Capítulo 4

Conjunto de Instruções

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Listar as principais características de um conjunto de instrução
- Definir e elencar os prós e contras das arquiteturas RISC e CISC

Neste capítulo vamos estudar o que é o Conjunto de Instruções e sua relevância no projeto de um processador. Sua características repercutem em todas principais características do processador, se tornando a principal e primeira decisão de projeto a ser tomada.

4.1 Introdução

O termo **Conjunto de Instruções** vem do inglês *Instruction Set Architecture* (ISA). ISA é a interface entre os softwares que serão executados pelo processador e o próprio processador. Ele define todas instruções de máquina serão interpretadas pela Unidade de Controle e executadas. Podemos então definir Conjunto de Instruções como sendo a coleção completa de todas instruções reconhecidas e executadas pela CPU. Esse Conjunto de instruções, também chamado de *Código de Máquina*, **é o ponto inicial para o projeto de uma arquitetura** e é essencial na definição de qualidade do sistema como um todo.

4.2 O projeto de um Conjunto de Instruções

4.2.1 Arquitetura

Um Conjunto de Instruções pode ser classificado como uma das quatro arquiteturas:

- Arquitetura de Pilha
- Baseada em Acumulador
- Registrador-Registrador ou Load/Store
- Registrador-Memória

4.2.1.1 Arquitetura de Pilha

A Arquitetura de Pilha[1] é a mais simples possível. Como pode ser observado na Figura 4.1 [46], os dados necessários para a execução das operações pela ULA são provenientes de registradores especiais organizados na forma de uma pilha. Note que toda operação é realizada entre o registrador que indica o Topo de Pilha (apontado pelo registrador TOS) e o registrador seguinte. *Esse conjunto de instruções é muito simples porque a Unidade de Controle nunca precisa decodificar a instrução para saber quais registradores serão utilizados* nas operações lógicas e aritméticas. Sempre será o topo da pilha e o registrador seguinte.

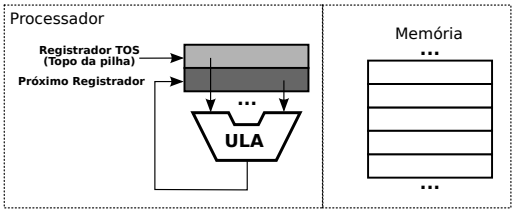

Figura 4.1: ISA baseado em Pilha

4.2.1.2 Arquitetura baseada em Acumulador

Já na arquitetura baseada em Acumulador (Figura 4.2 [46]) uma complexidade é adicionada. Um dos dados vem sempre do registrador Acumulador, mas o outro é mais livre. Na figura esse segundo dado vem da memória, mas poderia vir de um outro registrador designado na instrução. Neste caso, a instrução a ser decodificada pela Unidade de Controle precisará trazer de onde vem o segundo dado a ser utilizado, já que o primeiro é sempre proveniente do Acumulador.

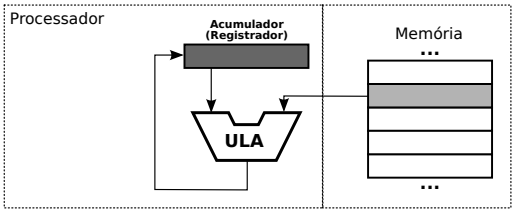

Figura 4.2: ISA baseado em Acumulador

4.2.1.3 Arquitetura Load/Store

Na sequência, a terceira arquitetura mais complexa é chamada Arquitetura Load/Store ou Arquitetura Registrador-Registrador (Figura 4.3 [47]). Nesta arquitetura, todas operações lógicas e aritméticas executadas pela ULA são provenientes de dois registradores a serem determinados pela instrução. A única forma de acessar dados da memória é através de duas instruções especiais: **LOAD**, para ler da memória e **STORE** para escrever o conteúdo de um registrador na memória. Assim, toda instrução a ser decodificada pela CPU deverá indicar o endereço de dois registradores que serão utilizados

[1] O funcionamento da estrutura Pilha é aprofundado na disciplina **Estrutura de Dados**.

na operação lógica ou aritmética, ou um endereço de memória se a instrução for um LOAD ou um STORE.

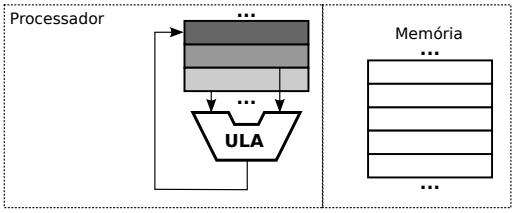

Figura 4.3: ISA Registrador-Registrador

4.2.1.4 Arquitetura Registrador-Memória

Por último, a mais complexa arquitetura é a Registrador-Memória (Figura 4.4 [47]). Esta arquitetura permite que a ULA execute operações lógicas e aritméticas envolvendo ao mesmo tempo um registrador indicado pela instrução e um conteúdo proveniente da memória. Esse tipo de instrução deve então trazer, em seu conteúdo, o código do registrador a ser utilizado e o endereço de memória do segundo dado.

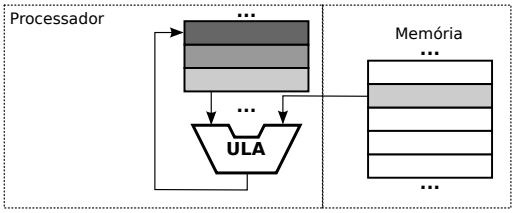

Figura 4.4: ISA Registrador-Memória

É muito importante aqui observar que arquiteturas mais simples, como a de Pilha por exemplo, fazem com as instruções a serem decodificadas pela Unidade de Controle sejam muito simples. Com instruções simples, a própria Unidade de Controle será simples, pequena e barata de produzir. Por outro lado, instruções simples resultam em operações limitadas. Assim, uma tarefa básica, como somar dois números, em uma Arquitetura de Pilha vai necessitar que pelo menos 4 operações sejam realizadas: colocar na pilha o primeiro dado, colocar na pilha o segundo dado, somar os dois elementos da pilha, salvar o resultado na pilha. Essa mesma operação numa arquitetura Registrador-Memória seria executada numa instrução única, indicando o registrador utilizado para ser o primeiro dado, o endereço de memória do segundo dado e o registrador que vai guardar o resultado.

4.3 Aspectos do Projeto do Conjunto de Instruções

A escolha da arquitetura é uma questão importantíssima de projeto e geralmente baseia-se na relação entre o desempenho que se quer atingir e o preço do processador ao final.

Após definida a arquitetura do Conjunto de Instruções, o projetista deve cuidar do projeto das instruções propriamente ditas. Neste projeto cinco pontos devem ser contemplados:

- Modelo de Memória
- Tipos de Dados
- Formato das Instruções
- Tipo de Instruções
- Modo de Endereçamento

4.3.1 Modelo de Memória

O modelo de memória define para cada instrução de onde vem e para onde vão os dados. Os dados podem vir da memória principal, ou memória Cache, de um registrador, ou de um dispositivo de Entrada e Saída. Além disso, o projeto do ISA deve definir alguns pontos cruciais.

4.3.1.1 Memória alinhada x não alinhada

As memórias geralmente são organizadas em bytes. Isso significa que o byte é a menor unidade de acesso à memória, e assim, cada endereço de memória acessa um byte. A Tabela 4.1 [48] apresenta um exemplo de organização de memória por bytes. Neste caso, o endereço 0 diz respeito ao Byte 0 da primeira linha. Já o endereço 1 ao Byte 1, o endereço 2 ao Byte 2 e o endereço 3 ao Byte 3. Enquanto que os endereços de 4 a 7 já dizem respeitos aos bytes da segunda linha.

Tabela 4.1: Exemplo de uma memória organizada por bytes

Endereço	Byte 0	Byte 1	Byte 2	Byte 3
0				
4	13	23		
8				
12				
16				
. . .				

A memória é organizada desta forma, mas há diferentes tipos de dados. Uma variável inteira, por exemplo, pode ser declarada como um byte, ou um inteiro curto de 2 bytes, um inteiro de 4 bytes, ou até mesmo um inteiro longo de 8 bytes. Se o conjunto de instruções fixar que a memória será apenas *acessada de forma alinhada*, significa que um dado não pode ultrapassar uma linha. Assim, os dados de 4 bytes deverão obrigatoriamente ser armazenados nos endereços 0, 4, 8, 12, 16 etc.

Esta decisão visa facilitar e acelerar o trabalho do processador ao acessar a memória. Cada vez que a memória precisar ser acessada para buscar um número de 4 bytes, o processador deve apenas verificar se o endereço é um múltiplo de 4. Se ele não for, o acesso é negado e o programa é encerrado. A desvantagem desta abordagem é que muitas áreas podem ficar desperdiçadas. Por exemplo, se um dado de 2 bytes é armazenado no endereço 4 (como mostra a Tabela 4.1 [48]), os Bytes 0 e 1 são utilizados, mas os Bytes 2 e ficam disponíveis. Mesmo assim, se um dado de 4 bytes precisar ser armazenado, ele não poderá ser feito em uma posição livre múltipla de 4 (0, 8, 12, 6 etc.), como por

exemplo, salvando os dois primeiros bytes nos endereços 6 e 7 (segunda linha) e os outros 2 bytes nos endereços 8 e 9 (terceira linha).

Já se o conjunto de instruções permitir o acesso não alinhado à memória, todas essas restrições se acabam e os dados podem ser acessados em quaisquer posições. O uso da memória termina tendo um maior aproveitamento, já que menos áreas livres existirão. O problema agora é com o desempenho. Todo processador acessa a memória através de um barramento. Para otimizar o acesso, os barramentos geralmente são largos o suficiente para trazer uma linha inteira de memória, ou mais. Com a memória alinhada fica mais fácil, porque para buscar um dado a CPU só precisa saber em que linha ele está, daí é só trazê-la para a CPU.

Já na memória não alinhada, quando um dado é buscado a CPU precisa saber em que linha ele está, em que posição exatamente começa, se ele invade a próxima linha, e onde termina. Esta operação é mais uma responsabilidade para a CPU e torna o processo de leitura mais lento.

Processadores da arquitetura Intel, por exemplo, utilizam memórias alinhadas, porque percebeu-se que o ganho com o desperdício de memória não compensa o tempo gasto buscando dados em linhas diferentes.

4.3.1.2 Memória para dados e instruções

Outra decisão importante com relação ao acesso à memória é *se a faixa de endereços da memória de dados será a mesma da memória de instruções*. Desde a criação das Arquiteturas Harvard (em substituição às de Von Neumann) os dados passaram a serem armazenados em áreas de memória separadas das instruções. Isso ajuda principalmente no Pipeline, porque a CPU pode, no mesmo ciclo, buscar uma instrução e buscar os dados de uma outra instrução.

Na definição do Conjunto de Instruções o projetista deve decidir como serão os endereços de dessas áreas de memória. Por exemplo, se uma memória tiver 2 milhões de bytes (2MB), como cada byte possui um endereço, ela terá endereços de 0 a 1.999.999. Digamos que a primeira parte de memória seja para os dados e a segunda para as instruções. Neste caso, haverá duas opções. A memória pode ser organizada como uma faixa única de endereços, então os endereços de 0 a 999.999 serão utilizados para armazenar os dados, e os endereços de 1.000.000 a 1.999.999 armazenarão as instruções.

O problema desta abordagem, seguindo o mesmo exemplo, é que neste sistema só poderá haver 1 milhão de dados 1 milhão de instruções. Não será possível o armazenamento de nenhuma instrução a mais do que isso, mesmo que a área de dados esteja com posições disponíveis.

Como solução, o Conjunto de Instruções pode definir instruções especiais para buscar dados e instruções para buscar instruções. Quando uma instrução de busca for decodificada a Unidade de Controle saberá que se trata de um dado ou um endereço e vai buscar a informação exatamente na memória especificada. No exemplo dado, as instruções ficariam armazenadas na memória de instrução nos endereços de entre 0 e 1.999.999, e os dados na memória de dados também no endereço de 0 e 1.999.999. Mas observe que a memória do exemplo só possui 2 milhões de bytes (2MB), mesmo assim, neste exemplo, o sistema seria capaz de endereçar 4 milhões de bytes (4MB). Isso a tornaria pronta para um aumento futuro do espaço de memória.

4.3.1.3 Ordem dos bytes

No início do desenvolvimento dos computadores, os engenheiros projetistas tiveram que tomar uma decisão simples, mas muito importante. Quando temos dados que ocupam mais de um byte devemos começar salvando os bytes mais significativos, ou os menos significativos? Esta decisão não afeta em nada o desempenho ou o custo dos sistemas. É simplesmente uma decisão que precisa ser tomada.

Aqueles projetos que adotaram a abordagem de salvar os dados a partir dos bytes mais significativos foram chamados de Big Endian, enquanto que aqueles que adotaram iniciar pelo byte menos significativo foram chamados de Little Endian.

Imagine que uma instrução pede para que a palavra UFPB seja salva no endereço 8 da memória. Uma palavra pode ser vista como um vetor de caracteres, onde cada caracter ocupa 1 byte. Na Tabela 4.2 [50] é apresentada a abordagem Big Endian. Neste caso, o byte mais significativo (o mais a esquerda) é o que representa a letra U. Desta forma, ele é armazenado no Byte mais a esquerda, seguido pela letra F, a letra P e a letra B.

Tabela 4.2: Exemplo de uma memória Big Endian

Endereço	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**
0				
4				
8	U	F	P	B
12				
16				
. . .				

Já na abordagem Little Endian mostrada na Tabela 4.3 [50] a mesma palavra é armazenada iniciando pelo byte menos significativo, da direita para a esquerda. Apesar de parecer estranho, note que o que muda é a localização dos Bytes 0, 1, 2 e 3. Comparando as abordagens Big Endian e Little Endian no exemplo, em ambos a letra U é armazenada no Byte 0, a F no Byte 1, a P no Byte 2 e a B no Byte 3. A diferença é que no Little Endian os bytes são contados da direita para a esquerda.

Tabela 4.3: Exemplo de uma memória Little Endian

Endereço	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0				
4				
8	B	P	F	U
12				
16				
. . .				

Importante

O importante é observar que se um computador Little Endian precisar trocar dados com um computador Big Endian, eles precisarão antes ser convertidos para evitar problemas. A palavra UFPB salva num computador Big Endian, por exemplo, se for transmitida para um Little Endian, deve antes ser convertida para BPFU para evitar incompatibilidade.

Nota
Computadores Intel, AMD e outras arquiteturas mais populares utilizam o modelo Little Endian, enquanto que a arquitetura da Internet (TCP/IP) e máquinas IBM adotam o Big Endian.

4.3.1.4 Acesso aos registradores

A forma de acesso aos registradores também é essencial para a definição das instruções de máquina. Cada endereço é referenciado através de um código, como se fosse um endereço, assim como nas memórias. *O Conjunto de Instruções deve definir quantos endereços de registradores serão possíveis e o tamanho deles, e as políticas de acesso*. Essas decisões são de extrema importância para o projeto do Conjunto de Instruções e do processador como um todo.

4.3.2 Tipos de Dados

Quando escrevemos programas em linguagens de programação somos habituados a utilizar diversos tipos de dados, como inteiros, caracteres, pontos flutuantes e endereços. O que muitos esquecem, ou não sabem, é esses tipos são definidos pelo processador, no momento do projeto do Conjunto de Instruções. Os tipos mais comuns de dados são:

- Inteiros
- Decimais
- Pontos flutuantes
- Caracteres
- Dados lógicos

Cada um destes dados pode ainda possuir diversas variantes, como por exemplo:

- Inteiros e Pontos flutuantes: com e sem sinal, curto, médio ou grande
- Caracteres: ASCII, Unicode ou outras
- Dados lógicos: booleanos ou mapas de bits

O Conjunto de Instruções de máquina pode adotar todos os tipos e variedades, ou um subconjunto delas. Se um processador não utilizar um tipo de dado, o compilador deverá oferecer uma alternativa ao programador e, no momento de compilação, fazer a relação entre o tipo utilizado na linguagem de programação e o tipo existente na linguagem de máquina.

Sempre que for determinado que um tipo de dado será utilizado pelo processador, é também necessário que se criem instruções para manipular esses dados. Por exemplo, se for definido que a arquitetura vai suportar números de ponto flutuante de 32 bits, será necessário também criar instruções para executar operações aritméticas com esses números, criar registradores para armazena-los e barramentos para transporta-los. É uma decisão que impacto em todo o projeto do processador.

4.3.3 Formato das Instruções

Em seguida, é preciso também definir quais serão os formatos de instrução aceitos pela Unidade de Controle. No geral, toda instrução de máquina deve ter pelo menos o código da operação (ou *Opcode*) e os endereços para os parâmetros necessários, que podem ser registradores, posições de memória ou endereços de dispositivos de Entrada e Saída.

Nesta definição de formatos é necessário que se defina quantos endereços cada instrução poderá trazer. Para ilustrar, suponha que na linguagem de alto nível a operação `A=B+C` seja escrita e precise ser compilada. Se o Conjunto de Instruções adotar instruções com 3 endereços, fica simples. O código da operação de soma é `ADD` e com 3 endereços ela ficaria algo semelhante a:

```
ADD A, B, C
```

Onde `A`, `B` e `C` são endereços que podem ser de memória, registrador ou dispositivo de Entrada e Saída. Isso não vem ao caso neste momento.

Mas se o Conjunto de Instruções suportar apenas 2 endereços, ele pode adotar que o resultado sempre será salvo no endereço do primeiro parâmetro, sendo assim, a instrução teria que ser compilada da seguinte forma:

```
MOV B, R
ADD R, C
MOV R, A
```

A instrução `MOV` teve que ser adicionada para copiar o conteúdo de `B` para `R` (um registrador qualquer). Em seguida a instrução `ADD R, C` é executa, que soma o conteúdo de `R` com `C` e salva o resultado em `R`. No final, a instrução `MOV` é chamada novamente para salvar o resultado de `R` em `A` e finalizar a instrução. Note que todas instruções neste exemplo utilizaram no máximo 2 endereços.

Já se a arquitetura utilizar instruções de apenas 1 endereço, será necessário utilizar uma arquitetura baseada em Acumulador e toda operação será entre o Acumulador e um endereço, e o resultado será também salvo no acumulador. Assim, a instrução seria compilada como:

```
ZER ACC
ADD B
ADD C
STO A
```

Aqui, quatro instruções foram necessárias. A primeira zeta o conteúdo do Acumulador (`ACC`). A segunda soma o Acumulador com o conteúdo apontado pelo endereço `B` e salva o resultado no Acumulador. A terceira soma o conteúdo do Acumulador com `C` e salva no Acumulador e, por fim, a última instrução transfere o resultado do Acumulador para o endereço de `A`. Note aqui também que todas instruções não passaram de um endereço para os parâmetros.

A última opção seria não utilizar endereços em suas instruções principais. Isso é possível se for utilizada uma Arquitetura Baseada em Pilha. Neste caso, duas instruções são adicionadas: `POP` e `PUSH`. A instrução `PUSH` adiciona um valor ao topo da pilha, enquanto que `POP` remove um elemento do topo da pilha e adiciona seu conteúdo em um endereço. Dessa forma, a instrução seria compilada assim:

```
POP B
POP C
ADD
PUSH A
```

Neste caso a primeira instrução colocou o conteúdo do endereço de `B` na pilha, a segunda instrução fez o mesmo com `C`. A terceira instrução é a única que não precisa de endereço. Ela faz a soma com os dois últimos valores adicionados à pilha (`B` e `C`, no caso) e salva o resultado de volta na pilha. A instrução `PUSH` vai remover o último valor da pilha (resultado da soma) e salvar na variável endereçada por `A`.

É possível perceber que quanto menos endereços, mais simples são as instruções. Isso é bom porque a Unidade de Controle não precisará de muito processamento para executá-las. Por outro lado, o programa compilado se torna maior e termina consumindo mais ciclos. A decisão se as instruções serão mais simples ou mais complexas é muito importantes e vamos tratar dela mais a frente neste capítulo.

Geralmente as arquiteturas adotam abordagem mistas. Por exemplo, aquelas que suportam instruções com 2 endereços geralmente também suportam instruções com 1 endereço e com nenhum. Isso também é bastante interessante, porque torna o processador bastante versátil. Por outro lado, aumenta a complexidade da Unidade de Controle. Cada instrução que for executada precisa antes ser classificada e depois despachada para uma unidade de execução específica.

4.3.4 Tipos de Instruções

Além de definir como serão as instruções, é necessário também definir que tipos de instruções serão executadas pelo processador. Os principais tipos de instrução são:

Movimentação de dados
: Como os dados serão transferidos interna e externamente ao processador.

Aritméticas
: Que operações serão realizadas, como exponenciais, trigonométricas, cálculos com pontos flutuantes, com e sem sinais, com números curtos e longos etc.

Lógicas
: AND, OR, NOT e XOR.

Conversão
: De caracteres para números, de inteiros para reais, decimal para binário etc.

Entrada e Saída
: Haverá instruções específicas ou haverá um controlador específico, como um DMA (Direct Memory Access).

Controle
: Instruções para controlar os dispositivos diversos do computador.

Transferência de Controle
: Como a execução deixará um programa para passar para outro, para interromper um programa, chamar subprogramas, instruções de desvio e de interrupção.

4.3.5 Modos de Endereçamento

Por último, no projeto de um Conjunto de Instruções é necessário determinar de que forma os endereços das instruções serão acessados. Ou seja, o que cada endereço de parâmetros de uma instrução podem representar.

Os principais modos de endereçamento são:

- **Imediato**
- **Direto** e **Direto por Registrador**
- **Indireto** e **Indireto por Registrador**
- **Indexado** e **Indexado por Registrador**

4.3.5.1 Endereçamento Imediato

O endereçamento Imediato é o mais simples possível e indica que o endereço na verdade é uma constante e pode ser utilizada imediatamente.

Por exemplo, na instrução a seguir:

```
ADD A, 5, 7
```

Significa que a variável `A` deve receber o conteúdo da soma de 5 com 7. O endereços 5 e 7 são considerados Imediatos, já que não representam um endereço, e sim uma constante.

4.3.5.2 Endereçamento Direto

No endereçamento Direto o endereço representa um endereço de memória, como mostrado na instrução a seguir.

```
ADD A, B, 7
```

Neste exemplo, o valor 7 continua sendo endereçamento de forma imediata, mas os endereços de `A` e `B` são endereços de memória e por isso, são chamados de Endereços Diretos.

4.3.5.3 Endereçamento Direto por Registrador

No endereçamento Direto por Registrador os endereços representam registradores, e poderíamos ver o exemplo citado da seguinte forma:

```
ADD R1, R2, C
```

Neste exemplo `R1` e `R2` são códigos para registradores e o endereçamento é chamado Direto por Registrador, enquanto que `C` é acessado através de Endereçamento Direto.

4.3.5.4 Endereçamento Indireto

Já o endereçamento Indireto é aplicado quando é necessário que um acesso Direto seja feito antes para buscar o endereço alvo e só então o acesso é feito. O exemplo a seguir mostra um caso onde este endereçamento é utilizado.

```
ADD A, (B), 7
```

Neste exemplo o endereço `B` entre parêntesis representa o acesso indireto. Isso indica que primeiramente o endereço de `B` deve ser acessado, mas lá não está o conteúdo a ser somado com 7, mas o endereço de memória onde o valor deverá ser encontrado. Este tipo de endereçamento é muito utilizado nas linguagens de programação para representar variáveis dinâmicas através de apontadores (ou ponteiros).

4.3.5.5 Endereçamento Indireto por Registrador

O endereçamento Indireto também pode ser feito por Registrador. No exemplo a seguir o valor sete não é somado ao conteúdo de `R1`, mas ao dado que está na memória no endereço apontado por `R1`.

```
ADD A, (R1), 7
```

4.3.5.6 Endereçamento Indexado

Outro endereçamento possível é o Indexado. Neste modo de endereçamento é necessário indicar o endereço do dado e um valor chamado Deslocamento. No exemplo a seguir, o endereçamento Indexado é aplicado para `B[5]`.

```
ADD A, B[5], 7
```

Isso indica que o dado a ser utilizado está no endereço `B` de memória adicionado de 5 posições. Ou seja, se a variável `B` estiver no endereço 1002 de memória, o valor `B[5]` estará no endereço 1007. O endereço de `B` é chamado de Endereço Base e o valor 5 é chamado de Deslocamento. Para realizar um endereçamento Indexado é necessário um somados que não seja a ULA para agilizar o processamento e realizar cada deslocamento. Este tipo de endereçamento é utilizado quando são utilizadas instruções de acesso a vetores.

4.3.5.7 Endereçamento Indexado por Registrador

Há também a possibilidade de realizar o endereçamento Indexado por Registrador, que utiliza um registrador ao invés da memória para armazenar o Endereço Base, como exibido a seguir:

```
ADD A, R1[5], 7
```

4.4 RISC x CISC

O projeto do Conjunto de Instruções inicia com a escolha de uma entre duas abordagens, a abordagem RISC e a CISC. O termo RISC é a abreviação de **Reduced Instruction Set Computer**, ou Computador de Conjunto de Instruções Reduzido e CISC vem de **Complex Instruction Set Computer**, ou Computador de Conjunto de Instruções Complexo. Um computador RISC parte do pressuposto de que um conjunto simples de instruções vai resultar numa Unidade de Controle *simples, barata e rápida*. Já os computadores CISC visam criar arquiteturas complexas o bastante a ponto de *facilitar a construção dos compiladores, assim, programas complexos são compilados em programas de máquina mais curtos*. Com programas mais curtos, *os computadores CISC precisariam acessar menos a memória para buscar instruções e seriam mais rápidos*.

A Tabela 4.4 [56] resume as principais características dos computadores RISC em comparação com os CISC. Os processadores RISC geralmente adotam arquiteturas mais simples e que acessam menos a memória, em favor do acesso aos registradores. A arquitetura Registrador-Registrador é mais adotada, enquanto que os computadores CISC utilizam arquiteturas Registrador-Memória.

Tabela 4.4: Arquiteturas RISC x CISC

Características	RISC	CISC
Arquitetura	Registrador-Registrador	Registrador-Memória
Tipos de Dados	Pouca variedade	Muito variada
Formato das Instruções	Instruções poucos endereços	Instruções com muitos endereços
Modo de Endereçamento	Pouca variedade	Muita variedade
Estágios de Pipeline	Entre 4 e 10	Entre 20 e 30
Acesso aos dados	Via registradores	Via memória

Como as arquiteturas RISC visam Unidades de Controle mais simples, rápidas e baratas, elas geralmente optam por instruções mais simples possível, com pouca variedade e com poucos endereços. A pouca variedade dos tipos de instrução e dos modos de endereçamento, além de demandar uma Unidade de Controle mais simples, também traz outro importante benefício, que é a *previsibilidade*. Como as intrusões variam pouco de uma para outra, é mais fácil para a Unidade de Controle prever quantos ciclos serão necessários para executa-las. Esta previsibilidade traz benefícios diretos para o ganho de desempenho com o *Pipeline*. Ao saber quantos ciclos serão necessários para executar um estágio de uma instrução, a Unidade de Controle saberá exatamente quando será possível iniciar o estágio de uma próxima instrução.

Já as arquiteturas CISC investem em Unidades de Controle poderosas e capazes de executar tarefas complexas como a Execução Fora de Ordem e a Execução Superescalar. Na execução Fora de Ordem, a Unidade de Controle analisa uma sequência de instruções ao mesmo tempo. Muitas vezes há dependências entre uma instrução e a seguinte, impossibilitando que elas sejam executadas em Pipeline. Assim, a Unidade de Controle busca outras instruções para serem executadas que não são as próximas da sequência e que não sejam dependentes das instruções atualmente executadas. Isso faz com que um programa não seja executado na mesma ordem em que foi compilado. A Execução Superescalar é a organização do processador em diversas unidades de execução, como Unidades de Pontos Flutuante e Unidades de Inteiros. Essas unidades trabalham simultaneamente. Enquanto uma instrução é executada por uma das unidades de inteiros, outra pode ser executada por uma das unidades de Pontos Flutuantes. Com a execução Fora de Ordem junto com a Superescalar, instruções que não estão na sequência definida podem ser executadas para evitar que as unidades de execução fiquem ociosas.

Nota
É importante ressaltar que a execução fora de ordem não afeta o resultado da aplicação pois foram projetadas para respeitar as dependências entre os resultados das operações.

Estas características de complexidade tornam os estágios de Pipeline dos processadores CISC mais longos, em torno de 20 a 30 estágios. Isto porque estas abordagens de aceleração de execução devem ser adicionadas no processo de execução. Já os processadores RISC trabalham com estágios mais curtos, em torno de 4 a 10 estágios.

Os processadores CISC também utilizam mais memória principal e Cache, enquanto que os processadores RISC utilizam mais registradores. Isso porque os processadores CISC trabalham com um maior volume de instruções e dados simultaneamente. Esses dados não poderiam ser armazenados em registradores, devido à sua elevada quantidade e são, geralmente, armazenados em memória Cache. Enquanto que os processadores RISC trabalham com menos instruções e dados por vez, o que possibilita a utilização predominante de registradores.

4.4.1 Afinal, qual a melhor abordagem?

Sempre que este assunto é apresentado aos alunos, surge a pergunta crucial sobre qual é a melhor abordagem, a RISC ou a CISC? Esta é uma pergunta difícil e sem resposta definitiva. A melhor resposta que acho é de que depende do uso que se quer fazer do processador.

Processadores RISC geralmente resultam em projetos menores, mais baratos e que consumem menos energia. Isso torna-os muito interessante para dispositivos móveis e computadores portáteis mais simples. Já os processadores CISC trabalham com clock muito elevado, são mais caros e mais poderosos no que diz respeito a desempenho. Entretanto, eles são maiores e consomem mais energia, o que os torna mais indicados para computadores de mesa e notebooks mais poderosos, além de servidores e computadores profissionais.

Os processadores CISC iniciaram com processadores mais simples e depois foram incorporando mais funcionalidades. Os fabricantes, como a Intel e a AMD, precisavam sempre criar novos projetos mas mantendo a compatibilidade com as gerações anteriores. Ou seja, o Conjunto de Instruções executado pelo 486 precisa também ser executado pelo Pentium para os programas continuassem compatíveis. O Pentium IV precisou se manter compatível ao Pentium e o Duo Core é compatível com o Pentium IV. Isso tornou o projeto dos processadores da Intel e AMD muito complexos, mas não pouco eficientes. Os computadores líderes mundiais em competições de desempenho computacional utilizam processadores CISC.

Já o foco dos processadores RISC está na simplicidade e previsibilidade. Além do benefício da previsibilidade do tempo de execução ao Pipeline, ele também é muito interessante para aplicações industriais. Algumas dessas aplicações são chamadas de Aplicações de Tempo Real. Essas aplicações possuem como seu requisito principal o tempo para realizar as tarefas. Assim, o Sistema Operacional precisa saber com quantos milissegundos um programa será executado. Isso só é possível com processadores RISC, com poucos estágios de Pipeline, poucos tipos de instrução, execução em ordem etc. Mesmo que os processadores RISC sejam mais lentos do que os CISC, eles são mais utilizados nessas aplicações críticas e de tempo real, como aplicações industriais, de automação e robótica.

4.5 Recapitulando

Este capítulo apresentou uma etapa de extrema importância no projeto de um processador, que é a definição do Conjunto de Instruções. O Conjunto de Instruções define não apenas como os programas serão compilados, mas também características críticas e de mercado, como tamanho, consumo de energia e preço.

Feedback sobre o capítulo
Você pode contribuir para melhoria dos nossos livros. Encontrou algum erro? Gostaria de submeter uma sugestão ou crítica?
Para compreender melhor como feedbacks funcionam consulte o guia do curso.