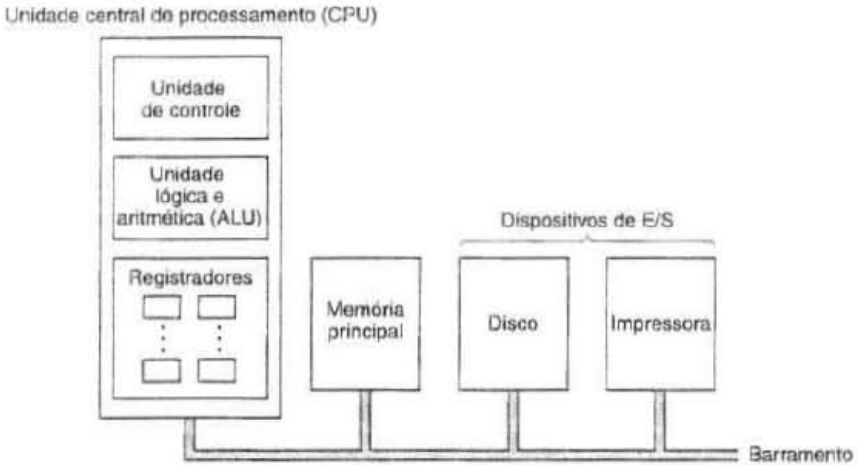
Organização de Sistemas de Computadores

Cap. 2 (Tanenbaum), Cap. 3 (Weber)

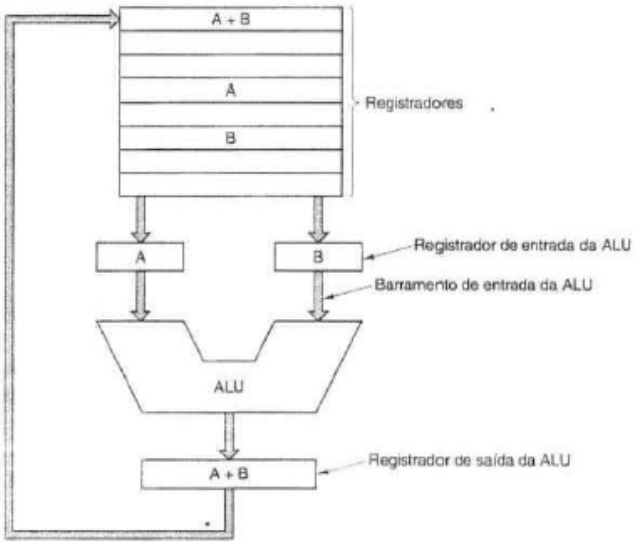
2.1 Processadores



CPU

- UC = buscar instruções na memória principal e determinar o seu tipo
- ULA = adição e AND
- Registradores = memória de alta velocidade para armazenar resultados temporários / controle de informações
  - Registradores mais importantes:
    - PC: indica a próxima instrução a ser buscada
    - IR: instrução que está sendo executada

2.1.1 Organização da CPU



2.1.1 Organização da CPU

- Os registradores alimentam 2 registradores de entrada da ULA
- O resultado pode ser armazenado em um registrador e, posteriormente, na memória
- Dois tipos de instrução:
  - Registrador – Memória (acessa memória)
  - Registrador – Registrador (não acessa memória)
- O processo de passar dois operandos pela ULA e armazenar o resultado recebe o nome de: ciclo do caminho de dados

ULA - Introdução

- Realiza operações aritméticas e lógicas sobre uma ou mais operandos
- Ex: soma de dois operandos, negação de um operando, AND ou OR de dois operandos, etc
- São opções geralmente muito simples
- Funções mais complexas são realizadas pela ativação sequencial das várias operações básicas disponíveis (ex: multiplicação)

ULA - Alguns Códigos de Condição

- Overflow: estouro de campo indicando que o resultado de uma operação aritmética não pode ser representado no espaço disponível
- Sinal: indica se o sinal de uma operação é positivo ou negativo
- Carry: em soma representa o “vai-um” (carry out) e em subtração o “vem-um” (borrow out)
- Zero: indica o resultado zero em uma operação

ULA - Sinais de Controle

- Devem ser fornecidos para a ULA
- Servem para selecionar a operação desejada entre as operações básicas disponíveis
- Contém salientar que a ULA não armazena o resultado, nem os operandos e os códigos de condição gerados

ULA - Modelo Estrutural

ULA - Características

- Comprimento em bits dos operandos
- Número e tipo de operações
- Códigos de condição gerados

2.1.2 Execução de Instrução
Buscar – Decodificar - Executar

1. Trazer a próxima instrução da memória até o registrador
2. Alterar o PC
3. Determinar o tipo de instrução trazida
4. Se usa a memória, determinar onde está na RAM
5. Trazer a palavra para dentro de um registrador (se for necessário
6. Executar a instrução
7. Voltar a etapa 1

Interpretador – Computador Simples (Java)

```
public class Interp {
  static int PC;                          // contador de programa contém endereço da próxima instr
  static int AC;                          // o acumulador, um registrador para efetuar aritmética
  static int instr;                       // um registrador para conter a instrução corrente
  static int instr_type;                  // o tipo da instrução (opcode)
  static int data_loc;                    // o endereço dos dados, ou -1 se nenhum
  static int data;                        // contém o operando corrente
  static boolean run_bit = true;          // um bit que pode ser desligado para parar a máquina

  public static void interpret(int memória[ ], int starting_ address) {
    // Esse procedimento interpreta programas para uma máquina simples com instruções que têm
    // um operando da memória. A máquina tem um registrador AC (acumulador), usado para
    // aritmética. A instrução ADD soma um inteiro na memória do AC, por exemplo.
    // O interpretador continua funcionando até o bit de funcionamento ser desligado pela instrução HALT.
    // O estado de um processo que roda nessa máquina consiste em memória,
    // contador de programa, bit de funcionamento e AC. Os parâmetros de entrada consistem
    // na imagem da memória e no endereço inicial.

  PC = starting_address;
  while (run_bit) {
    instr = memory[PC];                   // busque a próxima instrução e armazena em instr
    PC = PC + 1;                          // incremente contador de programa
    instr_type = get_instr_type(instr);   // determine tipo da instrução
    data_loc = find_data(instr, instr_type);  // localize dados (-1 se nenhum)
    if (data_loc >= 0)                    // se data_loc é -1, não há nenhum operando
      data = memory[data_loc];            // busque os dados
    execute(instr_type, data);            // execute instrução
    }
  }
  private static int get_instr_type(int addr) { ... }
  private static int find_data(int instr, int type) { ... }
  private static void execute(int type, int data) { ... }
}
```

2.1.3 RISC x CISC

- David Patterson e Carlo Séquin projetaram o RISC na UC Berkeley em 1980 (deu origem ao SPARC)
- John Hennessy projetou o MIPS em 1984 em Stanford
- RISC = 50 instruções & CISC = entre 200 e 300
- Intel 486 = instruções simples no núcleo RISC e complexas no núcleo CISC

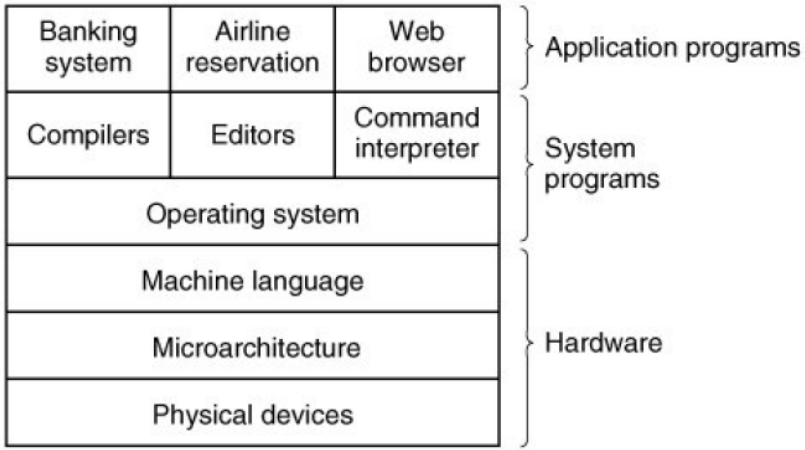
CISC – Complex Instruction Set Computer

- Por possuir um conjunto de instruções complexas, diminui o número de instruções que um programa necessita para sua implementação
- Faz uso de microprogramação/microcódigo da microarquitetura

CISC

- O número de ciclos por instrução pode aumentar (além do questão do clock da máquina...)
- Exemplos: IBM 360, DEC VAX, Motorola 68030, IBM 8088, 8086, 80286 e 80386.

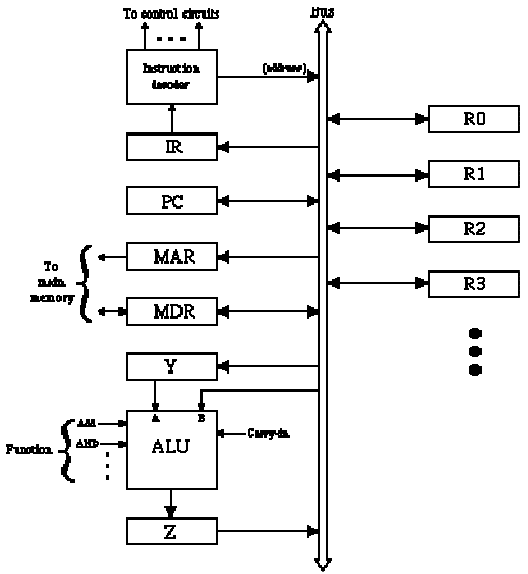
Microarquitetura

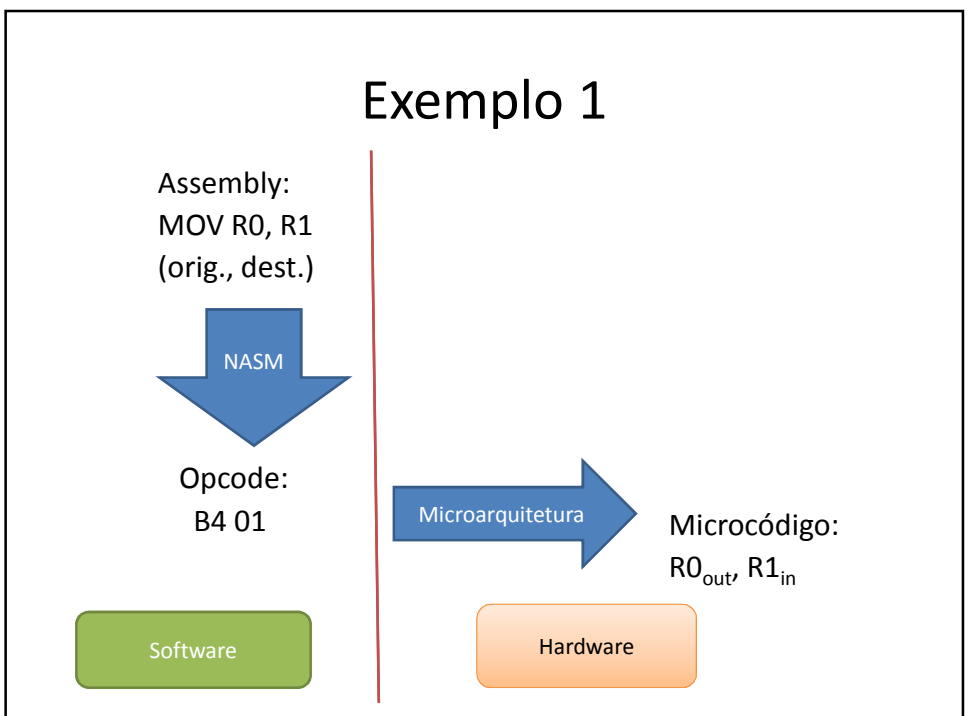


Microarquitetura

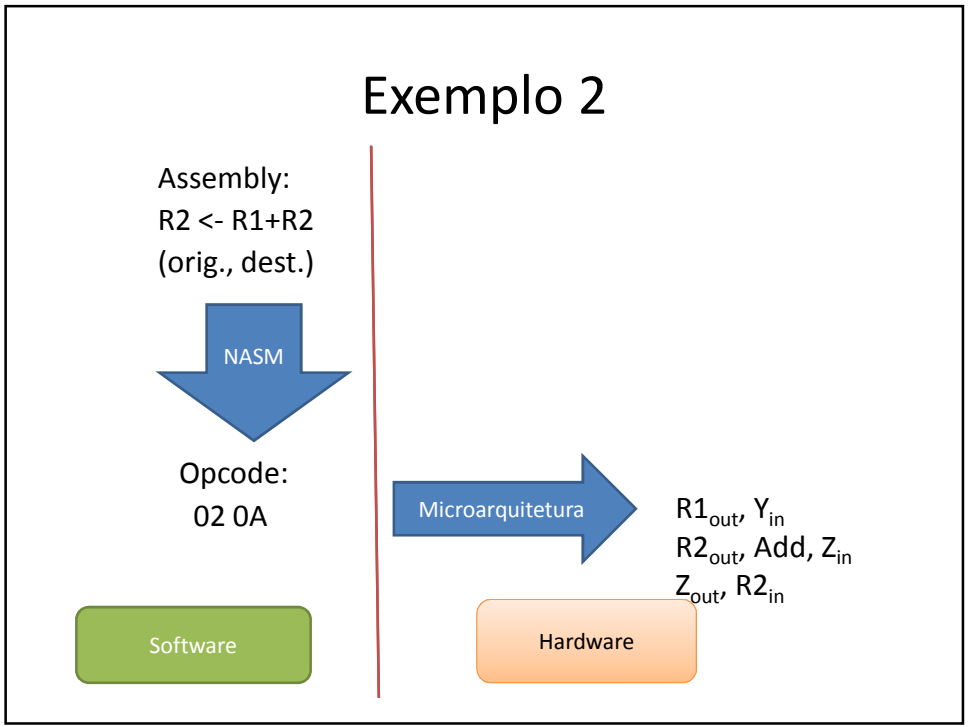
- Está acima do nível lógico
- Implementa o nível ISA (Instruction Set Architeture)
- RISC = ISA muito simples
- CISC = ISA muito complexa

Exemplo – Arquitetura com um barramento




Exemplo 1
Assembly:
MOV R0, R1
(orig., dest.)
NASM
Opcode:
B4 01
Microarquitetura
Microcódigo:
R0out, R1in
Software
Hardware

Exemplo 2


Assembly:
R2 <- R1+R2
(orig., dest.)
NASM
Opcode:
02 0A
Microarquitetura
$R1_{out}$, Y_{in}
$R2_{out}$, Add, Z_{in}
Z_{out}, $R2_{in}$
Software
Hardware

RISC – Reduced Instruction Set Computer

- Geralmente as instruções executam em um único ciclo do processador
- A funcionalidade migrou para o software
- O hardware realiza somente procedimentos básicos
- O compilador precisa ser otimizado

RISC

- No final de 1970 foram feitos experimentos com instruções complexas
- Existia a discussão entre o que as máquinas faziam e o que as linguagens de alto nível precisavam
- Computador 801 (somente protótipo)

RISC

- 1980: em Berkeley começaram a produzir chips VLSI sem interpretação (microprogramação) -> SPARC
- 1981: em Stanford projetaram o MIPS -> MIPS
- Poucas instruções
- Várias instruções iniciadas por segundo

RISC x CISC (1980)

- CISC = “Set” de 200 a 300 instruções
- RISC = “Set” de 50 instruções

- RISC = Alegavam que o melhor modo era um pequeno número de instruções simples que executassem em um único ciclo

- Por mais que tivessem que executar várias instruções, comparado ao CISC, venceriam pois não eram interpretadas

Processadores “híbridos”

- Intel (486 em diante) possui um núcleo RISC para executar instruções mais simples
- Permite competitividade com máquinas puramente RISC
- A maioria das instruções dos programadores é simples e é executada pelo núcleo RISC

Princípios de Projeto Moderno

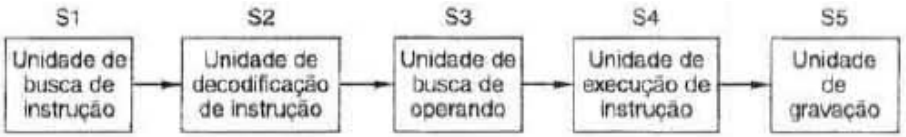
- Todas instruções executadas diretamente por hardware
- Maximizar a taxa de execução das instruções
- Instruções fáceis de decodificar
- Somente LOAD e STORE referenciam memória
- Providencie muitos registradores

2.1.5 Pipelining

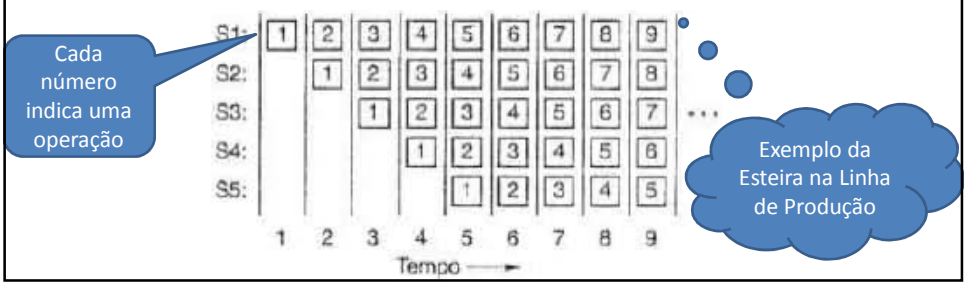
- O processo de buscar instruções na memória é lento
- A saída é uma busca antecipada, dividindo a execução em várias partes
- Pipeline = paralelismo, tubulação
- Cada parte é manipulada por uma parte do hardware (todas podem executar em paralelo)

Ex: Pipeline com 5 unidades

- São também denominados de estágios



- Funcionamento do Pipeline em função do tempo



Pipeline

- O pipeline permite o compromisso com:
- Latência = o tempo em que demora para executar uma instrução
- Largura de Banda de Processador = quantos MIPS a CPU tem