Introdução aos Computadores
Notas e Exercícios

Ana Paula Tomás
Nelma Moreira

Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
email: {apt,nam}@ncc.up.pt

1996

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Linguagem *Assembly* do MC68000

1.1 MC68000 — generalidades

Bibliografia: T. L. Harman, B. Lawson, The Motorola MC68000 Microprocessor Family - Assembly Language, Interface Design, and System Design, Prentice-Hall, 1985

MC68000 como circuito integrado

O microprocessador MC68000 é constituído por aproximadamente 68000 componentes integradas numa única peça (*chip*) de silicone, sendo por isso um produto **VLSI** (*very large scale integration*). O chip tem cerca de $246 \times 280 \times 10^{-3}$polegadas, ou seja cerca de 68000×10^{-6}polegadas^2 de área, e por coincidência também contém cerca de 68000 transístores.

Um grande área de ROM no chip é usada para guardar microcódigo que controla os circuitos para execução das instruções máquina.

A tecnologia do MC68000 é **HMOS** (*high-density metal-oxide-semiconductor*). O tamanho dos elementos é reduzido (relativamente à tecnologia **MOS**) permitindo aumentar a velocidade e reduzir o consumo de energia de cada transístor no CPU.

O processador

Como em muitos microprocessadores, no MC68000 as operações ao nível dos circuitos são controladas por uma sequência de instruções microprogramadas e guardadas em duas secções de ROM no chip do CPU. Quando as instruções em linguagem máquina são carregadas da memória externa e descodificadas, determinam a execução dum microprograma. Este microprograma controla toda a actividade nas linhas de sinais e todas as transferências de dados ou operações dentro do CPU durante a execução da instrução. Como o microcódigo não pode ser modificado excepto por criação dum novo chip com outro microcódigo, o utilizador raramente se interessa pelos detalhes das operações do processador a este nível. No entanto, para determinar exactamente o que o processador está a fazer em resposta a uma instrução é necessário compreender o microcódigo.

Microprocessadores como o MC68000 são incorporados em sistemas de computadores para controlar toda operação do sistema. Este processadores podem gerir a actividade

do sistema, executando programas e as funções de input/output necessárias. Mais ainda, separam o processamento em dois modos ou estados — *supervisor* e *utilizador*, permitem administração e protecção de memória, e detectam vários tipos de erros. Os processadores anteriores (classe de 8-bits) não tinham em geral estas capacidades. O MC68000 pertence à classe dos processadores de 16 bits.

Do ponto de vista de programação, os processadores de 16 bits oferecem um conjunto geral e potente de instruções assim como diversos modos de referir o operando em memória (*modos de endereçamento*). Muitos destes processadores suportam técnicas especiais de programação e ajudas à detecção de erros (*debugging*).

A interacção entre o processador e as outras componentes de hardware (equipamento) do sistema efectua-se pelo **bus** do sistema, onde ocorrem transferências de sinais de controlo, endereços e dados.

A capacidade e flexibilidade do processador neste aspecto é determinada pelas funções das linhas de sinal. Por exemplo, capacidades sofisticadas de I/O (input/output) e de interrupção obviam a necessidade de bastante hardware especializado, simplificando o sistema. Uma *interrupção* é determinada por um sinal exterior ao microcomputador, que causa uma transferência de controlo do programa em execução para um programa especial que responderá ao pedido. Depois de processar a interrupção, o controlo regressa ao programa que foi interrompido.

Em geral os processadores de 16 bits fornecem o estado do processador e outras informações aos circuitos externos à medida que o processador vai trabalhando, característica que permite simplificar o hardware.

O sistema

O designer do sistema preocupa-se com o funcionamento geral do sistema, incluindo a sua performance e fiabilidade. Determina também o uso da memória e a forma como estão protegidas as áreas de memória ocupadas pelo sistema de operação. Esta protecção é vital se o sistema for usado para desenvolver software, no qual possam ocorrer por exemplo erros de endereçamento, ou programas que saem fora dos limites.

Quando vários periféricos estão ligados ao sistema, o planeamento da porção de I/O é deveras importante para assegurar uma coordenação apropriada entre programas e hardware durante as transferências de dados. O MC68000 dispõe de dois modos de execução — modo supervisor e modo utilizador — que permitem evitar que erros de aplicações afectem o funcionamento geral do sistema de operação. Como seria de esperar, o sistema de operação é executado em modo supervisor, incluindo todas as rotinas para efectuar I/O e interrupções. Tipicamente, as aplicações correm em modo utilizador. Neste modo, algumas das instruções e áreas de memória serão inacessíveis. A atribuição do modo dos vários programas é feita pelo designer de sistema, e as transições entre os dois modos são cuidadosamente controladas.

O designer do sistema especifica o número e tipo de periféricos necessários a uma tarefa. O design do subsistema de I/O torna-se complicado quando muitos dispositivos estão ligados, uma vez que unidades diferentes podem demorar intervalos de tempo diferentes para completar as transferências de dados pedidas. Devido a estas variações, o sistema

permite normalmente que os periféricos enviem interrupções através de linhas de sinais de controlo da interface para o CPU. Um pedido é enviado quando o dispositivo se encontra preparado para receber ou transmitir dados ou quando uma condição de erro é detectada.

Programação em linguagem de Assemblador

A facilidade do desenvolvimento (escrita, correcção e teste) dum programa em linguagem de assemblador depende mais das características do processador, do que do desenvolvimento de software.

Editores, assembladores, e outras ajudas ao desenvolvimento variam em qualidade e eficiência, mas um bom sistema de desenvolvimento não substitui um processador inadequado. Os processadores da classe MC68000 são apropriados para a maioria das aplicações por disporem dum conjunto potente de instruções, numerosos modos de endereçamento e características especiais. O termo *Arquitectura do Computador* é por vezes usado para referir o conjunto de características do computador que são importantes para o programador. Uma descrição da arquitectura inclui uma definição da organização global do sistema assim como uma descrição completa das características de programação do CPU. O conjunto de instruções do processador e os modos de endereçamento constituem dois dos tópicos mais importantes desta descrição.

O conjunto de instruções dum microprocessador é a colecção de instruções disponíveis para o programador. Cada instrução pode ser descrita pela sua operação ou função, e pelo número e tipo dos seus operandos.

O MC68000 tem um conjunto de instruções aritméticas razoavelmente completo. Os processadores anteriores (de 8-bits), não tinham instruções de divisão e multiplicação. Isto permite simplicar a programação e aumentar a velocidade de execução dos programas. O MC68000 pode efectuar operações sobre operandos considerados com 8 bits, 16 bits ou 32 bits. Normalmente, os processadores de 8-bits permitiam operandos apenas de 8 bits.

Os endereços para operandos guardados em memória podem ser indicados na instrução por inteiros de 24 bits. Este método de endereçamento directo diz-se *endereçamento absoluto*. São possíveis outras formas de endereçamento, designando-se em geral por *modos de endereçamento*. Por exemplo, no MC68000 existem 14 modos de endereçamento distintos. Assim, as instruções de MC68000 devem indicar, além da operação a fazer, o modo de endereçamento usado para referir o endereço do operando. Quando a instrução é executada, o CPU calcula um endereço absoluto, que neste contexto será designado por *endereço efectivo*

Quando analisado em termos de variedade de modos de endereçamento, o sistema MC68000 é mais parecido com um computador grande do que com um microcomputador, uma vez que os microprocessadores anteriores tinham normalmente poucas capacidades de endereçamento. Por exemplo, vários tipos de estruturas de dados, como listas e vectores, podem ser facilmente criadas em memória usando os modos de endereçamento do MC68000.

A modularização de programas é uma boa técnica de programação que facilita a detecção de erros e o teste. Cada módulo implementa uma subtarefa, e o programa completo junta os diversos módulos. O MC68000 tem algumas instruções que suportam a pro-

gramação modular, incluindo instruções para chamada de subprogramas (módulos).

O MC68000 tem algumas características que ajudam à detecção de erros em programas. Quando uma instrução que causa um erro é executada ocorre um "Trap". Por exemplo, uma instrução ilegal, uma tentativa para dividir por zero, causam um "trap". Alguns erros de endereçamento e condições aritméticas podem também provocar trap. Nesse caso, o controlo passa a uma subrotina do sistema de operação que procede de acordo com o tipo de erro detectado. É guardada informação sobre o estado do processador e o endereço da instrução ofensiva, para facilitar o diagnóstico do problema.

As instruções que o processador executa estão em memória codificadas em linguagem máquina. Estas instruções podem ser escritas por um programador que escreve directamente as sequências em binário. Em geral um programador de assembler não trabalha directamente com linguagem máquina. No entanto, um conhecimento sobre o formato das instruções máquina pode ser necessário para a compreensão das capacidades do processador.

1.2 Programação do MC68000

1.2.1 Registos Programáveis do MC68000

O CPU contém um grande número de posições de memória especiais — **registos** –formadas por um ou mais elementos (cada um dos quais podendo guardar um bit de informação). A **capacidade** ("length") de um registo é o número de bits que podem ser lidos ou guardados simultaneamente no registo. A maior parte dos registos do CPU guardam informação específica necessária à execução de instruções não estando disponíveis para o programador. Existem registos (em número reduzido) que podem ser programados – **registos programáveis**.

No MC68000 os registos programáveis dividem-se em registos de *uso geral* (utilizados para guardar endereços ou dados manipulados por uma instrução), registo de *sequência do programa* ("program counter"; 32 bits; guarda o endereço da próxima instrução a ser executada), registo *estado* ("status register"; 16 bits; indica o estado do processador e dá informação sobre resultados de operações aritméticas ou similares).

No MC68000 existem 17 registos de uso geral: 8 **registos de dados** ("data registers") e 9 **registos de endereço** ("address registers"). A velocidade de execução de um programa aumenta se se usar os registos como operandos pois o acesso a posições de memória ou a periféricos é normalmente mais demorado.

Registos de Dados

Podem ser usados para guardar (e aceder a) 8 bits, 16 bits, ou 32 bits de informação, podendo ainda manipular-se apenas um bit. São, por exemplo, operandos em algumas instruções aritméticas. Quando o seu conteúdo é o operando efectivo numa dada instrução, toma-se o byte menos significativo (bits 0 a 7; bit 7 indica o sinal do valor) para operando

de 8 bits, os 2 bytes menos significativos (bits 0 a 15; bit 15 indica sinal) para operando de 16 bits, e o valor completo (bits 0 a 32) quando é de 32 bits.

Registos de Endereço

Contêm endereços de posições de memória. Pode-se aceder aos 16 bits menos significativos ou ao conteúdo integral (32 bits) do registo. Apenas os 24 bits menos significativos são usados para representar um endereço (por restrição do número de linhas de endereço no BUS). O registo A7 contém o endereço do topo da pilha do sistema. Como o processador pode funcionar em dois modos distintos (user e supervisor) o registo A7 (por vezes também denotado por SP) pode apontar duas pilhas: em modo utilizador será USP (user stack pointer) e em modo supervisor SSP (supervisor stack pointer). Os elementos da **pilha do sistema** no MC68000 são de 16 bits. O registo A7 aponta o último elemento colocado. Pode-se retirar ou colocar um elemento no topo da pilha (processo **LIFO** "last in first out"). Os elementos estão colocados em posições de endereços decrescentes (topo é o elemento de menor endereço).

A Tabela 1.1 sumariza os registos programáveis do MC68000.

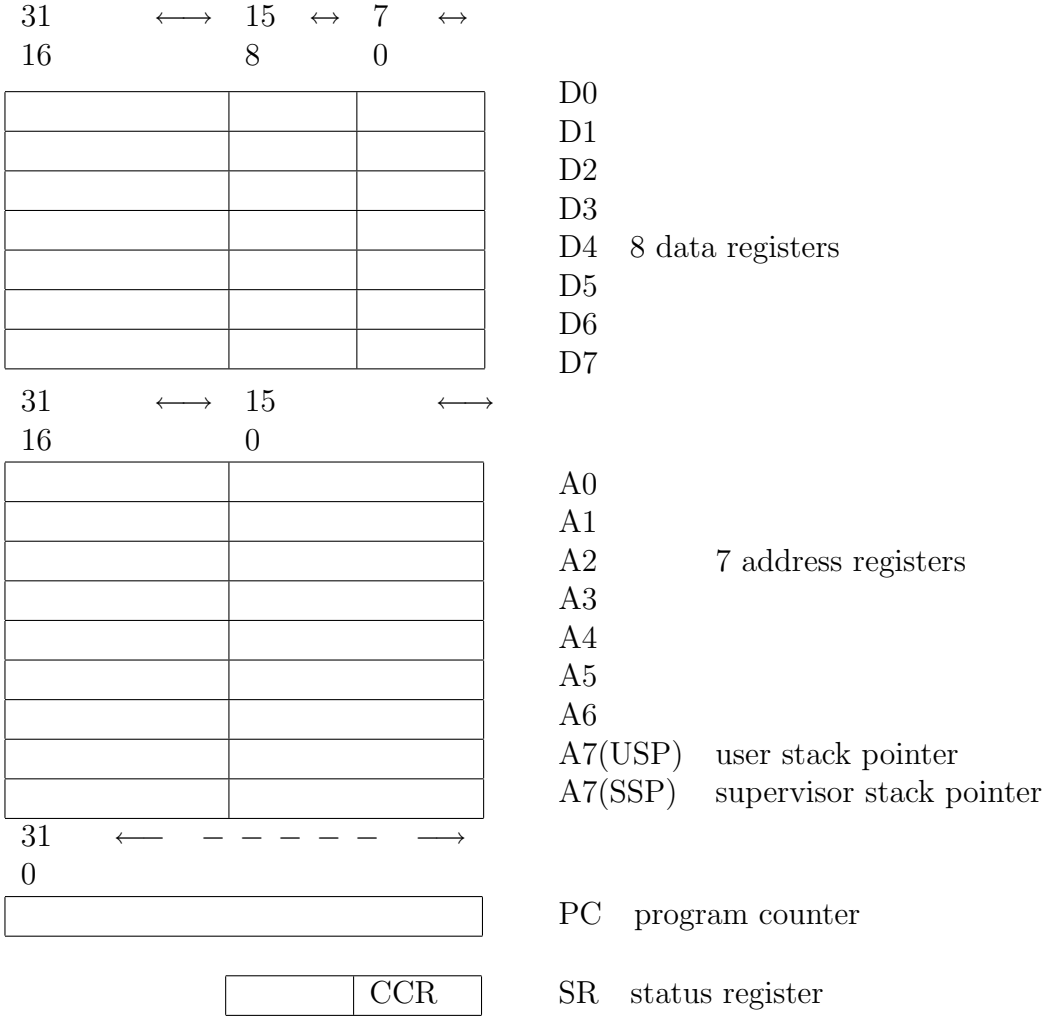
1.2.2 Organização da Memória

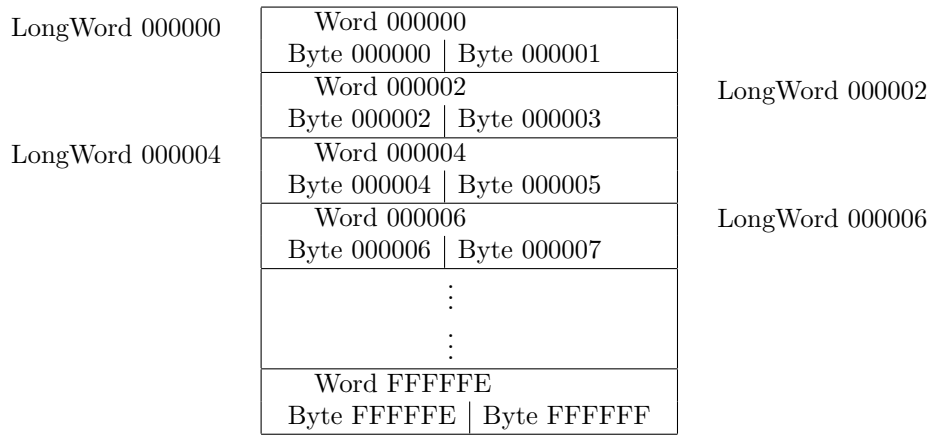
A memória é um conjunto de posições de 8 bits organizadas em **palavras**. No sentido usado na bibliografia do MC68000 uma palavra é um conjunto de 16 bits com endereço par. O processador pode aceder a cada uma das posições (8 bits). Cada instrução em memória é sempre um grupo destas palavras, assim como os operandos de 16 bits e de 32 bits, tendo por isso endereços **pares**.

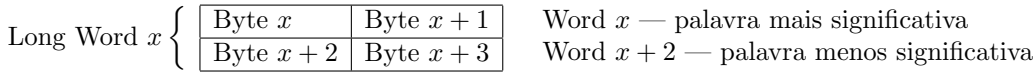
Dos **16 MB** endereçáveis (000000 a FFFFFF), 1024 (1KB) bytes estão reservados contendo endereços de instruções iniciais de subprogramas do sistema de operação.

Quando um valor de 16 bits é guardado em memória (numa palavra) o byte mais significativo do valor é guardado na posição de endereço menor. Do mesmo modo, se um valor de 32 bits é guardado como palavra longa a partir de uma posição de endereço E, então o byte mais significativo ocupa a posição E e o menos significativo a posição $E + 3$.

Tabela 1.1: Registos Programáveis do MC68000







1.2.3 Modos de Endereçamento

O microprocessador MC68000 permite 14 modos de endereçamento diferentes:

1. **Literal (ou Imediato)** (Immediate Data Addressing): O operando efectivo é o operando codificado. Sintaxe: $\#n$. Exemplo: MOVE.B #3,D0

2. **Directo com Registo** (Register Direct Addressing): O operando efectivo é o conteúdo do registo de dados ou de endereço indicado (tomado com o tamanho especificado na operação). Sintaxe: Dn ou An. Exemplos: CLR.B D3 ou o segundo operando em MOVE.B #3,D0.

3. **Directo dando o endereço efectivo** (Absolute data Addressing): O operando codificado é o endereço da palavra de memória que contém o operando efectivo. O operando codificado é um endereço em 16 (endereço curto; "absolute short") ou 32 bits (endereço longo; "absolute long"). Um endereço curto é sempre extendido (a 24 bits, prolongando o sinal) antes de ser usado: endereços 0 a 7FFF correspondem às posições 000000 a 007FFF; mas os endereços 8000 a FFFF são interpretados como FF8000 a FFFFFF. Uma instrução máquina em que se usou endereço curto ocupa menos 16 bits do que uma idêntica mas que usa o longo.

   NB: O prolongamento do sinal significa que o bit mais significativo da representação (16 bits) do endereço curto é usado para obter a representação em 24 bits. Os endereços são valores sem sinal. Exemplos: CLR.B $1001 (Como se descreve mais à frente $ indica ao assemblador que a constante 1001 é um hexadecimal).

4. **Indirecto** (Indirect Addressing): Sempre com registo de endereço.

(a) **Indirecto com registo de endereço** (Address Register Indirect). Sintaxe: (An). O operando efectivo encontra-se na posição apontada por An (ou seja, na posição de memória cujo endereço é o conteúdo de An). Exemplo: CLR.B (A0)

(b) **Indirecto com pós-incremento do conteúdo do registo de endereço** (Indirect with postincrement). Sintaxe: (An)+. O operando efectivo encontra-se na posição apontada por An, e o valor de An é depois incrementado de 1, 2 ou 4 unidades conforme o operando seja byte, palavra ou palavra longa, respectivamente. Exemplo: CLR.B (A0)+

(c) **Indirecto com pré-decremento do conteúdo do registo de endereço** (Indirect with predecrement). Sintaxe: -(An). O conteúdo do registo An decrementado de 1, 2 ou 4 unidades conforme o operando seja byte, palavra ou palavra longa. O novo conteúdo de An é o endereço do operando efectivo. Exemplo: CLR.B -(A2)

(d) **Indirecto com deslocamento** (Indirect with displacement). Sintaxe: d_{16}(An) onde d_{16} é um inteiro com sinal e em 16 bits. O endereço efectivo é obtido somando o conteúdo em An com o valor d_{16} depois de prolongar esse valor a 32 bits. O conteúdo de An não é alterado. Exemplo: CLR.B 2(A0).

(e) **Indirecto com índice** (Indirect with index). Sintaxe: d_8(An,Ri) em que Ri designa um registo de endereço ou de dados, e d_8 é um inteiro com sinal e em 8 bits. O endereço efectivo é obtido somando o conteúdo em An com o valor d_8 depois de prolongar esse valor a 32 bits, e com o conteúdo do registo Ri. Os conteúdos de An e Ri não são alterados. Exemplos: em CLR.B -2(A0,D0.W) apenas a palavra menos significativa de D0 é tomada; em CLR.B -2(A0,D0.L) é adicionado o valor correspondente aos 32 bits para determinar o endereço operando efectivo.

5. **Relativo**(Relative Addressing): Modo indirecto utilizando o registo de sequência do programa. Sintaxe: d_{16}(PC) e d_8(PC,Ri).

6. **Implícito** (Implied Addressing): uso implícito de registos SR, USP, SP, ou PC por certas instruções (exemplos: saltos, chamadas de subprogramas).

1.2.4 Formato das Instruções em Assembly 68000

Qualquer instrução é constituída por 4 campos: *endereço* de instrução (label), *mnemónica* de operação, *operandos* e *comentários*. Estes campos são separados por **espaços** e podem estar vazios. Por exemplo,

inicio	MOVE.W	D1,D2	;guardar valor de D1
endereço	mnemónica e tamanho do(s) operando(s)	operandos	comentário

- O primeiro campo, que contém o endereço da instrução, é na maior parte das instruções opcional; muitas vezes é um endereço simbólico (label), como por exemplo **inicio**.

- O segundo campo tem que conter a mnemónica de uma instrução ou para o processador ou para o assemblador, como por exemplo MOVE. Se a instrução tiver operandos que possam ser tomados com vários comprimentos (Byte, Word, ou Long word), o considerado será indicado a seguir à mnemónica estando separado dela por um ponto, como em MOVE.W.

- A instrução pode ter zero, um ou dois operandos, sendo no último caso separados por uma vírgula, sem qualquer espaço de separação.

- Quaisquer caracteres que ocorram depois dos operandos (separados deles por um espaço ou ;) não são traduzidos pelo assemblador, constituindo um comentário. Se uma instrução tiver na primeira coluna o caracter *, é interpretada como uma linha de comentário.

Expressões (avaliadas pelo Assemblador)

Uma expressão é uma combinação de símbolos, constantes, operadores algébricos e parentesis que pode ocorrer no campo de operandos da instrução e que é avaliada pelo e assemblador para determinar o endereço ou o valor de um operando.

Uma constante pode representar um número ou um caracter (ASCII). Uma constante númerica pode representar um valor inteiro decimal ou hexadecimal de 8 bits, 16 bits ou 32 bits. Um hexadecimal é precedido de um simbolo $. Para alguns assembladores, as constantes octais são precedidas por @ e as binárias por %. Assim, %100, $64, @144, e %01100100 são representações do mesmo inteiro em decimal, hexadecimal, octal e binário respectivamente.

Um literal ASCII é uma sequência de caracteres (no máximo 4) entre ' ', a qual é reconhecida pelo assemblador e convertida no seu código ASCII. Por exemplo, ao literal 'ESTE' são associados 4 bytes correspondendo a $45535445.

Os operadores +, -, *, / correspondendo a adição, subtracção, multiplicação e divisão podem ser usados em expressões. O resultado de qualquer operação aritmética é um inteiro de 32-bits.

O operador - pode ser usado como operador unário para definir números negativos, como por exemplo na instrução MOVE.W #-1,D1. Esta é equivalente a MOVE.W #$FFFF,D1 sendo $FFFF a representação hexadecimal em complemento para 2 do valor -1 (tomado com 16 bits). Essa representação é calculada pelo assemblador e guardada com a instrução máquina.

Comandos para o Assemblador (Assembler Directives)

Não são instruções para o processador (isto é, instruções executáveis) mas para o assemblador. Podem servir para controlar o código gerado, para definir símbolos, dados (constantes) e reservar espaço na memória. Alguns exemplos:

END — indica ao assemblador o fim de texto do programa; não é obrigatória

DC.t — (Define Constant) t pode ser B, W, ou L; permite atribuir certos valores a posições de memória. Exemplo: pela instrução **dados DC.B 5,2,2,3** guarda-se os valores 5, 2, 2, e 3 em quatro bytes contíguos na memória a partir da posição de endereço **dados**

DS.t — (Data Storage) t é B, W, ou L; permite reservar posições de memória sendo os seus conteúdos inicializados com zero. Por exemplo, qualquer das instruções seguintes reserva 8 bytes a partir da posição **mem**

```
mem DS.B 8
mem DS.W 4
mem DS.L 2
```

EQU — (Equate) Atribui o valor de uma certa expressão a um identificador ou símbolo (label). Por exemplo, pela instrução **maxval EQU 100** qualquer ocorrência no programa do símbolo **maxval** é substituida pelo inteiro 100. Se uma constante ocorre muitas vezes num programa, deve ser substituida por um símbolo, pois se pretender alterar o valor da constante basta atribuir um novo valor ao símbolo que a representa.

Está prevista a utilização dum interpretador de Assembly 68000 — **e68k**.

Aconselha-se a consulta do manual para informações relativas ao formato das instruções, e directivas.

1.3 Instruções do MC68000

1.3.1 Instruções para Transferência de Dados

CLR — (Clear an Operand) Coloca zero no operando efectivo

EXG — (Exchange Registers) Troca os conteúdos de dois registos

SWAP — (Swap Register Halves) Troca os 16 bits menos significativos de um registo de dados com os seus 16 bits mais significativos

MOVE — (Move Data) Copia o conteúdo do operando origem para o operando destino

MOVEQ — (Move Data Quick) Copia um dado imediato de até 8 bits para um registo Dn, afectando os 32 bits do registo

MOVEM — (Move Multiple Registers) Se o operando destino for o endereço de uma posição de memória, copia os conteúdos de uma lista de registos (Dn e An) para posições de memória contíguas a partir da indicada. Se o operando destino for uma lista de registos, estes são carregados com os conteúdos de posições de memória contíguas sendo o endereço da primeira dado no primeiro operando.

É uma transferência ordenada: começa pelos registos de dados D0 a D7 (se ocorrerem na lista) e termina com os de endereço A0 a A7 (se ocorrerem). Os elementos da lista são separados por / como por exemplo em D0/A0. Quando se pretende referir uma sequência de registos do mesmo tipo e com números consecutivos pode-se usar uma notação abreviada Ri-Rj por exemplo D4-D6.

Exemplos:

- MOVEM.L D1/D4-D6/A4,(A7)
  copia os conteúdos de D1, D4, D5, D6, e A4 para posições consecutivas a partir da de endereço dado em A7; Se em A7 tiver x, então D1 é colocado a partir de x, D4 de $x+4$, D5 de $x+8$, ..., A4 de $x+16$. O valor em A7 não é alterado.
- MOVEM.L D1/D4-D6/A4,-(A7)
  se em A7 tiver x, então copia A4 para $x-4$, D6 para $x-8$, D5 para $x-12$, D4 para $x-16$, e D1 para $x-20$. O valor em A7 passa a ser $x-20$. *Quando o modo de endereçamento é indirecto com pré-decremento a ordem dos registos é invertida.*
- MOVEM.L (A7)+, D1/D4-D6/A4
  se em A7 tiver x, então copia (x) para D1, $(x+4)$ para D4, $(x+8)$ para D5, $(x+12)$ para D6, e $(x+16)$ para A4. Em A7 fica $x+20$.

1.3.2 Exercícios

1. Escrever instruções assembly para:
   (a) atribuir o valor 15 ao byte menos significativo do registo D2;
   (b) atribuir o valor -35 ao byte menos significativo do registo D2;
   (c) colocar o endereço longo **aqui** em A6;
   (d) colocar o conteúdo da posição **dado** no byte menos significativo de D6;
   (e) Trocar o conteúdo dos registos **D0** e **D1** (**.L**) usando apenas uma posição de memória auxiliar (p.ex.: **$F110**).
   (f) Trocar as palavras (**.W**) nas posições de memória **$E00** e **$F00** usando apenas um registo temporário.
   (g) Inicializar **5** posições de memória (**.W**) com o valor **$0** apartir do endereço **$F110**.
   (h) colocar o conteúdo da palavra cujo endereço é o conteúdo de A1, na palavra de endereço **aqui**;

(i) colocar o conteúdo da palavra longa cujo endereço é o conteúdo de A1 menos 4, na palavra longa de endereço **aqui**;

(j) copiar o conteúdo do byte menos significativo de D3 para a posição de memória cujo endereço é o conteúdo de A3 menos 11;

(k) copiar o conteúdo da palavra longa cujo endereço é a soma do conteúdo de A3 com 250 para a posição cujo endereço é dado por A5, incrementando A5 de forma a apontar a palavra longa seguinte;

(l) colocar os valores de D3, D4, D5, A2, e A4 em cinco palavras longas contíguas supondo que o endereço da primeira é dado por A6;

(m) copiar os valores de 5 palavras longas contíguas para D3, D4, D5, A2, e A4 sendo o endereço da primeira dado por A6;

(n) copiar a palavra mais significativa de D2 para a menos significativa de D3;

(o) copiar um valor de 64 bits que está colocado a partir do endereço **VALOR1** para outro endereço **VALOR2**;

(p) colocar a sequência de caracteres 'FIM' a partir da posição **mensg**

2. *Rodar* os conteúdos (**.L**) dos registos **D1**,**D2**, **D3**, **D4** e **D5** usando:

   (a) apenas um registo auxiliar (p.ex.: **D6**).

   (b) usando a instrução `EXG`.

3. Copiar os últimos 16 *bits* do registo **D2** para os últimos 16 *bits* do registo **D5** sem alterar os primeiros 16 *bits* de nenhum deles.

4. Dadas duas posições de memória `BYTE` e `PALAVRA`, contendo, respectivamente, inteiros de 1 e 2 *bytes*, pretende-se copiar:

   (a) o inteiro em `BYTE` para `PALAVRA`;

   (b) o inteiro em `PALAVRA` para `BYTE` (eventualmente truncando o tamanho);

   (c) resolva o mesmo exercício, começando por colocar os endereços `BYTE` e `PALAVRA` em **A0** e **A1**, respectivamente.

5. Para cada um dos seguintes extractos de programa indique o significado de cada instrução e (sempre oportuno!) simplifique o programa.

a)
```
MOVE.W  #$FFFF,D0
SWAP    D0
MOVE.W  #$FFFF,D0
```

b)
```
MOVE.B   #$D0,D0
MOVEA.L  #1000,A0
MOVE.B   D0,(A0)
```

c)
```
ah  DS.B     6
    MOVEM.L  D0-D1,-(A7)
    MOVE.L   #$41534349,D0
    MOVE.W   #$4921,D1
    MOVE.L   D0,ah
    MOVE.W   D1,ah+4
    MOVEM.L  (A7)+,D0-D1
```

1.3.3 Operações Aritméticas

Representação de inteiros — Algumas notas

Seja x um inteiro positivo que pode ser representado por m bits. Por definição, o *complemento para 2 de x* é o inteiro $2^m - x$. A representação (em computador) mais comum para um inteiro negativo $-x$ é o complemento para 2 de x. Assim, por exemplo, numa representação em 8 bits usando complemento para 2 o inteiro -3 é o binário correspondente a $2^8 - 3$, ou seja $253 = 11111101$.

Num sistema de representação em m bits que permite denotar inteiros negativos (em complemento para 2) e não negativos metade das representações são entendidas como números positivos e a outra metade como números negativos. Assim, de 0 a $2^{m-1} - 1$ tem-se números não negativos e de 2^{m-1} a $2^m - 1$ números negativos. Usando a definição de complemento para 2, pode-se concluir que $2^m - 1$ representa -1 em complemento para 2, e que 2^{m-1} representa -2^{m-1}, pelo que os valores possíveis dos inteiros representáveis variam de -2^{m-1} a $2^{m-1} - 1$.

Exercício: Quais os inteiros que podem ser representados num sistema de complemento para 2 em 8, 16, e 32 bits?

Uma vez que $2^m - x = (2^m - 1) - x + 1$, pode-se usar a seguinte regra prática para calcular o complemento para 2 de um inteiro positivo x: toma-se a representação em binário de x em m bits; troca-se nessa representação cada 1 por 0 e cada 0 por 1 (o que equivale a subtrair x de $2^m - 1$) e ao resultado soma-se 1. (Nota: $(2^m - 1) - x$ diz-se o *complemento para 1 de x*)

Por exemplo, para representar o inteiro -3 em 8 bits e complemento para 2, toma-se 00000011 (binário 3); efectua-se a troca indicada obtendo-se 11111100 e soma-se 1 resultando 11111101.

Exercício: Quais os inteiros representados em 8-bits e complemento para 2 por \$F2, \$4F, \$A3, 10010000, 00000000, 10000000, \$7F?

Adição e Subtracção binária de Inteiros não negativos

- Adição: Segue algoritmo habitual de adição binária; se se tiver uma representação de m bits e a soma for superior a $2^m - 1$ (o maior inteiro positivo com m bits) diz-se que há transporte (**carry**); no MC68000 isso é assinalado pelo registo **C** (toma valor 1) do registo de condições **CCR**, sendo uma indicação de que o valor calculado está errado. Por exemplo: o valor de C é 1 depois de efectuar a soma em 8-bits dos inteiros positivos 10010000 e 11111101, e o resultado é 10001101

- Subtracção: Segue algoritmo habitual de subtracção binária; se o valor da diferença for inferior a zero, diz-se que há transporte (**borrow**); no MC68000 isso é assinalado pelo registo **C** (toma valor 1) do registo de condições **CCR**, sendo indicação de que o valor obtido está errado. Por exemplo: a diferença em 8-bits $10010000 - 11111101 = 10010011$ (resultado errado) afecta o registo **C** colocando-o em 1.

O resultado da soma ou diferença de inteiros não negativos (colocado no operando destino) está correcto se e só se o valor do registo **C** depois da operação for 0.

Adição de Inteiros em Complemento para 2

Se os inteiros são não negativos, a soma é calculada pelo algoritmo habitual. Se a representação tiver m bits e a soma for superior a $2^{m-1} - 1$ (o maior inteiro positivo com m bits num sistema de complemento para 2) diz-se que há **overflow**; no MC68000 isso é assinalado pelo registo **V** (toma valor 1) do registo de condições **CCR**, sendo indicação de que o valor obtido está errado.

Por exemplo: a soma em 8-bits e complemento para 2 de 01111111 com 00000001 é 10000000 ou seja, a soma de $+127$ com $+1$ é -128(!). Neste caso, o registo **V** tomaria o valor 1. Note-se que no registo **C** ficaria 0 pois se essas representações fossem interpretadas como inteiros positivos não haveria carry.

Se os inteiros são ambos negativos (em complemento para 2) a soma é calculada pelo algoritmo habitual "desprezando-se"o transporte. Se se usa uma representação em m bits, o "desprezar"o transporte corresponde a subtrair ao resultado obtido 2^m. Basta notar que quando se efectua $(-x) + (-y)$ se pretende obter o complemento para 2 de $x + y$, isto é $2^m - (x + y)$. A relação desse número com os complementos para 2 de x e y é $2^m - (x + y) = (2^m - x) + (2^m - y) - 2^m$. Ou seja, para obter a representação em complemento para 2 de $(-x) + (-y)$ adicionam-se as representações de $(-x)$ e $(-y)$ e subtrai-se 2^m.

Se a representação tiver m bits e a soma for inferior a -2^{m-1} (o menor inteiro negativo com m bits num sistema de complemento para 2) diz-se que há **underflow**; no MC68000 isso é assinalado pelo registo **V** (toma valor 1), sendo indicação de que o valor obtido está errado.

Por exemplo: a soma em 8-bits e complemento para 2 de 10000000 com 11111111 seria 01111111 ou seja, $-128 + (-1) = +127$ (!). Neste caso, o registo **V** tomaria o valor 1.

Note-se que no registo **C** ficaria 1 pois se essas representações fossem interpretadas como inteiros positivos haveria carry.

Analogamente, a soma de um inteiro positivo com um inteiro negativo é o binário correspondente à adição das representações dos operandos, "desprezando-se"o transporte se o houver. Não haverá overflow nem underflow.

O resultado da soma de inteiros em complemento para 2 (colocado no operando destino) está correcto sse o valor do registo **V** depois da operação for 0.

Subtracção de Inteiros em Complemento para 2

A subtracção binária de inteiros representados em complemento para 2 pode reduzir-se à adição de inteiros.

	Exemplos
(-x)-(-y) = (-x)+y	10010000-11111101=10010000+00000011=10010011
(-x)-y=(-x)+(-y)	10010000-00000011=10010000+11111101=10001101
y-(-x)=y+x	00000011-10010000=00000011+ 01110000=01110011
y-x=y+(-x)	00000011-00010000=00000011+11110000=11110011

Do mesmo modo que para a adição de inteiros, o resultado da diferença de inteiros em complemento para 2 (colocado no operando destino) está correcto sse o valor do registo **V** depois da operação for 0.

Instruções para Adição e Subtracção

ADD — (Add Binary) Adiciona o conteúdo do primeiro operando (não literal) ao segundo (operando destino, não An). O tamanho da operação deve ser especificado, podendo ser byte (8 bits), word (16 bits) ou long word (32 bits). Se o primeiro operando for um dos registos de endereço, o tamanho da operação só pode ser **W** e **L**. Afecta apenas a parte do operando destino correspondente ao tamanho da operação. Altera o registo **CCR** de acordo com o resultado da operação.

ADDA — (Add Address) Análogo a ADD mas o operando destino é um registo de endereço. Afecta os 32 bits do registo de endereço (independente do tamanho dos operandos); Não afecta **CCR**.

ADDI — (Add Immediate) Análogo a ADD mas o primeiro operando é um literal (Immediate data); afecta **CCR**.

ADDQ — (Add Quick): Análogo a ADD mas o primeiro operando é um literal em 3 bits. 1–7 representam valores de 1–7, e 0 representa 8; afecta **CCR** se operando destino não for um dos registos de endereço.

ADDX — (Add Extended) Adiciona o conteúdo do primeiro operando, com o segundo e com **X** (do registo **CCR**); Aplica-se em adição em precisão múltipla. Por exemplo, para efectuar a soma de dois inteiros de 64 bits, somando primeiramente os 32 bits menos significativos e depois os restantes com **X**, que representa o carry da primeira adição anterior.

SUB — (Subtract Binary) $destino - origem$

SUBA — (Subtract Address)

SUBI — (Subtract Immediate)

SUBQ — (Subtract Quick)

SUBX — (Subtract Extended) $destino - origem - \mathbf{X}$

A sintaxe das instruções, os modos de endereçamento dos operandos, e os tamanhos possíveis de operação são os indicados na Tabela 1.2.

Tabela 1.2: Instruções de Adicção e Subtracção

Mnem.	Sintaxe	Endereçamento e Tamanho de Operandos(0)
ADD	ADD $<ea>$,Dn	all (B,W,L but An – W,L)
	ADD Dn,$<ea>$	all (B,W,L) but Dn, An, d(PC) to Imm
ADDA	ADDA $<ea>$,An	all (W,L)
ADDI	ADDI #$<data>$,$<ea>$	all (B,W,L) but An, d(PC) to Imm(1)
ADDQ	ADDQ #$<data>$,$<ea>$	all (B,W,L but An – W,L) but d(PC)–Imm(2)
ADDX	ADDX Dx,Dy	— B, W, L
	ADDX -(Ax),-(Ay)	— B, W, L
SUB	SUB $<ea>$,Dn	all (B,W,L but An – W,L)
	SUB Dn,$<ea>$	all (B,W,L) but Dn, An, d(PC)–Imm
SUBA	SUBA $<ea>$,An	all (W,L)
SUBI	SUBI #$<data>$,$<ea>$	all (B,W,L) but An, d(PC)– Imm(1)
SUBQ	SUBQ #$<data>$,$<ea>$	all (B,W,L but An – W,L) but d(PC)–Imm(2)
SUBX	SUBX Dx,Dy	— B, W, L
	SUBX -(Ax),-(Ay)	— B, W, L

(0) Ordem: Dn, An, (An), (An)+, -(An), d_{16}(An), d_8(An,Xi), Abs.W, Abs.L, d_{16}(PC), d_8(PC,Xi), Imm
(1) immediate data matches the operation size
(2) 3 bits; valores entre 1 e 8

O registo de condições **CCR** é afectado consoante o resultado das operações efectuadas. Para saber se o resultado colocado no operando destino está ou não correcto, o utilizador/programador deve testar os registos **C** ou **V** de **CCR**:

- **C** se os operandos representam inteiros sem sinal;
- **V** se os operandos representam inteiros com sinal (complemento para 2)

Por exemplo, sejam 01111111 e 00000001 os valores nos bytes menos significativos de D0 e D1 respectivamente. Após a execução da instrução $ADD.BD0, D1$ tal byte de D1 conteria 10000000 e o estado do registo de condições seria X=0, N=1, Z=0, V=1, C=0. Como foi visto anteriormente, se se considerar que aqueles valores são representações com sinal de inteiros o resultado em D1 não estaria correcto, facto que se concluiria de V=1. Se se considerar que os valores representam inteiros positivos (sem sinal) então o resultado está correcto (comprovado por C=0).

Exemplos: Em cada um dos exemplos a primeira parcela é o conteúdo do byte menos significativo de D3, e a segunda e o resultado é o conteúdo do byte menos significativo de D2. Supõe-se que se executa $ADD.BD3, D2$. Quais são os valores decimais envolvidos nas adições (complemento para 2) seguintes? Quais são as erradas? Quais seriam as erradas se se interpretasse as representações como inteiros sem sinal?

negativos					negativo e positivo					positivos				
	10010000					00010000					00010000			
+	11111101				+	11111111				+	00111101			
1	10001101				1	00001111				0	01001101			
X	N	Z	V	C	X	N	Z	V	C	X	N	Z	V	C
1	1	0	0	1	1	0	0	0	1	0	0	0	0	0

positivos					negativos				
	01111010					10000110			
+	01110001				+	10001111			
0	11101011	< 0			1	00010101	> 0		
overflow					underflow				
X	N	Z	V	C	X	N	Z	V	C
0	1	0	1	0	1	0	0	1	1

Multiplicação e Divisão de Inteiros

O Assembly 68000 tem instruções para efectuar o produto e a divisão inteira de inteiros. Tais instruções são diferentes consoante se trate de inteiros representados em complemento para 2 (isto é, com sinal), ou apenas não negativos (sem sinal): **MULU**, **MULS**, **DIVU**, e **DIVS**. Os sufixos **U** (unsigned) e **S** (signed) das mnemónicas dessas instruções indicam o tipo de multiplicação ou divisão considerado.

MULU — (Unsigned Multiply) Calcula o produto de dois inteiros (sem sinal) de 16 bits. O resultado é tomado com 32 bits, e colocado no segundo operando da instrução,

o qual é um registo de dados. Sintaxe: MULU < *ea* >,Dn. O primeiro operando admite todos os modos de endereçamento excepto An.

−	∗	∗	0	0
X	N	Z	V	C

X não é alterado
N = 1 se bit mais significativo do produto for 1
Z = 1 se produto for zero

MULS — (Signed Multiply) Análogo ao anterior mas os operandos são tomados como inteiros em complemento para 2.

DIVU — (Unsigned Divide) Divide o operando destino (segundo operando) pelo primeiro operando tomando-os como inteiros sem sinal. O segundo operando é palavra longa (32 bits) e o primeiro palavra (16 bits). O resultado ocupa 32 bits: a palavra menos significativa é o **quociente**, e a mais significativa é o **resto**. A divisão por zero causa um erro assinalado pelo processador; Se ocorrer overflow (quociente não representável em 16 bits, ou seja superior a $2^{16} - 1$) o bit **V** toma o valor 1 e os operandos não são alterados. Sintaxe: DIVU < *ea* >,Dn. O primeiro operando admite todos os modos de endereçamento excepto An.

−	∗	∗	∗	0
X	N	Z	V	C

X não é alterado
N = 1 se bit mais significativo do quociente for 1; indefinido se overflow
Z = 1 se quociente for zero; não definido se overflow
V = 1 se quociente não for representável em 16 bits

DIVS — (Signed Divide): Análogo ao anterior mas os operandos são tomados como inteiros em complemento para 2. O sinal do resto é igual ao do dividendo, a menos que o resto seja zero. Ocorre overflow se o quociente for inferior a -2^{15} ou superior a $+2^{15} - 1$.

−	∗	∗	∗	0
X	N	Z	V	C

X não é alterado
N = 1 se quociente for negativo; indefinido se overflow
Z = 1 se quociente for zero; não definido se overflow
V = 1 se quociente não for representável em 16 bits

1.3.4 Exercícios

1. Somar o conteúdo da posição de memória **VALOR/** (**.W**) a todos os registos de dados (D_n).

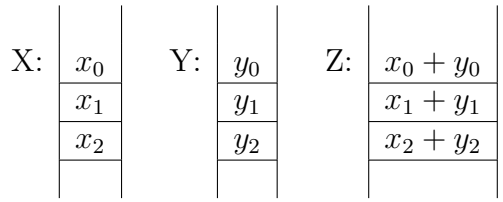


Figura 1.1: Soma de vectores

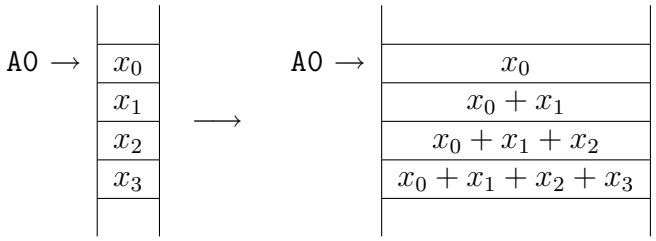


Figura 1.2: Somas parcelares

2. Somar o conteúdo de duas posições de memória contidas nos registos **A1** e **A3** e colocar o resultado em cada um desses endereços.

3. Calcular em **D0** o valor de $2 \times$ **D0** $+ 1$.

4. Considerem-se **3** vectores de **3** bytes (**.B**) em memória apontados por **X**,**Y** e **Z** (constantes). Somar os elementos dos vectores **X** e **Y** um a um e colocar os resultados nas posições respectivas em **Z** (ver figura 1.1).

5. Considerem-se **4** palavras (**.W**) na memória apartir da posição apontada por **A0** (x_0,x_1,x_2 e x_3). Faça um programa que calcule as somas parcelares (x_0,$x_0 + x_1$,$x_0 + x_1 + x_2$,$x_0 + x_1 + x_2 + x_3$) e as coloque nas posições de, respectivamente, (x_0,x_1, x_2 e x_3) (ver figura 1.2).

1.3.5 Registo de Condições CCR (Condition Code Register)

É o byte menos significativo do registo **SR** (Status Register), e contém os registos de 1 bit ou flags ("bandeiras") apresentados na tabela 1.3.5.

Estes registos são alterados de forma consistente, de modo a darem uma informação útil sobre o resultado da operação realizada pelo processador. As instruções que são casos particulares de outras mais gerais afectam o registo **CCR** da mesma forma que estas. Em qualquer utilização de uma mesma instrução o registo **CCR** é sempre alterado da mesma maneira. Em apêndice é dada uma tabela que apresenta o modo como cada operação altera o registo **CCR**. Por exemplo, a operação **MOVE**, não afecta **X**, coloca 1 em **N** se o resultado for negativo (bit mais significativo do resultado é 1), coloca 1 em **Z** se o resultado for zero, e coloca zero nos bits **C** e **V**.

C	(Carry) toma valor 1 se houver "carry"numa adição, ou "borrow"numa subtracção; Caso contrário é zero.
V	(Overflow) toma valor 1 se houver "overflow"numa operação aritmética, o que significa que o resultado não é representável com o número de bits dos operandos; Caso contrário é zero.
Z	(Zero) toma valor 1 se o resultado for zero; Caso contrário 0.
N	(Negative) toma valor 1 se o bit mais significativo do resultado for 1; Caso contrário é zero.
X	(Extend) quando é afectado toma o mesmo valor que o bit C tomar. É um dos operandos em cálculos em precisão múltipla.

Tabela 1.3: Registos do CCR

Existem **instruções condicionais** (por exemplo **BNE**, **BCC**, **BCS**,...) que testam um ou mais bits do registo de condição. A mnemónica duma instrução condicional tem como sufixo alguma das mnemónicas de condição indicadas numa tabela dada em Apêndice. Na tabela é apresentado também o teste equivalente a cada condição. Por exemplo, a condição **NE** (*Not Equal*) equivale a testar se ¬Z (isto é, se o resultado da operação anterior não era zero); a condição **CS** (*Carry Set*) equivale a testar **C** (ou seja, se houve "carry"na operação anterior) enquanto **CC** (*Carry Clear*) testa ¬C.

1.3.6 Exercícios

1. Indique o conteúdo dos registos e estado das *flags* **C,V,Z,N** após a execução das instruções:

   (a)
   ```
   CLR.L D0
   ```

   (b)
   ```
   CLR.L D0
   MOVE.W #$FFFF,D0
   ```

   (c)
   ```
   MOVE.L #$FFFF,D0
   MOVE.B #0,D0
   ```

   (d)
   ```
   MOVE.L #$FF,D0
   ADDI.L #1,D0
   ```

   (e)
   ```
   MOVE.L #$FF,D0
   ADDI.B #1,D0
   ```

   (f)
   ```
   MOVE.W #10,D0
   SUBI.W #20,D0
   ```

   (g)
   ```
   MOVE.B #$7F,D0
   ADDI.B #$F0,D0
   ```

   (h)
   ```
   MOVE.B #$7F,D0
   ADDI.B #$7F,D0
   ```

2. Suponha que x e y são dois inteiros representados em n dígitos em complemento para dois. Indique, justificando, em qual dos casos poderá obter situações de "transporte" (*carry* ou *borrow*) e/ou *overflow* ou *underflow*:

   (a) $x \geq 0$ e $y \geq 0$, efectuo $x + y$;

   (b) $x \geq 0$ e $y < 0$, efectuo $x + y$;

   (c) $x < 0$ e $y < 0$, efectuo $x + y$;

1.3.7 Instruções de Comparação e Teste

As instruções **CMP** (*Compare*) e TST (*Test*) modificam **CCR** de acordo com o valor dos seus operandos, sendo usadas antes de instruções condicionais. Convém notar que por vezes (por exemplo, após uma adição) não é necessário, nem desejável, usar instruções **CMP** e **TST** antes de instruções condicionais atendendo-se antes ao modo como cada operação altera o registo **CCR**.

As instruções de comparação têm dois operandos e calculam a diferença entre o segundo operando e o primeiro operando, alterando o registo **CCR** de acordo com essa diferença. Têm as mnemónicas seguintes:

Mnemónica	Nome
CMP	compare
CMPI	compare immediate
CMPM	compare memory

A instrução de teste (**TST**) tem apenas um operando e calcula a diferença entre esse operando e zero, modificando o registo **CCR** conforme o resultado. Os modos de endereçamento, os tamanhos dos operandos e o modo como as alterações do registo **CCR** são calculadas estão descritos nas tabelas apresentadas em anexo.

1.3.8 Instruções de Transferência de Controlo

Instruções Incondicionais

As instruções **JMP** (*JuMP*) e **BRA** (*BRanch Always*) permitem alterar sempre o valor do registo de sequência do programa. O novo endereço de instrução deve ser ser par, isto é um endereço de palavra. A instrução "jump"permite saltar (**salto absoluto**) para uma qualquer instrução na memória. A instrução "branch"só permite efectuar um **salto relativo** com amplitude limitada. Essa amplitude é dada por um inteiro em complemento para 2 representável em 8 ou 16 bits. Convém notar que o valor do registo **PC** quando a instrução de "branch"está a ser executada é igual ao endereço da instrução dessa instrução mais 2 (uma vez que **PC** aponta a próxima instrução a ser executada).

Instruções Condicionais

A instrução **Bcc** (*BRanch conditionally*), em que **cc** é substituído por a mnemónica da condição desejada (por exemplo BNE, BCS, ldots), só altera o valor do registo **PC** se o conteúdo do registo **CCR** satisfizer a condição **cc**.

Uma outra instrução condicional que por vezes pode ser útil para implementar algoritmos iterativos é **DBcc** (*Test Condition, Decrement, and Branch*). Consultar as tabelas dadas para mais detalhes.

1.3.9 Exercícios

1. Diga como se pode traduzir para Assembly 68000 as seguintes formas gerais de instruções (da linguagem P):

   a) $x \leftarrow constante$

   b) $x \leftarrow y$

   c)
   ```
   Se (condição) então
       {instruções};
   instrução seguinte;
   ```

   d)
   ```
   Se (condição) então
       {instruções1};
   senão {instruções2};
   ```

   e)
   ```
   enquanto (condição) faça
       {instruções};
   instrução seguinte;
   ```

   f)
   ```
   repita
       {instruções};
   até (condição);
   instrução seguinte;
   ```

2. Copiar o número de palavras (**.W**) indicado pelo primeiro byte (**.B**) do registo **D2** da posição de memória **ORIGEM** para **DESTINO**.

3. Inverter a ordem dos bytes (**.B**) colocados entre as posições de memória **INICIO** e **FIM**.

4. Inverter a posição dos bytes (**.B**) de cada uma de **N** palavras (**.W**) colocadas apartir da posição de memória **INICIO**.

5. *Rodar* por uma posição as palavras longas (`.L`) colocadas entre as posições de memória **INICIO** e **FIM** (ver figura 1.3).

6. O mesmo problema anterior mas agora rodando **M** posições. (i.é., **M** vezes a operação anterior).

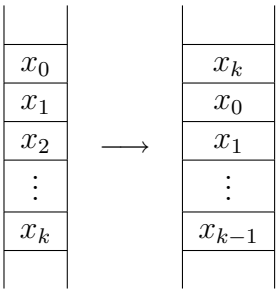


Figura 1.3: Rotação

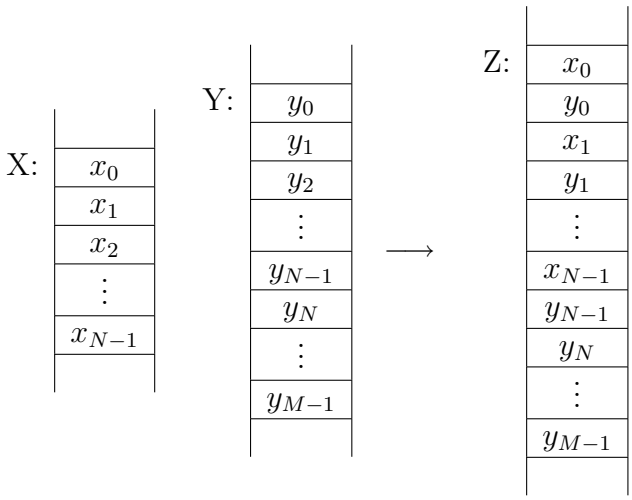


Figura 1.4: Interlaçar

7. Dado um inteiro com sinal em **D0**, escreva um programa que calcule o seu valor absoluto em D1.

8. Calcular a soma de todos os números de **1** até uma constante **N**, dada em D0.

9. Somar **N** palavras (**N** em **D0**) (**.W**) colocados apartir da posição de memória **INICIO**.

10. O mesmo problema de 1.3.4.4, mas agora com vectores de tamanho **N**.

11. O mesmo problema de 1.3.4.5, mas agora com **N** palavras.

12. Dados dois vectores de palavras (**.W**) nas posições **X** e **Y** (constantes) e de tamanhos respectivamente, **N** e **M** (eventualmente diferentes), faça um programa que *interlace* os elementos dos dois vectores apartir da posição **Z**. Caso um dos vectores tenha mais elementos que o outro, os elementos restantes devem ser copiados para o fim do vector **Z** (ver figura 1.4).

13. Considere uma sequência de caracteres terminada com 0, começando em `PALAVRA`. Escreva um programa que copie a sequência pela ordem inversa para a àrea começando em `INVERSA`.

14. Dadas duas sequências de caracteres (apontadas por A0 e A1), escreva um programa que copie para a àrea de memória começando em `PREFIXO` o maior prefixo comum às duas sequências.

1.3.10 Instruções Condicionais e Aritméticas

Já foi anteriormente mencionado que quando se pretende testar condições depois de operações aritméticas se deve distinguir entre estar a tomar os operandos sem sinal (positivos ou zero), ou com sinal (representação em complemento para 2). Assim, por exemplo, as condições a testar para saber se de uma certa operação com sinal resultou overflow é **VS** (*overflow set*, equivale a **V**) ou **VC** (*overflow clear*, equivale a ¬V). Por exemplo, a instrução **BVS errado**, permite efectuar um desvio para a instrução de endereço **errado** se o bit **V** estiver com o valor 1. Na tabela 1.4 ilustra-se transferências de controlo após operações aritméticas.

Tabela 1.4: Instruções Condicionais e Aritméticas

Instruções	*Branch* com Condição Sem Sinal	Com Sinal
ADD.t X,Y SUB.t X,Y MOVE.t X,Y	(Y) = 0 BEQ (Y) ≠ 0 BNE	(Y) = 0 BEQ (Y) ≠ 0 BNE (Y) ≥ 0 BPL (Y) < 0 BMI
Out of range for arithmetic instructions	C = {1} BCS C = {0} BCC	V = {1} BVS V = {0} BVC
CMP X,Y	(X) = (Y) BEQ (X) ≠ (Y) BNE (X) > (Y) BHI (X) ≥ (Y) BCC (X) < (Y) BCS (X) ≤ (Y) BLS	(X) = (Y) BEQ (X) ≠ (Y) BNE (X) > (Y) BGT (X) ≥ (Y) BGE (X) < (Y) BLT (X) ≤ (Y) BLE
TST X	(X) = 0 BEQ (X) ≠ 0 BNE (X) > 0 BNE (X) < 0 — (X) ≥ 0 — (X) ≤ 0 —	(X) = 0 BEQ (X) ≠ 0 BNE (X) > 0 BGT (X) < 0 BLT, BMI (X) ≥ 0 BGE, BPL (X) ≤ 0 BLE
MOVE admite ainda BGE, BLT, BGT, e BLE se os operandos tiverem sinal		

1.3.11 Exercícios

Escrever programas em Assembly 68000 para resolver cada um dos problemas seguintes.

1. Calcular o valor da expressão seguinte, supondo que / denota a divisão inteira

$$-(40 \times (152 + 38 - 85)/6 + 4 \times 62) \times (152 + 38 - 85)$$

   Se o resultado estiver errado, no byte **erro** deve ficar o valor 1; senão zero.

2. Dado um valor inteiro de 16 bits em x calcular em f o valor $f(x)$ assim definido:

$$f(x) = \begin{cases} x - 5 & \text{se } x - 5 \geq 0 \\ -(x - 5) & \text{se } x - 5 < 0 \end{cases}$$

   Se o resultado estiver errado, no byte **erro** deve ficar o valor 1; senão zero.

3. Determinar o valor absoluto de um número inteiro **X** de 16-bits dado em complemento para 2. O resultado (em 16 bits) deve ficar em **ABS**. Para que valor(es) de **X** ocorre overflow? Nessas condições o resultado está errado?

4. Dados dois inteiros positivos de 16 bits **X** e **Y** calcular por adições sucessivas, e em 32 bits, o produto de **X** por **Y**. O resultado deve ficar em **R**. Poderá existir *carry* em alguma das adições?

5. Calcular em **D0** o quociente e o resto da divisão inteira de um inteiro não negativo de 32 bits dado em **D0**, por um inteiro positivo de 16 bits **Y** sem utilizar instruções **DIV**. A palavra mais significativa de **D0** deve conter o resto da divisão e a menos significativa o quociente. Devem ser assinalados dois tipos de erro: overflow do quociente — **D1.B** conterá 1; e divisão por zero —**D1.B** conterá 2;

6. Transferir **N** bytes de informação de uma sequência iniciada na posição **AQUI** para uma outra com ínicio em **ALI**. Supor que **N** é um inteiro não negativo em 16 bits, e que a sequência dos endereços dos elementos é decrescente.

7. Determinar o número de inteiros não negativos de uma sequência de inteiros de 16 bits, sendo **n** (inteiro não negativo de 32 bits) o comprimento da sequência. Suponha que os elementos da sequência estão colocados em memória em posições de endereços crescentes sendo o primeiro o conteúdo (longo) do registo **A1**.

8. Calcular a parte inteira da média aritmética de uma sequência de inteiros não negativos de 16 bits guardados em palavras de memória contíguas (e de endereços crescentes) a partir da palavra de endereço **DADOS**. O número de elementos da sequência é o conteúdo da palavra de endereço **NVAL**. O resultado deve ser colocado em **MEDIA**. Poderá ocorrer overflow na divisão ou carry na soma?

9. Determinar o valor máximo (em **MAXINT**) de uma sequência de inteiros (com sinal) de 16 bits colocados em memória a partir da posição **DADOS**, para cada um dos casos seguintes.

   (a) O número de inteiros da sequência é o conteúdo da palavra de endereço **NINT**.

   (b) A sequência termina por $8000, não sendo dado o seu número de elementos.

10. Reescrever uma dada sequência de caracteres (em código ASCII) colocada a partir da posição de endereço **SEQ**, substituindo as letras minúsculas por maiúsculas.

11. Compactar uma sequência de caracteres dada, retirando os *caracteres de controlo* (código ASCII inferior a 32), supondo que **A0** aponta o primeiro caracter da sequência.

12. Dado um inteiro positivo em 16 bits, **Int**, determinar a sequência de caracteres correspondente à sua representação decimal, colocando-a a partir da posição **DEC**. Considere também o problema inverso: dada uma sequência de caracteres numéricos, a partir da posição **DEC** e terminada por 0, escrever um programa que calcule o número decimal correspondente, deixando o resultado em D0.L, ou colocar o valor -1 no registo D1, se ocorrer um erro.

13. Determinar a sequência de caracteres da representação com ponto decimal de um número racional positivo inferior a 1, supondo uma aproximação de **N** (inteiro não negativo de 8 bits) casas decimais. O numerador e denominador do número são dados nos bytes **Num** e **Den**. O resultado deve ser colocado na memória a partir de **RAC**. O processo para determinar os dígitos é o a seguir ilustrado:

$$\begin{array}{rclcl} 4/7 & = & 0.1(40/7) & = & 0.1(5+5/7) \\ & = & .5+0.1(0.1*(50/7)) & = & .5+0.1(0.1*(7+1/7)) \\ & = & .57+0.01*1/7 & = & \ldots \end{array}$$

1.3.12 Subprogramas ou Subrotinas

Um subprograma ou sub-rotina é uma sequência de instruções que pode ser tratada como um módulo independente de um programa maior. O mesmo subprograma pode ser chamado várias vezes, para executar uma tarefa específica, durante a execução de um programa. O uso de subprogramas é uma técnica de programação que facilita a concepção de programas. Em geral, um programa para efectuar uma certa tarefa pode ser partido em blocos independentes (módulos), cada um deles correspondendo a uma parte (bem determinada) dessa tarefa. Cada subprograma pode ser testado independentemente do programa que o chama permitindo, por exemplo, detectar erros mais facilmente.

Quando um subprograma é chamado, as suas instruções são executadas e depois o controlo regressa à instrução do programa principal imediatamente a seguir à instrução de chamada da sub-rotina. O endereço da primeira instrução da sub-rotina (endereço inicial) deve ser conhecido para que esta possa ser chamada. Se uma sub-rotina e o programa que a chama forem traduzidos pelo assemblador ao mesmo tempo, então o endereço inicial

da sub-rotina pode ser definido por um endereço simbólico (label), que ocorrerá na sua chamada no programa. Senão, tal endereço tem que ser dado explicitamente na instrução de chamada.

O endereço (longo) da instrução imediatamente a seguir à da chamada da sub-rotina é guardado na pilha do sistema (apontada por SP ou A7), quando a instrução de chamada é executada. Nenhum outro valor é automaticamente guardado. da responsabilidade do programador guardar os conteúdos dos registos que contiverem valores relevantes para o programa. Tais valores podem ser guardados na pilha do sistema antes da chamada da sub-rotina, se se souber quais os registos que a sub-rotina altera, e repostos depois de regressar. No entanto, é preferível efectuar essas operações dentro da sub-rotina, à entrada e imediatamente antes da saída. Uma mesma sub-rotina pode ser chamada várias vezes num programa, e assim pode-se diminuir o código do programa.

Instruções para Chamada de Sub-rotina

As instruções de chamada de uma sub-rotina são instruções de transferência de controlo para a instrução inicial da sub-rotina, podendo tais transferências ser relativas ou absolutas. No primeiro caso, o assemblador calcula o valor a somar ao conteúdo do registo PC para que tal registo passe a conter o endereço inicial da sub-rotina. Esse valor é guardado como um inteiro representado em complemento para 2 em 16 bits, o que limita a amplitude do "salto"a efectuar. A instrução de chamada é **BSR** (*Branch to Subroutine*), e é em tudo análoga a **BRA** excepto o facto do endereço (32 bits) da instrução seguinte (conteúdo do registo PC) ser guardado na pilha do sistema.

Se a transferência for absoluta o valor do registo PC é guardado na pilha do sistema, e o seu novo valor passa a ser o endereço da instrução inicial da sub-rotina. A instrução de chamada é **JSR** (*Jump to Subroutine*).

Instruções de Retorno da Sub-rotina

Em geral, a última instrução a ser executada num subprograma carrega o registo PC com o valor (32 bits) que está no topo da pilha do sistema, que deve ser o endereço da próxima instrução a executar. A instrução de retorno é **RTS** (*Return from Subroutine*), e não tem operandos.

A instrução **RTR** (*Return and Restore Condition Codes*) repõe o valor do registo CCR anterior à chamada da sub-rotina (se esse valor tiver sido guardado na pilha do sistema a seguir ao endereço de retorno) e o valor do registo PC. O valor de 16 bits que estiver no topo da pilha do sistema é usado para repor o valor do registo CCR mas o byte mais significativo do registo SR não é alterado. A instrução **MOVE SR,-(A7)** pode ser usada para colocar na pilha do sistema o valor do registo SR (o qual inclui o registo CCR).

A estrutura geral duma chamada a subrotina pode ser:

```
subr   MOVE.W    SR, -(A7)              ; guarda as condições
       MOVEM.L   D0-D7/A0-A6, -(A7)     ; guarda valor dos registos
       ...
       ...
       MOVEM.L   (A7)+,D0-D7/A0-A6      ; restaura valor dos registos
       RTR
```

Na figura 1.5 ilustra-se o uso da pilha de sistema durante a chamada a uma subrotina (neste caso JSR subr).

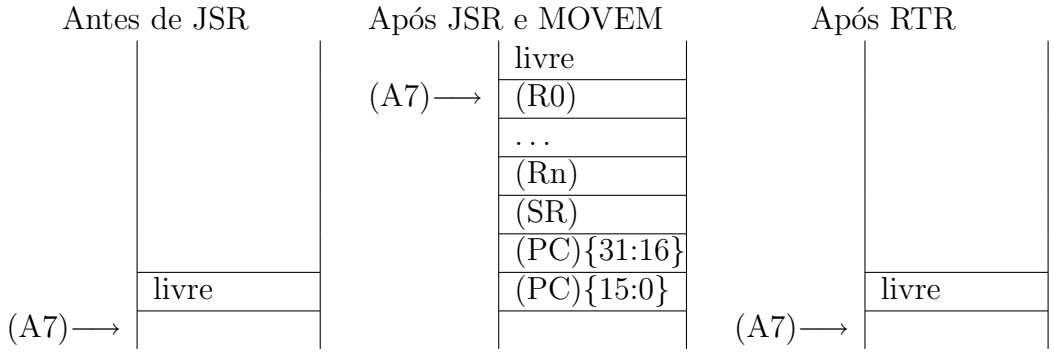


Figura 1.5: Uso da Pilha de Sistema durante a chamada a uma subrotina

1.3.13 Exercícios

Escrever subprogramas em Assembly 68000 para resolver cada um dos problemas seguintes:

1. Determinar o inteiro não negativo cuja representação decimal é uma sequência de caracteres (código ASCII) dada. A sequência está colocada a partir da posição **REP**, e o resultado deve ficar em **Val**, não devendo exceder uma palavra longa.

2. Dadas duas sequências de caracteres verificar se a primeira é subsequência da segunda. As sequências encontram-se em duas pilhas de endereços crescentes a partir das posições **SEQ1** e **SEQ2**, respectivamente. O primeiro caracter de cada sequência está no topo da pilha correspondente. Se a sequência **SEQ1** for subsequência de **SEQ2** , no bit zero[1] da posição **SIM** deve ficar o valor 1.

3. Dadas duas sequências de caracteres nas condições do exercício anterior determinar a maior subpalavra da primeira que é subsequência da segunda. Como resultado devem ser indicados em **ED** os endereços longos do primeiro e último caracter da subsequência comum. Se não existir uma subpalavra comum, no bit zero da posição **SIM** deve ficar o valor 0.

[1]Existem instruções para manipular apenas um bit de um operando

4. Comparar duas sequências de caracteres usando a ordem *lexicográfica.*[2] As sequências encontram-se em duas pilhas de endereços crescentes a partir das posições **SEQ1** e **SEQ2**, respectivamente. Supor que o primeiro byte de cada sequência (topo da pilha respectiva) contém o comprimento dessa sequência. No fim, o bit zero de D0 deve ser 1 se a primeira for maior do que a segunda, o bit um deve ser 1 se a segunda for maior, e ambos zero se as sequências forem iguais.

5. Verificar se uma dada sequência de caracteres, em código ASCII, é capícua. A sequência encontra-se numa pilha de endereços crescentes. O topo da pilha (primeira posição vazia) é apontado por A2 e a posição anterior contém o número de caracteres da sequência. Se a sequência for capícua, o bit menos significativo de D0 deve ficar com o valor 1.

6. Dada uma base b (inteiro positivo em 8 bits) da forma 2^k, em que $k \leq 4$, e a sequência de carateres da representação em binário de um inteiro (não negativo), reescrever tal sequência de forma a obter a representação base b. A sequência de caracteres encontra-se a partir da posição **rep2** devendo ser substituída pelo resultado (libertando posições desnecessárias). O registo A1 deve apontar o 1 caracter do resultado.

7. Considerar uma estrutura de dados em que cada elemento é constítuido por 3 campos: o primeiro é uma palavra chave e ocupa 2 bytes, o segundo contém um nome e ocupa 20 bytes, e o terceiro indica um grupo e ocupa 1 byte. Os elementos estão colocados em posições de memória contíguas de endereços crescentes. A primeira chave é apontada por A0, e o último elemento da sequência tem chave zero. Dada uma chave em D0 (16 bits menos significativos) pretende-se encontrar o grupo correspondente a essa chave. O resultado deve ser colocado em D0. Se não existir nenhum elemento com a chave dada, o valor do grupo será zero.

1.3.14 Operações Lógicas e para Manipular um bit

Em geral, uma variável lógica é uma variável que toma apenas dois valores. Tais variáveis são usadas quando se pretende indicar o resultado de uma operação se tal resultado só puder ser um de dois valores. São por vezes designadas por "flags". Por exemplo, o registo de condições CCR é um conjunto de flags que indica resultados de operações aritméticas. Em certas aplicações, pode ser conveniente tratar cada "bit"de um operando como uma variável lógica. Um operando pode ser visto como um conjunto de variáveis lógicas.

Existem instruções em Assembly 68000 que permitem trabalhar com 8, 16 e 32 variáveis lógicas em simultâneo, e instruções para manipular apenas uma (um bit).

[2]Um caracter a é inferior a um caracter b sse o código de a é inferior ao de b. Uma sequência caracteres s_1 é inferior (lexicograficamente) a uma sequência caracteres s_2 sse para o primeiro par de caracteres em que diferem, seja (c_1, c_2), se tem c_1 inferior a c_2; ou se tal par não existir e s_2 tiver mais caracteres. Por exemplo, $esta < este$, $estas < este$, $est < este$.

Operações Lógicas

AND — (*And Logical*) Calcula no operando destino o e-lógico desse operando com o primeiro. Sintaxe: AND.t < *ea* >,Dn ou AND.t Dn,< *ea* >. O tamanho da operação deve ser especificado, podendo ser 8, 16, ou 32 bits. Nenhum dos operandos pode ser um registo de endereço (ver nas tabelas os modos de endereçamento permitidos). Altera o registo CCR de acordo com o resultado da operação.

ANDI — (*And Immediate*) Análogo a AND mas o primeiro operando é literal. Sintaxe: ANDI.t #< *data* >,< *ea* >. Em geral, os assembladores traduzem automaticamente a instrução AND.t #< *data* >,Dn como se fosse ANDI.t #< *data* >,Dn.

ANDI — (*And Immediate to CCR*) O segundo operando é CCR. Sintaxe: ANDI.B #< *data* >,CCR

OR — (*Inclusive Or Logical*) Calcula ou-inclusivo dos operandos. Análogo a AND.

ORI — Sintaxe: ORI.t #< *data* >,< *ea* >

ORI — Sintaxe: ORI.B #< *data* >,CCR

EOR — (*Exclusive Or Logical*): Calcula ou-exclusivo dos operandos. Análogo a AND, mas o primeiro operando é sempre um registo de dados.

EORI — Sintaxe: ORI.t #< *data* >,< *ea* >

EORI — Sintaxe: ORI.B #< *data* >,CCR

NOT — (*Logical Complement*) Determina o complemento para 1. É a negação lógica do valor de cada variável que constitui o operando. Sintaxe: NOT.t < *ea* >

Máscaras

As operações lógicas podem ser usadas para efectuar certas operações de um modo mais eficiente. Por exemplo, para saber se algum dos bits 0, 1, ou 5 do registo D1 tem valor 1, pode-se colocar $23 no byte menos significativo de D2 e determinar se o resultado de AND.B D1,D2 é zero. O valor $23 corresponde à configuração 0010 0011. Ou seja, os bits 0, 1, e 5 têm o valor 1, e os restantes são zero pelo que o resultado de AND.B D1,D2 é zero sse nenhum dos bits 0, 1, e 5 do registo D1 for 1. Tal valor designa-se por *máscara*.

Operações de Deslocamento e Rotação

Estas operações "*deslocam*"*os bits de um operando um certo número de posições para a esquerda ou para a direita, e incluem: deslocamentos aritméticos (que correspondem à divisão ou produto do operando em complemento para 2 por uma potência de 2), deslocamentos lógicos, e deslocamentos cíclicos ou rotações.*

Exemplos:

- *deslocamento aritmético de uma posição à esquerda (***ASL***)*

dado	*resultado*	*efeito sobre CCR*				
00011111	00111110	*X*	*N*	*Z*	*V*	*C*
		0	*0*	*0*	*0*	*0*
10011111	00111110	*X*	*N*	*Z*	*V*	*C*
		1	*0*	*0*	*1*	*1*

- *deslocamento aritmético de uma posição à direita (***ASR***)*

dado	*resultado*	*efeito sobre CCR*				
00011111	00001111	*X*	*N*	*Z*	*V*	*C*
		1	*0*	*0*	*0*	*1*
10011111	11001111	*X*	*N*	*Z*	*V*	*C*
		1	*1*	*0*	*0*	*1*

- *deslocamento lógico de uma posição à esquerda (***LSL***); registo* **V** *fica sempre zero*

dado	*resultado*	*efeito sobre CCR*				
00011111	00111110	*X*	*N*	*Z*	*V*	*C*
		0	*0*	*0*	*0*	*0*
10011111	00111110	*X*	*N*	*Z*	*V*	*C*
		1	*0*	*0*	*0*	*1*

- *deslocamento lógico de uma posição à direita (***LSR***); registo* **V** *fica sempre zero*

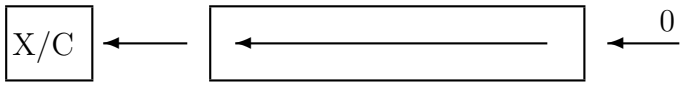
dado	*resultado*	*efeito sobre CCR*				
00011111	00001111	*X*	*N*	*Z*	*V*	*C*
		1	*0*	*0*	*0*	*1*
10011111	01001111	*X*	*N*	*Z*	*V*	*C*
		1	*0*	*0*	*0*	*1*

A figura 1.6 contém as mnemónicas de cada uma das instruções e uma descrição do efeito de cada uma.

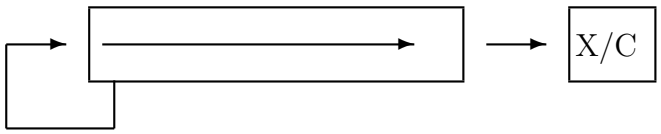
A sintaxe das instruções, os modos de endereçamento dos operandos, e os tamanhos possíveis de operandos são os indicados na tabela seguinte.

Figura 1.6: Operações de Deslocamento e Rotação

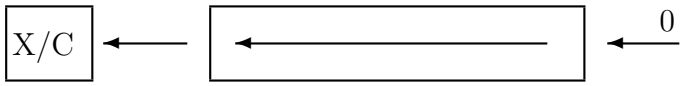
Deslocamento Artimético à esquerda: ASL



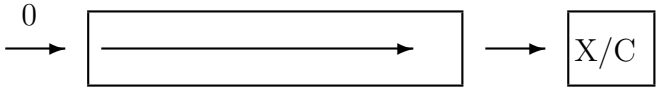
Deslocamento Aritm'etico 'a direita: ASR



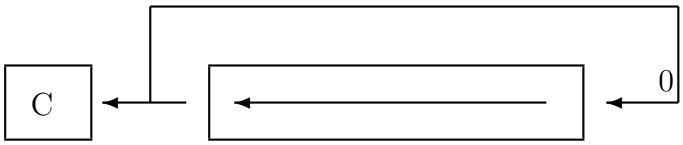
Deslocamento L'ogico 'a esquerda: LSL



Deslocamento L'ogico 'a esquerda: LSL



Rotac{c}ao 'a esquerda: ROL



Rotac{c}ao 'a direita: ROR

MNE é uma das ASL, ASR, LSL, LSR, ROR, ROL, ROXR, ROXL

Sintaxe	*Endereçamento e Tamanho de Operandos(0)*
MNE Dm,Dn	*— B,W,L*
MNE #< data >,Dn	*— B,W,L(2)*
MNE < ea >	*— all (W) but Dn,An,d(PC) to Imm*

(0) Ordem: Dn, An, (An), (An)+, -(An), d_{16}(An), d_8(An,Xi), Abs.W, Abs.L, d_{16}(PC), d_8(PC,Xi), Imm
(2) 3 bits; valores entre 1 e 8

O primeiro operando indica o número de posições do deslocamento a efectuar sobre o segundo operando, excepto para MNE < ea > em que o deslocamento é apenas de uma *posição. Para MNE Dm,Dn o deslocamento é de 0 a 63 posições sendo dado por Dm módulo 64 (isto é, resto da divisão de Dm por 64).*

Operações para Manipular apenas 1 bit

*Existem instruções que permitem alterar (*bit change*), atribuir valor 0 (*bit clear*), atribuir valor 1 (*bit set*), e testar (*bit test*) apenas um bit do operando destino. Têm uma forma geral MNE Dn,< ea > ou MNE #< bn >,< ea >. O primeiro operando indica o número do bit de < ea > a alterar ou testar. No quadro seguinte indica-se a operação correspondente a cada uma das instruções, supondo-se que b_n representa o bit a manipular:*

Instrução	*Efeito*
BCHG Dn,< ea > *BCHG #< b_n >,< ea >*	$Z = \neg b_n$ *e* $b_n \leftarrow \neg b_n$
BCLR Dn,< ea > *BCLR #< b_n >,< ea >*	$Z = \neg b_n$ *e* $b_n \leftarrow 0$
BSET Dn,< ea > *BSET #< b_n >,< ea >*	$Z = \neg b_n$ *e* $b_n \leftarrow 1$
BTST Dn,< ea > *BTST #< b_n >,< ea >*	$Z = \neg b_n$

O operando destino pode ser tomado com 32 bits se for um dos registos de dados; caso contrário será apenas um byte. O primeiro operando contém um inteiro que é tomado como o número do bit a manipular (o bit menos significativo é o bit zero). Consultar as tabelas gerais para mais detalhes.

1.3.15 Exercícios

Escrever um subprograma em Assembly 68000 para resolver cada um dos problemas seguintes.

1. *Calcular o produto de dois inteiros não negativos de 16 bits usando operações de deslocamento e soma. Os dois inteiros são dados em DO (16 bits mais significativos*

e 16 bits menos significativos), e o resultado deve ser colocado em D0 em 32 bits.

2. *Análogo ao anterior mas os valores são inteiros (não necessariamente positivos) representados em complemento para 2.*

3. *Ordenar por ordem crescente uma sequência de inteiros em 16 bits,*

   (a) *pelo* Método da Bolha*;*

   (b) *por* Inserção*.*

   A sequência encontra-se em posições de memória a partir da posição apontada por A0 (endereço 32 bits), sendo a sequência de endereços decrescente. O número de elementos da sequência é um inteiro não negativo de 32 bits dado em D0. O resultado deve ser apontado por A0 e ocupar a posição da sequência dada.

4. *Determinar a posição de um inteiro de 16 bits numa dada sequência de inteiros não negativos de 16 bits, sem repetições para cada um dos casos seguintes.*

   (a) *a sequência pode não estar ordenada;*

   (b) *a sequência está ordenada por ordem crescente.*

   A sequência encontra-se em posições de memória a partir da posição apontada por A0 (endereço 32 bits), sendo a sequência de endereços crescente. O número de elementos da sequência é um inteiro não negativo de 32 bits dado em D0. Como resultado, D0 deve conter a ordem do elemento. Se o elemento não ocorrer na sequência o bit 1 de D1 conterá o valor 1.

5. *Dadas duas sequências de inteiros positivos* **SEQ1** *e* **SEQ2** *respectivamente, sem repetições e ordenadas por ordem crescente, construir a sequência ordenada e sem repetições* **MERGE** *obtida por junção dos elementos dessas sequências. Cada sequência encontra-se em posições de memória de endereços decrescentes. Os endereços (32 bits) do primeiro elemento de cada uma* **SEQ1**, **SEQ2**, *e* **MERGE** *estão colocados por esta ordem em posições contíguas da pilha do sistema, e* **SEQ1** *é o topo da pilha imediatamente antes da chamada da sub-rotina. O primeiro elemento de cada sequência é o seu número de elementos.*

1.3.16 Processos Recursivos

Recorde que o factorial de um inteiro não negativo pode ser definido como

definição 1
$$\begin{cases} 0! = 1! = 1 \\ n! = n(n-1)! \quad se\ n \geq 2 \end{cases}$$

definição 2
$$\begin{cases} 0! = 1! = 1 \\ n! = n(n-1)\ldots 2.1 \quad se\ n \geq 2 \end{cases}$$

A primeira definição diz-se por recorrência e determina um processo recursivo *para calcular* $n!$*, enquanto que a segunda determina um processo* iterativo.

A sub-rotina seguinte permite calcular, usando recursão, o factorial de um inteiro não negativo dado em D0 em 16 bits. O resultado é calculado em D0 e tomado com 16 bits. Se o valor calculado exceder $2^{16} - 1$*, o bit 0 de D1 assinala o erro.*

```
fact    CMPI.W   #1,D0
        BLS      zero
        MOVE.W   D0,-(SP)
        SUBQ.W   #1,DO
        BSR      fact
        MULU     (SP)+,DO
        CMPI.L   #$0000FFFF,D0
        BLS      pronto
        BSET     #0,D1          ;assinala erro em D1
pronto  RTS
zero    MOVE.W   #1,D0
        RTS
```

Exercício: *Siga o programa supondo que o conteúdo inicial de DO é 5 (e também quando é 9). Qual é o menor inteiro cujo factorial não pode ser calculado?*

Como a sub-rotina se chama a si própria, a pilha do sistema é usada para guardar os valores dos registos que são alterados na chamada se esses valores forem importantes (ou seja, necessários quando se retorna).

1.3.17 Exercícios

Escrever um subprograma em Assembly 68000 para resolver cada um dos problemas seguintes.

1. *Dados dois inteiros não negativos de 16 bits* x *e* y *em D1 e D2 respectivamente, calcular recursivamente, com 32 bits, e em D1 o valor de* $f(x, y)$ *assim definido*

$$f(x,y) = \begin{cases} y & \text{se } x = 1 \\ yf(x-1,y) + 4 & \text{se } x \neq 1 \end{cases}$$

   Se ocorrer algum erro aritmético o bit 16 de D2 deve tomar valor 1.

2. *Análogo ao anterior mas o resultado deve ser dado em D2, e o erro indicado por D1.*

3. *Calcular recursivamente o valor de* $Ack(m, n)$*, sendo* m *e* n *inteiros não negativos dados em D0 e D2 em 8 bits, e Ack a função de Ackermann assim definida:*

$$\begin{array}{lcll} Ack(0,n) & = & n+1, & \text{se } n \neq 0 \\ Ack(m,0) & = & Ack(m-1,1) & \text{se } m \neq 0 \\ Ack(m,n) & = & Ack(m-1, Ack(m,n-1)) & \text{se } n \neq 0 \text{ e } m \neq 0 \end{array}$$

O resultado deve ser calculado com 16 bits em D3; se ocorrer algum erro aritmético o valor do bit 16 de D3 deve ser 1.

4. *Dado um inteiro não negativo, seja N, calcular o termo de ordem N da sucessão de Fibonnacci definida por*

$$\begin{aligned} &fib(0) = fib(1) = 1 \\ &fib(n) = fib(n-1) + fib(n-2) \quad se\ n \geq 2 \end{aligned}$$

Como deve ser dado o valor de N? Onde é colocado o resultado? Qual o parâmetro que indica erros aritméticos?

5. *Escreva uma rotina recursiva que calcule o valor da função $f(x, y)$ dados x e y, inteiros não negativos, nos registos `D0` e `D1` (em 16 bits),*

$$\begin{array}{lcl} f(0,y) & = & 4 * y \\ f(x+1,0) & = & f(x,x) + 1 \\ f(x+1,y+1) & = & f(x,y) + f(x+1,0) \end{array}$$

6. *Escreva uma rotina recursiva que calcule o valor da função $f(x, y)$ dados x e y, inteiros não negativos, nos registos `D0` e `D1` (em 16 bits),*

$$\begin{array}{lcll} f(x,0) & = & x & \\ f(0,y) & = & 2 \times f(0,y-1) & se\ y > 0 \\ f(x,y) & = & f(x-1, f(x-1,y)) & se\ y > 0,\ x > 0 \end{array}$$

O resultado deve ser calulado com 32 bits no registo `D2` e se ocorrer algum erro aritmético o valor do bit 16 de `D0` deve ser 1.

1.3.18 Estruturas de Dados

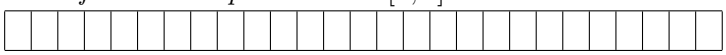
Estruturas Indexadas (*Arrays*) — vectores e matrizes

Exercício:

- *Como representar em memória uma matriz de inteiros $M[n, m]$ sendo cada inteiro representado por um byte (ou 16 bits, 32 bits, 64 bits)?*
- *Escrever uma sub-rotina para atribuir um valor ao elemento $M[i, j]$ dados i e j.*
- *Escrever uma sub-rotina para determinar a linha e coluna do elemento da matriz que é igual a um certo valor dado.*

- *Escrever uma sub-rotina para ordenar os elementos da matriz por ordem crescente, supondo que são inteiros em 8 bits e sem sinal.*

uma forma de representar $M[3,9]$ *sendo cada entrada um byte*



$x \qquad x + 9 \qquad x + 2 \times 9$

endereço de $M[i,j] = x + (i-1) \times 9 + (j-1)$, $1 \leq j \leq 9$, $1 \leq i \leq 3$

- *Escreva uma subrotina que permita calcular o produto de duas matrizes quadradas* $n \times n$.

Estruturas Ligadas — Listas e Árvores

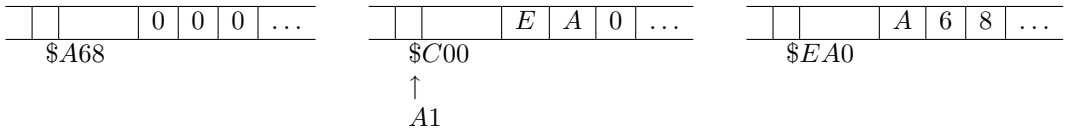
Uma lista *pode ser definida genericamente como sendo: ou a* lista vazia *ou um* par formado por um elemento de um certo tipo e uma lista. *Um tipo é formalmente definido como um conjunto de elementos.*

Uma lista pode ser implementada como uma estrutura ligada em que cada elemento tem um campo que é um apontador para o elemento seguinte, ou seja o endereço do elemento seguinte na lista. Pode considerar-se que o último elemento é o que tem no campo de endereço o valor zero (significando que não aponta nenhum outro).

Exercício

- *Suponha dada em memória uma lista em que cada elemento tem 2 campos sendo o endereço do primeiro campo sempre par. O primeiro campo de um elemento é uma sequência de caracteres 9 bytes e o segundo um endereço em 3 bytes. O registo A1 contém o endereço do primeiro elemento da lista.*

EXEMPLO DE LISTA COM 3 ELEMENTOS (apontador de 3 bytes)



$A1 \rightarrow elemento(\$C00) \rightarrow elemento(\$EA0) \rightarrow elemento(\$A68) \rightarrow 000$

1. *Determinar o endereço do primeiro elemento cujo conteúdo é igual a uma palavra de 9 caracteres dada; problema análogo para o último elemento nessas condições.*
2. *Inserir um elemento numa certa posição da lista. Supondo que o elemento a inserir ocupa posições contíguas a partir de NEL (endereço longo e par). Suponha que os 9 primeiros bytes contêm a sequência dos códigos dos caracteres desse elemento. O novo elemento deve ocupar a 3 posição na lista (ou a última se o número de elementos na lista for inferior a 3).*

3. *Suponha a lista ordenada por ordem (lexicográfica) crescente. Pretende-se inserir um elemento nas condições descritas na alínea anterior mantendo a ordenação.*

Uma **árvore binária** *pode ser implementada como uma estrutura ligada em que cada nó tem dois campos que são apontadores para os seus descendentes directos. Pode-se considerar que cada folha*

- *ou tem nesses campos (apontador) o valor zero, ou*
- *não tem tais campos.*

Exercício

- *Suponha dada em memória uma árvore binária em que cada elemento tem 4 campos sendo o endereço do primeiro sempre par:*
  - *primeiro campo — um inteiro de 16 bits em complemento para 2;*
  - *segundo campo — um caracter;*
  - *terceiro campo — o endereço em 24 bits do descendente à esquerda;*
  - *quarto campo — o endereço em 24 bits do descendente à direita.*

  As folhas e os nós são estruturas idênticas. O registo A1 contém o endereço da raíz da árvore.

  1. *Dado um caracter determinar o nó, mais à esquerda e mais próximo da raíz, que tem tal caracter no segundo campo.*
  2. *Dado um caracter determinar o nó, mais à direita e mais próximo da raíz, que tem tal caracter no segundo campo e cujo primeiro campo contém o maior inteiro.*

1.3.19 Exercícios Vários

1. *(exame) Supondo que x, y, z e w são posições de memória onde estão números de 8 bits, avaliar, sem usar instruções nem rotinas de multiplicação, a seguinte expressão:*

$$w = \begin{cases} 2y - 16z, & \text{se } x \text{ é par} \\ y + 2z, & \text{se } x \text{ é ímpar} \end{cases}$$

2. *(exame)Considere um vector em que cada um dos seus elementos tem dois campos:*

campo 1: um caracter (1 byte)
campo 2: um inteiro positivo (2 bytes)

Suponha que o primeiro elemento do vector está na posição de memória de endereço EL1 e que o número de elementos está na posição NEL. Escreva um programa que dentre os elementos cujo campo 1 é um dígito (0 a 9) determine o elemento que tem no campo 2 o menor inteiro.

3. *(exame) Escreva um programa para calcular $h(n, m)$ assim definido*

$$\begin{array}{lcll} h(0,m) & = & m+1, & \textit{se } m \geq 0 \\ h(n,0) & = & n, & \textit{se } n > 0 \\ h(n,m) & = & h(n-m,m), & \textit{se } n \geq m > 0 \\ h(n,m) & = & h(n,m-n), & \textit{se } m > n > 0 \end{array}$$

supondo **n** *e* **m** *posições de memória (1 byte) dadas contendo inteiros (sem sinal). O valor de $h(n, m)$ deverá ficar na posição* **result**.

4. *Escreva uma rotina recursiva que calcule o valor da função $f(x, y)$ dados x e y, inteiros não negativos, nos registos* `D0` *e* `D1` *(em 16 bits),*

$$\begin{array}{lcl} f(0,y) & = & 4 * y \\ f(x+1,0) & = & f(x,x)+1 \\ f(x+1,y+1) & = & f(x,y)+f(x+1,0) \end{array}$$

5. *(exame) Dado um vector em que cada elemento é de 8 bits, escrever uma sub-rotina para encontrar o maior elemento do vector e colocar na posição de memória* **R** *o valor da soma dos elementos contíguos do último máximo (pode apenas existir um se o máximo ocorrer num dos extremos). Supor que o primeiro elemento e o número de elementos do vector se encontram na posições* **AR** *e* **LEN**, *respectivamente.*

6. *Escreva um programa que calcule todos os números primos até n, usando o* crivo de Eratóstenes*:*

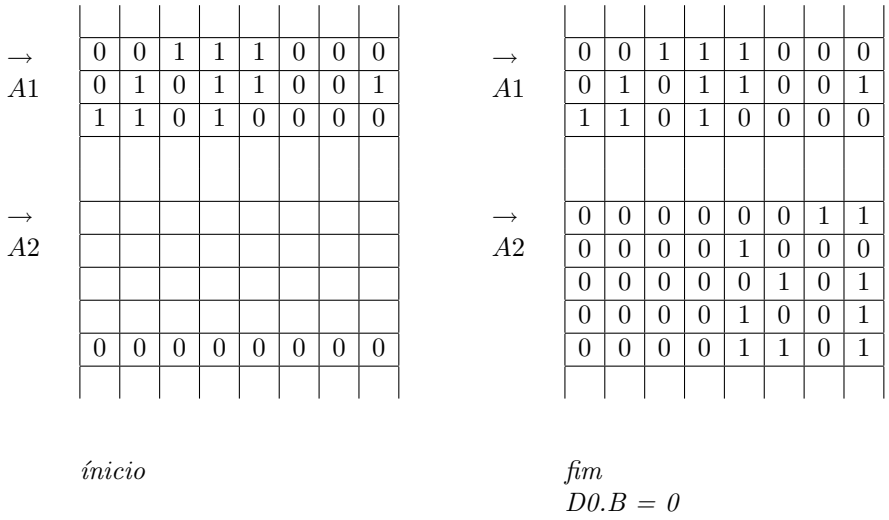
   - *começe por construir uma lista com todos números ímpares de 3 até n;*
   - *"marque" todos os múltiplos de 3 maiores que 3 (i.e. escreva por cima um código especial — ex.: 0);*
   - *"marque" todos os múltiplos de 5 maiores que 5;*
   - *proceda da mesma forma para cada número na lista ainda não marcado; os números que restam são primos.*

   Assuma que n está em `D0` *e que pode construir a lista com os números (*`.W`*) num espaço de memória a partir da posição* `LISTA`.

7. *Escrever uma sub-rotina para efectuar a concatenação de duas sequências de caracteres. Os operandos são apontados por A0 e A1 e o resultado deve ficar no espaço apontado por A2 que é terminado por 0. Caso o resultado não caiba, D0 deve tomar valor 1; senão D0 deve ser 0. Suponha que qualquer das sequências de caracteres dadas termina pelo caracter de código 0.*

8. *No sistema de numeração BCD (Binary Coded Decimal) cada dígito decimal é representado em 4 bits, e cada inteiro (sem sinal) é representado pela sequência de bits obtida por conversão de cada um dos seus dígitos decimais para BCD. Por exemplo, 3859 = 0011 1000 0101 1001(bcd). Escrever uma sub-rotina para resolver cada um dos problemas seguintes.*

   (a) *Dada uma sequência de bytes, cada um dos quais contendo dois dígitos BCD, descompactá-la (isto é, re-escrever a sequência de forma que cada byte contenha apenas um dígito BCD). Supor que a posição apontada por A1 contém os dígitos mais significativos da sequência dada, e que o fim da sequência é assinalado por um qualquer valor em 4 bits que não representa nenhum dígito BCD. O resultado deve ficar no espaço apontado por A2 que é terminado por 0. Caso a sequência descompactada não caiba no espaço disponível o byte menos significativo de D0 deve ser $FF; senão, $00. Sugestão: Utilize a instrução de mnemónica AND para descompactar cada byte, e a de mnemónica Scc (*Set According to Condition*) para assinalar o erro.*



   (b) *Dada uma sequência de bytes, cada um dos quais contendo um dígito BCD, compactá-la (isto é, reescrever a sequência de forma que cada byte contenha dois dígitos BCD). Supor que a posição apontada por A1 contém inicialmente os dígito mais significativo da sequência dada e que no fim contém os dois dígitos mais significativos da sequência compactada. Supor ainda que o resultado satisfaz as condições descritas em 38.1 para a sequência que é dada.*

9. *(exame) Escreva uma rotina que determine se uma sequência de caracteres* `s`*, com um comprimento múltiplo de 6, verifica as seguintes condições: os dois primeiros caracteres são iguais ao sexto; o terceiro é igual ao sétimo; o sétimo é igual ao oitavo e ao décimo segundo; o nono é igual ao décimo terceiro... e o padrão repete-se enquanto houver caracteres. A sequência termina com um caracter de código zero. Por exemplo, a sequência*

`aacfgacc1uuc11B561BBBjkBBBhUUBhhiP0h`

é da forma pretendida.

A sequência deve ser percorrida uma só vez e no fim o registo `D0` *deve conter o valor* `1` *se a sequência verifica as condições e o valor* `0`*, caso contrário.*

Suponha que s está em memória, e que ao entrar na rotina o respectivo endereço está no registo `A0`*. Não necessita de verificar o comprimento da sequência. Indique o significado dos contéudos dos restantes registos que utilizar.*

10. *(exame) Considere uma sequência formada por letras minúsculas (de 'a' a 'z') terminada pelo caracter '0'. Escreva uma rotina que a transforme do modo seguinte: sempre que um caracter ocorra consecutivamente entre* ***duas*** *e* ***nove*** *vezes, essa subsequência é substituída pelo caracter correspondem ao número de ocorrências seguido do caracter em causa. Por exemplo* `ffloooooppppddddddddddddddddadddd0` *será transformada em* `2fl5o3p9d5da4d0`*. Note que se um caracter ocorrer mais de 9 vezes seguidas, consideram-se subsequências distintas.*

    Suponha que a sequência está em memória e ao entrar na rotina o respectivo endereço está no registo `A0`*. Descreva brevemente o algoritmo a utilizar e o significado dos conteúdos dos registos que utilizar.*

11. *(exame) Escreva uma rotina que dada uma matriz de inteiros (16 bits) M de* $(n+1) \times (n+1)$*, (n inteiro não negativo, de 8 bits) permita determinar a soma dos valores da diagonal principal (elementos em posições* $M[i,i], 0 \leq i \leq n$*). Deve indicar explicitamente como é que a matriz está guardada em memória (incluindo a sua dimensão) e como é guardado o resultado. Se o endereço do primeiro elemento da matriz se encontrar no registo* `A0`*, pode considerar que os elementos da matriz estão guardados na memória, da seguinte forma:*