CAPÍTULO 3 – NÍVEL ISA

3.1 Introdução ao Nível de Arquitetura do Conjunto de Instruções

- O Nível de Arquitetura do Conjunto de Instruções (ISA - *Instruction Set Architecture*) é a interface entre *software* e *hardware*.
- É o nível mais próximo ao *hardware* em que o usuário pode programar.

- Problema: programar neste nível não é muito amigável.
- Solução: organização em níveis,

  - Cada nível da organização possui uma linguagem associada.
  - A medida que a organização evolui em direção ao usuário, a linguagem se torna mais conveniente para ele.
  - Usuário escreve um programa fonte em linguagem de alto nível ou em linguagem de montagem, mais amigáveis.
  - Programa fonte é traduzido para programa Objeto numa linguagem intermediária (Linguagem de Máquina);
  - As instruções de máquina são interpretadas pelo *Hardware*.

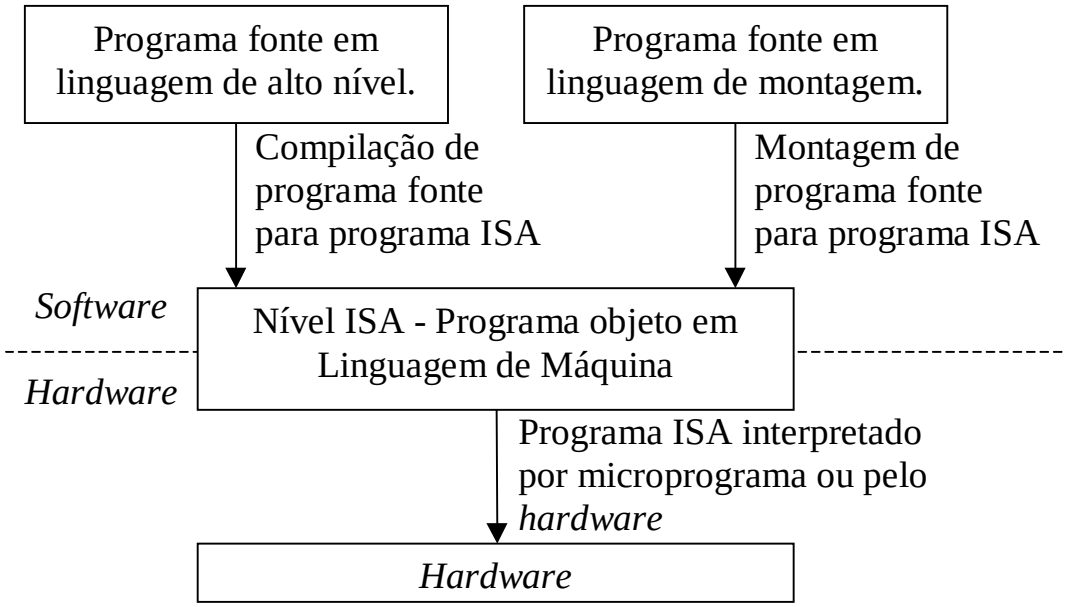


Figura 2.1. Nível ISA - interface *software* / *hardware*.

- Comparação entre as linguagens de cada nível:

Linguagem de alto nível:	A=5;
Linguagem de montagem:	MOVE A, #5;
Linguagem de máquina:	0011001100000101

Observações:

- A linguagem de montagem e a linguagem de máquina possuem uma relação de uma para um, ou seja, cada instrução de montagem possui uma instrução de máquina equivalente;
- A diferença entre as linguagens de montagem e de máquina é que a primeira é uma representação simbólica da segunda, que é puramente numérica;
- A linguagem de alto nível utiliza instruções que operam sobre estruturas de dados complexas. A compilação de uma seqüência de instruções em linguagem de alto nível geralmente gera uma seqüência maior de instruções ISA que operam sobre estruturas de dados mais simples e diversas daquelas presentes no programa fonte.

Características de projeto do nível ISA:

- No nível de linguagem de máquina está definida a interface entre *Software* e *Hardware*.
- Deve ser o mais simples possível, para facilitar projeto do *hardware*.
- Deve facilitar a geração de código por parte do compilador.

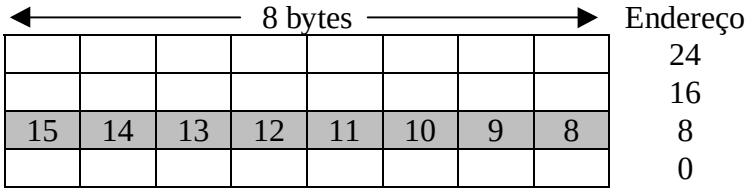
- Projeto do nível de linguagem de máquina deve dar suporte aos níveis superiores, possibilitando o uso de estruturas de dados tais como procedimentos, variáveis locais, variáveis globais, constantes, etc., utilizados pelas linguagens de alto nível;

Aspectos importantes no projeto desse nível:

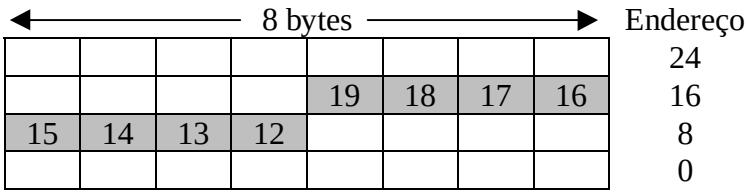
- Modelo da memória;
- Conjunto de registradores;
- Formato de instruções;
- Modos de endereçamento;
- Tipos de instruções;
- Fluxo de controle;

3.2 Modelos de Memória

- Antes de ser executado o programa em linguagem de máquina é armazenado na memória principal;
- Memória dividida em células referenciadas por endereços consecutivos;
- Célula pode ter qualquer tamanho, mas um byte é padrão atualmente;
- Os bytes são agrupados em palavras;
- Espaço de endereçamento: endereços começam em 0 e vão até um valor máximo Duas alternativas adotadas:
  - Espaço de endereçamento único (mais comum). Exemplo: o espaço de endereçamento [0 → (2^{64}-1)] requer 64 bits para especificar um endereço;
  - Espaço de endereçamento separado para instruções e dados: busca de instrução no endereço 1000 e busca de dado no endereço 1000 não acessam o mesmo endereço. Exemplo: o espaço [0 → (2^{64}-1)], dividido em [0 → (2^{32}-1)] para dados e [0 → (2^{32}-1)] para instruções, requer 32 bits para especificar um endereço.
- Alinhamento de palavras: muitas arquiteturas exigem alinhamento das palavras nos endereços corretos, de acordo com o seu tamanho.
  - Exemplo: palavras de oito bytes alinhadas só podem começar nos endereços 0, 8, 16, etc.
  - CPU's costumam transferir os bytes de uma palavra de uma só vez. Freqüentemente, a arquitetura utiliza apenas endereços múltiplos do tamanho da palavra. Neste caso, dados desalinhados teriam que ser buscados em duas referências à memória e os dados teriam de ser reconstruídos a partir das duas palavras lidas.



(a) Palavra de 8 bytes alinhada



(b) Palavra de 8 bytes não alinhada

Figura 3.2. Alinhamento de Palavra. a) Alinhada. b) Não alinhada.

Organização da memória no Pentium II:

- Espaço de endereçamento constituído por 16 k segmentos, cada segmento com endereços de 0 a (2^{32} - 1).
- Maioria dos sistemas operacionais só suporta um segmento com espaço de endereçamento linear de 0 a (2^{32} - 1).
- Cada byte tem seu próprio endereço.
- Palavras de 32 bits, *little-endian*.

3.3 Registradores

- O nível ISA disponibiliza um conjunto de registradores visíveis ao programador.
- Registradores de propósito geral: utilizados para armazenar resultados intermediários e variáveis locais, agilizando o acesso a esses dados;
- Registradores de propósito específico:
  - PC (contador de programa): ponteiro que armazena o endereço da próxima instrução a ser executada.
  - SP (apontador de pilha): aponta para o endereço no topo da pilha.
  - Registrador de *Status* (PSW - *Program Status Word*): contém bits necessários à operação do processador. Exemplo: habilitação de interrupções, modo de execução da máquina, códigos de condição, etc.
    - Códigos de condição são modificados a cada ciclo de máquina e indicam o estado do resultado do processamento da ULA. Exemplos:
      - N - Igual a 1 quando o resultado da ULA for negativo.
      - Z - Igual a 1 quando o resultado da ULA for zero.
      - V - Igual a 1 quando o resultado da ULA gerou um *oVerflow*.
      - C - Igual a 1 quando o resultado da ULA gerou um Vai-um (*carry*) para o bit mais à esquerda.
      - A - Igual a 1 quando o bit 3 do resultado da ULA gerou um *carry* Auxiliar.
      - P - Igual a 1 quando o resultado da ULA apresentar paridade par.
    - Códigos de condição são utilizados por instruções de comparação e desvio condicional.

Registradores do Pentium II

- Registradores de propósito geral, 32 bits, (compatibilidade com 16 e 8 bits): EAX (aritmético), EBX (ponteiros), ECX (*loops*), EDX (multiplicação e divisão, junto com EAX).
- Registradores de propósito geral, 32 bits: ESI e EDI (ponteiros para manipulação de *strings*), EBP (ponteiro para base de quadro de pilha).
- Registrador de propósito específico, 32 bits: ESP (apontador de pilha).
- Registrador de segmento, 32 bits: CS, SS, DS, ES, FS, GS. (compatibilidade com versões anteriores).
- Registrador de 32 bits: EIP (contador de programa).
- Registrador de 32 bits: EFLAGS (*status*),

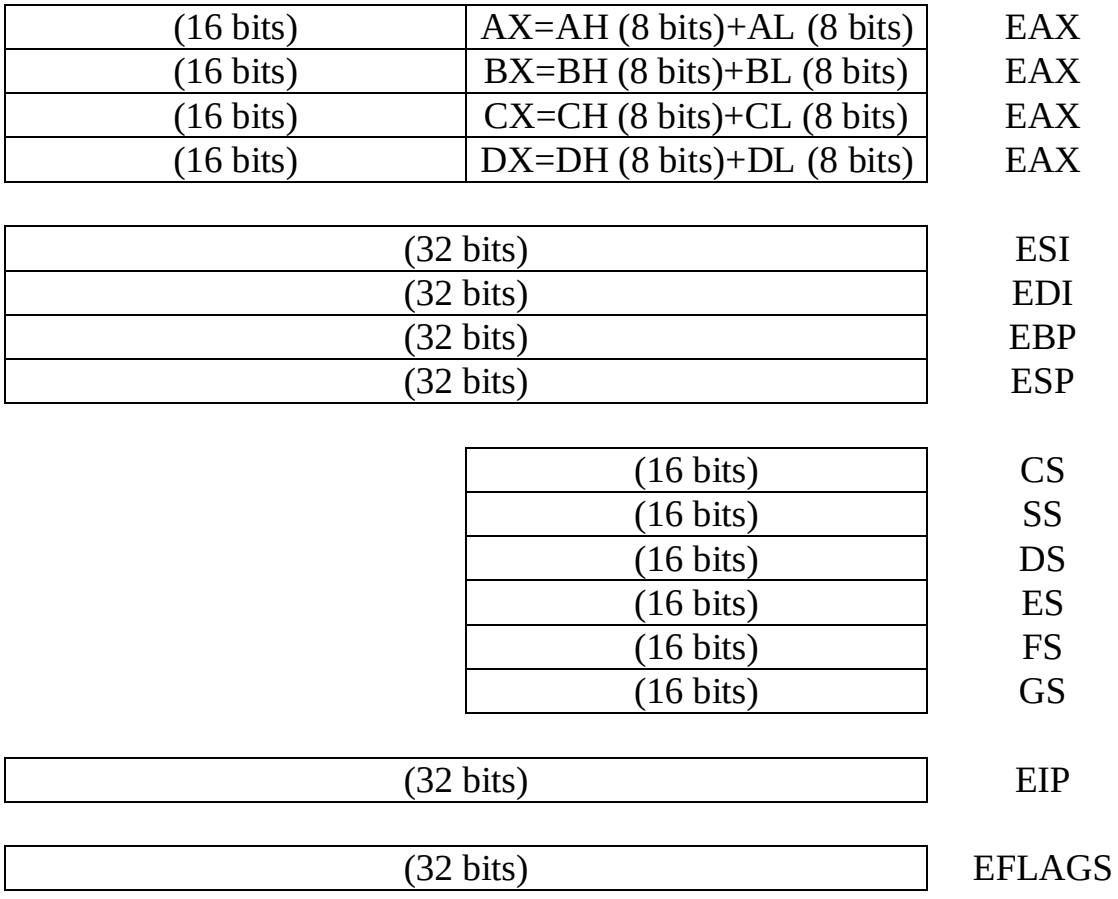
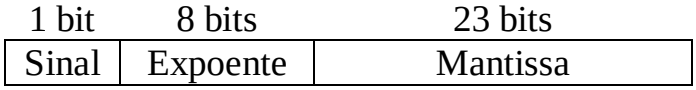


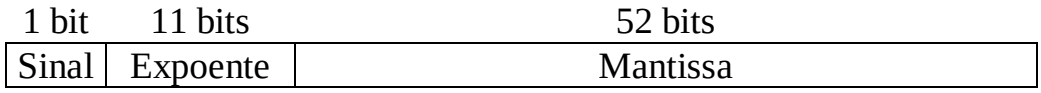
Figura 3.3. Registradores do Pentium II.

3.4 Tipos de Dados

- Dados Numéricos:
  - Números inteiros: 8, 16, 32, e 64 bits, com sinal (em complemento de dois) ou sem sinal.
  - Números decimais BCD: 4 bits por dígito (*hardware* não muito eficiente).
  - Números em ponto flutuante: 32 e 64 bits.



(a)



(b)

Figura 3.4. Padrão IEEE 754. a) Precisão simples. b) Precisão dupla.

Item	Precisão Simples	Precisão Dupla
Bits no campo de sinal	1	1
Bits no campo do expoente	8	11
Bits no campo da mantissa	23	52
Número total de bits	32	64
Sistema de representação do expoente	Excesso de 127	Excesso de 1023
Faixa de variação do expoente	-126 a 127	-1022 a 1023
Menor número normalizado	2^{-126}	2^{-1022}
Maior número normalizado	aprox. 2^{128}	aprox. 2^{1024}
Faixa de variação decimal	aprox. de 10^{-38} a 10^{38}	aprox. de 10^{-308} a 10^{308}
Menor número não normalizado	aprox. 10^{-45}	aprox. 10^{-324}

Figura 3.5. Características de números em ponto flutuante.
Padrão IEEE 754.

Normalizado	±	0 < Exp < Max	Qualquer combinação de bits
Não normalizado	±	0	Qualquer combinação de bits ≠ 0
Zero	±	0	0
Infinito	±	111111 ... 1	0
NaN	±	111111 ... 1	Qualquer combinação de bits ≠ 0

Figura 3.6. Tipos numéricos IEEE 754.

- Dados Não Numéricos:
  - *Strings* de caracteres ASCII.
  - Valores booleanos.
  - Mapas de bits (matriz de valores booleanos).
  - Ponteiros (endereços de máquina).

- Tipos de dados no Pentium II:
  - Números inteiros em complemento de dois.
  - Números inteiros sem sinal.
  - Números decimais codificados em binário (BCD).
  - Números em ponto flutuante padrão IEEE 754.
  - *Strings* de caracteres ASCII de comprimento conhecido.
  - *Strings* de caracteres ASCII de comprimento desconhecido (final da *string* marcado por caractere especial).

Tipo	8 bits	16 bits	32 bits	64 bits
Inteiro com sinal	x	x	x	
Inteiro sem sinal	x	x	x	
Inteiro BCD	x			
Ponto flutuante			x	x

Figura 3.7. Tipos de dados numéricos no Pentium II.

3.5 Formato das Instruções

- Dois campos essenciais de uma instrução:
  - Código de operação (OP CODE): identifica a operação a ser realizada pelo processador.
    - Deve identificar de forma única cada ação a ser executada pelo processador;
    - Pode ser de tamanho fixo ou variável.
  - Endereço: indica a localização do dado (operando) a ser manipulado pela instrução.
    - Em geral indica um endereço de memória ou de um registrador onde está contido o dado, ou onde ele será armazenado;
    - Cada instrução pode possuir 0, 1, 2 ou mais campos de endereço.

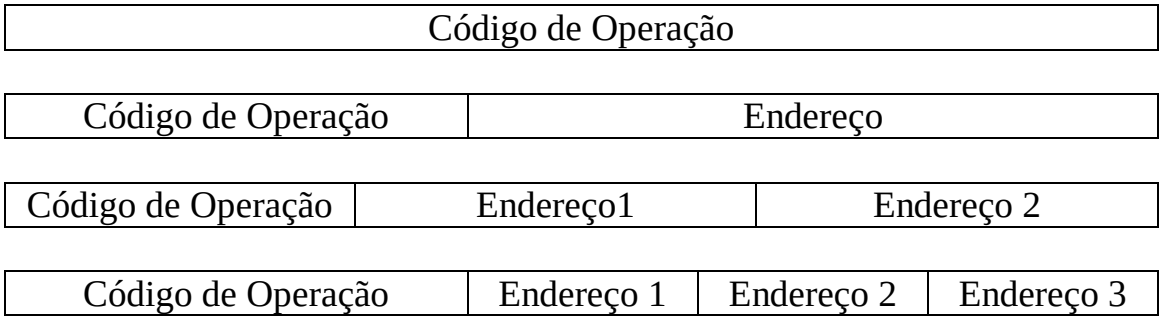


Figura 3.8. Formatos de instruções típicos.

- Na prática os formatos de instrução são bem mais complexos:
  - O conjunto de instruções pode possuir formatos diferentes;
  - O comprimento das instruções também pode ser variável, geralmente múltiplo ou submúltiplo do tamanho da palavra:
    - instruções menores que uma palavra;
    - instruções maiores que uma palavra;
    - instruções do tamanho de uma palavra.

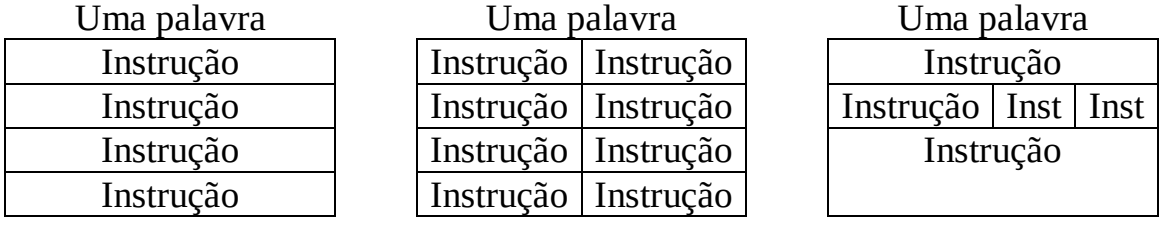


Figura 3.9 Várias relações entre tamanho de instrução e de palavra.

- Critérios de projeto de formatos de instrução:

- Tamanho da instrução:

  - Vantagens das instruções curtas:
    - Quanto maior o tamanho da instrução, maior é o espaço ocupado por um programa na memória.
    - N instruções de 32 bits ocupam o dobro da memória ocupada por N instruções de 16 bits.
  - Desempenho:
    - Banda passante da memória: t bps.
    - Tamanho médio das instruções: r bits.
    - Memória pode buscar no máximo t/r instruções por segundo.
    - Velocidade que o processador pode executar as instruções é limitada pelo tamanho das instruções.
    - Quanto menor o tamanho das instruções mais rápido serão buscadas pelo processador.
  - Desvantagens das instruções curtas:
    - São mais difíceis de decodificar.

- Tamanho do código de operação:

  - Deve possuir espaço suficiente para identificar todas as operações da máquina.
  - Ex.: 2^n instruções precisam de pelo menos n bits no campo de código de operação.

- Numero de bits no campo de endereço:
  - Tamanho maior: maior resolução no acesso à memória; espaço de endereçamento maior.
  - Tamanho menor: menor espaço necessário para o armazenamento das instruções.

- Expansão do código de operação:

  - Número de bits do campo código de operação depende do número de instruções que a máquina possui.
  - Ex.: Um conjunto com $256 = 2^8$ instruções precisa de pelo menos 8 bits para representar o código de qualquer uma das instruções da máquina.
  - O código de operação pode ser de tamanho fixo ou variável.
  - Códigos de operação de tamanho fixo são mais fáceis de implementar e manipular.
  - Código de operação com tamanho variável é muitas vezes útil;
  - Exemplo:
    - Máquina com instruções de 16 bits;
    - Endereços são representados por 4 bits;
    - 15 instruções de 3 endereços;
    - 14 instruções de 2 endereços;
    - 31 instruções de 1 endereço;
    - 16 instruções sem endereço.

  - As 15 instruções de 3 endereços são identificadas pelos quatro bits mais significativos (b_{15}, b_{14}, b_{13}, b_{12}) com valor entre 0000 e 1110.

  - Quando o código de escape (b_{15}, b_{14}, b_{13}, b_{12}) = 1111 aparece no início da instrução, os 4 bits (b_{11}, b_{10}, b_9, b_8) com valores entre 0000 e 1101 podem ser usados para especificar as 14 instruções de 2 endereços.

  - Quando os oito bits mais significativos assumem o código de escape 1111 1110 ou 1111 1111, os próximos 4 bits podem ser usados para especificar 31 instruções de 1 endereço.

  - Finalmente quando todos os primeiros 12 bits forem 1, os últimos 4 bits podem ser usados para especificar 16 instruções sem endereço.

15 Instruções de 3 endereços

Cod Operação	End1	End2	End3
0000	Xxxx	yyyy	zzzz
0001	Xxxx	yyyy	zzzz
⋮	⋮	⋮	⋮
1110	Xxxx	yyyy	zzzz

14 Instruções de 2 endereços

Cod Operação		End1	End2
1111	0000	yyyy	zzzz
1111	0001	yyyy	zzzz
⋮	⋮	⋮	⋮
1111	1101	yyyy	zzzz

31 Instruções de 1 endereço

Cod Operação			End1
1111	1110	0000	zzzz
1111	1110	0001	zzzz
⋮	⋮	⋮	⋮
1111	1110	1111	zzzz
1111	1111	0000	zzzz
1111	1111	0001	zzzz
⋮	⋮	⋮	⋮
1111	1111	1110	zzzz

16 Instruções sem endereço

Cod Operação			
1111	1111	1111	0000
1111	1111	1111	0001
⋮	⋮	⋮	⋮
1111	1111	1111	1111

Figura 3.10. Expansão do código de operação.

3.6 Modos de Endereçamento

Características da Especificação do Endereçamento:

- As instruções em geral utilizam:
  - pequena quantidade de bits parar o código de operação.
  - grande quantidade para especificar endereços dos dados.
  - Exemplo: instrução de adição:
    - Precisa dos endereços dos dois valores a serem somados.
    - Precisa do endereço do local onde o valor deve ser armazenado.
    - Supondo que os endereços de memória sejam de 32 bits: são necessários 96 bits de endereços.
    - Os endereços dos operandos na memória principal devem ser determinados em tempo de compilação.
  - 1ª Solução: utilizar registradores para armazenar os operandos.
    - Máquina com 32 registradores, endereços de registrador de 5 bits: 15 bits de endereços, para o caso acima citado;
    - Acesso mais rápido.
    - Problema: precisa de instruções adicionais que utilizariam endereços de 32 bits para carregar os operandos da memória para os registradores.
    - Uso repetido de uma variável compensa o problema.
  - 2ª Solução: indicar um ou mais endereços de forma implícita.
    - Exemplo: instrução só precisa indicar o ENDEREÇO_1, quando o código de operação é decodificado já se sabe que o registrador R1 deve ser utilizado.

      R1:=R1+ENDEREÇO_1

    - Uma pilha também pode ser usada para que nenhum endereço seja especificado.

- Existem outros métodos de codificar o campo de endereços. Estes podem utilizar mais ou menos bits e o endereço pode ser determinado estaticamente, em tempo de compilação, ou dinamicamente, em tempo de execução.
- O método utilizado para codificar o campo de endereço de uma instrução é denominado **Modo de Endereçamento**. Indica a forma pela qual o campo de endereço é interpretado para buscar o dado.

- para exemplificar os modos de endereçamento existentes considere o seguinte formato de instrução:

COD OPERAÇÃO	**ENDEREÇO**

- e a seguinte instrução neste formato:

ADD	**FONTE**

- A instrução acima soma o valor endereçado por FONTE a um registrador chamado ACUMULADOR e armazena a soma no próprio ACUMULADOR, ou seja,

ACUMULADOR:=ACUMULADOR+FONTE;

- Endereçamento Imediato: o campo de endereço contém o próprio dado.
  - No ciclo de busca-decodificação-execução, a memória só é acessada para buscar a instrução.
  - O dado é obtido imediatamente quando a instrução é buscada.
  - O dado fica limitado ao tamanho do campo de endereço da instrução.
  - É utilizado para passar constantes de valor pequeno.
  - Exemplo:

ADD	4

ACUMULADOR:=ACUMULADOR+4;

  - Se o valor inicial do acumulador for 16, o seu valor será modificado para 16 + 4 = 20, após a execução da instrução acima.

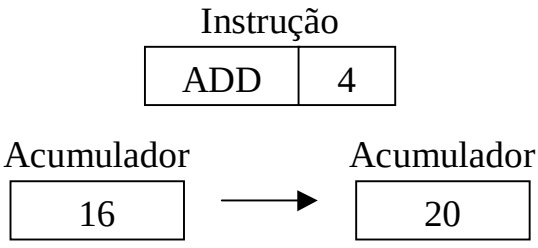


Figura 3.11. Exemplo de endereçamento imediato.

- Endereçamento Direto: o campo de endereço contém o endereço de memória onde está armazenado o dado.
  - É necessária uma referência extra à memória para buscar o dado, além daquela feita para buscar a instrução.
  - Dado não fica limitado ao tamanho do campo de endereço.
  - Utilizado para implementar variáveis globais.

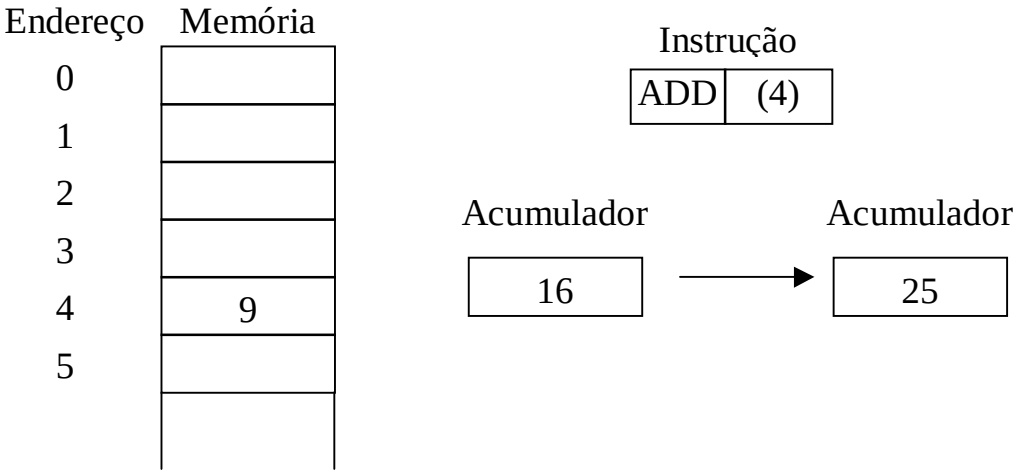


Figura 3.12. Endereçamento Direto.

- Endereçamento Via Registrador: o endereço especificado no campo de endereço é de um registrador onde está contido o dado.
  - Utiliza um endereço de registrador, que é menor que um endereço de memória principal;
  - Acesso aos registradores é mais rápido.
  - Número de registradores é limitado.
  - É utilizado para acessar variáveis locais.

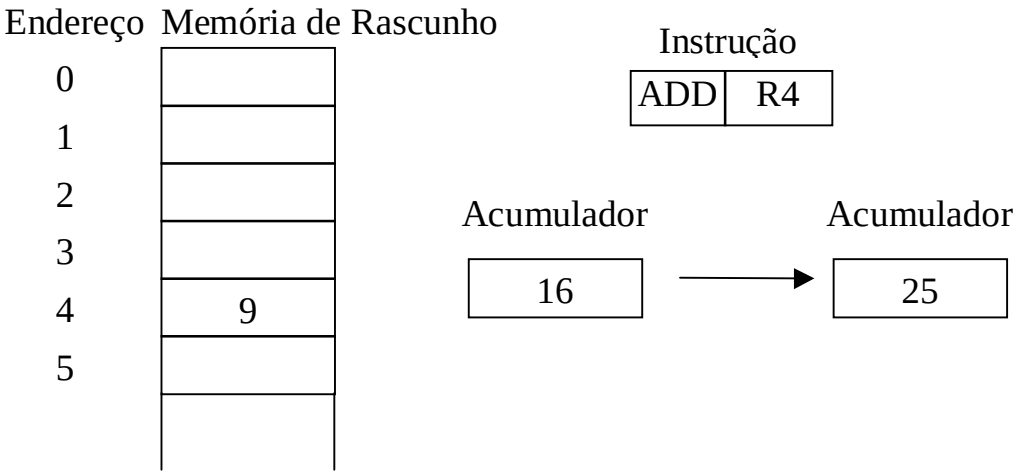


Figura 3.13. Endereçamento Via Registrador.

- Endereçamento Indireto Via Registrador: campo de endereço contém um endereço de registrador na memória de rascunho, onde está armazenado o endereço do dado na memória principal.
  - Endereço intermediário é chamado de ponteiro.
  - Quando o ponteiro se encontra em um registrador é possível acessar um endereço de memória através de um endereço de registrador.
  - Acesso a vetores.

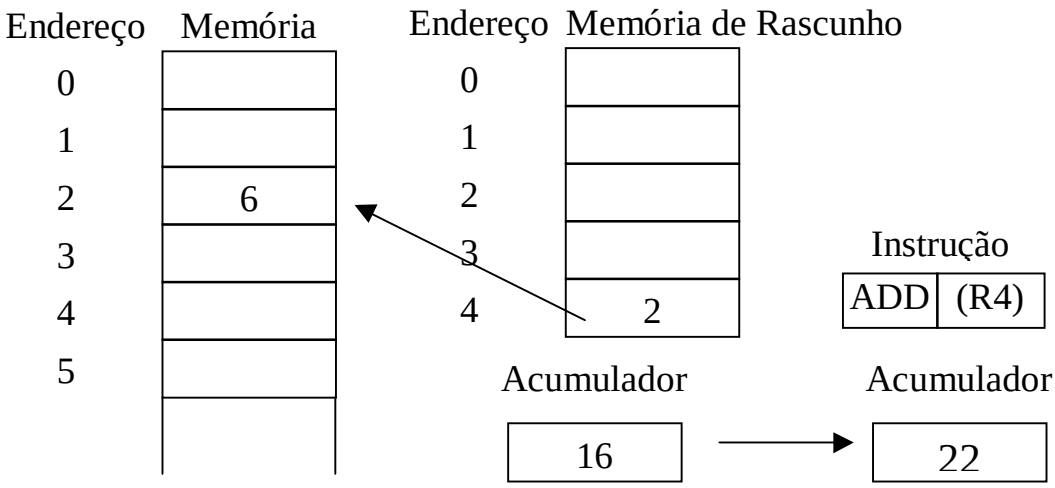


Figura 3.14. Endereçamento Indireto.

- Exemplo: soma dos elementos de um vetor com 1024 elementos.
  - Elementos inteiros, palavras de 4 bytes.
  - Elementos são somados um a um em laço.
  - Registrador R1 armazena a soma acumulada.
  - Registrador R2 armazena o endereço A do primeiro elemento.
  - Registrador R3 armazena o primeiro endereço fora do vetor.
  - As instruções do laço só endereçam operandos em registradores.

```
        MOV R1, #0          ; acumula soma em R1; valor inicial zero
        MOV R2, #A          ; R2 = endereço do vetor A
        MOV R3, #A+4096     ; R3 = endereço de 1ª palavra após A
LOOP:   ADD R1, (R2)        ; modo indireto de obter operando com base em R2
        ADD R2, #4          ; incrementa R2 de uma palavra (4 bytes)
        CMP R2, R3          ; compara R2 e R3 para testar fim de laço
        BLT LOOP            ; se R2 < R3, não terminou, continua o laço
```

Figura 3.15. Exemplo de uso de modos de endereçamento.

- Endereçamento Indexado: endereço do dado é obtido somando o valor no campo de endereço com o valor contido em um registrador de índice.
  - É utilizado para acessar posições de memória localizadas a uma distância conhecida a partir do conteúdo de um registrador.
  - Registrador de índice é incrementado a cada utilização de forma a acessar endereços contíguos.
  - Acesso a vetores e variáveis locais.

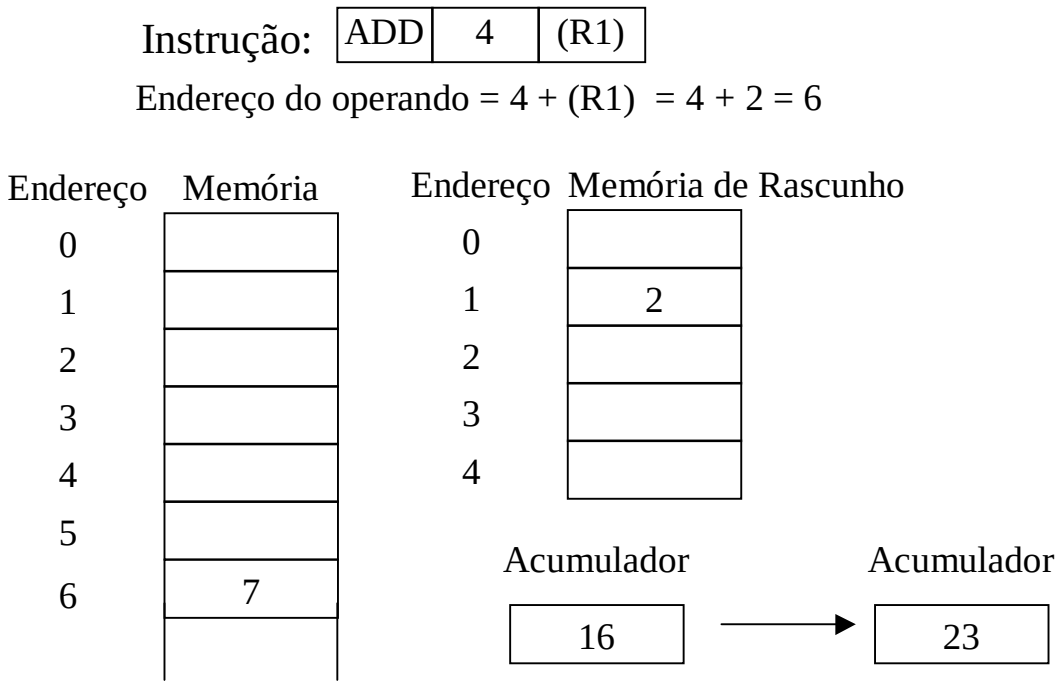


Figura 3.16. Endereçamento Indexado.

- Exemplo: soma dos elementos de um vetor, resultado da soma de dois vetores A e B, cada um com 1024 elementos.
  - Elementos inteiros, palavras de 4 bytes.
  - Elementos são somados um a um em laço.
  - Registrador R1 armazena a soma acumulada dos elementos.
  - Registrador R2 = índice i usado para percorrer os vetores.
  - Registrador R3 armazena o primeiro índice fora dos vetores.
  - Registrador R4 (rascunho) armazena a soma A(i) + B(i).
  - Instrução MOV R4, A(R2): fonte usa modo indexado, com A sendo o deslocamento a ser somado ao conteúdo do registrador de índice R2; destino endereçado via registrador R4.
  - Instrução ADD R4, B(R2): idem, com B = deslocamento.

```
        MOV R1,#0       ; Acumula soma de elementos em R1, valor inicial zero
        MOV R2,#0       ; R2 = índice i da soma corrente A(i) + B(i)
        MOV R3,#4096    ; valor do 1º valor inválido do índice
LOOP:   MOV R4,A(R2)    ; R4 = A(i)
        ADD R4,B(R2)    ; R4 = A(i) + B(i)
        ADD R1,R4       ; soma acumulada dos elementos de A(i) + B(i)
        ADD R2,#4       ; incrementa índice do tamanho da palavra: i = i + 4
        CMP R2,R3       ; compara R2 e R3 para testar fim de laço
        BLT LOOP        ; se R2 < R3, não Terminou, continua o laço
```

Figura 3.17. Exemplo de uso de Endereçamento Indexado.

- Endereçamento Base-Indexado: endereço do dado é calculado somando os valores armazenado em um registradores de índice e o outro de base.
  - O conteúdo da base é fixo enquanto que o conteúdo do índice é incrementado, como no caso anterior.
  - É bastante semelhante ao endereçamento indexado, diferenciando somente no uso do registrador de base.
  - No exemplo acima, as instruções indexadas podem ser substituídas por base-indexadas, com os endereços de A e B em R5 e R6:

```
LOOP:   MOVE R4,(R2+R5)
        ADD R4, (R2+R6)
```

Instrução: | ADD | (R1+R4) |

Endereço do operando = (R1) + (R4) = 2 + 3 = 5

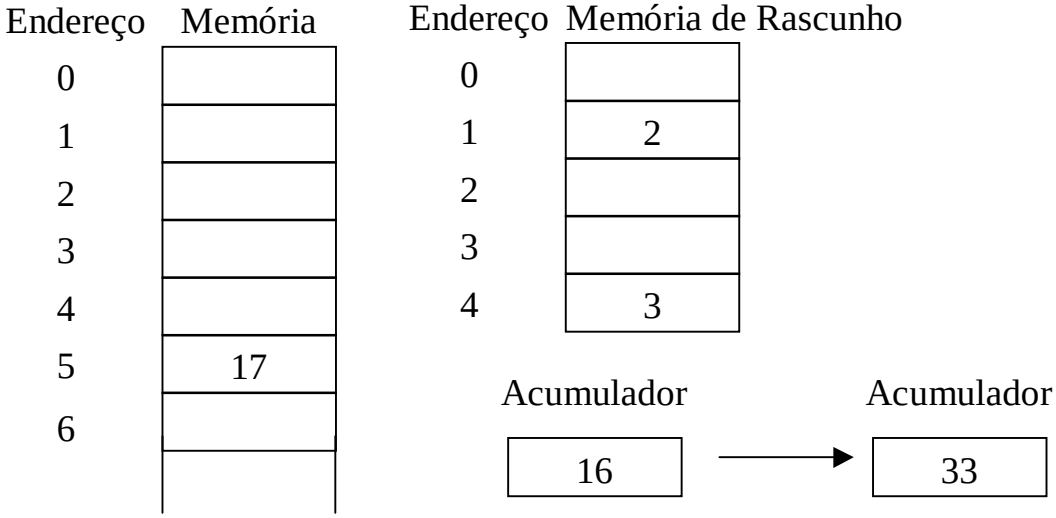


Figura 3.18. Endereçamento Base-Indexado.

- Endereçamento Via Estrutura de Pilha: os dados são buscados a partir do topo da pilha;
  - Estrutura de dados LIFO armazenada em endereços consecutivos.
  - utiliza um registrador Ponteiro de Pilha (SP - *Stack Pointer*) que aponta para o topo da pilha (último item armazenado).
  - Instruções para gerenciar a pilha:

PUSH	X
POP	Y

Empilha X ⇒SP :=SP+1; (SP) := X;
Desempilha em Y⇒ Y := (SP); SP := SP-1;

  - Algumas instruções de pilha não necessitam especificar endereço.
  - Os dados são buscados a partir de SP.

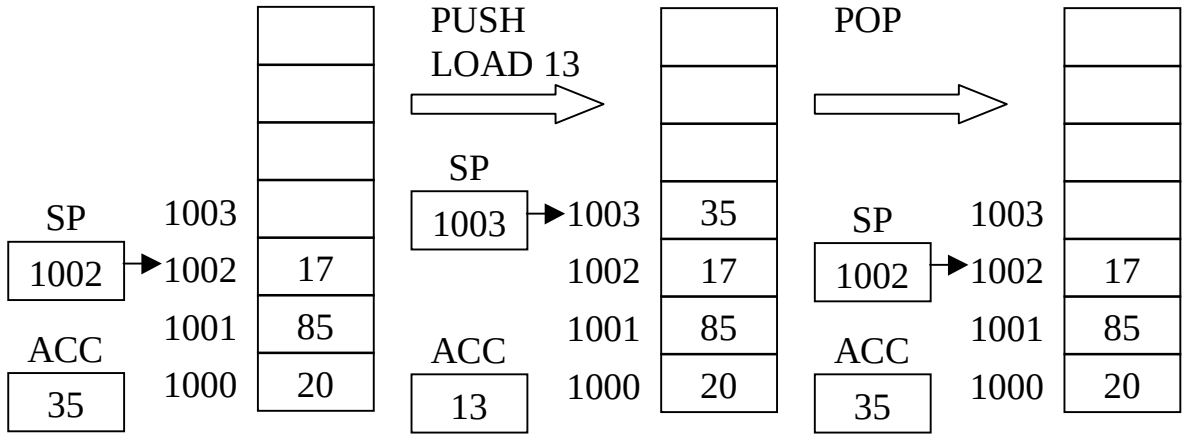


Figura 3.19. Funcionamento da Pilha.

- Exemplo: adição de dois valores usando endereçamento via estrutura de Pilha.
  - Dois valores são desempilhados,
  - Os valores são somados.
  - O resultado é empilhado.

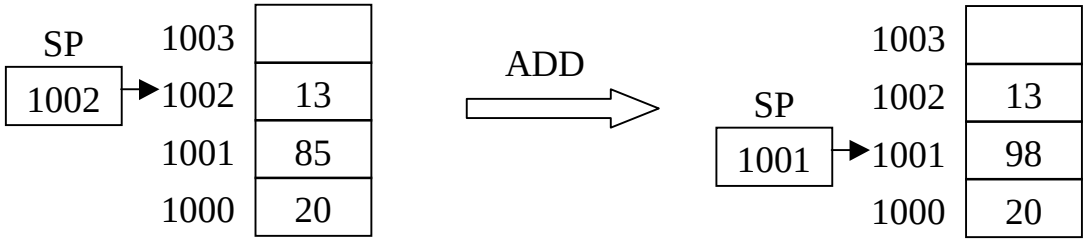


Figura 3.20. Exemplo de uso de endereçamento via Estrutura de Pilha.

3.7 Tipos de Instruções

- Instruções de movimento de dados:
  - criam uma cópia de um dado armazenado em algum lugar (fonte) e armazenam essa cópia em outro lugar (destino);
  - precisam especificar o endereço da fonte e do destino;
- Operações diádicas:
  - Combinam dois operandos para produzir um resultado;
  - Exemplo: operações aritméticas (adição, subtração, multiplicação e divisão), funções booleanas (AND, OR, NOR, NAND, ...)
- Operações monádicas:
  - Utilizam um único operando para produzir um resultado.
  - Ex.: NOT lógico, negação, deslocamento, rotação, incremento.

Instruções de movimentação de dados	
MOV DST, SRC	Move SRC para DST
PUSH SRC	Coloca SRC na pilha
POP DST	Tira valor da pilha e armazena em DST
XCHG DS1, DS2	Troca DS1 com DS2
Instruções aritméticas	
ADD DST, SRC	Soma DST (destino) com SRC (fonte)
SUB DST, SRC	Subtrai DST de SRC
MUL SRC	Multiplica o valor de EAX por SRC
INC DST	Soma uma unidade a DST
DEC DST	Subtrai uma unidade de DST
NEG DST	Nega DST (subtrai seu valor de 0)
Instruções booleanas	
AND DST, SRC	AND booleano entre SRC e DST
OR DST, SRC	OR booleano entre SRC e DST
XOR DST, SRC	EXCLUSIVE OR booleano entre SRC e DST
NOT DST	Substitui DST por seu complemento a 1
Instruções de deslocamento/rotação	
SAL/ SAR DST, #N	Desloca DST para esquerda/direita #N bits
ROL/ ROR DST, #N	Rotaciona DST para esquerda/direita #N bits

Figura 3.21. Instruções de movimentação, aritméticas, booleanas e de deslocamento/rotação do Pentium II.

- Instruções de comparação e desvio condicional:
  - Alguns programas precisam testar seus dados e alterar a seqüência de instruções executadas de acordo com o teste feito.
  - Exemplo: cálculo do fatorial de um número (n!).
  - Em linguagem de alto nível teremos:

```
if ( n >= 0 ){
     .
     .   // instruções para calcular n!
     .
}
else
     printf("ERRO: número negativo!\n");
```

  - Instruções em linguagem de máquina geralmente testam algum bit da máquina e desviam para um *label* (rótulo) de acordo com o valor do bit testado;

Programa em linguagem de alto nível	Programa em linguagem de montagem

`if(A!=B){`
`  . //instruções`
`  . para A!=B`
`  .`
`}`
`else{`
`  .`
`  . //instruções`
`  . para A==B`
`}`
	`  CMP A,B;`
	`  JZS ELSE;`
	`  .  //instruções`
	`  .  para A!=B`
	`  .`
	`ELSE:  .`
	`  .  //instruções`
	`  .  para A==B`

Figura 3.22. Exemplo do uso de instruções de comparação e desvio.

`CMP SRC1, SRC2`	Liga os flags com base em SRC1 e SRC2
`JMP ADDR`	Desvia para ADDR
`Jxx ADDR`	Desvia condicionalmente para ADDR

Figura 3.23. Instruções de comparação e desvio no Pentium II.

- Instruções de chamada de procedimento:
  - Procedimentos: conjunto de instruções que realizam uma determinada tarefa.
  - Podem ser chamados várias vezes de qualquer parte do programa.
  - Quando um procedimento é chamado através de instrução de chamada de procedimento (CALL), o programa é desviado para a primeira instrução do procedimento.
  - Quando o procedimento termina sua tarefa o programa é desviado para a instrução imediatamente seguinte a instrução de sua chamada através de instrução de retorno de procedimento (RET).
  - Quando o procedimento é chamado, é necessário armazenar o endereço de retorno.
  - O endereço de retorno pode ser armazenado numa pilha, permitindo que um procedimento chame outro procedimento.
  - Quando um procedimento termina o endereço de retorno é desempilhado e colocado no PC.

CALL ADDR	Chama o procedimento em ADDR
RET	Retorno de procedimento

Figura 3.24. Instruções de procedimento no Pentium II.

- Instruções de Controle de Laço:
  - Execução de um grupo de instruções (laço) repetida um número fixo de vezes.
  - Baseada em contador que é incrementado ou decrementado a cada execução do laço (*loop*).
  - Contador deve ser testado a cada execução do laço, quando o teste assumir uma determinada condição, o laço é interrompido.
  - Duas implementações: com teste no final ou com teste no inicio do laço.

```
        LOAD R1, #1;
L1      1ª instrução do laço;
                .
                .
                .
        Última instrução do laço;
        INC R1;
        CMP R1, #N;
        JLT L1;
        1ª instrução após o laço;
```

(a)

```
        LOAD R1, #1;
L1      CMP #N, R1;
        JLT L2;
        1º instrução do laço;
                .
                .
                .
        Última instrução do laço;
        INC R1;
        JMP L1
L2      1ª instrução após o laço;
```

(b)

Figura 3.25. Laço com teste: a) no fim, b) no começo.

- Instruções de E/S (Entrada/Saída) de dados:

  (1) E/S programada com espera ocupada:

  - Método de implementação simples.
  - Comum em sistemas de baixo desempenho (sistemas embarcados, sistemas em tempo real).
  - Cada dispositivo possui dois registradores associados: *status* e *buffer* de dados.
  - Processador testa registrador de *status* periodicamente, em laço, até verificar se o dispositivo esta pronto para receber (saída) ou se disponibilizou um dado (entrada). ⇒ Espera Ocupada.
  - Espera ocupada mantém o processador ocioso enquanto realiza operação de entrada ou saída. ⇒ Baixo desempenho. Aplicações dedicadas.
  - Instruções IN e OUT são providas para ler e escrever nos registradores.
  - Instruções selecionam um dos dispositivos de E/S disponíveis.
  - 1 caractere é lido ou escrito por vez no registrador de dados.
  - Processador precisa executar seqüência explícita de instruções para cada caractere lido ou escrito.
  - Exemplo: terminal com dois dispositivos de E/S: 1 de entrada (teclado) e 1 de saída (vídeo).

- **Entrada:**
  - Processador fica em *loop* lendo registrador de *status* do teclado até que o bit **Caractere disponível** seja ligado pelo dispositivo.
  - Quando esse bit é ligado significa que o teclado acaba de escrever um novo caractere no registrador *buffer* de dados do teclado.
  - Quando isso ocorre, o programa lê o caractere do registrador de dados do teclado, desligando o bit **Caractere disponível**.

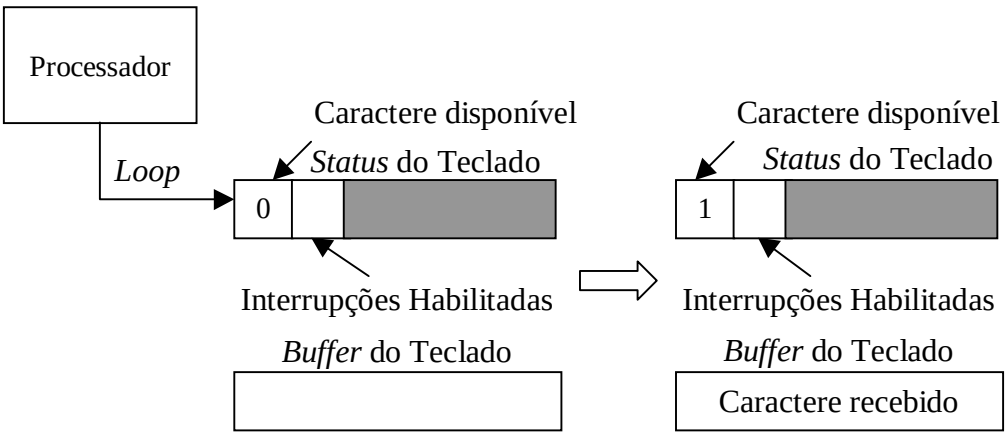


Figura 3.26. Entrada Programada com Espera Ocupada.

- **Saída:**
  - Processador fica em *loop* lendo registrador de *status* do vídeo até que o bit PRONTO seja ligado pelo dispositivo.
  - Quando esse bit é ligado significa que o vídeo está pronto para receber um novo caractere.
  - Quando isso ocorre, o programa coloca o caractere no registrador de dados do vídeo, desligando o bit PRONTO.

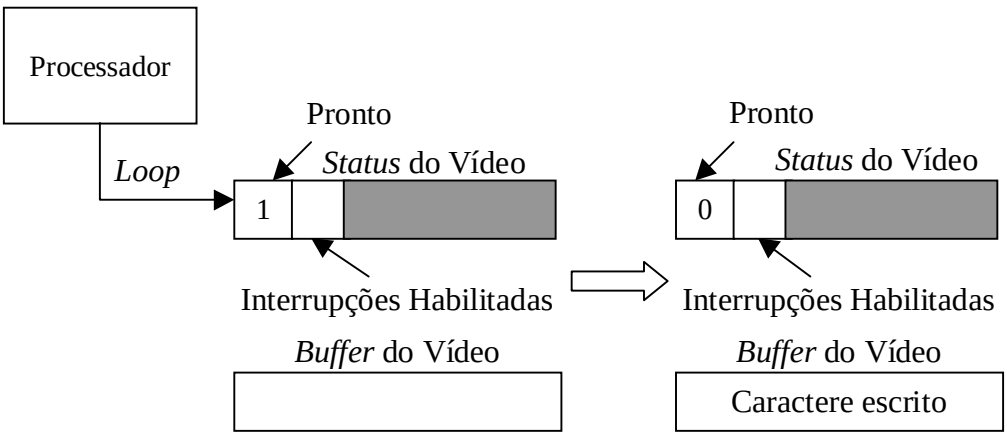


Figura 3.27. Saída Programada com Espera Ocupada.

(2) E/S dirigida por interrupção:

- Processador apenas inicia a operação de E/S.
- Processador habilita interrupções.
- Processador sai do processo, ficando livre para outras tarefas.
- Quando o caractere é escrito ou recebido, o dispositivo gera uma interrupção, (ativa sinal no pino de interrupção do processador), avisando que a operação de E/S foi concluída.
- Sinal de interrupção = (bit Pronto ou bit Caractere Disponível) AND (bit Habilita Interrupções).
- Vantagem: o processador não precisa esperar que o dispositivo acabe operação de E/S.
- Desvantagem: a cada caractere transmitido é necessário tratar uma interrupção.
- Exemplo de uso de interrupção em uma operação de saída de dados:
  - Dispositivo está Pronto para receber um novo caractere.
  - Processador coloca um caractere no registrador buffer de dados do dispositivo, o que desliga o bit Pronto.
  - Processador liga o bit Habilita Interrupções e sai do processo;
  - Dispositivo implementa a saída do caractere.
  - Concluída a operação de saída, o dispositivo liga o bit Pronto.
  - Sinal de interrupção para o processador é gerado como (bit Habilita Interrupção) AND (bit Pronto).
  - Processador desliga bit Habilita Interrupções.
  - Processador interrompe o programa corrente e executa rotina de atendimento a interrupção.
  - Processador retoma o programa interrompido.

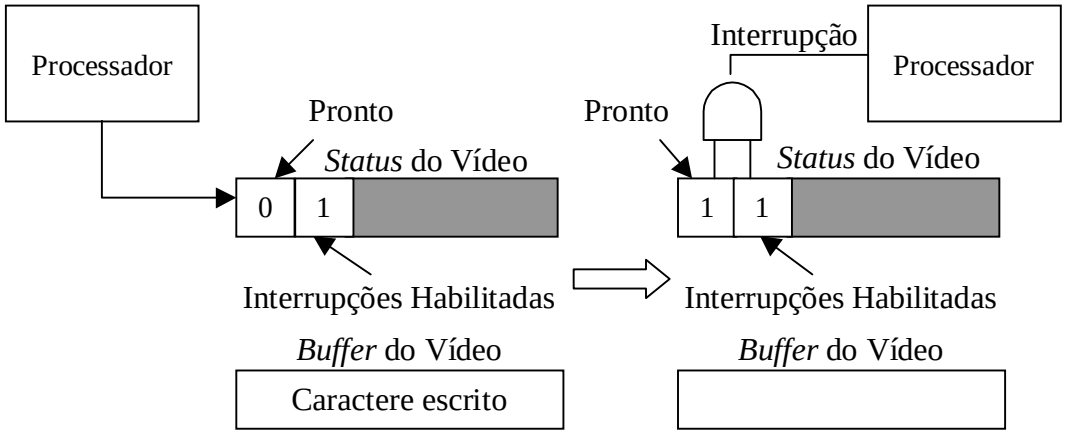


Figura 3.28. Saída dirigida por Interrupção.

(3) E/S com acesso direto à memória (DMA):

- Utiliza um controlador de DMA dedicado que toma posse do barramento e realiza E/S programada, avisando ao processador, por meio de interrupção, quando a operação de E/S estiver finalizada.
- Vantagens:
  - O processador não precisa ficar em espera ocupada, assim fica livre para realizar outras tarefas.
  - Não é necessário tratar uma interrupção por caractere transmitido, a interrupção só é gerada após a transmissão de um bloco de caracteres de tamanho especificado.
- Desvantagens:
  - Problema: toda vez que o controlador requisita o barramento, seja para acessar a memória ou para acessar o dispositivo, ele tem prioridade sobre o processador.
  - Diz-se que o controlador de DMA “rouba” ciclos de barramento do processador.
- Controlador de DMA possui, no mínimo, quatro registradores:
  - Endereço: armazena o endereço de memória a ser lido ou escrito.
  - Contador: armazena o número de bytes a serem lidos ou escritos.
  - Dispositivo: armazena o número do dispositivo E/S a ser usado.
  - Direção: indica se é operação de leitura ou escrita no dispositivo.

- Exemplo: escrever 256 bytes, armazenados na memória principal a partir do endereço 1023, para um dispositivo de saída identificado pelo número é 7. Considere Leitura = 0, Escrita = 1.
  - Processador inicia os registradores do controlador de DMA: Endereço = 1023, Contador = 256, Dispositivo = 7, Direção = 1.
  - Controlador de DMA requisita barramento para ler o endereço 1023 da memória, fazendo a sua leitura e obtendo um byte.
  - Controlador solicita escrita no dispositivo 7 para enviar um byte.
  - Quando o byte for enviado, controlador incrementa o endereço e decrementa o contador, verificando se este chegou em zero.
  - Caso o contador não seja igual a zero processo todo é repetido para enviar o próximo byte.
  - Quando o contador chegar a zero, o controlador de DMA para o processo e avisa o fim da operação de E/S ao processador por meio de uma interrupção.

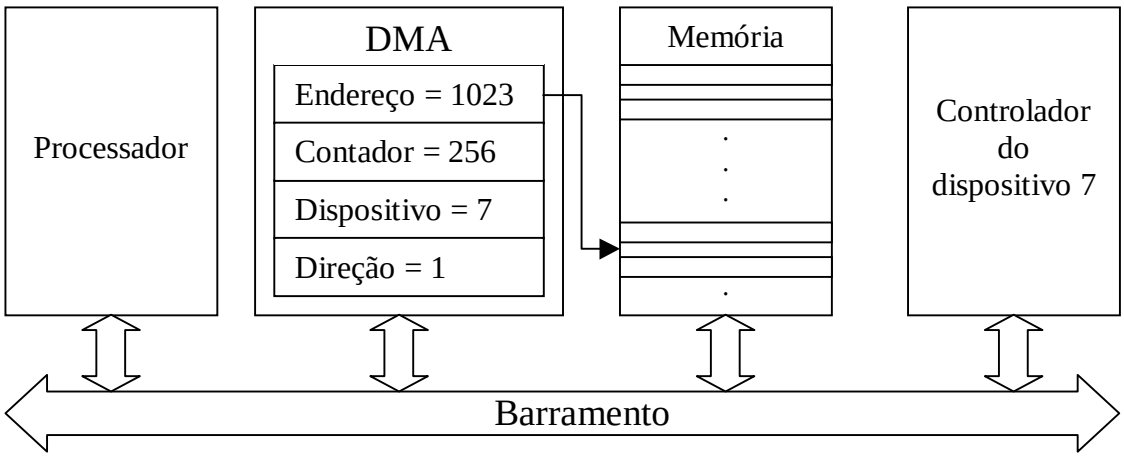


Figura 3.29. E/S baseada em Controlador de DMA.

3.8 Fluxo de Controle

- O fluxo de controle é seqüência na qual as instruções são executadas.
- A execução da maioria das instruções não altera o fluxo de controle. Em um fluxo de controle normal as instruções são buscadas em palavras consecutivas da memória. Após a execução de uma instrução, a próxima instrução é buscada na memória incrementando o PC do tamanho da instrução que acaba de ser executada.
- O fluxo de controle normal é alterado por desvios, chamadas de procedimentos, co-rotinas, armadilhas (*traps*) e interrupções.

- Desvios:
  - quando ocorre um desvio em um programa a próxima instrução a ser buscada não é mais a instrução na posição consecutiva da memória e sim a instrução armazenada no endereço de destino do desvio.
  - as instruções de desvio (condicional ou incondicional) em baixo nível são utilizadas para implementar as estruturas **if**, **while**, **for** e outras presentes na maioria das linguagens de alto nível.

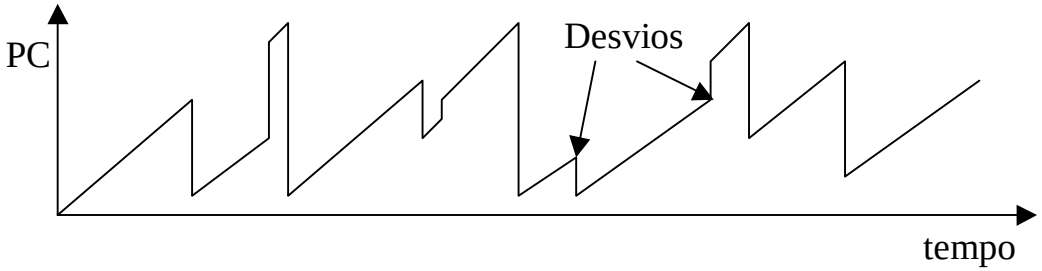


Figura 3.30. Mudança do fluxo de controle em instruções de desvios.

- Procedimentos:

  - Procedimento é uma seqüência de instruções que pode ser executada várias vezes a partir de diferentes partes de um programa (ou de outro procedimento).
  - Em lugar de repetir as linhas de código correspondentes ao procedimento em várias partes do programa, o procedimento aparece apenas uma vez.
  - Duas instruções são fornecidas:
    - Chamada de procedimento (CALL): substitui as instruções do procedimento em cada parte do programa onde elas deveriam ser executadas. Recebe como parâmetro o endereço da primeira instrução do procedimento. A execução da chamada desvia o fluxo de controle para a primeira instrução do procedimento. Exemplo: CALL PROCED.
    - Retorno de procedimento (RET): última instrução do procedimento. Sua execução desvia o fluxo para a instrução seguinte à chamada corrente do procedimento correspondente.

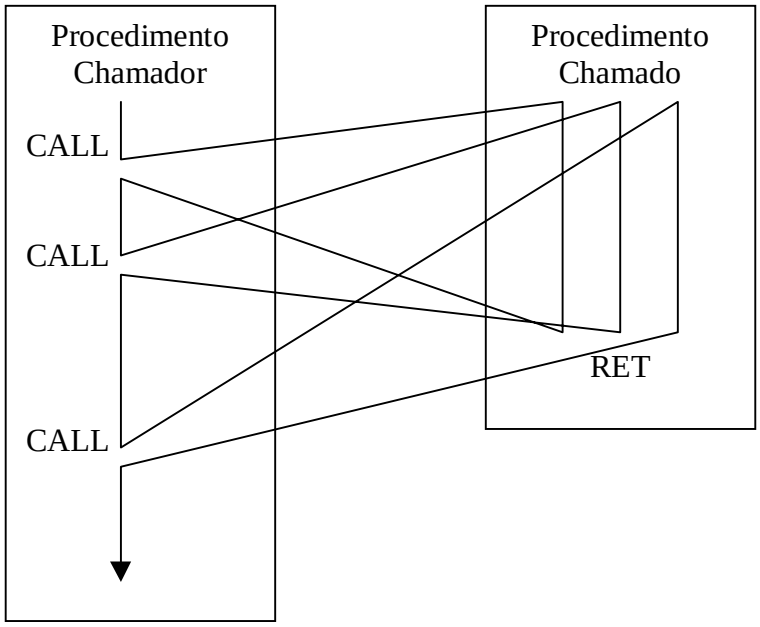


Figura 3.31. Chamada de procedimento.

  - Para poder retornar, é preciso armazenar o endereço de retorno do procedimento em algum lugar.
  - A pilha é o lugar mais indicado para armazenar o endereço de retorno, pois permite implementar procedimentos que chamam outros procedimentos.

- Os parâmetros e variáveis locais do procedimento são armazenados também na pilha.
- Quando um procedimento é chamado, empilha-se uma estrutura de pilha, que contém os parâmetros, variáveis locais e endereço de retorno do procedimento.
- A estrutura no topo da pilha é a estrutura do procedimento corrente;
- Além do ponteiro de pilha (SP) é necessário outro ponteiro, a Base Local (LB - *Local Base*), que aponta para um endereço fixo da estrutura.
- Variáveis locais e parâmetros do procedimento são acessados utilizando um deslocamento fixo com relação a LB;
- LB é necessário, pois pode ser necessário armazenar outros valores na pilha durante o procedimento, com isso o deslocamento das variáveis locais e parâmetros com relação a SP não é fixo.

Para exemplificar considere o seguinte programa em C:

```
void soma( int, int, int);
void subtracao( int, int, int);
void multiplicacao( int i, int j, int k);
int main(){
    soma( 1, 1, 1);
    return 0;
}
void soma( int i, int j, int k){
    int s;
    s = i+j+k;
    subtracao( s, j, k);
    multiplicacao( s, j, k);
}
void subtracao( int i, int j, int k){
    int s;
    s = i-j-k;
    multiplicacao( s, j, k);
}
void multiplicacao( int i, int j, int k){
    int m;
    m = i*j*k;
}
```

* Quando o programa principal vai fazer a chamada `soma(1,1,1);` primeiro são empilhados os três parâmetros e em seguida uma instrução CALL é executada empilhando o endereço de retorno.
* Após isso o procedimento faz o seguinte:
  - Empilha LB anterior;
  - Atualiza LB com o valor de SP;
  - Avança SP para reservar espaço para as variáveis locais;
* Feito isso a pilha estará como mostrado em (a);
* Neste exemplo, onde os endereços aumentam a medida que a pilha cresce, as variáveis locais do procedimento (no caso s) são acessadas utilizando um deslocamento positivo de endereços com relação a LB;
* Caso seja necessário acessar os parâmetros, é feito um deslocamento negativo de endereços com relação a LB;
* Na seqüência do procedimento, s=3 é calculado e é feita a chamada `subtracao(3,1,1);`
* Uma nova estrutura é empilhada e a pilha fica como em (b);
* Em seguida a função `subtracao` calcula s=1 e chama `multiplicacao(1,1,1);`
* Após essa chamada a pilha fica como mostrada em (c);
* A função `multiplicacao` calcula m=1 e retorna, desempilhando uma estrutura; assim a pilha volta a ficar com a configuração anterior como mostra (d);
* Em seguida `subtracao` também retorna e a pilha fica como mostra (e);
* Quando `soma` retoma o controle do programa é feita uma chamada para `multiplicacao(3,1,1);`
* Uma nova estrutura é empilhada ficando a pilha como mostra (f) .
* Esse processo continua até que o controle volte para o programa principal;

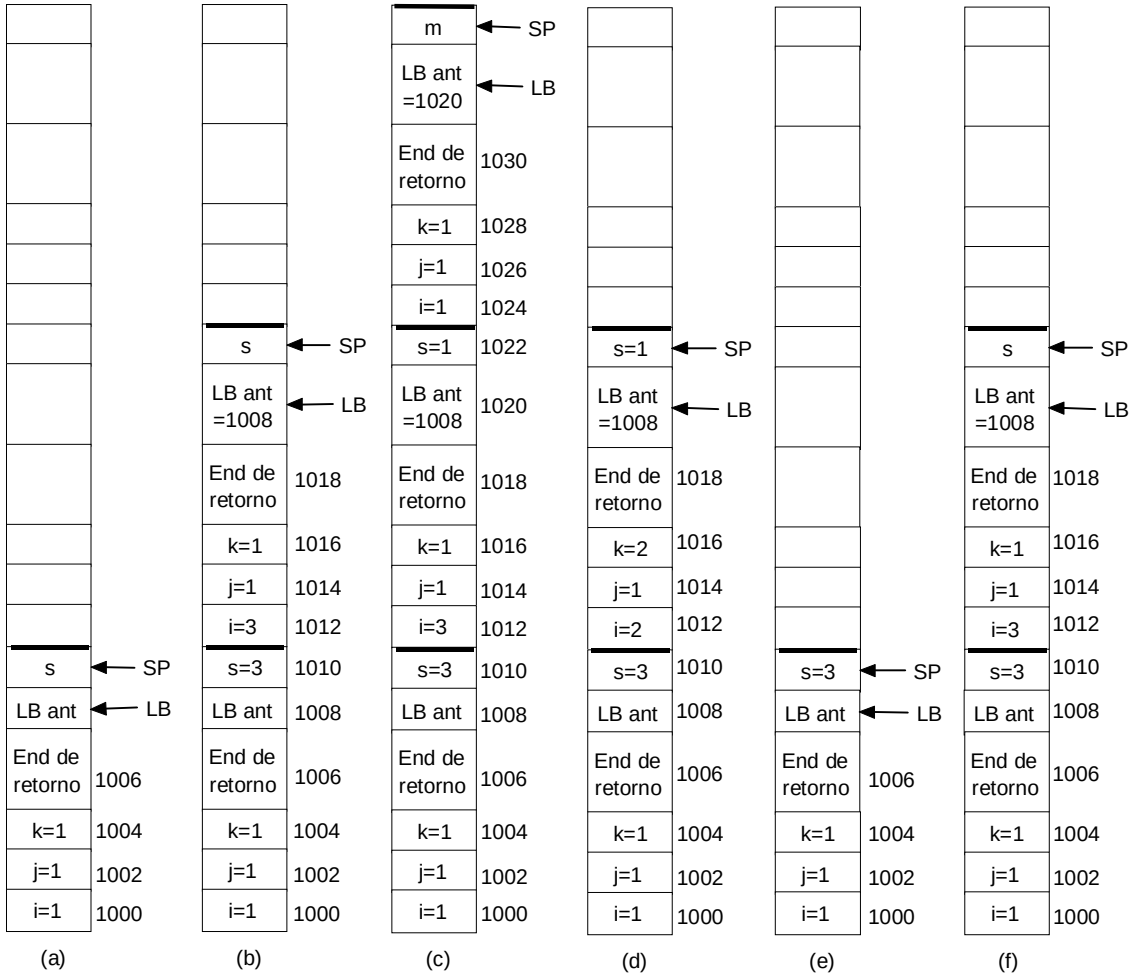


Figura 3.32. Comportamento da pilha em procedimentos.

- Co-Rotinas:

  - Co-Rotinas são pares de seqüências de instruções que executam tarefas interdependentes em paralelo.
  - Aplicações mais comuns: paralelismo simulado, teste de software para multiprocessadores.
  - Implementadas através de instrução REATIVA: empilha endereço da próxima instrução da co-rotina corrente e desvia para a instrução seguinte à instrução mais recentemente executada da outra co-rotina.
  - Dadas duas co-rotinas A e B, quando A precisa retornar o controle para B (REATIVA B), desvia para o comando seguinte àquele que originou a chamada a A (REATIVA A), (e vice-versa).
  - A Instrução REATIVA desempilha o endereço de destino, empilha o PC (endereço de retorno) e coloca o endereço de destino no PC, ou seja, logicamente corresponde a uma troca entre o (PC) e o (SP).

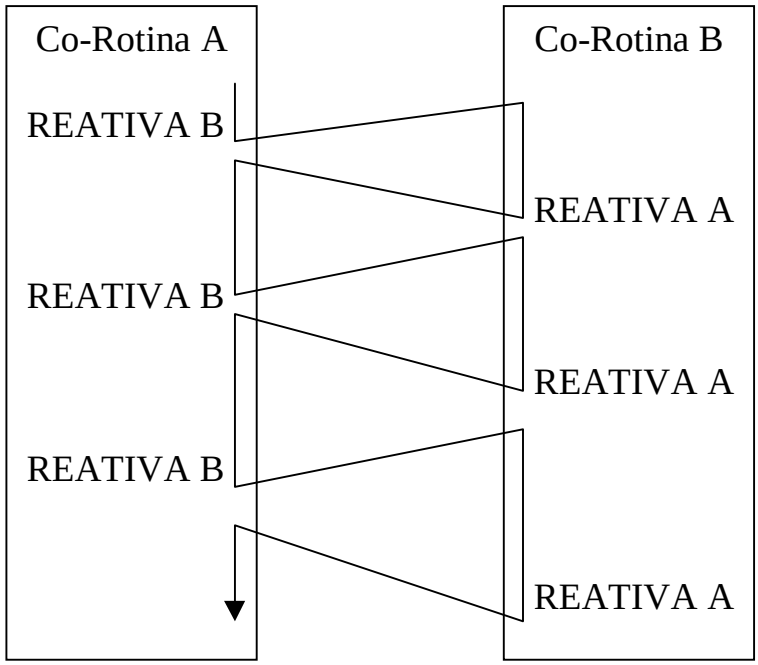


Figura 3.33. Fluxo de controle em Co-Rotinas.

- Armadilhas (*Traps*):

  - Uma armadilha é uma chamada automática de procedimento, causada pela ocorrência de uma situação especial (exceção) na execução do programa, detectada pelo *hardware* ou pelo microprograma.
  - Durante a execução de um programa, uma *trap* pode ser gerada, por exemplo, quando ocorre um *overflow*, uma violação de proteção de memória, uma divisão por zero, estouro de pilha, etc.
  - Quando um ocorre uma *trap*, o fluxo de controle é alterado para uma posição fixa da memória, que contém uma chamada para uma rotina de tratamento de *trap*.
  - As *traps* são causadas pelo próprio programa.
  - As *traps* são ditas síncronas, já que, na medida em que toda vez que um programa é executado com uma mesma entrada, a *trap* é gerada sempre no mesmo ponto.

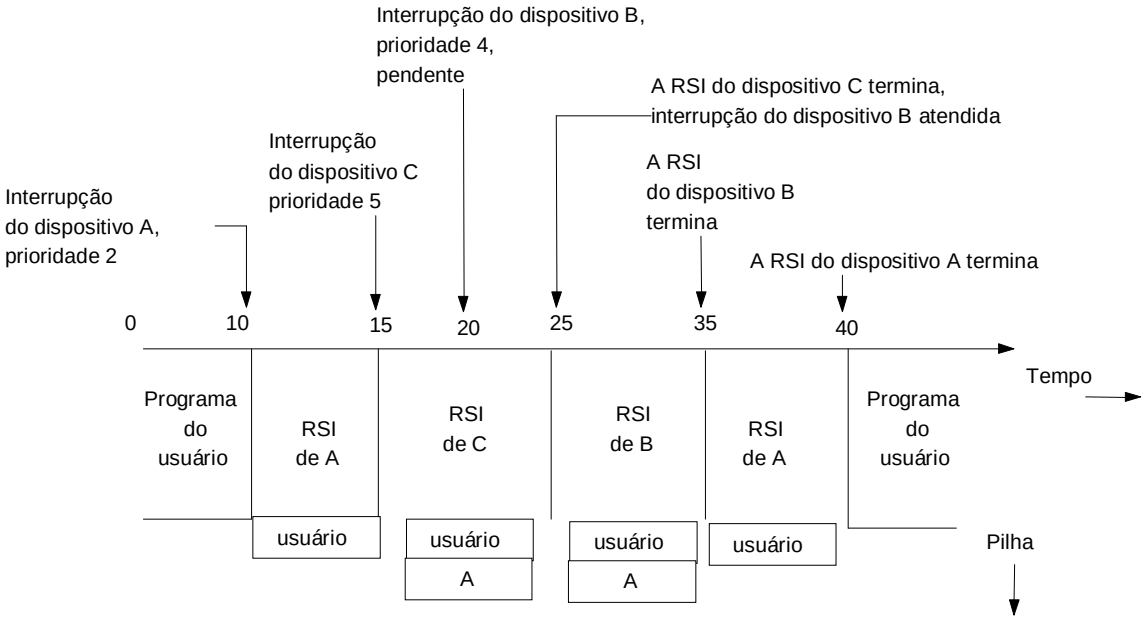
- Interrupções:

  - Uma interrupção é uma alteração no fluxo de controle causada por um evento externo ao programa (geralmente, operações de E/S).
  - Quando ocorre uma interrupção, o programa corrente para de ser executado e o controle é transferido para o tratamento da interrupção através de uma Rotina de Serviço de Interrupção (RSI).
  - O endereço da primeira instrução da RSI está armazenado numa estrutura denominada Vetor de Interrupção.
  - Os vários dispositivos de E/S são identificados por um número, o qual é utilizado para indexar o vetor de interrupção.
  - Ao término da rotina de serviço, o fluxo de controle retorna para o programa interrompido.
  - Antes de uma rotina de tratamento de interrupção ser iniciada, o estado atual da máquina (valores do PC, do SP, etc., por exemplo), é salvo, normalmente, na pilha.
  - Quando a rotina de tratamento termina o estado da máquina anterior à interrupção deve ser restaurado.
  - A interrupção é dita transparente ao programa interrompido, o qual continua a sua execução como se nada tivesse acontecido.
  - Ao contrário das *traps*, as interrupções são assíncronas, pois podem ocorrer aleatóriamente, em qualquer ponto de um programa a cada nova execução do mesmo.

  - Problema: quando existem vários dispositivos de E/S, uma interrupção pode ocorrer quando outra está em andamento.

    - Solução 1: quando uma interrupção é iniciada, qualquer outra é inibida. Esta solução pode causar problemas de perda de dados em um dispositivo que receba os dados muito rapidamente.
    - Solução 2: atribuir prioridades aos dispositivos.
      - Dispositivos com tempo crítico possuem maior prioridade.
      - Quando uma nova interrupção ocorre com prioridade maior do que a RSI corrente, o estado atual é empilhado e a nova RSI é executada. Caso a nova interrupção tenha prioridade menor do que a atual, ela não é atendida e fica pendente.
      - Interrupções por *software* também são possíveis. Programas mais críticos devem ter maior prioridade (armazenada na PSW). Programas do usuário têm a mais baixa prioridade.

- Exemplo: computador com três dispositivos de E/S, A, B e C, cada um com prioridade 2, 4 e 5 respectivamente. O programa do usuário, inicialmente em execução, possui prioridade zero
  - O programa do usuário é interrompido pelo dispositivo A.
  - O estado atual da máquina é empilhado e a RSI de A é executada.
  - RSI de A é interrompida pelo dispositivo C (prioridade maior).
  - O estado atual da máquina é empilhado e a RSI de C é executada.
  - Dispositivo B gera interrupção.
  - Como a prioridade do dispositivo B é menor que a prioridade do dispositivo C, o seu pedido de interrupção fica pendente e a RSI de C continua.
  - Quando a RSI de C termina, o estado da RSI de A é desempilhado e nesse mesmo instante a interrupção de B recebe *acknowledge*.
  - O estado atual da máquina (correspondente à RSI de A) é empilhado novamente. E a RSI de B é executada.
  - Ao fim da RSI de B, o estado da RSI de A é desempilhado e esta rotina continua até o fim.
  - Ao fim da RSI de A, o estado do programa do usuário é desempilhado e o programa continua a sua execução a partir do ponto interrompido, como se nada tivesse ocorrido.



3.34. Atendimento a interrupções com níveis de prioridade.

- Ações de *Hardware* e *Software* numa interrupção:

  - Exemplo: imprimir uma linha de caracteres em um terminal.
    - Caracteres armazenados em um *buffer*. Variável global *ptr* (ponteiro) aponta para o próximo caractere a ser escrito. Variável *count* armazena o número de caracteres a imprimir.
    - Utilizando o esquema de E/S dirigida por interrupção, o processador verifica se o dispositivo está pronto. Caso esteja, um caractere é colocado no *buffer* de dados.
    - O processador vai realizar outra tarefa e fica a espera da interrupção.
    - Quando o caractere é exibido, a seguinte seqüência de eventos é realizada:

  - **Ações do Hardware:**

1. Controlador do dispositivo ativa a linha de interrupção no barramento para iniciar a interrupção.
2. Processador em condições de atender a interrupção ativa linha de reconhecimento (*acknowledge*) da interrupção no barramento.
3. Controlador do dispositivo coloca número inteiro (vetor de interrupção) que o identifica na via de dados do barramento.
4. Processador lê o vetor de interrupção e o armazena.
5. Processador empilha PC e registrador de *Status* (PSW).
6. Processador localiza o endereço da RSI do dispositivo utilizando o vetor de interrupção como índice para uma tabela no início da memória. Este endereço é carregado no PC.

- **Ações do Software:**

7. Rotina de tratamento salva (na pilha, por exemplo), o valor atual de todos os registradores.
8. Dispositivo que causou a interrupção é identificado.
9. Códigos de estado e outros dados podem ser obtidos agora.
10. Eventuais erros de E/S podem ser tratados neste momento.
11. *ptr* é incrementado de um byte e *count* é decrementado de uma unidade, caso o contador *count* seja maior do que zero, ainda há caracteres a serem transferidos. O byte apontado por *ptr* é colocado no *buffer* de dados do dispositivo.
12. Se necessário, gera-se código especial para avisar ao dispositivo que uma interrupção está sendo processada.
13. Os registradores salvos na pilha são restaurados.
14. A rotina de tratamento retorna (instrução RETURN FROM INTERRUPT) e tudo volta ao estado anterior à interrupção.