U.PORTO

Faculdade de Engenharia

Microprocessadores

Apoio às Aulas Práticas

4ª Edição, Setembro 2005
Departamento de Engenharia Electrotécnica e de Computadores

Primeiros Passos na Programação em Linguagem Assembly

João Paulo Sousa
jpsousa@fe.up.pt

Setembro 2005

Conteúdo

1 Objectivos **2**

2 Introdução **2**

3 Programação em linguagem *assembly* **3**
3.1 Vantagens . . . 3
3.2 Desvantagens . . . 3
3.3 Conclusão . . . 3

4 Modelo de programação **4**
4.1 Registos principais da família i51 . . . 4
4.2 Grupos de instruções da família i51 . . . 4
4.3 Algumas regras de sintaxe e modos de endereçamento da família i51 . . . 5

5 O assemblador **5**
5.1 Caraterísticas gerais . . . 6
5.2 Nomes simbólicos e operadores simples . . . 6
5.3 Segmentos . . . 7
5.3.1 Segmentos absolutos de código . . . 7
5.3.2 Segmentos absolutos de dados . . . 7
5.4 Formato das listagens . . . 8

6 Problemas **8**

1 Objectivos

Familiarização com o modelo de programação simplificado da família 51 da Intel, com a sintaxe da sua linguagem *assembly* e com alguns comandos do assemblador.

2 Introdução

Trabalhar segundo o paradigma do *programa residente*[1] obriga, como o próprio nome indica, a armazenar em memória um programa (conjunto de ordens) num formato que o microprocessador entenda. No seu nível mais baixo, essas ordens não são mais do que códigos binários armazenados em memória que o microprocessador vai extraindo, interpretando e executando. Um programa nessa forma diz-se estar em *código máquina*. Por diversas razões programar directamente em código máquina não é viável mesmo para programas pequenos. De facto, constata-se facilmente que um programa em código máquina:

- é muito difícil de perceber e depurar,
- não descreve a tarefa a executar de um modo inteligível,
- é muito extenso e demora muito tempo a escrever,
- é muito sensível a distracções do programador que normalmente conduzem a erros muito difíceis de detectar.

Os dois últimos problemas podem ser atenuados escrevendo os códigos em hexadecimal em vez de binário; mas uma melhoria substancial é atribuir um nome (mnemónica) a cada código de instrução de modo a tornar o programa minimamente inteligível – chama-se a isso programar em linguagem *assembly*. Na figura 1 um mesmo programa aparece escrito em linguagem *assembly* da família 51 e também em código máquina (binário e hexadecimal).

Linguagem *assembly*	Bin	Hex
`MOV A,#64`	`01110100`	`74`
	`01000000`	`40`
`ADD A,#10`	`00100100`	`24`
	`00001010`	`0A`

Figura 1: Programa em *assembly* e código máquina

Por simples inspecção facilmente se conclui que é muito mais fácil programar em linguagem *assembly* do que directamente no código máquina de um microprocessador.

De acordo com o convencionado pelo fabricante, os códigos `74h` e `24h` representam, respectivamente, as instruções `MOV A,#n` (copiar para o registo A o valor n) e `ADD A,#n` (adicionar ao registo A o valor n, ficando o resultado em A), onde `n` representa um valor constante e `A` o registo acumulador. Por outro lado, os códigos `40h` e `0Ah` representam os operandos das instruções; pode assim concluir-se que o programa adiciona ao valor 64 o valor 10 ficando o resultado da adição no registo acumulador. Note-se a necessidade de converter os números 64 (40h) e 10 (0Ah) para binário (ou hexadecimal) quando se programa directamente em código máquina.

[1] cfr. Aulas teóricas da primeira semana

3 Programação em linguagem *assembly*

A linguagem assembly é, pois, o nível mais baixo em que se pode programar com alguma comodidade. A tradução da linguagem *assembly* de cada microprocessador para o código máquina correspondente pode ser feita à mão, mediante uma tabela de conversão, mas normalmente é feita recorrendo a um *assemblador*, ferramenta que na maior parte dos casos é oferecida pelo fabricante do microprocessador.

Nas secções seguintes apresentam-se as vantagens e desvantagens de programar em linguagem *assembly*; um bom assemblador atenua algumas das desvantagens da linguagem mas há outras que lhe são intrínsecas.

3.1 Vantagens

A principal vantagem de programar em linguagem *assembly* é poder optimizar o código de modo a aproveitar ao máximo as características particulares do *hardware* onde vai ser executado, conseguindo assim melhores resultados quer em tempo de execução quer em tamanho de código gerado.

Outra vantagem é a existência de assembladores para todos os microprocessadores, muitas vezes oferecidos pelos fabricantes, pelo que é sempre possível programar em *assembly*, qualquer que seja o microprocessador escolhido. O mesmo não acontece com linguagens de alto nível, onde nem sempre é possível encontrar um compilador adequado para um determinado microprocessador.

3.2 Desvantagens

A principal desvantagem de uma linguagem de baixo nível é a grande desproporção que existe entre a complexidade do seu conjunto de instruções e as tarefas que o microprocessador normalmente é chamado a executar. Esta desproporção obriga o programador a decompor cada tarefa num conjunto de operações elementares que, além de ser um processo demorado e sujeito a erros, não ajuda a manter o código estruturado.

Outra desvantagem é a necessidade de conhecer em detalhe o modelo de programação do microprocessador, nomeadamente no que se refere aos registos de trabalho disponíveis, registos privilegiados ou especiais e registo de estado. Como consequência desta dependência relativamente aos detalhes internos de um microprocessador, a portabilidade dos programas é muito reduzida.

3.3 Conclusão

A tendência actual é a favor de uma programação mista, usando principalmente linguagens de mais alto nível (C em particular) e recorrendo à linguagem *assembly* apenas em rotinas onde a eficiência do código seja o objectivo principal a atingir. Esta tendência explica-se por três motivos:

- a pressão do mercado obriga a encurtar o tempo de desenvolvimento e a aumentar a facilidade de manutenção do código.
- existem actualmente compiladores de C para a maioria dos microprocessadores, alguns até de domínio público,
- os avanços na microelectrónica permitem que a rapidez de execução se consiga facilmente por aumento da frequência de funcionamento

Quando se está a estudar o funcionamento interno de um microprocessador o *assembly* é, no entanto, a linguagem mais adequada pelo que será a utilizada nesta disciplina.

4 Modelo de programação

O modelo de programação de um microprocessador descreve os recursos disponíveis para o programador: registos de trabalho, registos especiais, eventuais agrupamentos de registos, instruções e modos de endereçamento disponíveis, etc. Para começar a escrever pequenos programas basta conhecer os principais registos e instruções de um microprocessador. As secções seguintes apresentam o modelo de programação da família 51 da Intel.

4.1 Registos principais da família i51

A arquitectura base da família 51 da Intel disponibiliza um número apreciável de registos dos quais se destacam, numa primeira abordagem, os seguintes:

- Registos de trabalho – R0 a R7
- Registos privilegiados – A (acumulador) e B
- Registo de estado – PSW (*program status word*)

Os registos A e B são privilegiados no sentido de que existem algumas instruções que só podem ser executadas neles, por exemplo as operações aritméticas só podem ser efectuadas sobre o acumulador. O registo de estado (PSW) dá, entre outras, indicações sobre a paridade do valor contido no acumulador, se houve ou não transporte e/ou *overflow* na última operação aritmética efectuada, etc.

4.2 Grupos de instruções da família i51

As instruções disponíveis na família i51 podem dividir-se em cinco grupos consoante a sua função:

1. Instruções de movimentação de dados
2. Instruções aritméticas
3. Instruções lógicas
4. Instruções de salto e chamada de subrotinas
5. Instruções booleanas

As instruções de **movimentação de dados** permitem *copiar* valores: de um registo para memória, da memória para um registo, etc.

As instruções **aritméticas** permitem efectuar as quatro operações aritméticas elementares considerando ou não a existência de eventuais transportes. É sempre necessário recorrer ao registo acumulador e, por vezes, também ao registo B.

As instruções **lógicas** permitem efectuar operações lógicas elementares assim como rotações de bits para a esquerda ou para a direita. Funcionam também exclusivamente com o acumulador.

As instruções de **salto e chamada** de subrotinas permitem alterar a ordem de execução de um programa de forma condicional ou incondicional.

As instruções **booleanas** permitem manipular bits individualmente. A maior parte delas obriga a utilizar um dos bits do registo de estado – a *flag* CY (*carry*) – que funciona para o processamento booleano como o acumulador para o processamento aritmético e lógico.

O resumo de instruções oficial de um dos actuais fabricantes desta família [1, páginas 13..16] ou o resumo [2] dão uma panorâmica geral das instruções disponíveis pelo que, um ou outro, deve ser estudado em pormenor.

4.3 Algumas regras de sintaxe e modos de endereçamento da família i51

A forma geral de uma instrução assembly da família 51 é

```
mnemónica [operando1[,operando2[,operando3]]]
```

ou seja, há instruções com 3, 2, 1 ou nenhum operando. De acordo com o estabelecido pelo fabricante original um operando pode ser:

- um número, representando um endereço de memória (endereçamento directo),
- o nome de um registo (endereçamento registo),
- um valor constante, se precedido do carácter `#` (endereçamento imediato),
- um apontador, se precedido do carácter `@` (endereçamento indirecto).

Estas regras de sintaxe entendem-se melhor com a apresentação de alguns exemplos simples:

```
MOV R5,40h          ; Copia para o registo R5 o conteúdo da posição de memória 64.
MOV R5,#40h         ; Coloca no registo R5 o valor 64.

ADD A,R5            ; Adiciona ao acumulador o registo R5. Resultado no acumulador.
ORL A,#10010001b    ; Faz o OR lógico do acumulador com o valor 91h.
                    ; O resultado fica no acumulador.
LJMP 4358h          ; Continua a execução do programa no endereço 4358h.

; Copia para o acumulador...
MOV A,R0            ; ...o que está em R0
MOV A,@R0           ; ...o conteúdo da posição de memória apontada por R0.
```

Note-se que dos registos de trabalho R0 a R7, apenas R0 e R1 podem assumir o papel de apontadores. Quando o manual de programação [1] emprega a designação genérica *Rn* está a referir-se aos registos R0 a R7 mas quando usa a designação *Ri* está a referir-se apenas aos registos R0 e R1.

5 O assemblador

Uma ferramenta essencial para programar em linguagem *assembly* é o assemblador. Os primeiros assembladores pouco mais faziam do que a tradução para código máquina mas os mais recentes têm muitas outras capacidades, nomeadamente:

- permitem atribuir nomes simbólicos a endereços de memória, dispositivos de entrada saída, variáveis e grupos de instruções,
- permitem trabalhar em diversas bases de numeração bem como converter caracteres nos seus códigos ASCII,

- permitem efectuar cálculos aritméticos simples com valores constantes ou nomes simbólicos,
- permitem definir os endereços de memória onde o programa e os dados irão ser armazenados,
- permitem reservar áreas de memória para armazenamento temporário de informação,
- permitem a construção e utilização de bibliotecas de funções, ajudando assim a programar de modo modular e a reutilizar código já escrito em *assembly* ou noutras linguagens,
- permitem a configuração dos parâmetros que alteram a geração de código máquina e o formato das listagens produzidas.

Existem actualmente diversos assembladores comerciais e de domínio público para a família 51 da Intel. Um dos melhores é o da KEIL, disponível em versão de demonstração (mas 90% funcional) nas salas de computadores e no CD que acompanha o livro recomendado como bibliografia principal. Nas secções seguintes apresenta-se um resumo das suas capacidades.

5.1 Caraterísticas gerais

Não há distinção entre maiúsculas e minúsculas. Qualquer texto precedido do carácter '`;`' é considerado comentário; um comentário prolonga-se sempre até ao fim da linha. O fim do código fonte é indicado pelo comando `END`; qualquer texto que apareça depois desse comando é ignorado pelo assembler.

5.2 Nomes simbólicos e operadores simples

Uma das grandes vantagens de utilizar um assemblador é poder definir nomes simbólicos para variáveis, constantes e endereços de programa. Para as as variáveis essa definição faz-se com o comando `DS` (*define storage*) e para as constantes com o comando `EQU` (*equate*) cuja sintaxe se depreende do exemplo apresentado abaixo, retirado de um programa de controlo de uma máquina de encher e empacotar garrafas:

```
; === Definição de constantes ============================================
Garrafas    EQU 12          ; Número de garrafas por caixa
tempo       EQU 250         ; Tempo de engarrafamento (ms)
V1          EQU 8000h       ; Endereço de E/S da válvula 1
V2          EQU V1+1        ; Endereço de E/S da válvula 2
V3          EQU V1+2        ; Endereço de E/S da válvula 3

; === Declaração de variáveis ============================================
Contador:   DS  1           ; Reserva um byte em memória
Total:      DS  2           ; Reserva dois bytes em memória
```

No comando `DS` note-se a necessidade de '`:`' imediatamente a seguir ao nome.

A definição de nomes simbólicos para endereços de programa é feita implicitamente ao colocar uma etiqueta antes da instrução que se quer referenciar:

```
        .
        .
init:   mov a,#100
        dec a
        .
        .
```

5.3 Segmentos

Ao programar em *assembly* é possível escolher os endereços onde ficarão as variáveis e as diferentes partes de um programa. Essa escolha é feita pela definição de segmentos de memória. Existem duas classes de segmentos de memória – absolutos e relativos (ou recolocáveis), interessando, numa primeira abordagem, referir apenas os segmentos absolutos. Um segmento absoluto é uma zona de memória com endereço inicial conhecido.

5.3.1 Segmentos absolutos de código

Um segmento absoluto de código é uma zona de memória com endereço inicial conhecido onde estão instruções do programa e/ou operandos com valor constante. Este tipo de segmento é definido recorrendo ao comando `CSEG` cuja sintaxe se depreende sem dificuldade do exemplo apresentado abaixo:

```
; === Programa principal ================================================
            CSEG AT 0000h   ; Segmento com início no endereço 0000h
            mov a,#255      ; Move o valor 255 para o acumulador
            mov r0,a        ; r0=255
            mov r1,a        ; r1=255
            .
            .
 ; --- Rotinas de E/S -----------------------------------------------------
            CSEG AT 0080h   ; Segmento com início no endereço 0080h
            orl a,#7fh
            cpl a
            .
            .
            end             ; Fim do código fonte
```

Neste exemplo, o programa está distribuído por dois segmentos de código: um, com início no endereço 0000h, para o programa principal e outro, com início no endereço 0080h, onde reside um conjunto de rotinas auxiliares.

5.3.2 Segmentos absolutos de dados

Um segmento absoluto de dados é uma zona de memória com endereço inicial conhecido onde estão alojadas variáveis. Este tipo de segmento é definido recorrendo, entre outros, ao comando `DSEG` cuja sintaxe se depreende do exemplo apresentado abaixo:

```
; === Declaração de variáveis ===========================================
            DSEG AT 40h     ; Segmento com início no endereço 40h
cont:       DS 1            ; Variável cont ocupa 1 byte

; === Programa ==========================================================
            CSEG AT 0000h
            mov  cont,#100  ; cont=100
            .
            .
            end             ; Fim do código fonte
```

Em programas complexos a utilização de segmentos absolutos é uma tarefa ingrata pois é necessário garantir que não haja sobreposição entre eles. Para evitar problemas e facilitar a reutilização de código é recomendável trabalhar, sempre que possível, com segmentos relativos ou recolocáveis. A definição de segmentos relativos será, no entanto, abordada mais tarde.

5.4 Formato das listagens

O assemblador da KEIL permite definir vários parâmetros de formatação para as listagens que gera. Se nada for dito em contrário, as listagens são formatadas de modo a evidenciar o endereço de memória em que cada instrução começa e os códigos (e operandos) de cada instrução. Um exemplo é mais elucidativo do que muitas explicações:

```
LOC  OBJ            LINE     SOURCE

----                   1             dseg at 40h
0040                   2     cont:   ds 1               ; cont ocupa 1 byte
                       3
----                   4             cseg at 0
0000 754064            5             mov     cont,#100  ; cont=100
0003 1540              6     loop:   dec     cont       ; cont=cont-1
0005 E540              7             mov     a,cont     ; a=cont
0007 70FA              8             jnz     loop       ; Salta, isto é, repete tudo, se A<>0
                       9
                      10             end
```

Examinando a 1ª coluna da listagem (endereços) facilmente se percebe que a variável `cont` ficou colocada no endereço 40h e que o programa começa efectivamente no endereço 0000h. Examinando 2ª coluna (códigos gerados) pode concluir-se, por exemplo, que a primeira instrução foi codificada em três bytes (75h, 40h, 64h) – o primeiro é o código da instrução propriamente dita, o segundo, o endereço do primeiro operando da instrução (variável cont) e o terceiro, o valor do segundo operando da instrução, neste caso o valor 100.

6 Problemas

Apresenta-se de seguida um conjunto de problemas simples que podem ser resolvidos recorrendo a instruções aritméticas, lógicas e de movimentação de dados. Recorra aos documentos de apoio necessários [1, páginas 13..16], [2] para conhecer as instruções disponíveis para os resolver.

1. Escreva programas para efectuar as seguintes operações:
   (a) Copiar ***R5*** para ***A***
   (b) Copiar ***R5*** para ***R3***
   (c) Trocar ***R5*** com ***R3*** sem estragar mais nenhum registo
2. Escreva um programa que coloque a zero os registos ***R0*** e ***R1*** e de seguida calcule o quadrado do valor que está contido no acumulador. O resultado deverá ficar em ***R1*** (parte mais significativa) e ***R0*** (parte menos significativa).
3. Considere as variáveis ***montante***, ***jusante1*** e ***jusante2*** declaradas no segmento de dados apresentado:

```
; === Variáveis ============================================================
            DSEG AT 50h   ; Segmento de dados
montante:   DS 1          ; Variável montante ocupa 1 byte
jusante1:   DS 1          ; Variável jusante1 ocupa 1 byte
jusante2:   DS 1          ; Variável jusante2 ocupa 1 byte
```

   Apresente, num segmento de código absoluto com início no endereço 0000h, dois modos diferentes (CPL, XRL) de negar o conteúdo da variável ***montante***, ficando os resultados nas variáveis ***jusante1*** e ***jusante2***.

4. Traduza à mão, para código máquina, a sua solução para o problema anterior. Nota: será necessário utilizar o manual de programação [1] para responder a esta questão.

5. Apresente dois modos diferentes (MUL, RLA) de multiplicar por 4 o valor da variável ***velocidade*** guardada na posição de memória com o endereço 60h.

6. Considerando que ***mem[x]*** representa a posição de memória com endereço ***x***, escreva programas para efectuar as seguintes operações:

   (a) ***mem[62] = mem[61] − mem[60]***

   (b) ***mem[61]*** $= 3 \times$ ***mem[60]***, com ***mem[60]<86***. Porquê?

7. Considere duas variáveis – ***minutos*** e ***segundos*** – guardadas em posições de memória com endereços consecutivos. Declare as variáveis num segmento de dados e escreva um programa para efectuar a operação ***segundos = 60 × minutos***, podendo ***minutos*** variar entre 0 e 200.

8. Declare a variavel ***x*** num segmento de dados absoluto com início no endereço 50h e escreva um programa que calcule ***x = − x*** (complemento para 2) considerando que:

   (a) ***x*** é de 8 bits

   (b) ***x*** é de 16 bits

9. Escreva um programa que calcule ***x = x + y*** para variáveis de 16 bits. Declare as variáveis num segmento de dados absoluto com início no endereço 40h.

10. Declare ***xx*** e ***yy*** num segmento de dados absoluto com início no endereço 48. Escreva um programa que copie os 4 bits mais significativos de ***xx*** para os 4 bits menos significativos de ***yy***, considerando que são variáveis de 8 bits. Os 4 bits mais significativos de ***yy*** devem ser colocados a zero.

11. Considere que ***temptot*** contém a soma das medições de temperatura efectuadas em 16 pontos diferentes de uma estufa. Apresente 3 modos diferentes (DIV, RRA, SWAP) de calcular a temperatura média no interior da estufa. Suponha que está a trabalhar com variáveis de 8 bits e o resultado é guardado em ***tempmed***. Declare todas as variáveis num segmento de dados absoluto com início no endereço 35h.

Referências

[1] Philips semiconductors; *80C51 family programmer's guide and instruction set*; Setembro de 1997.

[2] Ferreira, José Manuel; *Resumo das instruções do 80C51*; FEUP, Setembro de 2000.