Tópico 10

Microcontroladores

Autores: José Raimundo de Oliveira e Wu Shin-Ting

DCA - FEEC - Unicamp

Outubro de 2019

Pelo exposto até agora, é claro que a execução completa de uma tarefa não basta termos uma CPU de altíssimo desempenho. No mínimo são adicionalmente necessários uma memória para armazenar as instruções e os dados, um periférico para interagir com o mundo físico e um barramento para interligá-los, como mostra Figura 4.1.

Você já imaginou um micro-computador integrado num único circuito integrado, ou seja um sistema numa única pastilha (em inglês, ***system on chip*** ou **SoC**), embutível em qualquer objeto de seu uso pessoal de forma ubíqua? Um micro-computador que contém um processador, um sistema de memória, e módulos de conversão entre os sinais analógicos e digitais. Um processador com um repertório de instruções limitado. Um sistema de memória constituído por RAM, ROM, EPROM ou *flash* com uma capacidade total de armazenamento na ordem de *kilobytes*. Enfim, um sistema computacional miniaturizado para execução completa de uma tarefa específica, de baixo custo, de baixo consumo de energia (operando na base de bateria) e suficientemente robusto para funcionar em condições de operação das mais diversas possíveis.

O desenvolvimento de um microcontrolador iniciou-se com o desenvolvimento de microprocessadores, quando a *Intel* integrou em 1971 numa única pastilha, Intel 4004 de 4 *bits*, as funções de uma CPU. No mesmo ano, os engenheiros Gary Boone e Michael Cochran da *Texas Instruments* projetaram o primeiro microcontrolador TMS 1000, disponível comercialmente em 1974. Hoje em dia existe uma parafernália de variedades de sistemas digitais na forma de microprocessadores, microcontroladores e microcomputadores. Como distinguir um microcontrolador dos outros sistemas?

Considera-se como um **microcontrolador** um sistema computacional de baixo custo, de baixo consumo de energia, robusto suficiente para executar uma tarefa

específica em condições de operação bem diversas, e tem embutido nele os seguintes componentes (Figura 10.1):

- unidade de processamento central,
- memória para instruções (ROM ou flash),
- memória para dados (RAM),
- temporizadores e contadores,
- portas de entrada/saída,
- interfaces de comunicação serial,
- circuito de relógio, e
- mecanismo de interrupção.

Microcontroladores modernos integram ainda circuitos de interface com os periféricos, como SPI (*Serial Peripheral Interface*), I2C (*Inter Integrated Circuit*), circuitos de conversão AD e DC, CAN (*Controller Area Network*), USB (*Universal Serial Bus*), e muitos outros circuitos dedicados. Portanto, um microcontrolador é um circuito de integração de muita larga escala, em inglês *very large scale integration* (VLSI). Não podemos deixar de mencionar que, tão importante quanto os módulos funcionais embarcados, os barramentos embarcados são fundamentais para assegurar transferências apropriadas dos sinais que devem circular entre os módulos embarcados durante a execução de uma tarefa.

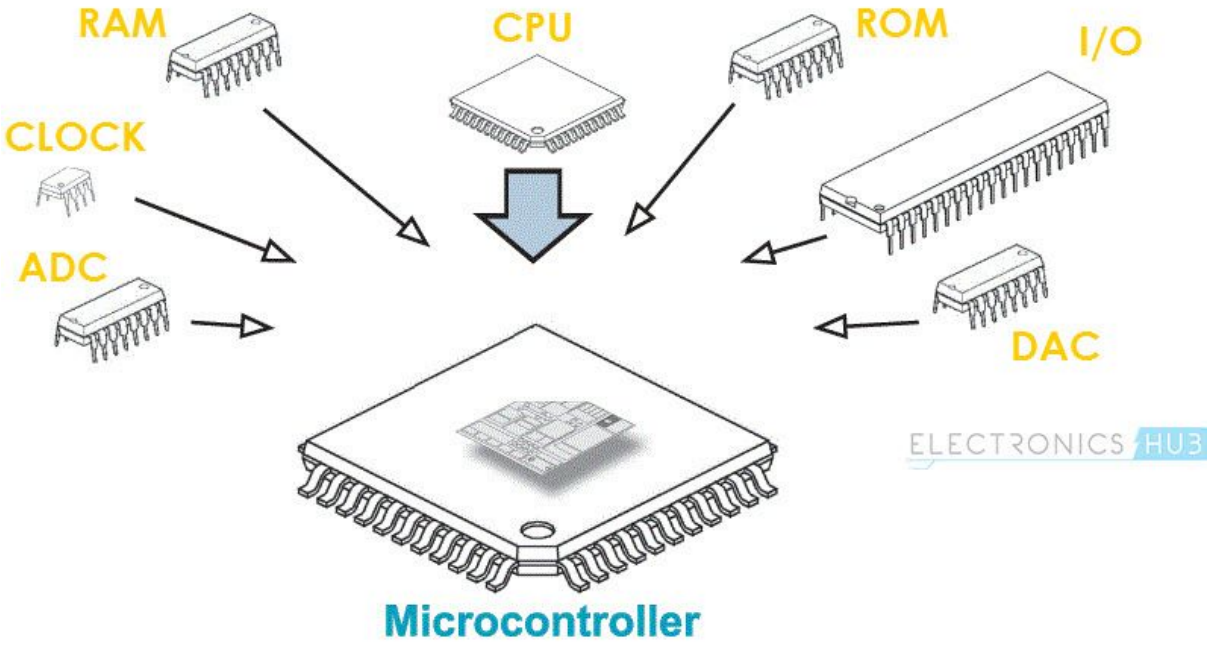


Fig. 10.1: Microcontrolador (Fonte: [1]).

Tendo todos os principais circuitos integrados num mesmo *chip*, com todos os problemas de interfaceamento que discutimos nos capítulos anteriores devidamente solucionados, desenvolver projetos dedicados com uso de microcontroladores vem reduzindo drasticamente o tempo de desenvolvimento de *hardware* de um sistema

computacional e os projetistas podem se focar na implementação das funções sobre este *hardware*. Figura 2 ilustra três dispositivos em que os microcontroladores estão embutidos para realizar tarefas bem específicas. Chamam atenção as condições de operação bem distintas em que eles são submetidos: alta pressão (robô mergulhador), alta temperatura (aquecedor programável), baixa temperatura (robô limpador de neve).

Figura 10.2: Microcontrolador em objetos sob condições de operação bem distintas.

10.1 Organização

(baseado em [1])

Figura 10.3 apresenta uma organização básica de um microcontrolador. Além dos três módulos clássicos da arquitetura de von Neumann, CPU (Capítulo 4), memória (Capítulo 5 e 6), entrada/saída (Capítulo 7), e barramento (Capítulo 9), há módulos de suporte que ampliam o espectro de aplicações do microcontrolador. Alguns desses módulos, o conversor analógico-digital (ADC), o conversor digital-analógico (DAC) (Capítulo 8) e circuitos de interface de comunicação serial, incluindo a interface de programação e teste de circuitos digitais JTAG (Capítulo 9), já vimos nos capítulos anteriores. Outros, o controlador de interrupções e temporizadores/ contadores, são módulos fundamentais num microcontrolador serão detalhados neste capítulo.

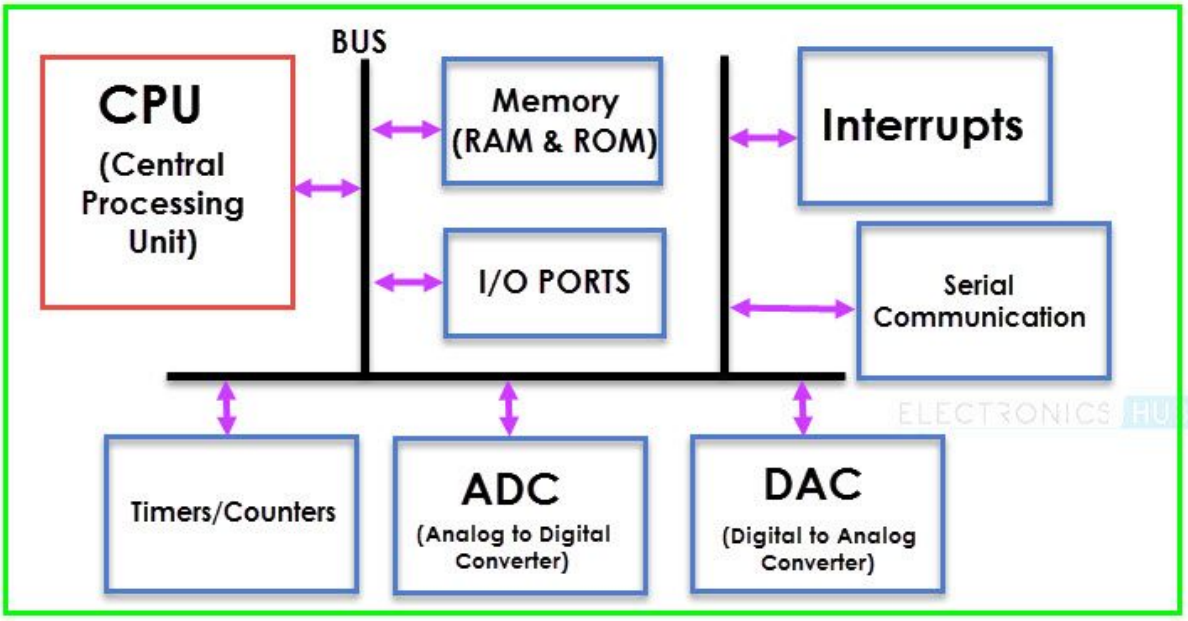


Figura 10.3: Organização básica de um microcontrolador (Fonte: [1]).

10.2 Módulos Embarcados

Nesta seção faremos uma breve revisita dos módulos computacionais que vimos nos capítulos anteriores recondicionados para microcontroladores que tem uma organização mais simples visando a baixo custo e baixo consumo.

10.2.1 Microprocessadores Embarcados

A arquitetura do microprocessador (núcleo, em inglês *core*) nos microcontroladores modernos é predominantemente a **arquitetura Harvard** ou **arquitetura Harvard modificada**. Como vimos na Seção 4.2, a principal diferença entre a arquitetura de Von Neumann e a arquitetura Harvard está na separação da memória principal em memória de dados e memória de instruções na segunda arquitetura. A arquitetura Harvard modificada é uma arquitetura em que a CPU e as memórias de dados e de instruções tem conexões separadas, mas compartilham um mesmo espaço de endereçamento, ou, como veremos na Seção 10.2.7, é uma arquitetura em que alguns tipos específicos de dados, como os registradores de função especial de dados, os dados constantes e as tabelas de funções, podem ser armazenados na memória de instruções.

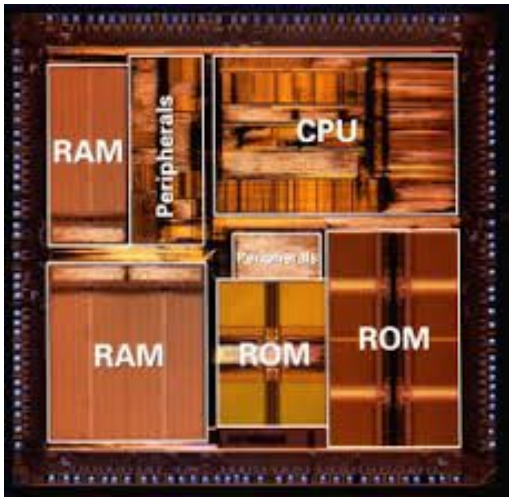
A arquitetura *Harvard* foi adotada em 1975 no desenvolvimento dos microcontroladores pela *Microchip Technology*. O microcontrolador foi originalmente batizado como **Controlador de Interface Programável**, em inglês *Programmable Interface Controller* (PIC), mas foi logo substituído por **Computador Inteligente Programável**, em inglês *Programmable Intelligent Computer*. Os primeiros

microprocessadores ARM, como Cortex M0, eram da arquitetura de *von Neumann*. E os mais modernos, como Cortex M4 e os sucessores, são da arquitetura *Harvard* ou *Harvard* modificada. A arquitetura do núcleo do microcontrolador AVR é, por exemplo, da arquitetura *Harvard* modificada de 8 *bits*. Com a arquitetura *Harvard*, onde os barramentos de dados e de memória são independentes, é mais efetiva a **segmentação de instruções** (*pipeline*), que vimos na Seção 4.2.2.

10.2.2 Memórias Embarcadas

Funcionalmente, as memórias embarcadas são equivalentes aos módulos de memória convencionais. As memórias embarcadas apresentam, porém, um desempenho bem superior, em decorrência da sua velocidade e da largura dos barramentos dedicados. Apesar do empenho dos projetistas de *chips*, as memórias embarcadas ocupam ainda um grande espaço da pastilha e são complexas tanto para projetar quanto para fabricar principalmente quando se integra vários tipos de memória num *chip* (Figura 10.4.(a)) .

As técnicas de otimização aplicáveis para os módulos de memórias clássicos não são apropriadas para as memórias embarcadas [2]. Na Figura 10.4.(b) apresenta comparativamente as diferenças entre uma memória *Flash* clássica e a sua versão embarcada. Note que a ocupação da área de uma memória embarcada num *chip* é muito menos crítica do que a ocupação da memória convencional (uma percentagem menor). Portanto, é comum adotar estratégias diferenciadas para projetar as células de memória, como usar 6 transistores numa célula de SRAM ao invés do projeto otimizado que envolve somente 4 transistores, ou implementar duas portas flutuantes nas EPROMs para acessos separados de leitura e de escrita, As memórias EPROMs são feitas com uma camada de polisilício[1].



	Embedded	Stand-Alone
Typical Array Size (Bits)	2K - 2M	1M - 16M
Typical Percent of Chip Area	5% - 40%	100%
Cell Size Very Critical	No	Yes
Cell Type	NOR	NOR/NAND/AND
Dual Gate-Oxide	Likely	Not Likely
Multiple Modules	Yes	No
Redundancy Repairing	Rare	Yes

Source: Motorola/ICE, "Memory 1997" 20813

[1] Polisilício, ou silício policristalino, é um material que consiste em pequenos cristais de silício.

(a) Ocupação da memória em relação a CPU.

(b) Embarcada e clássica

Figura 10.4: Memória embarcada (Fonte: [2]).

As memórias ROM e RAM estáticas (SRAMs) são as mais difundidas no mundo dos embarcados. Em muitos microcontroladores há ainda uma pequena quantidade de memória SRAM integrada no *chip* como *cache* Essa memória é conhecida como a memória primária ou *cache* nível 1 (L1). Como módulos de memória convencionais, as memórias embarcadas de EPROM/EEPROM estão sendo substituídas pelas memórias *Flash*. Figura 10.5 mostra comparativamente os diferentes tipos de memória não voláteis em aplicações embarcadas.

	ROM	EPROM	Single Gate Flash	Split Gate Flash	EEPROM
Density	+++	++	+	-	---
Electrically Prog.	---	+	+	+	+
Electrically Erase.	---	---	+	+	+
Byte Erasable	-	-	-	-	+
Program Disturb	+	+	--	-	++
Over Erase/Program	+	+++	-	+	++
Process Complexity	+++	++	--	+	-
Manufacturability	+++	++	-	+	+
Cost	+++	++	+	+	--
	Worst	---	--	-	+	++	+++	Best

Source: Motorola/ICE, "Memory 1997" 20810

Figura 10.5 Memórias não voláteis em sistemas embarcados (Fonte: [2]).

As únicas memórias que desafiam os projetistas são as memórias DRAMs. Em desenvolvimento por várias companhias desde os anos 90, as DRAMs embarcadas são ainda as menos utilizadas devido à complexidade do processo da sua fabricação. O seu elevado desempenho de largura de banda, como mostra a Figura 10.6, as torna extremamente atraentes para aplicações gráficas e multimídias. Já em 1996, Mitsubishi apresentou em ISSCC[2] um microcontrolador multimídia com um processador RISC de 32 *bits*, uma DRAM de 16 M*bits* e 16 K*bits* de SRAM *cache* embutidas.

[2] *International Solid-State Circuits Conference*.

Memory Type	Bandwidth
Standard (256K x 16)	240MB/sec.
High-Speed (256K x 16)	400MB/sec.
2MB x 8 RDRAM (Rambus DRAM)	500MB/sec.
SDRAM (Synchronous DRAM)	640MB/sec.
Embedded DRAM (32K x 256)	2,560MB/sec.

Source: Silicon Magic Corp/ICE, "Memory 1997" 20811A

Figura 10.6: Largura de banda de diferentes memórias embarcadas (Fonte: [2]).

10.2.3 Portas de Entrada e Saída

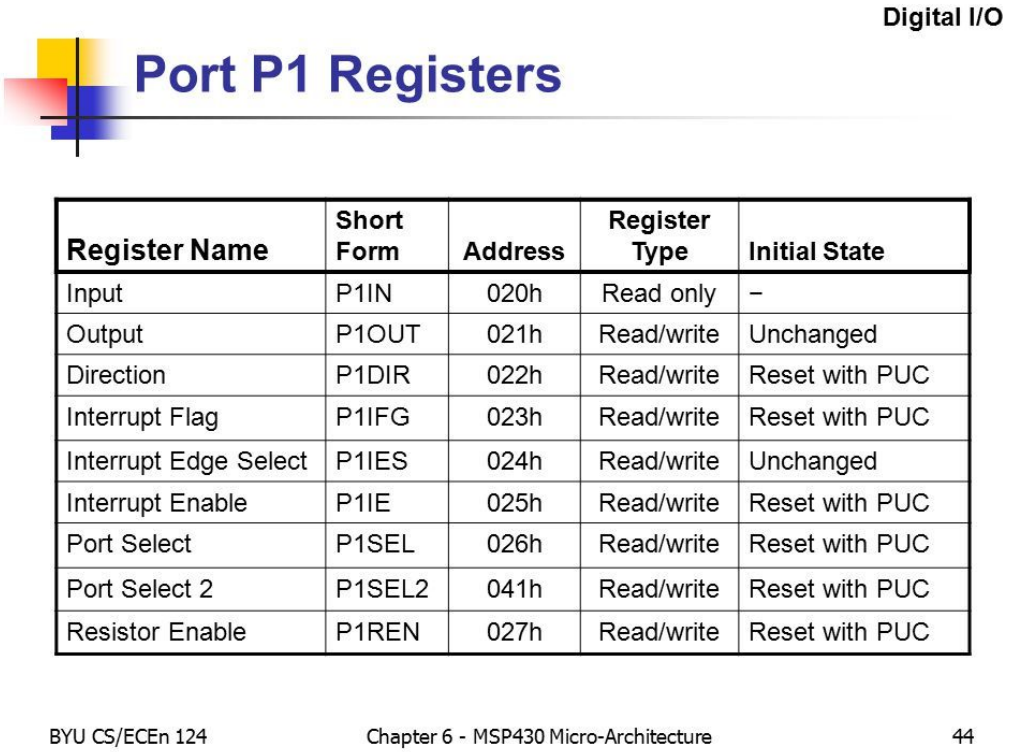
Para se comunicar com o mundo físico, todos os microcontroladores são providos de uma série de pinos. Diferentes dos pinos dos módulos que vimos até agora, como os pinos dos módulos das memórias (Capítulo 5) ou dos microprocessadores (Capítulo 4), são associadas aos pinos dos microcontroladores as portas. **Portas** são abstrações de uma série de registradores embutidos nos microcontroladores que permitem que um *software*/*firmware* controle, modifique ou leia os estados dos pinos individualmente. A correspondência entre os pinos e os *bits* das portas é biunívoca. Ou seja, cada pino é controlado por um *bit* de uma porta.

10.2.3.1 Mapeamento

Nos microcontroladores modernos, os registradores de controle, de dados e de estado dos pinos são mapeados no espaço de memória do processador (Seção 5.5) de forma que se usa as mesmas instruções de acesso às memórias para acessar os dados nos registradores. Tipicamente, são associados a um pino

- *bit* de controle de direção de dados (entrada/saída),
- *bit* de controle do tipo de saída (*pull-up*, *pull-down*, *push-pull*, impedância alta),
- *bit* de controle de alternância do estado corrente,
- *bit* de controle de evento de interrupção (borda, nível),
- *bit* de dado de entrada,
- *bit* de dado de saída.

Por questão de modularidade e simplicidade na implementação em *hardware*, estes *bits* se encontram em diferentes registradores associados a cada porta, como registrador de controle de direção, registrador de controle de habilitação de interrupção, registrador de dados de entrada e registrador de dados de saída.Figura 10.7 ilustra o conjunto de registradores de controle, de dado e de estado associados à porta P1 do microcontrolador MSP430. Estes registradores são mapeados no mesmo espaço de endereços da memória como mostra a terceira coluna.



Digital I/O

Port P1 Registers

Register Name	Short Form	Address	Register Type	Initial State
Input	P1IN	020h	Read only	–
Output	P1OUT	021h	Read/write	Unchanged
Direction	P1DIR	022h	Read/write	Reset with PUC
Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
Port Select	P1SEL	026h	Read/write	Reset with PUC
Port Select 2	P1SEL2	041h	Read/write	Reset with PUC
Resistor Enable	P1REN	027h	Read/write	Reset with PUC

BYU CS/ECEn 124 Chapter 6 - MSP430 Micro-Architecture 44

Figura 10.7: Registradores da porta GPIO do microcontrolador MSP430.

10.2.3.2 Multiplexação

Visando a minimizar a quantidade de pinos, e portanto as dimensões de um microcontrolador, cada pino é tipicamente multiplexado nos microcontroladores modernos. Há *bits* de controle de configuração das funções atribuídas aos pinos. Figura 10.8 mostra uma implementação do circuito de controle associado a um pino físico, para que ele seja bi-direcional (*Output Data* e *Input Data*) e multiplexável na saída (*Output Multiplexers*).

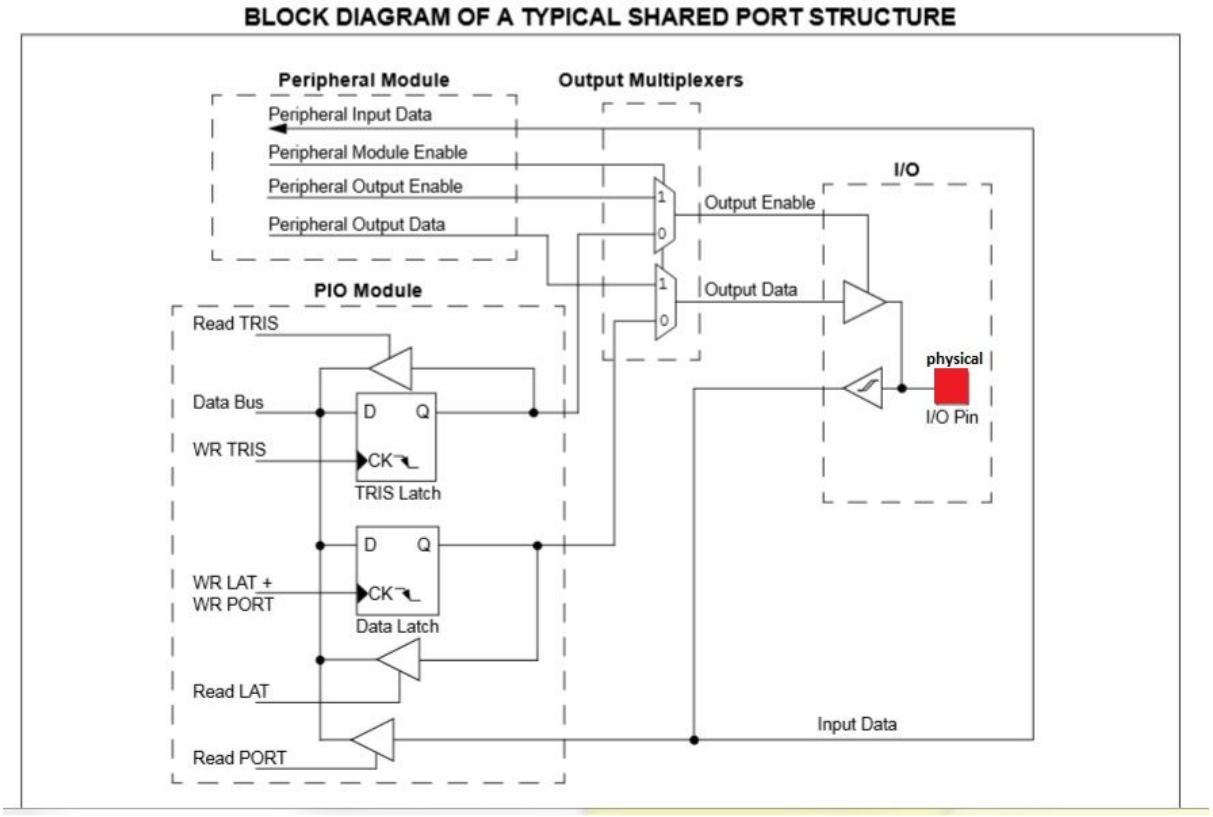


Figura 10.8: Circuito de um pino embarcado (Fonte: [3]).

Figura 10.9 mostra a pinagem de um microcontrolador AVR (Arduino) de 8 *bits*, associada a quatro portas A, B, C e D. Exceto os pinos de alimentação, todos os pinos tem duas funções, sendo uma delas é a função de pino digital de propósito genérico, em inglês *General Purpose Input/Output* (*GPIO*). Por exemplo, o pino GPIO5 é multiplexado entre a entrada digital (função GPIO) e o sinal de relógico SCL da interface de comunicação (função SCL), enquanto o pino ADC0 é dedicado à entrada analógica.

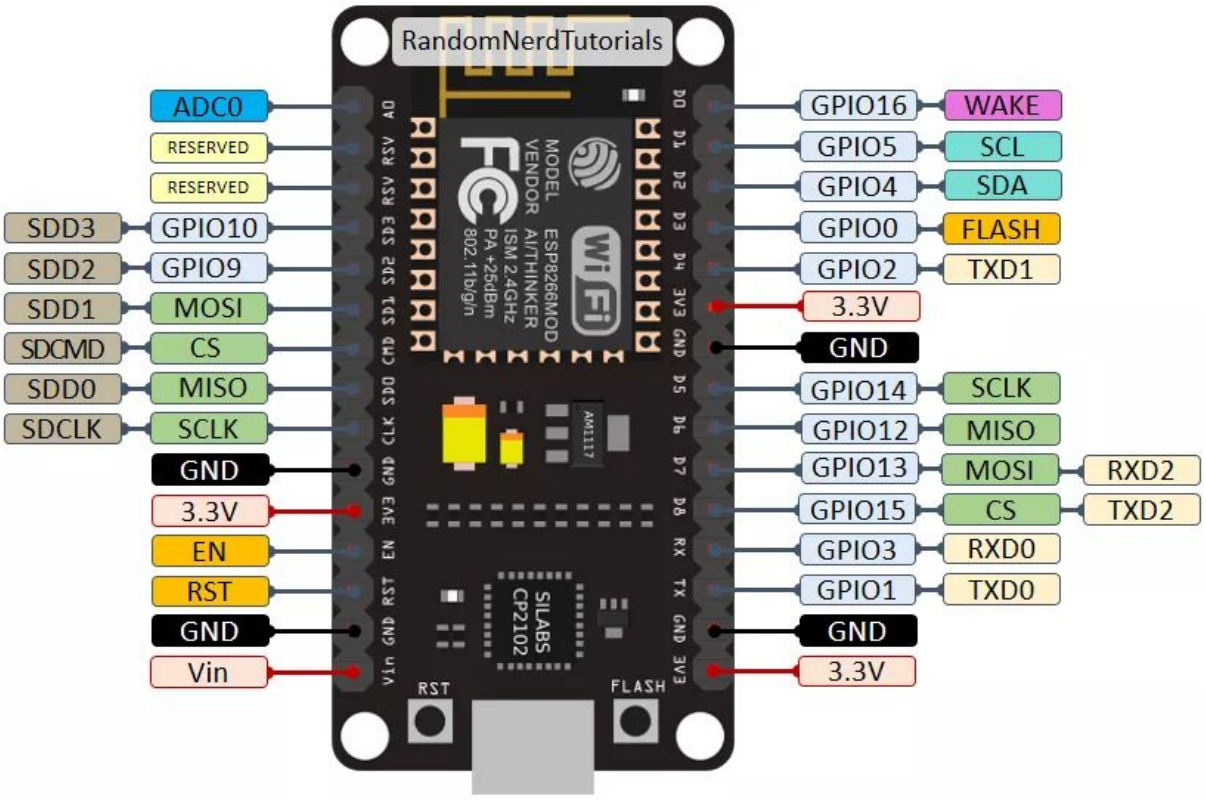


Figura 10.9: Pinos associados a uma porta no microcontrolador NodeMCU (Fonte: [13]).

10.2.3.3 Programação em C

Embora os pinos de entrada/saída possam ter o seu estado alterado pelo mundo externo ao microcontrolador, eles são mapeados no mesmo espaço de memória. Como o conteúdo da memória é tradicionalmente modificado pelo processador (Seção 4.1), os mecanismos de otimização implementados nos compiladores procuram reduzir a quantidade de instruções de um programa removendo as instruções desnecessárias como aquelas em que operam sobre variáveis cujo conteúdo não é alterado. Por exemplo, para o trecho do código abaixo o mecanismo de otimização removeria o laço “while” porque o conteúdo da variável “foo” não é modificado por nenhuma instrução:

```
static int foo;
void bar(void) {
    foo = 0;
    while (foo != 128)
        ;
}
```

O código otimizado com uma comparação a menos ficaria:

```
void bar_optimized(void) {
    foo = 0;
    while (true)
        ;
}
```

No entanto, se a variável por um registrador de entrada de uma das portas, cujo estado é alterado pelo fenômeno externo ao processador, o fluxo de controle do código otimizado não atenderia a nossa expectativa. Como podemos “instruir” o compilador a não otimizar o trecho de código envolvendo a variável “foo”?

Para “avisar” o sistema computacional sobre os registradores que contém *bits* cujos valores possam ser modificados externamente, usamos o qualificador *volatile* em linguagem C. Se reescrevermos o trecho de código na forma abaixo, podemos evitar que o laço “while” seja removido durante o processo de otimização:

```
static volatile int foo;
void bar (void) {
    foo = 0;
    while (foo != 255)
        ;
}
```

Além da possibilidade do conteúdo dos endereços mapeados no espaço de memória ser modificado pelo mundo externo, os registradores das portas podem ter funções diferentes às de armazenamento de dados como veremos na Seção 10.2.7. Para acessá-los de forma discriminada, precisamos ter conhecimento dos endereços em que o fabricante do microcontrolador mapeou. Estas informações de mapeamento podem ser encontrada no manual de referência do microcontrolador. O que um projetista precisa saber é como declarar estes endereços para o compilador de uma linguagem de programação de alto nível.

Na linguagem C, podemos tornar o nosso código mais legível redefinindo estes endereços por nomes simbólicos mais “inteligíveis” por meio de macros[3], como os seguintes endereços 0x6000030x dos registradores do microcontrolador ESP8266 [50] mapeados em macros GPIO_OUT_x através da diretiva “define” em C:
#define GPIO_OUT *(uint32_t *)0x60000300
#define GPIO_OUT_W1TS *(uint32_t *)0x60000304
#define GPIO_OUT_W1TC *(uint32_t *)0x60000308

Nas instruções de atribuição, podemos substituir os endereços pelas macros, como na atribuição do valor 0b00000010 no endereço 0x600000300:

[3] Macros são nomes simbólicos que são substituídos antes do código ser compilado.

GPIO_OUT = 0b00000010

As seguintes redefinições são as dos endereços de alguns registradores das portas do microcontrolador AVR [51,52]:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr)
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define PORTB _SFR_IO8(0x05)
#define PORTC _SFR_IO8(0x08)
#define PORTD _SFR_IO8(0x0B)
```

E, para MSP430, seguem-se as redefinições de alguns pinos das portas [53]:

```
#define P1IN *((byte *)0x20)
#define P1OUT *((byte *)0x21)
#define P4IN *((byte *) 0x1c )
#define P4OUT *((byte *) 0x1d )
```

10.2.4 Barramentos Embarcados

Nos microcontroladores (SoC) não só são importantes os componentes que são integrados neles como também a forma como eles são interligados. Diferentes dos sistemas computacionais de propósito geral modernos, em que as conexões entre a CPU e a memória são separadas das conexões entre a CPU e os periféricos de entrada e saída (Figura 4.22), a arquitetura do barramento ainda muito encontrada nos microcontroladores é a da primeira geração em que todos os dispositivos conectados ao microprocessador compartilham o mesmo barramento, de forma que a CPU precisa inserir os estados de espera, em inglês *wait states*, nas transferências de dados, ou baixar a frequência do seu relógio. Veremos na Seção 10.4 que, com a integração de um circuito de interrupção mais complexo do que vimos na Seção 4.5, os microcontroladores toleram melhor as discrepâncias nas velocidades entre os dispositivos conectados a um barramento.

Quatro principais arquiteturas de barramentos internos (Capítulo 9) para microcontroladores são *Advanced Microcontroller Bus Architecture* (AMBA) da ARM [4], *CoreConnect* da IBM [6], *Avalon* da Altera/Intel [7] e *Wishbone Bus Architecture* da *Silicore Corporation* [8]. Todos eles compartilham os objetivos de facilitar o reuso e compartilhamento de códigos proprietários e módulos de circuitos (*hardware*) proprietários, em inglês ***Intellectual Property (IP) cores***, de ter um alto desempenho, e de apresentar um baixo consumo de energia.

10.2.5 Controladores de Barramentos Externos e Globais Embarcados

Para que os controladores sejam de fato úteis, é necessário que eles se comunicam com o mundo ciber-físico. Vimos no Capítulo 9 que esta comunicação se dá através dos barramentos externos que, por sua vez, podem ser paralelos ou seriais. Vimos na Seção 9.5 que o protocolo de comunicação das interfaces paralelas são muito simples, similares às comunicações entre a CPU e os módulos de memória (Seção 5.6) em que podemos sintetizar os sinais de controle do receptor usando os sinais gerados pelo transmissor. Circuitos de gerenciamento de um protocolo de comunicação serial já não pode ser tão simples. Veja na Figura 9.35 que precisamos, no mínimo, ter um circuito de conversão paralelo-serial no transmissor e um circuito de conversão serial-paralelo no receptor.

Além das conversões, é necessário prover sinais de relógio ou sinais/*bits* de controle adicionais para assegurar que ambos os lados interpretem os sinais segundo o mesmo padrão de interface serial (Seção 9.8). Figura 10.10 ilustra o protocolo de comunicação serial síncrona I2C. Observe os detalhes das relações temporais dos sinais de relógio (SCL) e dos sinais de dados (SDA).

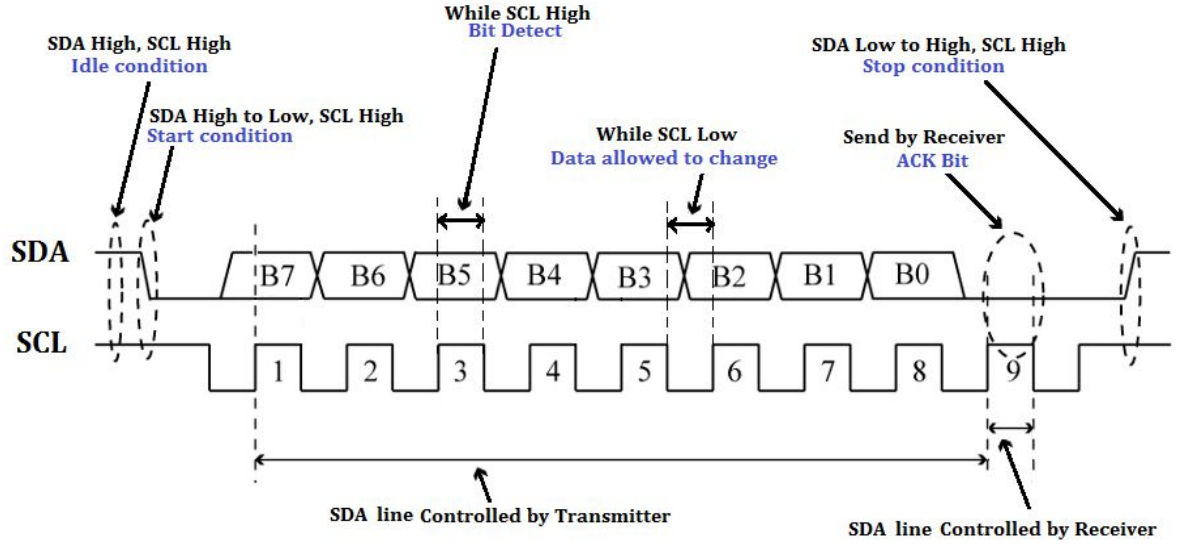


Figura 10.10: Protocolo de comunicação I2C (Fonte: [12]).

Uma das características dos microcontroladores modernos é terem integrados num mesmo *chip* diversos controladores para comunicações seriais, como UART (Seção 9.7.1), I2C (Seção 9.8.1) e SPI (Seção 9.8.3). Isso reduz o projeto de uma interface de comunicação serial com os periféricos na programação dos registradores de controle, de dado e de estado associados a cada módulo de controlador de um protocolo específico. Figura 10.11 mostra o conjunto de registradores de um

controlador da interface de comunicação I2C do AVR (Arduino). Através destes registradores, mapeados no espaço de memória do processador, consegue-se configurar as características elétricas e temporais dos sinais requeridos pelo periférico I2C conectado ao barramento com uso das mesmas instruções de acesso à memória.

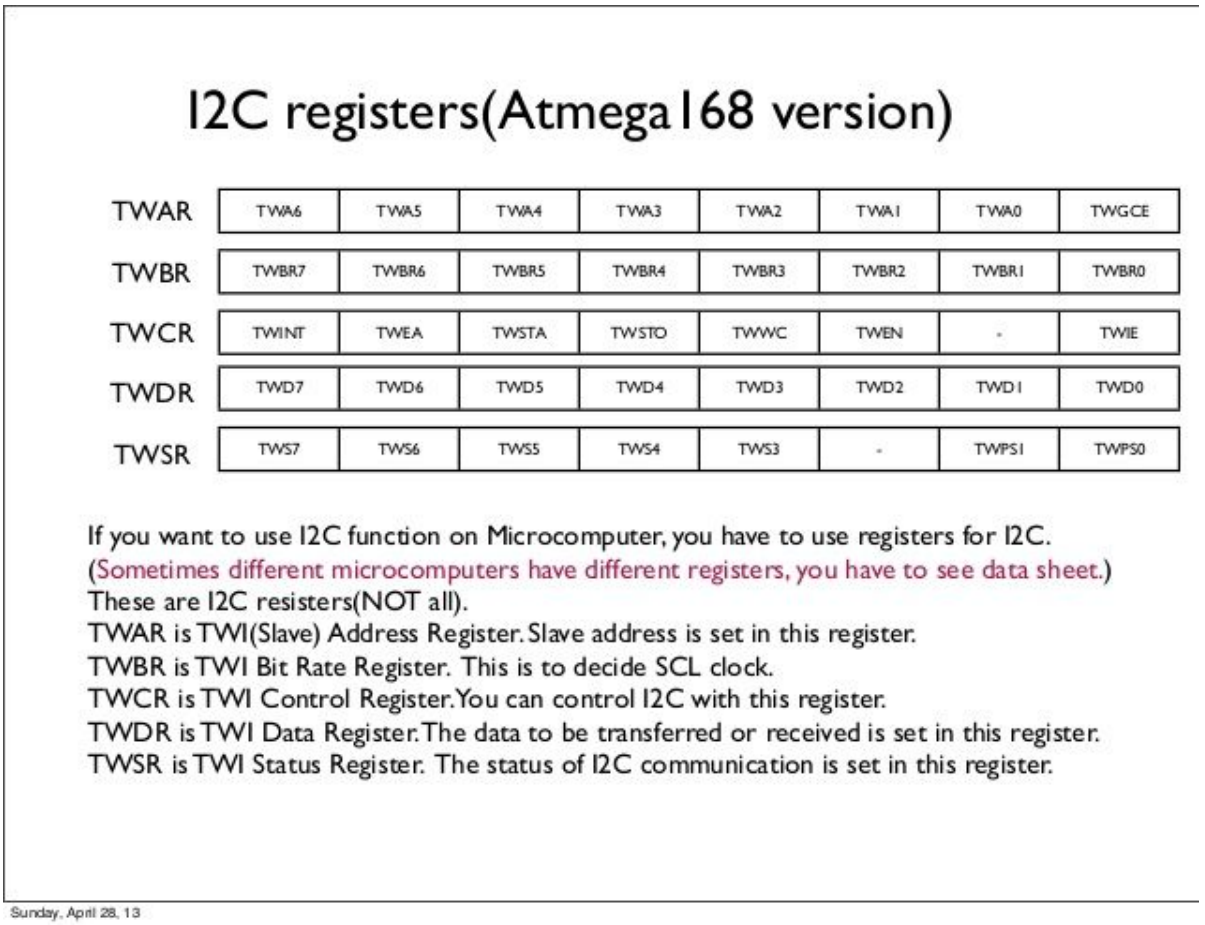


Figura 10.11: Registradores associados a um módulo I2C do AVR (Fonte: [14]).

Cada vez mais frequentes nos sistemas embarcados são os barramentos globais (Capítulo 13) que conectam uma diversidade de microcontroladores embutidos num dispositivo, como uma cadeira de rodas, um carro ou uma aeronave. Por exemplo, em indústrias automobilísticas temos o barramento *Controller Area Network* (CAN) [9] para conectar as unidades de controle eletrônico, em inglês *eletronic control units* (*ECUs*), e um outro mais novo FlexRay. Embora os fios/trilhas de conexões destes barramentos não se encontram integrados nos microcontroladores, os seus controladores são tipicamente embutidos para facilitar os projetos de conexões.

Vamos apresentar sucintamente quatro tipos de circuitos responsáveis pelos sinais seriais de comunicação comumente integrados nos microcontroladores.

10.2.5.1 UART

UART, acrônimo de *Universal Asynchronous Receiver/Transmitter* (**Receptor/Transmissor Universal Assíncrono**), é um circuito que transforma dados paralelos em dados seriais, e vice-versa, apropriados para serem transmitidos numa única linha de transmissão. O circuito realiza todas as funções necessárias para comunicações seriais, como a temporização dos sinais seriais e a verificação de erros de paridade, de forma que para transmitir por exemplo sinais seriais do padrão RS-232 basta acoplar um *driver* que transforma os níveis de tensão lógicos de saída do UART para níveis de tensão estabelecidos pelo padrão. A sequência de *bits* gerada depende do "acordo" de comunicação estabelecido entre um transmissor e um receptor. Como vimos na Seção 9.8.1, a sequência mais aplicada é a do protocolo *start-stop* NRZ[4] (Figura 9.48).

A comunicação entre um microprocessador e um UART é controlado por uma série de registradores [56]. Esses registradores, de 8 *bits*, podem ser acessados pelo microprocessador para monitorar e modificar o estado do UART. Nos *chips* modernos, como 16550, existem integrados não só um circuito de controle de acesso direto à memória (DMA) com também buffers de dados (64 *bytes*) do tipo FIFO (Capítulo 13) para acomodar melhor os fluxos de dados. As primeiras interfaces seriais compatíveis com os computadores pessoais foram implementadas em IBM XT com uso de 8250 UART.

10.2.5.2 Módulos I2C e SPI

Os módulos I2C e SPI são dois circuitos bem populares entre os microcontroladores modernos para transformarem dados paralelos em dados seriais, respectivamente, do padrão I2C (Seção 9.9.1) e SPI (Seção 9.9.3).

O protocolo I2C foi desenvolvido em 1982 pela Philips *Semiconductors*, hoje em dia NXP *Semiconductors*, objetivando a padronizar comunicações entre os *chips* de uma mesma placa. Não é necessário pagar nenhuma taxa para usar ou implementar o padrão I2C. É, porém, necessário, pagar uma taxa no registro de um endereço para uma linha de dispositivos. O endereçamento a um dispositivo se dá pela colocação do endereço (único até 7 *bits*) do dispositivo no barramento como mostra a Figura 10.12. O protocolo I2C só tem 2 linhas, SDA e SCL, para estabelecer comunicações seriais, síncronas e multi-*points*. Ambas as linhas requerem conexões com saídas dreno/coletor aberto, ou seja, com dispositivos capazes de acionar somente o nível lógico 0. Resistores *pull-up* são necessários para acionar o nível lógico 1 quando nenhum dispositivo estiver ativo, como vimos na Figura 9.59. O protocolo I2C suporta ***clock stretching*** entre um mestre rápido e um escravo lento, permitindo que o escravo force o sinal de relógio se manter no nível baixo (*bit* 9 na Figura 10.10) até que o escravo conclua a sua operação. Observe ainda que, por causa do possível *clock stretching*, é necessário

[4] Non-return-to-zero.

resincronizar o início de transmissão do próximo *byte* através das condições de START e STOP estabelecidas pelo protocolo.

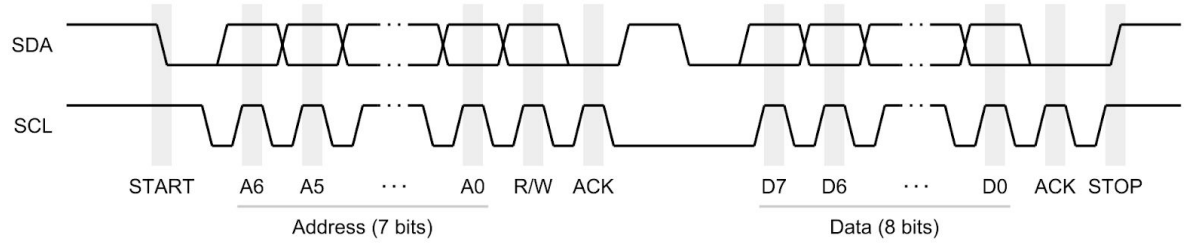


Figura 10.12: Sequência de *bits* estabelecida pelo protocolo I2C para transmissão de um *byte* (Fonte: [57]).

O protocolo SPI foi desenvolvido pela Motorola em 1980 para comunicações entre microcontroladores e os periféricos *onboard*, como as memórias EEPROM [57]. Ele não especifica os níveis de tensão elétricos, taxa de transmissão nem esquema de endereçamento. Como vimos na Seção 9.9.3, ele é um padrão “de facto” e a sua implementação pode variar entre os fabricantes. Detalhes de cada circuito devem ser consultados nas folhas de dados fornecidas pelo seu fabricante. Na Seção 9.9.3 mostramos que 4 linhas são usadas nas suas comunicações seriais síncronas *multi-drop*. A linha SS é usada pelo mestre para selecionar o escravo com quem ele pretende se comunicar. As conexões das linhas MOSI e MISO permitem formar um *buffer* circular *interchip*, como mostra a Figura 10.13, e os *bits* desse *buffer interchip* são deslocados por um sinal de relógio SCLK comum.

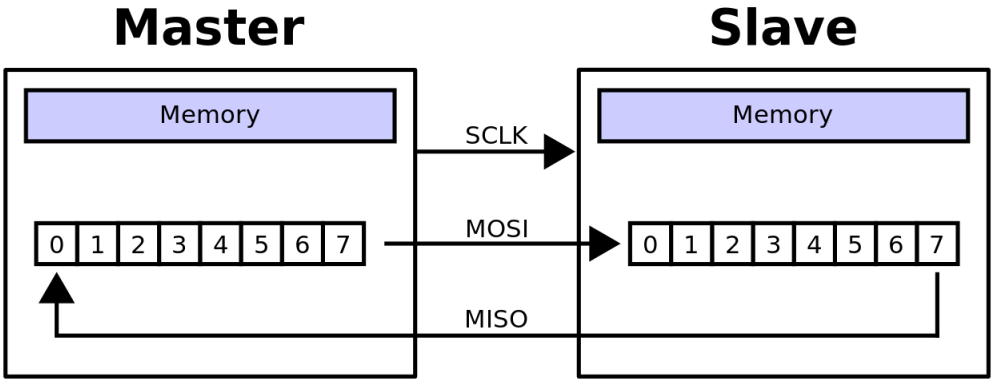


Figura 10.13: *Buffer* circular *interchip* em SPI (Fonte: [58]).

SPI suporta quatro modos de amostragem dos sinais de dados no receptor (Figura 10.14):

- Modo 0: dados amostrados (no meio do *bit*) na borda de subida do sinal de relógio, sinal de relógio ocioso no nível baixo.
- Modo 1: dados amostrados na borda de descida do sinal de relógio, sinal de relógio ocioso no nível baixo.

- Modo 2: dados amostrados na borda de descida do sinal de relógio, sinal de relógio ocioso no nível alto.
- Modo 3: dados amostrados na borda de subida do sinal de relógio, sinal de relógio ocioso no nível alto.

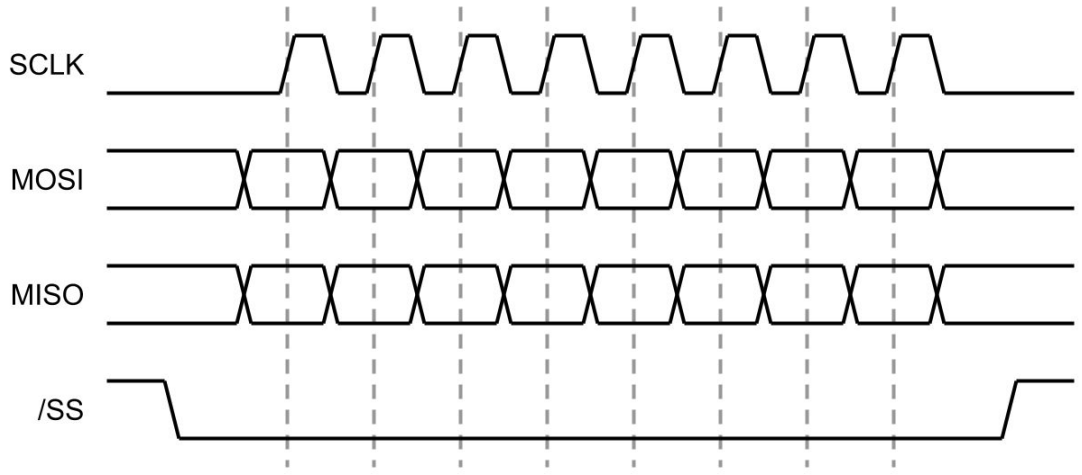


Figura 10.14: Relação temporal dos sinais do protocolo SPI.

Tabela 10.1 mostra comparativamente as características dos protocolos de comunicação I2C e SPI.

Tabela 10.1: I2C e SPI

I2C (NXP)	SPI (Motorola)
Protocolo serial multi-mestre e multi-escravo	Protocolo serial multi-escravo
Half-duplex	Full-duplex
Suporta *clock stretching*	Não suporta *clock stretching*
2 fios de transmissão (SDA e SCL)	3 a 4 fios (MOSI, MISO, SCL e *Chip select*)
Mais lento que SPI	Mais rápido que I2C
Consome mais energia que SPI	Consome menos energia que SPI
Menos susceptível aos ruídos que SPI	Mais susceptível aos ruídos que I2C
Mais barato que SPI	Mais caro que I2C
Conexão em Lógica-OU com *pull-up*	Não há resistor *pull-up*
Bit ACK para confirmar a recepção	Não há ACKnowlegment para verificação
Topologia multi-point	Topologia multi-drop
Há overhead: *start* e *stop bits*	Não há overhead de *bits*
Seleção de escravo por endereçamento	Selação de escravo por uma linha de seleção
Distância de transmissão maior do que SPI	Distância de transmissão menor do que I2C

10.2.5.2 USART

USART, acrônimo de *Universal Synchronous/Asynchronous Receiver/Transmitter* (**Receptor/Transmissor Universal Síncrono/Assíncrono**), é também conhecido como **Interface de Comunicação Serial**, em inglês *Serial Communication Interface* (***SCI***). É um

circuito capaz de converter os dados paralelos em dados seriais em ambos os modos de transmissão serial. No modo assíncrono o circuito opera de forma similar ao circuito UART, no modo de comunicação *full-duplex*. E no modo síncrono, ele pode gerar sequências de sinais compatíveis com diferentes protocolos de comunicação, como I2C, SPI, Smart Card [59]. Figura 10.15 apresenta um esboço dos dois modos de transmissão serial pelo circuito USART [60].

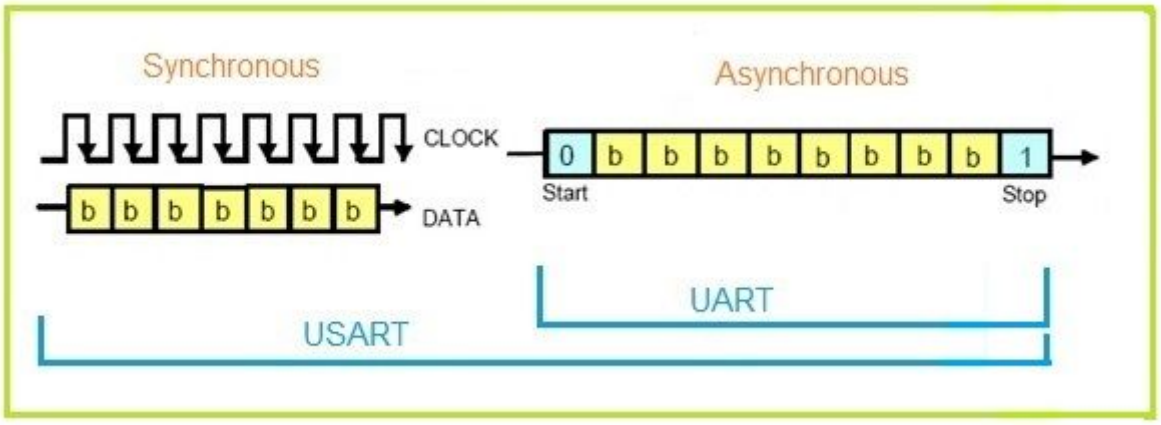


Figura 10.15: UART e USART (Fonte: [60]).

10.2.6 Circuitos de Depuração

(baseado em [11])

Mencionamos na Seção 9.7.5 uma série de funções da interface serial JTAG, entre as quais está a lógica de depuração em microprocessadores e microcontroladores. A interface JTAG é o padrão IEEE *Standard Test Access Port and Boundary Scan Architecture* que nos permite verificar o funcionamento de vários pontos dos circuitos embutidos num SoC. Originalmente, foi concebida para "varrer a borda", em inglês *boundary scan*, de um circuito impresso (PCB) de forma a permitir monitorar e controlar os sinais nos pinos de E/A e testar a conectividade entre os dispositivos conectados nele [62] (Figura 10.16(b)). Hoje em dia, a interface JTAG é embutida em todos os microcontroladores para facilitar acessos aos seus circuitos internos, controlar *hardware* e depurar os códigos, embora nem todos os circuitos integrados sejam compatíveis com JTAG.

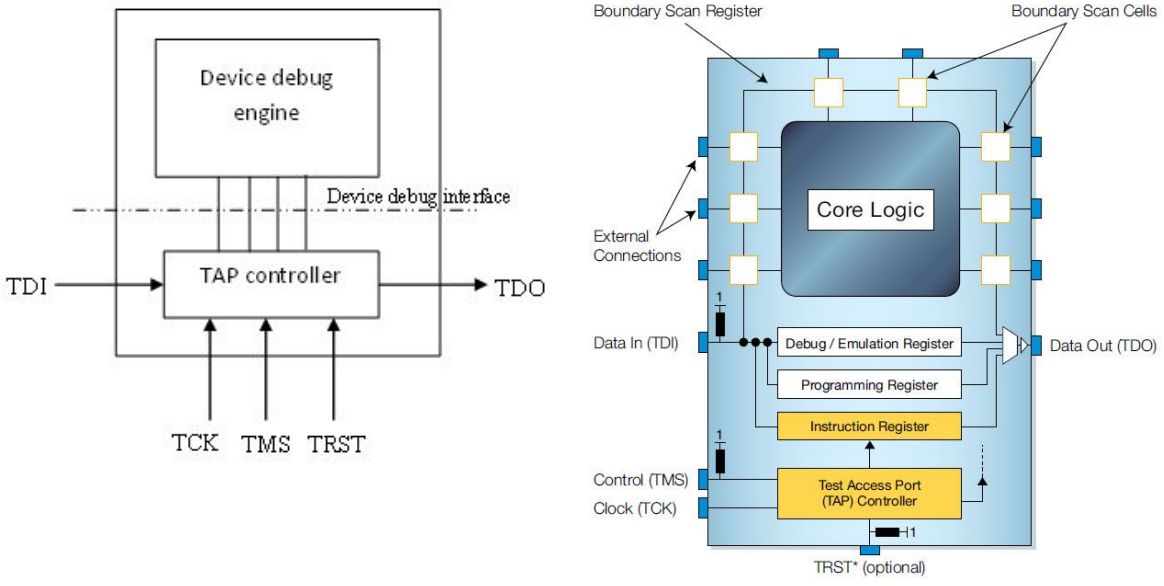
Através da interface JTAG e um programador apropriado, pode-se "programar" a memória de um microcontrolador e controlar, por *software* específico, os pulsos de relógio do seu processador. Isso permite que o projetista insira pontos de parada, em inglês *breakpoints*, ao longo da execução de um código, reinicie, pause e interrompa a execução do código no *hardware* em qualquer instrução para verificar o estado do sistema. Entre os exemplos do mecanismo de depuração podemos citar EJTAG (MIPS32) e OCI (8051) [11]. Vale comentar que o padrão JTAG é bem flexível quanto aos detalhes de implementação. Por isso, há uma grande variedade

de implementações de JTAG para poder atender as demandas específicas de cada fabricante dos *chips*.

O padrão define que a interface de JTAG contenha 5 sinais acessíveis através dos pinos associados à **porta *Test Access Port*** (TAP):

- *Test data input* (TDI)
- *Test data output* (TDO)
- *Test mode select* (TMS)
- *Test clock input* (TCK)
- *Test reset input* (TRST*) - opcional

O controlador TAP inclui um circuito que controla qual mecanismo de depuração deve ser utilizado para depuração, como ilustra a Figura 10.16(a). O controlador TAP tem um registrador de instrução (IR) e um ou mais registradores de dados (DR) acessíveis pelos seus 4 *bits* para controlar o fluxo de monitoramento e monitorar o estado de execução de um programa. Como nos outros módulos, esses registradores são mapeados no espaço de endereços da memória nos microcontroladores modernos.



(a) Interface JTAG (b) escaneamento pela borda

Figura 10.16: Controlador TAP e a interface de depuração (Fonte: [11]).

Em alguns *kits* de desenvolvimento em que os microcontroladores são implementados, como FRDMKL25Z [20], pode-se acessar diretamente a interface JTAG para “programar” a memória de instruções do microcontrolador através do ambiente integrado de desenvolvimento, em inglês *Integrated Development Environment* (*IDE*), disponível. Isso é porque o adaptador JTAG já se encontra

integrado no *kit* de desenvolvimento. Em outros *kits*, como o Arduino [18], ESP-MSP430 [19] e e NodeMCU [21], precisa-se de um dispositivo adicional, respectivamente, JTAG *In-Circuit Emulator* (ICE) (Figura 10.17), eZ430-RF2500 (Figura 10.18) e ARM-USB-OCD-H JTAG (Figura 10.19), para programar e depurar os códigos nas memórias de instruções via a interface JTAG.

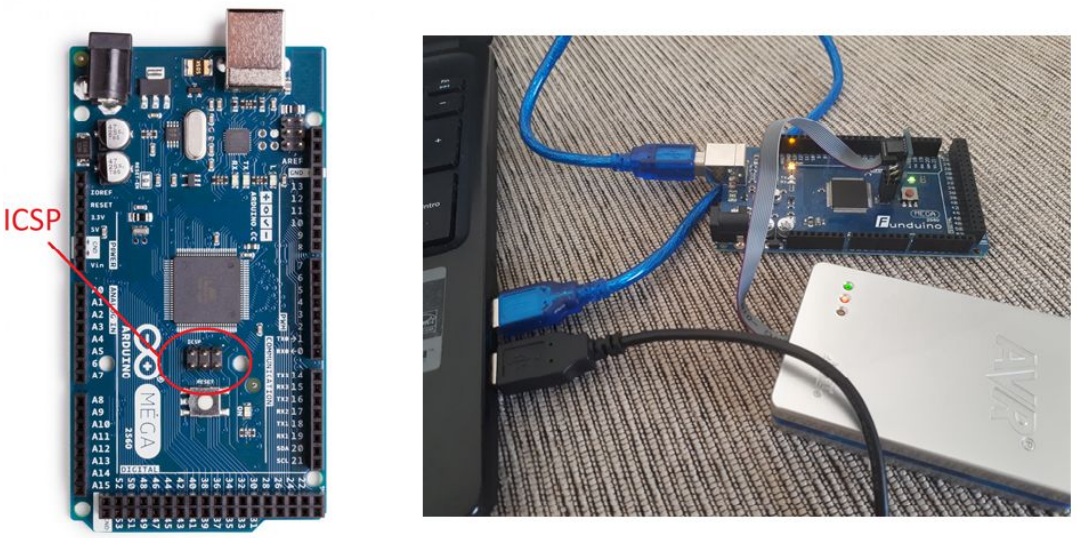


Figura 10.17: Depuração do AVR via JTAG ICE (Fonte: [18]).

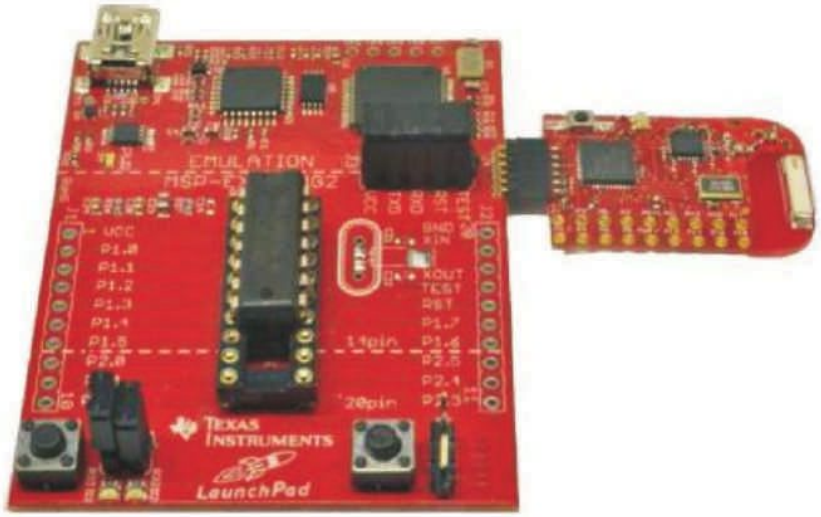


Figura 10.18: Depuração do MSP430 via eZ430-RF2500 (Fonte: [10]).

Figura 10.19: Depuração do ESP8266 com Olimex ARM-USB-OCD-H (Fonte: [61]).

10.2.7 Registradores de Função Especial

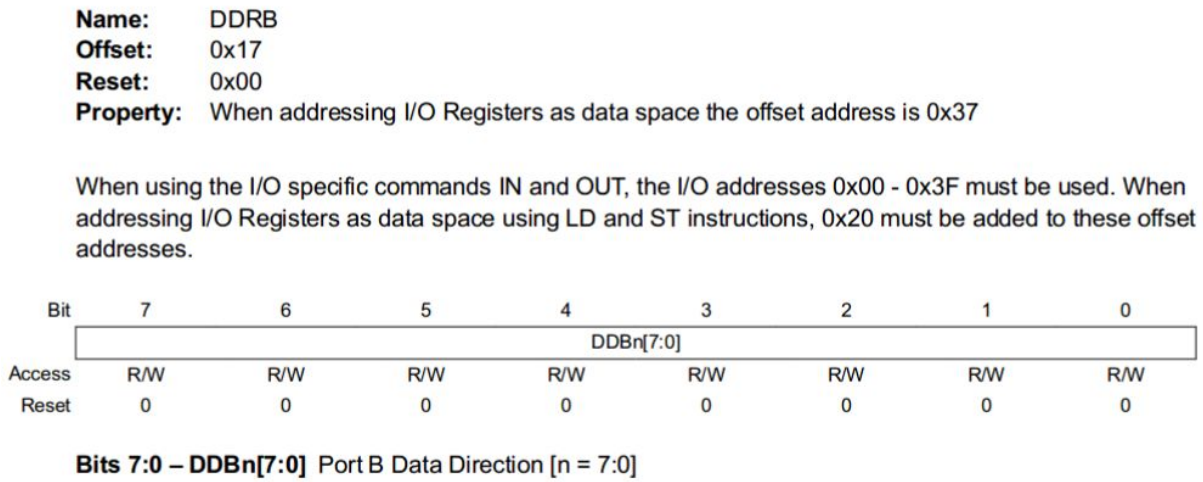
Vimos no Capítulo 4 que os microprocessadores tem um conjunto de **registradores de propósito geral** onde são armazenados os operandos e o resultado de uma operação lógico-aritmética. Nos microcontroladores, as funções dos periférico, circuitos de interface ou circuitos adicionais embutidos no *chip* do microcontrolador são mapeados em diferentes campos dos seus registradores de controle como podemos ver na Figura 10.12. Esses registradores são denominados os **registradores de função especial**, em inglês *special function registers* (SFR). Eles são mapeados no espaço de memória do microcontrolador e são processados como dados de uma memória.

10.2.7.1 Exemplos

Para otimizar o uso do espaço de memória, que é um recurso bem escasso em microcontroladores, é muito comum compactar num registrador de função especial vários campos de *bits* de controle. Sendo esses registradores mapeados no espaço de memória, a configuração passa a ser simples operações de escrita nos *bits* de interesse. Vamos ilustrar as operações, *bit* a *bit*, nos registradores de função especial para configurar as portas de três diferentes microcontroladores.

Nos microcontroladores da família MSP430x2xx, cada *bit* do registrador PxREN (8 *bits*) habilita ou desabilita o resistor *pullup*/*pulldown* do pino correspondente da porta x [48]. Se o valor do *bit* for 0, o resistor é desabilitado, do contrário, ele é habilitado. Para configurar a forma de conexão do resistor, *pullup* ou *pulldown*, há um outro registrador PxOUT. Quando o *bit* é 0 em PxOUT, o pino é "*pulled down*", senão ele é "*pulled up*". Por exemplo, se escrevermos 0x22 no registrador P1REN, somente os pinos 1 e 5 terão seus resistores *pullup*/*pulldown* habilitados. E se escrevermos 0x02 no registrador P1OUT, o pino 1 será '"*pulled up*" e o pino 5, "*pulled down*".

O registrador DDRB do microcontrolador AVR é o registrador de direção dos sinais da porta B [42]. Ele permite configurar a direção do sinal de cada um dos 8 pinos individualmente. Quando o *bit* é 1, o pino é configurado como pino de saída; do contrário, ele é um pino de entrada. Por exemplo, se escrevermos 0x10 em DDRB, todos os pinos da porta B serão pinos de entrada exceto o pino 4 que operará como saída (Figura 10.20).



Name: DDRB
Offset: 0x17
Reset: 0x00
Property: When addressing I/O Registers as data space the offset address is 0x37

When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these offset addresses.

Bit	7	6	5	4	3	2	1	0
	DDBn[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDBn[7:0] Port B Data Direction [n = 7:0]

Figura 10.20: Registrador de controle de direções dos pinos da porta B no AVR (Fonte: [42]).

O microcontrolador ESP8266 tem 11 pinos de entrada e saída de propósito geral, correspondentes aos bits 0-5 e 12-16. Os *bits* 6-11 são reservados para comunicações com a memória FLASH embarcada no microcontrolador. Os pinos são multiplexados. A cada pino é associado um registrador de controle de 16 *bits*. Este é, por sua vez, dividido em vários campos de *bits* de controle. Por exemplo, os *bits* 4 e 5 do registrador são para configurar a função do pino, o *bit* 7 é para habilitar ou desabilitar *pullup*, e o *bit* 6, pulldown. Se quisermos habilitar o resistor *pullup* do pino 2, precisamos setar (em 1) somente o *bit* 7 do registrador PERIPHS_IO_MUX_GPIO2_U, mapeado no endereço 0x60000838 [50]..

10.2.7.2 Programação em C

Na Seção 10.2.7.1 vimos que, embora o tamanho e a definição dos registradores de função especial variam entre os microcontroladores, o princípio é o mesmo. Essencialmente, quando desejamos alterar somente um campo de *bits* de controle, não podemos simplesmente fazer atribuição do valor daquele *bit* ao registrador, como habilitar o resistor *pullup* do pino 2 em ESP8266 pela instrução:

PERIPHS_IO_MUX_GPIO2_U = 0x0080,

pois estaremos zerando todos os outros *bits* de controle modificando a operação do pino 2. Precisamos, então, **mascarar** os *bits* 0-6 e 8-15 que não sejam de interesse para que a atribuição não afete estes *bits*. Como podemos alcançar este efeito?

Essencialmente, temos 3 tipos de mascaramento para manipular o conteúdo de um endereço a nível de *bits*:

- AND *bit* a *bit* para extrair o conteúdo de um registrador. Por exemplo,
  máscara: 0b00011000
  valor: 0b11010101
  resultado: 0b00010000
  Ao deslocarmos o resultado 3 *bits* para direita, obteremos o valor do campo, 0b00000010 (3)
- OR *bit* a *bit* para setar o conteúdo de um registrador. Por exemplo, se zerarmos primeiro o conteúdo do campo de interesse com o inverso da máscara acima (NOT bit a bit) com uma operação AND *bit* a *bit* seguida de OR com o novo valor que desejamos atribuir
  máscara invertida: 0b11100111
  valor : 0b11010101
  resultado 1 : 0b11000101
  novo valor : 0b00001000 (= 0b01 deslocado de 3 *bits* para esquerda)
  resultado : 0b11001101
- XOR *bit* a *bit* para alternar o estado dos *bits*, lembrando que 1 XOR 0 = 1, 0 XOR 1 = 1, 1 XOR 1 = 0, 0 XOR 0 = 0,
  máscara : 0b00011000
  valor : 0b11010101
  resultado : 0b11001101

A pergunta é como transmitir essas ideia para uma máquina através da linguagem C. Tabela 10.2 mostra os operadores, *bit* a *bit*, em C. Usando esses operadores, a tradução é quase direta.

Tabela 10.2: Operadores lógicos em C

Operador em C	Descrição	Mnemônico em a*ssembly*
&	E lógico bit a bit.	AND
|	OU lógico bit a bit.	ORR
^	OU exclusivo (XOR) *bit* a *bit*.	EOR
~	Complemento bit a bit.	MVN
<<	Deslocamento para esquerda.	LSL
>>	Deslocamento para direita.	LSR

Vamos colocar em prática os seguintes casos:

- Setar 1 na posição *n* de uma palavra: fazer um deslocamento de *n* bits do valor 01

  1 << n.

  A ideia é análoga para resetar o valor do *bit* em 0.
- Setar 1...1 em *m bits* deslocados de *n bits* do bit *0* do registrador Reg

  Reg = Reg | (1...1 <<n) equivalente a Reg |= (1...1 << n).

  Análogo para resetar em 0...0
- AND *bit* a *bit* para extrair o conteúdo de um registrador a partir do bit *m* de um registrador. No exemplo acima, podemos traduzir em
  resultado = (máscara & valor) >> m = (0b00011000 & 0b11010101) >> 3
- OR *bit* a *bit* para setar o conteúdo y a partir do *bit* m de um registrador. Para o exemplo acima,
  resultado = (~máscara & valor) | (y << m)
  = (0b11100111 & 0b11010101) | (0b01 << 3)
- XOR *bit* a *bit* para alternar o estado dos *bits*. Para o exemplo acima, podemos usar a seguinte instrução em C
  resultado = 0b00011000 ^ 0b11010101.

10.3 Temporizadores Embarcados

(baseado em [16])

Vimos na Seção 4.6 que existem circuitos integrados de temporização, mas são muito modestas as suas aplicações. Especificamente na Seção 4.6, vimos que o temporizador foi aplicado na emissão de *beeps* durante o *boot* de um sistema computacional. A menos que se esteja preocupado com a avaliação do desempenho temporal de um programa, dificilmente alguém se ocuparia em

determinar quantos microsegundos foram gastos na execução de uma sequência de instruções. Uma das características notáveis dos microcontroladores, que os distinguem dos sistemas computacionais convencionais, é a sua “intimidade” com o tempo para poder interagir com o mundo físico em instantes pré-estabelecidos. Todos os microcontroladores têm ao menos um sistema de temporização embarcado no seu *chip*.

O principal componente de um sistema de temporização, em inglês *timer*, embarcado é um **contador de corrida livre**, em inglês *free running counter*, de forma totalmente independente da CPU. Esse contador é incrementado ciclicamente de 0 até 2^n-1 por um sinal de relógio, de forma que se consegue computar um intervalo de tempo t pela quantidade *m* de contagem e pelo período fixo T do sinal de relógio:

$$t = m \times T .$$

Sendo a contagem máxima de um sistema de temporização dependente da quantidade de *bits* do seu contador, é comum especificar um temporizador por esta quantidade. Figura 10.21 ilustra um sistema de temporização de 16 *bits*. O sinal de *Overflow* indica que o contador atingiu o valor máximo e o contador volta para 0. Esse sinal pode ser usado para contar a quantidade k de ciclos completos contados.

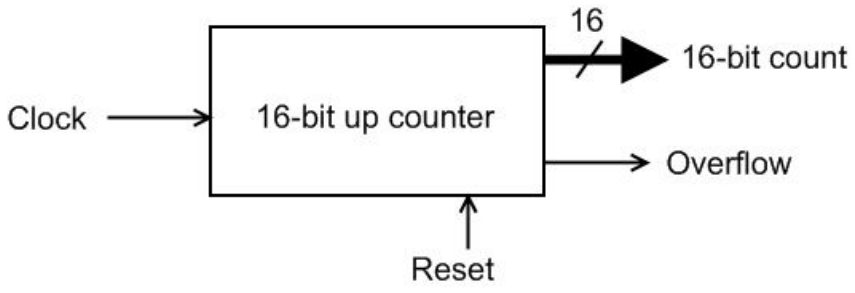


Figura 10.21: Componente básico num *timer*: contador (Fonte: [15]).

Supondo que sejam *i* o valor inicial registrado inicialmente num contador de *n* bits e *j* o valor final de uma contagem, podemos determinar com precisão a quantidade total dos pulsos contados através das seguintes relações:

- se o número de ciclos completos for 0,

  m = j-i pulsos

- se o número de ciclos completos for maior que 0,

  $(2^n - 1 - i) + j + 1 = 2^n - i + j$ pulsos além dos ciclos completos.

  Somando com os pulsos dos (k-1) ciclos completos, temos no total

  $m = (k-1)(2^n) + 2^n - i + j = 2^n(k) - i + j$ pulsos.

Em diversos *timers*, a sua fonte do sinal de relógio é configurável. Ela pode ser o mesmo sinal de relógio do microprocessador, ou um derivado deste sinal. A técnica

de **divisor de frequência** é a mais utilizada para gerar sinais de frequências menores. A fonte do sinal de relógio do contador pode ser também externa. Neste caso, a frequência de contagem pode ser até aperiódica. Visando à redução do consumo de energia, deve-se parar o contador do *timer* quando ele não é mais necessário. Adicionalmente, os temporizadores são projetados para gerarem interrupções ou *ticks* com base nos eventos especificados. Exemplos destes eventos são “estouro” ou “contagem atingir um valor pré-definido”. Isso torna viável, por *software*, medir o intervalo de tempo entre dois eventos, ou controlar precisamente o tempo de duração do estado de um dispositivo.

Um temporizador que existe em todos os microcontroladores é o **temporizador *watchdog***, em inglês *watchdog timer*. Este temporizador é responsável por disparar um *reset* ao sistema quando ocorre um estouro na sua contagem. Isso garante que o sistema consiga sair de uma condição de falha ou de erro numa emergência. Por isso, quando esse temporizador estiver ativado precisaremos periodicamente zerar o seu contador para evitar *resets* emergenciais.

Figura 10.22 mostra os registradores de controle, de dados e de estado, associados ao sistema de temporização BCM2835 embutido no microcontrolador Raspberry Pi. Os endereços do espaço de memória em que eles estão mapeados são mostrados na primeira coluna. Observe que dois registradores de 32 *bits* foram usados para acessar o conteúdo de um contador de corrida livre de 64 *bits*.

BCM2835 System Timer registers

Offset	Name	Description
+0x00	CS	System Timer Control and Status
+0x04	CLO	System Timer Counter Lower 32 bits
+0x08	CHI	System Timer Counter Upper 32 bits
+0x0C	C0	System Timer Compare 0; corresponds to IRQ line 0.
+0x10	C1	System Timer Compare 1; corresponds to IRQ line 1.
+0x14	C2	System Timer Compare 2; corresponds to IRQ line 2.
+0x18	C3	System Timer Compare 3; corresponds to IRQ line 3.

Figura 10.22: Registradores associados ao temporizador BCM3835 (Fonte: [17])

10.4 Controlador de Interrupções Embarcado

Diferentemente dos computadores de uso genérico, os microcontroladores usam

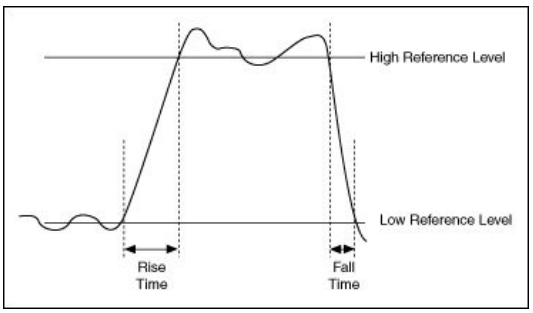
intensamente as suas portas de entrada e de saída para interagir com os outros componentes no sistema em que ele está embutido. Através dos periféricos de entrada, o microcontrolador consegue coletar, via os seus pinos, os dados necessários para processamento, e, através dos periféricos de saída, os resultados do processamento podem ser exibidos ou usados diretamente em atuadores. Como processadores tem uma velocidade de processamento muito maior do que a dos periféricos, o grande desafio é como compatibilizar temporalmente os dois componentes sem sacrificar o desempenho do primeiro e sem postergar a comunicação dos sinais com o segundo quando este estiver pronto para tal. Por exemplo, para detectar a mudança do estado de uma botoeira que depende da ação de um usuário que ocorre esporadicamente, será que o processador precisa executar periodicamente instruções de amostragem (*polling*) só para verificar o estado da botoeira?

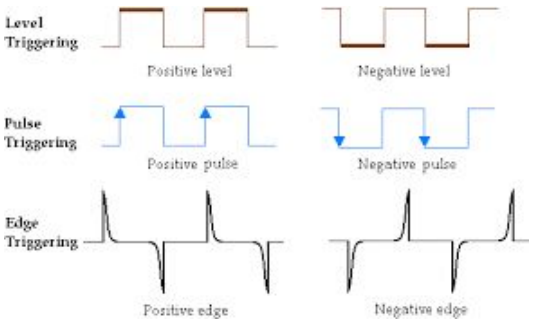
10.4.1 Fluxo de Controle

O mecanismo de interrupção implementado nos microcontroladores é uma solução eficiente para compatibilizar um microcontrolador com o mundo físico. Ele provê uma forma eficiente tanto para capturar as variações "inesperadas" nos estados dos periféricos quanto para respondê-las, desviando automaticamente do fluxo corrente de execução para uma **rotina de serviço de interrupção** pré-definida (*Interrupt Service Routine* - ISR). Os sinais assíncronos (em relação ao relógio do sistema) oriundos dos periféricos e capazes de interromper o fluxo de execução corrente são chamados de **requisições de interrupções**, em inglês *interrupt request* (*IRQ*). Podemos distinguir 3 formas de ativação de interrupção, conforme as características desses sinais detectáveis pelos circuitos eletrônicos (Figura 10.23):

- Gatilhos por nível: quando a ativação é por nível lógico (0 ou 1) de um pulso com uma largura mínima em tempo.
- Gatilho pela borda: quando a ativação é pela borda (de subida ou de descida) de um pulso quadrado.
- Gatilho pela aresta: quando a ativação é por um pulso rápido (de subida ou de descida).

Portas (de entrada) que suportam interrupções incluem nos seus circuitos registradores de controle que permitem customizar o tipo de gatilho ao qual os pinos são sensíveis. O intervalo de tempo entre o instante em que ocorre uma interrupção e o instante em que se inicia a execução da rotina de serviço correspondente é conhecida por **latência de interrupção**.





(a) Bordas e níveis (Fonte: [25]) (b) Tipos de gatilhos (Fonte: [63])

Figura 10.23: Gatilhos de interrupções.

Ao interromper um fluxo de execução, é esperado que após a execução de uma rotina de serviço o processador retorna exatamente ao estado em que se encontrava. Para assegurar o retorno ao ponto em que ocorreu o desvio após a execução de uma ISR, deve-se salvar o estado corrente da CPU (os registradores e as variáveis) e o endereço de retorno do sistema antes de carregar no contador de programa o endereço da ISR. Estes dados são tipicamente empilhados na pilha do sistema como no AVR (Arduino). Voltaremos à estrutura pilha no Capítulo 12. Nos microcontroladores ARM, o endereço de retorno é salvo num registrador especial denominado ***link register***, a fim de melhorar o desempenho. Além disso, o processador salva automaticamente todos os dados antes do desvio. Figura 10.24 ilustra o fluxo de atendimento de uma interrupção. Observe que o desvio está condicionado a dois eventos: requisição de interrupção e habilitação da interrupção.

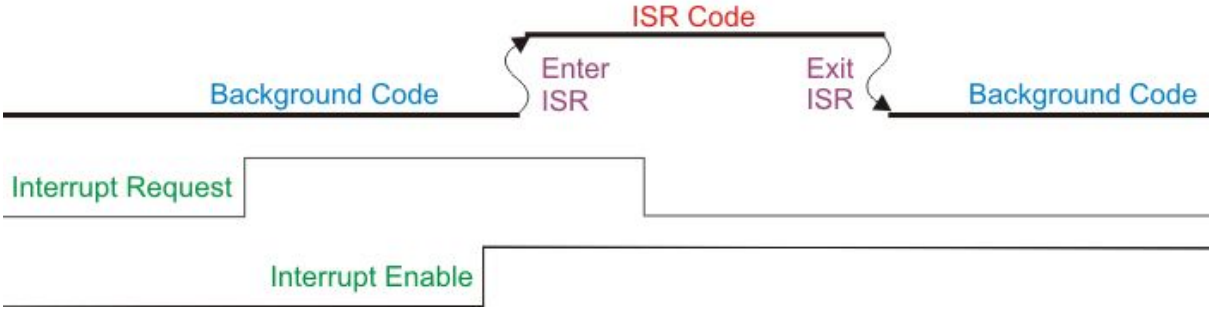


Figura 10.24: Desvio e retorno ao fluxo de controle (Fonte: [27]).

Um microcontrolador pode ter centenas de fontes de interrupção. Quando há mais de uma interrupção solicitando atendimento do microprocessador, elas são atendidas de acordo com o seu nível de prioridade (de atendimento) pré-fixado ou configurado pelo projetista. Em alguns microcontroladores, a próxima interrupção só será atendida depois de concluído um atendimento. Em outros, que suportam interrupções aninhadas, em inglês ***nested vector interrupt*** (***NVI***), o processamento de uma rotina de serviço pode ser interrompido por uma nova interrupção de prioridade maior, assegurando que o atendimento das interrupções de prioridade maior seja sempre concluído antes das outras solicitações. Figura 10.25 ilustra o

processamento de uma IRQ pelo controlador NVI, em inglês ***Nested Vector Interrupt Controller*** (**NVIC**), embutido num microcontrolador ARM. Através dos valores dos *bits* de controle SETPEND, CLRPEND decide-se se a solicitação deva ficar pendente ou prosseguir no processamento. Os *bits* de controle SETENA e CLRENA são os que definem se a solicitação deva ser habilitada ou não. Caso seja habilitada, verifica-se ainda a prioridade de atendimento para então, finalmente ser ativada e processada.

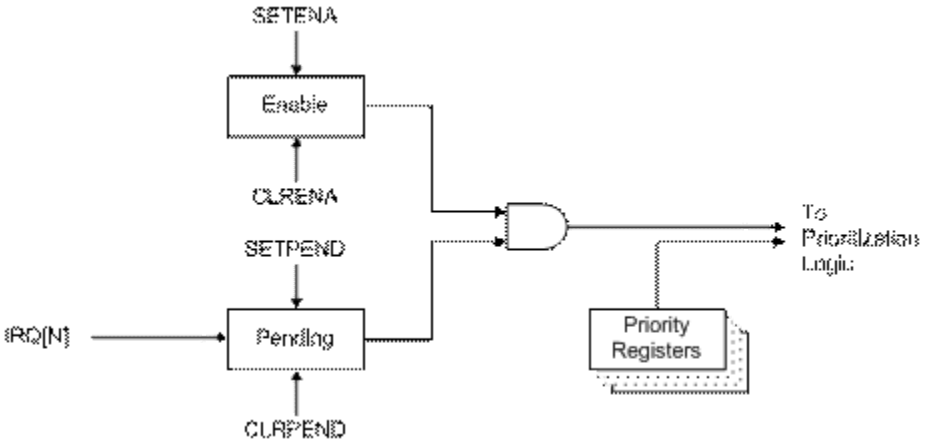


Figura 10.25: Processamento de uma IRQn no NVIC (Fonte:[23]).

Nem sempre uma única interrupção é associada a uma rotina de serviço. Quanto há mais de uma interrupção tratável por uma mesma rotina de serviço, é necessário identificar a fonte da interrupção depois de tê-la detectada. Figura 10.26 mostra multi-requisições processadas como uma única requisição *n* pelo controlador NVI do processador ARM Cortex-M3.

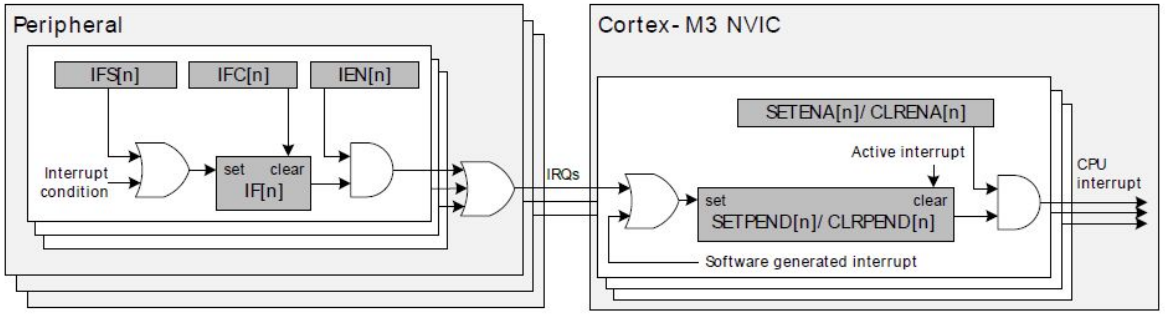


Figura 10.26: IRQ geradas pelos periféricos e processadas como uma única requisição (Fonte: [24]).

Finalmente, vale comentar que nos microcontroladores modernos um evento só consegue gerar uma interrupção se ele estiver habilitado para tal. Isso não só reduz o consumo de energia como também evita que eventos inesperados gerem interrupções e não terem rotinas de tratamento apropriadas associadas. Figura

10.24 ilustra o desvio do fluxo de controle para uma rotina de serviço somente quando ambos sinais, IRQ e IE (*Interrupt Enable*), estiverem ativos.

10.4.2 Programação em C

Note que há dois pontos no processamento de uma interrupção altamente dependentes da aplicação: a prioridade de atendimento e o processamento esperado. De fato, estas informações precisam ser configuradas **A PRIORI** pelo projetista para que a execução flua suavemente sem intercorrências. O projetista precisa escrever o código de tratamento do evento de interesse (ISR x) de um evento x e setar a sua prioridade de atendimento num registrador de controle. A pergunta que se faz agora é como um microcontrolador consegue sistematicamente relacionar as ISR x programadas e as prioridades definidas com os eventos detectados? A solução mais aplicada é atribuir a cada evento processável um **número de exceção**, em inglês *exception number*. Nos dispositivos modernos, esse número de exceção é fornecido pelo próprio dispositivo solicitante. Dizemos que é uma **interrupção vetorizada**. Quando o dispositivo não consegue fornecer a sua identificação, cabe ao controlador prover este número. Neste caso, dizemos que é uma **interrupção auto-vetorizada**.

O ponto-chave do processamento de uma solicitação de interrupção IRQ de maior prioridade é conseguir desviar o fluxo de controle para o início do segmento de instruções (ISR) pré-programado pelo projetista. Este procedimento equivale a buscar da memória o endereço inicial de ISR e carregá-lo no contador de programa (PC). Para isso, precisa-se ter armazenadas na memória as instruções de ISR, saber a priori o endereço inicial de ISR ou ter o endereço inicial de ISR armazenado em alguma palavra da memória antes de iniciar o processamento de interrupções e saber onde está armazenado o endereço do ISR. O local onde se armazena os dados necessários para iniciar a execução de uma ISR é estabelecido pelo fabricante do microcontrolador.

Dependendo da quantidade *N* de interrupções que um microcontrolador suporta é reservado na memória um espaço de *N* dados, cada um correspondente à informação necessária para realizar o desvio. Os dados podem ser uma instrução de desvio ou um endereço que o processador automaticamente carrega no PC [26]. O espaço de memória onde são armazenados estes dados é conhecido por **tabela de vetores de interrupção**, em inglês *interrupt vector table*. Assim, a partir do número de exceção x do evento identificado pelo controlador, consegue-se acessar na tabela o endereço da sua ISR. Figura 10.27 ilustra a tabela de vetores de interrupção do microcontrolador ATmega328P. As colunas 1 e 3 contém, respectivamente, o número do vetor e o endereço da rotina de serviço representado pelo nome da rotina de serviço. É importante frisar que a busca pelo endereço da

rotina de serviço e o seu carregamento no contador de programa (PC) são automáticos. Portanto, é crucial que o projetista assinale na tabela de vetores de interrupção endereços válidos de ISRs. Do contrário, o fluxo de controle pode ser desviado para um endereço fisicamente inexistente ou com conteúdo totalmente inválido.

Table 11-1. Reset and Interrupt Vectors in ATmega328P

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset, brown-out reset and watchdog system reset
2	0x002	INT0	External interrupt request 0
3	0x0004	INT1	External interrupt request 1
4	0x0006	PCINT0	Pin change interrupt request 0
5	0x0008	PCINT1	Pin change interrupt request 1
6	0x000A	PCINT2	Pin change interrupt request 2
7	0x000C	WDT	Watchdog time-out interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 compare match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 compare match B
10	0x0012	TIMER2 OVF	Timer/Counter2 overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 capture event
12	0x0016	TIMER1 COMPA	Timer/Counter1 compare match A
13	0x0018	TIMER1 COMPB	Timer/Counter1 compare match B
14	0x001A	TIMER1 OVF	Timer/Counter1 overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 compare match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 compare match B
17	0x0020	TIMER0 OVF	Timer/Counter0 overflow
18	0x0022	SPI, STC	SPI serial transfer complete
19	0x0024	USART, RX	USART Rx complete
20	0x0026	USART, UDRE	USART, data register empty
21	0x0028	USART, TX	USART, Tx complete
22	0x002A	ADC	ADC conversion complete
23	0x002C	EE READY	EEPROM ready
24	0x002E	ANALOG COMP	Analog comparator
25	0x0030	TWI	2-wire serial interface
26	0x0032	SPM READY	Store program memory ready

Figura 10.27: Tabela de vetor de interrupção do ATmega328P (Fonte: [64])

Essencialmente, a tarefa do projetista consiste em escrever no elemento x da tabela de vetores de interrupção o endereço da rotina de serviço que ele programou. Em muitos ambientes de desenvolvimento, estes endereços já são “pré-estabelecidos” na tabela e a tarefa do projetista se reduz a programar a rotina de tratamento ISR adequada no endereço pré-estabelecido. O elo de ligação entre uma ISR programada e o endereço escrito na tabela de vetores de interrupção pode ser delegado ao compilador e ao ligador de C se estivermos programando em C. Basta inserirmos na tabela de vetores de interrupção o mesmo nome da função ISR implementada, o compilador vai entender que são as mesmas variáveis e instruir o ligar para conectá-los. Por este motivo, nos ambientes em que os endereços já são

pré-estabelecidos na tabela de vetores de interrupção, é mandatório o uso de nomes de ISR dados nos manuais de uso.

A programação de ISR é uma programação concorrente no sentido de que trechos de códigos de ISR distintos possam ser executados concorrentemente como mostra a Figura 10.28 em que várias ISR podem ser interrompidas para dar lugar ao evento de maior prioridade. É fundamental assegurar que os recursos compartilhados por diferentes trechos de instruções não entrem em estado inconsistente. Este problema de concorrência será discutido no Capítulo 11.

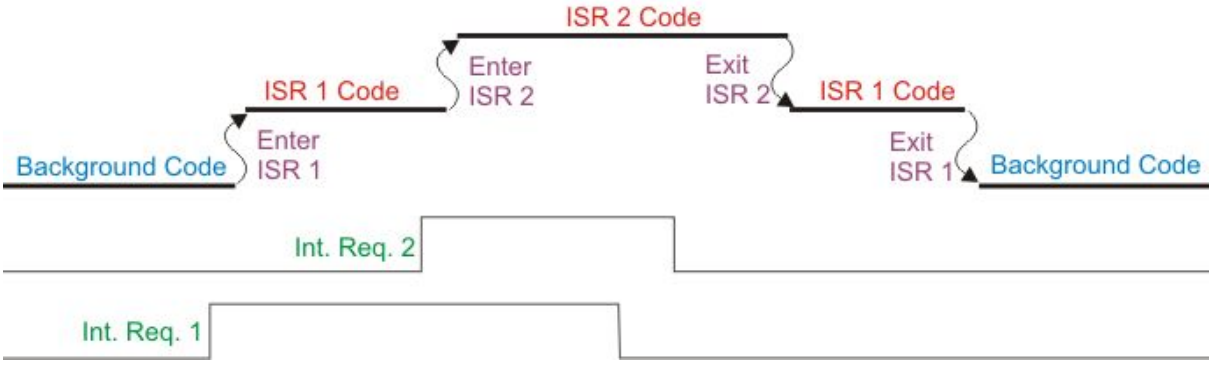


Figura 10.28: Interrupções aninhadas para dar prioridade de atendimento ao evento de maior prioridade (Fonte:[27]).

10.4.3 Um Exemplo

Na arquitetura ARM as interrupções são consideradas como casos especiais de exceções. Sob o ponto de vista de processamento, é utilizado o mesmo modelo de tratamento exceção para ambos os tipos de eventos [28]. Os endereços de rotinas de serviço correspondentes às exceções e às interrupções estão armazenados na mesma Tabela de Vetores de Interrupção (*Interrupt Vector Table*) que pode se alocar na inicialização de um aplicativo. Figura 10.29 mostra uma inicialização em linguagem C. Observe que esta tabela contém as entradas não só de interrupções como as de exceções e os nomes das rotinas ISR já são pré-definidos, indicando que o programador precisa usar estes nomes para programar as suas próprias rotinas ISR. O ligador[5] dos códigos é o responsável pela conexão destes símbolos na geração do código executável.

[5] Montador produz código-objeto a partir da linguagem de montagem. Compilador produz o código na linguagem de montagem a partir do código-fonte. Ligador produz o código executável a partir da ligação dos códigos-objeto.

```
/* The Interrupt Vector Table */
void (* const InterruptVector[])() __attribute__ ((section(".vectortable"))) = {
    /* Processor exceptions */
    (void(*)(void)) &_estack,
    __thumb_startup,
    NMI_Handler,
    HardFault_Handler,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    SVC_Handler,
    0,
    0,
    PendSV_Handler,
    SysTick_Handler,

    /* Interrupts */
    DMA0_IRQHandler, /* DMA Channel 0 Transfer Complete and Error */
    DMA1_IRQHandler, /* DMA Channel 1 Transfer Complete and Error */
    DMA2_IRQHandler, /* DMA Channel 2 Transfer Complete and Error */
    DMA3_IRQHandler, /* DMA Channel 3 Transfer Complete and Error */
    MCM_IRQHandler, /* Normal Interrupt */
    FTFL_IRQHandler, /* FTFL Interrupt */
    PMC_IRQHandler, /* PMC Interrupt */
    LLW_IRQHandler, /* Low Leakage Wake-up */
    I2C0_IRQHandler, /* I2C0 interrupt */
    I2C1_IRQHandler, /* I2C1 interrupt */
    SPI0_IRQHandler, /* SPI0 Interrupt */
    SPI1_IRQHandler, /* SPI1 Interrupt */
    UART0_IRQHandler, /* UART0 Status and Error interrupt */
```

Figura 10.29 Inicialização de uma Tabela de Vetores de Interrupção declarada como um vetor InterruptVector em C.

As interrupções são gerenciadas por um circuito extra, *Nested Vector Interrupt Controller* (NVIC), como mostra a Figura 10.30. Este circuito, em conjunto com o processador, consegue reduzir o tempo de latência no atendimento de duas ou mais interrupções consecutivas (*tail-chaining)* e no atendimento de uma interrupção prioritária com retardo (*late-arrival*). No núcleo do microcontrolador está ainda integrado um sistema de temporização (*timer*) de 24-*bits*, denominado *SysTick*, que pode ser utilizado como um gerador de interrupções periódicas. Estas interrupções são eventos assíncronos gerenciados pelo NVIC.

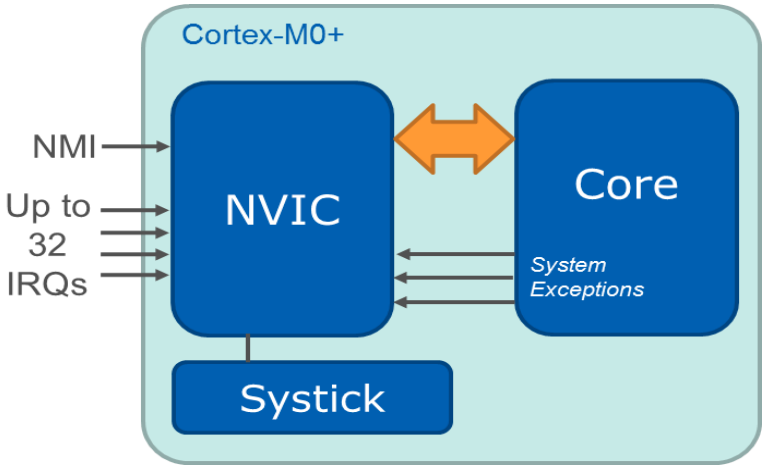


Figura 10.30: Módulos do núcleo dos microcontroladores da família Kinetis L: processador ARM Cortex-M0+, NVIC e SysTick (opcional) (Fonte: [28])

10.5 Famílias de Microcontroladores

Há uma grande variedade de microcontroladores disponíveis comercialmente, como mostra o levantamento apresentado em Wikipedia [29]. Podemos classificar esses microcontroladores de acordo com a largura do seu barramento, repertório de instruções, arquitetura de memória, e localização da memória. Para uma mesma família de microcontroladores, podemos ter variações nestes itens [30].

Quanto à quantidade de *bits* disponíveis nos registradores de operação lógico-aritméticas, temos tipicamente microcontroladores de 4, 8, 16 e 32 bits (Seção 4.1.3). Quanto maior a quantidade de *bits*, maior é a precisão dos cálculos numéricos. Microcontroladores de 32 *bits* são tipicamente usados nos dispositivos médicos, de controle, e de automação de escritório e secretariado. **Quanto ao repertório de instruções**, temos microcontroladores de arquitetura CISC (*Complex Instruction Set Computer*) e de arquitetura RISC (*Reduced Instruction Set Computer*) (Seção 4.1.1). Na arquitetura CISC o uso do espaço de memória é otimizado, por ter menos instruções por programa. Na arquitetura RISC o tempo de execução é usualmente menor por padronizar as micro-instruções (propicia *pipeline*) e reduzir a quantidade de ciclos de relógio por instrução. **Quanto à arquitetura de memória**, distinguem-se a arquitetura von Neumann (Princeton) e a arquitetura *Harvard* (Seção 4.2). A arquitetura é a clássica e a de Princeton evita *hazards* estruturais (Seção 4.2.2) em *pipeline*. E, finalmente **quanto à localização da memória**, há microcontroladores com as unidades de memória principal embutidas num mesmo *chip* e existem microcontroladores, como 8031, com memória principal externa.

Nesta seção vamos apresentar alguns microcontroladores para ilustrar a sua evolução e a sua diversidade. O *link* [39] permite acessar o banco de dados Keil de um grande volume de dispositivos e permitir buscas pelos microcontroladores com características definidas. Antes de iniciarmos, vale chamar atenção a um equívoco frequente: *Raspberry* Pi é um sistema computacional on *chip* (SoC) embutível/embarcável, provido de um microprocessador ARM e uma série de periféricos como UART, temporizadores, controlador de vídeo (GPU), controlador de Ethernet, controlador de HDMI e controladores de interface das mídias de armazenamento (USB, SSD). Ele é muito mais complexo do que um microcontrolador e o seu primeiro estágio do processo de *boot* é controlado pela GPU[6] [54,55]. Por isso, ele é mais reconhecido como um *single board computer* (SBC) do que um microcontrolador [44].

10.5.1 Intel 8051 - 1981

Microcontroladores 8051 é um microcontrolador de 8 *bits* baseado na arquitetura CISC. Ele tem 40 pinos com 4 portas de entrada e saída, sendo as saídas dos pinos da porta 0 de dreno aberto e os pinos da porta 3 multiplexados com outras funções como comunicação serial UART, sinais de relógio do seu sistema de temporização, e gatilhos para controladores de interrupção. Há ainda dois pinos para interfacear com um cristal externo necessário à geração dos sinais de relógio do sistema. Figura 10.31 apresenta uma versão simplificada do diagrama de blocos do Intel 8051. Observe que ele tem dois sistemas de temporização cujos sinais de relógio são fornecidos externamente.

[6] Graphics Processing Unit.

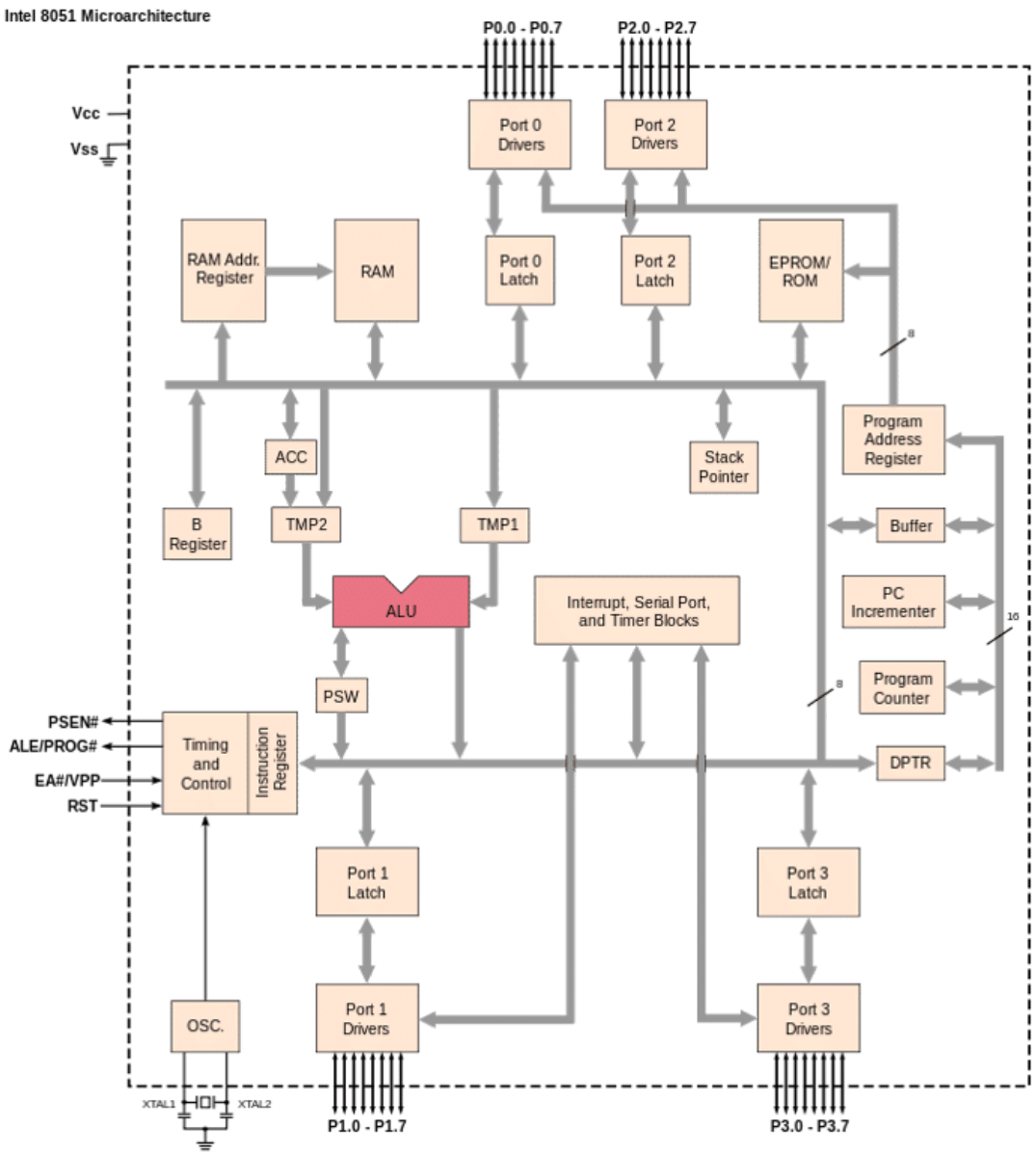
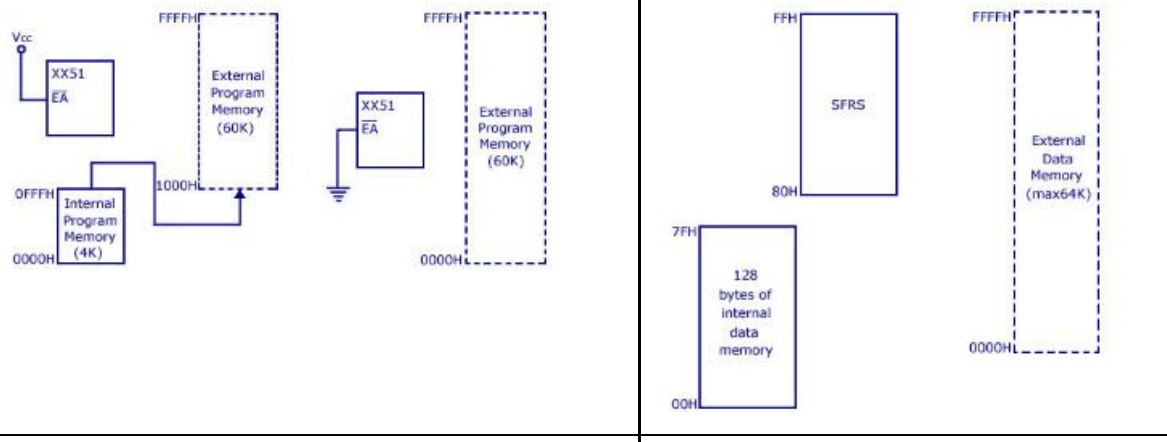


Fig 10.31: Intel 8051 (Fonte: [32]).

A memória principal do Intel 8051 é dividida em duas partes: memória de instruções (ROM) e memória de dados (RAM). Os espaços das duas memórias são extensíveis através de memórias externas, como mostra a Figura 10.32. Nas versões mais modernas, com memórias FLASH, pode-se programar através da interface JTAG [40].



(a) Memória de instruções | (b) Memória de dados

Figura 10.32: Memórias externas no Intel 8051 (Fonte: [30]).

Pela facilidade de integração num outro dispositivo, Intel 8051 é um microcontrolador encontrado em diversos dispositivos, como painel de toque, sistema de gerenciamento de energia, automóveis e aparelhos médicos.

10.5.2 Motorola HC11 - 1984

HC11 é um microcontrolador de 8 *bits*, de arquitetura CISC e von Neumann. As instruções e os dados são armazenados num mesmo espaço de memória. Foi introduzido pela Motorola e hoje produzido pela empresa NXP Semiconductors. É descendente do microprocessador 6800 da Motorola. É o primeiro microcontrolador que inclui a memória CMOS EEPROM de 2048 *bytes*. A versão mais popular de HC11 tem 5 portas A, B, C. D e E, como mostra a Figura 10.33. Todas as portas tem 8 pinos, exceto a porta D com 6 pinos. Pode operar com as instruções armazenadas na memória interna RAM (1 até 768 *bytes*) ou com as instruções armazenadas na memória externa extensível até 64 *kilobytes*. Neste último caso, as portas B e C são usados como barramento de endereços e de dados e a porta C é multiplexada entre os dados e os *bytes* menos significativos dos endereços.

Está integrado nele um conversor AD por aproximações sucessivas (SAR) de 8 *bits* e 8 canais. Há um sistema de temporização de 16 *bits*, com 3 canais de *Input Capture*[7] (IC), 4 canais de *Output Compare*[8] e um canal configurável. Este sistema inclui o temporizador de *watchdog*, interrupções periódicas internas e lógica de controle de interrupções externas IRQ. As comunicações seriais assíncronas

[7] Ao capturar um evento, o valor do contador no temporizador é automaticamente armazenado.
[8] Quando o contador do temporizador atingir o valor pré-definido, gera-se no pino de saída o sinal pré-configurado.

adotam a codificação *non-return-to-zero* (NRZ)[9] e o controlador de comunicação serial SPI é embutido no *chip*. Além dos pinos de alimentação digital, é necessário prover as tensões de referência para o conversor AD e sinais do cristal para geração de sinais de relógio do sistema. Não suporta diretamente interface JTAG, mas há dispositivos como M88x3Fxx [38] que possam ser conectados caso queira agregar esta funcionalidade.

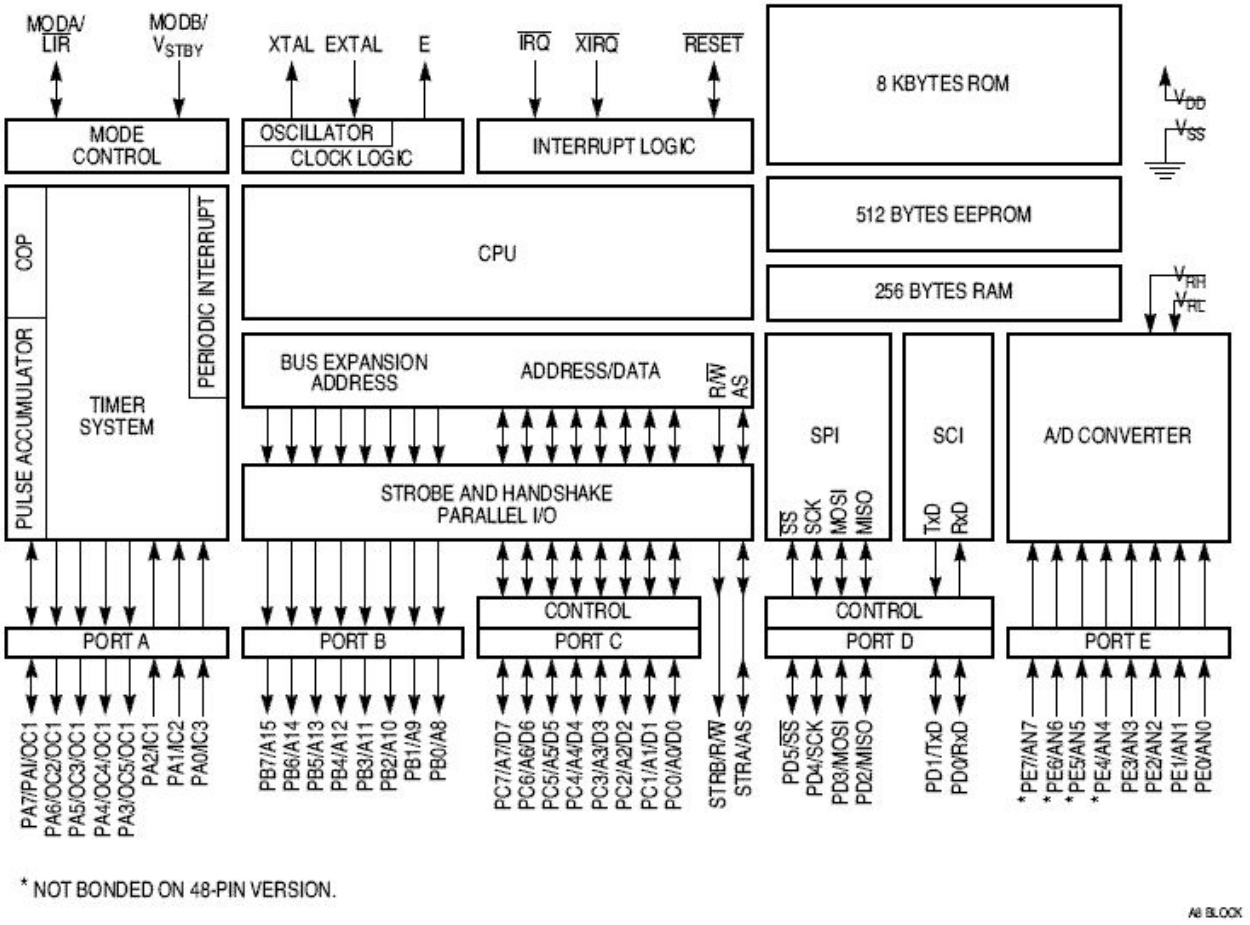


Fig 10.33: Diagrama de blocos da Motorola MC68H11 (Fonte: [33]).

10.5.3 PIC - 1993

PIC é acrônimo de *Peripheral Interface Controller*, que evoluiu para *Programmable Intelligent Computer*. É designado por PICnnCxxx (CMOS) ou PICnnFxxx (FLASH). nn indica a quantidade de bits usados para representar uma instrução (12 *bits*, nn=10 ou 12; 14 *bits*, nn=16; 24 *bits*, nn=24). PIC16x84 foi o primeiro microcontrolador de 8 *bits* da Microchip provido de EEPROM embutido no *chip*. Hoje em dia todos os modelos usam a memória FLASH para armazenar as instruções. A arquitetura deste microcontrolador é *Harvard* modificada. O seu repertório de

[9] É uma codificação em que 1s são representados por uma condição significativa, usualmente um nível de tensão positivo, e os 0s por uma outra condição significativa, usualmente um nível de tensão negativo.

instruções é do tipo RISC, contendo apenas 35 instruções. O tamanho das instruções pode ser 12, 14, 16 ou 24 *bits*. Exceto as instruções de desvio que demandam 2 ciclos de instrução, todas requerem apenas um ciclo de instrução. O tamanho dos dados pode ser de 8, 16 e 32 *bits*. A família de PIC mostrada na Figura 10.34 suporta instruções de 14 *bits* e dados de 8 *bits*.

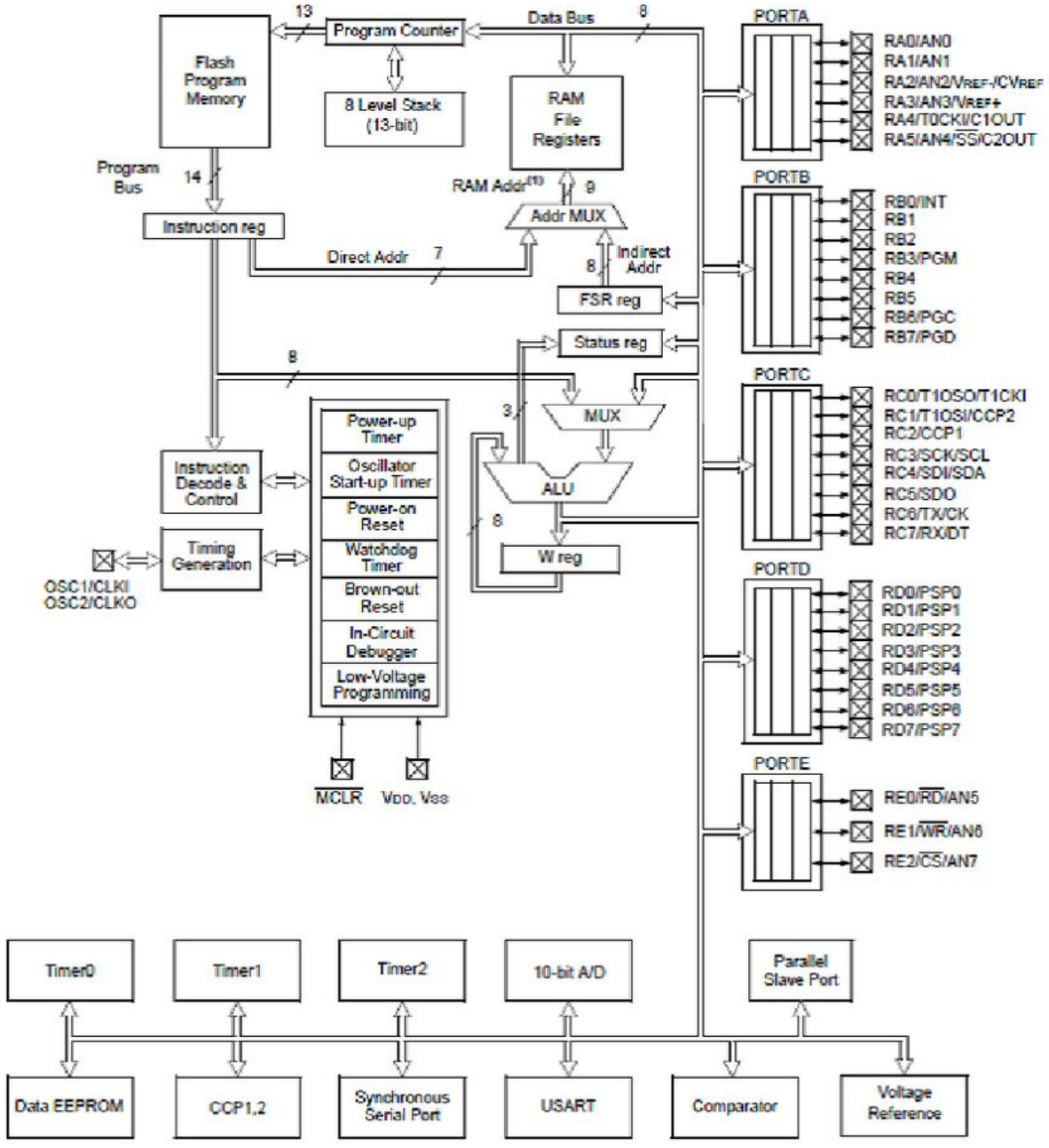


Figura 10.34: Diagrama de blocos de MIcrochip PIC16F87XA (Fonte: [35]).

Distinguem-se três tipos de memória nos PICs: memória de instruções, memória volátil (RAM) de dados e memória não-volátil de dados EEPROM. As duas primeiras memórias são usadas na execução de um programa e a última memória para armazenar dados. Para PIC16F87Xa, o espaço de memória de instruções contém 2^{13}-1 endereços, de 0x0000 a 0x1FFF, o espaço de memória de dados contém 4

bancos de 128 *bytes* (2^7-1 endereços, de 0x00 a 0x7F) - é neste espaço que estão mapeados todos os registradores do sistema, e um espaço de EEPROM de até 256 bytes (2^8-1 endereços).

São ainda embutidos no *chip*: 3 temporizadores, 1 sistema de temporização com Input Capture, Output Compare e PWM[10] , comunicações seriais assíncronas (UART) e síncronas I2C e SPI e 1 conversor AD por aproximações sucessivas de 10 *bits*. Um circuito de interrupção gerencia as interrupções vetorizadas e autovetorizadas, internas (sinais de interrupção dos circuitos periféricos para indicar diferentes eventos) e externas (através da porta B na Figura 10.34). Pode-se também gerar interrupções por *software* via a instrução SWI [41]. Para permitir que todos esses módulos embutidos se comuniquem com o mundo físico, os pinos do PIC são multiplexados. Vale observar que os PICs modernos são providos de uma interface JTAG embutida, mas, por padrão, não é habilitada. A Microchip desenvolveu seus depuradores e programadores proprietários [37].

Os PICs são bastante populares, tanto para desenvolvedores industriais quanto para hobbistas, em decorrência d o seu baixo custo, ampla disponibilidade, grande base de usuários, extensa coleção de notas de aplicação, disponibilidade de ferramentas de desenvolvimento de baixo custo ou grátis, e capacidade de programação serial e reprogramação com memória flash [34].

10.5.4 AVR - 1996

É uma família de microcontroladores desenvolvidos pela Atmel, que foi vendida para *Microchip Technologies* em 2016. A arquitetura AVR foi concebida por dois estudantes da Instituto de Tecnologia da Noruega (NTH), Alf-Egil Bogen e Vegard Wollan, nos meados de 1990. Os microcontroladores são de 8 *bits*, com um repertório de instruções RISC e uma arquitetura de memória *Harvard* modificada. Por ser organizado em *pipeline* de 2 estágios (o estágio de busca em paralelo com o estágio de execução), AVR tem um desempenho superior a muitos microcontroladores de 8 *bits*. Por ser projetado visando à execução de códigos de máquina gerados pelo compilador C, a melhor linguagem de programação para AVR é C [43].

É um dos primeiros microcontroladores que usou a memória FLASH embutida no *chip* para armazenar as instruções. O tamanho da memória de instruções é usualmente indicado no nome do microcontrolador, como ATmega16x tem 16 k*bytes*. Além da memória FLASH, são embutidos no seu *chip* uma memória de dados SRAM para execução de programas e uma memória EEPROM para

[10] *Pulse Width Modulation*

armazenamento de dados. Os registradores e os periféricos são mapeados no mesmo espaço de endereços da memória. Em muitos variantes de AVR, a memória EEPROM é considerado como um periférico e o seu conteúdo é acessado indiretamente via registradores mapeados no espaço de memória.

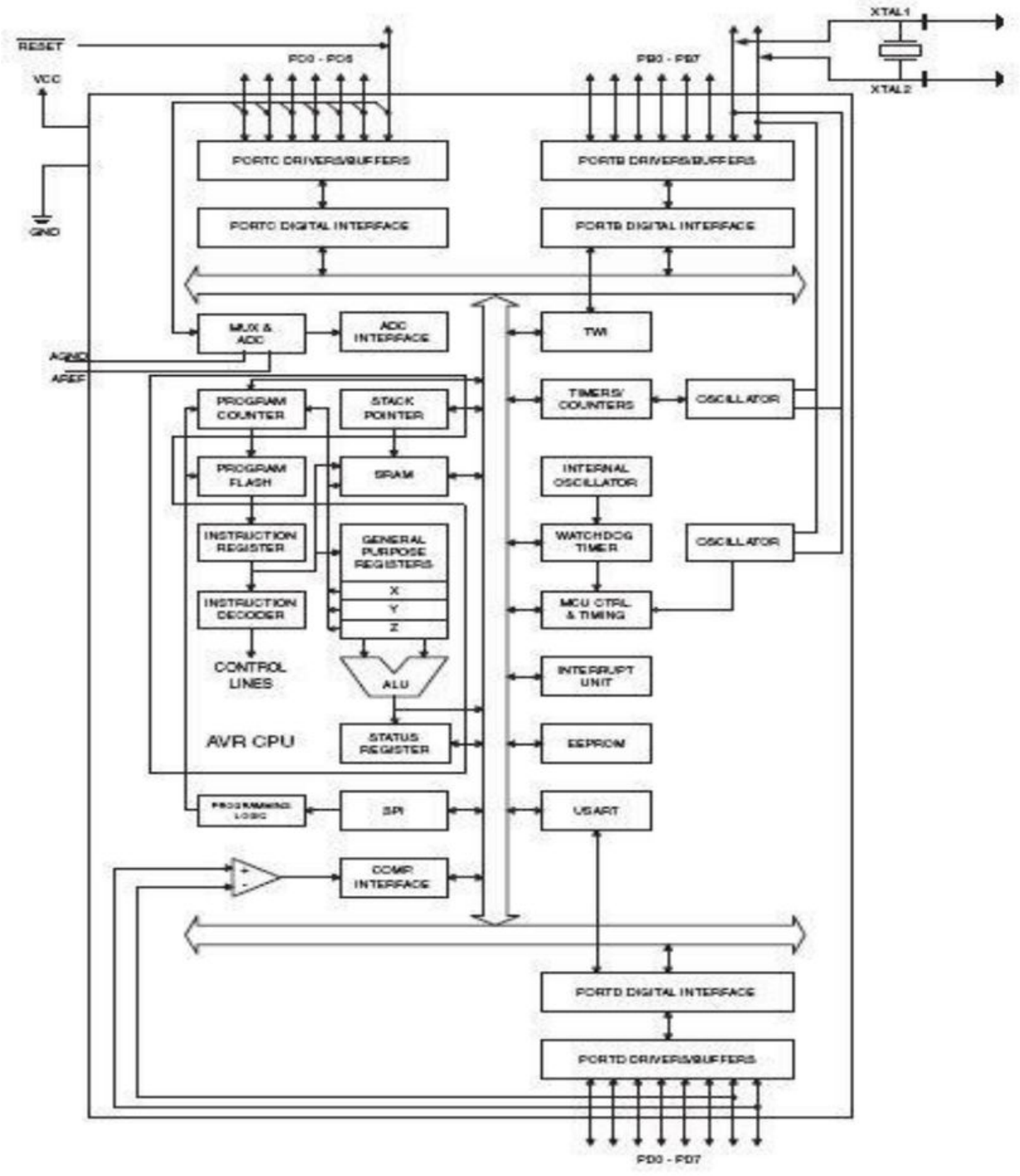


Figura 10.35: Diagrama de bloco de Atmega8 (Fonte: [42]).

AVR oferece uma grande gama de funcionalidades (Figura 10.35): 4 portas de entrada e saída de 8 *bits* configurável, incluindo dreno aberto com resistor *pull-up*, sistemas de temporização de 8 e 16 *bits*, 1 conversor AD de 10 ou 12

bits, 1 conversor DA de 12 *bits*, 1 comparador analógico, uma grande variedade de circuitos de interface de comunicação serial, 1 controlador de interrupções internas (dos temporizadores e dos periféricos) e externas (3 pinos INT0, INT1 na porta D e INT2 na porta B), e uma interface JTAG (Seção 10.2.6). A frequência do sistema é estabelecida pela frequência de um cristal externo.

O módulo de sistema de sinais de relógio e sua distribuição é complexo, como mostra a Figura 10.36. O módulo gera, a partir da fonte do sinal de relógio, sinais de relógio para o microprocessador (CPU), memória FLASH, conversor ADC e módulos de Entrada/Saída.

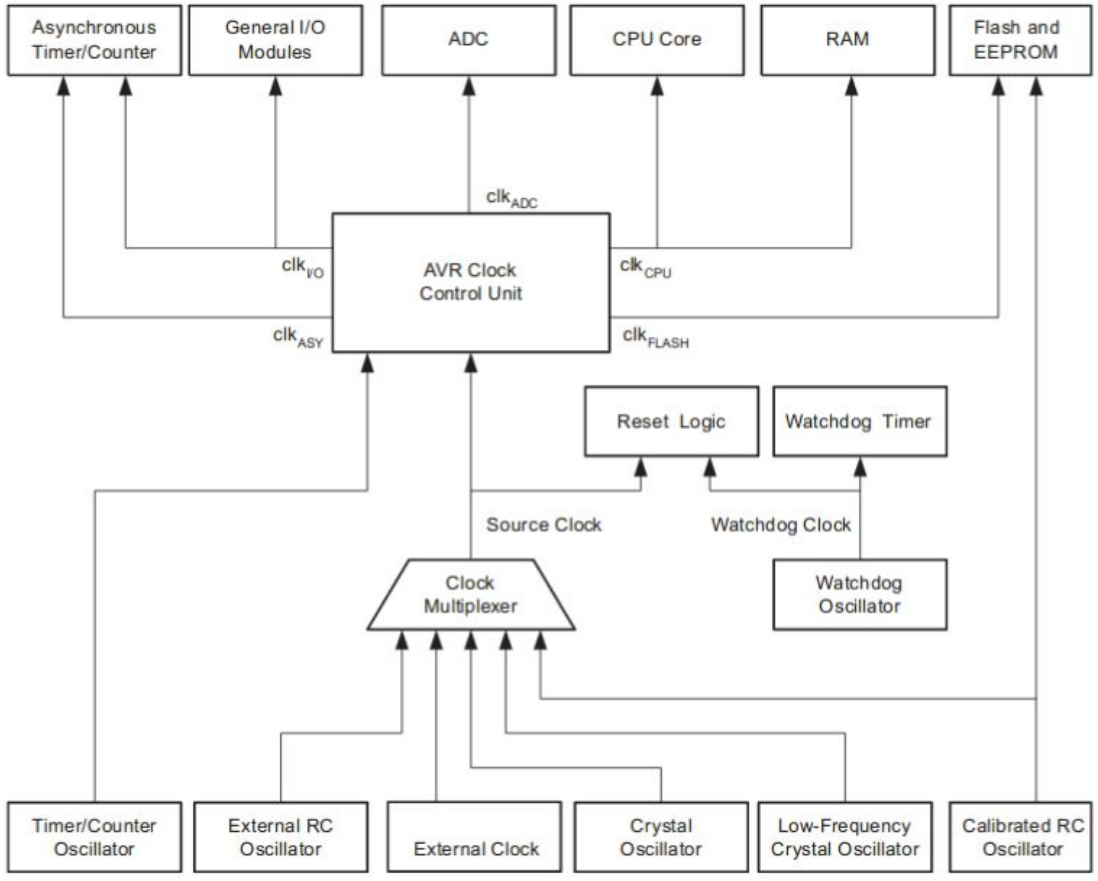


Figura 10.36: Diagrama de bloco da unidade de relógio do AVR (Fonte: [42]).

Microcontroladores AVR tornaram bastante populares pela sua integração na grande maioria dos *kits* de desenvolvimento de *hardware* aberto Arduino. Eles são muito apreciados pelos hobbistas e usados no ensino, principalmente pelo volume de material compartilhado na internet.

10.5.5 Kinetis KL25Z128 - 2012

Microcontroladores da sub-família Kinetis KL25 são microcontroladores de 32 *bits*

cujo núcleo é um microprocessador ARM (*Advanced RISC Machine*) da arquitetura *Harvard*. As instruções são armazenadas na memória FLASH (até 128KB) e os dados na memória SRAM (até 16KB). Todos os registradores do núcleo, dos periféricos e dos circuitos de interface de comunicação são mapeados no mesmo espaço de endereços da memória. Em relação aos microcontroladores apresentados nas seções anteriores, esses microcontroladores têm integrados no seu *chip* muito mais funcionalidades. Destacam-se o controlador de acesso direto à memória (DMA), os circuitos de interface com sensor de toque, em inglês *Touch Sensing Interface* (*TSI*), e as 2 portas (A e D) com 32 pinos capazes de gerarem interrupções externas, além das interrupções internas. Figura 10.37 ilustra os principais componentes do microcontrolador Kinetis KL25Z128.

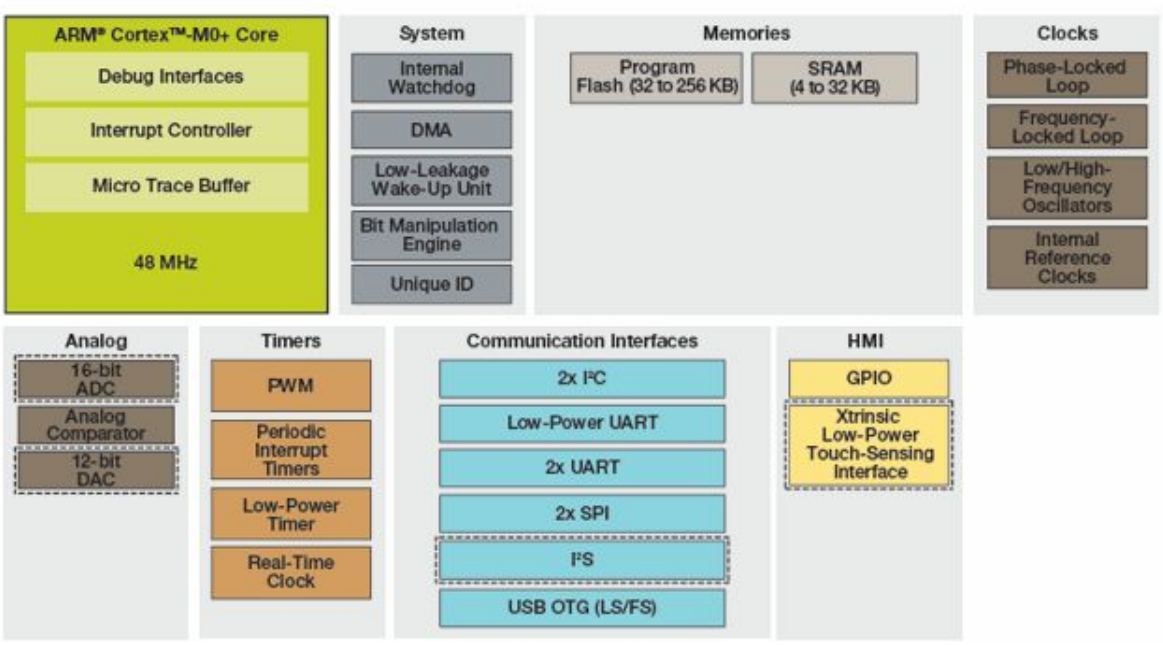


Figura 10.37: Principais componentes do Kinetis KL25Z128 (Fonte: [47]).

Em relação aos circuitos de interface de comunicação serial, eles contém não só as interfaces SPI, I2C e UART, como também um circuito de interface USB. É incluído no *chip* um relógio de tempo real, em inglês *real time clock* (RTC). E cada *chip* de um microcontrolador tem um número de identificação única, em inglês *Unique Identification Register*, de 80 *bits*. Essa UID é atribuída durante o processo de fabricação. Ela facilita a identificação de um *chip* numa rede, rastreamento de um *chip*, segurança no pareamento com o dispositivo em que o *chip* é embutido e a criptografia das entradas.

Vale também destacar o módulo de geração de sinais de relógio para multi-propósitos embutido no *chip* Kinetis KL25 [46]. Está integrado neste módulo circuitos FLL (*Frequency Lock Loop*) e PLL (*Phase Lock Loop*) para gerar simultaneamente diferentes frequências de sinais de relógio de sinais de relógio apropriados para os diversos módulos embutidos no *chip*. Veja na Figura 10.38 que

podemos o módulo MCG consegue gerar, a partir de um cristal externo, sinais de relógio para o núcleo/microprocessador, barramento, memória FLASH e periféricos.

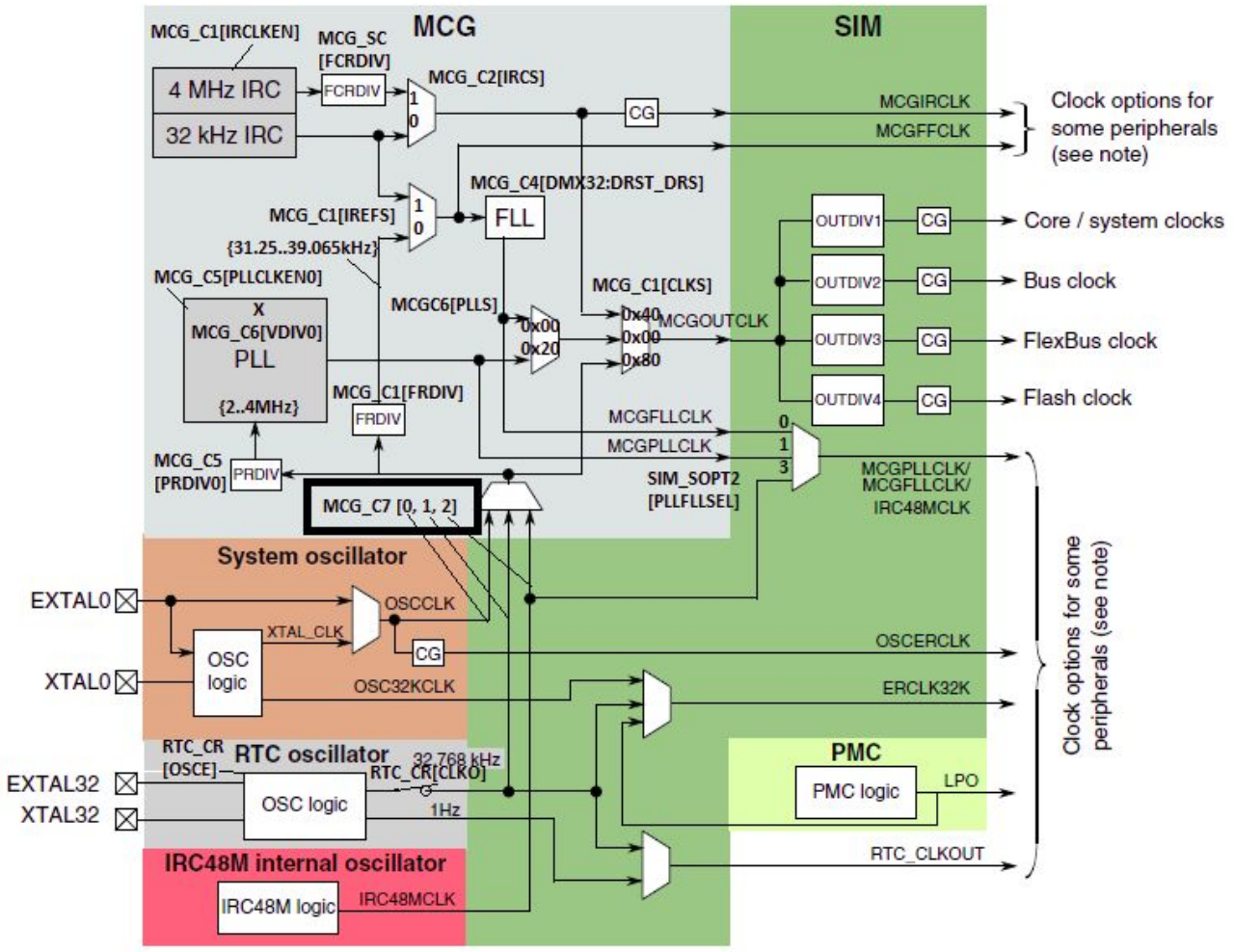


Figura 10.38: Diagrama de blocos do módulo MCG embutido no Kinetis KL25Z128 (Fonte: [46]).

10.6 Exercícios

1. Qual é a diferença entre um microprocessador e um microcontrolador?
2. Como pode ser classificado o Raspberry Pi? É um microprocessador? Um microcontrolador? Ou um microcomputador num *chip*?
3. É similar o projeto de memórias convencionais e o das memórias embarcadas? Justifique.
4. O que você entende por um pino multiplexado? Por que esta técnica é vastamente aplicada nos projetos de microcontroladores?
5. Sejam os seguintes registradores do microcontrolador ESP8266: GPIO_OUT (registrador de saída), GPIO_ENABLE (registrador de estado de habilitação)

e GPIO_IN (registrador de entrada). Num programa em C, qual(is) destes registradores devem ser declarados com o qualificador volatile? Justifique.

6. Quais são as características almejadas para um barramento embarcado? Há alguma diferença em relação aos barramentos externos de um sistema computacional?
7. O que você entende por uma interface JTAG embutida num microcontrolador? Para que serve esta interface?
8. Os microcontroladores AVR de 8 *bits* tem registradores de função especial de 16 *bits* no sistemas de temporização de 16 *bits* embutidos. Em qual memória são mapeados os registradores de função especial de 16 *bits*? Justifique.
9. O que é um sistema de temporização num microcontrolador? Qual é o componente principal deste sistema?
10. Dado um sistema de temporização de 8 *bits* com um relógio de frequência 16MHz. Qual é o intervalo de tempo máximo que este sistema consegue contar sem *overflow*? Para monitorar um intervalo de tempo de 0.5s quantos *overflows* podem ocorrer?
11. Modulação por largura de pulso, em inglês *pulse width modulation* (PWM), consiste na modulação da razão cíclica, em inglês *duty cycle*, de um sinal. Uma das aplicações é o controle de potência média entregue a uma carga. Os sistemas de temporização que suportam a geração de sinais PWM (1) têm um registrador onde se configura a quantidade de *ticks* em que o seu contador deve alternar o estado do sinal de saída, e (2) permitem a configuração de setar (1)/resetar (0) o sinal de saída quando ocorre um *overflow*. Para sistemas de temporização de 8 *bits*,
    a. qual é a quantidade de larguras de pulsos que se pode configurar?
    b. para um *duty cycle* de 25%, qual deve ser o valor setado no registrador de *ticks* que controla a alternância do estado do sinal de saída?
12. Saída baseada na comparação é um modo de operação de um sistema de temporização. Ela consiste na atribuição de um estado de saída pré-configurado quando o valor do contador iguale o valor de *ticks* pré-fixado. Para um sistema de temporização de 315MHz, qual deve ser o valor do divisor de frequência que devemos inserir para que o sinal de saída tenha a frequência próxima da nota musical Dó (253Hz).
13. O que vocẽ entende por uma exceção e por uma interrupção?
14. Qual é a diferença entre um a interrupção vetorizada e auto-vetorizada? Qual delas é da geração antiga?
15. Quando ocorre uma interrupção, por quê o conteúdo de contador de programa (PC) é salvo na pilha do sistema? O que aconteceria se não for salvo?

16. Por quê a introdução de *linking register* melhorou o tempo de processamento de interrupções?
17. É preciso inserir nos códigos as chamadas das rotinas de serviço ISR nos pontos em que a sua execução é esperada? Justifique.
18. É possível armazenar a tabela de vetores de interrupção numa memória não volátil já que as interrupções já são decididas na fase do projeto? Justifique. Caso seja possível, qual(is) seria(m) a(s) restrição(ões)?
19. Explique o que poderia acontecer se as rotinas de serviços de interrupções de maior prioridade sejam muito longas.
20. O desenvolvimento do projeto do curso consiste um conjunto de exercícios relacionados com os microcontroladores. Procurem associar as características dos microcontroladores que vocês estão trabalhando com os apresentados neste capítulo.

10.7 Referências

[1] Electronics Hub. Basics of Microcontrollers - History, Structure and Applications. https://www.electronicshub.org/microcontrollers-basics-structure-applications/

[2] Embedded memory. http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC11.PDF

[3] StackExchange. Pin and port in microcontroller. https://electronics.stackexchange.com/questions/134605/pin-and-port-in-microcontroller

[4] Wikipedia. Advanced Microcontroller Bus Architecture. https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture

[5] Wikipedia. Volatile (Computer Programming). https://en.wikipedia.org/wiki/Volatile_(computer_programming)

[6] Wikipedia. CoreConnect. https://en.wikipedia.org/wiki/CoreConnect

[7] Intel. Avalon Interface Specifications. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

[8] Silicore. Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. https://cdn.opencores.org/downloads/wbspec_b3.pdf

[9] Wikipedia. CAN bus. https://en.wikipedia.org/wiki/CAN_bus

[10] Wikipedia. FlexRay. https://en.wikipedia.org/wiki/FlexRay

[11] Pillai, V.P. JTAG basics and usage in microcontroller debugging. https://www.embeddedinn.xyz/articles/tutorial/JTAG-basics-and-usage-in-microcontroller-debugging/

[12] Embedded C. I2C Bus Communication Protocol Tutorial with Example. http://www.mbeddedc.com/2017/05/i2c-bus-communication-protocol-tutorial.html

[13] Random Nerd Tutorials. ESP8266 Pinout Reference: Which GPIO pins should you use? https://randomnerdtutorials.com/esp8266-pinout-reference-gpios/

[14] SlideShare. I2C Programming with C and Arduino.
https://www.slideshare.net/sato262/i2c-programming-with-c-and-arduino
[15] R-B. Timers and Counters. http://embedded-lab.com/blog/timers-and-counters/
[16] Mike Silva. Introduction to Microcontrollers - Timers.
https://www.embeddedrelated.com/showarticle/478.php
[17] Embedded Xinu. BCM2835 System Timer.
https://embedded-xinu.readthedocs.io/en/latest/arm/rpi/BCM2835-System-Timer.html
[18] GranaSAT. Debugging Arduino Mega 2560 with JTAGICE3.
https://granasat.ugr.es/2017/10/debuggin-arduino-mega-2560-with-jtagice3/
[19] Texas Instruments. MSP-EXP430G2 LaunchPad Development Kit.
http://www.ti.com/lit/ug/slau318g/slau318g.pdf
[20] Freescale. FRDM-KL25Z User's Manual
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/FRDMKL25Z.pdf
[21] SysProgs. Debugging NodeMCU Firmware over JTAG.
https://visualgdb.com/tutorials/esp8266/nodemcu/jtag/
[22] SysProgs. Preparing Raspberry PI for JTAG Debugging.
https://sysprogs.com/VisualKernel/tutorials/raspberry/jtagsetup/
[23] ARM. Interrupt Behavior of Cortex-M1.
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0211a/index.html
[24] Silicon Labs Community. EFM32 and EFR32 Interrupt Handling.
http://community.silabs.com/t5/tkb/articleprintpage/tkb-id/2@tkb/article-id/1193
[25] National Instruments. Timing and Transition.
http://zone.ni.com/reference/en-XX/help/371268P-01/expresswb/timing_and_transition/
[26] Mike Silva. Introduction to Microcontrollers - Interrupts.
https://www.embeddedrelated.com/showarticle/469.php
[27] Mike Silva. Introduction to Microcontrollers - More On Interrupts.
https://www.embeddedrelated.com/showarticle/472.php
[28] ARM. Cortex-M0+ Nested Vector Interrupt Controller.
http://microchip.wikidot.com/32arm:m0-nvic
[29] Wikipedia. List of common microcontrollers.
https://en.wikipedia.org/wiki/List_of_common_microcontrollers
[30] EL-PRO-CUS. Microcontrollers - Types and Applications.
https://www.elprocus.com/microcontrollers-types-and-applications/
[31] Intel 8051 Datasheet. http://www.keil.com/dd/docs/datashts/intel/80xxah_ds.pdf
[32] Fernando Deluno Garcia. Arquitetura Intel 8051.
https://www.embarcados.com.br/arquitetura-intel-8051/
[33] NXP. M68HC11E Family. https://www.nxp.com/docs/en/data-sheet/M68HC11E.pdf
[34] Wikipedia. PIC microcontrollers. https://en.wikipedia.org/wiki/PIC_microcontrollers
[35] Microchip. PIC16F87XA Datasheet.
https://ww1.microchip.com/downloads/en/devicedoc/39582b.pdf
[36] Wikipedia. AVR Microcontrollers. https://en.wikipedia.org/wiki/AVR_microcontrollers
[37] Microchip. Introduction to JTAG. https://microchipdeveloper.com/jlink:jtag
[38] ST. AN1176 Application Note - 68HC11/M88 Flash+PSD Design Guide.
http://search.alkon.net/cgi-bin/pdf.pl?pdfname=anglia/st/6945.pdf

[39] ArmKeil. Parametric Search.
http://www.keil.com/dd/search_parm.asp?_ga=2.81314327.1203347296.1570267867-235674196.1564315605
[40] Silicon Labs. AN105 Programming FLASH Through the JTAG Interface.
https://www.silabs.com/documents/public/application-notes/an105.pdf
[41] Khaled Magdy. Interrupts in PIC Microcontroller.
https://deepbluembedded.com/interrupts-in-pic-microcontrollers/
[42] Atmel. 8-bit Microcontroller with 8K Bytes In-System Programmable Flash.
https://www.gme.cz/data/attachments/dsh.958-112.1.pdf
[43] Wikipedia. AVR microcontrollers.
https://en.wikipedia.org/wiki/AVR_microcontrollers
[44] Raspberry. https://www.raspberrypi.org/documentation/faqs/#introWhatIs
[45] NXP. Kinetis KL25 Sub-Family 48 MHz Cortex-M0+ Based Microcontroller with USB. ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/KL25P80M48SF0.pdf
[46] uTasker. MCG (Multipurpose Clock Generator) Support - How to use the Kinetis MCG with ease? http://www.utasker.com/kinetis/MCG.html
[47] http://www.ganssle.com/rants/frdm-kl25z.html
[48] Texas Instruments. MSP430x2xx Family User's Guide.
http://www.ti.com/lit/ug/slau144j/slau144j.pdf
[49] ESP8266 Technical Reference.
https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf
[50]
https://github.com/scottjgibson/esp8266/blob/master/esp_iot_sdk_v0.6/include/eagle_soc.h
[51]
https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/iom328p.h
[52] https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/sfr_defs.h
[53] http://eleceng.dit.ie/frank/msp430/msp430ports.pdf
[54] Sérgio Prado. Raspberry Pi e o processo de boot.
https://sergioprado.org/raspberry-pi-e-o-processo-de-boot/
[55] Raspberry Pi. The boot folder.
https://www.raspberrypi.org/documentation/configuration/boot_folder.md
[56] Lammert Bies. Serial UART information.
https://www.lammertbies.nl/comm/info/serial-uart.html
[57] Saleae Articles. SPI vs I2C Protocol Difference and Things to Consider.
https://articles.saleae.com/logic-analyzers/spi-vs-i2c-protocol-differences-and-things-to-consider
[58] Wikipedia. Serial Peripheral Interface.
https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
[59] Jacob Beningo. USART vs UART: Know the difference.
https://www.edn.com/electronics-blogs/embedded-basics/4440395/USART-vs-UART--Know-the-difference
[60] RF Wireless World. UART vs USART: Difference between UART and USART.
https://www.rfwireless-world.com/Terminology/UART-vs-USART.html

[61] SysProgs. Debugging ESP8266 firmware with Olimex ARM-USB-OCD-H. https://visualgdb.com/tutorials/esp8266/olimex/
[62] XJTAG. What is JTAG and how can I make use of it? https://www.xjtag.com/about-jtag/what-is-jtag/
[63] https://rmd.ac.in/dept/it/notes/3/DPSD/unit3.pdf
[64] Microchip Technology. ATmega328P Datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf