Parte2:
A Arquitetura do Conjunto de Instruções

Bibliografia

[1] Miles J. Murdocca e Vincent P. Heuring, “Introdução à Arquitetura de Computadores”

[2] Marcelo Rubinstein – Transparências do curso de Computadores Digitais (UERJ)

[3] Andrew S. Tanenbaum, “Organização Estruturada de Computadores”

[4] José Paulo Brafman – Transparências do curso de Organização de Computadores (UFRJ)

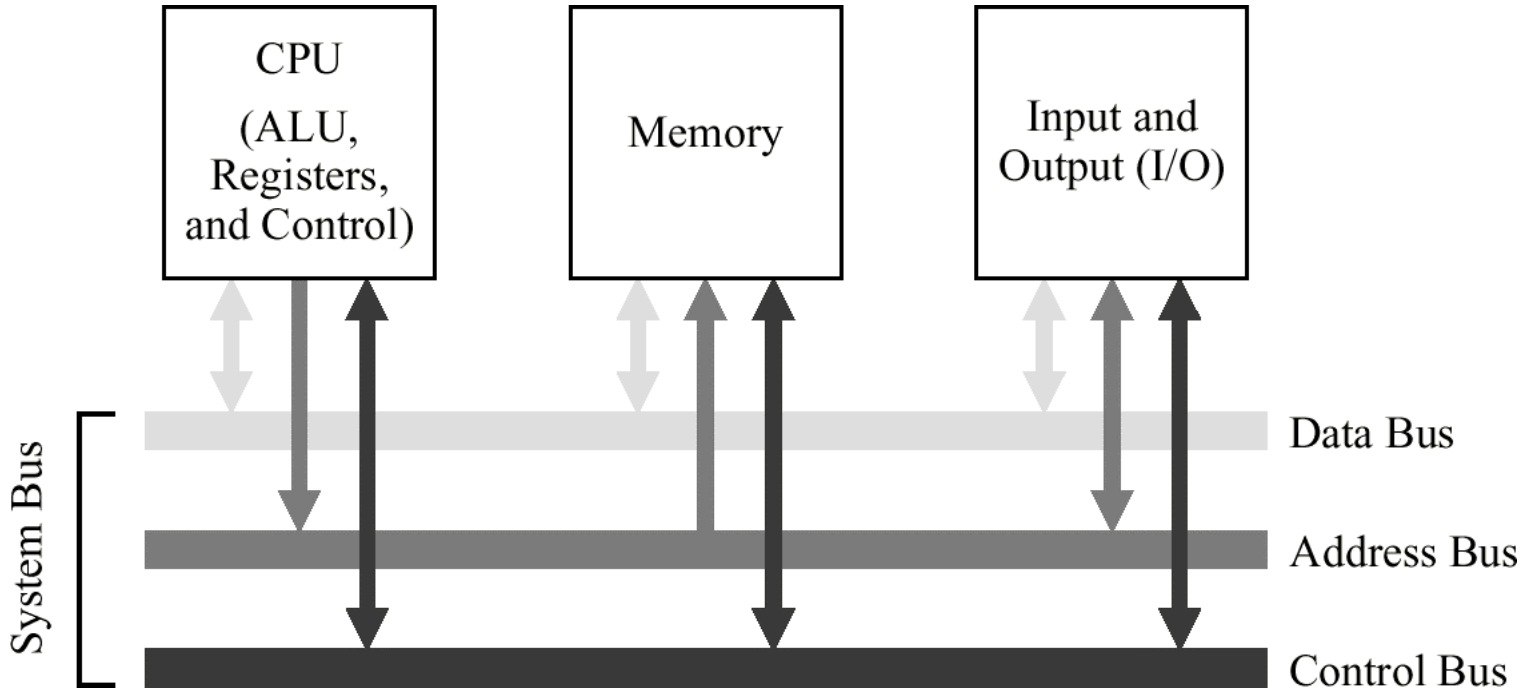
[5] Victor Paulo Peçanha Esteves – Apostila de Arquitetura de Microcomputadores, Módulo 1

Arquitetura do Conjunto de Instruções

- **A Arquitetura do Conjunto de Instruções (*Instruction Set Architecture* - ISA) de uma máquina corresponde aos níveis de linguagem de montagem (assembly) e de linguagem de máquina.**
- **O *compilador* traduz uma linguagem de alto nível, que é independente de arquitetura, na linguagem assembly, que é dependente da arquitetura.**
- **O *assembler* (ou montador) traduz programas em linguagem assembly em códigos binários executáveis.**
- **Para linguagens completamente compiladas, como C e Fortran, os códigos binários são executados diretamente pela máquina-alvo. O Java pára a tradução no nível de byte code. A *máquina virtual Java*, que está no nível da linguagem assembly, interpreta os byte codes (implementações em hardware da JVM também existem, caso em que o byte code Java é executado diretamente.)**

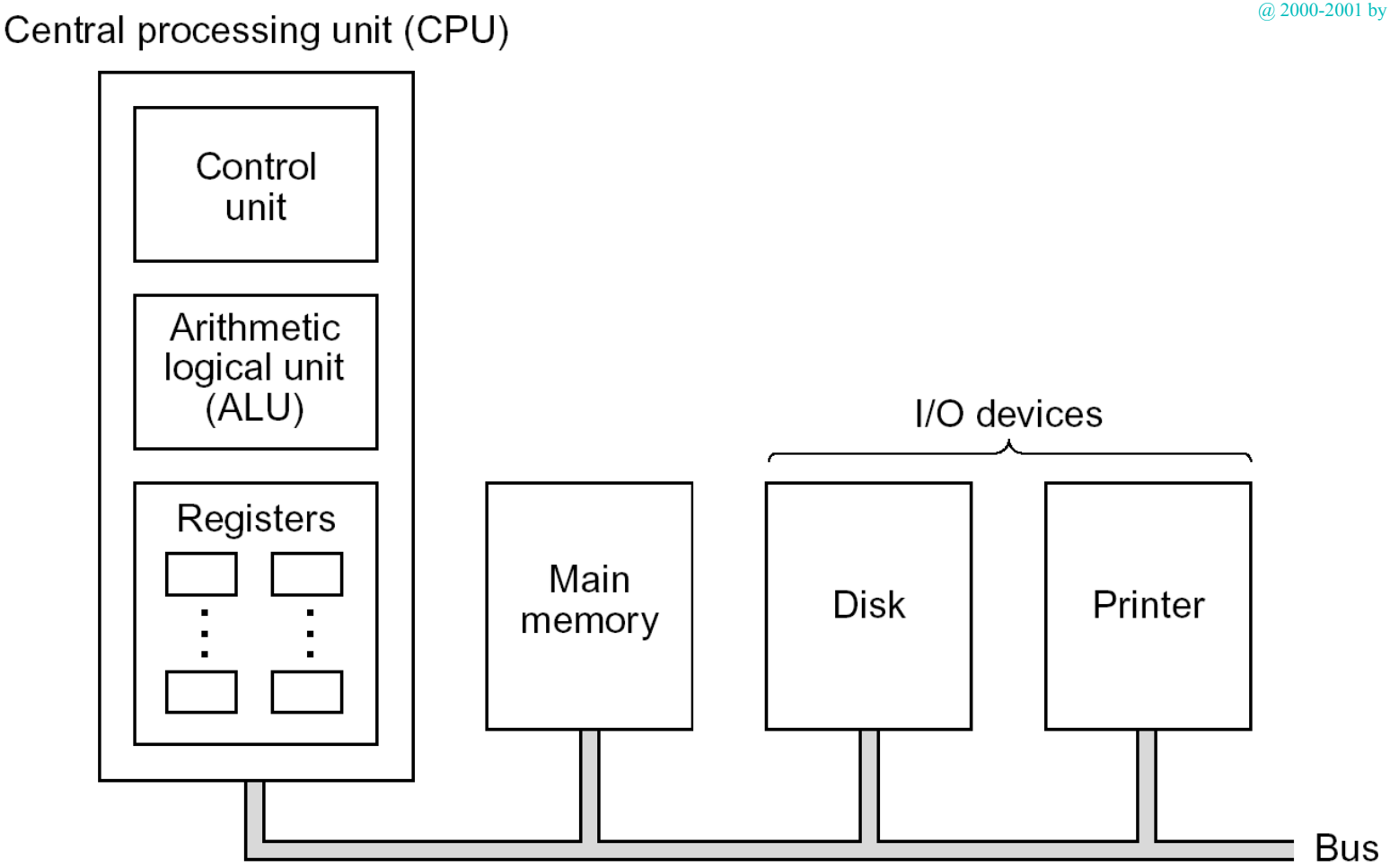
O Modelo de Barramento de Sistemas Revisitado

- **Um programa compilado é copiado do disco rígido na memória. A CPU lê as instruções e os dados da memória, executa as instruções, e armazena os resultados de volta na memória.**



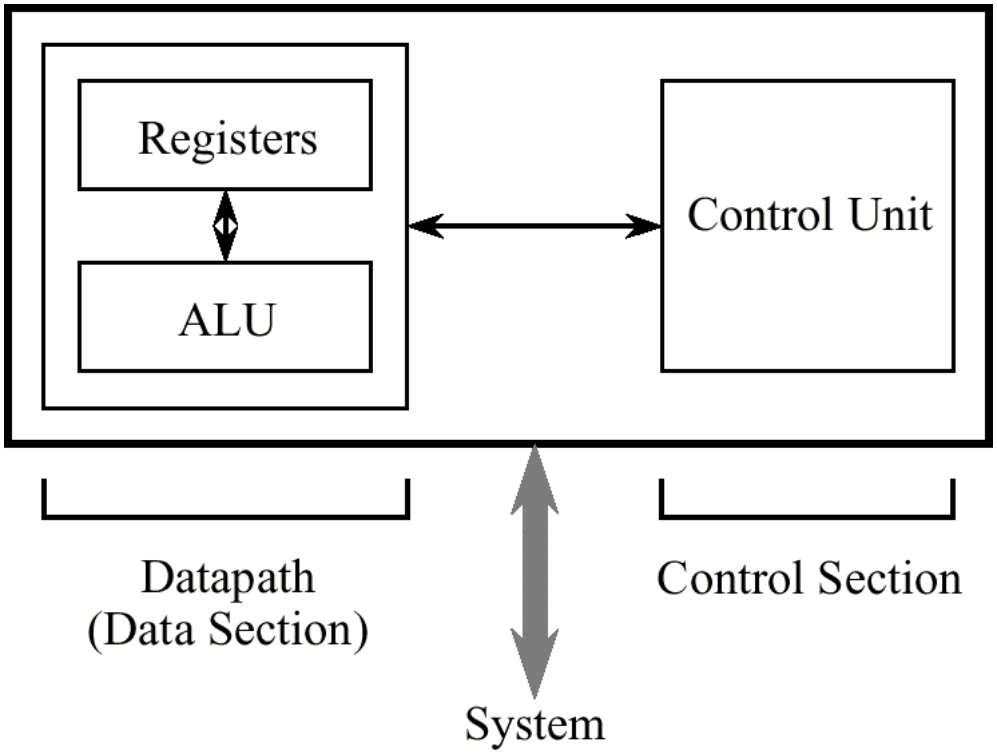
Organização de um Computador Simples

(fonte: Tanenbaum)



Visão Abstrata de uma CPU

- **A CPU consiste de uma seção (ou caminho) de dados (*datapath*) que contém registradores e uma ALU, e uma seção de controle, que interpreta instruções e efetua transferências entre registradores.**

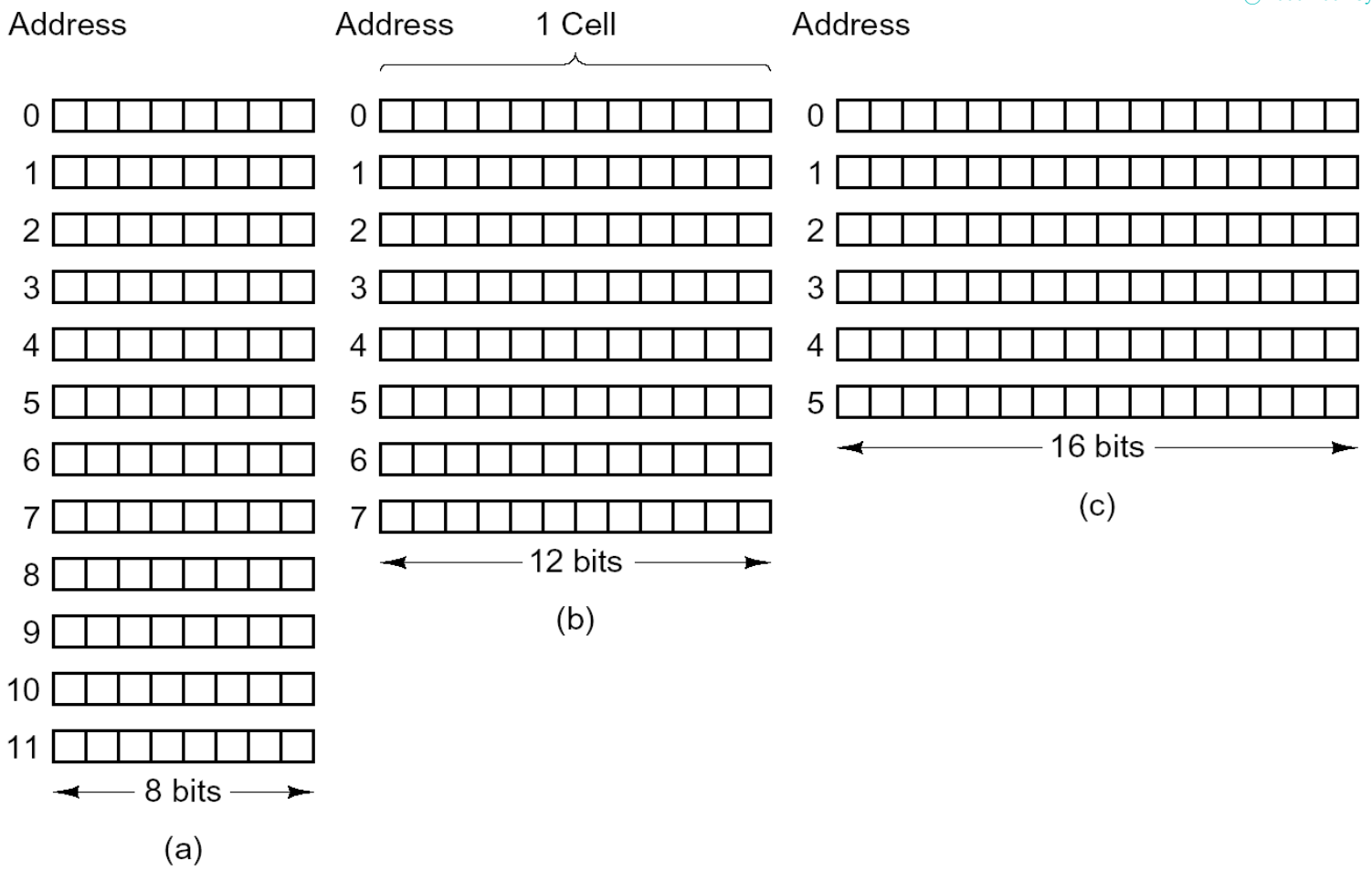


Memória

- Onde os programas e os dados são armazenados
- Sua unidade básica é o bit
- É formada por um conjunto de células (ou posições)
  - O número de bits de uma célula é chamado palavra
  - Células referenciadas por um endereço

Memória (cont.)

Organização de uma memória de 96 bits (fonte: Tanenbaum)



Processador

- Cérebro do computador
- Também conhecido como CPU
- Sua função é executar instruções
- Constituído de
  - Unidade de controle
    - Busca instruções na memória principal e determina o tipo de cada instrução
  - Unidade lógica e aritmética (ALU)
    - Realiza um conjunto de operações necessárias à execução de instruções
- Possui uma memória pequena e de alta velocidade formada por um conjunto de registradores

Processador (cont.)

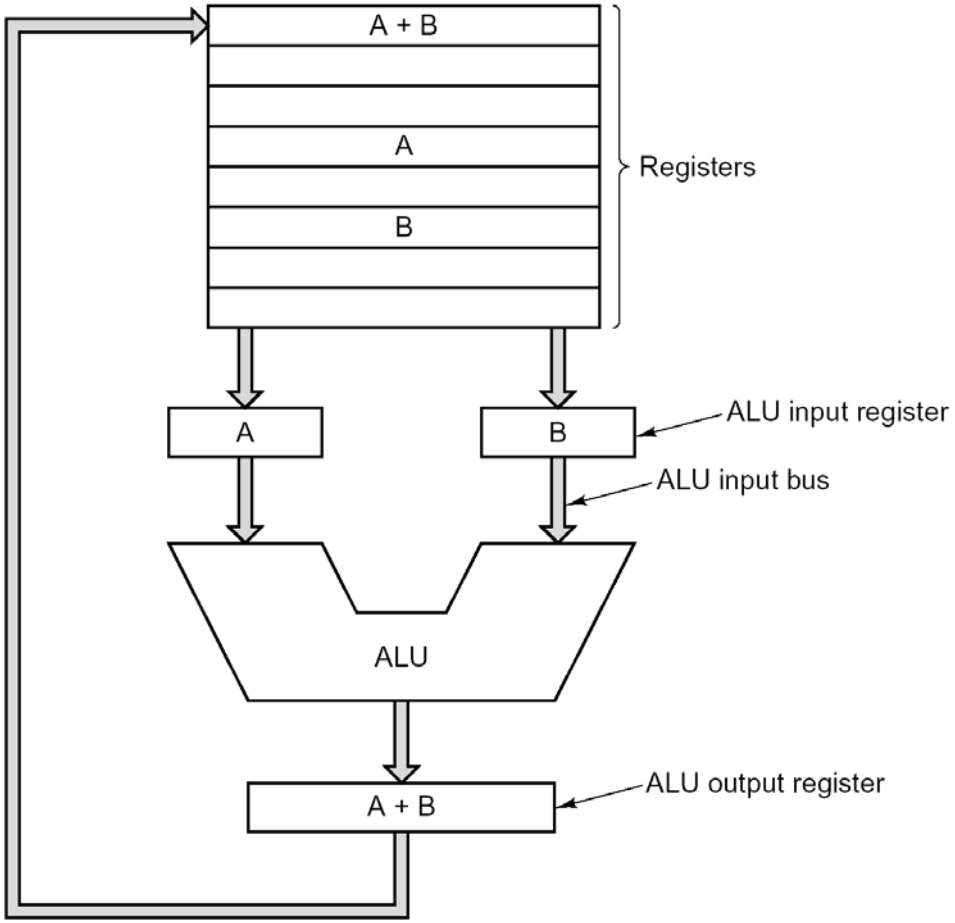
- Registrador é constituído de *n* flip-flops, cada flip-flop armazenando um bit
  - PC (*Program Counter*): aponta para a próxima instrução a ser buscada na memória para ser executada
  - IR (*Instruction Register*): armazena a instrução que está sendo executada
  - Outros de uso geral ou específico

Processador (cont.)

- Organização do processador
  - Caminho de dados constituído de
    - Registradores
    - ULA
    - Barramentos: conjunto de fios paralelos que permite a transmissão de dados, endereços e sinais de controle
  - Instruções do processador
    - Registrador-memória
    - Registrador-registrador

Processador (cont.)

Caminho de dados de uma típica máquina de Von Neumann (fonte: Tanenbaum)



Ciclo de Busca e Execução

- **Os passos que a Unidade de Controle segue durante a execução de um programa são:**

  (1) Busca na memória da próxima instrução a ser executada.

  (2) Decodificação do opcode.

  (3) Leitura dos operandos da memória, se necessário.

  (4) Execução da instrução e armazenamento dos resultados.

  (5) Volta ao passo 1.

Este é conhecido como ciclo de busca e execução *(fetch-execute cycle)*, ou busca-decodificação-execução.

Busca e Execução (det.)

1 - Busca da próxima instrução na memória e armazenamento da instrução em IR

2 - Atualização de PC

3 - Determinação do tipo de instrução do IR

4 - Caso necessário, busca dos dados que estão na memória e armazenamento dos mesmos em registradores

5 - Execução da instrução

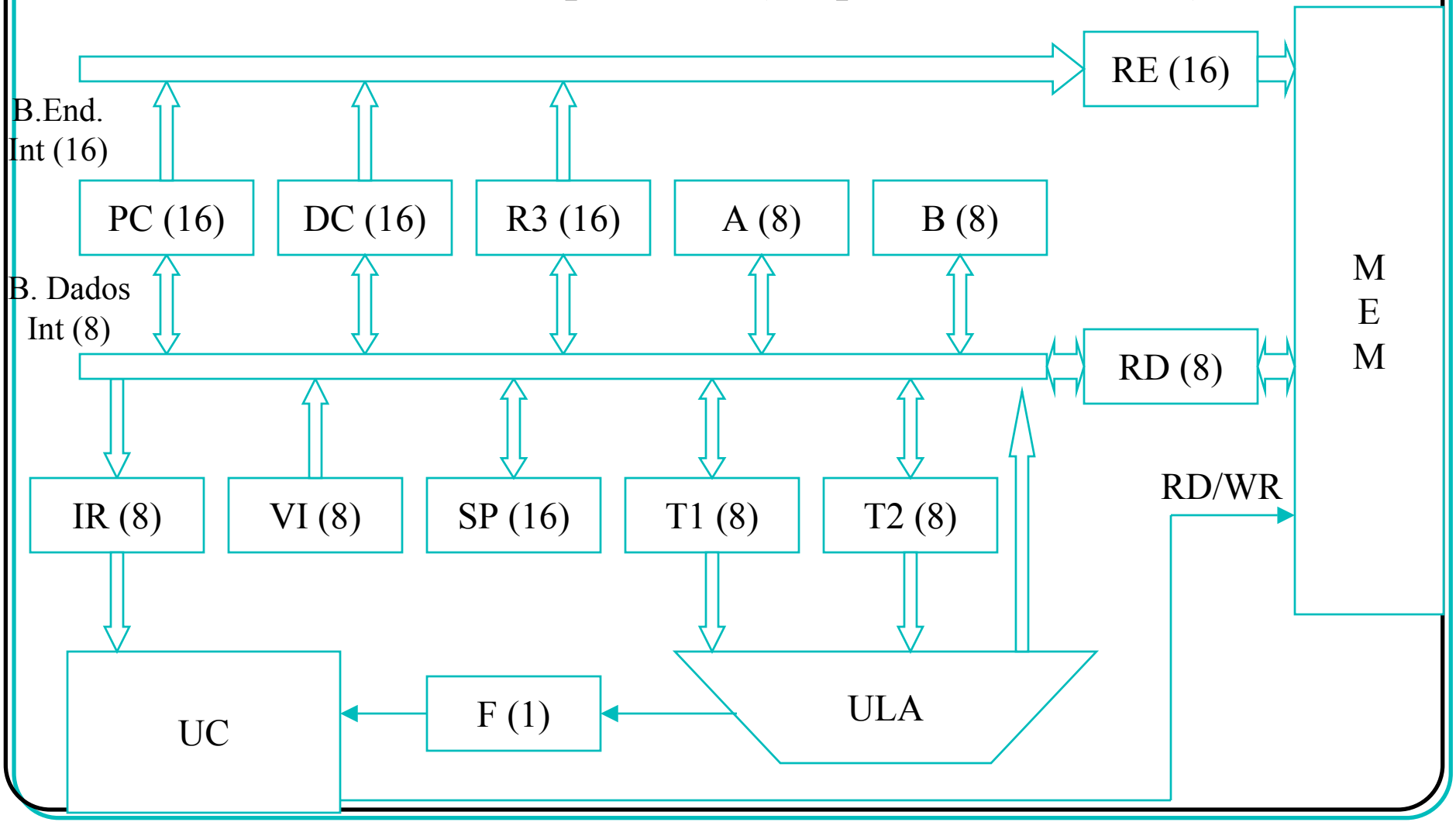
6 - Caso necessário, armazenamento do resultado na memória

Execução de Instruções

- Unidade de controle “dispara” cada um dos passos
- Registradores armazenam temporariamente dados e instruções
- Unidade lógica e aritmética “trata” os dados e permite a atualização dos apontadores

Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)

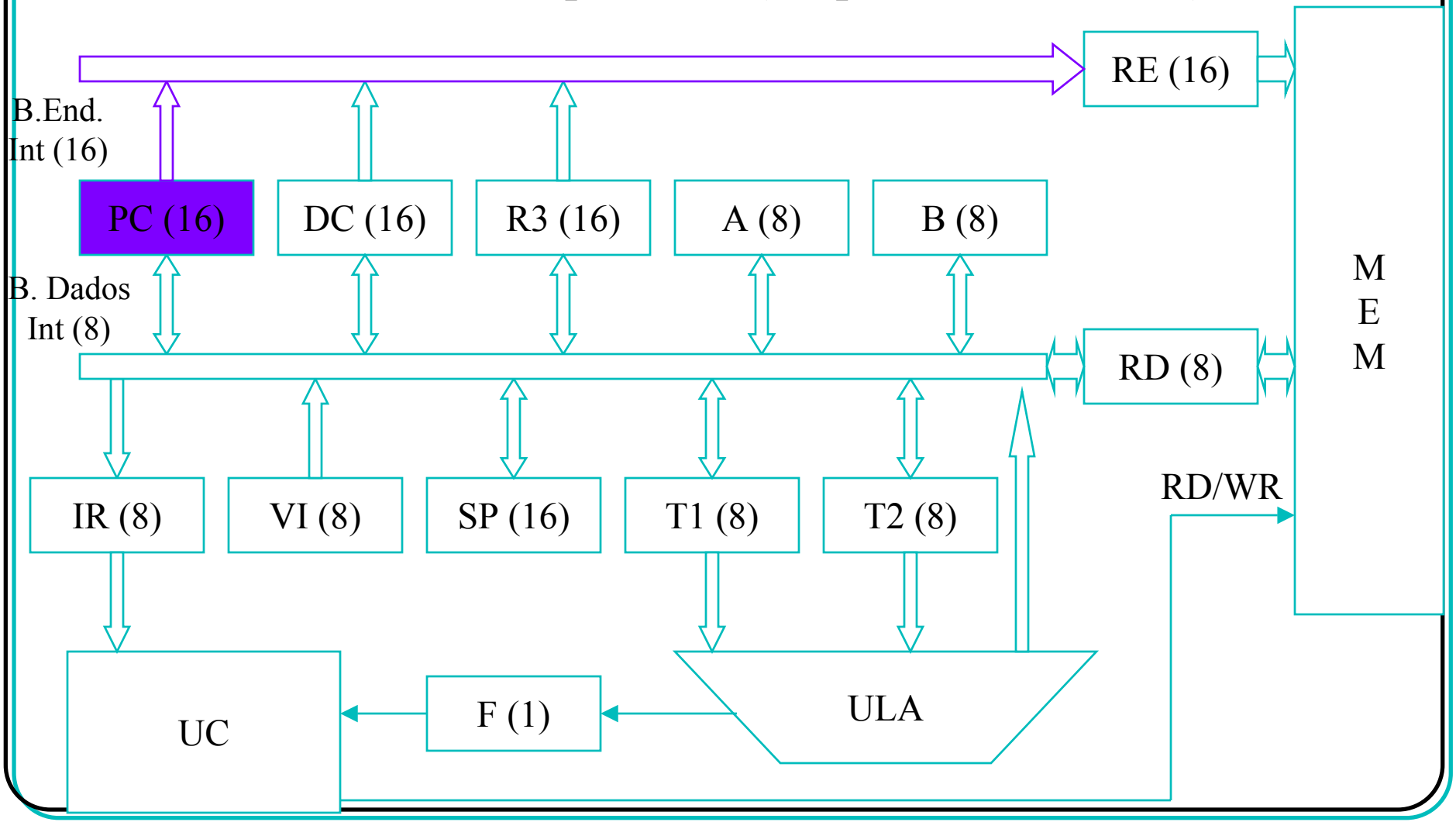


Processador (cont.)

- Busca na memória

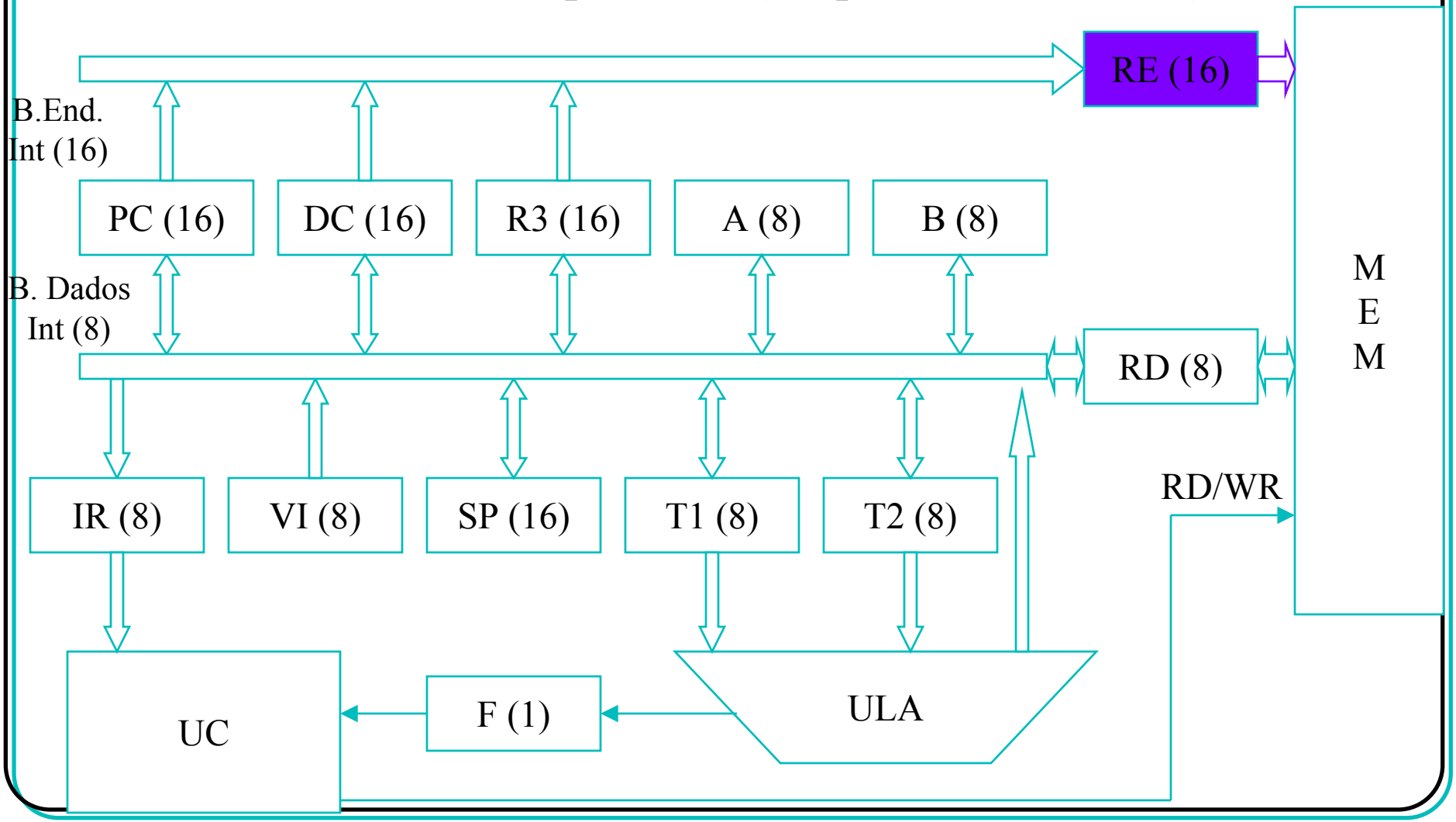
Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)



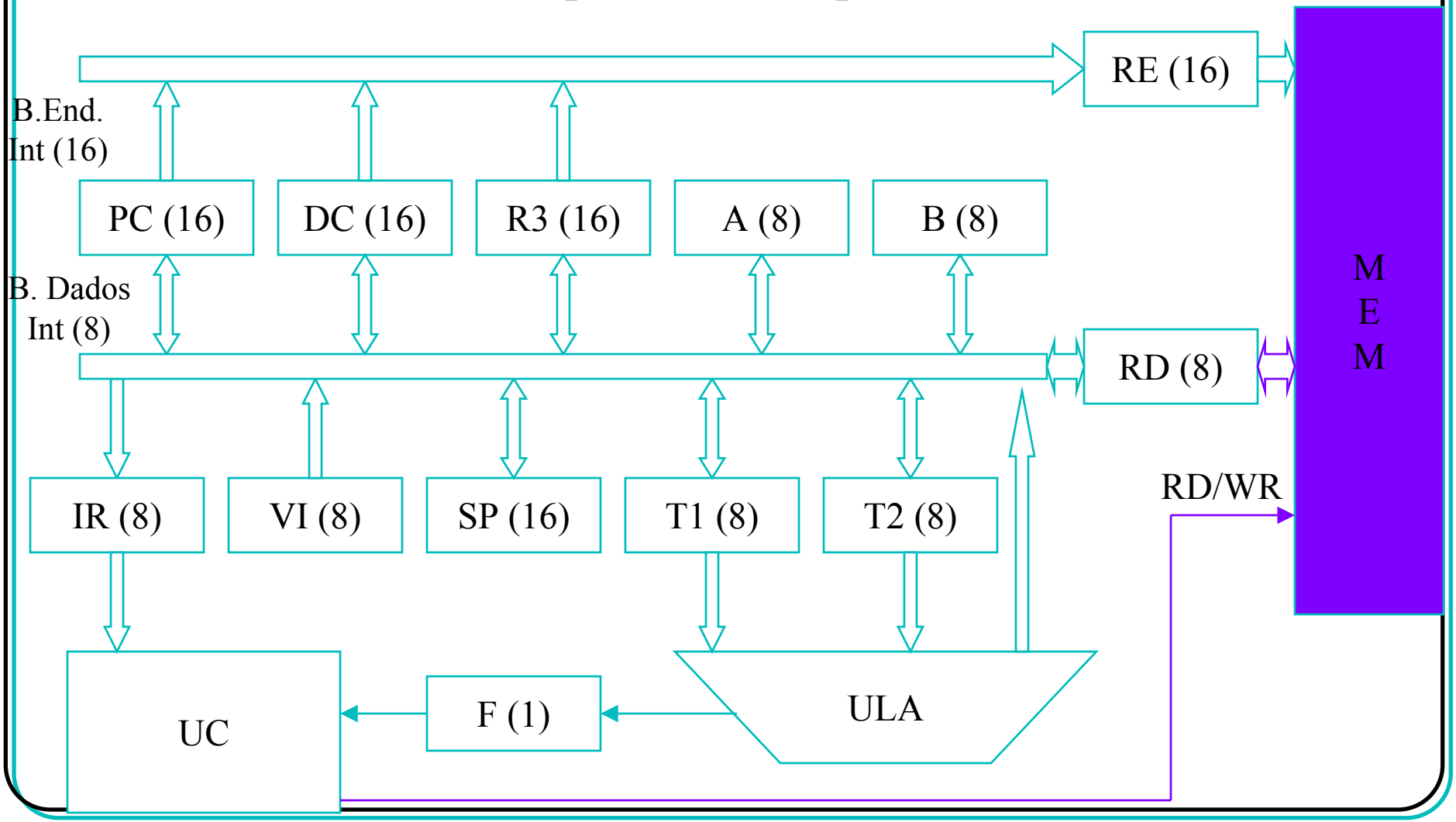
Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)



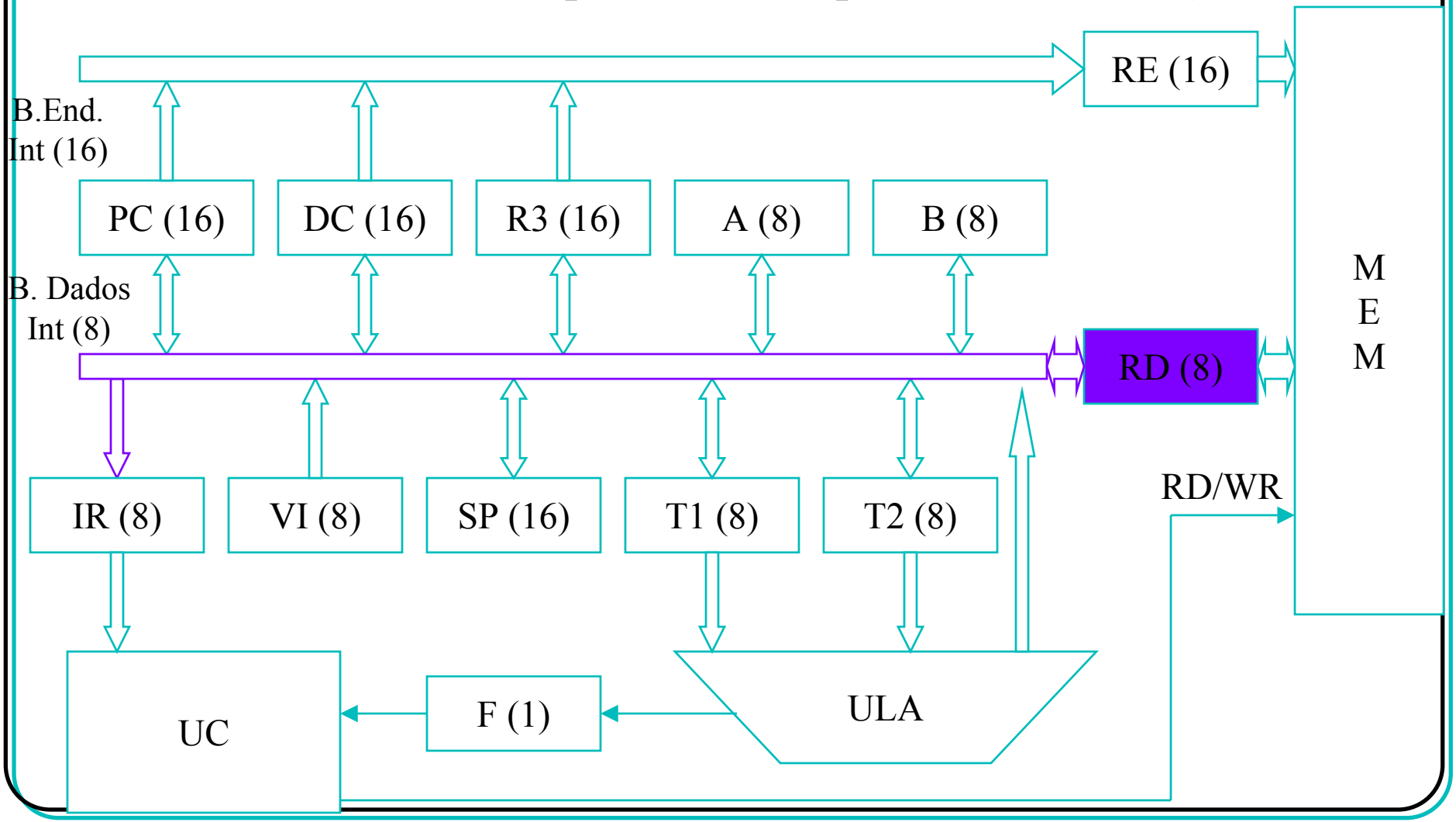
Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)



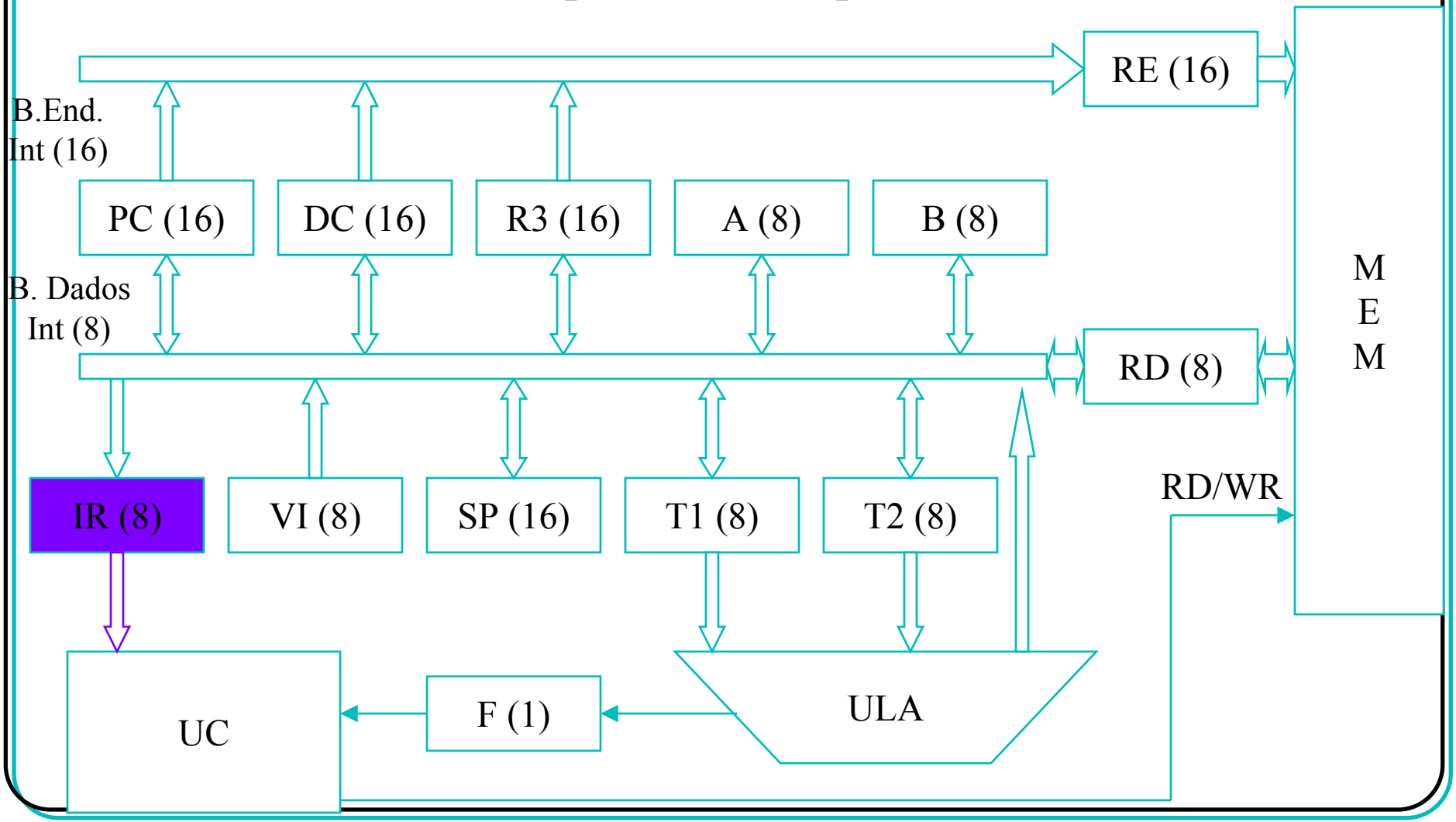
Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)



Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)

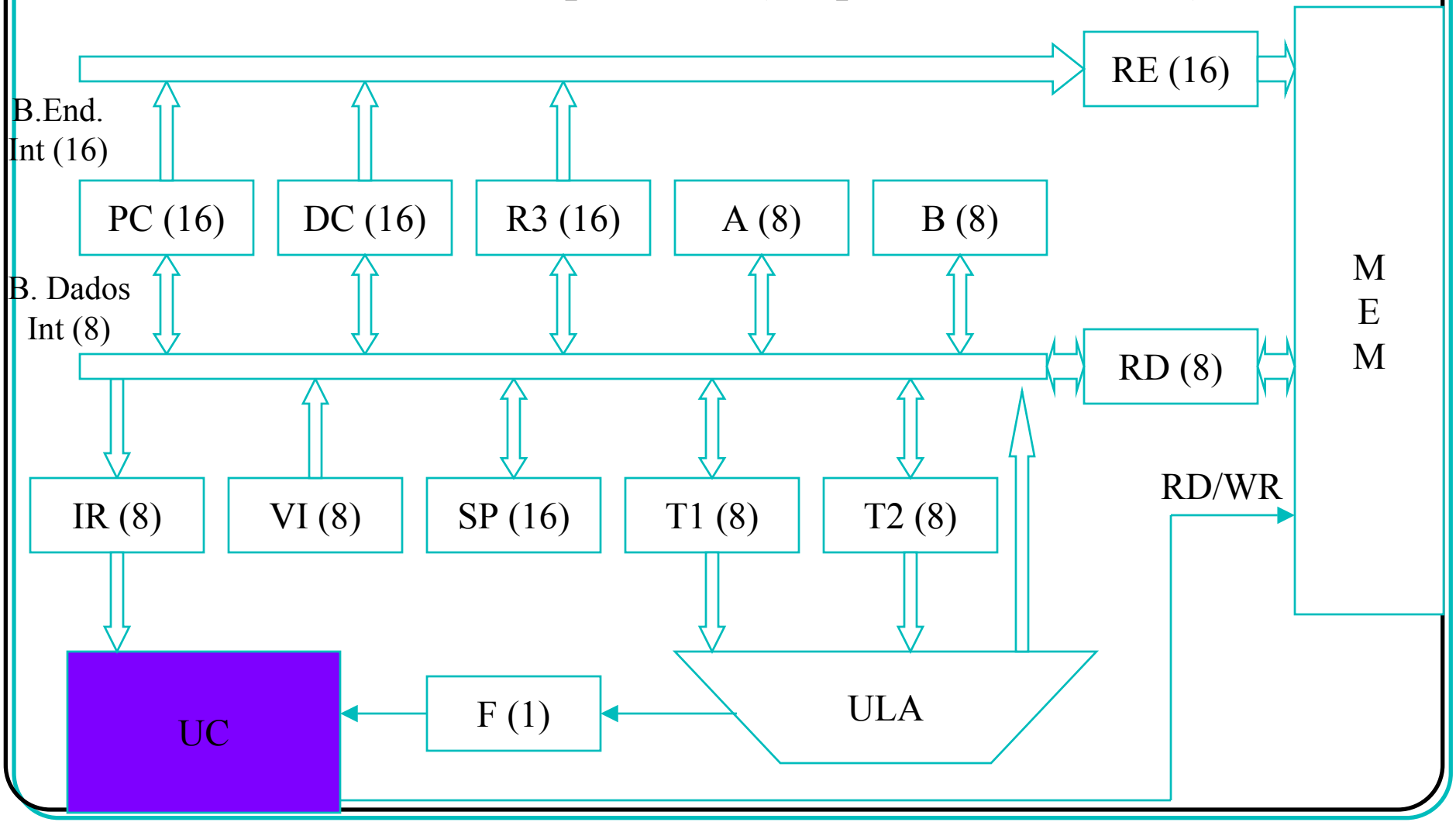


Processador (cont.)

- Determinação do tipo de instrução do IR

Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)

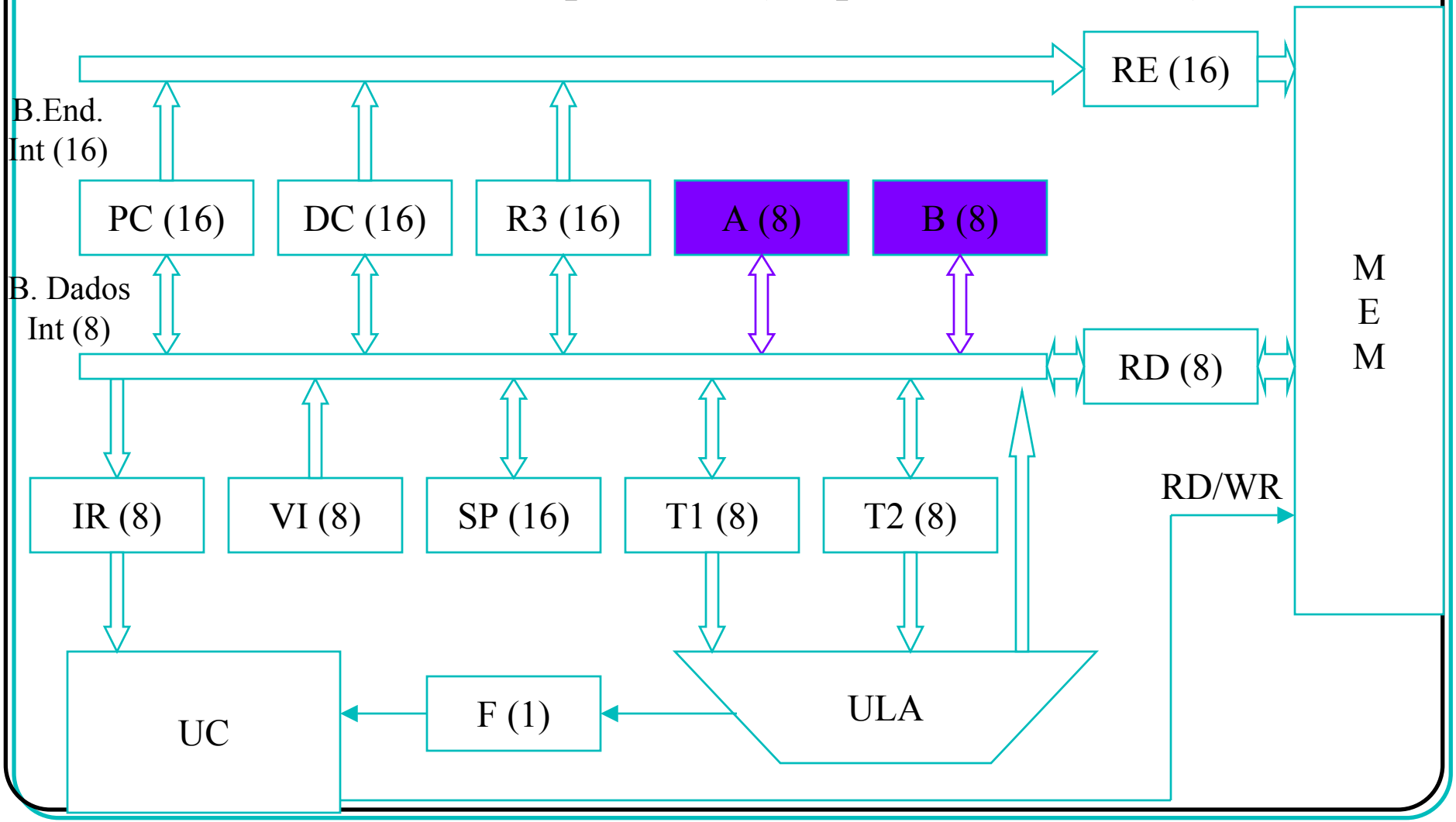


Processador (cont.)

- Execução da instrução

Processador (cont.)

CPU 8080 simplificada (adaptado de Brafman)



Processador (cont.)

- Distinção entre CPUs
  - Registradores (número, tamanho)
  - ULA (número de operações, tamanho)
  - UC (máquina de estado convencional, microprogramada)
- Tamanhos do B. Dados, do B. interno e da ULA definem a categoria do processador
  - 8 bits: 8085, Z80, 8031
  - 16 bits: 8086, 8088 (B. Dados de 8 bits), 68000

Tamanhos de Tipos de Dados Comuns

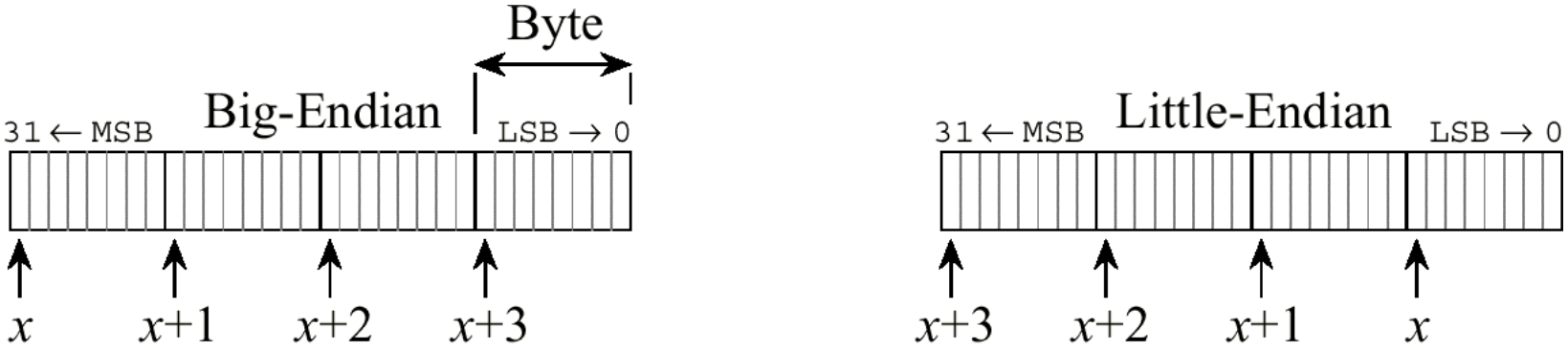
- **Um byte (ou octeto) é composto de 8 bits. Dois *nibbles* formam um byte.**
- **meias-palavras, palavras *(words),* palavras duplas *(doublewords)*, e *quadwords* são compostas de bytes como abaixo:**

Bit	0
Nibble	0110
Byte	10110000
16-bit word (halfword)	11001001 01000110
32-bit word	10110100 00110101 10011001 01011000
64-bit word (double)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101
128-bit word (quad)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001

Formatos *Big-Endian* e *Little-Endian*

- **Numa máquina endereçável por byte, o menor dado que pode ser referenciado na memória é o byte. Palavras de múltiplos bytes são armazenadas como uma seqüência de bytes, na qual o endereço da palavra na memória é o mesmo do byte da palavra que possui o *menor* endereço.**

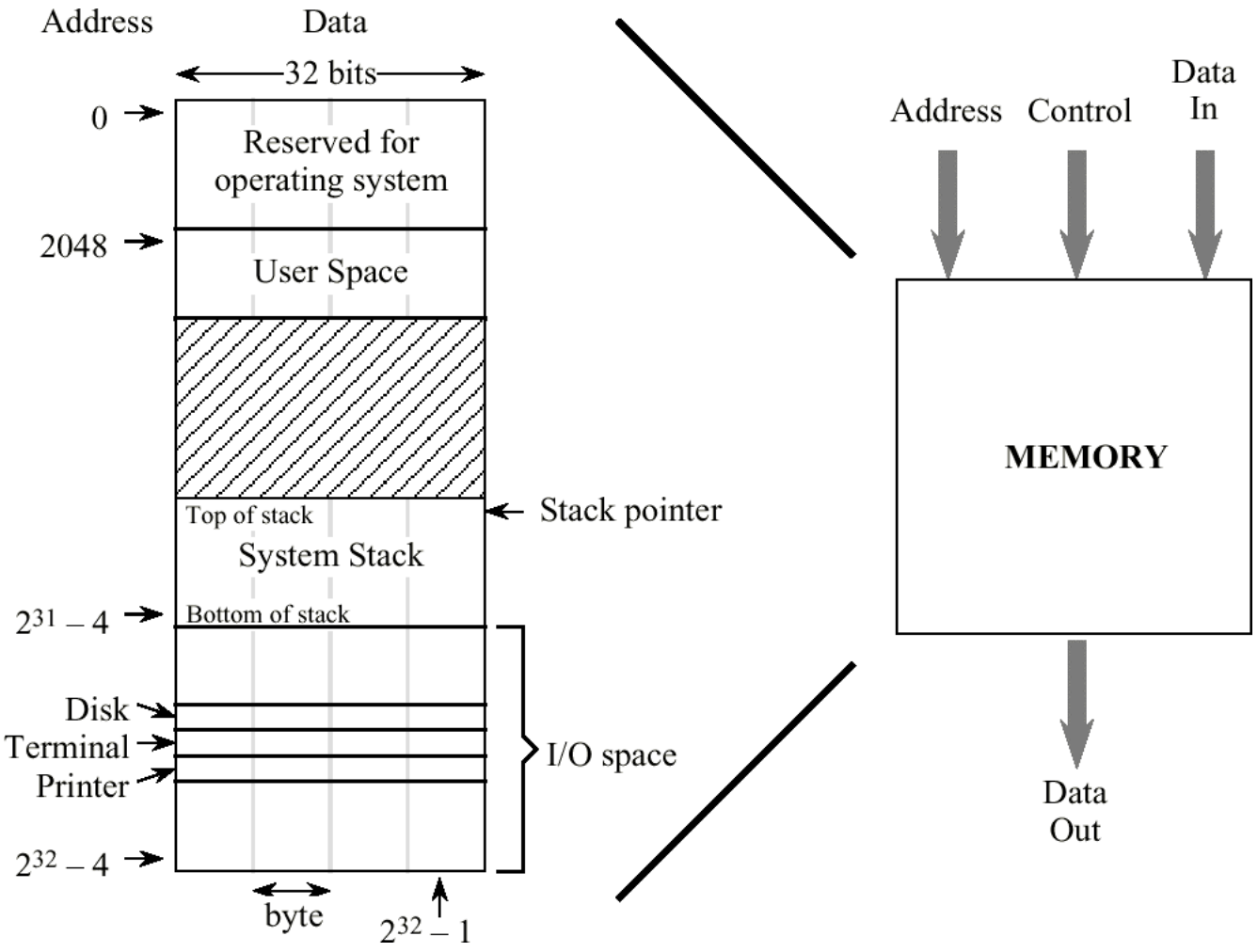
- **Quando palavras de múltiplos bytes são usadas, existem duas escolhas para a ordem na qual os bytes são armazenados na memória: o byte *mais significativo* no endereço *mais baixo*, chamada *big-endian*, ou o byte *menos significativo* armazenado no endereço *mais baixo*, chamada *little-endian*.**



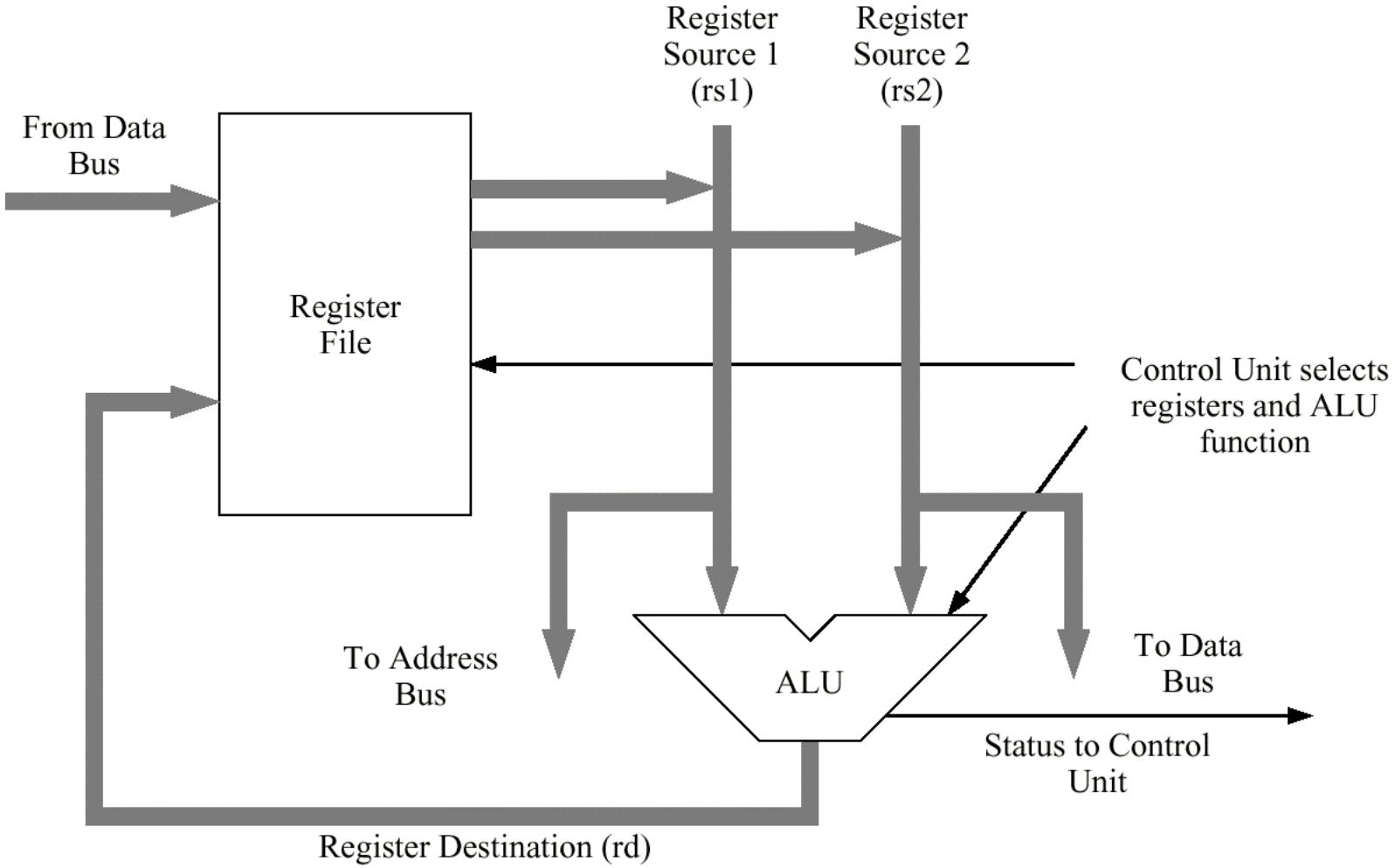
Word address is x for both big-endian and little-endian formats.

Mapa de Memória do ARC

- **As localizações na memória são arrumadas linearmente em ordem. Cada localização numerada corresponde a uma palavra ARC. O número único que identifica cada palavra é o seu *endereço*.**



Um Exemplo de Caminho de Dados



- **O caminho de dados do ARC possui uma coleção de registradores, conhecido como *register file,* e uma unidade aritmética e lógica (ALU).**

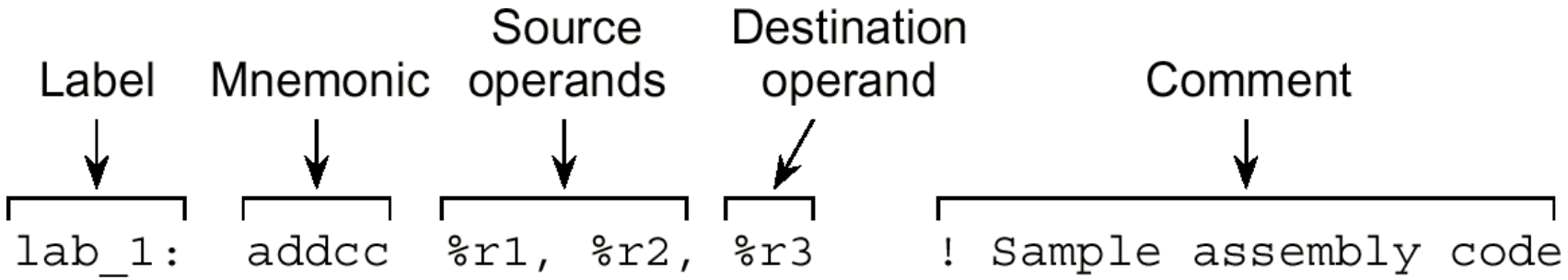
O ISA ARC

- **O ISA ARC é um sub-conjunto do ISA SPARC.**

	Mnemonic	Meaning
Memory	**ld**	Load a register from memory
	st	Store a register into memory
Logic	**sethi**	Load the 22 most significant bits of a register
	andcc	Bitwise logical AND
	orcc	Bitwise logical OR
	orncc	Bitwise logical NOR
	srl	Shift right (logical)
Arithmetic	**addcc**	Add
Control	**call**	Call subroutine
	jmpl	Jump and link (return from subroutine call)
	be	Branch if equal
	bneg	Branch if negative
	bcs	Branch on carry
	bvs	Branch on overflow
	ba	Branch always

Formato da Linguagem Assembly ARC

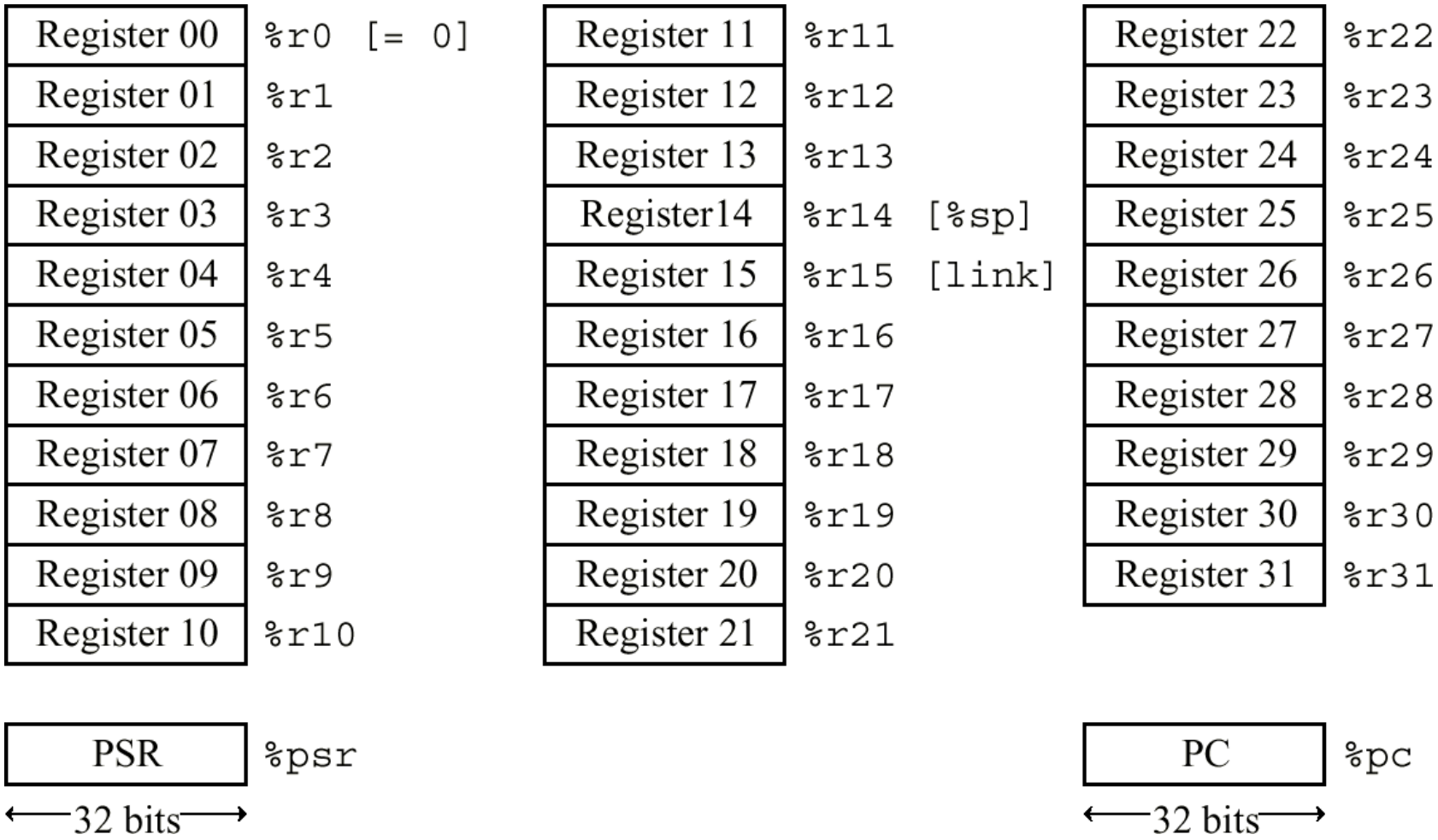
- **O formato da linguagem assembly ARC é o mesmo da linguagem assembly SPARC.**



Formato do Assembly ARC

- Rótulo: opcional
  - não pode começar por número
  - termina por dois pontos (:)
- Mnemônico
  - campo de código da operação
- Operandos origem (se houver operando)
  - Um ou mais campos separados por vírgula
- Operando destino (se houver operando)
- Comentário (opcional)
  - Começa por exclamação (!)

Registradores do ARC Visíveis pelo Usuário



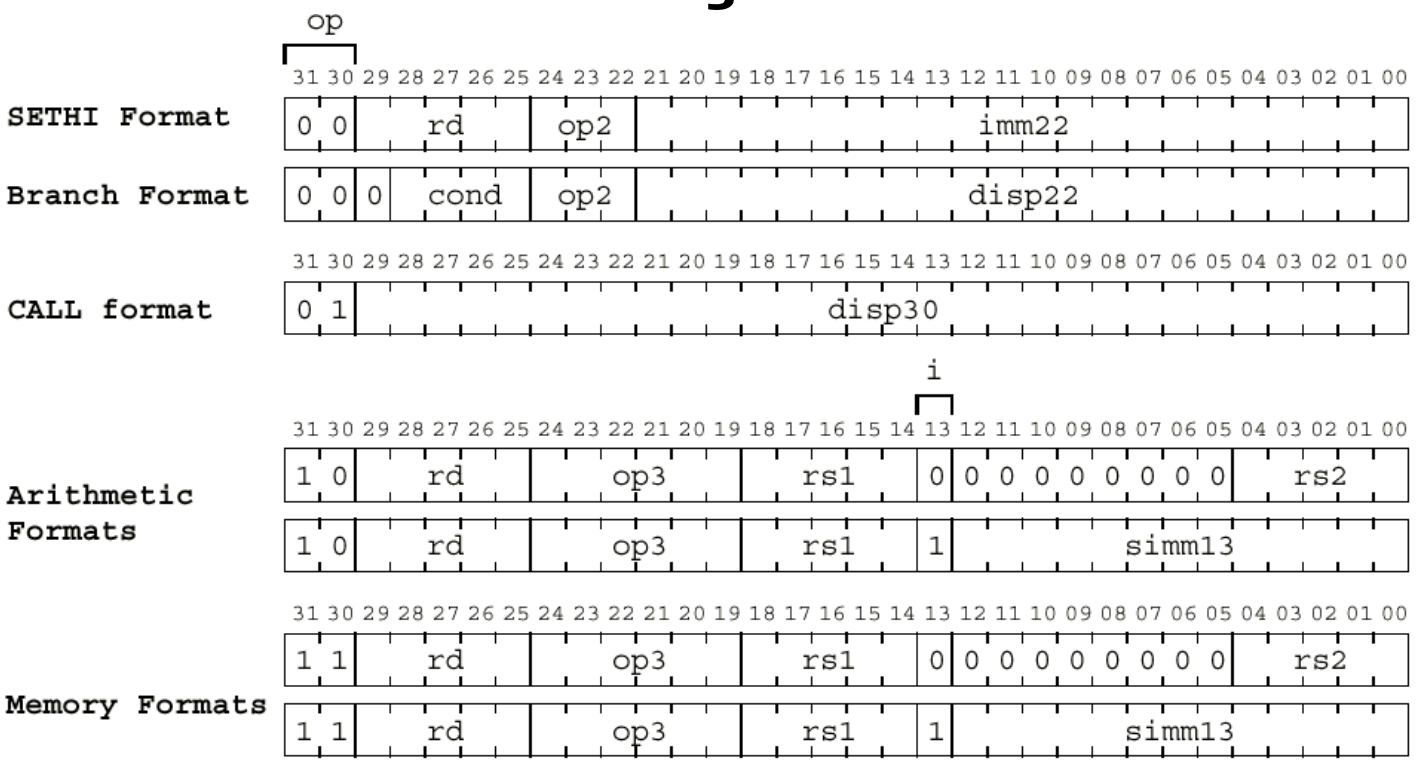
Registradores do ARC

- PC – Program Counter
  - Endereço da próxima instrução a ser executada
- IR – Instruction Register
  - Instrução em execução

- PSR – Processor Status Register
  - Informações sobre o estado do processador
  - Informações a respeito do resultado de operações aritméticas

  - Códigos de condição
  - z – valor zero
  - n – valor negativo
  - c – excedente da ALU de 32 bits (*carry* ou vai-um)
  - v - *overflow*

Formato de Instruções e do PSR ARC



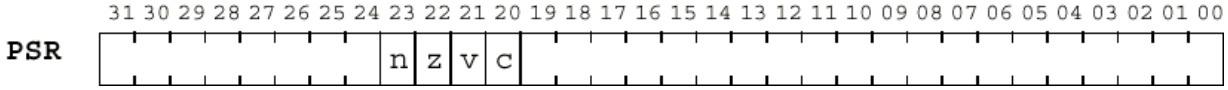
op	Format
00	SETHI/Branch
01	CALL
10	Arithmetic
11	Memory

op2	Inst.
010	branch
100	sethi

op3 (op=10)	
010000	addcc
010001	andcc
010010	orcc
010110	orncc
100110	srl
111000	jmpl

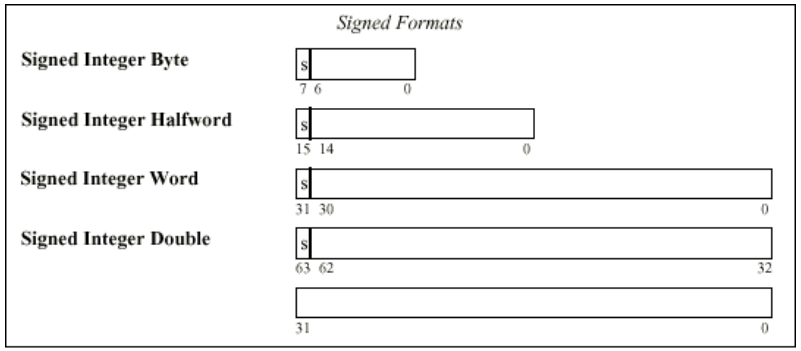
op3 (op=11)	
000000	ld
000100	st

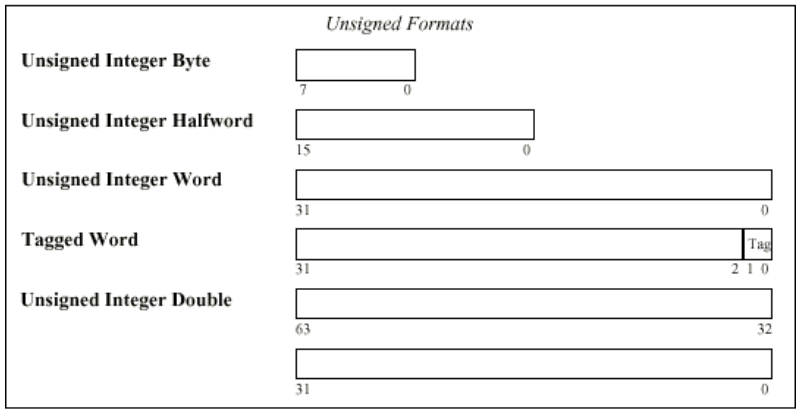
cond	branch
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba

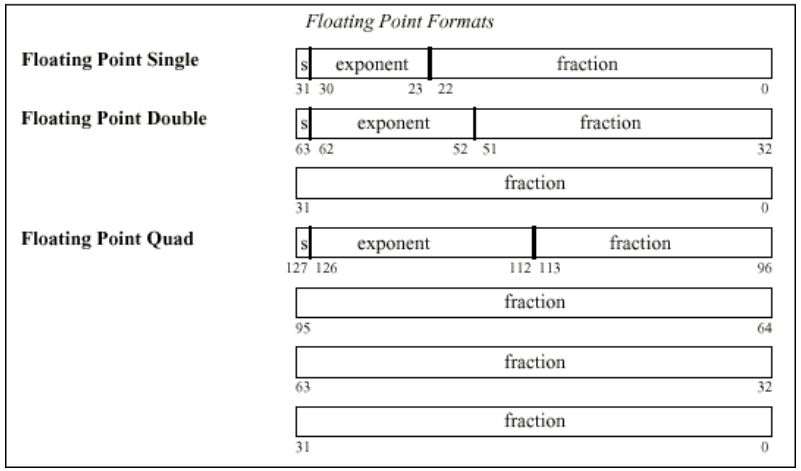


Formatos de Instruções do ARC

- Formato SETHI e de Desvio
  - rd – registrador de destino no SETHI
  - cond – identifica o tipo de desvio, baseado nos bits do PSR
  - Instruções com “cc” no final do mnemônico alteram o PSR
- Formato de Chamada (CALL)
  - disp30 – tamanho do desvio para endereço da rotina chamada
- Formato aritmético
- Formato memória
  - rd – origem para st, destino para outras instruções
  - rs1, rs2 – registrador origem
  - simm3 – constante como segunda origem em vez de rs2







Formato de Dados ARC

ARC Pseudo-Ops

Pseudo-Op	Usage	Meaning
.equ X	.equ #10	Treat symbol X as $(10)_{16}$
.begin	.begin	Start assembling
.end	.end	Stop assembling
.org	.org 2048	Change location counter to 2048
.dwb	.dwb 25	Reserve a block of 25 words
.global	.global Y	Y is used in another module
.extern	.extern Z	Z is defined in another module
.macro	.macro M a, b, ...	Define macro M with formal parameters a, b, ...
.endmacro	.endmacro	End of macro definition
.if	.if <cond>	Assemble if <cond> is true
.endif	.endif	End of .if construct

- **Pseudo-ops são instruções para o assembler, não são parte do ISA.**

Exemplo de Programa ARC

- **Um programa em assembly ARC que soma dois inteiros:**

```
! This programs adds two numbers
        .begin
        .org 2048
prog1:  ld      [x], %r1            ! Load x into %r1
        ld      [y], %r2            ! Load y into %r2
        addcc   %r1, %r2, %r3       ! %r3 ← %r1 + %r2
        st      %r3, [z]            ! Store %r3 into z
        jmpl    %r15 + 4, %r0       ! Return
x:      15
y:      9
z:      0
        .end
```

Programa Um Pouco Mais Complexo

- **Um programa ARC que soma 5 inteiros.**

```
! This program sums LENGTH numbers
! Register usage:        %r1 - Length of array a
!                        %r2 - Starting address of array a
!                        %r3 - The partial sum
!                        %r4 - Pointer into array a
!                        %r5 - Holds an element of a
            .begin                ! Start assembling
            .org  2048            ! Start program at 2048
a_start     .equ  3000            ! Address of array a
            ld    [length], %r1   ! %r1 ← length of array a
            ld    [address],%r2   ! %r2 ← address of a
            andcc %r3, %r0, %r3   ! %r3 ← 0
loop:       andcc %r1, %r1, %r0   ! Test # remaining elements
            be    done            ! Finished when length=0
            addcc %r1, -4, %r1    ! Decrement array length
            addcc %r1, %r2, %r4   ! Address of next element
            ld    %r4, %r5        ! %r5 ← Memory[%r4]
            addcc %r3, %r5, %r3   ! Sum new element into r3
            ba    loop            ! Repeat loop.

done:       jmpl  %r15 + 4, %r0   ! Return to calling routine

length:       20                  ! 5 numbers (20 bytes) in a
address:      a_start
            .org  a_start         ! Start of array a
a:             25                 ! length/4 values follow
              -10
               33
               -5
                7
            .end                        ! Stop assembling
```

Instruções de 1, 2 ou 3 Endereços

- Considere como a expressão A = B*C + D pode ser calculada por tipos de instrução de um, dois ou três endereços.

- Hipóteses:
  - Endereços e palavras de dados possuem dois bytes.
  - Opcodes tem tamanho de 1 byte.
  - Operandos são movidos para e da memória uma palavra (dois bytes) de cada vez.

Instruções de 3 Endereços

- Usando instruções de 3 endereços, a expressão A = B*C + D pode ser codificada como:

```
mult   B, C, A
add    D, A, A
```

Ou seja, multiplique B por C e armazene o resultado em A. (As operações `mult` e add são genéricas; não são instruções ARC.)

Então, some D a A e armazene o resultado em A.

Instruções de 3 Endereços

- O tamanho do programa é 7x2 = 14 bytes
  - cada instrução = 1 + 2 + 2 + 2 = 7 bytes
    - opcode + operando + operando + operando
- Tráfego da memória
  - Para cada instrução:
  - busca da instrução (7 bytes) + tráfego de dados (6 bytes)
  - Total = (7+6) + (7+6) = 26 bytes

Instruções de 2 endereços

- Em uma instrução de 2 endereços, um dos operandos é sobrescrito com o resultado. Um código possível para o cálculo da expressão A = B*C + D é:

```
load   B, A
mult   C, A
add    D, A
```

O tamanho do programa agora é: 3 × (1 + 2 + 2), ou 15 bytes.

Tráfego de memória:

busca das instruções: 5 por instrução

primeira inst.: 2 × 2 bytes buscados

outras 2 instruções: 2 × 2 bytes buscados, 2 bytes armazenados.

total: (5 + 4) + (5 + 6) + (5 + 6) = 31 bytes

Instruções de 1 endereço

- Ou de Acumulador: Instruções de 1 endereço utilizam um único registrador aritmético na CPU, conhecido como o acumulador. O código para a expressão A = B*C + D agora é:

```
load   B
mult   C
add    D
store  A
```

A instrução load carrega B no acumulador;

mult multiplica C pelo acumulador e armazena o resultado no acumulador;

e add realiza a soma correspondente.

A instrução store armazena o valor do acumulador em A.

Instruções de 1 endereço

- O tamanho do programa é 3 x 4 = 12 bytes
  - cada instrução = 1 + 2 = 3 bytes
    - opcode + operando
- Tráfego da memória
  - Cada instrução gera apenas 2 bytes de cada vez, sendo buscados OU armazenados na memória
  - Total = (3+2) + (3+2) + (3+2) + (3+2) = 20 bytes

Modos de Endereçamento

Addressing Mode	Syntax	Meaning
Immediate	#K	K
Direct	K	M[K]
Indirect	(K)	M[M[K]]
Register	(Rn)	M[Rn]
Register Indexed	(Rm + Rn)	M[Rm + Rn]
Register Based	(Rm + X)	M[Rm + X]
Register Based Indexed	(Rm + Rn + X)	M[Rm + Rn + X]

- **Quatro formas de calcular o endereço de um valor na memória: (1) uma constante conhecida em tempo de montagem, (2) o conteúdo de um registrador, (3) a soma de dois registradores, (4) a soma de um registrador e de uma constante. A tabela dá nomes a estes e outros modos de endereçamento.**

Ligação de Sub-rotinas

- Processo de passar argumentos (e valores de retorno) entre rotinas
- Convenções de chamada
  - Diferentes maneiras de passagem de argumentos
  - Registradores
  - Área de ligação de dados
  - Pilha

Ligação de Sub-rotinas por Registradores

- **Passagem de parâmetros através de registradores.**

```
! Calling routine
      .
      .
      .
     ld      [x], %r1
     ld      [y], %r2
     call    add_1
     st      %r3, [z]
      .
      .
      .
x:   53
y:   10
z:    0
```

```
! Called routine
! %r3 ← %r1 + %r2


add_1: addcc   %r1, %r2, %r3
       jmpl    %r15 + 4, %r0
```

Ligação de Sub-rotinas por Registradores

- Método rápido e simples
- Mas não funciona se
  - Número de argumentos excede o número de registradores livres
  - Chamadas de subrotina alinhadas em profundidade

Ligação de Sub-rotinas por Área de Ligação de Dados

- **Passagem de parâmetros numa área de memória separada. O endereço inicial da área de memória é passado em um registrador (%r5 aqui).**

```
! Calling routine
      .
      .
      .
    st     %r1, [x]
    st     %r2, [x+4]
    sethi  x, %r5
    srl    %r5, 10, %r5
    call   add_2
    ld     [x+8], %r3
      .
      .
      .
! Data link area
x:  .dwb   3
```

```
! Called routine
! x[2] ← x[0] + x[1]


add_2:  ld     %r5, %r8
        ld     %r5 + 4, %r9
        addcc  %r8, %r9, %r10
        st     %r10, %r5 + 8
        jmpl   %r15 + 4, %r0
```

Ligação de Sub-rotinas por Área de Ligação de Dados

- Blocos de dados arbitrariamente grandes podem ser passados entre rotinas
- Um único registrador é copiado na chamada da sub-rotina

- Problemas
- Recursão
  - Uma rotina que chama a si mesma precisa de várias áreas de ligação de dados
- Tamanho da área de ligação
  - Deve ser conhecido em tempo de montagem

Ligação de Sub-rotinas por Pilha

- Pilha: “o último a entrar é o primeiro a sair”
- Idéia básica
  - Rotina principal coloca os argumentos na pilha
  - Rotina chamada retira os argumentos da pilha,
    executa,
    e coloca valores de retorno na pilha
  - Rotina principal recupera os valores de retorno da pilha
  - Registrador especial aponta o topo da pilha
    - SP – *stack pointer*
- Algumas máquinas possuem instruções *push* e *pop* que *colocam* e *retiram* elementos da pilha e atualizam o SP automaticamente

Ligação de Sub-rotinas por Pilha

- **Passagem de parâmetros utilizando a pilha.**

```
  ! Calling routine
       .
       .
       .
%sp  .equ   %r14
     addcc  %sp, -4, %sp
     st     %r1, %sp
     addcc  %sp, -4, %sp
     st     %r2, %sp
     call   add_3
     ld     %sp, %r3
     addcc  %sp, 4, %sp
       .
       .
       .
```

```
! Called routine
! Arguments are on stack.
! %sp[0] ← %sp[0] + %sp[4]
     %sp  .equ   %r14
add_3:  ld     %sp, %r8
        addcc  %sp, 4, %sp
        ld     %sp, %r9
        addcc  %r8, %r9, %r10
        st     %r10, %sp
        jmpl   %r15 + 4, %r0
```

Ligação de Sub-rotinas por Pilha

- **Tamanho da pilha cresce e diminui *dinamicamente***
- **Aninhamento arbitrariamente profundo de chamadas**

Exemplo de Ligação pela Pilha

- **Programa em C com chamadas de função aninhadas.**

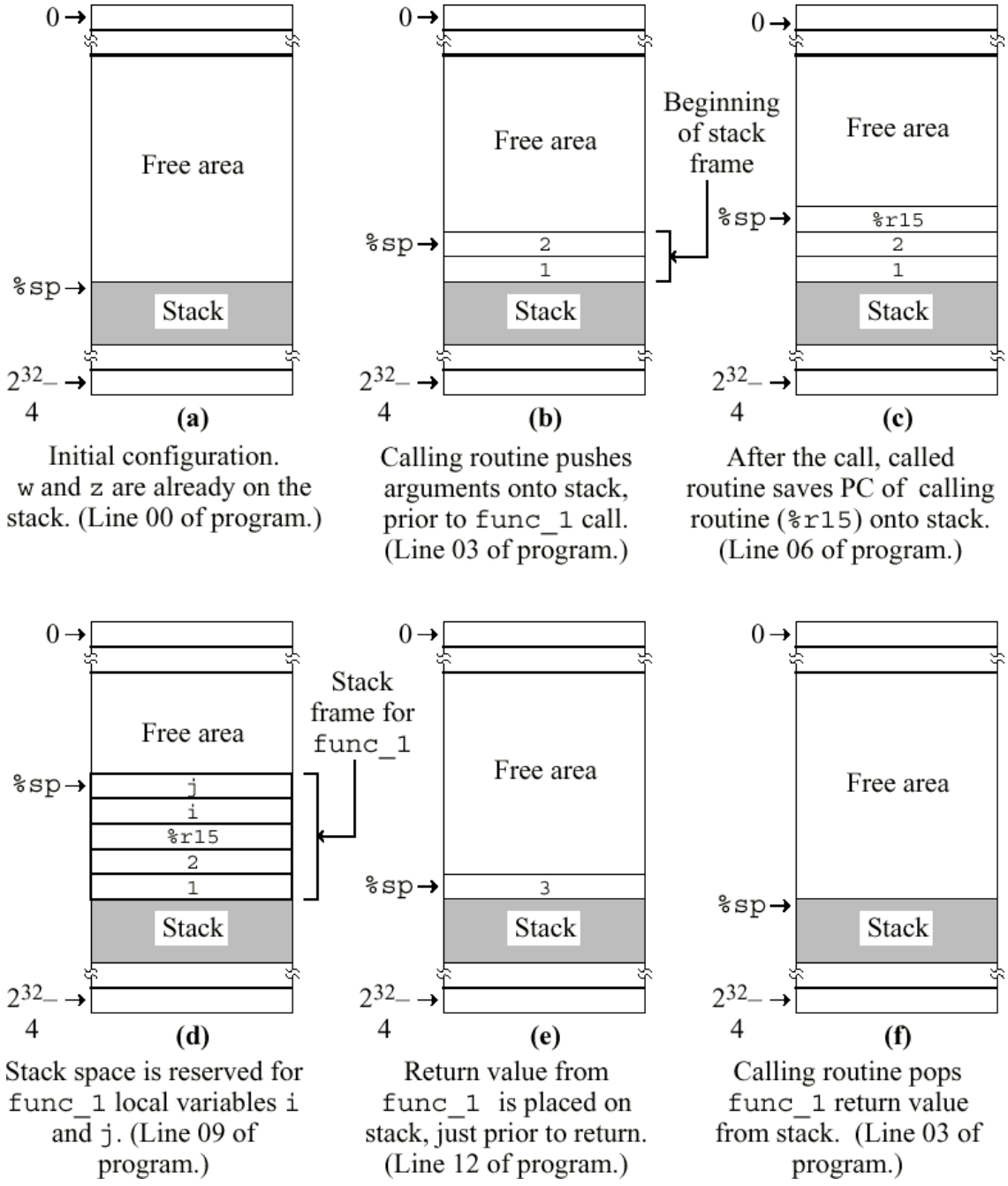
```
Line No.  /* C program showing nested subroutine calls */
00  main()
01  {
02      int w, z;           /* Local variables */
03      w = func_1(1,2);    /* Call subroutine func_1 */
04      z = func_2(10);     /* Call subroutine func_2 */
05  }                       /* End of main routine */

06  int func_1(x,y)         /* Compute x * x + y */
07  int x, y;           /* Parameters passed to func_1 */
08  {
09      int i, j;          /* Local variables */
10      i = x * x;
11      j = i + y;
12      return(j);    /* Return j to calling routine */
13  }

14  int func_2(a)           /* Compute a * a + a + 5 */
15  int a;                  /* Parameter passed to func_2 */
16  {
17      int m, n;           /* Local variables */
18      n = a + 5;
19      m = func_1(a,n);
20      return(m);          /* Return m to calling routine */
21  }
```

Aninhamento

- Quando sub-rotina termina de executar, %r15 (reg. de ligação) é usado para calcular endereço de retorno
  - Problema: e se várias chamadas são aninhadas?
- Solução
  - Valor atual de %r15 deve ser armazenado na pilha
    - Junto com quaisquer outros registradores que precisem ser restaurados
  - Conjunto dos registradores “preservados”: *Quadro de pilha*

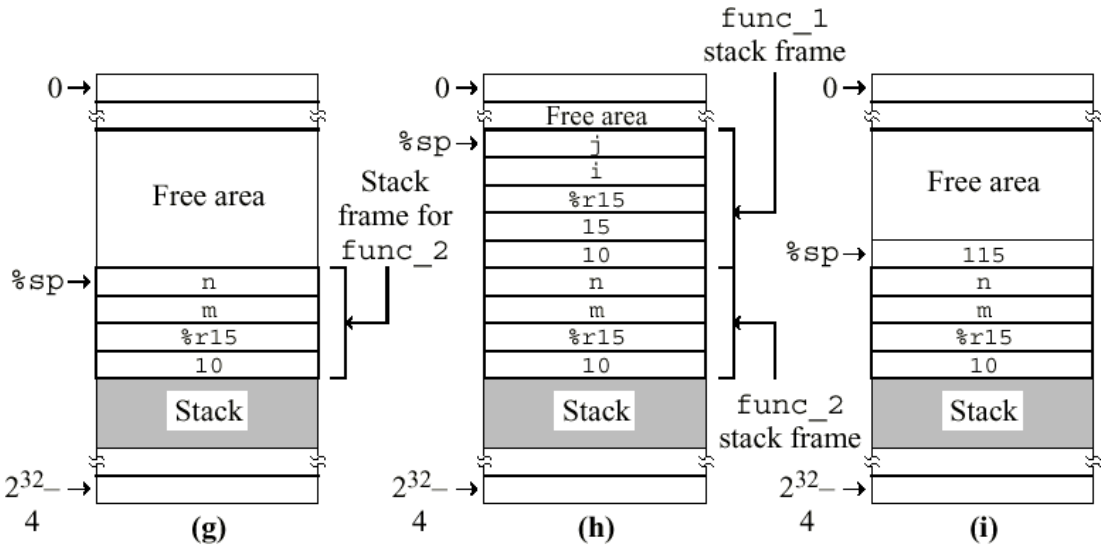
Exemplo de Ligação pela Pilha (cont.)

- **(a-f) Comportamento da pilha durante a execução do programa em C.**



Exemplo de Ligação pela Pilha (cont.)

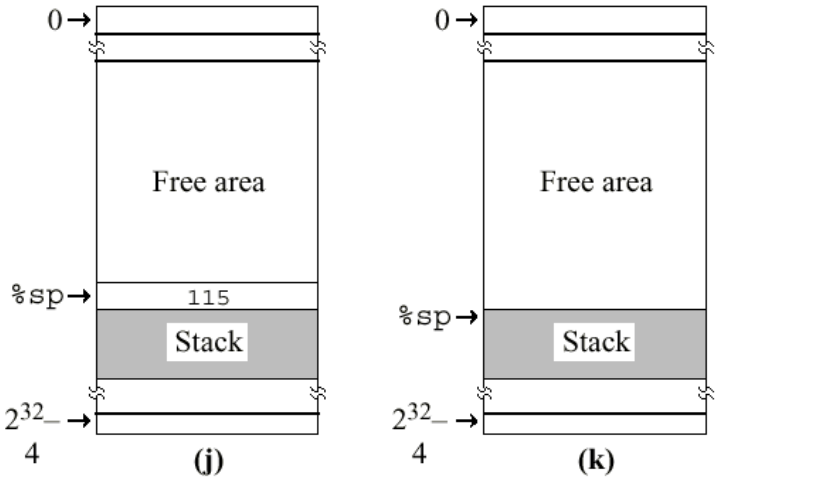
- **(g-k) Comportamento da pilha durante a execução do programa em C.**



A stack frame is created for `func_2` as a result of function call at line 04 of program.

A stack frame is created for `func_1` as a result of function call at line 19 of program.

`func_1` places return value on stack. (Line 12 of program.)



`func_2` places return value on stack. (Line 20 of program.)

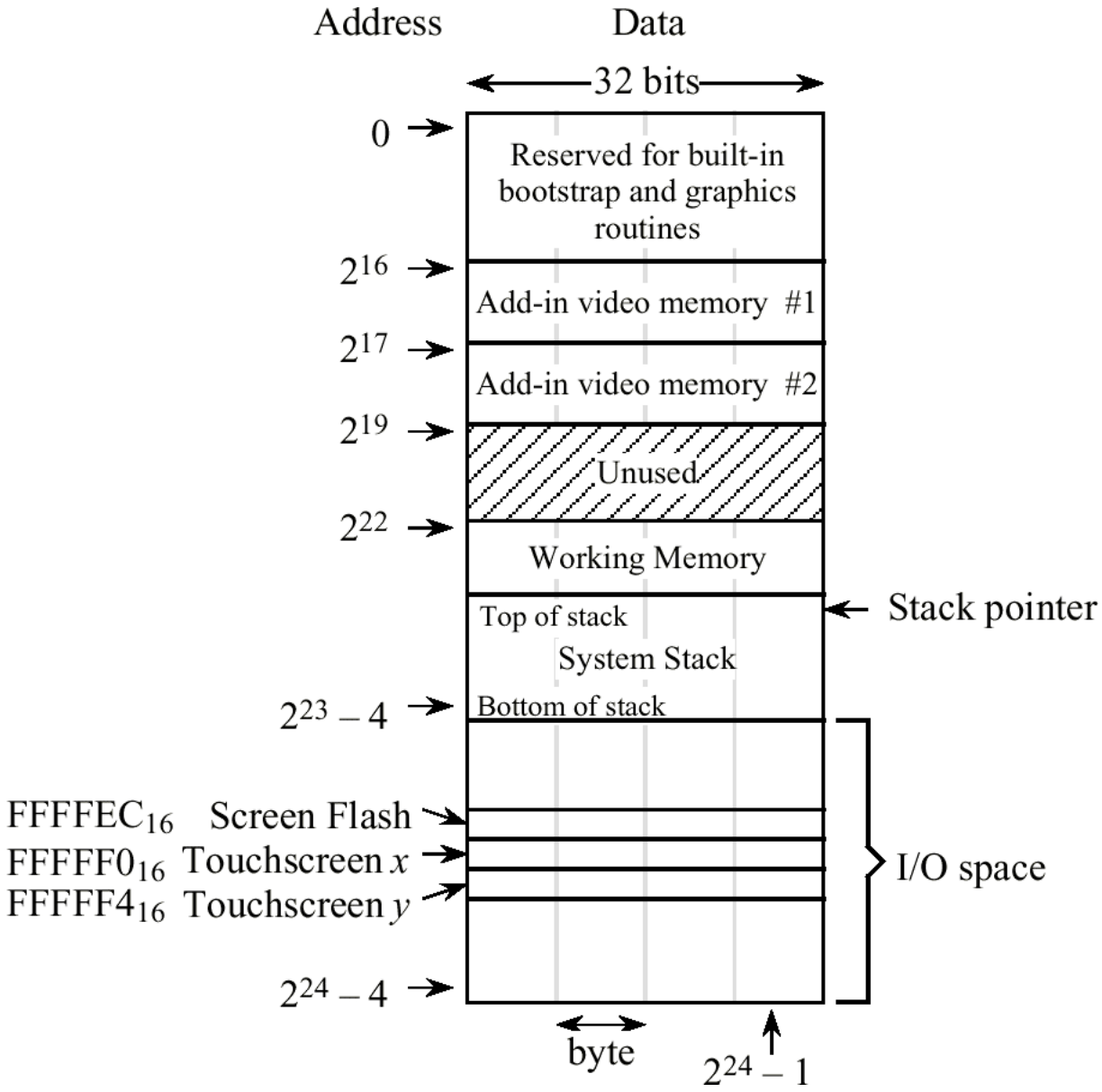
Program finishes. Stack is restored to its initial configuration. (Lines 04 and 05 of program.)

Entrada e Saída

- Duas formas
  - Instruções especiais e barramento especial de entrada e saída
  - Entrada e saída mapeada em memória
    - acesso é feito como para posições de memória, em seções do espaço de endereçamento onde não existe memória real
- ARC
  - E/S mapeada em memória

Entrada e Saída no ISA ARC

- **Mapa de memória do ARC, mostrando a E/S mapeada em memória.**

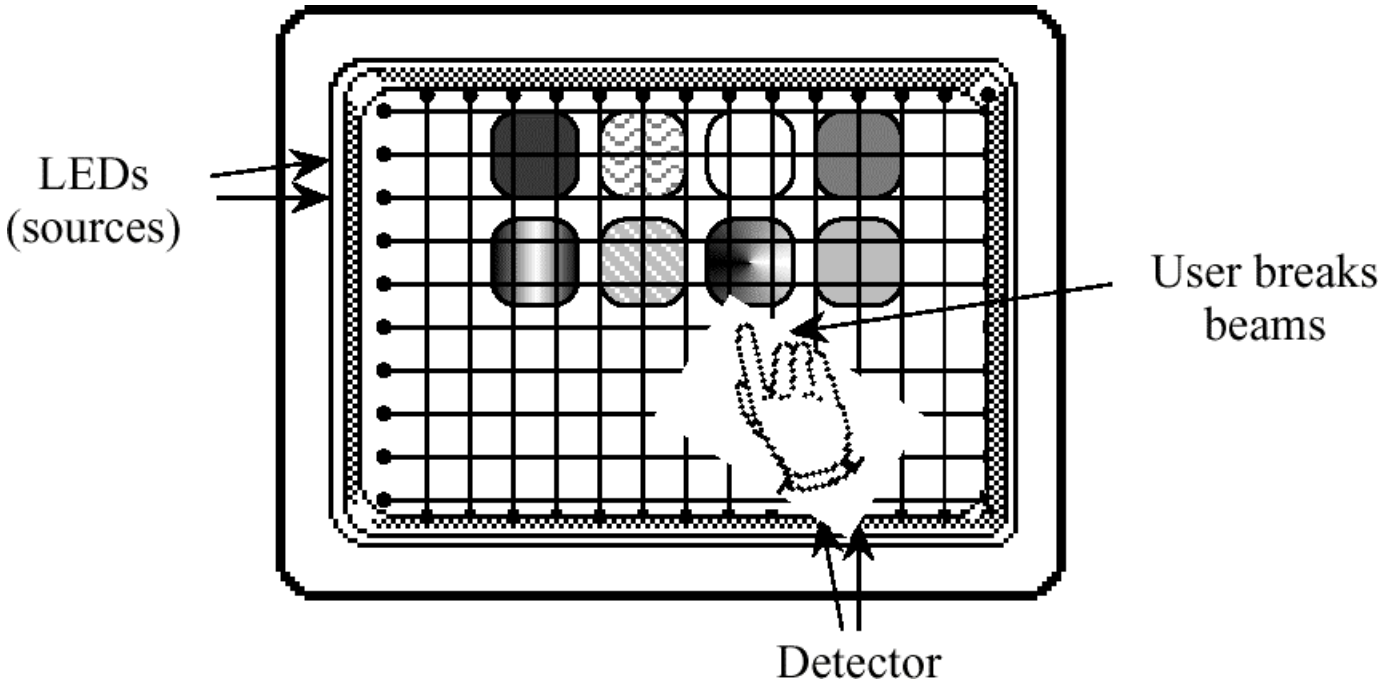


Regiões de Endereçamento do ARC

- Memória real
  - Entre os endereços 2^{22} e $2^{23} – 1$
- Rotinas de inicialização e rotinas gráficas básicas
  - Entre os endereços 0 e $2^{16} – 1$
- Módulos de vídeo
  - Entre os endereços 2^{16} e $2^{19} – 1$
- Dispositivos de E/S
  - Entre os endereços 2^{23} e $2^{24} – 1$

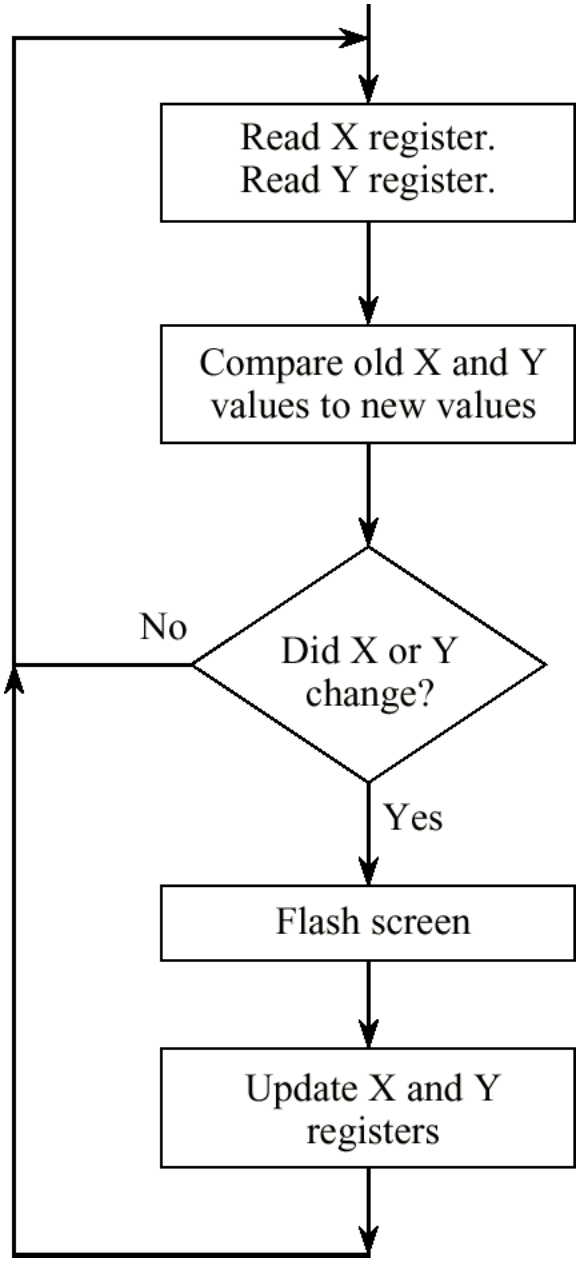
Disp. de E/S: *Touchscreen*

- **Um usuário seleciona um objeto em um *touchscreen*:**

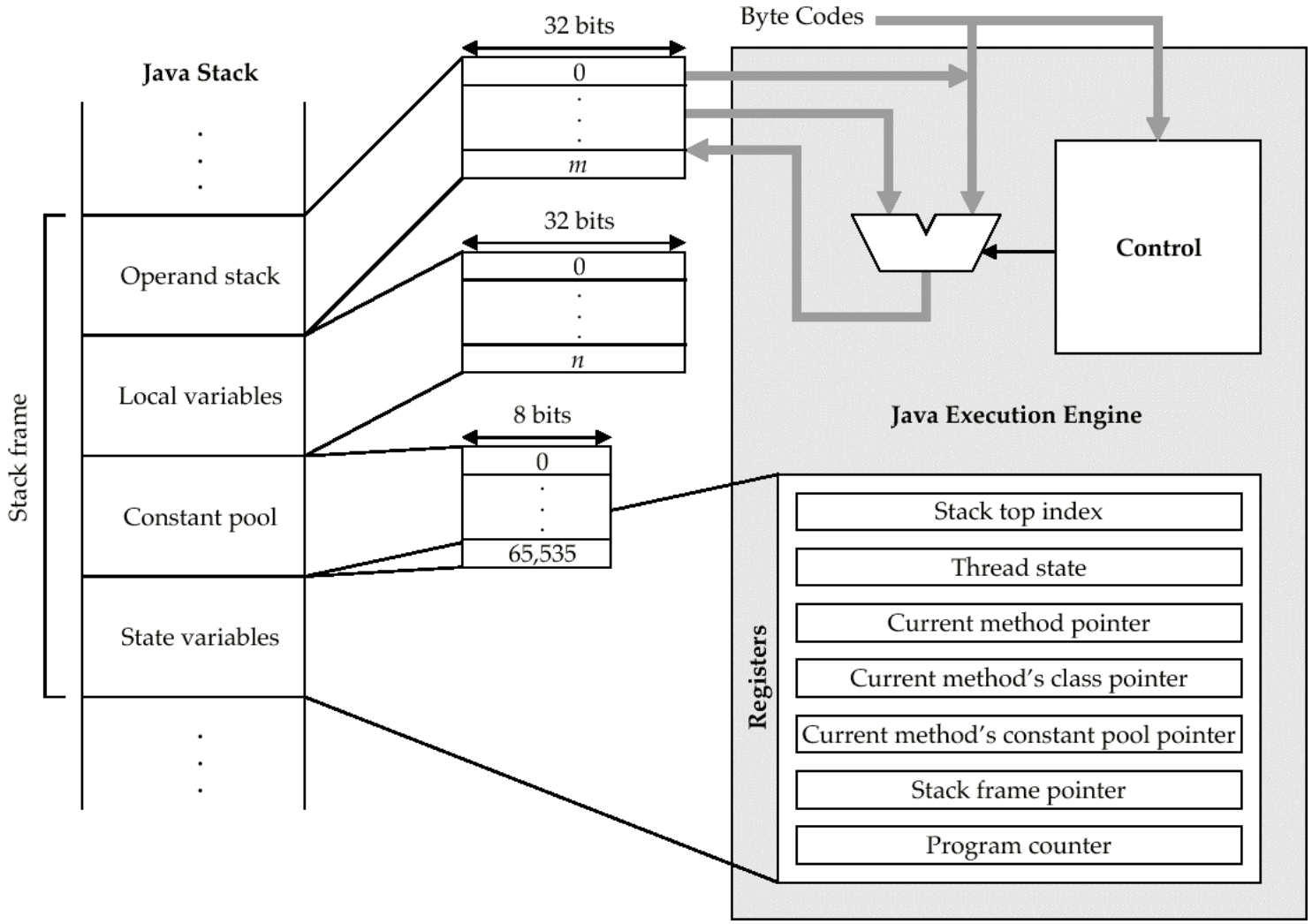


Fluxograma para Disp. de E/S

- **Estrutura de controle de um programa de rastreamento do *touchscreen*.**



Arquitetura da Máquina Virtual Java

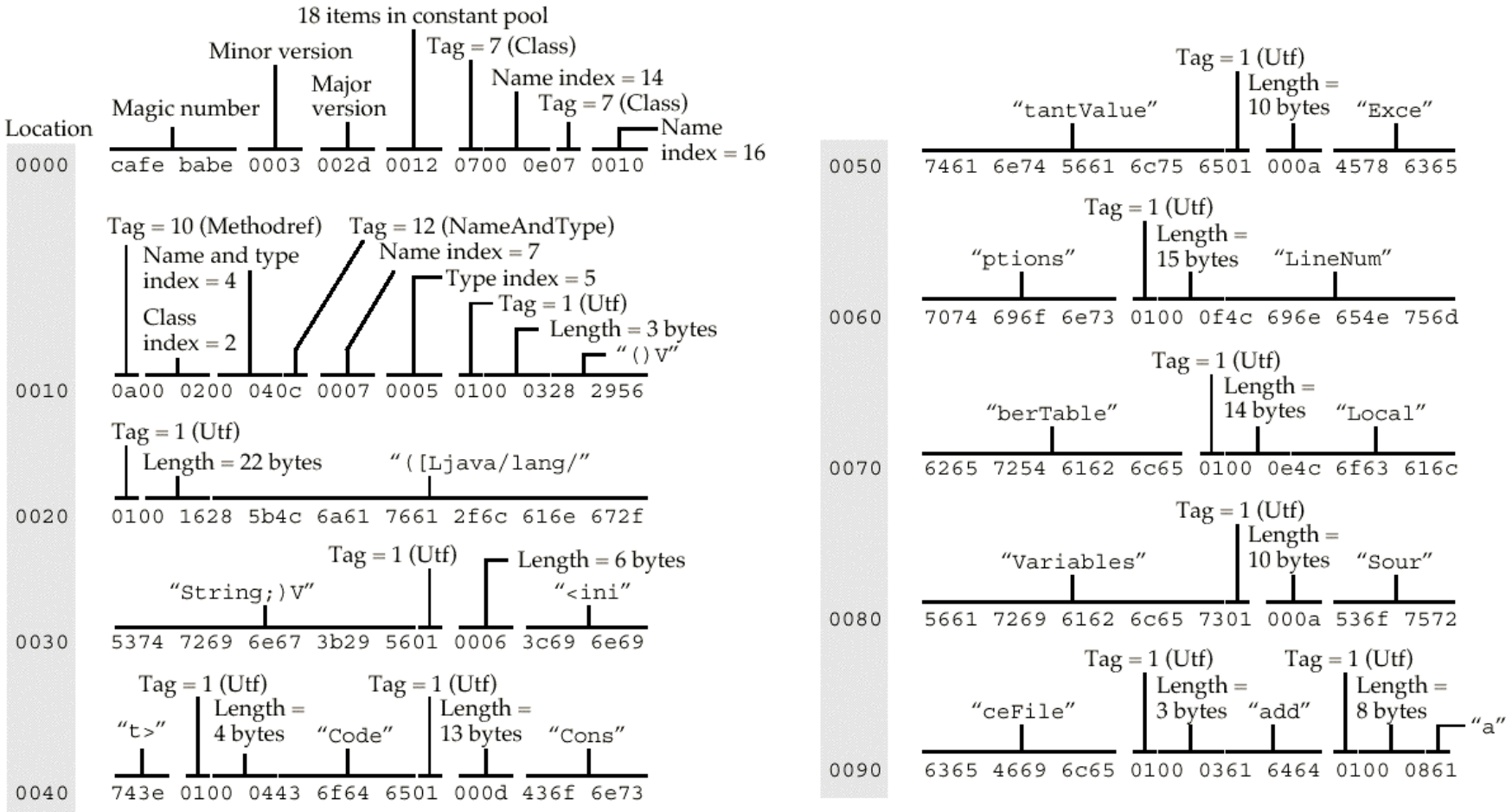


Programa Java e Arquivo de Classe Compila-do

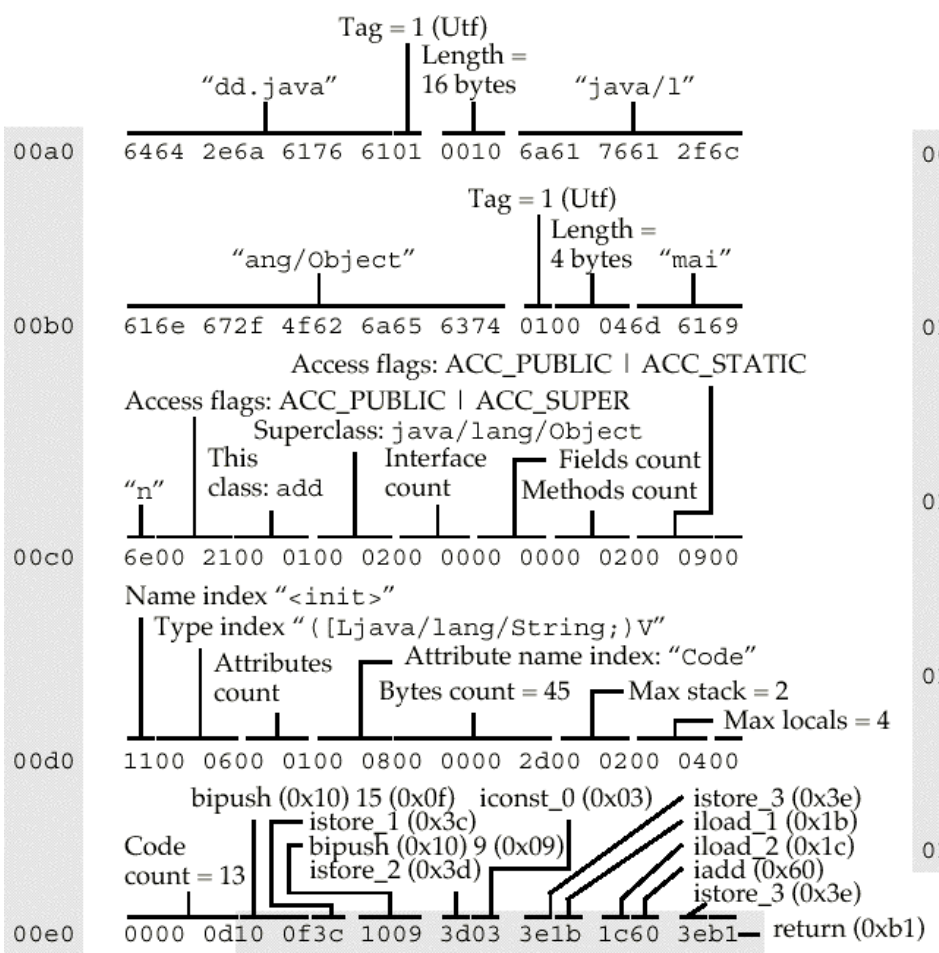
```
// This is file add.java

public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
        }
    }
```

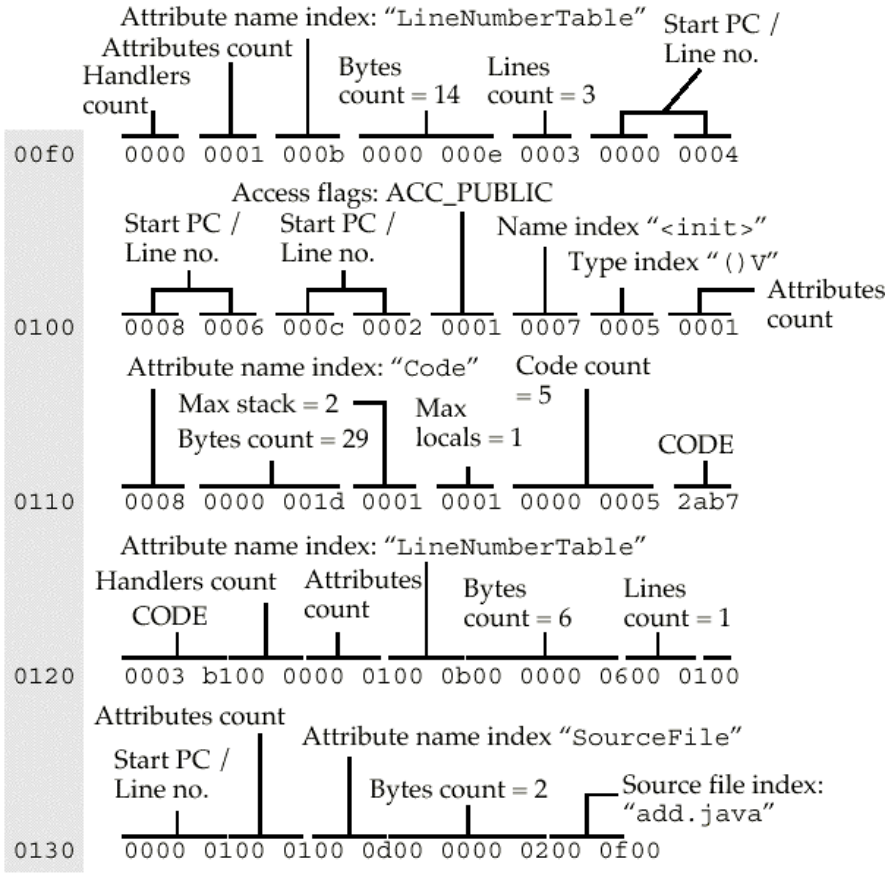
```
0000 cafe babe 0003 002d 0012 0700 0e07 0010   ..............
0010 0a00 0200 040c 0007 0005 0100 0328 2956   ............()V
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f   ...([Ljava/lang/
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69   String;)V...<ini
0040 743e 0100 0443 6f64 6501 000d 436f 6e73   t>...Code...Cons
0050 7461 6e74 5661 6c75 6501 000a 4578 6365   tantValue...Exce
0060 7074 696f 6e73 0100 0f4c 696e 654e 756d   ptions...LineNum
0070 6265 7254 6162 6c65 0100 0e4c 6f63 616c   berTable...Local
0080 5661 7269 6162 6c65 7301 000a 536f 7572   Variables...Sour
0090 6365 4669 6c65 0100 0361 6464 0100 0861   ceFile...add...a
00a0 6464 2e6a 6176 6101 0010 6a61 7661 2f6c   dd.java...java/l
00b0 616e 672f 4f62 6a65 6374 0100 046d 6169   ang/Object...mai
00c0 6e00 2100 0100 0200 0000 0000 0200 0900   n.!.............
00d0 1100 0600 0100 0800 0000 2d00 0200 0400   ..........-.....
00e0 0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1   .....<..=.>..`>.
00f0 0000 0001 000b 0000 000e 0003 0000 0004   ................
0100 0008 0006 000c 0002 0001 0007 0005 0001   ................
0110 0008 0000 001d 0001 0001 0000 0005 2ab7   ..............*.
0120 0003 b100 0000 0100 0b00 0000 0600 0100   ................
0130 0000 0100 0100 0d00 0000 0200 0f00        ..............
```

Um Arquivo de Classes Java



Um Arquivo de Classes Java (Cont.)





Byte Code do Programa Java

- Byte code desmontado do programa Java do slide anterior.

Location	Code	Mnemonic	Meaning
0x00e3	0x10	bipush	Push next byte onto stack
0x00e4	0x0f	15	Argument to bipush
0x00e5	0x3c	istore_1	Pop stack to local variable 1
0x00e6	0x10	bipush	Push next byte onto stack
0x00e7	0x09	9	Argument to bipush
0x00e8	0x3d	istore_2	Pop stack to local variable 2
0x00e9	0x03	iconst_0	Push 0 onto stack
0x00ea	0x3e	istore_3	Pop stack to local variable 3
0x00eb	0x1b	iload_1	Push local variable 1 onto stack
0x00ec	0x1c	iload_2	Push local variable 2 onto stack
0x00ed	0x60	iadd	Add top two stack elements
0x00ee	0x3e	istore_3	Pop stack to local variable 3
0x00ef	0xb1	return	Return