

[MAC0211] Laboratório de Programação I

Aula 2 –

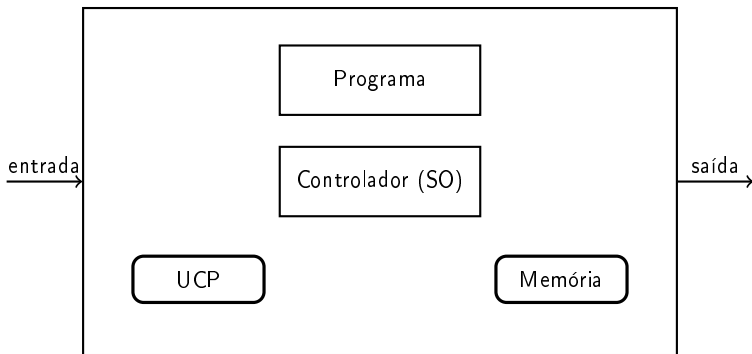
Alair Pereira do Lago

DCC-IME-USP

26 de fevereiro de 2015

Arquitetura de von Neumann

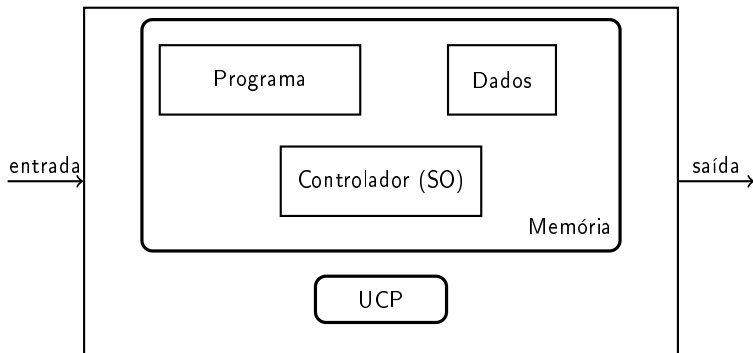
- ▶ Primeiros computadores eletrônicos (como o ENIAC): não armazenavam programas; cada novo cálculo exigia que plugues e cabos fossem movidos



Arquitetura do ENIAC

Arquitetura de von Neumann

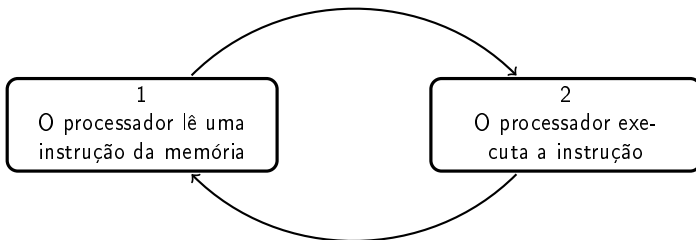
- ▶ John von Neumann (matemático consultor do projeto ENIAC) publicou o conceito de “programa armazenado” em 1945



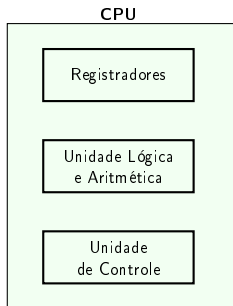
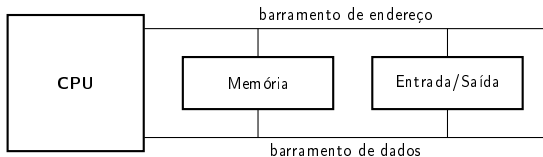
Arquitetura de von Neumann (usada nos computadores atuais)

O ciclo de busca e execução

- ▶ Programa: lista de instruções
- ▶ Processador efetua uma computação por meio do ciclo de busca e execução:



Modelo simplificado de um computador



Dentro de uma CPU (processador), temos...

Unidade Lógica e Aritmética (ULA)

- ▶ Realiza todas as tarefas relacionadas a operações lógicas (and, or, not, etc.) e aritméticas (adições, subtrações, etc.)

Unidade de Controle (UC)

- ▶ Controla as ações realizadas pelo computador, comandando todos os demais componentes de sua arquitetura

Dentro de uma CPU (processador), temos...

Registradores

- ▶ Um registrador é uma coleção de circuitos que armazenam bits
- ▶ Os registradores de um processador não precisam armazenar uma mesma quantidade de bits (mas é mais fácil de se lidar com eles quando eles são assim)
- ▶ A quantidade de bits que se pode armazenar em um registrador típico do processador é um dos atributos que determinam sua classificação (Ex.: processador de 32-bits, ou de 64-bits, etc.)
- ▶ Cada registrador possui uma função própria. Exemplos:
 - ▶ Contador de programa (PC, de *program counter*) – aponta para a próxima instrução a executar
 - ▶ Registrador de instrução (IR, de *instruction register*) – armazena a instrução em execução
 - ▶ Armazenamento de resultados intermediários

Processadores – conjunto de instruções

- ▶ As instruções são as operações que um processador é capaz de realizar; elas são a parte do processador que é “visível” para os programadores
- ▶ Cada processador possui o seu próprio conjunto de instruções
- ▶ Por meio de uma linguagem de montagem (*assembly*), podemos usar o conjunto de instruções diretamente
- ▶ Mesmo processadores com arquiteturas internas diferentes podem ter um mesmo conjunto de instruções (ex.: Intel Pentium e AMD Athlon)

Processadores – conjunto de instruções

Operações

As instruções de um processador se relacionam às seguintes funcionalidades:

- ▶ operações matemáticas e lógicas
- ▶ movimentação de dados (transferência de dados da memória para os registradores e vice-versa)
- ▶ operações de entrada/saída (leitura ou escrita de dados em dispositivos de entrada e saída)
- ▶ controle do fluxo de execução (desvios condicionais ou incondicionais)

Processadores – CISC × RISC

Complex instruction set computer (CISC)

- ▶ capaz de executar várias centenas de instruções complexas diferentes (= **versatilidade**)

Reduced instruction set computer (RISC)

- ▶ capaz de executar apenas algumas poucas instruções simples (= **rapidez, menor custo**)

Processadores – CISC × RISC

Exemplo

Comando em linguagem de alto nível:

```
A = B + C    /* considerando que todas as quantidades  
               estão em memória */
```

Em linguagem de máquina em um processador CISC:

```
add    mem(B), mem(C), mem(A)
```

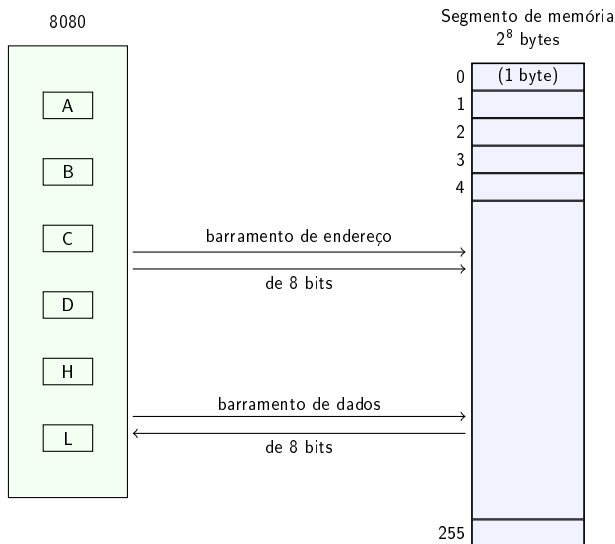
Em linguagem de máquina em um processador RISC:

```
load    mem(B), reg(1);  
load    mem(C), reg(2);  
add     reg(1), reg(2), reg(3);  
store   reg(3), mem(A);
```

Processadores Intel da família x86 (ou 80x86)

- ▶ Ótimo exemplo de processadores CISC
- ▶ Característica importante: processadores dessa família mantém compatibilidade com os seus antecessores, ou seja, programas desenvolvidos para um processador mais antigo da família podem ser executados em um processador mais novo da família
- ▶ Processadores membros: 8086, 80186, 80286, 80386, 80486, Pentium (Pro, II, III, 4), Core...

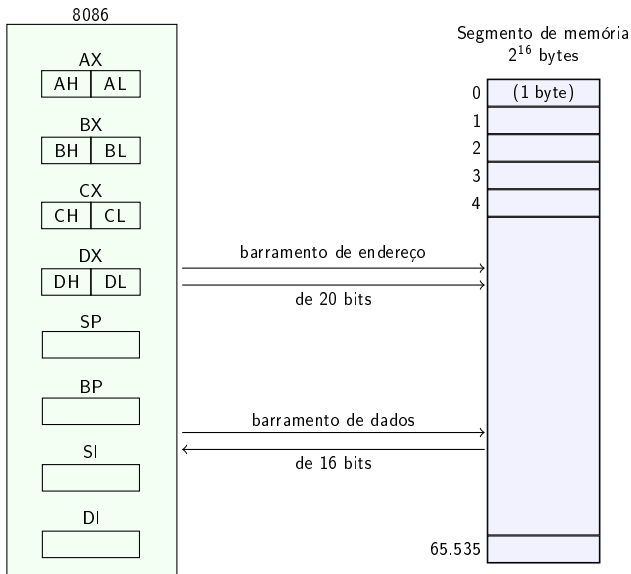
Registradores do processador Intel 8080 (8 bits)



Registradores de uso geral

- ▶ A (Acumulador) – usado em movimentações de dados, operações aritméticas e entrada/saída
- ▶ B (Base) – usado como ponteiro para acesso a memória; também recebe o valor de retorno de algumas interrupções
- ▶ C (Contador) – usado para controlar o número de vezes que um laço deve ser executado ou o número de *shifts* a ser realizado; também usado em operações aritméticas
- ▶ D (Dados) – usado em operações de entrada e saída; também usado em operações de multiplicação ou divisão de números grandes
- ▶ H (*High*) e L (*Low*) – usados de forma conjunta, como um registrador de 16 bits, que pode ser usado para endereços

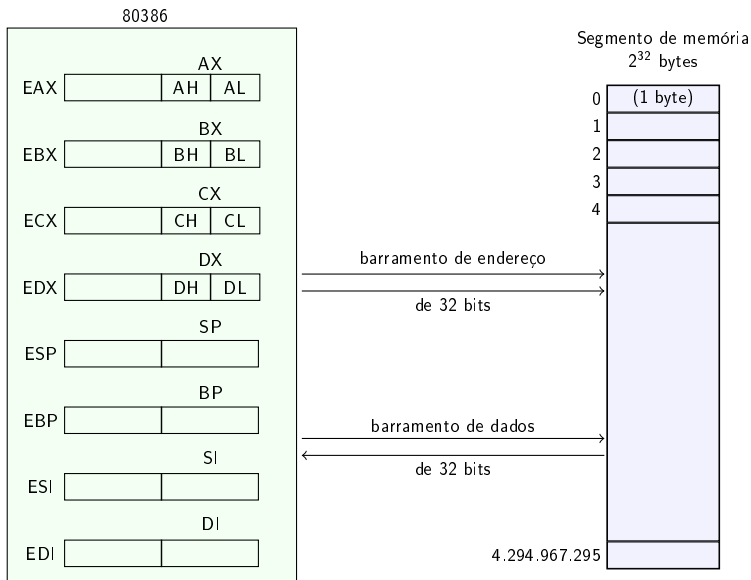
Registradores do processador Intel 8086 (16 bits)



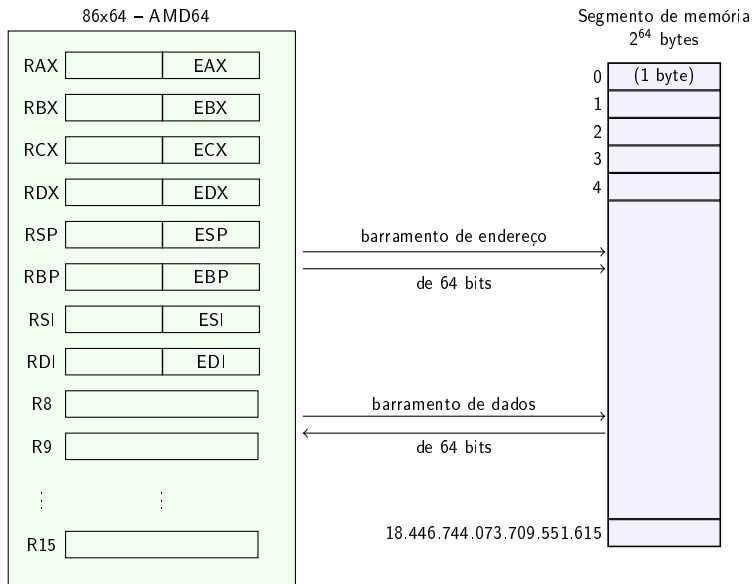
Registradores para índices e ponteiros

- ▶ SP (*Stack pointer register*) – armazena o endereço do topo da pilha de dados
- ▶ BP (*Stack base pointer register*) – armazena o endereço da base da pilha de dados
- ▶ SI (*Source index register*) – usado na manipulação de strings e vetores
- ▶ DI (*Destination index register*) – usado na manipulação de strings e vetores

Registradores do processador Intel 80386 (32 bits)



Registradores dos proc. Intel x86-64 ou AMD64 (64 bits)



O ciclo de busca e execução (revisitado)

A UCP executa cada instrução por meio de uma série de pequenos passos:

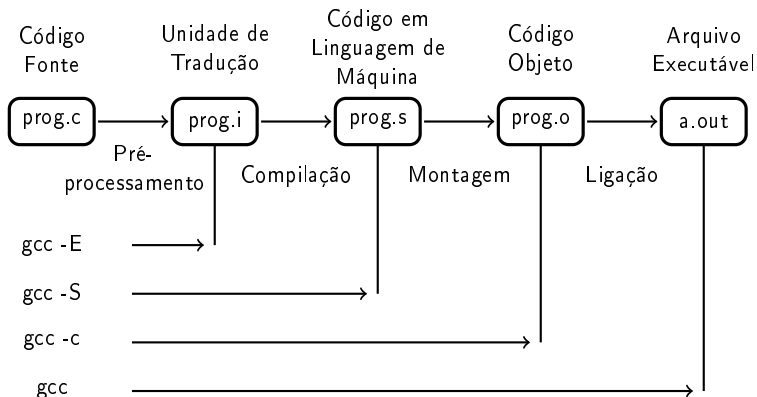
1. Lê a próxima instrução na memória e a armazena no registrador de instrução
2. Muda o registrador contador de programa, para que ele aponte para a instrução seguinte
3. Determina o tipo da instrução que acabou de ser lida
4. Se a instrução usa algum dado da memória, determina onde ele está
5. Carrega o dado, se necessário, em um registrador da UCP
6. Executa a instrução
7. Volta para o passo 1, para começar a execução da instrução seguinte

Do código fonte de alto nível ao código executável

GCC – Gnu C Compiler

- A compilação feita pelo GCC tem vários estágios:

```
alair@linux$ gcc prog.c
```



Linguagem de montagem (*Assembly*)

Porque usar

- ▶ Para aprender como uma dada CPU funciona
- ▶ Para ter o controle total do seu código
- ▶ Para melhorar o desempenho de partes do programa, quando o tempo de execução é uma preocupação crítica
- ▶ Para escrever um novo SO ou portar um já existente para uma nova máquina (há partes do código que precisam ser escritas em linguagem de montagem)
- ▶ Para obter acesso a modos de programação não usuais do seu processador (ex.: modo 16-bits, firmware, etc.)
- ▶ Para otimizar manualmente códigos gerados por um compilador não muito bom
- ▶ Para produzir códigos otimizados para uma determinada configuração de hardware

Linguagem de montagem (*Assembly*)

Porque não usar

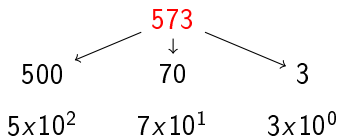
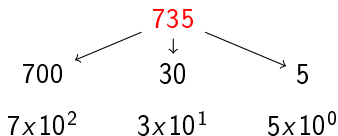
- ▶ Código longo (e entediante!)
- ▶ Código muito susceptível a erros; e os erros são difíceis de serem localizados
- ▶ Código de difícil manutenção
- ▶ O resultado não é portátil para outras arquiteturas

Sistemas de numeração

O ENIAC usava o sistema de numeração decimal. Depois dele, todos os computadores eletrônicos usam em seus cálculos aritméticos o sistema de numeração binário.

Sistema decimal (base 10)

- ▶ Usa dez dígitos distintos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- ▶ É um sistema posicional
 - ▶ Valor de um dígito depende da posição em que ele se encontra no conjunto de dígitos que representa uma quantidade
 - ▶ O valor total do número é a soma dos valores relativos de cada dígito



Sistema binário (base 2)

- ▶ Usa dois dígitos distintos (0, 1)
- ▶ Estrutura de pesos dos números binários:

$$\dots 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0, \ 2^{-1} \ 2^{-2} \ 2^{-3} \ 2^{-4} \ 2^{-5} \dots$$

Conversão de binário para decimal

Exemplo: $(111001,1)_2$

$$\begin{aligned} &= (1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1})_{10} \\ &= (32 + 16 + 8 + 1 + 0,5)_{10} \\ &= (57,5)_{10} \end{aligned}$$

Conversão de decimal para binário

Exemplo:

$$(57,3125)_{10} = (111001,0101)_2$$

Parte inteira – Método das divisões sucessivas

57	÷	2	=	28	com resto	1	→ bit menos significativo
28	÷	2	=	14	com resto	0	
14	÷	2	=	7	com resto	0	
7	÷	2	=	3	com resto	1	
3	÷	2	=	1	com resto	1	
1	÷	2	=	0	com resto	1	→ bit mais significativo

Tomando-se os restos na ordem inversa da que foram gerados, temos o número **111001**.

Logo, temos que $(57)_{10} = (111001)_2$.

Conversão de decimal para binário

Exemplo:

$$(57,3125)_{10} = (111001,0101)_2$$

Parte fracionária – Método das multiplicações sucessivas

0,3125	×	2	=	0,	625	→ bit mais significativo
0,625	×	2	=	1,	25	
0,25	×	2	=	0,	5	
0,5	×	2	=	1,	0	→ bit menos significativo

Tomando-se os restos na ordem em que foram gerados, temos o número **0101**.

Logo, temos que $(0,3125)_{10} = (0,0101)_2$.

Aritmética binária

Soma

$$0 + 0 = 0$$

$$0 + 1 = 1 + 0 = 1$$

$$1 + 1 = 10$$

$$\text{Exemplo: } 1111 + 11100 = 101011$$

$$\begin{array}{r} \\ \\ + \\ \hline 1 \end{array}$$

Aritmética binária

Subtração

$$0 - 0 = 0$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

$$10 - 1 = 1$$

$$\text{Exemplo: } 10001 + 1110 = 101011$$

		1		1	
	0	10	10	10	
	1	0	0	0	1
-		1	1	1	0
<hr/>					
	0	0	0	1	1

Conversão de binário para decimal

Outro exemplo:

$$(1010101010101110)_2$$

parte 1 (8 bits)	parte 2 (8 bits)
<hr/> 10101010	<hr/> 10101110

$$(170)_{10} \quad (174)_{10}$$

Logo, temos que $(1010101010101110)_2 = (170 \times 2^8 + 174)_{10} = (170 \times 256 + 174)_{10} = (43694)_{10}$.

Bibliografia e materiais recomendados

- ▶ Texto bastante didático que descreve todos os estágios do processo de compilação do GCC
<http://www.redhat.com/magazine/002dec04/features/gcc/>
- ▶ Capítulo 2 do *Linux Assembly HOWTO*, de K. Boldyshev e F.-R. Rideau
<http://www.tldp.org/HOWTO/pdf/Assembly-HOWTO.pdf>
- ▶ Capítulos 1, 2 e 3 do livro *Linux Assembly Language Programming*, de B. Neveln
- ▶ Livro *Structured Computer Organization*, de A. S. Tanenbaum
- ▶ Notas das aulas de MAC0211 de 2010, feitas pelo Prof. Kon
<http://www.ime.usp.br/~kon/MAC211>

Cenas dos próximos capítulos...

- ▶ Sistemas de numeração (continuação)
- ▶ Linguagem de montagem – conceitos básicos e primeiras instruções