CAPÍTULO 4

CONJUNTO DE INSTRUÇÕES

4.1. INTRODUÇÃO

Todos os membros da família MCS-51 executam o mesmo conjunto de instruções. As instruções são otimizadas para aplicações de controle de 8 bits. Elas permitem um rápido endereçamento da RAM interna, facilitando operações byte a byte em estruturas de dados pequenas. O conjunto de instruções também oferece um grande suporte para manipulações e operações de variáveis de um bit, facilitando os sistemas lógicos que necessitam de operações booleanas.

A seguir serão mostradas todas as instruções da família MCS-51. Para uma descrição mais detalhada dessas instruções deve-se consultar o manual do MCS-51. (Aqui se supõe que o leitor tenha uma pequena experiência em programação assembly).

Para ajudar os programadores, os programas montadores para o MCS-51 têm os endereços dos SFR e dos bits previamente declarados. É como se em todo programa já existisse as pseudo-instruções:

```
Acc     EQU     0E0H
B       EQU     0F0H
PSW     EQU     0D0H
etc.
```

Para a exposição das instruções, utilizar-se-ão as seguintes abreviaturas:

Rn	→	qualquer registro R0, R1, ...,R7
direto, dir	→	endereço da RAM interna (8 bits)
@Ri	→	@R0 ou @R1, usado para endereçamento indireto
#data, #dt	→	constante de 8 bits (byte)
#data16, #dt16	→	constante de 16 bits
adr16	→	endereço de 16 bits (endereça 64 KB)
adr11	→	endereço de 11 bits (endereça 2 KB)
rel	→	deslocamento relativo (complemento a 2: -128 a +127)
bit	→	endereço de um bit da RAM interna
A	→	acumulador (registro)
Acc	→	endereço do acumulador.

4.2. MODOS DE ENDEREÇAMENTO

De acordo com o "Data Sheet" do MCS-51, existem 6 modos de endereçamento com a seguinte nomenclatura :

- → Imediato
- → Direto
- → Indireto
- → Registrador
- → Registrador Específico
- → Indexado

Imediato → O valor da constante é colocado no opcode.

MOV A, #100

Carrega 100 no acumulador (Acc=100). O byte 100 é um dado imediato. Deve-se notar a presença do sinal **#** que indica operação imediata.

Direto → O operando especifica um endereço de 8 bits da RAM interna.

MOV A, 20

Transfere para o acumulador o conteúdo do endereço 20 da RAM Interna. Todo endereçamento direto usa a RAM Interna.

Indireto → Aqui se especifica um registro onde está o endereço do operando. Só pode ser usado para endereçamento indireto: R0, R1 ou DPTR.

MOV A, @R0

Coloca no acumulador o conteúdo do endereço que está em R0

Registro → No código de operação da instrução existe um campo de 3 bits (pois são 8 registradores, de R0 a R7) onde é especificado o registro a ser utilizado. Essa forma é eficiente e evita utilizar um byte adicional para indicar o registro.

MOV A,R0

Coloca no acumulador o conteúdo de R0. É uma instrução de um só byte.

Registro Especifico → Algumas operações são especificas para certos registros. Por exemplo, algumas instruções sempre operam com Acc ou DPTR e não necessitam de espaço no opcode para especificar isto.

MOVX A,@DPTR

Esta é uma instrução para leitura da Memória de Dados Externa. Coloca no acumulador o conteúdo do endereço da RAM Externa que está no DPTR. Como sempre são usados

Acc e DPTR, não é necessário especificá-los, o que faz com que a instrução empregue apenas 1 byte.

Indexado → O endereço do operando é formado pela soma de um endereço base com um registro de indexação. Somente a Memória de Programa pode ser endereçada deste modo.

MOVC A,@A+DPTR

A → índice,

DPTR → endereço base.

A soma do DPTR com o acumulador forma um endereço da Memória de Programa e o conteúdo deste endereço é transferido para o acumulador. Essa instrução é ótima para "look up table".

Observação: também existe um desvio indexado

4.3. SOBRE AS INSTRUÇÕES

Agora serão mostradas as instruções do MCS-51. Estas não serão muito detalhadas. Em cada instrução são apresentados o número de bytes do opcode e o número de ciclos de máquina (MC=12 clocks) necessários para sua execução.

Existem 111 instruções na família MCS-51. Elas estão distribuídas da seguinte forma:

- Aritméticas 24 → 22%
- Lógicas 25 → 23%
- Transferência de dados 28 → 25%
- Booleanas 17 → 15%
- Desvios 17 → 15%

4.4. INSTRUÇÕES ARITMÉTICAS

4.4.1. Soma de 8 Bits:

O acumulador é um dos operandos e também armazena o resultado. Existem 4 instruções de soma:

		bytes	MC	CY	AC	OV
ADD A,	Rn	1	1	X	X	X
	direto	2	1			
	@Ri	1	1			
	#data	2	1			

4.4.1. Soma de 8 Bits com Carry:

O acumulador é um dos operandos e também armazena o resultado. São 4 instruções de soma com carry:

		bytes	MC	CY	AC	OV
ADDC A,	Rn	1	1	X	X	X
	direto	2	1			
	@Ri	1	1			
	#data	2	1			

4.4.3. Subtração de 8 Bits com Borrow:

O acumulador é um dos operandos e também recebe o resultado. Deve-se lembrar que a subtração sempre usa o borrow. São 4 instruções de subtração com borrow:

		bytes	MC	CY	AC	OV
SUBB A,	Rn	1	1	X	X	X
	direto	2	1			
	@Ri	1	1			
	#data	2	1			

4.4.4. Incremento de 8 Bits:

Existem 4 instruções para incremento de 8 bits :

		bytes	MC	CY	AC	OV
INC	A	1	1	-	-	-
	Rn	1	1			
	direto	2	1			
	@Ri	1	1			

4.4.5. Decremento de 8 Bits:

Há 4 instruções de decremento de 8 bits e muito se assemelham às instruções de incremento:

		bytes	MC	CY	AC	OV
DEC	A	1	1	-	-	-
	Rn	1	1			
	direto	2	1			
	@Ri	1	1			

4.4.6. Incremento de 16 Bits:

Existe apenas um incremento de 16 bits (não há decrementos de 16 bits) :

		bytes	MC	CY	AC	OV
INC	DPTR	1	2	-	-	-

4.4.7. Multiplicação e Divisão de 8 Bits:

Na multiplicação e na divisão sempre são usados os registros A e B. Estas são as instruções que necessitam mais tempo de execução.

Multiplicação: A * B → Resultado: A=LSB B=MSB

Divisão: A / B → Resultado: A=quociente B=resto

		bytes	MC	CY	AC	OV
MUL	AB	1	4	0	-	X
DIV	AB			0	X	0

4.4.8. Ajuste Decimal:

Esta instrução existe para permitir operações com representação BCD. O ajuste é valido apenas para as somas (ADD e ADDC). Existe uma única instrução:

		bytes	MC	CY	AC	OV
DA	A	1	1	X	X	-

OBS 1: DA A não pode simplesmente converter um número hexadecimal no Acumulador para BCD nem se aplica à subtração.

Exemplo: A = 56H (0101 0110B) representando 56 decimal

R3 = 67H (0110 0111B) representando 67 decimal

Carry = 1

ADDC A, R3 ➔ A = 0BEH e C (carry) e AC (auxiliar carry) iguais a 0

DA A ➔ A = 24H (= 0010 0100B) e Carry=1

Algoritmo de ajuste da soma pela instrução DAA:

1- Se os bits 3-0 de A forem maiores que 9 ou se AC=1 ➔ 6 é somado a A para gerar o dígito BCD menos significativo. Esta soma interna pode ligar o carry se o carry da nibble menos significativa propagar por toda a nibble mais significativa mas não apagará o carry.

2- Se o carry está agora em 1 ou se os bits 7-4 excederem 9, esta nibble é incrementada de 6 para produzir o dígito BCD mais significativo. O carry indicará se a soma é maior do que 99. O OV (overflow) não é afetado.

Na realidade, soma-se 00H, 06H, 60H ou 66H conforme o valor inicial de A e PSW.

OBS 2: Números BCD podem ser incrementados somando 01H ou decrementados somando 99H e usando o ajuste decimal DA A.

Exemplo: Se A = 30H (representando 30 decimal), as instruções

```
ADD A,#99H
DA A
```

resultarão em A = 29H (representando 29 decimal) e C (Carry) = 1 pois 30 + 29 = 129

4.5 INSTRUÇÕES LÓGICAS

As instruções lógicas são as que realizam as operações de AND, OR e XOR. Deve-se notar que essas instruções são muito semelhantes. O resultado da operação não obrigatoriamente deve ser colocado no acumulador; também se pode indicar um endereço para receber o resultado.

4.5.1. AND de 8 Bits:

Usa-se o símbolo ANL (Logical AND) para representar esta instrução. Existem 6 instruções reunidas em dois grupos de acordo com o destino do resultado:

		bytes	MC	CY	AC	OV
ANL A,	Rn	1	1	-	-	-
	direto	2	1			
	@Ri	1	1			
	#data	2	1			
ANL direto	A	2	1			
	#data	3	2			

4.5.2. OR de 8 Bits:

Emprega-se o símbolo ORL (Logical OR) para representar esta instrução. Existem 6 instruções, reunidas em dois grupos de acordo com o destino do resultado:

		bytes	MC	CY	AC	OV
ORL A,	Rn	1	1	-	-	-
	direto	2	1			
	@Ri	1	1			
	#data	2	1			
ORL direto	A	2	1			
	#data	3	2			

4.5.3. XOR de 8 Bits:

Emprega-se o símbolo XRL (Logical Exclusive OR) para representar esta instrução. Existem 6 instruções, reunidas em dois grupos de acordo com o destino do resultado:

		bytes	MC	CY	AC	OV
XRL A,	Rn	1	1	-	-	-
	direto	2	1			
	@Ri	1	1			
	#data	2	1			
XRL direto	A	2	1			
	#data	3	2			

4.5.4. Operações Lógicas com o Acumulador:

Pode-se inicializar, complementar ou rodar o acumulador. As rotações podem se realizar para a direita ou para a esquerda utilizando ou não o bit de carry. Também se pode trocar de posição as nibbles do acumulador (uma nibble é formada por 4 bits, ou seja, um byte possui 2 nibbles). Todas estas instruções são classificadas como de registro específico pois sempre operam com o acumulador (não utilizam bytes de opcode para especificar isto). As instruções são:

- **CLR A** → inicializa com zero o acumulador
- **CPL A** → complementa de 1 o acumulador (inverte os bits)
- **RL A** → roda o acumulador à esquerda
- **RLC A** → roda o acumulador à esquerda com carry
- **RR A** → roda o acumulador à direita
- **RRC A** → roda o acumulador à direita com carry
- **SWAP A** → intercambiar as nibbles do acumulador

		bytes	MC	CY	AC	OV
CLR	A	1	1	-	-	-
CPL				-	-	-
RL				-	-	-
RLC				X	-	-
RR				-	-	-
RRC				X	-	-
SWAP				-	-	-

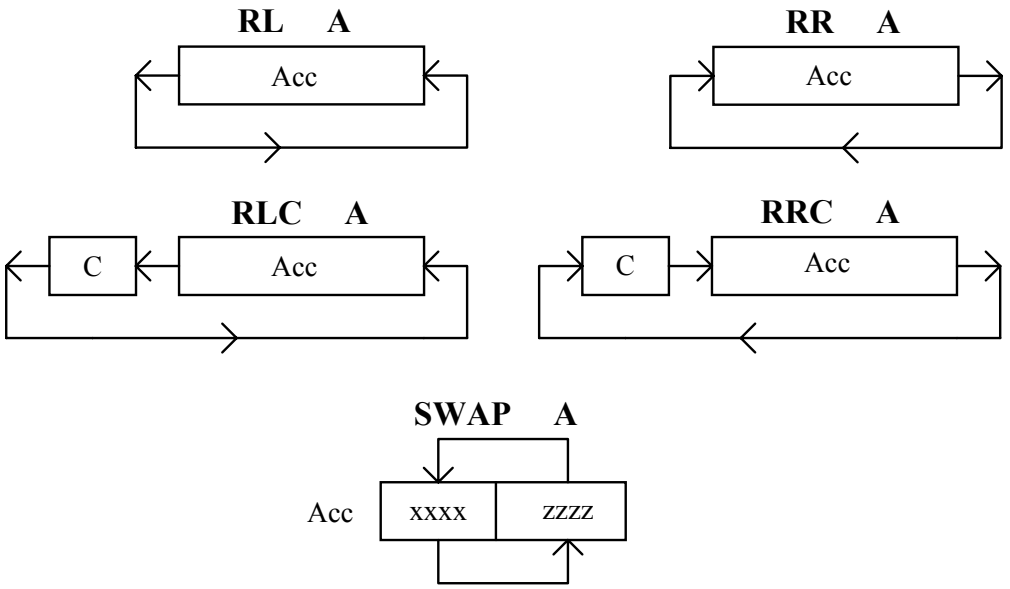


Figura 4.1. Gráfico das rotações e do swap.

4.6. INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

Existem 3 tipos de instruções de transferência de dados:

- as que trabalham com a RAM Interna (22),
- as que trabalham com a Memória de Dados Externa (4) e
- as que trabalham com a Memória de Programa (2).

Em geral, as transferências de dados operam na RAM Interna, a menos que se indique o contrário.

4.6.1. Transferência de Dados:

Todas as operações se realizam na RAM Interna. Existem quase todas as transferências que se possa imaginar.

		bytes	MC	CY	AC	OV
MOV A,	Rn	1	1			
	direto	2	1			
	@Ri	1	1			
	#data	2	1			
MOV Rn,	A	1	1			
	direto	2	2			
	#data	2	1			
MOV direto,	A	2	1	-	-	-
	Rn	2	2			
	direto	3	2			
	@Ri	2	2			
	#data	3	2			
MOV @Ri	A	1	1			
	direto	2	2			
	#data	2	1			

Pode-se pensar que existem todas as combinações possíveis entre **A**, **Rn**, **direto**, **@Ri** e **#data**, mas existem algumas exceções:

Todas as combinações possíveis:

MOV	A	,	A
	Rn		Rn
	direto		Direto
	@Ri		@Ri
			#data

Instruções que não existem:

MOV	A	,	A
	Rn	,	Rn
	Rn	,	@Ri
	@Ri	,	@Ri
	@Ri	,	Rn

4.6.2. Permutação de Bytes:

Toda permutação opera na RAM Interna. Estas instruções são muito úteis pois permitem a troca de conteúdo entre dois registros sem a necessidade de usar um outro auxiliar. Existem 3 permutações e todas usam o acumulador como um dos operandos.

		bytes	MC	CY	AC	OV
XCH A,	Rn	1	1	-	-	-
	direto	2	1			
	@Ri	1	1			

4.6.3. Permutação de Nibbles:

Existe só uma instrução e esta opera com a RAM Interna. Nesta instrução sempre se usa o acumulador e um operando (por endereçamento indireto). É útil para trabalhar com BCD e em conversões de hexadecimal para código ASCII.

		bytes	MC	CY	AC	OV
XCHD	A,@Ri	1	1	-	-	-

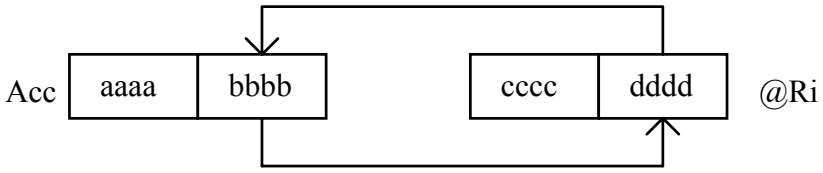


Figura 4.2. Gráfico da permutação de nibbles.

4.6.4. Operações com a Pilha:

Como já foi estudada, a pilha opera exclusivamente com a RAM Interna. Existem apenas duas instruções de pilha e trabalham com endereçamento direto. Quer dizer, sempre usam dois bytes para o opcode e não se pode usar a instrução PUSH A, mas sim PUSH Acc. Deve-se lembrar que Acc é o label para o endereço do acumulador. Na instrução PUSH, primeiro incrementa-se o ponteiro (SP) e em seguida escreve-se a informação. Exemplo: se SP=7, PUSH Acc colocará o conteúdo do acumulador no endereço 8. A instrução POP trabalha ao contrário.

		bytes	MC	CY	AC	OV
PUSH	direto	2	2	-	-	-
POP	direto					

4.6.5. Transferências de Dados com a Memória de Dados Externa:

Existem instruções para leitura e escrita, as quais sempre usam o acumulador como fonte ou destino do dado. É possível fazer dois acessos, um com endereçamento de 16 bits usando o DPTR e outro com endereçamento de 8 bits usando @Ri (@R0 ou @R1). Deve-se lembrar que quando se usa endereçamento de 8 bits, o conteúdo de P2 permanece inalterado e representa os 8 bits mais significativos do endereço. A instrução usada é MOVX, onde X indica que se trabalhará com Memória de Dados Externa.

LEITURA		bytes	MC	CY	AC	OV
MOVX A,	@Ri	1	2	-	-	-
	DPTR					

ESCRITA		bytes	MC	CY	AC	OV
MOVX @Ri,	A	1	2	-	-	-
MOVX @DPTR,						

4.6.6. Leitura da Memória de Programa:

Quando se trabalha com Memória de Programa separada da Memória de Dados Externa, a Memória de Programa só pode ser lida (sinal *PSEN). Existem duas instruções para realizar esta operação e também são as únicas que usam endereçamento indexado. São muito convenientes para construção de tabelas de consulta. O mnemônico empregado muda para MOVC onde o C indica Memória de Códigos.

LEITURA		bytes	MC	CY	AC	OV
MOVC A,	@A+DPTR	1	2	-	-	-
	@A+PC					

4.7. INSTRUÇÕES BOOLEANAS

São chamadas de instruções booleanas as que trabalham com bits. Todas essas instruções operam com um só bit por vez e o CARRY (C) funciona como se fosse o acumulador, ou seja, é um dos operandos e normalmente armazena o resultado. Os bits são endereçados diretamente, mas quando se usa o CARRY este é endereçado implicitamente. Existem também desvios baseados no estado dos bits; estes são relativos à posição atual do PC e se pode saltar 127 bytes adiante ou 128 bytes para trás.

4.7.1. Zerar/ Setar /Complementar um Bit:

Para carregar zero: usa-se a instrução → **CLR**

Para carregar um: usa-se a instrução → **SETB**

Para complementar: usa-se a instrução → **CPL**

		bytes	MC	CY	AC	OV
CLR	C	1	1	0	-	-
	bit	2	1	-	-	-
SETB	C	1	1	1	-	-
	bit	2	1	-	-	-
CPL	C	1	1	X	-	-
	bit	2	1	-	-	-

4.7.2. AND/OR Booleano:

Há dois AND e dois OR. Nestas instruções o carry (C) trabalha como o acumulador, quer dizer, é um dos operandos e também recebe o resultado. Exemplo: ANL C,bit → C = C AND bit. Empregam-se os mnemônicos ANL para Logical AND e ORL para Logical OR. Nas instruções em que um operando é "/bit", indica que a operação se realiza com o complemento deste bit. Essas operações são muito convenientes quando se implementam operações booleanas.

		bytes	MC	CY	AC	OV
ANL C,	bit	2	1	X	-	-
	/bit					
ORL C,	bit	2	1			
	/bit					

4.7.3. Movimento de Bits:

São duas as instruções para movimento de bits e sempre envolvem o CARRY.

	bytes	MC	CY	AC	OV
MOV C , bit	2	1	X	-	-
MOV bit , C	2	2	-	-	-

4.7.4. Desvios Baseados em Bits:

São 5 as instruções de desvio: duas se baseiam no CARRY e as outras três operam com qualquer bit. Os mnemônicos **JB** e **JC** são usados para desviar se o bit está ativado (em um) e

JNB e **JNC** para desviar se o bit está desativado (em zero). Existe o mnemônico **JBC** que desvia se o bit está ativado e depois complementa este bit.

		bytes	MC	CY	AC	OV
JC	rel	2	2	-	-	-
JNC						
JB	bit,rel	3	2			
JNB						
JBC						

4.8. INSTRUÇÕES DE DESVIO

Nas instruções de desvio estão incluídas as chamadas de subrotinas (CALL) e os desvios (JMP); estes podem ser realizados com 11 ou 16 bits. Também estão incluídos os desvios condicionais (JZ, JNZ, CJNE), os loops (DJNZ) e as instruções de retorno (RET, RETI).

4.8.1. Chamadas de Subrotinas:

São duas as instruções de chamada de subrotinas: a instrução **LCALL,** que usa um endereço de 16 bits e por isso permite acesso a qualquer posição nos 64 KB de programa, e a instrução **ACALL,** que usa apenas 11 bits, ou seja, permite acesso dentro de um bloco de 2 KB da memória de programa; os 11 bits menos significativos do PC são trocados pelos bits especificados na instrução. A vantagem é que **ACALL** necessita apenas de 2 bytes para o opcode enquanto que **LCALL** precisa de 3 bytes.

		bytes	MC	CY	AC	OV
LCALL	adr16	3	2	-	-	-
ACALL	adr11	2	2			

4.8.2. Retorno de Subrotinas:

São 2 as instruções de retorno de subrotinas. A instrução **RET** é o retorno usual. A instrução **RETI** é usada para retorno das subrotinas que atendem as interrupções, indicando o fim de uma rotina de interrupção. Mais adiante, no capítulo de interrupções, será abordado este tema com maior detalhe.

	bytes	MC	CY	AC	OV
RET	1	2	-	-	-
RETI					

4.8.3. Desvios:

São quatro as instruções de desvios. Há dois desvios que são idênticos às chamadas de subrotinas e que também permitem saltar numa faixa de 2 KB (AJMP) ou de 64 KB (LJMP). Os outros dois desvios são relativos.

		bytes	MC	CY	AC	OV
AJMP	adr11	2	2	-	-	-
LJMP	adr16	3	2			
SJMP	Rel	1	2			
JMP	@A+DPTR	1	2			

4.8.4. Desvios Condicionais:

Há dois desvios que são feitos com os valores do acumulador: **JZ,** que salta se o conteúdo do acumulador é zero, e **JNZ,** que salta se o conteúdo do acumulador não é zero. Deve-se notar que não existe o flag zero (como nos Z80 e 8085); a decisão para o desvio é feita com o conteúdo atual do acumulador. Há outras 4 instruções que usam o mnemônico **CJNE**, que significa "compare dois elementos e desvie se forem diferentes". Todos estes desvios são relativos e por isso têm um alcance de 127 bytes para adiante e 128 bytes para trás.

			bytes	MC	CONDIÇÃO	CY	AC	OV
JZ	rel		2	2	se Acc = 0	-	-	-
JNZ					se Acc 0			
CJNE	A	,direto ,rel	3	2	se Acc Direto			
CJNE	A	,#data ,rel	3	2	se Acc (#data)			
	Rn				se Rn (#data)			
	@Ri				se @Ri (#data)			

4.8.5. Loops :

As instruções de loops são do tipo "decremente e desvie se for diferente de zero" (DJNZ). Há duas instruções, as quais são muito úteis para a repetição de determinadas partes do programa. Utilizam como contador um registro ou um byte da RAM Interna e por isso têm um limite de até 256 repetições. Como são baseadas em desvios relativos, têm um alcance de 127 bytes para adiante e de 128 bytes para trás.

			bytes	MC	CY	AC	OV
DJNZ	Rn	,rel	2	2	-	-	-
	direto						

4.8.6. Não Operação :

Há uma instrução que não faz nada, consumindo apenas ciclos da CPU. É muito útil para reservar espaços na memória de programa quando se trabalha com EPROM.

	bytes	MC	CY	AC	OV
NOP	1	1	-	-	-

4.9. INSTRUÇÕES E FLAGS

Para a perfeita utilização das instruções é necessário conhecer como estas alteram os flags. Em seguida apresenta-se uma pequena tabela com um resumo dessas informações. Na tabela, 0 (zero) indica que o flag é apagado, 1 (um) indica que é ativado, X (don't care) significa que o flag será ativado ou apagado de acordo com o resultado da instrução e "-" indica que o flag não é alterado.

INSTR/FLAG	C	OV	AC
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	-
DIV	0	X	-
DA	X	-	-
RRC	X	-	-
RLC	X	-	-
SETB C	1	-	-
CLR C	0	-	-
CPL C	X	-	-
ANL C,bit	X	-	-
ANL C,/bit	X	-	-
ORL C.bit	X	-	-
ORL C,/bit	X	-	-
MOV C,bit	X	-	-
CJNE	X	-	-

Figura 4.3. Comportamento dos flags.

4.10. OBSERVAÇÕES

Aqui convém fazer uma pausa para alguns comentários e exemplos sobre endereçamento e uso das instruções.

4.10.1. Bancos de Registros :

Existem quatro bancos, selecionados por RS1 e RS0. Estes bancos podem ser acessados como registros, com endereçamento direto ou com endereçamento indireto.

	BANCO 0	**BANCO 1**	**BANCO 2**	**BANCO 3**
	R0 R1...R7	R0 R1...R7	R0 R1...R7	R0 R1...R7
Endereço:	00 01 07	08 09 15	16 17 23	24 25 31

Supondo que o banco 1 esteja selecionado, as instruções seguintes dão como resultado a mesma operação mas algumas consomem menos bytes que outras.

MOV	A,R1	1 Byte	1MC	(A,Rn)
MOV	A,9	2 Bytes	1MC	(A,direto)

Também se pode fazer uma seqüência para endereçamento indireto:

MOV	R0,#9	2 Bytes	1MC
MOV	A,@R0	1 Byte	1MC
	Total	3 Bytes	2MC

Os registros de outros bancos são acessados empregando endereçamento direto ou indireto :

MOV	A,23	2 bytes	1 MC	(23 e/ou R7 do banco 2)

Aqui são apresentadas algumas instruções interessantes para realizar transferências entre registros (supõe-se que o banco 3 esteja selecionado) :

MOV	R1,R6	(não existe)
MOV	R1,30	(equivale a MOV R1,R6)
MOV	25,R6	(equivale a MOV R1,R6)
MOV	R1,R1	(não existe)
MOV	R1,25	(equivale a MOV R1,R1)
MOV	25,R1	(equivale a MOV R1,R1)

4.10.2. Registros Especiais (SFR) :

Deve-se notar que todos os SFR, exceto o Acumulador, são acessados empregando endereçamento direto. Para que exista compatibilidade com o 8052, não se permite endereçamento indireto para os SFR. No 8052, quando se usa endereçamento indireto com endereços maiores que 07FH, acessa-se a região chamada 128 UPPER. As duas instruções a seguir dão o mesmo resultado; na verdade são idênticas e geram o mesmo opcode.

```
MOV     B,#10       2 bytes     1 MC
MOV     0F0H,#10    2 bytes     1 MC
```

O "B" que se usa aqui é na verdade um label para o endereço 0F0H da RAM Interna. Todos os programas montadores que trabalham com a família MCS-51 trazem os endereços dos SFR já definidos, quer dizer, é como se existissem as seguintes declarações (isto já havia sido afirmado no início deste capítulo) :

```
B       EQU     0F0H
Acc     EQU     0E0H
PSW     EQU     0D0H
        ...
SP      EQU     081H
P0      EQU     080H
```

Deve-se ter cuidado com o acumulador pois existem dois símbolos para o mesmo: A e Acc:

A → para as instruções com registro especifico

Acc → para usar acumulador por endereçamento direto

Esses dois símbolos permitem construir instruções interessantes que, se não forem levadas em conta, podem consumir muita memória. Exemplos :

```
MOV     R7,A        1 byte      1 MC  (MOV Rn,A)
MOV     R7,Acc      2 bytes     1 MC  (MOV Rn,direto)
MOV     A,Acc       2 bytes     1 MC  (MOV A,direto)
MOV     A,B         2 bytes     1 MC  (MOV A,direto)
MOV     Acc,B       3 bytes     2 MC  (MOV direto,direto)
```

4.11. CÓDIGOS DE OPERAÇÃO (OPCODE)

As instruções da família MCS-51, para que possam ser executadas pelo microcontrolador, devem ser traduzidas em códigos de operação (opcode). São estes opcodes que a CPU compreende e executa. A cada instrução é associado somente um opcode.

4.11.1. Tabelas de Instruções :

As seguintes tabelas mostram a distribuição dos opcodes da família MCS-51. As tabelas devem ser vistas por colunas. Notar que o opcode "A5" não tem instrução.

	0_	1_	2_	3_	4_	5_	6_	7_
_0	NOP	JBC bit,rel	JB bit,rel	JNB bit,rel	JC rel	JNC rel	JZ rel	JNZ rel
_1	AJMP adr11	ACALL adr11	AJMP adr11	ACALL adr11	AJMP adr11	ACALL adr11	AJMP adr11	ACALL adr11
_2	LJMP adr16	LCALL adr16	RET	RETI	ORL dir,A	ANL dir,A	XRL dir,A	ORL C,bit
_3	RR A	RRC A	RL A	RLC A	ORL dir,#dt	ANL dir,#dt	XRL dir,#dt	JMP @A+DPTR
_4	INC A	DEC A	ADD A,#dt	ADDC A,#dt	ORL A,#dt	ANL A,#dt	XRL A,#dt	MOV A,#dt
_5	INC dir	DEC dir	ADD A,dir	ADDC A,dir	ORL A,dir	ANL A,dir	XRL A,dir	MOV dir,#dt
_6	INC @R0	DEC @R0	ADD A,@R0	ADDC A,@R0	ORL A,@R0	ANL A,@R0	XRL A,@R0	MOV @R0,#dt
_7	INC @R1	DEC @R1	ADD A,@R1	ADDC A,@R1	ORL A,@R1	ANL A,@R1	XRL A,@R1	MOV @R1,#dt
_8	INC R0	DEC R0	ADD A,R0	ADDC A,R0	ORL A,R0	ANL A,R0	XRL A,R0	MOV R0,#dt
_9	INC R1	DEC R1	ADD A,R1	ADDC A,R1	ORL A,R1	ANL A,R1	XRL A,R1	MOV R1,#dt
_A	INC R2	DEC R2	ADD A,R2	ADDC A,R2	ORL A,R2	ANL A,R2	XRL A,R2	MOV R2,#dt
_B	INC R3	DEC R3	ADD A,R3	ADDC A,R3	ORL A,R3	ANL A,R3	XRL A,R3	MOV R3,#dt
_C	INC R4	DEC R4	ADD A,R4	ADDC A,R4	ORL A,R4	ANL A,R4	XRL A,R4	MOV R4,#dt
_D	INC R5	DEC R5	ADD A,R5	ADDC A,R5	ORL A,R5	ANL A,R5	XRL A,R5	MOV R5,#dt
_E	INC R6	DEC R6	ADD A,R6	ADDC A,R6	ORL A,R6	ANL A,R6	XRL A,R6	MOV R6,#dt
_F	INC R7	DEC R7	ADD A,R7	ADDC A,R7	ORL A,R7	ANL A,R7	XRL A,R7	MOV R7,#dt

	8_	9_	A_	B_	C_	D_	E_	F_
_0	SJMP rel	MOV DPTR,#dt16	ORL C,/bit	ANL C,/bit	PUSH dir	POP dir	MOVX A,@DPTR	MOVX @DPTR,A
_1	AJMP adr11	ACALL adr11	AJMP adr11	ACALL adr11	AJMP adr11	ACALL adr11	AJMP adr11	ACALL adr11
_2	ANL C,bit	MOV bit,C	MOV C,bit	CPL bit	CLR bit	SETB bit	MOVX A,@R0	MOVX @R0,A
_3	MOVC A,@A+PC	MOVC A,@A+DPTR	INC DPTR	CPL C	CLR C	SETB C	MOVX A,@R1	MOVX @R1,A
_4	DIV AB	SUBB A,#dt	MUL AB	CJNE A,#dt,rel	SWAP A	DA A	CLR A	CPL A
_5	MOV dir,dir	SUBB A,dir	----------	CJNE A,dir,rel	XCH A,dir	DJNZ dir,rel	MOV A,dir	MOV dir,A
_6	MOV dir,@R0	SUBB A,@R0	MOV @R0,dir	CJNE @R0,#dt,rel	XCH A,@R0	XCHD A,@R0	MOV A,@R0	MOV @R0,A
_7	MOV dir,@R1	SUBB A,@R1	MOV @R1,dir	CJNE @R1,#dt,rel	XCH A,@R1	XCHD A,@R1	MOV A,@R1	MOV @R1,A
_8	MOV dir,R0	SUBB A,R0	MOV R0,dir	CJNE R0,#dt,rel	XCH A,R0	DJNZ R0,rel	MOV A,R0	MOV R0,A
_9	MOV dir,R1	SUBB A,R1	MOV R1,dir	CJNE R1,#dt,rel	XCH A,R1	DJNZ R1,rel	MOV A,R1	MOV R1,A
_A	MOV dir,R2	SUBB A,R2	MOV R2,dir	CJNE R2,#dt,rel	XCH A,R2	DJNZ R2,rel	MOV A,R2	MOV R2,A
_B	MOV dir,R3	SUBB A,R3	MOV R3,dir	CJNE R3,#dt,rel	XCH A,R3	DJNZ R3,rel	MOV A,R3	MOV R3,A
_C	MOV dir,R4	SUBB A,R4	MOV R4,dir	CJNE R4,#dt,rel	XCH A,R4	DJNZ R4,rel	MOV A,R4	MOV R4,A
_D	MOV dir,R5	SUBB A,R5	MOV R5,dir	CJNE R5,#dt,rel	XCH A,R5	DJNZ R5,rel	MOV A,R5	MOV R5,A
_E	MOV dir,R6	SUBB A,R6	MOV R6,dir	CJNE R6,#dt,rel	XCH A,R6	DJNZ R6,rel	MOV A,R6	MOV R6,A
_F	MOV dir,R7	SUBB A,R7	MOV R7,dir	CJNE R7,#dt,rel	XCH A,R7	DJNZ R7,rel	MOV A,R7	MOV R7,A

4.11.2. Instruções em Ordem Alfabética com OPCODES :

A seguir serão apresentadas todas a instruções em ordem alfabética. Esta lista é para a construção dos opcodes. As tabelas também apresentam essas mesmas informações, mas aqui elas estão mais detalhadas.

OBSERVAÇÃO: Empregar-se-á a seguinte notação :

n = 0,1,2,3,4,5,6,7

i = 0,1

data = palavra de 8 bits (byte)

data16 = palavra de 16 bits

adr16 = um endereço de 16 bits (Memória Externa)

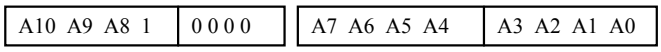
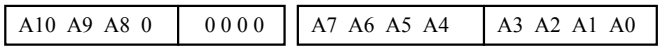
adr11 = um endereço de 11 bits (Memória Externa)

direct ou **dir** = um endereço da RAM Interna (8 bits)

rel = um deslocamento relativo

MSB = byte mais significativo de uma palavra de 16 bits

LSB = byte menos significativo de uma palavra de 16 bits

1)	ACALL	adr11			
2)	ADD	A,Rn	28+n		
3)	ADD	A,direct	25	direct	
4)	ADD	A,@Ri	26+i		
5)	ADD	A,#data	24	data	
6)	ADDC	A,Rn	38+n		
7)	ADDC	A,direct	35	direct	
8)	ADDC	A,@Ri	36+i		
9)	ADDC	A,#data	34	data	
10)	AJMP	adr11			
11)	ANL	A,Rn	58+n		
12)	ANL	A,direct	55	direct	
13)	ANL	A,@Ri	56+i		
14)	ANL	A,#data	54	data	
15)	ANL	direct,A	52	direct	
16)	ANL	direct,#data	53	direct	data
17)	ANL	C,bit	82	bit	
18)	ANL	A,/bit	B0	bit	
19)	CJNE	A,direct,rel	B5	direct	relativo
20)	CJNE	A,#data,rel	B4	data	direct
21)	CJNE	Rn,#data,rel	B8+n	data	relativo

22)	CJNE	@Ri,#data,rel	B6+i	data	relativo
23)	CLR	A	E4		
24)	CLR	bit	C2	bit	
25)	CLR	C	C3		
26)	CPL	A	F4		
27)	CPL	bit	B2	bit	
28)	CPL	C	B3		
29)	DA	A	D4		
30)	DEC	A	14		
31)	DEC	Rn	18+n		
32)	DEC	direct	15	direct	
33)	DEC	@Ri	16+i		
34)	DIV	AB	84		
35)	DJNZ	Rn,rel	D8+n	relativo	
36)	DJNZ	direct,rel	D5	direct	relativo
37)	INC	A	04		
38)	INC	Rn	08+n		
39)	INC	direct	05	direct	
40)	INC	@Ri	06+i		
41)	INC	DPTR	A3		
42)	JB	bit,rel	20	bit	relativo
43)	JBC	bit,rel	10	bit	relativo
44)	JC	rel	40	relativo	
45)	JMP	@A+DPTR	73		
46)	JNB	bit,rel	30	bit	relativo
47)	JNC	rel	50	relativo	
48)	JNZ	rel	70	relativo	
49)	JZ	rel	60	relativo	
50)	LCALL	adr16	12	MSB(adr16)	LSB(adr16)
51)	LJMP	adr16	02	MSB(adr16)	LSB(adr16)
52)	MOV	A,Rn	E8+n		
53)	MOV	A,direct	E5	direct	
54)	MOV	A,@Ri	E6+i		
55)	MOV	A,#data	74	data	
56)	MOV	Rn,A	F8+n		
57)	MOV	Rn,direct	A8+n	direct	
58)	MOV	Rn,#data	78+n	data	
59)	MOV	direct,A	F5	direct	

60)	MOV	direct,Rn	88+n	direct	
61)	MOV	direct,direct	85	dir(fonte)	dir(destino)
62)	MOV	direct,@Ri	86+i	direct	
63)	MOV	direct,#data	75	direct	data
64)	MOV	@Ri,A	F6+i		
65)	MOV	@Ri,direct	A6+i	direct	
66)	MOV	@Ri,#data	76+i	data	
67)	MOV	bit,C	92	bit	
68)	MOV	C,bit	A2	bit	
69)	MOV	DPTR,#data16	90	MSB(data16)	LSB(data16)
70)	MOVC	A,@A+DPTR	93		
71)	MOVC	A,@A+PC	83		
72)	MOVX	A,@Ri	E2+i		
73)	MOVX	A,@DPTR	E0		
74)	MOVX	@Ri,A	F2+i		
75)	MOVX	DPTR,A	F0		
76)	MUL	AB	A4		
77)	NOP		00		
78)	ORL	A,Rn	48+n		
79)	ORL	A,direct	45	direct	
80)	ORL	A,@Ri	46+i		
81)	ORL	A,#data	44	data	
82)	ORL	direct,A	42	direct	
83)	ORL	direct,#data	44	data	
84)	ORL	C,bit	72	bit	
85)	ORL	C,/bit	A0	bit	
86)	POP	direct	D0	direct	
87)	PUS	direct	C0	direct	
88)	RET		22		
89)	RETI		32		
90)	RL	A	23		
91)	RLC	A	33		
92)	RR	A	03		
93)	RRC	A	13		
94)	SETB	bit	D2	bit	
95)	SETB	C	D3		
96)	SJMP	rel	80	relative	
97)	SUBB	A,Rn	98+n		

98)	SUBB	A,direct	95	direct	
99)	SUBB	A,@Ri	96+i		
100)	SUBB	A,#data	94	data	
101)	SWAP	A	C4		
102)	XCH	A,Rn	C8+n		
103)	XCH	A,direct	C5	direct	
104)	XCH	A,@Ri	C6+i		
105)	XCHD	A,@Ri	D6+i		
106)	XRL	A,Rn	68+i		
107)	XRL	A,direct	65	direct	
108)	XRL	A,@Ri	66+i		
109)	XRL	A,#data	64	data	
110)	XRL	direct,A	62	direct	
111)	XRL	direct,#data	63	direct	data

4.12. JUMP E CALL

Segundo o modo de formação do endereço pode-se identificar 3 tipos de JUMPs e dois tipos de CALLs:

Os 3 JUMPs são: relativo, absoluto e longo.

Os 2 CALLs são: absoluto e longo.

Neste item serão esclarecidas estas instruções para evitar erros que são muito freqüentes quando se começa a trabalhar com a família MCS-51.

4.12.1. JUMPs Relativos :

Existem dois tipos de JUMPs relativos: os incondicionais e os condicionais. Estes últimos (os condicionais) realizam um desvio de acordo com o estado de um registro, de um flag ou de um bit, enquanto o outro tipo (incondicional) sempre realiza o desvio. Exemplo:

- JUMP incondicional: SJMP rel
- JUMP condicional: JNC rel

Nos JUMPs relativos especifica-se quantos bytes (deslocamento) se deseja desviar, para frente ou para trás. Este deslocamento é codificado em complemento a 2. Como há apenas um byte para este deslocamento, pode-se saltar 127 bytes para frente ou 128 bytes para trás.

Agora serão apresentados alguns exemplos com JUMPs relativos. Os programas, por simplicidade, sempre iniciarão no endereço 200H. Ao lado de cada instrução estarão os opcodes.

EXEMPLO 1 : DESVIO CONDICIONAL PARA FRENTE

```
        ORG     200H
        ADD     A,#30H
        JC      LABEL
        ADD     A,#20H
        DA      A
LABEL   MOV     B,A
```

O programa apresenta um desvio condicional para frente. A seguir tem-se este mesmo programa com os endereços de memória e seu conteúdo.

endereço	**conteúdo**	**instrução**		**deslocamento**
200	24	ADD	A,#30H	-4
201	30			-3
202	40	JC	LABEL	-2
203	03			-1
204	24	ADD	A,#20H	+0
205	20			+1
206	D4	DA	A	+2
207(LABEL)	F5	MOV	B,A	+3
208	F0			+4

É conveniente ressaltar que quando a instrução "JC LABEL" está sendo executada, o Program Counter (PC) já está apontando para a instrução seguinte; por isso, o deslocamento +0 está na instrução "ADD A,#20H".

EXEMPLO 2 : DESVIO CONDICIONAL PARA TRÁS

```
        ORG     200H
        INC     A
LABEL   MOV     A,B
        ADD     A,#20H
        DA      A
        JC      LABEL
        MOV     R0,A
```

O programa apresenta um desvio condicional para trás e a seguir pode-se observar os endereços de memória e seu conteúdo.

endereço	conteúdo	instrução		deslocamento
200	04	INC	A	-8 (F8)
201(LABEL)	E5	MOV	A,B	-7 (F9)
202	F0			-6 (FA)
203	24	ADD	A,#20H	5 (FB)
204	20			-4 (FC)
205	D4	DA	A	-3 (FD)
206	40	JC	LABEL	-2 (FE)
207	F9			-1 (FF)
208	F8	MOV	R0,A	+0
209	75	MOV	B,#50H	+1
20A	F0			+2
20B	50			+3

4.12.2. JUMPs e CALLs Absolutos :

Os JUMPs e CALLs absolutos são sempre incondicionais. Neste tipo de instrução apenas se trocam alguns bits do PC pelos bits especificados na instrução (são trocados os 11 bits menos significativos do PC). Isto significa que os JUMPs e CALLs estão limitados a um alcance de 2 KB, quer dizer, é como se a memória estivesse dividida em páginas de 2 KB e se pudesse saltar dentro dos limites de cada página. Estas instruções permitem JUMPs e CALLs que consomem poucos bytes. Deve-se ter cuidado ao escrever os opcodes.

EXEMPLO 3 : DESVIO ABSOLUTO

```
        ORG     200H
        ADD     A,#30H
        JNC     LABEL
        AJMP    LB1
LABEL   ADD     A,#20H
        ...
        ORG     765H
LB1     DA      A
        ...
```

O programa acima apresenta um desvio condicional para frente e abaixo se tem o programa com os endereços de memória e seu conteúdo.

endereço	conteúdo	instrução		deslocamento
200	24	ADD	A,#30H	-4
201	30			-3
202	50	JNC	LABEL	-2
203	02			-1
204	E0	AJMP	LB1	+0
205	65			+1
206(LABEL)	24	ADD	A,#20H	+2
207	20			+3
...				
765(LB1)	04	DA	A	--

OBSERVAÇÃO:

AJMP	LB1 →	A10	A9	A8	0	0	0	0	0	A7	A6	A5	A4	A3	A2	A1	A0
E0 65		1	1	1	0	0	0	0	0	0	1	1	0	0	1	0	1

EXEMPLO 4 : CALL ABSOLUTO

```
        ORG     400H
        ADD     A,#30H
        JNC     LABEL
        ACALL   LB1
LABEL   ADD     A,#20H
        ...
        ORG     234H
LB1     DA      A
        ...
```

O programa é semelhante ao anterior mas o AJMP foi trocado por um ACALL. Notar que a instrução ACALL está chamando uma rotina que está atrás, mas dentro do bloco de 2 KB.

endereço	conteúdo	instrução		deslocamento
400	24	ADD	A,#30H	-4
401	30			-3
402	50	JNC	LABEL	-2
403	02			-1
404	50	ACALL	LB1	+0
405	34			+1
406(LABEL)	24	ADD	A,#20H	+2

```
407          20                                          +3
...
234(LB1)     D4              DA      A                   --
```

OBSERVAÇÃO:

```
ACALL  LB1 →  A10 A9 A8 1     0 0 0 0     A7 A6 A5 A4     A3 A2 A1 A0
50  34          0   1  0  1     0 0 0 0      0  0   1  1       0  1 0   0
```

EXEMPLO 5 : DESVIO ABSOLUTO (COM ERRO)

```
            ORG     700H
            ADD     A,#30H
            JNC     LABEL
            AJMP    LB1
LABEL       ADD     A,#20H
            ...
            ORG     900H
LB1         DA      A
            ...
```

Este programa vai apresentar um erro do tipo "illegal range" devido ao fato que o AJUMP está no primeiro bloco de 2 KB (0 até 7FF) e o label está no segundo bloco (800 até FFF).

EXEMPLO 6 : DESVIO ABSOLUTO (COM ERRO)

```
            ORG     7FFH
            AJMP    LB1
            ...
            ORG     700H
LB1         DA      A
            ...
```

Aparentemente não há erro pois o AJMP e o label estão no mesmo bloco de 2 KB. Na verdade, há um erro porque quando se inicia a execução de uma instrução o PC já está apontando para a instrução seguinte, ou seja, quando a CPU inicia a execução do AJMP, o PC está em 801H (AJMP usa dois bytes). O PC e o label estão em dois blocos (de 2 KB) distintos. Os programas montadores devem ser suficientemente inteligentes para detetar este tipo de erro.

4.13. EXEMPLOS

A seguir é apresentada uma seqüência de exemplos que vão ilustrar a utilização das instruções do MCS-51. Supõe-se que a CPU é o 8031, que os programas serão usados como subrotinas e que sempre iniciam no endereço 200H. Nos primeiros exemplos serão apresentadas também as listagens em hexadecimal dos opcodes.

EXEMPLO 1. SUBROTINA SUM:

R7 = R1 + R0.

Colocar em R7 o resultado da soma de R0 e R1.

```
        ORG   200H
SUM:    MOV   A,R0
        ADD   A,R1
        MOV   R7,A
        RET
```

Listagem em hexadecimal com os opcodes:

```
200     E8      MOV     A,R0
201     29      ADD     A,R1
202     FF      MOV     R7,A
203     22      RET
```

Deve-se ter cuidado para que a soma (R0+R1) não seja maior que 256. A melhor solução é ter uma subrotina geral e por isso o resultado é armazenado em 2 registros: R7 (MSB) R6 (LSB). O resultado será maior que 256 quando existir um carry.

A seguir está a nova subrotina SUM (R7 R6 = R1 + R0).

```
        ORG     200H
SUM:    MOV     A,R0
        ADD     A,R1
        MOV     R6,A    ;R6 guarda o LSB
        CLR     A       ;zera o acumulador
        ADDC    A,#0    ;acrescenta o carry
        MOV     R7,A    ;R7 guarda o MSB
        RET
```

Listagem em hexadecimal com os opcodes:

```
200     E8      MOV     A,R0
201     29      ADD     A,R1
202     FE      MOV     R6,A
203     E4      CLR     A
204     34      ADDC    A,#0
205     00
206     FF      MOV     R7,A
207     22      RET
```

A instrução "ADDC A,#0" pode ser substituída pela instrução "RLC A", com a vantagem de se economizar um byte. Isto está na listagem hexadecimal a seguir.

Listagem em hexadecimal com os opcodes:

```
200     E8      MOV     A,R0
201     29      ADD     A,R1
202     FE      MOV     R6,A
203     E4      CLR     A
204     33      RLC     A
205     FF      MOV     R7,A
206     22      RET
```

EXEMPLO 2. SUBROTINA SUB:

R7 = R1 - R0.

Colocar em R7 o resultado da subtração de R1 e R0

```
        ORG     200H
SUB:    MOV     A,R1
        CLR     C       ;zera o carry (necessario na subtracao)
        SUBB    A,R0
        MOV     R7,A
        RET
```

Listagem em hexadecimal com os opcodes:

```
200     E9      MOV     A,R1
201     C3      CLR     C
202     98      SUBB    A,R0
203     FF      MOV     R7,A
```

```
204         22          RET
```

Um cuidado que se deve ter ao usar a instrução SUBB é o de zerar o carry pois o SUBB sempre o utiliza.

EXEMPLO 3. SUBROTINA MUL:

R7 R6 = R1 * R0.

Colocar em R7 (MSB) e em R6 (LSB) o produto de R0 e R1

```
            ORG     200H
MUL:        MOV     A,R0
            MOV     B,R1
            MUL     AB
            MOV     R6,A        ;guarda o LSB
            MOV     R7,B        ;guarda o MSB
            RET
```

Listagem em hexadecimal com os opcodes:

```
200         E8          MOV     A,R0
201         89          MOV     B,R1
202         F0
203         A4          MUL     AB
204         FE          MOV     R6,A
205         AF          MOV     R7,A
206         F0          MOV     R7,B
207         22          RET
```

EXEMPLO 4. SUBROTINA DIV:

R7(quoc.) R6(resto) R1/R0.

Dividir R1 por R0 e guardar o resultado em: R7(quociente) e R6(resto)

```
            ORG     200H
DIV:        MOV     A,R1        ;dividendo em A
            MOV     B,R0        ;divisor em B
            DIV     AB
            MOV     R7,A        ;guarda o quociente
            MOV     R6,B        ;guarda o resto
            RET
```

Listagem em hexadecimal com os opcodes:

```
200     E9      MOV     A,R1
201     88      MOV     B,R0
202     F0
203     84      DIV     AB
204     FF      MOV     R7,A
205     AE      MOV     R6,B
206     F0
207     22      RET
```

EXEMPLO 5. SUBROTINA DIV_7:

Verificar se o byte que está em R3 é múltiplo de 7 e , se for, ativar o bit 0 da porta P1.

```
        ORG     200H
DIV_7:  MOV     A,R3
        MOV     B,#7
        DIV     AB      ;dividir por 7
        MOV     A,B     ;coloca o resto em A
        JZ      LB      ;ha resto ?
        CLR     P1.0    ;zera bit P1.0
        RET
LB:     SETB    P1.0    ;seta bit P1.0
        RET
```

Listagem em hexadecimal com os opcodes:

```
        200     EB      MOV     A,R3
        201     75      MOV     B,#7
        202     F0
        203     07
        204     84      DIV     AB
        205     E5      MOV     A,B
        206     F0
        207     60      JZ      LB
        208     03
        209     C2      CLR     P1.0
        20A     90
        20B     22      RET
```

```
20C (LB)    D2          SETB        P1.0
20D         90
20E         22          RET
```

Há uma outra solução que ocupa um byte a menos. Inicia-se com a hipótese de que o número não é divisível por 7 (zerar P1.0) e em seguida se realiza o teste; caso o numero for divisível inverte-se P1.0. Economiza-se um RET.

```
200         C2          CLR         P1.0
201         90
202         EB          MOV         A,R3
203         75          MOV         B,#7
204         90
205         07
206         84          DIV         AB
207         E5          MOV         A,B
208         F0
209         70          JNZ         LB
20A         02
20B(LB)     B2          CPL         P1.0
20C         90
20D         22          RET
```

EXEMPLO 6. SUBROTINA INICIALIZAR:

Inicializar com zero toda a RAM interna, ou seja, os endereços de 0 a 127

```
            ORG     200H
ZERAR:      CLR     RS1         ;seleciona banco 0
            CLR     RS0         ;seleciona banco 0
            CLR     A
            MOV     R0,#127     ;maior endereço
LB:         MOV     @R0,A
            DJNZ    R0,LB
            RET
```

Listagem em hexadecimal com os opcodes:

```
200         C2          CLR     RS1
201         D4
```

```
202          C2          CLR     RS0
203          D3
204          E4          CLR     A
205          78          MOV     R0,#127
206          7F
207(LB)      F6          MOV     @R0,A
208          D8          DJNZ    R0,LB
209          FD
20A          22          RET
```

EXEMPLO 7. SUBROTINA MUL16:

R7 R6 R5 R4 = (R3 R2)*(R1 R0)

Em R1 e R0 existe um número de 16 bits e em R3 e R2 existe um outro. Multiplicar estes dois números de 16 bits e guardar o resultado em R7, R6, R5 e R4.

A solução é muito simples pois esta multiplicação pode ser vista como 4 multiplicações de 8 bits. A equação a seguir ilustra este efeito.

$$(2^8.R3 + R2)*(2^8.R1 + R0) = 2^{16}(R3.R1)+2^8(R3.R0+R2.R1)+(R2.R0)$$

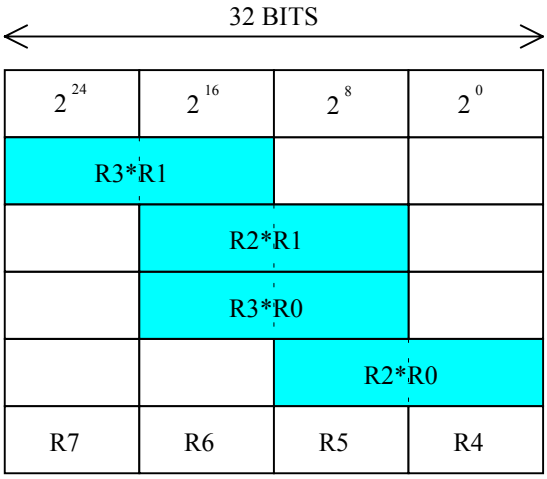


Figura 4.4. Explicação da rotina de multiplicação.

```
MUL16:   MOV     A,R2        ;
         MOV     B,R0        ;
         MUL     AB          ;R2*R0
         MOV     R4,A        ;R4 contem LSB(R2*R0)
         MOV     R5,B        ;R5 contem MSB(R2*R0)
         MOV     A,R3        ;;
         MOV     B,R0        ;;
         MUL     AB          ;R3*R0
```

```
        ADD     A,R5        ;A contem LSB(R3*R0)+MSB(R2*R0)
        MOV     R5,A        ;coloca em R5 o valor de A
        CLR     A           ;zera A
        ADDC    A,B         ;A contem Carry + 0 + MSB(R3*R0)
        MOV     R6,A        ;coloca em R6 o valor de A
        MOV     A,R2        ;
        MOV     B,R1        ;
        MUL     AB          ;R2*R1
        ADD     A,R5        ;A contem LSB(R2*R1)+LSB(R3*R0)+MSB(R2*R0)
        MOV     R5,A        ;coloca em R5 o valor de A
        MOV     A,B         ;A contem MSB(R2*R1)
        ADDC    A,R6        ;A contem Carry+MSB(R3*R0)+MSB(R2*R1)
        MOV     R6,A        ;coloca em R6 o valor de A
        MOV     A,R3        ;;
        MOV     B,R1        ;;
        MUL     AB          ;R3*R1
        ADD     A,R6        ;A contem LSB(R3*R1)+MSB(R2*R1)+MSB(R3*R0)
        MOV     R6,A        ;coloca em R6 o valor de A
        CLR     A           ;zera A
        ADDC    A,B         ;A contem Carry + 0 + MSB(R3*R1)
        MOV     R7,A        ;coloca em R7 o valor de A
        RET
```

EXEMPLO 8. SUBROTINA SUMBCD:

Em R1 e R0 existem 2 números BCD que deverão ser somados e o resultado deve ser armazenado em R7 e R6

```
SUMBCD: MOV     A,R1
        ADD     A,R0    ;A contem R0 + R1
        DA      A       ;ajuste decimal depois de uma soma BCD
        MOV     R6,A    ;coloca em R6
        CLR     A       ;zera A
        ADDC    A,#0    ;A contem 0 + Carry + 0
        MOV     R7,A    ;coloca em R7
        RET
```

EXEMPLO 9. SUBROTINA SUBBCD:

Em R1 e R0 encontra-se um número em BCD de quatro dígitos. Subtrair 86 deste número e colocar o resultado em R7 e R6.

Como o "decimal adjust" (DA A) só funciona em somas, será necessário realizar a subtração utilizando complemento a 10. Deve-se notar que 99 funciona como -1 e que 98 como -2. O complemento a 10 de 86 é 9914.

```
   76                76                10000
+  99 (-1)        +  98 (-2)        -    86
-------           -------           -------
 1 75              1 74              9914
   ↑                 ↑
ignorar este 1    ignorar este 1    complemento a 10 de 86
```

```
SUBBCD:  MOV    A,R0
         ADD    A,#14H   ;R0 + LSB de -86
         DA     A        ;ajuste decimal da soma
         MOV    R6,A     ;coloca em R6
         MOV    A,R1
         ADDC   A,#99    ;R1 + MSB de -86
         DA     A        ;ajuste decimal da soma
         MOV    R7,A     ;coloca em R7
         RET
```

EXEMPLO 10. SUBROTINA BINBCD:

Converter um número binário para BCD.

Existem vários casos de acordo com o valor do número a ser convertido:

caso a: 0 a 99 (2 algarismos BCD)
caso b: 0 a 255 (1 byte)
caso c: 0 a 999 (3 algarismos BCD)
caso d: 0 a 9999 (4 algarismos BCD)

CASO a: Em R4 existe um número de 0 a 99. Convertê-lo para BCD e colocar o resultado em R6. Ao dividir o número por 10, a unidade termina em Acc e a dezena em B.

```
BINBCD_99:  MOV    A,R4
            MOV    B,#10
            DIV    AB          ; A=0X   B=0Y
```

```
        SWAP    A       ; A=X0   B=0Y
        ADD     A,B     ; A=XY   B=0Y
        MOV     R6,A    ;coloca em R6 o valor de A
        RET
```

CASO b: Em R4 existe um número de 0 a 255. Convertê-lo para BCD e colocar o resultado em R7 e R6. Divide-se o número por 100 para separar a centena e depois usa-se a solução do caso a.

```
BINBCD_255:  MOV     A,R4
             MOV     B,#100
             DIV     AB      ;separar centena (divide por 100)
             MOV     R7,A    ;R7 guarda o quociente (digito BCD mais sigf.)
             MOV     A,B     ;igual a BINBCD_99
             MOV     B,#10
             DIV     AB
             SWAP    A
             ADD     A,B
             MOV     R6,A
             RET
```

CASO c: Em R5 e R4 existe um número de 0 a 999. Convertê-lo para BCD e colocar o resultado em R7 e R6. A solução usada aqui é a de somar 256 em R7 e R6 e decrementar R5 até que este seja 0; depois se usa uma rotina idêntica à BINBCD_255.

```
BINBCD_999:  CLR     A       ;zera A
             MOV     R7,A    ;poe zero em R7 e R6 para
             MOV     R6,A    ;receber os resultados
             MOV     A,R5
             JZ      LB1     ;se R5=0 (0<= num <=255), seguir adiante
             ;
LB2:         MOV     A,R6
             ADD     A,#56H  ;somar 256 BCD e decrementar
             DA      A       ;R5 ate que este chegue
             MOV     R6,A    ;a zero
             MOV     A,R7
             ADDC    A,#2
             DA      A
             MOV     R7,A
```

```
        DJNZ    R5,LB2      ;decrementar R5
        ;
LB1:    MOV     A,R4        ;parecido com BINBCD_255 (0<= num <=255)
        MOV     B,#100
        DIV     AB
        ADD     A,R7
        DA      A
        MOV     R7,A
        MOV     A,B
        MOV     B,#10
        DIV     AB
        SWAP    A
        ADD     A,B
        ADD     A,R6
        DA      A
        MOV     R6,A
        MOV     A,R7        ;ha posibilidade de que se
        ADDC    A,#0        ;necessite passar um carry adiante
        DA      A
        MOV     R7,A
        RET
```

CASO d: Em R5 e R4 existe um número do 0 a 9999. Convertê-lo para BCD e colocar seu resultado em R7 e R6.

A solução adotada no Caso c sugere um loop para zerar R5. Mas para números muito grandes esse loop pode ser excessivo em termos de tempo e por isso a melhor solução seria usar divisões por 1000, 100 e 10. O problema agora será como realizar divisões de 2 números de 16 bits usando a divisão de 8 bits que se tem disponível.

Uma maneira de solucionar este problema é converter o LSB para BCD e depois converter o MSB para BCD, multiplicá-lo por 256 e somar com o anterior. Uma multiplicação por 256 é fácil: multiplique por 200 (multiplique por 2 e desloque 2 posições BCD para a esquerda); multiplique por 50 (multiplique por 5 e desloque 1), multiplique por 6 e finalmente some os três resultados.

Com esta solução pode-se pensar em um caso e mais geral:

CASO e: Em R4 e R3 existe um número binário de 16 bits (0 a 9999). Convertê-lo para BCD e colocar seu resultado em R7, R6 e R5. Na solução serão utilizadas algumas subrotinas auxiliares.

Converte um byte do ACC para BCD e coloca o resultado em AUXH e AUXL

```
BCD8:   MOV     B,#100      ;parecido com BINBCD_255
        DIV     AB
        MOV     AUXH,A
        MOV     A,B
        MOV     B,#10
        DIV     A,B
        SWAP    A
        ADD     A,B
        MOV     AUXL,A
        RET
```

Recebe um número BCD em R2 e R1 e o incrementa o número de vezes especificado em R0 (R0 > 0).

```
ROT_INC:  MOV     R1AUX,R1
          MOV     R2AUX,R2
          DEC     R0
R_INC:    MOV     A,R1AUX
          ADD     A,R1
          DA      A
          MOV     R1,A
          MOV     A,R2AUX
          ADDC    A,R2
          DA      A
          MOV     R2,A
          DJNZ    R0,R_INC
          RET
```

Somar dois números BCD, um número está em R2 e R1, enquanto o outro está indicado pelo ponteiro R0. O resultado deve ser guardado no endereço apontado por R0.

[@(R0+1) e @R0] [@(R0+1) e @R0] + [R2 e R1]

```
SOMA:   MOV     A,@R0
        ADD     A,R1        ;soma os LSBs
        DA      A           ;ajuste decimal da soma
        MOV     @R0,A       ;guarda a soma em @R0
```

```
        INC     R0      ;vai para proximo endereco
        MOV     A,@R0
        ADDC    A,R2    ;soma os MSBs com Carry
        DA      A       ;ajuste decimal da soma
        MOV     @R0,A   ;guarda o MSB da soma
        RET
```

Aqui está a subrotina principal:

(R7, R6, R5) ← BCD(R4, R3)

```
BCD16:  MOV     A,R3    ;coloca em A o LSB
        ACALL   BCD8    ;converte o LSB p/ BCD e coloca
        ;               ;o resultado em AUXH e AUXL
        MOV     R5,AUXL ;o LSB em BCD vai para R5 e R6
        MOV     R6,AUXH
        MOV     A,R4    ;coloca em A o MSB
        ACALL   BCD8    ;converte o LSB p/ BCD e coloca
        ;               ;o resultado em AUXH e AUXL
        MOV     R2,AUXH ;o MSB em BCD vai para R1 e R2
        MOV     R1,AUXL
        MOV     R0,#6
        ACALL   ROT_INC ;Calcula BCD(MSB)*6
        MOV     R0,#5   ;Para que a soma seja colocada em R5 e R6
        ACALL   SOMA    ;(R6,R5) <- (R6,R5) + 6*BCD(MSB)
        CLR     A       ;zera A
        RLC     A       ;houve carry? (por seguranca)
        MOV     R7,A    ;coloca carry em R7
        ;
        MOV     R2,AUXH ;AUXH e AUXL ainda guardam o BCD(MSB)
        MOV     R1,AUXL
        MOV     R0,#5
        ACALL   ROT_INC ;Calcula BCD(MSB)*5
        ;               ; A   R2  R1   deslocar
        MOV     A,R2    ; 0X  0X  YZ   uma
        MOV     R0,#1
        XCHD    A,@R0   ; 0Z  0X  YX   posicao BCD
        SWAP    A       ; Z0  0X  YX   para a
        XCH     A,R1    ; YX  0X  Z0   esquerda
```

```
SWAP        A           ; XY  0X  Z0   (multipl. por 10)
XCH         A,R2        ; 0X  XY  Z0
MOV         R0,#5       ;Para que a soma seja colocada em R5 e R6
ACALL       SOMA        ;(R6,R5) <- (R6,R5)+ 6*BCD(MSB)+50*BCD(MSB)
MOV         A,R7
ADDC        A,#0        ;acrescenta o carry a R7
DA          A
MOV         R7,A
;
MOV         R2,AUXH     ;AUXH e AUXL ainda guardam o BCD(MSB)
MOV         R1,AUXL
MOV         R0,#2
ACALL       ROT_INC     ;Calcula BCD(MSB)*2
MOV         R0,#6       ;Para que a soma seja colocada em R6 e R7
ACALL       SOMA
RET
```

Note que podem ser introduzidas algumas melhorias nas subrotinas de multiplicação: a multiplicação por 2 se resume a uma soma.

$2n = n + n$

$5n = 2n + 2n + n$

$6n = 5n + n$

Deve-se notar que existem muitas outras rotinas para converter BIN para BCD.

EXEMPLO 11. SUBROTINA NIB_ASC:

Converter uma nibble que está no Acc para seu correspondente ASCII

Exemplos: 0110 (6) 36H

1010 (A) 41H

A solução é simples:

Se n < 10 → somar 30H

Se n >= 10 → somar 37H

```
NIB_ASC:    PUSH        Acc         ;coloca o conteudo de A na pilha
            CLR         C           ;zera Carry
            SUBB        A,#10       ;A-10 -> se CY=1 => A<10
            POP         Acc         ;       -> se CY=0 => A>=10
            JNC         NIB_ASC1    ;
```

```
            ADD     A,#30H      ;de 30 a 39 (algarismos)
            RET
NIB_ASC1:   ADD     A,#37H      ;de 41 a 46 (letras)
            RET
```

EXEMPLO 12 SUBROTINA HEX_ASC:

Converter o byte que está em Acc em seus 2 correspondentes ASCII, guardando o mais significativo em Acc e o menos significativo em b.

Exemplo: Acc = 0110 1010 Acc = 36H e B = 41H

```
HEX_ASC:    MOV     B,A         ;coloca em B uma copia de A
            ANL     A,#0FH      ;zera nibble mais sigf. de A
            ACALL   NIB_ASC
            XCH     A,B         ;troca os valores de A e B
            ANL     A,#0F0H     ;zera a nibble menos sigf. de A
            SWAP    A           ;troca as nibbles de A
            ACALL   NIB_ASC
            RET
```

EXEMPLO 13. SUBROTINA ASC_HEX:

Converter os 2 caracteres ASCII que se encontram em Acc (mais significativo) e B (menos significativo) em seu binário correspondente

Exemplo: Acc = 43H e B = 38H => Acc =1100 1000

C 8

```
ASC_HEX:    ACALL   ASCNIB
            SWAP    A           ;troca as nibbles em A
            XCH     A,B         ;troca valores entre A e B
            ACALL   ASCNIB
            ADD     A,B         ;coloca as 2 nibbles em A
            RET
```

Subrotina auxiliar, recebe um caracter ASCII no Acc e retorna sua nibble correspondente.

```
ASCNIB:     CLR     C           ;zera Carry
            SUBB    A,#30H      ;A-30H -> se CY=1 => A<30 (ERRO)
            JC      ERRO        ;       -> se CY=0 => A>=30
            PUSH    Acc         ;guarda A-30H na pilha
            SUBB    A,#10       ;A-10 -> se CY=1 => A<10
```

```
            JNC     ASCNIB1     ;    -> se CY=0 => A>=10
            POP     Acc         ;restaura o valor A-30H
            RET
            ;
ASCNIB1:    SUBB    A,#6        ;A-6 -> se CY=1 => A<6
            POP     Acc         ;       -> se CY=0 => A>=6
            JNC     ERRO
            SUBB    A,#6        ;Notar que CY=1 (ou seja, A-37H)
            RET
            ;
ERRO:       CPL     P1.7        ;indica erro
            SJMP    ERRO        ;fica em loop
```

EXEMPLO 14. UMA SUGESTÃO:

Uma solução para divisões quando o dividendo é fixo e tem mais de um byte: seja o caso em que é necessário dividir um número N por 836; não se pode fazer diretamente N/836 mas 836 =(aprox.) 65536/78 e então se usa o seguinte truque: (N*78)/65536, ou seja, multiplica-se por 78 e se deixam de fora os dois últimos bytes.

No caso de conversão de binário para BCD de um número de 0 até 9999 é necessário dividi-lo por 1000. Esta divisão pode ser feita através de uma multiplicação seguida de uma divisão por um número que seja uma potencia de dois. Deve-se lembrar que dividir um número por 2 corresponde a 1 shift right, por 4 corresponde a 2 shift right, por 8 corresponde a 3 shift right, ...

A divisão por 1000 pode ser feita usando vários valores de N e n; a tabela a seguir ilustra o erro cometido.

$$1000 = \frac{N}{n}$$

N	n	erro %	delta %	bits fora
1024	1	97,656	-2,344	10
2048	2	97,656	-2,344	11
4096	3	97,656	-2,344	12
8192	4	97,656	-2,344	13
16384	16	97,656	-2,344	14
32768	33	100,708	+0,708	15
65536	66	100,708	+0,708	16
131072	131	99,945	-0,055	17
262144	262	99,945	-0,055	18
524288	524	99,945	-0,055	19
1048576	1049	100,040	+0,044	20
2097152	2097	99,993	-0,007	21
4194304	4194	99,993	-0,007	22

Um bom exercício é desenvolver uma rotina para converter de binário para BCD usando este truque para a divisão por 1000. Verificar também a precisão (deve-se usar valores de N e n que resultem em delta % < 0 e, para obter o resto, multiplica-se o resultado por 1000 e subtrai-se do original).