Tópico 4

Microprocessadores

Autores: José Raimundo de Oliveira e Wu Shin-Ting
DCA - FEEC - Unicamp
Agosto de 2019

Um microprocessador, usualmente denominado por **unidade central de processamento**, em inglês *central processing unity* (CPU), é considerado o cérebro de um sistema computacional, pois é a partir dele que se geram sinais de controle centrais para coordenar as operações do restante dos componentes do sistema, a fim de executar uma ou mais tarefas. Figura 4.1 ilustra um sistema computacional de mesa (*desktop*) com os seus componentes, incluindo a CPU no canto direito inferior.

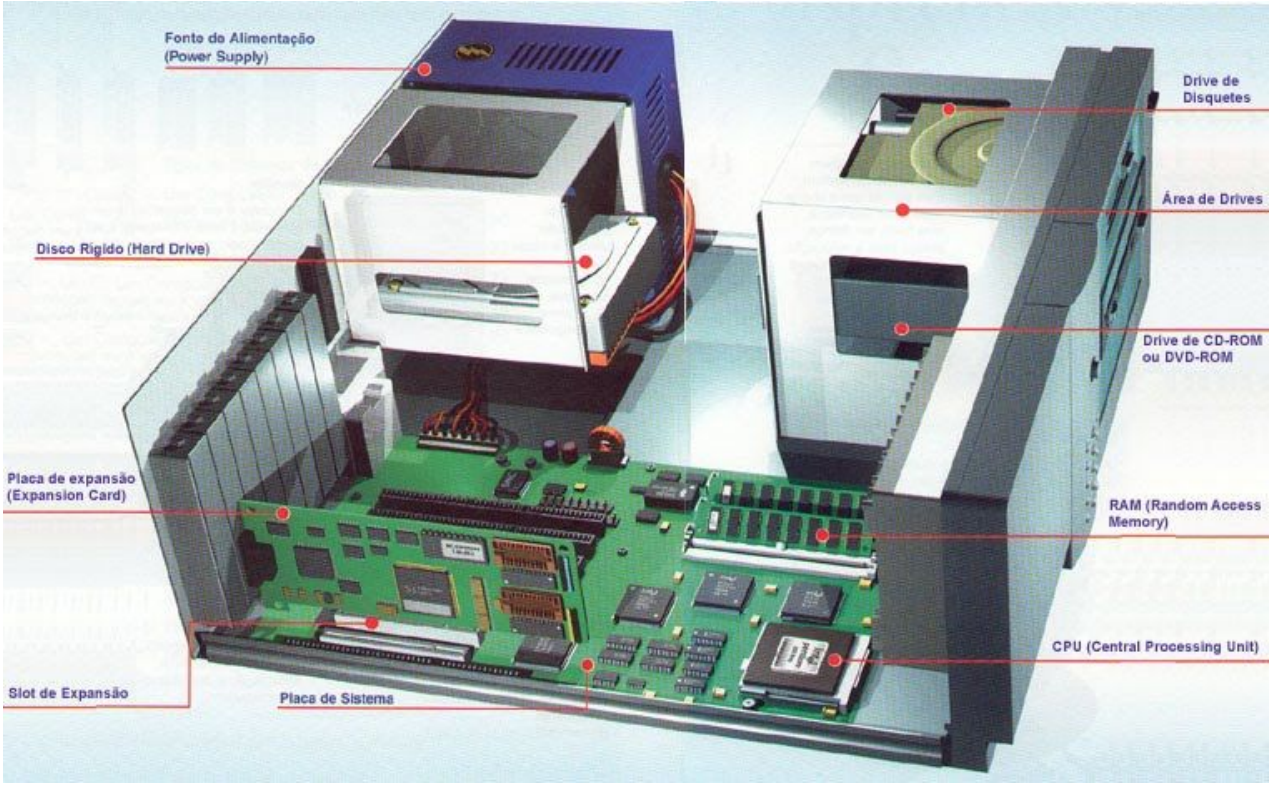


Figura 4.1: Um sistema computacional de mesa (*desktop*) e seus componentes (Fonte: [32]).

É importante frisar que, diferente do cérebro humano, um microprocessador não gera espontaneamente os sinais de controle. Ele simplesmente executa sequencialmente as instruções carregadas na memória, depois de transferi-las, uma por uma, para a CPU (unidade de controle e unidade lógico-aritmética). Na CPU, cada instrução é decodificada em **códigos de operação** e formas de acesso aos **operandos**. A sequência de sinais de controle é oriunda da unidade de controle da CPU, a partir dos códigos de operação decodificados, e é executada na frequência do sinal de **relógio**, em inglês *clock*. Portanto, quanto mais concisa e eficiente for a sequência de instruções programada pelo projetista, melhor será o desempenho do microprocessador.

O conjunto de instruções que um microprocessador consegue decodificar é denominado o seu **jogo de instrução**. Este jogo varia com a **arquitetura** do microprocessador, podendo impactar no tamanho e no tempo de execução de uma instrução. Como há uma grande diferença entre a velocidade de geração dos sinais de controle e a velocidade de busca dos operandos na memória, todas as CPUs modernas têm registradores internos para armazenar tanto os operandos necessários à execução dos sinais de controle quanto os resultados gerados com a execução destes sinais. Veremos neste capítulo que se pode ter para uma mesma arquitetura diferentes **organizações**, isto é diferentes números de registradores e interconexões com memória, resultando em diferentes modelos com preços e características de desempenho bem distintos.

Vale ressaltar que, sob o ponto de vista de projetos de sistemas embarcados, não é necessário projetar ou construir um microprocessador. Há no mercado uma variedade de famílias e modelos disponíveis como circuitos integrados completamente encapsulados, a um custo bem acessível. Porém, uma boa compreensão da organização e arquitetura de cada um deles permite uma melhor escolha de um microprocessador para a aplicação em foco, um melhor projeto dos circuitos de interface do selecionado com os outros módulos mostrados na Figura 4.1, e uma melhor programação deste cérebro eletrônico para execução das tarefas em questão. Certamente, como já comentamos no Capítulo 1, isso é uma atividade que demanda conceitos e habilidades de diversas áreas de conhecimento, como eletrônica, circuitos elétricos, processamento de sinais, controle, arquitetura e organização dos sistemas computacionais, programação e leitura de folhas técnicas dos componentes. Vamos começar com alguns conceitos relevantes sobre os microprocessadores.

4.1 Arquitetura

A arquitetura de um processador, ou mais precisamente a **arquitetura do jogo de instruções**, em inglês *instruction set architecture* ISA, inclui o jogo de instruções, quantidade de *bits* usados para representar as instruções, os operadores e os tipos de operandos, as técnicas de endereçamento dos operandos, e interface de entrada e saída.

Sob o ponto de vista de programação, uma CPU é constituída de uma **unidade de controle**, uma **unidade lógico-aritmética**, em inglês *arithmetic-logic unity* (ALU), registradores e conexões entre eles. Além dos **registradores de propósito geral**, há três registradores de propósito específico, **contador de programa**, em inglês *program counter* (PC), **ponteiro de pilha**, em inglês *stack pointer* (SP) e r**egistrador de palavra de estado do programa**, em inglês *program status word* (PSW). Em alguns processadores existe ainda um quarto registrador especial, denominado **acumulador**, em inglês *accumulator* (ACC), onde são armazenados temporariamente os resultados de uma operação lógico-aritmética como veremos na Seção 4.2.1.

O **contador de programa** é o registrador que guarda o endereço de memória da instrução corrente. Ele é automaticamente incrementado após o acesso de leitura de uma instrução. O **ponteiro de pilha** é o registrador que armazena o endereço do topo da pilha alocada para guardar temporariamente as variáveis locais de um segmento de instruções (uma função ou uma rotina) cujos valores não são relevantes para o processamento de instruções fora do escopo desse segmento. E o registrador de **palavra de estado do programa** é o registrador que armazena o estado da execução de um programa, como ***bits* de condição** do resultado de uma operação lógico-aritmética (**zero**, **positivo**, **negativo** e/ou **transbordamento**, em inglês *overflow*), **máscaras de interrupção, estado de privilégio** (usuário comum ou super-usuário) e o subconjunto de registradores em uso.

4.1.1 Jogo de Instrução

Essencialmente distinguem-se cinco tipos de instruções na **linguagem da máquina**:

- transferência de operandos entre a memória e os registradores da CPU,
- operações lógico-aritméticas,
- operações de controle de fluxo de execução,
- operações do co-processador, caso exista, e
- operações de controle do sistema.

Estas instruções são codificadas em códigos binários compostos de ‘0’ e ‘1’ e distinguíveis em três campos: o campo que contém os **códigos de operação**, em inglês *operation code* (opcode), o campo que especifica os endereços dos **operandos** sobre os quais o correspondente código de operação deve operar, e o campo de **modo de endereçamento**, ou seja, como os operandos devem ser acessados pelos “endereços” codificados.

Conforme a complexidade das instruções codificadas em códigos de operação, temos hoje em dia três grandes classes de arquitetura:

- **Computador com um conjunto complexo de instruções**, em inglês *Complex Instruction Set Computer* (**CISC**)**:** é uma arquitetura que dominou a década de 80. Ela suporta várias centenas de instruções, algumas simples e outras bem complexas, podendo demandar vários ciclos de relógio para serem executadas. Para poder codificar de forma única uma grande gama de instruções com diferentes modos de endereçamento, o tamanho dessas instruções é bastante variado. Os microprocessadores da família x86 e da família Motorola 68000 são exemplos de arquitetura CISC.
- **Computador com um conjunto restrito de instruções**, em inglês *Reduced Instruction Set Computer* (**RISC**)**:** ao contrário da CISC, a arquitetura RISC apresenta um conjunto pequeno de instruções com poucos modos de endereçamento. O tamanho das suas instruções lógico-aritméticas é o mesmo e o tempo de execução destas instruções é 1 ciclo de relógio. Isso decorre do fato de que os operandos destas instruções são sempre previamente carregados nos registradores. E os acessos de leitura e de escrita são os mesmos para todas as transferências dos dados entre a CPU e a memória. Essencialmente, o paradigma consiste em trabalhar diretamente com as instruções elementares de um microprocessador, a fim de tirar melhor proveito dos circuitos disponíveis. A arquitetura **ARM**, acrônimo de *Advanced RISC Machine*, de 32 *bits* tornou-se a arquitetura RISC mais popular. Observou-se, no entanto, que, desdobrar uma instrução complexa numa sequência de instruções simples, pode onerar o espaço da memória. Isso levou à proposta da variante **Thumb** (16 *bits*), cujas instruções tem a metade do tamanho das instruções ARM, a custo da restrição de algumas funções da ARM [17]. Para superar essas limitações foi então proposta a segunda variante **Thumb-2** que abrange as vantagens dos dois jogos de instruções, contendo instruções de 16 e 32 *bits*. Em 2010 iniciou-se na Universidade da Califórnia o projeto **RISC-V** que é uma versão livre da RISC focada em dispositivos modernos, como computação em nuvem, dispositivos móveis, sistemas embarcados e *internet* das coisas. São da arquitetura RISC os microprocessadores da família ARM desenvolvida pela empresa britânica *ARM Holdings*, os núcleos Krait e Kryo de Qualcomm incluídos nos microcontroladores Snapdragon, amplamente usados nos dispositivos móveis como *smartphones* e *tablets*.

- **Computação explícita de instruções paralelas**, em inglês *Explicitly Parallel Instruction Computing* (EPIC): como resposta ao limite de desempenho alcançado pela arquitetura RISC (1 instrução por ciclo de relógio), nasceu em torno de 1989 a ideia de codificar numa única **palavra de instrução muito longa**, em inglês *Very Long Instruction Word* (VLIW), múltiplas operações paralelizáveis mantendo a simplicidade da arquitetura RISC. Ela consiste essencialmente em aumentar o **paralelismo a nível de instrução**, em inglês *Instruction Level Parallelism* (ILP).

Pela filosofia de executar as operações lógico-aritméticas somente sobre os operandos que estão armazenados nos registradores, os modos de endereçamento da arquitetura RISC é muito mais uniforme e simples do que os modos de endereçamento da arquitetura CISC e o tamanho das suas instruções é menor do que os códigos de operação da arquitetura EPIC. Por outro lado, programas baseados em CISC e EPIC são usualmente mais densos em termos de quantidades de instruções necessárias para programar uma mesma tarefa. Entre CISC e EPIC, temos que na arquitetura CISC as instruções são mapeadas diretamente no circuito físico, enquanto na arquitetura EPIC as dependências entre as operações devem ser detectadas antes da execução para evitar os *hazards* (Seção 4.2.2).

Vale comentar aqui que a ISA é um dos diferentes níveis de abstração da organização de um microprocessador. Ele descreve as operações básicas que um microprocessador suporta. Essa descrição não reflete necessariamente a sequência de sinais físicos gerados, ou a **microarquitetura**, de um microprocessador. Diferentes microprocessadores podem suportar uma mesma ISA, como é o caso da x86 ISA da arquitetura CISC. Por exemplo, os microprocessadores AMD Athlon e a Intel Core 2 Duo tem a mesma arquitetura do jogo de instruções x86, porém as suas operações são implementadas por **microarquiteturas** com diferentes desempenhos e eficiências. Figura 4.2 ilustra a microarquitetura de i8080.

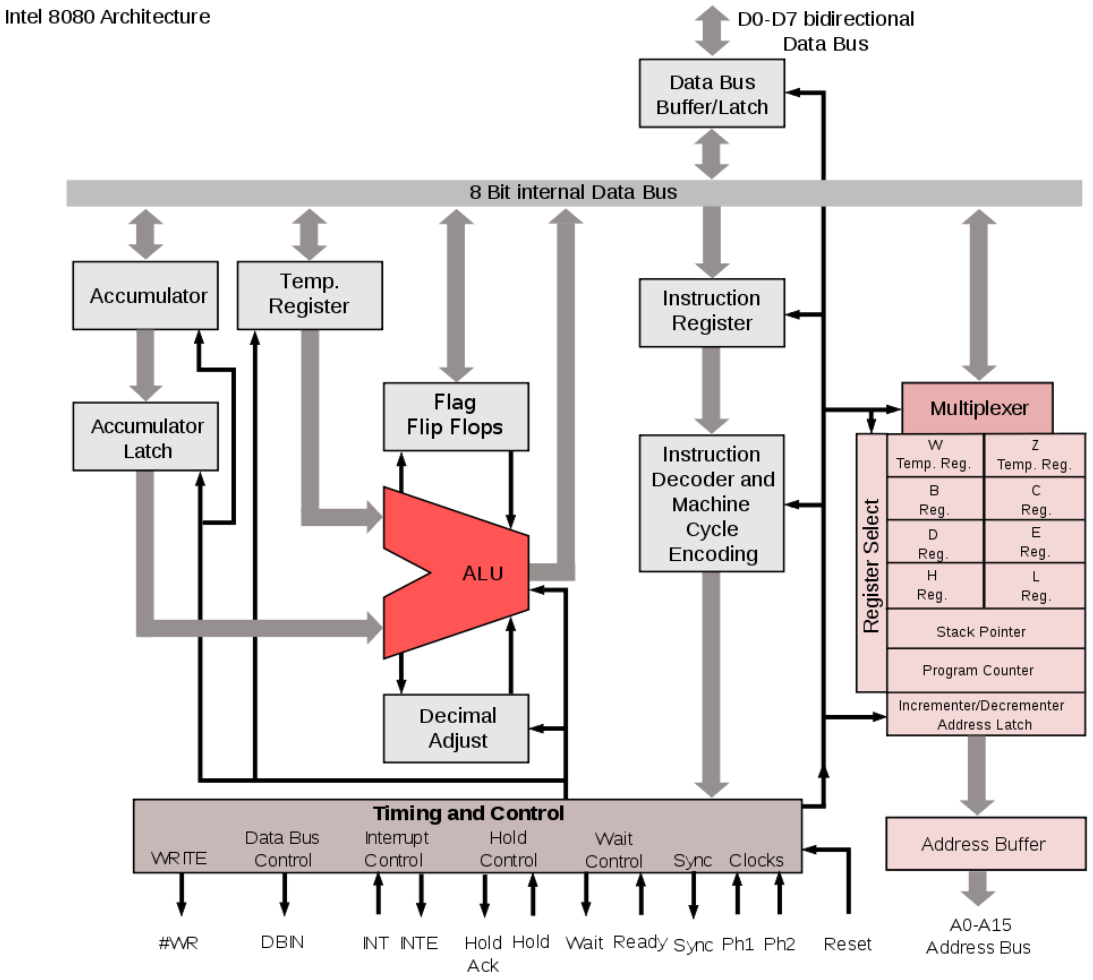


Figura 4.2: Microarquitetura de i8080 (Fonte: [34]).

Mesmo a nível de arquitetura, há diferentes níveis de abstração de um circuito físico (*hardware*) (Figura 4.3): o circuito físico propriamente dito (circuitos combinacionais e sequenciais), a sua abstração no nível de transferência entre registradores, em inglês *register transfer level* (RTL) [16], a nível de microarquitetura (sequência de sinais de controle capaz de realizar uma instrução) e a nível da ISA. É comum usar o termo **núcleo**, em inglês *core*, para denominar uma específica microarquitetura utilizada para implementar uma ISA. Por exemplo, o núcleo ARM (*Advanced RISC Machines*) Cortex A-5 é uma das diversas implementações da ISA ARMv7-A.

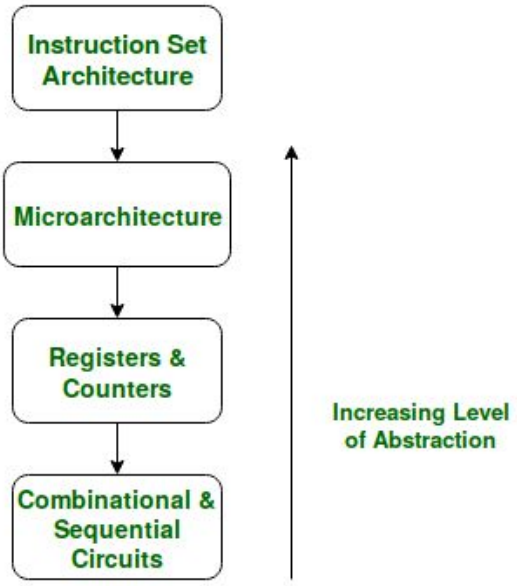


Figura 4.3: Níveis de abstração do circuito de uma CPU (Fonte: [33]).

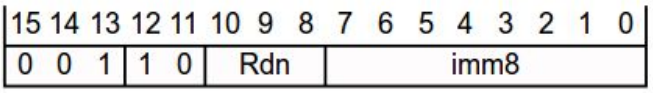
4.1.2 Códigos de Operação

Todas as instruções dos cinco grupos de instruções listados na seção 4.1.1 precisam ser codificadas em códigos binários de forma única. Portanto, é de se esperar que a quantidade de *bits* reservados para o campo de *opcode* depende da quantidade de instruções disponíveis no jogo de instrução. Os detalhes dos códigos binários de cada instrução de um jogo de instruções são encontradas nos manuais de referência de cada jogo. Por exemplo, em [15] encontramos a descrição do jogo de instruções x86, ou amd64, e em [7] todos os detalhes do jogo de instruções da arquitetura RISC ARMv6-M.

Figura 4.4 ilustra um campo de 5 *bits* reservados para o códigos de operação ADD (soma) entre um operando armazenado no registrador Rdn e o outro operando codificado no campo imm8 (*bits* 0-7) da instrução para um microprocessador da arquitetura ARM Thumb-2 [7].

Encoding T2 All versions of the Thumb instruction set.

```
ADDS <Rdn>,#<imm8>
```




```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Figura 4.4: Campo de código de operação numa instrução ARM Thumb-2 (Fonte: [7]).

4.1.3 Operandos

Cada microprocessador possui uma unidade de processamento natural, denominada a sua **palavra**, em inglês *word*. Há microprocessadores de 8, 16, 32 e 64 *bits*. Um microprocessador de 16 *bits* acessa, por exemplo, sempre em cada acesso 2 *bytes* em endereços pares. Um microprocessador de 32 *bits* acessa 4 *bytes* em endereços múltiplos de 4. Um microprocessador de 64 *bits* acessa 8 *bytes* em endereços múltiplos de 8. Quando a CPU faz acessos à memória com os endereços que sejam múltiplos do tamanho da sua palavra, dizemos que é um **acesso alinhado**, como ilustra a Figura 4.5. Quando os dados que queremos acessar não tem o tamanho alinhado com o tamanho da palavra do microprocessador, o microprocessador lê uma quantidade de palavras que abrange os dados desejados para depois extraí-los das palavras lidas, ou seja, o microprocessador preenche automaticamente os dados com campos adicionais para que fique com um tamanho que seja um múltiplo do tamanho da sua palavra (unidade natural).

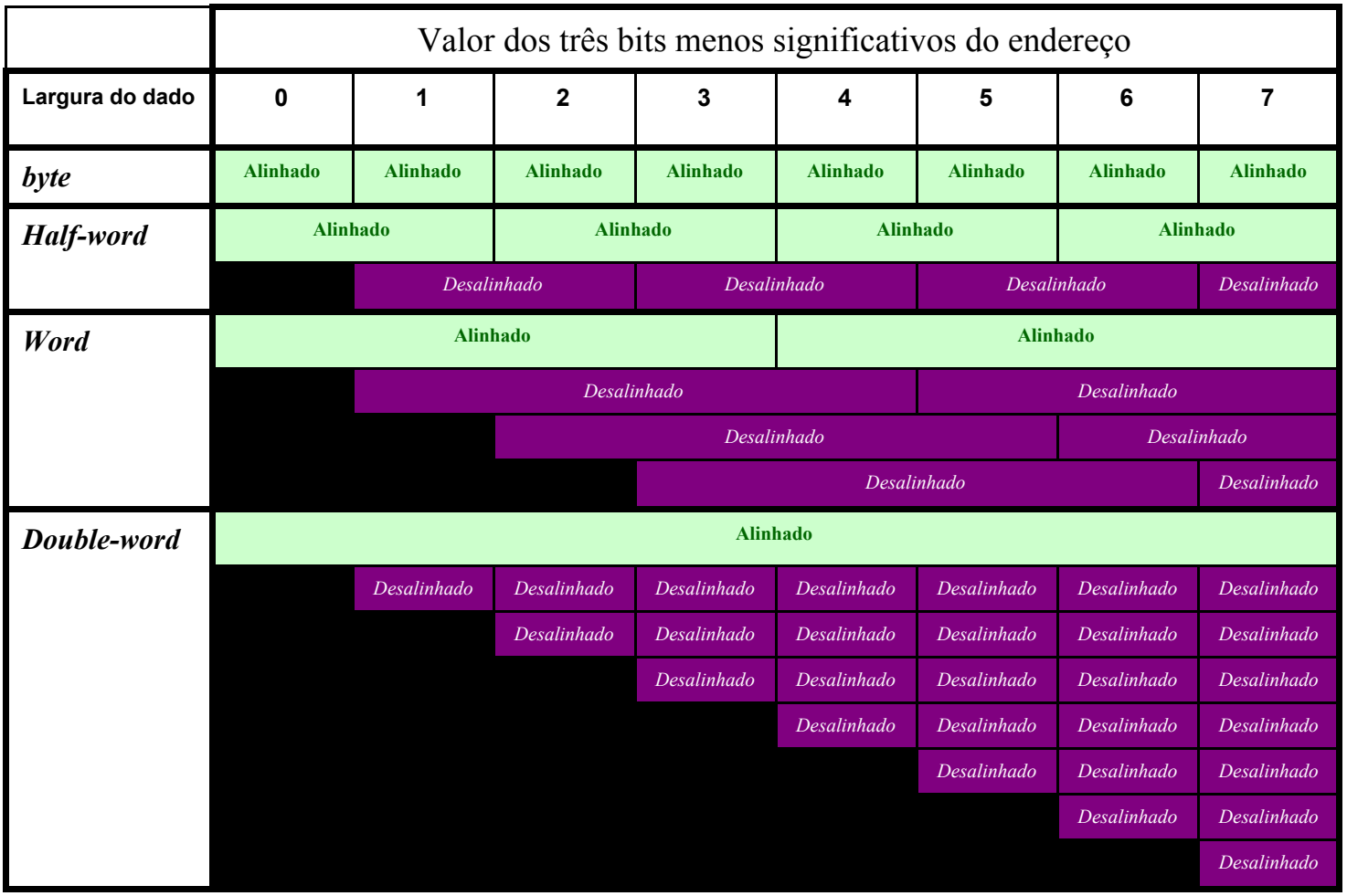


Figura 4.5: Alinhamento de endereço.

Veremos no Capítulo 5 que a menor unidade de endereçamento de memória (principal) é o *byte* (8 *bits*). Para armazenar uma instrução ou um operando que tem um tamanho maior de um *byte*, mais de um endereço de memória deve ser usado. Surge-se então a dúvida como ordená-los numa memória. Os dois métodos mais encontrados nos sistemas computacionais são (Figura 4.6):

- **ordenação menor primeiro**, em inglês *little-endian*[1]: na ordem crescente dos "pesos numéricos", do *byte* menos significativo para o mais, em endereços sucessivos crescentes da memória, e

[1] Danny Cohen introduziu os termos Little-Endian e Big-Endian para ordenamento de bytes em um artigo publicado pelo IETF (*Internet Engineering Task Force*) de 1980. No artigo "*On Holy Wars and a Plea for Peace*" ele faz um exame técnico e político das questões de ordenação de *bytes*, os nomes "*endian*" foram extraídos da sátira de Jonathan Swift de 1726, "As Viagens de Gulliver", na qual uma guerra civil eclode a partir da discussão se a parte maior (*big end*) ou a parte menor (*little end*) de um ovo cozido é o lado apropriado para rachar e abrir. Cohen fez uma analogia desta discussão àquela de qual o final que contém o *bit* mais significativo ou o *bit* menos significativo.

- **ordenação maior primeiro**, em inglês *big-endian*: na ordem decrescente dos “pesos numéricos”, do *byte* mais significativo para o menos, em endereços sucessivos crescentes da memória.

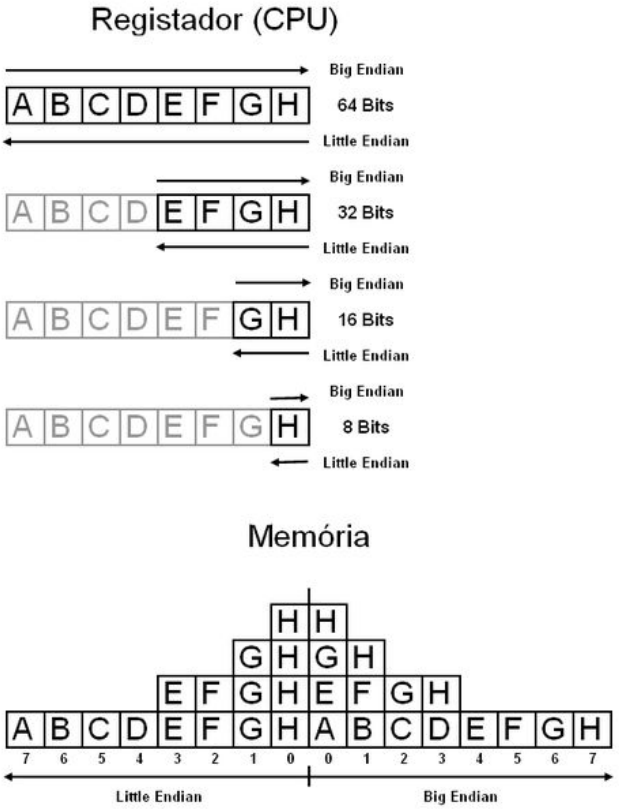


Figura 4.6: Ordenação de um dado na memória (Fonte: [35]).

4.1.4 Modos de endereçamento

Os modos de endereçamento variam muito entre os microprocessadores. Eles variam com a arquitetura destes microprocessadores. A arquitetura RISC apresenta muito menos modos de endereçamento do que a arquitetura CISC. Tanto que o campo de modo de endereçamento é suprido em alguns jogos de instruções dos microprocessadores da arquitetura RISC.

Nesta seção vamos ilustrar os diferentes modos de endereçamento do microprocessador 8086, uma máquina de 2 endereços (Seção 4.2.1), para mostrar a diversidade dos modos numa arquitetura CISC [10]. Considerando que as operações binárias (OP) sigam sempre a seguinte regra de execução:

operando-destino = operando-destino OP operando-fonte

segue-se a definição de cada modo:

- **implícito**: nenhum endereço está codificado na instrução.

- **endereçamento imediato**: o operando-fonte é um valor de 8 ou 16 *bits*.
- **por registrador**: os endereços são registradores da CPU.
- **indireto por registrador**: os **endereços efetivos** dos operandos estão contidos nos registradores codificados na instrução.
- **auto indexado (incremento/decremento)**: similar ao modo de endereçamento por registrador com o conteúdo do registrador automaticamente incrementado/decrementado de 1 após o acesso ao conteúdo do endereço contido no registrador.
- **endereçamento direto (absoluto)**: o endereço do operando está codificado na instrução.
- **endereçamento indireto**: o endereço do operando está contido no endereço ou no registrador codificado na instrução.
- **endereçamento indexado**: o endereço do operando-fonte/destino é a soma do conteúdo do registrador de índice da fonte/destino e o valor de 8 ou 16 *bits* codificado na instrução.
- **endereçamento indexado basal**: o endereço do operando-fonte/destino é a soma do conteúdo do registrador de índice da fonte/destino e o conteúdo do registrador de base.
- **endereçamento relativo a PC**: o endereço efetivo é a soma do valor codificado na instrução e o conteúdo do registrador PC.
- **endereçamento relativo a um registrador basal**: o endereço efetivo é a soma do valor codificado na instrução e o conteúdo do registrador de base.

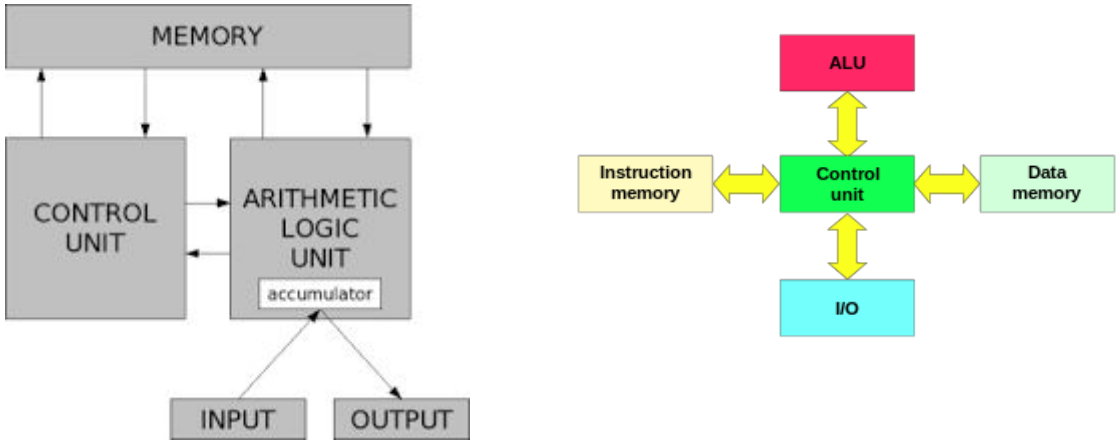
4.1.5 Linguagem *Assembly*

Programar um microprocessador usando os códigos binários, ou os equivalentes hexadecimais, é uma tarefa extremamente tediosa e propensa a erros. Como uma forma de melhorar a legibilidade dos programas, foram desenvolvidas **linguagens *assembly***, que são de fato uma representação simbólica direta dos códigos binários da ISA, acrescidas de algumas diretivas como alinhamento dos operandos aos endereços da memória [30]. Essas instruções são traduzidas, uma por uma, para as instruções em códigos binários por um “tradutor” denominado ***assembler***. Sendo cada linha de instrução uma ação que muda o estado do sistema computacional sob controle, dizemos que a programação em linguagem de *assembly* é uma **programação imperativa**.

Até hoje, a linguagem *assembly* é ainda muito utilizada no desenvolvimento de aplicativos que demandam, sob o ponto de vista de ocupação de memória e eficiência, um controle maior nas instruções a serem executadas. Um exemplo de aplicação da linguagem *assembly* é o ***firmware***, que consiste de um conjunto de instruções armazenadas em memórias não-voláteis para o controle de um circuito bem específico de um equipamento eletrônico.

4.2 Organização

Uma das características impactantes no tempo de execução de uma instrução é a quantidade de acessos à memória principal, cujo tempo é ainda na ordem de 100ns, enquanto a velocidade de processador já atingiu o patamar de Gigahertz (1ns). Diferentes organizações nos elementos de armazenamento (registradores e memória principal) e segmentação de instruções foram propostas com o intuito de minimizar essa discrepância no desempenho temporal. O espaço de endereços de um sistema de memória, que veremos no Capítulo 5, pode ser o mesmo para ambos os operandos e as instruções (arquitetura de von Neumann [36]) ou ser dividido em dois sub-espaços, um para operandos e outro para as instruções (arquitetura de Harvard [37]). Figura 4.7 esquematiza a organização das duas arquiteturas.



(a) Arquitetura de von Neumann (Fonte: [38])

(b) Arquitetura de Harvard (Fonte: [39])

Figura 4.7: Arquiteturas com diferentes abstrações do sistema de memória.

4.2.1 Quantidade de Endereços em Instruções

A quantidade de endereços codificados numa instrução impacta diretamente na quantidade de posições de memória a serem buscadas por instrução, decodificadas e executadas para executar uma operação lógico-aritmética, como mostram as seguintes organizações da CPU:

- **Organização por Pilha**, ou instruções de zero endereço: todas as expressões lógico-aritméticas em notação infixa são convertidas para a notação Polonesa inversa (pós-fixada) antes da sua execução. Na Figura 4.8, a expressão infixa A*B é reescrita na notação pós-fixada AB*, de forma que se empilha os dois operandos numa pilha (Seção 14.6.1) para então

desempilhar os dois operandos, aplicar a multiplicação e re-empilhar o resultado.

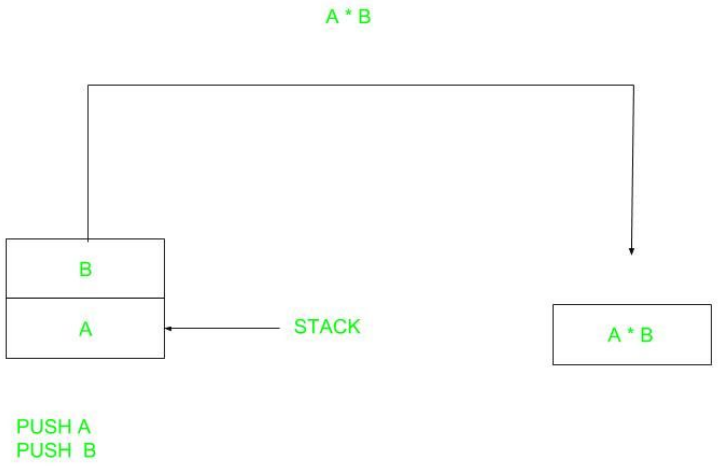


Figura 4.8: Organização por pilha (Fonte: [40]).

- **Organização por Acumulador**, ou instrução de um endereço (Figura 4.9): o registrador ACC é usado para armazenar o segundo operando e o resultado de uma operação lógico-aritmética. O operando é sempre transferido da memória para o registrador ACC. Quando se trata de um operador binário, é aplicada a operação sobre o dado transferido e o conteúdo do ACC e o resultado armazenado no ACC.

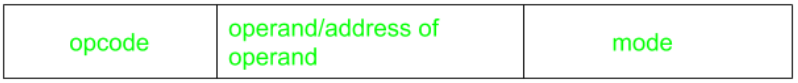


Figura 4.9: Organização por acumulador (Fonte: [40]).

- **Registrador-Memória**, ou instrução de dois endereços (Figura 4.10): ambos os operandos de uma operação binária tem seus endereços codificados na instrução, um do operando-fonte e outro do operando-destino. Usualmente o resultado da operação sobrescreve o conteúdo do endereço do operando-destino.

Figura 4.10: Organização registrador-memória (Fonte: [40]).

- **Register-Register/*Load-Store***: é também uma instrução de dois endereços. Nesta arquitetura a CPU usa somente os registradores para realizar as suas operações lógico-aritméticas em um ciclo de relógio. Caso não se encontram os operandos nos registradores, instruções dedicadas para transferência **LOAD** e **STORE** são usadas para transferir os dados entre os registradores da CPU e a memória principal (Capítulo 4). Por isso, são também conhecidos como **microprocessadores sem estágios intertravados de *pipeline***, em inglês *Microprocessor without Interlocked Pipeline Stages* (MIPS).

4.2.2 Segmentação de instruções

Para acelerar a execução de uma instrução, a técnica de **segmentação de instruções**, em inglês *pipeline*, pode ser aplicada. Esta técnica consiste essencialmente em dividir uma instrução numa sequência de microinstruções e paralelizar a execução destas (micro)instruções. Isso reduz o tempo de ociosidade dos recursos computacionais. Ela tem sido inicialmente introduzida na arquitetura RISC dos processadores MIPS, mas hoje é aplicada em todas as arquiteturas.

No caso de um microprocessador de 32 *bits* com instruções codificadas em 16 *bits* e 2 ciclos de relógio por execução de uma instrução (1 ciclo para busca e pré-decodificação, e outro para pós-decodificação e execução) [1], podemos segmentar uma instrução em dois segmentos na execução de 4 instruções como mostra na Figura 4.11. Neste caso, um ciclo de instrução passa de 2 ciclos de relógio para 1 ciclo de relógio.

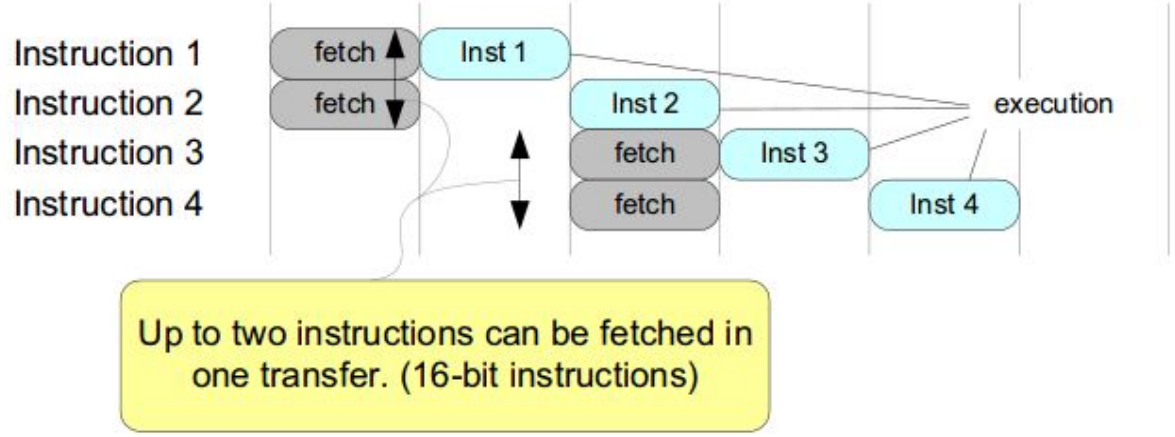


Figura 4.11: *Pipeline* de 2 estágios. Fonte: [1]

Com microprocessadores da arquitetura EPIC é possível fazer a busca e a execução de mais de uma instrução ao mesmo tempo. Vamos ilustrar o procedimento aplicado na execução de uma instrução lógico-aritmética (Figura 4.12) em 5 ciclos de relógio: (1) IF: **ciclo de busca** de instrução, em inglês *fetch*, (2) ID: **ciclo de decodificação** de instrução, em inglês *decode*, (3) EX: ciclo de execução instrução ou de obtenção do endereço efetivo, (4) MEM: ciclo de acesso à memória, e (5) WB: ciclo de *write-back*.

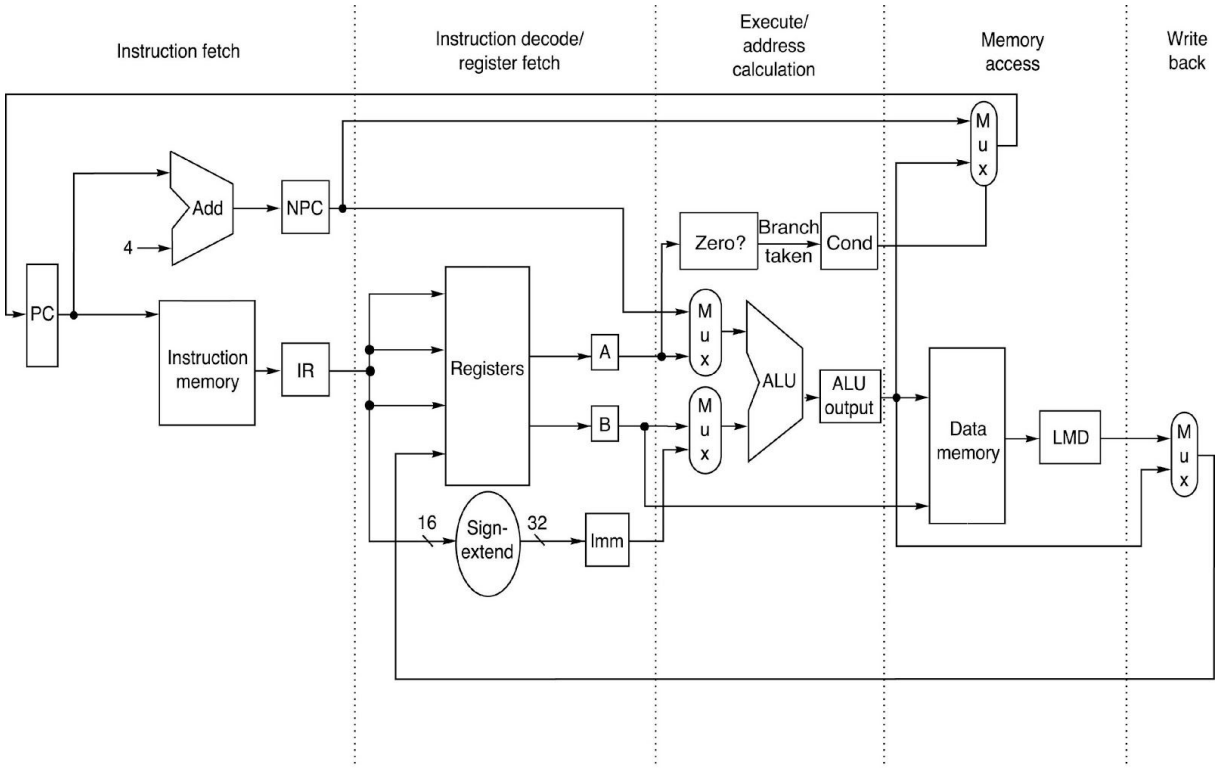


Figura 4.12: Arquitetura MIPS sem *pipeline*.

Considerando ainda que a arquitetura seja de Harvard, os acessos às instruções IM e aos dados DM, destacados na Figura 4.13, são totalmente independentes. Portanto, alguns recursos podem ser utilizados em paralelo. Figura 4.14 mostra a sobreposição da execução de 5 instruções diferentes, deslocadas de um ciclo de relógio. Note que, ao invés de gastarmos 5x5=25 ciclos de relógio, levaremos em média 5 ciclos de relógio para executarmos as 5 instruções. Isso aumenta a **vazão**, em inglês *throughput*, do sistema. O tempo de um **ciclo de instrução com *pipeline*** passará a ser

$$ciclo\ de\ instrução\ com\ pipeline\ = \frac{ciclo\ de\ instrução\ sem\ pipeline}{número\ de\ segmentos\ por\ instrução}$$

Para o caso ilustrado na Figura 4.13 especificamente, o tempo de um ciclo de instrução reduziu de 5 ciclos para 1 ciclo.

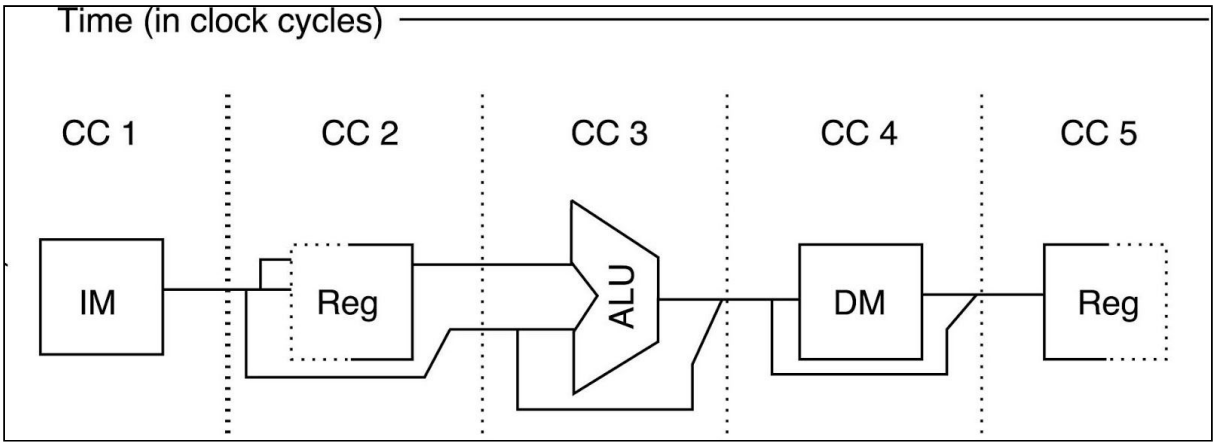


Figura 4.13: Segmentação de uma instrução lógico-aritmética.

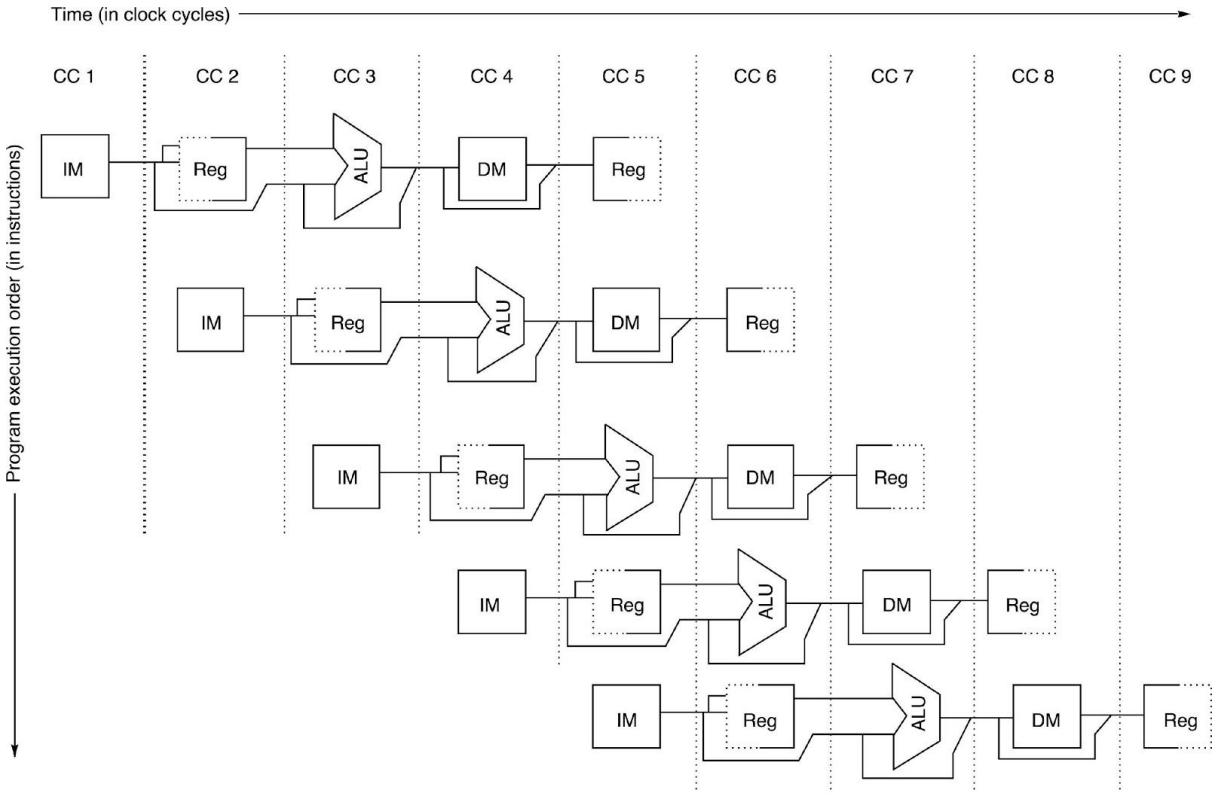


Figura 4.14: Paralelização por *pipeline* de 5 estágios.

Veja uma plausível organização para um fluxo de controle de uma instrução lógico-aritmética numa arquitetura MIPS sem segmentação de instruções (Figura 4.12) e numa arquitetura MIPS com segmentação de instruções (Figura 4.15).

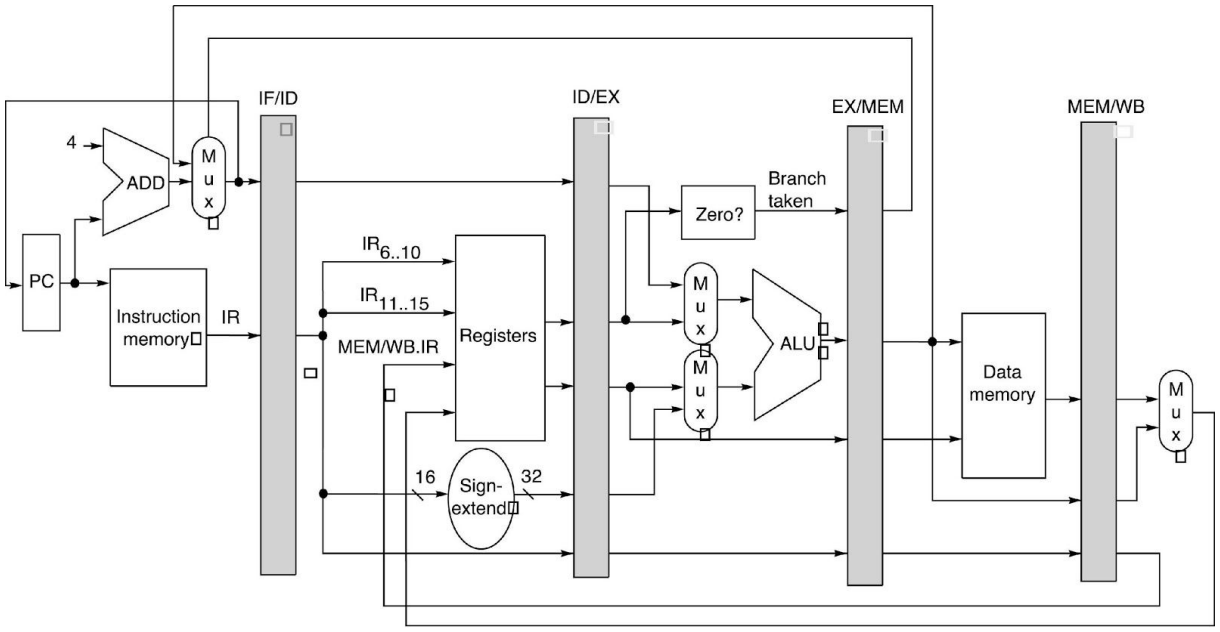


Figura 4.15: Arquitetura MIPS com *pipeline*.

A busca de uma instrução em paralelo com a execução de uma outra instrução pode causar alguns **perigos**, em inglês *hazards*, classificados em:

- **estruturais**: surgem de conflitos de recursos quando o circuito não pode suportar todas as combinações possíveis de instruções simultaneamente em execução. Nos processadores modernos, estes *hazards* estruturais ocorrem principalmente em unidades funcionais de propósito específico que são menos usadas, como divisão em ponto flutuante ou outras instruções complexas de longa duração. Eles não são o principal fator que afeta o desempenho. Assume-se, regra geral, que os programadores e os desenvolvedores de compiladores (Seção 4.7.2) estejam cientes do menor *throughput* dessas instruções.
- **de dados**: surgem quando uma instrução depende dos resultados de uma instrução de uma forma que é exposta pela sobreposição de instruções no fluxo.
- **de controle:** surgem da segmentação de instruções de desvio e de outras instruções que mudam o conteúdo do registrador PC.

Interessados em projetos de sistemas embarcados, não usaremos a arquitetura EPIC ao longo deste curso. E a ocorrência dos *hazards* de segmentação de instrução é menos frequente na arquitetura de von Neumann CISC e RISC, pois os dados e as instruções são acessadas de forma sequencial, compartilhando o mesmo conjunto de conexões como mostra a Figura 4.7(a). Voltaremos a discutir

este problema no Capítulo 10, quando veremos com mais detalhes a arquitetura de Harvard (Figura 4.7(b)) para a qual muitos microcontroladores estão migrando.

4.3 Exemplos de Microprocessadores

O consumo de energia, as altas temperaturas e o espaço demandado pela placa-mãe para alojar estes microprocessadores inviabilizariam economicamente a implementação de um projeto simples como o central de um sistema de alarmes ou as funções de um dispositivo móvel (*smartphones* e *tablets*). Mostraremos no Capítulo 10 que a alternativa preferida pelos projetistas de sistemas embarcados são os **microcontroladores**, em que os microprocessadores são integrados como um núcleo numa mesma pastilha junto com outros componentes fundamentais de um sistema computacional. Entretanto, as técnicas empregadas para desenvolver os microcontroladores são, de fato, uma evolução das técnicas utilizadas no desenvolvimento das unidades encapsuladas individualmente. Um entendimento melhor das limitações das técnicas e soluções apresentadas proporciona uma compreensão melhor de uma série de práticas aplicadas nos projetos modernos.

As folhas técnicas dos microprocessadores contém as **características elétricas, temporais e funcionais** dos sinais da sua interface, as caracterìsticas mecânicas inclusive a pinagem, a sua **organização** (componentes integrados e as conexões entre estes componentes) e a sua **arquitetura de jogo de instruções**. Adicionalmente, são incluídas informações sobre circuitos básicos de alimentação, de relógio e de conexão com memória e com os periféricos para formar um sistema computacional. Ressaltamos aqui que os pinos de saída dos microprocessadores, conectados aos barramentos (Capítulo 10), são do tipo **tri-state**, em inglês *three-state* (Capítulo 2). Assim eles podem compartilhar com outros dispositivos um mesmo conjunto de linhas, ficando no estado de alta impedância (Z) quando não fazem uso das linhas.

Apesar da preferência pelos microcontroladores, é importante que os projetistas de sistemas embarcados estejam sintonizados com todas as tecnologias dos microprocessadores que podem muito bem serem integrados como núcleos nos futuros microcontroladores. Vamos dar uma rápida passada por algumas outras famílias de microprocessadores.

Os microprocessadores que suportam paralelismo a nível de instruções (ILP) são chamados de **microprocessadores superescalares**. Um exemplo é o processador IBM *RISC System*/6000 [14]. Esses microprocessadores exploram paralelismo de maneira a capacitar a execução de mais de uma instrução por ciclo de relógio. Ele consegue decodificar múltiplas instruções de uma vez e a sincronização do

paralelismo para assegurar um fluxo ininterrupto e sem *hazards* ocorre, geralmente, no tempo de compilação (Seção 4.8.2).

Os microprocessadores que apresentamos até agora tem seus circuitos internos definidos pelos seus fabricantes. Usualmente, eles são de propósito geral, customizáveis pelos programas (instruções) que são carregados neles. Sendo de propósito geral, dificilmente eles contém circuitos dedicados como circuito de processamento de áudio ou de vídeo. Já imaginaram um processador baseado em tecnologia de **arranjo de portas programáveis em campo**, em inglês *Field Programmable Gate Array* (FPGA), cujo circuito é um grande arranjo de blocos lógicos configuráveis? Já imaginaram usar, por exemplo, a linguagem VHDL [11] para transformar o circuito de uma pastilha de FPGA numa "CPU" dedicada? Estas CPUs implementadas com circuitos que contém lógicas programáveis são conhecidas por **microprocessadores "flexíveis"**, em inglês *soft microprocessor*. Exemplos de processadores *soft* são MicroBlaze da Xilink [28] e Nios II da Altera [29]. Em [12] encontra-se uma breve análise comparativa entre uma CPU e um *soft microprocessador* implementado com FPGA.

Como a rápida evolução de tecnologias de processamento de sinais digitais de áudio, vídeo e de voz, existe no mercado uma linha de microprocessadores especializados para processá-los. São conhecidos como **processadores de sinais digitais**, em inglês *digital signal processing* (DSP). Estes processadores são providos de circuitos dedicados para realizar as funções que eram tradicionalmente implementadas em circuitos analógicos até os anos 70, tanto no domínio temporal/espacial quanto no domínio espectral/*wavelet*:

- **filtragem**
- **convolução**, ou mistura
- **correlação**, ou comparação
- **retificação**
- **amplificação**
- **aplicação das transformadas**

DSP é, portanto, um processador de sinais em tempo-real amostrados em intervalos regulares e convertidos para digital, como mostra Figura 4.16. Note que o sinal analógico é filtrado antes de ser amostrado e quantizado no conversor A/D (Capítulo 8) para evitar o efeito de **alias**, em inglês *aliasing*, que consiste a interferência dos sinais de altas frequências em sinais de baixas frequências. O sinal digital resultante pode voltar para o formato analógico, passando por um conversor D/A (Capítulo 8) e por um filtro de suavização. Exemplos de DPS são os processadores das famílias DSP56000 da Motorola [23] e TMS320 da Texas Instruments [24].

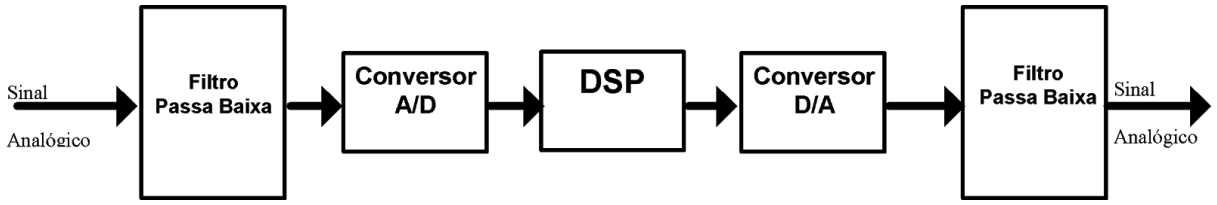


Figura 4.16: Microprocessador dedicado: DSP.

4.4 Coprocessadores

Vimos na lista dos grupos de instruções um grupo de instruções de acesso às instruções de coprocessadores. Um coprocessador é um circuito de processamento de funções bem específicas, usualmente complementares às funções do processador ao qual ele está conectado. Sendo um módulo complementar, é necessário que os seus sinais sejam compatíveis com o processador que ele complementa elétrica, temporal e funcionalmente. Figura 4.17 mostra o diagrama de blocos de um sistema constituído por um processador e um co-processador. Coprocessadores mais conhecidos são as unidades **de ponto flutuante aritmético**, em inglês *floating-point unit* (FPU)**,** e **unidade de processamento gráfico**, em inglês *graphics processing unit* (GPU).

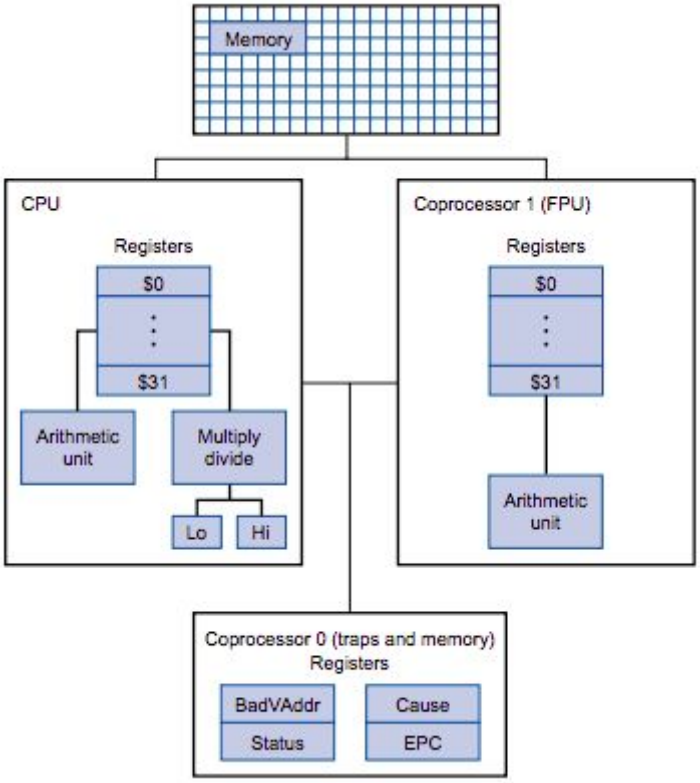


Figura 4.17: Processador e coprocessador (Fonte: [41]).

Os coprocessadores FPU já se encontram integrados na maioria dos microprocessadores, como Intel 80486 [13]. Exemplos mais famosos de

coprocessadores encapsulados como unidades são as GPUs, da arquitetura SIMD (*Single Instruction and Multiple Data*), desenvolvidas pela Nvidia [25] e pela AMD [26]. Esses coprocessadores foram originalmente destinados para jogos digitais em sistemas computacionais de mesa (*desktop*). Há hoje várias GPUs para dispositivos móveis como Adreno da Qualcomm (originalmente desenvolvidas pela AMD), Mali da *ARM Holding*, Tegra da Nvidia, PowerVR da *Imagination Technologies* [27]. A grande capacidade paralela de processamento numérico demonstrada pelas GPUs fez elas ganharem popularidade em computações científicas também. São hoje intensamente aplicadas na execução de uma variedade de algoritmos numéricos.

4.5 Controladores de Interrupção

Dois conceitos importantes relacionados com a melhoria do uso da CPU é o **processo** e ***thread***. Eles permitem que uma única CPU seja vista como múltiplas CPUs virtuais, atendendo pseudo-**paralela**mente várias tarefas. Um **processo** é a execução de um programa com uma CPU e um espaço de endereçamento virtual separado. Um ***thread*** é um fluxo de controle num mesmo espaço de endereçamento. Usualmente, é associado um *thread* a um processo. Porém, quando há múltiplas entradas e saídas (**um processo limitado pela E/S**, em inglês *I/O bound*), associar múltiplos *threads* a um processo pode aumentar o desempenho do sistema. Figura 4.18 ilustra a diferença entre multiprogramação e *multithreading*.

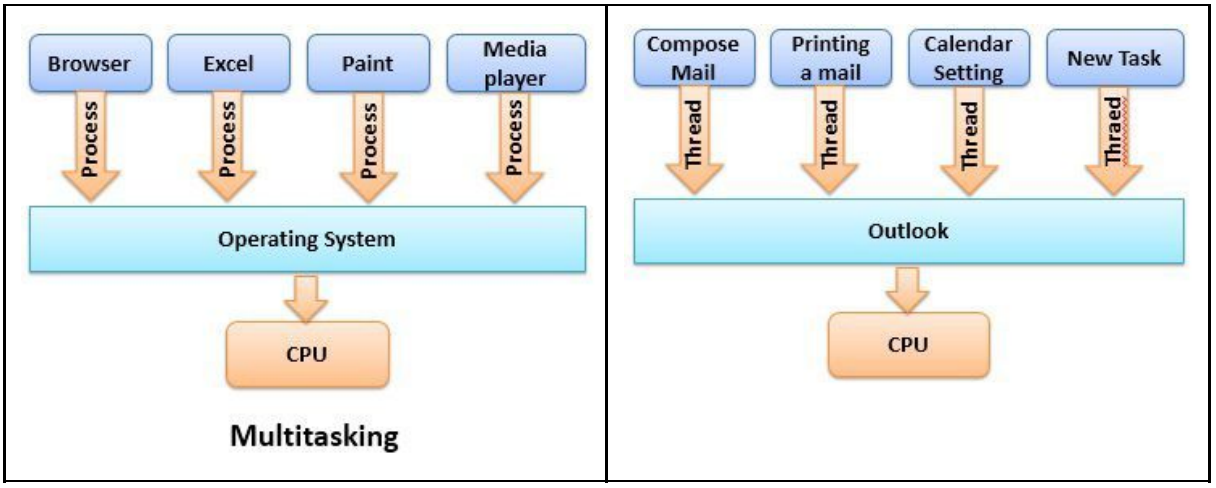


Figura 4.18: Multiprogramação e *multithreading* (Fonte: [42]).

Para dar aos usuários a ilusão de todos os processos/*threads* sejam atendidos simultaneamente, a CPU se chaveia entre eles num intervalo de tempo pré-definido ou sob demanda, embora ela continue executando somente um programa a cada instante. Vale observar que, diferente da execução de um mesmo **fluxo de controle sequencial** de um único processo, chaveamentos constantes entre os programas numa **multiprogramação** ou n**uma execução multithread** podem resultar num

fluxo de controle “costurado” por trechos de instruções de programas diferentes. A predizibilidade e a reprodutibilidade dos resultados são, portanto, muito mais difíceis.

No caso de múltiplos dispositivos de entrada e saída (teclado, *mouse*, disco rígido), o desempenho do sistema aumenta quando se conecta à CPU um **controlador programável de interrupção**, em inglês *programmable interrupt controller* (PIC). Este controlador é responsável pelo processamento dos eventos de interrupção gerados pelos dispositivos de entrada e saída. Ele se encarrega da identificação do endereço da rotina de serviço, em inglês *event handler*, associada ao evento e da atualização do conteúdo do PC com este endereço. O endereço da rotina é pré-configurado na tabela de **vetores de interrupções**. Com isso, **as chamadas das rotinas de serviço não são mais pré-programadas pelo projetista. Elas acontecem de forma assíncrona, orientadas aos eventos pré-definidos, e são gerenciadas por um controlador de interrupção**. Cabe à CPU, através da execução da rotina de serviço programada pelos projetista, salvar uma cópia do conteúdos dos registradores na pilha do sistema antes de iniciar a execução das instruções correspondentes ao tratamento da interrupção, e restaurar o conteúdo do PC.

Figura 4.19 ilustra o diagrama de blocos do controlador de interrupção PIC 8259 da Intel. Este controlador foi introduzido em 1976 como parte da família dos processadores MCS-85. Operou também com o processador da Intel 8088 nos originais computadores pessoais da IBM. Evoluiu-se para **controladores programáveis avançados de interrupção**, em inglês *Advanced Programmable Interrupt Controller* (APIC), viabilizando a construção de sistemas de multiprocessamento, em que dois ou mais processadores compartilham o uso de uma memória principal. Além disso, ao invés de serem disponíveis em unidades encapsuladas separadas, os controladores avançados se encontram integrados nos ***chipsets*** modernos (Capítulo 4). Embora não seja mais usado, alguns *chipsets*, como os da placa-mãe x86, ainda suportam conexões com o CI 8259A.

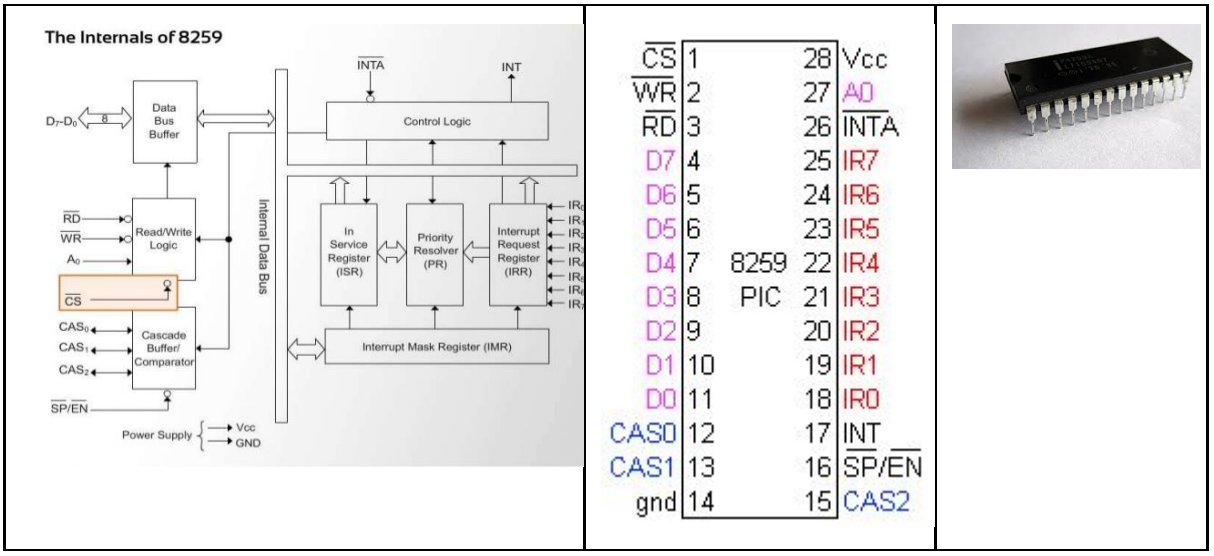


Figura 4.19: Intel 8259: controlador de interrupções programáveis.

Mesmo não estando os programas executando de fato em paralelo, o fato deles compartilharem o mesmo recurso físico, como o espaço de endereçamento, propicia a **concorrência** no acesso aos recursos. A falta de uma coordenação apropriada para tal concorrência pode levar à **condição de corrida**, em inglês *race condition*, **impasse**, em inglês *deadlock*, ou **inanição de recursos**, em inglês *resource starvation*. A **condição de corrida** surge quando dois ou mais segmentos de códigos em execução compartilham um mesmo recurso. O **impasse** acontece quando um programa fica bloqueado, aguardando por um evento ou um recurso que só pode ser liberado por um outro programa que está também bloqueado. E a i**nanição de recursos** ocorre quando o progresso de execução de um programa seja impedido pela execução de outros ocupando os recursos que ele precisa. Diferentes técnicas de sincronização foram propostas para proteger regiões críticas, isto é segmentos de instruções do programa em que possa ocorrer acessos concorrentes, como veremos no Capítulo 13.

4.6 Circuitos de Contagem e Temporização

A melhoria do processamento de interrupções assíncronas pela CPU propiciou o desenvolvimento de módulos encapsulados de circuitos dedicados para contagem de eventos e fazer cronometragem específica demandada por um dispositivo, sem ocupar a CPU. Um desses circuitos dedicados é o temporizador de intervalos programáveis, em inglês *Programmable Interval Timer* (PIT) 8253 da Intel. Figura 4.20 mostra a arquitetura, a pinagem e um encapsulamento deste temporizador.

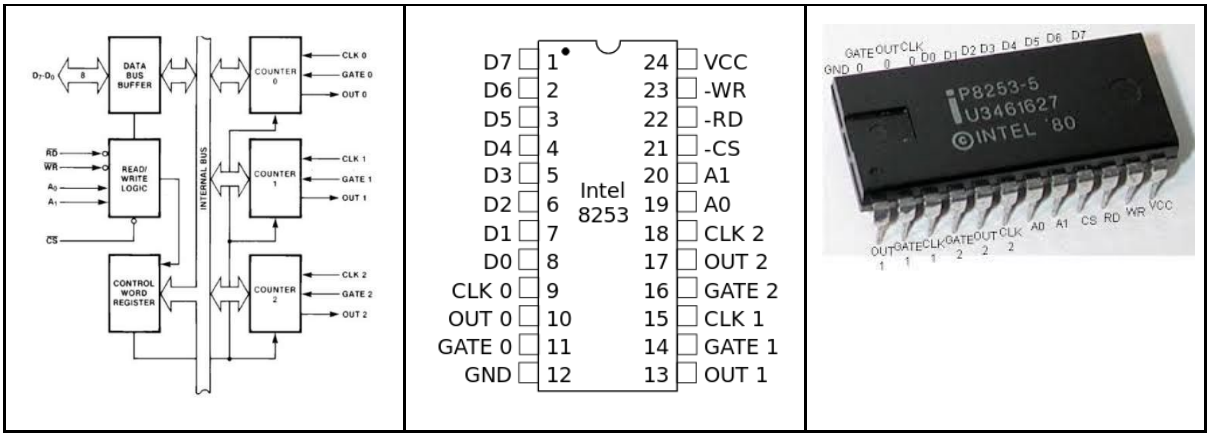


Figura 4.20: PIT 2853: temporizador de intervalos programáveis (Fonte: [43]).

Assim que se liga um computador, é executado o **sistema integrado de entrada e saída**, em inglês *Basic Input/Output System* (BIOS), para realizar uma série de testes chamados de **auto-teste na ligação**, em inglês *Power-On Self Test* (POST). Os dados do POST são mostrados na tela, numa forma tabular, antes do carregamento do sistema operacional. POST detecta, testa e inicializa os componentes instalados num sistema computacional. Falhas detectadas são avisadas através de uma sequência de *beeps* sonoros [31]. O problema era como emitir tais *beeps* antes mesmo da inicialização do sistema operacional. Foi desenvolvido um circuito de temporização e contagem capaz de fazer tais contagens e gerar sinais quando o contador atinge um valor pré-definido. O CI PIT 8253 foi usado nos computadores pessoais compatíveis com IBM PC, XT e AT (Figura 4.21). Mais tarde, o circuito passou a ser usado para regenerar a memória dinâmica DRAM que apresentaremos no Capítulo 5.

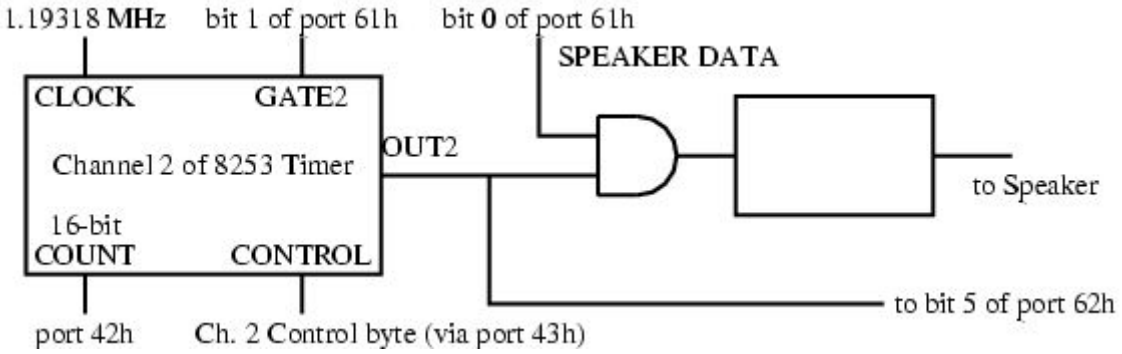


Figura 4.21: Circuito de interface entre PIT e alto-falante (Fonte: [44]).

4.7 *Chipsets*

Tanto na arquitetura de von Neumann quanto na arquitetura de Harvard, um microprocessador só tem mesmo utilidade se ele estiver se comunicando com uma

memória, de onde ele obtém a sequência de instruções que ele precisa executar, e com um periférico, através do qual ele consegue interagir como o mundo físico que o cerque. O circuito integrado que o permite fazer estas comunicações é o ***chipset***. Um *chipset* é um circuito dedicado ao gerenciamento de fluxo de dados entre microprocessadores, sistema de memória e periféricos. É normalmente encontrado na placa-mãe e projetado para operar em conjunto com uma família específica de microprocessadores. Uma divisão tradicional é considerar que um *chipset* seja constituído por duas pontes: a **ponte norte**, em inglês *northbridge*, que controla o fluxo de alta velocidade entre a CPU e a memória primária; e a **ponte sul**, em inglês *southbridge*, que controla fluxos de velocidade menor entre as conexões do sistema com os periféricos. Figura 4.22 ilustra estas conexões com um processador Athlon.

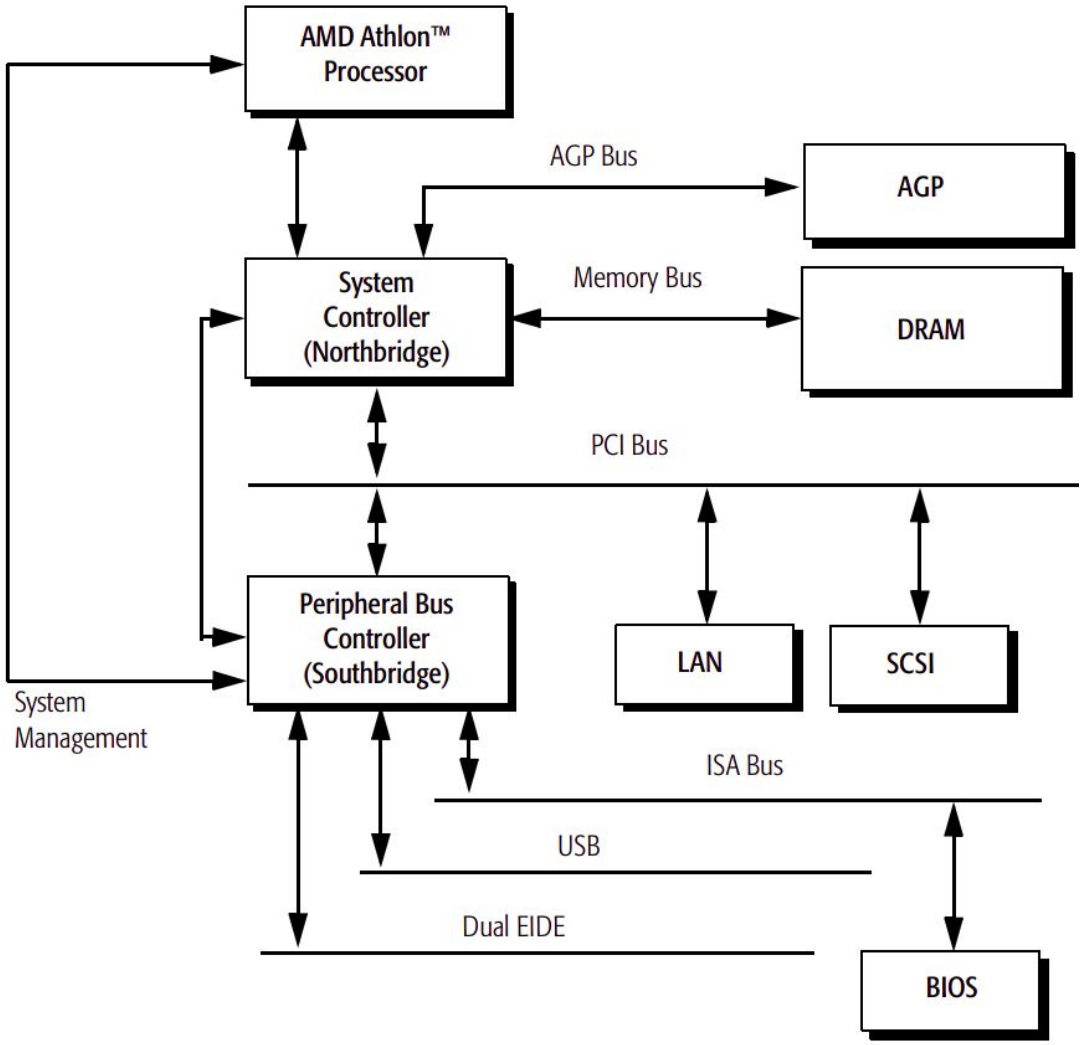


Figura 4.22: Conexões entre CPU e restante do sistema. Fonte: [19]

4.8 Níveis de Abstrações

Para programar um sistema computacional como o ilustrado na Figura 4.1 de forma otimizada, o desenvolvedor de aplicativos precisa dominar não só a arquitetura do

jogo de instruções que está sendo utilizada, como todos os detalhes de funcionamento de outros componentes eletrônicos e os sinais de conexão entre eles. Isso seria um trabalho extremamente difícil, se não seria inviável para o grau de complexidade que os sistemas modernos atingiram. Para abstrair todos esses detalhes em um modelo computacional mais simples e mais claro, a maioria dos sistemas computacionais tem integrado neles um aplicativo básico conhecido por **sistema operacional**. O sistema operacional é o responsável pelo gerenciamento dos recursos computacionais entre os aplicativos, como esquematiza a Figura 4.23. Usualmente, a linguagem C é utilizada para a implementação de um sistema operacional [20]. Os sistemas operacionais mais conhecidos entre os computadores pessoais são Windows, Linux, ou Free BSD ou Mac OS.

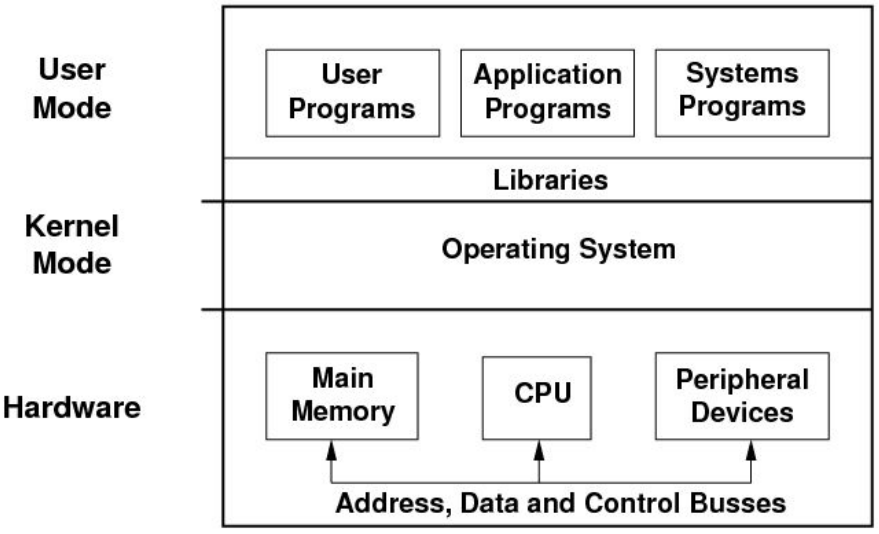


Figura 4.23: Sistema operacional: interface entre os programas dos usuários e os circuitos eletrônicos (Fonte: [45]).

4.8.1 *Software* Básico

No desenvolvimento de programas dedicados a uma tarefa específica com grande restrição temporal no topo de um sistema computacional provido de recursos bem limitados, é importante ponderarmos o custo e o benefício de integrarmos ao nosso sistema computacional um sistema operacional. Na primeira geração de sistemas embarcados, escrevia-se *firmwares* diretamente sobre os circuitos eletrônicos. Depois, incluiu-se somente o ***bootloader*** para iniciar o microprocessador, o relógio e a memória principal. Em seguida, o fluxo é desviado diretamente à sequência de instruções desenvolvidas pelo projetista. Neste caso, não tendo nenhuma camada de abstração dos recursos físicos, o projetista é diretamente responsável pela inicialização e pelo gerenciamento destes recursos através dos seus programas. A estes paradigmas de programação denomina-se **programação em metal puro**, em inglês *bare metal programming*. Hoje em dia, com o aumento da capacidade dos

processadores, já existem disponíveis sistemas operacionais embarcados, como o **sistema operacional em tempo real** (em inglês *real-time operating system* RTOS) para a arquitetura ARM.

4.8.2 *Software* de Desenvolvimento

Independente do nível de abstração em que os circuitos elétricos são visíveis aos projetistas, é necessário que estes projetistas instruam, através dos seus programas, os sistemas computacionais como pensar e o que fazer usando os recursos disponíveis no nível de abstração visível. Qual linguagem usar? Linguagem de Máquina? *Assembly*? Fortran? C? Java? Python? A escolha vai depender da demanda da aplicação para a qual está desenvolvendo o projeto [21]. Neste curso, procuramos dar uma visão geral das características relevantes dos componentes de um sistema computacional, a fim de subsidiar melhor as tomadas de decisão de um projetista de sistemas embarcados sem entrar nos detalhes das características específicas de cada projeto.

Como desenvolvedor de aplicativos de sistemas embarcados, a linguagem mais utilizada para programar os microprocessadores é ainda a linguagem C [22]. Diferente dos programas em *assembly*, os programas escritos nesta linguagem precisam ser pré-processados, compilados e ligados antes de serem efetivamente executáveis como ilustra Figura 4.24. Note que os arquivos-objeto, de extensão .o, podem estar numa biblioteca, como a biblioteca-padrão de C cuja interface é stdio.h.

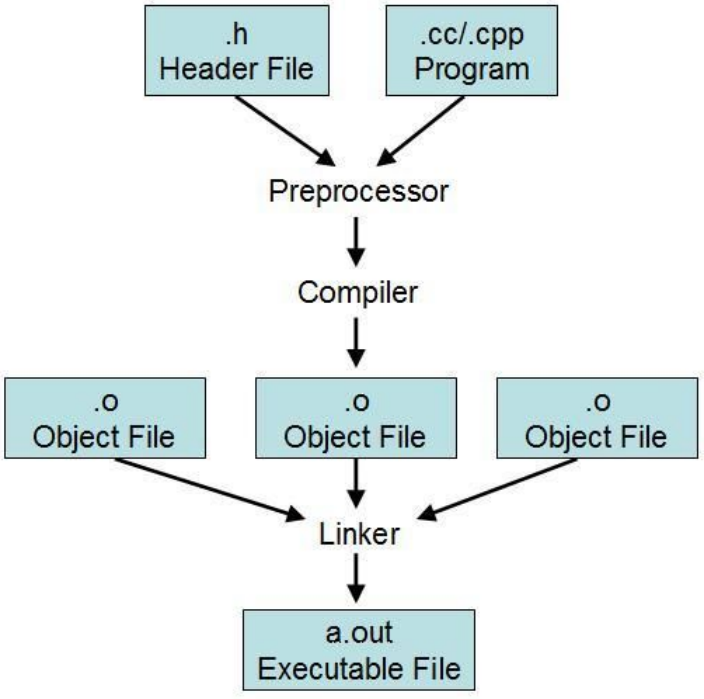


Figura 4.24: Fluxo para construção de um código executável, na linguagem de máquina, a partir dos programas-fonte em C.

4.8.3 Aplicativos

Com o suporte dos *softwares* de desenvolvimento, os projetistas de aplicativos de porte médio para cima utilizam, regra geral, linguagens de alto nível. Com o intuito de mostrar as correspondências entre as instruções de uma linguagem de alto nível e de uma linguagem de máquina, faremos uma breve introdução a algumas instruções mais utilizadas da linguagem C, com foco na sua tradução para a linguagem de máquina ARM Thumb. Mesmo que esta tradução seja feita de forma otimizada e transparente pelos compiladores modernos, ter uma melhor compreensão de como ela é feita pode ajudar na escolha das instruções apropriadas para um programa em que a eficiência temporal seja crítica. Vale também comentar que, sendo ARM Thumb uma arquitetura RISC, um comando complexo tem que ser desdobrado em instruções simples. Se o microprocessador for da arquitetura CISC, a tarefa de tradução seria mais simples.

A linguagem C dispõe de uma variedade de **operadores elementares** para manipular de forma bem flexível os valores dos seus dados, como os operadores lógicos e aritméticos, operadores sobre os *bits* de um registrador e operadores relacionais [2]. Esses operadores são usados para definir funções lógico-aritméticas mais complexas, como uma equação algébrica ou uma função lógica envolvendo várias variáveis. Um programa é, de fato, diversas sequências destas funções lógico-aritméticas conectadas pelos **comandos de decisão e de repetição**. Por meio destes comandos, podemos **programar** o controle do fluxo de execução dos operadores elementares para executar uma tarefa complexa específica. Muitas vezes, por modularidade, por concisão de códigos e/ou por inteligibilidade, é comum agrupar o fluxo de controle de execução de uma tarefa específica numa **função**, ou numa **rotina**.

Tabela 4.1 mostra a correspondência direta entre os **operadores lógico-aritméticos** e **relacionais** da linguagem C e a linguagem mnemônica ARM Thumb.

C	Descrição	ARM Thumb
+	Soma (entre inteiros)	ADD
-	Subtração (entre inteiros)	SUB
*	Multiplicação (entre inteiros)	MUL
/	Divisão	(implementado por *software*)

%	Resto da divisão	(implementado por *software*)
++	Incremento	ADD (imediato)
--	Decremento	SUB (imediato)
&	E lógico bit a bit	AND
|	OU lógico bit a bit	ORR
^	OU exclusivo bit a bit	EOR
~	Complemento bit a bit	MVN
<<	Deslocamento para esquerda	LSL
>>	Deslocamento para direita	LSR
&&	E lógico	CMP e *bits* de condição
||	OU lógico	CMP e *bits* de condição
!	Reverte o estado lógico	NEG
==	Igual a	CMP e *bits* de condição
!=	Não é igual a	CMP e *bits* de condição
>	Maior que	CMP e *bits* de condição
<	Menor que	CMP e *bits* de condição
>=	Maior ou igual a	CMP e *bits* de condição
<=	Menor ou igual a	CMP e *bits* de condição

Tabela 4.1: Correspondência entre instruções de C e ARM Thumb.

A linguagem C suporta a **junção do operador de atribuição "=" com um operador lógico-aritmético** como explicitam as equivalências na Tabela 4.2.

Atribuição em C	Equivalência
a = b;	a=b;
a+=b;	a = a+b;
a-=b;	a = a-b;
a*=b;	a = a*b;
a/=b;	a = a/b;

a%=b;	a = a%b;
a&=b;	a = a&b;
a|=b;	a = a|b;
a^=b;	a = a^b;
a<<=b;	a = a<<b;
a>>=b.	a = a>>b;

Tabela 4.2: Junção de operadores.

A linguagem C é provida ainda de um conjunto de comandos básicos de controle com os quais podemos construir, junto com os seus operadores, programas que conseguem tomar decisões ou iterar conforme os resultados obtidos. O primeiro grupo de comandos é conhecido por **comandos condicionais**, em inglês *branching* [5], enquanto o segundo grupo é chamado de **comandos de iteração**, em inglês *looping* [6]. Existem ainda comandos que podem interromper um fluxo de **controle incondicional**mente, em inglês *breaking*.

Dos comandos condicionais temos:

- **if … else**

  if (expressão lógica**) {**
  sequência de instruções quando verdadeiro
  } else {
  sequência de instruções quando falso
  }

  que pode assumir a seguinte forma reduzida na linguagem C

  expressão lógica **?** valor 1 quando verdadeiro **:** valor 2 quando falso**;**

- **switch**

  switch(expressão lógico-artitmética**)**
  {
  case valor1**:** sequência de instruções quando a expressão assume valor1**;**
  break;
  case valor2**:** sequência de instruções quando a expressão assume valor2**;**
  break;
  :
  :
  case valorN**:** ***sequência de instruções quando a expressão assume valorN*****;**
  break;
  default : sequência para casos não contemplados**;**
  }

Figura 4.25 e Figura 4.16 ilustram a tradução de uma instrução condicional, destacada pela cor vermelha, em instruções da arquitetura ARM Thumb, em azul, pelo compilador do GNU ARM *toolchain*.

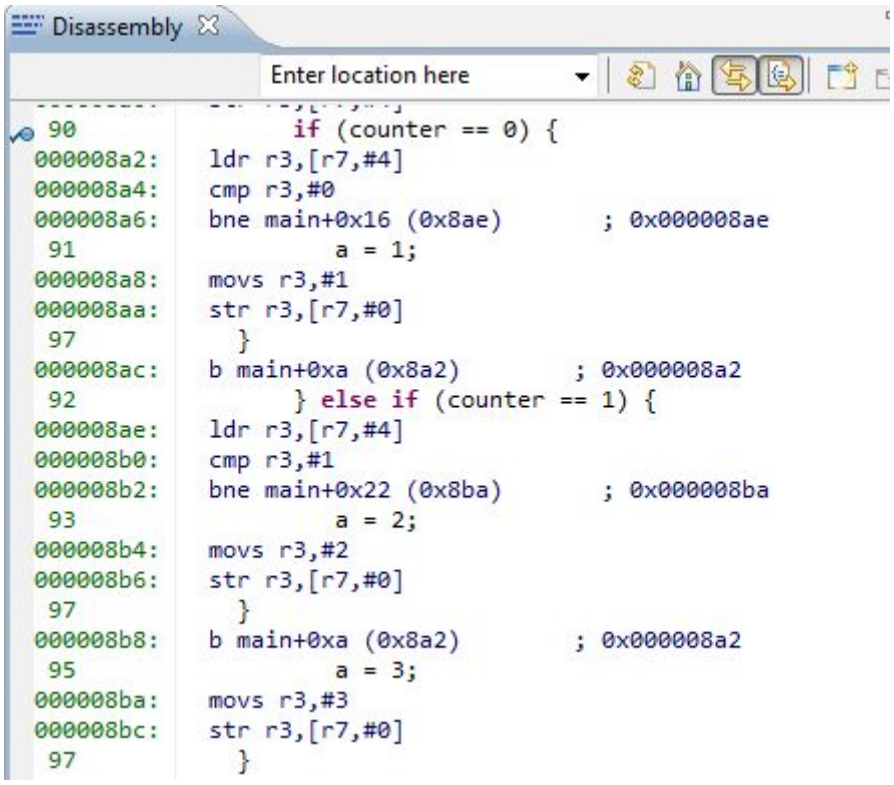


Figura 4.25: Correspondência entre uma instrução **if...else if...else** em C (em vermelho) e as instruções de ARM Thumb (em azul).

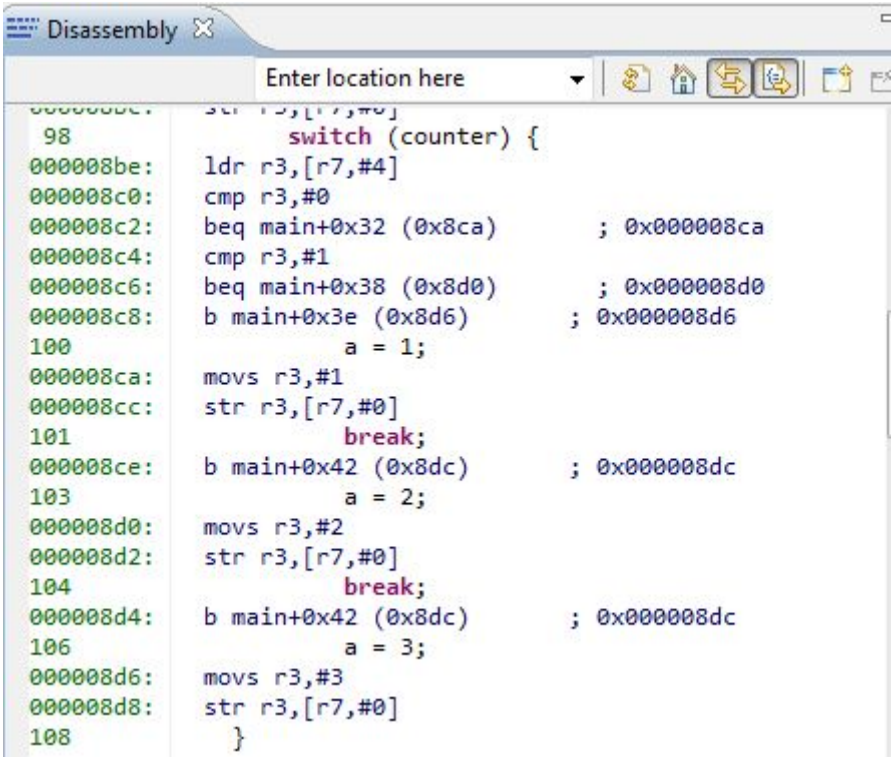


Figura 4.26: Correspondência entre uma instrução **switch** em C (em vermelho) e as instruções de ARM Thumb (em azul).

Os fluxos de controle conhecidos como **laços** são aqueles que permitem não só repetir uma sequência de instruções como também controlar a quantidade de iterações desejada explicitamente ou condicionada a uma expressão lógico-aritmética. A linguagem C dispõe dos seguintes comandos para controlar estes fluxos:

- **while**
  while (expressão lógica**) {**
  sequência de instruções
  }

```
 25           while (i8a != i8b) {
000015b2:   b main+0x20 (0x15b8)      ; 0x000015b8
 26               int a = 0;
000015b4:   movs r3,#0
000015b6:   str r3,[r7,#36]
 25           while (i8a != i8b) {
000015b8:   ldr r2,[r7,#48]
000015ba:   ldr r3,[r7,#44]
000015bc:   cmp r2,r3
000015be:   bne main+0x1c (0x15b4)      ; 0x000015b4
```

- **do**
  do {
  sequência de instruções
  } while (expressão lógica-aritmética**);**

```
 33               int a = 0;
000015d8:   movs r3,#0
000015da:   str r3,[r7,#28]
 34           } while (i8a != i8b);
000015dc:   ldr r2,[r7,#48]
000015de:   ldr r3,[r7,#44]
000015e0:   cmp r2,r3
000015e2:   bne main+0x40 (0x15d8)      ; 0x000015d8
```

- **for**
  for (inicialização**;** expressão lógica-aritmética**;** condição**) {**
  sequência de instruções
  }

```
 25           for (i8a = 0; i8a < 12; i8a++) {
000015b2:   movs r3,#0
000015b4:   str r3,[r7,#68]
000015b6:   b main+0x24 (0x15c2)      ; 0x000015c2
 26               int a = 0;
000015b8:   movs r3,#0
000015ba:   str r3,[r7,#44]
 25           for (i8a = 0; i8a < 12; i8a++) {
000015bc:   ldr r3,[r7,#68]
000015be:   adds r3,#1
000015c0:   str r3,[r7,#68]
000015c2:   ldr r3,[r7,#68]
000015c4:   cmp r3,#11
000015c6:   bls main+0x20 (0x15b8)      ; 0x000015b8
```

Finalmente, há dois comandos de **quebra** de fluxo que é muito útil em programas de sistemas embarcados, embora os seus usos não sejam recomendados para um programa bem estruturado [3]:

- **goto:** desvio incondicional para o endereço do rótulo

  inicio: **while** (1) {
  sequência de instruções 1
  if (entrada) **goto** inicio;
  sequência de instruções 2
  }

- **continue:** interrompe o laço de iterações em que se encontra

  while (1) {
  sequência de instruções 1
  if (entrada) **continue**;
  sequência de instruções 2
  }

4.9 Exercícios

1. Seja uma máquina de 32 *bits* cujas instruções tem tamanho de 4 *bytes*. Na execução de uma sequência de instruções, como o contador de programa PC é incrementado após a busca de uma instrução?
2. Quais dos seguintes microprocessadores tem a arquitetura CISC: x86, ARM, Krait, MIPS, Motorola 68000, PDP-11? E quais são da arquitetura RISC?
3. Uma mesma instrução de uma linguagem *assembly* é traduzida em um mesmo conjunto de sinais de controle em todas as máquinas que suportam o jogo de instruções correspondente? Justifique.
4. Seja uma CPU de 32 bits. Qual seria a forma mais eficiente, em termos de quantidade de acessos, para armazenar uma *string* de 51 caracteres? E se for uma máquina de 8 bits?
5. O valor representado pelo número 0x5249 5343 5643 5055 (hexadecimal) deve ser armazenado em uma **palavra dupla alinhada** de 64 *bits*.
    a) Considerando que este valor armazenado seja lido *byte* a *byte*, escreva o valor a ser armazenado usando a ordenação *Big Endian*.
    b) Usando o mesmo arranjo físico da parte (a), escreva o valor para ser armazenado usando a ordem de byte *Little Endian*.
    c) Quais são os valores hexadecimais de todas as palavras desalinhadas de 2 *bytes* que pode ser lido a partir da palavra dupla de 64 bits fornecida quando armazenado em ordem de *byte Big Endian*?
    d) Quais são os valores hexadecimais de todas as palavras desalinhadas de 2 *bytes* que pode ser lido a partir da palavra dupla de 64 bits quando armazenado em ordem de byte *Little Endian*?
    e) Quais são os valores hexadecimais de todas as palavras de 4 *bytes* desalinhadas que podem ser lidos a partir da palavra dupla de 64 *bits* quando armazenado em ordem de *byte Little Endian*?
6. Seja um jogo de instruções ARM Thumb onde 8 registradores são endereçáveis e de organização register-register. Sabendo que são
    a) 5 *bits* reservados para *opcode* de deslocamento de *n* bits para esquerda pelo modo imediato, qual é a quantidade máxima de *bits* que se pode programar com a instrução?
    b) 6 *bits* reservados para *opcode* de soma de três operandos, dois armazenados nos registradores e um terceiro endereçável por um dos modos definido por um *bit* reservado para o modo de endereçamento. Qual é o tamanho do terceiro operando se o modo de endereçamento for por registrador? E se for endereçado pelo modo imediato?

7. Dada uma sequência de instruções, entre as quais temos uma instrução de desvio condicional. O que pode acontecer de segmentarmos esta instrução e paralelizar os seus segmentos com segmentos de outras instruções?
8. Qual é a função dos registradores entre os segmentos de uma instrução lógico-aritmética na Figura 4.15?
9. Sendo a GPU uma unidade processadora, dedicada ao processamento gráfico, podemos remover o microprocessador da placa-mãe, passando o controle do sistema computacional integralmente para ela? E em relação a um processador DSP32C da Lucent, o DPS32C pode assumir o controle de um sistema computacional? Justifique suas respostas.
10. Interrupções são eventos que podem acontecer de forma assíncrona ao fluxo de controle pré-programado. É possível predizer os instantes em que ocorrem tais eventos e inserir as suas chamadas nos pontos apropriados de um programa sequencial? Qual é a solução mais aplicada para tratá-los nos sistemas atuais?
11. Condições de corrida, impasse e inanição de recursos são três situações que podem ocorrer numa execução concorrente de processos. Vimos na Seção 4.7 que o *chipset* é responsável pelo fluxo de dados entre a CPU e o restante do sistema computacional. Se conectarmos, através do *chipset*, à CPU uma impressora e um *led* cujo estado alterna a uma frequência de 1KHz quando se inicia um processo de impressão e vá para o estado apagado quando se conclui uma impressão. Podem ocorrer condições de corrida, impasse e inanição de recursos numa impressão de um arquivo? Explique.
12. Qual configuração você escolheria na compra de um novo PC com SO Windows? Um PC com um processador *dual-* ou *quad-core* 2.8GHz ou um com um processador *single-core* 3.4GHz? Justifique.
13. A velocidade da execução da BIOS reduz com o aumento da velocidade da CPU? Justifique.
14. Para emitir os sinais sonoros de alerta pelos erros detectados na inicialização de um sistema computacional, não seria mais simples usar uma das funções de tempo disponíveis em sistemas operacionais? Justifique.
15. Entre os circuitos eletrônicos integrados e um aplicativo há vários níveis de abstração de um sistema operacionais tanto na camada de *hardware* quanto na camada de *software*. Quais são os níveis na camada de *hardware*? E quais na camada de *software*?
16. Uma mesma instrução de controle da linguagem C é traduzida sempre para uma mesma sequência de instruções em linguagem de máquina quando as máquinas tiverem a mesma CPU?
17. Analise as traduções das instruções de controle em linguagem C para linguagem de máquina ARM Thumb mostradas na Seção 4.8.3. Quais são as instruções usadas na linguagem de montagem para controlar desvios de um fluxo de controle sequencial?

4.10 Referências

[1] ARM Cortex-M Programming Guide to Memory Barrier Instructions (Application Note 321)
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf
[2] Tutorialspoint. C Operators.
https://www.tutorialspoint.com/cprogramming/c_operators.htm.
[3] Sérgio Prado. Você usa goto nos seus códigos em C?
https://sergioprado.org/voce-usa-goto-nos-seus-codigos-em-c/
[4] Michael S. Schlansker, B. Ramakrishna Rau. EPIC: An Architecture for Instruction-Level Parallel Processors.
https://pt.scribd.com/document/63760972/EPIC-Architecture-ILP
[5] Tutorialspoint. *C Decision Making*.
https://www.tutorialspoint.com/cprogramming/c_decision_making.htm
[6] Tutorialspoint. C Loops.
https://www.tutorialspoint.com/cprogramming/c_loops.htm
[7] ARM. ARMv6-M Architecture Reference Manual
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/ARMv6-M.pdf
[8] ARM and Thumb-2 Instruction Set - Quick Reference Card
http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf
[9] ARM7TDMI - Technical Reference Manual.
http://ww1.microchip.com/downloads/en/DeviceDoc/DDI0029G_7TDMI_R3_trm.pdf
[10] GeeksforGeeks. Addressing Modes.
https://www.geeksforgeeks.org/addressing-modes/
[11] Weijun Zhang. VHDL Tutorial: Learn by Example.
http://esd.cs.ucr.edu/labs/tutorial/
[12] hardwarebee. FPGA vs. CPU - What is the difference.
http://hardwarebee.com/fpga-vs-cpu-difference/
[13] Intel. Intel 8086
https://www.archive.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:8086-datasheet.pdf
[14] Wikipedia. RS/6000. https://pt.wikipedia.org/wiki/RS/6000
[15] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual.
https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
[16] GeekksForGeeks. RTL (Register Transfer Level) design vs Sequential logic design

https://www.geeksforgeeks.org/rtl-register-transfer-level-design-vs-sequential-logic-design/
[17] kmittal82. ARM/Thumb/Thumb-2. https://kmittal82.wordpress.com/2012/02/17/armthumbthumb-2/ (informações jogo de instruções desatualizadas)
[18] Intel. Datasheet, Vol. 1: Intel® Core™ i7 Processor for LGA2011-v3 Socket. https://www.intel.com.br/content/www/br/pt/products/docs/processors/core/core-i7-6xxx-lga2011-v3-datasheet-vol-1.html
[19] AMD. AMD Athlon Processor Model 4 Data Sheet. https://www.dexsilicium.com/AMD_Athlon.pdf
[20] Tanenbaum, A. S. Sistemas Operacionais Modernos. ISBN: 978-8543005676. Pearson Universidade. 2015
[21] Maharajan, Veerabahu. 5 Roles played by an Embedded Software Engineer. https://www.linkedin.com/pulse/5-roles-played-embedded-engineer-maharajan-veerabahu?trk=prof-post
[22] Maharajan, Veerabahu. 3 Steps to become an Embedded Software Engineer. https://www.linkedin.com/pulse/3-steps-become-embedded-software-engineer-maharajan-veerabahu?trk=pulse_spock-articles
[23] Motorola. DSP56000 24-BIT DIGITAL SIGNAL PROCESSOR FAMILY MANUAL. https://www.nxp.com/docs/en/reference-manual/DSP56000UM.pdf
[24] Kun-Shan Lin Gene A. Frantz Ray Simar, Jr..The TMS320 Family of Digital Signal Processors APPLICATION REPORT: SPRA396. http://www.ti.com/lit/an/spra396/spra396.pdf
[25] *Site* em português da Nvidia. https://www.nvidia.com/pt-br/
[26] *Site* em portughuês da AMD. https://www.amd.com/pt/graphics/radeon-rx-graphics
[27] Quora. What are Adreno, Mali, Tegra and PowerVR? And which of them does my Samsung Galaxy Tab2 use? https://www.quora.com/What-are-Adreno-Mali-Tegra-and-PowerVR-And-which-of-them-does-my-Samsung-Galaxy-Tab2-use
[28] Xilink. MicroBlaze Processor Reference Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug984-vivado-microblaze-ref.pdf
[29] Altera. Nios II Classic Processor Reference Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf
[30] Oracle. Assembler Directives. https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
[31] Adriano Meirelles. Hardware Manual Completo - Códigos de erro do BIOS. https://www.hardware.com.br/livros/hardware-manual/codigos-erro-bios.html
[32] Componentes de um Computador. https://sites.google.com/site/admtopicosdeinformatica/componentes-de-um-computador
[33] GeeksForGeeks. Microarchitecture and Instruction Set Architecture. https://www.geeksforgeeks.org/microarchitecture-and-instruction-set-architecture/
[34] Wikipedia. Intel 8080. https://en.wikipedia.org/wiki/Intel_8080#/media/File:Intel_8080_arch.svg

[35] Wikipedia. Extremidade (ordenação).
https://pt.wikipedia.org/wiki/Extremidade_(ordena%C3%A7%C3%A3o)
[36] Elaine Cecília Gatto. Arquitetura de John Von Neumann.
https://www.embarcados.com.br/arquitetura-de-john-von-neumann/
[37] Harvard Computer Architecture.
https://konstantin.solnushkin.org/teaching_reports/intro_to_hpc/2007/harvard_architecture.pdf
[38] https://en.wikipedia.org/wiki/File:Von_Neumann_architecture.svg
[39] https://pt.wikipedia.org/wiki/Arquitetura_Harvard
[40] GeeksForGeeks. Computer Organization | Instruction Formats (Zero, One, Two and Three Address Instruction).
https://www.geeksforgeeks.org/computer-organization-instruction-formats-zero-one-two-three-address-instruction/
[41] Operating Systems. Mips coprocessor 0.
http://www.it.uu.se/education/course/homepage/os/vt18/module-1/mips-coprocessor-0/
[42] TechDifferences. Difference Between Multitasking and Multithreading in OS.
https://techdifferences.com/difference-between-multitasking-and-multithreading-in-os.html
[43] TutorialsPoint. Intel 8253 - Programmable Interval Timer.
https://www.tutorialspoint.com/microprocessor/microprocessor_intel_8253_programmable_interval_timer
[44] Célio Cardoso Guimarães. 8353 Timer Chip.
http://www.ic.unicamp.br/~celio/mc404s2-03/8253timer.html
[45] Minnie. Introduction to Operating System.
https://minnie.tuhs.org/CompArch/Lectures/week07.html