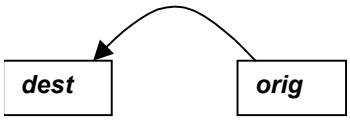
Instruções de movimentação de dados

Instrução MOV

O 8051 possui instruções que permitem copiar o conteúdo de um registrador ou localidade para outro registrador ou localidade de memória. Nas seções anteriores foram vistos os registradores do 8051. Para que a informação contida em um registrador possa ser "movimentada" para outro registrador ou para a memória usa-se a instrução:

MOV ***dest,orig***



Significado: Copia o conteúdo de origem do para destino.

Vejamos alguns exemplos:

`MOV A,B`	copia o conteúdo do registrador B para o registrador A.
`MOV R1,R5`	copia o conteúdo de R5 em R0.

Os exemplos anteriores mostram a forma como é feita a cópia de um registradora para outro. Entretanto, existem outras formas de movimentação de dados. Além de registradores, podem envolver endereços da memória RAM interna. Essas formas são vistas a seguir:

Modos de movimentação

No 8051, existem 3 formas básicas de movimentação de dados:

1. modo imediato
2. modo direto
3. modo indireto

1) IMEDIATO

```
MOV A,#03h
```

Coloca no registrador A o valor 03h. Para dizer que 03h é um valor imediato, deve ser precedido por "grade" (#).

Podemos observar que o valor de origem está contido na própria instrução.

2) DIRETO

```
MOV A,03h
```

O valor 03h agora é um endereço da memória RAM interna e o seu conteúdo é que será colocado no registrador A.

Se tivermos, por exemplo, o byte 4Dh no endereço 03h, esse valor (4Dh) será colocado no registrador A.

Podemos observar que o valor de origem vem da memória RAM interna.

3) INDIRETO

```
MOV A,@R0
```

O µP toma o valor contido no registrador R0, considera-o como sendo um endereço, busca o byte contido nesse endereço e coloca-o em A.

Supondo que R0 contém 03h, o µP pega o valor de R0 (03h), considera esse valor como sendo um endereço, busca o byte contido no endereço 03h (4Dh, por exemplo) e coloca esse byte em A (A=4Dh).

Podemos observar que o valor de origem também vem da memória RAM interna.

IMPORTANTE!!!
Esse modo usa somente os registradores RO e R1. (@R0 e @R1).

EXERCÍCIOS

para os exercícios a seguir, considere a memória interna com o seguinte conteúdo:

END.	CONTEÚDO
0000	10 00 02 FC 49 64 A0 30
0008	1D BC FF 90 5B 40 00 00
0010	00 00 01 02 03 04 05 06
0018	07 08 09 0A FF FE FD FC
0020	FB FA A5 5A DD B6 39 1B
...	...

Qual é o resultado das seguintes instruções:

a)
```
MOV A,#7EH
```

b)
```
MOV A,03H
```

c)
```
MOV A,0CH
```

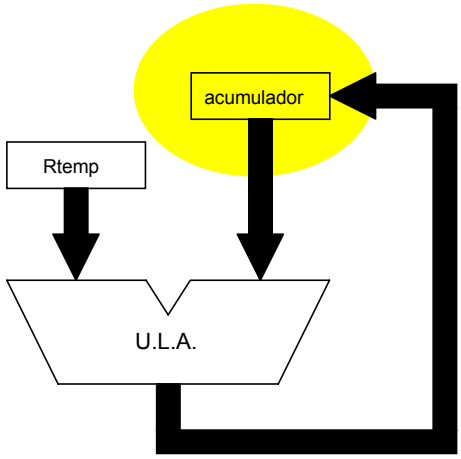
d)
```
MOV R0,#1CH
MOV A,R0
```

e)
```
MOV R1,1BH
MOV A,@R1
```

f)
```
MOV R0,#12
MOV R1,#0EH
MOV A,@R0
MOV @R1,A
```
Obs.: 12 está em decimal. 12 = 0CH

Instruções Lógicas e Aritméticas

O 8051 possui uma unidade que realiza operações lógicas e aritméticas (ULA - Unidade Lógica e Aritmética) que realiza funções de soma, subtração, multiplicação e divisão, AND, OR, XOR, etc.



Todas as operações precisam de dois parâmetros: Por exemplo: para somar, são necessários os dois números que serão somados. 2 + 5 é um exemplo de soma onde existem dois parâmetros: 2 e 5.

Mas, como se sabe, o acumulador é um dos registradores empregados nessa unidade. Portanto, o acumulador é obrigatoriamente um dos parâmetros e o resultado é colocado de volta no acumulador. No exemplo acima, o valor 2 estaria no acumulador, ocorreria a instrução de soma com o valor 5 e o resultado (7) seria colocado no acumulador no lugar do valor 2.

1. INSTRUÇÃO DE SOMA

```
ADD A,parâm  ;"add"
```

Realiza a soma A+*parâm* e coloca o resultado em A. O valor anterior de A é perdido. Exemplos:

```
ADD A,#03h   ;A+3 → A
ADD A,03h    ;A+"end.03h" → A
ADD A,B      ;A+B → A
ADD A,R0     ;A+R0 → A
ADD A,@R0    ;A+"end.@R0" → A
```

A instrução de soma afeta o registrador de 1 bit CARRY. Sempre que o resultado for um número de mais de 8 bits (um valor acima de FFH) a ULA coloca o "vai-um" automaticamente no CARRY. Isso significa que,

quando o valor "extrapolar" 8 bits o CARRY=1. Quando a soma não extrapolar, o CARRY=0.

2. INSTRUÇÃO DE SOMA COM CARRY

Parecida com a instrução anterior, a soma com carry pode utilizar o CARRY gerado do resultado anterior junto na soma:

```
ADDC A,parâm ;"add with carry"
```

Realiza a soma A + *parâm* + CARRY e coloca o resultado em A. Como o CARRY é um registrador de um único bit, o seu valor somente pode ser 0 ou 1. Então, quando CARRY=0, o efeito das instruções ADD e ADDC é o mesmo.

É muito importante notar que, as operações ADD e ADDC podem resultar valores acima de FFH e, com isso, afetar o CARRY como resultado da soma. A instrução ADDC usa o CARRY na soma. Portanto, o valor contido no CARRY antes da operação ADDC pode ser diferente do valor após a operação.
Exemplo: supondo A=F3H e CARRY=0:

```
ADDC A,#17H  ;F3H+17H+0=10AH. A=0AH e CARRY=1
ADDC A,#17H  ;0AH+17H+1=22H.  A=22H e CARRY=0
ADDC A,#17H  ;22H+17H+0=39H.  A=39H e CARRY=0
```

3. INSTRUÇÃO DE SUBTRAÇÃO COM CARRY

Por razões técnicas, não seria possível implantar ambos os tipos de subtração. Por ser mais completa, foi implantada a subtração com carry.

```
SUBB A,parâm ;"subtract with borrow"
```

Realiza a subtração A – *parâm* – CARRY e coloca o resultado em A.

No caso da subtração, o resultado pode ser positivo ou negativo. Quando o resultado for positivo, CARRY=0. Quando for negativo, CARRY=1.

No caso de se desejar fazer uma subtração simples, deve-se assegurar que o valor do CARRY seja 0 antes da instrução para não afetar o resultado. Por isso, é comum usar a instrução:

```
CLR C        ;"clear carry"
```

Exemplo:

```
CLR C        ;carry=0
SUBB A,B     ;A-B-0 = A-B
```

INSTRUÇÕES LÓGICAS

As instruções lógicas trabalham com os bits dos parâmetros. A operação é feita bit-a-bit, ou seja, uma operação do tipo AND, faz um AND lógico do primeiro bit do parâmetro 1 com o primeiro bit do parâmetro 2, o segundo com o segundo, e assim por diante.

1. INSTRUÇÃO AND LÓGICO

```
ANL A,parâm  ;"and logic"
```

Faz um AND lógico bit-a-bit de A com *parâm*. O resultado é colocado em A.

Exemplo: supondo A=9EH e B=3BH, a operação

```
ANL A,B      ;9EH "and" 3BH = 1AH
```

pois:

Função lógica AND.

A	B	L
0	0	0
0	1	1
1	0	1
1	1	1

```
        9EH = 10011110
AND     3BH = 00111011
-----------------------
        1AH = 00011010
```

2. INSTRUÇÃO OR LÓGICO

```
ORL A,param  ;"or logic"
```

Faz um OR lógico bit-a-bit de A com *parâm*. O resultado é colocado em A.

Exemplo: supondo A=9EH e B=3BH, a operação

```
ORL A,B      ;9EH "or" 3BH = __H
```

pois:

Função lógica OR.

A	B	L
0	0	
0	1	
1	0	
1	1	

```
        9EH = 10011110
OR      3BH = 00111011
-----------------------
          H =
```

3. INSTRUÇÃO EXCLUSIVE-OR LÓGICO

```
XRL A,param  ;"exclusive-or logic"
```

Faz um XOR lógico bit-a-bit de A com *parâm*. O resultado é colocado em A.

Exemplo: supondo A=9EH e B=3BH, a operação

```
XRL A,B      ;9EH "xor" 3BH = __H
```

pois:

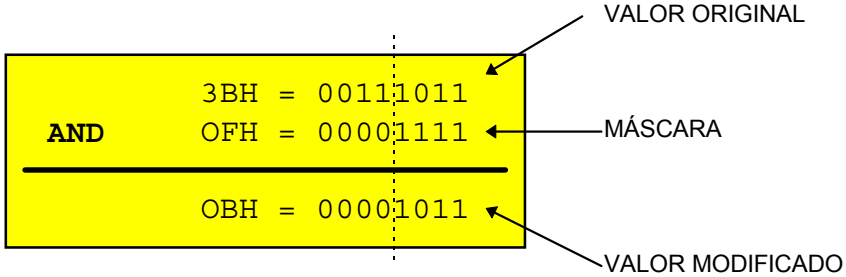
Função lógica XOR.

A	B	L
0	0	
0	1	
1	0	
1	1	

```
        9EH = 10011110
XOR     3BH = 00111011
-----------------------
         H =
```

MÁSCARAS LÓGICAS

A idéia de máscaras lógicas surgiu da observação do comportamento das operações lógicas. Como foi visto anteriormente, não é difícil realizar uma operação lógica entre dois parâmetros; esses parâmetros podem ser chamados **entradas**, e o resultado **saída**. Se mudarmos o ponto de vista, considerando a primeira entrada como sendo **valor original** e a outra entrada como a **máscara**, e o resultado como o **valor modificado**, podemos entender como as máscaras lógicas funcionam.



No exemplo acima foi escolhida uma máscara com o valor 0FH para melhor ilustrar os exemplos. Fique bem claro: a máscara pode conter qualquer valor de 8 bits.

No caso da função **AND**, os bits da máscara com valor zero modificam os bits do valor original para zero. Os bits uns, não modificam o valor original. Observe que os 4 bits MSB da máscara são zeros e transformam os 4 bits MSB do valor original em zero. Já os 4 bits LSB da máscara não causam alterações nos 4 bits LSB do valor original.

```
          3BH = 00111011
OR        OFH = 00001111
-------------------------
          3FH = 00111111
```

No caso da função **OR**, o efeito dos bits da máscara é diferente:
Os bits zeros da máscara _____
Os bits uns da máscara _____

```
          3BH = 00111011
XOR       OFH = 00001111
-------------------------
          34H = 00110100
```

No caso da função **XOR**, o efeito dos bits da máscara é:
Os bits zeros da máscara _____
Os bits uns da máscara _____

Isso nos permite concluir que, com o uso das máscaras podemos causar qualquer efeito em cada um dos bits do valor original: pode-se zerá-los, setá-los ou invertê-los.

```
               XXXXXXXX
AND     OFH = 00001111
-----------------------
               0000XXXX
```

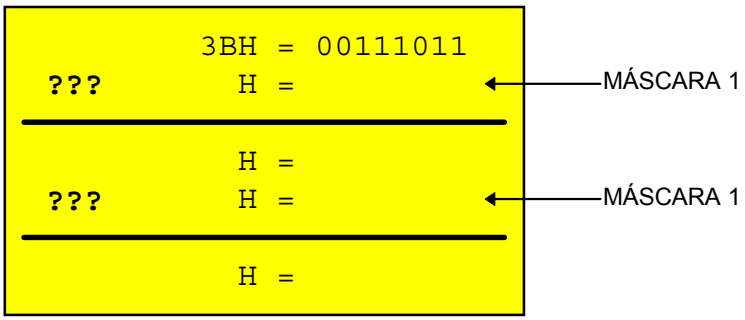
```
               XXXXXXXX
OR      OFH = 00001111
-----------------------
               XXXX1111
```

```
               XXXXXXXX
XOR     OFH = 00001111
-----------------------
               XXXXXXXX
```

Nos exemplos anteriores foi utilizado um valor pré-estabelecido 3BH como valor original. Poderíamos partir da premissa de que o valor original é desconhecido. Desse modo, o valor original poderia ser expresso como XXXXXXXX onde cada um dos X pode ser 0 ou 1. Assim, o efeito das máscaras binárias poderia ser expresso conforme as figuras a seguir:

Para obter efeitos mesclados, como por exemplo, ter alguns bits do valor original zerados e outros invertidos devem ser usadas duas máscaras para obter o valor desejado.

Exemplo: a partir de um valor original A=3BH escrever o trecho de programa que permita zerar os bits 1 e 5 e inverter os bits 3, 4 e 7 desse valor.

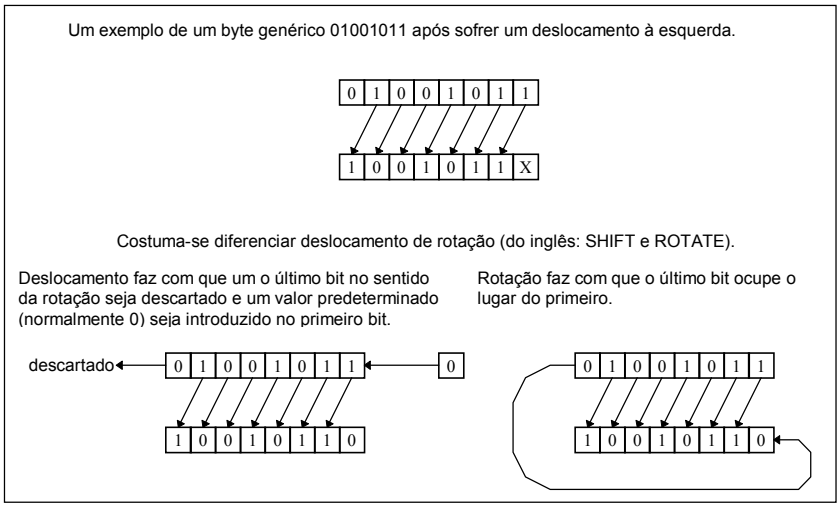


PROGRAMA:

```
MOV A,#3BH    ;A=3BH
              ;
              ;
```

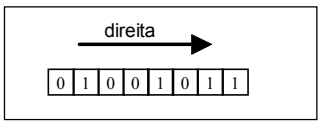
INSTRUÇÕES DE ROTAÇÃO DE BITS

As instruções de rotação e deslocamento de bits exercem sua influência sobre os bits do parâmetro. Antes de continuar, é importante estudar a diferença entre rotação e deslocamento. O quadro ao lado mostra essa diferença.

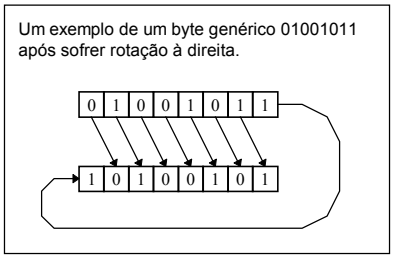


1. INSTRUÇÃO DE ROTAÇÃO À DIREITA

RR A ;"rotate right"

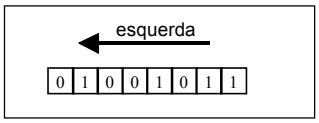


Faz a rotação dos bits de A uma posição em direção ao LSB. O bit LSB é colocado no lugar do MSB. As figuras ao lado exemplificam essa rotação.

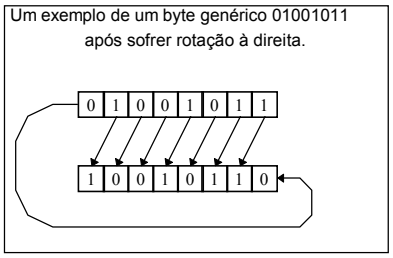


2. INSTRUÇÃO DE ROTAÇÃO À ESQUERDA

```
RL  A  ;"rotate left"
```

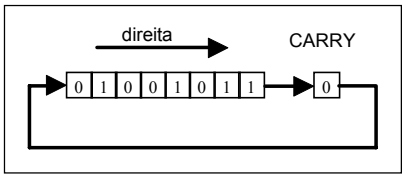


Faz a rotação dos bits de A uma posição em direção ao MSB. O bit MSB é colocado no lugar do LSB. As figuras ao lado exemplificam essa rotação.

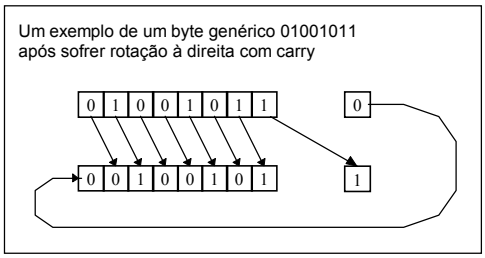


3. INSTRUÇÃO DE ROTAÇÃO À DIREITA COM CARRY

```
RRC  A        ;"rotate right with carry"
```

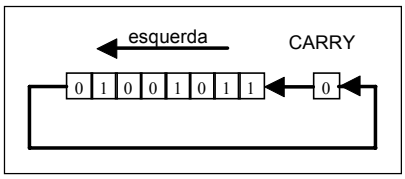


Faz a rotação dos bits de A uma posição em direção ao LSB. O bit LSB é colocado no CARRY e, este, no lugar do MSB.

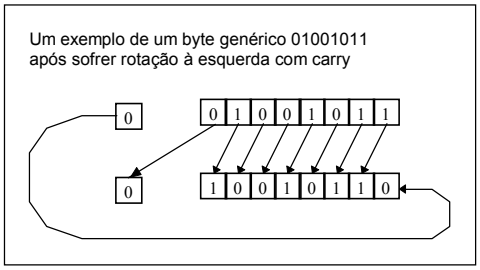


4. INSTRUÇÃO DE ROTAÇÃO À ESQUERDA COM CARRY

RLC A ;"rotate left with carry"



Faz a rotação dos bits de A uma posição em direção ao MSB. O bit MSB é colocado no CARRY e, este, no lugar do LSB.



EXEMPLOS

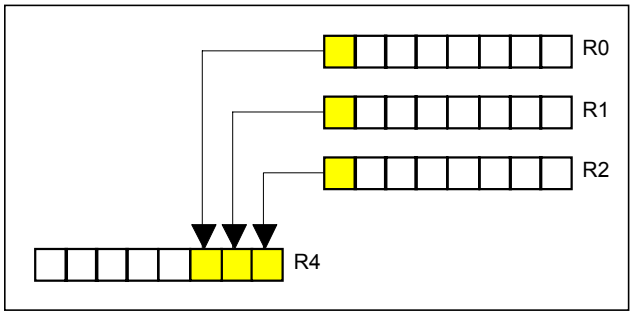
EXEMPLO 1: Esse trecho de programa inverte o nibble MSB com o nibble LSB do acumulador.
nibble=grupo de 4 bits
byte =grupo de 8 bits.

```
RL A
RL A
RL A
RL A
```

EXEMPLO 2: Esse trecho de programa pega cada bit MSB dos registradores R0 até R7 e coloca-os um a um nos bits do acumulador.

```
MOV R4,#0
MOV A,R0
RLC A
MOV A,R4
RLC A
MOV R4,A
MOV A,R1
```

```
RLC A
MOV A,R4
RLC A
MOV R4,A
MOV A,R2
RLC A
MOV A,R4
RLC A
MOV R4,A
```



Instruções de incremento e decremento

INSTRUÇÃO DE INCREMENTO (SOMA 1)

```
INC destino
```

Faz o conteúdo de destino ser somado de 1.

Exemplo:

```
MOV A,#14h
INC A
```

Sintaxes válidas:

```
INC A
INC DPL
INC 02h
INC @RO
```

INSTRUÇÃO DE DECREMENTO (SUBTRAI 1)

```
DEC destino
```

Faz o conteúdo de destino ser subtraído de 1.

Exemplo:

```
MOV B,#23h
DEC A
```

Sintaxes válidas:

```
DEC R5
DEC 79h
DEC @R1
DEC #03h
```

A última instrução não é válida: mesmo que se faça a subtração de um valor imediato (no exemplo acima, 03-1=02) a instrução não especifica onde deve ser guardado o resultado.

EXERCÍCIOS

Instruções de desvio

Esse tipo de instruções são de extrema utilidade na programação dos microprocessadores. Até então, as instruções eram executadas em seqüência, uma após a outra.

As instruções de desvio permitem alterar a seqüência de execução das instruções.

As instruções de desvio possuem diferentes graus de abrangência. Nos 8051 temos:

1. DESVIO LONGÍNQUO (LONG JUMP)

```
LJMP    endereço
```

Essa instrução, quando executada, permite que o µP desvie o seu "trajeto normal" da execução das instruções e vá direto para "endereço" continuando lá a execução das instruções.

Exemplo:

```
0160    74 03       MOV    A,#03h
0162    75 11 0D    MOV    11h,#0Dh
0165    25 11       ADD    A,11h
0167    02 01 0D    LJMP   016Bh
016A    04          INC    A
016B    25 11       ADD    A,11h
016D    FB          MOV    R3,A
```

2. DESVIO MÉDIO

```
AJMP    endereço
```

Como visto anteriormente, a instrução LJMP pode desviar para qualquer endereço de 16 bits, ou seja, abrange qualquer localidade dentro de 64 KBytes. Já o AJMP não consegue abranger uma quantidade de memória tão grande, mas apenas endereços de 11 bits, ou seja, 2 KBytes. Mas de onde vêm os bits de endereço restantes A_{10} até A_{15}? Resposta: esses bits vêm do registrador PC. Isso permite concluir o seguinte:

Imaginemos a memória de 64 KBytes segmentada em 64 partes de 1 KByte:

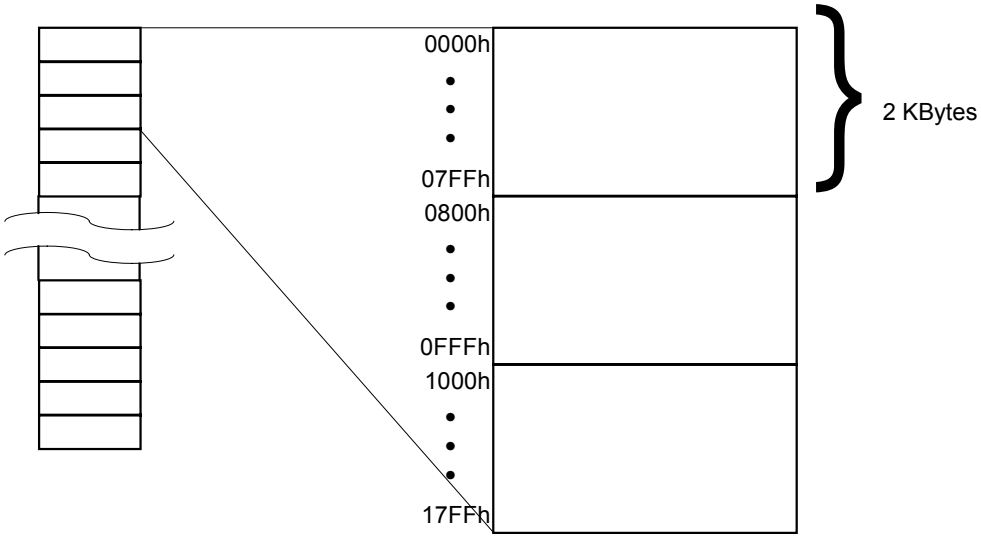


Figura 1. Área de abrangência do desvio médio AJMP em blocos de 2 Kbytes.

A área de "abrangência" de cada AJMP é o "bloquinho" de 2 KBytes de memória onde a instrução está sendo executada. Assim, um AJMP que está dentro do 1º bloquinho somente pode desviar para endereços dentro do mesmo bloquinho, nunca para fora dele.

A única vantagem da instrução AJMP é que ela ocupa 2 bytes da EPROM enquanto LJMP ocupa 3 bytes.

As instruções de desvio LJMP e AJMP causam desvios absolutos. Uma instrução LJMP 03A5h refere-se exatamente ao endereço 03A5h. A seguir, veremos um tipo de desvio diferente.

3. DESVIO RELATIVO CURTO

```
SJMP    endereço_relativo
```

Este tipo de desvio é diferente dos anteriores (absolutos): é um desvio relativo. Nesse tipo de instrução o endereço final é obtido a partir da localização da instrução. Desvio relativo significa dizer, por exemplo:

```
SJMP   +05h
```

"desviar +05 endereços a partir do endereço atual".

Esse tipo de desvio abrange um desvio relativo de, no máximo, -128 até +127.

Exemplo:

```
0160    74 03        MOV    A,#03h
0162    75 F0 02     MOV    B,#02h
0165    25 F0        ADD    A,B
0167    80 01        SJMP   +01 (016Ah)
```

```
0169     04           INC     A
016A     AE F0        MOV     R6,A
```

Para encontrarmos o endereço para onde será feito o desvio, o endereço de referência quando a instrução SJMP estiver sendo executada é o endereço da próxima instrução após o SJMP, que no exemplo acima é 0169h. Sobre esse endereço de referência, aplicamos o valor relativo do desvio, que no exemplo é +01. Fazendo então 0169h + 01 = 016Ah que é o endereço para onde é feito o desvio. O endereço final 016Ah é comumente chamado de endereço efetivo (endereço final obtido após os cálculos).

Outro exemplo:

```
0160     74 03        MOV     A,#03h
0162     75 F0 02     MOV     B,#02h
0165     25 F0        ADD     A,B
0167     80 01        SJMP    -09 (______h)
0169     04           INC     A
016A     AE F0        MOV     R6,A
```

Pergunta: para onde será feito o desvio quando for executada a instrução SJMP acima?
resp.: _____

A grande vantagem do desvio relativo é que ele é relativo à localização da instrução SJMP, não importando onde essa instrução esteja. Aqueles bytes do exemplo acima poderiam ser escritos em qualquer lugar da memória que o programa funcionaria sempre do mesmo modo. No primeiro exemplo, o programa foi colocado no endereço 0160h e o SJMP desviava para 016Ah. Se esse mesmo programa do exemplo estivesse em outro endereço, não haveria nenhum problema, pois, o desvio é relativo e sempre desviaria +01 à frente. Isso dá origem a um outro conceito muito importante: o de código realocável.

Vejamos os exemplos a seguir:

Exemplo 1:

```
0150     74 03        MOV     A,#03h
0152     75 F0 02     MOV     B,#02h
0155     25 F0        ADD     A,B
0157     80 01        SJMP    -09 (______h)
0159     04           INC     A
015A     AE F0        MOV     R6,A
```

Exemplo 2:

```
0290     74 03        MOV     A,#03h
0292     75 F0 02     MOV     B,#02h
0295     25 F0        ADD     A,B
0297     80 01        SJMP    -09 (______h)
0299     04           INC     A
029A     AE F0        MOV     R6,A
```

Observe que os bytes que compõem o programa são absolutamente os mesmos mas estão localizados em endereços diferentes. Entretanto, em ambos os casos, o trecho de programa funciona perfeitamente. No primeiro caso, a instrução SJMP causa desvio para o endereço 0159h - 09 = 0150h. No segundo caso, a instrução SJMP causa desvio para o endereço 0299h - 09 = 0290h.

DESVIOS CONDICIONAIS

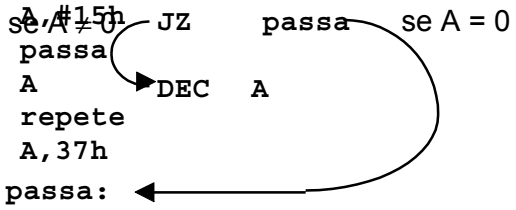
A programação é uma seqüência de procedimentos para que o μP realize todos os sinais de controle necessários no sistema digital onde está associado. Sem desvios, a execução das instruções ocorreria de forma linear: uma após a outra, até o final da memória. Com os desvios incondicionais, é possível alterar o "trajeto" da execução das instruções (por exemplo: executar 15 instruções, fazer um desvio para o endereço 023Dh e continuar lá a execução, podendo ter ainda outros desvios), entretanto, o trajeto será sempre o mesmo. Com os desvios condicionais é possível tomar decisões sobre qual trajeto seguir, pois o desvio dependerá de uma condição. São, portanto, a essência da programação.

Instrução JZ (Jump If Zero)

```
JZ    endereço_relativo
```

Realiza um desvio relativo para **endereço_relativo** caso, nesse instante, o conteúdo do registrador A seja zero. Se não for, a instrução não surte nenhum efeito e a instrução seguinte ao JZ é executada. Exemplo:

```
        MOV     A,#15h
repete: JZ      passa
        DEC     A
        LJMP    repete
passa:  MOV     A,37h
```



Nesse programa, a primeira instrução manda colocar o valor 15h no Acumulador. A segunda instrução é um desvio, mas depende de uma condição. A condição é: o Acumulador deve ser zero nesse instante. Como isso é falso, a instrução não faz efeito e o desvio não é gerado, passando-se para a próxima instrução: DEC A. Não é difícil de se perceber que o valor de A é 15h originalmente, mas vai sendo decrementado a cada repetição (15h, 14h, 13h, ...) até que alcance o valor 0. Nesse momento, a instrução de desvio será realizada, pois a sua condição finalmente é verdadeira.

Instrução JNZ (Jump If Not Zero)

```
JNZ    endereço_relativo
```

Semelhante à anterior para o caso de A diferente de zero.

Exemplos: Fazer um programa que realize a contagem de 1 a 10 no registrador B.

solução:

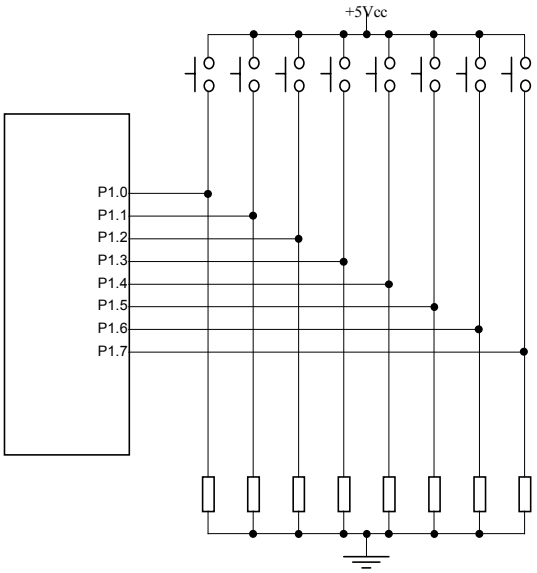
```
        MOV     B,#1
        MOV     A,#9
repete: INC     B
```

```
        DEC         A
        JNZ         repete
```

outra solução:

```
        MOV         B,#1
repete: INC         B
        MOV         A,B
        XRL         A,#0Ah   ; poderia ser XRL A,#10
        JNZ         repete
```

Exemplo: Fazer um pequeno programa que procure detectar uma chave pressionada, considerando o circuito ao lado:



solução:

```
repete: MOV         A,P1    ;lê o conteúdo
                      das chaves
        JZ          repete  ;se nada for
                      pressionado,
                      repete
```

Instrução JC (Jump If Carry Set)

```
        JC    endereço_relativo
```

Causa o desvio se o CARRY=1.

Instrução JNC (Jump If Carry Not Set)

```
        JNC    endereço_relativo
```

Causa o desvio se o CARRY=0.

Instrução JB (Jump If Direct Bit Set)

```
        JB    end-bit,endereço_relativo
```

Causa o desvio se o o conteúdo do "endereço de bit" = 1.

Instrução JNB (Jump If Direct Bit Not Set)

JNB end-bit,endereço_relativo

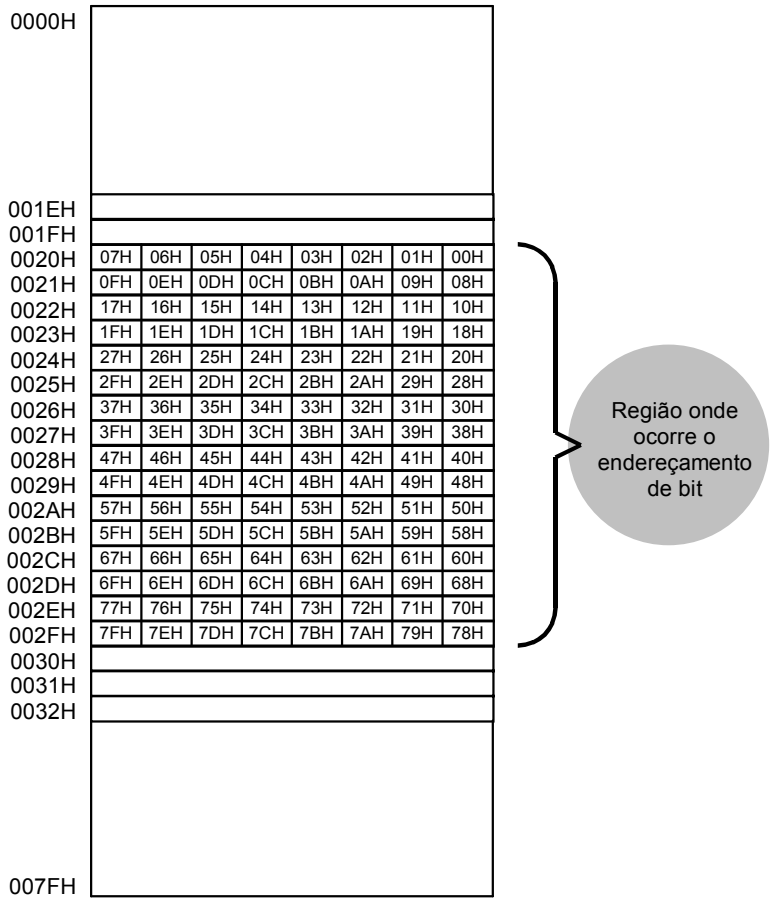


Figura 2. Mapeamento da memória RAM interna dos 8051. Endereçamento de byte e Endereçamento de bit.

Causa o desvio se o o conteúdo do "endereço de bit" = 0.

Para compreender o funcionamento dessas duas instruções devemos analisar um novo conceito: Os 8051 foram construídos para terem dois modos de endereçamento da memória RAM interna:

- endereçamento de byte
- endereçamento de bit

Endereçamento de byte é o que vimos até então; exemplo: MOV A,1CH.

O endereçamento de bit ocorre a partir do endereço 0020H até 002FH. Observe o esquema a seguir (figura 2):

Onde existe o endereço de byte 0020H exitem 8 bits endereçá-veis através de endereçamento de bit: endereços: 00H, 01H, 02H, 03H, 04H, 05H, 06H e 07H. Portanto, podemos dizer que nesse local da memória RAM interna existem dois endereça-mentos.

O endereço de byte é usado nas instruções que acessam todos os 8 bits que compõem um byte nesse local da memória (ver figura 3a). O endereço de bit é usado por outras instruções que acessam 1 único bit nesse local da memória (ver figura 3b).

Os endereços não são coinciden-tes e nem poderiam ser. O avan-ço de 1 byte é o mesmo que o avaço de 8 bits.

Supondo que o conteúdo do endereço 0020H é C5H, veremos como esse valor é acessado através das figuras 3a e 3b.

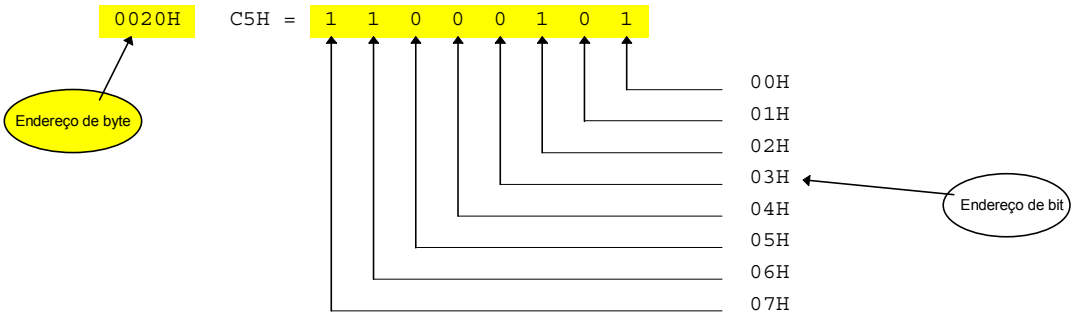


Figura 3a. Acesso ao endereço de byte 0020H.

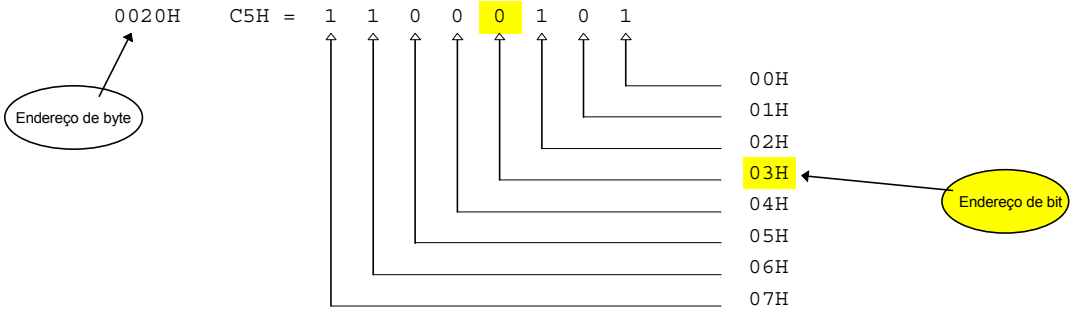


Figura 3b. Acesso ao endereço de bit 03H.

Dessa forma, a instrução

```
JB 00H,+12
```

Diz para fazer um desvio +12H dependendo do conteúdo do endereço de bit 00H. Como o conteúdo desse endereço é 1, a condição é verdadeira e a instrução irá causar o desvio relativo +12.

Mas como fazer para saber quando um determinado endereço refere-se a endereço de byte ou de bit? A própria instrução vai deixar isso claro. As instruções que utilizam endereço de bit são:

```
nome     exemplo
JB       JB          03H,+0CH
JNB      JNB         03H,+0CH
CLR      CLR         03H
SETB     SETB        03H
CPL      CPL         03H
ANL      ANL         C,03H
ORL      ORL         C,03H
XRL      XRL         C,03H
MOV      MOV         C,03H
MOV      MOV         03H,C
```

Algumas dessas instruções ainda não foram vistas, outras apenas sofreram uma alteração no seu parâmetro principal que, em lugar do Acumulador é usado o Carry. Vejamos, então, qual é o funcionamento das instruções que realizam manipulação de bit, citadas acima:

Instrução CLR (Clear bit)

```
CLR    endereço_de_bit
```

Zera o conteúdo do endereço de bit. (=0).

Instrução SETB (Set Bit)

```
SETB   endereço_de_bit
```

"Seta" o conteúdo do endereço de bit. (=1).

Instrução CPL (Complement Bit)

```
CPL    endereço_de_bit
```

Complementa (inverte) o conteúdo do endereço de bit.

Instruções ANL, ORL, XRL (Operações lógicas entre o Carry e um bit)

```
ANL    C,endereço_de_bit
ORL    C,endereço_de_bit
XRL    C,endereço_de_bit
```

Faz a operação lógica correspondente entre o Carry e o conteúdo do endereço de bit. O resultado é colocado no Carry.

Instrução MOV (MOV bit)

```
MOV    C,endereço_de_bit
MOV    endereço_de_bit,C
```

Copia o conteúdo do endereço de bit para o Carry ou vice-versa.

Podemos agora visualizar e diferenciar esses dois tipos de endereçamento:

```
MOV  A,06H          usa endereço de byte 06H.
MOV  C,06H          usa endereço de bit 06H.
```

Exemplo: Para colocar o valor C5H no endereço de byte 0020H pode-se fazer:

```
MOV 20H,#C5H
```

ou

```
SETB 00H
CLR  01H
SETB 02H
CLR  03H
CLR  04H
CLR  05H
```

```
SETB 06H
SETB 07H
```

Parece muito melhor usar uma única instrução que manipula byte do que usar 8 instruções que manipulam bit. Entretanto, se desejarmos zerar apenas um determinado bit (o terceiro) teríamos que fazer:

```
MOV A,20H
ANL A,#11111011B      ;é o mesmo que ANL A,#FBH
MOV 20H,A
```

ou

```
CLR 02H
```

Aí está a grande diferença!

Instrução CJNE (Compare and Jump if Not Equal)

```
CJNE    param1,param2,endereço_relativo
```

Compara **param1** com **param2**; se forem diferentes, a condição é verdadeira e faz um "jump" para **endereço_relativo**. Se **param1**=**param2** a condição é falsa.

Exemplo:

```
        MOV  R3,#05H        ;valor inicial=5
ref     INC  R3             ;aumenta
        CJNE R3,#25H,ref    ;se R3 diferente de 25h faz
                            jump
```

Instrução DJNZ (Decrement and Jump if Not Zero)

```
DJNZ    param,endereço_relativo
```

Trata-se de uma instrução complexa, pois é a junção de dois procedimentos em uma única instrução:

1. Decrementa o conteúdo de **param**.
2. Após ter decrementado, se o valor de **param** for diferente de zero, a condição é verdadeira e faz o "jump" para **endereço_relativo**.

O decremento acontece sempre, independente de a condição ser verdadeira ou falsa.

Essa instrução é utilíssima, porque esses procedimentos são muitíssimos freqüentes nos programas. É com essa instrução que implementamos qualquer tipo de contagem:

```
MOV  A,#valor1  ;val.inicial
```

```
        MOV  B,#valor2  ;val.inicial
        MOV  R2,#valor3 ;val.inicial
        MOV  R7,#5      ;num.de repetições
pt      ADD  A,B        ;procedimento
        SUBB A,R2       ;procedimento
        DJNZ R7,pt      ;controla a repetição dos
                        procedimentos
        ...próximas instruções
```

O programa acima demonstra como utilizar um registrador de propósito geral (no caso o registrador R7) para fazer um controle do número de repetições dos procedimentos de soma e subtração. Primeiramente são colocados os valores iniciais nos registradores, depois é estabelecido o número de repetições igual a 5 e depois entram os procedimentos. Ao final, decrementa R7 e desvia para "pt". Essa repetição ocorre 5 vezes até que em **DJNZ R7,pt** o valor de R7 alcança zero, o desvio não ocorre e passa-se para as próximas instruções.

EXERCÍCIOS

EX. 1 – Fazer um programa que leia 15 vezes a porta P1 para o regist. R4 e depois termine.

EX. 2 – Fazer um programa que leia indefinidamente a porta P1 e só termine quando o valor lido for igual a 3BH.

EX. 3 – Fazer um programa que faça 12 vezes a soma entre A e B.

EX. 4 – Fazer um programa que faça uma "somatória fatorial", ou seja, S = n + (n-1) + (n-2) + ... + 2 + 1. O valor de n deve ser fornecido no registrador B e a resposta deve ser colocada no acumulador.

EX. 5 – Nos sistemas digitais antigos, a multiplicação era implementada como diversas somas sucessivas. Assim, 8x3 é o mesmo que 8 + 8 + 8 . Dados os valores a ser multiplicados nos registradores A e B, fazer um programa que realize a multiplicação. O resultado deverá estar no acumulador.

EX. 6 –Fazer um programa que some os conteúdos dos endereços 0020h até 0030h.

EX. 7 – Fazer um programa que some 12 conteúdos a partir do endereço 0050h.

Estruturas de armazenamento sequenciais

ESTRUTURAS DE DADOS USADAS EM MEMÓRIAS

O armazenamento de informações em memórias é algo absolutamente imprescindível. Em se tratando de um grande volume de informações, essa tarefa não é tão fácil. Existem várias estruturas de armazenamento de dados em memórias bastante comuns, entre elas: fila, pilha, lista simples, circular, duplamente encadeada, árvores, matrizes esparsas etc.

Para que uma estrutura seja apropriada ela deve ser eficiente de acordo com o necessário. Por exemplo: uma lista telefônica com os nomes de pessoas de uma cidade inteira precisa ser bem organizada para permitir um acesso rápido à informação desejada. Não é apropriado usar uma estrutura seqüencial pois se a informação for uma das últimas da seqüência demorará muito para ser encontrada. Por outro lado, quando pouca informação estiver envolvida, não é necessário usar estruturas complexas que acabam gastando muito espaço somente com seus indexadores.

Das estruturas mencionadas, estudaremos duas: fila e pilha.

1. FILA

Também conhecida como memória FIFO (First In, First Out). É uma estrutura seqüencial.

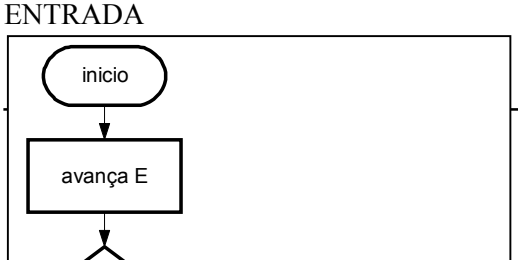
As informações são colocadas nessa estrutura de memória em seqüência e a retirada ocorre na mesma ordem em que as informações foram colocadas. A primeira informação retirada é a primeira que havia sido colocada.

Para a implementação dessa estrutura são necessárias 4 variáveis.

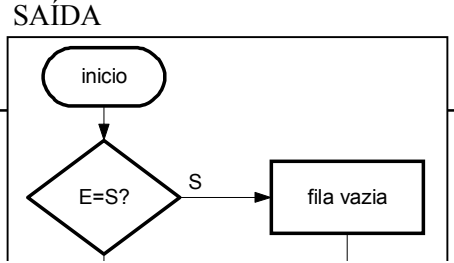
endereço de início (INI)
endereço de fim (FIM)
endereço de entrada (E)
endereço de saída (S)

Os procedimentos de entrada e saída de uma informação (um byte) nessa estrutura são:

ENTRADA



SAÍDA



É fácil de verificar que a capacidade de armazenamento dessa estrutura é: CAP = FIM – INI

Como exemplo, imaginemos uma FILA com:

INI = 30H
FIM = 37H

No exemplo acima, as informações são colocadas na memória na seqüência: 9Dh, F1h, 00h, 3Fh, 20h e 1Ah.

Quando se deseja retirar as informações dessa estrutura começa-se pela primeira que entrou. Os dados são retirados na ordem: 9Dh, F1h, 00h, 3Fh, 20h e 1Ah.

2. PILHA

Também conhecida como memória LIFO (Last In, First Out). Como a fila, a pilha também é uma estrutura seqüencial.

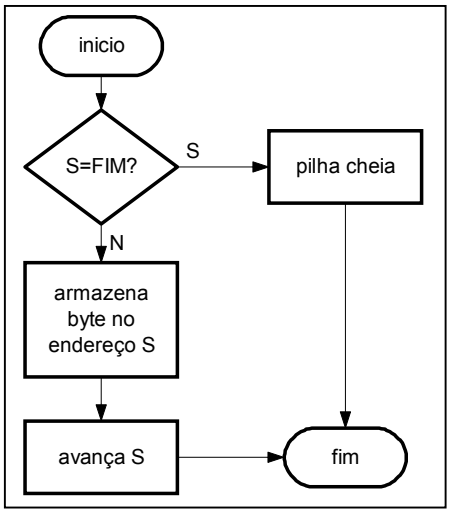
As informações são colocadas nessa estrutura de memória em seqüência e a retirada ocorre na ordem inversa em que as informações foram colocadas. A primeira informação retirada é a última que foi colocada.

Para a implementação dessa estrutura são necessárias 3 variáveis.

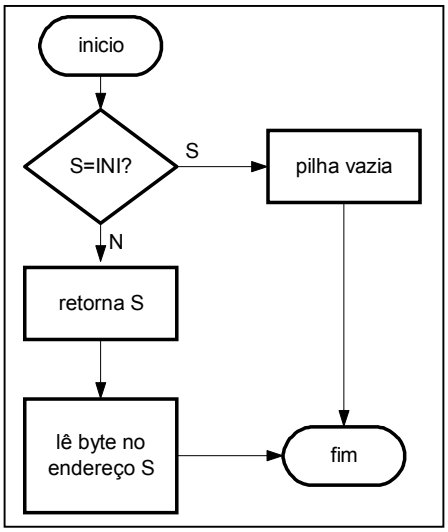
endereço de início (INI)
endereço de fim (FIM)
endereço de pilha (S)

Os procedimentos de entrada e saída de uma informação (um byte) nessa estrutura são:

ENTRADA



SAÍDA



Como exemplo, imaginemos uma PILHA com:

INI = 30H
FIM = 37H

Nesse exemplo, os byte foram colocados na ordem: 9Dh, F1h, 00h, 3Fh, 20h e 1Ah. Quando se deseja retirar as informações dessa estrutura começa-se pela última que entrou. Os bytes são retirados na ordem: 1Ah, 20h, 3Fh, 00h, F1h e 9Dh.
A característica mais importante dessas estruturas de memória: _____.

O µP utiliza a estrutura da PILHA internamente, por ser especialmente adequada ao controle do retorno das chamadas às subrotinas. Entretanto é importante ressaltar que o µP não possui as variáveis INI e FIM. Possui apenas o endereço de pilha, que é representado pelo registrador S (nos 8051). (Obs. todos os µPs possuem essa estrutura de memória mas na maioria deles o registrador é chamado de SP, sigla de 'Stack Pointer'). Portanto, CUIDADO! É possível colocar bytes indefinidamente na pilha sobre-escrevendo todos os endereços da RAM interna, inclusive endereços onde há outras informações, apagando-as. Também é possível retirar bytes além do limite mínimo (pilha vazia).

EXERCÍCIOS:

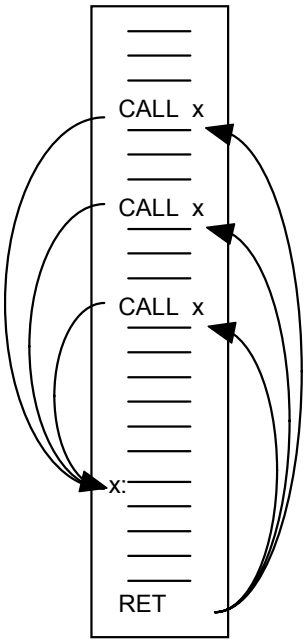
Para os exercícios a seguir, considere a memória com o seguinte conteúdo:

ENDER.	CONTEÚDO							
0000	10	00	02	FC	49	64	A0	30
0008	1D	BC	FF	90	5B	40	00	00
0010	00	00	01	02	03	04	05	06
0018	07	08	09	0A	FF	FE	FD	FC
0020	FB	FA	A5	5A	DD	B6	39	1B

Subrotinas

O fluxo habitual de execução das instruções em um programa é chamado de *rotina*. Além desse significado, há outro: Todo *programa* pode ser dividido em *trechos de programa* que realizam tarefas específicas. Um trecho de programa que realiza a leitura da porta P1 continuamente até que um dos dois últimos bits seja igual a 1, por exemplo, realiza uma tarefa específica (ler a porta P1 e fazer verificação); também é referido como *rotina*.

Habitualmente, dizemos que uma *subrotina* é como um subprograma. Na verdade, uma parte de programa que é "chamada" pelo programa principal. Uma chamada faz o µP partir de uma dada posição no programa principal para onde está a subrotina e, ao final da sua execução, deve retornar para o local de onde havia partido a chamada. A grande característica é que uma mesma subrotina pode ser chamada várias vezes, de diferentes pontos do programa principal, e o retorno sempre se dará para o ponto de chamada original.



Na rotina principal ao lado, ocorrem 3 referências à subrotina "x". A chamada à subrotina é feita através da função de chamada CALL (chamada) que possui 2 formas:

1. CALL longínquo:

LCALL *end-16* (semelhante a LJMP *end-16*)

2. CALL médio

ACALL *end-11* (semelhante a AJMP *end-11*)

Quando uma chamada é realizada, o µP faz o seguinte:

- armazena na pilha o endereço de retorno, que indica a instrução seguinte ao LCALL.
- Desvia para a subrotina

Ao final da execução da subrotina está a instrução **RET**. Quando o µP encontra essa instrução, faz o seguinte:

- retira da PILHA o endereço de retorno lá armazenado
- desvia para esse endereço.

Desse modo, é possível fazer várias chamadas a uma mesma subrotina sem que o µP se perca, pois o endereço de retorno estará armazenado na PILHA e nunca deixará o µP se perder.

Como os endereços possuem 16 bits, eles são armazenados na pilha na forma de 2 bytes.

Funcionamento das subrotinas

Para ilustrar o funcionamento das subrotinas, consideremos o seguinte programa:

```
 1   0200                          ORG 0200H
 2
 3   0200   78 30                 MOV R0,#30H
 4   0202   74 00                 MOV A,#0
 5   0204   7C 0A                 MOV R4,#10
 6   0206   26              rep1  ADD A,@R0
 7   0207   08                    INC R0
 8   0208   DC FC                 DJNZ R4,rep1
 9   020A   78 20                 MOV R0,#20H
10   020C   74 00                 MOV A,#0
11   020E   7C 0A                 MOV R4,#10
12   0210   26              rep2  ADD A,@R0
13   0211   08                    INC R0
14   0212   DC FC                 DJNZ R4,rep2
15   0214   78 70                 MOV R0,#70H
16   0216   74 00                 MOV A,#0
17   0218   7C 0A                 MOV R4,#10
18   021A   26              rep3  ADD A,@R0
19   021B   08                    INC R0
20   021C   DC FC                 DJNZ R4,rep3
     ...    ...                   ...
```

O objetivo deste trecho de programa é fazer uma somatória de 10 bytes consecutivos da memória a partir de um dado endereço. Essa mesma tarefa foi feita a partir do endereço 0030H e repetida a partir dos endereços 0020H e 0070H. Desta forma, as três rotinas são praticamente idênticas.

Observe agora:

```
 1   0200                          ORG 0200H
 2
 3   0200   78 30           inicio MOV   R0,#30H
 4   0202   12 02 0F               LCALL soma
 5   0205   78 20                 MOV   R0,#20H
 6   0207   12 02 0F               LCALL soma
 7   020A   78 70                 MOV   R0,#70H
 8   020C   12 02 0F               LCALL soma
 9                                ...
10                                LJMP  inicio
11
12   020F   74 00           soma   MOV A,#0
13   0211   7C 0A                 MOV R4,#10
14   0213   26              rep    ADD A,@R0
15   0214   08                    INC R0
16   0215   DC FC                 DJNZ R4,rep
17   0217   22                    RET
```

Escrito na forma de subrotina “soma”, o trecho de programa que soma os 10 bytes é escrito uma só vez, colocado em separado do programa principal e é terminado pela instrução **RET**. No programa principal, apenas o parâmetro relevante é usado (colocar em **R0** o endereço de início); depois ocorre a chamada à subrotina “soma” que faz o resto.

É importante observar que as subrotinas devem ser colocadas em separado. O programador não deve permitir que, em seu programa, a rotina principal acabe “entrando” dentro de uma subrotina sem que tenha havido uma chamada com LCALL ou ACALL. Se isso ocorrer, não haverá endereço de retorno na pilha e o resultado é imprevisível.

As subrotinas também podem chamar outras subrotinas dentro delas. Considere o seguinte programa:

```
--------------------- PROGRAMA -----------------------

    1   0100                            org 0100h
    2   0100   74 7D           refaz:   MOV A,#7Dh
    3   0102   75 F0 55                 MOV B,#55h
    4   0105   12 01 00                 LCALL sub_1
    5   0108   FA                       MOV R2,A
    6   0109   74 13                    MOV A,#13h
    7   010B   75 F0 45                 MOV B,#45h
    8   010E   12 01 00                 LCALL sub_1
    9   0111   FB                       MOV R3,A
   10   0112   74 9F                    MOV A,#9Fh
   11   0114   75 F0 11                 MOV B,#11h
   12   0117   12 01 00                 LCALL sub_1
   13   011A   21 00                    AJMP refaz
   14
   15
   16   011C   12 01 06         sub_1:  LCALL sub_2
   17   011F   25 F0                    ADD A,B
   18   0121   22                       RET
   19
   20   0122   04               sub_2:  INC A
   21   0123   12 01 0D                 LCALL sub_3
   22   0126   15 F0                    DEC B
   23   0128   22                       RET
   24
   25   0129   54 0F           sub_3:  ANL A,#0Fh
   26   012B   22                       RET
```

O programa acima foi escrito em um editor de textos simples (o EDIT do DOS ou o BLOCO DE NOTAS do Windows) e foi montado pelo programa ASM51.EXE, que faz parte do pacote de programas AVSIM para o 8051.

Para verificar o seu funcionamento, devemos analisar o programa passo-a-passo tentando imaginar o que o µP realizaria após a execução de cada instrução. Iniciemos pelas primeiras instruções localizadas a partir do endereço 0100h:

```
0100   74 7D           refaz:  MOV A,#7Dh
0102   75 F0 55                 MOV B,#55h
```

Como resultado: A=7Dh B=55h R2=?
PILHA= (vazia)

A próxima instrução é:

```
0105   12 01 00                LCALL sub_1
```

Para executar essa instrução, o µP realiza os seguintes passos:
1) Pega o endereço da próxima instrução (0108h) e coloca-o na PILHA.
2) desvia para o endereço onde está sub_1 (011Ch)

Como resultado: A=7Dh B=55h R2=?
PILHA= 01h, 08h

Depois que ocorreu o desvio, a próxima instrução passa a ser:

```
011C   12 01 06         sub_1:  LCALL sub_2
```

Novamente ocorre uma chamada a uma subrotina, somente que agora essa chamada ocorreu dentro da subrotina sub_1. A chamada é feita para sub_2. Para executar essa instrução, o µP realiza os seguintes passos:
1) Pega o endereço da próxima instrução (011Fh) e coloca-o na PILHA.
2) desvia para o endereço onde está sub_2 (0122h)

Como resultado: A=7Dh B=55h R2=?
PILHA= 01h, 08h , 01h, 1Fh

Depois que ocorreu o desvio, a próxima instrução passa a ser:

```
0122   04               sub_2:  INC A
```

que irá incrementar o registrador A (para 7Eh). A próxima instrução é:

```
0123   12 01 0D                LCALL sub_3
```

cujo funcionamento é similar ao que já foi descrito anteriormente:
1) Pega o endereço da próxima instrução (0126h) e coloca-o na PILHA.
2) desvia para o endereço onde está sub_3 (0129h)

Como resultado: A=7Eh B=55h R2=?
PILHA= 01h, 08h , 01h, 1Fh , 01h, 26h

Depois que ocorreu o desvio, a próxima instrução passa a ser:

```
0129   54 0F            sub_3:  ANL A,#0Fh
```

que faz a operação AND (A AND 0Fh). O valor de A fica sendo 0Eh. Depois disso, vem a instrução:

```
012B    22                      RET
```

Essa instrução faz o seguinte:
1) Retira um endereço da PILHA (0126h).
2) Desvia para esse endereço.

Dá para perceber que esse é o endereço de retorno para a última chamada que partiu da subrotina sub_2. Assim, a próxima instrução é:

```
0126    15 F0                   DEC B
```

que decrementa o registrador B (para 54h).

Como resultado: A=7Eh B=54h R2=?
PILHA= 01h, 08h , 01h, 1Fh

Olhando atentamente para a subrotina sub_2, verificamos que a execução das instruções foi desviada para sub_3 mas retornou exatamente para a próxima instrução de sub_2. Isso quer dizer que a seqüência de execução das instruções na subrotina sub_2 ficou intacta. Então, o mesmo deverá ocorrer com sub_1 e com a rotina principal. Vejamos:

A próxima instrução é:

```
0128    22                      RET
```

Essa instrução retira um endereço da pilha (011Fh) e desvia para esse endereço.

Como resultado: A=7Eh B=54h R2=?
PILHA= 01h, 08h

Então, a próxima instrução será:

```
011F    25 F0                   ADD A,B
```

De fato, essa instrução está na seqüência correta dentro da subrotina sub_1. Essa instrução manda o µP fazer a soma A+B e colocar o resultado em A.

Como resultado: A=D2h B=54h R2=?
PILHA= 01h, 08h

A próxima instrução é:

```
0121    22                      RET
```

O µP retira um endereço da pilha (0108h) e desvia para esse endereço.

Como resultado: A=D2h B=54h R2=?
PILHA= (vazia)

A próxima instrução será:

```
0108   FA                          MOV R2,A
```

Do mesmo modo como ocorreu para as subrotinas, essa instrução está na seqüência correta dentro da rotina principal. Essa instrução movimenta dados de um registrador para outro, no caso, de A para R2.

Como resultado: A=D2h B=54h R2=D2h
PILHA= (vazia)

A análise do restante do programa fica como exercício para o aluno.

ERROS DE PROGRAMAÇÃO

1. DESVIO PARA FORA DA SUBROTINA

Observe o programa:

```
 1   0030                    ORG 0030H
 2
 3   0030   E5 90    inicio  MOV   A,P1
 4   0032   12 00 3B         LCALL verif_letra
 5   0035   20 00 00 parte_2 JB    00H,pula
 6
 7                           ;(instruções que colocam a letra no visor)
 8
 9   0038   02 00 30 pula    LJMP  inicio
10
11   003B           verif_letra
12   003B   C2 00            CLR   00H   ;bit 0 = 0 (sinaliza não-letra)
13   003D   C0 E0            PUSH  A     ;guarda reg A
14   003F   C3               CLR   C     ;não influir na subtração
15   0040   94 41            SUBB  A,#41H      ;< 41H = 'A' ?
16   0042   50 F1            JNC   parte_2 ;sim: não é letra: vai p/ parte_2
17   0044   D0 E0            POP   A     ;resgata A
18   0046   C0 E0            PUSH  A     ;guarda A
19   0048   C3               CLR   C     ;não influir na subtração
20   0049   94 5B            SUBB  A,#5BH      ;> 5AH = 'Z' ?
21   004B   40 02            JC    nao_e ;sim: não é letra
22   004D   D2 00            SETB  00H   ;bit 0 = 1 (sinaliza letra)
23   004F   D0 E0    nao_e   POP   A     ;resgata A
24   0051   22               RET         ;retorna
```

Esse programa apresenta um grande erro: na quinta instrução "JNC parte_2" é um desvio para fora da subrotina. Isso significa que, quando a subrotina foi chamada, dois bytes foram colocados na pilha, equivalentes ao endereço de retorno. A subrotina deveria utilizar a instrução RET para retornar à rotina principal, pois retira esses dois bytes da pilha para formar o endereço de retorno.

Como o programa é cíclico, cada vez que houver um desvio para "fora" da subrotina, dois bytes sobram esquecidos na pilha. Na próxima vez que for

chamada, corre o risco de deixar mais dois. Em pouco tempo a pilha crescerá e fará com que o registrador S alcance todos os endereços da memória interna sobre-escrevendo informações.

2. ALTERAÇÃO DA PILHA ANTES DO RETORNO

Outro erro possível de acontecer é trabalhar com a pilha dentro da subrotina de forma errada deixando um bytes a mais ou a menos do que havia no início.

No programa a seguir, ocorre a chamada à subrotina "verif_letra". O μP coloca o endereço de retorno na pilha (PILHA = 00h, 35h). Logo a seguir é colocado o valor do acumulador (suporemos 97h) na pilha (PILHA = 00h, 35h, 97h). Se o fluxo de execução das instruções na subrotina for uma instrução após a outra até o seu final, ocorrerá o seguinte:
No endereço 0044h existe uma instrução POP A (portanto, PILHA = 00h, 35h).

No endereço 004Dh há outra instrução POP A (portanto, PILHA = 00h).
Ocorre a instrução RET no final da subrotina que tentará retirar 2 bytes da pilha para formar o endereço de retorno, o que não é possível. Serão retirados 00h e um outro byte desconhecido (abaixo do ponto inicial da pilha).

```
 1   0030                          ORG 0030H
 2
 3   0030   E5 90          inicio  MOV     A,P1
 4   0032   12 00 3B               LCALL   verif_letra
 5   0035   20 00 00       parte_2 JB 00H,pula
 6
 7                                 ;(instruções que colocam a letra no visor)
 8
 9   0038   02 00 30       pula    LJMP    inicio
10
11   003B                  verif_letra
12   003B   C2 00                  CLR     00H   ;bit 0 = 0 (sinaliza não-letra)
13   003D   C0 E0                  PUSH    A     ;guarda reg A
14   003F   C3                     CLR     C     ;não influir na subtração
15   0040   94 41                  SUBB    A,#41H  ;< 41H = 'A' ?
16   0042   50 09                  JNC     nao_e ;sim: não é letra
17   0044   D0 E0                  POP     A     ;resgata A
18   0046   C3                     CLR     C     ;não influir na subtração
19   0047   94 5B                  SUBB    A,#5BH  ;> 5AH = 'Z' ?
20   0049   40 02                  JC nao_e      ;sim: não é letra
21   004B   D2 00                  SETB    00H   ;bit 0 = 1 (sinaliza letra)
22   004D   D0 E0          nao_e   POP     A     ;resgata A
23   004F   22                     RET           ;retorna
```

Esse programa poderia ser modificado, deixando de colocar a última instrução "POP A": O programa ficaria assim:

```
 1   0030                        ORG 0030H
 2
 3   0030   E5 90          inicio  MOV     A,P1
 4   0032   12 00 3B              LCALL   verif_letra
 5   0035   20 00 00        parte_2 JB 00H,pula
 6
 7                               ;(instruções que colocam a letra no visor)
 8
 9   0038   02 00 30        pula    LJMP    inicio
10
11   003B                   verif_letra
12   003B   C2 00                   CLR     00H   ;bit 0 = 0 (sinaliza não-letra)
13   003D   C0 E0                   PUSH    A     ;guarda reg A
14   003F   C3                      CLR     C     ;não influir na subtração
15   0040   94 41                   SUBB    A,#41H  ;< 41H = 'A' ?
16   0042   50 09                   JNC     nao_e ;sim: não é letra
17   0044   D0 E0                   POP     A     ;resgata A
18   0046   C3                      CLR     C     ;não influir na subtração
19   0047   94 5B                   SUBB    A,#5BH  ;> 5AH = 'Z' ?
20   0049   40 02                   JC nao_e      ;sim: não é letra
21   004B   D2 00                   SETB    00H   ;bit 0 = 1 (sinaliza letra)
22   004D   22             nao_e   RET           ;retorna
```

Nesse caso, a subrotina começa com a PILHA = 00h, 35h. Se o fluxo de execução das instruções na subrotina for uma após a outra, o erro não ocorre mais. Entretanto, há outro caso que pode gerar erro:

Quando ocorre a instrução “PUSH A” no end. 003Dh, a pilha fica (PILHA = 00h, 35h, 97h). Se a condição da instrução “JNC nao_e” no end. 0042h for verdadeira, haverá um desvio para onde está o rótulo “nao_e”, no end. 004Dh. Quando for executada a instrução de retono RET, os dois bytes retirados da pilha serão 97h e 35h que formarão o endereço 3597h. Esse não é o endereço de retorno correto. Nem sequer sabemos se há algum programa nesse endereço.

Como regra geral, o programador deve tomar o cuidado de que quando a instrução RET for executada, existam na pilha apenas os dois bytes de retorno.