Arquitetura e Organização de Computadores

Professor: Lucas Cambuim

Aula: Conjunto de Instruções

Introdução

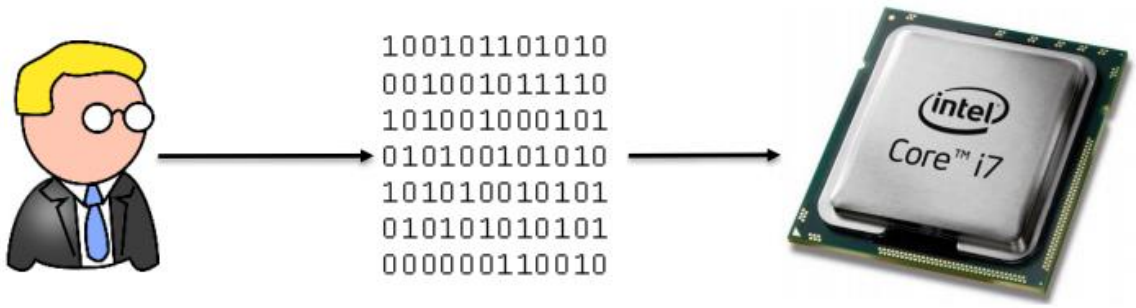
- Organização de um computador.
- Representação das instruções em linguagem de máquina.
- O processador MIPS.
- Instruções para soma e subtração.
- Registradores no MIPS.
- Instruções para transferência de dados.
- Endianness.
- Instrução para soma com constantes.
- Instruções para operações lógicas

Que Linguagem o HW entende?

- HW entende sinais elétricos
- Alfabeto da linguagem entendida por HW possui dois valores:
  - Ligado (On), Desligado (Off)
  - Ou 0 e 1 (números binários)
- **Instruções** são sequências de números binários para que o computador realize uma determinada ação

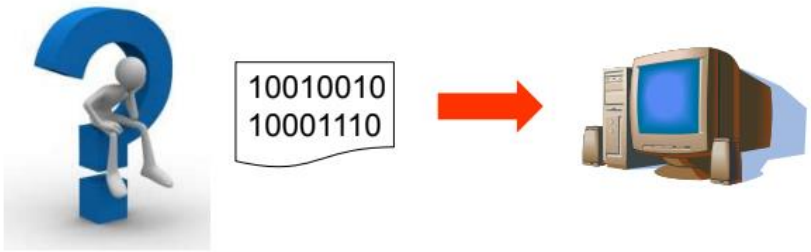
Linguagem de máquina

- Para programar um computador é necessário definir instruções baseadas numa **linguagem de máquina.**
- Linguagem de máquina = Conjunto de instruções do processador



Abstraindo a Linguagem de Máquina

- Escrever um programa em linguagem de máquina é impraticável!
- Conceitos de HW foram abstraídos para que ser humano pudesse instruir o computador
- Criação de linguagens de programação



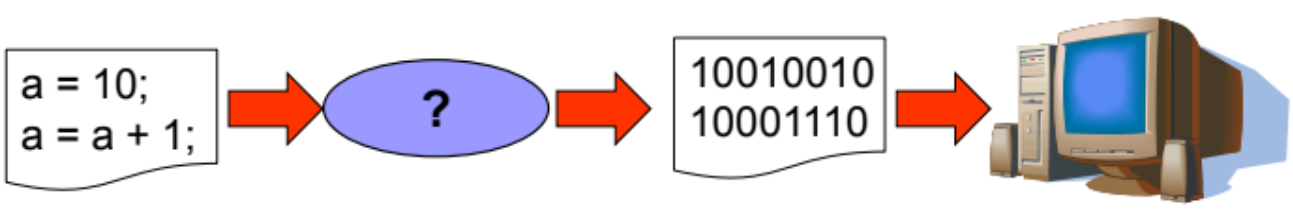
Linguagens de Programação

- Os programas têm que ser escritos em uma linguagem de programação:
  - que possa ser entendida pelo computador

Linguagens de Programação

- Os programas têm que ser escritos em uma linguagem de programação:
  - que possa ser entendida pelo computador

  - que possa ser **traduzida** para a linguagem entendida pelo computador



Níveis de Abstração de Linguagens

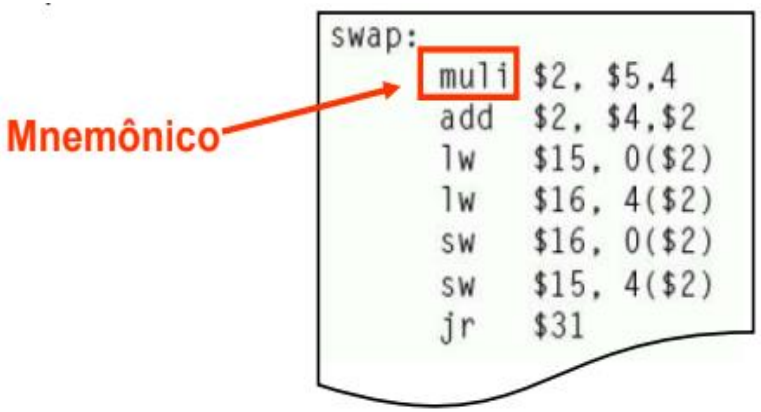
- Linguagens de programação variam de acordo com o seu nível de abstração
  - ↑ conhecimento da máquina onde programa será executado
  - ↓ nível de abstração

  - ↓ conhecimento da máquina onde programa será executado
  - ↑ nível de abstração

- Podem ser classificadas em 4 níveis:
  - Linguagem de máquina
  - Linguagem de montagem (assembly)
  - Linguagem de alto nível (Java, C, Pascal, C++, etc)
  - Linguagem de 4a geração (PL/SQL, NATURAL, MATLAB, Python etc)

Níveis de Abstração de Linguagens

- Linguagem assembly é dependente da máquina, porém utiliza palavras reservadas para codificar instruções (mnemônicos)



- Outros níveis são independentes de máquina e facilitam leitura e escrita dos programas por parte do ser humano
  - Complexidade atual de programas exigem cada vez mais o emprego destas linguagens

Como o Computador Entende um Programa?

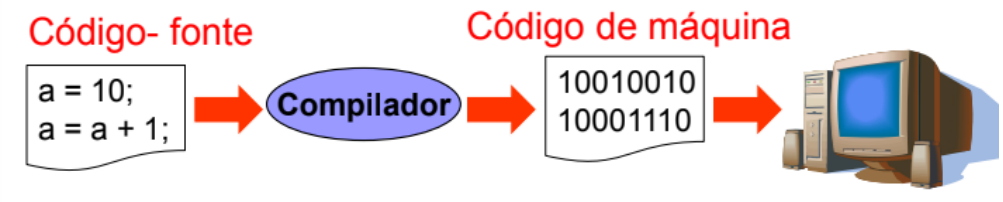
- Deve-se traduzir um programa para a linguagem de máquina

- Um compilador é um programa que traduz um programa escrito (código fonte) em uma determinada linguagem de programação para outra linguagem (linguagem destino)
  - Se a linguagem destino for a de máquina, o programa pode, depois de compilado, ser executado

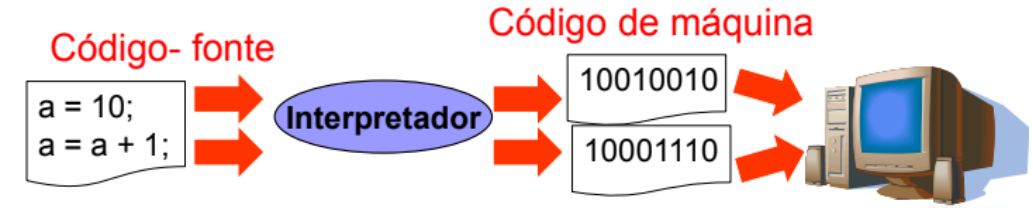
- Um interpretador é um programa que traduz instrução por instrução de um programa em linguagem de máquina e imediatamente executa a instrução (em tempo de execução)

Compilação x Interpretação

- Compilação



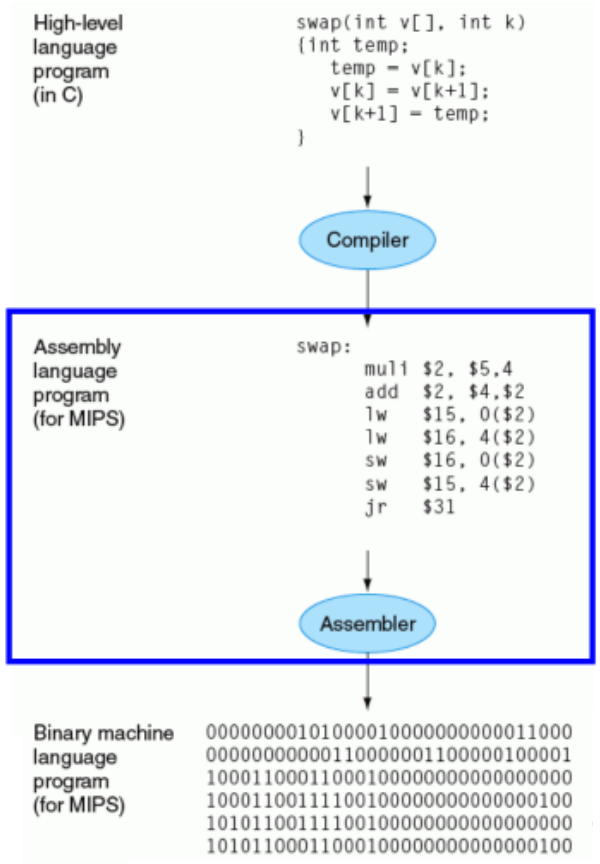
- Interpretação



Compilação x Interpretação

- Existem vários exemplos tanto de linguagens interpretadas como de linguagens compiladas
- A linguagem C é um exemplo de linguagem compilada
- Java é uma linguagem de programação que utiliza um processo híbrido de tradução
  - O compilador Java traduz o código-fonte em um formato intermediário independente de máquina chamado bytecode
- Interpretador Java específico da máquina onde irá rodar o programa então traduz os bytecodes para linguagem de máquina e executa o código

Exemplo de Compilação em 2 etapas



Maioria dos compiladores C omitem esta parte. Compilam diretamente para linguagem de máquina

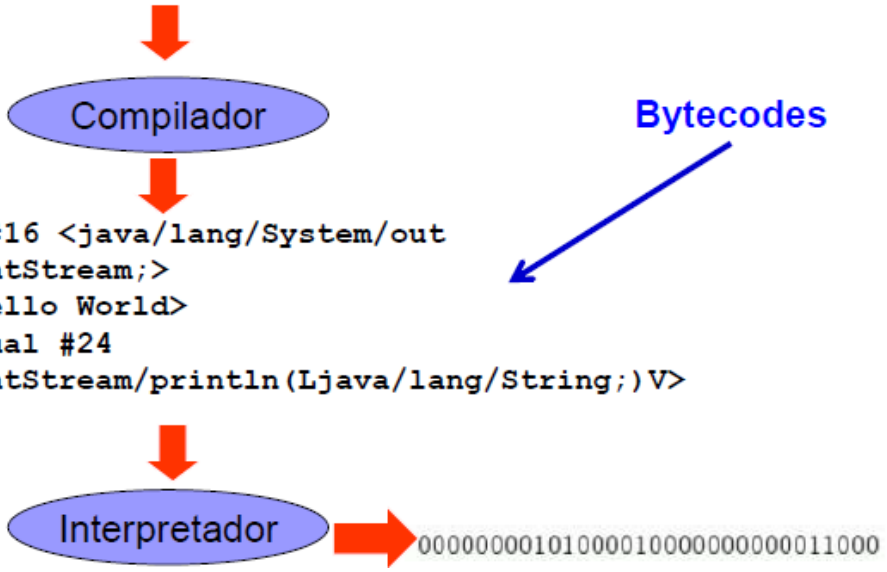
Exemplo de Compilação e Interpretação

```
public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello World");

  }

}
```

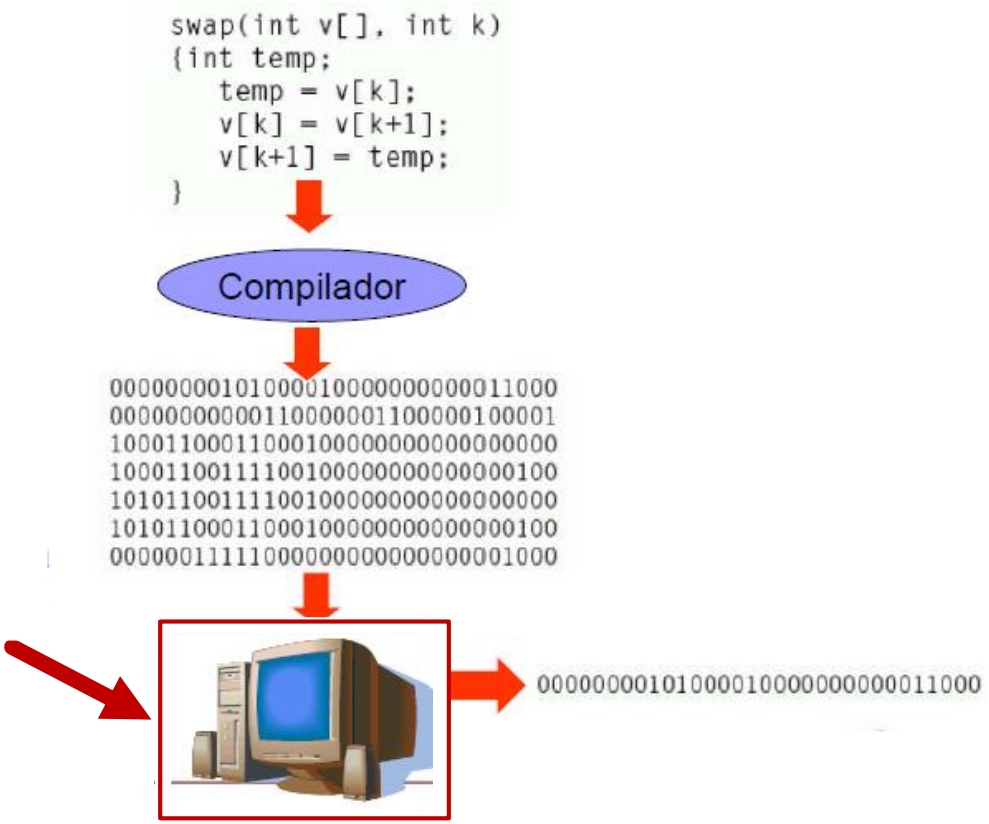


Compilação x Interpretação

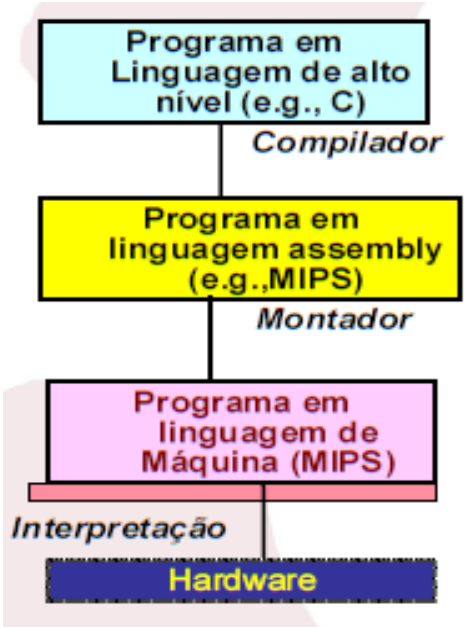
Tradutor	Vantagens	Desvantagens
Compilador	Execução mais veloz.	Várias etapas de tradução.
	Programas mais complexos, com mais funcionalidades.	Maior consumo de memória para a execução, uma vez que o programa final é maior.
	Permite otimização do código-fonte.	Processo de depuração de cada linha é mais demorado.
Interpretador	Depuração mais simples.	Execução mais lenta.
	Menor consumo de memória.	Programas mais simples.
	Resultado imediato da execução das instruções por ser passo a passo.	É necessário que a máquina possua o programa fonte.

Compilação em SW e Interpretação em HW

HW interpreta instrução a instrução



Compilação em SW e Interpretação em HW



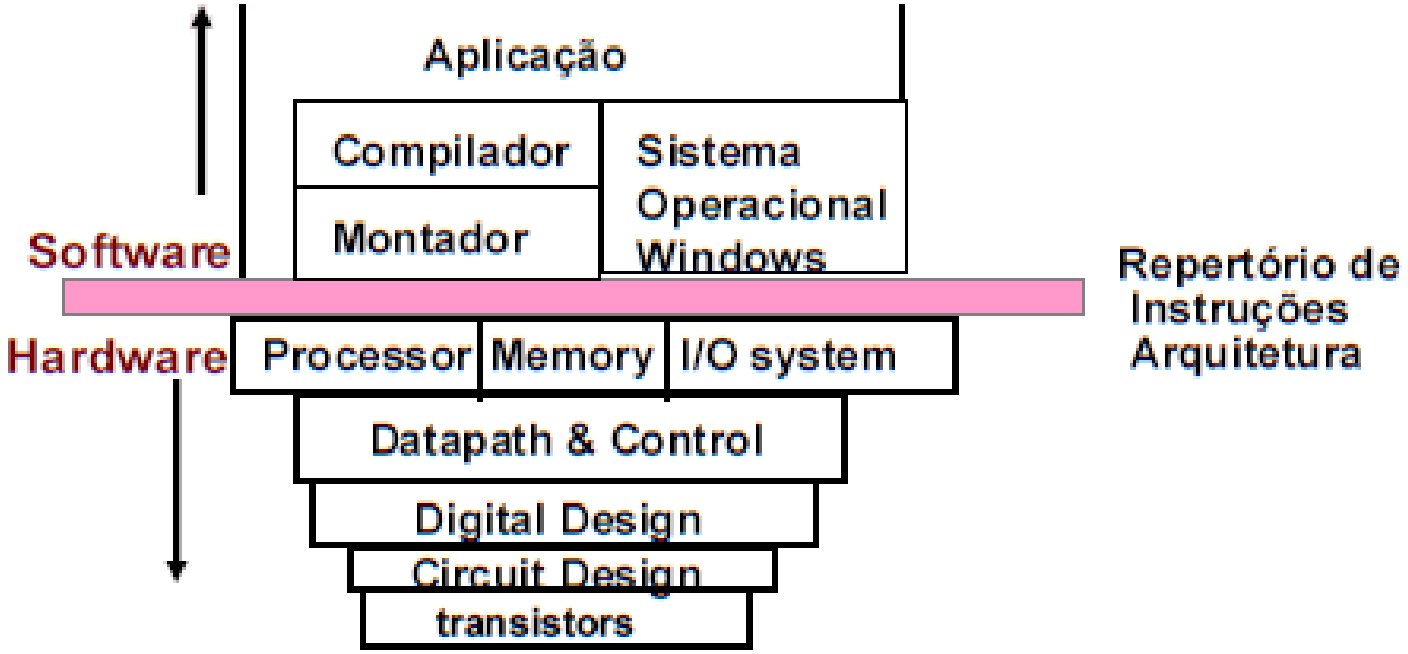

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $to,   0($2)
lw  $t1,   4($2)
sw $t1,   0($2)
sw $t0,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

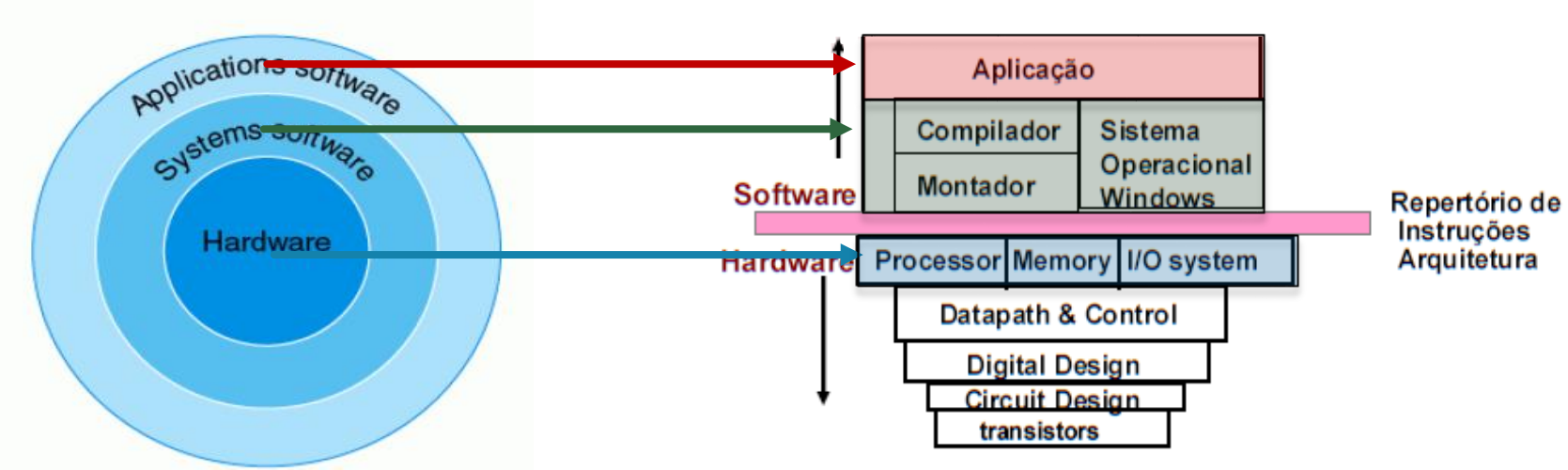
Abstrações de um Computador

- Faz-se necessário a criação de camadas de abstrações que escondam detalhes de implementação de um computador para desenvolver as aplicações atuais cada vez mais complexas



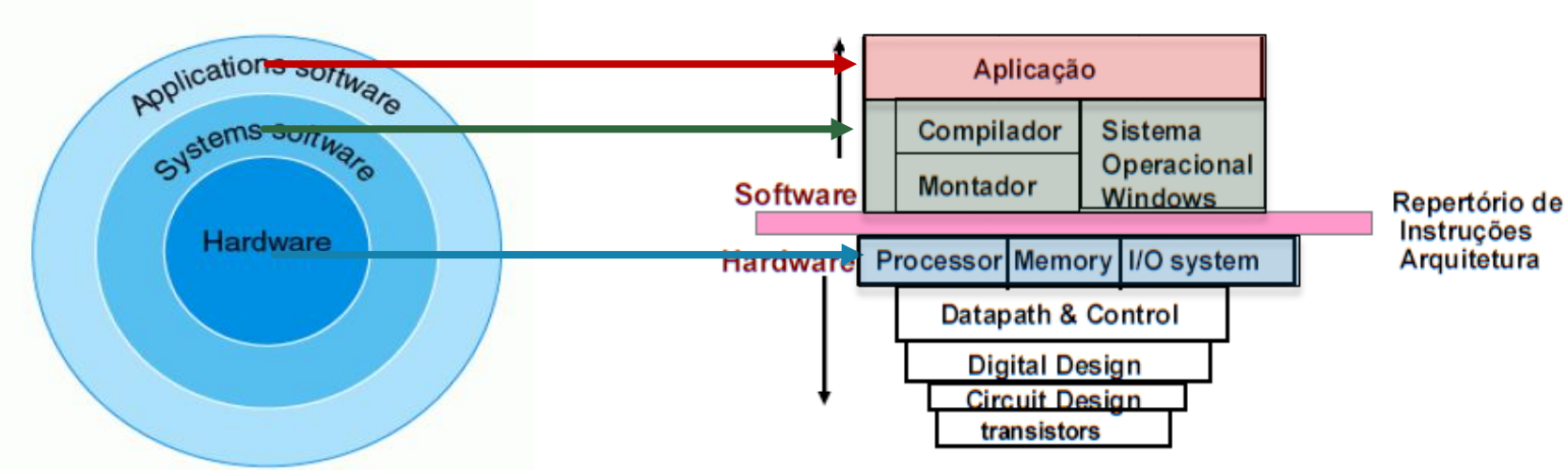
Abstrações de um Computador

- Faz-se necessário a criação de camadas de abstrações que escondam detalhes de implementação de um computador para desenvolver as aplicações atuais cada vez mais complexas



Abstrações de um Computador

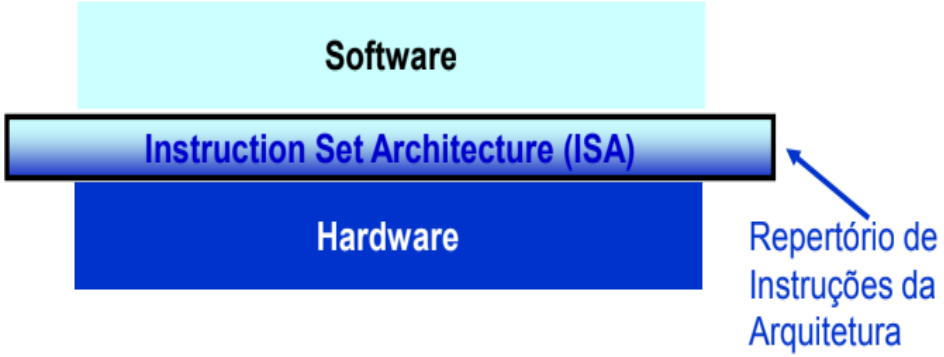
- Aplicação: abstração de dados, armazenamento, procedural
- Softwares de sistema
  - Compiladores: abstração do repertório de instruções da máquina
  - Sistema Operacional: abstração de concorrência, recursos de HW, hierarquia de memória



Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?

Interface HW/SW: Repertório de Instruções da Arquitetura

- Todo processador já é fabricado de modo a conter em seu interior um grupo dessas instruções, chamado de conjunto de instrucões.



- Última abstração do HW vista pelo SW
- Computadores diferentes podem ter diferentes ISAs
  - Mas com muitos aspectos em comum
- Visando portabilidade de código, indústria se alinha em torno de quantidade pequena de ISAs diferentes

Evolução de ISAs

- Até metade da década de 60 computadores tinham ISAs com quantidade reduzida de instruções e instruções simples
  - Simplifica implementação
- Fim da década de 60 surge ISAs com grande número de instruções complexas
  - **Complex Instruction Set Computer (CISC)**
  - Difícil implementação e existência de muitas instruções pouco usadas
- Começo da década de 80 ISAs com instruções simples voltam a ser comuns
  - **Reduced Instruction Set Computer (RISC)**

Exemplos de Processadores CISC e RISC

- **CISC**
  - Intel x86, Pentium, AMDx86, AMD Athlon
  - Muito utilizados em PCs
- **RISC**
  - MIPS, SPARC, ARM, PowerPC
  - Muito utilizados em sistemas embarcados
- Tendência hoje é termos processadores híbridos
  - Ideias de RISC foram incorporados a CISC e vice-versa

Aprendendo as Operações Simples

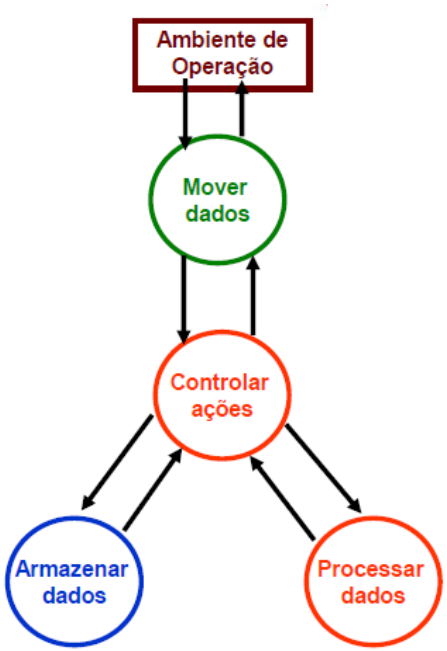
- Embora o conjunto *de instruções de máquina* seja uma característica associada a arquitetura, em geral, o conjunto de instruções de diferentes arquiteturas **são bastante similares.**
  - Isso acontece devido:
    - Aos projetos possuírem **princípios básicos semelhantes**
    - Algumas **operações básicas** devem ser oferecidas por todos os computadores.
- Essas operações que iremos aprender para podermos ser capazes de compreender as instruções mais complexas.

Repertório (ISA) do Processador MIPS

- Utilizado como exemplo nesta disciplina
- Desenvolvido no começo de 80, é um bom exemplo de uma arquitetura RISC
- Muito utilizado no mercado de sistemas embarcados
  - Aplicações em eletrônicos diversos, equipamento de rede/armazenamento, câmeras, impressoras, bluerays, smart watches,etc

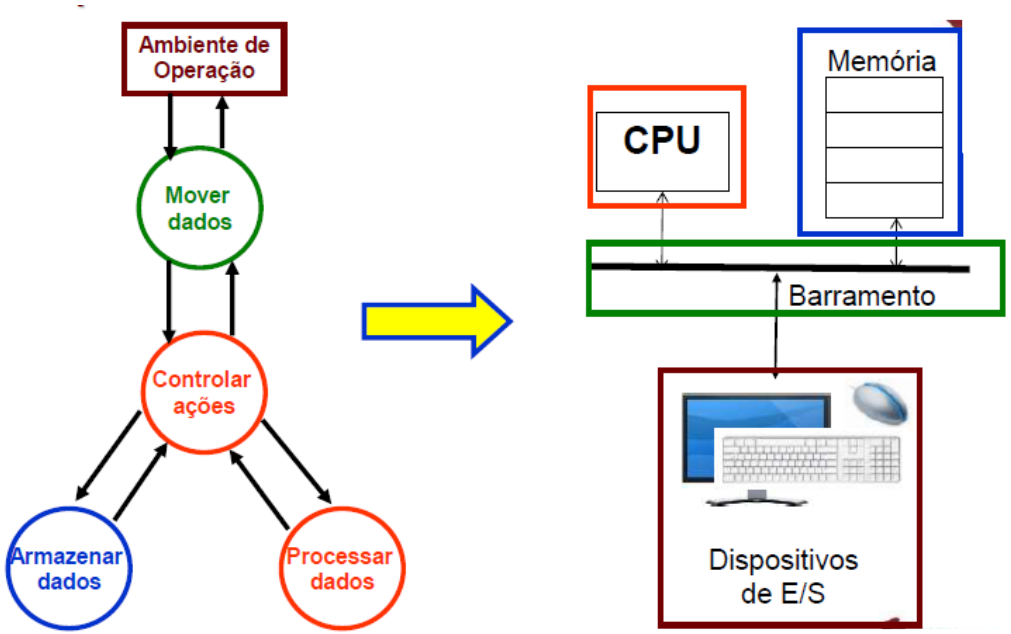
Mas antes, vamos entender algumas coisas...
Visão Funcional de um Computador

- O HW de um computador deve realizar 4 ações:
  - Mover dados
  - Armazenar dados
  - Processar dados
  - Controlar as ações mencionadas



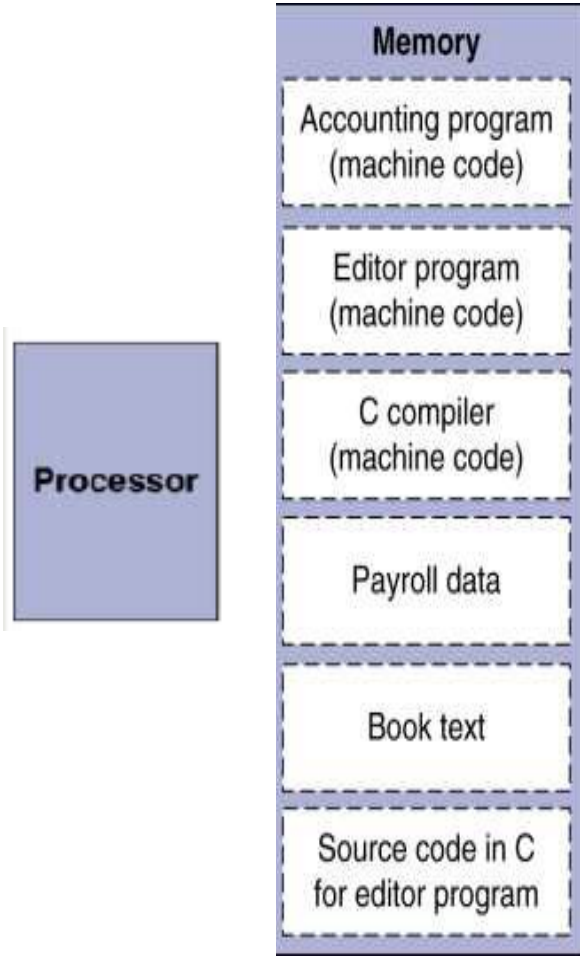
Mas antes, vamos entender algumas coisas...

Mapeando Funcionalidades em um Computador



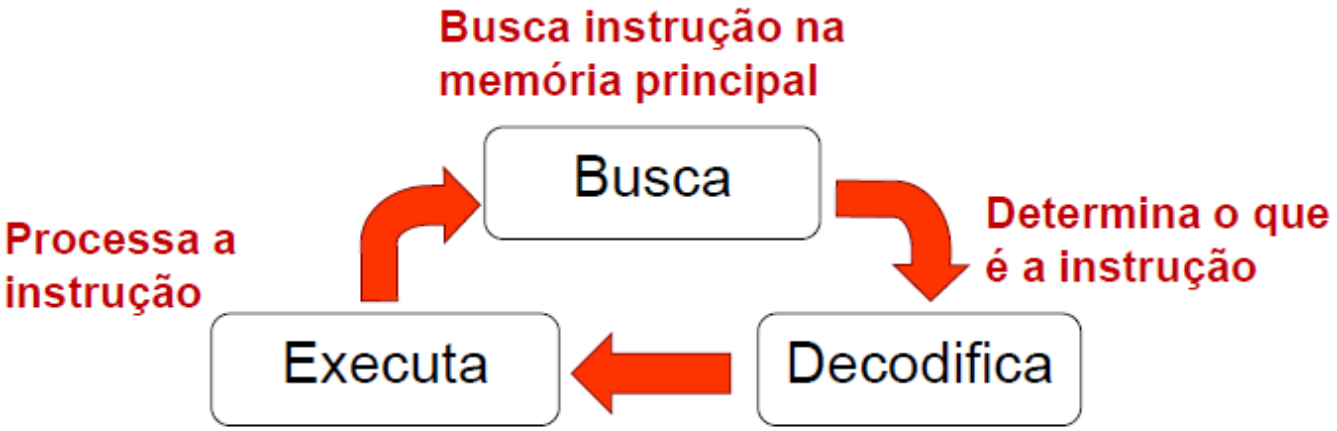
Mas antes, vamos entender algumas coisas...
Como Funciona um Computador?

- Conceitos básicos para funcionamento de um computador:
  - —Dados e instruções são armazenados na memória
  - —Para simplificar, vamos considerar que é uma única memória para instruções e dados
- Conteúdo da memória é acessado através de um endereço, não importando o tipo de dado armazenado
- Execução ocorre de maneira sequencial (a não ser que seja explicitamente especificado), uma instrução após a outra



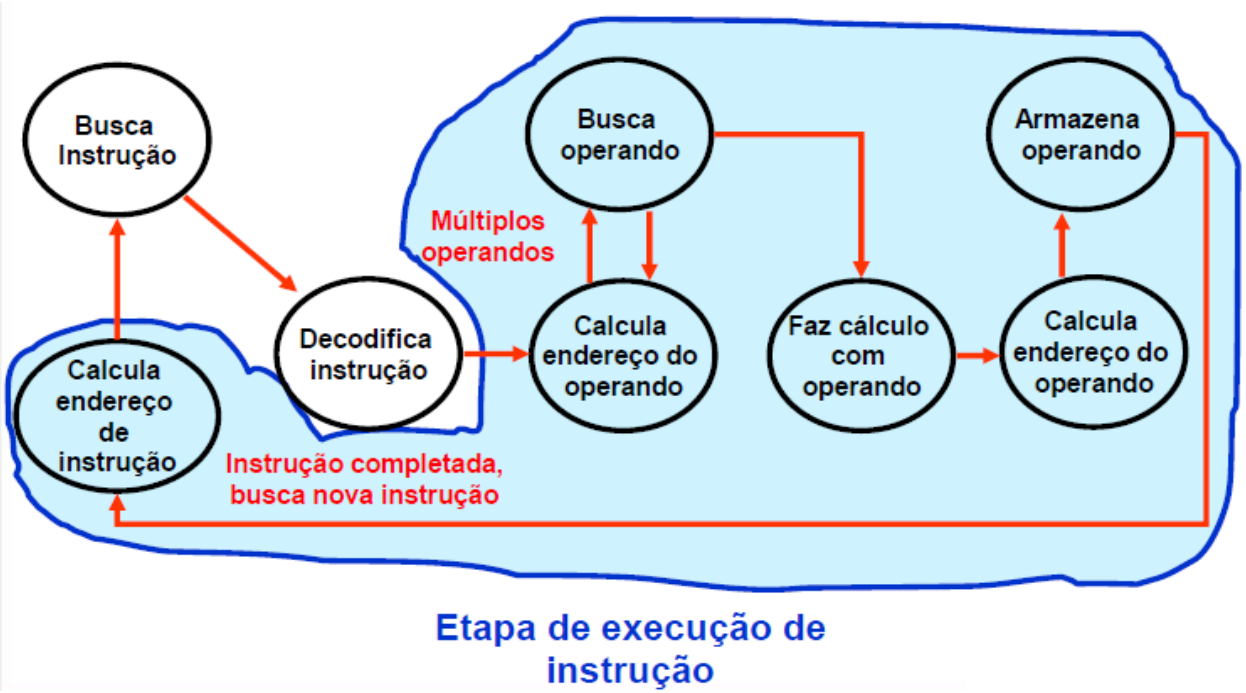
Mas antes, vamos entender algumas coisas...
Visão Simplificada de Processamento de Instrução

- CPU faz continuamente 3 ações:



Mas antes, vamos entender algumas coisas...

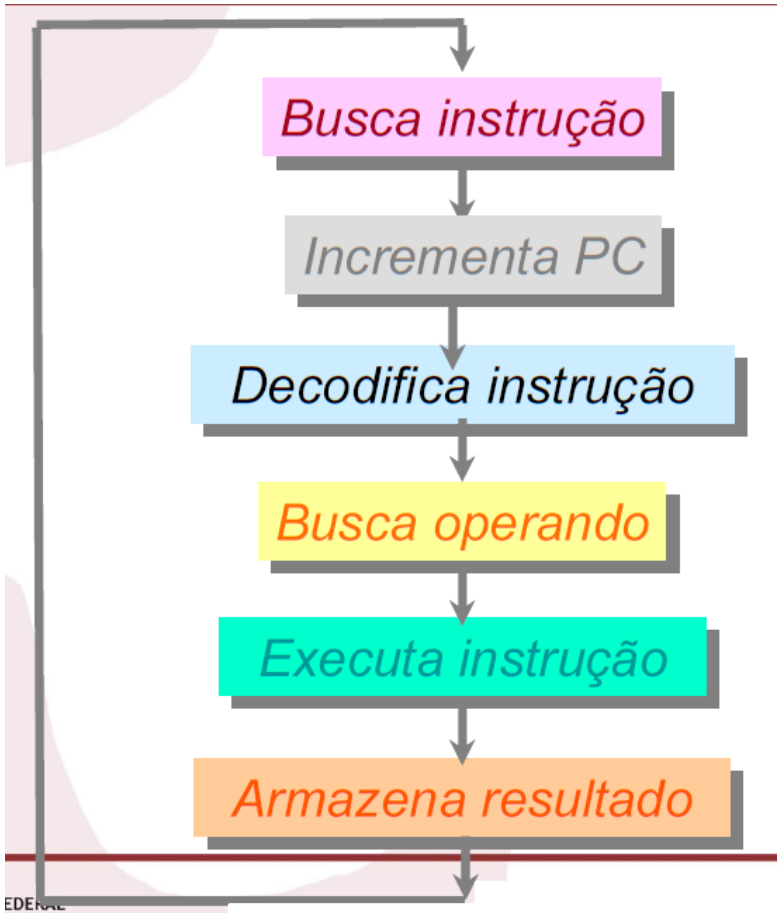
Visão Detalhada da Execução de uma Instrução



Etapa de execução de instrução

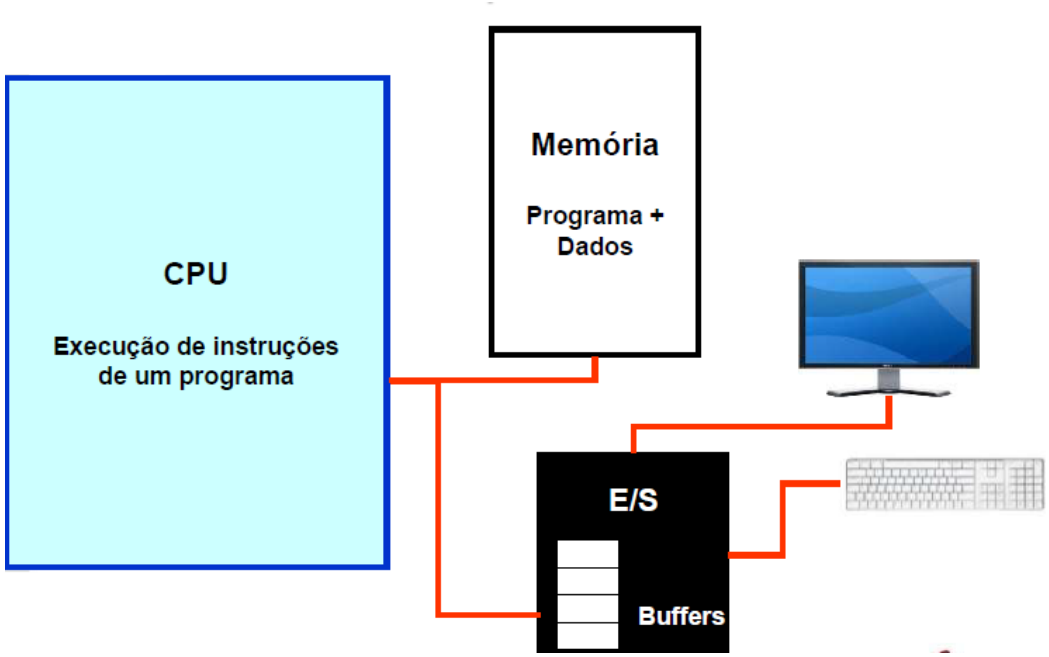
Mas antes, vamos entender algumas coisas...

Visão Detalhada da Execução de uma Instrução



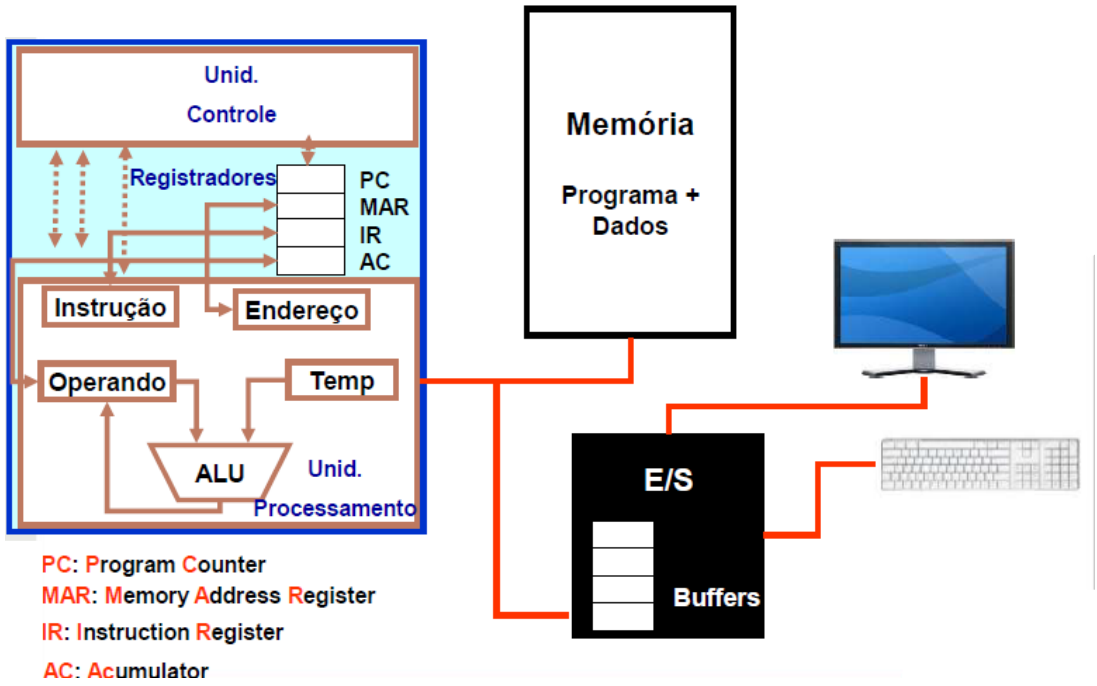
Mas antes, vamos entender algumas coisas...

Componentes de um Computador



Mas antes, vamos entender algumas coisas...

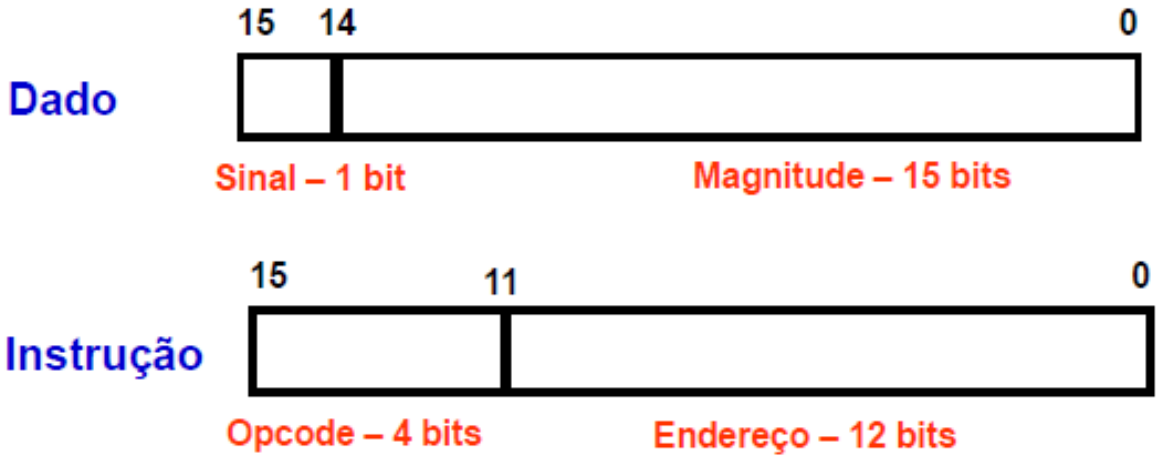
Mais Detalhes de uma CPU



Mas antes, vamos entender algumas coisas...

Executando um Programa em um Computador Hipotético

- Instruções e Dados ocupam 16/32/64 bits na memória
- Memória composta por **palavras** de 16 bits
- Formato de Dados e Instruções:



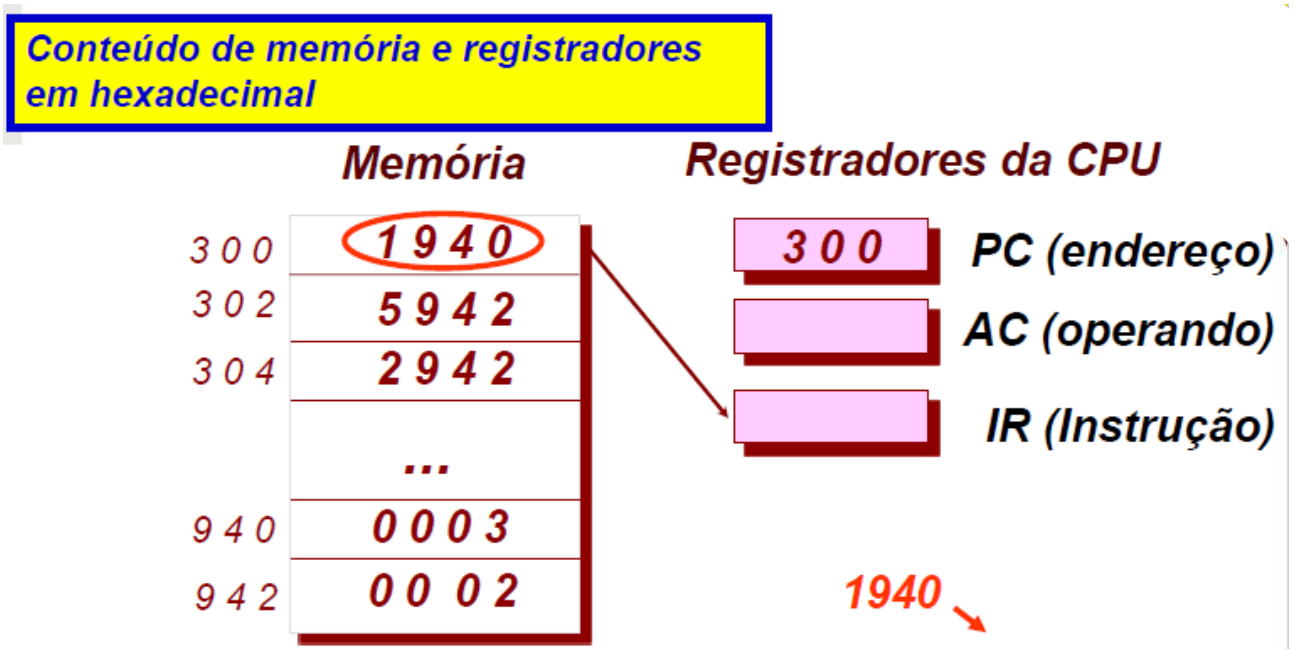
- 2^4 = 16 instruções possíveis nesta arquitetur

Mas antes, vamos entender algumas coisas...
Executando um Programa em um Computador Hipotético

- Por simplicidade, examinaremos 3 registradores
  - —**PC** - Contém o endereço da instrução a ser executada
  - —**AC** - Contém um operando
  - —**IR** - Contém a instrução executada
- Repertório de Instruções

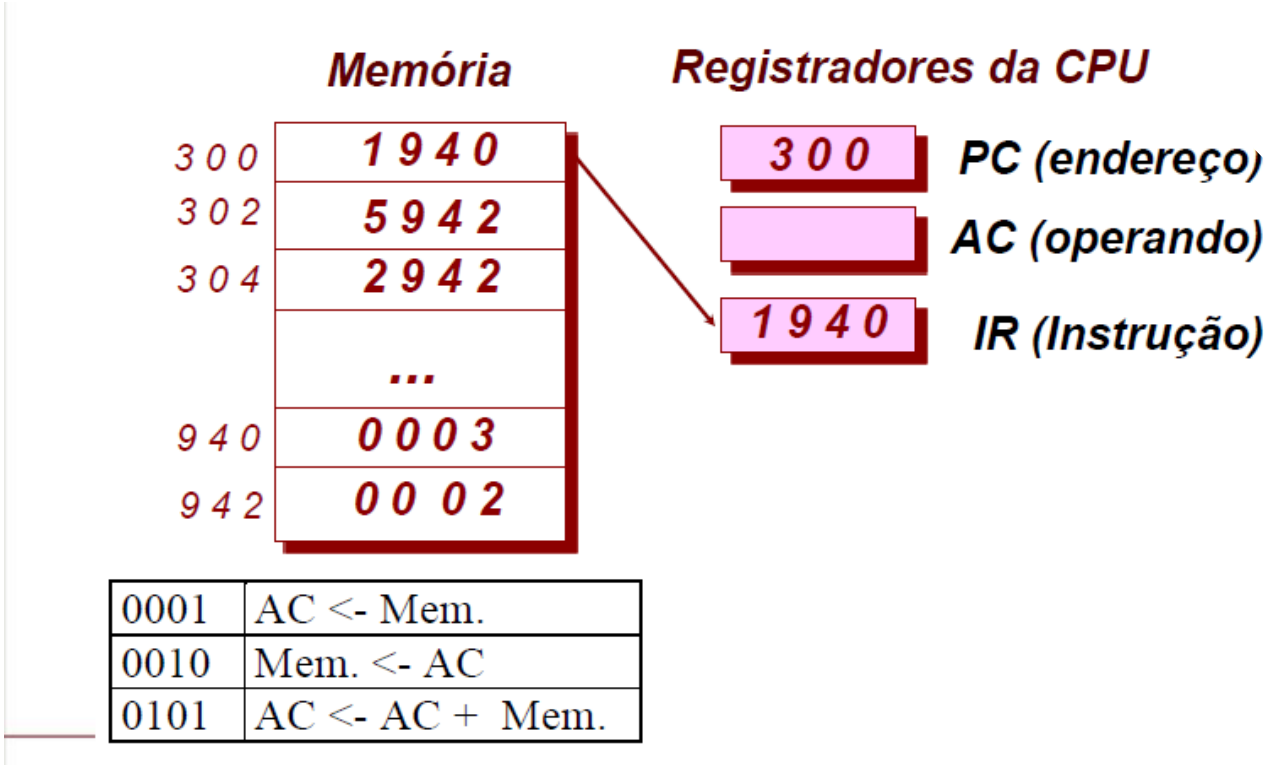
Opcode	Significado	Descrição
0001	AC ← Mem	Carrega em AC conteúdo de memória
0010	Mem ← AC	Salva na memória conteúdo de AC
0101	AC ← AC + Mem	Soma a AC conteúdo de memória

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



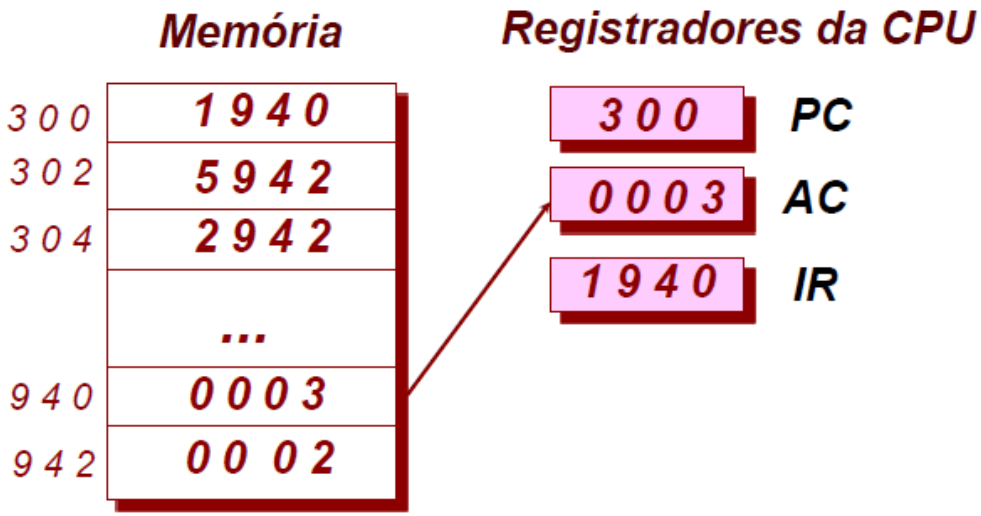
0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



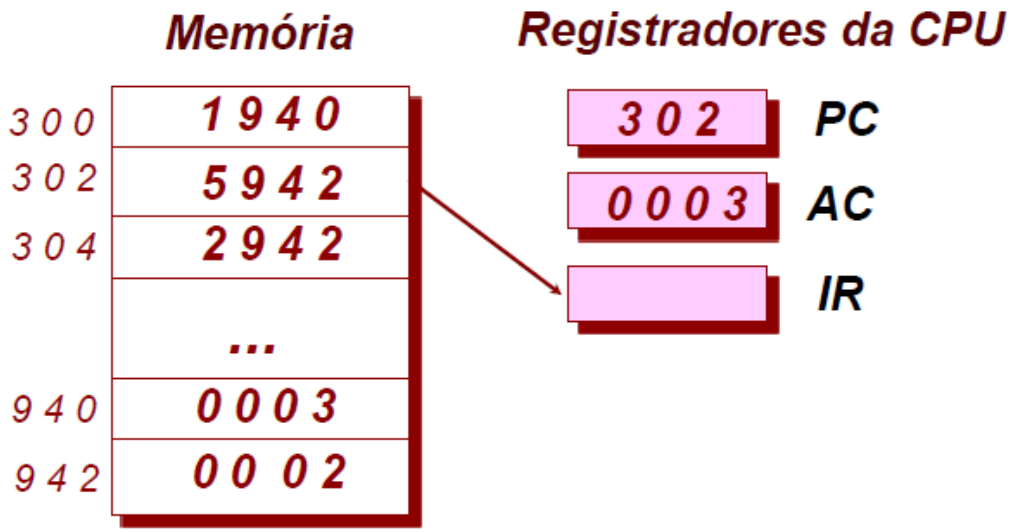
0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



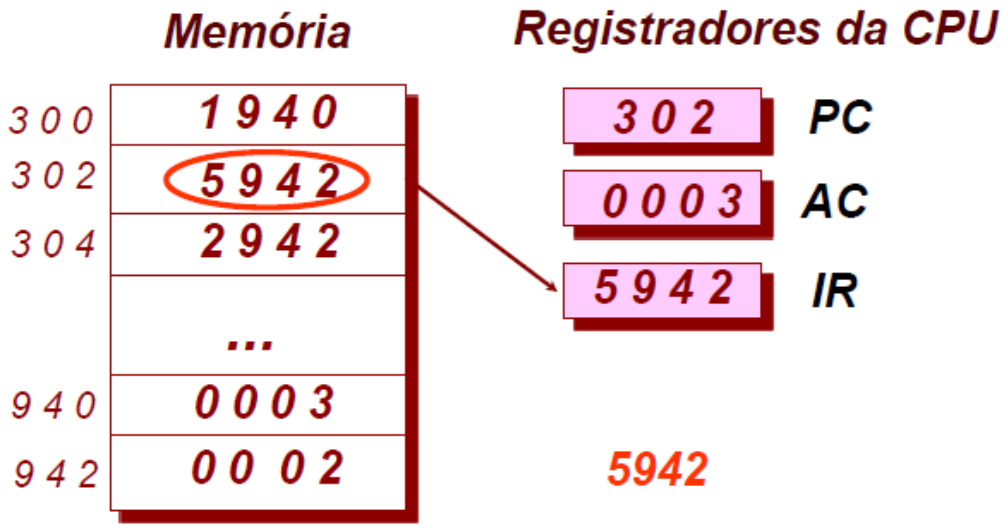
0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



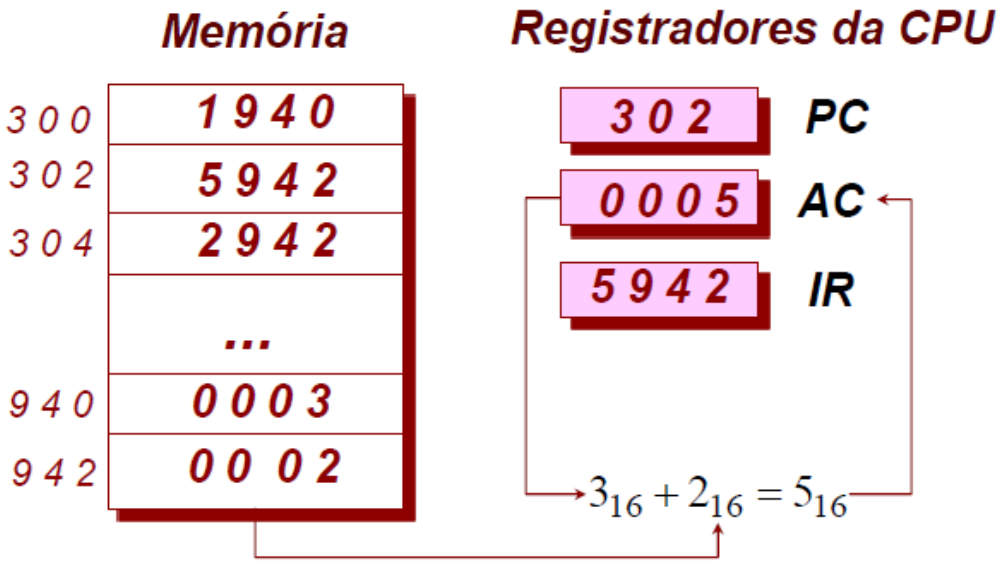
0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



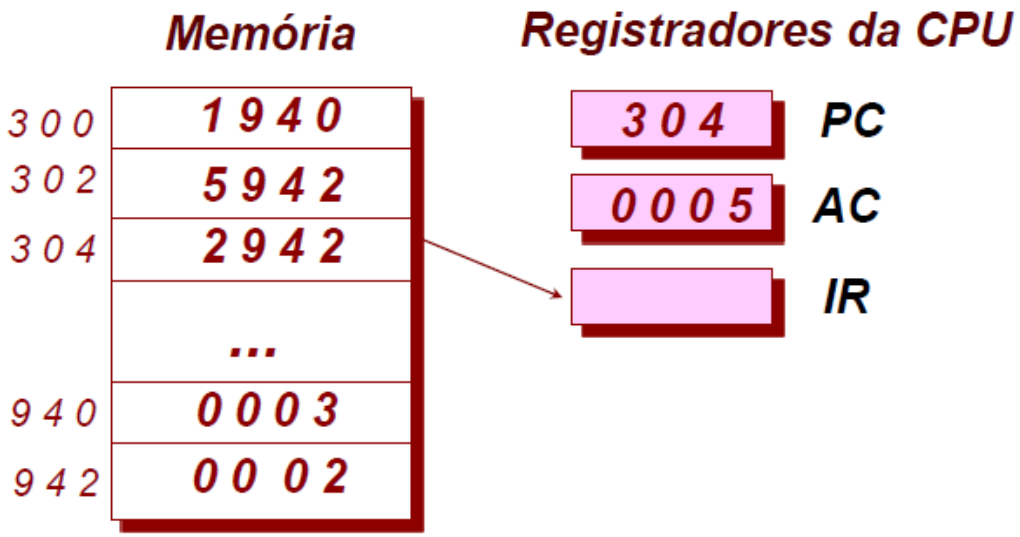
0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



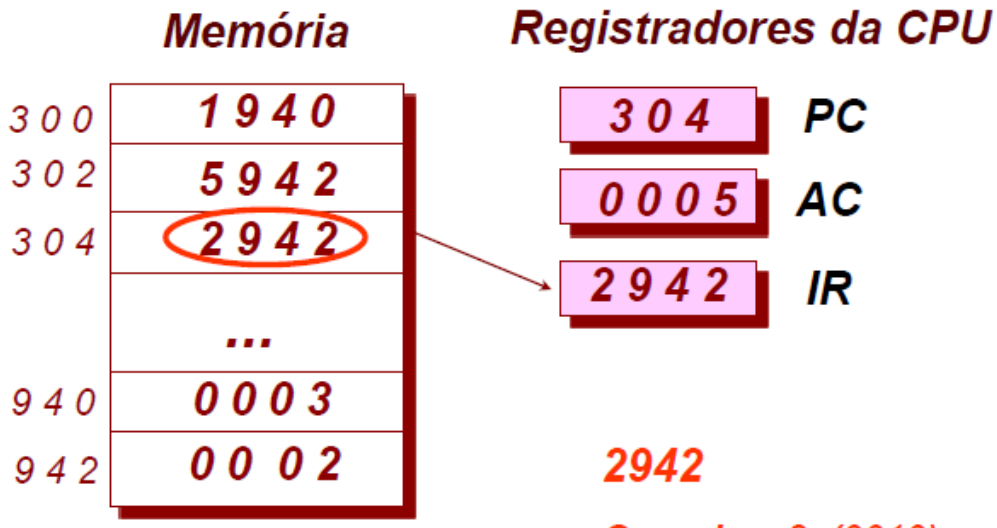
0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



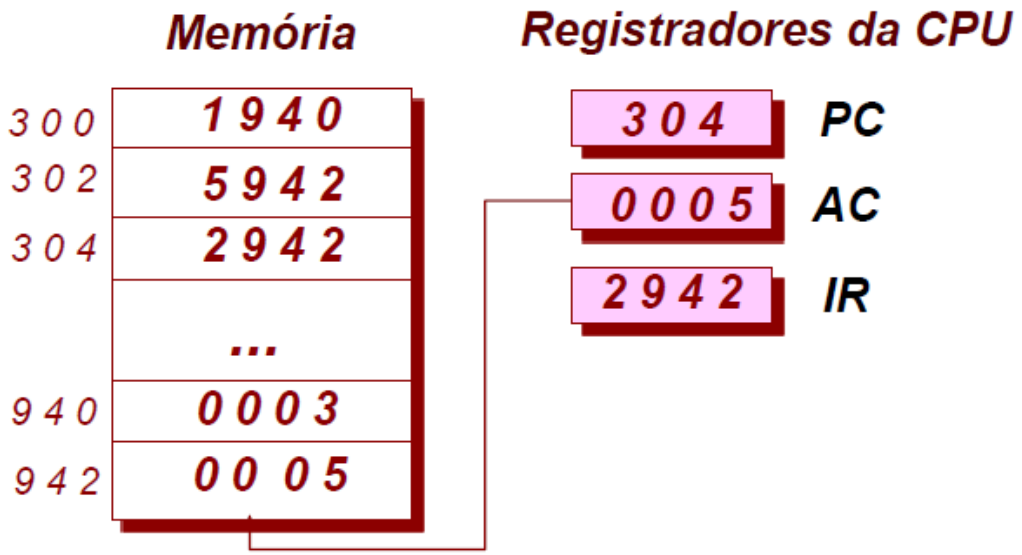
0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Mas antes, vamos entender algumas coisas...
Passo a Passo da Execução de um Programa



0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

Princípios de Projeto do MIPS (RISC)

- **Simplicidade é favorecida pela regularidade**
  - Instruções de tamanho fixo
  - Poucos formatos de instruções
  - Opcode sempre utiliza os primeiros 6 bits
- **Quanto menor, mais rápido**
  - Repertório de instruções limitados
  - Quantidade de registradores limitados
  - Número reduzido de modos de endereçamento
- **Torne rápido o caso mais comum**
  - Existência de instruções que contém operandos
- **Bom projeto requer boas escolhas (compromissos)**
  - Diferentes formatos de instruções complica decodificação, CONTUDO permite instruções de tamanho fixo

O processador MIPS

- Um pouco sobre o MIPS:
  - —Criado na década de 80 por John L. Hennessy.
  - —Microprocessador bastante utilizado.
    - Em 2002, foram fabricados 100 milhões de unidades.
    - Encontrados em produtos de várias empresas.
      - ATI, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, Texas Instrument, Toshiba, etc.

O processador MIPS

- Algumas Características
  - Instruções simples, sempre realizam uma única operação.

O processador MIPS

- Algumas Características
  - —Instruções simples, sempre realizam uma única operação.
  - —As instruções possuem tamanho fixo de 32 bits.

O processador MIPS

- Algumas Características
  - No total, o MIPS **possui 32 registradores**. Cada um deles, de 32 bits.
    - Estes 32 bits representam a palavra (word), ou seja, a unidade de acesso natural de um computador.
    - Em geral, o tamanho da palavra é de 32 bits, porém nos novos processadores como os Core2 Duo, esta palavra é de 64 bits.

O processador MIPS

- Algumas Características
  - Cada instrução MIPS sempre trabalha com 3 operandos.
    - Esta característica torna o **hardware mais simples**.

O Processador MIPS

- Princípios fundamentais de projeto MIPS:
  - “*Simplicidade favorece a regularidade*”
  - “Menor significa mais rápido”
  - “*Agilize os casos mais comuns*”
  - “Um bom projeto exige bons compromissos”.

Instruções para soma e subtração

- Todo computador precisa ser capaz de realizar aritmética
- As instruções de adição e subtração do MIPS são, respectivamente:
  - add e sub.
  - Exemplos:
    - add a, b, c # A soma de b + c é colocada em a.
    - sub a, b, c # A subtração de b - c é colocada em a.
    - add a, a, c # A soma de a + c é colocada em a.
  - Notação rígida:
    - MIPS realiza apenas uma operação e
    - **Sempre** precisa ter exatamente três operandos: dois de origem e um de destino.
      - destino, fonte 1, fonte 2
    - Faz parte do princípio de projeto 1: “Simplicidade favorece a regularidade”
    - O hardware para o número de variável de operandos é mais complicado do que o hardware para um número fixo.
  - As palavras à direita do símbolo # são comentários

Instruções para soma e subtração

- Como é compilada as seguintes instruções em java?

  a = b + c;

  d = a - e;

Instruções para soma e subtração

- Como é compilada as seguintes instruções em java?

  a = b + c;

  d = a - e;

- Resposta:

  add a, b, c

Instruções para soma e subtração

- Como é compilada as seguintes instruções em java?

  a = b + c;

  d = a - e;

- Resposta:

  add a, b, c

  sub d, a, e

Instruções para soma e subtração

- E no caso de uma atribuição mais complexa, como por exemplo:
  - f = (g + h) – (i + j) ?
    - Neste caso, é necessário gerar mais de uma instrução.
    - Temos também de utilizar variáveis temporárias.
    - Resposta ??

Instruções para soma e subtração

- E no caso de uma atribuição mais complexa, como por exemplo:
  - f = (g + h) – (i + j) ?
    - Neste caso, é necessário gerar mais de uma instrução.
    - Temos também de utilizar variáveis temporárias.
    - Resposta ??
      - add t0, g, h

Instruções para soma e subtração

- E no caso de uma atribuição mais complexa, como por exemplo:
  - f = (g + h) – (i + j) ?
    - Neste caso, é necessário gerar mais de uma instrução.
    - Temos também de utilizar variáveis temporárias.
    - Resposta ??
      - add t0, g, h
      - add t1, i, j

Instruções para soma e subtração

- E no caso de uma atribuição mais complexa, como por exemplo:
  - f = (g + h) – (i + j) ?
    - Neste caso, é necessário gerar mais de uma instrução.
    - Temos também de utilizar variáveis temporárias.
    - Resposta ??
      - add t0, g, h
      - add t1, i, j
      - sub f, t0, t1

Instruções para soma e subtração

Resumo até agora: Essas instruções são representações simbólicas daquilo que o processador MIPS realmente entende

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add a,b,c`	a = b + c	Always three operands
	subtract	`sub a,b,c`	a = b - c	Always three operands

Exercícios

- Considere os seguintes registradores para cada variável: a = $s0, b = $s1, c = $s2, d = $s3, e = $s4, f = $s5.

1. a = b - c
2. b = a + c
3. d = (a + b - c)
4. f = (a + b) - d
5. c = a - (b + d)
6. e = (a - (b - c)
7. e = (a - (b - c) + f)
8. f = e - (a - b) + (b - c)

Registradores no MIPS

- Operandos do hardware de um computador
  - Ao contrário dos programas nas linguagens de alto nível, os operandos das instruções aritméticas são restritos.
    - Os operandos de uma instrução aritmética são **registradores**.
    - leitura e escrita em registradores são muito mais rápidas que em memória
  - Lembrem-se que no MIPS só temos 32 registradores.
  - Os grupos de 32 bits ocorrem com tanta frequência que recebem o nome de word (palavra) na arquitetura MIPS.
  - Qual o motivo para termos uma quantidade tão pequena de registradores?

Registradores no MIPS

- Além de fatores ligados ao consumo de energia, complexidade do hardware e preço final do produto, também temos um outro fator, nosso *Princípio de Projeto 2*:
  - **"Menor significa mais rápido"**
    - Uma quantidade muito grande de registradores pode aumentar o tempo do ciclo do clock, simplesmente porque os sinais eletrônicos levam mais tempo quando precisam atravessar uma distância maior.
      - Cuidado. Em alguns casos ter mais registradores promove um melhor desempenho.
      - Em um projeto de hardware devemos sempre pesar o limite deste número de registradores (ou seja, o momento em que não temos mais melhora de desempenho), bem como sua viabilidade.
  - Outro motivo para não usar mais de 32 é o número de bits que seria necessário no formato da instrução.

Registradores no MIPS

- Os nomes dos registradores MIPS obedecem ao seguinte padrão:
  - Utilizamos “$” seguido por dois caracteres para representar um registrador.
  - $s0, $s1, $s2,... representam os registradores que correspondem às **variáveis dos programas** em C e Java.
  - $t0, $t1, ... Representam os registradores que armazenam valores **temporários**.

Registradores no MIPS

- Logo, para o programa anterior, teríamos:
  - f = (g + h) - (i + j);

Registradores no MIPS

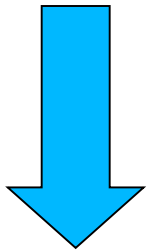
- Logo, para o programa anterior, teríamos:
  - f = (g + h) - (i + j);

Assembly

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

Registradores no MIPS

- Logo, para o programa anterior, teríamos:
  - f = (g + h) - (i + j);



Assembly

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Registradores no MIPS

- Pergunta:
  - O processador MIPS possui apenas **32 registradores**. Como fazemos para trabalhar com variáveis complexas (por exemplo, arrays, estrutura de dados)

Registradores no MIPS

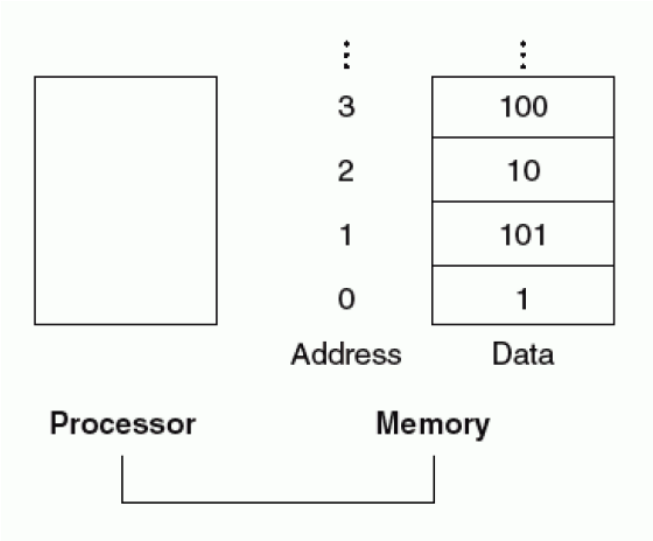
- Pergunta:
  - O processador MIPS possui apenas **32 registradores**. Como fazemos para trabalhar com variáveis complexas (por exemplo, arrays, estrutura de dados)
    - A quantidade de elementos de dados nestas variáveis é muito maior do que a quantidade de registradores em um computador?

Instruções para transferência de dados

- Utilizando a **memória principal**.
  - Memória é muito utilizada para armazenar dados compostos
    - Arrays, estruturas, dados dinâmicos

Instruções para transferência de dados

- Precisamos de instrução para realizar **transferência de dados** entre a memória e os registradores.
- Para acessar uma word na memória, a instrução precisa fornecer o **endereço** de memória.
  - A memória é apenas uma sequência grande e unidimensional, com o endereço atuando como índice para esse array, começando do 0.



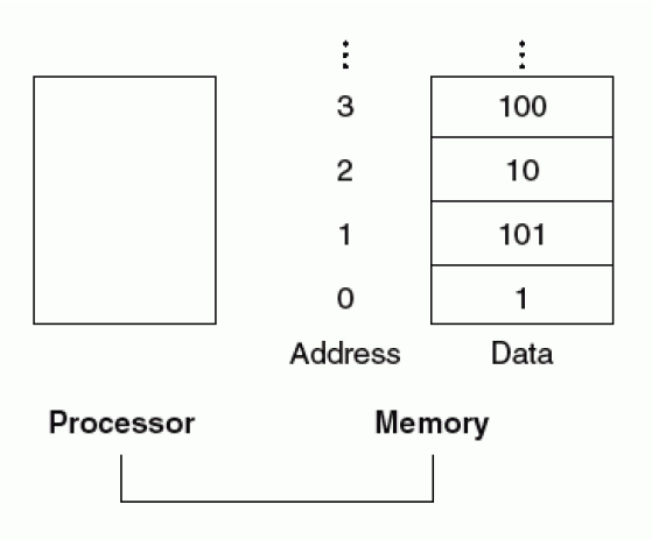
Instruções para transferência de dados

- A instrução de transferência de dados que copia dados da memória para o registrador é o *load (lw)*
  - Sintaxe:
    - lw $r1, const($r2),
    - **$r1** é o registrador que armazena o conteúdo da memória.
    - **const** representa um valor constante que é somado ao endereço presente no registrador $r2.
    - **Endereço** = const + $r2

Instruções para transferência de dados

- Qual código assembly para o seguinte trecho de código, considere A um array de inteiros?
- Suponha que A seja uma sequência de 100 words
- Suponha que g e h estão associados aos registradores $s1 e $s2
- Suponha que o endereço inicial ou *endereço base* esteja em $s3

```
g = h + A[8];
```



Instruções para transferência de dados

- Qual código assembly para o seguinte trecho de código, considere A um array de inteiros?

```
g = h + A[8];
lw $t0,8($s3);
add $s1,$2,$t0;
```

A constante é chamado de *offset* e o registrador acrescentado para formar o endereço é chamado de *registrador base*

Mas cuidado com o valor 8 ...

Instruções para transferência de dados

- Memória endereçada por byte (8 bits)
  - Cada endereço sinaliza para uma célula de 1 byte apenas.
- **Contudo** memória é vista como uma sequência de palavras de 32 bits
  - Cada posição do array ocupa 1 palavra (4 bytes ou 32 bits).
  - **Endereços de palavras devem ser múltiplos de 4!**

Instruções para transferência de dados

- Por exemplo: no processador MIPS
  - Inteiros de 32 bits
  - Array de inteiros chamado de a
  - Logo, para acessarmos o inteiro na posição 8, temos de pular os 8 inteiros que aparecem antes no array, assim temos: 4 * 8 = 32

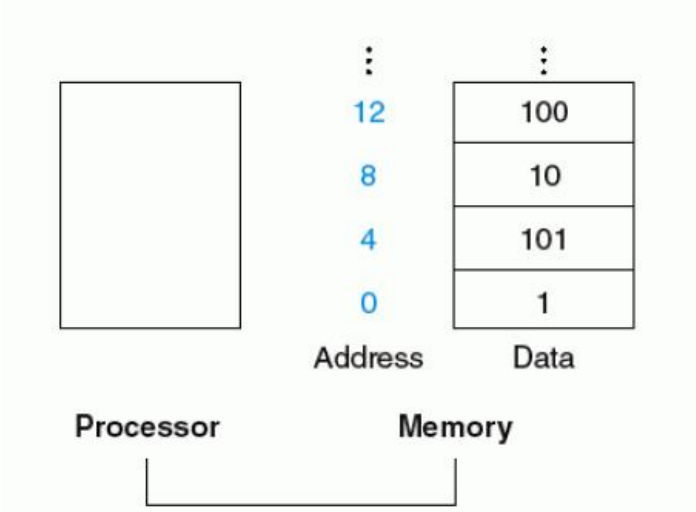
End (a[0]) = 10
End (a[1]) = 14

End(a[i]) = End-inicial + i x 4

Instruções para transferência de dados

- Qual código assembly para o seguinte trecho de código, considere A um array de inteiros?

```
g = h + A[8];
lw $t0,32($s3);
add $s1,$2,$t0;
```



Exercícios

- Converta as instruções abaixo:

- a = b[15] - c;
- b = a[5] + c[3];
- c = b - a[21];

- Use $s0 para a, $s1 para b e $s2 para c.

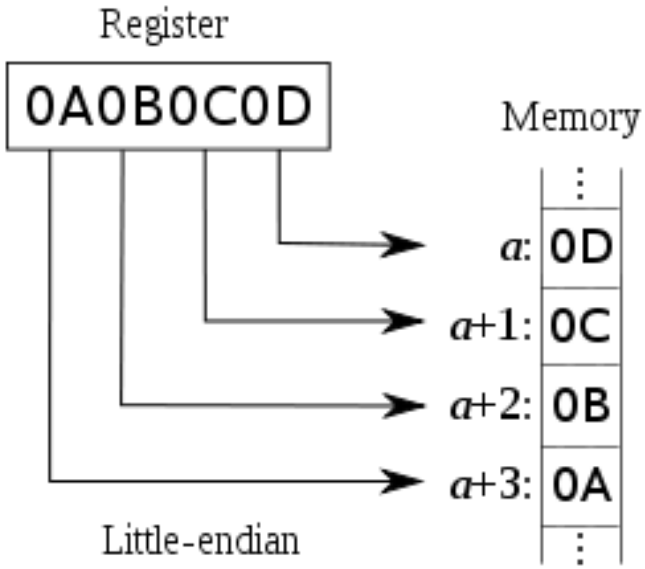
Endianness

- Grande parte dos processadores fazem restrição com relação ao endereço em que começam suas palavras.
  - No Mips, por exemplo, suas palavras precisam começar em endereços que sejam múltiplos de 4.
  - Esse requisito é denominado ***restrição de alinhamento***.
- Com relação ao byte de endereçamento, os processadores podem ser *big endian* ou *little endian*.

Endianness

- Processadores *Little-Endian* numeram os bytes dentro de uma palavra (word) de forma que o byte com o valor mais baixo é o mais à direita.

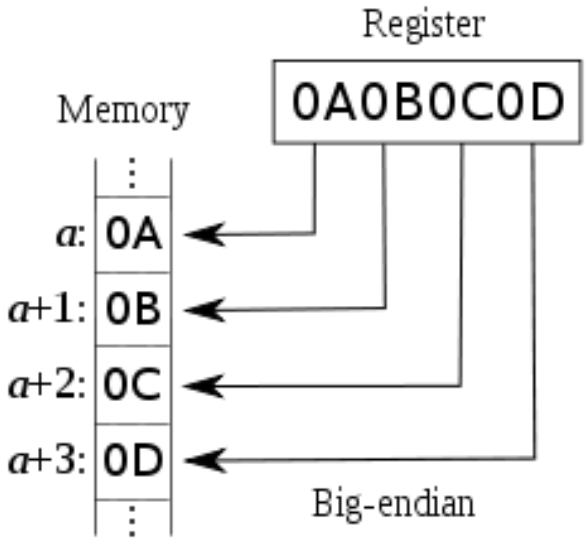
Byte #			
3	2	1	0



Endianness

- Processadores Big-Endian numeram os bytes dentro de uma palavra (word) de forma que o byte com o valor mais baixo é o mais à esquerda.
- MIPS é Big-Endian

Byte #			
0	1	2	3



Instruções para transferência de dados

- A instrução que desempenha função inversa ao load word (lw) é a instrução store word (sw).
- Basicamente ela transfere o conteúdo de um registrador para um endereço específico da memória principal.
- Formato é semelhante ao lw.
- Sintaxe:
  - Similar ao do lw.
    - sw $r1, const($r2)

Exercícios

- Converta as instruções abaixo:

- A[10] = f - g;
- B[245] = h + g;
- C[0] = i - f;

- Use $s0 para f, $s1 para g, $s2 para h e $s3 para i.
- Os endereços base de A, B e C são $s4, $s5, $s6, respectivamente.

Instruções para transferência de dados

- Exemplo:
  - A[12] = h + A[8];

h em $s2,
endereço base de a em $s3

Instruções para transferência de dados

- Exemplo:
  - A[12] = h + A[8];
    - Resposta:

      lw $t0, 32($s3)

h em $s2,
endereço base de a em $s3

Instruções para transferência de dados

- Exemplo:
  - A[12] = h + A[8];
    - Resposta:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
```

h em $s2,
endereço base de a em $s3

Instruções para transferência de dados

- Exemplo:
  - A[12] = h + A[8];
    - Resposta:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48, $s3;
```

h em $s2,
endereço base de a em $s3

Outro exemplo

- Array com variável de indexação
  - g = g + a[i];

g em $s1,
i em $s2,
endereço base de a em $s3

Outro exemplo

- Array com variável de indexação
  - g = g + a[i];
    - Resposta:

      add $t1, $s2, $s2

g em $s1,
i em $s2,
endereço base de a em $s3

End(a[i]) =
base + i + i + i + i

Outro exemplo

- Array com variável de indexação
  - g = g + a[i];
    - Resposta:

```
add $t1, $s2, $s2
add $t1, $t1, $t1
```

g em $s1,
i em $s2,
endereço base de a em $s3

End(a[i]) =
base + i + i + i + i

Outro exemplo

- Array com variável de indexação
  - g = g + a[i];
    - Resposta:

```
add $t1, $s2, $s2
add $t1, $t1, $t1
add $t1, $t1, $s3
```

g em $s1,
i em $s2,
endereço base de a em $s3

End(a[i]) =
base + i + i + i + i

Outro exemplo

- Array com variável de indexação
  - g = g + a[i];
    - Resposta:

```
add $t1, $s2, $s2
add $t1, $t1, $t1
add $t1, $t1, $s3
lw $t0, 0($t1)
```

g em $s1,
i em $s2,
endereço base de a em $s3

End(a[i]) =
base + i + i + i + i

Outro exemplo

- Array com variável de indexação
  - g = g + a[i];
    - Resposta:

```
add $t1, $s2, $s2
add $t1, $t1, $t1
add $t1, $t1, $s3
lw $t0, 0($t1)
add $s1, $s1, $t0
```

g em $s1,
i em $s2,
endereço base de a em $s3

End(a[i]) =
base + i + i + i + i

Exercícios

- Converta as seguintes instruções considerando g = $s0, h = $s1, i = $s2. Os endereços base de A, B e C são $s3, $s4, $s5, respectivamente. Tente reutilizar os registradores temporários.

a) A[34] = B[3] + g - h

b) A[45] = i - g + D[67]

c) A[79] = i - C[18] + h

d) A[82] = B[2] - C[4]

Mais exercícios

Considere A, B, C arrays de inteiros. Sendo A em $s0, B em $s1 e C em $s2. Converta para assembly.

```
for (int i:= 0; i < 3; i++)
    C[i] := A[i] + B[i];
```

Mais exercícios

Considere A, B, C arrays de inteiros. Sendo A em $s0, B em $s1.
Converta para assembly.

```
for (int i:= 0; i < 3; i++)
    B[i] := A[i] - i;
```

Registradores x Memória

- Acesso a registradores é mais rápido
  - Lembre-se que o MIPS apenas realiza as operações aritméticas com operandos em registradores e o resultado também é armazenado em registrador
- Utilização da memória requer *loads* e *stores*
  - É por isso que o MIPS é chamado de arquitetura LOAD/STORE
  - Mais instruções a serem executadas
- Compilador deve maximizar a utilização de registradores
  - **Otimização de registradores é importante!**

Representação das instruções em linguagem de máquina.

- Embora até o momento só termos visto instruções em assembly, cada uma destas instruções possui uma correspondência binária.
- A estas instruções, em binário, damos o nome de **linguagem de máquina**.
- Por exemplo a instrução add $t0, $s1, $s2 possui a seguinte representação:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

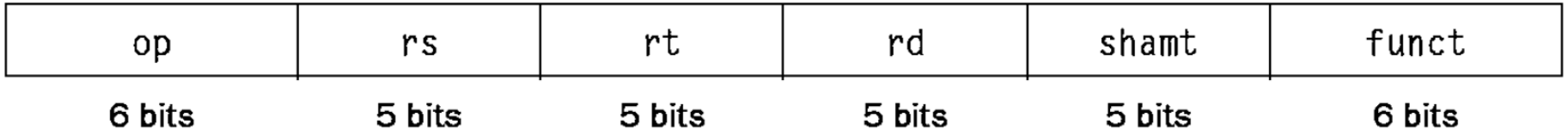
- Vamos entender melhor como funciona essa representação

Representação das instruções

- Codificação das instruções
- Mapeamento de nomes de registradores para números
  - $s0 a $s7 : 16 a 23
  - $t0 a $t7 : 8 a 15
  - $t8-$t9 : 24-25

Formato da Instrução ADD e SUB

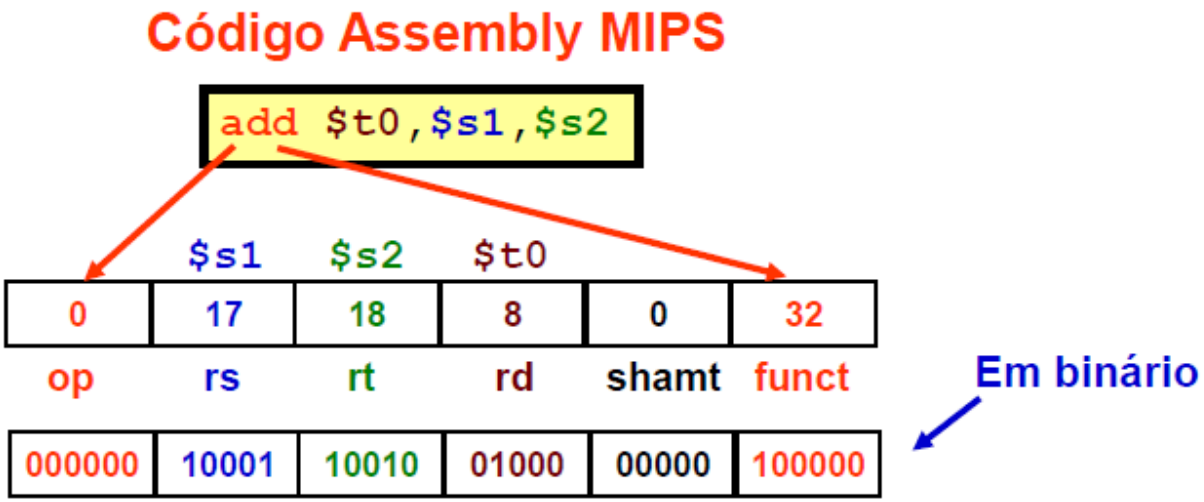
- Campos de uma instrução MIPS:
  - **Instruções tipo-R (de registrador) ou formato R**



  - op: Operação básica da instrução (opcode)
  - rs: registrador do primeiro operando de origem
  - rt: registrador do segundo operando de origem
  - rd: registrado do operando de destino
  - shamt: “shift amount” (quantidade de deslocamento).
    - Não é utilizado para **add** e **sub**
  - funct: Função que estende o opcode.

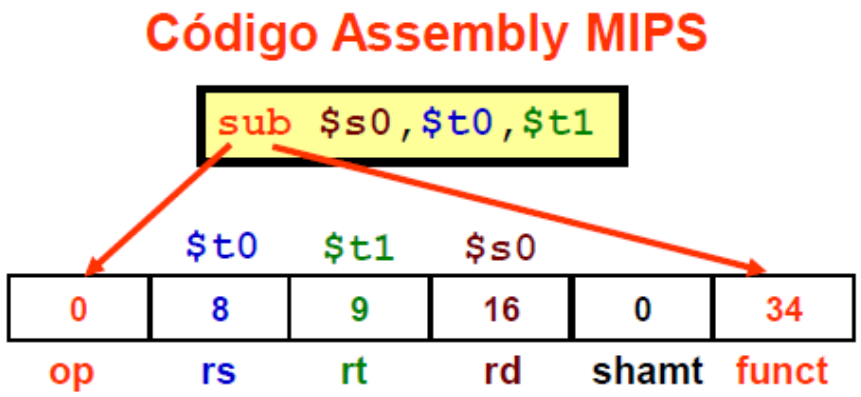
Representando ADD na Máquina

- Número dos registradores
  - $s0 - $s7: 16 - 23
  - $t0 - $t7 : 8-15
  - $t8-$t9 : 24-25



Representando SUB na Máquina

- Número dos registradores
  - $s0 - $s7: 16 - 23
  - $t0 - $t7 : 8-15
  - $t8-$t9 : 24-25



Questão importante

- Existe um problema quando uma instrução precisa de campos maiores.

Questão importante

- Existe um problema quando uma instrução precisa de campos maiores.
- Se usássemos um dos campos de 5 bits para o lw/sw a constante da instrução seria limitada a apenas $2^5 = 32$.

Questão importante

- Existe um problema quando uma instrução precisa de campos maiores.
- Se usássemos um dos campos de 5 bits para o lw/sw a **constante** da instrução seria limitada a apenas $2^5 = 32$.
- Arrays ou estruturas de dados normalmente possui uma quantidade de elementos muito maior do que 32.

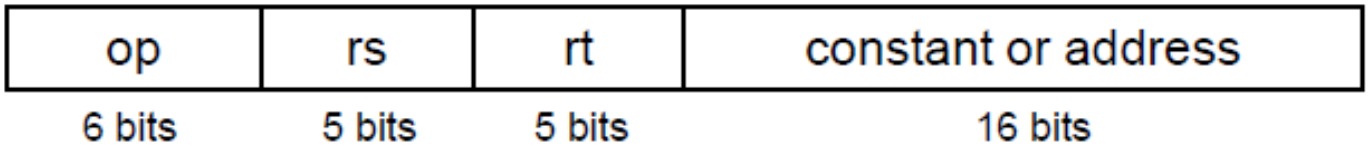
Questão importante

- Existe um problema quando uma instrução precisa de campos maiores.
- Se usássemos um dos campos de 5 bits para o lw/sw a **constante** da instrução seria limitada a apenas $2^5 = 32$.
- Arrays ou estruturas de dados normalmente possui uma quantidade de elementos muito maior do que 32.
- Logo, temos um **conflito** entre manter o mesmo tamanho e manter o mesmo formato

Formato da Instrução LW e SW

- Campos de uma instrução MIPS:
  - **Instruções tipo-I (imediato) ou formato I**

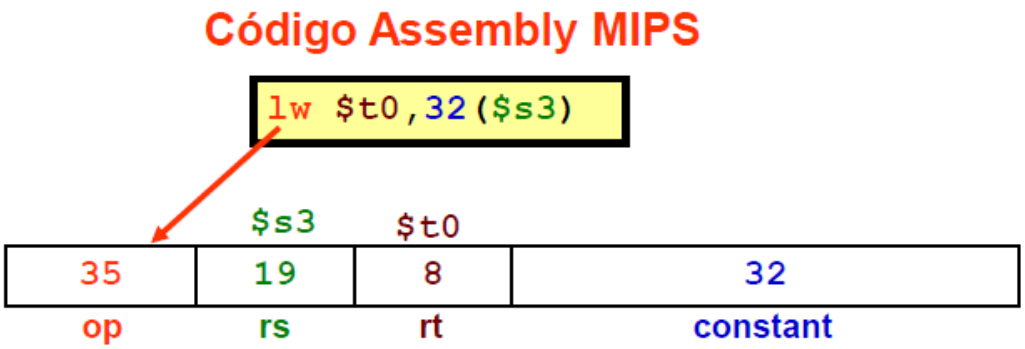
Formato I de Instrução



  - op: Operação básica da instrução (opcode)
  - rs: registrador que neste caso contém endereço base
  - rt: registrador fonte ou destino
  - Constant: constante representa o offset

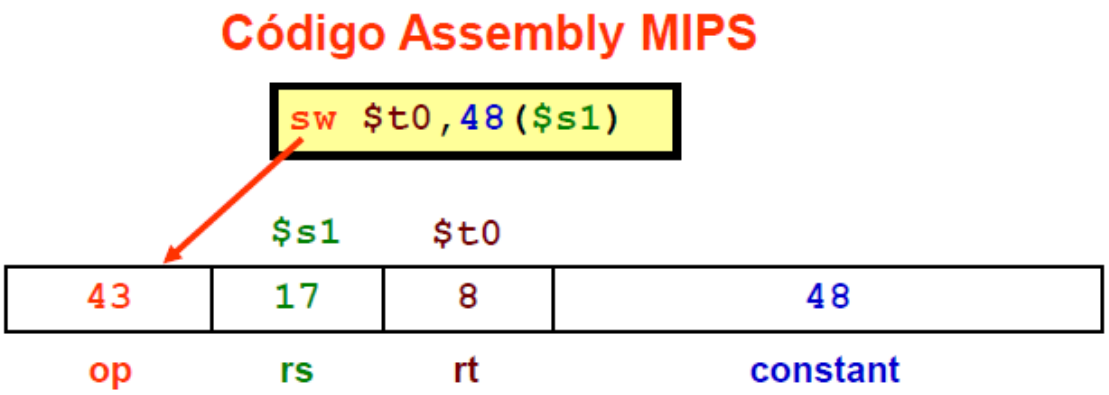
Representando LW na Máquina

- Número dos registradores
  - **$s0 - $s7: 16 - 23**
  - **$t0 - $t7 : 8-15**
  - **$t8-$t9 : 24-25**



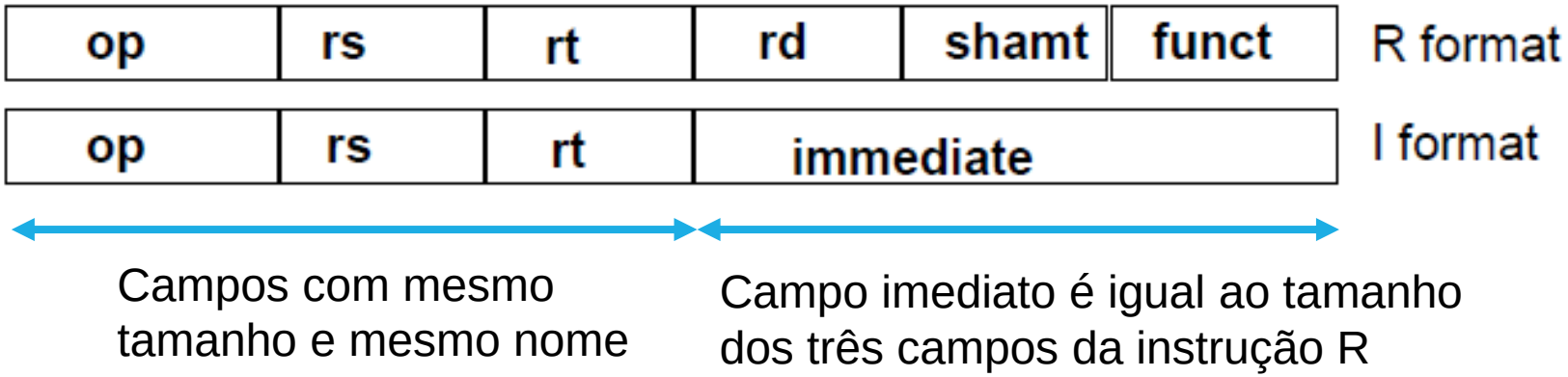
Representando SW na Máquina

- Número dos registradores
  - **$s0 - $s7: 16 - 23**
  - **$t0 - $t7 : 8-15**
  - **$t8-$t9 : 24-25**



Semelhança entre formatos

- Os formatos são semelhantes



Como operamos com constante?

- **Frequentemente** utilizamos pequenas constantes em programas
  - Ex: **a = a + 1;**
  - **Ex: Instruções load e stores**
- Possíveis soluções

Como operamos com constante?

- **Frequentemente** utilizamos pequenas constantes em programas
  - Ex: **a = a + 1;**
  - **Ex: Instruções load e stores**
- Possíveis soluções
  - Armazenar constantes na memória e depois carregá-las
    - Processamento lento

Como operamos com constante?

- **Frequentemente** utilizamos pequenas constantes em programas
  - Ex: **a = a + 1;**
  - **Ex: Instruções load e stores**
- Possíveis soluções
  - Armazenar constantes na memória e depois carregá-las
    - Processamento lento
  - Ter registradores que armazenam a mesma constante
    - MIPS possui o registrador **$zero** que armazena 0
    - Um registrador para cada número ?
    - Poderia precisar de muitos registradores

Como operamos com constante?

- **Frequentemente** utilizamos pequenas constantes em programas
  - Ex: **a = a + 1;**
  - **Ex: Instruções load e stores**
- Possíveis soluções
  - Armazenar constantes na memória e depois carregá-las
    - Processamento lento
  - Ter registradores que armazenam a mesma constante
    - MIPS possui o registrador **$zero** que armazena 0
    - Um registrador para cada número ?
    - Poderia precisar de muitos registradores

Instruções Imediatas

- **Ter instruções especiais que contêm constantes!**
- MIPS oferece instruções onde **uma constante** está embutida na própria instrução
- Isto evita o atraso de termos de ler uma constante da memória para depois utilizá-la em uma soma

Instruções Imediatas

- Instruções imediatas contêm 3 operandos:
  - destino, fonte, constante

- Basicamente, ela realiza uma soma com um valor constante.
  - Sintaxe:
    - **addi $r1, $r2, const**

      #armazena o valor de $r2 + const no registrador $r1.

Instruções Imediatas

- Existe a adição imediata (addi), mas não existe a subtração imediata
  - Subtração : Soma com uma constante negativa

Código C

```
a = b + 8;

a = a - 2;
```

a em $s1,
b em $s2

Instruções Imediatas

- Existe a adição imediata (addi), mas não existe a subtração imediata
  - Subtração : Soma com uma constante negativa

a em $s1,
b em $s2

Código C

```
a = b + 8;

a = a - 2;
```

Código Assembly MIPS

```
addi $s1, $s2, 8
addi $s1, $s1, -2
```

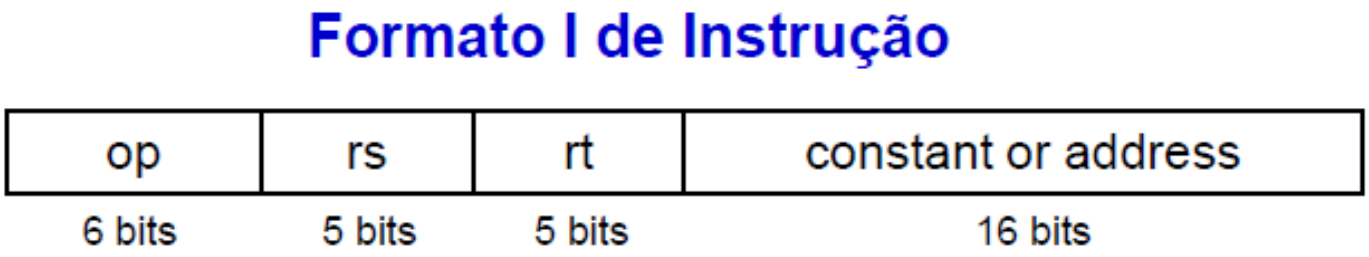
  - O compilador converter para complemento de 2.

Instruções Imediatas

- Essa instrução obedece ao princípio de projeto 3:
  - —"*Agilize os casos mais comuns*"
  - — Operandos com constante ocorrem com frequência
  - — Instruções com constantes são mais rápidas que instruções que precisam ler a constante da memória.

Formato da Instrução ADDI

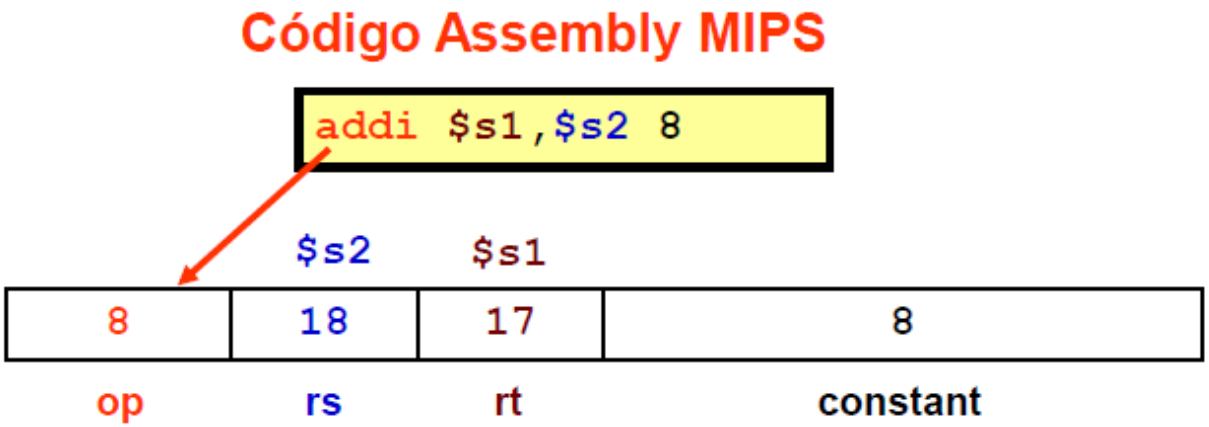
- Campos de uma instrução MIPS:
  - Instruções tipo-I (imediato) ou formato I



  - op: Operação básica da instrução (opcode)
  - rs: registrador fonte
  - rt: registrador destino
  - Constant: constante embutida na instrução

Representando ADDI na Máquina

- Número dos registradores
  - **$s0 - $s7: 16 - 23**
  - **$t0 - $t7 : 8-15**
  - **$t8-$t9 : 24-25**



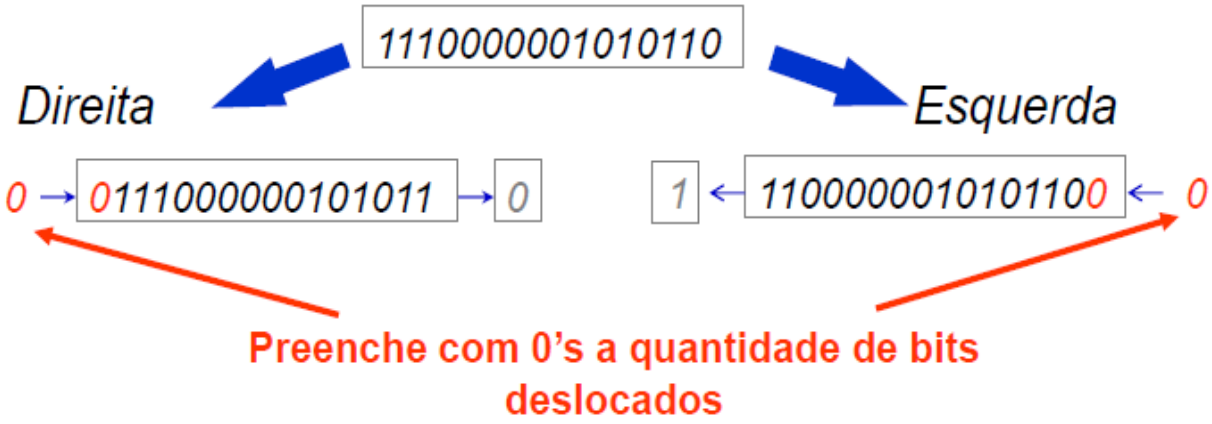
Operações Lógicas

- Permitem manipulação bit a bit dos dados
- Exemplo de uso: Examinar caracteres dentro de uma word.
- Úteis para extrair ou inserir um grupo de bits em uma palavra
  - Podem modificar o formato de um dado

Operações Logicas	Operadores C	Operadores Java	Instruções MIPS
Shift à esquerda	<<	<<	sll
Shift à direita	>>	>>>	srl
AND bit a bit	&	&	and, andi
OR bit a bit	|	|	or, ori
NOT bit a bit	~	~	Nor

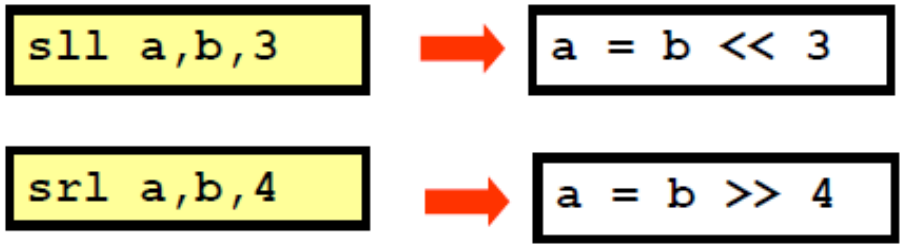
Operações Lógicas de Deslocamento (Shift)

- Afeta a localização dos bits em um dado
- Permite o deslocamento para esquerda ou direita de bits de um dado
- Insere grupo de bits no dado



Operações Lógicas de Deslocamento no MIPS

- Instruções de deslocamento no MIPS possuem 3 operandos:
  - destino, fonte, quantidade de bits deslocados



- Deslocamento para esquerda de i bits de um valor é equivalente a multiplicar o valor por 2^i
- Deslocamento para direita de i bits de um valor é equivalente a a dividir o valor por 2^i

Instruções para operações lógicas

- Sintaxe de uso dos operadores de deslocamento (sll, slr).
  - —sll $t2, $s0, 4; # reg $t2 = $s0 << 4 bits.
  - —Para $s0 = 9, teríamos $t2 = 144;

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

Multiplicação com SLL

- Multiplicação do valor por 4 (2^2)

g em $s1

Código C

```
g = g * 4;
```

Código Assembly MIPS

```
sll $s1, $s1, 2
```

g armazena valor 4 (100_2)

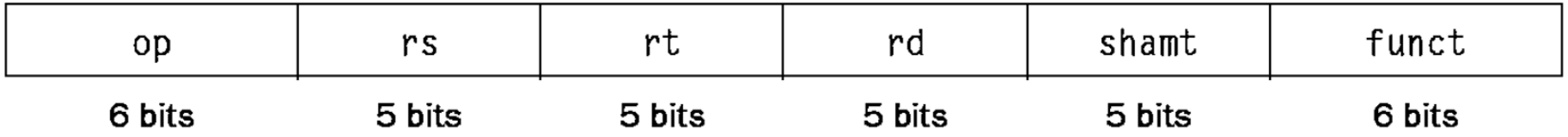
0000000000000100

g armazena valor 16 (10000_2)

0000000000010000

Formato da Instrução SLL e SRL

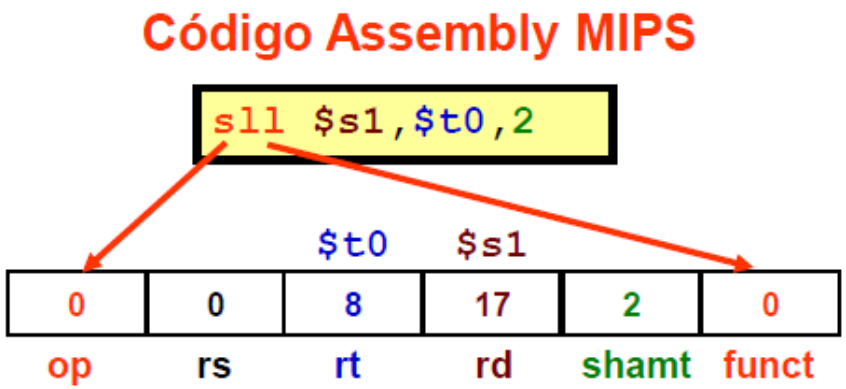
- Campos de uma instrução MIPS:
  - **Instruções tipo-R (de registrador) ou formato R**



  - op: Operação básica da instrução (opcode)
  - rs: não utilizado para **sll** e **srl**
  - rt: registrador que contém operando fonte
  - rd: registrador destino que contém resultado
  - shamt: “shift amount” (quantidade de deslocamento).
  - funct: Função que estende o opcode.

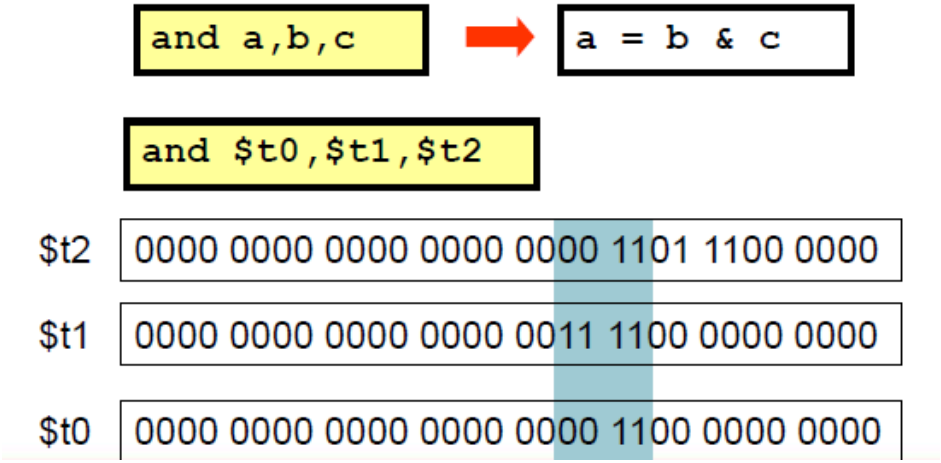
Representando SLL na Máquina

- Número dos registradores
  - **$s0 - $s7: 16 - 23**
  - **$t0 - $t7 : 8-15**
  - **$t8-$t9 : 24-25**



Outras Operações Lógicas

- AND ,OR, XOR, NOR, NOT
- Úteis para extrair grupos de bits
  - "Máscara" para encontrar padrões de disposição de bits
- No MIPS, possui 3 operandos como (ADD) e tem formato R



Outras Operações Lógicas

- AND, OR

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

```
and   $t1, $t2, $t3
```

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

```
or    $t1, $t2, $t3
```

Outras Operações Lógicas

- XOR, NOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

```
xor  $t1, $t2, $t3
```

Note the similarity to not-equals!

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

```
nor  $t1, $t2, $t3
```

Outras Operações Lógicas

- E a operação NOT ??
  - Mips não possui uma instrução not.

Outras Operações Lógicas

- E a operação NOT ??
  - MIPS não possui uma instrução not.
  - MIPS implementa como *A NOR 0*. Por que?

Outras Operações Lógicas

- E a operação NOT ??
  - MIPS não possui uma instrução not.
  - MIPS implementa como *A NOR 0*. Por que?
    - A NOR 0 = NOT (A OR 0) = NOT (A)

Outras Operações Lógicas

- E a operação NOT ??
  - Exemplo: nor $t0, $t1, $t3 # reg $t0 = ~ (reg $t1 | reg $t3)

Onde $t1 é

0000 0000 0000 0000 0011 1100 0000 0000

e $t3 é

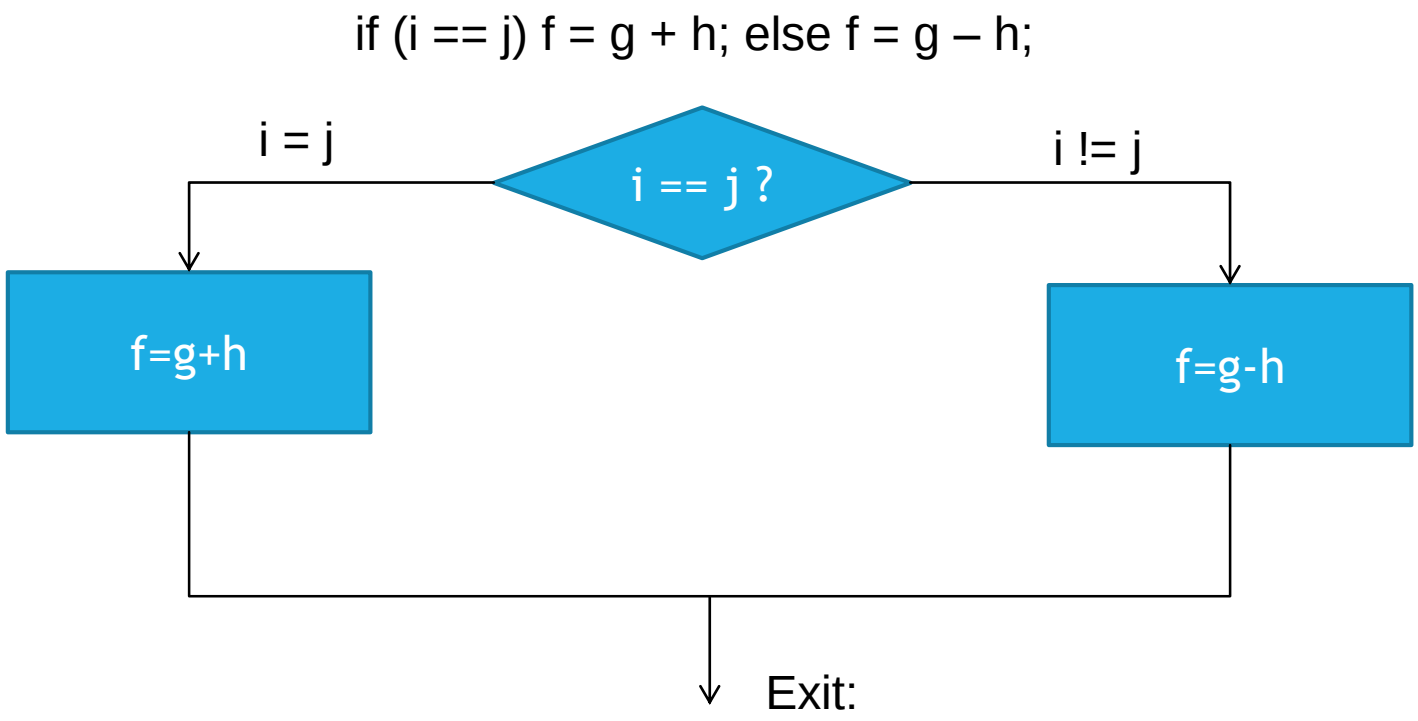
0000 0000 0000 0000 0000 0000 0000 0000

Resultado:

1111 1111 1111 1111 1100 0011 1111 1111

Instruções para tomada de decisão

- Altera a ***sequência*** de execução das instruções



Exemplos

Disposição das instruções em memória

- Em um ambiente computacional, instruções e dados são armazenados em memória.
- Cada instrução e dado possui um endereço associado.
- Algumas instruções dependem desses endereços. Quais?
  - Instruções de tomada de decisão
- Em assembly, podemos utilizar rótulos para representar tais endereços.

Disposição das instruções em memória

Address	Source
0x00400000	1: addi $0, $0, 100
0x00400004	2: add $0, $0, $1
0x00400008	3: sub $0, $0, $1
0x0040000c	4: addi $0, $0, 100
0x00400010	5: add $0, $0, $1
0x00400014	6: sub $0, $0, $1
0x00400018	7: L1: addi $0, $0, 100
0x0040001c	8: add $0, $0, $1
0x00400020	9: sub $0, $0, $1

Instruções para a tomada de decisões.

- beq (Branch equal)
  - Instrução utilizada quando desejamos comparar a igualdade de dois valores armazenados em registradores.
  - Sintaxe:
    - beq $r1, $r2, L1,
      - Onde $r1, $r2 são os registradores cujos valores armazenados vão ser comparados.
      - Se os valores são iguais, a sequência de execução pula para a instrução que possui o rótulo L1.

Instruções para a tomada de decisões.

- bnq (Branch not equal)
  - Instrução utilizada quando desejamos comparar se dois valores armazenados em registradores são diferentes.
  - Sintaxe:
    - bnq $r1, $r2, L1,
      - Onde $r1, $r2 são os registradores cujos valores armazenados vão ser comparados.
      - Se os valores são diferentes, a seqüência de execução pula para a instrução que possui o rótulo L1.
- beq e bnq são instruções conhecidas como ***desvios condicionais***.
- **Desvios condicionais** instruções que requerem a comparação de dois valores e a partir disso, realiza desvio a um novo endereço.

Desvio incondicional

- O assembly do Mips também dá suporte a instrução de desvio incondicional.
  - Basicamente, um desvio incondicional é uma instrução que ***sempre diz que o processador deverá seguir o desvio***.
  - A instrução para isso é a instrução j (jump).
  - Sintaxe:
    - *j rotulo # pula para a instrução precedida por*
    - *# rótulo*

Instruções para a tomada de decisões.

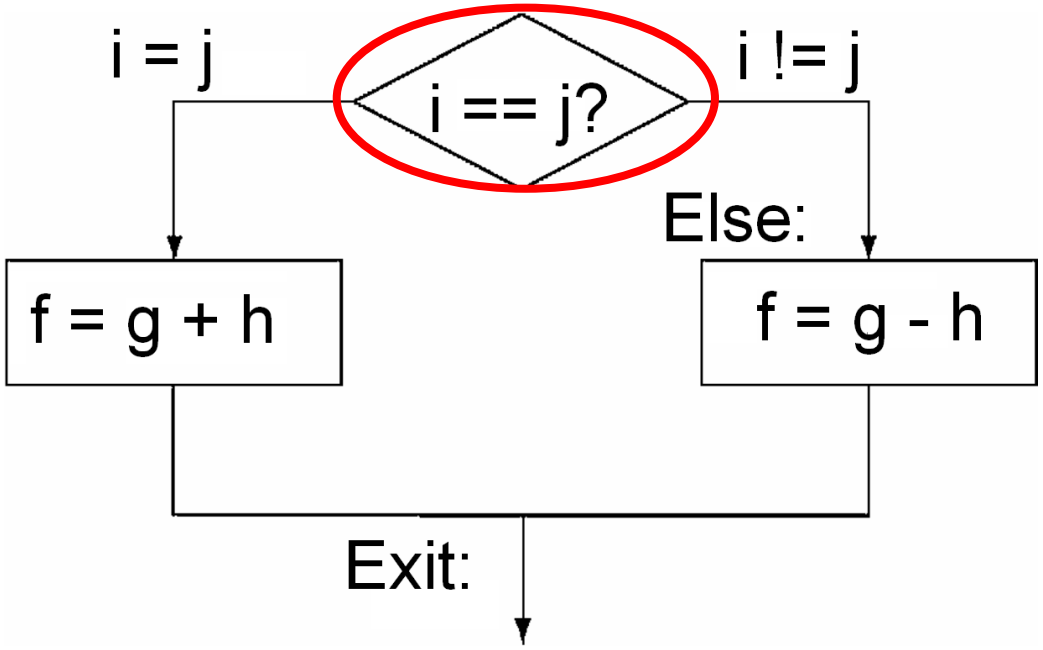
- Dado o seguinte código em Java, qual o assembly obtido?

```
if ( i == j)
  f = g + h;
else
  f = g – h;
```

f em $s0,
g em $s1,
h em $s2,
i em $s3,
j em $s4

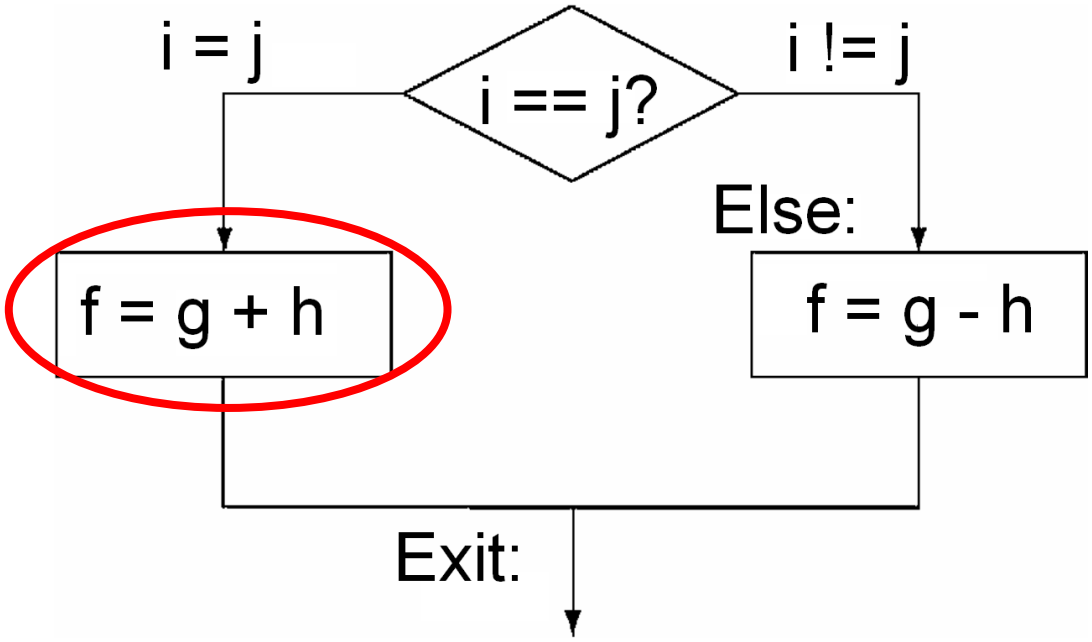
Instruções para a tomada de decisões.

```
Bne $s3, $s4, Else
```



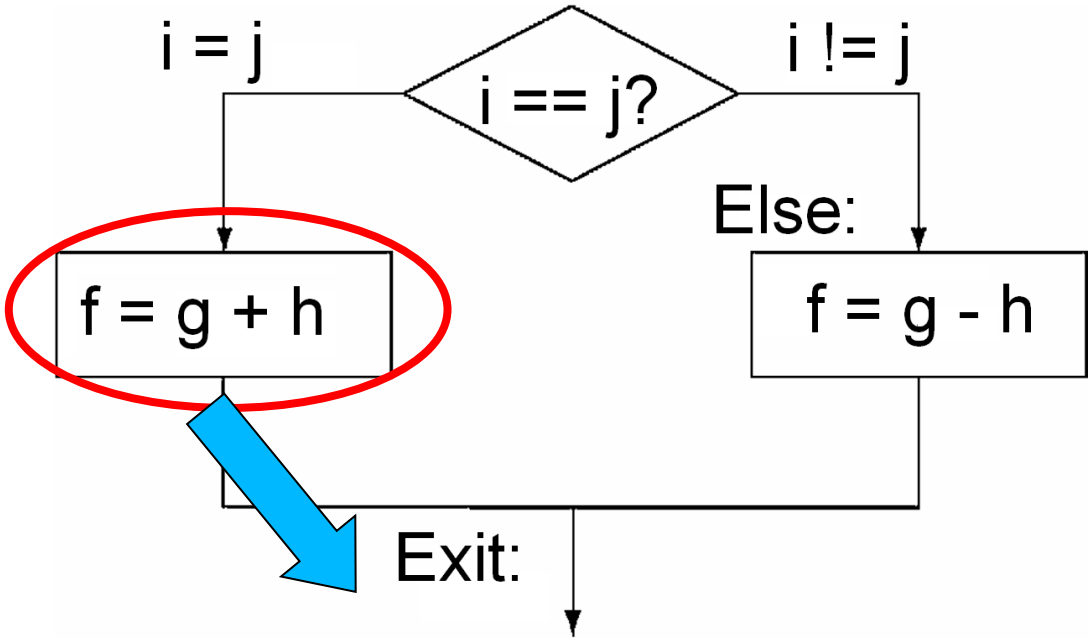
Instruções para a tomada de decisões.

```
Bne $s3, $s4, Else
add $s0, $s1, $s2
```



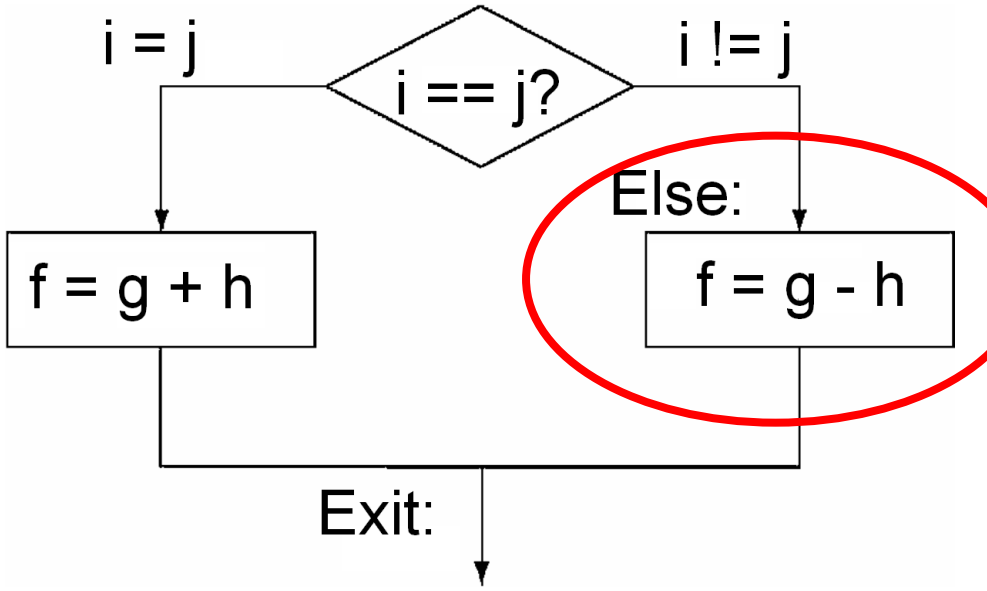
Instruções para a tomada de decisões.

```
Bne $s3, $s4, Else
add $s0, $s1, $s2
J Exit;
```



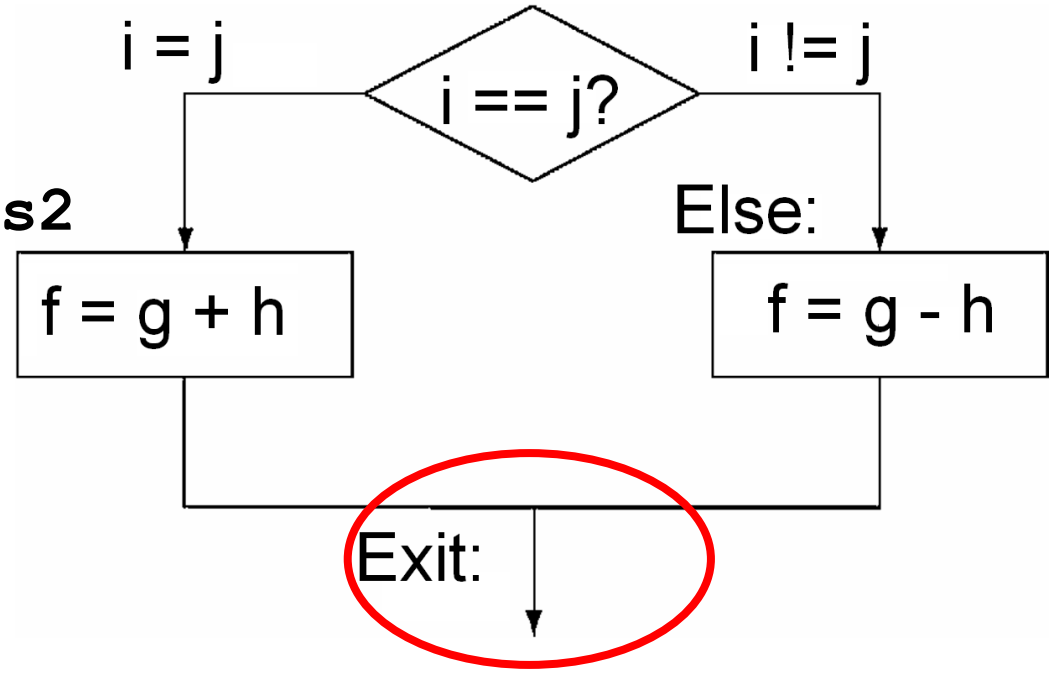
Instruções para a tomada de decisões.

```
Bne $s3, $s4, Else
add $s0, $s1, $s2
J Exit;
ELSE:sub $s0, $s1, $s2
```



Instruções para a tomada de decisões.

```
Bne $s3, $s4, Else
add $s0, $s1, $s2
J Exit;
ELSE:sub $s0, $s1, $s2
Exit:
```



Implementando Loops - while

- Qual seria o assembly correspondente?

Código C

```
while (save[i] == h)
  i += 1;
```

h em $s5,
i em $s3
e endereço base de save em $s6

Instruções para a tomada de decisões.

- 1. Realizar a leitura de save[i]

```
Loop: sll   $t1, $s3, 2        # $t1 = 4 * i
      add   $t1, $t1, $s6      # $t1 = endereço de save[i]
      lw    $t0, 0($t1)        # $t0 = save[i]
```

Instruções para a tomada de decisões.

- 1. Realizar a leitura de save[i]

```
Loop: sll    $t1, $s3, 2          # $t1 = 4 * i
      add    $t1, $t1, $s6        # $t1 = endereço de save[i]
      lw     $t0, 0($t1)          # $t0 = save[i]
```

- 2. Teste do loop, terminando se save[i] != k

```
Bne $t0, $s5, Exit     #vá para exit se save[i] != k
```

Instruções para a tomada de decisões.

- 1. Realizar a leitura de save[i]

```
Loop: sll     $t1, $s3, 2          # $t1 = 4 * i
      add  $t1, $t1, $s6           # $t1 = endereço de save[i]
      lw     $t0, 0($t1)             # $t0 = save[i]
```

- 2. Teste do loop, terminando se save[i] != k

```
Bne $t0, $s5, Exit     #vá para exit se save[i] != k
```

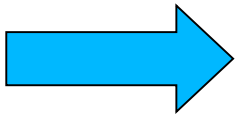
- 3. Senão, adiciona 1 a i e volta para o início.

```
add $s3, $s3, 1              # i = i + 1
j       Loop
Exit:
```

Instruções para a tomada de decisões.

- Código fonte

```
while (save[i] == k)
   i += 1;
```



Resultado Final

```
Loop: sll $t1, $s3, 2
add $t1, $t1, $s6
lw $t0, 0($t1)
Bne $t0, $s5, Exit
add $s3, $s3, 1
j       Loop
Exit:
```

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

```
loop: sll $t1, $s3, 2
```

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

```
loop: sll $t1, $s3, 2
add $t2, $s4, $t1
```

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

```
loop: sll $t1, $s3, 2
add $t2, $s4, $t1
lw $t3, 0($t2)
```

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

```
loop: sll $t1, $s3, 2
add $t2, $s4, $t1
lw $t3, 0($t2)
add $t4, $t3, 1
```

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

```
loop: sll $t1, $s3, 2
add $t2, $s4, $t1
lw $t3, 0($t2)
add $t4, $t3, 1
sw $t4, 0($t2)
```

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

```
loop: sll $t1, $s3, 2
add $t2, $s4, $t1
lw $t3, 0($t2)
add $t4, $t3, 1
sw $t4, 0($t2)
add $s3, $s3, 1
```

Implementando Loops - for

- Converta o seguinte trecho de código

```
Int i
Int A[10]
for(i=0;i<10;i++){
   A[i] = A[i] + 1;
}
```

i em $s3 e está com o valor 0
e endereço base de A em $s4
10 está em $t0

```
loop: sll $t1, $s3, 2
add $t2, $s4, $t1
lw $t3, 0($t2)
add $t4, $t3, 1
sw $t4, 0($t2)
add $s3, $s3, 1
bne $s3, $t0, loop
```

Instruções para a tomada de decisões.

- Estas sequências de instruções sem desvios (exceto, possivelmente, no final) e sem destinos de desvio ou rótulos de desvio (exceto, possivelmente, no início) são conhecidas como ***blocos básicos***.

- Blocos básicos são muito importante e constituem uma das etapas do projeto de compiladores, basicamente, através deles, podemos realizar algumas otimizações no programa.

Instruções para a tomada de decisões.

- As instruções slt e slti
  - A instrução slt é usada quando desejamos verificar se o valor armazenado em um registrador é menor que o valor armazenado em um outro registrador.
  - A instrução slti é usada quando desejamos verificar se o valor armazenado em um registrador é menor que o valor de uma constante literal.

Instruções para a tomada de decisões.

- Sintaxe de uso
  - slt $t1, $r1, $r2
    - Basicamente, se o valor em $r1 for menor que o valor em $r2, $t1 recebe o valor 1. Caso contrário, $t1 recebe o valor 0.
  - slti $t1, $r1, constante
    - Basicamente, se o valor em $r1 for menor que o valor da constante literal, $t1 recebe o valor 1. Caso contrário, $t1 recebe o valor 0.

Instruções para a tomada de decisões.

- Nenhuma instrução de desvio para <, >=, … ?????
  - Combinar em uma só instrução branch e comparações (>, <, >= …) , requer mais trabalho por instrução
  - Tornando o processamento mais lento

Instruções para a tomada de decisões.

- **slt** pode ser utilizada junto com **beq, bne**

```
slt $t0,$s1,$s2
bne $t0,$zero, L1
```

se ($s1 < $s2) desvie para L1

- MIPS possui registrador que armazena valor 0
  - **$zero**

- slt, slti, beq, bne e o valor fixo 0 (sempre à disposição com a leitura do registrador $zero)
  - podemos criar todas as condições relativas: igual, diferente, menor que, menor ou igual, maior que , maior ou igual.

Instruções para a tomada de decisões.

- Exemplo
  - Implemente a seguinte instrução assembly
  - if (i < j) then
    - i = j;

i em $s0,
j em $s1,

Instruções para a tomada de decisões.

- Exemplo
  - Implemente a seguinte instrução assembly
  - if (i < j) then
    - i = j;

  - slt $t0, $s0, $s1;
  - beq $t0, $zero, exit;
  - add $s0, $s1, $zero;
  - exit:

Atividades

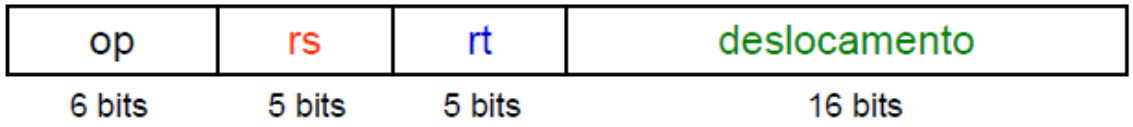
- Codifique o seguinte programa em assembly:
  - if (x < y) then
    for (int i = 0; i < 10; i++)
        save[i] = i * 2;

Mais Sobre Branchs no MIPS

- O operando relativo ao rótulo nas instruções de branch corresponde na verdade ao deslocamento em relação ao endereço da instrução contida no **PC (Program Counter)**
  - **PC já incrementado de 4!**
  - **PC = PC + (deslocamento * 4) se reg1 == reg2**

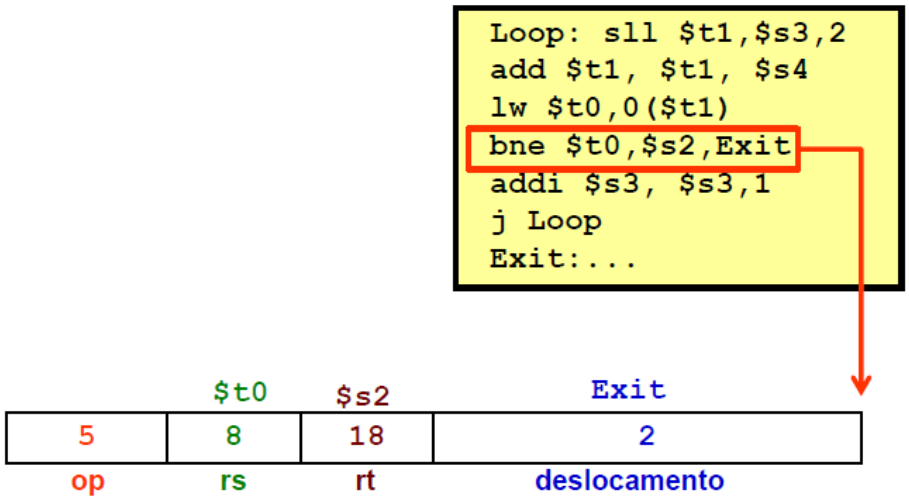
```
beq rs,rt,deslocamento
```

Formato I de Instrução



Representando BNE na Máquina

- Número dos registradores
  - **$s0 - $s7: 16 - 23**
  - **$t0 - $t7 : 8-15**
  - **$t8-$t9 : 24-25**

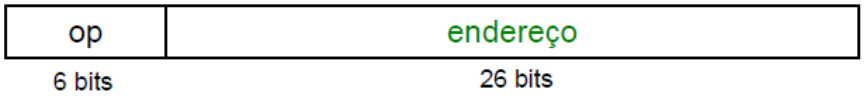


Mais Sobre Jumps no MIPS

- O operando relativo ao label nas instruções de jumps corresponde na verdade ao endereço (número) da instrução a ser executada
  - **PC = endereço**

```
j endereco
```

Formato J de Instrução



Suporte a chamadas de função em hardware

- Sub-rotinas são utilizadas para estruturar um programa
  - Facilita o entendimento
  - Aumenta o reuso de código
  - Exs: Procedimentos, funções e métodos
- Chamada de sub-rotinas faz com que programa execute as instruções contidas na subrotina
- Ao término da execução de uma subrotina, computador deve executar instrução seguinte à chamada de subrotina

Suporte a chamadas de função em hardware

- Para a execução de uma sub rotina deve-se:
  1) Colocar os parâmetros em um local onde a sub rotina possa acessá-los
  2) Transferir o controle a sub rotina
  3) Adquirir os recursos necessários ao sub rotina
  4) Executar a tarefa
  5) Colocar o resultado em um local onde o programa possa acessá-lo
  6) Retornar o controle ao ponto onde a sub rotina foi chamado

Suporte a chamadas de função em hardware

- Para a execução de uma sub rotina deve-se:
  1) ***Colocar os parâmetros em um local onde a sub rotina possa acessá-los***
  2) Transferir o controle a sub rotina
  3) Adquirir os recursos necessários ao sub rotina
  4) Executar a tarefa
  5) Colocar o resultado em um local onde o programa possa acessá-lo
  6) Retornar o controle ao ponto onde a sub rotina foi chamado

Passagem de Argumentos

Ling. alto nível

```
int media(int w, int x, int y, int z) {
...
 (corpo da função)
...
}
/* Programa principal */
int main() {
  int m;
  m = media(2,3,6,2);
....
}
```

Passando Argumentos

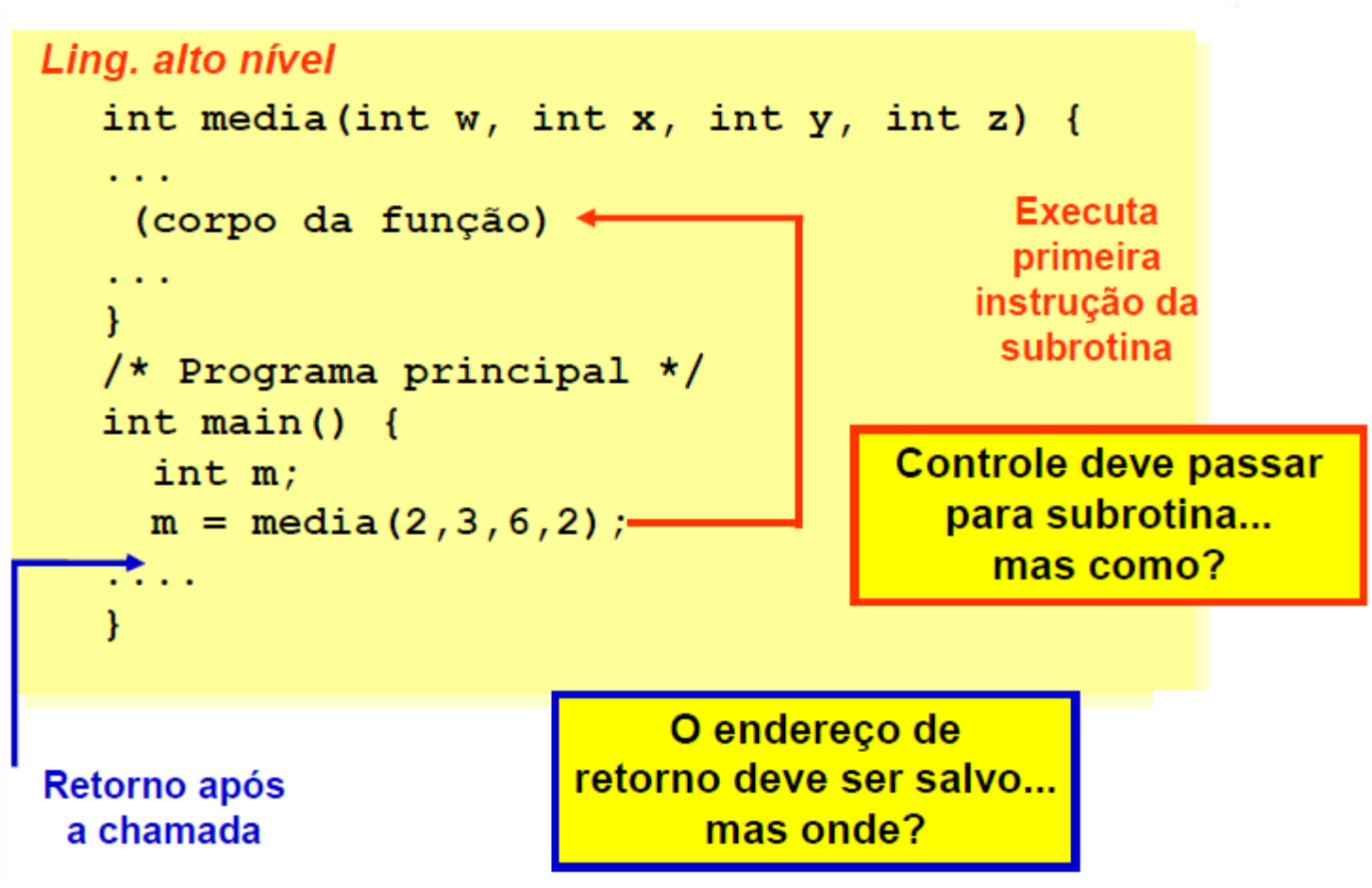
No MIPS, 4 registradores são destinados para armazenar argumentos

- a0 - $a3 – números 4 a 7

Suporte a chamadas de função em hardware

- Para a execução de uma sub rotina deve-se:
  1) Colocar os parâmetros em um local onde a sub rotina possa acessá-los
  2) ***Transferir o controle a sub rotina***
  3) Adquirir os recursos necessários ao sub rotina
  4) Executar a tarefa
  5) Colocar o resultado em um local onde o programa possa acessá-lo
  6) Retornar o controle ao ponto onde a sub rotina foi chamado

Tranferência de Controle Para Subrotina



Instrução Para Chamada de Subrotinas

- MIPS oferece uma instrução para fazer a chamada a subrotina –
  - **Jump And Link**
- Instrução para chamar a subrotina possui um operando:
  - Label da subrotina

```
jal label
```

- Instrução pula para endereço inicial da subrotina e salva endereço de retorno (instrução após chamada)
  - $ra – *return address* (número 31)– registrador que armazena endereço de retorno
  - Armazena **PC + 4**

Suporte a chamadas de função em hardware

- Para a execução de uma sub rotina deve-se:
  1) Colocar os parâmetros em um local onde a sub rotina possa acessá-los
  2) Transferir o controle a sub rotina
  3) *Adquirir os recursos necessários ao sub rotina*
  4) *Executar a tarefa*
  5) *Colocar o resultado em um local onde o programa possa acessá-lo*
  6) Retornar o controle ao ponto onde a sub rotina foi chamado

Armazenamento e Retorno de Valores

- Variáveis podem ser salvas em registradores disponíveis
- No MIPS, 2 registradores para valores retornados
  - **$v0 - $v1 - números 2 a 3**

Ling. alto nível

```
int media(int w, int x, int y, int z) {
  int result;
  result = (w + x + y + z)/4;
  return result;
}
/* Programa principal */
int main() {
  int m;
  m = media(2,3,6,2);
....
}
```

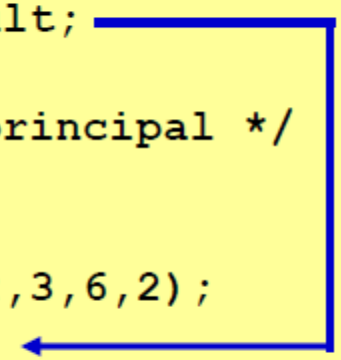
Variável local

Retorna valor

Retorno da Subrotina

Ling. alto nível

```
int media(int w, int x, int y, int z) {
  int result;
  result = (w + x + y + z)/4;
  return result;
}
/* Programa principal */
int main() {
  int m;
  m = media(2,3,6,2);
....
}
```



Controle deve voltar à instrução após chamada

Instrução Para Retorno de Subrotinas

- MIPS oferece uma instrução que pode ser utilzado para retornar da subrotina

Jump **R**egister

- Instrução para retornar da subrotina possui um operando:
  - Registrador que contém um endereço

```
jr registrador
```

- Instrução pula para endereço armazenado no registrador
  - No caso de retorno de subrotina, o registrador deve ser o $ra

```
jr $ra
```

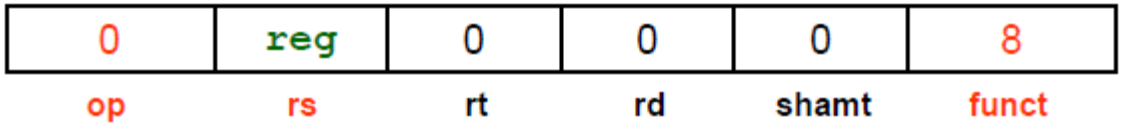
Formato de Instruções

```
jal label
```

Formato J de Instrução


```
jr reg
```

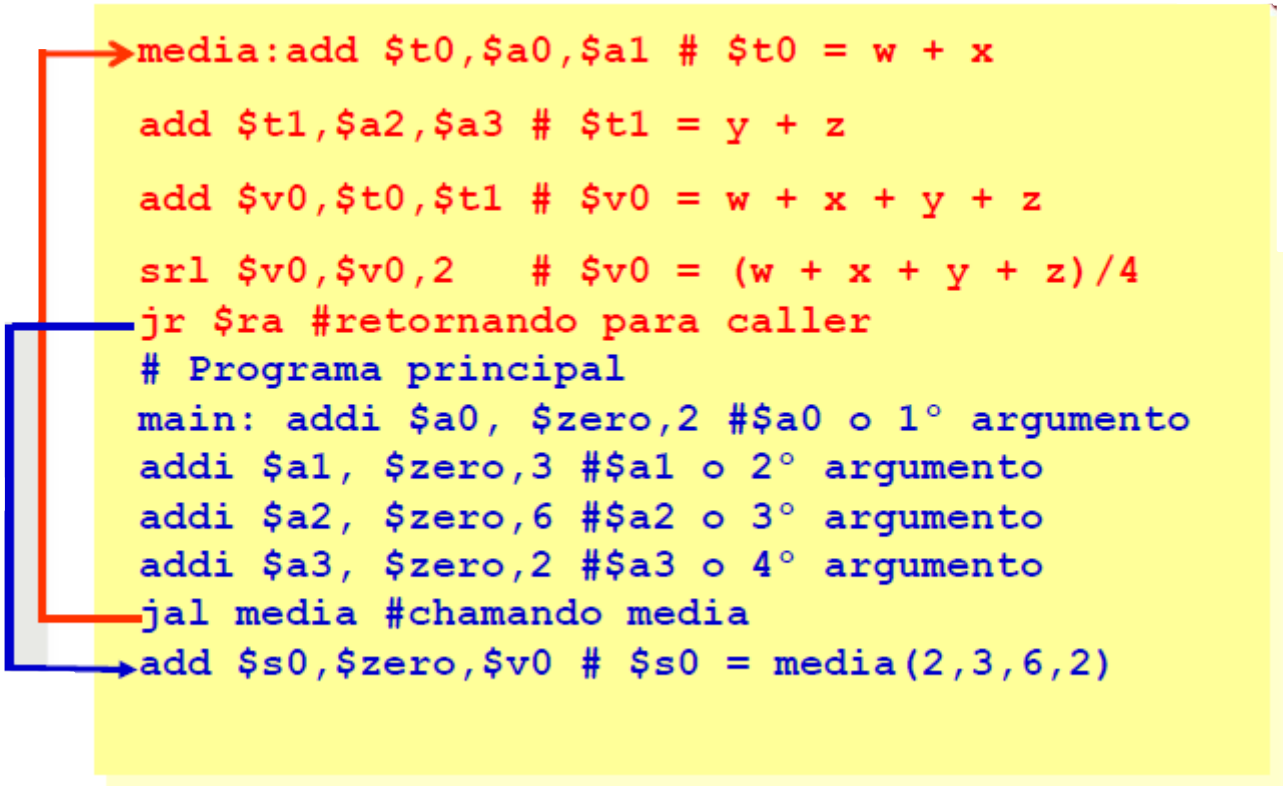
Formato R de Instrução



Resumo de chamada de subrotina

- O programa que chama coloca os valores de parâmetro em $a0 - $a3
- utiliza a instrução **jal X** para desviar para o procedimento X
- O procedimento X então:
  - Realiza os cálculos
  - Coloca os resultados em **$v0-$v1** e
  - Retorna o controle para o programa que o chamou usando **jr $ra**.

Código Assembly



```
media:add $t0,$a0,$a1 # $t0 = w + x
 add $t1,$a2,$a3 # $t1 = y + z
 add $v0,$t0,$t1 # $v0 = w + x + y + z
 srl $v0,$v0,2   # $v0 = (w + x + y + z)/4
jr $ra #retornando para caller
# Programa principal
main: addi $a0, $zero,2 #$a0 o 1° argumento
addi $a1, $zero,3 #$a1 o 2° argumento
addi $a2, $zero,6 #$a2 o 3° argumento
addi $a3, $zero,2 #$a3 o 4° argumento
jal media #chamando media
add $s0,$zero,$v0 # $s0 = media(2,3,6,2)
```

Modos de Endereçamento do MIPS

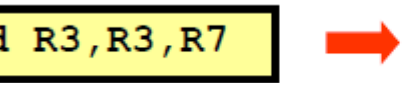
- Modo de endereçamento se refere às maneiras em que instruções de uma arquitetura especificam a localização do operando
  - Onde e como pode ser acessado

- No MIPS, operandos podem estar em:
  - Registradores
  - Memória
  - Na própria instrução

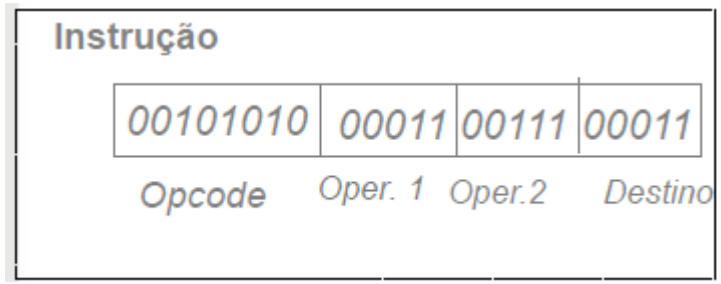
Endereçamento de Registrador

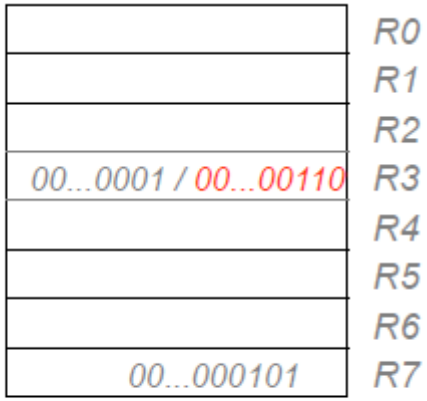
- **Operações aritméticas:**
  - O operando está em um registrador e a instrução contem o número do registrador

```
add R3,R3,R7
```



```
R3 <- R3 + R7
```

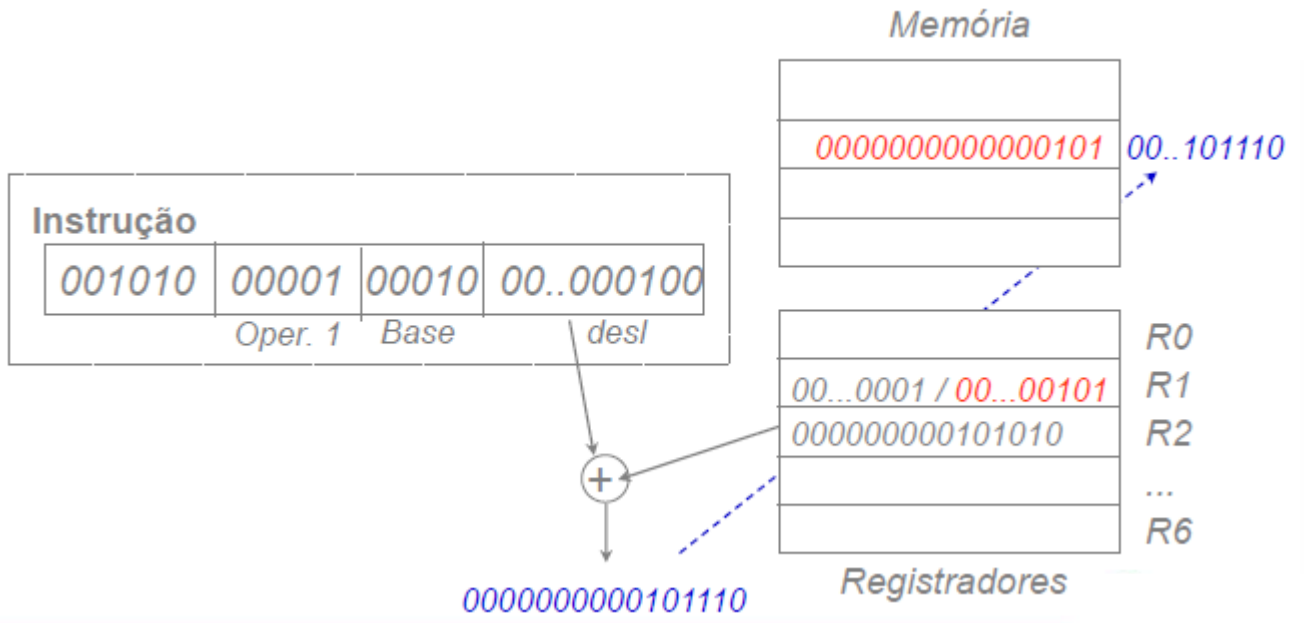




Endereçamento Base

- **Instruções de acesso à memória:**
  - Instrução:deslocamento
  - Registrador de base:end- inicial

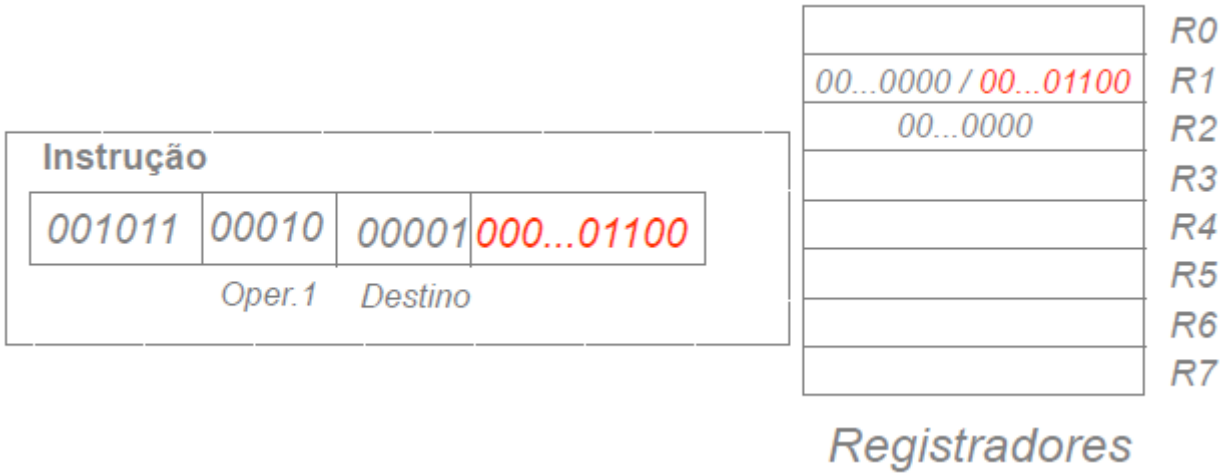
```
lw R1,desl(R2)
```



Endereçamento imediato

- **Operações aritméticas e de comparação:**
  - O operando é especificado na instrução

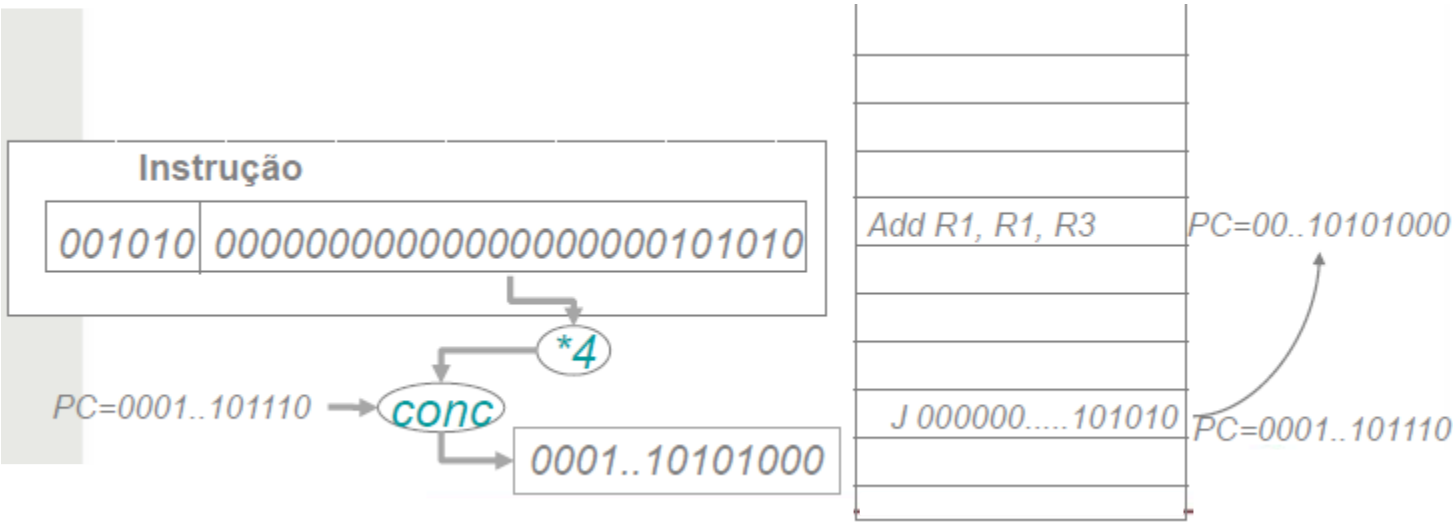
```
addi R1,R2, 12
```



Endereçamento (Pseudo)Direto

- **Instrução de Desvio Incondicional:**
  - o (pseudo)endereço da próxima instrução (endereço da palavra) é especificado na instrução
  - 4 bits mais significativos do PC são concatenados ao endereço especificado multiplicado por 4

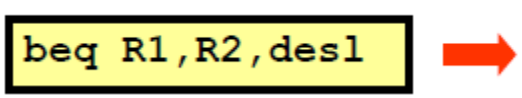
```
j endereco  ➡  PC <- conc(PC(4 bits), endereco* 4)
```



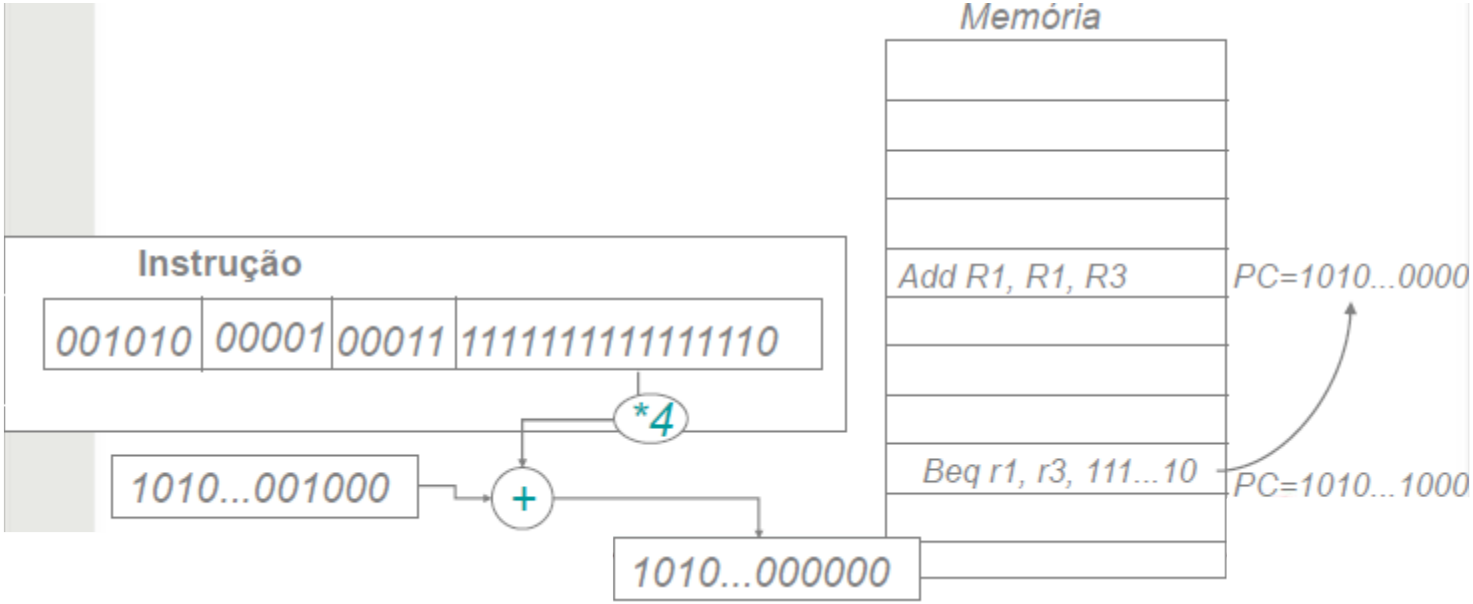
Endereçamento Relativo a PC

- **Instrução de Desvio Condicional:**
  - o número de instruções a serem puladas a partir da
  - instrução é especificado na instrução

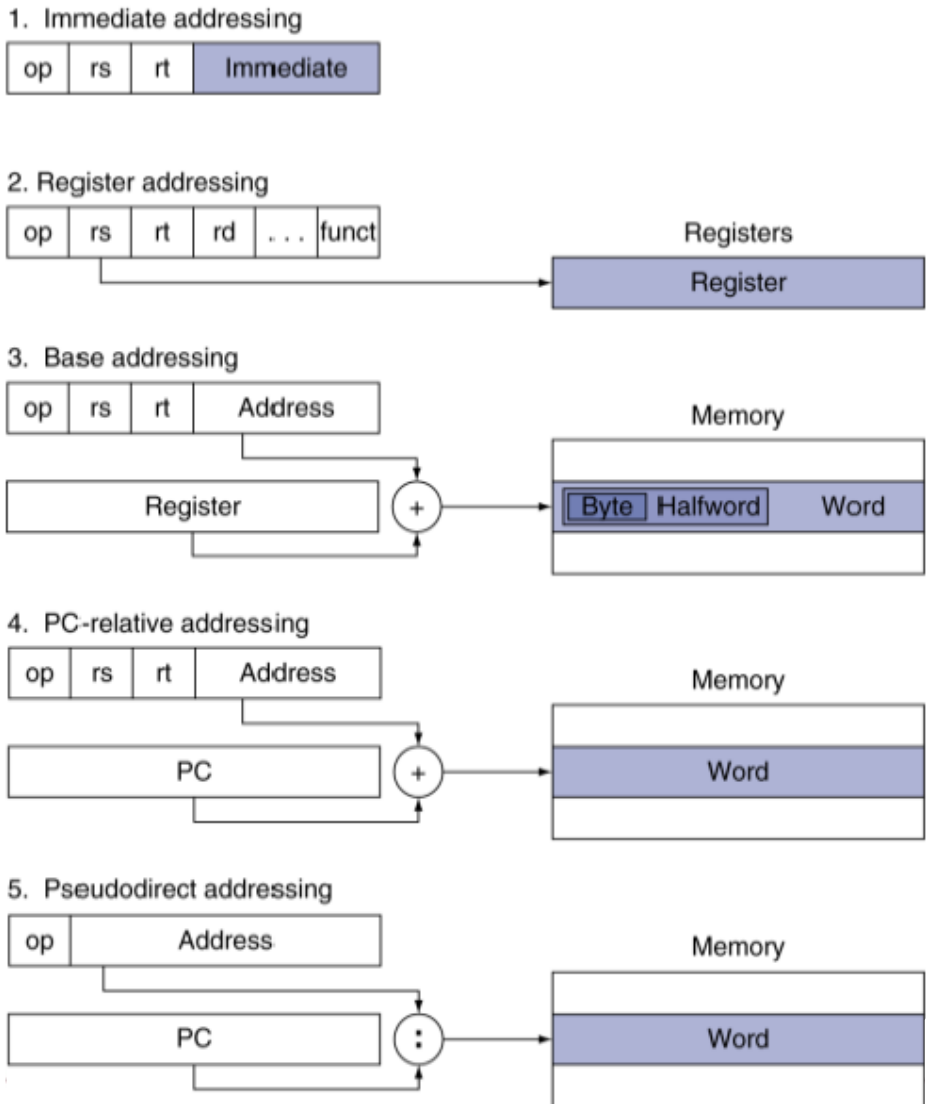
```
beq R1,R2,desl
```



```
PC <- PC + desl * 4
```



Resumo dos Modos de Endereçamento do MIPS



Resumo de Instruções do MIPS

Instrução	Descrição
nop	No operation
lw reg, end(reg_base)	reg. = mem (reg_base+end)
sw reg, end(reg_base)	Mem(reg_base+end) = reg
add regi, regj,regk	Regi. <- Regj. + Regk
sub regi, regj, regk	Regi. <- Regj. – Regk
and regi, regj,regk	Regi. <- Regj. and Regk
xor regi, regj, regk	Regi = regj xor regk
srl regd, regs, n	Desloca regs para direita n vezes sem preservar sinal, armazena valor deslocado em regd .
sra regd, regs, n	Desloca regs para dir. n vezes preservando o sinal, armazena valor deslocado em regd.
sll regd, regs, n	Desloca regs para esquerda n vezes, armazena valor deslocado em regd.
ror regd, regs, n	Rotaciona regs para direita n vezes, armazena valor deslocado em regd.
rol regd, regs, n	Rotaciona regs para esquerda n vezes, armazena valor deslocado em regd.
beq regi, regj, desl	PC = PC + desl*4 se regi = regj
bne regi, regj, desl	PC = PC + desl *4 se regi <> regj
slt regi, regj, regk	Regi =1 se regj < regk senão regi=0
j end	Desvio para end
jr reg	Pc = reg
jal end	Reg31 = pc, pc = endereço
break	Para a execução do programa