3 Organização básica de computadores

Agora que já sabemos um pouco da codificação da informação na memória, vamos voltar a examinar o funcionamento básico de um computador. A Fig. 3.1 mostra o esquema simplificado de um computador típico, ilustrando a interligação entre 3 componentes: memória, processador e um bloco representando dispositivos de entrada e saída, tais como monitor de vídeo, teclado, impressora, discos, etc.

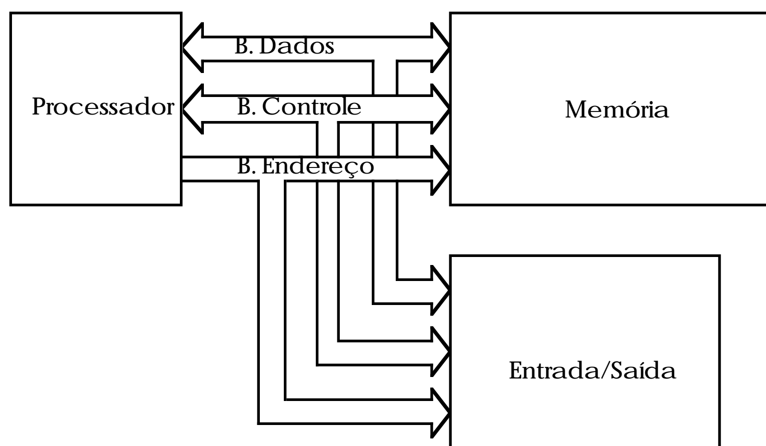


Fig. 3.1 Esquema simplificado de um computador típico.

Interligando os 3 componentes vemos os *barramentos*. Um barramento é simplesmente um conjunto de fios, cada um dos quais a cada momento pode ou não ter uma tensão elétrica. Se existe uma tensão

elétrica, convenciona-se que o fio contém, ou carrega, o valor 1; caso contrário, o fio contém o valor 0. Dessa forma, a cada momento um barramento contém uma informação que pode ser interpretada da mesma forma que uma informação na memória. Os barramentos são usados para transferir informações entre dois componentes. No momento da transferência, o componente que possui a informação coloca as tensões adequadas nos fios do barramento, e o componente destino faz a leitura das tensões nos fios e desta forma adquire a informação.

Note que na Figura 3.1 são usados três barramentos: um de dados, um de endereço e um de controle. Note também que os barramentos de dados e controle são desenhados com setas nas duas extremidades, indicando que os dados podem trafegar nas duas direções, enquanto que no barramento de endereços a informação trafega em uma única direção. O barramento de endereços contém tantos fios (bits) quantos forem necessários para endereçar todas as palavras da memória. O número de fios (bits) do barramento de dados é igual ao tamanho de cada palavra na memória. Assim, para comunicação com uma memória organizada como 2MB palavras de 32 bits (total de 8MB), o processador deve dispor de um barramento de dados com 32 fios, e um barramento de endereço com pelo menos 21 fios.

A comunicação entre o processador e a memória é feita da seguinte maneira. Quando o processador necessita de um dado na memória, ele especifica pelo barramento de endereços qual o endereço de memória da palavra que contém o dado. Feito isto, o processador especifica pelo barramento de controle que a operação é de leitura em memória. A memória coloca então o valor da palavra especificada no barramento de dados, que é finalmente lido pelo processador.

A operação de escrita na memória é similar. O processador coloca no barramento de endereços o endereço da palavra que deve ser modificada, e no barramento de dados o valor a ser escrito. O processador indica pelo barramento de controle que a operação é de escrita, e a operação é executada pela memória.

3.1 Funcionamento do processador

Vamos examinar mais detalhadamente o funcionamento do computador através da introdução de um processador didático, o Faíska, mostrado na Fig. 3.2.

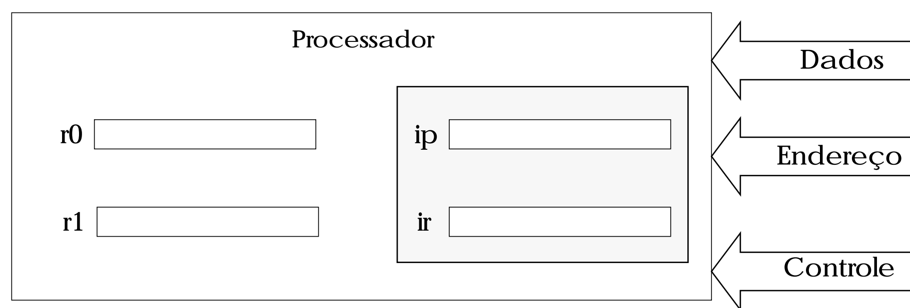


Fig. 3.2 Esquema simplificado de um processador

Os componentes mostrados no interior do processador (*r0*, *r1*, *ip* e *ir*) são chamados registradores. Um registrador é basicamente uma palavra de memória interna ao processador. A diferença é que acesso a um registrador é muito mais rápido que o acesso a qualquer palavra de memória externa ao processador.

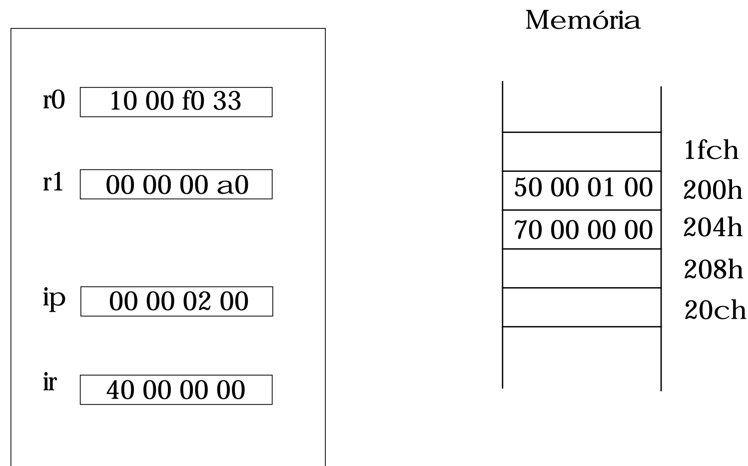
Os registradores *r0* e *r1* são de propósito geral, e podem ser usados para manipular dados do usuário. Apesar de apenas *r0* e *r1* serem mostrados no desenho, o Faíska possui 256 registradores de propósito geral, designados por *r0*, *r1*, ..., *r255*. Todos os registradores do Faíska são de 32 bits.

Os outros dois registradores mostrados na Figura 3.2 são internos ao processador. O registrador de instruções *ir* (em inglês, *instruction register*) armazena ocódigo da instrução que está sendo executada, enquanto que o registrador apontador de instruções *ip* (em inglês, *instruction pointer*) contém o endereço da próxima instrução a ser executada. O usuário não tem acesso direto aos registradores internos *ip* e *ir;* eles são mostrados nas figuras deste capítulo para facilitar o entendimento do funcionamento do processador.

O processador funciona em passos – cada instrução é composta por um número variável de passos, dependendo da complexidade da instrução. Os passos básicos de uma instrução são:

- busca de instrução,
- busca de operandos,
- execução e
- armazenamento dos resultados.

Vamos ilustrar o funcionamento do processador através da execução de uma instrução típica. Suponha que o código 50000100h represente a instrução “adicione o valor do registrador *r0* ao valor do registrador *r1* e coloque o resultado em *r1*”, e que em um dado momento a memória e os registradores contenham os seguintes valores:



A execução do próximo passo é

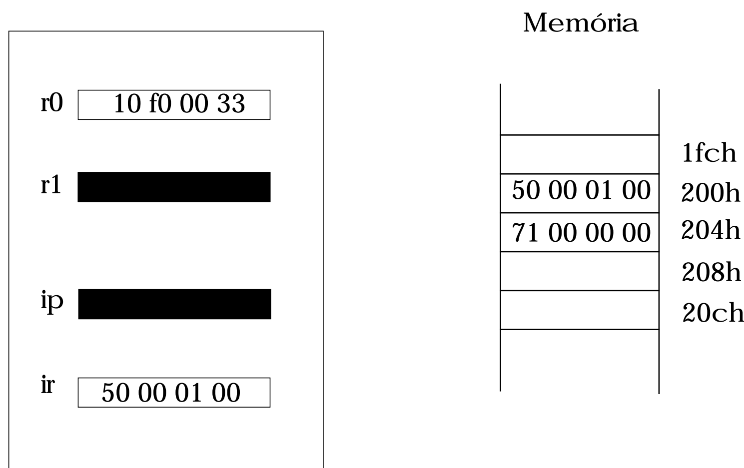
1. <u>Busca de instrução</u>: o processador lê a palavra de memória apontada por *ip* (endereço 0200h) e coloca o valor lido (50000100h) em *ir*. O processador *ip* é avançado do número de bytes lidos: *ip* passa a valer 0204h.

2. <u>Execução:</u> o processador executa a instrução correspondente ao código 50000100h: o processador toma os valores dos registradores *r0* e *r1* e efetua a soma.

3. <u>Armazenamento de resultado</u>: o resultado da soma é armazenado no registrador *r1*.

A configuração da memória (que neste caso não é alterada) e do

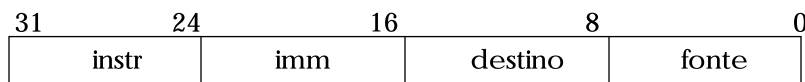
processador após a execução desse passo é mostrada a seguir:



Nesse caso, a próxima execução a ser executada é a de código 71000000h, presente no endereço de memória 204h.

3.1.1 Codificação das instruções

A codificação de instruções no Faíska é bastante simples. Toda instrução tem uma ou duas palavras. A primeira palavra é sempre dividida em quatro campos de um byte cada, como mostra a figura abaixo.



O campo *instr* representa o tipo da instrução a ser executada; no exemplo anterio mostramos que este campo deve ter o valor 50h para instruções de soma. Os campos *fonte* e *destino* codificam os registradores usados como fonte e/ou destino a serem utilizados na instrução. Os registradores são identificados nestes campos pelos seus números, ou seja, o registrador *r0* tem como representação o valor 0, e o registrador *r1* tem como representação o valor 1. A utilização dos campos *fonte* e *destino* dependem da instrução em questão; algumas utilizam apenas o campo destino, outras apenas o campo *fonte* e algumas utilizam ambos os campos.

Podemos agora entender a codificação da instrução “adicione o valor

do registrador *r0* ao valor do registrador *r1* e coloque o resultado em *r1*" :

instr	ender	destino	fonte
50	00	01	00

Note como os campos *fonte* e *destino* foram utilizados. A instrução de soma requer dois operandos fonte; por convenção, no Faíska neste caso o registrador destino é utilizado também como um dos operandos.

A codificação do Faíska é exageradamente simples. Em processadores reais a codificação é bem mais complexa, devido a fatores internos da arquitetura de cada processador (como por exemplo a maneira pela qual é feita a decodificação das instruções) e devido também ao fato de que existe um compromisso entre simplicidade e compactabilidade do código.

3.2 Modos de endereçamento

Na seção anterior nós vimos um exemplo de instrução que utiliza apenas registradores como operandos. Obviamente, os processadores possuem também instruções de transferência de dados entre memória e registradores. Instruções de leitura de memória escrevem ("carregam") um valor de uma posição de memória em um registrador, e instruções de escrita em memória escrevem o valor de um registrador em uma posição de memória.

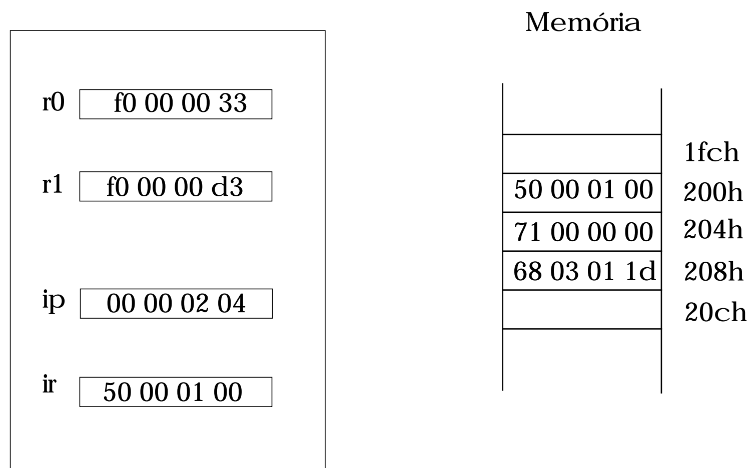
Em instruções que fazem acesso à memória é necessário indicar qual a palavra de memória que deve ser utilizada. Ou, mais precisamente, é necessário especificar como deve ser calculado o endereço da palavra de memória a ser utilizada (chamado *endereço efetivo* do operando).

Existem diversas formas possíveis de cálculo do endereço efetivo. Essas diferentes formas dão origem a diferentes *modos (ou tipos) de endereçamento*; cada instrução do processador define precisamente o modo de endereçamento utilizado. Nesta seção vamos apresentar apenas um modo de endereçamento, o chamado endereçamento imediato. Outros modos de endereçamento serão vistos mais adiante.

3.2.1 Endereçamento imediato

O modo mais simples de endereçamento é o endereçamento chamado *imediato*, que especifica que o valor do operando faz parte do próprio código da instrução.

No Faíska, por exemplo, a instrução "carregue o registrador com valor constante" é codificada utilizando-se duas palavras de memória. A primeira palavra é o próprio código da instrução (70010000h), e a segunda palavra contém o valor que dever ser carregado no registrador destino. Vamos examinar o funcionamento do processador em uma instrução de transferência de dados entre memória e registrador, usando endereçamento imediato. Suponha a seguinte configuração de processador e memória:



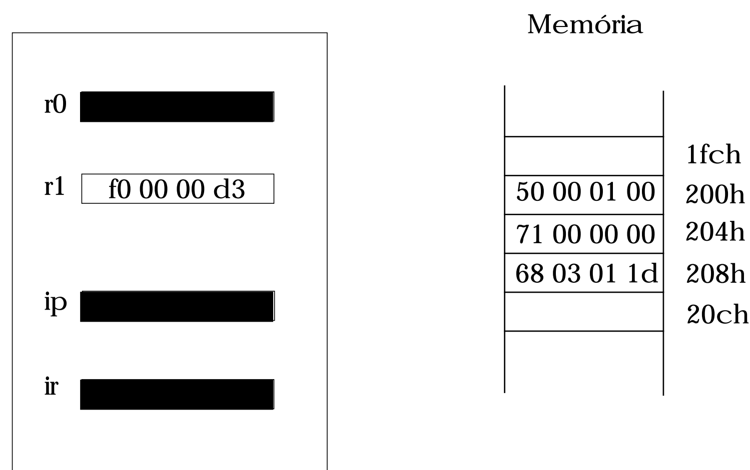
A seqüência de operações realizadas pelo processador no próximo passo a partir dessa configuração será:

1. <u>Busca de instrução</u>: o processador lê a palavra de memória apontada por *ip* (endereço 0204h) e coloca o valor lido (71000000h) em *ir*. O processador *ip* é avançado do número de bytes lidos: *ip* passa a valer 0208h. Note que neste momento o registrador *ip* aponta para a palavra de memória que contém o valor a ser carregado.

2. <u>Busca de operando</u>: o processador lê a palavra de memória apontada por *ip* (6803011d). O registrador *ip* é avançado do número de bytes lidos: *ip* passa a valer 020Ch.

3. <u>Armazenamento de resultado</u>: o valor lido é armazenado no registrador *r0*.

A configuração final da memória e do processador é a seguinte:



Note que também neste caso ao término da execução da instrução o registrador *ip* aponta para a próxima instrução a ser executada.

3.2.2 Um pequeno programa

Com a informação de como é a codificação das instruções no processador, vamos escrever um "programa" bastante simples:

Exemplo 3.1: Escrever um trecho de programa que calcule a expressão (7014h + 39BCh)*2 e coloque o resultado em *r1*.

```
70000000    carrega imediato em r0
00007014    este valor
70000100    carrega imediato em r1
000039bc    este valor
50000001    soma r0 com r1 e coloca o resultado em r1
50000101    soma r1 com r1 e coloca o resultado em r1
```

Colocando este trecho de programa em qualquer posição da memó-

ria, e fazendo com que o registrador *ip* aponte para o endereço inicial do trecho, a seqüência de instruções especificada será executada.

3.3 Introdução a linguagens de montagem

Já deve ter ficado claro que a programação feita no estilo do exemplo anterior seria uma tarefa árdua, pois teríamos que conhecer a codificação de cada instrução. Além disto, este método é muito sujeito a erros.

Entretanto, este foi o método utilizado pelos primeiros programadores, no início da era dos computadores (cérebros eletrônicos!). Esses programadores pioneiros logo descobriram que necessitavam de uma maneira mais confiável de montar as seqüências de instruções de um programa. Assim surgiram as primeiras linguagens de montagem (em inglês, *assembly languages*).

Uma linguagem de montagem é basicamente uma linguagem de programação bastante simples; tão simples que o compilador de uma linguagem de montagem não é chamado propriamente de compilador, mas apenas de *montador* (em inglês, *assembler*). Para facilitar a tarefa do montador, o formato de um programa em linguagem de montagem é simples e considerávelmente rígido. Por exemplo, deve haver apenas uma instrução de linguagem de montagem em cada linha do programa.

As instruções da linguagem de montagem são identificadas por *palavras reservadas* do montador, isto é, palavras que têm um significado fixo para o montador e não podem ter o seu significado redefinido pelo programador. Normalmente cada instrução em linguagem de montagem é traduzida pelo montador em uma instrução de máquina. Operandos devem aparecer à direita da instrução, e, se mais de um operando é necessário, eles devem ser separados por vírgulas. No Faíska usaremos a convenção de que o operando que será modificado pela instrução aparece mais à esquerda na lista de operandos.

Um exemplo de instrução em linguagem de montagem é

```
set     r1, aa00h
```

Quando encontrada pelo montador, esta instrução em linguagem de montagem é traduzida na seqüência de palavras

```
70000100h
0000aa00h
```

Para documentação do programa, a linguagem de montagem permite a inclusão de comentários. Um comentário é iniciado pelo caractere ';' e se estende até o final da linha. Comentários e linhas em branco são desconsiderados pelo montador.

A linguagem de montagem permite também a associação de um rótulo a uma posição de memória. Rótulos devem aparecer obrigatóriamente no início de uma linha e são usados para definir pontos importantes em um programa (início de um procedimento, por exemplo), para definir variáveis, ou simplesmente para documentação.

Cada linha de um programa em linguagem de montagem pode portanto ter o seguinte formato, onde a notação [*comp*] indica que a presença do componente *comp* é opcional (a ordem de cada um dos componentes na linha, no entanto, é fixa):

[*rótulo*:] [*instr*] [*lista de operandos*] [; *comentário*]

Exemplo 3.2: Escreva um trecho de programa em linguagem de montagem que calcule a expressão (7014h + 39BCh)*2 e coloque o resultado em *r1*.

```
; trecho que calcula a expressão
; (7014h + 39bch)*2
inicio:
        set     r0, 7014h       ; carrega primeiro termo
        set     r1, 39bch       ; carrega segundo termo
        add     r0, r1          ; soma os dois termos
        add     r1, r1          ; e dobra o valor
```

Note que o montador traduz este trecho de programa em linguagem de montagem exatamente na seqüência de instruções de máquina do Exemplo 3.2.

3.4 Diretivas do Montador

Além das instruções uma linguagem de montagem inclui também *diretivas*. Diretivas da linguagem de montagem, ao contrário das instruções, não são traduzidas em código de máquina. Elas servem para transmitir informações adicionais ao montador, como por exemplo definições de constantes. A linguagem de montagem do Faíska inclui diretivas para definir constantes, reservar espaço na memória para armazenamento de variáveis e indicar o endereço inicial de montagem.

3.4.1 Diretiva de definição de constantes

Constantes podem ser definidas em qualquer parte do programa em linguagem de montagem, mas usualmente as definimos no início do programa, para facilidade de leitura. Uma constante é introduzida pela diretiva EQU, cuja sintaxe é

nome EQU *valor*

Após a diretiva, toda ocorrência de *nome* é substituída por *valor*.

Exemplo 3.3:

```
; definição de constantes                1
                                         2
TRUE      equ    ffh                     3
FALSE     equ    0                       4
MAXVAL    equ    1000                    5
MINVAL    equ    MAXVAL/2                6
```

3.4.2 Diretiva de reserva de espaço em memória

Podemos reservar espaço na memória do Faíska de duas maneiras: inicializando o espaço com um valor conhecido, ou deixando o espaço com um valor desconhecido. Para simplesmente reservar espaço em memória usamos a diretiva DS, que tem o formato geral

[*rótulo*:] DS <*expressão_inteira*> [; *comentário*]

Esta diretiva reserva <*expressão_inteira*> bytes de memória (os valores iniciais são indefinidos).

Exemplo 3.4:

```
; exemplo de reserva de espaço na memória para variáveis do programa   1
; vamos primeiro definir uma constante                                 2
SIZE         equ   4                                                   3
                                                                       4
contador:    ds    1                                                   5
x:           ds    SIZE                                                6
final:       ds    4                                                   7
```

3.4.3 Diretivas de reserva e inicialização de espaço em memória

Para reservar e inicializar espaço em memória, usamos as diretivas DB (para reservar e inicializar bytes) e DW (para reservar e inicializar palavras).

Reserva e inicialização de bytes

A diretiva DB reserva e inicializa bytes:

[*rótulo*:] db <*lista_de_valores*> [; *comentário*]

Exemplo 3.5:

```
; exemplo de reserva de espaço na memória para variáveis do programa   1
; vamos primeiro definir uma constante                                 2
MAXVAL       equ   256                                                 3
                                                                       4
contador:    db    1                                                   5
estado:      db    0                                                   6
x:           db    MAXVAL - 1                                          7
letra:       db    'a'                                                 8
num:         db    -1                                                  9
```

O espaço de memória reservado pela seqüência de diretivas do Exemplo 3.5 é mostrado na Figura 3.3., onde a memória é apresentada como um vetor de bytes.

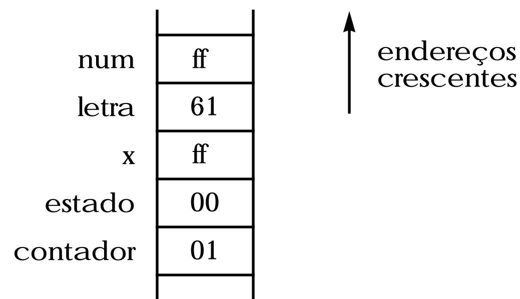


Fig. 3.3 Memória reservada pelo Exemplo 3.5.

Para facilitar a definição de seqüências de bytes, os valores podem ser separados por vírgulas.

Exemplo 3.6:

```
; exemplo de reserva de espaço na memória para variáveis do programa   1
; vamos primeiro definir uma constante                                 2
MAXVAL      equ     256                                                3
                                                                       4
dir:        db      128, feh, 'a', MAXVAL-1                            5
esq:        db      1, 0, 33                                           6
```

O espaço de memória reservado pela seqüência de diretivas do Exemplo 3.5 é mostrado na Figura 3.4.

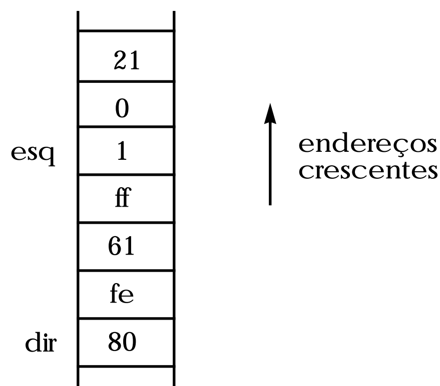


Fig. 3.4 Memória reservada pelo Exemplo 3.6.

Uma outra abreviação permitida é a utilização de aspas simples para a definição de cadeias de caracteres. Ou seja, a diretiva

```
Cadeia:     db      'biriba', 0
```

é equivalente a

```
Cadeia:     db     'b','i','r','i','b','a', 0
```

Reserva e inicialização de palavras

A diretiva de reserva de espaço de memória em palavras *dw* é similar à diretiva *db*, mas uma palavra inteira (4 bytes) é reservada e inicializada. O formato geral da diretiva dw é

[*rótulo*:]   dw   <*lista_de_valores*> [; comentário]

onde <*lista_de_valores*> é uma lista, separada por vírgulas, onde cada elemento pode ser uma constante numérica, um caracter, ou uma seqüência de caracteres entre aspas simples.

Exemplo 3.7:

```
; exemplo de uso da diretiva dw              1
ALTO        equ     32000h                   2
BAIXO       equ     2000h                    3
                                             4
var_x:      dw      ALTO - 1                 5
var_y:      dw      BAIXO * 2                6
            dw      31                       7
ultimo:     dw      -1                       8
```

O espaço de memória reservado pela seqüência de diretivas do Exem-

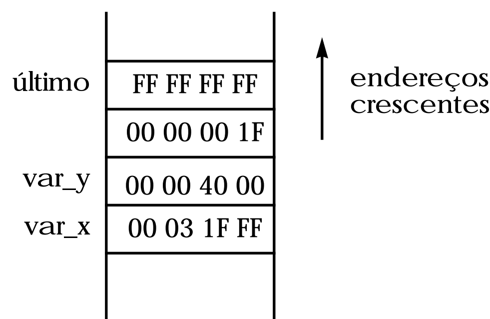


Fig. 3.5 Memória reservada pelo Exemplo 3.7

plo 3.8 é mostrado na Figura 3.5, onde a memória é apresentada como um vetor de palavras de 32 bits.

3.4.4 Diretiva para indicar endereço de montagem

Durante a montagem, o montador mantém um *contador de montagem* que indica o endereço corrente de montagem, isto é, o endereço da próxima palavra a ser montada. Durante o processo de montagem, esse contador é incrementado do número de palavras da instrução a cada instrução montada. O mesmo acontece quando o montador reserva espaço na memória: a cada espaço reservado o contador é incrementado do número de bytes ou palavras correspondente ao espaço.

Para alterar o valor do contador de montagem, utilizamos a diretiva *org*, que tem o formato

ORG <*expressão_inteira*> [; *comentário*]

Ao encontrar esta diretiva, o montador toma <*expressão_inteira*> como o novo valor do contador de montagem.

Exemplo 3.8:

```
; exemplo de reserva de espaço na memória para variáveis do programa    1
; vamos primeiro definir algumas constantes                             2
TAM_BLOCO   equ   16                                                     3
NUM_BLOCO   equ   64                                                     4
                                                                         5
; agora definimos o endereço inicial de montagem do bloco               6
            org   TAM_BLOCO * (NUM_BLOCO - 1)                            7
buffer1:    dw    0, 1, 2                                                8
; este outro bloco deve começar em outro endereço                       9
            org   200h                                                  10
buffer2:    db    10, 11, 12, 13, 14                                    11
```

O espaço de memória reservado pela seqüência de diretivas do Exemplo 3.8 é mostrado na Figura 3.6.

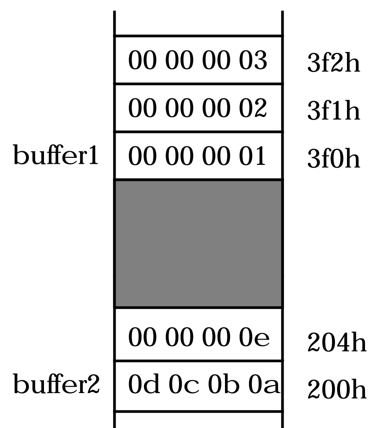


Fig. 3.6 Memória reservada pelo Exemplo 3.8.

3.5 Exercícios