8051

António Abreu
Departamento de Engenharia Electrotécnica
Escola Superior de Tecnologia de Setúbal
Campus do IPS, 2910 761 Setúbal
abreu@est.ips.pt

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Numa perspectiva simplista, a máquina computacional, ou seja, aquela que é capaz de executar instruções, é constituída por três blocos principais: i) a unidade central de processamento (CPU), onde são executadas as instruções; ii) um sistema de memória, para armazenar programa e dados; e iii) I/O (*Input/Output*) ou E/S (Entrada/Saída), que permite à CPU comunicar com dispositivos com dispositivos exteriores através de sinais.

A própria CPU também segue uma arquitectura semelhante, sendo constituída por: i) uma unidade funcional (unidade lógica aritmética–ALU) e um bloco interpretador de instruções; ii) registos, onde se armazena informação de significado diverso: contador de programa (*Program Counter)*, acumulador, apontador de pilha (*Stack Pointer)*, registos auxiliares, etc.); e iii pinos, por onde flui a informação necessária à execução dos programas (E/S).

A presença dos três blocos é, por vezes, insuficiente para emular outras máquinas de uma forma eficiente, pelo que é costume acrescentar outros dispositivos que implementam funções especializadas. Exemplos são a contagem de tempo ou de acontecimentos, através de contadores, e a comunicação entre máquinas, através de UART (Universal Asynchronous Receiver Transmitter).

Com o desenvolvimento das tecnologias de desenho e fabrico de circuitos integrados, que têm provocado um melhor e maior acomodamento das funções no silício, tem-se observado o aparecimento de circuitos integrados reunindo um número crescente de funções ou funções mais especializadas. A reunião dos três blocos previamente mencionados num só circuito integrado não escapou a esta tendência. Chega-se, finalmente, ao conceito de microcontrolador.

O termo microcontrolador refere-se, então, a circuitos integrados digitais

com capacidade de executar instruções por intermédio de uma CPU, e onde estão implementadas internamente funções que, no caso dos microprocessadores, não existem à partida. Tal é o caso da comunicação série, contagem ou temporização, conversão analógica para digital, elevado número de pinos de E/S, *watchdog*, controlo do *duty-cycle* de um onda quadrada, etc.

Por este motivo, o microcontrolador é um dispositivo electrónico que permite implementar sistemas digitais segundo a filosofia *single-chip*, significando que a quantidade de circuitos integrados, para além do próprio microcontrolador, é bastante reduzida quando comparada com o sistemas baseados em microprocessador. É também comum encontrar a designação *single-chip-computer* para fazer referência a microcontroladores, porque, com efeito, este tipo de dispositivo reúne internamente alguns dos dispositivos digitais pertencentes à organização de um computador, como sejam os previamente mencionados (E/S, memória, comunicação série com o exterior, etc.).

Para além desta centralização de capacidades, um microcontrolador apresenta certos melhoramentos ou incrementos relativamente à arquitectura de um microprocessador com o mesmo número de *bits*. Nomeadamente, existência da memória de programa no próprio *chip*, memória para dados relativamente extensa, capacidade de operar directamente sobre bits, instruções para multiplicação e/ou divisão, etc.

A união de todas as características anteriores facilita, com efeito, a implementação de sistemas digitais controladores de sistemas reais, como sejam, por exemplo, telecomandos de televisões, abertura automática de portas, temporização num forno de micro-ondas, alarmes de tipo variado, etc. A facilidade ou adequação do uso deste dispositivo para certos tipos de aplicações terá tido certamente influência no nome que lhe foi associado - micro + controlador.

Quer do ponto de vista aplicativo, quer do ponto de vista comercial, os microcontroladores são dispositivos muito importantes. Este facto é confirmado pela quantidade de fabricantes que produzem uma variedade de dispositivos muito grande, e pela larga escala da sua aplicação a diversos ramos da economia: a indústria de um modo geral e a electrónica de consumo em particular. Este aspecto é muito importante no que respeita o projecto electrónico de sistemas digitais ou híbridos (analógicos e digitais). Com efeito, é fundamental efectuar um estudo relativamente vasto no que respeita às diversas ofertas dos vários fabricantes, relativamente às características globais e particulares de cada microcontrolador, uma vez que existem boas garantias de

se encontrar o microcontrolador certo para as necessidades de cada projecto.

Complementarmente ao valor do próprio microcontrolador, deve ter-se em atenção a qualidade das ferramentas de desenvolvimento, uma vez que as ofertas do mercado são muito variadas, seja pelo preço, facilidade de aprendizagem, universalidade, flexibilidade, etc. O aspecto do desenvolvimento da aplicação, do ponto de vista do custo nas mais variadas vertentes, apresenta uma importância fundamental, devendo, em grande parte dos casos, ser analizado criteriosamente. Deve, então, considerar-se a qualidade das ferramentas de desenvolvimento e a dinâmica do fabricante do microcontrolador.

No sentido do parágrafo anterior, este documento não deve ser entendido como uma tentativa de declaração da superioridade dos microcontroladores da Intel relativamente a outros. Antes, porém, trata-se de um texto técnico dedicado aos aspectos pedagógicos do projecto digital baseado num microcontrolador concreto. Naturalmente, a importância da Intel no mercado dos dispositivos microcontroladores é conhecida e apreciada. O sucesso que se pode retirar deste livro terá mais origem nas capacidades do leitor e menos nos atributos do microcontrolador abordado.

O microcontrolador 8051 foi desenvolvido pela Intel em 1980. A partir desse momento, a sua popularidade cresceu de uma maneira sustentada. Este efeito surge devido à existência de um número relativamente elevado de fabricantes que produzem versões parcial ou totalmente compatíveis. Este aspecto é vantajoso, principalmente devido à diversidade de características possíveis de encontrar e devido às garantias de disponibilidade comercial. O aspecto da multi-origem do 8051 é ainda importante por outras razões, destacando-se a da compatibilidade. Com efeito, qualquer que seja o microcontrolador em causa, a maior parte dos dispositivos existentes actualmente apresentam, de um modo geral, arquitecturas mais ou menos complexas, o que leva a que o seu estudo possa ser demorado, podendo implicar custos de formação e atrasos no desenvolvimento. Neste sentido, o comprometimento com o 8051 é vantajoso quando comparado com outros microcontroladores, dado existir em múltiplas formas: com mais ou menos memória interna, com PWM (*Pulse Width Modulation)*, com ADC (*Analog to Digital Converter)* de um ou mais canais, mais rápido (40MHz), etc.; todas parcialmente compatíveis entre si, seja ao nível do *assembly* ou ao nível dos pinos. Adicionalmente, existem versões com interpretadores de BASIC ou FORTH no próprio *chip*, que não degradam significativamente o desempenho final, e que proporcionam um desenvolvimento relativamente rápido. Existem ainda diversas companhias que fornecem compiladores de C e de PASCAL para o *assembly* do 8051, tal

como sistemas de desenvolvimento com diversas ferramentas integradas (emuladores, *assemblers*, *linkers*, execução passo a passo sobre o *hardware*). Para terminar, relativamente a preços, estes variam razoavelmente em função dos recursos presentes no microcontrolador e das quantidades compradas. Para aqueles que apenas compram alguns microcontroladores por ano, os preços variam desde os 5 a cerca de 20 euros. Todas as características anteriores justificam a venda de cerca de 126 milhões de unidades no ano de 1993.

1.1 O Documento

O presente documento está organizado em duas partes fundamentais:

1. A primeira parte é dedicada à apresentação das características dos microcontroladores MCS 51 da Intel. Trata-se de uma parte essencialmente descritiva que deve ser vista como objecto de estudo, de forma a que os diversos conceitos sejam apreendidos na sua totalidade. Cada aspecto da arquitectura do microcontrolador é analizado separadamente por capítulos. Sempre que possível, são fornecidos exemplos de aplicação para que a compreensão seja mais facilitada. Grande parte da informação presente foi retirada de *data books*, pelo que o objectivo principal foi o de proporcionar uma leitura mais descontraída, comparativamente com a que se faz daqueles, que é mais atenta e concentrada. Naturalmente, este livro não os pretende substituir. Na verdade, eles deverão ser utilizados por diversas razões: complemento/suplemento de informação, verdadeira garantia das características dos microcontroladores e meio privilegiado de actualização. Sempre que apropriado, os exemplos de aplicação são apresentados nas linguagens assembly e C.

2. A segunda parte é dedicada a exemplos de aplicação que poderão ser implementados pelo leitor com successo, desde que sejam usados os componentes mencionados nos respectivos esquemas e sejam seguidas as boas regras da execução do projecto. Assim, esta parte deve ser utilizada quer no sentido de cimentar os conhecimentos adquiridos na primeira parte, quer no sentido simples de executar as montagens apresentadas.

1.2 Aspectos de Notação

Relativamente à notação utilizada, assume-se o seguinte:

- Utiliza-se a letra *B* como designador de byte.
- Utiliza-se a letra *b* como designador de bit.
- LSB designa *Least Significant Byte* (byte menos significativo), MSB significa *Most Significant Byte* (byte mais significativo), e LSb e MSb designam o mesmo mas para bits, respectivamente.
- Utiliza-se a notação C/$\overline{\text{T}}$ para designar os dispositivos contadores e temporizadores.
- A designação *bps* representa bits por segundo.
- A notação E/S designa Entrada/Saída, e I/O representa *Input/Output*.
- As letras H, D ou B referem-se às bases hexadecimal, decimal e binária, respectivamente. A letra B pode então surgir em duas situações diferentes: em referência a byte ou em referência à base binária. O contexto da frase onde este tipo de situação acontece permitirá remover a ambiguidade. Sempre que a letra B (no contexto de base binária) se confundir com o dígito equivalente da base hexadecimal, opta-se por utilizar-se o subscrito 2, por exemplo, 0011_2 em vez de 0011B.

Na maior parte das vezes é feita referência ao membro original da família MCS-51, através da designação 8051, quando, efectivamente, tal designação comercial não existe. Trata-se de um aspecto relativo ao conforto da leitura que pretende evitar as referências aos componentes segundo as suas designações comerciais, que são difíceis de ler. Por outro lado, qualquer membro da família MCS-51 herda as características do membro original. Assim, a aprendizagem das suas características constitui um meio obrigatório para conhecimento posterior de outros derivados; sejam da Intel ou de outros fabricantes de compatíveis.

Capítulo 2

Visão Genérica

2.1 Sistema de Memória

Relativamente ao acesso a programa e a dados, esta família de microcontroladores segue a arquitectura HARVARD, significando que dados e programa estão armazenados em memórias fisicamente separadas. O barramento de endereços é comum aos dois tipos de memória e é de 16 bits, levando a que se possam endereçar 2^{16}=64KB de programa e 64KB de dados no máximo (*vide* figuras 2.1 e 2.3). Este tipo de arquitectura apresenta algumas vantagens. Por exemplo, a memória de programa pode ser só de leitura, estando assegurada a protecção do programa contra escritas acidentais. Por outro lado, podem ser desencadeadas operações de escrita e de leitura sobre a memória de dados a partir de endereços de oito bits (apesar do barramento ser de 16 bits), o que constitui uma vantagem do ponto de vista da eficiência, dado que operações a oito bits são mais eficientes em processadores de oito bits.

Para endereçar duas memórias diferentes são necessários dois registos apontadores: o *Program Counter* (PC), onde é armazenado o endereço da próxima instrução a ser executada, e o *Data Pointer* (DPTR), onde reside o endereço do byte de dados a ser lido ou escrito. As memórias de programas e dados podem estar no interior do microcontrolador, mas também podem estar no exterior, ainda que mais raramente.

No caso em que ambas as memórias estão no exterior, elas estão acoplados ao microcontrolador através dos mesmos *buses*, i.e., existe partilha. O controlo de acesso é determinado por pinos diferentes. No caso da memória de dados, os pinos $\overline{WR}$ e $\overline{RD}$ determinam as operações de escrita e de leitura,

respectivamente. No caso da memória de programa, o pino $\overline{PSEN}$ determina o controlo de leitura. Com o sentido de se reduzir o número de pinos do circuito integrado, o barramento de dados é partilhado com os oito bits menos significativos do barramento de endereços. Isto significa que o barramento apresenta dois comportamentos diferentes em função do tempo: numa dada altura dedica-se ao transporte da parte menos significativa do endereço do dado a ler, ao passo que no intervalo de tempo seguinte serve de meio de comunicação para o próprio dado.

No caso em que ambas as memórias residem no interior do microcontrolador, deixa de ser necessário saber como é que o seu acesso está organizado. As soluções single-chip usam preferencialmente memórias internas ao microcontrolador, pelo que estas versões são tipicamente as preferidas pelos técnicos.

A memória interna de dados é composta por 256B à qual estão atribuídas várias funções. Esta memória está dividida em duas partes iguais. O espaço ocupado pelos 128B superiores, denominado espaço SFR - *Special Function Registers*, é constituído por registos de funções especiais, dos quais se destacam o acumulador, o PC, o DPTR, e aqueles usados para configuração dos diversos dispositivos existentes no microcontrolador. É possível aceder directamente aos bits de alguns dos registos SFR (aqueles cujo endereço termina em 0 ou 8). Os 128B mais baixos estão totalmente dedicados ao armazenamento de dados do utilizador. Da sua constituição destacam-se três partes fundamentais: quatro bancos de oito registos (32B), memória endereçável por bit (16B=128b) e memória RAM convencional (80B), vulgarmente conhecida por *scratch pad*. Os bancos de registos participam em funções importantes, como sejam as aritméticas, lógicas e de endereçamento. É sobre a memória endereçável por bit que opera o denominado processador *booleano*, que permite executar instruções sobre bits; novamente aritméticas, lógicas, movimentação e saltos condicionais. Note-se que a memória interna de dados (256B) é independente da memória externa de dados (64KB). As figuras 2.3 e 2.4 ilustram a organização referida.

Relativamente à memória de programa, a versão 8051 (o membro original da família) possui 4KB de memória interna. Esta faz parte do espaço total (64KB) de endereçamento. É por intermédio do pino $\overline{EA}$ - *External Address*, que se determina se os primeiros 4KB de programa são procurados na memória interna ou na memória externa. Caso o programa ocupe um espaço inferior a 4KB, então não há necessidade de recorrer à memória externa, uma vez que o programa será integralmente armazenado no próprio microcontro-

lador. Nas figuras 2.1 e 2.2 mostram-se os dois esquemas alternativos de organização deste tipo de memória. Actualmente, o mercado disponibiliza versões com vários KB (64, 128, etc.).

Relativamente à memória de programa, a Intel adoptou a seguinte terminologia:

1. A designação original refere-se a dispositivos cuja memória de programa é do tipo ROM. Por exemplo, a versão 80C51BH é composta por uma memória interna de programa de 4KB do tipo ROM. Note-se que, devido ao tipo de memória, estas versões são tipicamente utilizadas após o fim da fase de teste, i.e., quando o projecto está completamente estável, podendo passar-se às fases de montagem e comercialização do produto. Naturalmente, estas são as versões mais baratas. Estas versões têm desaparecido com o passar do tempo.

2. Caso a memória seja do tipo EPROM, o prefixo da designação do componente passa a ser 87, em vez do habitual 80. Por exemplo, o 87C51 é composto por uma memória interna de programa de 4KB do tipo EPROM. As versões com este tipo de memória são tipicamente utilizadas no desenvolvimento de protótipos. Ultimamente têm-se imposto as versões constituídas por memórias do tipo EEPROM ou FLASH, de tal modo que é praticamente impossível adquirir versões com EPROM.

3. Finalmente, quando não existe memória de programa interna, a parte sufixa da designação deixa de ser 51 para passar a ser 31. Por exemplo, a versão 80C31BH não tem memória de programa interna. Neste caso, o programa reside numa memória externa que está acoplada ao microcontrolador através do barramento de endereços e do barramento de dados.

O interesse das três versões mencionadas é praticamente histórico. Com efeito, dada a evolução do mercado da electrónica, que significa produzir melhor com menor preçço, actualmente os microcontroladores com memória de programa com FLASH é a norma utilizada por todos os fabricantes.

2.2 Portos de Entrada/Saída - E/S

Uma característica importante dos microcontroladores diz respeito às suas capacidades de ligação a outros *chips*. No caso do 8051 esta capacidade é

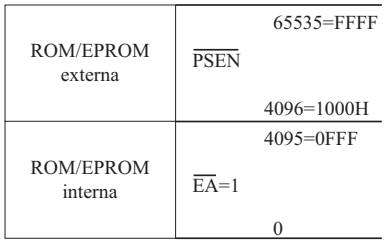

Figura 2.1: Memória de programa dividida por uma parte interna e por uma externa, prefazendo os 64KB possíveis de endereçar.

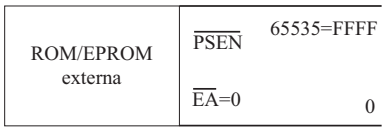

Figura 2.2: Memória de programa externa num total de 64KB.

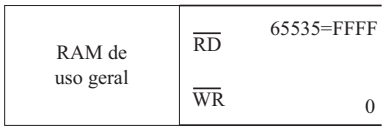

Figura 2.3: Memória de dados exterior, totalizando 64KB.

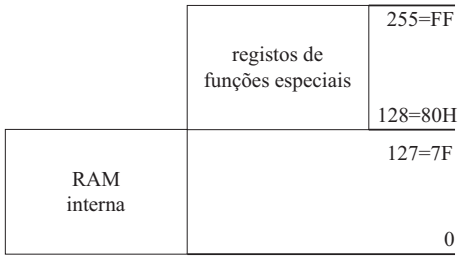

Figura 2.4: Configuração da memória de dados interna, num total de 256B.

levada a cabo por 32 linhas de E/S, ou, por outras palavras, por quatro portos de E/S de oito bits cada. Cada linha pode ser configurada como entrada ou saída. Note-se que se a memória de programa for externa, então 16 linhas de E/S deixam de o ser, passando a constituir os barramentos de endereços (AB0...AB15) e de dados (DB0...DB7). Se, adicionalmente, a memória de dados também for externa, então, para além daqueles, outros dois pinos deixam de ser de E/S para passarem a desempenhar o papel do controlo de acesso ($\overline{WR}$ e $\overline{RD}$). Verifica-se assim que certos pinos do microcontrolador têm duas funções, das quais só uma pode ser utilizada em dado momento. As capacidades de ligação de um processador ao seu exterior é fundamental, pelo que a quantidade de linhas de E/S é geralmente vista como um factor de escolha. Existem várias possibilidades para se ter acesso a linhas suplementares de E/S. Estes esquemas serão abordados posteriormente.

2.3 Contagem e Temporização

Outro aspecto de importância fundamental no projecto de sistemas digitais baseados em microcontrolador diz respeito aos mecanismos de realização de contagem e/ou de temporização. Ambos podem ser implementados por *software* no microcontrolador, mas, se assim for, poderão existir condicionalismos importantes, como sejam o uso de interrupções e o maior grau de complexidade de escrita do programa. Idealmente, estes processos deverão ser desempenhados por *hardware* dedicado. No caso dos microcontroladores, é típico verificar-se a existência no próprio *chip* de dispositivos que desempenham esta função. No caso específico do 8051 existem dois contadores/temporizadores (designados por C/$\overline{T}$0 e C/$\overline{T}$1) de 16 bits cada. A sua configuração é feita por intermédio de registos existentes no espaço SFR, devendo determinar-se, para uma dada aplicação, se se pretende a função de contagem ou de temporização, i.e., se o sinal de relógio que alimenta o contador é externo (função contador) ou interno (função temporizador). Adicionalmente, cada C/$\overline{T}$ dispõe de quatro modos de operação.

2.4 Canal de Comunicação Série

Todos os microcontroladores da série 51 dispõem de um canal de comunicação série do tipo assíncrono, que permite, com simplicidade, implementar o pro-

tocolo de comunicação RS-232 presente em cada computador pessoal. O sucessivo abaixamento do preço dos PC's tornou muito importante esta característica nos microcontroladores. Com efeito, a título de exemplo, é preferível desenvolver *hardware* dedicado a uma função específica e reservar o PC para as tarefas de processamento mais intensivo. Considere-se o caso da supervisão ou monitorização do estado de um dado sistema real, que, idealmente, deverá ser feito por intermédio de grafismos. Estes deverão ser executados num PC, dado ser mais fácil programar numa linguagem de alto nível e dado que o PC disponibiliza um bom meio para a interface homem-máquina (placas gráficas potentes e écrans grandes). Opostamente, este tipo de tarefa é demasiadamente difícil quando implementada em microcontroladores.

A operação com o canal de comunicação série do 8051 faz-se por intermédio de registos pertencentes ao espaço SFR, onde se determinam a taxa a que se realiza a comunicação e a constituição da sequência de bits a enviar/receber - a trama[1]. Existe ainda a possibilidade de ser gerada uma interrupção tão logo esteja terminado o processo de envio/recebimento de uma trama, criando uma espécie de semi-paralelismo entre o programa principal e a comunicação de dados com o exterior.

2.5 Resumo das Características

Relativamente ao microcontrolador 8051, que constitui o membro original da família, e que por isso contém o conjunto mínimo de características ou funcionalidades, apresenta-se, de seguida, o conjunto resumido das suas características:

- CPU de oito bits.
- Capacidade de processamento directo ao nível do bit através do processador *booleano*.
- Espaço endereçável de 64KB para memória de programa.
- Espaço endereçável de 64KB para memória de dados.
- Memória de programa no próprio *chip* de 4KB ou superior.
- Memória de dados no próprio *chip* de 128B ou superior.

[1] *frame* na denominação anglo-saxónica.

- Existência de 32 linhas de E/S individualmente endereçáveis.
- Existência de dois contadores/temporizadores de 16 bits cada.
- Existência de uma UART[2] *full duplex*[3].
- Existência de cinco fontes de interrupção organizadas em dois níveis de prioridade.
- Existência de oscilador no próprio *chip*.
- Frequência típica de operação de 12MHz ou superior.
- *Assembly* de 111 instruções, das quais 64 são executadas num só ciclo máquina.
- Multiplicação e divisão no próprio *chip*.

Todos os restantes membros da família 51 disponibilizados pela Intel são baseados nestas características, podendo ter outras suplementares e/ou as apresentadas mas mais fortalecidas, como, aliás, o resumo anterior deixa transparecer.

2.6 Descrição dos Pinos

O microcontrolador 8051 está disponível em vários tipos de empacotamento: DIL - *Dual InLine* (ou DIP - *Dual InLine Package*), QFP - *Quad Flat Pack,* e PLCC - *Plastic Leaded Chip Carier*, para citar os mais comuns. Relativamente ao tipo de empacotamento DIL (ou DIP), o circuito integrado é composto por 40 pinos, tal como é apresentado na figura 2.5.

Uma descrição resumida do significado de cada pino é apresentada de seguida, chamando-se a atenção para o facto de certos pinos terem associadas duas funções diferentes.

- P0.0, ..., P0.7 - Pinos associados ao porto zero de E/S. Alternativamente, estes pinos podem comportar-se como a parte menos significativa do barramento de endereços e como barramento de dados.

[2]Acrónimo anglosaxónico designador de *Universal Asynchronous Receiver Transmitter*.
[3]Designa a capacidade de emitir e receber dados simultaneamente.

- P1.0, ..., P1.7 - Pinos associados ao porto um de E/S.
- P2.0, ..., P2.7 - Pinos associados ao porto dois de E/S. A segunda função destes pinos corresponde à parte mais significativa do barramento de endereços.
- RST - Pino de entrada para reinicialização do microcontrolador (reset).
- RxD/P3.0 - Pino zero do porto três de E/S ou pino por onde se faz a recepção no canal série.
- TxD/P3.1 - Pino um do porto três de E/S ou pino por onde se faz a transmissão no canal série.
- $\overline{INT}0$/P3.2 - Pino dois do porto três de E/S ou pino de entrada associado à interrupção externa zero.
- $\overline{INT}1$/P3.3 - Pino três do porto três de E/S ou pino de entrada associado à interrupção externa um.
- T0/P3.4 - Pino quatro do porto três de E/S ou pino de entrada do relógio externo para o contador/temporizador zero - C/$\overline{T}0$.
- T1/P3.5 - Pino cinco do porto três de E/S ou pino de entrada do relógio externo para o contador/temporizador um - C/$\overline{T}1$.
- $\overline{WR}$/P3.6 - Pino seis do porto três de E/S ou pino de saída para controlo de escrita na memória externa de dados.
- $\overline{RD}$/P3.7 - Pino sete do porto três de E/S ou pino de saída para controlo de leitura da memória externa de dados.
- XTAL1, XTAL2 - Pinos de entrada associados à fonte de oscilação para o relógio do microcontrolador.
- Vss - Pino de massa.
- Vcc - Pino de alimentação (+5V).
- $\overline{EA}$ - Pino de entrada que determina se os primeiros 4KB de programa devem ser procurados na memória interna ($\overline{EA}$=1) ou na memória externa ($\overline{EA}$=0).

- $\overline{ALE}$ - Pino de saída para controlo do *latch* que armazena a parte menos significativa do barramento de endereços.
- $\overline{PSEN}$ - Pino de saída para controlo da acção de leitura da memória externa de programa.

Nos próximos capítulos é feita uma exposição de maior pormenor técnico acerca de cada uma das características mencionadas.

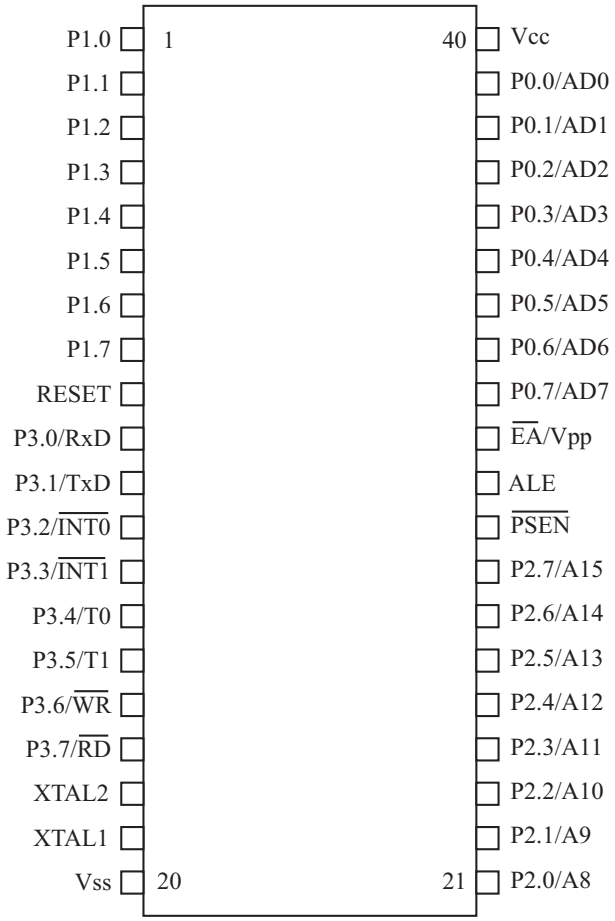

Figura 2.5: *Pinout* do 8051.

Capítulo 3

Sistema de memória

Foi visto anteriormente que o 8051 é constituído por dois espaços de memória disjuntos, segundo a arquitectura *Harvard*: memórias de programa e dados. Neste capítulo são apresentadas as características de cada tipo de memória. Adicionalmente, introduz-se lentamente o *assembly*, de modo a que o leitor possa ficar com uma visão prática dos conceitos introduzidos.

3.1 Memória de programa

Nas versões com memória de programa no interior do microcontrolador, as mais habituais presentemente, o processo de leitura de uma instrução é completamente transparente para o programador, pelo que este caso não merece, para já, considerações adicionais. O mesmo não pode ser dito relativamente à leitura de uma instrução que resida numa memória externa. Este é, com efeito, o objectivo fundamental desta secção. No entanto, dado que actualmente todos os microcontroladores têm memória de programa interna, não é importante que o leitor tente compreender, pelo menos numa fase inicial, os conhecimentos aqui transmitidos. Poderá, assim, saltar o próximo parágrafo.

O processo de leitura de um byte proveniente da memória externa (sejam os códigos das instruções ou os seus argumentos) é controlado por dois pinos: $\overline{\text{PSEN}}$ e ALE. O pino $\overline{\text{PSEN}}$ (*Program Store Enable)*, quando activo, determina o intervalo de tempo durante o qual se faz a leitura de um byte da memória de programa. O pino ALE (*Address Latch Enable)* é o responsável pela multiplexagem[1] entre a parte baixa do *bus* de endereços e o *bus* de dados,

[1] Certos estrangeirismos como este são adoptados sem hesitação.

da maneira seguinte. Durante uma leitura à memória, quer seja de dados ou de programa, o porto P0 apresenta dois comportamentos distintos. Durante a primeira fase, o byte menos significativo do endereço está disponível em P0 (nesta fase é um porto de saída) e o restante byte está disponível em P2, perfazendo os 16 bits que compõem o endereço. Durante a segunda fase, o byte correspondente é lido através de P0 (que passa a porto de entrada). Ou seja, existe partilha de P0 entre a parte baixa do endereço e o dado a ler nesse endereço. Durante o intervalo de tempo composto pelas duas fases, o endereço deve permanecer estável nos pinos da memória. Torna-se então necessário armazenar a parte baixa do endereço (durante a primeira fase) através de um dispositivo adequado, tipicamente um *latch* (*e.g.* 74XX373), para que, na segunda fase, P0 possa receber o byte endereçado. O pino ALE fornece o sinal de controlo para o armazenamento da parte baixa do endereço no *latch*. Este modo de funcionamento está ilustrado na figura 3.1. Relativamente a essa figura, observe-se que um ciclo de instrução (M1 e M2) é composto por seis estados (S1 a S6) ou 12 períodos de relógio (S1P1 a S6P2). Caso o microcontrolador seja alimentado com um cristal de 12MHz, então cada ciclo máquina demora 1 μseg.

No *assembly* do 80C51 existem três tipos de instrução relativamente ao número de bytes ocupados: instruções de um, dois e três bytes. Estes podem ainda ser divididos em: i) instruções de um byte-um ciclo máquina, dois bytes-um ciclo máquina, e ii) um byte-dois ciclos máquina, dois bytes-dois ciclos máquina e três byte-dois ciclos máquina. Na primeira divisão, o código da instrução é recebido pelo microcontrolador durante M1S1, é descodificado durante M1S2 e executado desde M1S3 até M1S6. Na segunda divisão tem-se: o código da instrução é recebido durante M1S1, é descodificado durante M1S2 e executado desde M1S3 a M2S6. Os segundo e terceiro bytes são carregados durante M1S4, M2S1 ou M2S4. Observe-se também a existência de dois bytes carregados pelo microcontrolador em cada ciclo máquina. Caso o segundo byte não seja necessário à execução da instrução (cujo código foi carregado no primeiro byte), então o microcontrolador descarrega imediatamente o segundo byte.

A versão do microcontrolador implicitamente referida na figura 3.1 pode ser do tipo 80C31, *i.e.*, microcontrolador sem memória interna de programa. Neste caso deve atribuir-se obrigatoriamente $\overline{\text{EA}}$=0 (*External Address*). Este pino de entrada permite decidir de que memória se recolherão as instruções: da memória interna ($\overline{\text{EA}}$=1) ou da memória externa ($\overline{\text{EA}}$=0). A figura 3.2 apresenta o esquema de ligações de uma memória do tipo EPROM a um

microcontrolador 80C31.

Tipicamente, escolhe-se um microcontrolador com memória interna com tamanho superior para armazenar o programa. Nesse caso, faz-se $\overline{\text{EA}} = 1$. As vantagens são várias. Ao evitar-se a memória de programa externa, poupam-se pinos de E/S, tem-se um sistema mais pequeno e também de menor consumo. Note-se que não faz muito sentido usar um programa cujo tamanho ultrapassa o tamanho da memória interna do microcontrolador. Isto significaria que teria de haver uma memória externa, onde se guardaria a parte que não coube na memória de programa interna. Seria preferível adquirir uma versão do microcontrolador sem memória interna (8031) e colocar uma memória externa, como dito anteriormente, ou, situação que é a mais correcta, adquirir uma versão com memória interna cujo tamanho seja superior ao programa.

O pino $\overline{\text{PSEN}}$ fornece o sinal de controlo sempre que o acesso ao programa seja feito externamente. O mesmo não se passa com o pino *ALE*, que emite continuamente o sinal de sincronismo. Assim, nas versões com memória de programa com memória interna (as mais habituais actualmente), $\overline{\text{PSEN}}$ mantém-se a nível 1 e *ALE* comuta continuamente. Este comportamento é aproveitado para um procedimento básico de teste de funcionamento do microcontrolador - se *ALE* não comuta entre 0 e 1, então o programa não pode estar a ser executado.

Quando o espaço ocupado pelo programa é inferior ou igual à capacidade da memória interna, os portos P0 e P2 poderão ser utilizados para E/S e o pino ALE como uma fonte de relógio de frequência $f_{osc}/6$, em que f_{osc} representa a frequência fundamental do cristal ou da fonte de relógio acoplada ao microcontrolador.

3.2 Memória de dados

Existe a possibilidade de se armazenarem dados na memória interna ou na memória externa. Dado tratar-se de memórias endereçadas separadamente, tem-se que os mecanismos de endereçamento são naturalmente diferentes. Neste sentido, a sua análise e feita em separado.

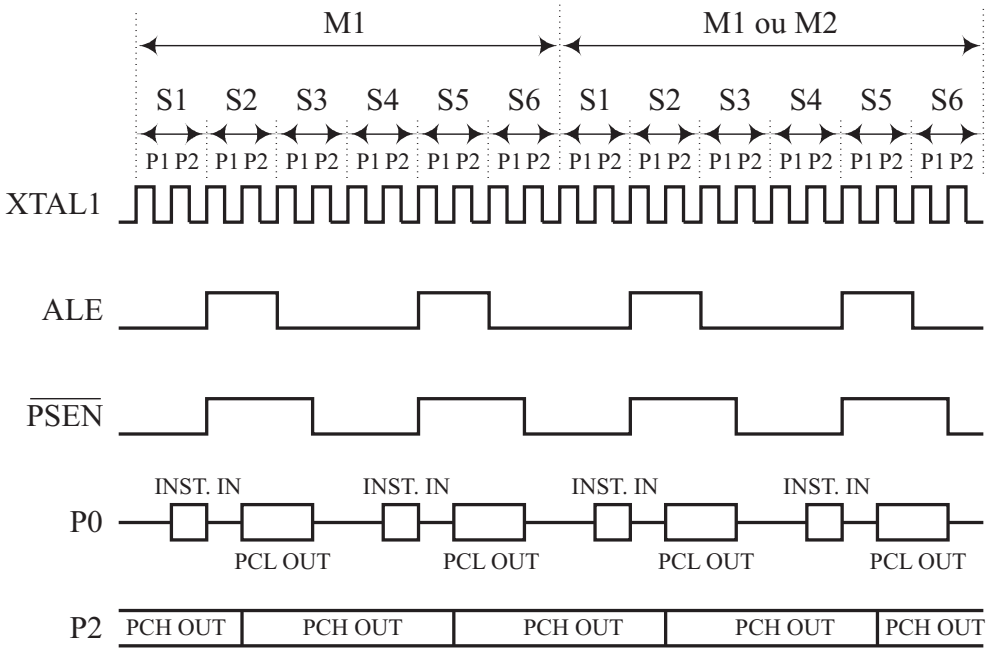

Figura 3.1: Diagrama temporal referente a um ciclo máquina.

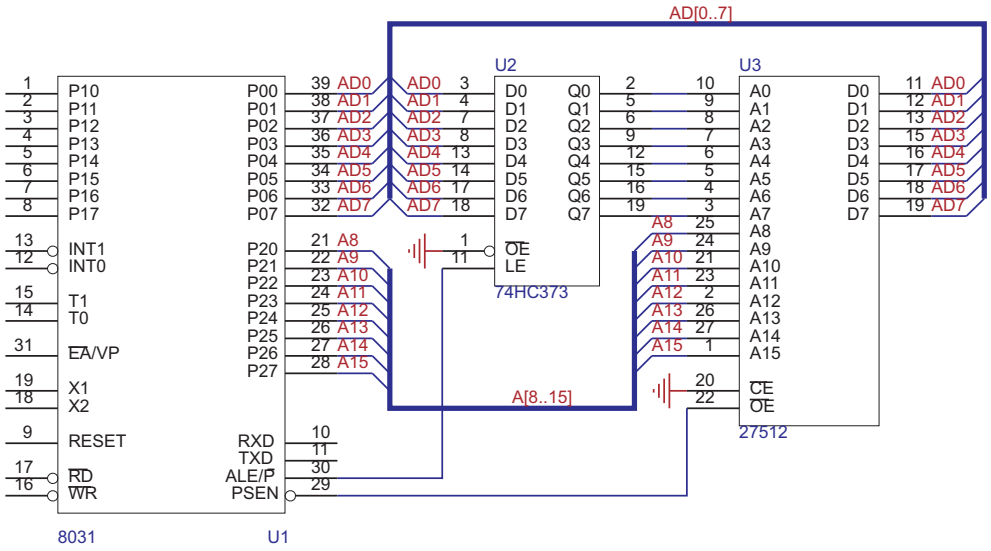

Figura 3.2: Esquemático da ligação entre a memória de programa e o microcontrolador.

3.2.1 Memória externa

A memória externa de dados é endereçada pelo registo DPTR, pertencente ao espaço SFR. Dado tratar-se de um registo de 16 *bits*, o total de memória endereçada é de 2^{16}=64KB. Esta característica é desejável para aplicações onde o volume de dados a processar seja relativamente grande.

O controlo da memória é feito através dos pinos $\overline{\text{WR}}$, $\overline{\text{RD}}$ e ALE. Neste caso, o pino $\overline{\text{PSEN}}$ está inoperante. O esquema das ligações de uma memória RAM a um microcontrolador do tipo 8031 é semelhante àquele visto para a memória de programa (*vide* figura 3.2), estando apresentado na figura 3.3. O facto de, nessa figura, a memória não ser de 64KB não deve levantar qualquer tipo de dúvida.

Por exemplo, imagine-se que é pretendido carregar o acumulador com o conteúdo do endereço 1000H da memória de dados externa. O *assembly* correspondente seria,

```
MOV DPL,\#00H ; carregar a parte baixa do DPTR
MOV DPH,\#10H ; carregar a parte alta do DPTR
MOVX A,@DPTR  ; ler o dado para o acumulador
```

ou, em alternativa,

```
MOV DPTR,\#1000H
MOVX A,@DPTR
```

Alguns registos de 16b em microcontroladores de 8b estão separados por dois registos de 8b; é esse o significado de DPL (*Data Pointer Low*) e DPH (*Data Pointer High*). A última instrução permite ler a memória na posição indicada, garantindo que o dado é armazenado no acumulador. Certos pormenores sintácticos presentes no conjunto de instruções anterior são propositadamente deixados sem mais explicações. Naturalmente, em lugar próprio, eles serão totalmente esclarecidos. Deve então, de algum modo e dentro do que for possível, funcionar a intuição do leitor.

3.2.2 Memória interna

Tal como previamente afirmado, a memória interna pode ser dividida em duas partes fundamentais: o espaço RAM, composto por 128B, e o espaço SFR, composto por 128B (*vide* figura 3.4). O espaço RAM está dedicado ao armazenamento de dados da aplicação, ao passo que no espaço SFR estão os

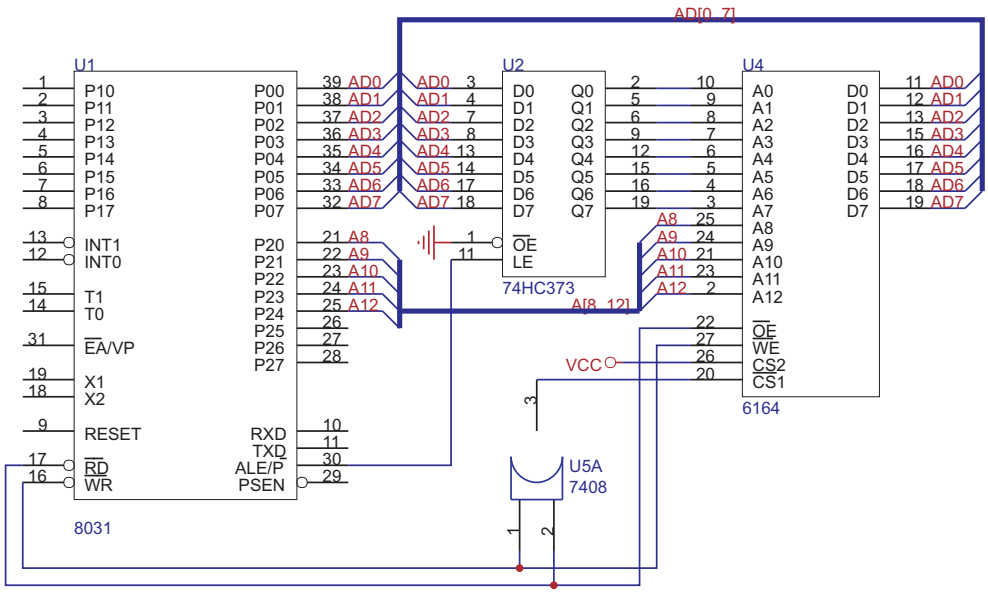

Figura 3.3: Esquema de ligações entre a memória de dados e o microcontrolador.

registos de funções especiais. A quantidade de registos pertencentes ao espaço SFR é dependente do microcontrolador, dado que versões com mais dispositivos internos requerem mais registos. As considerações tecidas nas secções subsequentes deste capítulo são referentes à versão mais simples (8051), que é composto por 21 registos de funções especiais.

Espaço RAM

Este espaço é constituído por 128 bytes, cujos endereços vão de 00H até 7FH. No próximo capítulo haverá a oportunidade de compreender os modos de endereçamento que permitem aceder a qualquer byte deste espaço. Para além da visão linear desta memória, existe uma outra que é interessante apresentar. Assim, também é possível afirmar que esses 128 bytes estão divididos em três partes principais: bancos de registos, memória endereçável por bit e memória de uso geral (de acordo com a figura 3.5). Note-se que, fisicamente, a memória é uma só, existindo duas interpretações para o seu uso.

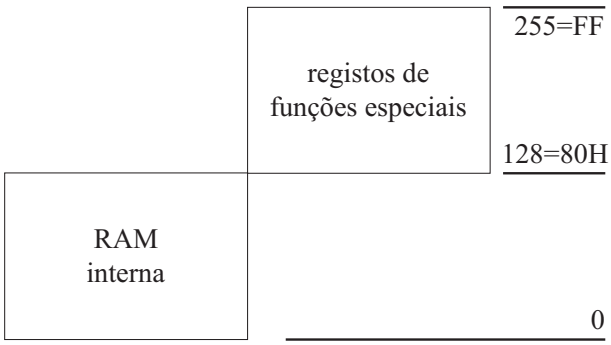

Figura 3.4: Memória de dados interna dividida nos dois blocos principais: espaço SFR e RAM.

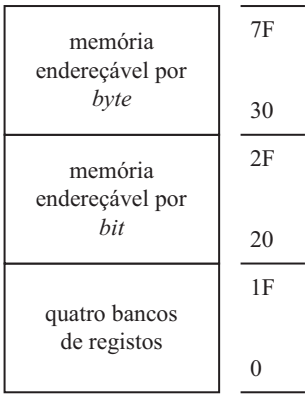

Figura 3.5: Separação do espaço RAM interno em três partes principais: bancos de registos, memória endereçável por bit e RAM génerica.

RS1	RS2	Banco	ASM	C
0	0	0	MOV PSW,#00H	PSW=0;
0	1	1	MOV PSW,#08H	PSW=x8;
1	0	2	MOV PSW,#10H	PSW=x10;
1	1	3	MOV PSW,#18H	PSW=x18;

Tabela 3.1: Como fazer a selecção do banco em função dos bits RS1 e RS2.

Ordem	7	6	5	4	3	2	1	0
Bit	CY	AC	FO	RS1	RS2	OV	-	P

Tabela 3.2: Configuração do registo PSW pertencente ao espaço SFR.

Banco de registos Existem quatro bancos de registos que ocupam os endereços 00H a 1FH, de acordo com a figura 3.6. Cada banco é composto por oito registos: R0,..., R7. Dado que os registos partilham os nomes entre bancos, então primeiro deve seleccionar-se o banco e só depois se poderá aceder a um dado registo. A maneira de seleccionar um banco é feita por intermédio de dois bits existentes no registo PSW(D0H), pertencente ao espaço SFR (*vide* tabelas 3.2 e 3.1).

Por exemplo, o assembly que permite escrever o dado 00H em R0 é,

```
MOV R0,#00H
```

Memória endereçável por bit Esta memória é composta por 16 bytes que são endereçveis por bit, *i.e.*, existem $16 \times 8 = 128$ bits endereçáveis. Na figura 3.7 são apresentados os endereços dos 128 bits em função dos endereços dos 16 bytes. Os endereços dos bits podem ser utilizados no contexto de qualquer instrução que opere sobre bits, como se verificará no próximo capítulo.

Por exemplo, para se escrever 1 no primeiro endereço da memória endereçável por bit, basta fazer SETB 00H. É importante compreender que o endereço 00H é referente a um bit e não a um byte. O interpretador de instruções no interior do microcontrolador, "sabe" que a instrução SETB opera sobre bits, pelo que o seu argumento é um endereço de um bit.

endereço	nome	RS2	RS1	banco
1F	R7	1	1	3
1E	R6			
1D	R5			
1C	R4			
1B	R3			
1A	R2			
19	R1			
18	R0			
17	R7	1	0	2
16	R6			
15	R5			
14	R4			
13	R3			
12	R2			
11	R1			
10	R0			
F	R7	0	1	1
E	R6			
D	R5			
C	R4			
B	R3			
A	R2			
9	R1			
8	R0			
7	R7	0	0	0
6	R6			
5	R5			
4	R4			
3	R3			
2	R2			
1	R1			
0	R0			

Figura 3.6: Pormenor dos bancos, evidenciando o endereço e nome de cada registo e o código dos bits de selecção.

bit								byte
7F	7E	7D	7C	7B	7A	79	78	2F
77	76	75	74	73	72	71	70	2E
6F	6E	6D	6C	6B	6A	69	68	2D
67	66	65	64	63	62	61	60	2C
5F	5E	5D	5C	5B	5A	59	58	2B
57	56	55	54	53	52	51	50	2A
4F	4E	4D	4C	4B	4A	49	48	29
47	46	45	44	43	42	41	40	28
3F	3E	3D	3C	3B	3A	39	38	27
37	36	35	34	33	32	31	30	26
2F	2E	2D	2C	2B	2A	29	28	25
27	26	25	24	23	22	21	20	24
1F	1E	1D	1C	1B	1A	19	18	23
17	16	15	14	13	12	11	10	22
0F	0E	0D	0C	0B	0A	09	08	21
07	06	05	04	03	02	01	00	20

Figura 3.7: Mapa da memória endereçável por bit, onde se apresentam os endereços de cada bit e dos bytes correspondentes.

Memória de uso geral Esta memória estende-se desde o endereço 30H até 7FH, como ilustrado na figura 3.5.

Por exemplo, para escrever na posição 40H o byte 40H, basta fazer MOV 40H,#40H, em que o primeiro 40H refere-se ao endereço e o segundo refere-se ao dado.

Espaço SFR Como anteriormente afirmado, o espaço SFR é constituído por um conjunto de registos destinados à configuração dos dispositivos existentes no interior do microcontrolador, e por certos registos fundamentais. A figura 3.8 apresenta a lista de todos os registos SFR existentes na versão 80C51. Noutras versões apetrechadas com mais dispositivos internos, outros registos existirão para além dos apresentados.

Apesar de uma coluna da figura 3.8 referir o significado de cada registo, pode não ser claro para o leitor a função real de certos registos. Em espaço próprio, os dispositivos internos são analisados em pormenor, ficando claro o significado de cada registo.

Um nota adicional para referir que todos os registos cujo endereço termine em 8 ou 0 são acessíveis por bit. Por exemplo, o registo associado ao porto P0 tem endereço 80H. Porque este endereço termina em 0, então pode-se aceder ao primeiro bit de P0, através de P0.0, ao segundo bit de P0, através de P0.1, etc. Esta facilidade confere uma utilização mais confortável ao programador.

Outros exemplos de *assembly*. Para "limpar" o acumulador, poder-se-á fazer MOV A,#00H. Para colocar o bit mais significativo do porto P1 a 1, faz-se SETB P1.7. Para complementar o bit menos significativo de P3, faz-se CPL P3.0

Nome	Significado	Endereço
A	acumulador	E0
B	registo B	F0
PSW	*Program Status Word*	D0
SP	*Stack Pointer*	81
DPL	parte baixa do DPTR	82
DPH	parte alta do DPTR	83
P0	porto zero	80
P1	porto um	90
P2	porto dois	A0
P3	porto três	B0
IP	configuração das prioridades das interrupções	B8
IE	configuração da activação das interrupções	A8
TMOD	controlo do modo dos C/$\overline{\text{T}}$	89
TCON	controlo dos C/$\overline{\text{T}}$	88
TH0	parte alta do C/$\overline{\text{T}}$0	8C
TL0	parte baixa do C/$\overline{\text{T}}$0	8A
TH1	parte alta do C/$\overline{\text{T}}$1	8D
TL1	parte baixa do C/$\overline{\text{T}}$1	8B
SCON	configuração da porta série	98
SBUF	*buffer* da porta série	99
PCON	configuração dos modos de adormecimento	87

Figura 3.8: Lista dos registos de funções especiais existentes na versão 80C51.

Capítulo 4

Conjunto de instruções

Todos os membros da família MCS 51 executam o mesmo conjunto de instruções. Como foi anteriormente afirmado, o aspecto da compatibilidade do *assembly,* entre os microcontroladores produzidos pela Intel e os compatíveis produzidos por outros fabricantes, é vantajoso do ponto de vista do projecto, dado que, por si só, permite uma redução nos custos de desenvolvimento. Em termos práticos, as diferenças que alguns microcontroladores mais evoluídos introduzem ao nível da programação, situam-se ao nível dos registos SFR que estão associados aos recursos que disponibilizam. Ou seja, o conjunto de instruções mantém-se inalterado independentemente do microcontrolador.

O conjunto de instruções executadas pelo microcontrolador 8051 é relativamente extenso, sendo composto por 111 instruções, das quais 64% são executadas num ciclo máquina (a que correspondem 12 períodos de relógio) e, das restantes, a maior parte é executada em dois ciclos máquina. O microcontrolador original aceita uma frequência máxima de relógio de 12MHZ, levando a que possa processar um milhão de instruções por segundo (1 MIP).

Em termos gerais, este microcontrolador apresenta uma arquitectura interna baseada em acumulador com pequenas aproximações à arquitectura baseada em registo. Isto significa que grande parte das instruções dizem respeito ao acumulador. Por exemplo, o resultado final de uma dada operação fica armazenado no acumulador, ou um determinado teste é feito relativamente ao conteúdo do acumulador. Existem, contudo, instruções do mesmo tipo mas que são referentes a registos particulares: por exemplo R0 e R1 de qualquer banco de registos, pelo que, em certa medida, a arquitectura da família MCS 51 é algo híbrida. Adicionalmente, existem conjuntos de instruções do tipo lógico, de atribuição ou relativas a saltos condicionais,

que processam *bits* directamente. Esta capacidade do *assembly* é importante dado facilitar o desenvolvimento de programas. Finalmente, existem seis modos de endereçamento cuja variedade é geralmente do agrado dos programadores.

Neste capítulo são apresentadas as instruções do *assembly* do MCS 51, relativamente à sua sintaxe, semântica e modos de endereçamento.

4.1 Modos de Endereçamento

Os modos de endereçamento referem-se às maneiras de determinar qual ou quais os operandos a que uma dada instrução é aplicada. Tipicamente os modos mais comuns são: i) **directo**, onde se refere explicitamente o endereço do operando, ii) **indirecto**, onde se refere indirectamente o endereço do operando e iii) **imediato**, onde se refere na própria instrução o valor do operando. Estes três modos básicos estão presentes no *assembly* do 8051 e ainda algumas derivações, como de seguida se apresenta.

4.1.1 Constante Imediata

O valor do operando aparece explicitamente depois do código da instrução. Por exemplo, a instrução `MOV A,\#25H` permite carregar directamente no acumulador a constante (imediata) 25H. Este modo é determinado pelo uso do símbolo #.

4.1.2 Endereçamento Directo

O endereço do dado a ser processado é referido após o código da operação. Por exemplo, a instrução `MOV A,25H` permite carregar no acumulador o conteúdo do endereço 25H da memória de dados interna. Note-se que, neste tipo de endereçamento, só é permitido o acesso à memória de dados interna (de 00H a FFH), dado que o endereço mencionado é sempre de oito *bits*.

4.1.3 Endereçamento Indirecto

Neste tipo de endereçamento, o conteúdo de um dado registo determina o endereço do valor a ser processado, *i.e.*, o registo funciona como um pon-

teiro ou apontador. A memória apontada pode ser a memória interna ou a memória externa.

No caso da memória interna, somente os registos R0 ou R1, de qualquer banco de registos, podem ser utilizados para referir endereços da metade inferior da memória (de 00H a 7FH). Por exemplo, a execução das instruções `MOV R1,#24H` e `MOV @R1,25H` permite copiar para o endereço 24H o conteúdo do endereço 25H. O endereçamento indirecto está patente no símbolo @.

No caso da memória externa, é possível fazer endereçamento utilizando os registos de oito *bits* $R0$ e $R1$ e o registo de 16 *bits* $DPTR$. No caso dos registos de oito *bits*, as instruções `MOV R0,#24H` e `MOVX @R0,A`, por exemplo, permitem copiar para o endereço 24H (da memória externa, devido ao X em MOVX) o conteúdo do acumulador. No caso do $DPTR$, as instruções correspondentes seriam `MOV DPTR,#0024H` e `MOVX @DPTR,A`.

A diferença entre os dois tipos de acesso à memória de dados externa está no porto P2. No primeiro caso, o endereço de oito *bits* é colocado somente em $P0$, significando que $P2$ não participa. Este porto poderá então ser utilizado para E/S. No segundo caso, o endereço de 16 *bits* é dividido por $P0$ e $P2$. A quantidade de memória endereçável também é diferente. Caso o acesso seja a 8 *bits* ou a 16 *bits*: 2^8=256B, para o primeiro caso, e 2^{16}=65536B, para o segundo.

Existe ainda uma outra particularidade devida ao número de *bits* envolvido no acesso. Imagine-se que uma dada aplicação é executada numa versão 87CXX (a memória de programa é interna). Caso a memória de dados interna seja insuficiente, poder-se-á recorrer às instruções de movimentação de dados com acesso a 8 *bits*, significando que o porto $P0$, funcionando previamente como E/S, é trocado por 256B de memória de dados. É uma solução de compromisso. Caso o acesso tenha de ser a 16 *bits*, por necessidade, P0 e P2 passam a estar dedicados ao barramento de endereços e dados. A partir deste momento deixa de haver vantagem na utilização do microcontrolador com memória de programa interna. Naturalmente, o problema não se coloca caso o sistema seja baseado na versão 80C3X (a memória de programa é externa), uma vez que $P0$ e $P2$ funcionam como barramentos, pelo que é perfeitamente igual fazer-se o acesso a 8 *bits* ou a 16 *bits*.

4.1.4 Endereçamento Indexado

Este tipo de endereçamento permite ler valores de tabelas (armazenadas na memória de programa). Dado não ser possível escrever nesta memória, então

as tabelas só podem ser lidas. Neste tipo de endereçamento faz-se uso dos registos PC ou $DPTR$ e A. Os registos PC ou $DPTR$ devem conter o endereço do primeiro elemento da tabela (endereço base), ao passo que em A é colocado o valor da linha a ser lida, *i.e.*, o registo A funciona como um índice. Por exemplo, imagine-se uma tabela definida da seguinte maneira:

Endereço	**Conteúdo**
0FFFH:	×
1000H:	10H
1001H:	11H
1002H:	12H
1003H:	13H
1004H:	×

em que × representa um valor não importante no contexto criado. O conjunto de instruções que permitem ler a terceira linha da tabela é:

```
MOV DPTR,#1000H   ; endereço da base
MOV A,#02H        ; atribuição do índice
MOVC A,@A+DPTR    ; leitura do valor
```

O exemplo anterior faz uso do registo $DPTR$. De seguida apresenta-se um exemplo da leitura de uma tabela fazendo uso do registo PC.

```
TABELA: INC A
        MOVC A,@A+PC
        RET
        DB 10H          ; início dos dados
        DB 20H
        DB 30H
        DB 40H
        DB 50H
```

Note-se que a soma entre o conteúdo de A e o conteúdo do PC, executada pela instrução MOVC, é feita com base no valor actual do PC. Este contém o endereço seguinte à instrução MOVC, ou seja, a instrução RET. Assim, para ultrapassar a instrução RET, há que somar o número de *bytes* por ela ocupados ao resultado de $A + PC$, que no caso presente é 1B. Esta é a razão que justifica o aparecimento da instrução `INC A` no início da subrotina de leitura da tabela.

Relativamente à instrução DB, compete afirmar que não se trata de uma instrução do *assembly*, mas de uma directiva para o *assembler*. Este é o programa que traduz o código *assembly* em código máquina; o que é executado pelo microcontrolador. O *assembler* será posteriormente apresentado neste documento.

4.1.5 Instruções de Registo

Neste subconjunto de instruções referem-se os registos R0,...,R7 de qualquer banco de registos. Realce-se que esta classe de instruções é relativamente eficiente, dado que o índice do registo alvo da operação está implícito no *byte* da instrução. No entanto, só é possível endereçar oito valores diferentes, uma vez que somente um banco está activo num determinado momento. Por exemplo, a instrução `INC R4` permite incrementar o conteúdo do registo R4 do banco que estiver seleccionado.

4.1.6 Instruções Específicas a Registos

Algumas instruções referem-se implicitamente a determinados registos no próprio código da instrução. Por exemplo, as instruções `INC DPTR`, `SWAP A` e `MUL AB` são executadas sobre os registos mencionados na própria instrução. No caso da primeira instrução, é incrementado o valor contido em $DPTR$. No segundo caso, os *nibbles* de A (quatro *bits* menos significativos e quatro *bits* mais significativos) são trocados. A última instrução permite multiplicar o conteúdo de A pelo conteúdo de B.

4.2 Tipos de Salto

Como é sabido, o registo PC (*Program Counter*) contém o endereço da próxima instrução a ser executada. Na maior parte dos casos, a ordem pela qual as instruções de um dado programa são executadas é dada pela ordem em que as instruções aparecem no programa. Ou seja, se a instrução que está a ser executada num determinado instante está na n-ésima linha do programa, então, na maior parte dos casos, a próxima instrução a executar estará na $n+1$-ésima linha do programa. Neste sentido, o PC é incrementado no total de *bytes* ocupados pela instrução que está a ser executada, passando

a apontar para a próxima instrução. Existem excepções a este comportamento relativamente ordenado e que estão patentes nas instruções de salto e de chamadas a subrotinas. Com efeito, uma das operações realizadas durante a fase de execução deste tipo de instruções é alterar o conteúdo do PC.

As instruções de salto incondicional simplesmente alteram o PC para o valor presente no operando da instrução. Por exemplo, a instrução `JMP ABCDH` faz com que a próxima instrução a ser executada seja aquela existente no endereço ABCDH. Assim, as instruções de salto incondicional permitem alterar o fluxo de execução do programa independentemente do valor lógico de qualquer condição ou estado.

Qualquer que seja o tipo de salto a realizar (condicional ou incondicional), existem três variedades de salto: i) salto longo, ii) salto absoluto e iii) salto relativo, que têm em vista a construção de programas que ocupem pouco espaço em memória.

4.2.1 Salto Longo

Neste caso pode realizar-se um salto para qualquer endereço dentro dos 64K endereçáveis. Dado que o código da instrução ocupa um *byte*, e dado que os endereços são de 16 *bits*, então este tipo de instrução ocupa três *bytes*. Exemplos de instruções deste tipo são: `LCALL E000H` e `LJMP F000H`.

4.2.2 Salto Absoluto

Este caso requer alguma atenção, dado permitir efectuar saltos para qualquer posição de memória dentro de um segmento de 2KB, através de instruções que ocupam somente dois *bytes*. Para melhor entender o mecanismo subjacente, apresenta-se um exemplo concreto, recorrendo à instrução AJMP. A sintaxe da instrução é `AJMP end11`, em que *end11* representa um endereço de 11 *bits*.

$$end11 = A_{10}A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0$$

Em base binária, a instrução apresenta a seguinte codificação:

$$A_{10}A_9A_800001A_7A_6A_5A_4A_3A_2A_1A_0$$

em que a sequência 00001 representa o código da instrução. O cálculo do endereço para onde será realizado o salto é dado por,

PC=(5 MSb do PC) concatenação (bits 5, 6 e 7 do MSB da instrução)
concatenação (LSB da instrução).

Considere-se o exemplo, 0050H: `AJMP 1F0H` Antes de se efectuar o salto, o valor de *PC* é actualizado, passando a valer 50H+2=52H, equivalente a 0000 0000 0101 0010 em base binária. Por seu lado, o endereço do salto é 1F0H=001 1111 0000. O cálculo do endereço para onde se efectuará o salto é dado por,

00000	001	11110000
(5 MSb's do PC)	(bits 5, 6 e 7 do MSB da instrução)	(LSB da instrução)

O endereço anterior vale 01F0H. Observe-se que os cinco *bits* mais significativos (00000) representam o endereço ou código da página de 2KB onde se efectua o salto. Existem então $2^5 = 32$ páginas de 2KB. Note-se ainda que este tipo de salto é absoluto dentro da página considerada, ou seja, podem efectuar-se saltos para qualquer posição dentro da página.

Para terminar, faz-se uma chamada de atenção para os casos fronteira. Por exemplo, a instrução `AJMP 000H`, quando presente no endereço 07FEH=0000 0111 1111 1110_2, não significa que o salto seja feito para a primeira posição da página 00000_2, como é, por vezes, típico pensar-se. Ou seja, a actualização do *PC*, feita pela aritmética 07FEH+2+=0800H, provoca uma mudança de página. Assim, o salto é feito para a posição 000H da página 00001_2 e não da página 00000_2. Esta situação merece atenção por parte do projectista, uma vez que os *assemblers* não a detectam[1].

4.2.3 Salto Relativo

No terceiro e último tipo, são permitidos saltos relativos com base no valor actual do *PC*. O cálculo é feito somando o *PC* com o valor representativo do salto, que deve ser entendido como estando representado em complemento para dois. Como exemplo ilustrativo, considere-se a instrução SJMP FEH residente no endereço 1000H. O *PC* vale 1002H (1000H+2) durante a execução da instrução. A soma do *PC* (1002H=0001 0000 0000 0010) com FEH=1111 1111 1111 1110B (valor representativo do salto em complemento

[1]Com efeito, os *assemblers* não são obrigados a detectá-la, dado não se tratar de uma situação errónea. Quem tem de tomar precauções é o programador.

para dois) é 0001 0000 0000 0000B=1000H. Ou seja, foi realizado um salto para trás de duas posições de memória [2]. Verifica-se então que as instruções de salto relativo permitem realizar saltos dentro de um intervalo de 256B centrado em torno do valor do PC – [PC-128,PC+127].

4.3 Assembly

Nesta secção são apresentadas as instruções pertencentes ao *assembly* dos microcontroladores MCS 51 (denominado ASM 51) em classes de operação ou funcionalidade. Os grupos básicos existentes são os seguintes: saltos incondicionais, saltos condicionais, instruções sobre *bits*, transferência de dados, instruções lógicas, instruções aritméticas e intruções relativas ao *stack*. Para cada grupo apresentam-se as instruções em função dos modos de endereçamento que suportam, o número de *bytes* que ocupam e a quantidade de períodos de relógio que consomem.

4.3.1 Saltos Incondicionais

Na tabela 4.1 são apresentadas as instruções do ASM 51 que permitem efectuar saltos do tipo incondicional. As instruções `AJMP end11`, `LJMP end16` e `SJMP rel` executam saltos do tipo absoluto, longo e relativo, respectivamente, segundo a secção 4.2.

A instrução `JMP @A+DPTR` merece uma menção especial, dado permitir implementar a estrutura de programação conhecida por `case` (caso da linguagem PASCAL) ou `switch` (caso da linguagem C). O endereço do salto é calculado através da soma do conteúdo do $DPTR$ com o conteúdo de A. Na situação mais comum, a $DPTR$ é atribuído o endereço base de uma tabela de saltos, e a A é atribuído o valor da linha da tabela associada ao salto que se pretende efectuar. Considere-se o seguinte exemplo onde se pretende executar um de três blocos de código com base no valor do acumulador.

[2] Note-se que 1000H é o endereço da própria instrução de salto, pelo que o salto é como tivesse sido feito para a própria instrução. Na realidade, é um salto para trás porque, durante a interpretação da instrução, o PC foi incrementado.

Adicionalmente, o leitor deve reparar que se trata de um ciclo infinito, o que, do ponto de vista da teoria da computação, é uma situação indesejável, no sentido de que um algoritmo deve terminar sempre. No entanto, na programação de microcontroladores, este tipo de construção é muito utilizada.

Mnemónica	Operação	Períodos	Bytes
`AJMP end11`	salto para o endereço de 11b	24	2
`LJMP end16`	salto para o endereço de 16b	24	3
`SJMP rel`	salto para o endereço relativo	24	2
`JMP @A+DPTR`	salto para o endereço A+DPTR	24	1

Tabela 4.1: Instruções que permitem realizar saltos incondicionais.

```
                MOV DPTR,\#caso1 ; endereço da base
                MOV A,\#x ; x representa o índice do bloco e vale 0, 1 ou 2
                RL A ; duplicar A
                JMP @A+DPTR ; salto para a tabela de saltos
      depois:
         ...    ...
       caso1:   AJMP codigocaso1 ; endereço do primeiro bloco de código
       caso2:   AJMP codigocaso2 ; endereço do segundo bloco de código
       caso3:   AJMP codigocaso3 ; endereço do terceiro bloco de código
         ...    ...
  codigocaso1:  ... ; código do primeiro bloco
                ...
                AJMP depois
  codigocaso2:  ... ; código do segundo bloco
                ...
                AJMP depois
  codigocaso3:  ... ; código do terceiro bloco
                ...
                AJMP depois
```

Observe-se que cada instrução de salto ocupa dois *bytes* (`AJMP XXH`), pelo que torna-se necessário transformar o valor de A (que pode ser 0, 1 ou 2) num número par; razão da existência da instrução RL (*Rotate Left*), que efectua uma rotação dos *bits* para a esquerda. A instrução `AJMP depois` garante o retorno ao programa principal. Verifique-se ainda que, apesar de se usar o registo $DPTR$, a memória mencionada é a memória de programa.

4.3.2 Saltos Condicionais

Este tipo de instruções está discriminado na tabela 4.2, em que `<byte>`, sendo função do modo de endereçamento, representa o endereço do *byte* que se pretende testar.

Neste tipo de saltos, o fluxo de execução do programa pode ser alterado com base na verificação de determinadas condições, permitindo implementar a conhecida estrutura de programação *if-then*. No entanto, a condição de teste é algo pobre, verificando somente se o conteúdo do acumulador é zero (JZ) ou diferente de zero (JNZ).

É também possível implementar directamente as estruturas repetitivas *repeat-until*, *do-while* e *for*, através das instruções CJNE (*Compare and Jump if Not Equal*) e DJNZ (*Decrement and Jump if Not Zero*), respectivamente. Um exemplo ilustrativo do primeiro caso é mostrado de seguida, onde se decrementa o conteúdo do endereço 10H (da memória de dados interna) até ser igual à constante (imediata) 15H.

```
        MOV 10H,#20H
outra:  DEC 10H
        CJNE 10H,#15H,outra
```

No próximo exemplo apresenta-se um ciclo baseado na instrução DJNZ. A compreensão do significado é deixada ao leitor.

```
        MOV 10H,#10H
outra:  DJNZ 10H,outra
```

4.3.3 Instruções Lógicas

As instruções lógicas permitem realizar as operações simples pertencentes à lógica proposicional ou de BOOLE. As operações implementadas são a conjunção (ANL), a disjunção inclusiva (ORL), a disjunção exclusiva (XRL) e a negação (CPL). Existe ainda um outro conjunto de instruções onde estão incluídas as rotações e a troca de *nibbles*. A lista de instruções está patente na tabela 4.3, onde se pode constatar a diversidade de tipos de endereçamento que podem ser usados para determinação dos operandos. A este respeito, apresentam-se alguns exemplos relativos à instrução ANL.

Mnemónica	Operação	Endereço	P	B
`JZ rel`	salto para `rel` se A=0	só A	24	2
`JNZ rel`	salto para `rel` se A≠0	só A	24	2
`DJNZ \verb¡byte¿",rel"`	decremento de `\verb¡byte¿` e salto para `rel` se `\verb¡byte¿`≠0	dir. e reg.	24	2 ou 3
`CJNE A,\verb¡byte¿",rel"`	salto para `rel` se A≠`\verb¡byte¿`	dir. e imed.	24	3
`CJNE \verb¡byte¿",#dado,rel"`	salto para `rel` se `\verb¡byte¿`≠dado	ind. e reg.	24	3

Tabela 4.2: Instruções que permitem realizar saltos condicionais, onde a coluna P refere-se à quantidade de períodos de relógio consumidos e a coluna B refere-se aos *bytes* ocupados.

```
ANL A,7FH     (endereçamento directo)
ANL A,@R1     (endereçamento indirecto)
ANL A,R6      (endereçamento por registo)
ANL A,\#53H   (constante imediata)
```

Relativamente às rotações, note-se que, nas instruções RL e RR, o *bit* que sai entra pelo outro extremo (*vide* figura 4.1), não havendo alteração do registo PSW. No caso das instruções RLC e RRC, o *bit* C (oitavo *bit* de PSW) serve de entreposto para o MSb ou LSb, respectivamente, como também se mostra na figura 4.1. Finalmente, no caso da instrução `SWAP A`, os *nibbles* do acumulador são trocados, *i.e.*, se inicialmente se tem $A = X_1X_2X_3X_4Y_1Y_2Y_3Y_4$, após a execução da instrução ter-se-á $A = Y_1Y_2Y_3Y_4X_1X_2X_3X_4$, em que X_i e Y_i representam os dígitos binários ('1' e '0').

4.3.4 Instruções Aritméticas

Estas instruções, patentes na tabela 4.4, permitem realizar as quatro operações aritméticas básicas: adição (ADD e ADDC), subtracção (SUBB), multiplicação (MUL) e divisão (DIV). No caso das duas primeiras, a aritmética é realizada a oito *bits*, *i.e.*, os operandos e o resultado ocupam oito *bits*, respec-

Mnemónica	**Operação**	**Endereça**	**P**	**B**
`ANL A,<byte>`	A=A AND `<byte>`	dir., ind., reg. e imed.	12	1 ou 2
`ANL <byte>,A`	`<byte>`=`<byte>` AND A	dir.	12	2
`ANL <byte>,\#dado`	`<byte>`=`<byte>` AND `dado`	dir., ind., reg. e imed.	24	3
`ORL A,<byte>`	A=A OR `<byte>`	dir., ind., reg. e imed.	12	1 ou 2
`ORL <byte>,A`	`<byte>`=`<byte>` OR A	dir	12	2
`ORL <byte>,#dado`	`<byte>`=`<byte>` OR `dado`	dir., ind., reg. e imed.	24	3
`XRL A,<byte>`	A=A XOR `<byte>`	dir., ind., reg. e imed.	12	1 ou 2
`XRL <byte>,A`	`<byte>`=`<byte>` XOR A	dir.	12	2
`XRL <byte>,#dado`	`<byte>`=`<byte>` XOR `dado`	dir., ind., reg. e imed.	24	3
`CRL A`	A=00H	só A	12	1
`CPL A`	A=NOT A	só A	12	1
`RL A`	*rotate left*	só A	12	1
`RLC A`	*rotate left through carry*	só A	12	1
`RR A`	*rotate right*	só A	12	1
`RRC A`	*rotate right through carry*	só A	12	1
`SWAP A`	*nibble exchange*	só A	12	1

Tabela 4.3: Instruções que permitem realizar operações lógicas. Novamente, a coluna P apresenta a quantidade de períodos de relógio necessários à execução da instrução e a coluna B apresenta a quantidade de *bytes* ocupados.

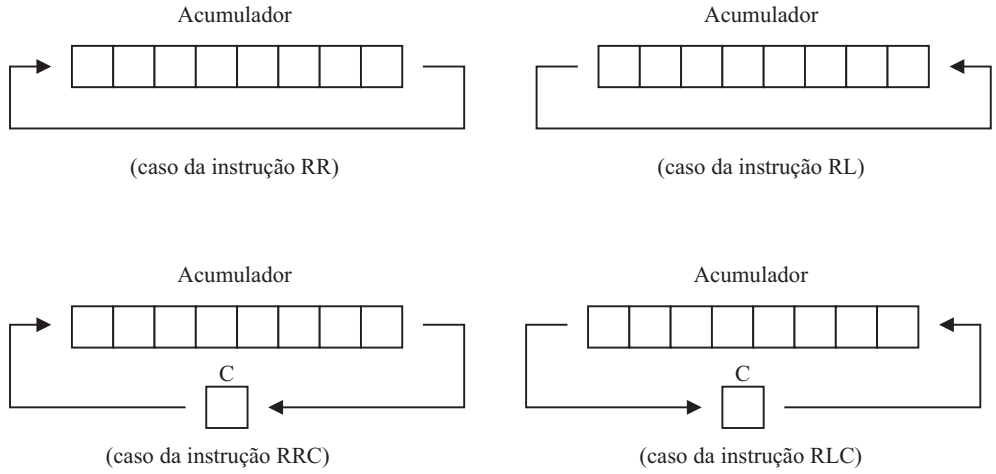

Figura 4.1: Ilustração das rotações para a esquerda e direita, directas e passando pelo *bit* C existente em PSW.

tivamente. Nos dois últimos casos, apesar dos operandos serem de oito *bits*, o resultado fica distribuído por dois registos num total de 16 *bits*. Na multiplicação, o LSB fica armazenado em A e o MSB fica em B. Na divisão, em A fica armazenado o quociente da divisão inteira e em B fica armazenado o resto. Saliente-se ainda que ambas gastam 48 períodos de relógio até estarem concluídas.

Para além da aritmética básica, é ainda possível realizar incrementos (sobre a memória de dados interna e sobre o DPTR) e realizar decrementos (somente sobre a memória interna).

Note-se que as instruções aritméticas podem modificar os *bits carry*, *auxiliary carry* e *overflow* pertencentes ao PSW.

A instrução `DA A` merece uma menção especial, dado permitir a realização simples, mas parcial, de aritmética em BCD (*Binary Coded Decimal*). O ajuste decimal deve ser feito quando o resultado da soma[3] de dois números codificados em BCD é superior a 9. Nestas circunstâncias, deve somar-se o 6 para que a soma seja correcta. A instrução `DA A` tem a capacidade de reconhecer a situação em que é necessário proceder ao ajuste, bem como realizar o ajuste propriamente dito. Dado que um *byte* contém dois dígitos BCD, as situações nas quais se deve realizar o ajuste são: quando C=0 e AC=1, C=1 e AC=0 e C=1 e AC=1. Nestas situações, o ajuste corresponde

[3]O ajuste também deve ser feito na subtracção, naturalmente.

Mnemónica	**Operação**	**End.**	**P**	**B**
`ADD A,<byte>`	A=A+`<byte>`	dir., ind., reg. e imed.	12	1 ou 2
`ADDC A,<byte>`	A=A+`<byte>`+C	dir., ind., reg. e imed.	12	1 ou 2
`SUBB A,<byte>`	A=A-`<byte>`-C	dir., ind., reg. e imed.	12	1 ou 2
`INC <byte>`	`<byte>`=`<byte>`+1	A, dir., ind. e reg.	12	2
`DEC <byte>`	`<byte>`=`<byte>`-1	A, dir., ind. e reg.	12	2
`INC DPTR`	DPTR=DPTR+1	só DPTR	12	2
`MUL AB`	B:A=B×A	só A e B	48	1
`DIV AB`	A=int(A/B), B=mod(A/B)	só A e B	48	1
`DA A`	ajuste decimal	só A	12	1

Tabela 4.4: Instruções que permitem realizar operações aritméticas.

à soma de 06H, 60H ou 66H, respectivamente. Deve ter-se em atenção que a instrução `DA A` só deve ser executada após uma instrução de soma (ADD ou ADDC) e nunca após uma instrução de subtracção, tendo-se então uma limitação.

Há ainda que referir que a instrução `DA A` não permite transformar a representação em binário puro para BCD, como por vezes se tende a pensar. Como afirmado anteriormente, a instrução realiza somente o ajuste decimal.

4.3.5 Instruções sobre *bits*

O tipo de instruções que operam directamente sobre *bits* fazem parte do denominado processador booleano, *i.e.*, uma espécie de processador dentro do processador do 8051 que opera sobre *bits*. Este processador é constituído

por uma memória (a memória endereçável por *bit*), por um acumulador (*bit* C de PSW), por um conjunto de instruções (ver tabela 4.5) e por capacidades de E/S (escrita e leitura directamente nos portos de E/S). Este processador apresenta grande interesse para os programadores, dado permitir evitar o uso de máscaras sobre *bytes* para actuar sobre *bits* isoladamente.

O conjunto de instruções está presente na tabela 4.5, onde ¡bit¿ representa o endereço do *bit* sobre o qual a instrução opera. As instruções podem ser divididas nos seguintes subconjuntos: lógica (ANL e ORL), movimentação (MOV), atribuição (CLR, SETB e CPL) e salto condicional (JC, JNC, JB, JNB e JBC).

Saliente-se a existência de instruções que são referentes ao estado de um *bit* ou à sua negação (lógica positiva *versus* lógica negativa). Observe-se ainda que nem todas as instruções são referentes ao acumulador do processador booleano (neste caso o *bit* C), mas sim a qualquer *bit* que possa ser endereçável (memória interna de dados de 20H a 2FH e todos os registos SFR cujo endereço termine em 0H ou 8H). Finalmente, saliente-se que no caso da instrução JBC, o *bit* endereçado só é colocado a zero caso o teste ¡bit¿=1 seja verdadeiro.

4.3.6 Transferência de Dados

Este tipo de instrução é dos mais utilizados em qualquer programa. Este facto prende-se com o tipo de programação em *assembly*, que, sendo de baixo nível, obrigada a muitas movimentações de dados. As instruções de transferência de dados permitem a movimentação/cópia de valores entre as várias posições de memória (interna e externa). No caso do ASM 51, as instruções deste tipo podem ser agrupadas em dois blocos principais: transferências de dados entre a memória interna (apresentadas na tabela 4.6) e transferências de dados entre a memória externa e a memória interna (apresentadas nas tabelas 4.7 e 4.8). Dentro deste último conjunto, existe um subgrupo de instruções que permite transferir valores da memória externa para a memória interna, sendo denominadas por instruções de leitura de tabelas (apresentadas na tabela 4.8).

Transferências Entre a Memória Interna

Cada instrução deste tipo (ver tabela 4.6) pode assumir várias formas, em virtude dos vários modos de endereçamento que podem ser utilizados. Por

Mnemónica	**Operação**	**P**	**B**
`ANL C,<bit>`	C=C AND `<bit>`	24	2
`ANL C,/<bit>`	C=C AND não `<bit>`	24	2
`ORL C,<bit>`	C=C OR `<bit>`	24	2
`ORL C,/<bit>`	C=C OR não `<bit>`	24	2
`MOV C,<bit>`	C=`<bit>`	12	2
`MOV <bit>,C`	`<bit>`=C	24	2
`CLR C`	C=0	12	1
`CLR <bit>`	`<bit>`=0	12	2
`SETB C`	C=1	12	1
`SETB <bit>`	`<bit>`=1	12	2
`CPL C`	C=não C	12	1
`CPL <bit>`	`<bit>`=NOT `<bit>`	12	2
`JC rel`	salto para *rel* se C=1	24	2
`JNC rel`	salto para *rel* se C=0	24	2
`JB <bit>,rel`	salto para *rel* se `<bit>`=1	24	3
`JNB <bit>,rel`	salto para *rel* se `<bit>`=1	24	3
`JBC <bit>,rel`	salto para *rel* e `<bit>`=0 se `<bit>`=1	24	3

Tabela 4.5: Instruções que permitem efectuar operações sobre *bits*.

Mnemónica	Operação	Modo	P	B
`MOV A,<ft>`	A=`<ft>`	dir., ind., reg., ime.	12	1, 2
`MOV <dest>,A`	`<dest>`=A	dir., ind., reg.	12	1, 2
`MOV <dest>,<ft>`	`<dest>`=`<ft>`	dir., ind., reg., ime.	24	2, 3
`MOV DPTR,\#dado16`	DPTR=`dado16`	imediato	24	3
`XCH A,<byte>`	A e `<byte>` trocam conteúdos	dir., ind., reg.	12	1, 2
`XCHD A,@Ri`	A e @Ri trocam *nibbles* menos significativos	indirecto	12	1

Tabela 4.6: Instruções que permitem realizar transferência de dados entre a memória interna.

exemplo, no caso da instrução genérica `MOV <dest>,<ft,>` em que `<dest>` representa o destino e `<ft>` representa a fonte, tem-se:

```
MOV 10H,20H     ; endereço directo para endereço directo
MOV 10H,@R1     ; endereço indirecto para endereço directo
MOV 10H,R1      ; registo para endereço directo
MOV 10H,\#10H   ; constante imediata para endereço directo
MOV @R0,10H     ; endereço directo para endereço indirecto
MOV @R0,\#10H   ; constante imediata para endereço indirecto
MOV R6,10H      ; endereço directo para registo
MOV R6,\#10H    ; constante imediata para registo
```

Saliente-se a possibilidade de copiar valores de qualquer endereço da memória sem haver a necessidade de recorrer ao acumulador.

Transferências de dados entre as memórias externa e interna

Neste tipo de instruções (ver tabela 4.7), a movimentação de dados é feita por intermédio do acumulador. O acesso à memória externa pode ser feito através de endereços de oito *bits* (caso das instruções `MOVX A,Ri@` e `MOVX Ri@,A`) ou de 16 *bits* (caso das instruções `MOVX A,@DPTR` e `MOVX @DPTR,A`). No caso do acesso a oito *bits*, o endereço é colocado em $P0$, ficando $P2$ disponível para

Mnemónica	**Operação**	**Períodos**	*Bytes*
`MOVX A,@Ri`	A=conteúdo do conteúdo de Ri	2	1
`MOVX @Ri,A`	conteúdo do conteúdo de Ri=A	2	1
`MOVX A,@DPTR`	A=conteúdo do conteúdo de DPTR	1	1
`MOVX @DPTR,A`	conteúdo do conteúdo de DPTR=A	1	1

Tabela 4.7: Instruções que permitem realizar transferência de dados entre a memória externa e o acumulador.

Mnemónica	**Operação**	**Per**	**B**
`MOVC A,@A+DPTR`	A=conteúdo do endereço DPTR+A	2	1
`MOVC A,@A+PC`	A=conteúdo do endereço PC+A	2	1

Tabela 4.8: Instruções que permitem realizar leitura de tabelas.

E/S. Neste caso acede-se a 256B de memória externa. Caso se necessite aceder a alguns KB de memória sem utilizar o porto $P2$ na sua totalidade, *i.e.*, utilizando as instruções MOVX sem DPTR, dever-se-ão utilizar alguns *bits* de $P2$[4] para controlar *bit* a *bit* as linhas mais significativas do endereço, como se mostra na figura 4.2.

Leitura de Tabelas

Este tipo de instrução está patente na tabela 4.8. O respectivo modo de funcionamento foi abordado na secção 4.1.4.

4.3.7 Instruções relativas ao *Stack*

O *stack* é uma área de memória utilizada para guardar valores de carácter temporário. Os valores são armazenados no *stack* por ordem do programador

[4]Os portos de E/S $P1$ ou $P3$ também poderão ser utilizados para este fim.

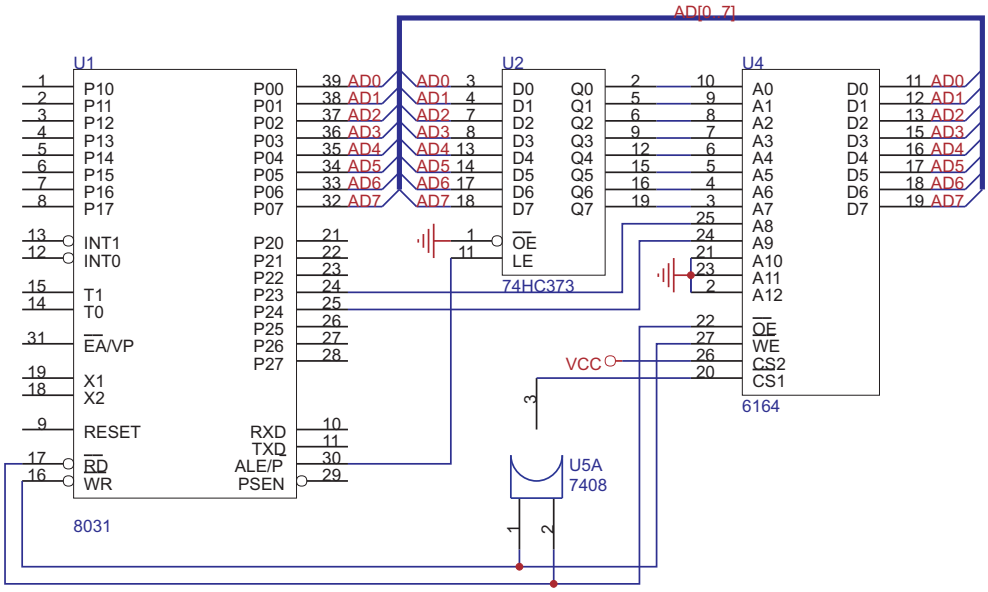

Figura 4.2: Acesso a 8 *bits* a uma memória externa de 1KB, em que os dois *bits* mais significativos do endereço são controlados por dois pinos de E/S (P2.3 e P2.4).

(caso das instruções POP e PUSH) ou por iniciativa do processador (caso de ACALL, LCALL, RET e RETI). A memória de *stack* (ou o *stack*) é geralmente entendida como um depósito de valores na qual não é importante saber onde e como são guardados os valores. Esta filosofia é diferente daquela utilizada para a restante memória, na qual o programador deve manter um registo dos endereços utilizados. A única característica fundamental relativamente à interacção com o *stack*, diz respeito à ordem de entrada e saída dos valores, que, no caso presente é o último valor a entrar é o primeiro a sair[5]. Este tipo de interacção está directamente ligada ao uso de subrotinas. Com efeito, a ordem pela qual termina a execução das subrotinas é a inversa da ordem da sua chamada, lembrando, por exemplo, a acção de empilhar blocos, seguida da acção inversa, como se pode observar na figura 4.3. Devido a esta analogia, a memória de *stack* é, em português, denominada memória de pilha.

Quando é dada ordem de execução de uma subrotina, o conteúdo do PC é armazenado no *stack* antes da execução da primeira instrução. Quando terminar a execução da subrotina, o conteúdo do PC é recuperado do *stack* para que a execução do programa continue na instrução seguinte àquela onde foi feita a chamada. Este esquema de operações está patente na figura 4.4. Estas operações são automaticamente desencadeadas pelo processador. Considere-se o seguinte exemplo,

Endereço	**Instrução**
1000H:	ACALL 2000H
1003H:	×
...	...
2000H:	×
...	...
××××:	RET

(4.1)

onde o símbolo × representa informação não importante para o efeito. Durante a execução da instrução ACALL o PC vale 1003H, que é o endereço da instrução a executar após o fim da subrotina. Em termos do próprio *assembly*, a instrução ACALL realiza as seguintes instruções:

```
PUSH PC
MOV PC,#2000H
```

(4.2)

[5]FCFS, First Come First Served, ou LIFO, Last In First Out.

Ou seja, o conteúdo do PC é guardado no *stack* (através da instrução PUSH) e de seguida é-lhe atribuído o endereço da primeira instrução da subrotina[6]. A instrução de retorno da subrotina (RET) permite repor no PC o endereço da instrução seguinte à de chamada, *i.e.*, é desencadeada a instrução POP PC.

As instruções que acedem à memória de *stack* estão presentes na tabela 4.9, ao passo que as acções desencadeadas por cada instrução são apresentadas na tabela 4.10, onde (SP) representa o conteúdo do registo SP (*Stack Pointer*) pertencente ao espaço SFR. Este registo contém o endereço da útima posição da memória de *stack* (por vezes denominado topo do *stack*). A pilha reside na memória de dados interna.

Como se observará posteriormente, após uma acção de reinicialização[7] do microcontrolador, é atribuído o valor 07H ao registo SP. Dado que as instruções PUSH, ACALL e LCALL incrementam o registo SP e só depois armazenam o valor, então o primeiro valor a ser escrito no *stack* ficará residente no endereço 08H, a que corresponde o registo R0 do segundo banco de registos. Este facto poderá originar um programa defeituoso caso o programador utilize "esse" R0 para outros fins. Nesta situação, o programador deverá modificar a origem do topo do *stack,* atribuindo um valor diferente a SP. O valor a escolher depende do programa em causa. A decisão deverá ter em atenção a ocupação da memória de dados interna.

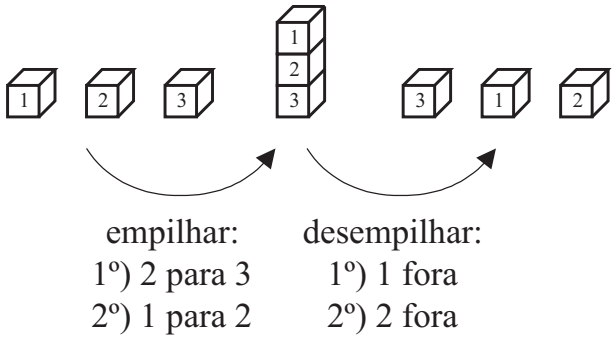

Figura 4.3: Ilustração das acções de empilhar e desempilhar, evidenciando que a ordem da primeira é a inversa da segunda.

[6]Note-se que a instrução PUSH é utilizada de uma forma sintaticamente incorrecta. Com efeito, como posteriormente se verificará, o operando é de oito *bits* e não de 16. Este exemplo permite somente introduzir a funcionalidade interna da chamada de subrotinas.

[7]*Reset* na literatura anglo-saxónica.

Mnemónica	**Operação**	**Per**	**B**
PUSH end8	guardar no *stack* o conteúdo do endereço *end8*	24	2
POP end8	retirar um *byte* do *stack* e armazena-lo no endereço *end8*	24	2
ACALL end11	executar a subrotina que se inicia no endereço absoluto *end11*	24	2
LCALL end16	executar a subrotina que se inicia no endereço longo *end11*	24	3
RET	retorno da subrotina	24	1
RETI	retorno da subrotina de atendimento de uma interrupção	24	1

Tabela 4.9: Instruções que efectuam operações sobre a memória de *stack*.

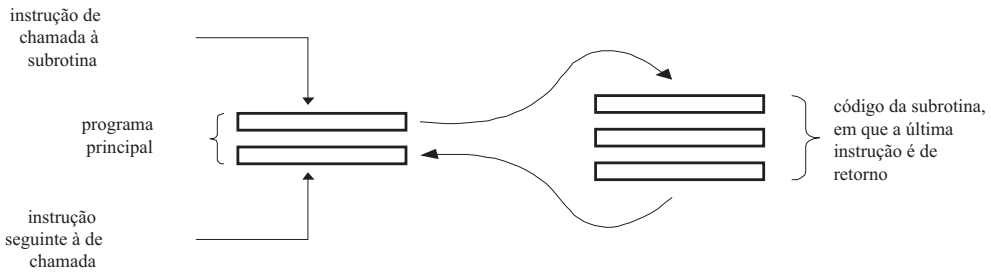

Figura 4.4: Ilustração da chamada a uma subrotina. O endereço do retorno ao programa principal é recuperado a partir do *stack*.

Instrução	**Acção**
PUSH	(SP)←(SP)+1 ((SP))←(directo)
POP	(directo)←((SP)) (SP)←(SP)-1
ACALL	(PC)←(PC)+2 (SP)←(SP)+1 ((SP))←(PC_{7-0}) (SP)←(SP)+1 ((SP))←(PC_{15-8}) (PC)←endereço da página
LCALL	(PC)←(PC)+3 (SP)←(SP)+1 ((SP))←(PC_{7-0}) (SP)←(SP)+1 ((SP))←(PC_{15-8}) (PC)←endereço 15-0
RET	(PC_{15-8})←((SP)) (SP)←(SP)-1 (PC_{7-0})←((SP)) (SP)←(SP)-1
RETI	(PC_{15-8})←((SP)) (SP)←(SP)-1 (PC_{7-0})←((SP)) (SP)←(SP)-1

Tabela 4.10: Acções desempenhadas pelas instruções que actuam sobre a memória de *stack*.

4.4 O Registo PSW - Program Status Word

O registo *PSW*, pertencente ao espaço SFR, apresenta a configuação presente na tabela 4.12. A descrição de cada *bit* é feita nas próximas secções.

4.4.1 C - *bit* de *carry*

Este *bit* apresenta uma dupla funcionalidade: serve de acumulador para o denominado processador booleano e de *bit* de empréstimo do *bit* 7 originado em qualquer instrução aritmética ("vai um"). Por exemplo, a instrução `SETB C` realiza a atribuição C=1, ou, usando notação alternativa, (C)←1. Nesta aplicação, *C* desempenha o papel de acumulador do processador booleano. No exemplo `ADD A,R0`, de modo a que A+R0¿255, tem-se C=1, dado haver empréstimo do *bit* 7.

4.4.2 AC - *bit* de *carry* auxiliar

Sempre que, numa instrução aritmética, se tiver um empréstimo (ou "vai um") do *bit* 3 para o *bit* 4 do acumulador, tem-se AC=1. Um exemplo interessante, e que permite demonstrar o tipo de aplicações que podem ser resolvidas por este microcontrolador, diz respeito à aritmética em BCD (*Binary Coded Decimal*). Considere-se que se pretende somar os dígitos BCD 8 e 3. Dado que a soma é superior a 9, então há que somar 6 ao resultado, o que provoca AC=1. Por outras palavras, se a soma de 8 com 3 e 6 fornecer AC=1, então a soma de 8 com 3 ultrapassa o máximo em BDC (que é 9), o que significa que deve somar-se 6 ao resultado, e que o valor de AC deve ser somado ao dígito BCD que ocupa o lugar das dezenas. O valor de AC é, neste exemplo, duplo: ajuda a determinar se foi ultrapassado o dígito 9 e determina o empréstimo para a posição mais significativa seguinte. Considera-se ainda o exemplo ilustrado na figura 4.5.

4.4.3 F1, F0 - *bits* disponíveis

A estes *bits* não está associado nenhum significado pré-estabelecido. Antes, porém, estão disponíveis para utilização pelo programador.

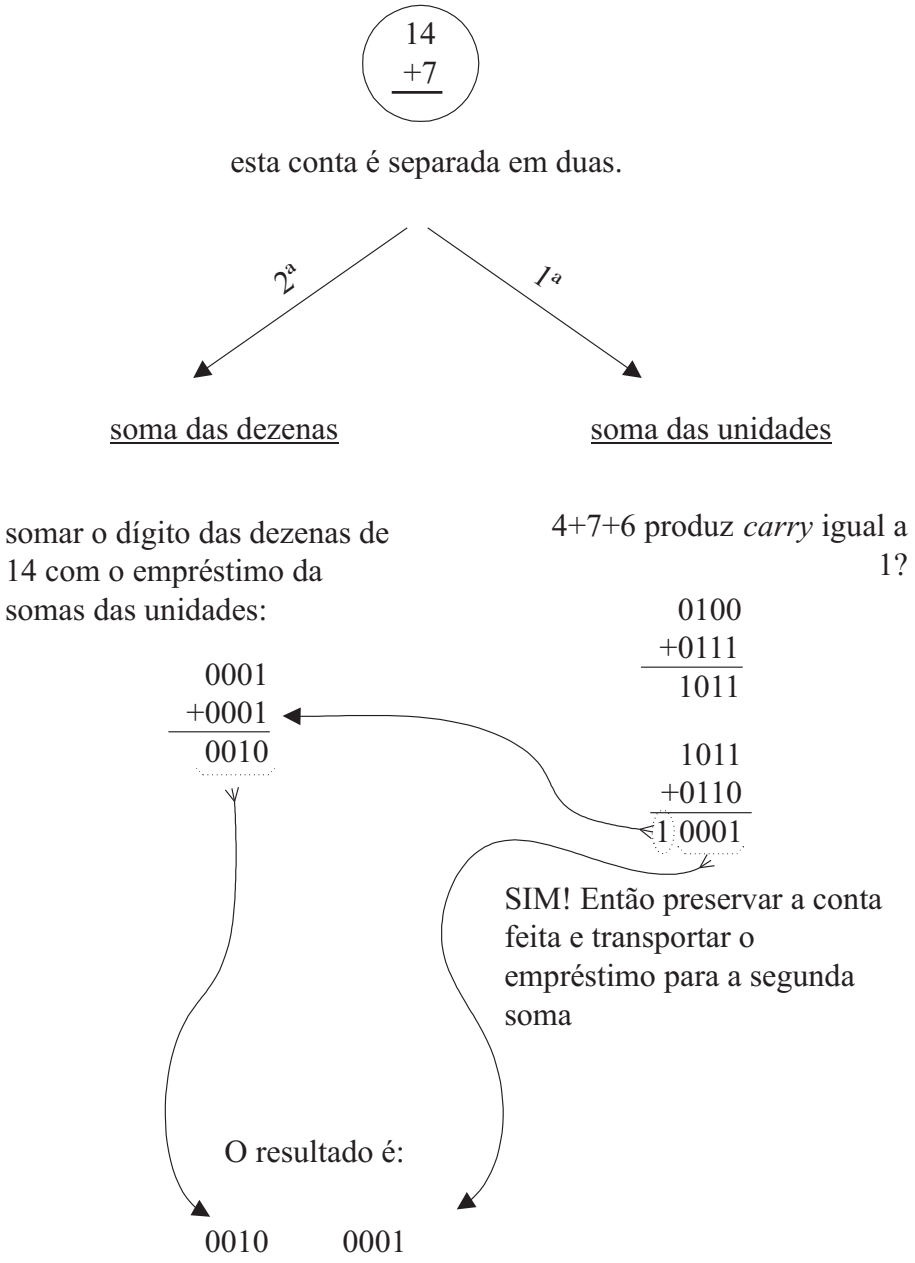

Figura 4.5: Ilustração de uma conta em BCD com mais de um dígito de significância, onde se pretende demonstrar a vantagem em usar o *bit* AC do registo PSW.

RS1	RS0	Banco
0	0	0
0	1	1
1	0	2
1	1	3

Tabela 4.11: Codificação do banco activo em função de $RS1$, $RS0$.

4.4.4 RS1, RS0 - selecção do banco de registos

É por intermédio destes *bits* que se faz a selecção do banco de registos. A codificação de cada banco é apresentada na tabela 4.11.

4.4.5 P - *bit* de paridade

Este *bit* reflecte a paridade do conteúdo do acumulador, de modo a que a soma dos *bits* em A com o *bit* P seja '0'. Por outras palavras, o cálculo a efectuar para se determinar o valor de P, em função do valor de A, é,

$$P = A_7 \oplus A_6 \oplus A_5 \oplus A_4 \oplus A_3 \oplus A_2 \oplus A_1 \oplus A_0$$

4.4.6 OV - *bit* de sobrecarga

Este *bit* sinaliza a situação de sobrecarga na aritmética realizada. Vejamos os casos possíveis.

Instrução de Multiplicação

Sempre que se pretende multiplicar dois números, é sabido que uma parcela deve ser colocada em A e a restante em B. O resultado fica distribuído pelo par B:A. Se o resultado for superior a FFH, então tem-se OV=1, significando que o resultado ocupa 16 *bits*. Caso OV=0 após uma multiplicação, então fica a saber-se que o resultado está totalmente acomodado em A ($A \times B \leq 255$).

Note-se que C é sempre colocado a 0 após uma multiplicação, dado que o resultado nunca é superior a FFFFH.

D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H
C	AC	F0	RS1	RS2	OV	F1	P

Tabela 4.12: Configuração do registo *PSW*.

Instruções Aritméticas Normais - Soma e Subtração

Foi visto que os *bits* *C* e *AC* poderão ser modificados sempre que se utilizar uma instrução aritmética. Se *C* foi modificado, então é porque o resultado ultrapassou FFH, pelo que existe sobrecarga. Esta funcionalidade é útil caso os argumentos estejam codificados em binário puro. Se os argumentos estiverem codificados em complemento para dois, o que permite operar sobre números positivos e negativos, este funcionamento é insuficiente para tirar conclusões acerca do resultado. É neste ponto que entra o *bit* *OV*. Tem-se a atribuição OV=1 sempre que haja empréstimo a partir do *bit* 6 mas não a partir do *bit* 7, ou se tenha empréstimo do *bit* 7 e não do *bit* 6. Sempre que se tiver empréstimo a partir do *bit* 6 e não a partir do *bit* 7, então é porque o MSb de ambos os números é 0 (os números são positivos) e o resultado é negativo (o MSb do resultado é 1). Trata-se da situação em que a soma de dois números positivos resulta num número negativo, o que é falso; como tal, OV=1 sinaliza o erro. Sempre que se tiver empréstimo do *bit* 7 mas não do *bit* 6', então é porque o MSb de ambos os números é 1. O MSb do resultado passa a ser '0', o que determina um resultado positivo, que é igualmente falso.

Fica então assegurada a detecção de resultados falsos em aritmética com sinal. Falta agora endereçar a questão de saber se OV=1 é proveniente da soma de dois números positivos ou de dois números negativos. A questão não deve, tão pouco, ser colocada, dado a consulta do MSb de um dos números dar a resposta imediatamente. Na maior parte das vezes, pretende-se somente saber se o resultado é correcto ou falso, e, neste sentido, *OV* cumpre plenamente o objectivo.

Capítulo 5

O *Assembler* ASM51 da Intel

Após um dado fabricante ter produzido um microcontrolador ou microprocessador, fornece uma série de ferramentas (*hardware* e *software*) destinadas a facilitar o projecto.

Como exemplos de dispositivos (i.e., *hardware*) têm-se os processadores escravos de funcionalidade limitada, como é o exemplo das UARTS, contadores, controladores de interrupções, controladores de portos de E/S, etc. Estes são fornecidos pelo fabricante na perspectiva de estenderem as capacidades do processador principal. Outro tipo de dispositivos são os emuladores, que permitem observar o comportamento do *hardware* desenvolvido e do estado interno do processador em tempo real; ou ainda plataformas de *hardware* que permitem executar código escrito directamente numa memória, seja por intermédio de um teclado, que é parte integrante da plataforma, ou através da ligação a um computador.

Relativamente às ferramentas de software, tem-se igualmente alguma diversidade. Por exemplo: emuladores em *software*, que permitem somente observar o estado do processador à medida que a execução do programa evolui no tempo, seja passo-a-passo ou a uma velocidade perto da real; compiladores de linguagens de alto nível para o *assembly* respectivo; ou, finalmente, o denominado *assembler*, que produz código executável a partir de um ficheiro que contém código fonte em *assembly*. É na análise deste último que está centrado o interesse deste capítulo.

Apesar do *assembler* constituir a ferramenta mais básica que um fabricante disponibiliza, a compreensão que um projectista dele deve ter é importante. O assunto é até bastante vasto. Infelizmente, este capítulo não endereça todos os tópicos com a profundidade desejada.

A partir do anteriormente exposto, deve permanecer a ideia da importância da qualidade das diversas ferramentas e dispositivos associados a um dado microprocessador/microcontrolador disponibilizadas pelo fabricante. Possuir e dominar os dispositivos e as ferramentas significa produzir com eficiência e qualidade.

5.1 *Assemblers*: um Resumo

O programa ASM51 é o compilador do código *assembly* para a série MCS 51 de microcontroladores da Intel. Esta afirmação significa que o *assembler* transforma o código *assembly*, escrito num ficheiro ASCII, em código binário, escrito num outro ficheiro denominado objecto. É este código que tem de ser escrito na memória de programa para que o microcontrolador desempenhe a sua função.

Um microcontrolador original tem um *assembly*, e o seu fabricante disponibiliza um *assembler*. Outros fabricantes poderão fabricar microcontroladores dos quais não tenham sido os criadores; bastando que tenham uma autorização do fabricante original e lhe paguem os direitos de autor que são devidos. Contudo, os *assemblers* não estão sujeitos a direitos de autor, pelo que tipicamente existem vários *assemblers* para um *assembly*

O *assembler* ASM51 é executável em computadores pessoais, que operam a partir de um microprocessador. O código máquina gerado pelo *assembler* não é executável no microprocessador do computador que executou a compilação. Por este facto, o ASM51 é classsificado como um *cross assembler* (*assembler* cruzado). Alguns computadores pessoais são fornecidos com o *assembler* para o microprocessador no qual esses computadores se baseiam. Este é correctamente denominado *assembler*. Concluíndo, existem *assemblers* e *cross assemblers*: os primeiros são programas que correm num computador e produzem código executável para o microprocessador do computador; os segundos são programas que são executados num computador mas não geram código para o microprocessador do computador. Geralmente é feita uma redução do significado correcto dos termos, sendo ambos chamados *assemblers*. Agora que o significado de ambos fica diferenciado, é possível aderir a esta redução, porque, efectivamente simplifica o texto.

A melhor maneira de se apreciar o valor de um *assembler* é produzir um programa na forma exacta como deve existir em memória para execução. A tarefa é bastante fastidiosa. Para além do processo de tradução entre duas

linguagens, um *assembler* fornece certas facilidades com vista à produção eficiente de *software*: seja através do uso de etiquetas, uso de expressões aritméticas, de diversas bases de numeração, definição de variáveis ou símbolos, modularização do programa, etc.

Um programa em *assembly* é habitualmente constituído por três partes:

- Instruções em *assembly*. Estas codificam um dado comportamento que se pretende para o sistema ou aplicação.
- Directivas do *assembler*. Estas permitem definir a estrutura do programa e utilizar certos símbolos pré-definidos e/ou definidos pelo utilizador.
- Controlos do *assembler*. Permitem controlar a execução do próprio *assembler*.

Este texto estende-se basicamente pelas directivas do *assembler*. Relembre-se que as instruções do *assembly* foram cobertas em capítulo próprio.

O *assembler* tem como entrada um ficheiro fonte cuja extensão é da escolha do utilizador. As extensões mais utilizadas são: A51, ASM e SRC. Após o processo de compilação são criados dois ficheiros: o ficheiro objecto, de extensão *.OBJ, e um ficheiro de listagem, com extensão *.LST, que lista informações úteis à depuração de erros. A figura 5.1 ilustra o processo.

Considere-se um exemplo. Pretende-se desenvolver um programa simples que gera uma temporização no pino $P1.0$. O ficheiro fonte, de nome A.A51, é apresentado na figura 5.2. Diversos aspectos são merecedores de explicações. A primeira e última linhas não contêm *assembly*, mas directivas. A primeira determina a posição na memória de programa onde o código será escrito. A última determina que o *assembler* não procurará mais código (*assembly* ou directivas) a partir desta posição. Isto não significa que o ficheiro tenha que terminar neste ponto. É possível criar-se espaço entre a directiva END e o fim do ficheiro para escrever texto livre, possivelmente para alguma explicação importante ou para documentar certos aspectos[1].

Observe-se o uso que se faz das denominadas etiquetas (*labels*), que permitem tornar o programa mais claro para quem o lê. Observe-se também a existência de três partes fundamentais em certas linhas: na primeira parte referem-se as etiquetas (terminadas com o carácter ':'), na segunda aparecem

[1]Infelizmente, esta característica não está disponível em todos os *assemblers*.

as instruções e na terceira os comentários (após o carácter ';'). Relativamente às etiquetas, repare-se que só aquelas referidas na coluna da esquerda é que são complementadas com o carácter ':', e não as da coluna ao centro. Note-se também que qualquer endereço deve ser iniciado, sem excepção, por um dígito numérico. A razão prende-se com a indecisão entre o endereço FFH e o símbolo FFH criado pelo utilizador.

O processo de compilação é executado em resposta ao seguinte comando: `ASM51 A.A51`. O resultado da compilação é apresentado na figura 5.3.

Introduz-se agora, propositadamente, um erro na segunda linha do programa: em vez de P1.0 escreve-se P1.9. O resultado da compilação é apresentado na figura 5.4. Observe-se o texto relativo ao erro, que constitui uma ajuda apreciável, podendo não dispensar a ajuda de um manual. As colunas LOC e OBJ traduzem o que de mais importante acontece na compilação: determinação da localização das instruções em memória e substituição das mnemónicas pelos códigos das instruções acompanhadas dos respectivos argumentos.

Uma nota final para referir o que se passa no final do ficheiro. Todos os símbolos são avaliados com vista à sua substituição pelos endereços respectivos. Os símbolos presentes são: DELAY, LARGE, P1, SMALL e START. Com excepção de $P1$, todos foram definidos pelo utilizador. A avaliação do endereço correspondente é feita pelo *assembler* a partir do endereço expresso na directiva ORG e no espaço ocupado por cada instrução. O endereço associado ao símbolo $P1$ é conhecido pelo *assembler*. A sua utilização fornece maior clareza ao programa. Naturalmente, aconselha-se o uso dos símbolos existentes, como sejam os associados aos registos SFR e aos bits endereçáveis de registos SFR, bem como símbolos criados pelo próprio programador.

5.2 O processo de Ligação (*Linking*)

Apesar de na figura 5.4 se observar que o programa se desenvolve a partir do endereço 0000H, o ficheiro A.OBJ não é executável. Este deve ser aplicado a um outro programa: o RL51, que é um *Relocatable Linker*[2], apresentando a capacidade de gerar o código executável a partir de um ou mais ficheiros objecto. A figura 5.5 retoma e completa o esquema de blocos apresentado na figura 5.1.

[2]Para já, o processo de ligação é exposto com simplicidade, sendo analisado posteriormente neste texto.

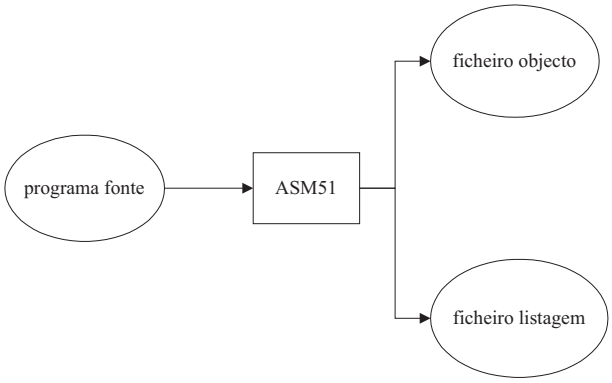

Figura 5.1: Diagrama de blocos ilustrativo do processo de compilação, identificando entradas e saídas.

```
        ORG 0000H       ; início do programa
START:  CPL P1.0        ; complementar o pino
        ACALL DELAY     ; esperar
        AJMP START      ; mais uma iteração
DELAY:  MOV R0,#10H     ; rotina que implementa a espera
LARGE:  MOV R1,#0FFH
SMALL:  DJNZ R1,SMALL
        DJNZ R0,LARGE
        RET             ; fim da rotina
        END             ; fim do ficheiro
```

Figura 5.2: Conteúdo do ficheiro objecto utilizado para exemplo.

```
DOS 7.0 (033-P) MCS-51
MACRO ASSEMBLER, V2.2

Copyright 1979, 1983, 1986 Intel Corporation

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figura 5.3: Resultado da compilação do ficheiro A.A51.

```
MCS-51 MACRO ASSEMBLER  A

DOS 7.0 (033-P) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN .\A.OBJ
ASSEMBLER INVOKED BY:  .\ASM51.EXE .\A.A51

     LOC      OBJ      LINE  SOURCE
     0000               1          ORG 0000H
     0000     B200      2    START: CPL P1.9
  *** ERROR #9, LINE #2 (0), BAD BIT OFFSET IN BIT ADDRESS EXPRESSION
     0002     1106      3          ACALL DELAY
     0004     0100      4          AJMP START
     0006     7810      5    DELAY: MOV R0,#10H
     0008     79FF      6    LARGE: MOV R1,#0FFH
     000A     D9FE      7    SMALL: DJNZ R1,SMALL
     000C     D8FA      8          DJNZ R0,LARGE
     000E       22      9          RET
                       10          END

SYMBOL TABLE LISTING
------------------------------------

NAME     TYPE     VALUE    ATTRIBUTES

DELAY    C ADDR   0006H    A
LARGE    C ADDR   0008H    A
P1       D ADDR   0090H    A
SMALL    C ADDR   000AH    A
START    C ADDR   0000H    A

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, 1 ERROR FOUND (2)
```

Figura 5.4: Resultado da compilação do ficheiro A.A51 com erros.

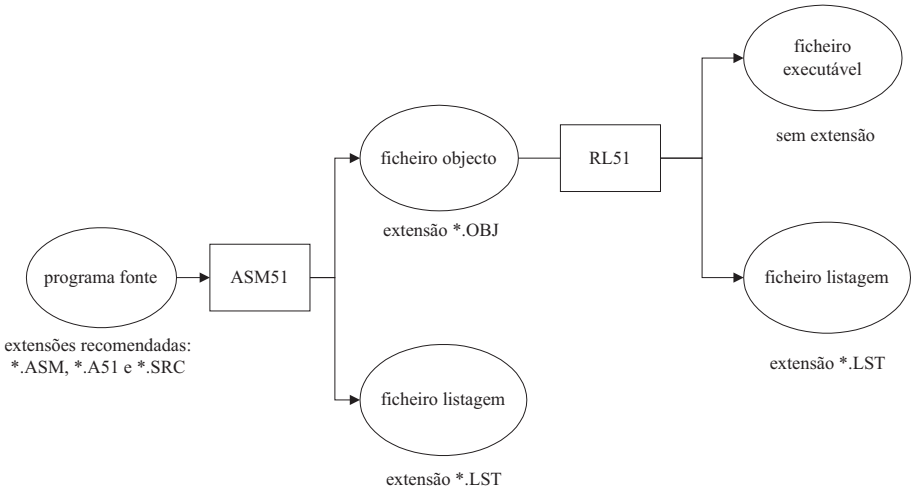

Figura 5.5: Diagrama de blocos ilustrativo do processo de compilação e ligação, identificando entradas e saídas.

5.3 O Formato Hexadecimal

Alguns produtos, como é o caso de certos emuladores, não aceitam o ficheiro objecto em binário, mas sim um ficheiro com extensão *.HEX, que codifica todo o programa no denominado formato hexadecimal.

É de referir que não existe um só formato hexadecimal. O abordado nesta secção é respeitante ao conhecido por Intel Hex. Um ficheiro em formato hexadecimal é constituído por dois tipos de registos: (vários) registos de dados e um registo de fim de ficheiro. Um registo, independentemente do seu tipo, é constituído pelos seguintes campos:

```
:<tamanho><endereço><tipo><dado><dígito de controlo>
```

em que:

`<tamanho>:` representa o número de campos do tipo `dado`, sendo descrito por dois dígitos hexadecimais;

`<endereço>:` indica o endereço a partir do qual os dados referidos no campo do tipo `dado` serão escritos, sendo representado por quatro dígitos hexadecimais;

`<tipo>:` vale '0' para os registos de dados e '1' para o registo de fim de ficheiro, sendo descrito por dois dígitos hexadecimais;

`<dado>:` contém desde um até 16 pares de dígitos hexadecimais, perfazendo um máximo de 16 *bytes* de dados por registo;

`<controlo>:` corresponde à soma de controlo (*checksum*) entre o tamanho do registo, endereço de início, tipo de registo e campo de dados. O cálculo é feito de modo a que a soma dos campos referidos com os dois dígitos hexadecimais do controlo seja '0'.

O registo `:00000001FF` serve para marcar o fim do ficheiro.

No conjunto de programas dos quais o *assembler* ASM51 faz parte, existe um programa que converte os ficheitos em formato objecto para o formato hexadecimal, que recebe o nome de OBJHEX.

5.3.1 Um exemplo

5.4 Operandos e Expressões

Qualquer instrução do *assembly* apresenta a seguinte forma geral:

```
[label:] mnemónica [operando] [,operando] [;comentário]
```

em que a notação [] representa opcional. Assim, o uso de etiquetas e a existência de um comentário é opcional. O número de operandos depende da instrução. Neste momento, o interesse desta exposição diz respeito à possibilidade de referir os operandos através de expressões aritméticas que atendem a uma sintaxe. A referência pode mencionar números, símbolos (pré-definidos ou definidos pelo programador), expressões aritméticas ou outros operadores do *assembler*.

5.4.1 Etiquetas

As etiquetas são símbolos, como tal seguem regras de contrução próprias. Uma etiqueta é o primeiro campo de uma linha de programa, podendo ser precedida por qualquer número de espaços. Para que uma etiqueta seja entendida como tal, deve ser complementada pelo carácter ':'. Só é permitida uma etiqueta por linha. Uma etiqueta pode ser utilizada tão logo seja

definida, e depois de definida não pode ser redefinida. Na utilização de uma etiqueta só deve figurar o nome e não o símbolo ':'.

5.4.2 Avaliação de Expressões em Tempo de Compilação

Uma expressão é uma combinação de números, caracteres, símbolos e operadores dispostos segundo regras de sintaxe próprias, com vista a obter um dado significado. As expressões devem avaliar em números representativos de endereços ou dados. Os números podem ser especificados nas bases hexadecimal (usando o carácter 'H'), decimal (usando 'D'), octal (usando 'O' ou 'Q') e binária (usando o carácter 'B'). Os dígitos disponíveis para cada base estão patentes na tabela 5.6.

Eis alguns exemplos:

base hexadecimal:	0abh, 0ABh, 0ABH, 0abH
base decimal:	40, 40D, 40d
base octal:	16O, 16o, 16Q, 16q
base binária:	0B, 0b, 1B, 1b, 101B

É possível utilizar caracteres ASCII em expressões, havendo que escrevelos entre plicas. Eis alguns exemplos:

'a'	caracter *a*
'ab'	*string ab*
'''	plica
'a'''	*string a*' (a e plica)
''	*string* vazia

base	**dígitos permitidos**
B	0, 1
O ou Q	0, 1, 2, 3, 4, 5, 6
D	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
H	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Figura 5.6: Caracteres representativos das diversas bases acompanhados dos respectivos dígitos.

5.4.3 Operadores Pré-estabelecidos

Operadores Aritméticos

Os operadores disponibilizados aparecem listados na tabela 5.7. Eis alguns exemplos: +27, 13-2, 4*5-21, -32, 2*3/2-1, 100 MOD 50. Neste último caso, há que separar o operador MOD dos argumentos por, pelo menos, um espaço.

operador	símbolo
adição	+
subtracção	-
multiplicação	*
divisão inteira	/
resto da divisão inteira	MOD

Figura 5.7: Lista dos operadores aritméticos.

Operadores Lógicos

Os operadores disponibilizados estão apresentados na tabela 5.8. Eis alguns exemplos:

```
MOV A,#10101010B OR 01010101B
MOV B,#10101010B AND 01010101B
MOV A,#NOT 01010101B
```

operador	símbolo
conjunção	AND
disjunção inclusiva	OR
disjunção exclusiva	XOR
negação	NOT

Figura 5.8: Lista dos operadores lógicos.

Operações Sobre *Bytes* ou 16 *Bits*

Os operadores disponibilizados estão presentes na tabela 5.9. Eis alguns poucos exemplos:

```
LOW 1135H
HIGH (1135H SHL 8)
```

operador	símbolo
deslocamento para a direita	SHR
deslocamento para a esquerda	SHL
byte mais significativo de 16b	HIGH
byte menos significativo de 16b	LOW

Figura 5.9: Lista dos operadores que actuam sobre 8 e 16 *bits*.

Operadores Relacionais

Os operadores disponibilizados estão presentes na tabela 5.10. O resultado de expressões com operadores relacionais pode ser falso, representado pelo decimal '0', ou verdadeiro, representado pelo hexadecimal FFFFH. Sempre que se utilizarem as mnemónicas (por exemplo, EQ em vez de =), há que as separar dos seus argumentos por pelo menos um espaço. Não é necessário ter o mesmo cuidado quando se utilizam os símbolos.

operador	símbolo	mnemónica
igual	=	EQ
diferente	¡¿	NE
menor	¡	LT
menor ou igual	¡=	LE
maior	¿	GT
maior ou igual	¿=	GE

Figura 5.10: Lista dos operadores relacionais.

Precedências

Os operadores anteriores seguem naturalmente uma ordem (precedência), que deve ser levada em consideração na elaboração das expressões. A precedência de todos os operadores está presente na tabela 5.11. É sabido que o operador *parentesis* permite atribuir precedência ou sobrepôr-se à precedência pré-estabelecida. Para os operadores que apresentem a mesma precedência, a sequência de cálculo corresponde à ordem pela qual os operadores estão dispostos, i.e., da esquerda para a direita. Relativamente ao operador '-', há que distinguir entre o operador binário, como por exemplo 2-3, e o operador unário, como por exemplo -3.

operador	precedência
()	alta
HIGH, LOW	
*, /, MOD, SHL, SHR	
+, - (unário e binário)	
EQ, NE, LT, LE, GT, GE, =, ¡¿, ¡, ¿=, ¿, ¿=	
NOT	
AND	
OR, XOR	baixa

Figura 5.11: Ordenação dos operadores em função da sua precedência.

5.5 Símbolos

5.5.1 Símbolos Reservados e Símbolos Definidos Pelo Utilizador

Os símbolos definidos de raiz no *assembler* estão naturalmente reservados, i.e., não poderão ser redefinidos pelo utilizador. A lista de nomes reservados é relativamente extensa. Nela se encontram todas as mnemónicas das instruções, os nomes de todos os operadores (apresentadas na tabela 5.11), directivas do *assembler* (a definir posteriormente), os registos SFR e os seus *bits* endereçáveis.

5.5.2 Regras de Construção de Nomes de Símbolos

Os nomes a utilizar para símbolos devem seguir as seguintes regras de construção:

- devem começar com uma letra ou um dos seguintes caracteres especiais: '?' ou '_';
- podem seguir-se letras, números ou caracteres especiais (? e _), até um máximo de 255 caracteres, embora só os primeiros 31 sejam significativos;
- os nomes que difiram pelo facto das letras serem maiúsculas ou minúsculas constituem o mesmo símbolo.

5.6 Directivas do *Assembler*

As directivas do *assembler* constituem um conjunto de instruções que são interpretadas e executadas durante o processo de compilação, pelo que não são *assembly*. Estas instruções têm, contudo, influência na maneira como o código é gerado pelo *assembler*. é então para o *assembler* que as directivas se destinam.

As directivas do *assembler* são de vários tipos. Nas próximas secções são apresentadas e definidas algumas das consideradas mais importantes.

5.6.1 Definição de Símbolos

EQU

O formato desta directiva é,

```
nome_símbolo EQU expressão
```

Esta directiva permite atribuir um valor numérico ou um símbolo especial do *assembler* (por exemplo; A, $P0$, $SBUF$, etc.) ao símbolo que se está a definir. Por exemplo, a instrução `LED EQU P1.0` permite atribuir ao símbolo LED o valor $P1.0$, que representa o endereço do *bit* menos significativo de $P1$. A partir deste instante, o programa poderá mencionar o símbolo LED as vezes que forem necessárias, tendo-se a vantagem de atribuir maior clareza ao programa.

Um símbolo definido pela directiva EQU não pode ser redefinido. Naturalmente, o nome a dar ao símbolo segue as regras sintácticas de construção de nomes.

Exemplos:

```
ACUMUL EQU A
MENSAGEM EQU 'OLA MUNDO'
```

SET

O formato é o seguinte,

```
nome_símbolo SET expressão
```

Esta directiva é semelhante à anterior, permitindo então criar e atribuir valores a símbolos. A única diferença está na capacidade de poder-se redefinir o

símbolo posteriormente. Um símbolo definido através de EQU não pode ser redefinido por meio de SET.

Exemplos:

```
INICIO SET 1
FIM SET 10
PASSO SET (FIM-INICIO)/10+INICIO
```

BIT

O formato é o seguinte,

```
nome_símbolo BIT endereço
```

Esta directiva permite atribuir valores de endereços de bits a símbolos. Um símbolo definido por BIT não pode ser posteriormente redefinido.

Exemplos:

```
LED BIT P1.0
LIGADO BIT 60H
```

DATA

O formato é o seguinte,

```
nome_símbolo DATA expressão
```

Esta directiva é semelhante à anterior, permitindo atribuir endereços de dados da memória interna a símbolos. Um símbolo definido através de DATA não pode ser redefinido.

Exemplos:

```
SERIE DATA SBUF
BASE DATA 70H
FIM DATA 80H
```

XDATA

O formato é o seguinte,

```
nome_símbolo XDATA expressão
```

Esta directiva é semelhante à anterior, permitindo atribuir endereços de dados da memória externa a símbolos. Um símbolo definido através de XDATA não pode voltar a ser definido.

Exemplos:

```
BASE XDATA 700H
FIM XDATA 800H
```

IDATA

O formato é o seguinte:

```
nome_símbolo IDATA expressão
```

É semelhante às directivas DATA e XDATA, permitindo atribuir a símbolos endereços de dados da memória interna que seja referida indirectamente. A expressão depois de avaliada não poderá resultar em números superiores a 127. Um símbolo definido através de IDATA não pode voltar a ser definido.

Exemplos:

```
INIT_BUFFER IDATA 60H
TAMANHO EQU 20H
END_BUFFER IDATA INIT_BUFFER+TAMANHO-1
```

CODE

O formato é o seguinte,

```
nome_símbolo CODE expressão
```

Esta directiva permite atribuir um endereço de código a um símbolo. Um símbolo criado por intermédio de CODE não pode ser redefinido em nenhuma circunstância.

Exemplos:

```
RESTART CODE 00H
INTR_EXT0 CODE 03H
INTR_CT0 CODE 0BH
INTR_SERIE CODE 23H
```

5.6.2 Reserva e Inicialização de Espaço

Estas directas permitem reservar e inicializar espaço para conjuntos de *bits*, *bytes* ou palavras (16b). O espaço reservado é iniciado na posição indicada pelo contador de localização.

DS

O formato desta directiva é o seguinte,

```
[etiqueta:] DS expressão
```

Esta directiva permite reservar espaço para *bytes*. A expressão não pode conter referências a símbolos por definir. Quando o *assembler* encontra uma directiva DS, o valor do contador de localização dita o início do espaço que se reserva, e o contador é incrementado no montante do espaço reservado.

DBIT

O formato é o seguinte,

[etiqueta:] DBIT expressão

Esta directiva permite reservar espaço para *bits*. A expressão não pode conter referências a símbolos por definir. Quando o *assembler* encontra uma directiva DBIT, o valor do contador de localização de *bits* dita o início do espaço que se reserva e o contador é incrementado no montante do espaço reservado.

DB

O formato é,

```
[etiqueta:] DB lista_de_expressões
```

Esta directiva permite reservar espaço para *bytes* na memória de programa. Os valores a escrever a seguir a DB devem ser separados por vírgulas, podendo ser tantos quanto se queira, desde que não se ultrapasse o endereço máximo. Também é possível escrever *strings* directamente. Os valores são colocados em memória pela ordem em que são escritos.

Exemplos:

```
PRIMOS DB 1,2,3,5,7,11,13,17,19,23,29,31,37
MENSAGEM DB 'ERRO: VALOR NEGATIVO'
```

Para se aceder ao primeiro número primo menciona-se PRIMOS, para se aceder ao segundo menciona-se PRIMOS+1, etc. De igual modo, MENSAGEM pode ser visto como um ponteiro para o primeiro caracter, MENSAGEM+1 refere-se ao segundo caracter, etc.

DW

O formato desta directiva é,

```
[etiqueta:] DW lista_de_expressões
```

Esta directiva permite inicializar a memória de programa com uma lista de valores de 16 *bits*. Os valores devem estar separados por vírgulas. Cada item presente na lista é escrito em memória pela ordem que é escrito, sendo o *byte* mais significativo escrito em primeiro lugar (endereços mais baixos). Caso se pretenda mencionar *strings*, estas não poderão ter mais que dois caracteres.

Exemplos:

```
CHEGADAS DW 'AM',0910,'PM',0910
INVENTARIO DW 'A',100,'B',34,'C',726
TABELA_SALTOS DW ROT_A, ROT_B, ROT_C
```

No último exemplo, apresenta-se uma tabela de saltos, onde os items da lista são os endereços (neste caso em forma de etiquetas) de início de rotinas.

5.6.3 Controlo do Estado do *Assembler*

END

O formato desta directiva é,

```
END
```

Cada programa deve terminar com a directiva END. A seguir a esta directiva só pode aparecer um comentário.

ORG

O formato é,

```
ORG expressão
```

Esta directiva permite alterar o conteúdo do contador de localização, de modo a criar uma nova posição para a colocação do código. Na expressão não podem ser mencionados símbolos ainda não definidos.

Exemplos:

```
INTR_EXT0 SET 03H
ORG INTR_EXT0
(segue-se o código da rotina de atendimento)
INTR_CT0 SET 0BH
ORG INTR_CT0
(segue-se o código da rotina de atendimento)
etc.
```

5.6.4 Segmentos

Um segmento é um bloco de código ou de dados onde existe atribuição a endereços de memória. Um segmento pode ser fixo (diz-se um segmento absoluto) ou pode ser colocado em qualquer posição de memória (diz-se relocalizável). Os segmentos relocalizáveis têm um nome, um tipo e atributos, podendo estar distribuídos por vários segmentos parciais. Os segmentos absolutos não têm nome e não podem ser combinados com outros segmentos.

5.6.5 Módulos

Um módulo é composto por um ou mais segmentos, absolutos ou relocalizáveis, parciais ou não. Qualquer módulo tem um nome. As definições de um módulo permitem determinar o âmbito dos símbolos definidos.

Um ficheiro fonte pode ser constituído por mais de um módulo. Um programa consiste em um módulo absoluto, que corresponde à união de todos os segmentos absolutos e relocalizáveis existentes em vários módulos, que, por sua vez, podem estar distribuídos por vários ficheiros fonte.

Capítulo 6

Circuito de Relógio e Reinicialização

6.1 Circuito de Relógio

Os microcontroladores da série MCS 51 são constituídos por um oscilador interno. Esta afirmação significa que basta aplicar nos pinos XTAL1 e XTAL2 um sinal oscilante com a frequência desejada; o microcontrolador encarrega-se de o transformar numa onda quadrada de nível TTL (níveis de tensão 0V e 5V) e com *duty cycle*[1] de 50%. Por consequência, o circuito a utilizar para alimentar o oscilador interno é composto por poucos componentes, como se pode comprovar pela figura 6.1. O valor da capacidade dos condensadores deve ser de 30pF$\pm$10. O cristal pode apresentar uma frequência de oscilação com qualquer valor entre 6MHz e 12MHz. Presentemente, a Intel disponibiliza versões de microcontroladores que podem ser alimentados com cristais de 16, 20 ou 24MHz.

Alimentar o circuito de oscilação com um cristal é a situação mais comum. Em alternativa, poder-se-á aplicar nos mesmos pinos um oscilador cerâmico, que sendo menos dispendioso, é menos estável. Se o sinal de relógio aplicado a um microcontrolador não tiver uma frequência (relativamente) fixa, então poderá deixar de fazer sentido executar temporizações precisas através de instruções de programa ou através dos dispositivos contadores/temporizadores.

Em alternativa, pode ligar-se uma fonte de relógio externa (de nível TTL),

[1]Percentagem do tempo em que a onda está no nível '1' relativamente ao tempo em que está a '0'.

caso exista, ao pino XTAL1. Se essa fonte de relógio alimentar demasiados dispositivos, poderá ser aconselhável intercalar um porta TTL (inversora, por exemplo) entre a fonte e o pino XTAL1, de maneira a repor as necessidades energéticas do circuito oscilador interno do microcontrolador. O sinal de relógio externo a aplicar deverá estar devidamente tratado, i.e., entre 0V e 5V e ter *duty cycle* de 50%.

Uma característica interessante diz respeito à possibilidade de se poder desligar o oscilador interno por meio de *software*, atribuindo o nível lógico "1" ao *bit* PD existente no registo PCON. Este registo permite escolher um de dois modos de adormecimento do microcontrolador, permitindo uma redução considerável no consumo de energia eléctrica. Este assunto será abordado posteriomente com mais pormenor.

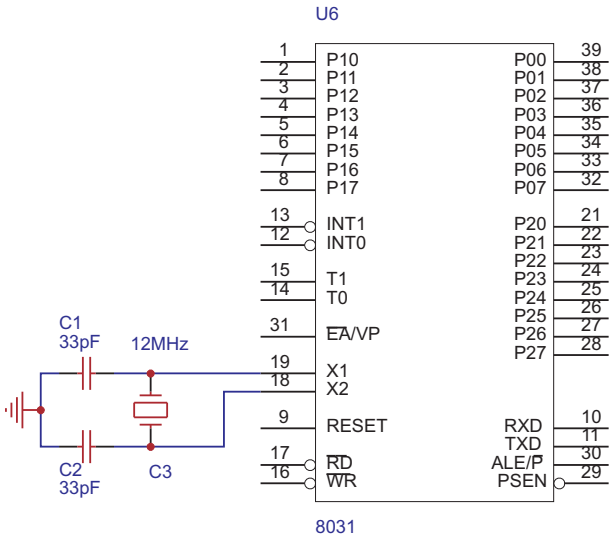

Figura 6.1: Esquemático que evidencia a ligação de um cristal aos pinos XTAL1 e XTAL2.

6.2 Reinicialização

A acção de reinicialização[2] é garantida quando se aplicar uma tensão de 5V durante pelo menos dois ciclos máquina (24 períodos de relógio de 12MHz)

[2] *Reset* na literatura anglo-saxónica.

no pino RST, seguida de uma transição para 0V. Neste sentido, diz-se que a reinicialização (ou o *reset*) do 80C51 é activa a "1". Esta acção deve acontecer após a ligação do sistema digital à alimentação (*power-on reset*), ou por iniciativa de um operador humano através de um botão de pressão (*reset* forçado ou manual). Em qualquer dos casos, deve garantir-se a temporização mínima, que depende da frequência de oscilação aplicada ao microcontrolador. Caso contrário, não é possível garantir o estado normal de arranque do microcontrolador. Por exemplo, o registo PC poderá não ser inicializado correctamente, levando a que o programa não seja executado desde o início, o que poderá levar a um mau funcionamento ou a uma avaria definitiva do *hardware* circundante.

O circuito típico que pode ser utilizado é apresentado na figura 6.2. Caso se deseje prover o circuito com a possibilidade de efectuar um *reset* forçado, então poder-se-á utilizar o esquema apresentado na figura 6.3.

Após a subida da tensão de alimentação até ao nível correcto, é executada automaticamente uma rotina de inicialização que atribui aos registos SFR os valores presentes na tabela 6.1, onde o símbolo $\times$ representa o valor "1" ou "0". Salientam-se dois aspectos particulares. Primeiro, note-se que o registo PC é inicializado com o valor 0000H, o que determina que a primeira instrução do programa deve existir nesse endereço da memória de programa (externa ou interna). Em segundo lugar, deve ter-se em atenção o valor de inicialização do registo SP (*Stack Pointer*), que aponta para o registo R7 do primeiro banco de registos. Dado que uma acção de armazenamento de um dado valor na memória de *stack* é iniciada por um incremento de SP, então o primeiro valor a guardar no *stack* será colocado em R0 do segundo banco de registos. Este pormenor não deverá ser esquecido pelo programador, de outro modo poderá incorrer-se num tipo de erro de difícil remoção e que poderá ser prejudicial para todo o sistema digital. Note-se, para finalizar, que após o fim da execução da rotina interna de reinicialização não é possível determinar o conteúdo da memória de dados interna.

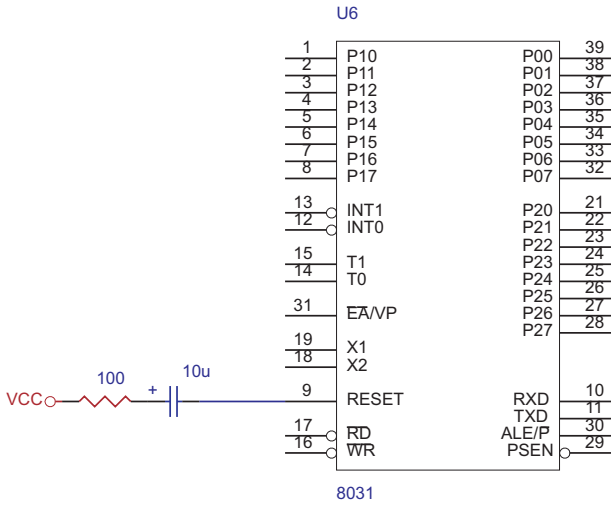

Figura 6.2: Circuito necessário para garantir um *power-on reset* correcto.

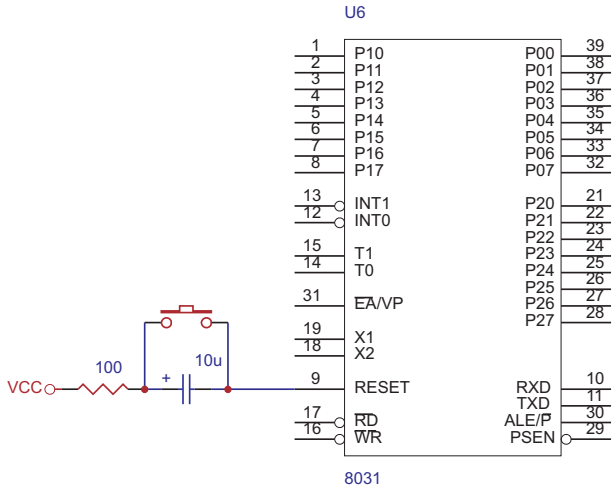

Figura 6.3: Circuito necessário para garantir um *power-on reset* e um *reset* manual correctos.

Registo	**Conteúdo**
A	00000000
B	00000000
PSW	00000000
SP	00000111
DPH	00000000
DPL	00000000
P0	11111111
P1	11111111
P2	11111111
P3	11111111
IP	xxx00000
IE	0xx00000
TMOD	00000000
TCON	00000000
TH0	00000000
TL0	00000000
TH1	00000000
TL1	00000000
PCON	0xxx0000

Tabela 6.1: Conteúdo em binário dos registos SFR após um *reset*.

Capítulo 7

Portos de entrada/saída

A capacidade de ligação de um processador com o exterior é fundamental. Os sinais provenientes do exterior podem ser gerados por dispositivos digitais acoplados ao processador ou por sensores que traduzem o estado de grandezas físicas em valores numéricos. Estes sinais, depois de tratados, tendem a gerar outros que por sua vez exercem controlo sobre outros componentes, semdo este o ciclo típico de uma aplicação baseada em microcontroladores.

As capacidades de entrada/saída do 8051 são sintetizadas pela existência de quatro portos bidireccionais de oito bits, que possibilitam a leitura e a escrita de níveis lógicos nos pinos respectivos. Na realidade, cada pino de cada porto pode ser acedido individualmente, levando a que existam 32 bits de E/S. Esta capacidade é de grande potencial.

7.1 Operação

Do ponto de vista da utilização, os portos de E/S são bastante simples. Como foi declarado no segundo capítulo, a cada porto está associado um registo SFR cujos endereços estão presentes na tabela 7.1. Dado que o endereço de cada porto de E/S termina em 0H ou 8H, então é possível aceder directamente a qualquer bit do registo respectivo. Nas tabelas 7.2, 7.3, 7.4 e 7.5 são apresentados os endereços de cada bit de cada porto.

A interacção que se faz com os pinos de E/S do microcontrolador é realizada por intermédio de leituras e escritas de valores (sejam *bytes* ou *bits*) no registo desejado (através da instrução MOV, por exemplo). Caso se pretenda determinar o nível de tensão no pino, deve escrever-se o valor correspondente

no registo respectivo. Por exemplo: `MOV P0,#XXH` coloca o valor XXH no porto P0, `SETB P1.2` coloca o pino P1.2 a alto, e `CLR P3.5` leva o pino P3.5 a zero. A partir deste instante, os pinos mencionados na instrução emitem continuamente o valor escrito.

Vê-se assim que os portos não necessitam de configuração prévia à sua utilização, significando que não é necessário determinar previamente a função de cada porto ou pino, nomeadamente leitura, escrita ou ambos.

No caso da leitura, existe um aspecto particular que tem de ser compreendido, e que se prende com a arquitectura interna associada aos pinos de E/S. Com efeito, caso se pretenda utilizar um dado pino para entrada, dever-se-á nele escrever o valor lógico 1 previamente à acção de leitura.

Antes de se explicar a razão desta particularidade, são apresentados nas figuras 7.1 e 7.2 os esquemas simplificados da electrónica existentes no porto 0 e nos portos 1, 2 e 3, respectivamente. A observação dessas figuras permite concluir que caso o valor lógico 0 esteja armazenado no *flip-flop* tipo D, então o transístor FET estará no estado de condução, como tal não será possível ler o valor correcto de tensão (nível TTL) aplicado no pino. Opostamente, se no *latch* estiver armazenado o valor lógico 1, então o transístor não conduz e o valor de tensão no pino estará presente no barramento de dados interno, realizando-se a respectiva leitura. Por este motivo se compreende a razão de se anteceder a leitura por uma escrita de 1 no porto. Por exemplo, para ler para o acumulador o valor de oito bits aplicado no porto 1, deve fazer-se,

```
MOV P1,#0FFH
MOV A,P1
```

Caso se pretenda ler o bit $P3.2$ para o bit C do registo PSW, deve fazer-se,

```
SETB P3.2
MOV C,P3.2
```

Após uma acção de reinicialização, o conteúdo de cada registo de portos de E/S é FFH, pelo que o comportamento por defeito é o poder realizar leituras dos pinos. Assim, caso dado porto seja somente de leitura numa dada aplicação, então não há necessidade de repetir o comando `MOV Px,#0FFH` (para o caso de bytes) ou `SETB Px.y` (para o caso de bits) de cada vez que se pretende fazer uma leitura.

O porto 0 é verdadeiramente bidireccional, dado que quando se atribui o valor lógico 1 a um bit, o pino correspondente fica num estado flutuante;

diz-se que é um pino em dreno aberto. Como tal, deve ser a electrónica externa ao microcontrolador a determinar o valor de tensão. Por outro lado, nos restantes portos tem-se sempre um nível de tensão bem determinado. O próprio microcontrolador coloca o valor lógico 1 no pino através de uma resistência de *pull-up* ligada ao dreno do transístor. Por esta razão, estes portos são apelidados de quase bidireccionais. O porto 0 consegue absorver ou suprir a carga de oito portas TTL, ao passo que os restantes portos podem ser ligados a somente quatro portas TTL.

Com base nas figuras 7.1 e 7.2, pode levantar-se a questão acerca do que é lido numa instrução de leitura; se o valor armazenado no *lacth* ou o valor presente no pino. Uma instrução de leitura lê o valor armazenado no pino.

Apesar de as operações de leitura e escrita mencionarem sempre o registo SFR, internamente têm-se operações naturalmente diferentes. Estas são devidamente coordenadas pelo *hardware* associado ao porto de modo inteiramente transparente para o utilizador.

7.1.1 Instruções ler-modificar-escrever

Mencionou-se anteriormente que todas as instruções de leitura de um porto lêm o pino e não o *latch* associado. Existe um comportamento de excepção representado pelas instruções do tipo ler-modificar-escrever. Nestas é lido o *latch* e não o pino. Por exemplo, na instrução `INC P3`, que permite incrementar o conteúdo do registo associado ao porto 3, é lido o valor do registo, o conteúdo é incrementado e o novo valor é reescrito em $P3$. A instrução INC, quando referida a um porto de E/S, é uma instrução do tipo ler-modificar-escrever. Deve ser claro que as três operações são automaticamente desempenhadas pelo *hardware*; não havendo intervenção do programador. As instruções deste tipo são:

- as instruções ANL, ORL e XRL que tenham por destino um porto;
- as instruções JBC e CPL, quando o operador de destino é um bit de um porto;
- as instruções INC, DEC e DJNZ, quando o primeiro argumento é um porto;
- as instruções MOV, CLR e SETB, quando o destino é um bit de um porto.

Registo	**Endereço**
P0	80H
P1	90H
P2	A0H
P3	B0H

Tabela 7.1: Endereços dos registos associados aos portos de E/S.

80H	80H	81H	82H	83H	84H	85H	86H	87H
P0	P0.0	P0.1	P0.2	P0.3	P0.4	P0.5	P0.6	P0.7

Tabela 7.2: Configuração do Porto 0.

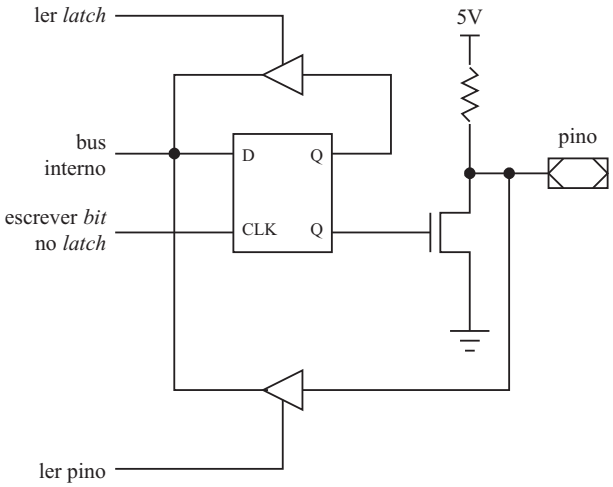

Figura 7.1: Esquema da electrónica simplificado existente em $P1$, $P2$ e $P3$.

90H	90H	91H	92H	93H	94H	95H	96H	97H
P1	P1.0	P1.1	P1.2	P1.3	P1.4	P1.5	P1.6	P1.7

Tabela 7.3: Configuração do Porto 1.

A0H	A0H	A1H	A2H	A3H	A4H	A5H	A6H	A7H
P2	P2.0	P2.1	P2.2	P2.3	P2.4	P2.5	P2.6	P2.7

Tabela 7.4: Configuração do Porto 2.

7.2 Funções especiais

Uma característica fundamental do microcontrolador 8051 diz respeito à dupla função atribuída aos portos $P0$, $P2$ e $P3$. Com efeito, já foi visto que o porto $P0$, para além de ser um porto de E/S, permite o transporte da parte menos significativa do endereço de 16 *bits* multiplexado com o transporte do *byte* de dados. Relembre-se que o uso de uma função impede o uso da restante. De modo semelhante, o porto $P2$ pode ser utilizado como porto de E/S ou pode emitir a parte alta do endereço de 16 *bits*. O facto do programa estar armazenado em memória externa determina automaticamente a não utilização de $P0$ e $P2$ para efeitos de E/S.

O porto $P3$ tem também associadas duas funções a cada pino. Como se pode observar na tabela 7.6, a função alternativa está associada a: (i) utilização dos dispositivos internos; como é o caso dos contadores e do canal de comunicação série; (ii) utilização de interrupções e (iii) acesso à memória de dados externa. Para se fazer uso da função alternativa é necessário escrever o valor lógico 1 no bit respectivo.

Note-se que ao utilizar-se uma determinada função especial, está automaticamente a determinar-se a utilização dos pinos correspondentes no porto $P3$ para o uso específico correspondente, pelo que, naturalmente, ficam indisponíveis para funções de E/S. Por outro lado, as funções especiais são independentes entre si, o que significa que, caso seja desejável, os restantes pinos de $P3$ podem ser utilizados para E/S. Com efeito, é o próprio microcontrolador que faz a comutação entre as utilizações como E/S ou como funções especiais, tão logo se faça o uso de um dos dispositivos associados a estas funções.

B0H	B0H	B1H	B2H	B3H	B4H	B5H	B6H	B7H
P3	P3.0	P3.1	P3.2	P3.3	P3.4	P3.5	P3.6	P3.7

Tabela 7.5: Configuração do Porto 3.

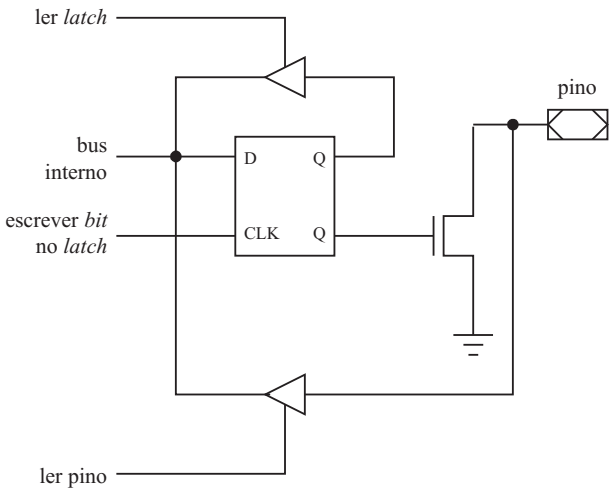

Figura 7.2: Esquema da electrónica simplificado existente em $P0$, em que a diferença para $P1$, $P2$ e $P3$ está na resistência de *pull-up* interna.

Pino	**Função Alternativa**
$P3.0$	entrada do canal série (RXD)
$P3.1$	saída do canal série (TXD)
$P3.2$	interrupção externa 0 ($\overline{INT}0$)
$P3.3$	interrupção externa 1 ($\overline{INT}1$)
$P3.4$	entrada do contador 0 ($C/\overline{T}0$)
$P3.5$	entrada do contador 1 ($C/\overline{T}1$)
$P3.6$	sinal de escrita da memória de programa externa ($\overline{WR}$)
$P3.7$	sinal de leitura da memória de programa externa ($\overline{RD}$)

Tabela 7.6: Funções alternativas do porto 3.

Capítulo 8

Contadores/Temporizadores

0 80C51 possui dois dispositivos contadores (designados por C/$\overline{T}$0 e C/$\overline{T}$1) que funcionam independentemente entre si e do resto do microcontrolador. Este tipo de dispositivo é muito útil no projecto de sistemas digitais, dado permitir a contagem de acontecimentos exteriores (por exemplo, o número de peças numa passadeira rolante existente num processo de fabrico) e medição de temporizações (por exemplo, desencadear uma dada acção ao fim de um determinado intervalo de tempo ou gerar a taxa de emissão/recepção do canal de comunicação série).

No caso de sistemas digitais baseados em microprocessador, as acções de contagem e de temporização são tipicamente desempenhadas por instruções de programa. Esta via apresenta as desvantagens de obrigar a que o processador esteja totalmente dedicado a essas acções e a algum esforço suplementar por parte do programador, caso se deseje garantir a temporização correcta. Alternativamente, estas acções poderão ser executadas por circuitos integrados contadores ligados ao processador. Neste caso, as desvantagens que se apontam prendem-se com o encarecimento do projecto, devido ao aumento da área de placa de circuito impresso e ao aumento do número de circuitos integrados. Por outro lado, o número de avarias tenderá a ser maior, dado que a quantidade de avarias é proporcional ao número de circuitos integrados presentes e às temperaturas desenvolvidas durante o funcionamento. Assim, a existência de contadores internos ao processador reúne as seguintes vantagens:

- Paralelismo - o processo de contagem decorre independentemente da execução do programa.

- Flexibilidade - a configuração dos C/$\overline{T}$ é feita através do *assembly*, utilizando dois registos existentes no espaço SFR. Adicionalmente, um dado C/$\overline{T}$ pode funcionar como contador ou como temporizador em intervalos de tempo disjuntos.

- Redução de recursos - o projecto encontra-se mais perto da solução *single-chip*, reduzindo-se o número de circuitos integrados por área da placa de circuito impresso.

As características dos C/$\overline{T}$ existentes no 80C51 podem ser resumidas da seguinte maneira:

- A contagem é crescente de 8, 13 ou 16b.

- O valor inicial da contagem é ajustável.

- Pode ser desencadeada uma interrupção do programa após o fim da contagem.

- A frequência máxima da contagem é 1/12 da frequência de oscilação. Caso se utilize um cristal de 12MHz, ter-se-á um ritmo de contagem de 1MHz.

- Existem quatro modos de contagem.

- O ritmo da contagem pode ser determinado por um sinal exterior (operação como contador) ou por um sinal interno (operação como temporizador).

8.1 Registos de Contagem - TH0 (8CH), TL0 (8AH), TH1 (8DH) e TL1 (8BH)

Cada C/$\overline{T}$ tem associados dois registos onde é feita a contagem, totalizando 16b. Os nomes dos pares de registos e seus endereços no espaço SFR são apresentados na tabela 8.1. Dado que os dois C/$\overline{T}$ são iguais, a explicação dos modos de operação e respectiva configuração é feita para um só, recorrendo à seguinte notação: THn e TLn denotam os registos associados aos *bytes* mais significativo e menos significativo do registo de 16b associado ao C/$\overline{T}$n, respectivamente, em que n=0 ou n=1.

C/$\overline{T}$	**Registo**	**Endereço**
C/$\overline{T}$0	TH0	8CH
	TL0	8AH
C/$\overline{T}$1	TH1	8DH
	TL1	8BH

Tabela 8.1: Registos de contagem e seus endereços.

O valor de início da contagem é escrito em THn e TLn, anteriormente à ordem de início da contagem. O valor máximo atingido depende do modo de contagem (a apresentar posteriormente). No estado seguinte ao estado a que corresponde o valor máximo da contagem dá-se a situação de sobrecarga[1], podendo ser gerado um pedido de interrupção. Neste instante, o fluxo de execução do programa poderá então passar para a subrotina de atendimento, caso se tenha determinado previamente que a interrupção associada ao C/$\overline{T}$n está activa. Esta, dentro de outras acções, pode ou não parar o processo de contagem e/ou recarregar novos valores nos registos THn e TLn. Para atribuir um dado valor aos registos de contagem, pode utilizar-se qualquer instrução de movimentação de dados, por exemplo, MOV TH0,#10H.

8.2 Registos de Configuração

A configuração de cada C/$\overline{T}$ faz-se por intermédio da escrita de valores em determinados registos do espaço SFR. Nas próximas secções do texto são apresentadas descrições de cada registo.

TMOD (89H) - *Timer Mode Register*

Como se pode observar na tabela 8.2, este registo está dividido em duas partes. A metade à esquerda permite configurar o C/$\overline{T}$1 e a metade à direita permite configurar o C/$\overline{T}$0. O significado de cada *bit* é o seguinte:

GATEn Permite determinar o modo como o C/$\overline{T}$n é iniciado ou parado. Quando se atribuir *GATEn*=0, o início ou a paragem da contagem é determinado pelo *bit* TRn do registo TCON. Quando se atribuir GATEn=1, o C/$\overline{T}$n inicia a contagem quando o *bit* TRn do registo

[1] *overflow* na literatura anglo-saxónica

Bit	GATE1	C/T1	M1.1	M1.0	GATE0	C/T0	M0.1	M0.0
Ordem	7	6	5	4	3	2	1	0

Tabela 8.2: *Bits* do registo TMOD.

M1	**M0**	**Modo**
0	0	contagem a 13b
0	1	contagem a 16b
1	0	contagem a 8b com recarga automática
1	1	modo de expansão

Tabela 8.3: Modos de contagem

TCON e o pino $\overline{INT}$n estiverem ambos no nível lógico '1'. A contagem é parada quando pelo menos um deles for '0'. Este aspecto encontra-se ilustrado nas figuras 8.1, 8.2 e 8.3.

Este *bit* é útil para contar a largura do nível superior de uma onda quadrada TTL (*duty cycle*) proveniente de um sinal externo. Para isso aplica-se o sinal no pino $\overline{INT}$n e utiliza-se o C/$\overline{T}$n como temporizador. A contagem é iniciada logo que exista uma transição 0→1 no pino $\overline{INT}$n, desde que TRn e GATEn estejam a '1'. A contagem pára quando se verificar a transição 1→0 no pino $\overline{INT}$n. Nessa altura, o valor existente no par de registos THn e TLn é proporcional à largura do referido patamar[2].

C/$\overline{T}$n Permite determinar se a função do dispositivo é a de contagem (=1) ou de temporização (=0).

Mn.0, Mn.1 Estes *bits* determinam o modo da contagem segundo a tabela 8.3.

TCON (88H) - *Timer Control Register*

O conteúdo deste registo é apresentado na tabela 8.4. O significado de cada *bit* é o seguinte:

[2]A largura do patamar que se pretende medir não poderá ser arbitrariamente grande, apresentando um limite superior. Este limite pode ser tornado maior recorrendo a esquemas de programação.

TRn Permite iniciar ou parar a contagem no C/$\overline{T}$n. Caso se tenha GATEn=0, a atribuição TRn=1 permite (re)começar a contagem, ao passo que TRn=0 permite pará-la.

TFn Quando for atingido o fim da contagem (estado seguinte ao que corresponde o valor máximo no registo da contagem), o microcontrolador coloca automaticamente TFn=1 para assinalar o pedido de interrupção. Após a execução da rotina de atendimento, o microcontrolador coloca automaticamente TFn=0.

Por exemplo, suponha-se que se pretende activar o C/$\overline{T}$1 e desligar o C/$\overline{T}$0. As instruções *assembly* necessárias seriam:

$$\textit{SETB 94H} \tag{8.1}$$
$$\textit{CLR 92H} \tag{8.2}$$

em que 94H = 88H (endereço base de TCON) + 6H (ordem do *bit* a modificar) e 92H = 88H + 4. Relembre-se que os *assemblers* permitem o uso das seguintes expressões totalmente equivalentes e naturalmente de significado mais claro,

$$\textit{SETB TR1} \tag{8.3}$$
$$\textit{CLR TR0} \tag{8.4}$$

Os *bits* de ordem zero a três referem-se a interrupções, por isso são aqui encobertos. As interrupções são analisadas em capítulo próprio.

Note-se que o processo de contagem não pára após a sobrecarga da contagem e respectiva geração do pedido de atendimento. Garante-se assim uma verdadeira independência do microcontrolador relativamente aos processos de contagem. Caso haja necessidade de parar a contagem, recorre-se, como anteriormente afirmado, ao *bit* TRn.

8.3 Modos de Contagem

8.3.1 Modo '0' - contador de 13 *bits*

Nos oito *bits* de THn e nos cinco *bits* menos significativos de TLn é colocado o valor inicial da contagem, perfazendo 13b. O C/$\overline{T}$n inicia a contagem caso

Bit	TF1	TR1	TF0	TR0	-	-	-	-
Ordem	7	6	5	4	3	2	1	0
Endereço	8F	8E	8D	8C	8B	8A	89	88

Tabela 8.4: *Bits* do registo TCON.

TRn=1 e GATEn=0 ou TRn=1, GATEn=1 e INTn=1, ao ritmo do relógio interno (caso C/$\overline{T}$n=0) ou ao ritmo de um relógio externo (caso C/$\overline{T}$n=1). Note-se que ao serem considerados somente os cinco *bits* menos significativos de TLn, a escrita/leitura dos restantes *bits* não tem qualquer significado, *i.e.*, é inconsequente mas inóquo.

Quando no par de registos THn:TLn se verificar a transição do valor máximo (2000H[3]) para o valor mínimo (0000H), o *bit* TFn comuta de '0' para '1', sinalizando o pedido de interrupção. Relembre-se que a contagem é crescente, pelo que se se pretender contar B unidades, o valor a escrever no par de registos THn:TLn é 2000H-*B*.

A figura 8.1 apresenta um esquema ilustrativo do modo '0' de contagem.

8.3.2 Modo '1' - contador de 16 *bits*

Este modo, igualmente ilustrado na figura 8.1, é semelhante ao anterior. Com efeito, os registos THn e TLn são totalmente aproveitados para proporcionar um contador de 16b, podendo contar-se desde 0000H até FFFFH=65535. Como em todos os modos de contagem, logo após a sobrecarga do contador dá-se o pedido de atendimento da interrupção associada ao C/$\overline{T}$n, que, de entre outras acções, poderá modificar o valor inicial da contagem.

8.3.3 Modo '2' - contagem a oito *bits* com recarga automática

Em THn é escrito o valor inicial da contagem e em TLn é colocado o valor da recarga. Logo que exista sobrecarga em THn (passagem de FFH para 00H), é gerado o pedido de interrupção associado ao C/$\overline{T}$n (através da atribuição TFn=1) e o valor de TLn é automaticamente copiado para THn, prosseguindo a contagem a partir deste valor. Relembre-se que a contagem

[3]Porque existem 13b destinados à contagem, o valor máximo permitido neste modo é de $2^{8+5}=2^{13}=8192=2000$H.

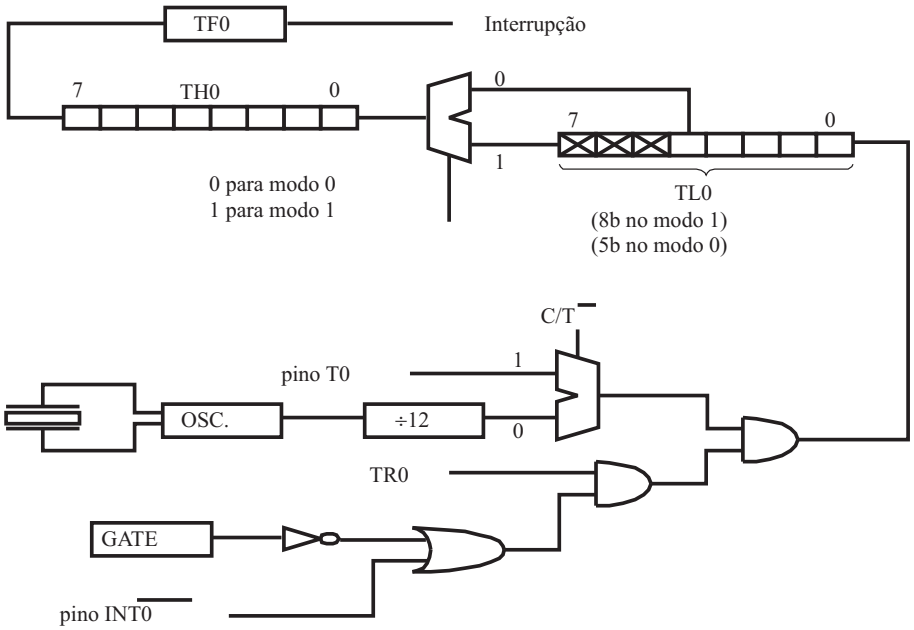

Figura 8.1: Diagrama ilustrativo dos modos '0' e '1' de contagem.

não pára quando se dá a sobrecarga do contador. O modo de operação correspondente está ilustrado na figura 8.2.

8.3.4 Modo '3' - modo de expansão

Trata-se do modo mais singular. Em primeiro lugar salienta-se que só o C/$\overline{T}$0 funciona neste modo. Ou seja, o C/$\overline{T}$1 só poderá funcionar nos modos '0', '1' e '2'. A contagem neste contador pára quando for configurado para o modo '3'.

Relativamente ao C/$\overline{T}$0 tem-se o seguinte esquema. Os registos TH0 e TL0 transformam-se em dois contadores de oito *bits* cada. Seja C1 o contador associado ao registo TH0 e seja C0 o contador associado ao registo TL0. O contador C1 é comandado pelos *bits* TR1, GATE1 e C/$\overline{T}$1, presentes nos registos TCON e TMOD, ao passo que o contador C0 é comandado pelos *bits* TR0, GATE0 e C/$\overline{T}$0. O pedido de interrupção gerado por C1 é assinalado no *bit* TF1, ao passo que o *bit* TF0 está associado a C0. O contador C1 só pode funcionar como temporizador e C0 pode funcionar como contador ou temporizador. Este modo de operação está ilustrado na figura 8.3

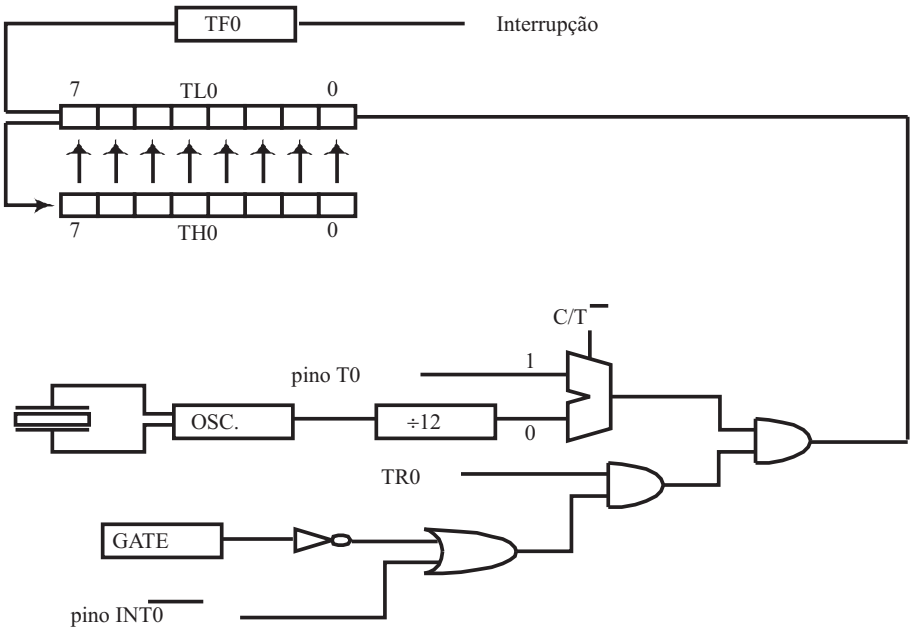

Figura 8.2: Diagrama ilustrativo do modo '2' de contagem.

Foi visto que o contador C/$\overline{T}$1 não funciona no modo '3', podendo faze-lo em qualquer um dos restantes, não gerando, porém, pedido de interrupção. Assim, com o modo '3', os projectistas dos microcontroladores da família MCS51 pretenderam criar três contadores, ainda que com potencialidades limitadas. Uma utilização típica deste microcontrolador corresponde a usar o C/$\overline{T}$1 como gerador da taxa de emissão/recepção[4] para o canal série, ao passo que o C/$\overline{T}$0, podendo ser desbobrado em dois contadores, servirá para as restantes tarefas de contagem e/ou de temporização, conforme o anteriormente exposto.

8.4 Geração de Uma Onda Quadrada

Neste exemplo ilustrativo das capacidades e da operação dos C/$\overline{T}$'s, pretende-se gerar uma onda quadrada de 10KHz no pino P1.0. Note-se que o período correspondente a esta frequência é de 100μs, pelo que pretende-se alternar o nível lógico em P1.0 de 50μs em 50μs.

[4] *baud rate* na literatura anglo-saxónica.

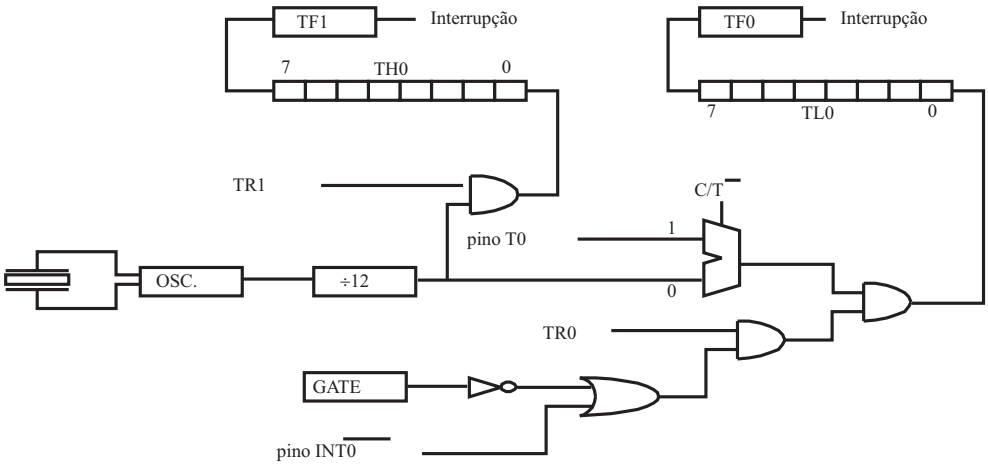

Figura 8.3: Diagrama ilustrativo do modo '3' de contagem.

O problema pode ser abordado por via de *poling* sobre o *bit* TF0 ou através da interrupção associada ao C/$\overline{T}$0. Opta-se pela primeira via. O código *assembly* que permite realizar o objectivo é apresentado de seguida.

```
        MOV TMOD,#02H   ; C/T0 em modo '2'
        MOV TH0,#-50    ; valor da recarga
        SETB TR0        ; iniciar contagem
LOOP:   JNB TF0,LOOP    ; foi atingido o máximo?
        CLR TF0         ; limpar TF0
        CPL P1.0        ; complementar o pino
        SJMP LOOP       ; mais meio período
```

Relativamente ao código apresentado, eis alguns esclarecimentos obrigatórios. Certos *assemblers* permitem a menção de constantes em diversas bases de numeração. A constante -50 está em base decimal, dado não ter nenhum sufixo, ou seja, por defeito é assumida a base decimal. A instrução JNB TF0,LOOP, correspondente a um salto condicional para a própria linha, não representa nenhum ciclo infinito. Com efeito, após a sobrecarga do C/$\overline{T}$0, o bit TF0 transita para '1', tornando a condição falsa. Finalmente, apesar da última instrução implementar um salto do tipo relativo, não foi feito nenhum cálculo para determinação do endereço de salto. Mais uma vez aproveitam-se certas características dos *assemblers* favoráveis para os programadores, como é, neste caso, o uso de etiquetas.

8.5 Qual é a Frequência Máxima Possível de Gerar num 80C51?

A questão pode ser analisada por duas vias: recorrendo somente a *software* ou através de um dos contadores.

Na primeira abordagem, a solução seria:

```
LOOP:  CPL P1.0
       SJMP LOOP
```

Até que o nível no pino seja complementado são executadas duas instruções. Assumindo que o microcontrolador funciona com um cristal de 12MHz, o tempo de execução de CPL é de 1μs e para SJMP é de 2μs. Assim, de 1+2=3μs em 3μs o nível no pino é complementado, fazendo com que T/2=3μs, ou seja, f=166Khz.

Na abordagem por via dos temporizadores, baseada no código da secção anterior, o valor máximo a carregar em C/$\overline{T}$0 é 255, que leva a uma frequência de 83KHz, proveniente do seguinte cálculo:

$$\begin{aligned} T &= 2(t_{JNB}+t_{CLR}+t_{CPL}+t_{SJMP})=2(2+1+1+2)=12\mu s \\ f &= 83KHz \end{aligned}$$

Capítulo 9

Canal Série do 80C51

O microcontrolador 80C51 está equipado com um canal série que permite efectuar comunicações segundo a norma RS-232, entre outras. O canal série é comummente denominado UART (*Universal Asynchronous Receiver/Transmitter*), significando que implementa comunição assíncrona, *i.e.*, os dados são enviados sem a presença de um sinal de sincronismo; este recuperado pelo receptor.

Uma UART permite enviar e receber dados de oito *bit* em série. No caso da emissão é composta uma sequência de *bit* cujo formato é o seguinte: um bit de início[1], uma sequrepresentando o dado propriamente dito, um *bit* programável (opcional) e um *bit de fim*[2]. Este conjunto de *bit* é habitualmente denominado trama[3](*vide* figura 9.1). Após esta composição, a UART envia cada *bit* pelo canal de comunicação. Observe-se que o *bit de início* é definido por uma transição 1→0, ao passo que o *bit de fim* garante que a linha fique no estado inicial (chanado de repouso), pelo que pode haver ou não uma transição.

9.1 Canal série

A porta série do 80C51 pode ser utilizada para expansão do número de entradas/saídas ou como UART. Eis um resumo das características:

- Expansão de E/S de oito *bit.*

[1]tradução do termo anglo-saxónico start bit.

[2]Tradução do termo anglo-saxónico *stop bit.*

[3]A palavra *trama* é um neologismo que surge da tradução do termo anglo-saxónico *frame.*

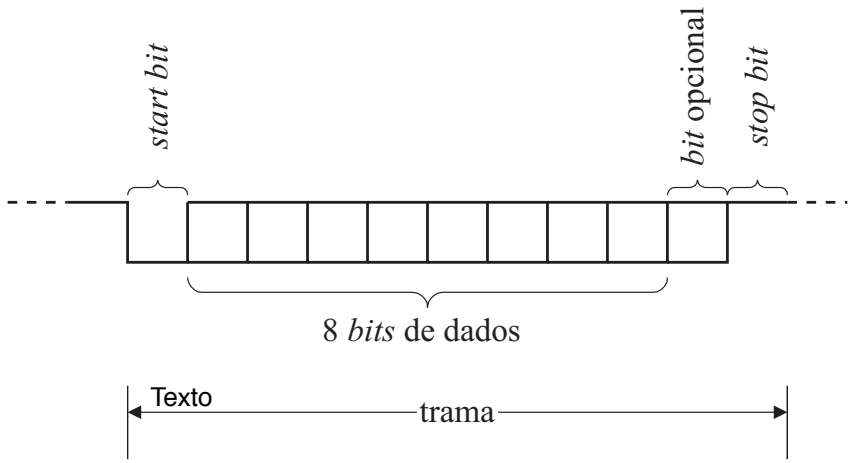

Figura 9.1: Configuração de uma trama RS232.

- UART.
- Circuitos de transmissão e recepção independentes para comunicação *duplex*[4].
- Selecção do tamanho da trama: 10 ou 11 *bit*.
- Taxa de emissão/recepção[5] máxima de 1 Mhz, para $f_{osc} = 12$ MHz.
- Escolha de taxas diferentes para a emissão e recepção (em algumas versões).
- Capacidade de comunicação entre vários microcontroladores 80C51.
- Detecção de erros *overrun* e *framing*[6].
- Existem quatro modos de operação, segundo a tabela 9.1.

.

[4]Termo anglo-saxónico que se refere à capacidade de enviar e receber informação simultaneammente, i.e., bidireccional.

[5]*baud rate* na denominação anglo-saxónica.

[6]Estes termos serão definidos posteriormente neste capítulo.

Modo	**Taxa**	**Dado**	**9° *bit***
0	$12\times t_{clk}$	8 *bit*	-
1	taxa de sobrecarga no $C/\overline{T}1$	8 *bit*	-
2	$32\times t_{clk}$ ou $64\times t_{clk}$	9 *bit*	0, 1 ou paridade
3	taxa de sobrecarga no $C/\overline{T}1$	9 *bit*	0, 1 ou paridade

Tabela 9.1: Características dos modos de funcionamento.

9.2 Registos de configuração

A configuração da porta série passa pela escrita de valores em registos pertencentes ao espaço SFR. Após esta tarefa, para se enviar um byte basta escreve-lo no registo SBUF, e para receber um byte basta lê-lo do mesmo registo após o bit RI transitar para '1'. Os processos de emissão e recepção podem ser coordenados por rotinas de atendimento de interrupções, que são executadas após o fim da emissão (flag TI transita para '1') ou da recepção (flag RI transita para 1), ou através de *poling*. Procede-se, de seguida, à explicação dos registos envolvidos.

9.2.1 SBUF (99H) - Registo de escrita/leitura

É neste registo que se escreve o dado a ser enviado (iniciando-se assim o processo de envio) e que se lê o dado recebido. Fisicamente existem dois registos internos, que separam o dado recebido do dado a ser enviado (*vide* figura 9.2), embora este facto seja transparente para o programador. O registo apropriado é seleccionado pelo próprio microcontrolador conforme o tipo de instrução: escrita ou leitura. Esta característica está subjacente à possibilidade de enviar e receber dados simultaneamente.

9.2.2 PCON (87H) - Duplicação da taxa

O registo PCON permite configurar o 80C51 relativamente ao modo de adormecimento e relativamente ao canal série. Nesta secção só se aborda a última funcionalidade. O *bit* SMOD (*bit* sete de PCON) está associado à duplicação da referida taxa (ver modos de operação na secção 9.3). Note-se que PCON não é endereçável por *bit*.

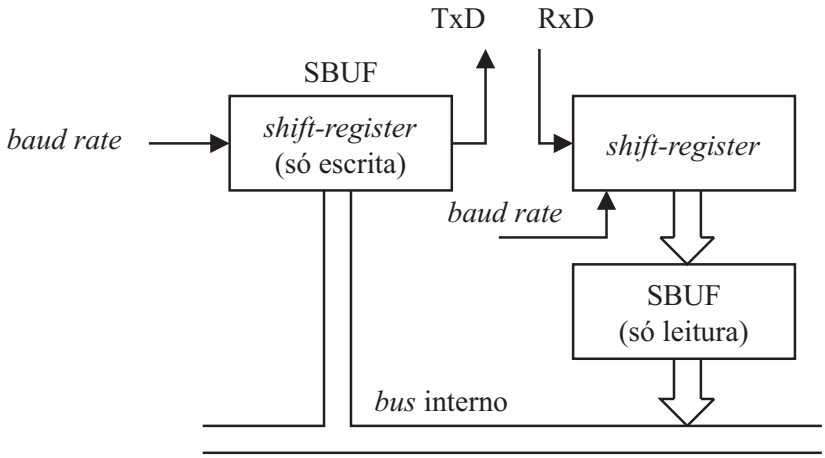

Figura 9.2: Diagrama de blocos ilustrativo da separação existente entre os registos de emissão e de recepção, de modo a implementar comunicação *full-duplex*.

9.2.3 SCON (98H) - Configuração da UART

A organização interna do registo SCON está representada na tabela 9.2. O significado de cada *bit* é o seguinte:

SM1, SM0: Permitem escolher um de quatro modos de operação, de acordo com a tabela 9.3, onde *fosc* representa a frequência de oscilação do microcontrolador.

REN: Activa (REN=1) ou inibe (REN=0) a recepção de dados.

TB8: *Bit* programável (10° *bit*) da trama a enviar, para o caso dos modos '2' e '3', cuja atribuição é da responsabilidade do programador.

RB8: Local onde é armazenado o *bit* programável (10° *bit*) da trama recebida, para o caso dos modos '2' e '3'. No modo '1' é armazenado o *bit de fim*.

TI: *Bit* de sinalização do fim da emissão. Após o envio do último *bit* é gerado automaticamente o pedido de interrupção, através da transição TI=0→1. No fim da rotina de atendimento da interrupção o programador deve atribuir TI=0, de outro modo não é possível detectar-se a recepção de outra trama.

*B*it	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
Ordem	7	6	5	4	3	2	1	0
Endereço	9FH	9EH	9DH	9CH	9BH	9AH	99H	98H

Tabela 9.2: Organização do registo SCON.

SM0	**SM1**	**Modo**	**Descrição**	**Taxa**
0	0	0	registo de deslocamento	$f_{osc}/12$
0	1	1	UART de 8 *bit*	variável
1	0	2	UART de 9 *bit*	$f_{osc}/64$ ou $f_{osc}/32$
1	1	3	UART de 9 *bit*	variável

Tabela 9.3: Selecção e características dos modos existentes.

RI: *Bit* de sinalização de fim da recepção. O significado é semelhante ao *bit* TI.

SM2: Este *bit* deve ser colocado a '0' no modo '0'. Se se tiver SM2=1 no modo '1' e se o *bit de fim* recebido for '0' (*stop bit* inválido), então os dados recebidos são ignorados, ou seja, SBUF, RB8 e RI não são alterados. Se SM2=0 o *byte* é recebido independentemente do estado do *bit de fim*. Nos modos '2' e '3' este *bit* possibilita a comunicação entre vários 80C51. Esta funcionalidade será posteriormente esclarecida.

9.3 Modos de operação

O canal série do 80C51 pode operar em um de quatro modos. Já foi afirmado que a selecção do modo é feita por intermédio dos *bit* SM0 e SM1 existentes no registo SCON. Nas próximas secções procede-se à explicação das características de cada modo.

9.3.1 Modo '0' - registo de deslocamento

Este modo permite expandir as capacidades de E/S, acrescentando um quinto porto bidireccional de oito *bit* por intermédio de uma interface série, oposta-

mente aos portos P0, P1, P2 e P3 que apresentam interfaces do tipo paralela. O envio e a recepção dos caracteres de oito *bit* faz-se pelo pino RxD, ao ritmo do sinal de sincronismo presente no pino TxD. O *bit* menos significativo é o primeiro a ser enviado e o primeiro a ser recebido. Este modo funciona a uma frequência fixa de 1/12 da frequência de oscilação a que o microcontrolador é alimentado. Na figura 9.3 são apresentados os diagramas temporais correspondentes à emissão e recepção.

Nas figuras 9.4 e 9.5 são apresentados esquemas de circuitos que permitem implementar um quinto porto de entrada e de saída, respectivamente[7]. Como se pode observar, as interfaces série permitem diminuir o número de pistas da placa de circuito impresso, reduzindoassim a complexidade do seu desenho. Note-se, contudo, que são gastos oito ciclos máquina para ler ou escrever um *byte*, ou seja, a leitura e escrita de valores é (no mínimo[8]) oito vezes mais lenta que aquela verificada nos portos P0, P1, P2 e P3. Ainda assim, é possível ler/escrever um *byte* em 8μs (ver nota anterior), caso $f_{osc} = 12$MHz. Dependendo da aplicação, este intervalo de tempo poderá ser satisfatório.

O programa que permite configurar o 80C51 para o modo '0' é o seguinte:

```
CLR 9FH   ; bit SM0=0
CLR 9EH   ; bit SM1=0
CLR 9DH   ; bit SM2=0
```

ou, alternativamente, de maneira mais explícita:

```
CLR SM0
CLR SM1
CLR SM2
```

Relativamente às instruções anteriores compete afirmar o seguinte:

- O *bit* SMOD do registo PCON não é utilizado dado a frequência ser fixa.
- Os *bit* TB8 e RB8 não têm aplicação no modo '0'.
- Os *bit* TI e RI não fazem parte da configuração mas sim da operação do canal série.

[7] A derivação do circuito que permita implementar um porto bidirecional é deixada como exercício.

[8] Não se considera o tempo gasto pelas instruções necessárias ao tratamentos dos dados a enviar ou a receber.

- O *bit* REN deve ser colocado a '1' caso se deseje receber um *byte*, o que pode acontecer diversas vezes no mesmo programa, pelo que poderá não fazer parte da configuração.

O controlo da emissão/recepção pode ser abordado de duas maneiras: por intermédio de *poling* (é o programa principal que se encarrega de observar o valor dos *bit* TI e RI) ou utilizando a interrupção disponibilizada para o efeito. Para o primeiro caso têm-se:

Caso da emissão: Assume-se que se pretende enviar o conteúdo de oito *bytes*, por exemplo, endereços 20H a 27H da memória. Para o efeito utiliza-se o registo R0 como apontador para essa região de memória. O código é o seguinte:

Linha	**Etiqueta**	**Instrução**	**Comentário**
1		MOV IE,#00H	
2		MOV R0,#20H	; apontar para o 1° valor
3	salto:	MOV SBUF,@R0	
4		JNB TI,$	; testar se TI=0
5		CLR TI	
6		INC R0	; próximo *byte*
7		CJNE R0,#28H,salto	; repetir o envio

Na primeira linha inibe-se o atendimento de todas as interrupções, apesar de só ser necessário inibir aquela que diz respeito ao canal série. Na segunda linha prepara-se R0 para apontar para o primeiro valor dos dados a enviar. A terceira linha corresponde à primeira instrução do processo iterativo de envio dos oito *bytes*. Esta linha, por se ter o registo SBUF como destino, representa o envio propriamente dito. Na quarta linha estabelece-se um processo iterativo de espera até que TI esteja a '1', altura em que terminou o envio. Após esta instrução é iniciado o processo de preparação de novo envio, caso ainda não se tenham enviado os oito *byte*começando por repor o valor '0' em TI e apontando para o *byte* seguinte.

Caso da recepção: Assume-se que os oito *byte* recebidos são colocados na memória de dados interna a partir do endereço 20H. Novamente, utiliza-se o registo R0 como apontador para a memória. O código *assembly* é

o seguinte:

```
        MOV IE,#00H
        MOV R0,#20H
outro:  JNB RI,$          ; testar se RI=0
        CLR RI
        MOV @R0,SBUF      ; guardar o valor
        INC R0            ; apontar para o próximo
        SJMP outro
```

Na abordagem por interrupções ter-se-ia:

Caso da emissão: O código correspondente será:

```
0023H:  CJNE R0,#28H,mais_um
        CLR TI
        RETI
mais_um MOV SBUF,@R0
        INC R0
        CLR TI
        RETI
principal: MOV R0,#20H
        MOV IE,#90H           ; interrupção do canal série
        MOV SBUF,@R0          ; 1° valor a enviar
        ...
```

Observe-se que a rotina de atendimento da interrupção é armazenada a partir do endereço 23H da memória de programa. Dado que após um *reset* SM0=0, SM1=0 e SM2=0, não há necessidade de proceder à configuração da porta série para o modo '0'.

Caso da recepção: O código correspondente será:

```
0023H:  CLR RI
        MOV @R0,SBUF
        INC R0
        RETI
principal: MOV R0,#20H
        SETB REN
        MOV IE,#90H       ; interrupção do canal série
        ...
```

Diagrama temporal para a emissão

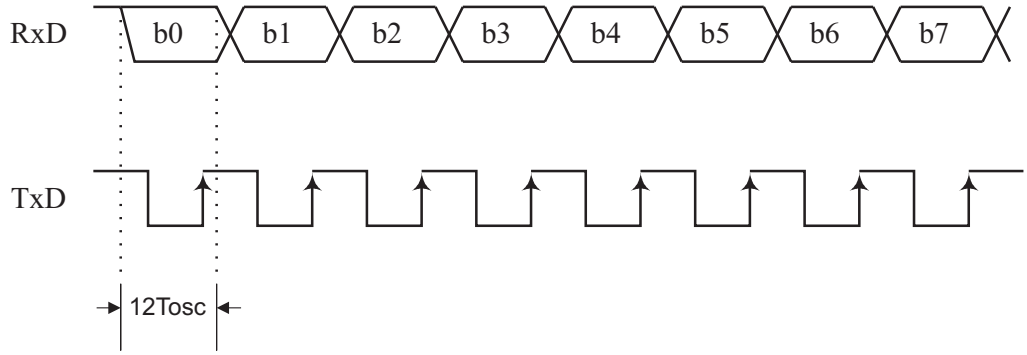

Diagrama temporal para a recepção

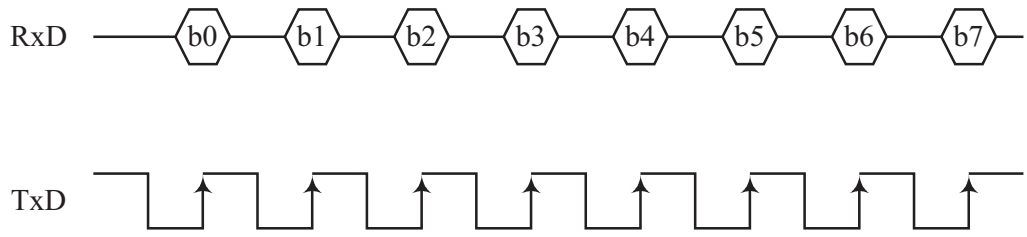

Figura 9.3: Diagramas temporais para a emissão e recepção referentes ao modo '0'.

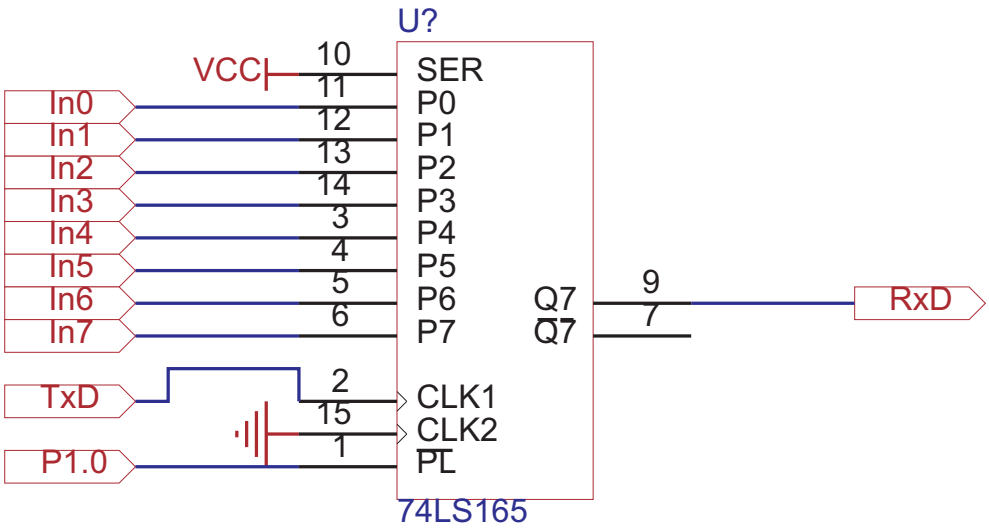

Figura 9.4: Esquemático do circuito que permite implementar um quinto porto de entrada.

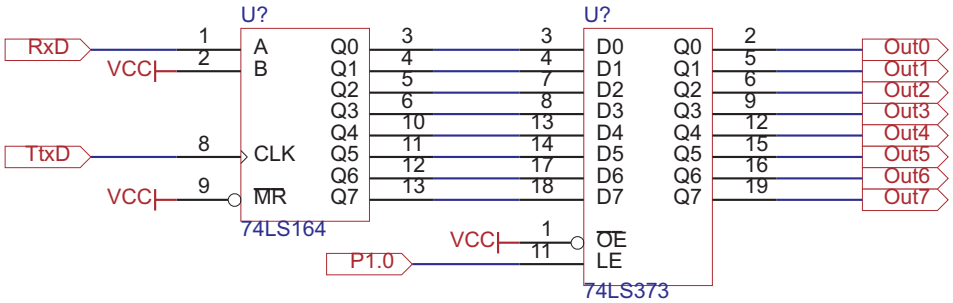

Figura 9.5: Esquemático do circuito que permite implementar um quinto porto de saída.

9.3.2 Modo '1' - UART de 10 bit

Este modo é um dos mais utilizados na comunicação com computadores pessoais. A trama é composta por um *bit de início* (start bit), oito bit de dados e um *bit de fim* (*stop bit*). A taxa de emissão/recepção (B) é determinada através do ritmo da sobrecarga no C/$\bar{\text{T}}$1. O valor de B é dado pela equação 9.1, em que SMOD representa o conteúdo do bit mais significativo de PCON (ver secção 9.2.2), e $f_{C/T1}$ representa a taxa de sobrecarga do C/$\bar{\text{T}}$1. Note-se que, nesta situação, deve inibir-se a interrupção do C/$\bar{\text{T}}$1. Apesar de se poder utilizar qualquer dos modos '0', '1' e '2' de contagem, na situação típica utiliza-se o modo '2', passando o cálculo de B a ser definido pela equação 9.2, em que f_{OSC} representa a frequência do cristal e TH1 representa o valor (em base decimal) que é carregado no registo TH1. A tabela 9.4 apresenta uma lista de taxas (nem todas RS232) em função da frequência de oscilação do cristal, do bit SMOD e da configuração do C/$\bar{\text{T}}$1.

$$B = \frac{2^{SMOD}}{32} \times f_{C/T1} \tag{9.1}$$

$$B = \frac{2^{SMOD}}{32} \times \frac{f_{osc}}{12 \times [256 - (TH1)]} \tag{9.2}$$

Neste modo, o envio dos bit dá-se pelo pino TxD e a recepção pelo pino RxD. O bit menos significativo é o primeiro a ser enviado/recebido.

Se as condições:

1) RI=0
2) SM2=0 ou SM2=1 e *stop bit*=1

forem satisfeitas, então o dado recebido é colocado em SBUF, o *stop bit* recebido é colocado em RB8 e RI é colocado a '1', seguindo-se o pedido de interrupção. Entre várias acções, a rotina de atendimento deverá copiar o valor de SBUF para outra posição de memória e recolocar RI=0, indicando ao microcontrolador que o dado recebido foi processado pelo programa da aplicação.

Caso a primeira condição não seja satisfeita (*i.e.*, RI=1) no final da recepção de um byte, então tem-se um erro de *overrun*, significando que anteriormente foi recebido um byte que não foi reconhecido pelo programa. Todos os *bytes* recebidos posteriormente serão ignorados até que se atribua RI=0. Ou seja, os conteúdos de SBUF, RB8 e RI não são alterados. Isto

que dizer que todos os bytes seguintes ao que levou a RI=1 são perdidos. Note-se ainda que o programa não se apercebe da perda de uma trama, por não haver transição $RI = 0 \rightarrow 1$.

Relativamente à segunda condição, o byte recebido pode ser ou não aceite pelo microcontrolador, dependendo do modo que estiver seleccionado. No caso do modo '1', SM2 pode ser utilizado para rejeitar o byte recebido caso o *stop bit* não seja válido. Assim, se SM2=1, o valor recebido é ignorado caso o *stop bit* seja inválido (*stop bit*=0)[9]. Com SM2=0 o *stop bit* é colocado em RB8 e o valor recebido é colocado em SBUF, independentemente do valor do *stop bit*. Ou seja, com SM2=0 a detecção do erro de *framing* é da responsabilidade do programador.

Na figura 9.6 é apresentado um esquema electrónico que permite implementar comunicações segundo a norma RS-232. De seguida é apresentado o código de um programa que permite enviar a mensagem "HELLO WORLD", onde o controlo da emissão é feito por intermédio de *poling*.

[9]Este tipo de situação é conhecida por erro de *framing* na denominação anglo-saxónica original. Este erro acontece sempre que existir perda de sincronismo entre o emissor e o receptor.

```
            ORG 00H
            ; configuração
            MOV TMOD,#00100000B
            MOV TH1,#0FDH
            MOV TCON,#01000000B
            MOV SCON,#01000000B
            ; programa principal
            MOV R0,#0
            MOV DPTR,HELLO_MSG
OUTRO:      MOV A,R0
            MOVC A,@A+DPTR
            ACALL SEND
            INC R0
            CJNE A,#'#',OUTRO
            AJMP $
HELLO_MSG:  DB 'HELLO WORLD#'       ; # é o terminador da string
            ; rotina de envio (poling)
            MOV SBUF,A
            JNB TI,$
            CLR TI
            RET
            END
```

De seguida apresenta-se o código de outro programa que permite à porta série ecoar todos os caracteres que lhe cheguem. Desta feita, o controlo é desempenhado por interrupções.

```
        CSEG AT 00H
        AJMP INICIO
        CSEG AT 23H
        AJMP SERIE
        ; configuração
INICIO: MOV IE,#90H
        MOV TMOD,#20H
        MOV TH1,#0FDH
        MOV TCON,#40H
        MOV SCON,#50H
        ; programa principal
        SJMP $
        ; rotina da porta série
SERIE:  JB RI,RX
        CLR TI
        RETI
RX:     CLR RI
        MOV A,SBUF
        MOV SBUF,A
        RETI
        END
```

9.3.3 Modo '2'

Neste modo a trama é composta por 11 bit: um *start bit*, oito bit de dados, um bit programável e um *stop bit*. No caso da emissão, o bit programável é recolhido de TB8 (escrito previamente pelo programador), ao passo que na recepção o 10° bit recebido é colocado em RB8. A taxa de emissão/recepção é fixa sendo determinada pela seguinte expressão:

$$B = f_{\text{osc}} \frac{2^{\text{SMOD}}}{64} \tag{9.3}$$

Relativamente aos pinos por onde se faz a emissão e a recepção, são válidos os comentários tecidos para o modo '1'.

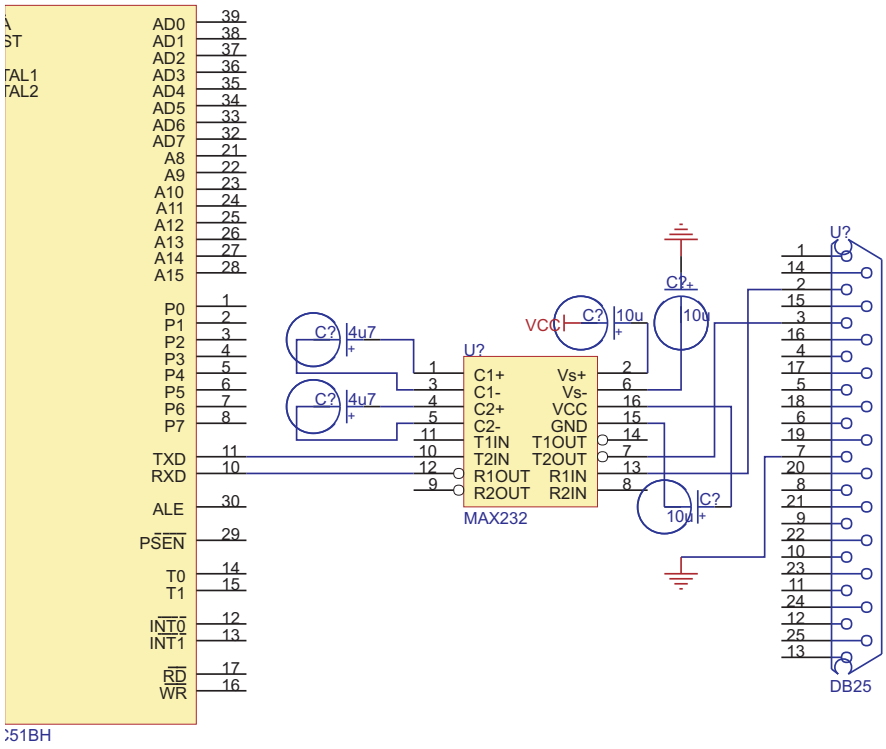

Figura 9.6: Esquemático do circuito que permite implementar comunicação RS232 através dum conector de 25 pinos.

B	f_{osc}	**SMOD**	**C/$\overline{\mathbf{T}}$**	**Modo**	**Recarga**
1MHz (Modo 0)	12MHz	×	×	×	×
375KHz (Modo 2)	12MHz	1	×	×	×
62.5KHz (Modo 1 e 3)	12MHz	1	0	2	FFH
19.2K	11.059MHz	1	0	2	FDH
9.6K	11.059MHz	0	0	2	FDH
4.8K	11.059MHz	0	0	2	FAH
2.4K	11.059MHz	0	0	2	F4H
1.2K	11.059MHz	0	0	2	E8H
300	11.059MHz	0	0	2	A0H
110	6MHz	0	0	2	72H
110	6MHz	0	0	1	FEEBH

Tabela 9.4: Taxas de emissão/recepção geradas pelo C/$\overline{\text{T}}$1, em que a coluna modo refere-se ao modo de contagem.

9.3.4 Modo '3'

Este modo é semelhante ao anterior, com excepção do cálculo da *baud rate* que é feito pelas expressões apresentadas no modo '1'.

9.3.5 Sistemas multiprocessador

Neste exemplo de aplicação é aproveitada a particularidade dos modos '2' e '3' que diz respeito ao bit programável, da seguinte maneira:

> Se o bit programável (o 10° bit) recebido for '1', com SM2=1, então o dado recebido é colocado em SBUF, sendo gerado o pedido de interrupção respectivo.
>
> Se o 10° bit for '0', com SM2=1, então os dados recebidos são ignorados, não sendo gerado pedido de interrupção.

Ou seja, o bit SM2 permite receber o dado em função do estado do bit programável.

Um exemplo de um sistema multiprocessador composto por um mestre e vários escravos é apresentado nesta secção. Neste sistema, somente o escravo tem a capacidade de enviar a informação e de iniciar a comunicação. Os escravos, naturalmente, estão limitados a recebe-la e só respondem ao mestre. Eis o protocolo:

> A cada escravo é atribuído um endereço (único). Inicialmente, cada escravo coloca SM2=1. O mestre envia tramas com TB8=1 sempre que pretenda endereçar um escravo. De modo semelhante, coloca TB8=0 sempre que pretenda enviar dados para o escravo. Quando os escravos recebem uma trama (enviada sempre pelo mestre) fazem uma comparação do dado recebido com o seu endereço. O único escravo endereçado faz SM2=0, com vista a preparar-se para a recepção dos dados. Os restantes escravos mantêm-se com SM2=1. O mestre envia então os dados em tramas com TB8=0. Dado que o SM2 do escravo seleccionado é zero, este recebe e processa os dados. Os restantes ignoram constantemente a informação recebida. Após a transmissão dos dados, o mestre envia um dado código que o escravo reconhece como fim do envio dos dados. Em resposta a este código, o escravo faz SM2=1, ficando, tal como os restantes, em modo de escuta.

O diagrama de blocos do esquema de comunicações mestre-escravo está patente na figura 9.7.

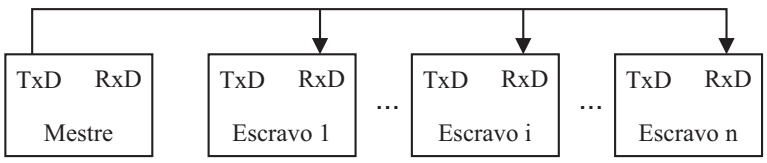

Figura 9.7: Diagrama ilustrativo das ligações entre diversos microcontroladores de modo a implementar um sistema de comunicação entre um mestre e diversos escravos.

Capítulo 10

Interrupções

O processo de interrupção permite suspender o fluxo normal de execução de um dado programa (programa principal e suas rotinas) em resposta a um determinado acontecimento. Este, por definição, apresenta uma prioridade alta, que requer atenção imediata, daí a suspensão do programa. O fluxo de execução é então canalizado para a rotina de atendimento da interrupção, onde está codificada a resposta a dar ao acontecimento. Esta facilidade é relativamente confortável para o programador, que se pode concentrar nas respostas aos acontecimentos possíveis, não necessitando de se preocupar com a determinação da ocorrência do acontecimento.

Considere-se o alarme de uma residência onde são monitorados sinais provenientes de sensores de presença, de humidade, de calor, etc. Cada sensor está relacionado com uma fonte de alarme, que, por seu turno, está associada a um dado comportamento do dispositivo de alarme. O processo de determinação da existência de alarme e da respectiva reacção pode ser feito de duas maneiras.

No primeiro caso tem-se uma abordagem iterativa ou cíclica, *i.e.*, cada sensor é testado para se verificar se está em situação de alarme. Dependendo do valor lido no sensor, poderá ser executada uma rotina de alarme ou passar-se à leitura do próximo sensor na sequência. Este processo continua até que todos os sensores tenham sido testados; um novo ciclo será iniciado.

Em resumo, existe um conjunto de tarefas a executar, sendo cada tarefa executada sequencialmente sem que nenhum tipo de acontecimento a possa interromper. Existem algumas limitações a este tipo de comportamento, nomeadamente, caso seja importante reagir rapidamente às várias situações de alarme ou caso existam situações mais alarmantes que outras. Por exem-

plo, poderá acontecer uma situação de alarme cuja determinação seja tardia, em virtude do teste ao sensor respectivo se encontrar no fim da sequência determinada. Na literatura anglo-saxónica, este tipo de estratégia é vulgarmente conhecida por *poling*.

O segundo caso recorre a interrupções, que, em termos simples, podem ser vistas como a capacidade do processador observar todos os sensores (fontes da interrupção) em simultâneo e a capacidade de executar a rotina associada a um dado sensor (rotina de atendimento) tão logo seja reconhecida a situação de alarme. Dado que é possível associar prioridades à execução de rotinas de atendimento, é possível parar a execução de uma dada rotina para dar lugar à execução de outra mais importante ou prioritária. Este tipo de estratégia é do tipo semi-paralelo, uma vez que, em termos genéricos, permanece a ideia da execução de várias tarefas (o test aos sensores) em simultâneo, quando, efectivamente, tal não acontece. Várias tarefas só são executadas simultaneamente caso existam vários processadores. As limitações apontadas a este tipo de estratégia prendem-se com o número limitado de fontes de interrupção geralmente disponíveis nos processadores, embora existam dispositivos que permitem alargar o número de interrupções; tal é o caso dos PIC - *Programmable Interrupt Controllers*[1].

As rotinas de atendimento de interrupção não fazem parte do programa principal. Destinguem-se das rotinas convencionais pela maneira como são terminadas. As rotinas convencionais terminam, no caso do *assembly* do 80C51, com a instrução RET, que permite voltar à instrução seguinte àquela de chamada. No caso das rotinas de atendimento, a terminação é feita com a instrução RETI.

A partir da inspecção do programa principal é possível determinar a ordem de execução das rotinas convencionais, e, desta maneira, determinar o fluxo de execução de todo o programa. O mesmo não pode ser dito relativamente às rotinas de atendimento de interrrupção, que, devido à sua natureza, não permitem determinar quer no tempo quer no espaço o momento da chamada. Por este motivo, devem ser colocados certos cuidados nas rotinas de atendimento, nomeadamente, é boa prática guardar na memória de *stack* todos os conteúdos de registos e de endereços de memória que sejam usados nas rotinas de atendimento.

A utilização extensiva de interrupções permite implementar o paradigma de programação denominado *event-driven* ou baseado em acontecimentos,

[1]Na verdade, um simples codificador prioritário pode ser usado para este efeito.

no sentido que a execução de qualquer rotina só acontece em resposta a um acontecimento. Um exemplo bem conhecido é o existente nos sistemas operativos baseados em janelas, onde, por exemplo, ao movimento do rato é associada a execução de subrotinas que determinam o que fazer com base na sua posição espacial.

Como foi anteriormente afirmado, o 80C51 é sensível a cinco fontes de interrupção: duas relacionadas com acontecimentos externos ao microcontrolador, duas associadas ao estado dos contadores/temporizadores e uma associada ao canal de transmissão/recepção série. Naturalmente, os microcontroladores melhor apetrechados da família MCS 51 têm disponíveis um maior número de interrupções. Por exemplo, o 87C51GB é composto por um conversor analógico-digital que gera uma interrupção tão logo tenha sido feita uma conversão em qualquer um dos oito canais analógicos existentes.

10.1 Activação de interrupções - global e particular

Cada fonte de interrupção pode ser individualmente activada, *i.e.*, é possível determinar se existe atendimento do pedido de uma dada interrupção. Igualmente, é possível activar na globalidade (*i.e.*, num só passo) todas as interrupções.

A acção de activação/desactivação de interrupções é controlada por intermédio do registo IE(A8H) existente no espaço SFR (ver a tabela 10.1).

O significado de cada bit é apresentado nas secções seguintes.

10.1.1 EA - activação global

Nenhuma interrupção será atendida se EA=0. De outro modo, a possibilidade de haver atendimento de uma dada interrupção está dependente do estado dos bits ES, ET1, ET2, EX0 e EX1.

10.1.2 ES - activação da interrupção associada ao canal série

Existe atendimento da interrupção associada ao canal série se EA=1 e ES=1. Existe pedido de interrupção quando o canal série termina o envio ou a recepção de um byte. O canal série é abordado em capítulo próprio.

10.1.3 ET0, ET1 - activação das interrupções associadas aos contadores 0 e 1, respectivamente

Estes bits determinam a activação ET=1 ou desactivação (ET=0) da interrupção associada ao C/$\bar{T}$ respectivo, desde que se tenha EA=1. Existe pedido de interrupção quando se verificar uma sobrecarga no contador. Os contadores são abordados em capítulo próprio.

10.1.4 EX1, EX0 - activação das interrupções externas

Estes bits determinam a activação (EX=1) ou desactivação (EX=0) das interrupções associadas às fontes externas existentes, caso se tenha EA=1.

Os bits de ordem '5' e '6' do registo IE estão associados a interrupções existentes em outros microcontroladores, por consequência não têm aplicação na versão 80C51. O programador deverá atribuir-lhes o valor '0' para que a migração para um desses microcontroladores não provoque comportamentos indesejados.

Considere-se que se pretende activar as interrupções associadas ao canal série e ao C/$\bar{T}$1. O código respectivo seria,

```
SETB 0AFH
SETB 0ACH
SETB 0ABH
```

em que AFH = A8H (endereço base de IE) + 7 (ordem do bit a modificar). O cálculo anterior é válido dado que os registos SFR que terminam em 0H ou 8H são endereçáveis por bit. Em alternativa poderia fazer-se,

```
MOV 0A8H,#98H
```

em que 98H=10011000B, ou ainda, de maneira explícita, e preferencialmente,

```
SETB EA
SETB ES
SETB ET1
```

Observe-se o uso do número 0' antes do carácter 'A'. Se assim não fosse o compilador interpretaria AFH, ACH e ABH como variáveis e não como endereços. Não estando tais variáveis definidas (através de EQU ou SET)

Bit	EA	-	-	ES	ET1	EX1	ET0	EX0
Ordem	7	6	5	4	3	2	1	0
Endereço	AFH	AEH	ADH	ACH	ABH	AAH	A9H	A8H

Tabela 10.1: Organização do registo IE.

ter-se-ia um erro de compilação. Exige-se assim que os endereços comecem por dígitos numéricos. Esta convenção é útil para que o *assembler* saiba discernir, por exemplo, entre SETB EA, com EA representando o bit EA do registo IE (i.e., uma variável), e SETB 0EAH, com EA representando um endereço.

10.2 Prioridades

Como foi anteriormente afirmado, é possível atribuir prioridades ao atendimento de interrupções. Com efeito, estão disponíveis dois níveis de prioridade que podem atribuir-se a cada uma das fontes de interrupção. O registo IP(B8H), presente na tabela 10.2, permite associar prioridades a interrupões. O significado de cada bit é apresentado nas próximas secções.

10.2.1 PS - prioridade da interrupção do canal série

Permite definir a prioridade atribuída à interrupção associada ao canal série, podendo ser alta (PS=1) ou baixa (PS=0).

10.2.2 PT1, PT0 - prioridades das interrupções dos contadores

Idem para ao C/$\bar{\text{T}}$0 e C/$\bar{\text{T}}$1.

10.2.3 PX1, PX0 - prioridades das interrupções externas

Idem para as interrupções externas $\bar{\text{I}}$NT0 e $\bar{\text{I}}$NT1.

Os restantes bits do registo IP não têm aplicação na versão 80C51. Uma vez mais, por questões de segurança na migração para outros microcontro-

Bit	-	-	-	PS	PT1	PX1	PT0	PX0
Ordem	7	6	5	4	3	2	1	0
Endereço	BFH	BEH	BDH	BCH	BBH	BAH	B9H	B8H

Tabela 10.2: Organização do registo IP relativamente às interrupções.

ladores, onde estes bits possam ter aplicação, é conveniente atribuir-lhes o valor '0'.

Considere-se uma aplicação onde está previsto o atendimento de duas interrupções: as associadas ao canal série e ao C/$\bar{T}$1. Neste caso o registo IE teria o seguinte conteúdo IE=10011000B. O código seria MOV IE,#98H. Considere-se ainda que se pretende atribuir maior prioridade à interrupção associada ao C/$\bar{T}$1. O conteúdo de IP seria IP=00001000B. O código seria MOV IP,#08H. Basicamente, este facto significa que a interrupção pedida pelo C/$\bar{T}$1 pode interromper a execução da rotina da interrupção associada ao canal série. O oposto não se verifica, naturalmente. Em conclusão, uma interrupção com um dado nível de prioridade não pode ser interrompida nem interromper outra da mesma prioridade, mas pode ser interrompida por uma de prioriade superior.

Uma situação rara mas possível de acontecer, corresponde ao caso em que os dois pedidos são feitos em simultâneo. Nesta situação será atendida a interrupção de prioridade mais alta. Caso ambas as interrupções apresentem o mesmo nível proritário, então a decisão de qual atender primeiro é baseada numa ordenação pré-estabelecida (a apresentar posteriormente).

10.3 Estado de IE e IP após reset

Nesta secção relembram-se os estados dos registos IE e IP após a reinicialização do microcontrolador. Os valores de interesse estão apresentados na tabela 10.3. Verifica-se que, por defeito, não há atendimento de interrupções e que é atribuída prioridade baixa a todas as interrupções.

registo	valor
IE	$0 \times \times 0\ 0\ 0\ 0\ 0$
IP	$\times \times \times 0\ 0\ 0\ 0\ 0$

Tabela 10.3: Valores dos registos IE e IP após um *reset*.

10.4 Interrupções externas

Existem duas fontes de interrupção associadas a acontecimentos externos ao microcontrolador: $\bar{\text{I}}$NT0 e $\bar{\text{I}}$NT1. A configuração é feita por intermédio do registo TCON(88H) (ver tabela 10.4). O significado de cada bit é apresentado nas próximas secções.

10.4.1 IT1, IT0 - sensibilidade ao nível ou à transição

Estes bits permitem determinar que tipo de acontecimento produz o pedido de interrupção, podendo ser a transição $1 \rightarrow 0$ ou patamar 0. Se ITn=1:n=0,1, então só existe pedido de interrupção quando se verificar a transição descendente no pino $\bar{\text{I}}$NTn:n=0,1. Caso o programador decida atribuir ITn=0:n=0,1, então o pedido de interrupção é sensível à presença do nível lógico '0' no pino $\bar{\text{I}}$NTn:n=0,1.

10.4.2 IE1, IE0 - sinalização do pedido de interrupção

Estes bits sinalizam a existência do pedido de interrupção. Caso ITn=1:n=0,1 e sempre que se verifique uma transição descendente no pino $\bar{\text{I}}$NTn:n=0,1, então o microcontrolador coloca automaticamente IEn=1:n=0,1, sinalizando o pedido de interrupção. Note-se que havendo pedido de interrupção não significa que haja atendimento. O atendimento está dependente do conteúdo do registo IE. Logo que a rotina de atendimento termine, IEn:n=0,1 é recolocado automaticamente a '0'.

No caso em que ITn=0:n=0,1 tem-se que IEn:n=0,1 é igual ao complemento do nível lógico presente no pino $\bar{\text{I}}$NTn:n=0,1, não havendo armazenamento do estado. Ou seja, tão logo no pino $\bar{\text{I}}$NTn:n=0,1 se tenha 0, ter-se-á IEn=1:n=0,1, havendo pedido de interrupção. Quando $\bar{\text{I}}$NTn=1:n=0,1 ter-se-á IEn=0:n=0,1, deixando de haver pedido de interrupção. A reposição de IEn:n=0,1 a 0 não é necessária, dado o bit seguir o estado do pino $\bar{\text{I}}$NTn:n=0,1. Não é então anormal que, caso não se tomem providências,

Bit	-	-	-	-	IE1	IT1	IE0	IT0
Ordem	7	6	5	4	3	2	1	0
Endereço	8FH	8EH	8DH	8CH	8BH	8AH	89H	88H

Tabela 10.4: Organização do registo TCON relativamente às interrupções

se tenha uma execução contínua de rotinas de interrupção externas enquanto se tiver 0 no pino $\bar{\text{I}}$NTn:n=0,1. A ideia de base deste comportamento tem a ver com o protocolo *handshaking* de ligação a dispositivos lentos. Ou seja, um dado dispositivo coloca o valor 0 no pino $\bar{\text{I}}$NTn:n=0,1. O 80C51, de entre várias acções, deve sinalizar ao dispositivo que o pedido foi reconhecido, talvez por intermédio de um pino de E/S. Este, por sua vez, deve desfazer o 0 em $\bar{\text{I}}$NTn:n=0,1, disponibilizando o valor 1.

Verifica-se que, quer no caso da transição, quer no caso do patamar 0, há que garantir uma temporização mínima para que o microcontrolador possa aperceber-se do pedido de interrupção. A temporização mínima corresponde a um ciclo máquina.

Os bits de ordem quatro a sete do registo TCON são relativos à configuração dos C/$\bar{\text{T}}$, pelo que são abordados em capítulo próprio.

Por exemplo, para que haja atendimento de uma interrupção quando se verificar uma transição descendente no pino $\bar{\text{I}}$NT0, deverá executar-se a seguinte instrução,

```
SETB 88H
```

ou,

```
SETB IT0
```

de outro modo, o bit IT0 não deve ser modificado, dado valer 0 por defeito (ou seja, após um *reset*).

10.5 Interrupções associadas aos dispositivos

Em termos resumidos, os contadores/temporizadores têm a capacidade de gerar uma interrupção após atingirem o estado de sobrecarga (transição do valor máximo para 0). A interrupção é sinalizada através dos bits TF0 e TF1, associados ao C/$\bar{\text{T}}$0 e C/$\bar{\text{T}}$1, respectivamente, que passam a valer 1. A reposição do estado original, *i.e.*, TF0=0 e TF1=0, é da responsabilidade do programador, devendo ser feita por software.

interrupção	bit	**registo SFR**	**colocação no estado original**
externa 0	IE0	TCON	pelo *hardware*
externa 1	IE1	TCON	pelo *hardware*
C/T1	TF1	TCON	pelo *software*
C/T0	TF0	TCON	pelo *software*
porta série	TI	SCON	pelo *software*
porta série	RI	SCON	pelo *software*

Tabela 10.5: Lista dos bits pertencentes a registos SFR que sinalizam a existência de interrupções.

Relativamente ao canal série, é feito o pedido de interrupção quando terminar o processo de envio ou recepção de uma trama. Dado existir uma só interrupção associada ao canal série, há que saber diferenciar entre interrupção devida ao envio e interrupção devida à recepção. O mecanismo implementado é o seguinte: ao envio está associado o bit TI e à recepção está associado o bit RI, ambos pertencentes ao registo SCON. O estado normal destes bits é 0. Quando uma das situações acontecer, *i.e.*, fim do envio ou fim da recepção, o bit correspondente transita para 1, e com esse valor permanece até que o programa o recoloque no estado original. Uma das primeiras acções a tomar na rotina de atendimento da interrupção será determinar qual a fonte da interrupção: TI ou RI.

Na tabela 10.5 são apresentados os bits que determinam ou sinalizam o pedido de interrupção associados aos diversos dispositivos, em que "software" representa "pelo programador" e "hardware" representa "automaticamente pelo microcontrolador". Note-se que estes bits podem ser alterados directamente pelo programa, permitindo simular o pedido ou o cancelamento de interrupções.

10.6 Sequência das interrupções

Apesar de se ter transmitido a ideia de que existe uma forma de semi-paralelismo associado às insterrupções, no sentido da existência de um "olho" que observa todas as fontes de interrupção em simultâneo, tal não é verdade. Com efeito, as condições de verificação do pedido de interrupção são

ordem	fonte	prioridade
1	IE0	alta
2	TF0	
3	IE1	
4	TF1	
5	RI, TI	baixa

Tabela 10.6: Sequência de verificação das interrupções existentes no 80C51.

feitas num determinado instante do ciclo máquina, segundo uma ordem pré-estabelecida. A sequência interna de *poling* é apresentada na tabela 10.6. Assim, é possível afirmar que existindo uma ordem de verificação das interrupções, existe igualmente uma atribuição de prioridades por defeito.

Na figura **??** é apresentado um diagrama de blocos que permite resumir tudo o que foi abordado: as diversas fontes de interrupção, os bits de registos SFR modificados aquando da existência de interrupções, a activação global e particular, prioridades e sequência de verificação por defeito.

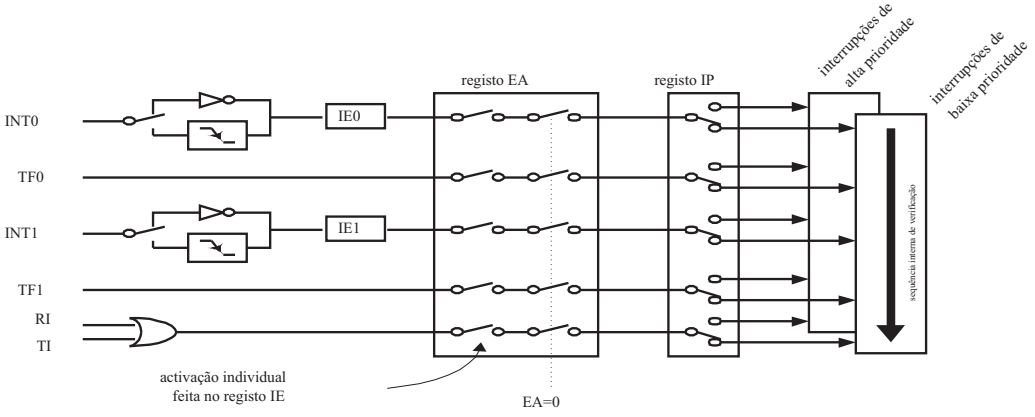

Figura 10.1: Diagrama de blocos ilustrativo das interrupções no 80C51.

10.7 Rotinas de atendimento

Após o pedido de uma interrupção, o fluxo de execução do programa passa para a rotina de atendimento correspondente. Estas devem existir a partir de posições de memória bem definidas, comummente denominadas vectores. No caso do 80C51, as posições de início das rotinas são apresentadas na tabela 10.7. Deve ter-se em atenção que só o conteúdo do registo PC é guardado na memória de *stack*. Compete ao programador utilizar esta memória (através das instruções POP e PUSH) para guardar os valores relevantes à aplicação em causa.

Como se pode observar na tabela 10.7, o espaço disponibilizado é relativamente escasso: oito bytes para cada rotina, com excepção da última que não apresenta limite superior. Caso este espaço não seja suficiente, o programador poderá utilizar instruções de salto (LJMP, AJMP e SJMP, ou JMP) para outras posições de memória, de maneira a criar um espaço maior entre as várias rotinas de atendimento. Este aspecto está ilustrado no seguinte exemplo.

Endereço	**Etiqueta**	**Instrução**
0000H:		JMP principal
0003H:		JMP externa0
000BH:		JMP contador0
0013H:		JMP externa1
001BH:		JMP contador1
0023H:		JMP serie
	externa0:	...
		RETI
	contador0:	...
		RETI
	externa1:	...
		RETI
	contador1:	...
		RETI
	serie:	...
		RETI
	principal:	...

Note-se que não se pretende transmitir a ideia de que o espaço é propositadamente limitado, levando o programador a seguir a estratégia anterior

Fonte	**Vector**
externa 0	0003H
contador 0	000BH
externa 1	0013H
contador 1	001BH
canal série	0023H

Tabela 10.7: Endereços de início das rotinas de atendimento de interrupções.

sem excepção. Muitas vezes os oito bytes disponibilizados são suficientes.

Capítulo 11

Programação em C

Neste capítulo aborda-se a programação em linguagem C para o 8051. Isto significa que tem de se assumir um dado compilador. Com efeito, apesar de a linguagem C estar devidamente padronizada, os fabricantes de compiladores, em função dos computadores para os quais desenvolvem os seus compiladores, introduzem especificidades. Assim é também com o 8051. O compilador que se usa nos exemplos apresentados é o da Keil (`http://www.keil.com`). Note-se, contúdo, que neste capítulo, não se fazem explicações sobre o IDE (Integrated Development Environment) da Keil.

Assume-se, obviamente, umm conhecimento pelo menos intermédio desta linguagem.

Porquê o C? A linguagem C é justamente classificada como de nível intermédio, i.e., entre o baixo nível, tal como o assembly, e o alto nível, como o JAVA. Esta classificação justifica-se pelo facto de o C permitir aceder a recursos da máquina, nomeadamente posições de memória (e seus bits), discos, teclado, UART, etc., e recursos de alto nível, como ficheiros, aritmética em vírgula flutuante, etc.

11.1 Vantagens

Desenvolver aplicações em C para o 8051 trás algumas vantagens. Por um lado, a clareza do programa deverá aumentar, uma vez que um programa em C terá necessariamente menos instruções que o mesmo programa escrito em *assembly*. Por outro lado, certas instruções são mais claras que outras, embora se trate de um pormenor de estilo e não de facilidade em escrever

programas. Por exemplo, a instrução `EA=1`, em C, é mais clara que a instrução `SETB EA`, em assembly. Melhor exemplo é o que se ilustra nas figuras 11.1 e 11.2, onde se apresenta um exemplo da instrução *if-then-else* escrita em C e *assembly*.

```
if(X==4) {
    instr1;
    instr2;
}
else {
    instr3;
    instr4;
}
instr5;
...
```

Figura 11.1: Exemplo genérico de um *if-then-else* em C.

```
      mov A,X
      cjne A,#4,else
      instr3
      instr4
      jmp after
else: instr1
      instr2
after: instr5
      ...
```

Figura 11.2: Exemplo genérico de um *if-then-else* em *assembly*.

Naturalmente, devido ao C ser de nível mais alto que o *assembly*, o programador tem mais oportunidade de se concentrar na solução e menos nos pormenores de implementação (i.e., menos preocupado com a arquitectura do microcontrolador).

Depois, há aplicações onde é manifestamente difícil programar em *assembly*, tanto mais quando o desenvolvimento de tais aplicações é facilitado com

a existência de recursos apropriados do próprio C. Tal é o caso de aplicações com aritmética real.

Vamos então aos pormenores técnicos do C respeitantes ao 8051.

11.2 Tipos de dados

O C fornece mais alguns tipos de variáveis relativamente aos existentes no *assembly*. Os mais substanciais são os que permitem tratar números com parte decimal (`float` e `double`). Os inteiros convencionais (`int`) são também valiosos, pois 16 *bits* permitem mais precisão que os 8 *bits* do 8051.

Caso não hajam requisitos especiais, declarar variáveis não requer muitos cuidados, a não ser o facto de que o C consome bastante memória do microcontrolador. Havendo requisitos relativos ao tipo de memória onde as variáveis devem ficar alojadas, então há que ter em atenção o mapa de memória de dados do 8051.

Para se declarar variáveis que residam nos primeiros 128 *bytes*, deve usar-se o modificador `data`, tal como nos exemplos seguintes.

```
int data x;
char data c[4];
```

Para variáveis que devam ficar armazenadas em qualquer parte da memória, usa-se o modificador `idata`, tal como no exemplo seguinte.

```
int idata x;
```

Para variáveis cujos bits se pretendam endereçar, usa-se o modificador `bdata`, tal como nos exemplos seguintes.

```
int bdata x;
char bdata c[4];
```

Note-se que, no primeiro exemplo, não se está a declarar `x` como sendo do tipo *bit*, mas sim um conjunto de 16 *bits* que são endereçáveis. Para se aceder aos respectivos bits, faz-se, por exemplo, `x^2;`, que permite aceder ao *bit* 2 de `x`, e `c[2]^7;`, que permite aceder ao MSB do terceiro elemento de `c`.

Para se declararem variáveis do tipo *bit*, faz-se como no exemplo seguinte.

```
sbit bit2=x^2;
sbit bit15=c[2]^15;
```

Note-se que estas expressões representam declarações e não definições. Ou seja, no caso do primeiro exemplo, `bit2` é uma variável que reflecte sempre o valor do *bit* 2 de `x`. Se se tratasse de uma definição, então mudar `x^2` não implicaria mudanças em `bit2`.

Exemplos de uso são: `bit2=0;`, `bit15=1;`.

Relativamente ao uso de `sbit` em tipos de dados agregados, veja-se o seguinte exemplo.

```
union a {
   float af;
   long al;
};

bdata struct b {
   char bc;
   union a c;
} d;

sbit y31=d.c.al^31;
sbit y0=d.bc^0;
```

`sbit` é útil para se aceder a *bits* de registos SFR, tal como o caso seguinte.

```
sbit EA=0xAF;
```

As formas desta declaração são:

```
sbit nome=nome-sfr ^ position-bit;
sbit name=sfr-address ^ bit-position;
```

Eis alguns exemplos de uso:

```
sfr IE=0xA8;
sbit EA=IE^7;
sbit EA=0xA8^7;
```

Note-se que se declarou a variável `IE` como sendo do tipo `sfr`. Tipicamente, não há necessidade de declarar variáveis deste tipo, pois os registos SFR de um dado microcontrolador estão todos declarados em ficheiros com extensão *h* que se incluem no início dos ficheiros em C, tal como se exemplifica de seguida.

```
#include <reg51.h>
```

Outra forma de se aceder a *bits* é através do tipo `bit`, como no exemplo seguinte.

```
static bit b=0;
```

Vejamos o caso de uma função que recebe dois *bits* e retorna a sua disjunção exclusiva.

```
bit fxor(bit a, bit b)
{
return a^b;
}
```

As variáveis do tipo *bit* são colocadas na área 20H a 2FH.

Não se podem declarar ponteiros para `bit`, tal como `bit *pb;`, nem declarar conjuntos de variáveis do tipo `bit`, tal como `bit ab[8];`.

11.3 Interrupções

Relembrem-se os vectores de interrupção do 8051.

número	descrição	endereço
0	externa 0	0003H
1	timer/counter 0	000BH
2	externa 1	0013H
3	timer/counter 1	001BH
4	porta série	0023H

É necessário diferenciar as funções normais das de atendimento de interrupções (ISR–Interrupt Service Routine). Tal é feito através de um modificador que se coloca no final do cabeçalho da função. Veja-se o exemplo,

```
unsigned char second;

void timer0ISR (void) interrupt 1
{
second++;
}
```

nome	valor	significado
nan	0xFFFFFFF	not a number
+inf	0x7F80000	$+\infty$
-inf	0xFF80000	$-\infty$

Tabela 11.1: Tipos de excepções em aritmética real.

Tal como se percebe no exemplo, a rotina de atendimento que se define é a associada ao contador 0, que, na tabela anterior, tem ordem 1. Ou seja, o número que aparece depois da palavra *interrupt* é referente à ordem da ISR que se define.

As rotinas de atendimento de interrupções não têm argumentos e não retornam valores. Também não podem ser chamadas explicitamente.

A tabela apresentada refere-se ao 8051. Microcontroladores mais evoluídos têm mais interrupções; a estas é atribuído um número de ordem pelo fabricante, respeitando sempre a ordem da tabela apresentada.

11.4 Erros em aritmética real

No C em consideração não existe suporte para erros provenientes de aritmética real. Note-se que estes erros acontecem em tempo de execução. Assim, tem de ser o próprio programador a prever as situações possíveis e disponibilizar as medidas correctivas.

Algumas definições importantes são apresentadas na tabela 11.1.

A figura 11.3 apresenta um exemplo de um tratamento simples de uma excepção aritmética.

O tratamento de situações de excepção, tal como *nan* ou infinito, é tão mais importante quanto mais sensíveis são as grandezas que se manipulam.

11.5 Ponteiros

Os ponteiros são simultaneamente uma fonte de desempenho e de preocupação. Temos assim duas boas razões para os entender bem: aumentar o desempenho e diminuir as preocupações.

No C em estudo existem dois tipos de ponteiros: genéricos e específicos a memória.

```
#define NaN 0xFFFFFFF
#define plusINF 0x7F80000
#define minusINF 0xFF80000

union f {
    float f;
    unsigned long ul;
}

void main(void)
{
float a, b;
union f x;

...
x.f=a*b;
if(x.ul==Nan || x.ul==plusINF || x.ul==minusINF)
    /* tratar o erro */
    ...
else
    /* fazer os cálculos */
    ...
}
```

Figura 11.3: Tratamento simples de uma excepção aritmética.

Antes de abordarmos os pormenores, vejamos exemplos de declaração de ponteiros dos dois tipos. Ponteiros genéricos:

```
char* s;
char* data num;
```

Ponteiros específicos a memória.

```
char data* s;
char data* idata s;
```

Vejamos cada caso separadamente.

11.5.1 Ponteiros genéricos

Generic pointers are stored using three bytes. They can point to any variable independently of where it is stored (internal versus external memory, etc.). Because of this, they are in fact generic/universal.

Code generated from generic pointers is slower in comparison to memory-specific pointers. In other words, if speed is a priority, then you should use memory-specific pointers.

When declaring a generic pointer, one can specify the memory type where the pointer shall be stored. For example:

```
char * xdata ps;
```

In this case, the variable to which `ps` points can reside in any memory type. Variable `ps`, however, is stored in external memory (`xdata`).

Memory-specific pointers

As seen in the example before, memory-specific pointers include a reference to the memory type to which they point to.

```
char data* str;
```

Means a pointer to a string residing in `data` memory.

Like generic pointers, you may specify the memory area where the pointer is to be stored.

```
char data* xdata str;
```

This is a pointer stored in `xdata` pointing to a `char` in `data`.

Naturally, a pointer to a variable of some type cannot point to a variable of a different type.

Also, a memory-specific pointer pointing to a specific memory type cannot point to a different memory type.

Memory-specific pointers are eficient but not flexible.

Pointer conversion is a serious matter that should be studied meticulously; it is not the focus of these slides, however.

```
#include "reg_c51.h"

char uart_data;

/**
 * FUNCTION_PURPOSE: This file set up uart in mode 1 (8 bits uart) with
 * timer 1 in mode 2 (8 bits auto reload timer).
 * FUNCTION_INPUTS: void
 * FUNCTION_OUTPUTS: void
 */
void main (void) {
    SCON = 0x50;                        /* uart in mode 1 (8 bit), REN=1 */
    TMOD = TMOD | 0x20 ;          /* Timer 1 in mode 2 */
    TH1  = 0xFD;                  /* 9600 Bds at 11.059MHz */
    TL1  = 0xFD;                       /* 9600 Bds at 11.059MHz */
    ES = 1;                               /* Enable serial interrupt */
    EA = 1;                             /* Enable global interrupt */
    TR1 = 1;                            /* Timer 1 run */

   while(1);                     /* endless */
}

/**
 * FUNCTION_PURPOSE: serial interrupt, echo received data.
 * FUNCTION_INPUTS: P3.0(RXD) serial input
 * FUNCTION_OUTPUTS: P3.1(TXD) serial output
 */
void serial_IT(void) interrupt 4 {

    if (RI == 1)
    {                                  /* if reception occur */
      RI = 0;                        /* clear reception flag for next recepti
      uart_data = SBUF;           /* Read receive data */
      SBUF = uart_data;              /* Send back same data on uart*/
    }
    else TI = 0;         /* if emission occur */
                           /* clear emission flag for next emission*/
}
```

Figura 11.4: EXemplo