperatura cujo endereço base se encontra em \$0:

```
a     IS    $0
unit  IS    0
T     IS    4
      LDT   rX,a,T       Carrega a temperatura.
      SAVE  rSP,a,a
      PUSH  rX
      CALL  putnum
      REST  rSP,a,a
      SETW  rX,2
      LDBU  rY,a,unit    Carrega a unidade.
      INT   #80
      INT   0 ▌
```

Note como podemos usar apelidos para os nomes dos campos, tornando o programa mais claro.

Exercício

1. Suponha que um programa do departamento de trânsito precise guardar as seguintes informações sobre um carro: as três letras e os quatro dígitos da placa do carro, o ano de fabricação e o nome do dono, que é uma seqüência de no máximo 20 caracteres. Lembre-se de que, no MAC216, o endereço base de um wyde é um múltiplo de 2, de um tetra um múltiplo de 4 e de um octa um múltiplo de 8. Ordene os campos e forneça o deslocamento de cada campo de modo que o tipo de dado composto que guarda as informações acima ocupe o menor espaço possível na memória. Em seguida, ordene os campos de forma a fazer com que o mesmo tipo composto ocupe o maior espaço possível.

§21. Gerenciamento dinâmico de memória: o sistema de irmãos

Na §19 usamos o segmento do heap para guardar o vetor v. Diferentes partes de um programa mais complexo precisam guardar informações na memória, e torna-se cada vez mais difícil para o programador manter um mapa de que parte do programa acessa qual parte da memória. Por isso existem rotinas de gerenciamento de memória, que nos permitem reservar blocos de memória quando são necessários e liberá-los quando não mais precisamos deles.

Qualquer programa em C razoavelmente complexo usa as funções *malloc* e *free* da biblioteca padrão para o gerenciamento de memória. Veremos nesta seção e nas seguintes uma possível implementação dessas funções para o MAC216. Para acompanhar a exposição a seguir, você deve estar familiarizado com listas ligadas.

Há diversos métodos de gerenciamento de memória, cada um com vantagens e desvantagens. O método que vamos utilizar é o "sistema de irmãos" ou "buddy system". (A palavra *buddy* é mais próxima de companheiro ou camarada, mas estas notas tentam ficar o mais longe possível do português de uma assembléia do DCE.) A apresentação a seguir é baseada na descrição de Knuth [2], §2.5C.

Supomos que a região de memória a ser gerenciada tem 2^m bytes, com endereços entre 0 e $2^m - 1$. Para cada $0 \le k \le m$, mantemos uma lista duplamente ligada de

blocos livres de tamanho 2^k. Todas as listas começam vazias, exceto pela m-ésima lista, que contém o único bloco de tamanho 2^m.

Quando queremos reservar um bloco de memória de tamanho 2^k, procuramos pelo menor $j \geq k$ para o qual há pelo menos um bloco livre de tamanho 2^j. Removemos um tal bloco de sua lista e, se $j > k$, dividimo-lo em dois blocos de tamanho 2^{j-1} cada, colocamos um deles na lista de blocos livres de tamanho 2^{j-1}, fazemos $j \leftarrow j-1$ e repetimos o processo.

Os dois blocos que surgem da divisão de um bloco durante o algoritmo de reserva são chamados de *irmãos.* É fácil provar por indução (prove!) que o endereço inicial de todo bloco de tamanho 2^k é um múltiplo de 2^k. Em binário, um bloco de tamanho 32 tem um endereço como $xx \ldots x00000$, onde cada x representa um 0 ou 1. Quando ele é dividido, os blocos resultantes têm endereços $xx \ldots x00000$ e $xx \ldots x10000$. Assim, é fácil calcular o endereço do irmão de um bloco, dado o endereço do bloco: se $\text{irmão}_k(x)$ é o endereço do irmão do bloco de endereço x e tamanho 2^k, então

$$\text{irmão}_k(x) = \begin{cases} x + 2^k, & \text{se } x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{se } x \bmod 2^{k+1} = 2^k. \end{cases}$$

Pode-se então calcular o endereço do irmão de um bloco usando-se a operação `XOR` do `MAC216` (como?).

Para liberar um bloco, tentamos juntá-lo novamente com seu irmão, caso este também esteja livre, e repetimos essa operação enquanto possível, construindo assim o maior bloco livre que podemos.

Mais precisamente, para liberar um bloco de tamanho 2^k e endereço x, verificamos se seu irmão está livre. Para fazê-lo, precisamos guardar em cada bloco um bit de informação que nos diz se o bloco está livre ou não. Se o irmão está livre, temos duas possibilidades. Se o tamanho do irmão é diferente de 2^k, então o algoritmo termina e o bloco x é colocado na lista de blocos livres de tamanho 2^k. Se o tamanho do irmão é 2^k, então combinamos o bloco x com seu irmão e repetimos o processo com o novo bloco de tamanho 2^{k+1}. Durante a liberação, torna-se necessário saber o tamanho atual de um bloco; assim, precisamos guardar essa informação em cada bloco e atualizá-la durante divisões e uniões.

Com a descrição dada acima, você pode tentar implementar os algoritmos de reserva e liberação em `MACAL`, antes de o fazermos em detalhes nas seções seguintes.

§22. Algoritmos de reserva e liberação

Vejamos uma descrição mais detalhada dos algoritmos de reserva e liberação. Cada bloco, livre ou reservado, deve ter dois campos:

- um campo *tag*, que é igual a 0 se e somente se o bloco está reservado;
- um campo *kval* que guarda o valor k para o qual o tamanho do bloco é 2^k.

Guardamos blocos livres em listas duplamente ligadas. Assim, cada bloco livre deve ter dois outros campos:

- *next*, que contém o endereço do próximo bloco na lista;
- *prev*, que contém o endereço do bloco anterior na lista.

Para facilitar a remoção de um bloco de uma lista, vamos usar listas circulares com cabeças falsas (*dummy head*). Denote por $avail[k]$ o endereço da cabeça da lista de blocos livres de tamanho 2^k. Inicialmente, todas as listas, exceto uma, estão vazias: para $k \neq m$ temos

$$next(avail[k]) = prev(avail[k]) = avail[k].$$

A lista $avail[m]$ tem apenas um bloco de tamanho 2^m, que começa no endereço 0:

$$\begin{aligned}&tag(0) = 0, \quad kval(0) = m,\\ &next(0) = prev(0) = avail[m],\\ &next(avail[m]) = prev(avail[m]) = 0.\end{aligned}$$

Algoritmo 13 (*Reserva de bloco no sistema de irmãos*). O algoritmo encontra e reserva um bloco livre de tamanho 2^k ou indica falha.

1. [Encontra um bloco.] Seja j o menor inteiro tal que $0 \leq j \leq m$ para o qual $next(avail[j]) \neq avail[j]$, i.e., para o qual a lista dos blocos livres de tamanho 2^j não esteja vazia. Se um tal j não existe, o algoritmo termina sem sucesso.

2. [Remove bloco da lista.] Faça $L \leftarrow next(avail[j])$, $P \leftarrow next(L)$, $next(avail[j]) \leftarrow P$, $prev(P) \leftarrow avail[j]$, $tag(L) \leftarrow 0$ e $kval(L) \leftarrow k$.

3. [Precisa dividir?] Se $j = k$, o algoritmo termina devolvendo o bloco encontrado.

4. [Faz divisão.] Faça $j \leftarrow j - 1$, $P \leftarrow L + 2^j$, $tag(P) \leftarrow 1$, $kval(P) \leftarrow j$, $next(P) \leftarrow prev(P) \leftarrow avail[j]$, $next(avail[j]) \leftarrow prev(avail[j]) \leftarrow P$. (Isto divide o bloco em dois e coloca a metade que ficará sem uso na lista $avail[j]$, que estava até então vazia.) Volte ao passo 3. ▌

Algoritmo 14 (*Liberação de bloco no sistema de irmãos*). Libera um bloco dado seu endereço inicial L.

1. $k \leftarrow kval(L)$.

2. [O irmão está disponível?] Faça $P \leftarrow \text{irmão}_k(L)$. Se $k = m$ ou se $tag(P) = 0$, ou se $tag(P) \neq 0$ e $kval(P) \neq k$, vá para o passo 4.

3. [Combina bloco com irmão.] Faça $next(prev(P)) \leftarrow next(P)$ e $prev(next(P)) \leftarrow prev(P)$, removendo assim o bloco P da lista $avail[k]$. Faça $k \leftarrow k+1$ e, se $P < L$, então faça $L \leftarrow P$. Vá para o passo 2.

4. [Coloca bloco na lista.] Faça $tag(L) \leftarrow 1$, $kval(L) \leftarrow k$, $P \leftarrow next(avail[k])$, $next(L) \leftarrow P$, $prev(P) \leftarrow L$, $prev(L) \leftarrow avail[k]$, $next(avail[k]) \leftarrow L$. ▌

§23. Decisões de implementação

Antes de implementar o sistema de irmãos, precisamos tomar algumas decisões. Por exemplo, onde guardar as listas ligadas de blocos livres? Qual região da memória gerenciar? Como guardar as informações adicionais em cada bloco?

Os algoritmos descritos acima supõem que gerenciamos o intervalo de endereços 0, …, $2^m - 1$. Isso não é essencial, mas facilita a implementação dos algoritmos. Ademais, o segmento do heap começa no endereço 0, então parece justo que seja esse o intervalo que gerenciamos. Para determinar m, temos de chamar o sistema operacional: a interrupção de código #`70` devolve o maior número de bytes que podemos utilizar no heap; veja §25.

Os campos *tag* e *kval* ocupam um byte cada em cada bloco, livre ou reservado. Quando o usuário pede pelo endereço de memória de um bloco livre, é uma boa idéia sempre devolver um endereço múltiplo de 8 (o que poderia dar errado caso contrário?). Dessa forma, para guardar os campos *tag* e *kval* gastamos um octabyte inteiro, embora seis dos bytes sejam desperdiçados. Assim, todo bloco deve ter pelo menos dois octabytes: um octabyte para os campos *tag* e *kval* e um octabyte que pode ser usado.

Blocos livres também têm campos *next* e *prev*. Se usamos um octabyte para cada um desses campos, precisamos de pelo menos três octabytes por bloco livre. Como o tamanho de um bloco deve ser uma potência de dois, isso significa que um bloco deve ter no mínimo quatro octabytes.

Não precisamos, entretanto, usar um octabyte para cada um desses campos. Se usamos um tetrabyte para cada um deles, podemos acessar os endereços 0, ..., $2^{32}-1$, o que já é bastante, e precisamos de apenas um octabyte a mais por bloco livre. Assim, o tamanho mínimo de um bloco deve ser de dois octabytes, uma potência de 2 como queremos.

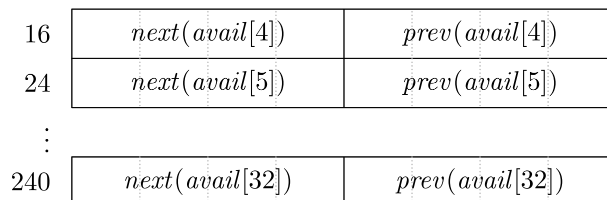
A estrutura de um bloco com endereço `#100` fica então assim:

Dessa forma, os deslocamentos dos campos são:

$$\begin{array}{ll} tag: 0, & next: 8, \\ kval: 1, & prev: 12. \end{array}$$

Note que os campos *next* e *prev* só são utilizados quando o bloco está livre. Se o bloco é reservado, então o primeiro byte que pode ser utilizado por quem o reservou é o que está na posição `#108`.

Ainda resta decidir onde e como guardar a cabeça de cada lista. Os únicos campos importantes em cada cabeça são *next* e *prev*; estes ficam guardados da seguinte forma:



Portanto, em geral o campo *next* da cabeça da lista $avail[k]$ fica no tetrabyte na posição $8k-16$ e o campo *prev* fica no tetrabyte seguinte. Dados os deslocamentos apresentados acima, temos então que

$$avail[k] = 8k - 24.$$

Assim, para manter as listas de blocos livres usamos os bytes de endereço 16, ..., 247. Você pode estar pensando que essa não é uma boa idéia. As listas ficam na mesma área de memória que gerenciamos: como podemos impedir que um usuário reserve um bloco que contém parte dos endereços de memória que usamos para manter as listas? Há uma solução simples para esse problema: o algoritmo de reserva só altera o primeiro octabyte do bloco que é reservado. Assim, logo após inicializar as listas, podemos reservar um bloco de tamanho 256. Esse bloco terá endereço inicial 0 e a partir daí os endereços nos quais as cabeças das listas residem estarão reservados.

§24. Implementação do algoritmo de reserva

Vejamos uma implementação da sub-rotina `malloc`, que recebe o tamanho n, em bytes, de um bloco a ser alocado e devolve o endereço $a \neq 0$ do primeiro octabyte livre de um bloco que contém pelo menos os bytes $a, \ldots, a+n-1$. Caso um tal bloco não esteja disponível, `malloc` devolve 0.

Começamos com uma implementação do Algoritmo 13. O algoritmo tenta reservar um bloco de tamanho 2^k, $k \geq 4$. Caso obtenha sucesso, devolve o endereço do primeiro octabyte livre; caso contrário devolve 0.

```
1  tag IS 0; kval IS 1; next IS 8; prev IS 12;
2  k IS $0; j IS $1; a IS $2; L IS $3; P IS $4;
3  reserve  SUBU  j,k,1
4           SLU   a,j,3
5           SUBU  a,a,24       a ← avail[k − 1].
6  find_j   ADDU  j,j,1
7           CMPU  rX,j,32
8           JP    rX,failure   Não há bloco disponível.
9           ADDU  a,a,8
10          LDTU  L,a,next
11          CMPU  rX,L,a
12          JZ    rX,find_j
13          LDTU  P,L,next     Bloco encontrado; faz reserva.
14          STTU  P,a,next
15          STTU  a,P,prev
16          XOR   rY,rY,rY
17          STBU  rY,L,tag     Marca bloco como reservado.
18          STBU  k,L,kval     Guarda o tamanho final do bloco.
19          SETW  rY,1         Para marcar bloco livre.
20 split    CMPU  rX,j,k       Precisamos dividir o bloco?
21          JZ    rX,success
22          SUBU  j,j,1
23          SUBU  a,a,8
24          SETW  P,1          P ← L + 2^j.
25          SLU   P,P,j
26          ADDU  P,P,L
27          STBU  rY,P,tag     Adiciona P à avail[j].
28          STBU  j,P,kval
29          STTU  a,P,next
30          STTU  a,P,prev
31          STTU  P,a,next
32          STTU  P,a,prev
33          JMP   split
34 failure  XOR   rA,rA,rA     Devolve 0 em caso de falha.
35          RET   1
36 success  ADDU  rA,L,8       Devolve primeiro endereço livre de L.
37          RET   1  ▌
```

No algoritmo acima, a guarda $avail[j]$. Começamos com $j = k - 1$ na linha 3 pois no início de cada iteração do laço da linha 6 fazemos $j \leftarrow j + 1$. Assim, o primeiro valor de j considerado é $j = k$, como queremos.

A sub-rotina `malloc` recebe o número n de bytes que o usuário quer poder usar no bloco alocado. Ela calcula o menor $k \geq 4$ tal que um bloco de tamanho 2^k seja suficiente, ou seja, tal que $2^k - 8 \geq n$ (já que um octabyte é usado para os campos *tag* e *kval*). Em seguida, o algoritmo de reserva é utilizado.

```
1          EXTERN  malloc
2  nbytes  IS      rY
3  bsize   IS      rZ
4  malloc  SUBU    nbytes,rSP,16     Determina número de bytes a alocar.
5          LDOU    nbytes,nbytes,0
6          ADDU    nbytes,nbytes,8   Um octabyte para tag e kval.
7          SETW    k,4
8          SETW    bsize,16
9  find_k  CMPU    rX,nbytes,bsize
```

```
10         JNP     rX,reserve
11         ADDU    k,k,1
12         SLU     bsize,bsize,1
13         JMP     find_k ▌
```

§25. Inicialização

A sub-rotina `minit` inicializa o sistema, como explicado acima. Ela termina chamando a sub-rotina `malloc` para reservar os primeiros 256 bytes do segmento do heap, que é onde guardamos as cabeças das listas de blocos livres.

```
1  tag IS 0; kval IS 1; next IS 8; prev IS 12;
2  N IS rA; k IS $0; a IS $1; bsize IS $2;
3              EXTERN  minit
4  minit       INT     #70              N ← tamanho máximo do heap.
5              SETW    bsize,1          Prepara inicialização.
6              SLU     bsize,bsize,32   bsize ← 2^32.
7              SETW    k,32
8              SETW    a,232            a ← avail[32].
9  init_list   CMPU    rZ,k,4           Se k < 4, terminamos.
10             JN      rZ,end
11             CMPU    rZ,bsize,N       Se bsize > N, a lista fica vazia.
12             JP      rZ,init_empty
13             XOR     rX,rX,rX         Se não, temos m = k.
14             SETW    rY,1             Inicializamos o bloco.
15             STBU    rY,rX,tag
16             STBU    k,rX,kval
17             STTU    a,rX,next        Adiciona bloco à avail[k].
18             STTU    a,rX,prev
19             STTU    rX,a,next
20             STTU    rX,a,prev
21             XOR     N,N,N            Evita inicialização de bloco menor.
22             JMP     next_iter
23 init_empty  STTU    a,a,next         Inicializa lista vazia.
24             STTU    a,a,prev
25 next_iter   SUBU    k,k,1            Pula para próxima iteração.
26             SUBU    a,a,8
27             SRU     bsize,bsize,1
28             JMP     init_list
29 end         SETW    rX,240           Reserva primeiros 256 bytes.
30             PUSH    rX
31             CALL    malloc
32             RET     0 ▌
```

Na linha 4 usamos a interrupção de código `#70`, que faz com que o sistema operacional coloque no registrador rA o maior número de bytes que podemos usar no segmento do heap.

Em seguida, calculamos o maior m tal que $2^m \leq$ rA. Inicializamos as listas de modo que todas estejam vazias, exceto pela lista $avail[m]$, que contém o bloco de tamanho 2^m e endereço inicial 0.

A quantidade 240, usada na linha 29 para a chamada à `malloc`, não tem nada de especial e poderia ser trocada por qualquer número que fizesse com que `malloc` reservasse um bloco de tamanho 256.

Exercícios

1. Prove ou dê contra-exemplo: se nunca usamos o algoritmo de liberação, então após qual-

quer seqüência de reservas cada lista de blocos livres ou está vazia ou tem exatamente um bloco.

2. Mostre uma seqüência de operações que leva uma lista a ter mais de um bloco. Qual o maior número de blocos que a lista $avail[k]$ pode ter? Qual seqüência de operações leva a esse maior número de blocos?

3. Implemente uma sub-rotina `free` que recebe um endereço devolvido por uma chamada à `malloc` e libera a memória do bloco correspondente. Pode ser interessante alterar a sub-rotina `minit` para que guarde o expoente m do tamanho do bloco inicial.

4. Escreva um programa para gerar um mapa gráfico da memória após alguns usos de `malloc` e `free`. Faça alguns experimentos aleatórios com o seu programa.

5. Reescreva o Programa 12 da §19 para que use a sub-rotina `malloc` para reservar espaço para o vetor v, em vez de simplesmente usar o início do segmento to heap.

§26. Árvores de busca binária

O Exercício 19.6 pede-nos um programa que lê um texto texto da entrada padrão e imprime as palavras do texto em ordem lexicográfica, sendo que cada palavra deve ser impressa apenas uma vez.

Uma forma de resolver esse problema é primeiro guardar todas as palavras que ocorrem no texto, depois colocar as palavras em ordem lexicográfica e finalmente eliminar palavras duplicadas durante a impressão. Uma forma melhor é utilizar uma árvore de busca binária para guardar uma única cópia de cada palavra lida. Nesta seção veremos como representar uma árvore de busca binária simples (i.e., sem nenhum esquema de balanceamento) e também veremos uma implementação do algoritmo de inserção. Nenhum dos programas que estudamos até aqui beneficia-se pelo uso de um esquema de gerenciamento dinâmico de memória como o discutido nas seções acima; não é o caso das árvores de busca binária: implementá-las sem usar um tal esquema é bem mais complicado.

A descrição a seguir supõe que você conheça os conceitos básicos acerca de árvores binárias e árvores de busca binária. Cada nó interno de nossa árvore precisa guardar os endereços de seus filhos esquerdo e direito. As chaves de nossa árvore são seqüências de caracteres. Uma forma de guardá-las seria colocar em cada nó interno um campo que guarda o endereço de sua chave; esta forma de proceder está mais próxima daquilo que faríamos ao implementar árvores de busca binária numa linguagem como C. Nossa abordagem será outra: cada nó guardará a seqüência de caracteres que constitui sua chave; assim, há nós de diferentes tamanhos.

Resumindo, cada nó interno tem os seguintes campos:

- *left*, que guarda o endereço do nó que é a raiz da subárvore esquerda;
- *right*, que guarda o endereço do nó que é a raiz da subárvore direita;
- *key_f*, que guarda o primeiro caractere da chave; os demais caracteres são guardados nos bytes seguintes e a chave termina no primeiro byte 0, como sempre.

O endereço 0 em *left* ou *right* indica um nó externo. A árvore vazia é aquela cuja raiz é um nó externo, tendo portanto endereço 0.

O deslocamento dos campos é 0, 4 e 8, respectivamente. Um nó de endereço `#100` que contém a chave `"Hello"` ficaria assim:

#100				#104				#108					
	left				*right*			72	101	108	108	111	0

Os números a partir da posição `#108` são os códigos ASCII dos caracteres da chave, com um zero ao final. Qual é o tamanho do bloco que a sub-rotina `malloc` reservaria para guardar o nó acima?

Para inserir uma nova chave na árvore, usamos o seguinte algoritmo iterativo:

Algoritmo 15 (*Inserção iterativa em árvore de busca binária*). Insere a seqüência de caracteres *key* na árvore de busca binária de raiz de endereço r e devolve a raiz da nova árvore; se a seqüência já está na árvore, então a árvore não é alterada. A variável p guarda, durante a execução, o endereço do pai do nó a ser inserido. A variável c é igual a -1 se devemos inserir na subárvore direita ou 1 se devemos inserir na subárvore esquerda.

1. $p \leftarrow 0$, $c \leftarrow 0$, $r' \leftarrow r$.

2. [Podemos inserir?] Se $r = 0$, crie um nó *new* com a seqüência *key* e com filhos esquerdo e direito iguais a 0. Se $p = 0$, então devolva *new*. Caso contrário, se $c < 0$, então faça $\mathit{right}(p) \leftarrow \mathit{new}$; se $c > 0$, então faça $\mathit{left}(p) \leftarrow \mathit{new}$. Devolva r'.

3. [Esquerda ou direita?] Faça $c = -1$, 0, ou -1 se $\mathit{key_f}(r)$ for menor, igual ou maior que *key*. Se $c = 0$, encontramos a chave; devolva r'. Caso contrário, faça $p \leftarrow r$. Se $c < 0$, então faça $r \leftarrow \mathit{right}(r)$ (inserimos na subárvore direita); se $c > 0$, então faça $r \leftarrow \mathit{left}(r)$ (inserimos na subárvore esquerda). Vá para o passo 2. ▌

Vejamos como fica uma implementação do algoritmo acima em `MACAL`. A sub-rotina `bst_insert` recebe como argumentos o endereço da raiz r da árvore e a chave *key* a ser inserida e devolve a raiz da árvore após a inserção. Por exemplo, o programa abaixo cria uma árvore de busca binária com duas palavras:

```
r       IS      $0            Endereço da raiz.
        EXTERN  main
name1   STR     "Jose"
name2   STR     "Filipe"
main    CALL    minit
        XOR     r,r,r         Inicializa árvore como vazia.
        GETA    rX,name1
        PUSH    rX
        PUSH    r
        CALL    bst_insert    Insere "Jose".
        OR      r,rA,0
        GETA    rX,name2
        PUSH    rX
        PUSH    r
        CALL    bst_insert    Insere "Filipe".
        OR      r,rA,0
        INT     0 ▌
```

A implementação abaixo utiliza a sub-rotina `new_node` para criar um novo nó. Essa sub-rotina encontra-se descrita mais adiante; ela cria um novo nó com chave *key* e devolve seu endereço no registrador rA. Abaixo, usamos a sub-rotina `strcmp` do Exercício 19.2.

```
1  left IS 0; right IS 4; key_f IS 8;
2  r IS $0; key IS $1; p IS $2;
3  c IS $3; r_key IS $4; bp IS $5;
4                EXTERN  bst_insert
5  bst_insert    SUBU    bp,rSP,24
6                LDOU    r,bp,8
7                LDOU    key,bp,0
8                XOR     p,p,p
9  branch        JZ      r,insert_new
10               SAVE    rSP,r,key
```

```
11                 ADDU   r_key,r,key_f       Compara key_f(r) com key.
12                 PUSH   key
13                 PUSH   r_key
14                 CALL   strcmp              rA ← resultado da comparação.
15                 REST   rSP,r,key
16                 JZ     rA,ret_r            A chave foi encontrada; paramos.
17                 OR     p,r,0
18                 JN     rA,branch_right     Segue à direita.
19                 LDTU   r,r,left            Segue à esquerda.
20                 JMP    branch
21 branch_right    LDTU   r,r,right
22                 JMP    branch
23 insert_new      OR     c,rA,0              c ← resultado da comparação.
24                 SAVE   rSP,p,c
25                 CALL   new_node
26                 REST   rSP,p,c
27                 JZ     p,end               Devolve o novo nó.
28                 JN     c,insert_right      Devemos inserir na direita?
29                 STTU   rA,p,left           Insere na esquerda.
30                 JMP    ret_r
31 insert_right    STTU   rA,p,right
32                 JMP    ret_r
33 ret_r           SUBU   rA,rSP,16
34                 LDOU   rA,rA,0
35 end             RET    2
```

A sub-rotina `new_node` usada acima cria um novo nó com filhos esquerdo e direito iguais a 0 e com chave *key* e devolve o endereço do novo nó em rA. Usamos as sub-rotinas `strlen` e `strcat` do Exercício 19.2.

```
36 new_node   SAVE  rSP,key,key
37            PUSH  key
38            CALL  strlen
39            ADDU  rA,rA,9
40            PUSH  rA
41            CALL  malloc         Aloca espaço para o nó.
42            XOR   rX,rX,rX
43            STOU  rX,rA,left     left(rA) ← right(rA) ← 0.
44            REST  rSP,key,key
45            SAVE  rSP,rA,rA
46            ADDU  rX,rA,key_f
47            PUSH  key
48            PUSH  rX
49            CALL  strcat         Copia chave.
50            REST  rSP,rA,rA
51            RET   0  ▮
```

Note que para guardar um nó precisamos de 8 bytes para os endereços dos filhos, mais um byte para cada caractere da chave, mais um byte para o byte 0 que marca o fim da chave. Por isso na linha 39 somamos 9 ao comprimento da chave calculado por `strlen`. Os endereços dos filhos esquerdo e direito de um nó ocupam os dois tetrabytes de um octabyte. Assim, na linha 43 inicializamos ambos os campos com 0.

Exercícios

1. A sub-rotina `bst_insert` devolve a raiz da árvore modificada. Quando ela devolve uma raiz diferente da que recebe?

2. Resolva o Exercício 19.6 usando árvores de busca binária. Você vai precisar escrever uma sub-rotina que imprime o conteúdo de uma árvore de busca binária em ordem crescente (ou seja, uma sub-rotina que visita os nós da árvore em ordem).

3. Escreva uma sub-rotina que recebe o endereço da raiz de uma árvore de busca binária e libera a memória ocupada pela árvore, destruindo todos os seus nós.

4. Escreva uma sub-rotina que calcula a altura e o número de nós de uma árvore de busca binária.

§27. Notas históricas

A idéia de que o programa sendo executado fica guardado na memória junto aos dados sobre os quais opera parece-nos natural hoje em dia, mas nem sempre foi assim.

Um problema fundamental no início do Século XX era fornecer uma definição matemática rigorosa para o conceito de algoritmo. Em seu artigo clássico de 1937, Alan Turing [3] propôs uma tal definição na forma do que hoje chamamos de *máquina de Turing*. Uma máquina de Turing é um modelo matemático simples de computador, tão poderoso do ponto de vista computacional quanto qualquer computador moderno.

Hoje em dia, há uma separação entre o algoritmo que implementamos e o computador físico que o executa. Em contrapartida, uma máquina de Turing *é* um algoritmo, ou seja, pode-se dizer que o algoritmo em si esteja codificado em seu hardware. Para grande surpresa de seus contemporâneos, Turing observou que é possível construir uma *máquina universal*, que recebe como entrada a descrição de uma máquina de Turing M e a entrada x a ser dada a ela e simula a execução da máquina M em x. Assim, a máquina universal é distinta do programa que executa; o hardware é distinto do software.

Mesmo assim, programar os primeiros computadores era uma tarefa física. O programador tinha que fazer um recabeamento, ou seja, praticamente mudar o hardware para escrever um novo programa. A idéia de manter um programa na memória é devida a John von Neumann. Segundo Ulam [4]:

> [...] during 1944 and 1945, he [von Neumann] formulated the now fundamental methods of translating a set of mathematical procedures into a language of instructions for a computing machine. The electronic machines of that time (e.g., the Eniac) lacked the flexibility and generality which they now possess in the handling of mathematical problems. Speaking broadly, each problem required a special and different system of wiring, in order to enable the machine to perform the prescribed operations in a given sequence. Von Neumann's great contribution was the idea of a fixed and rather universal set of connections or circuits in the machine, a "flow diagram", and a "code" so as to enable a fixed set of connections in the machine to have the means of solving a very great variety of problems. While, a priori at least, the possibility of such an arrangement might be obvious to mathematical logicians, the execution and practice of such a universal method was far from obvious with the then existing electronic technology.

§28. Referências

[1] D.E. Knuth, *MMIX: A RISC Computer for the New Millennium*, The Art of Computer Programming, Volume 1, Fascicle 1, Addison-Wesley, Upper Saddle River, 2005.

[2] D.E. Knuth, *The Art of Computer Programming, Volume 1* (Third Edition), Addison-Wesley, Reading, 1997.

[3] A. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society, Series 2* 42 (1937) 230–265.

[4] S. Ulam, John von Neumann (1903–1957), *Bulletin of the American Mathematical Society* 64(3) (1958) 1–49.