Arquitetura de Computadores

Instruction Set Architecture (ISA)

UFMT

Fevereiro de 2025

Agenda

- Introdução.
- Programa Armazenado e Modelo de Von Newman.
- Arquiteturas CISC *versus* RISC.
- MIPS:
    1. Visão geral.
    2. Assembly *vs* Linguagem de Máquina.
    3. Conjunto de Instruções.

Parte 1

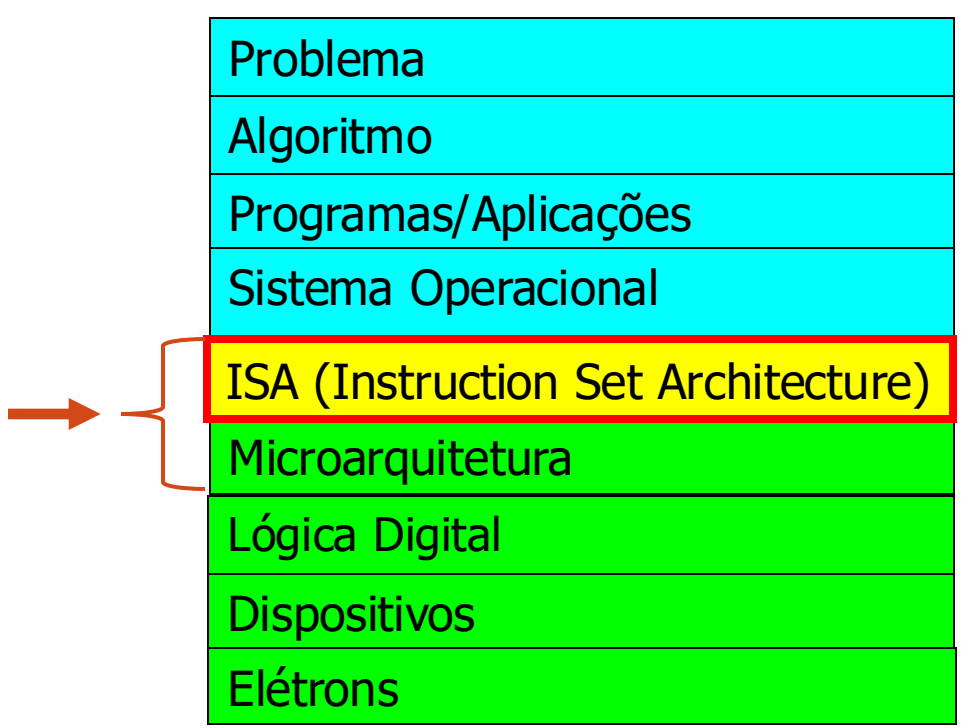
    4. Programação.
    5. Modos de Endereçamento.
    6. Compilação, Montagem, Ligação e Carga.
    7. Pseudo-instruções.
    8. Exceções.
    9. Números em Ponto Flutuante.

Parte 2

Introdução à Arquitetura de Computadores

A Hierarquia de Transformações

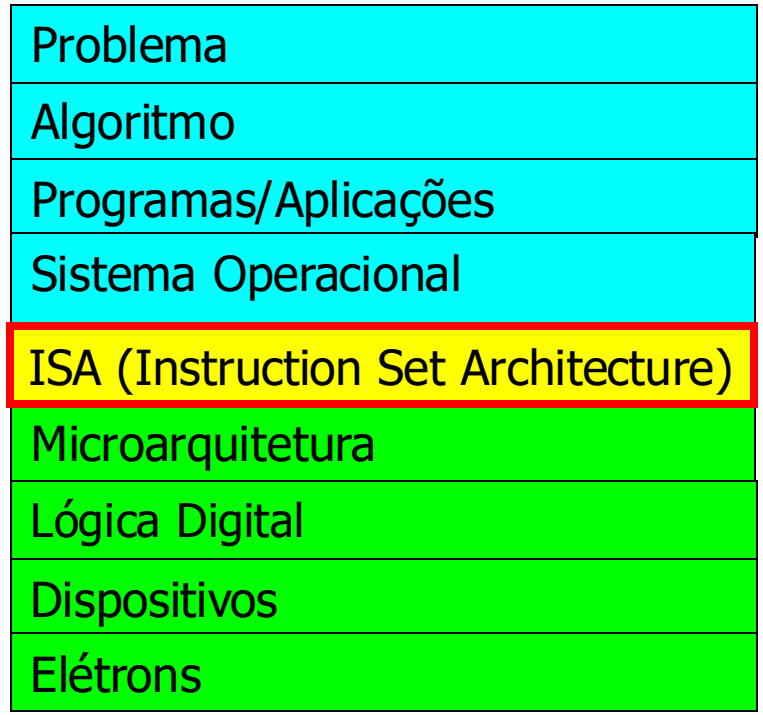
Foco dessa disciplina:
ISA e Microarquitetura
do Processador



Um computador pode ser estudado analisando o processamento de informações em diferentes níveis de abstração

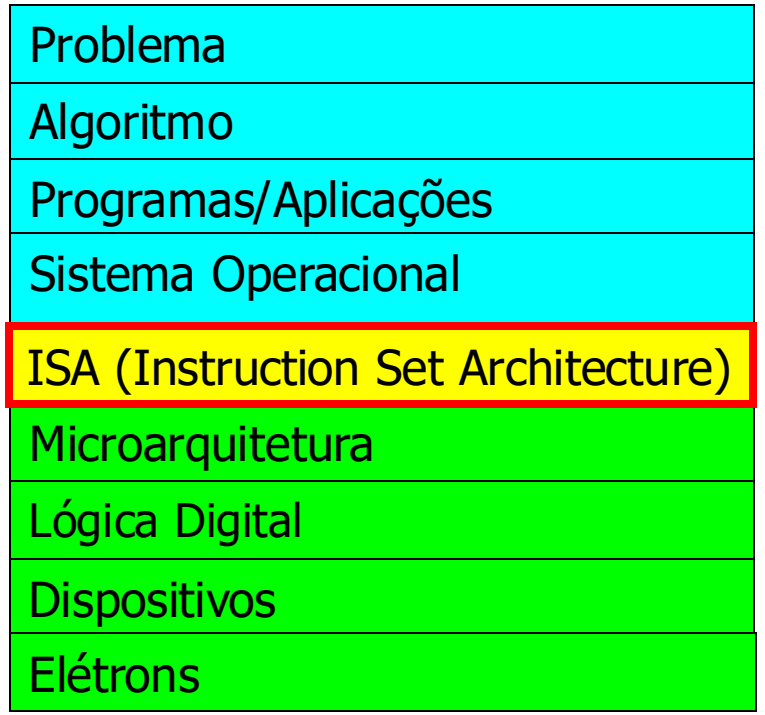
Instruction Set Architecture (ISA)

- A ISA é a interface que define como o software se comunica com o hardware.
- Consiste na especificação do **conjunto de instruções** que o processador pode executar, incluindo modos de endereçamento, tipos de dados e organização da memória.
- Cada instrução é um **comando** que o processador entende e executa. De modo geral, as instruções são divididas em **lógicas**, **aritméticas**, **de movimentação de dados** e de **controle de fluxo** de execução.



Instruction Set Architecture (ISA)

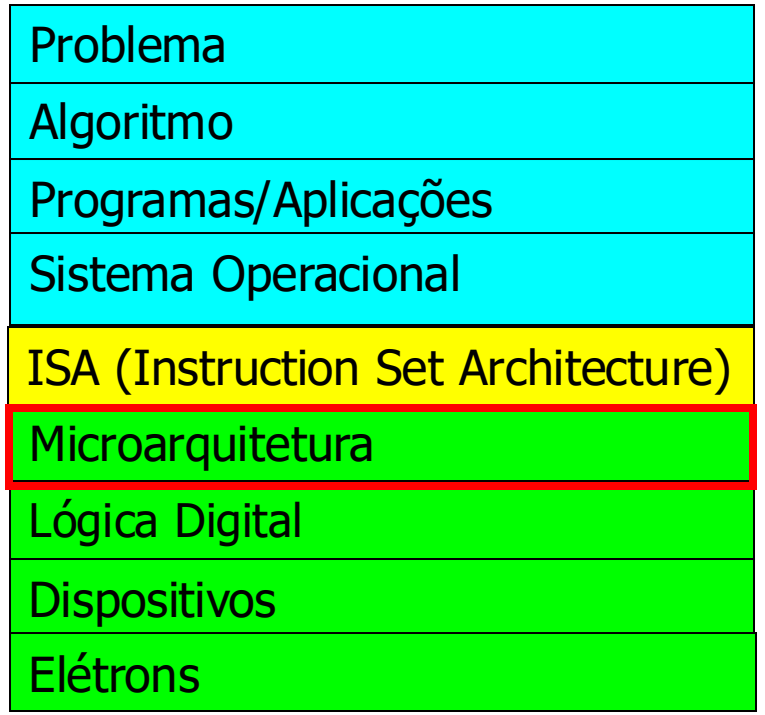
- Programas escritos em linguagem de alto nível (como C ou Python) precisam ser compilados/traduzidos para uma sequência de instruções que fazem parte da ISA da máquina para serem executados.
- Portanto, a ISA determina a visão que o programador (em linguagem de máquina) tem do computador, isto é, o que ele pode fazer.
- Existem muitas arquiteturas diferentes, como **RISC-V**, ARM, x86, **MIPS**, SPARC e PowerPC.



Microarquitetura

- Enquanto a ISA define as funcionalidades esperadas do processador, a microarquitetura trata da **implementação física** do hardware que executa as instruções definidas pela ISA.

- Envolve decisões sobre o projeto físico do processador:
  - Unidades funcionais, como a ULA.
  - Pipelines.
  - Cache e Hierarquia de Memória.
  - Unidade de Controle
  - Interligação de componentes, etc.



Microarquitetura

- Frequentemente, existem diversas implementações diferentes de hardware (microarquiteturas) para uma mesma ISA. Embora executem o mesmo conjunto de instruções, podem ser otimizados para oferecer diferentes compromissos em termos de desempenho, preço e consumo de energia.
- Por exemplo, um processador que implementa uma determinada ISA pode ser otimizado para operar como um servidor de alto desempenho, enquanto outros para oferecer longa duração de bateria em laptops.
- A Intel e a AMD vendem diversos microprocessadores que implementam à mesma arquitetura x86. Todas elas podem executar os mesmos programas, mas utilizam hardwares subjacentes diferentes.

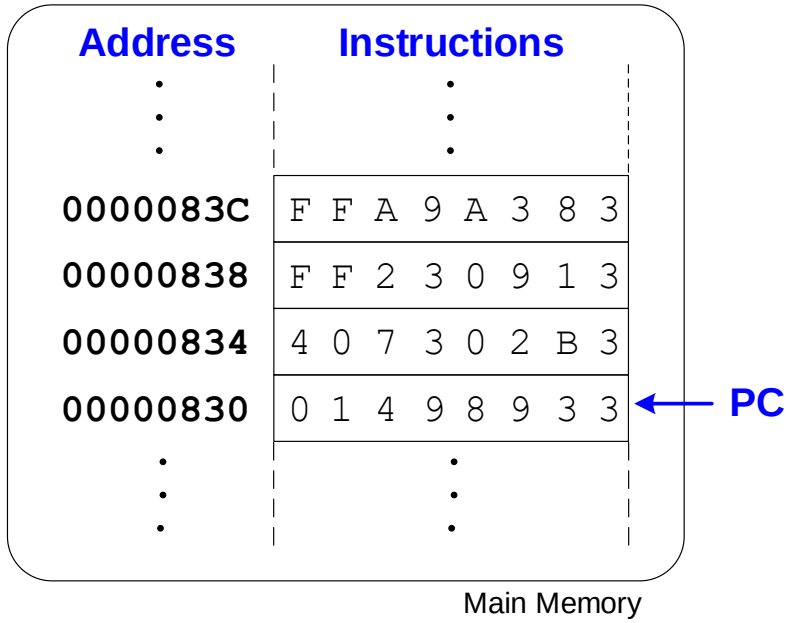
A Arquitetura dos Primeiros Computadores

- Os primeiros computadores eram especializados, isto é, projetados para resolver tarefas muito específicas, tais como cálculos matemáticos complexos.
  A "programação" era feita ajustando fisicamente o hardware, num processo que envolvia reconectar cabos, configurar painéis de controle, ajustar interruptores e, em alguns casos, trocar cartões ou fitas perfuradas.

- Problemas:
  - Reconfigurar fisicamente um computador podia levar dias, dependendo da tarefa.
  - Erros na reconexão de cabos ou configuração de chaves eram comuns, o que causava falhas no programa.
  - As máquinas eram projetadas para realizar um conjunto específico de tarefas, e sair disso era complicado.

Computadores com Programa Armazenado

Assembly Code	Machine Code
`add  s2, s3, s4`	`0x01498933`
`sub  t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw   t2, -6(s3)`	`0xFFA9A383`



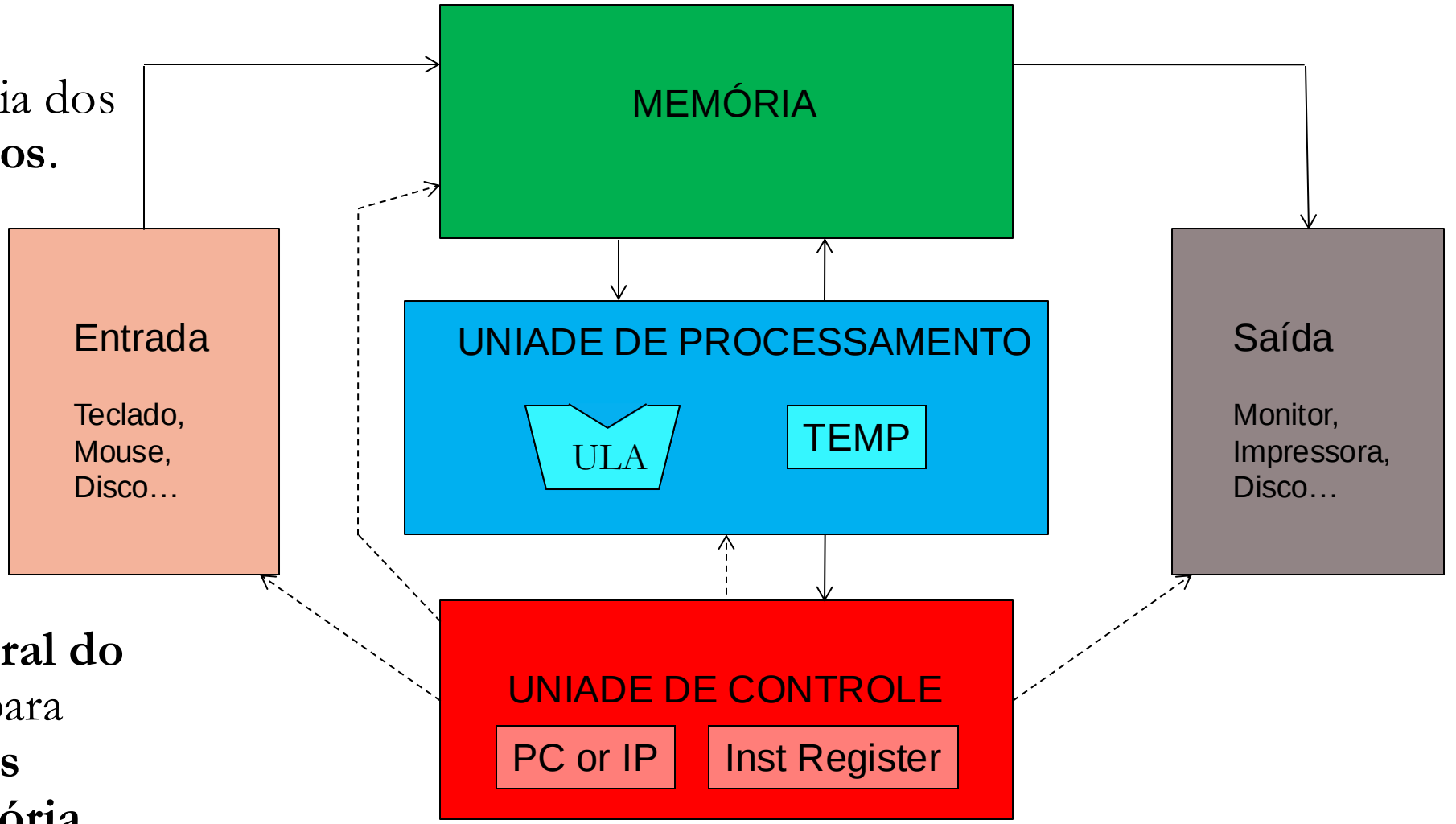
- Os computadores modernos de propósito geral possuem arquiteturas baseadas no conceito de **Programa Armazenado**.
- Cada tarefa é definida por um programa (sequência de instruções) carregado na memória juntamente com seus dados.
- O processador é projetado para executar as instruções que compõe os programas.
- Essa ideia elimina a necessidade de um novo hardware para cada nova tarefa/aplicação.

Alan Turing, 1912 - 1954

- Matemático e cientista da computação britânico, fundador da computação teórica, inventou a máquina de Turing, um modelo matemático de computação.
- Projetou o "Automatic Computing Engine", um dos primeiros computadores com **programa armazenado**.
- Em 1952, foi processado por ser homossexual. Dois anos depois morreu envenenado por cianeto.
- O Prêmio Turing, maior prêmio da computação, foi nomeado em sua homenagem.

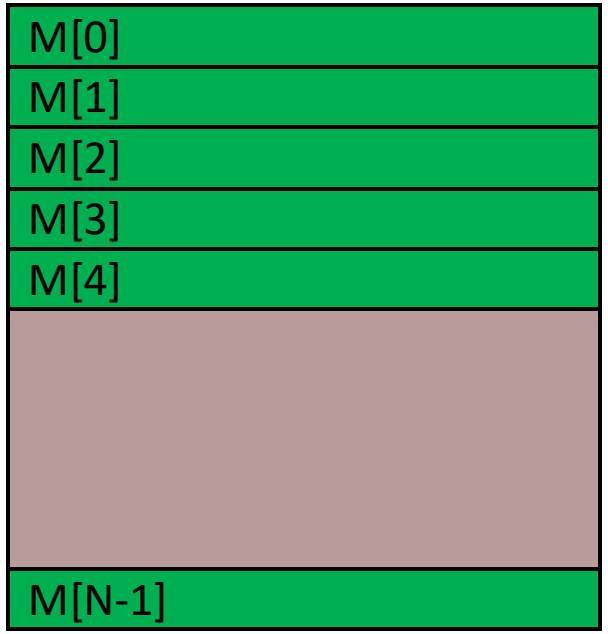
O Modelo de von Neumann

Modelo base para a maioria dos **computadores modernos.**



Define a **organização geral do hardware** necessário para **executar programas armazenados em memória.**

Estado Arquitetural Visível ao Programador



Memória

Array de localizações para armazenamento
Indexada por um endereço

Registradores

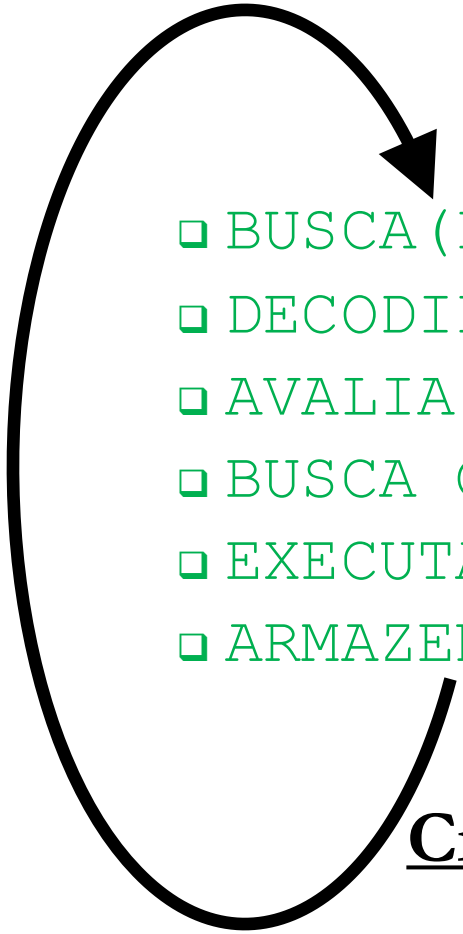
- Recebem nomes especiais na ISA.
- De propósito general vs. especial.

Contador de Programa

Endereço de memória da próxima instrução a ser executada,

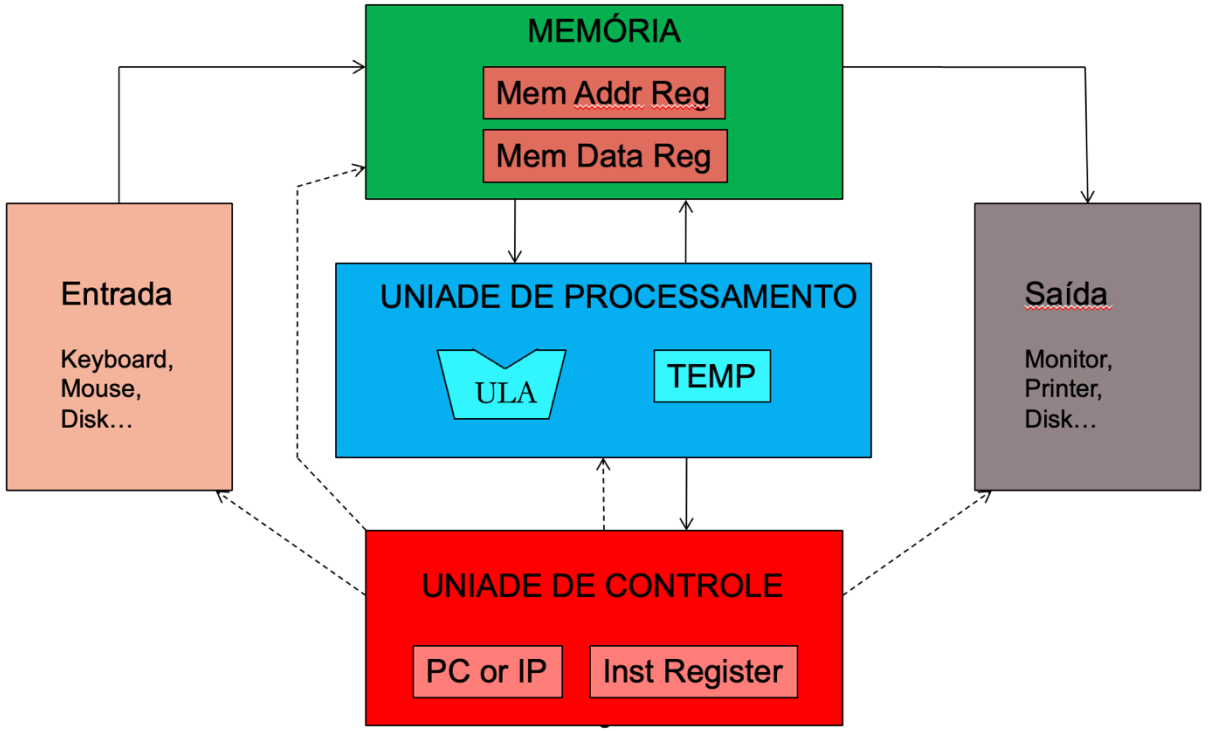
A execução de instruções transformam os valores que definem o estado arquitetural.

Programas Armazenado & Fases do Ciclo de Execução de Instruções



- BUSCA(FETCH)
- DECODIFICA (DECODE)
- AVALIA ENDEREÇO
- BUSCA OPERANDOS
- EXECUTA (EXECUTE)
- ARMAZENA RESULTADO

Ciclo de Instrução



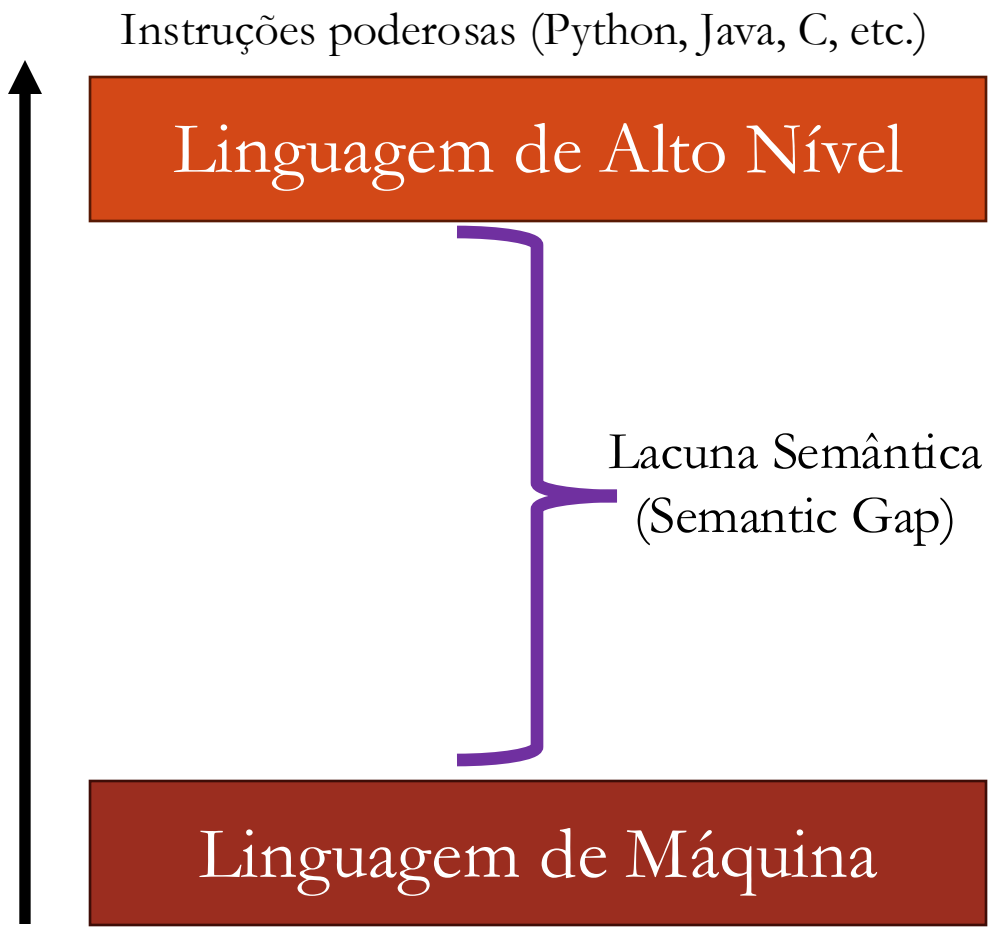
Programa Armazenado & Ciclo de Instrução

- As instruções e os dados ficam **armazenados na memória**. Tipicamente, uma instrução tem o comprimento (em bits) de uma palavra.
- O processador, mais precisamente a unidade de controle, opera um looping que processa **sequencialmente** as instruções que compõe o programa.
  - Instruções de desvio podem alterar o fluxo de execução sequencial.
- O endereço da próxima instrução é armazenado em um registrador especial conhecido como **Contador de Programa** (PC).
- No MIPS, o Sistema Operacional tipicamente inicializa o PC com o endereço 0x00400000, onde o programa é carregado.

Paradigmas Gerais de Arquiteturas de Computadores

CISC *versus* RISC

Lacuna Semântica entre Linguagens



A função do compilador é exatamente fechar a lacuna semântica

CISC

- Linguagens de programação de alto nível possuem instruções poderosas que são traduzidas em um grande número de instruções de máquina mais simples.
- No passado, pensava-se que tornar as instruções de máquina mais poderosas/complexas estreitaria a lacuna semântica, tornando o código compilado mais rápido.
  - Exemplos de tais instruções incluem acesso complicado a arrays em uma única instrução.
- Processadores com conjuntos de instruções complexas são chamadas CISC (*Complex Instruction Set Computer*). Complexo no sentido de que certas instruções da ISA fazem coisas complicadas, difíceis de implementar em hardware.

CISC

- Em alguns casos, as arquiteturas CISC resultaram em código compilado mais rápido. Mas, na década de 70, os pesquisadores notaram que:
  - Os compiladores faziam pouco uso das instruções CISC complexas.
  - As instruções complexas tendiam a ser mais lentas do que uma sequência equivalente de instruções mais simples e otimizadas.
  - Os programas reais, muitas vezes, passavam a maior parte do tempo executando instruções simples.
  - Instruções complexas resultam em microarquiteturas difíceis de implementar, com restrições que retardam a execução de instruções simples.

RISC

- As percepções empíricas sobre os projetos CISC levaram à instituição de princípios de projeto que mudaram radicalmente as ISAs e culminaram nas arquiteturas RISC (*Reduced Instruction Set Computer*).

Princípio de Projeto RISC

"*Otimizar ao máximo um número reduzido de instruções simples muito utilizadas na prática. Torná-las extremamente rápidas.*"

- As arquiteturas RISC tornam a tarefa do compilador muito mais difícil, mas o compilador precisa compilar um programa apenas uma vez.

RISC

- A arquitetura **MIPS** foi umas das primeiras arquiteturas RISC, o **RISC-V** é, de certa forma, sua evolução.
- Uma característica importante das arquiteturas RISC é que a memória principal só pode ser acessada por instruções de carregamento (`load`) e armazenamento (`store`). Todas as outras instruções são limitadas a operar com dados armazenados nos **registradores internos** do processador.
- Como veremos, essa abordagem simplifica drasticamente o projeto do processador, permitindo que as instruções tenham tamanho fixo, facilitando a implementação de pipelines eficientes e isolando a lógica necessária para lidar com atrasos nos acessos à memória.

RISC *versus* CISC

Característica	RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
Conjunto de Instruções	Pequeno e simples, com instruções de execução rápida	Grande e complexo, com instruções que realizam tarefas sofisticadas
Tamanho das Instruções	Tamanho fixo	Tamanho variável
Acesso à Memória	Apenas instruções *load* e *store* acessam a memória	Instruções podem acessar memória diretamente
Eficiência	Mais eficiente em energia e processamento paralelo	Maior densidade de código, reduzindo o uso de memória
Complexidade do Hardware	Simples, facilitando otimizações como pipelining	Mais complexo, com decodificação elaborada de instruções
Exemplos	ARM, MIPS, RISC-V	Intel x86, AMD64

Atualmente...

- Embora o RISC ofereça vantagens técnicas como simplicidade e eficiência energética, a família de chips para desktop/servidor mais popular (Intel x86) continua baseada em uma arquitetura CISC.

- As razões para isso incluem a grande quantidade de código legado (produtos Microsoft), e a compatibilidade com versões anteriores, desde o processador 8086 (1978). A Intel investiu pesadamente em pesquisa, mantendo seus chips CISC competitivos com as arquiteturas RISC. Além disso, os processadores Intel modernos implementam ideias inspiradas no RISC. As instruções complexas x86 são traduzidas, em tempo de execução, em micro-ops mais simples, permitindo uma execução eficiente sem que isso seja visível para o programador.

- No entanto, com o crescimento das arquiteturas ARM em diversos setores, a dominância tem mudado. Os telefones celulares são dominados por arquiteturas ARM, que seguem o modelo RISC, embora incorporem algumas otimizações modernas que lembram características de CISC.

Arquiteturas Especializadas

- Para certas cargas de trabalho altamente especializadas, pode fazer sentido complementar ou substituir arquiteturas tradicionais (incluindo RISC) por processadores para fins específicos:
  - GPUs (unidades de processamento gráfico) exploram o alto grau de paralelismo SIMD e SIMT, tornando-as ideais para algoritmos de processamento gráfico e treinamento de redes neurais.
  - O Bitcoin levou ao desenvolvimento de ASICs (circuitos integrados de aplicação específica) dedicados exclusivamente ao cálculo da função hash criptográfica SHA-256.
  - As TPUs (Tensor Processing Units) do Google aceleram a computação em redes neurais, oferecendo desempenho significativamente superior às GPUs em tarefas específicas, com ganhos de 15x a 30x em velocidade e até 70x mais eficiência energética em comparação com GPUs convencionais.

MIPS

MIPS: Visão Geral

- Uma das primeiras arquiteturas de processador RISC de sucesso.
- Originalmente desenvolvida por John Hennessy e seus colegas em Stanford na década de 1980.
- Processadores MIPS foram amplamente utilizados em diversas aplicações, incluindo estações de trabalho da Silicon Graphics, consoles de videogame da Nintendo e equipamentos de rede da Cisco.
- Atualmente, ainda encontra aplicação em sistemas embarcados, roteadores e dispositivos de rede, além de ser uma referência importante no ensino de arquitetura de computadores devido à sua simplicidade e eficiência.

John Hennessy

- Professor de Engenharia Eletrica e Ciência da Computação em Stanford desde 1977.
- Co-inventou o paradigma RISC (Reduced Instruction Set Computer) juntamente com David Patterson.
- Desenvolveu a arquitetura MIPS em Stanford em 1984 e co-fundou o MIPS Computer Systems.
- Em 2004, mais de 300 milhões de microprocessadores MIPS já tinham sido vendidos.

Princípios de Projeto Articulados por Hennessy e Patterson

1. **A simplicidade favorece a regularidade.**
2. **Torne rápido os casos comuns (mais usados).**
3. **Menor é mais rápido.**
4. **Bons projetos exigem bons compromissos.**

Instruções MIPS

Abordagem

- Começaremos estudando instruções MIPS básicas, em **Assembly** e **Linguagem Máquina**.
- Em seguida, estudaremos as instruções usadas em construções comuns de programação:
  - Testes de condições.
  - Loops.
  - Manipulações de array.
  - Chamadas de função.
- Finalmente, estudaremos exceptions e um pouco sobre a relação com o Sistema Operacional.

Linguagem de Máquina *versus* Assembly

- As instruções da ISA podem ser expressas em diferentes níveis de abstração:
  - **Linguagem de Máquina**: instruções codificadas usando 0's e 1's em formato **diretamente** executável pelo hardware.
  - **Linguagem Assembly:** Formato simbólico, que usa mnemônicos (e.g., `add`, `sub`) para facilitar a leitura por nós humanos.
- Instruções em Assembly precisam ser **traduzidas** para Instruções de Máquina para serem executadas.
- Quem faz a tradução é um programa chamado **Assembler**, na correspondência de uma instrução assembly para uma instrução de máquina.

MIPS
Instruções Aritméticas

Adição (`add`) e Subtração (`sub`) de inteiros

Adição de Números Inteiros

Código C

```
a = b + c;
```

Código Assembly MIPS

```
add a, b, c
```

- **`add:`** mnemônico que indica a operação a ser realizada.
- **`b, c:`** operandos fonte (sobre os quais a operação é realizada).
- **`a:`** operando de destino (para o qual o resultado é escrito).
- **`Resultado:`** o registrador `a` recebe a soma dos conteúdos dos registradores `b` e `c` (`a` ← `b+c`).
- Observação: neste ponto os parâmetros estão sendo apresentados de forma abstrata, sem nenhuma vinculação com a sua representação física.

Subtração de Números Inteiros

- Similar a adição – somente o mnemônico muda.

Código C

```
a = b - c;
```

Código Assembly MIPS

```
sub a, b, c
```

- **`sub:`** mnemônico.
- **`b, c:`** operandos fonte.
- **`a:`** operando de destino.
- **`Resultado:`** o registrador `a` recebe a subtração dos conteúdos dos registradores `b` e `c` (`a` ← `b+c`).

Princípios de Projeto em Ação

"A Simplicidade Favorece a Regularidade."

- Note a consistência no formato das instruções `add` e `sub`. Elas são simples e regulares (mesmo número de operandos: dois fontes e um destino).
- Isso facilita a codificação em linguagem de máquina (0s e 1s) e simplifica o hardware necessário para conduzir a execução.
- A ideia das arquiteturas RISC (*Reduced Instruction Set Computer*), tais como o MIPS, é ter um conjunto pequeno de instruções simples e eficientes, a fim de minimizar a complexidade do hardware e a codificação de instruções.

Tradução de Código mais Complexo

- Instruções mais complexas de linguagem de alto nível, que não possuem instruções MIPS equivalentes, são tratadas combinando múltiplas instruções MIPS mais simples.
  - Obs: no caso de arquiteturas CISC, o *gap* semântico é menor mas, ainda assim, em geral, há a necessidade de tradução para múltiplas instruções.

- Exemplo:

Código C

```
a = b + c - d;
```

Código MIPS em Assembly

```
1. add t, b, c    #t = b + c
2. sub a, t, d    #a = t - d
```

Exercício

- Apresente uma possibilidade de tradução para assembly MIPS do código C abaixo. Considere que todas as variáveis são inteiras de 4 bytes atribuídas aos registradores `$s0`, `$s1`, `$s2` e `$s3`.

```
f = (g + h) - (h + i);
```

Exercício

- Apresente uma possibilidade de tradução para assembly MIPS do código C abaixo. Considere que todas as variáveis são inteiras de 4 bytes atribuídas aos registradores `$s0`, `$s1`, `$s2` e `$s3`.

```
f = (g + h) - (h + i);
```

- **Assembly MIPS:**

```
1. add $t0, $s1, $s2
2. add $t1, $s2, $s3
3. sub $s0, $t0, $t1
```

Princípios de Projeto em Ação

“Torne os Casos Comuns Rápidos!”

- Usar múltiplas instruções Assembly para realizar operações mais complexas de alto nível é um exemplo do segundo princípio de projeto articulado Hennessy e Patterson.
- Instruções complexas (menos comuns em programas) são realizadas usando combinações de instruções simples **altamente otimizadas**.
- MIPS possui apenas instruções simples e muito usadas nos programas, com hardware de decodificação e execução otimizado para executá-las.

Operandos

- Cada instrução realiza uma pequena operação (e.g., somar ou subtrair) usando **operandos**.
- **Fisicamente**, há **três localizações possíveis** para os operandos de uma instrução:
  1. Registradores: extremamente rápidos, mas com capacidade limitada, ficam dentro do processador.
  2. Memória: mais lenta que os registradores porque exige acesso ao barramento de memória, mas com capacidade muito maior.
  3. Constantes (*imediatos*): valores embutidos na própria instrução, com capacidade limitada ao tamanho do campo disponibilizado.

Operandos: Registradores

- O acesso a um registrador é rápido, mas a capacidade de armazenamento é pequena. Por outro lado, o acesso à memória é relativamente lento, mas ela oferece mais espaço.

- Arquiteturas RISC, como o MIPS, tratam esse trade-off no design de instruções com uma abordagem conhecida como **load/store:** instruções lógicas e aritméticas operam **apenas com registradores e constantes**, enquanto o acesso à memória é feito exclusivamente por instruções de carregamento/leitura (`load`) ou armazenamento/escrita (`store`).

- Isso simplifica o design do processador e permite otimizações, mas também requer uma boa gestão dos registradores para minimizar acessos desnecessários à memória.

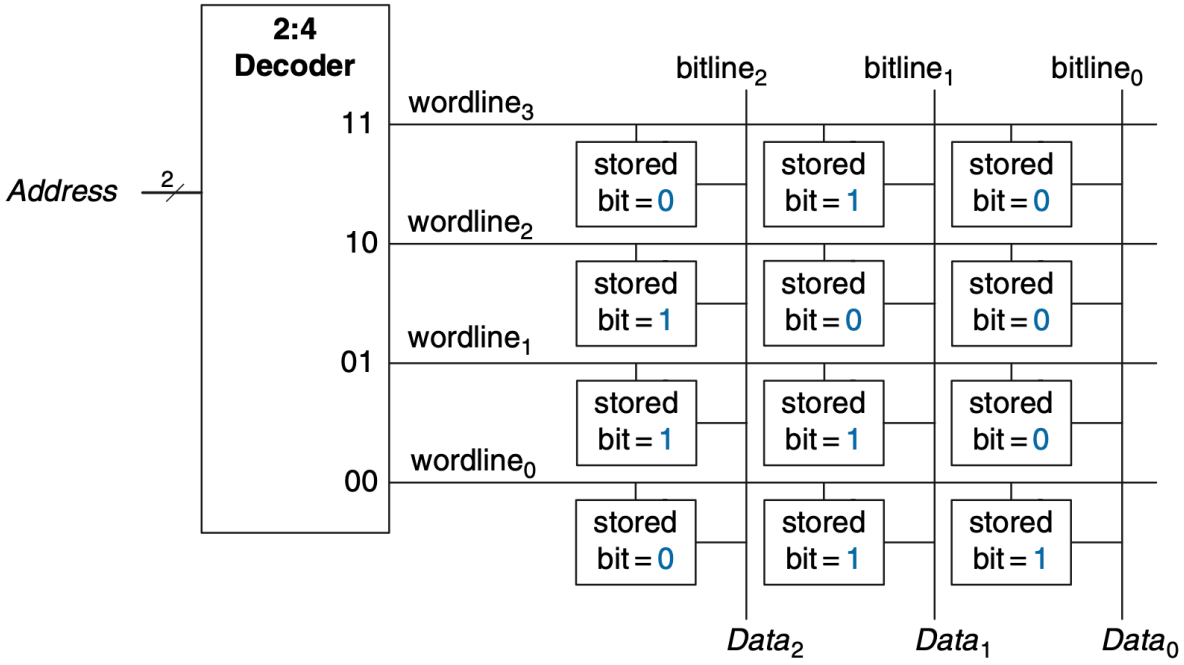
Operandos: Registradores

- A maioria arquiteturas especificam um pequeno número de registradores, chamado **Register File**, para manter **operandos** e resultados **intermediários** gerados pela execução de instruções.
- A arquitetura MIPS que estamos estudando emprega 32 registradores, cada um com 32-bits. Importante mencionar que há uma versão comercial de 64 bits.
- Decidir sobre o número de registradores envolve questões de custo, espaço e desempenho. Mais registradores significa mais espaço e mais custo, mas ler dados de um conjunto menor é mais rápido.

"Menor é mais rápido!"

Relembrando: Implementação do Register File

- O **Register File** fica no processador e é tipicamente construído como uma pequena SRAM (Static RAM).
- SRAMs usam um decodificador de endereço e bitlines conectadas às células de memória.
  - Decodificador: seleciona o registrador (palavra) correto.
  - Bitline: permitem a leitura ou escrita do registrador (palavra).



Conjunto de Registradores do MIPS

Nome simbólico	Número (em decimal)	Uso
`$0`	0	O valor constante 0.
`$at`	1	Temporário do assembler.
`$v0-$v1`	2-3	Valor de retorno de função.
`$a0-$a3`	4-7	Argumentos de Função.
`$t0-$t7`	8-15	Valores temporários.
`$s0-$s7`	16-23	Variáveis salvas.
`$t8-$t9`	24-25	Mais valores temporários.
`$k0-$k1`	26-27	Temporários do SO.
`$gp`	28	Ponteiro global.
`$sp`	29	Ponteiro de pilha.
`$fp`	30	Ponteiro de quadro (frame).
`$ra`	31	Endereço de retorno de função.

Operandos: Registradores

- **Nomenclatura simbólica dos Registradores MIPS:**
  - $ antes do nome. Exemplo: `$0`, registrador 0.
- Alguns registradores usados para **propósitos específicos**:
  - $0 sempre mantém o valor constante 0.
  - Saved registers, `$s0-$s7`, usados para armazenar variáveis ou valores temporários que precisam ser preservados após a chamada de funções.
  - Temporary registers, `$t0 - $t9`, usado para manter variáveis ou valores temporários que não precisam ser preservados após chamadas de funções.
- Discutiremos em detalhes o uso de todos os registradores, à medida que avançarmos com o curso.

Revisitando a Instrução `add`

- As instruções `add` e `sub` operam usando somente registradores. Portanto, as variáveis inteiras de alto nível (código C) devem ser mapeadas para registradores.
- **Formato geral: `add rd, rs, rt`**
  - **Mnemônico:** `add` indica soma dos registradores rs e rt.
  - **`rs, rt`:** registradores fonte com as palavras (representadas em comp. 2) a serem somadas.
  - **`rd`:** registrador de destino.
  - **Resultado:** `rd = rs + rt`.

Revisitando a Instrução `sub`

- **Formato geral:** `sub rd, rs, rt`
  - **Mnemônico:** `sub` indica a subtração dos valores em rs e rt.
  - **`rs, rt`:** registradores fonte com as palavras a serem subtraídas.
  - **`rd`:** registrador de destino.
  - **Resultado:** `rd = rs - rt`.

Revisitando as Instruções `add` e `sub`

- Exemplo:

Código C

```
a = b + c
```

```
a = b - c
```

Assembly MIPS

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

```
# $s0 = a, $s1 = b, $s2 = c
sub $s0, $s1, $s2
```

MIPS
Adição com Imediato

addi

Adição com Imediato: `addi`

- Os imediatos são operandos que ficam disponíveis na própria instrução, por isso o acesso a eles é muito rápido.
- A instrução `addi` soma o conteúdo de um registrador ao valor do *imediato*, tratando-o como um número com sinal representado em complemento de 2.
- **Formato geral: `addi rt, rs, imm`**
  - **Mnemônico:** `addi` indica soma da constante imm ao registrador rs.
  - **`rs`:** registrador fonte com a palavra a ser somada.
  - **`imm`:** constante de 16 bits a ser somada (estendida com sinal para 32 bits em complemento de 2).
  - **Resultado:** `rt = rs + sign-extended[imm].`

Adição com Imediato: `addi`

- Exemplo:

Código C

```
a = a + 4;
```

Assembly MIPS

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
```

- A constante não pode exceder 16 bits (com sinal): [-32768, +32767].
- **Pergunta**: a instrução `subi` é necessária? Por que?

Princípios de Projeto em Ação

“Bons projetos demandam bons compromissos!”

- Um único formato de instrução simplificaria o hardware, mas não é flexível. Como veremos, o conjunto de instruções MIPS está definido com três formatos principais:
  - Formato R, que usa 3 registradores como operando (e.g., `add`, `sub`).
  - Formato I, que usa 2 registradores e uma constante como operando (e.g., `addi`).
  - Formato `J`, que será discutido posteriormente, usa um imediato de 26 bits e nenhum registrador.
- O ideal é manter o número de formatos de instrução pequeno para simplificar o hardware, mas não tão pequeno a ponto de tornar a arquitetura inflexível.

MIPS
Instruções de Transferência de Dados

load/store de Palavras

Operandos: Memória

- Os registradores são rápidos, mas se fossem o único espaço de armazenamento para os operandos das instruções, no MIPS estaríamos confinados a programas com no máximo 32 variáveis.
- Por outro lado, a memória possui mais espaço, mas o acesso para leitura e gravação é mais lento.
- O MIPS aborda esse trade-off com o modelo load/store, no qual apenas instruções específicas acessam a memória para leitura e escrita.

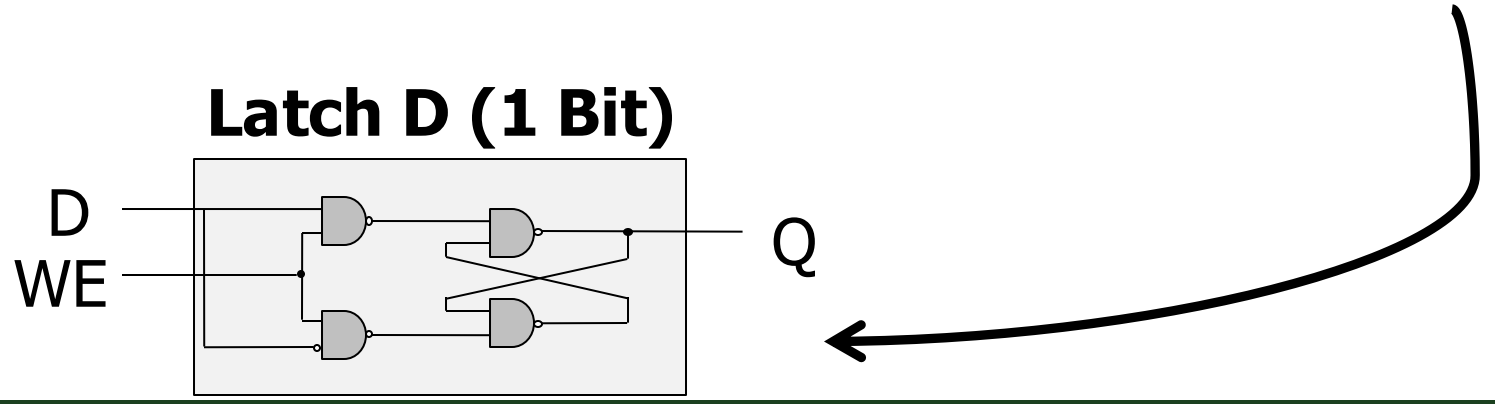
Relembrando: Matriz de Memória

- Uma matriz de memória com **endereçabilidade** de 3-bits **&** espaço de endereçamento de tamanho 2 (endereços de 1 bit) armazena um total de 6 bits.

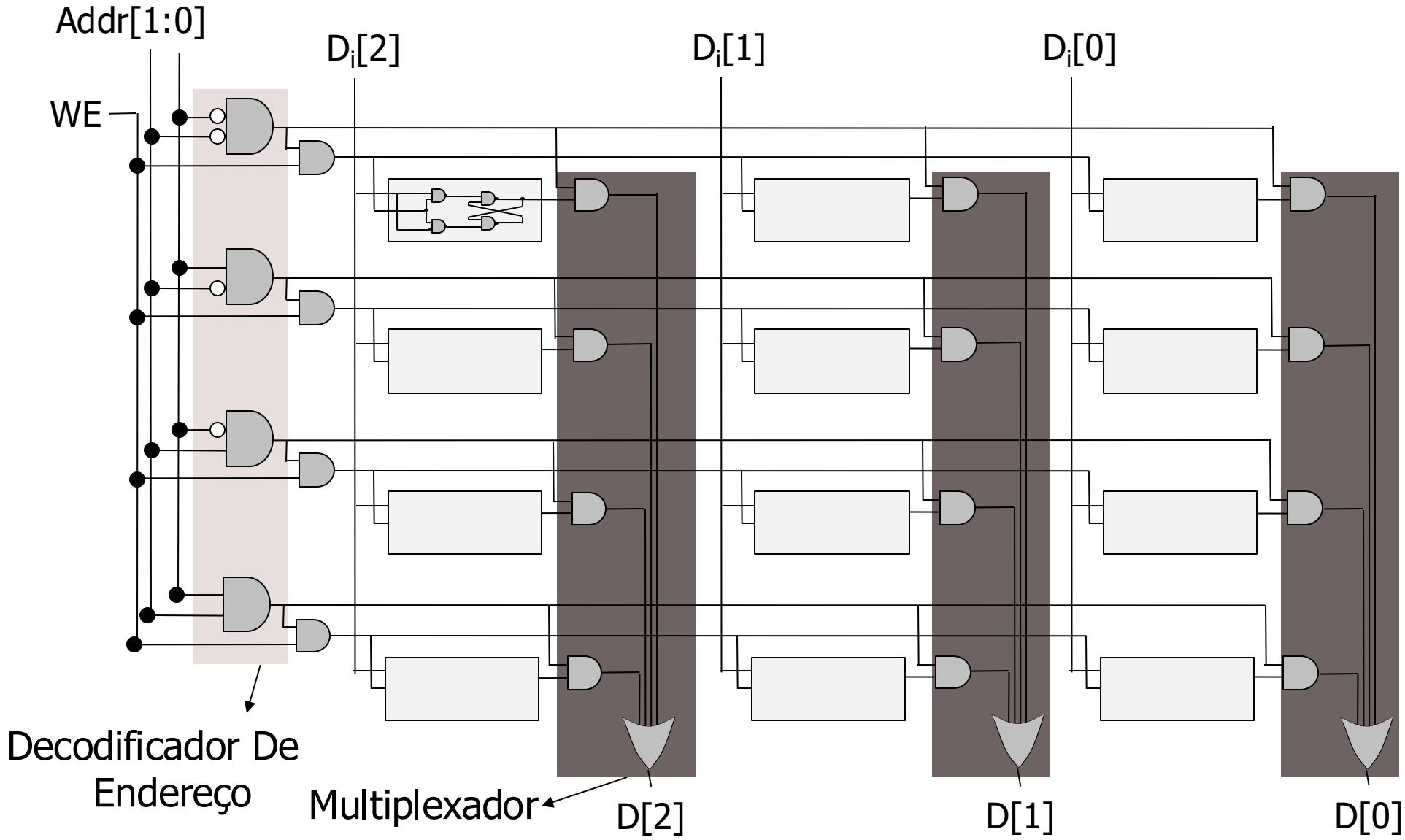
Matriz de Memória de 6 bits

Addr(0)	Bit_2	Bit_1	Bit_0
Addr(1)	Bit_2	Bit_1	Bit_0

Cada bit é armazenado em uma célula de memória que pode ser implementada de várias formas, por exemplo com um Latch D

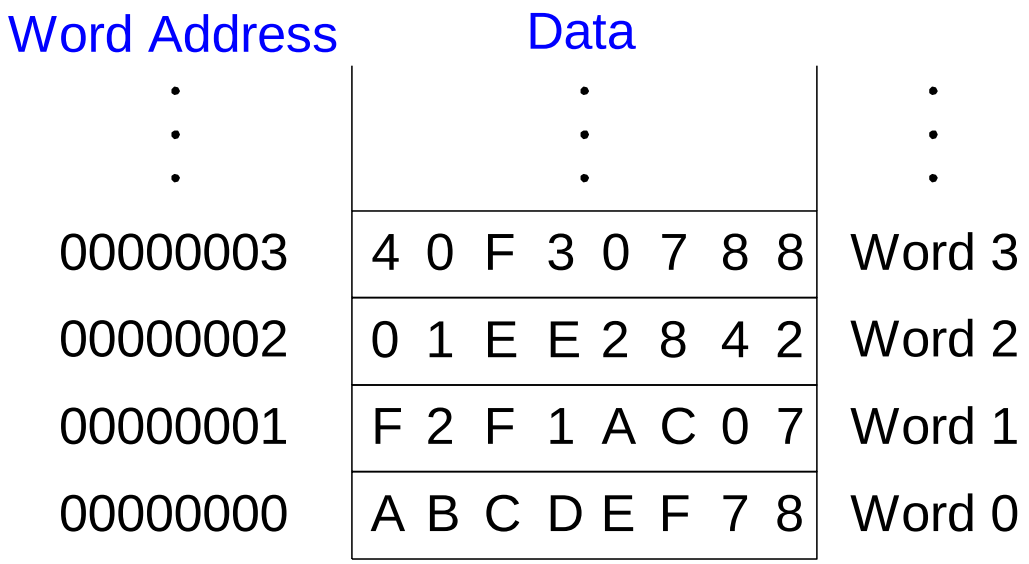


Relembrando: Matriz de Memória 4x3



Memória Endereçável por Palavra

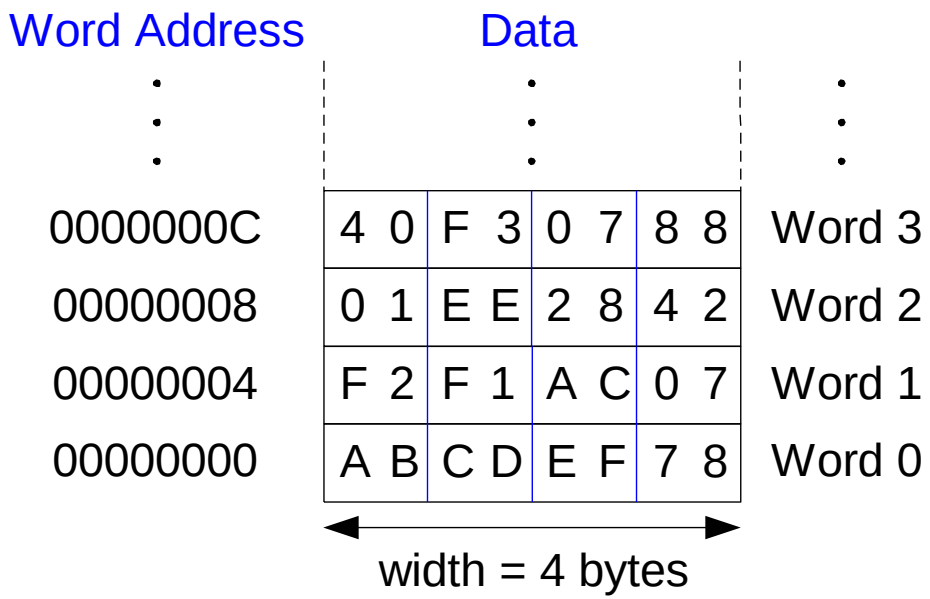
- Numa memória endereçável por palavra, cada palavra tem um endereço único e endereços consecutivos são espaçados de 1 em 1. Abaixo uma ilustração, onde cada palavras possui 32 bits.



Nota: MIPS usa memória **endereçável por bytes**, que discutiremos a seguir.

Memória Endereçável por Byte

- No MIPS, cada byte da memória tem um endereço único de 32 bits.
  - Número de localizações = 2^{32}
  - Capacidade máxima é de 2^{32} bytes = 4GB
- Há instruções para leitura/escrita de **palavras** e para leitura/escrita de **bytes específicos**.
- O endereço de uma palavra é o endereço de seu **primeiro byte**. Portanto, endereços de palavras são espaçados de 4 em 4.



Endereço de Palavras em Memórias Endereçáveis por Byte

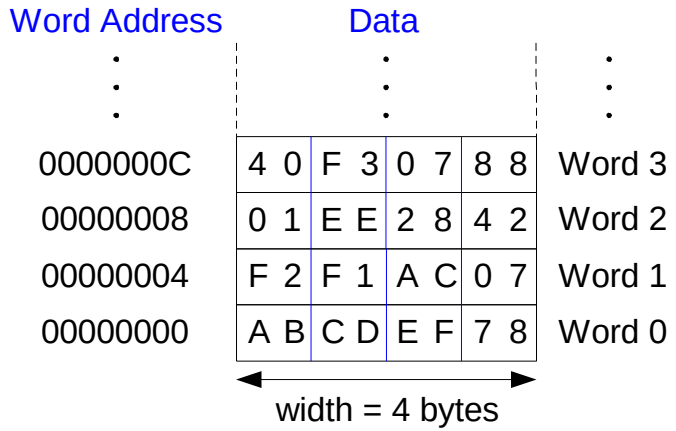
- O endereço de uma palavra de memória é obtido multiplicado o número da palavra pelo número de bytes que a compõe.
- No MIPS, cada palavra tem 4 bytes, portanto o endereços de palavras são calculados multiplicando o número da palavra por 4.
- Por exemplo, o endereço da palavra 0 é 0x4=0, o endereço da palavra 1 é 1x4=4, o endereço da palavra 2 é 2x4=8, e assim sucessivamente.
- Isso é chamado de **alinhamento de endereços**.

Leitura de uma Palavra da Memória

- `lw` (***load word***) é a instrução MIPS usada para ler uma **palavra da memória** para um **registrador** do Register File.
- **Formato geral:** **`lw rt, imm(rs)`**
  - **Mnemônico:** `lw` (*load word*) indica leitura de um palavra da memória para um registrador do Register File.
  - **Cálculo do Endereço:** `rs` (registrador base) + `imm` (offset de 16 bits estendido com sinal para 32 bits).
  - **Resultado:** `rt = Mem[rs+imm]`.
- Qualquer registrador do Register File pode ser usado como base do endereço. O valor final **deve ser alinhado** (múltiplo de 4).

Exemplo

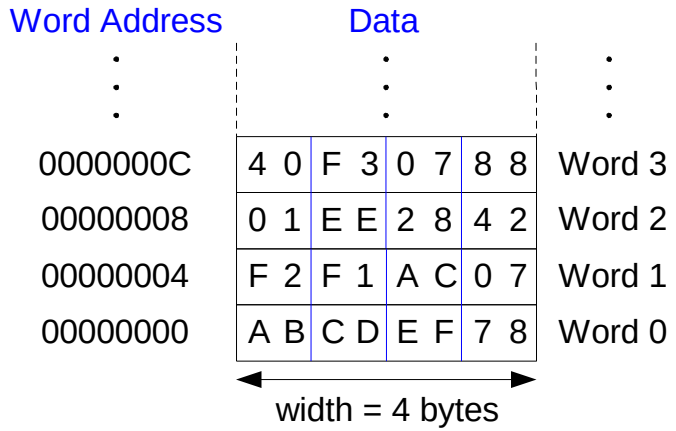
- Com a instrução `lw` leia a palavra "word 1" para o registrador `$s3`.



Nota: no MIPS a memória é endereçada por bytes, e os endereços das palavras são alinhados (múltiplos de 4).

Exemplo

- Com a instrução `lw` leia a palavra "word 1" para o registrador `$s3`.



Nota: no MIPS a memória é endereçada por bytes, e os endereços das palavras são alinhados (múltiplos de 4).

- Word 1 está no endereço 4, calculado como base + *offset* = $0 + 4.

Código Assemby

```
lw $s3, 4($0)   # lê a palavra 1 para $s3
```

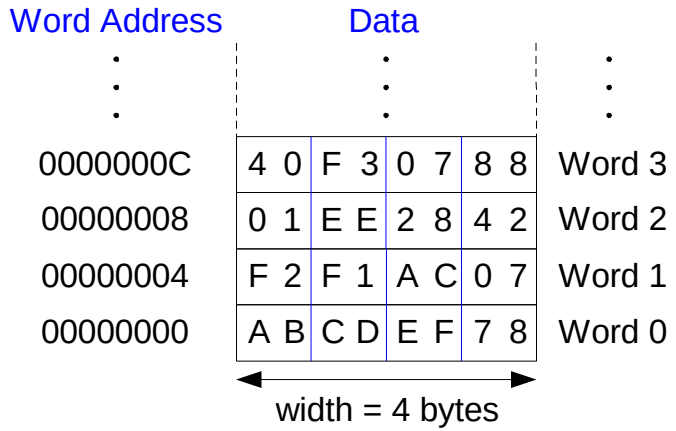
- Ao final da leitura `$s3` terá o valor **`0xF2F1AC07`**.

Gravação de uma Palavra na Memória

- `sw` (***store word***) é a instrução MIPS usada para gravar o conteúdo de um registrador do Register File em uma **palavra da memória**.
- **Formato geral:** **`sw rt, imm(rs)`**
  - **Mnemônico:** `sw` (*store word*) indica gravação do conteúdo de um registrador do Register File para uma palavra de memória.
  - **Cálculo do Endereço:** `rs` (registrador base) + `imm` (offset de 16 bits estendido com sinal para 32 bits).
  - **Resultado:** `Mem[rs+imm]` ← `rt`.
- Qualquer registrador do Register File pode ser usado como base do endereço. O endereço final **deve ser alinhado** (múltiplo de 4).

Exemplo

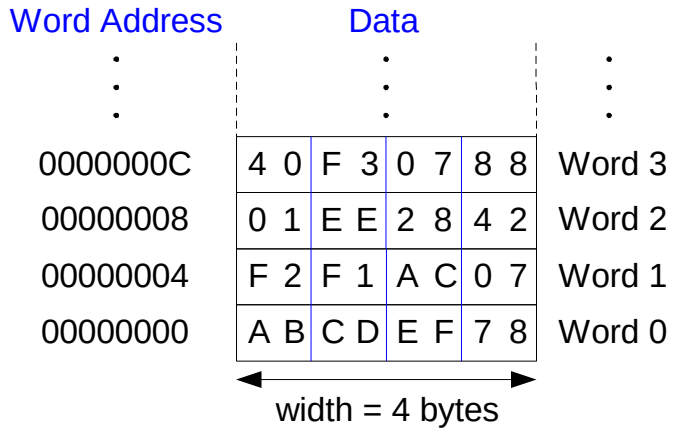
- Gravar na memória, no endereço de Word 2, a palavra contida em `$t4`.



Nota: No MIPS a memória é endereçada por bytes. Portanto, para essa instrução, o offset deve ser multiplicado por 4.

Exemplo

- Gravar na memória, no endereço de Word 2, a palavra contida em `$t4`.



Nota: No MIPS a memória é endereçada por bytes. Portanto, para essa instrução, o offset deve ser multiplicado por 4.

- Word 2 está no endereço 2x4 = 8, calculado como base + *offset* = $0 + 8.

Código Assembly

```
sw $t4, 8($0)
```

- Offsets podem ser escritos em decimal (default) ou hexadecimal.

MIPS
Instruções de Transferência de Dados

load/store de Bytes

Lendo um Byte da Memória

- `lb` (***load byte***) carrega um byte da memória para o byte menos significativo de um registrador do Register File. Os outros 3 bytes do registrador são carregados com o valor do bit mais significativo do byte transferido (extensão com sinal.
- **Formato geral: `lb rt, imm(rs)`**
  - **Mnemônico:** `lb` (*load byte*, leitura de byte).
  - **Cálculo do Endereço:** `rs` (registrador base) + `imm` (offset de 16 bits estendido para 32 bits com sinal).
  - **Resultado:** `rt`, recebe em seu byte menos significativo ($rt_{0:7}$) o byte lido da memória ($rt_{[0:7]}$ = `Mem[rs+offset]`). O restante do registrador é preenchido com o bit de sinal do byte lido.

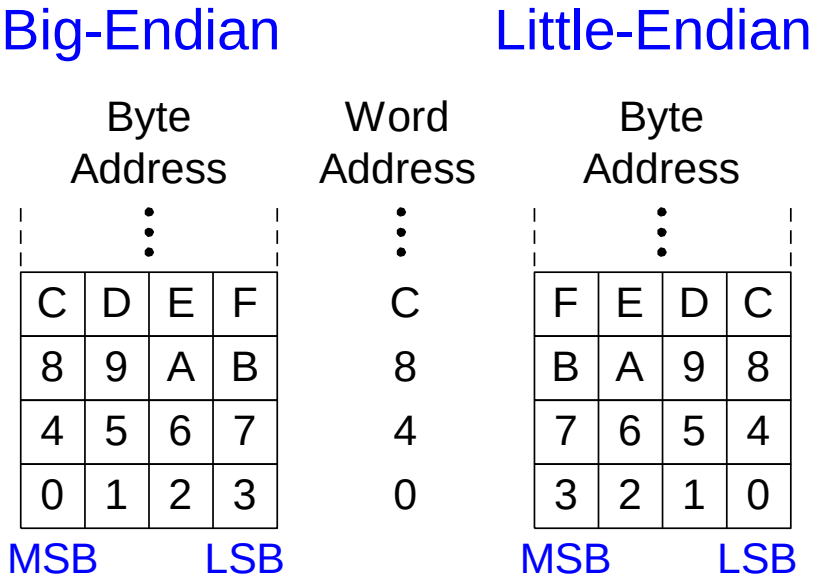
Gravando um Byte na Memória

- `sb` (***load byte***) grava o byte menos significativo de um registrador do Register File em um byte da memória.

- **Formato geral: `sb rt, offset(rs)`**
  - **Mnemônico:** `sb` (*store byte*) indica gravação do byte menos significativo de rt para a memória.
  - **Cálculo do Endereço:** `rs` (registrador base) + `imm` (offset de 16 bits estendido para 32 bits com sinal).
  - **Resultado:** O byte menos significativo do registrador é armazenado no byte endereçado da memória (`Mem[rs+offset]= rt`$_{[0:7]}$).

Ordenação de bytes: Big-Endian & Little-Endian

- Essa nomenclatura se refere à numeração dos bytes de uma palavra.
  - **Little-endian:** A numeração começa no byte menos significativo.
  - **Big-endian:** A numeração começa no byte mais significativo.
  - O **endereço da palavra** é o mesmo não importa se big ou little-endian.



Exercício

- Suponha `$t0` inicialmente contenha a palavra `0x23456789`. Depois que o código abaixo executar em um sistema **big-endian**, qual o valor em `$s0`? E em um sistema **little-endian**?

- Código Assbembly:

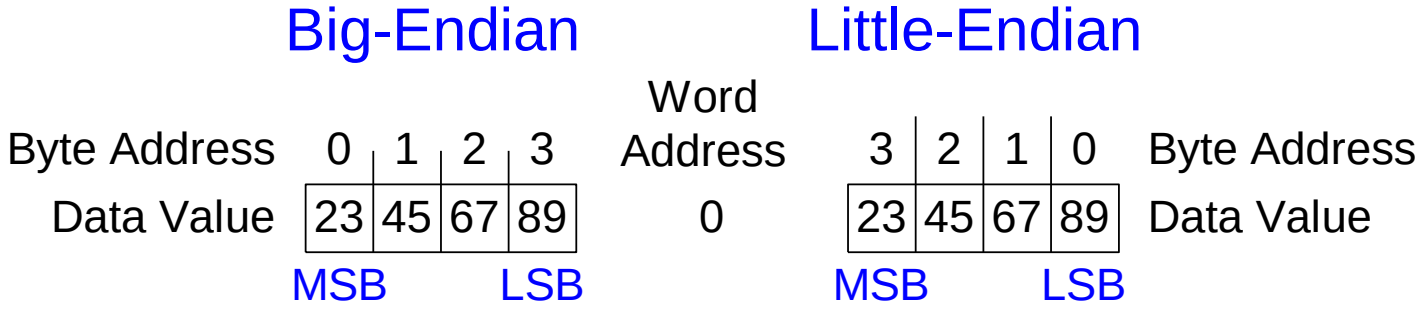
```
1. sw $t0, 0($0)
2. lb $s0, 1($0)
```

Exercício

- Suponha `$t0` inicialmente contenha a palavra `0x23456789`. Depois que o código abaixo executar em um sistema **big-endian**, qual o valor em `$s0`? E em um sistema **little-endian**?

- Código Assbembly:

```
1. sw $t0, 0($0)
2. lb $s0, 1($0)
```



- **Big-endian:** 0x00000045
- **Little-endian:** 0x00000067

Testando Instruções com o Simulador MARS

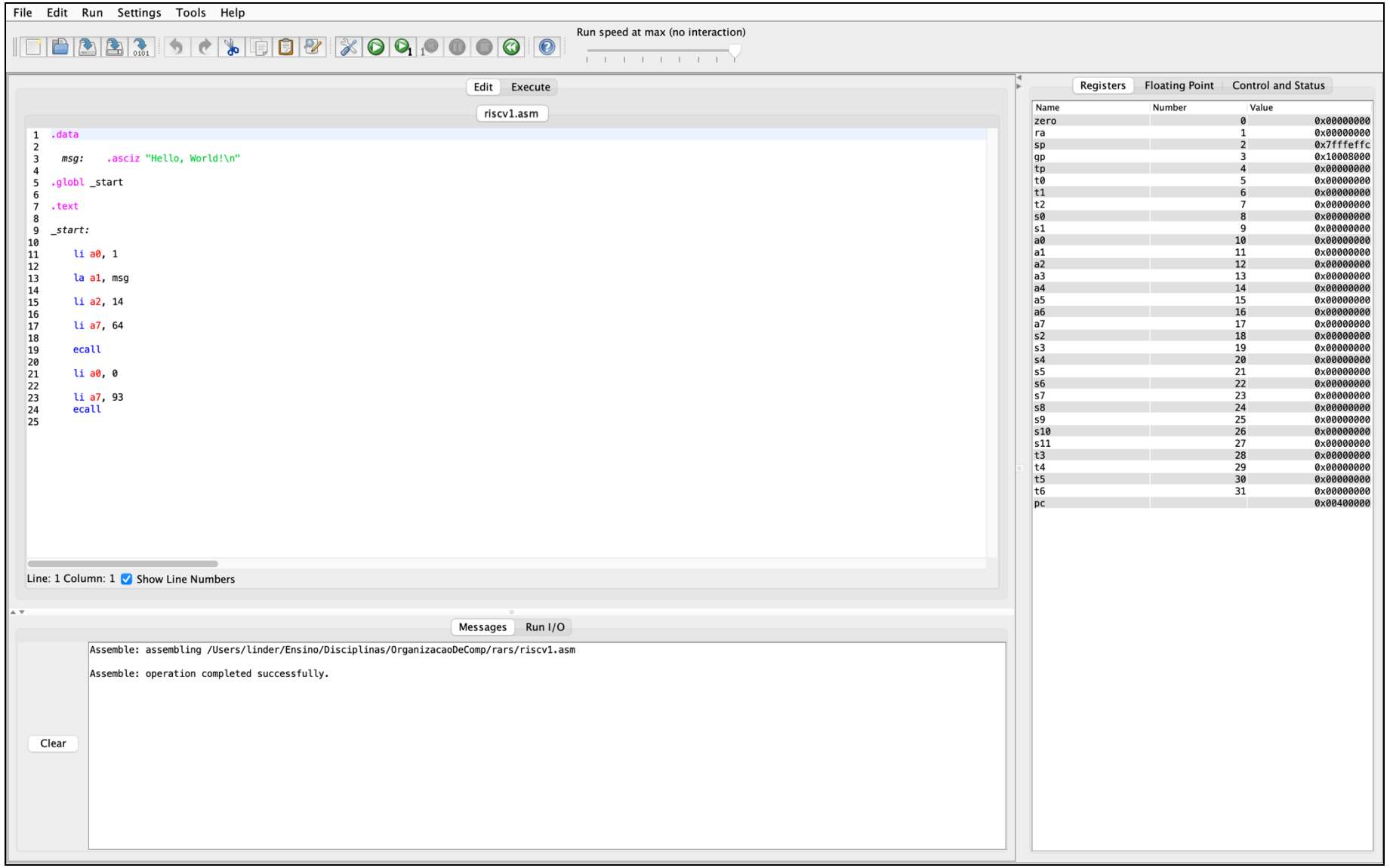
Simulação com MARS

- O MARS está disponível em https://dpetersanderson.github.io.
- Com a máquina virtual java instalada, digite o seguinte comando no terminal para executá-lo:

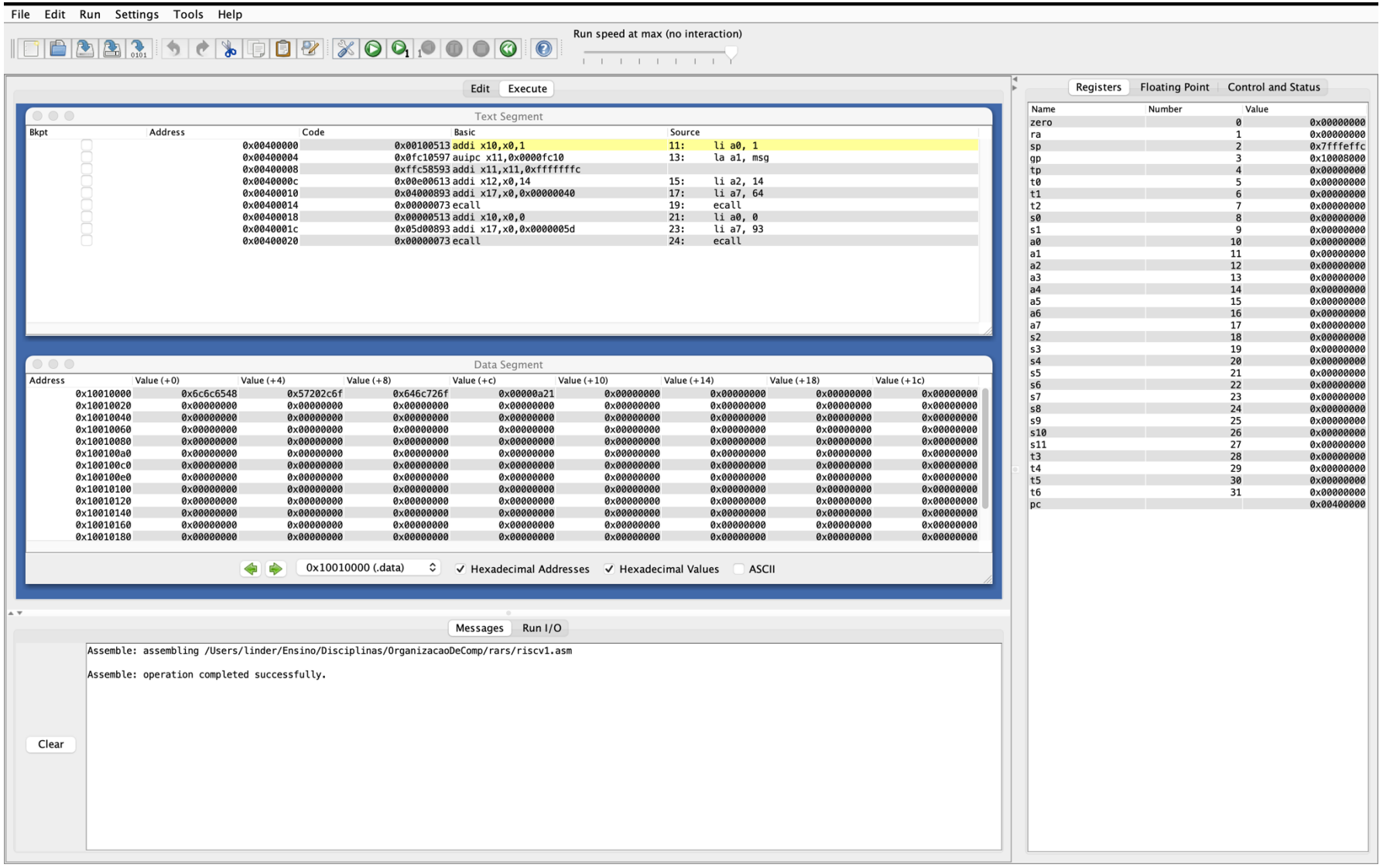
```
$java -jar mars4_5.jar
```

- Os programas assembly MIPS são escritos (ou copiados/colados) na aba `Edit` e depois simulados e executados na aba `Execute`.
- Os registradores e o conteúdo da memória podem ser visualizados enquanto o programa é executado passo a passo.
- Útil para entender/analisar a lógica de execução de programas escritos em assembly MIPS.

MARS: Aba de Edição



MARS: Aba de Execução



Linguagem de Máquina

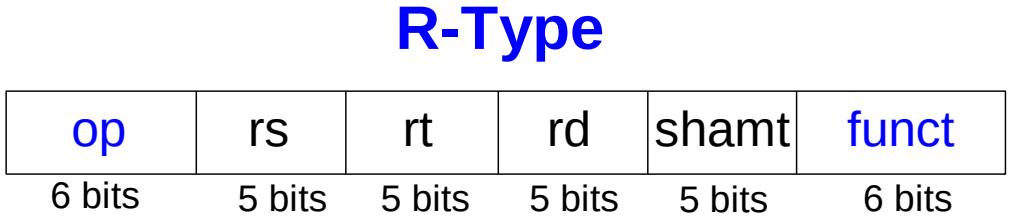
Principais Formatos de Instruções

Linguagem de Máquina

- **Representação binária** (1's e 0's) das instruções em um formato compreendido pelo processador.

- No MIPS, todas as instruções são codificadas com 32 bits e existem 3 formatos principais:
  - **Tipo-R:** 3 registradores de 32 bits como operandos.
  - **Tipo-I:** 2 registradores de 32 bits e um imediato de 16 bits como operandos.
  - **Tipo-J:** 1 imediato de 26 bits como operando.

Instruções Tipo-R



- 3 registradores como operandos, cada um identificado com 5 bits:
  - `rs`, `rt`: registradores fonte (source registers).
  - `rd`: registrador de destino.
- Outros campos:
  - op: opcode ou código de operação (sempre 0 para instruções Tipo-R).
  - funct: define a operação que o processador deve realizar.
  - shamt (shift amount): para operações de deslocamento, define o quanto deslocar. Para outras operações é sempre 0.

Exercício

Converta as instruções assembly Tipo-R abaixo para código de máquina

Assembly MIPS

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Exercício

Assembly MIPS

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Valores do Campos

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Código de Máquina

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note a diferença de ordem dos registradores no código em assembly!

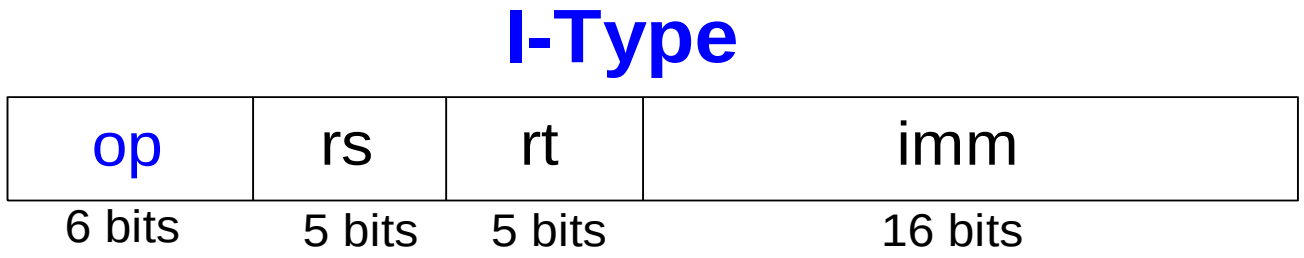
Exercício

- Traduza a instrução MIPS abaixo, apresentada em linguagem assembly, para linguagem de máquina.

```
add $t0, $s4, $s5
```

Nome simbólico	Número (em decimal)
$0	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$k0-$k1	26-27
$gp	28
$sp	29
$fp	30
$ra	31

Instruções Tipo-I



- 2 registradores de 32 bits e 1 imediato de 16 bits como operandos:
  - `rs`: registrador fonte (source register).
  - `rt`: registrador de destino.
  - `imm`: imediato (constante de 16 bits).
- Outros campos:
  - op: determina sozinho a operação. Simplicidade favorece a regularidade: todas as instruções possuem o campo opcode.

Exercício

Converta as instruções assembly Tipo-I abaixo para código de máquina

Assembly MIPS

```
addi $s0, $s1, 5

addi $t0, $s3, -12

lw   $t2, 32($0)

sw   $s1,  4($t1)
```

Exercício

Assembly MIPS

```
addi $s0, $s1, 5

addi $t0, $s3, -12

lw   $t2, 32($0)

sw   $s1,  4($t1)
```

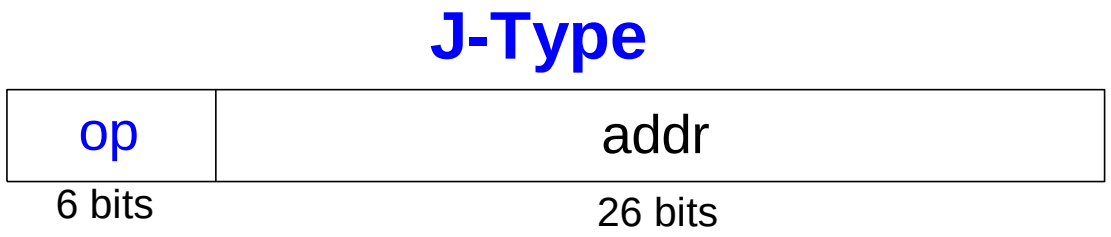
Valores do Campos

op	rs	rt	imm
8	17	16	5
8	19	8	–12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Código de Máquina

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Formato das Instruções Tipo-J



- O formato Tipo-J é usado para codificar a instrução de desvio incondicional `j` (*jump*), que veremos em breve.
- **Formato**: `j addr`
  - `addr`: imediato (constante) de 26 bits, que especifica o endereço de desvio.
  - op: opcode, ou código de operação, que determina a operação a ser realizada.

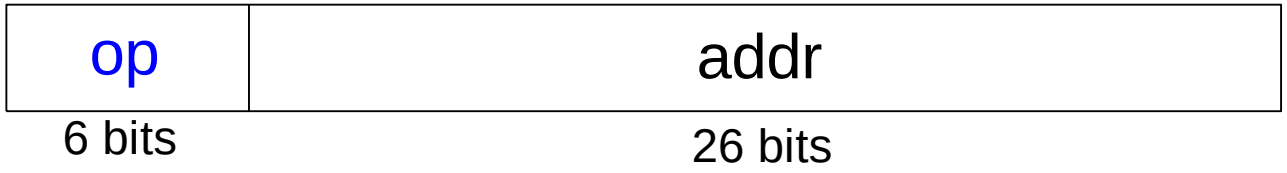
- Discussões adicionais sobre a instrução `j` serão apresentados em breve.

Sumário dos Principais Formatos de Instrução de Máquina do MIPS

R-Type

I-Type

J-Type

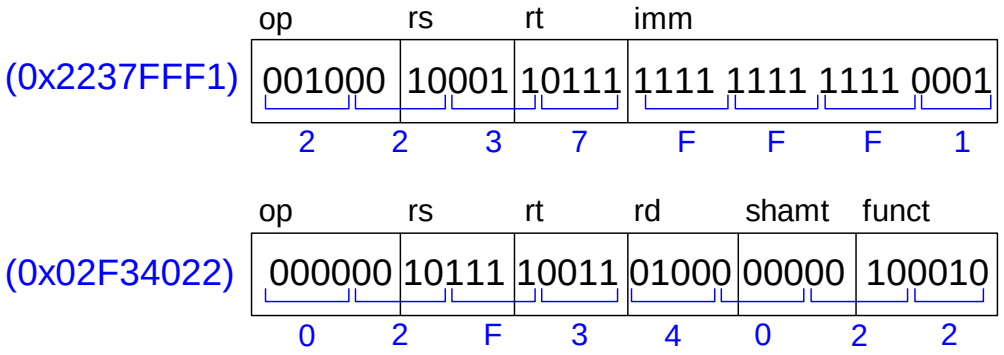


Como Interpretar Código de Máquina?

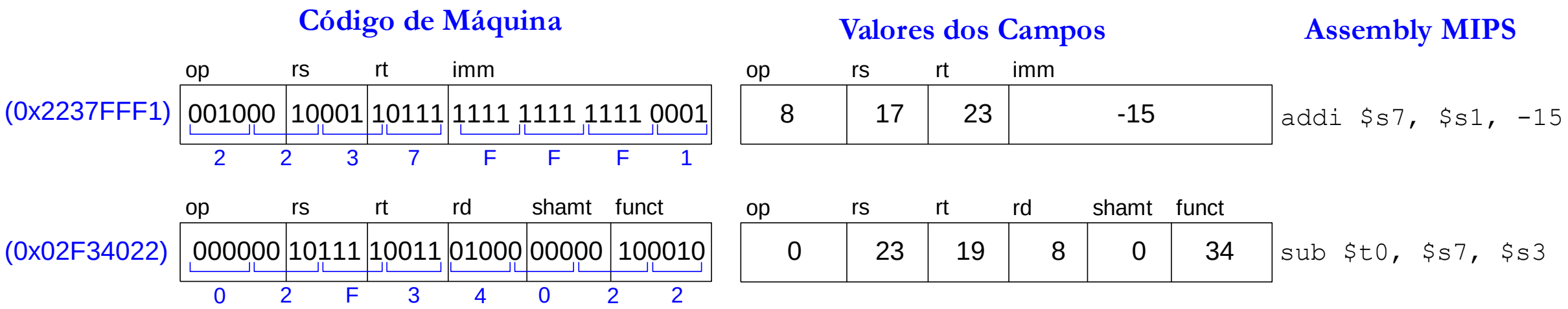
1. Comece com o opcode, tal como faz o processador. Ele diz como interpretar o restante da instrução.
2. Se o opcode contém apenas 0's
   - Instrução Tipo-R.
   - Os bits do campo `funct` ditarão qual é a operação.
3. Caso contrário (Tipo-I ou Tipo-J):
   - opcode por si só dita qual é a operação

Interprete as Instruções Abaixo

Código de Máquina



Interprete as Instruções Abaixo



MIPS
Instruções Lógicas Tipo-R

`and, or, xor e nor`

Instruções Lógicas Tipo-R

- **Formato geral em assembly**: **`op rd, rs, rt`**
  - **`op`**: `mnemônico` que pode ser `and`, `or`, `xor` ou `nor`.
  - **`rs`** e **`rt`**: registradores fonte.
  - **`rd`**: registrador de destino (no qual o resultado é escrito).

- **Resultado**: essas instruções realizam as operações lógicas bit-a-bit `and`, `or`, `xor` ou `nor`, com o conteúdo dos registradores **`rs`** e **`rt`**.

- O slide a seguir apresenta algumas aplicações para essas instruções.

Instruções Lógicas Tipo-R

- **`and`**: operação lógica de conjunção bit a bit. Útil para **mascaramento** de bits. Por exemplo, 0x000000FF é uma máscara para extrair o byte menos significativo de uma palavra usando a operação `and`: 0xF234012F AND 0x000000FF = 0x0000002F.
- **`or`**: operação lógica de disjunção bit a bit. Útil para **combinar** campos de bits. Por exemplo, 0xF2340000 OR 0x000012BC = 0xF23412BC.
- **`xor`**: Útil para verificar se dois valores são **iguais**. Por exemplo, 0xFFFF0000 XOR 0xFFFF0000 = 0x0000000.
- **`nor`**: Útil para **inverter** bits. Por exemplo, A NOR $0 = NOT A. Portanto, a operação `not` é desnecessária.

Instruções Lógicas Tipo-R

- Exemplo:

Registradores Fonte

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Código Assembly

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Resultado

$s3								
$s4								
$s5								
$s6								

Instruções Lógicas Tipo-R

- Exemplo:

Código Assembly

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Registradores Fonte

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Resultado

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

MIPS
Instruções Lógicas Tipo-I

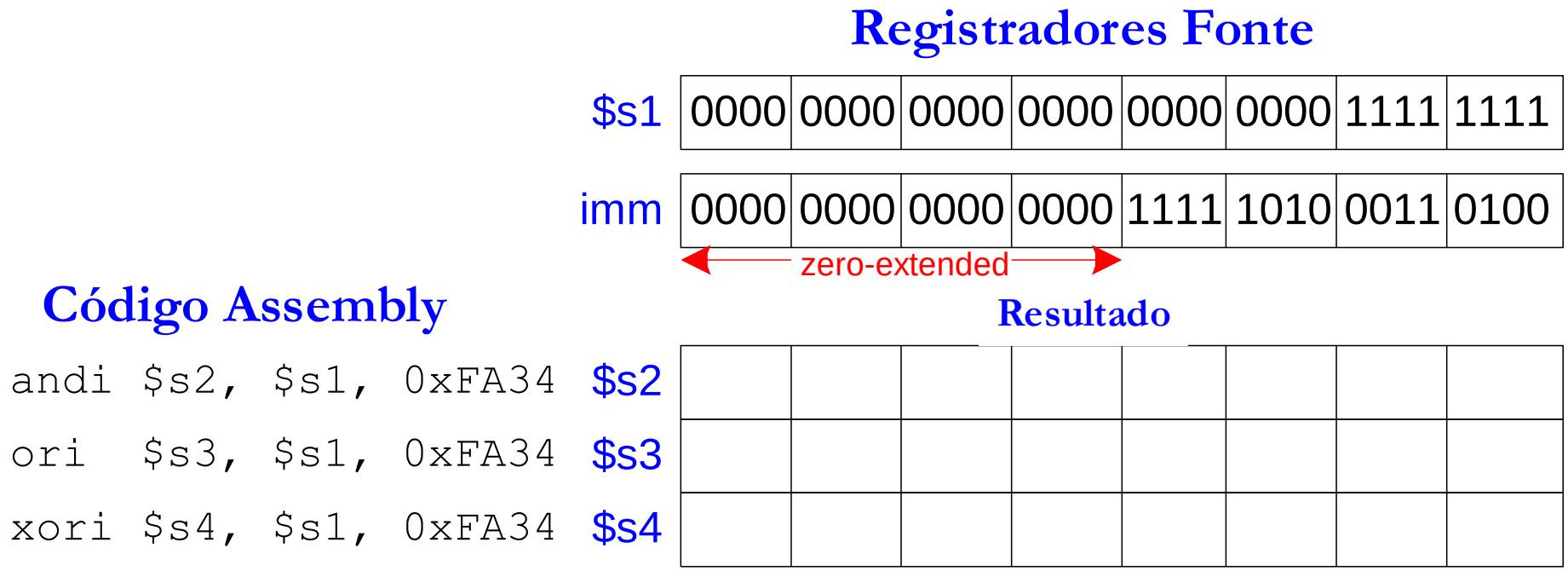
`andi, ori e xori`

Instruções Lógicas Tipo-I

- A ISA do MIPS disponibiliza três instruções lógicas bit-a-bit Tipo-I, isto é, que operam com imediatos: `andi`, `ori` e `xori`.
- **Formato geral em Assembly**: **`op rt, rs, imm`**
  - **`op`**: `mnemônico` que pode ser `andi`, `ori` ou `xori`.
  - **`rs`**: registrador fonte.
  - **`imm`**: imedito de 16 bits estendido com zeros para 32 bits (zero-extension).
  - **`rt`**: registrador de destino.
  - **Resultado**: `rt = rs | zero-extended[imm]`

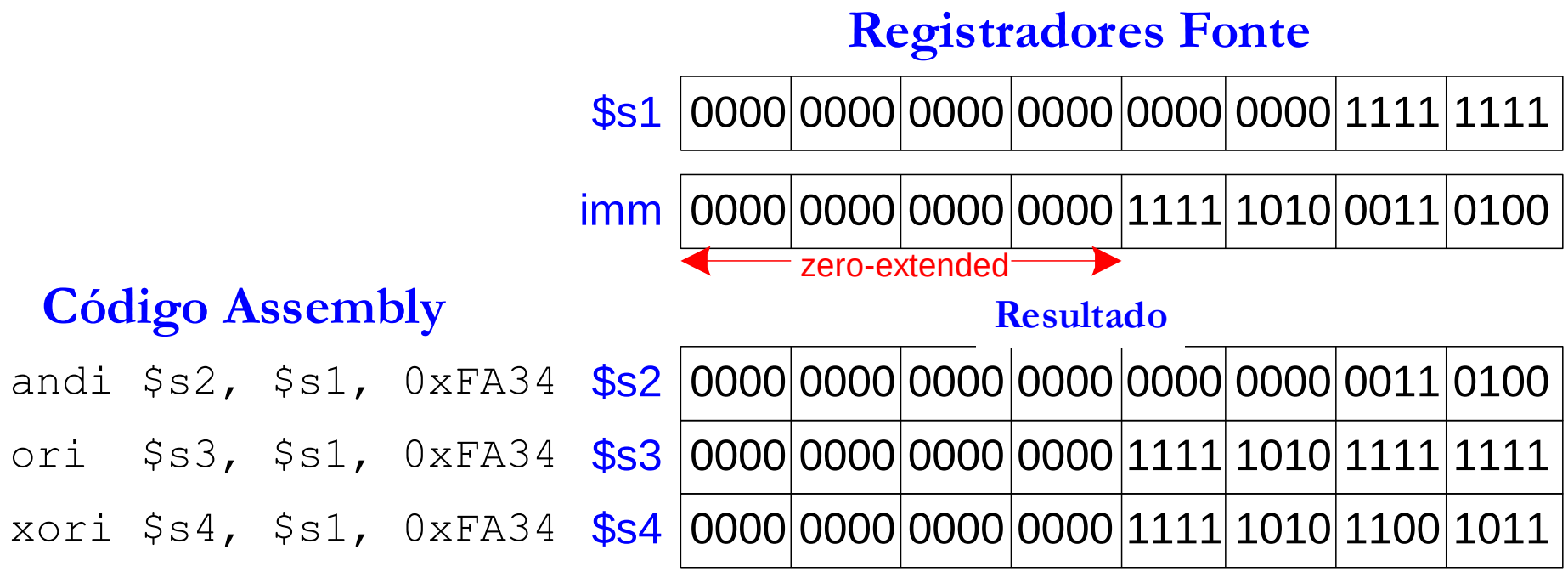
Instruções Lógicas Tipo-I

Exemplo:



Instruções Lógicas Tipo-I

Exemplo:



MIPS
Instruções de Deslocamento Tipo-R

`sll`, `srl`, `sra`, `sllv`, `srlv` e `srav`

Instruções Tipo-R de Deslocamento (shift)

- Usadas para deslocar em até 31 posições os bits de uma palavra contida em um registrador do Register File. Cada deslocamento à esquerda multiplica o número por 2 e cada deslocamento à direita divide o número por 2.
- `sll` (shift left logical); `srl` (shift right logical) e `sra` (shift right arithmetic).
- **Formato geral em Assembly**: **`op rd, rt, shamt`**
  - **`op`**: `mnemônico` pode ser `sll`, `srl` ou `sra`.
  - **`rs`**: não usado, deve ficar sempre em 0.
  - **`rt`**: registrador fonte.
  - **`rd`**: registrador de destino.
  - **`shamt`**: número de deslocamentos (0 a 31) especificado com 5 bits.

Instruções Tipo-R de Deslocamento (shift)

- `sll:` shift left logical – deslocamento lógico à esquerda.
  - `sll $t0, $t1, 5` **`#$t0 ← $t1 << 5`**
    - Preenche os bits menos significativos com zeros.

- `srl:` shift right logical – deslocamento lógico à direita.
  - **Exemplo:** `srl $t0, $t1, 5` **`#$t0 ← $t1 >> 5`**
    - Preenche os bits mais significativos com zeros.

- `sra:` shift right arithmetic– deslocamento aritmético à direita.
  - **Exemplo:** `sra $t0, $t1, 5` **`#$t0 ← $t1 >>> 5`**
    - Preenche os bits mais significativos com o valor do bit de sinal.

Instruções de Deslocamento (shift)

Código Assembly

Valore dos Campos

Código Assembly	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 4`	0	0	17	8	4	0
`srl $s2, $s1, 4`	0	0	17	18	4	2
`sra $s3, $s1, 4`	0	0	17	19	4	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Código de Máquina

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00100	000000	(0x00114100)
000000	00000	10001	10010	00100	000010	(0x00119102)
000000	00000	10001	10011	00100	000011	(0x00119903)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Registrador Fonte

$s1	1111	0011	0000	0000	0000	0010	1010	1000
shamt								00100

Resultado?

Instruções de Deslocamento (shift)

Código Assembly

Valore dos Campos

Código Assembly	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 4`	0	0	17	8	4	0
`srl $s2, $s1, 4`	0	0	17	18	4	2
`sra $s3, $s1, 4`	0	0	17	19	4	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Código de Máquina

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00100	000000	(0x00114100)
000000	00000	10001	10010	00100	000010	(0x00119102)
000000	00000	10001	10011	00100	000011	(0x00119903)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Registrador Fonte

$s1	1111	0011	0000	0000	0000	0010	1010	1000
shamt								00100

Resultado?

$t0	0011	0000	0000	0000	0010	1010	1000	0000
$s2	0000	1111	0011	0000	0000	0000	0010	1010
$s3	1111	1111	0011	0000	0000	0000	0010	1010

Instruções Tipo-R de Deslocamento

- As instruções `sllv`, `srlv` e `srav` são variações das instruções `sll`, `srl` e `sra`, onde o deslocamento é definido por um registrador em vez de um imediato.

- **Formato em assembly**: **`op rd, rt, rs`**.
  - **`op:`** `Mnemônico` que pode ser `sllv`, `srlv` ou `srav`.
  - **`rt`**: registrador com o valor a ser deslocado.
  - **`rs:`** quantidade de deslocamentos especificado nos 5 bits menos significativos.
  - **`rd`**: registrador de destino.
  - O campo **`shamt`**, nessas instruções, deve ficar zerado.

Instruções de Deslocamento (shift)

Código Assembly

Valore dos Campos

Código Assembly	op	rs	rt	rd	shamt	funct
`sllv $s3, $s1, $s2`	0	18	17	19	0	4
`srlv $s4, $s1, $s2`	0	18	17	20	0	6
`srav $s5, $s1, $s2`	0	18	17	21	0	7
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Código de Máquina

op	rs	rt	rd	shamt	funct	
000000	10010	10001	10011	00000	000100	(0x02519804)
000000	10010	10001	10100	00000	000110	(0x0251A006)
000000	10010	10001	10101	00000	000111	(0x0251A807)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Registradores Fonte

$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	0000	**1000**

Código Assembly

```
sllv $s3, $s1, $s2
srlv $s4, $s1, $s2
srav $s5, $s1, $s2
```

Resultado?

Instruções de Deslocamento (shift)

Código Assembly

Valore dos Campos

Código Assembly	op	rs	rt	rd	shamt	funct
`sllv $s3, $s1, $s2`	0	18	17	19	0	4
`srlv $s4, $s1, $s2`	0	18	17	20	0	6
`srav $s5, $s1, $s2`	0	18	17	21	0	7
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Código de Máquina

op	rs	rt	rd	shamt	funct	
000000	10010	10001	10011	00000	000100	(0x02519804)
000000	10010	10001	10100	00000	000110	(0x0251A006)
000000	10010	10001	10101	00000	000111	(0x0251A807)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Registradores Fonte

$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	0000	**1000**

Código Assembly

```
sllv $s3, $s1, $s2
srlv $s4, $s1, $s2
srav $s5, $s1, $s2
```

Resultado?

$s3	0000	0100	0000	0010	1010	1000	0000	0000
$s4	0000	0000	1111	0011	0000	0100	0000	0010
$s5	1111	1111	1111	0011	0000	0100	0000	0010

MIPS

Instruções Tipo-R para Multiplicação de Inteiros

`mult`, **`multu`** e **`mul`**

Multiplicação Inteira com Sinal

- A multiplicação de inteiros de 32 bits pode produzir até 64 bits. Por isso, o resultado de uma multiplicação não vai diretamente para os registradores comuns do Register File (32 bits).

- **Formato em assembly**: **`mult rs`**, **`rt`**.
  - **`mult`**: `Mnemônico` que indica multiplicação inteira com sinal.
  - **`rs`** e **`rt`**: registradores com os valores a serem multiplicados (comp. 2).
  - **`rd`**: sempre em 0.
  - **Resultado**: armazenado em dois registradores especiais de 32 bits chamados `hi` $=[rs \times rt]_{63:32}$ e `lo` $=[rs \times rt]_{31:0}$
  - As instruções `mflo rd` e `mfhi rd`, ambas tipo-R, são usadas para ler o conteúdo de `hi` e `lo` para o Register File.

Multiplicação Inteira sem Sinal

- A multiplicação de inteiros de 32 bits sem sinal pode produzir até 64 bits. Por isso, o resultado de uma multiplicação não vai diretamente para os registradores comuns do Register File.
- **Formato em assembly**: **`multu rs`**, **`rt`**.
  - **`multu`**: `Mnemônico` que indica multiplicação inteira sem sinal.
  - **`rs`** e **`rt`**: registradores com os valores a serem multiplicados (interpretados como números sem sinal).
  - **`rd`**: sempre em 0.
  - **Resultado**: armazenado em dois registradores especiais de 32 bits chamados `hi` $=[rs \times rt]_{63:32}$ e `lo` $=[rs \times rt]_{31:0}$
  - As instruções `mflo rd` e `mfhi rd`, ambas tipo-R, são usadas para ler o conteúdo de `hi` e `lo` para o Register File.

Multiplicação Inteira com Sinal

- A instrução `mul` faz a multiplicação de dois registradores de 32 bits e armazena o resultado diretamente em um registrador comum do Register File.
- **Formato em assembly**: **`mul rd, rs, rt`**.
  - **`mul`**: `Mnemônico` que indica multiplicação inteira com sinal.
  - **`rs`** e **`rt`**: registradores com os valores a serem multiplicados (comp. 2).
  - **`rd`**: sempre em 0.
  - **Resultado**: armazenado diretamente no registrador **`rd`** do Register File.

- Essa instrução deve ser usada apenas quando o programador tem certeza que o resultado da multiplicação pode ser armazenado em 32 bits, pois a ocorrência de overflow não é tratada.

MIPS

Instruções Tipo-R para Divisão Inteira

`div` e **`divu`**

Divisão Inteira com Sinal no MIPS

- A divisão inteira de números de 32 bits com sinal produz um quociente e um resto, ambos de 32 bits. A ISA do MIPS disponibiliza a instrução Tipo-R `div` para esse fim.
- **Formato em assembly**: **`div rs`**, **`rt`**.
  - **`div`**: `Mnemônico` que indica divisão inteira com sinal.
  - **`rs`** e **`rt`**: registradores com os valores a serem divididos (comp. 2).
  - **`rd`**: sempre em 0.
  - **Resultado**: O resultado é armazenado nos registradores especiais `hi` e `lo`: `hi = (rs / rt)` e `lo = (rs % rt)`
- Novamente, as instruções `mflo rd` e `mfhi rd` são usadas para ler o conteúdo de `hi` e `lo` para registradores do Register File:

Divisão Inteira sem Sinal no MIPS

- A divisão inteira de dois números de 32 bits (sem sinal) produz um quociente e um resto de 32 bits. A ISA do MIPS disponibiliza a instrução Tipo-R `divu` para esse fim.
- **Formato em assembly**: **`divu rs`**, **`rt`**.
  - **`divu`**: `Mnemônico` que indica divisão inteira sem sinal (unsigned).
  - **`rs`** e **`rt`**: registradores com os valores a serem divididos.
  - **`rd`**: sempre em 0.
  - **Resultado**: O resultado é armazenado nos registradores especiais `hi` e `lo`: `hi = (rs / rt)` e `lo = (rs % rt)`
- Novamente, as instruções `mflo rd` e `mfhi rd` são usadas para ler o conteúdo de `hi` e `lo` para registradores do Register File:

MIPS

Instruções de Desvios Condicionais e Incondicionais

`j`, `beq`, `bne`, `bltz`, `bgez`, `blez` e `bgtz`

Instruções de Desvio (Branching)

- Para executar instruções sequencialmente, a unidade de controle do MIPS incrementa o Contador de Programa, PC ← PC + 4, a cada ciclo de instrução.
- Entretanto, a execução sequencial de instruções não é suficiente para atender às necessidades da programação estruturada, a qual demanda construções para seleção, tais como **if-else** e **switch**, e repetição, tais como **for** e **while**.
- As instruções MIPS de desvio modificam o Contador de Programa (PC) para pular seções do código ou para repetir o código anterior.
- Há dois tipos de instruções de desvio:
  - **Desvio Condicional**: executam um teste e desviam somente se o teste for avaliado como **True** (verdadeiro).
  - **Desvio Incondicional**: também chamados de *jumps*, não realizam testes e sempre desviam.

Instrução Tipo-I de Desvio Condicional `beq`

- **Formato em assembly: `beq rs, rt, label`**
  - **`beq`:** `Mnemônico` que indica instrução desvio se `rs = rt`.
  - **`rs`** e **`rt`**: registradores com os valores a serem comparados.
  - **`label`**: indica o ponto de desvio no programa.
  - **Resultado**: `if ([rs] == [rt]) PC = BTA`
- O assembler computa imediato 16 bits, como sendo a distância (em número de instruções) do `PC+4` ao label.
- Em caso de desvio, o processador calcula o `BTA = PC + 4 + (imm << 2)` com base no `imm` e atualiza o PC. Note que no momento do cálculo do BTA o PC está com o endereço da instrução subsequente.

Exemplo

```
# assembly MIPS
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 beq  $s0, $s1, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0
target:                    # label não podem ser palavras reservadas
                           # e devem ser seguidos por :
 add  $s1, $s1, $s0
```

labels são traduzidos pelo assembler no valor de offset (imediato da instrução de máquina) necessário para cálculo do *Brach Target Address* (BTA).

Instrução Tipo-I de Desvio Condicional bne

- **Formato em assembly: `bne rs, rt, label`**
  - **`bne`:** Mnemônico que indica desvio caso `rs` ≠ `rt`.
  - **`rs`** e **`rt`**: registradores com os valores a serem comparados.
  - **`label`**: indica o ponto de desvio no programa.
  - **Resultado**: `if ([rs] ≠ [rt]) PC = BTA`

- O assembler computa imediato 16 bits, como sendo a distância (em número de instruções) do `PC+4` ao **`label`**.
- Em caso de desvio, o processador calcula o `BTA = PC + 4 + (imm << 2)` com base no `imm` e atualiza o PC.

Exemplo

```
# assembly MIPS
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 bne  $s0, $s1, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0

target:                    # label
 add  $s1, $s1, $s0
```

labels são traduzidos pelo assembler no valor de offset (imediato da instrução de máquina) necessário para cálculo do *Brach Target Address* (BTA).

Instrução Tipo-I de Desvio Condicional `bltz`

- **Formato em assembly:** **`bltz rs, label`**
  - **`bltz`:** `Mnemônico` que indica desvio caso `rs` <0.
  - **`rs`**: registrador com o valor a ser comparado.
  - **`rt`**: sempre 0 nessa instrução.
  - **`label`**: indica o ponto de desvio no programa.
  - **Resultado**: `if ([rs]< 0) PC = BTA`

- O assembler computa imediato 16 bits, como sendo a distância (em número de instruções) do `PC+4` ao **`label`**.
- Em caso de desvio, o processador calcula o `BTA = PC + 4 + (imm << 2)` com base no `imm` e atualiza o PC.

Exemplo

```
# assembly MIPS
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 bltz $s0, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0

target:                    # label
 add  $s1, $s1, $s0
```

labels são traduzidos pelo assembler no valor de offset (imediato da instrução de máquina) necessário para cálculo do *Brach Target Address* (BTA).

Instrução Tipo-I de Desvio Condicional `blez`

- **Formato em assembly:** **`blez rs, label`**
  - **`blez`:** `Mnemônico` que indica desvio caso `rs` ≤ 0.
  - **`rs`**: registrador com o valor a ser comparado.
  - **`rt`**: sempre 0 nessa instrução.
  - **`label`**: indica o ponto de desvio no programa.
  - **Resultado**: `if ([rs]` ≤ 0`) PC = BTA`

- O assembler computa imediato 16 bits, como sendo a distância (em número de instruções) do `PC+4` ao **`label`**.
- Em caso de desvio, o processador calcula o `BTA = PC + 4 + (imm << 2)` com base no `imm` e atualiza o PC.

Exemplo

```
# assembly MIPS
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 blez $s0, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0

target:                    # label
 add  $s1, $s1, $s0
```

labels são traduzidos pelo assembler no valor de offset (imediato da instrução de máquina) necessário para cálculo do *Brach Target Address* (BTA).

Instrução Tipo-I de Desvio Condicional `bgtz`

- **Formato em assembly**: **`bgtz rs, label`**
  - **`bgtz`**: `Mnemônico` que indica desvio caso `rs` > 0.
  - **`rs`**: registrador com o valor a ser comparado.
  - **`rt`**: sempre 0 nessa instrução.
  - **`label`**: indica o ponto de desvio no programa.
  - **Resultado**: `if ([rs]> 0) PC = BTA`

- O assembler computa imediato 16 bits, como sendo a distância (em número de instruções) do `PC+4` ao **`label`**.
- Em caso de desvio, o processador calcula o `BTA = PC + 4 + (imm << 2)` com base no `imm` e atualiza o PC.

Exemplo

```
# assembly MIPS
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 bgtz $s0, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0

target:                    # label
 add  $s1, $s1, $s0
```

labels são traduzidos pelo assembler no valor de offset (imediato da instrução de máquina) necessário para cálculo do *Brach Target Address* (BTA).

Instrução Tipo-I de Desvio Condicional bgez

- **Formato em assembly**: **`bgez rs, label`**
  - **`bgez`**: Mnemônico que indica desvio caso `rs` ≥ 0.
  - **`rs`**: registrador com o valor a ser comparado.
  - **`rt`**: sempre 0 nessa instrução.
  - **`label`**: indica o ponto de desvio no programa.
  - **Resultado**: `if ([rs]` ≥ 0`) PC = BTA`

- O assembler computa imediato 16 bits, como sendo a distância (em número de instruções) do `PC+4` ao **`label`**.
- Em caso de desvio, o processador calcula o `BTA = PC + 4 + (imm << 2)` com base no `imm` e atualiza o PC.

Exemplo

```
# assembly MIPS
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 bgez $s0, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0

target:                    # label
 add  $s1, $s1, $s0
```

labels são traduzidos pelo assembler no valor de offset (imediato da instrução de máquina) necessário para cálculo do *Brach Target Address* (BTA).

Instruções Tipo-J `j` (jump)

- O formato Tipo-J é usado para codificar a instrução de desvio incondicional `j` (*jump*).
- **Formato geral em assembly**: **`j label`**
  - **`label`**: indica o ponto de desvio no programa.
  - **`j`**: opcode que determina a realização da instrução de desvio incondicional.
  - **Resultado**: `PC = JTA` (jump target address)
- Com base no **`label`** o assembler conta o número de instruções relativo ao início do programa.
- O processador calcula o JTA = {(PC + 4)[31:28], imm× 4}e carrega no PC. Apenas 26 bits são fornecidos pela instrução **`j`** como imediato, portanto o desvio é limitado a um segmento de 256 MB do código.

Próximo Tópico

Programação