ARM Cortex-M:

Hardware e Software Embarcado

João Ranhel

1ª. Edição

São Paulo, 2025.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Ranhel, João

ARM Cortex-M: Hardware e Software Embarcado / João Ranhel. -- 1. ed. -- São Paulo : Ed. Do Autor, 2025.

ISBN 978-65-266-3491-2

1. Engenharia 2. Computação 3. Microprocessadores 4. Sistemas embarcados 5. Software embarcado. 6. Linguagem C I. Título.

CDD-621.399

Índice para catálogo sistemático:

1. ARM Cortex-M: Hardware e Software Embarcado : Engenharia Eletrônica 621.399

Aos meus alunos, monitores dessa disciplina, e colegas professores que compartilharam e compartilham o ministério das disciplinas Sistemas Microprocessados, Programação de Software Embarcado, Aplicações de Microcontroladores e Sistemas Embarcados na Universidade Federal do Pernambuco e na Universidade Federal do ABC (UFABC).

Aviso de Marcas Registradas (Disclaimer)

ARM®, Cortex®-M, STM32®, STM32CubeIDE®, STM32CubeMX® são marcas registradas ou comerciais da Arm Limited e STMicroelectronics.

Arduino® é uma marca registrada da Arduino AG.

Este livro não é afiliado, endossado ou patrocinado pelos detentores dessas marcas. Os nomes são usados exclusivamente para fins descritivos e referenciais, sem intenção de violação de direitos de propriedade intelectual.

Agradeço a meus amigos por valiosas sugestões e revisões no texto, agradecimento especial a Alessandro Alberto dos Santos, grato pela paciência em ler o texto ainda mal formatado. Também sou grato aos que tenham contribuído de alguma forma para a execução desse projeto, aos colegas, alunos e técnicos dos laboratórios da UFABC.

João Ranhel

Sumário

Sumário v
Introdução 1
Breve Histórico 2
De Microprocessadores a Microcontroladores 3
Parte 1 – FUNDAMENTOS 7
I – Microprocessador (MCU) 9
1.1 – Revisão de Eletrônica Digital 9
1.1.1 – Interconexão de saídas digitais 9
1.1.2 – Transceptor de Barramento 10
1.1.3 – Latches e Registradores 10
1.1.4 – Memórias 11
1.1.5 – Unidade Lógica e Aritmética 12
1.1.6 – ULA, Registradores e Datapath 13
1.2 – Processadores 14
1.2.1 – Sequência de Instruções 15
1.2.2 – Decisões e Inteligência 16
1.2.3 – Obter Dados e Salvar Resultados 17
1.3 – Funcionamento da Arquitetura Harvard 18
1.3.1 – Do Programa ‘C’ ao Assembly 18
1.3.2 – Interpretando Assembly 20
1.3.3 – Passo a Passo em Assembly 21
1.4 – Processador von Newman Didático 22
1.5 – Microcontrolador ARM Cortex-M 23
1.5.1 – Registradores do ARM Cortex-M 24
1.5.2 – Núcleo dos Processadores Cortex-M3/M4 25
1.5.3 – Pipeline no Cortex-M 26
1.5.4 – Unidades de Depuração e Rastreamento 28
1.5.5 – Circuitos Auxiliares do Core Cortex-M 29
1.5.5.1 – Circuitos Supervisores de Alimentação 30
1.5.5.2 – Circuitos de Clock e Reset 30
1.5.6 – Barramentos e Interfaces 32
1.5.7 – Mapeamento de Memória 34
1.5.7.1 – Boot 35
1.5.7.2 – Vantagens do Mapeamento e CMSIS 36
1.6 – Resumo e Conclusões do Capítulo 37
II – Ambiente de Desenvolvimento 39
2.1 – Projeto de Sistemas Embarcados 39
2.2 – Kits de Hardware 40
2.2.1 – Diagramas Elétricos 43
2.2 – Ambiente de Desenvolvimento 48

2.2.1 – Básico do STM32CubeIDE / STM32CubeMX....48
2.3 – Programando no STM32CubeIDE....55
2.4 – Compreendendo Assembly no ARM....62
2.4.1 – Diferença Entre Instruções ARM e Thumb....63
2.4.1.1 – Conjuntos ARM Modernos (Thumb e Thumb-2)....63
2.4.2 – Tabela reduzida AIS....64
2.4.3 – Resumo ARM Instruction Set....65
III – Programa em C e ARM Cortex-M....69
3.1 – Programação em Linguagem ‘C’....69
3.1.1 – Linguagem C em Embarcados....71
3.1.2 – Sub-rotina ou Função em C....72
3.1.3 – ARM e Chamada de Funções....73
3.2 – Pilha....73
3.2.1 – Pilha no ARM....74
3.2.2 – Pilha, Funções e Variáveis Locais em C....75
3.3 – Interrupções....76
3.4 – SysTick....78
3.5 – Foreground-Background e Superloop....80
3.5.1 – Software Dirigido por Eventos....81
3.5.2 – Código sem Bloqueios....82
3.5.3 – Condições de Corrida – Race Condition....84
3.6 – Memória RAM e Alocação de Variáveis....85
3.7 – Passagem de Parâmetros Entre Funções....87
3.7.1 – Parâmetros Passados por Valor....87
3.7.1.1 – Cortex-M e Passagem de Argumentos....87
3.7.2 – Parâmetros por Referência....89
3.7.3 – Parâmetros Passados por Arrays....91
3.8 – Resumo do Capítulo....92
IV – Cortex-M: Periféricos....94
4.1 – Pinos do Circuito Integrado....96
4.1.1 – Pino de Reset....96
4.1.2 – Estrutura do Pino I/O Padrão....97
4.2 – Periférico GPIO (General Purpose Input-Output)....98
4.2.1 – Programa Bare-Metal Mínimo Para GPIO....99
4.2.2 – Operação na GPIO Livre da Condição de Corrida.....100
4.2.3 – Conclusões Sobre Base Metal.....102
4.2.4 – Programação dos GPIOs com IDE....102
4.2.4.1 – Entrada e Saída Digital.....104
4.2.5 – GPIO Entrada Analógica....105
4.2.5.1 – GPIO Saída Analógica....106
4.2.6 – GPIO Entrada Digital Alternativa....106
4.2.7 – Interrupção Externa e GPIO....107

4.2.8 – GPIO Saída Digital Alternativa 108
4.3 – Temporizadores 108
4.3.1 – Descrição Geral do Temporizador 109
4.3.2 – Modos Geração de Saídas nos Temporizadores 111
4.3.2.1 – Programação e Configuração PWM 112
4.3.2.2 – Temporizador Gerando Base Temporal 113
4.3.2.3 – Temporizador no Modo de Pulso 115
4.3.2.4 – Saída com Sincronização Externa (Gated Mode) 117
4.3.3 – Modos de Medição/Entradas (Captura) dos Temporizadores 120
4.3.3.1 – Captura Simples em Sinal de Entrada 120
4.3.3.2 – Encoder de Quadratura e Timer 123
4.4 – Controlador de Interrupções - NVIC 126
4.4.1 – Prioridades e Starvation 127
4.5 – Controlador de DMA (Direct Memory Access) 127
4.5.1 – Interrupções Versus DMA 128
4.6 – ADC – Conversor Analógico para Digital 129
4.6.1 – ADCs nos ARM Cortex-M 129
4.6.2 – ADCs em Polling, DMA e Interrupções 130
4.6.3 – Modos de Captura nos ADCs Cortex-M 131
4.6.4 – Organização dos Canais 131
4.6.5 – Configuração e Programação com ADC 133
4.6.5.1 – Exemplo ADC em Polling 135
4.6.5.2 – Exemplo ADC por Interrupção 137
4.6.5.2 – Exemplo ADC com DMA 138
4.7 – Periféricos de Comunicação 139
4.8 – Resumo do Capítulo 140
V – Comunicação e Microcontroladores 142
5.1 – Tipos de Comunicação 143
5.1.1 – Resistores Pull-Up e Pull-Down 143
5.2 – Comunicação Paralela 144
5.2.1 – Problemas com Comunicação Paralela 144
5.2.2 – Comunicação Paralela em Microcontroladores 145
5.3 – Comunicação Serial 146
5.3.1 – Tipos de Interconexões 146
5.3.1.1 – Interconexão Single Ended 146
5.3.1.2 – Interconexão Diferencial 147
5.3.2 – Sincronização Entre Dispositivos 148
5.3.2.1 – Modelos de Sincronização 149
5.3.2.2 – Modelo Mestre-Escravo 151
5.3.3 – Taxas de Transferência – Baude x Bit rate 151
5.3.4 – Protocolos Seriais 152
5.4 – USART (Universal Synchronous and Asynchronous Receiver-Transmitter) 152

5.4.1 – Protocolo da UART ... 153
5.4.2 – Programando UART no ARM Cortex-M ... 154
5.4.3 – LIN (Local Interconnect Network) no ARM Cortex-M ... 156
5.4.6 – Protocolo IrDA: Infrared no ARM Cortex-M ... 157
5.5 – SPI (Serial Peripheral Interface) ... 158
5.5.1 – Protocolo da SPI ... 159
5.5.2 – Programando SPI e DMA ... 161
5.5.3 – Utilização da SPI ... 165
5.6 – I2C (IIC - Inter-Integrated Circuit) ... 166
5.6.1 – Protocolo da I2C ... 167
5.6.2 – Programando I2C no ARM Cortex-M ... 170
5.6.3 – Características e Utilização da I2C ... 173
5.7 – Rede CAN (Controller Area Network) ... 174
5.7.1 – Protocolo da CAN ... 174
5.7.2 – CAN no ARM Cortex-M ... 175
5.7.3 – Características e Utilização da CAN ... 176
5.8 – USB (Universal Serial Bus) ... 177
5.8.1 – Protocolo da USB ... 177
5.8.2 – Programando USB no ARM Cortex-M ... 178
5.8.3 – Características e Utilização da USB ... 180
Parte 2 – PRÁTICAS ... 183
VI – Práticas de Software Embarcado ... 185
6.1 – Prática 01: Piscar LEDs com SysTick e Delay ... 187
6.2 – Prática 02 – Superloop e Máquina de Estados Finitos ... 191
6.3 – Prática 03: Timers e PWM ... 195
6.4 – Prática 04: Interrupções Externas ... 205
6.5 – Prática 05: ADC e SPI ... 213
6.6 – Prática 06: Temporizador ... 225
6.7 – Prática 07: USART com Interrupções ... 235
6.8 – Prática 08: SPI e DMA ... 243
6.9 – Prática 09: I2C por Interrupção ... 251
6.10 – Prática 10: USB ... 259
Referências: ... 265

Introdução

Este livro mostra como são construídos os microprocessadores e como utilizar um microcontrolador para realizar um projeto de eletrônica embarcada. O leitor que nunca teve contato com processadores encontra no livro um capítulo que explica como são criados, a partir de blocos de circuitos eletrônicos digitais, e como funcionam, desde a execução de instruções passo a passo (*assembly*) até programação usando blocos de código em linguagem C, a linguagem mais usada em embarcados.

Os leitores já acostumados com microcontroladores de oito bits têm neste livro a oportunidade de ampliar seus conhecimentos ao utilizar processadores mais avançados, com exemplos práticos da maioria dos periféricos de microcontroladores de 32 bits.

Usamos nesse livro o termo MCU no masculino, como sinônimo de *microcontrolador*, apesar de a sigla em inglês se referir a *microcontroller unity*, cuja tradução literal seria *unidade microcontrolador* em português. Contudo, vamos usar MCU como sigla de microcontrolador, um circuito integrado composto de um processador associado a vários periféricos, que forma um verdadeiro sistema integrado em um único *chip*.

O livro tem como foco a família de MCUs ARM® Cortex-M®. Há inúmeras razões para essa escolha, por exemplo, a facilidade de se encontrar *kits* de desenvolvimento, o custo mais acessível dos *kits*, a disponibilidade de ferramentas de desenvolvimento e depuração de alto nível totalmente gratuitas, dentre outras.

O leitor que tem inclinação para aprender pela prática pode adquirir um *kit* de baixo custo e executar as várias práticas propostas nesse livro. A parte 2 do livro (Práticas) é toda dedicada ao processo “mãos na massa”. São práticas usadas e depuradas nos laboratórios de algumas disciplinas na Universidade Federal do ABC.

Com *kits* de baixo custo e exemplos passo a passo o leitor poderá não só aprender a configurar e utilizar os periféricos dos MCUs, mas também como programar dispositivos embarcados utilizando técnicas de construção de software corretas e mais adequadas para trabalhos profissionais.

Aprender a utilizar ARM Cortex-M é uma escolha estratégica se o leitor busca tecnologia moderna, relevante, flexível e amplamente utilizada no mercado. Além disso, oferece uma curva de aprendizado acessível e inúmeras possibilidades de evolução profissional.

Investir no aprendizado de ARM Cortex-M em vez de outras tecnologias de MCUs (como AVR/Arduino, PIC ou ESP32) pode ser uma decisão estratégica dependendo de seus objetivos e do contexto de uso. Aqui estão algumas razões para optar por Cortex-M:

1. *Domínio do Mercado e Relevância*

A arquitetura Cortex-M é amplamente usada por fabricantes como STMicroelectronics (STM32), NXP (LPC), Texas Instruments, e Nordic Semiconductor, e é líder no mercado de MCUs. Estima-se que MCUs Cortex-M respondam por mais de 50% dos *chips* ARM produzidos anualmente devido ao seu uso em dispositivos IoT, automação, saúde e outras áreas de alta demanda. Em uma estimativa conservadora é razoável estimar que mais de 100 bilhões de MCUs Cortex-M já tenham sido fabricadas globalmente.

Conhecer e saber trabalhar em projetos que usam Cortex-M é muito valorizado em setores como automação industrial, IoT, setor automotivo, dispositivos médicos, dentre outros.

O Cortex-M é licenciado para muitos fabricantes, o que significa que aprender ARM Cortex-M permite trabalhar com diversas famílias de MCUs (STM32, TMC4, LPC, etc.). É possível fazer portabilidade de código, ou seja, o conhecimento de Cortex-M é aplicável a diferentes fabricantes, devido à consistência da arquitetura ARM.

Cortex-M é otimizado para baixo consumo de energia, tornando-o ideal para sistemas embarcados alimentados por bateria ou energia obtida por conversão células fotovoltaicas, como nos dispositivos portáteis e IoT.

2. *Escalabilidade e Flexibilidade*

Há uma variedade de opções dentro da família Cortex-M com ampla gama de modelos, desde o Cortex-M0 (baixo custo e baixa potência) até o Cortex-M7 (alta performance e processamento DSP). Assim, esses dispositivos são aplicáveis em diferentes níveis; você pode usar Cortex-M em aplicações simples (sensores IoT) e em projetos complexos (controle de motores, processamento de áudio, sistemas automotivos).

3. *Ecossistema e Ferramentas de Desenvolvimento*

Existe um conjunto de ferramentas acessíveis que facilitam o desenvolvimento de projetos em nível profissional. ARM Cortex-M é suportado por ampla gama de IDE (*Integrated Development Environment* – ambiente integrado de desenvolvimento), como *ARM Keil MDK*, *IAR Embedded Workbench*, *Code Composer Studio* (CCS – Texas Instruments, tem versão gratuita), *MCUXpresso* (NXP - gratuito), *STM32CubeIDE* (STMicroelectronics, gratuito); além de todos esses ambientes existe suporte a compiladores *GCC* (*open-source*). O "ecossistema ARM" possui ampla documentação, exemplos de código, bibliotecas e suporte de comunidades.

4. *Modernidade e Inovação*

Os MCUs ARM apresentam um conjunto de instruções *Thumb-2*, compacto e eficiente, permite desenvolver programas otimizado para o quesito memória e/ou para velocidade. Os microcontroladores têm controle avançado de interrupções, suporte a ponto flutuante nos MCUs Cortex-M4 e superiores, recursos *TrustZone*, que garantem segurança para aplicações IoT (Cortex-M33 e M23).

5. *Forte Comunidade e Recursos de Aprendizado*

A família de MCUs ARM oferece manuais detalhados (*Datasheets*, TRMs - *Technical Reference Manual*) e guias de desenvolvimento (AN – *Application Notes*). Tudo isso se traduz em documentação abundante e detalhada. Há uma comunidade global com fóruns como ST *Community*, ARM *Developer*, e *Stack Overflow* que estão repletos de suporte em vários níveis, desde iniciantes até bastante avançados. Além disso, há diversos cursos online e materiais gratuitos, como *YouTube*, *Coursera* e livros voltados a Cortex-M.

6. *Preparação para o Futuro*

A ARM continua expandindo sua presença em mercados de ponta, como dispositivos IoT, *wearables*, e inteligência artificial embarcada. Aprender Cortex-M é um passo natural para evoluir para arquiteturas mais avançadas, como Cortex-A (usada em sistemas operacionais embarcados).

Apesar de poderosa, a arquitetura Cortex-M é amigável para iniciantes, como o leitor poderá ver nesse livro, especialmente com ferramentas como STM32CubeMX (que gera código automaticamente) e HAL (*Hardware Abstraction Layer*), uma biblioteca de interface de aplicações dos periféricos. É nessa linha que este livro segue – tentamos facilitar o aprendizado do leitor, além de apresentar uma série de práticas no capítulo final do livro.

As práticas desenvolvidas nesse livro começam por mostram o funcionamento básico do núcleo processador, para os leitores que querem compreender como o núcleo de processamento funciona. Em seguida, as práticas aumentam o nível de complexidade e mostram como programar e utilizar os periféricos mais comumente usados nos MCUs no contexto de eletrônica embarcada.

Breve Histórico

Os microprocessadores surgiram como um marco na história da computação, transformando a forma como os computadores eram projetados, fabricados e utilizados. Destaco os eventos mais marcantes na cronologia de desenvolvimento dessa indústria e do surgimento dos computadores como objetos pessoais.

1. Contexto e Necessidade (Década de 1960)

Na década de 1960 os computadores ocupavam grandes salas e eram compostos por diversos circuitos integrados que desempenhavam funções específicas. Na medida em que a demanda por dispositivos menores e mais eficientes crescia, a integração de componentes em um único *chip* se tornou um objetivo claro.

2. Primeiros Passos: Circuitos Integrados

Os circuitos integrados (CIs) surgiram como uma solução para reduzir o tamanho e o custo dos sistemas eletrônicos. Criados no final dos anos 1950 por Jack Kilby (Texas Instruments) e Robert Noyce (Fairchild Semiconductor), eles integravam vários transistores em uma única peça de silício.

Ou seja, no início dos anos 1960 surgiram os primeiros circuitos integrados lógicos[1], criados pela Fairchild em 1961, que deram origem às famílias de circuitos lógicos TTL e CMOS, nos anos seguintes.

Os primeiros circuitos integrados com portas lógicas e funções lógicas relativamente simples simplificavam a construção dos computadores, mas ainda eram montados sobre grandes placas de circuito integrados (PCB –

printed circuit boards). Fazia-se necessário diminuir o tamanho dessas PCBs, seus custos e, sobretudo, o consumo de energia que apresentavam.

3. Surgimento do Microprocessador (Década de 1970)

O microprocessador é essencialmente uma Unidade Central de Processamento (CPU – *central processor unity*) miniaturizada em um único *chip*. Sua criação foi motivada pela necessidade de simplificar projetos e reduzir custos. O prefixo "micro" no termo "*micro*processador" está relacionado ao tamanho reduzido dos componentes em comparação com os sistemas anteriores. Na década de 1970, a palavra "micro" era usada para indicar miniaturização e compactação. O termo "*micro*processador" surgiu porque o circuito integrado era capaz de executar as funções de uma *CPU*, algo que antes ocupava várias PCBs ou grandes gabinetes, mas agora estava integrado em um único *chip* muito pequeno.

Na época, a tecnologia de fabricação de semicondutores permitia criar transistores e outros componentes no *chip* com dimensões na ordem de *mícrons* (1 micrômetro = 10^{-6} m). Para se ter uma ideia, o primeiro microprocessador tinha transistores com largura de *gate* (porta) de aproximadamente dez *mícrons*. Aquele avanço tecnológico possibilitou a integração de milhares de transistores em um único *chip*, revolucionário para a época. Portanto, embora o prefixo "*micro*" tenha uma conexão indireta com o tamanho dos componentes, ele foi mais amplamente adotado como um termo que refletia a ideia de compactação e miniaturização da tecnologia. Ainda em termos de comparação, atualmente os transistores têm *gates* da ordem de grandeza de alguns nanômetros; embora continuemos a usar o termo de *micro*processadores ou *micro*controladores.

Os primeiros microprocessadores foram: (i) Intel 4004 (1971), desenvolvido por Federico Faggin, Ted Hoff e Stanley Mazor na Intel, é considerado o primeiro microprocessador comercialmente disponível. Era um *chip* de 4 bits projetado inicialmente para calculadoras eletrônicas. Em 1969, a Busicom (*Nippon Calculating Machine Corporation*) contratou a Intel para desenvolver um conjunto de *chips* personalizados para sua nova linha de calculadoras eletrônicas. Em 1971, o Intel 4004 foi concluído; um processador de 4 bits, operava a 740 KHz, capaz de realizar 92.000 operações por segundo e possuía 2.300 transistores, sendo extremamente inovador para sua época. Posteriormente vieram (ii) Intel 8008 (1972) e Intel 8080 (1974) com avanços significativos, aumentando a capacidade de processamento e abrindo caminho para computadores pessoais. O Intel 8080 tinha aproximadamente 6.000 transistores, operava em 2 MHz, permitia executar cerca de 640.000 instruções por segundo, e era capaz de acessar até 64 Kbytes de memória. Essas características permitiram o desenvolvimento dos primeiros computadores pessoais.

4. Consolidação e Popularização (Anos 1980)

Com a introdução de processadores mais avançados, como o Intel 8086 (base dos primeiros PCs da IBM) e seus concorrentes, os *microprocessadores* tornaram-se o núcleo de computadores pessoais, consoles de videogame, automação industrial e outros dispositivos.

5. Revolução Contínua

Desde então, os microprocessadores evoluíram em termos de potência, eficiência energética e complexidade. A Lei de Moore, proposta por Gordon Moore (cofundador da Intel), previu o aumento exponencial do número de transistores em um *chip*, algo que impulsionou a inovação por décadas. Atualmente (2025), os microprocessadores estão presentes em praticamente todos os dispositivos eletrônicos, desde smartphones até supercomputadores, sendo uma das maiores conquistas tecnológicas do século XX.

De Microprocessadores a Microcontroladores

As CPUs dos computadores pessoais são ainda hoje chamadas de microprocessadores. Contudo, durante os anos que seguem o surgimento dos computadores pessoais, as empresas começaram a integrar mais e mais circuitos em um único *chip*. O surgimento de unidades microcontroladoras (*MCU - microcontroller unit)* está intimamente ligado à necessidade de soluções mais compactas, acessíveis e otimizadas para produzir dispositivos com computação embarcada, comumente chamado de *sistemas embarcados*.

Podemos definir *sistema embarcado* toda eletrônica que executa algum tipo de computação, mas que não é um computador de uso genérico; ou seja, a característica da eletrônica embarcada é realizar computação específica, para resolver problemas pontuais de um certo dispositivo, produto, equipamento, etc.

MCUs se destacam pela integração de CPU, memórias RAM e ROM (não voláteis), associados ainda a uma série de periféricos em um único *chip*. MCUs têm baixo consumo de energia, o que é essencial para sistemas alimentados por bateria; representam economia porque possuem excelentes condições de custo-benefício, o que os tornam ideais para aplicações específicas e produção em larga escala; são relativamente fáceis de usar porque permitem projetos simplificados e têm conjuntos de ferramentas de desenvolvimento acessíveis. Tendo isso em conta, podemos desenhar um panorama histórico sobre como os MCUs surgiram:

1. Contexto: Limitações dos Microprocessadores (Anos 1960 e 1970)

Os primeiros microprocessadores, como o Intel 4004 (1971) e o Intel 8080 (1974), eram soluções poderosas para processamento geral, mas exigiam *periféricos externos* como memória, interfaces de entrada/saída (I/O) e temporizadores para formar um sistema completo. Essa abordagem resultava em placas de circuitos impresso (PCB) grandes e caras, inadequados para aplicações específicas que precisavam de soluções compactas e econômicas, como eletrodomésticos e equipamentos industriais. O problema a resolver era a necessidade de uma solução integrada para *sistemas embarcados*.

2. O Primeiro Passo: Intel 8048 (1976)

A Intel desenvolveu o 8048, amplamente reconhecido como o primeiro microcontrolador comercial. Ele integrava CPU de 8 bits, RAM (apenas 64 bytes) e ROM (1 KBytes); tinha ainda um periférico de entrada/saída (GPIO - *General-purpose input/output*) e temporizadores. O 8048 foi usado em aplicações como teclados do IBM PC e sistemas industriais. Sua criação respondeu diretamente à necessidade de integrar funcionalidade em um único *chip*, reduzindo custo e complexidade.

3. Competição e Expansão

A ideia de uma unidade integrada atraiu outras empresas que começaram a desenvolver seus próprios MCUs. Exemplos incluem a Texas Instruments TMS1000 (1974), considerado o primeiro MCU de 4 bits completamente integrado. Usado em brinquedos e calculadoras, foi projetado antes do Intel 8048, todavia era menos "maleável". Além dessa, os Motorola 6800 e 6801, que foram alternativas com mais recursos para sistemas embarcados.

4. Popularização: Automação e Consumo

Nos anos 1980, o avanço dos MCUs foi impulsionado pela crescente demanda por automação em indústrias e eletrodomésticos. Por exemplo, nos automóveis MCUs começaram a ser usados em sistemas de controle de motor e *airbag*; em eletrodomésticos MCUs simplificaram a operação de fornos de micro-ondas, máquinas de lavar e vários outros eletrodomésticos e dispositivos; e nos produtos de consumo MCUs foram e são usados em brinquedos eletrônicos, câmeras fotográficas e vídeo, vários dispositivos eletrônicos pessoais, etc.

5. Evolução Contínua (Anos 1990 e 2000)

O avanço na miniaturização e no processo de fabricação (submicrométrico) permitiu a criação de MCUs mais poderosos e eficientes. Exemplos são as famílias clássicas: PIC (Microchip), AVR (Atmel), e MSP430 (Texas Instruments). Com a miniaturização houve integração de novas tecnologias, nova interfaces, e periféricos, como as interfaces de comunicação (USART, SPI, USB, I^2C)[1], os conversores analógico-digital ADCs e digital-analógicos DACs; e, mais recente, a conectividade sem fio (*Wi-Fi*, *Bluetooth*).

A necessidade de MCUs mais potentes, escaláveis e com suporte para software moderno, sem sacrificar eficiência energética, levou a ARM (ARM Holdings – *Advanced RISC Machine*) a entrar no mercado de MCUs para preencher a lacuna entre processadores de alto desempenho e microcontroladores simples. Em 2004, a ARM introduziu o Cortex-M3, o primeiro microcontrolador da família Cortex-M. Baseado na arquitetura ARMv7-M, ele foi projetado especificamente para aplicações embarcadas que exigiam alta eficiência energética, baixo custo e processamento avançado. Os MCUs Cortex-M trouxeram inovações que transformaram o mercado, como *pipeline simplificado* com projetos eficientes em energia baseados no estilo RISC (*Reduced Instruction Set Computing* – computação com conjunto de instruções reduzidas); unidade de ponto flutuante (opcional) que dão suporte a cálculos mais complexos, controlador de interrupções capaz de gestão avançada de interrupções, o que é essencial para sistemas de tempo real, Além disso, possui um conjunto de instruções compacto e eficiente,

[1] I^2C (*inter-integrated circuit*) em inglês se pronuncia: "*ai squér si" (i ao quadrado C*), devido a dois 'I's seguidos na sigla. Em muitos documentos encontramos a grafia acima, com o '2' sobrescrito, o mesmo ocorre com I^2S (*inter-integrated circuit sound*). Contudo, no livro optou-se por grafar I2C e I2S (sem sobrescrito) porque são comumente usadas.

associado a um ecossistema robusto, com compatibilidade com ferramentas de desenvolvimento amplamente adotadas (Keil, Eclipse, GCC, etc.). Este livro foca no desenvolvimento de projetos e programas para linha de MCUs Cortex-M.

O impacto de todos esses aperfeiçoamentos listados acima é que MCUs revolucionaram a eletrônica embarcada, tornando viáveis dispositivos autônomos, baratos e compactos. Elas abriram caminho para inovações como a *Internet das Coisas* (IoT) e continuam evoluindo, incorporando tecnologias de conectividade, inteligência artificial e sensores avançados. Atualmente, os MCUs são onipresentes, desde controles remotos até sistemas complexos como drones e dispositivos médicos.

Espero que o leitor faça bom proveito do livro e que aprenda a lidar com toda essa tecnologia de ponta sem muito sofrimento, apesar de esperar sim um esforço contínuo e determinação. Absorver todo esse conteúdo e aprender a lidar com essa tecnologia não acontece *por osmose*. O leitor deve *praticar*, ampliar e buscar informações complementares para dirimir dúvidas que por ventura o livro tenha deixado. Manuais, tutoriais na internet, notas de aplicação (AN), vídeos, comunidades e grupos específicos deste assunto estão amplamente disponíveis online.

Aos alunos da UFABC, nos cursos de engenharia, é ministrado um curso de um quadrimestre somente sobre o assunto desse livro, na disciplina ESTI013, SISTEMAS MICROPROCESSADOS, um curso de introdução aos microprocessadores.

Após esse curso os alunos têm outras disciplinas específicas sobre técnicas de programação em embarcados (ESZI041, PROGRAMAÇÃO DE SOFTWARE EMBARCADO) e de aplicações de MCUs (ESZI025 – APLICAÇÕES DE MICROCONTROLADORES).

Como exemplo, em programação de software embarcado vemos técnicas e boas formas de se criar software para eletrônica embarcada, assunto para outro livro. Nela vemos problemas como *race condition*, que apenas arranhamos nesse livro, problemas de competição no uso de recursos do MCU, divisão do tempo da CPU entre várias tarefas (*tasks/threads*) por meio de RTOS; troca de dados e informações entre *tasks* por meio de filas (*queue*) e outros recursos; comunicação entre dispositivos, abordada com maior profundidade que neste livro; assuntos como *event-driven software*, *run to completion* e Máquinas de Estados Hierárquicas; e os paradigmas mais modernos na construção de software para embarcados, como *active object*, suas possibilidades e vantagens.

Note leitor que o assunto não se esgota no que se encontra neste livro, muito pelo contrário, ele começa aqui. Trata-se de uma bela empreitada, que o leitor deve encarar com serenidade, perseverança e determinação.

Boa jornada.

Parte 1 – FUNDAMENTOS

I – Microprocessador (MCU)

Os leitores que já conhecem como funcionam os núcleos de processamento (CPU – *central processing unit*) podem saltar para o tópico *1.5 Microcontrolador ARM Cortex-M* sem prejuízo na compreensão do restante do livro, embora uma revisão de conceitos nunca seja demais e ajude a compreender detalhes dos periféricos dos MCUs. Na primeira parte desse capítulo vamos mostrar como se constrói um *processador* usando elementos de eletrônica digital, para em seguida apresentar como é a constituição de um microcontrolador Cortex-M.

1.1 – Revisão de Eletrônica Digital

Uma breve revisão de conceitos basais de eletrônica digital para compreensão do funcionamento e construção de CPUs. Fundamentos mais básicos[2], por exemplo, o que são portas e funções lógicas estão fora do escopo desse livro[2]. Vamos ver algumas soluções técnicas particularmente importante para entender CPUs.

1.1.1 – Interconexão de Saídas Digitais

As entradas de circuitos digitais podem ser interconectadas porque as impedâncias de entradas das portas lógicas são muito altas, além de os sinais de entrada serem controlados externamente. Contudo, as saídas de circuitos lógicos não podem ser interconectadas umas com as outras. A razão pode ser vista na Fig.1.1; tomando como exemplo as saídas de dois inversores CMOS.

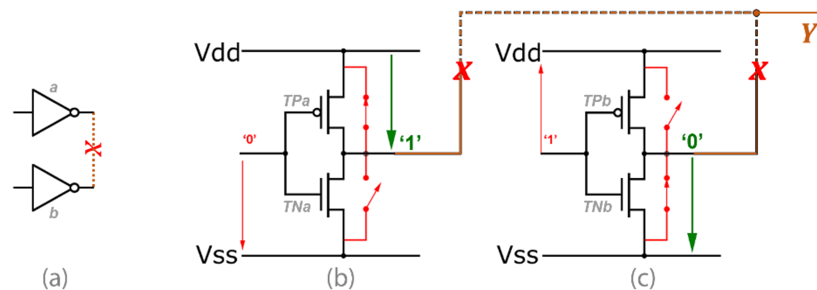


*Fig. 1.1: Saídas em curto-circuito. Nas saídas CMOS push-pull, um transistor liga a saída ao **+V** enquanto outro pode conectar sua saída ao **GND**. Por isso NÃO podemos fazer curto-circuito entre saídas.*

Quando o transistor canal P (TPa) do inversor 'a' funcionar como chave fechada ele joga +Vdd na saída. Se o transistor canal N (TNb) do inversor 'b' estiver em estado ON ele conectará da saída 'b' com Vss. Isso significa que nesse caso os circuitos inversores provocam um curto circuito entre Vdd e Vss. Para resolver a situação existem algumas soluções. A Fig. 1.2 ilustra as soluções mais usuais.

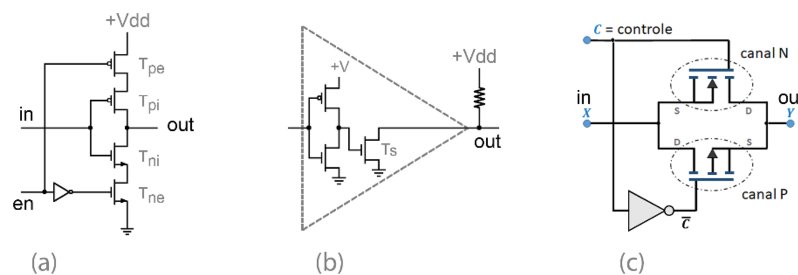


Fig. 1.2: Solução tri-state (a), solução open-drain (b), e a solução transmission gates (c).

A solução *tri-state* deixa a saída do circuito em alta impedância, desconectando +Vdd e Vss por meio dos transistores Tpe e Tne (Fig. 1.2.a) dos transistores *push-pull* (Tpi e Tni) de saída CMOS normal. A saída é colocada em estado de alta impedância quando a entrada de habilitação (*en* na Fig. 1.2.a). Quando esse sinal está em nível 0 os transistores Tpe e Tne funcionam como chave aberta.

---

[2] No livro "Eletrônica Digital, Verilog e FPGA" discuto em detalhe os conceitos e circuitos explicados abaixo, além do básico sobre sistemas de numeração e codificação binários, de portas lógicas, circuitos combinacionais e sequenciais, memória, linguagem Verilog, etc. ISBN 978-65-001-7065-8 https://clubdeautores.com/libro/eletronica-digital-verilog-e-fpga.

A solução *open-drain* insere um transistor (Ts na Fig. 1.2.b) em série com as saídas *push-pull* normais, porém esse transistor Ts não está polarizado (não tem seu dreno conectado à +V). A polarização do transistor deve ser feita por um resistor *pull-up* externo à porta de saída, como mostra a figura.

A terceira solução também utiliza um sinal de habilitação chamado *Controle* (C na Fig. 1.2.c). Esse sinal controla o estado de ativação dos transistores canal N e canal P ligados em contra fase na porta, também chamadas *transmission gates*, ou chaves analógicas. Pela Fig. 1.2.c, vemos que com o sinal de controle em nível lógico 1 o transistor canal N está ativado, e o mesmo ocorre com o canal P porque o sinal C é invertido antes de ativar o *gate* do canal P. Se C está em nível lógico 0 a porta se comporta como chave totalmente aberta porque os dois transistores têm seus *gates* despolarizados. Esse circuito é bom para transmitir valores contínuos sem distorcer os sinais, porque ao transmitir sinais perto de V_{DD} (ruim para o transistor canal N) o transistor canal P está completamente saturado e garante a chave em plena condução. Por outro lado, sinais com valores perto de Vss (ruim para o transistor canal P) o transistor canal N encontra-se totalmente saturado em plena condução.

Essa rápida revisão é necessária porque esses três tipos de solução tecnológica são usados em MCUs e nos periféricos que eles possuem. Por exemplo, transceptores de barramento usam lógica *tri-state*, as interfaces de comunicação I2C são baseadas em circuitos *open-drain*, e as entradas de sinais analógico para os conversores ADC passam por circuitos multiplexadores que usam *transmission gates*.

1.1.2 – Transceptor de Barramento

Baseado na tecnologia *tri-state* descrita acima, vamos compreender um circuito que é muito utilizado para controle do fluxo de dados em CPUs. Trata-se do circuito transceptor de barramento. A Fig. 1.3 mostra um circuito muito conhecido em lógica, o 74HC245 (*8-Bits Bus Transceiver*).

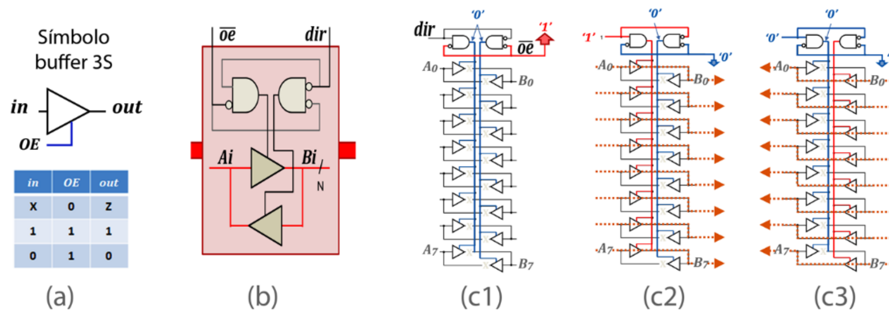


Fig. 1.3: Bus Transceiver. (a) Símbolo e tabela verdade de um buffer 3-state, (b) conexão de buffers 3-S em contra fase com sinais de habilitação ($\overline{oe}$) e sinal de direção (dir) comuns; (c1) quando ($\overline{oe}$=1) o barramento desconecta os lados 'A e 'B', (c2) com ($\overline{oe}$=0) e 'dir=1' ligando os fios do lado 'A' aos do 'B', (c3) com ($\overline{oe}$=0) e 'dir=0' ligando os fios de 'B' ao lado 'A'. Detalhes no texto.

A função do *bus transceiver* é controlar o fluxo de dados em um conjunto de fios. Um conjunto de fios recebe o nome de barramento (bus) que transportam os sinais digitais do menos para o mais significativo. No desenho vemos um lado do barramento que transporta 8 bits denominado $\boldsymbol{A}$ (de A_0 até A_7), e outro lado do barramento também com 8 bits denominado $\boldsymbol{B}$ (de B_0 até B_7).

O circuito desconecta completamente os sinais $\boldsymbol{A}$ dos fios $\boldsymbol{B}$ quando o sinal $\overline{oe}$=1, porque esse sinal tem que estar em nível baixo para habilitar o *transceiver*, como na Fig. 1.3.c1. Quando o sinal $\overline{oe}$=0, então o barramento conecta o lado $\boldsymbol{A}$ com o $\boldsymbol{B}$ dependendo do sinal *dir*. Se *dir* = 1 então os dados presentes nos fios do lado $\boldsymbol{A}$ passam para os fios do lado $\boldsymbol{B}$ (Fig.1.3.c2), mas se o sinal *dir* = 0 então os dados presentes nos fios do lado $\boldsymbol{B}$ passam para os fios do lado $\boldsymbol{A}$. Portanto, podemos pensar que o transceptor apresenta 3 estados operacionais: isola os lados $\boldsymbol{A}$ e $\boldsymbol{B}$, deixa passar dados de $\boldsymbol{A}$ para $\boldsymbol{B}$, ou deixa passar dados de $\boldsymbol{B}$ para $\boldsymbol{A}$. Os transceptores de barramento podem ter 8, 16, 32 ou mais bits (ou fios) dos lados $\boldsymbol{A}$ e $\boldsymbol{B}$.

1.1.3 – Latches e Registradores

Um *latch*[3], ou *flip-flop* é um circuito capaz de memorizar um bit[3]. Os símbolos dos *latches* ativados na borda de subida ou descida podem ser vistos na Fig. 1.4.a. Um latch tem uma entrada denominada *D*, uma saída denominada *Q*, um sinal de controle denominado *LE* (*latch enable*). Há casos de latches com saída *3-state* e um

[3] Um bom livro sobre projetos de sistemas digitais: *Digital Design (Verilog): An Embedded Systems Approach Using Verilog*; 2007, Peter J. Ashenden (autor), Morgan Kaufmann. Ver capítulo IV, Latches.

sinal de controle OE (*output enable*). Não vem ao caso aqui como é o circuito eletrônico em si porque é assunto de eletrônica digital, mas o funcionamento do *latch* interessa.

Quando aplicamos um sinal de habilitação no LE o *latch* deixa o bit na entrada D se propagar até a saída Q. Diz-se que o *latch* é transparente nesse tipo de construção. O sinal de habilitação pode ser LE=1 ou $\overline{LE}$=0, quando a habilitação é em nível alto ou nível baixo, respectivamente, e depende da construção do circuito. A *mágica* ocorre quando o sinal do LE muda do nível habilitado para o nível não habilitado. Na descida ↓ do sinal (transição de 1→0) ou na subida ↑ do sinal (transição de 0→1) o *latch* retém o último valor do bit presente na entrada D. Se o *latch* não tem saída 3-S, o valor retido aparece imediatamente na saída Q. Porém, quando o latch tem controle de saída OE, a habilitação de saída deve estar em OE=1 ou $\overline{OE}$=0 para que o dado Q' do *latch* seja mostrado na saída Q. Nesse caso, estamos chamando de Q' o valor interno do *latch*, antes de passar pelo *buffer 3-state*, como pode ser visto na Fig. 1.4.d.

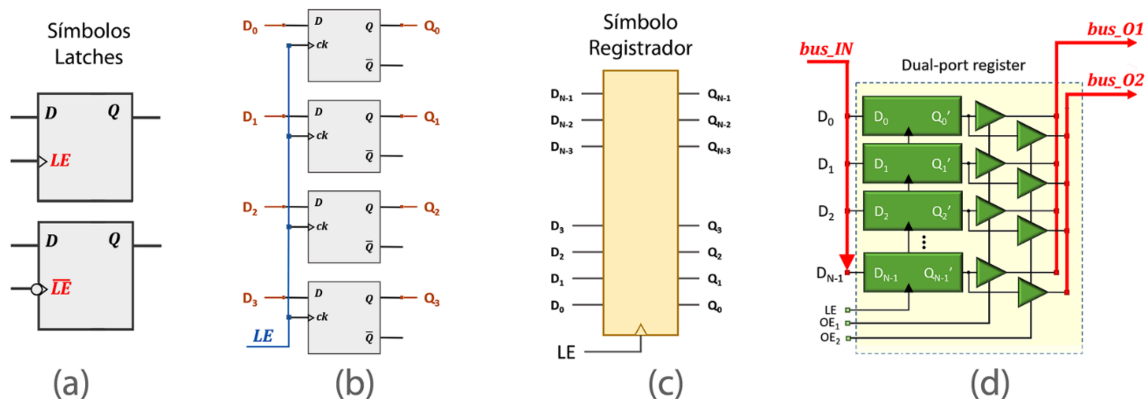


Fig. 1.4: Latches e Registradores. (a) Símbolos dos latches transparentes, (b) conexão de quatro latches formando um registrador de quatro bits; (c) símbolo de um registrador de N bits, (d) registrador com duas portas de saída de dados.

Podemos ligar vários *latches* em paralelo e conectar todos os LEs em um só sinal de controle, como mostra a Fig. 1.4.b. Nesse caso, criamos um circuito capaz de memorizar um vetor de bits. Esse circuito se chama *registrador* e podemos ter registradores de 8, 16, 32, 64 bits ou qualquer número de bits, dependendo da quantidade de *latches* ligados em paralelo. A Fig. 1.4.c ilustra o símbolo de um registrador transparente de N bits. Por outro lado, a Fig. 1.4.d ilustra um registrador construído com duas saídas *3-states*. Note que os bits chegam ao registrador pelo barramento (vetor de bits) *bus_IN*, e são copiados para dentro de cada *latch* quando o sinal LE for ativado. Contudo, o circuito tem duas saídas, uma habilitada pelo sinal OE_1 que, se habilitado, faz os dados Q_i' do registrador saírem pelo o barramento *bus_O*$_1$; e outra saída OE_2 que, se habilitada, faz os dados Q_i' do registrador saírem pelo o barramento *bus_O*$_2$. Esse tipo de registrador é chamado *dual-port register*.

Registradores são importantes para reter dados e mantê-los estáveis enquanto a unidade de lógica e aritmética da CPU realiza operações com os valores; ou seja, registradores retém momentaneamente os operandos para execução de operações computacionais.

1.1.4 – Memórias

Junte vários registradores e coloque um circuito de controle para que apenas um registrador esteja ativo por vez e você construirá uma memória RAM (*Random Access Memory*). O circuito de controle tem então que cuidar de três fatores: (i) endereçar apenas um dos registradores na memória, (ii) controlar se o dado está entrando ou saindo dos registradores, e (iii) copiar o dado presente na entrada para um registrador específico ou habilitar um registrador específico para colocar o dado na saída.

Para endereçar um dos registradores da memória usa-se um barramento chamado *bus address* (barramento de endereços). Esse barramento tem um vetor de bits que aponta para apenas uma *posição* da memória; portanto, cada registrador na memória de agora em diante será denominado apenas posição de memória. Para controlar se o processador está guardando dados (escrevendo na memória) ou buscando dados (fazendo uma leitura) temos alguns fios que realizam essas funções de sinalização. Por último, um barramento de dados serve tanto para entrada quanto para saída dos dados da memória. Quando dizemos que uma memória é de 8 bits quer dizer que o barramento de dados tem 8 fios em paralelo para trazer dados da CPU para a memória, ou para buscar da

memória para a CPU. Existem sistemas criados com barramento de dados de 16 bits, 32 bits, como é o caso do ARM Cortex-M que vamos trabalhar, e barramento de 64 bits já estão presentes em computadores pessoais.

Sobre memória, apenas mais dois fatores a serem observados: quando uma memória é de acesso aleatório ela recebe o nome de RAM, e a característica básica das RAMs é que elas perdem os dados quando a energia é desligada, são ditas voláteis. Contudo, existem outros tipos de memória que não são voláteis, ou seja, não perdem seus conteúdos quando desenergizadas. Essas são chamadas ROM (*Read Only Memory*). Apesar de serem regraváveis, as memórias *Flash* são comumente chamadas de ROM por não serem voláteis, e as *Flash* atualmente são de longe as mais utilizadas para memorizar programas em MCUs.

A Fig. 1.5 ilustra o circuito conceitual com o símbolo para uma memória com 2^k posições (endereços) que é capaz de memorizar n bits por posição. Diz-se, por exemplo, memória de 64K por 8 bits, ou 128K por 16 bits.

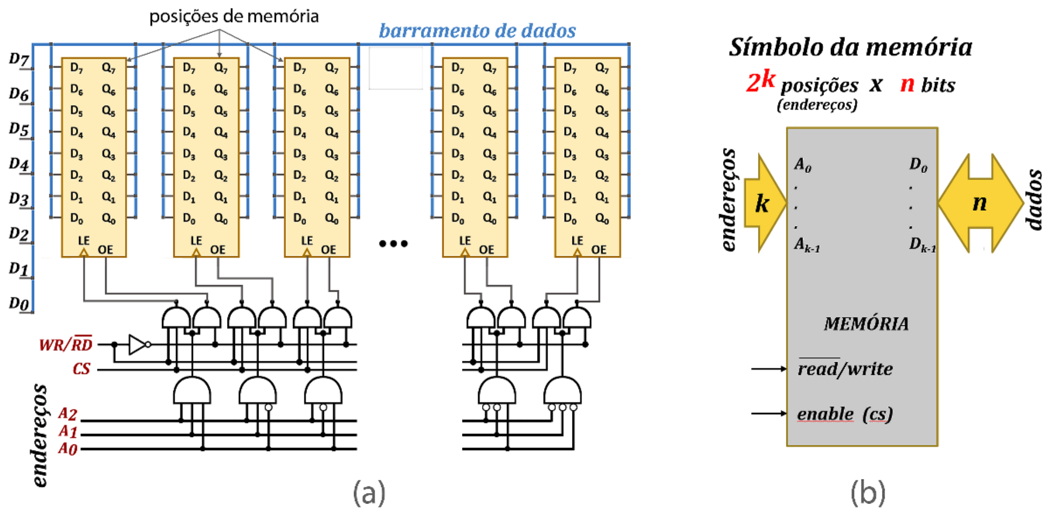


Fig. 1.5: Conceito de memória. (a) circuito de uma RAM com registradores em cada endereço; (b) símbolo das memórias, neste exemplo com 'k' linhas de endereços, 'n' bits para dados, um sinal que controla leitura (0) e escrita (1), e um chip enable.

Portanto, tanto computadores pessoais quanto os microcontroladores ambos possuem uma certa quantidade de memória RAM para trabalhar com dados em processamento, além de certa quantidade de memória ROM onde fica armazenado o programa inicial (*boot*) que os processadores precisam rodar quando sofrem *reset*. No caso de MCUs, os programas geralmente são gravados na sua integridade dentro das memórias *Flash*.

1.1.5 – Unidade Lógica e Aritmética

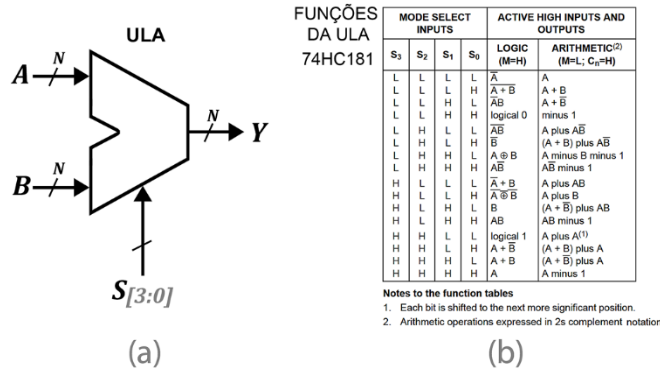
Em sistemas digitais podemos combinar portas lógicas e construir somadores, subtratores, multiplicadores, comparadores, e circuitos que executam operações lógica bit a bit (*bitwise operation*) em um vetor (ou *string*) de bits. Por depender apenas de combinações de portas esses circuitos são chamados 'combinacionais' e têm como característica serem muito rápidos, porque o atraso entre entrada dos vetores (operandos) e a saída do resultado depende apenas da propagação dos sinais pelas portas lógicas.

Quando juntamos todos esses circuitos em um único circuito chamamos o circuito resultante de ALU (*arithmetic logic unit*; em português ULA[4] - *Unidade Lógica Aritmética*)[4]. Uma ULA[5] tem como entrada um, dois ou mais operandos que são vetores de *'n'* bits, dependendo de como ela foi construída. Como ela é capaz de fazer várias operações, é preciso indicar qual operação queremos que ela execute. Para isso, há sempre nas ULAs um conjunto de fios que funcionam como seletores de operação. A Fig. 1.6 ilustra o símbolo de uma ULA.

Pela Fig. 1.6.a vemos que uma ULA terá sempre operandos de entrada, no caso as entradas A e B de '*n*' bits, dependendo da construção da ULA. A saída Y apresentará a resposta da operação pouco tempo depois de os valores de A e B estarem estáveis e do sinal de seleção S apontar qual operação deverá ser feita. Pode-se considerar a latência para a resposta na casa de dezenas, ou até mesmo unidades de nanosegundos.

[4] Ver *Sistemas Digitais: Princípios e Aplicações*, 2019, 12ª Edição Português, R. Tocci, N. Widmer, G. Moss (Autores), Editora Pearson Universidades. No livro há uma explicação detalhada sobreo CI 74HC382.
[5] É muito mais comum no Brasil se dizer ULA do que ALU, porisso vamos preferir usar a sigla ULA no livro.

Para fins de ilustração, a Fig. 1.6.b mostra a tabela de operações do circuito integrado comercial 74HC181, uma ULA de 4 bits capaz de realizar até 32 operações, porque se o sinal $\boldsymbol{M}$ = 1 (modo) a ULA faz operações lógicas, e se $\boldsymbol{M}$ = 0 a ULA faz operações aritméticas. Esse Por exemplo, se $\boldsymbol{S}$ = 0001 e $\boldsymbol{M}$ = 1, a ULA faz a operação lógica dos bits de ***A NOR*** bits de $\boldsymbol{B}$ e o resultado sai em $\boldsymbol{Y}$; ou seja, $\boldsymbol{Y} = (\overline{\boldsymbol{A} + \boldsymbol{B}})$. Pense cada bit A_i e B_i como as entradas de uma porta ***NOR***, e cada saída dessas portas fosse Y_i. Por outro lado, se o vetor $\boldsymbol{S}$ = 0001 e $\boldsymbol{M}$ = 0, então a ULA fará a operação aritmética de soma binária $\boldsymbol{Y} = \boldsymbol{A} + \boldsymbol{B}$.



MODE SELECT INPUTS				ACTIVE HIGH INPUTS AND OUTPUTS	
S_3	S_2	S_1	S_0	LOGIC (M=H)	ARITHMETIC(2) (M=L; C_n=H)
L	L	L	L	$\overline{A}$	A
L	L	L	H	$\overline{A + B}$	A + B
L	L	H	L	$\overline{A}B$	$A + \overline{B}$
L	L	H	H	logical 0	minus 1
L	H	L	L	$\overline{AB}$	A plus $A\overline{B}$
L	H	L	H	$\overline{B}$	(A + B) plus $A\overline{B}$
L	H	H	L	$A \oplus B$	A minus B minus 1
L	H	H	H	$A\overline{B}$	$A\overline{B}$ minus 1
H	L	L	L	$\overline{A} + B$	A plus AB
H	L	L	H	$\overline{A \oplus B}$	A plus B
H	L	H	L	B	$(A + \overline{B})$ plus AB
H	L	H	H	AB	AB minus 1
H	H	L	L	logical 1	A plus A(1)
H	H	L	H	$A + \overline{B}$	(A + B) plus A
H	H	H	L	A + B	$(A + \overline{B})$ plus A
H	H	H	H	A	A minus 1

Notes to the function tables

1. Each bit is shifted to the next more significant position.
2. Arithmetic operations expressed in 2s complement notation.

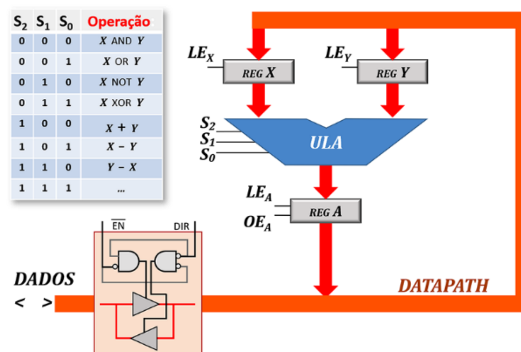
Fig. 1.6: ULA. (a) Símbolo da ULA que tem os operandos 'A' e 'B' como entrada de 'n' bits e a saída 'Y' também com 'n' bits. A entrada 'S' controla a operação desejada. (b) Tabela de operações da ULA em um circuito integrado comercial 74HC181.

Em um processador a ULA é o circuito crucial porque cada operação aritmética, de comparação, de lógica, de tomada de decisão, passa por uma operação na ULA.

Devido ao fato de a ULA precisar de ao menos um operando e da execução dos cálculos tomar algum tempo, usa-se registradores para armazenar e estabilizar os valores desses operandos durante os cálculos. Da mesma forma, depois de um tempo em que é conhecido o resultado estável da saída da ULA, usa-se outro(s) registrador(es) para memorizar os resultados, como veremos adiante.

1.1.6 – ULA, Registradores e Datapath

A Fig. 1.7 ilustra uma ULA associada a registradores e um caminho para os dados atingir os registradores que servem como memória de rascunho para os operandos da ULA.



S_2	S_1	S_0	Operação
0	0	0	X AND Y
0	0	1	X OR Y
0	1	0	X NOT Y
0	1	1	X XOR Y
1	0	0	X + Y
1	0	1	X - Y
1	1	0	Y - X
1	1	1	...

Fig. 1.7: Núcleo Computacional. Caminho para dados (datapath) com transceptor de barramento na entrada, registradores X e Y de entrada da ULA, e registrador A que recebe o resultado da operação da ULA, cuja operação pode ser controlada pelo vetor S.

O caminho que os dados podem percorrer recebe o nome de '*datapath*'. Note que há um transceptor de barramento que controla a direção dos dados que entram ou saem do *datapath*. Dados tanto podem entrar como sair desse núcleo. Quando queremos fornecer um operando podemos colocar o valor na entrada do *datapath*, habilitar o *bus-transceiver* ($\overline{EN}$=0), controlar a direção para dentro (DIR=1), e ao mesmo tempo vamos ligar o sinal de habilitação LE_X ou LE_Y (ou ambos) para que o(s) *latch(es)* memorize(m) esse valor.

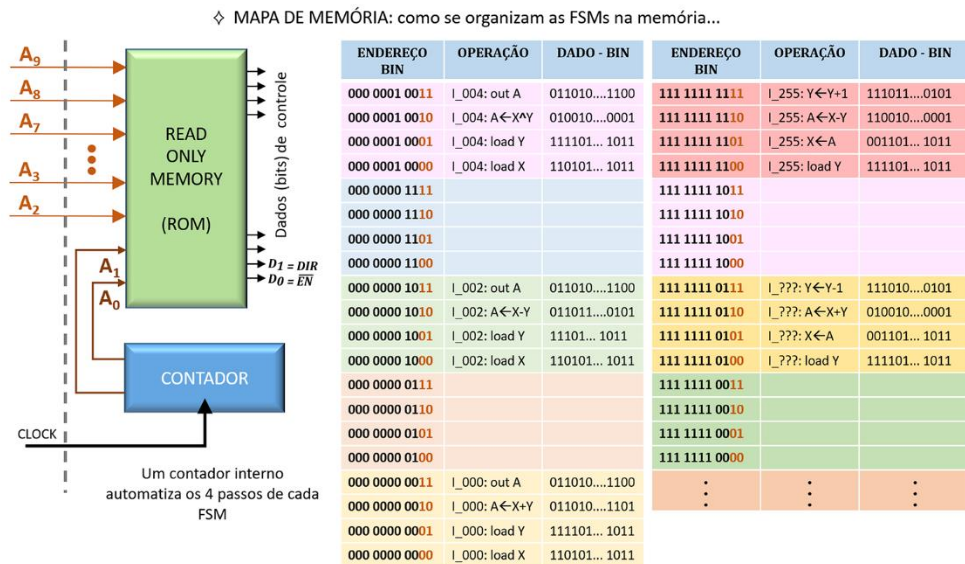
Podemos fazer as operações em etapas. Por exemplo, passo 1: colocamos o dado na entrada, habilitamos o transceptor para dentro e aplicamos LE_X para memorizar o operando #1 em $\boldsymbol{X}$. Em um passo 2, substituímos o dado na entrada do *datapath* e aplicamos LE_Y para memorizar o operando #2 em $\boldsymbol{Y}$. Concomitante, podemos ajustamos o valor do vetor S para indicar à ULA que operação executar. Os operandos estão memorizados em $\boldsymbol{X}$ e $\boldsymbol{Y}$, assim, depois de algum atraso, a ULA apresenta o resultado da operação selecionada. No passo 3, ativamos

o sinal LE_A para que o registrador $\boldsymbol{A}$ memorize esse resultado. O passo 4 nessa sequência é enviar o resultado para fora do núcleo. Para isso, habilitamos o *bus-transceiver* ($\overline{EN}$=0) e controlamos a direção para fora (DIR=0). Ao mesmo tempo, habilitamos a saída dos dados do registrador $\boldsymbol{A}$ ativando o sinal OE_A para que os dados do registrador sejam colocados no *datapath*. Os dados de $\boldsymbol{A}$ passam pelo *bus-transceiver* e podem ser utilizados por um circuito externo a esse núcleo. Enquanto nos passos 1 e 2 um circuito externo ao núcleo fornece dados ao *datapath*, no passo 4 esse circuito externo não pode tentar colocar dados no *datapath* para não ocorrer conflito. No passo 4 o circuito externo deve receber os dados vindos do núcleo.

1.2 – Processadores

No exemplo didático do núcleo anterior, vimos que temos que controlar vários sinais para que a sequência ocorra corretamente. Controlando os sinais $\overline{EN}$, DIR, LE_X, LE_Y, LE_A, OE_A e valor do vetor $\boldsymbol{S}$ podemos criar várias sequências de cálculo da forma que desejarmos. Podemos usar chaves físicas para manualmente controlar cada um dos sinais e assim criaremos uma ‘calculadora’ capaz de executar tantas operações quantas a ULA for projetada para executar. Porém, controlar cada um dos passos manualmente não parece muito produtivo.

Uma solução para melhorar as sequências de operações seria usar uma ROM e gravar nela como devem estar cada um desses sinais, $\overline{EN}$, DIR, LE_X, LE_Y, LE_A, OE_A e $\boldsymbol{S}$, em cada um dos passos. No exemplo acima precisamos de 4 passos, então podemos usar 2 bits menos significativos de endereço ($\boldsymbol{A_1}$, $\boldsymbol{A_0}$) da ROM para apontar posições subsequentes da memória, ou seja, nos endereços terminados em 00, 01, 10, 11. A Fig. 1.8 ilustra como pode ser esse circuito e o mapa de dados gravados na memória. Note que um contador controla os endereços ($\boldsymbol{A_1}$, $\boldsymbol{A_0}$), e esse contador começa sempre com 00 e conta até o valor 11, garantindo a sequência de passos.



ENDEREÇO BIN	OPERAÇÃO	DADO - BIN
000 0001 0011	I_004: out A	011010....1100
000 0001 0010	I_004: A←X^Y	010010....0001
000 0001 0001	I_004: load Y	111101... 1011
000 0001 0000	I_004: load X	110101... 1011
000 0000 1111		
000 0000 1110		
000 0000 1101		
000 0000 1100		
000 0000 1011	I_002: out A	011010....1100
000 0000 1010	I_002: A←X-Y	011011....0101
000 0000 1001	I_002: load Y	11101... 1011
000 0000 1000	I_002: load X	110101... 1011
000 0000 0111		
000 0000 0110		
000 0000 0101		
000 0000 0100		
000 0000 0011	I_000: out A	011010....1100
000 0000 0010	I_000: A←X+Y	011010....1101
000 0000 0001	I_000: load Y	111101... 1011
000 0000 0000	I_000: load X	110101... 1011

ENDEREÇO BIN	OPERAÇÃO	DADO - BIN
111 1111 1111	I_255: Y←Y+1	111011....0101
111 1111 1110	I_255: A←X-Y	110010....0001
111 1111 1101	I_255: X←A	001101... 1011
111 1111 1100	I_255: load Y	111101... 1011
111 1111 1011		
111 1111 1010		
111 1111 1001		
111 1111 1000		
111 1111 0111	I_???: Y←Y-1	111010....0101
111 1111 0110	I_???: A←X+Y	010010....0001
111 1111 0101	I_???: X←A	001101... 1011
111 1111 0100	I_???: load Y	111101... 1011
111 1111 0011		
111 1111 0010		
111 1111 0001		
111 1111 0000		
⋮	⋮	⋮

*Fig. 1.8: Controle via ROM. O mapa de memória mostra como dados na ROM podem automatizar a geração dos sinais $\overline{EN}$, DI_R, LE_X, LE_Y, LE_A, OE_A e S. Cada bloco de 4 endereços forma uma FSM que executa uma operação sequencial. Por exemplo: (1) carregar X, (2) carregar Y, (3) multiplicar X*Y e salvar em (A ← X * Y), (4) sair com resultado.*

Cada computação que desejamos executar é uma sequência finita de passos que pode ser modelada como uma máquina de estados finitos (*FSM* [6]*– Finite State Machine*). O que chamamos de *passo* está relacionado com um *estado* na FSM. Dessa forma, o que gravamos na ROM é uma tabela que controla a execução de máquinas de estados distintas, cada uma com até 4 estados – 4 passos ou 4 operações distintas.

Na ROM, vamos gravar cada bit de dado controlando um dos sinais $\overline{EN}$, DIR, LE_X, LE_Y, LE_A, OE_A e o vetor $\boldsymbol{S}$. Por exemplo, no bit $\boldsymbol{D_0}$ podemos gravar o sinal $\overline{EN}$, e no bit $\boldsymbol{D_1}$ podemos gravar DIR, e assim por diante. Nem todos os sinais estão destacados na Fig. 1.8 no barramento saída de dados $\boldsymbol{D_0}$ até $\boldsymbol{D_{N-1}}$.

A coluna ‘operação’ no mapa de memória mostra o mnemônico de cada operação. Dessa forma, $\boldsymbol{A} \leftarrow \boldsymbol{X+Y}$ é um mnemônico que significa que os dados gravados na ROM colocaram a ULA em operação de soma, porque o

[6] Vamos usar FSM no livro, por ser uma sigla mais amplamente usada para designar máquinas de estados finitos, em vez de MEF.

vetor $\boldsymbol{S}$ = ($\boldsymbol{S_2}$=1, $\boldsymbol{S_1}$=0, $\boldsymbol{S_0}$=0), ao mesmo tempo LE_A=1 foi ativado para copiar o dado para o registrador $\boldsymbol{A}$, enquanto OE_A=0, foi gravado de forma a colocar a saída *3-state* do *latch* em alta impedância.

Note que os endereços mais significativos da ROM ($\boldsymbol{A_9}$ até $\boldsymbol{A_2}$) apontam para uma região da ROM. Como essas linhas de endereços têm 8 bits, podem assumir valores que apontam $2^N = 2^8 = 256$ regiões na memória. Quer dizer que podemos usar os endereços altos da ROM para apontar 256 regiões diferentes e executar até 256 FSMs distintas. Em computação, chamamos '*instrução'* cada FSM gravada na ROM (na *unidade de controle - UC*). No nosso exemplo podemos gravar 256 *instruções* na *UC*, e os endereços $\boldsymbol{A_1}$ e $\boldsymbol{A_0}$, que saem de um contador interno, controlam a mudança sequencial dos estados de cada instrução variando de 00 até 11.

1.2.1 – Sequência de Instruções

Automatizamos até o momento uma parte do nosso processador: externamente podemos alterar os endereços mais significativos da ROM ($\boldsymbol{A_9}$ até $\boldsymbol{A_2}$) e com isso dizemos para o processador executar uma sequência de operações (uma FSM) que podem transferir dados, executar uma operação na ULA, e liberar o resultado pela saída do *datapath*. Isso é executar uma *instrução* no processador. Porém, se quisermos fazer uma sequência de instruções, deveremos alterar manualmente os valores de $\boldsymbol{A_9}$ até $\boldsymbol{A_2}$ para cada instrução que quisermos executar. Como automatizar a execução de uma série de instruções? Uma solução seria colocar um contador para gerar diretamente os valores de $\boldsymbol{A_9}$ até $\boldsymbol{A_2}$ em sequência, mas esse tipo de solução sempre executaria a mesma sequência de instruções porque o contador sempre contaria de 00000000 até 11111111 e diria para o núcleo processador executar a instrução 0, depois a instrução 1, até 2^N. Não parece uma boa ideia! Queremos executar instruções de forma sequencial, mas a ordem das instruções do núcleo processador poderia ser aleatória.

Precisamos então mapear uma sequência, como a que um contador pode fornecer, em uma sequência de instruções aleatoriamente escolhidas dentro do repertório que gravamos no núcleo de processamento. Portanto, podemos colocar no lugar do contador direto outra memória externa, e essa memória seria endereçada por um *contador de instruções*. A Fig. 1.9 mostra como poderia ser essa ideia.

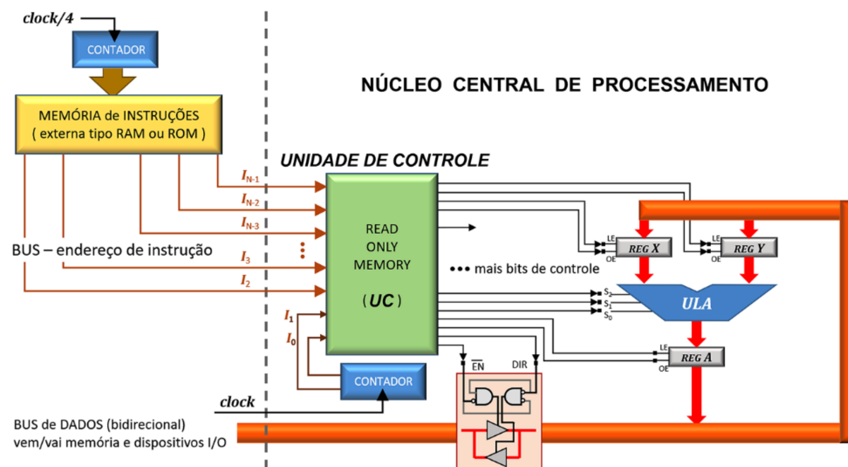


Fig. 1.9: Núcleo de Processamento. Externamente, usamos uma memória para armazenar uma sequência de instruções.

Uma memória externa pode ter gravada com qualquer sequência de instruções que desejarmos. Os endereços dessa memória são gerados por um *contador de instruções*. Esse contador aponta sempre qual das instruções está sendo executada em um dado momento. Lembre-se de que cada instruções tem quatro estados. Assim, com esse contador garantimos que as instruções gravadas na memória serão apresentadas ao núcleo central de processamento em sequência. A saída de dados dessa memória externa aponta qual das instruções (qual FSM) o núcleo central vai executar. Dessa forma, o que está gravado na memória externa pode ativar qualquer sequência de instruções – cada uma consumindo 4 passos (4 *clocks*) para ser executada. Quando necessário, trocamos o conteúdo da memória externa e assim executaremos outra sequência, outro '*programa*'.

Essa abordagem tem um problema porque sempre um programa será executado da instrução zero apontada pelo contador até a instrução 2^N que é o último valor que o contador poderá contar.

Esse tipo de solução sequencial não surgiu com a eletrônica e processadores eletrônicos. Antes, ainda no século 19, surgiram máquinas que liam cartões perfurados e a sequência dos cartões controlavam estados internos das

máquinas que instruíam a máquina a executar tarefas distintas. Exemplos são o Tear de Jacquard (1804), que introduziu o conceito de "programação" em automação industrial. Esses teares liam cartões em sequência e cada cartão instruía a máquina a levantar ou abaixar determinados fios, permitindo a criação de padrões complexos em tecidos. Outro exemplo é a Máquina de Hollerith (1890). Desenvolvida para processar dados do censo dos Estados Unidos da América de 1890, a máquina utilizava cartões perfurados para armazenar informações demográficas. Cada cartão representava um conjunto de dados que podia ser lido automaticamente pela máquina, acelerando imensamente o processo de contagem e análise. Essas máquinas fundou a base do que mais tarde se tornaria a IBM, e demonstrou o potencial dos cartões perfurados como um método eficaz de entrada, armazenamento e processamento de informações.

1.2.2 – Decisões e Inteligência

Apesar da solução anterior automatizar a execução de programas que podem ser facilmente alterados numa memória externa, a execução em sequência única não permite à máquina *tomar decisões* se continua executar o programa da próxima instrução ou se segue outra sequência mais adequada para solucionar o problema. Claro, tomar decisões é a questão chave nessa situação. Como uma máquina pode então decidir sobre algo? A base para toda e qualquer tomada de decisão é testar uma condição verdadeira ou falsa. A *inteligência* nas máquinas, mesmo a *Inteligência Artificial* (IA) que conhecemos hoje, depende de a máquina ter capacidade de testar condições. Um exemplo de teste de condição pode ser se a porta de um forno de micro-ondas está aberta (*Verdade*) ou fechada (*Falso*). Outro exemplo, um circuito comparador na ULA é capaz de comparar o valor do registrador $\boldsymbol{X}$ com o valor do registrador $\boldsymbol{Y}$, o que resulta em três possibilidades: $\boldsymbol{X} > \boldsymbol{Y}$, $\boldsymbol{X} = \boldsymbol{Y}$, ou $\boldsymbol{X} < \boldsymbol{Y}$. Então um conjunto de instruções pode mandar a ULA comparar $\boldsymbol{X}$ e $\boldsymbol{Y}$ e se o resultado for que $\boldsymbol{X} > \boldsymbol{Y}$ (*Verdade*) então a próxima instrução deve ser a que está numa dada posição da memória de programa, caso contrário (*Falso*), o programa vai seguir da instrução seguinte na memória de programa.

Da forma como construímos a solução acima, com um *contador de instrução* sempre apontando para um valor fixo como se estivesse lendo o próximo cartão, a máquina não tem capacidade de desviar o programa. Uma solução para executar esse tipo de automação é eliminar o contador externo de instrução e trazer o controle da busca da próxima instrução para dentro do próprio núcleo processador, como mostra a Fig. 1.10.

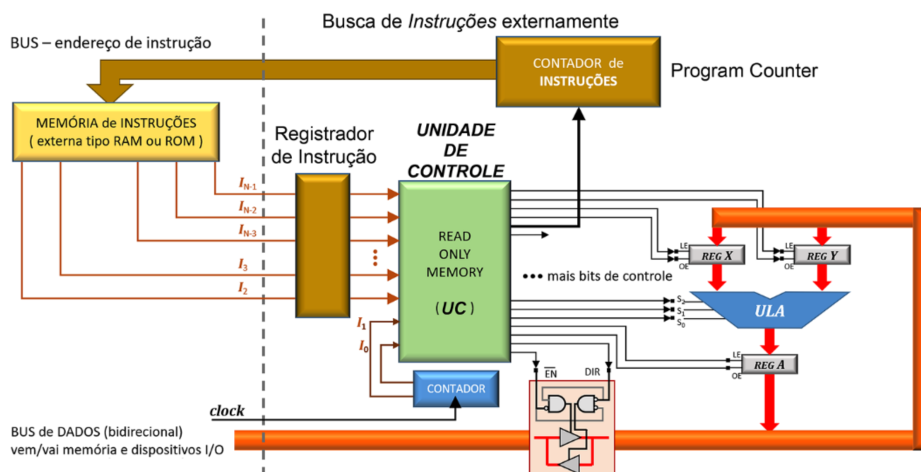


Fig. 1.10: Laço de Busca de Instrução. Externamente, uma memória armazena instruções. Um registrador interno ao núcleo (PC) determina em que endereço da memória está a próxima instrução, que é copiada para um registrador interno de instruções.

Como temos a ULA para realizar comparações e testes de condições dentro do núcleo processador, podemos usar a saída dessas operações para determinar em que posição da memória estará a próxima instrução. Dessa forma, nossa *UC* vai manter um registrador, chamado *Program Counter* (PC) para apontar para o endereço na memória externa onde está a próxima instrução. Adicionaremos um registrador (*de Instrução*) que copia a instrução em execução e mantém o estado dos endereços de $\boldsymbol{A_9}$ até $\boldsymbol{A_2}$ enquanto a instrução é executada. Note que, com essa modificação tomamos o controle da posição externa de memória onde buscaremos a próxima instrução a ser executada. Em nosso ciclo de operações deveremos então incluir um passo a mais, que é denominado busca de instrução (*instruction fetch*). Nossa máquina agora deve buscar uma instrução externamente, executá-la passo a passo como vimos acima, e calcular em que posição de memória vai buscar a próxima instrução, fazendo isso por meio do registrador contador de programas (PC).

1.2.3 – Obter Dados e Salvar Resultados

Automatizamos quase tudo, menos o fluxo dos dados. No tópico 1.16 vimos que é necessário fornecer dados externos à ULA para execução das instruções. Assim, é necessário controlar o fluxo de dados no *datapath* (*data-bus*) para que a ULA obtenha *operandos* sobre os quais opera. Da mesma forma, é necessário salvar dados em algum lugar ao término das instruções, caso contrário os resultados das operações se perdem.

A solução para gerir o fluxo de dados de forma automática é colocar uma outra memória externa ao núcleo de processamento. A forma como ligamos esses dados ao *datapath* gera ao menos duas arquiteturas diferente de computadores. Em uma arquitetura, apenas uma memória externa retém tanto dados quanto instruções. Essa arquitetura é chamada '*von Neumann*' em homenagem ao seu criador/inventor[7].

A outra arquitetura possui uma memória externa exclusiva para programas e outra memória e/ou periféricos que fornecem ou que retém dados depois de executada uma instrução. A arquitetura Harvard foi desenvolvida na década de 1930 no contexto do projeto do computador Harvard Mark I[8].

A Fig. 1.11 ilustra uma CPU do tipo Harvard. Podemos agora chamar o núcleo de processamento de CPU (*Central Processor Unity*), em português UCP (*unidade central de processamento*). Vamos usar a sigla CPU no livro por ser mais comumente usada no Brasil. Unidade central de processamento, como pode ser visto na ilustração, é um núcleo que tem capacidade de controlar também como os dados são transferidos para si e para fora desse circuito, quando termina a operação corrente.

Fig. 1.11: CPU Harvard. Externo à unidade central de processamento uma memória retém o programa (conjunto de instruções) enquanto outra memória e/ou dispositivos periféricos são usados para fornecer dados ou reter resultados das instruções.

Observem que um bloco de circuito lógico foi adicionado para controle do *endereçamento* e controle de *fluxo* de dados para a *memória* externa ou *periféricos*. Esse bloco lógico gera o endereço de qual posição de memória ou periférico (dispositivo de I/O – entrada/saída) estará ativo. Nesse esquema, a diferenciação entre dados da memória ou de periféricos pode ser feita pelo próprio endereçamento da memória/periférico. Em uma faixa de endereços a CPU pode escrever ou ler da memória, enquanto que e em outra faixa de endereços ela estará se comunicando com um dispositivo periférico.

A CPU determina o fluxo do dado por meio da ativação do sinal $RD/\overline{WR}$. Quando o dado vai para a memória ou periférico o sinal $RD/\overline{WR}$ = '0'; a CPU *grava* um dado na memória de dados ou periférico. Nesse caso, a CPU ajusta a direção do *transceptor de barramento* de dentro para fora dela. Por outro lado, quando $RD/\overline{WR}$ = '1', CPU vai *ler* um dado externo. Nesse caso, a memória ou o periférico é responsável por colocar os dados no barramento de dados; nessa hora, a CPU ajusta a direção do *transceptor de barramento* de fora para dentro dela.

[7] John von Neumann e equipe inventaram essa arquitetura em 1945, no projeto do EDVAC (*Electronic Discrete Variable Automatic Computer*), um dos primeiros computadores digitais da história

[8] Mark I, oficialmente conhecido como *Automatic Sequence Controlled Calculator*; uma máquina eletromecânica criada por Howard Aiken e sua equipe na Universidade de Harvard, com suporte da IBM, que foi finalizada em 1944.

Os processadores ARM Cortex-M são confeccionados usando a arquitetura Harvard. Claro, são muito mais complexos que esse exemplo didático. Contudo, não precisamos aprofundar em mais detalhes técnicos dos circuitos eletrônicos para compreender o básico de como funcionam os MCUs. A seguir, apresentamos uma descrição um pouco mais detalhada do núcleo de uma CPU ARM Cortex-M, com o objetivo de destacar o potencial desses processadores. Os conceitos de eletrônica digital discutidos acima são suficientes para entender a estrutura do Cortex-M sem sobrecarregar o leitor.

1.3 – Funcionamento da Arquitetura Harvard

Para melhor compreender o funcionamento de uma CPU, vamos executar um programa na arquitetura Harvard que construímos anteriormente. Tenha em mente que esta é uma CPU didática e que esse sistema processador é extremamente simples. Ainda assim, podemos ver como é o funcionamento dos processadores, que por mais sofisticados que sejam, as modernas CPUs seguem o mesmo modo de operação.

Partiremos de um programa escrito em linguagem C e vamos seguir o fluxo até tradução para código binário, que é o programa em código de máquina. Vamos escrever um programa bem básico, depois analisar o fluxo de um projeto de software para embarcados, e finalmente executar as instruções passo a passo.

1.3.1 – Do Programa ‘C’ ao Assembly

A tradução de um programa em C para linguagem de máquina (binário executável) em um ARM Cortex-M envolve diversas etapas dentro de um compilador, incluindo o *pré-processamento*, a *compilação* propriamente dita, a *montagem* e *linkagem*.

Descrevendo brevemente, o pré-processador do compilador remove comentários, expande macros definidas com `#define`, inclui o conteúdo de arquivos de cabeçalho (`#include`), resolve as outras diretivas de compilação condicional, como `#ifdef`, `#ifndef`, `#endif`, entre outras. Sai do pré-processamento um código-fonte intermediário (em C) sem macros e diretivas, pronto para compilação.

Em seguida, ocorre a compilação do código pré-processado. O compilador traduz o código em C para um código de montagem (*assembly*). O compilador pode realizar otimizações na sequência de processamento, embora às vezes tenhamos que tomar cuidado com o nível de otimização que o compilador aplica.

Outra obrigação do compilador é resolver estruturas como laços e condicionais. A saída do compilador é o *código em assembly*, que é legível por humanos, e é específico ao conjunto de instruções de cada CPU; por exemplo, o *assembly* específico para o ARM Cortex-M, ou o *assembly* para nosso processador didático, como mostrado adiante.

A tarefa seguinte é a montagem. No meio técnico, profissionais de software no Brasil preferem termos em inglês (*assembler*, *pointer*, *linker* e *linkagem*) por serem mais familiares e utilizados nas ferramentas e documentos de desenvolvimento de software. Não é muito usual encontrar *montador*, *ponteiro*, *vinculador* e *ligação*; embora em traduções mais formais as versões traduzidas possam aparecer.

O *assembler* (montador) converte o código em Assembly para código de máquina (instruções binárias que a CPU entende). A saída é um *arquivo de objeto*, extensões ‘.o’ ou ‘.obj’, contendo código em *linguagem de máquina* junto com informações auxiliares.

Ao final dessa tarefa o *assembler* gerou códigos de máquina; porém, os endereços das variáveis na memória (ponteiros) e das instruções iniciais das sub-rotinas e funções (ponteiros das funções) ainda não estão resolvidos; ou seja, não estão agregados aos arquivos objeto, permanecendo como referências a serem vinculadas pelo *linker* na etapa seguinte.

Em seguida, o processo de *linkagem* reúne um ou mais arquivos de objeto (e bibliotecas), resolve referências (como chamadas de função entre arquivos) e adiciona as bibliotecas necessárias.

O *linker* também organiza o código e dados no mapa de memória do MCU, insere seções de inicialização, e monta a tabela de ponteiros de vetores de interrupção, como veremos no momento oportuno.

A saída desse processo é um arquivo executável com extensões '.elf' (*Executable and Linkable Format*) ou binário '.bin' para gravação no MCU. Em resumo:

(1) Código em C → Pré-processador → C Expandido;
(2) C Expandido → Compilador → *Assembly* (e.g. *assembly* do ARM);
(3) *Assembly* ARM → *Assembler* → Objeto Binário;
(4) Objeto Binário + Bibliotecas → *Linker* → Executável.

Para fixar o conceito, vamos usar nosso processador didático para rodar um programa simples, escrito em C, para fixarmos as etapas acima.

```
/* Programa simples em linguagem 'C' */
#ifndef XVAL                  // diretiva: se ainda não foi definido XVAL
#define XVAL 3                // macro: define XVAL que será o valor inicial para X
#endif                        // fim da verificação de definição de XVAL
int main(void){
  int X = XVAL;               // declara a variável X e inicializa com XVAL
  int A = 0;                  // declara a variável que acumula somas
  for (int Y=1; Y<15; Y++) {  // laço FOR
    ++ X;                     // incrementa X
    A += X*Y;                 // acumula os valores de Y em A
  }
  while (1) {};               // laço infinito (dead loop)
  return 0;                   // retorno explícito para indicar sucesso
}
```

Na sequência, o pré-processador do compilador converte o arquivo em um código-fonte intermediário, um 'C' puro, pronto para compilar:

```
int main (void) {
  int X = 3;
  int A = 0;
  for (int Y=1; Y<15; Y++) {
    ++X;
    A += X*Y;
  }
  While (1) {};
  return 0;
}
```

Note que a variável *X* é inicializada com 3, porque o pré-compilador substitui o valor `XVAL` pelo valor definido na macro `#define XVAL 3`. Uma vez compilado, o resultado seria um hipotético *assembly* para nosso processador didático, como o código a seguir. Todavia, temos que considerar a inclusão de um novo registrador *R*, que não aparece na Fig. 1.11, a ser discutido adiante.

```
@ pseudo programa assembly: soma acumulativa de (X+1)*Y; Y varia de 1 a 14, X inicial = 3
main:                     @ label 'main', início do programa
        MOVS X, #3        @ mover X = 3 (inicializa o registrador X com 3)
        MOVS A, #0        @ mover A = 0 (inicializa o registrador A com 0)
        MOVS Y, #1        @ mover Y = 1 (início do laço)
loop:
        CMP Y, #15        @ compara Y com 15
        BGE fim           @ salta para label 'fim' se Y >= 15 (Branch if Great or Equal)

        ADD X, X, #1      @ incrementa X em 1 unidade (X += 1)
        MUL R, X, Y       @ multiplica X*Y e coloca resultado em 'R' (registrador novo)!
        ADD A, A, R       @ soma acumulativa A = A + R, ou (A += R)

        ADD Y, Y, #1      @ incrementa Y para controle do laço
        B   loop          @ Branch(salte) incondicionalmente para o label 'loop'
fim:
        B fim             @ loop infinito (fim do programa)
```

Este programa soma e acumula os valores de $(X+1) * Y$, com $X = 3$ no início, e Y variando de 1 até 14. Esse *pseudo-assembly* para nosso processador é bastante similar ao *assembly* ARM. À esquerda vemos marcadores