# Capítulo 2

# Conceitos de Arquiteturas de Microprocessadores

## 2.1. Computador de programa armazenado

Antes da década de 1950, os computadores eram criados para a solução de um problema específico, como no exemplo do processador que faz o cálculo da função *fatorial(N)*, mostrado no capítulo anterior. Para processar outras funções, eram necessárias demoradas e dispendiosas modificações no hardware.

Um computador de programa armazenado é um conceito que orienta a maioria das arquiteturas modernas de computadores, creditado em sua maior parte a John von Neumann. O conceito de programa armazenado determina que os programas (instruções) e os dados devem estar em uma memória de acesso direto (memória de acesso aleatório), permitindo que o programa (instruções) e os dados sejam tratados indiferentemente, possibilitando, inclusive, que ambos sejam modificados. A modificação das instruções do programa, nesse caso, possibilita a solução computacional de diferentes problemas.

Um programa é uma sequência ordenada de instruções codificadas por palavras binárias armazenadas sequencialmente na memória. Uma instrução consiste essencialmente na indicação de uma operação e dos dados sobre os quais ela deve operar. O processador, durante sua operação, busca uma instrução na memória, decodifica e executa esta instrução, repetindo sequencialmente essas ações para todas as instruções do programa.

O endereçamento de uma instrução na memória de programa ocorre mediante o uso de um registrador especial chamado Contador de Programa, ou *PC* (*Program Counter*), que pode estar contido no hardware da unidade de controle ou, em alguns casos, pode ser um dos registradores do banco. As linhas de saída do PC fornecem diretamente o endereço da próxima instrução a ser buscada na memória.

Na arquitetura de von Neumann ou de Harvard, as instruções são executadas sequencialmente. Para isso, a CPU implementa no hardware o laço mostrado na Figura 1.

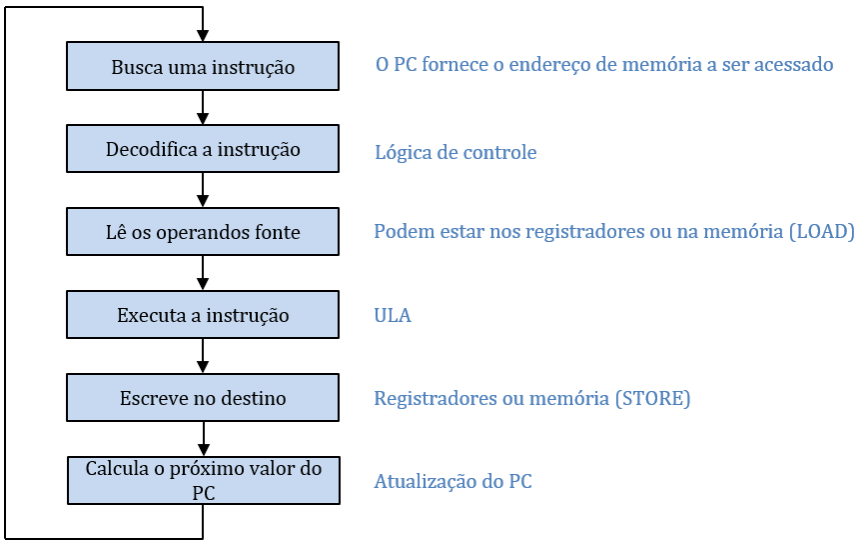



*Figura 1 – Laço de execução das instruções na arquitetura de von Neumann ou de Harvard.*

O laço inicia com o processador acessando a memória para buscar a próxima instrução a ser executada. O endereçamento da memória, como dito anteriormente, é feito pelo PC. A palavra endereçada armazenada na memória (instrução) entra no processador pela unidade de controle. O hardware da unidade de controle decodifica a instrução, gerando os sinais lógicos necessários para acessar os operandos fonte, que podem estar no banco de registradores ou armazenados na própria memória, caso a instrução seja do tipo LOAD. Nessa situação, um novo acesso à memória deve ser feito para buscar o operando. Em seguida, a instrução é executada, normalmente utilizando a ULA, cujos sinais de controle já foram previamente fornecidos na etapa de decodificação da instrução. O resultado da operação é escrito no registrador de destino, ou na memória, exigindo mais um acesso, caso a instrução seja do tipo STORE. Finalmente, o PC é automaticamente atualizado com o valor do endereço da próxima instrução a ser executada e o laço é reiniciado. Esse laço pode ser executado em um único pulso de clock, em processadores com microarquitetura conhecida como *single-cycle*, pode ser executado em vários pulsos de clock, nas arquiteturas conhecidas como *multi-cycle*, ou podem ser executados vários laços em paralelo nas arquiteturas conhecidas como *pipeline*.

Normalmente, a memória é endereçada byte a byte. Como as instruções estão armazenadas sequencialmente na memória, o próximo valor do PC será o valor atual acrescido do tamanho, em bytes, da instrução atualmente executada, o que fará o PC apontar para o endereço da próxima instrução na memória, a menos que a instrução atual diga o contrário, como ocorre com as instruções de desvio, por exemplo.

Do ponto de vista do hardware, dados e instruções são vistos como uma sequência de bits armazenados em palavras binárias na memória, como mostrado na Figura 2. O sistema faz a distinção entre ambos dependendo da forma como eles são tratados no processador: Se a palavra de memória acessada for um dado, ela é enviada para o *datapath* para ser processada, enquanto que se for uma instrução é enviada para a unidade de controle para ser decodificada e executada.

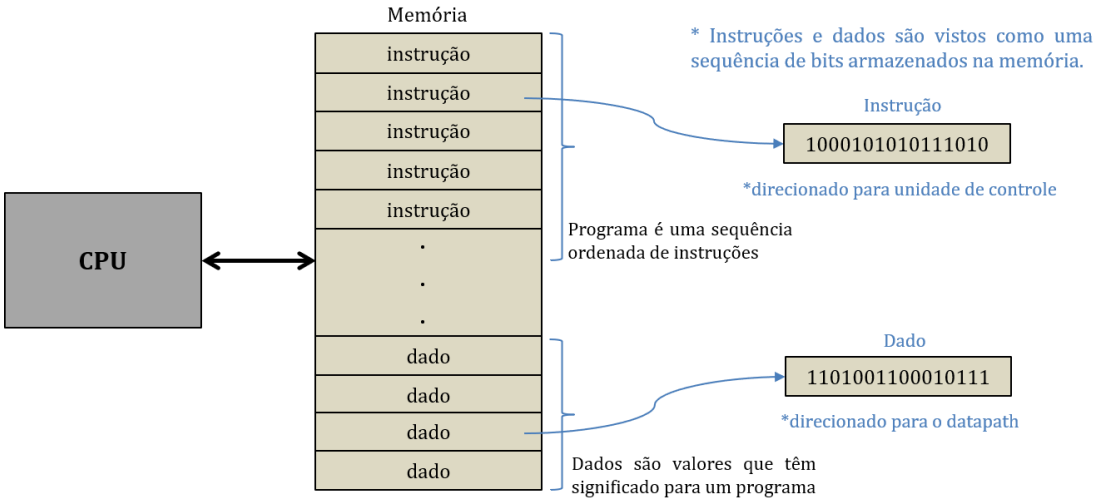



*Figura 2 – Memória do ponto de vista do hardware.*

De um modo geral, cada instrução armazenada na memória é formada por campos de bits com diferentes significados que serão interpretados pela unidade de controle do processador. O primeiro campo é o código da operação (abreviado como *opcode*), os demais campos indicam o(s) endereço(s) do(s) dado(s) sobre o qual, ou quais, ela deve operar, também conhecidos como operandos. Assim, podemos ver uma instrução como uma entidade cujo formato armazenado na memória possui basicamente os campos mostrados na Figura 3.

| *opcode* | *endereço dos operandos* |
|---|---|

*Figura 3 – Formato geral de uma instrução.*

O campo *opcode* é formado por *n* bits, onde *n* deve ser a quantidade mínima necessária para codificar e distinguir todas as *M* instruções básicas do processador. O valor de *n* é calculado pela função teto, mostrada abaixo:

$$n = \lceil log_2 M \rceil$$

Por exemplo, consideremos que um determinado processador seja capaz de executar $M$=90 instruções básicas distintas. Logo, o campo *opcode* deve possuir $n = \lceil log_2 90 \rceil = 7$ bits. Nem todos os códigos disponíveis para esse campo precisam ser utilizados pelo processador. Com $n$=7 bits, por exemplo, seria possível ter até $2^7$=128 diferentes códigos de instrução. Entretanto, para o processador do exemplo, apenas 90 são utilizados.

O campo de *endereço dos operandos* pode assumir diferentes formatos, conhecidos como "modos de endereçamento". Cada projeto de processador pode implementar uma variedade de modos, a fim de oferecer flexibilidade de programação ao usuário por meio de ponteiros para a memória, contadores para controle de loops, indexação de dados e reduzir o número de bits nos campos de endereço dos dados da instrução. Além disso, pode ou não haver um campo de endereço na instrução. Se houver, pode designar um endereço de memória ou um registrador do processador.

A disponibilidade de vários modos de endereçamento oferece ao programador experiente a capacidade de escrever programas que exigem menos instruções. No entanto, o efeito no rendimento e no tempo de execução deve ser cuidadosamente ponderado. Por exemplo, a presença de modos de endereçamento mais complexos pode resultar em menor rendimento e maior tempo de execução.

No modo de endereçamento direto, por exemplo, o campo de *endereço dos operandos* especifica diretamente o endereço de memória onde o dado está armazenado ou onde o resultado da operação será armazenado. No modo de endereçamento indireto, o campo de endereço fornece um endereço de memória no qual o endereço efetivo do dado é armazenado. A unidade de controle usa o campo de *endereço dos operandos* da instrução para acessar a memória a fim de ler o endereço efetivo do dado e acessar novamente a memória para fazer a leitura do dado. Isso permite acessar endereços de memória com maior capacidade (que exige mais bits de endereço), visto que o campo de endereços é limitado em quantidade de bits, o que limita o alcance do endereçamento.

Quando o campo de *endereço dos operandos* é formado por ponteiros que indicam os registradores fonte, onde os dados estão armazenados, e o registrador de destino do resultado da operação, dizemos que o endereçamento está no modo registrador. Nesse modo, os operandos estão em registradores do banco. O registrador específico é selecionado pelo campo de endereço no formato da instrução. Esse modo pode conter um, dois, três ou até mais ponteiros para registradores, dependendo do projeto do processador.

Consideremos o exemplo de um processador com um banco composto por 16 registradores de uso geral e que a ULA seja capaz de realizar operações que utilizam até 3 registradores simultaneamente, sendo 2 registradores fonte e um registrador de destino. Para indicar cada um desses operandos, o endereçamento exige o uso de 4 bits ($2^4$=16). Nesse caso, são necessários 12 bits de endereçamento dos operandos, sendo 8 para os dois operandos fontes e 4 para o destino. Se chamarmos esses campos com 4 bits de *Ra* (primeiro operando), *Rb* (segundo operando) e *Rc* (destino), a instrução poderia ter o seguinte formato detalhado:

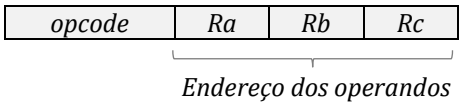



*Figura 4 – Formato de uma instrução com endereçamento no modo registrador.*

Esse formato de instrução informa que o processador deve executar a operação indicada pelo *opcode* sobre os dados armazenados nos registradores *Ra* e *Rb* e o resultado deve ser armazenado no registrador *Rc*, o que pode ser expresso por:

$$Rc \leftarrow op(Ra, Rb)$$

Alguns operandos podem não ser explicitamente indicados, porque sua localização é previamente definida pelo próprio *opcode* da instrução. Nesse caso, dizemos que o operando tem um endereçamento implícito. Se o endereço estiver implícito, não há necessidade de um campo de bits para esse operando na instrução. Por outro lado, se um operando tem um endereço na instrução, dizemos que o operando é explicitamente endereçado ou tem um endereço explícito, como é o caso do endereçamento direto.

O número de operandos explicitamente endereçados é um fator importante na definição do projeto do conjunto de instruções para um processador. Esse fator é tão importante na definição da natureza das instruções que ele atua como um meio de distinguir diferentes arquiteturas de conjunto de instruções. Ele também controla o comprimento, em bits, das instruções do processador.

Na Figura 5 são mostrados dois exemplos de endereçamento implícito: Em *(a)*, apenas o operando *Ra* é indicado, o outro operando está implícito no *opcode*, como ocorre nas instruções das arquiteturas baseadas em acumulador, onde o acumulador é o próprio operando implícito; Em (*b*), nenhum operando está explícito, existe apenas o *opcode*, como ocorre nas instruções das arquiteturas baseadas em pilha, onde o topo da pilha é sempre o endereço implícito.


*Figura 5 – Formatos de instruções com endereçamento implícito: (a) com um operando explícito, (b) sem operandos explícitos.*

No modo indireto com registrador, a instrução especifica um registrador no processador cujo conteúdo fornece o endereço do operando na memória. Em outras palavras, o registrador selecionado contém o endereço de memória do operando, em vez do próprio operando. Antes de usar uma instrução com modo de endereçamento indireto com registrador, o programador deve garantir que o endereço de memória esteja disponível no registrador. Uma referência ao registrador é então equivalente à especificação de um endereço de memória. A vantagem do modo indireto com registrador é que o campo de endereço da instrução usa menos bits para selecionar um registrador do que seria necessário para especificar diretamente um endereço de memória, sendo essas operações expressas por:

$$Rc \leftarrow op([Ra], Rb), \qquad Rc \leftarrow op(Ra, [Rb]) \qquad \text{ou} \qquad [Rc] \leftarrow op(Ra, Rb)$$

A primeira expressão indica que a operação é efetuada sobre o dado armazenado no endereço de memória especificado por *Ra*, também expresso por M[*Ra*], e o dado armazenado diretamente em *Rb*. Na segunda expressão, a operação é efetuada sobre o dado de *Ra* e o dado armazenado no endereço de memória especificado por *Rb*, também expresso por M[*Rb*]. Na terceira expressão, a operação é efetuada sobre os dados armazenados em *Ra* e *Rb* e o resultado é armazenado no endereço de memória especificado por *Rc*, também expresso por M[*Rc*].

No modo de endereçamento imediato, o operando é especificado na própria instrução. Em outras palavras, uma instrução como modo de endereçamento imediato possui um campo de operando em vez de um campo de endereço do operando. Esse campo contém o operando real a ser usado em conjunto com a operação especificada na instrução. Instruções no modo imediato são úteis, por exemplo, para inicializar um registrador com um determinado valor. O formato dessa instrução é mostrado na Figura 6.


*Figura 6 – Formato das instruções com endereçamento imediato.*

Alguns modos de endereçamento exigem que o campo de endereço da instrução seja adicionado ao conteúdo de um registrador especificado na CPU para obter o endereço efetivo do dado. Frequentemente, o registrador usado é o PC. No modo de endereçamento relativo, o endereço efetivo é calculado da seguinte maneira:

$$Endereço\ efetivo = campo\ de\ endereço\ da\ instrução + conteúdo\ do\ PC$$

Nesse modo, o campo do endereço da instrução é considerado um número com sinal, que pode ser positivo ou negativo. Quando esse número é adicionado ao conteúdo do PC, o resultado produz um endereço efetivo cuja posição na memória é relativa ao endereço da próxima instrução no programa.

Para esclarecer isso com um exemplo, vamos supor que o PC contenha o número 250 e o *campo de endereço da instrução* contenha o número 500. O cálculo do endereço efetivo do dado para o modo de endereçamento relativo é 250 + 500 = 750. O resultado é que o operando associado à instrução está armazenado na posição endereçada pelo número 750.

O endereçamento relativo geralmente é usado nas instruções de desvio do fluxo de execução do programa, quando o endereço do desvio está em um local próximo à instrução de desvio. O endereçamento relativo produz instruções mais compactas, já que o campo de endereço pode ser especificado com menos bits do que o necessário para designar diretamente o endereço de memória.

No modo de endereçamento indexado, o conteúdo de um registrador específico é adicionado à parte do endereço da instrução para obter o endereço efetivo, semelhante ao que ocorre no modo de endereçamento relativo. O registrador de índice, nesse caso, pode ser qualquer um do banco, sendo o PC o mais utilizado.

Ilustramos o uso do endereçamento indexado considerando uma matriz de dados na memória. O campo de endereço da instrução define o endereço inicial da matriz. Cada operando na matriz é armazenado na memória em relação ao endereço inicial. A distância entre o endereço inicial e o endereço do operando é o valor do índice armazenado no registrador. Qualquer operando na matriz pode ser acessado com a mesma instrução, desde que o registro de índice contenha o valor correto do índice. O registro de índice pode ser incrementado ou decrementado para facilitar o acesso a operandos consecutivos.

## 2.2. Arquitetura do conjunto de instruções (*ISA – Instruction Set Architecture*)

A sequência binária na qual as instruções são codificadas e armazenadas na memória é chamada de linguagem de máquina, ou linguagem de baixo nível, e chegou a ser utilizada, por um bom tempo, como a linguagem de programação dos computadores. Entretanto, a linguagem de máquina é pouco prática para ser compreendida de uma maneira direta e objetiva por um programador.

Uma linguagem que substitui os códigos das operações e os endereços binários por nomes simbólicos e que fornece outros recursos úteis ao programador é a chamada linguagem assembly, ou linguagem de montagem. Cada processador, ou família de processadores com características semelhantes, possui sua linguagem assembly individual.

A estrutura lógica dos processadores é normalmente descrita nos manuais de referência da sua linguagem assembly. Tais manuais explicam vários elementos internos do processador que são do interesse do programador, como os nomes e a quantidade dos registradores de uso geral, por exemplo. Os manuais também listam todas as instruções implementadas por hardware, especificam os nomes simbólicos, o formato do código binário das instruções e fornecem uma definição precisa de cada instrução. Assim, a programação utilizando a linguagem assembly para um determinado processador exige do programador o conhecimento da sua estrutura interna.

No passado, essa informação representava a arquitetura do computador. Um computador era composto de sua arquitetura, além de uma implementação dessa arquitetura. A implementação foi separada em duas partes: a organização e o hardware. A organização consiste em estruturas como o datapath, a unidade de controle, memórias e os barramentos que os interconectam. O hardware refere-se à lógica, às tecnologias eletrônicas empregadas e aos vários aspectos de design físico dos circuitos do processador.

Com o passar do tempo, os relacionamentos entre arquitetura, organização e hardware tornaram-se tão entrelaçados que um ponto de vista mais integrado se tornou necessário. De acordo com esse novo ponto de vista, a arquitetura conforme definida anteriormente é chamada mais restritamente de Arquitetura do Conjunto de Instruções (*ISA – Instruction Set Architecture*). A estrutura de uma implementação de hardware específica da *ISA* é chamada de microarquitetura ou organização do processador e o termo arquitetura é hoje usado para abranger todo o processador, incluindo arquitetura, organização e hardware do conjunto de instruções.

A arquitetura do set de instruções pode ser compreendida como a interface entre o software e o hardware de um determinado processador. É o que informa o que um determinado processador é capaz de fazer. Ela define as instruções básicas e seus formatos binários, quanto armazenamento dispõe (quantidade de registradores), além de informar precisamente como o software pode invocar essas instruções e acessar os dados. A *ISA* é a primeira camada de abstração do hardware. Ela especifica o que o hardware fornece, mas não especifica como ele é implementado, escondendo do usuário a complexidade de sua implementação. Por exemplo, a *ISA* pode fornecer uma operação de soma de dois operandos, mas não irá especificar como o hardware do somador foi implementado.

No exemplo apresentado na Figura 7, é ilustrada a relação entre o código de máquina e a linguagem assembly para um determinado processador. Nessa figura, temos uma instrução de adição armazenada na memória e codificada pela sequência binária mostrada.

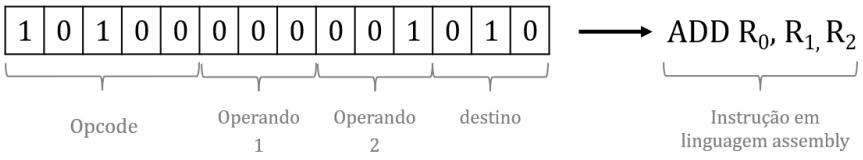



*Figura 7 – Relação entre código de máquina e linguagem assembly.*

A instrução é completamente codificada usando 14 bits. Os 5 primeiros bits da esquerda para a direita (*0b10100*) representam o código da operação de adição. Cada operando é indexado por 3 bits. O operando 1 é indexado como (*0b000*), o operando 2 é indexado como (*0b001*) e o destino é indexado como (*0b010*).

Essa instrução pode ser representada de uma maneira mais inteligível pela linguagem simbólica (assembly) cujo formato é mostrado do lado direito da figura. A instrução de adição é indicada pelas três letras ADD, chamadas de mnemônico da instrução. O mnemônico para uma instrução específica consiste em letras que sugerem a operação a ser executada por essa instrução. Por exemplo, atribuímos os mnemônicos ADD, SUB e SHL, respectivamente, a instruções de adição, subtração e deslocamento de bits à esquerda (*SHift Left*).

Os indexadores dos operandos fonte e do destino são substituídos pelos nomes dos registradores que serão usados na operação, isto é, R0 e R1 como fontes e R2 como destino. Essa instrução assembly executa a operação ($R_2 \leftarrow R_0 + R_1$), isto é, a adição do conteúdo dos registradores R0 e R1 e armazena o resultado em R2.

Cada instrução é representada na sua linguagem assembly por um mnemônico e um formato específico, permitindo que o programador reconheça imediatamente a instrução, facilitando o processo de desenvolvimento e depuração de código.

Os processadores fornecem um conjunto de instruções (set de instruções) para permitir a execução de tarefas computacionais. Há um conjunto básico de operações elementares que a maioria dos processadores inclui entre suas instruções.

A maioria das instruções elementares podem ser classificadas em três categorias principais: (1) Instruções de transferência de dados, (2) instruções de processamento de dados e (3) instruções de controle do fluxo de execução do programa.

As instruções de transferência de dados causam a transferência de dados de um local para outro sem alterar o conteúdo das informações binárias. Elas movem dados entre os registradores e entre os registradores e a memória. As instruções de processamento de dados executam operações aritméticas (adição, subtração, multiplicação, divisão, comparação, ...), lógicas (AND, OR, NOT, ...) e de deslocamento de bits nos dados. As instruções de controle do fluxo de execução do programa fornecem a tomada de decisão, condicionalmente modificam o conteúdo do contador de programa, alterando o caminho percorrido pelo programa quando este é executado. Além do conjunto de instruções básicas, um computador pode ter outras instruções que fornecem operações especiais para aplicações específicas.

**Instruções de transferência de dados**

As instruções de transferência de dados movem os dados de um local no computador para outro, sem alterá-los. As transferências típicas ocorrem entre os registradores e a memória, entre os próprios registradores, e entre os registradores e as portas de entrada e saída do processador.

A Tabela 1 fornece uma lista de sete instruções típicas de transferência de dados usadas em muitos processadores. Cada instrução é acompanhada por seu símbolo mnemônico. Diferentes processadores, no entanto, podem usar outros mnemônicos para o mesmo nome de instrução.

**Tabela 1-Instruções típicas de transferência de dados**

| Instrução | Mnemônico |
|---|---|
| Load | LDR |
| Store | STR |
| Move | MOV |
| Push | PUSH |
| Pop | POP |
| Input | IN |
| Output | OUT |

A instrução *Load* é usada para designar uma transferência da memória para um registrador do processador. A instrução *Store* designa uma transferência de um registrador do processador para uma posição de memória. A instrução *Move* é usada para designar uma transferência de um registrador para outro.

As instruções *Push* e *Pop* transferem dados entre uma pilha de memória e um registrador. A operação de *Push* coloca um novo item no topo da pilha. A operação *Pop* remove um item da pilha. A pilha de memória é essencialmente uma parte de um espaço de endereços de memória acessado por um ponteiro que é sempre incrementado ou decrementado, antes ou depois do acesso à memória. O registrador que contém o ponteiro da pilha é chamado de *Ponteiro da Pilha* (*Stack Pointer - SP*) e seu valor sempre aponta para o item no topo da pilha.

Um ponteiro de pilha é carregado com um valor inicial (endereço inicial do topo da pilha). A partir de então, o *SP* é automaticamente decrementado ou incrementado a cada operação *Push* ou *Pop*. A vantagem de uma pilha de memória é que o processador pode fazer referência a ela sem precisar especificar um endereço na instrução, pois o endereço está sempre disponível e atualizado automaticamente no ponteiro da pilha.

O par final de instruções de transferência de dados, *Input* e *Output*, transferem dados entre registradores e dispositivos de entrada e saída. Essas instruções são semelhantes às instruções de *Load* e *Store*, exceto que as transferências são de e para registradores externos em vez de posições de memória.

Um computador possui várias portas de entrada e saída dedicadas à comunicação com o ambiente externo. Uma porta geralmente é um registrador com linhas de I/O conectadas aos terminais (pinos) físicos do dispositivo. A porta específica é escolhida por um endereço, de modo semelhante à maneira como um endereço seleciona uma palavra na memória.

As instruções de entrada e saída incluem um campo de endereço em seu formato para indicar a porta específica selecionada para a transferência de dados. Os endereços de porta são atribuídos de duas maneiras: No sistema de I/O independente, os intervalos de endereços atribuídos à memória e às portas de I/O são independentes um do outro. O computador possui instruções distintas de entrada e saída, conforme listado na Tabela 1, contendo um campo de endereço separado que é interpretado pela unidade de controle e usado para selecionar uma porta de I/O específica. O endereçamento de I/O independente isola a seleção de memória e I/O para que o intervalo de endereços da memória não seja afetado pela atribuição do endereço da porta. Por esse motivo, o método também é chamado de configuração de I/O isolada.

Ao contrário do sistema de I/O independente, o sistema de I/O mapeado na memória atribui um subintervalo dos endereços de memória para endereçar os registradores das portas de I/O. Não há endereços separados para lidar com transferências de entrada e saída, pois as portas de I/O são tratadas como locais de memória em um intervalo de endereços comum. Cada porta de I/O é considerada um local de memória, semelhante a uma palavra de memória.

Computadores que adotam esse esquema não possuem instruções de entrada ou saída distintas, porque as mesmas instruções são usadas para manipular memória e dados de I/O. Por exemplo, as instruções Load e Store usadas para transferência de memória também são usadas para transferência de I/O, desde que o endereço associado à instrução seja atribuído à uma porta de I/O e não a uma palavra comum de memória. A vantagem desse esquema é a simplicidade que resulta com o mesmo conjunto de instruções servindo para acesso à memória e I/O.

A principal vantagem de se utilizar I/O independente é em CPUs com a capacidade de endereçamento limitada. Como o sistema de I/O independente separa o acesso de I/O do acesso de memória, o espaço para endereçamento pode ser usado inteiramente para a memória. Além disso, se torna evidente para uma pessoa lendo em linguagem assembly (ou mesmo analisando o código de máquina), a ocorrência de operações de I/O, pela utilização das instruções específicas para este fim.

Há duas grandes vantagens de se utilizar I/O mapeado em memória. Uma delas é que, descartando a complexidade extra trazida pela porta de I/O, a CPU precisa de menos lógica interna e, portanto, é mais rápida, é mais fácil de construir, é mais barata, consome menos energia e pode ser fisicamente menor. Isto segue as diretrizes principais de uma máquina *RISC* e é também vantajoso em sistemas embarcados. A outra vantagem é que por serem utilizadas instruções normais de memória para endereçar dispositivos, todos os modos de endereçamento das instruções estarão disponíveis tanto para I/O quanto para a memória comum e as instruções que efetuam uma operação da ULA diretamente em um operando - carregando o operando da memória, armazenando o resultado na memória, ou ambos - podem também ser utilizadas com registradores de I/O.

À medida que processadores de 8 e 16 bits se tornaram obsoletos e foram substituídos por processadores de 32 e 64 bits, reservar grandes intervalos de endereços de memória para I/O não era mais um problema, já que o espaço para endereçamento de memória é muito maior do que o necessário por todos os dispositivos de I/O. Assim, frequentemente é muito mais prático utilizar os benefícios do mapeamento de I/O em memória.

**Instruções de processamento de dados**

As instruções de processamento de dados executam operações nos dados e fornecem os recursos computacionais do processador. Elas geralmente são divididas em três tipos básicos: (1) Instruções aritméticas, (2) Instruções lógicas e de manipulação de bits e (3) Instruções de deslocamento de bits.

As quatro instruções aritméticas básicas são adição, subtração, multiplicação e divisão. A maioria dos processadores fornece instruções para todas as quatro operações. Alguns processadores mais simples, entretanto, fornece instruções apenas para adição e subtração, sendo as demais obtidas pela utilização sequencial das primeiras. Por exemplo, uma multiplicação pode ser realizada por sucessivas somas, embora o processamento se torne mais lento. Uma lista de instruções aritméticas típicas é fornecida na Tabela 2.

**Tabela 2-Instruções aritméticas típicas**

| Instrução | Mnemônico |
|---|---|
| Incremento | INC |
| Decremento | DEC |
| Adição | ADD |
| Subtração | SUB |
| Multiplicação | MUL |
| Divisão | DIV |
| Adição com carry | ADDC |
| Subtração com borrow | SUBB |
| Subtração reversa | SUBR |
| Negação | NEG |

A instrução de *Incremento* adiciona um ao valor armazenado em um registrador ou palavra de memória. Uma característica comum dessa operação é que um número binário com todos os bits iguais a 1 produz um resultado com todos os bits iguais a zero quando incrementado. Nesse caso, dizemos que houve um transbordamento (overflow), ou estouro, na palavra binária. A instrução de *Decremento* subtrai um do valor armazenado em um registrador ou palavra de memória. Quando é decrementado, um número com todos os bits iguais a zero produz um número com todos os bits iguais a 1, o que chamamos de underflow.

As instruções de *Adição*, *Subtração*, *Multiplicação* e *Divisão* podem estar disponíveis para diferentes tipos de dados. O tipo de dado que se supõe estar no registrador do processador durante a execução dessas operações aritméticas está incluído na definição do código de operação. Uma instrução aritmética pode operar sobre números binários inteiros com e sem sinal, ou dados de ponto flutuante (números decimais). A representação em ponto flutuante é usada para cálculos científicos e é realizada por uma unidade de processamento independente (coprocessador), chamada unidade de ponto flutuante (*FPU – Floating Point Unit*), encontrada apenas em processadores com mais recursos computacionais.

O bit *Carry* (C) do registrador de status é usado para armazenar o "vai-um" de uma operação de adição onde o resultado não pode ser representado pela quantidade de bits de um registrador. A instrução *Adição com carry* realiza a adição com dois operandos mais o valor do "vai-um" do cálculo anterior.

Da mesma forma, a instrução *Subtração com borrow* subtrai dois operandos e um "vem um" que pode ter resultado de uma operação anterior.

A instrução de *Subtração reversa* inverte a ordem dos operandos, executando B-A em vez de A-B.

A instrução *Negação* executa o complemento de 2 de um número com sinal, o que equivale a multiplicar o número por -1.

Instruções lógicas e de manipulação de bits executam operações binárias em palavras armazenadas em registradores ou palavras de memória. Elas são úteis para manipular bits individuais ou um grupo de bits que representam informações codificadas em binário.

Instruções lógicas consideram cada bit do operando separadamente e os tratam como uma variável binária. Pela aplicação adequada das instruções lógicas, é possível alterar valores de bits, limpar um grupo de bits ou inserir novos valores de bits em operandos armazenados em registradores ou memória. Algumas instruções lógicas e de manipulação de bits estão listadas na Tabela 3.

**Tabela 3-Instruções lógicas**

| Instrução | Mnemônico |
|---|---|
| Clear | CLR |
| Set | SET |
| Complemento | NOT |
| E | AND |
| Ou | OR |
| Ou exclusivo | XOR |
| Limpa carry | CLRC |
| Seta carry | SETC |
| Complementa carry | COMC |

A instrução *Clear* faz com que todos, ou alguns, bits do operando específico sejam substituídos por zeros. A instrução *Set* faz com que todos, ou alguns, bits do operando específico sejam substituídos por 1. A instrução *Complemento* inverte todos os bits do operando.

As instruções *E* (*AND)*, *Ou (OR)* e *Ou exclusivo* (*XOR)* produzem as operações lógicas correspondentes, realizadas bit a bit, em dois operandos. Embora as instruções lógicas executem operações booleanas, quando usadas em palavras, elas geralmente são vistas como operações de manipulação de bits.

A instrução *E* é usada para limpar um bit ou um grupo selecionado de bits de um operando. Para qualquer variável booleana X, o relacionamento X AND 0 = 0. Da mesma forma, o relacionamento X AND 1 = X. Portanto, a instrução AND é usada para limpar seletivamente os bits de um operando a partir de uma palavra que tem 0's nas posições de bits que devem ser limpas e 1's nas posições de bits que devem permanecer iguais. A instrução AND também é chamada de mascaramento porque, ao inserir 0's, mascara uma parte selecionada de um operando.

A instrução OR é usada para setar um bit ou um grupo selecionado de bits de um operando. Para qualquer variável booleana X, o relacionamento X OR 1 = 1. Da mesma forma, o relacionamento X OR 0 = X. Portanto, a instrução OR pode ser usada para setar seletivamente bits de um operando a partir de uma palavra com 1's nas posições de bit que devem ser definidos como 1 e 0's nas demais posições.

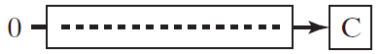
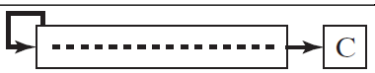
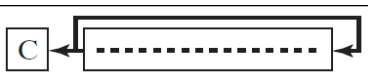
A instrução XOR é usada para complementar seletivamente os bits de um operando. Isso ocorre devido aos relacionamentos booleanos X AND 1 = X e X OR 0 = X. Uma variável binária é complementada quando é realizado uma operação XOR com 1, mas não altera seu valor quando a operação é realizada com zero. A instrução XOR às vezes é chamada de complemento de bit.

As outras instruções de manipulação de bits incluídas na Tabela 3 podem limpar, setar ou complementar (inverter o estado) o bit de transporte (*Carry*). Instruções adicionais podem operar sobre outros bits de status do processador de maneira semelhante.

Os processadores também fornecem instruções para alterar o conteúdo de um único operando. As operações de deslocamento de bits, por exemplo, são aquelas nas quais os bits de um operando são todos movidos para a esquerda ou para a direita. O bit recebido no final da palavra determina o tipo de deslocamento. Essas instruções podem especificar deslocamentos lógicos, deslocamentos aritméticos ou operações do tipo rotação de bits.

A Tabela 4, mostrada na página seguinte, lista quatro tipos de instruções de deslocamento de bits, nas versões à direita e à esquerda. Os pequenos diagramas apresentados na coluna da direita da tabela mostram como ocorre a movimentação dos bits para cada uma das instruções. Em todos os casos, o bit de saída é copiado no bit de status de transporte C (*carry*).

**Tabela 4-Instruções típicas de deslocamento de bits típicas**

| Instrução | Mnemônico | Diagrama |
|---|---|---|
| Deslocamento lógico à direita | SHR |  |
| Deslocamento lógico à esquerda | SHL |  |
| Deslocamento aritmético à direita | SHRA |  |
| Deslocamento aritmético à esquerda | SHLA |  |
| Rotação à direita | ROR |  |
| Rotação à esquerda | ROL |  |
| Rotação à direita com carry | RORC |  |
| Rotação à esquerda com carry | ROLC |  |

Os deslocamentos lógicos inserem 0 na posição do bit de entrada durante o deslocamento.

Os deslocamentos aritméticos estão em conformidade com as regras dos deslocamentos em complemento de 2 para números com sinal. O deslocamento aritmético à direita usa extensão de sinal, preenchendo a posição mais à esquerda com seu próprio valor durante a mudança. A instrução de deslocamento aritmético à esquerda insere 0 no bit recebido na posição mais à direita e é idêntica à instrução de deslocamento lógico à esquerda.

As instruções de rotação produzem um deslocamento circular. Os valores deslocados para fora do bit de saída são girados de volta para o bit de entrada.

As instruções de rotação com *carry* tratam o bit de transporte como uma extensão do registrador cuja palavra está sendo rotacionada. Assim, por exemplo, uma rotação para a esquerda com transporte transfere o bit de transporte para o bit recebido na posição mais à direita do registrador, transfere o bit de saída do bit mais à esquerda do registrador para o bit de transporte e desloca todo o registrador para a esquerda.

A maioria dos processadores possui um formato para a instrução de deslocamento que permite deslocar várias posições de bits, em vez de apenas uma. Um campo contém o código de operação e outro contém o número de posições que um operando deve ser deslocado. Com esse formato, é possível especificar o tipo de deslocamento, a direção do deslocamento e o número de posições a serem deslocadas, tudo em uma única instrução.

### Instruções de controle do fluxo de execução do programa

As instruções de um programa são armazenadas em locais sucessivos na memória. As instruções são lidas pela unidade de controle e executadas uma a uma, sucessivamente. Cada vez que uma instrução é buscada na memória, o PC é atualizado para que contenha o endereço da próxima instrução em sequência. Por outro lado, quando uma instrução de controle de fluxo de execução do programa é executada, pode ocorrer uma alteração do valor armazenado no PC, fazendo com que o fluxo de execução seja alterado.

A alteração no PC como resultado da execução de uma instrução de controle de fluxo do programa causa uma mudança na sequência de execução das instruções. Esse é um recurso importante dos computadores digitais, pois fornece controle sobre o fluxo de execução do programa e a capacidade de ramificação para diferentes segmentos do programa, dependendo dos resultados de processamentos anteriores. Essas instruções são equivalentes às declarações de controle em linguagens de alto nível como *if-else*, *do-while*, *switch-case*, *for*, *etc*, sendo as instruções resultantes da compilação dessas declarações.

Algumas instruções típicas de controle do fluxo de execução do programa estão listadas na Tabela 5.

**Tabela 5-Instruções típicas de controle do fluxo de execução do programa**

| Instrução | Mnemônico |
|---|---|
| Desvio (Branch) | BR |
| Salto (Jump) | JMP |
| Comparação (por subtração) | CMP |
| Teste (com AND) | TEST |
| Chamada (Call) | CALL |
| Retorno (Return) | RETURN |

As instruções de *Desvio* (*Branch*) e *Salto* (*Jump*) geralmente são usadas de forma intercambiável, embora algumas vezes sejam usadas para indicar diferentes modos de endereçamento da próxima instrução. Por exemplo, o *Salto* pode usar endereçamento direto ou indireto, enquanto o *Desvio* usa endereçamento relativo. O *Desvio*, ou *Salto*, geralmente é uma instrução que contém um campo de um endereço. Quando executada, a instrução de *Desvio* causa uma transferência do endereço efetivo para o PC. Como o PC contém o endereço da instrução a ser executada a seguir, a próxima instrução será buscada no local especificado pelo endereço efetivo.

As instruções de *Desvio* e *Salto* podem ser condicionais ou incondicionais. Uma instrução de *Desvio* incondicional causa um desvio no endereço efetivo especificado sem nenhuma condição. A instrução de *Desvio* condicional especifica uma condição que deve ser atendida para que o desvio ocorra, como a exigência do valor em um registrador especificado ser negativo, por exemplo. Se a condição for atendida, o PC será carregado com o endereço efetivo e a próxima instrução será buscada nesse endereço. Se a condição não for atendida, o PC não será alterado e a próxima instrução será buscada no próximo endereço em sequência. As instruções de *Desvio* ou *Salto* condicionais são usadas para alterar condicionalmente a sequência do programa.

A instrução de C*omparação* realiza uma comparação por meio de uma subtração, mas o resultado normalmente não é armazenado. Em vez disso, a *Comparação* pode causar um desvio condicional, alterar o conteúdo de um registrador ou alterar os bits de status do processador. Da mesma forma, a instrução de *Teste* executa um AND lógico de dois operandos sem reter o resultado e executa uma das ações listadas anteriormente.

As instruções de *Comparação* e *Teste* podem ser usadas de três modos distintos, dependendo da maneira como as decisões condicionais são tratadas: O primeiro modo executa toda a decisão como uma única instrução. Por exemplo, o conteúdo de dois registradores pode ser comparado e um *Desvio* ou *Salto* é executado se o conteúdo for igual. Como dois endereços de registradores e um endereço de memória estão envolvidos, essa instrução requer três endereços; O segundo modo também usa três endereços, sendo todos endereços de registradores. Considerando o mesmo exemplo, se o conteúdo dos dois primeiros registradores for igual, o valor 1 é armazenado no terceiro registrador, caso contrário, o valor 0 é armazenado. Esses dois modos de instrução evitam a alteração de bits de status do processador. No primeiro caso, esse bit não é necessário e, no segundo caso, um registrador é usado para simular sua presença; O terceiro modo redefine diretamente os bits de status do processador.

Uma instrução de *Desvio* condicional é uma instrução que pode ou não causar uma mudança no fluxo de execução do programa, dependendo do valor dos bits armazenados no registrador de status do processador. Cada instrução de *Desvio* condicional testa uma combinação diferente de bits de status. Se a condição for verdadeira, o controle é transferido para o endereço efetivo. Se a condição for falsa, o programa continua com a instrução seguinte.

A Tabela 6 fornece uma lista de instruções de *Desvio* condicional que dependem dos bits no registrador de status. Na maioria dos casos, a instrução mnemônica usa as letras BR (para "*branch*") e uma letra para o nome do bit de status que está sendo avaliado. A letra N (para "não") é incluída se o bit de status for testado para uma condição 0. Como exemplo, BRC é um desvio se *carry* = 1 e BRNC é um desvio se carry = 0. Nessas instruções, supõe-se que uma instrução anterior, como *Comparação* ou *Teste,* tenha atualizado os bits de status Z, C, N e V.

**Tabela 6-Instruções típicas de desvio condicional relacionadas aos bits de status**

| Instrução | Mnemônico | Condição testada |
|---|---|---|
| Desvie se zero | BRZ | Z=1 |
| Desvie se não zero | BRNZ | Z=0 |
| Desvie se carry | BRC | C=1 |
| Desvie se não carry | BRNC | C=0 |
| Desvie se negativo | BRN | N=1 |
| Desvie se não negativo | BRNN | N=0 |
| Desvie se overflow | BRV | V=1 |
| Desvie se não overflow | BRNV | V=0 |

Como afirmado anteriormente, a instrução de *Comparação* executa uma subtração de dois operandos, digamos, A - B. O resultado da operação não é transferido para um registrador de destino, mas os bits de status podem ser afetados. Os bits de status fornecem informações sobre a magnitude relativa entre A e B. Alguns processadores fornecem instruções de *Desvio* especiais que podem ser aplicadas após a execução de uma instrução de *Comparação*. As condições específicas a serem testadas dependem se os dois operandos são considerados como números com ou sem sinal.

A magnitude relativa entre dois números binários sem sinal A e B pode ser determinada subtraindo A - B e verificando os bits de status C e Z. As instruções de *Desvio* condicional para números sem sinal estão listadas na Tabela 7. Supõe-se que uma instrução anterior tenha atualizado os bits de status C e Z após uma subtração A-B ou alguma outra instrução semelhante. As palavras "acima", "abaixo" e "igual" são usadas para denotar a magnitude relativa entre dois números sem sinal. Por exemplo, se A está acima de B (A>B), o resultado da operação lógica OU entre os bits de status C e Z deve retornar zero.

**Tabela 7-Instruções típicas de desvio condicional para números sem sinal**

| Instrução | Mnemônico | Condição testada | Bits de status |
|---|---|---|---|
| Desvie se acima | BRA | A>B | C + Z = 0 |
| Desvie se acima ou igual | BRAE | A≥B | C=0 |
| Desvie se abaixo | BRB | A<B | C=1 |
| Desvie se abaixo ou igual | BRBE | A≤B | C + Z = 1 |
| Desvie se igual | BRE | A=B | Z=1 |
| Desvie se não igual | BRNE | A≠B | Z=0 |

As instruções de *Desvio* condicional para números com sinal estão listadas na Tabela 8. Novamente, assume-se que uma instrução anterior tenha atualizado os bits de status N, V e Z após uma subtração A - B. As palavras "maior", "menor" e "igual" são usadas para denotar a magnitude relativa entre dois números com sinal. Por exemplo, se A é maior que B, o resultado da operação lógica OU entre o bit de status Z e o resultado da operação lógica OU-EXCLUSIVO entre os bits de status N e V deve retornar zero.

**Tabela 8-Instruções típicas de desvio condicional para números com sinal**

| Instrução | Mnemônico | Condição testada | Bits de status |
|---|---|---|---|
| Desvie se maior | BRG | A>B | $(N \oplus V) + Z=0$ |
| Desvie se maior ou igual | BRGE | A≥B | $(N \oplus V) = 0$ |
| Desvie se menor | BRL | A<B | $(N \oplus V) = 1$ |
| Desvie se menor ou igual | BRLE | A≤B | $(N \oplus V) + Z = 1$ |
| Desvie se igual | BRE | A=B | Z=1 |
| Desvie se não igual | BRNE | A≠B | Z=0 |

As instruções de *Chamada* (*Call*) e *Retorno* (*Return*) são usadas na implementação de procedimentos (funções ou sub-rotinas). Um procedimento é uma sequência independente de instruções que executa uma determinada função computacional.

Durante a execução de um programa, um procedimento pode ser chamado para executar sua tarefa várias vezes em vários pontos do programa. Cada vez que o procedimento é chamado, um desvio é feito no início do procedimento para começar a executar seu conjunto de instruções. Após a execução do procedimento, é feito um desvio novamente para retornar ao programa que o chamou.

A instrução que transfere o controle para um procedimento (*Call*) é conhecida por diferentes nomes, incluindo chamada de procedimento, chamada de sub-rotina, salto para sub-rotina, desvio para sub-rotina e desvio com link. Vamos nos referir à rotina que contém a chamada de procedimento como o “procedimento de chamada” e a sub-rotina geralmente é chamada de “procedimento chamado”.

A instrução de *Chamada* de procedimento possui um campo de um endereço e executa duas operações: Primeiro, ela armazena o valor do PC, que é o endereço após a instrução de chamada, em um local temporário. Esse endereço é chamado de endereço de retorno (*link*) e a instrução correspondente é o ponto de continuação de execução normal do código no procedimento de chamada; Segundo, o endereço na instrução de chamada (o endereço da primeira instrução do procedimento chamado) é carregado no PC. Quando a próxima instrução é buscada, ela vem do procedimento que foi chamado.

A instrução final em todo procedimento deve ser uma instrução de *Retorno* (*Return*) ao procedimento de chamada. A instrução de *Retorno* pega o endereço de retorno (*link*) que foi armazenado pela instrução de *Chamada* e o coloca de volta no PC. Isso resulta em uma transferência da execução do programa de volta ao ponto de continuação no procedimento de chamada. Processadores diferentes usam locais temporários diferentes para armazenar o endereço de retorno. Alguns processadores armazenam em um local fixo na memória, alguns armazenam em um registrador específico e alguns armazenam em uma pilha de memória.

Além de armazenar o endereço de retorno, o programa também deve gerenciar adequadamente qualquer valor de parâmetro transferido para o procedimento chamado e os valores de resultados retornados ao procedimento de chamada, bem como valores temporários armazenados nos registradores exigidos pelo procedimento chamado ou pelo procedimento de chamada. O método usado por uma linguagem de programação ou compilador para garantir que os valores sejam gerenciados corretamente é conhecido como *convenção de chamada*. Ela especifica como os valores dos parâmetros são fornecidos ao procedimento chamado, como os valores dos resultados são retornados ao procedimento de chamada, quais registradores podem ser alterados e quais devem ser preservados pelo procedimento chamado para que seus valores possam ser usados pelo procedimento de chamada após o procedimento chamado retornar o controle para ele. Uma combinação de registradores e a pilha é frequentemente usada como parte da convenção de chamada para passar valores de parâmetros para o procedimento e retornar valores ao procedimento de chamada. A pilha também é usada para preservar valores de registradores durante uma chamada de procedimento.

## 2.3. Interrupções

Uma interrupção é um recurso de hardware encontrado nos microprocessadores e usado para lidar com uma variedade de situações que exigem uma saída da sequência normal do fluxo de execução do programa. Uma interrupção de programa transfere o controle de um programa que está sendo executado em um determinado momento para um procedimento de serviço como resultado de uma solicitação gerada externa ou internamente pelo hardware. O controle retorna ao programa original após a execução do procedimento de serviço.

Como o próprio nome sugere, uma interrupção serve para interromper imediatamente o fluxo regular de um programa, tomando atitudes instantâneas, quando um determinado evento acontece. Os eventos capazes de gerar uma interrupção são sinalizados para o hardware do processador através de pedidos de interrupção (*IRQs – Interrupt Requests*). O processamento da interrupção compõe uma troca de contexto para uma rotina de software especificamente escrita para tratar a interrupção. Essa rotina é chamada de rotina de serviço de interrupção (*ISR – Interrupt Service Routine*), rotina de atendimento à interrupção, ou tratador de interrupção (*interrupt handler*). Um processador pode ter várias rotinas de atendimento à diferentes interrupções. Os endereços iniciais dessas rotinas na memória são chamados de vetores de interrupção (*interrupt vector*) e são armazenados geralmente em uma tabela na memória chamada de tabela de vetores de interrupção (*interrupt vector table*).

As interrupções são ações tratadas diretamente pelo hardware, o que as tornam muito rápidas e disponíveis em qualquer ponto do programa. Assim sendo, quando uma interrupção acontece, o programa é interrompido, uma sequência específica de instruções (definida pelo programador) é executada e depois o programa continua a ser executado do mesmo ponto em que estava. Na Figura 8 é mostrado o fluxo de execuçao normal de um programa e o fluxo de execução do programa quando ocorre uma interrupção.

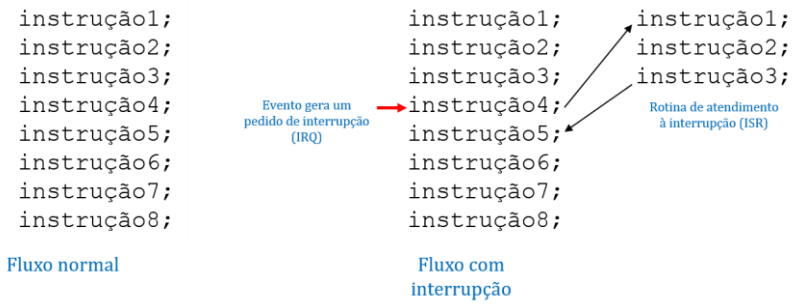



*Figura 8 – Fluxo normal de execução de um programa e fluxo com interrupção.*

Interrupções de hardware foram introduzidas como forma de evitar o desperdício de tempo computacional em rotinas de software à espera de eventos específicos (conhecidas como *polling loops*). Por exemplo, quando você pressiona um botão em um controle remoto de uma TV, o processador registrará e responderá imediatamente a essa ação mudando o canal, aumentando o volume, etc. Se o processador gastasse a maior parte do tempo verificando se cada um dos botões foi pressionado, isso não seria nada prático.

Em princípio, a rotina de atendimento a uma interrupção é semelhante a um procedimento utilizado com as instruções de *Chamada*, exceto em três aspectos:

1. A interrupção é geralmente iniciada em um ponto imprevisível do programa por um sinal externo ou interno, em vez da execução de uma instrução de *Chamada*.
2. O endereço da rotina de serviço que processa a solicitação de interrupção é pré-determinado pelo hardware, em vez do campo de endereço de uma instrução.
3. Em resposta a uma interrupção, é necessário armazenar informações que definam todo ou parte do conteúdo do conjunto de registradores no instante da requisição, em vez de armazenar apenas o contador de programa.

Após a interrupção do processador e a execução da rotina de serviço apropriado, o processador deve retornar exatamente ao mesmo estado em que estava antes da interrupção. Somente se isso acontecer, o programa interrompido poderá ser retomado, como se nada tivesse acontecido. O estado do processador no final da execução de uma instrução é determinado a partir do conteúdo do conjunto de registradores.

Além de conter os bits de status, o *PSR* pode especificar e habilitar quais interrupções podem ocorrer e se o processador está operando no modo de usuário ou sistema. A maioria dos processadores executa um sistema operacional que controla e supervisiona todos os outros programas. Quando o processador está executando um programa que faz parte do sistema operacional, ele é dito estar no modo sistema, no qual determinadas instruções são privilegiadas e podem ser executadas apenas nesse modo. O processador está no modo de usuário ao executar programas do usuário, no caso em que não pode executar as instruções privilegiadas. O modo do processador a qualquer momento é determinado a partir de um ou mais bits de status especiais no *PSR*.

Alguns processadores armazenam apenas o contador de programa ao responder a uma interrupção. Nessas máquinas, a rotina de atendimento à interrupção deve incluir instruções para armazenar o conteúdo essencial do conjunto de registradores. Outros processadores armazenam uma parte ou todo o conjunto de registradores automaticamente em resposta a uma interrupção. Alguns processadores possuem dois conjuntos de registradores, de modo que, quando o programa alterna do modo de usuário para o modo de sistema em resposta a uma interrupção, não é necessário armazenar o conteúdo dos registradores, porque cada modo utiliza seu próprio conjunto.

O procedimento de hardware para processar interrupções é muito semelhante à execução de uma instrução de *Chamada*: O conteúdo do conjunto de registradores é armazenado temporariamente, normalmente sendo empurrado para uma pilha de memória, e o endereço da primeira instrução da rotina de serviço da interrupção é carregado no PC. O endereço da rotina de serviço é definido pelo hardware. Alguns processadores atribuem um local único de memória ao endereço inicial da rotina de serviço: A rotina, então, deve determinar a fonte da interrupção, a partir da observação de bits de status, e prosseguir com o tratamento adequado para aquela fonte. Outros processadores atribuem um local de memória separado para cada fonte de interrupção possível. Às vezes, o próprio hardware da fonte de interrupção fornece o endereço da sua rotina de serviço. De qualquer forma, o processador deve possuir algum tipo de procedimento de hardware para selecionar um endereço específico para atender à interrupção.

Geralmente, uma rotina de atendimento à interrupção deve ser tão curta e rápida quanto possível. Se o programa usa várias fontes de interrupção, apenas uma pode ser atendida de cada vez. Quando várias fontes enviam pedidos de interrupção simultaneamente, as rotinas específicas de atendimento serão executadas, uma após a outra, segundo uma ordem que depende de suas prioridades, que normalmente são configuráveis.

A maioria dos processadores não responderá a uma interrupção até que a instrução que está sendo executada seja concluída. Então, pouco antes de ir buscar a próxima instrução, a unidade de controle verifica se há pedidos de interrupção. Se uma interrupção ocorreu, o controle passa para o ciclo de interrupção do hardware. Durante esse ciclo, o conteúdo de uma parte ou de todo o conjunto de registradores é armazenado na pilha, o endereço do desvio para a rotina de atendimento à interrupção específica é então transferido para o PC, e o controle busca a próxima instrução, que é o início da rotina de serviço daquela interrupção. A última instrução na rotina de serviço é uma instrução de *Retorno* da interrupção. Quando esse retorno é executado, é recuperado o endereço de retorno, que é transferido para o PC, bem como qualquer conteúdo armazenado do restante do conjunto de registradores, que é transferido de volta para os registradores apropriados.

Os três principais tipos de interrupções que causam desvio na execução normal de um programa são os seguintes:

1. Interrupções externas;
2. Interrupções internas;
3. Interrupções de software.

As interrupções externas são provenientes de dispositivos de entrada ou saída, de dispositivos de temporização, de circuitos que monitoram a fonte de alimentação ou de qualquer outra fonte externa ao processador. As condições que causam interrupções externas podem ser: Um dispositivo de entrada ou saída que solicita uma transferência de dados; um dispositivo externo que conclui uma transferência de dados; o tempo limite de um evento ou uma falha de energia iminente. Uma interrupção de tempo limite pode resultar de um programa que está em um loop sem fim e, portanto, excede sua alocação de tempo. Uma interrupção de falha de energia pode ter como rotina de serviço algumas instruções que transferem o conteúdo completo do conjunto de registradores para uma memória não volátil em alguns milissegundos antes que a energia cesse completamente.

As interrupções internas surgem do uso incorreto de uma instrução ou de dados. Interrupções internas também são chamadas de *traps* (ou exceções). Exemplos de interrupções causadas por condições internas são um estouro aritmético, uma tentativa de divisão por zero, um código de operação inválido em uma instrução, um estouro de pilha de memória ou uma violação de proteção. Uma violação de proteção é uma tentativa de acessar uma área da memória que não deveria ser acessada pelo programa atualmente em execução. As rotinas de serviço que processam interrupções internas determinam a medida corretiva a ser tomada em cada caso.

Interrupções externas e internas são iniciadas pelo hardware do processador. Por outro lado, uma interrupção de software é iniciada executando uma instrução. A interrupção do software é uma instrução especial de *Chamada* que se comporta como uma interrupção em vez de uma chamada de procedimento. Ela pode ser usada pelo programador para iniciar um procedimento de interrupção em qualquer ponto desejado no programa. O uso típico da interrupção de software está associado a uma instrução de *Chamada* do sistema.

Um termo alternativo para uma interrupção é uma exceção, que pode ser aplicada apenas a interrupções internas ou a todas as interrupções, dependendo do fabricante do processador. Dessa forma, o que um programador chama de rotina de tratamento de interrupção pode ser referido como rotina de tratamento de exceção por outro programador.

## 2.4. Linguagens de programação

Todas as linguagens de programação de computadores podem ser classificadas em três grandes categorias: Linguagem de Máquina, Linguagem Assembly e Linguagem de Alto Nível. Embora os computadores possam ser programados em muitas linguagens diferentes, existe apenas uma linguagem que é entendida pelo computador sem usar algum recurso de tradução. Essa linguagem é chamada de linguagem de máquina do computador, categorizada como uma linguagem de baixo nível.

Um computador usa dígitos binários para sua operação e consegue processar apenas as informações compostas por zeros e uns. Portanto, as instruções são codificadas e armazenadas na memória na forma de 1's e 0's. Como dito no tópico 2.2, um programa escrito dessa forma é chamado de linguagem de máquina.

O primeiro passo na evolução das linguagens de programação foi o desenvolvimento do que é conhecido como linguagem assembly. Nessa linguagem, os mnemônicos são usados para representar códigos de operação e cadeias de caracteres representam os endereços dos operandos. Como uma linguagem assembly é projetada principalmente para substituir cada código de máquina por um mnemônico compreensível e cada endereço por uma sequência alfanumérica simples, ela é dependente e corresponde à arquitetura de um computador específico.

Para executar um programa de linguagem assembly em um computador, ele deve primeiro ser traduzido para seu programa equivalente em linguagem de máquina. Isso é necessário porque o circuito do computador foi projetado para executar apenas os códigos de operação da máquina. Os mnemônicos devem ser convertidos em códigos binários absolutos para sua correta operação. Os endereços simbólicos dos operandos usados na linguagem assembly também devem ser convertidos em endereços binários absolutos. O tradutor que faz isso é conhecido como *assembler,* ou montador. A entrada para um *assembler* é o programa escrito em linguagem assembly, conhecido como o programa fonte. Sua saída é um programa de linguagem de máquina equivalente e é conhecido como programa objeto. O *assembler* é um programa de sistema fornecido pelo fabricante do computador.

As linguagens de programação que se destinam a serem independentes da máquina são chamadas de linguagens de alto nível. Exemplos dessas linguagens incluem Python, Java, C/C++/C#, FORTRAN, COBOL, BASIC, ALGOL, etc. As instruções escritas nessas linguagens são conhecidas como declarações, em vez de mnemônicos, e se aproximam bastante da nossa linguagem formal. Em geral, ao escrever um programa em uma linguagem de alto nível, o conhecimento da arquitetura do computador não é necessário. Programas escritos nessas linguagens podem ser facilmente usados para qualquer tipo de computador. Assim, dizemos que os programas escritos em uma linguagem de alto nível são portáveis. A escrita do programa é mais fácil e rápida, porque uma declaração de uma linguagem de alto nível corresponde a muitas instruções da linguagem assembly.

É necessário um programa especial, chamado compilador, para traduzir a linguagem de alto nível em códigos de máquina para a operação de um computador. Um compilador é muito mais poderoso e complexo que um *assembler*. Os programas em linguagem assembly, por outro lado, são compactos, exigem menos espaço em memória e são mais eficientes do que os programas escritos em linguagem de alto nível.

Na Figura 9 é mostrada a hierarquia das linguagens de programação e como elas se relacionam no processo de desenvolvimento de programas para computadores. ■

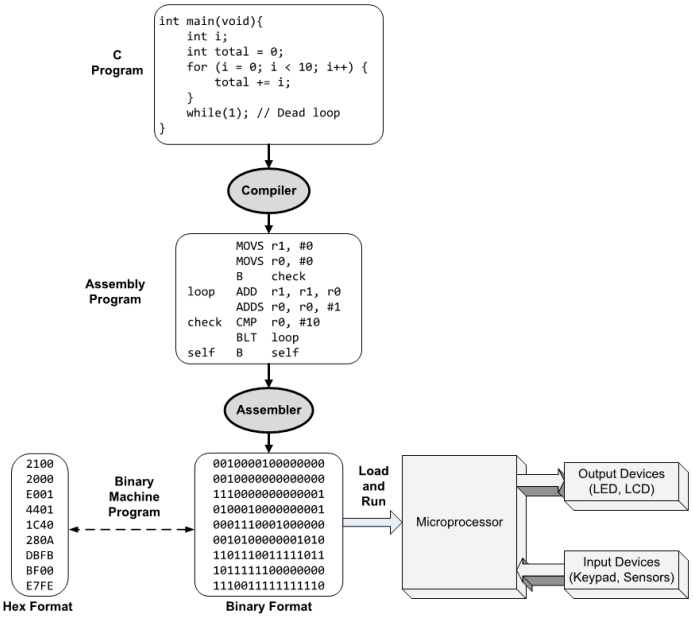



*Figura 9 – Hierarquia das linguagens de programação.*