QXD-005 - Arquitetura de Computadores

Arquitetura do Conjunto de Instruções

Prof. Pedro Botelho

Nas Aulas Passadas…

- Visão de Alto Nível do Computador
  - Estrutura do Computador
  - Instruções Básicas
- Sistemas de Memórias
  - Cache
  - Interna
  - Externa
- Entrada/Saída
- Questão: Como funciona o componente mais importante, o **processador**?

Nesta Aula…

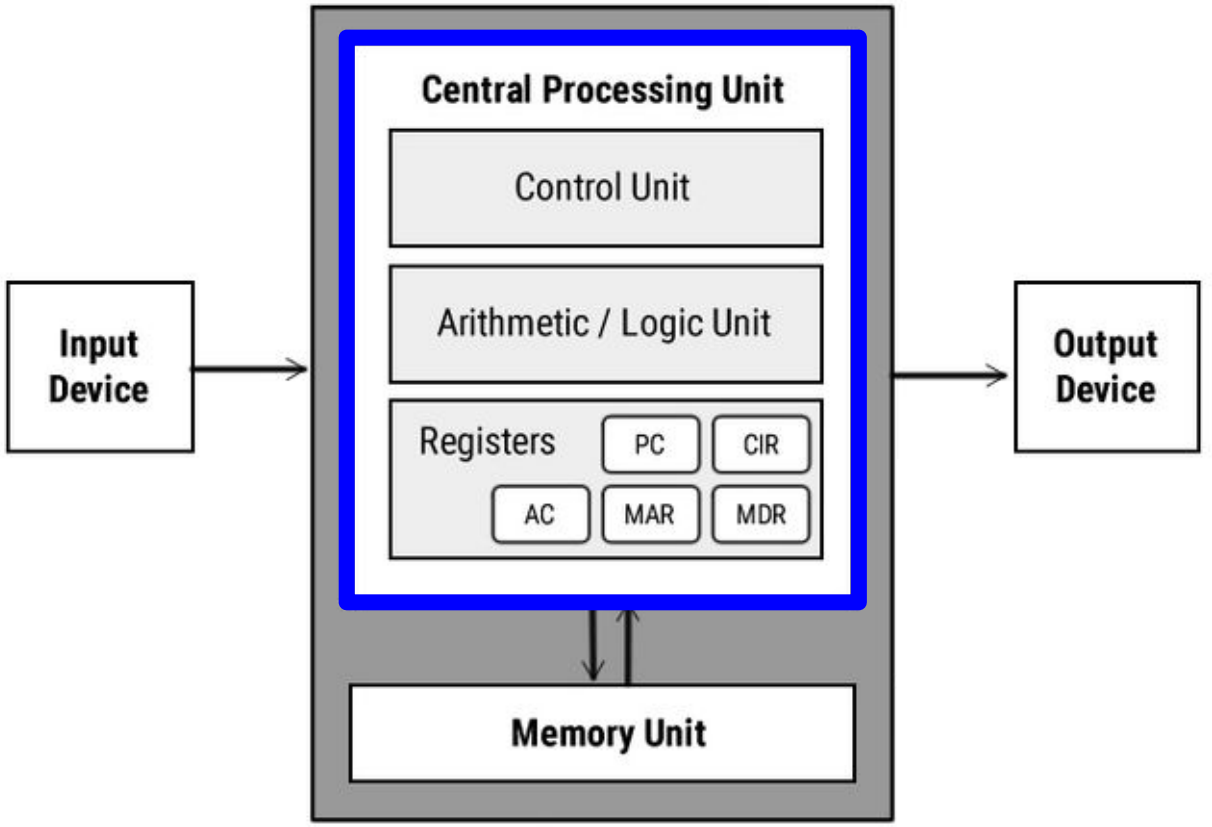
- Características das Instruções de Máquina
- Arquitetura v.s. Microarquitetura
- Processadores CISC
- Processadores RISC

Arquitetura do Conjunto de Instruções

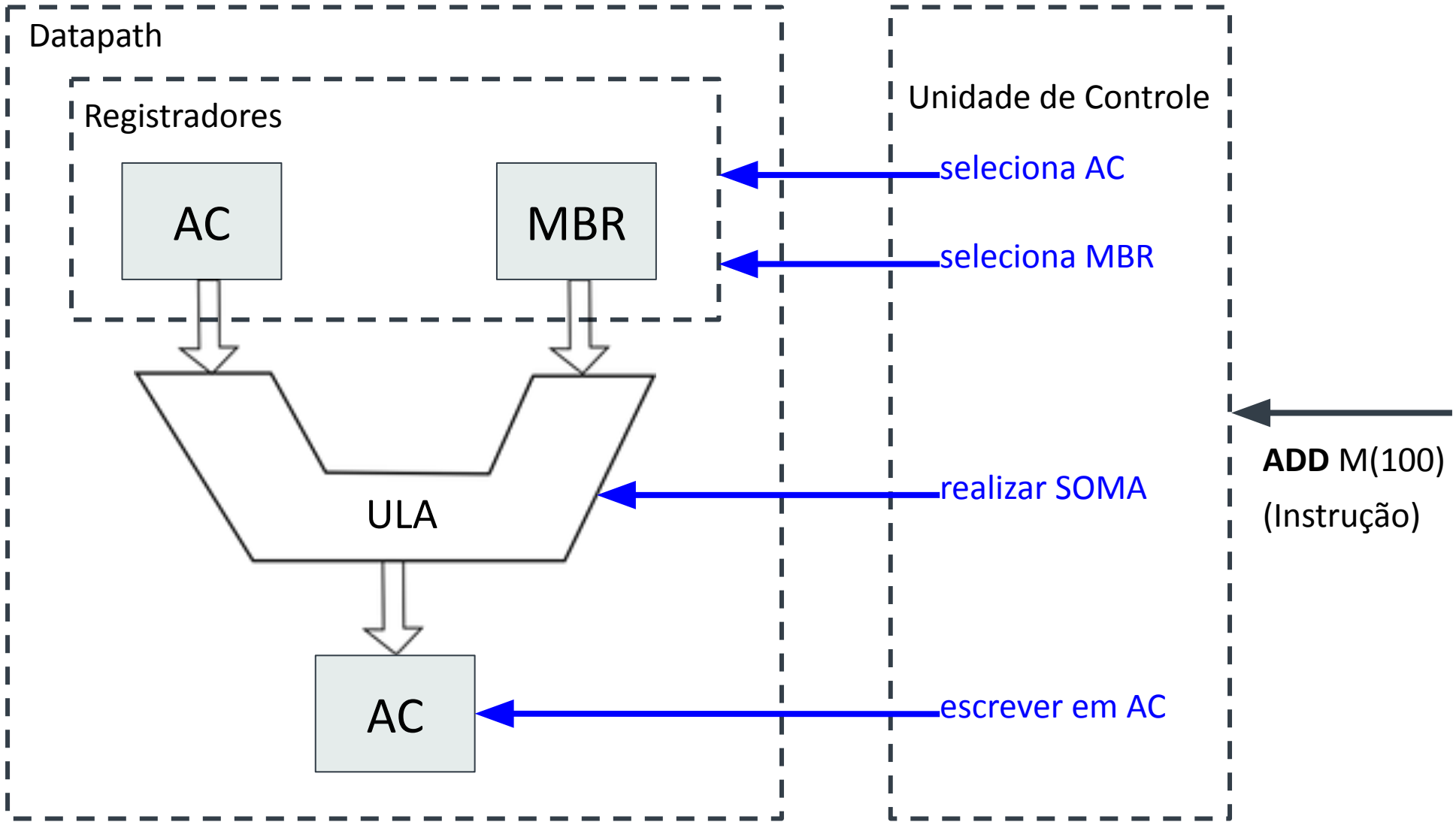
Características das Instruções de Máquina

Revisão do Processador

- Executa **operações** e processa **dados**
  - **Dados** e **operações** mantidos na memória
- Contém subunidades específicas
  - Operações executadas pela **ULA**
  - Operação controlada pela **Unidade de Controle**
  - **Registradores** mantém dados resultantes e a serem processados
- A operação do processador é determinada pelas **instruções** que ele executa
  - Operação que o processador consegue executar
  - Exemplo: Soma, subtração, multiplicação, etc…



Revisitando a Operação do Processador IAS



Conjunto de Instruções

- Repertório de instruções que são entendidas por uma CPU
  - "*Uma descrição do conjunto de instruções de máquina de um computador permite um grande entendimento sobre o processador*", Stallings, W.
- **Código de Máquina**: Instruções binárias que a CPU interpreta
  - Representação em binário ↔ Representação Simbólica (*Assembly*)

Mnemônico	Opcode Binário	Descrição
LOAD M(X)	0000 0001	Carrega dados da memória
STOR M(X)	0010 0001	Salva dados na memória
ADD M(X)	0000 0101	Adição
SUB M(X)	0000 0110	Subtração
MUL M(X)	0000 1011	Multiplicação
DIV M(X)	0000 1100	Divisão

Conjunto de Instruções

- Repertório de instruções que são entendidas por uma CPU
  - *"Uma descrição do conjunto de instruções de máquina de um computador permite um grande entendimento sobre o processador"*, Stallings, W.
- **Código de Máquina**: Instruções binárias que a CPU interpreta
  - Representação em binário ↔ Representação Simbólica (*Assembly*)

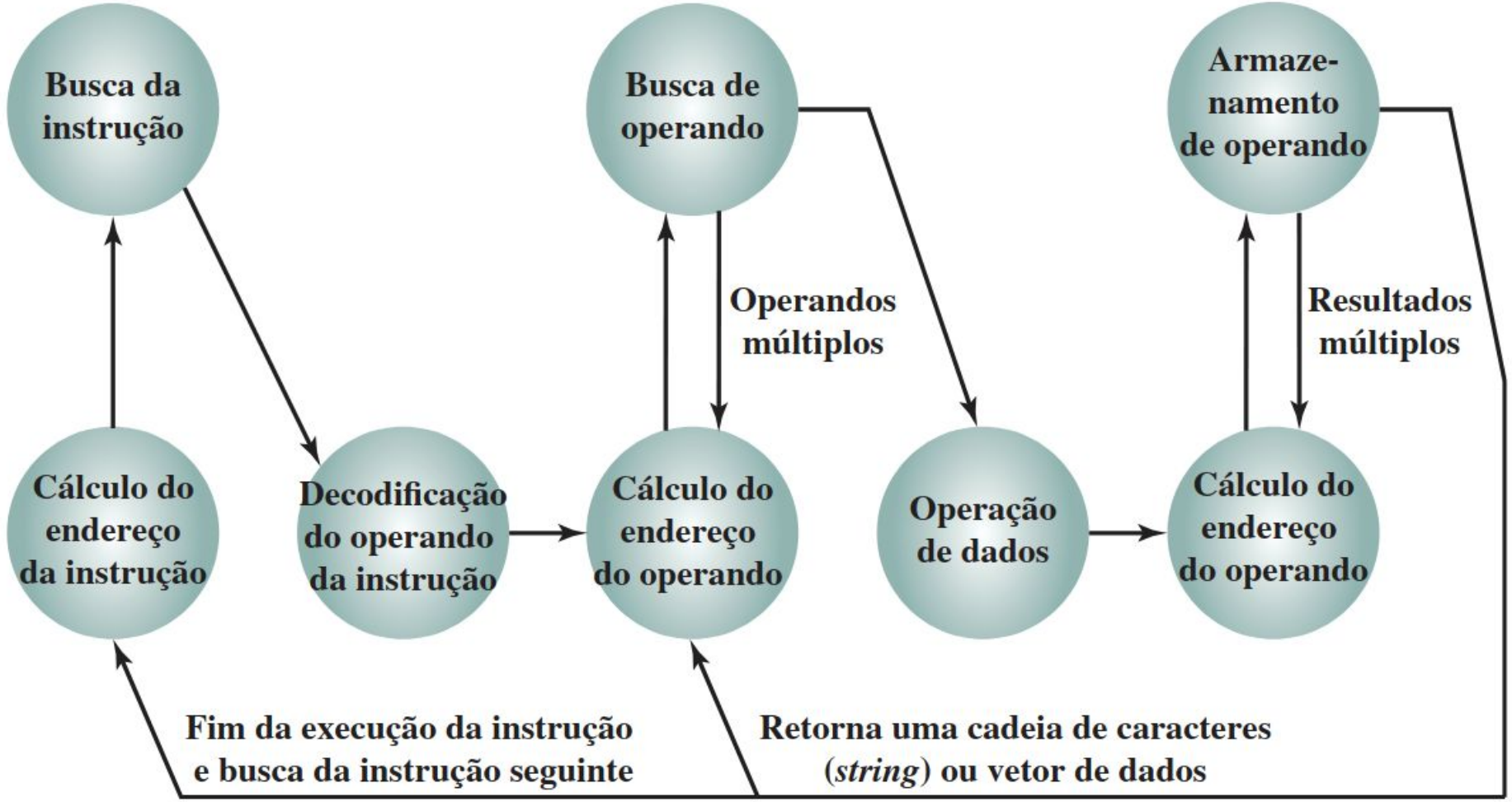
0000 0101 (Binário) ⟷

Mnemônico	Opcode Binário	Descrição
LOAD M(X)	0000 0001	Carrega dados da memória
STOR M(X)	0010 0001	Salva dados na memória
ADD M(X)	0000 0101	Adição
SUB M(X)	0000 0110	Subtração
MUL M(X)	0000 1011	Multiplicação
DIV M(X)	0000 1100	Divisão

⟷ ADD M(200) (Assembly)

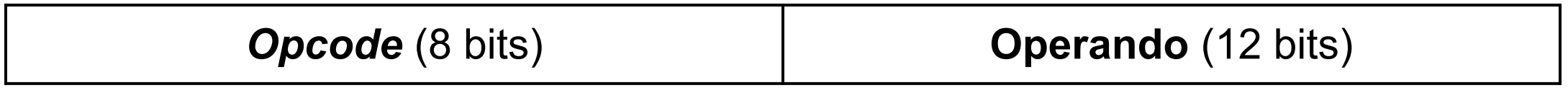
Ciclo de Instrução

- Passos tomados ao executar uma instrução: Veremos isso bastante!



Elementos de uma Instrução de Máquina

- Cada instrução deve conter as informações exigidas pelo processador para **execução**:
  - O que vai fazer?
  - Com o que?
  - Onde salvar?
- **Código de Operação** (*opcode*): Especifica a operação a ser realizada
- **Referência a Operando Fonte**: A operação pode ter um ou mais operandos de entrada
- **Referência a Operando de Resultado**: A operação deve produzir um resultado
- Representados por números binários na instrução
  - Podem ser "implícitos" ⟶ Por exemplo, **MBR** e **AC** na instrução **ADD M(120)** IAS



Exemplo de Elementos de uma Instrução

- Vejamos a instrução **ADD M(120)** do IAS:

0000 0101	0000 0111 1000
Opcode (8 bits)	**Operando** (12 bits)

- **Soma** realizada na ULA ⟶ ULA opera sobre **registradores**
  - Primeiro carrega MBR com M(120)
- Assim, é realizado AC = AC + MBR (**implícitos**)
- Problema… E pra processar mais dados?
  - f = (g + h) - (i + j)
  - Muitas operações! Como melhorar?
  - Aumentar a quantidade de operandos é uma boa!

```
LOAD   M(i)      // AC = i
ADD    M(j)      // AC = i + j
STOR   M(temp)   // temp = i + j
LOAD   M(g)      // AC = g
ADD    M(h)      // AC = g + h
SUB    M(temp)   // AC = (g + h) - temp
STOR   M(f)      // f = (g + h) - (i + j)
```

Instruções de Três Operandos

- Processadores que usam três operandos operam em dados sobre **registradores**
  - Operandos são “endereços” que referenciam esses registradores
  - Pequena memória dentro do processador ⟶ **Banco de Registradores**
- Suponha um processador com **32 registradores** (de x0 a x31) ⟶ Endereço de 5 bits
  - Exemplo: **ADD** X1, X2, X3

0000 0101 (**ADD**)	0 0001 (**X1**)	0 0010 (**X2**)	0 0011 (**X3**)
opcode (8 bits)	**rd** (5 bits)	**rs1** (5 bits)	**rs2** (5 bits)

- **Soma** realizada na ULA ⟶ ULA opera sobre esses **registradores**
- Assim, é realizado X1 = X2 + X3 (**explicitos**)

Programa com Instruções de Três Operandos

- Vamos refazer: f = (g + h) - (i + j)
  - Suponha que **X1 = g**, **X2 = h**, **X3 = i**, **X4 = j** e **X5 = f**.

```
LOAD   M(i)      // AC = i
ADD    M(j)      // AC = i + j
STOR   M(temp)   // temp = i + j
LOAD   M(g)      // AC = g
ADD    M(h)      // AC = g + h
SUB    M(temp)   // AC = (g + h) - temp
STOR   M(f)      // f = (g + h) - (i + j)
```

Instruções com 1 Operando

```
ADD    X6, X1, X2   // X6 = g + h
ADD    X7, X3, X4   // X7 = i + j
SUB    X5, X6, X7   // X5 = (g + h) - (i + j)
```

Instruções com 3 Operandos

- O que você acha melhor?

Arquitetura do Conjunto de Instruções

Arquitetura v.s. Microarquitetura

A Arquitetura do Conjunto de Instruções (ou ISA)

- Processadores possuem:
  - Instruções
  - Registradores
  - Modelo de Memória
- Como controlar essas máquinas? ⟶ Código Assembly
  - Usa instruções, acessa registradores e memória, e etc…
  - Todo o resto (e.g. HLL) vira Assembly eventualmente ⟶Ex: **C/C++ é compilado para Assembly**
- **Arquitetura do Conjunto de Instruções** define a **interface básica** entre *hardware* e *software*
  - Ou ISA (do inglês *Instruction Set Architecture*)
  - Define basicamente como o computador opera à nível de arquitetura
  - Não se preocupa com a organização
  - "*Processador tal define tais **registradores** e tais **instruções** que posso usar no **Assembly**.*"
  - Exemplos: x86, ARM, RISC-V, MIPS, etc…

Microarquitetura v.s. Arquitetura

- A **Arquitetura** (ISA) descreve os atributos de alto-nível.
  - Interface de *Hardware* e *Software*.
  - Modelo de Programação: Instruções, registradores, etc…
- A **Microarquitetura** define os detalhes da implementação física da CPU
  - Implementação do ISA: Define a organização física do *hardware*.
  - Exemplo: Define a tecnologia utilizada para implementar a memória (Harvard ou Von Neumann).
- O mesmo *SOFTWARE* pode ser compatível com múltiplos *HARDWARES…*
  - …desde que sejam da mesma **ARQUITETURA!**
- Exemplos:
  - A microarquitetura **Alder Lake** (usada no **Core i7 de 12ª geração**) implementa a arquitetura **x86-64**.
  - A microarquitetura **Cortex-A78** (usada no **Samsung Exynos 2100**) implementa a arquitetura **ARMv8**.

Computadores com Programa Armazenado

- **Instruções** representadas em **binário**, assim como **dados**
- Instruções e dados armazenados na **memória**
- Duas arquiteturas possíveis para memória:
  - **Von Neumann** → Mesma memória
  - **Harvard** → Memórias diferentes (dados e programa)
- **Compatibilidade binária** → Programas compilados funcionam em diferentes computadores
  - ISAs padronizados
- Exemplo: Linux compilado para o ISA x86-64 pode executar em qualquer processador x86-64:
  - Intel Core i3
  - Intel Core i5
  - Intel Core i7
  - Intel Core i9

A Arquitetura RISC-V

- RISC-V é uma arquitetura com foco em **simplicidade** e **poder**
  - Segue os princípios RISC (*Reduced Instruction Set Computer*)
- Desenvolvido na Universidade da Califórnia em Berkeley em 2010 → **ISA Aberto** (*open source*)
  - RISC-V Foundation (riscv.org) fornece o **ISA padronizado** → Descrição de como deve funcionar
- **Licença Permissiva (BSD)** → Não precisa compartilhar a implementação.
  - Empresas podem implementar a microarquitetura (seu IP) e vendê-la a sem pagar nada
  - Podem estender o conjunto de instruções base com **extensões** personalizadas
- Várias empresas usam esse modelo de negócios → Código fechado e **proprietário**:
  - **SiFive** → Cria núcleos RISC-V de alta performance.
  - **Western Digital** & **NVIDIA** → Usam núcleos RISC-V em seus discos rígidos e placas de vídeo.
- Já tem todo um ecossistema pronto → *Toolchain*, comunidade, etc…
  - Mas um custo: Muita coisa já existe para x86 e ARM! Deve ser recompilado…

Curiosidade 1: E quanto à Arquitetura ARM?

- Uma arquitetura moderna muito disseminada que também segue os princípios RISC
- Várias empresas (e.g. Qualcomm) produzem processadores **ARM**, mas pagam **royalties**.
- **ARM® Holdings** → Empresa inglesa que **projeta** processadores ARM
  - Mantém a arquitetura (ARMv9) e microarquiteturas (e.g. Cortex-A710)
  - Não fabrica processadores, apenas mantém a patente (IP)
  - Business partners fabricam os processadores **baseados** em ARM

Curiosidade 2: E quanto à Arquitetura x86?

- Apenas a **Intel** e **AMD** podem produzir processadores da arquitetura x86
  - (e a VIA Technologies, com 2% do mercado)
- AMD e Intel têm um acordo completo de **licenciamento cruzado de patentes**
  - Intel patentou o conjunto de instruções original (x86 de 32 bits, o IA-32)
  - AMD patentou o conjunto de instruções extendido (x86 de 64 bits, o AMD64)
  - Ambos precisam da patente um do outro ⟶ Não revogam a licença uma das outras

Arquitetura do Conjunto de Instruções

Processadores CISC

Cenário Histórico em 1970

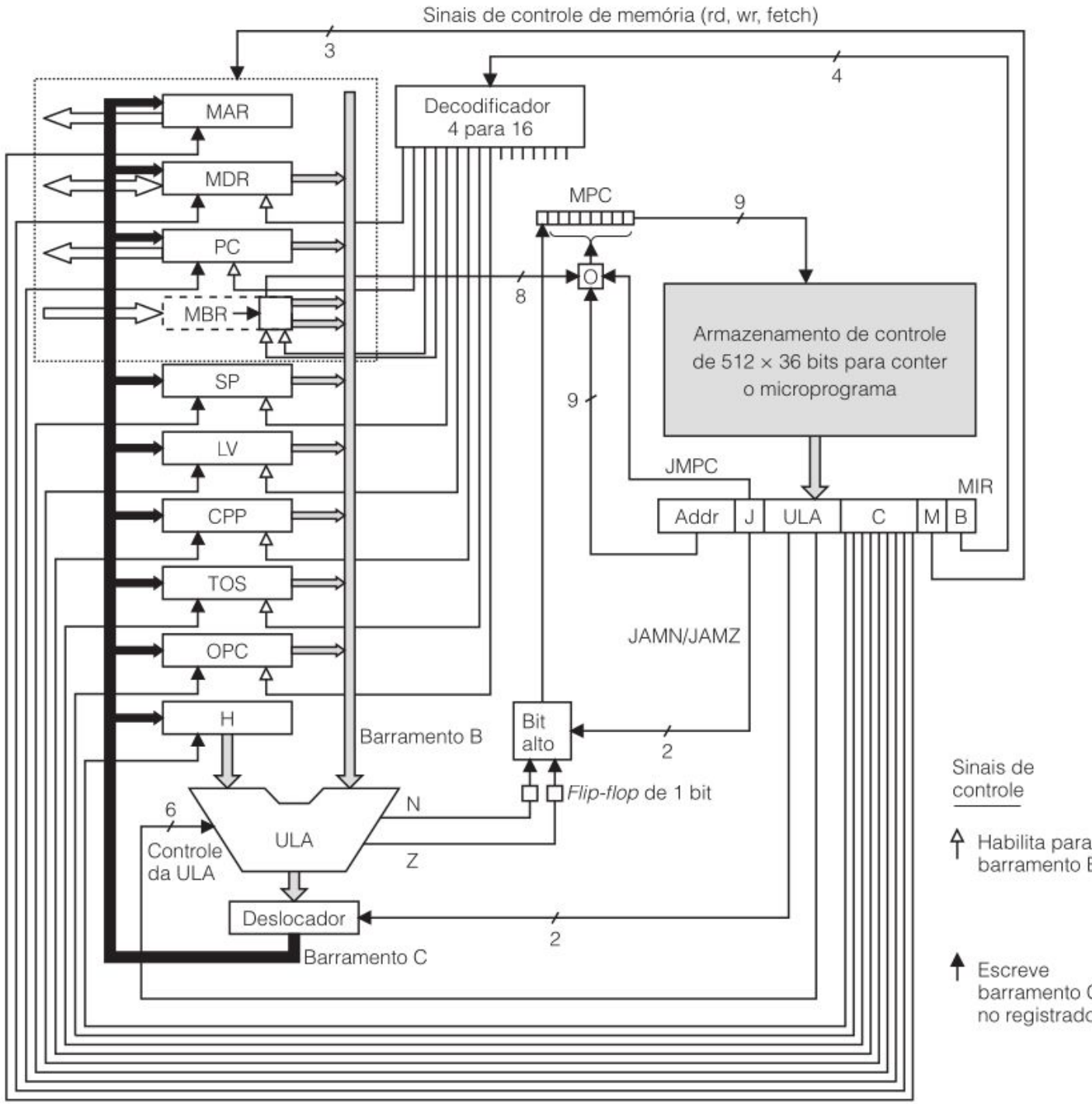
- Os custos do *software* excediam em muito os custos do *hardware*
- Linguagens de alto nível cada vez mais complexas
  - **Lacuna semântica** entre capacidades de *hardware* e demanda de *software*
- Instruções simples levavam a:
  - Código ineficiente
  - Tamanho excessivo do programa de máquina
  - Complexidade do compilador
  - Bancos de registradores pequenos e.g. acumulador (AC)
- Memória externa à CPU era muito cara e lenta
  - Lenta transição de núcleo de ferrite para DRAM
  - Solução: Reduzir a quantidade de instruções necessárias para realizar operação

Filosofia de Projeto CISC

- *Complex Instruction Set Computer*: Muitas instruções complexas
  - Visava reduzir o tamanho do programa e facilitar a programação
  - Baseado em **interpretação de instruções**
- Acesso à uma memória de controle dentro da CPU mais rápido do que à memória externa
  - **Memória de controle** (ROM) interna a CPU mantinha **microcódigo** que interpretava instruções
  - Uma **macroinstrução** buscada na memória → Executada por várias **microinstruções** na CPU
- Consequências:
  - Complexidade movida para o microcódigo
    - Programas *assembly* e compiladores mais simples
  - Conjuntos de instruções maiores e mais poderosos
  - Mais modos de endereçamento
  - Implementações em *hardware* de instruções HLL
- Inspirou o Intel x86, DEC VAX, Motorola 68000 e IBM System/360

Exemplo: MIC-1

- Processador baseado em CISC
  - Criado pelo Tanenbaum
- Programa em Macroassembly
  - Microprograma: Microassembly
  - Interpreta sequencialmente o programa
  - Comportamento da CPU personalizável em modificar *hardware*
- Link do simulador



Exemplo: Código de um CISC

- Filosofia CISC procurava diminuir a lacuna semântica
  - Acessos a memórias mais frequentes: HLLs não usavam registradores diretamente

1. S = W/(Z*(X + Y))

(HLL)

1. LOAD X
2. ADD Y
3. STOR T
4. LOAD MQ, T
5. MUL Z
6. LOAD MQ
7. STORE T
8. LOAD W
9. DIV T
10. LOAD MQ
11. STOR S

(IAS, anterior ao CISC)

1. MOV BX, [X]
2. ADD BX, [Y]
3. IMUL BX, [Z]
4. MOV AX, [W]
5. CWD
6. IDIV BX
7. MOV [S], AX

(x86, um CISC)

Cenário Histórico em 1980

- Avanços nas tecnologias de **semicondutores** e introdução da **memória cache**
  - Redução no **tempo de acesso** às instruções da memória
  - Uso de memória de controle ainda continua vantajosa?
- Compiladores cada vez mais inteligentes
  - Escrever código *assembly* manualmente cada vez menos vantajoso
  - Instruções complexas **raramente** utilizadas
- Técnica de *Pipelining*
  - Instruções complexas levam a mais hazards de pipelines
  - Instruções mais simples extraem maior desempenho do *pipeline*

Vale a Pena Investir em CISC?

- Programa ocupada menos memória, mas a memória é barata
  - Mais instruções exigem **opcodes mais longos**
  - Programa visualmente menor (menos instruções): Porém instruções levam vários bytes
  - Referências a memória exige mais bits que a registrador
- Instruções Complexas ↔Programas mais Rápidos?
  - Estudos indicaram que **instruções mais simples** são mais usadas
  - Compilador tem dificuldades de encontrar casos em que instruções complexas se encaixam
- Unidade de Controle mais **complexa**
  - **Tempo de interpretação** significativo
  - Instruções carregam vários argumentos de controle do microcódigo
  - Até instruções simples levam muito tempo pra executar
- Não é tão claro que CISC é a solução apropriada

Arquitetura do Conjunto de Instruções

Processadores RISC

Filosofia de Projeto RISC

- *Reduced Instruction Set Computer*: Pequeno conjunto de instruções simples
  - Visa simplificar o projeto da CPU ⟶ Aumenta a performance geral da execução
  - Ao invés de um microcódigo flexível ⟶ Unidade de controle *hardwired* (controle fixo)
- Principais características:
  - Grande número de **registradores** de uso geral e.g. 32 no RISC-V
  - Instruções específicas para **acessar a memória** (arquitetura *load-store*)
    - Operações de processamento ⟶ De registrador-a-registrador
  - **Conjunto de instruções** simples ⟶ Instruções de mesmo tamanho executadas em um ciclo
    - Poucos modos de endereçamento simples e.g. ARM tem três para acessar a memória
    - Poucos formatos de instrução disponíveis
  - Ênfase na otimização do **pipeline** de instrução e.g. predição de salto
  - Complexidade movida para o compilador e escrita de código em assembly
- Inspirou o ARM, MIPS, AVR, SPARC, PowerPC e RISC-V

Exemplo: Código de um RISC

- Otimiza recursos mais usados: Operações simples se sobresaem

1. Z = X + Y

(HLL)

1. MOV EAX, [X]
2. ADD EAX, [Y]
3. MOV [Z], EAX

(x86, um CISC)

1. MOV R0, #X
2. LDR R1, [R0]
3. MOV R1, #Y
4. LDR R2, [R0]
5. ADD R1, R1, R2
6. MOV R0, #Z
7. STR R1, [R0]

(ARM, um RISC)

Banco de Registradores

- Processadores CISC utilizam uma pequena quantidade de registradores comparado aos RISCs
  - Registradores →Armazenamento mais rápido o possível
  - Endereços bem menores que os de memória
- Processadores CISC utilizam mais as memórias cache para fornecer os dados e instruções
  - RISC foca em operações registrador-a-registrador: Minimizando acessos à memória

Grandes bancos de registradores	Cache
Todas variáveis locais escalares	Variáveis locais recentemente usadas
Variáveis individuais	Blocos de memória
Variáveis globais assinaladas pelo compilador	Variáveis globais recentemente usadas
Salvar/restaurar baseados na profundidade de aninhamento do procedimento	Salvar/restaurar baseado em algoritmos de atualização da cache
Endereçamento de registrador	Endereçamento de memória
Múltiplos operandos endereçados e acessados em um ciclo	Um operando endereçado e acessado por ciclo

Exemplo: ARM na Apple

- ARM transitou bastante na dúvida entre CISC e RISC!

1984

Motorola 68000
(CISC)

1994

Motorola PowerPC
(RISC)

2005

Intel x86
(CISC)

2020

Apple Silicon (ARM)
(RISC)

Características CISC vs. RISC

- **Ciclos por instrução**: Vários vs .um
- **Número de registradores**: Poucos (e.g. 8 no x86) vs. vários (e.g. 32 no MIPS)
- **Operandos por instrução**: Um/dois vs. três
  - No x86: ADD EAX, EBX
  - No ARM: ADD R0, R1, R2
- **Número de Instruções**: Muitas e complexas vs. poucas e simples
- **Comprimento das Instruções**: Variável (até 15 bytes no x86) vs. Fixo (4 bytes no MIPS)
- **Unidade de Controle**: Microprogramado vs. *Hardwired*
- **Complexidade**: *Hardware* (microcódigo) vs. *Software* (compilador)

Operações de Acesso à Memória

- Apenas algumas instruções acessam memória no RISC:
  - lw r2, 128(r3)
  - sw r4, 64(r0)
- Varias instruções podem acessar a memória no CISC:
  - add [esi], eax
  - sub ebx, [edi]
- Maioria das operações usam apenas registradores no RISC:
  - add r1, r2, r3
  - sub r4, r2, r1

Cenário Atual

- Quando o RISC foi introduzido: Foco eram *desktops* e servidores
  - Objetivo era **performance** e os limitadores eram **área** de *chip* e **complexidade** de projeto
- Atualmente: Foco em **sistemas embarcados** (tomados pelo ARM, baseado em RISC)
  - Processadores baseados em CISC ainda dominam sistemas maiores
- "*Se a filosofia RISC é melhor, por que ainda existem CPUs CISC?*"
  - Assunto delicado: Razões históricas, técnicas, econômicas e de compatibilidade
  - Legado e compatibilidade com o x86: Trocar plataforma seria inviável
  - Evolução da Microarquitetura: Microinstruções são basicamente RISC
- Convergência entre CISC e RISC
  - Processadores CISC com **microcódigo baseado em RISC** e.g. x86-64
  - Processadores RISC com **instruções complexas** e.g. Extensão NEON do ARM

Exemplos de Arquiteturas RISC

- AVR (8 bits)
  - Possui instruções de até 16 bits e permite processamento de 16 bits
- PIC (8 e 16 bits)
- ARM (32 bits)
  - Até o ARMv7
  - Versão de 64 bits no ARMv8
  - Versão “mesclada” de 16 e 32 bits (modo Thumb)
- MIPS (32 bits)
- RISC-V (32 bits)
  - Versão “comprimida” de 16 e 32 bits (extensão *compressed*)

Arquitetura do Conjunto de Instruções

Conclusão

Resumo da Aula

- O **Conjunto de Instruções** define a operação de um processador
- Instruções podem ser representadas em **binário** (executável) ou **assembly** (mais legível)
  - Instruções podem ter vários **formatos** de instrução
  - Podem ter vários **registradores** ou só um (AC)
- Arquiteturas (como **x86** e **ARM**) implementam decisões diferentes: Propósitos diferentes
- CISC e RISC não são regras de projeto, mas filosofias: Influenciaram no projeto de muitas CPUs
- CISC preza por flexibilidade e um conjunto de instruções rico
- RISC preza por performance e um projeto simplificado
  - Muitos registradores e acesso à memória por instruções específicas
  - Instruções simples que executam a um ciclo de clock
- Com o tempo, definições foram convergindo: CISC ↔ RISC
- Mas atenção: Quantidade de instruções não importa, mas sim sua performance

Conclusão

- Nessa Aula:
  - Arquitetura do Conjunto de Instruções
- Bibliografia Principal:
  - Arquitetura e Organização de Computadores; Stallings, W.; 10ª Edição (Capítulo 12)
- Próxima Aula:
  - Projeto e Implementação de Processadores