**Universidade de São Paulo**
**Escola de Artes, Ciências e Humanidades**

# ACH2034: Organização e Arquitetura de Computadores I

**Prof.ª Gisele da Silva Craveiro**
**Semestre 2024/1**
**EP OAC I - Relatório**

**Pedro Henrique Resnitzky Barbedo**
(N° USP: 14657691)
**Aline (Eduardo) Crispim de Moraes**
(N° USP: 14567051)

# Organização e Arquitetura MIPS

A arquitetura MIPS (*Microprocessor without Interlocked Pipeline Stages*) é uma arquitetura de conjunto de instruções desenvolvida para CPUs. Ela é conhecida por sua simplicidade e eficiência, sendo amplamente utilizada em sistemas embarcados, supercomputadores e ambientes educacionais.

Sobre as escolhas de organização e arquitetura:

- Conta com um conjunto de 32 registradores;
- As instruções têm tamanho fixo de 32 bits;
- Segue os formatos padronizados de instruções tipo R, I e J;
- Inclui instruções de operações aritméticas e lógicas básicas, de acesso à memória, de controle de fluxo e outras comuns na programação assembly;
- Utiliza endereçamento baseado em registradores.

Além disso, a arquitetura conta com algumas pseudo-instruções, ou seja, instruções que são na verdade um conjunto de execução de outras instruções, como, por exemplo, a pseudo-instrução bgt, que é uma junção das instruções addi, slt e bne.

Dentre os 32 registradores, existem 11 tipos, cada um com um propósito diferente dentro do código:

| Registradores | | |
|---|---|---|
| Tipo | Número | Uso |
| $zero | 0 | constante 0 |
| $at | 1 | reservado para o assembler |
| $v0-$v1 | 2-3 | avaliação de expressão e resultados de funções |
| $a0-$a3 | 4-7 | argumentos |
| $t0-$t9 | 8-15, 24-25 | temporários (não preservados) |
| $s0-$s7 | 16-23 | temporários salvos (preservados) |
| $k0-$k1 | 26-27 | reservado para o sistema operacional |
| $gp | 28 | ponteiro para a área global |
| $sp | 29 | ponteiro para a pilha |
| $fp | 30 | ponteiro para a pilha de chamadas de função |
| $ra | 31 | ponteiro para o endereço de retorno |

A numeração dos registradores presente nessa tabela será principalmente importante para a seção de explicação das micro-operações.

# Descrição do problema e código alto nível

Esse relatório analisa o problema número 11 da Lista de Problemas passada para o EP.

O problema envolve a criação de uma função "valor" que seja capaz de, a partir de um inteiro $k \geq 1$, calcular o valor de $F_k$ e $G_k$, sendo essas as respectivas funções:

$$\begin{cases} F_1 = 2 \\ F_2 = 1 \\ F_k = 2 * F_{k-1} + G_{k-2} \quad k \geq 3 \end{cases} \quad \begin{cases} G_1 = 1 \\ G_2 = 2 \\ G_k = G_{k-1} + 3 * F_{k-2} \quad k \geq 3 \end{cases} \tag{1}$$

Por serem duas funções recursivas que dependem uma da outra, a solução mais fácil encontrada para o código de alto nível foi criar uma função recursiva que implementa a recursão duas vezes.

A seguir, o código em C desenvolvido pela dupla:

```c
#include <stdio.h>
#include <stdlib.h>

int* valor(int k){

  int *valores = (int *)malloc(2 * sizeof(int));

  if (k == 1){

    valores[0] = 2;
    valores[1] = 1;
    return valores;
  }

  if (k == 2){

    valores[0] = 1;
    valores[1] = 2;
    return valores;
  }

  int *valores_anteriores1 = valor(k - 1);
```

```c
  int *valores_anteriores2 = valor(k - 2);

  valores[0] = 2 * valores_anteriores1[0] + valores_anteriores2
      [1];
  valores[1] = valores_anteriores1[1] + 3 * valores_anteriores2
      [0];

  return valores;
}

int main(){

    int k;

    scanf("%d", &k);

    int *res = valor(k);
    printf("%d %d\n", res[0], res[1]);
}
```

codigoC.c

# Código em Assembly desenvolvido

Tanto para o código em alto nível quanto para o código em assembly, decidimos não utilizar o .data e nem nenhum valor pré-estabelecido, mas sim uma entrada de variável $k$ decidida pelo usuário.

A seguir, o código em Assembly desenvolvido pela dupla:

```
.text
.globl main
main:
    li $v0, 5               # syscall de scan int
    syscall
    move $a1, $v0           # movimentacao do dado de v0 para a1
    li $a0, 8               # preparacao do syscall de alocacao
        de 8 bits
    li $v0, 9               # syscall de alocacao de 8 bits em
        v0
    syscall
    move $s0, $v0           # movimentacao do array alocado em
        v0 para s0
    jal valor               # jump and link pra funcao 'valor'
    lw $a0, 0($s0)          # load em a0 do primeiro elemento do
         array s0 (resultado de F)
```

```
13      li $v0, 1               # syscall de print int
14      syscall
15      li $a0, 32              # load imediato de 32 em a0, ASCII
            do caractere de espaco
16      li $v0, 11              # syscall de print char
17      syscall
18      lw $a0, 4($s0)          # load em a0 do segundo elemento do
            array em s0 (resultado de G)
19      li $v0, 1               # syscall de print int
20      syscall
21      li $a0, 10              # load imediato de 10 em a0, ASCII
            de nova linha
22      li $v0, 11              # syscall de print char
23      syscall
24      li $v0, 10              # syscall de fim de execucao
25      syscall
26
27  valor:
28      bgt $a1, 2, valor_maior     # branch se a1 > 2
29      beq $a1, 1, valor_um        # branch se a1 == 1
30      beq $a1, 2, valor_dois      # branch se a1 == 2
31
32  valor_um:
33      li $t0, 2               # load imediato de 2 em t0
34      sw $t0, 0($s0)      # guarda o resultado de F no primeiro
            indice do array s0
35      li $t0, 1               # load imediato de 1 em t0
36      sw $t0, 4($s0)      # guarda o resultado de G no segundo
            indice do array s0
37      j valor_fim             # jump para o fim da funcao
38
39  valor_dois:
40      li $t0, 1               # load imediato de 1 em t0
41      sw $t0, 0($s0)      # guarda o resultado de F no primeiro
            indice do array s0
42      li $t0, 2               # load imediato de 2 em t0
43      sw $t0, 4($s0)      # guarda o resultado de G no segundo
            indice do array s0
44      j valor_fim             # jump para o fim da funcao
45
46  valor_maior:            # funcao de inicializacao de pilha
47      subu $sp, $sp, 16       # abre 4 espacos na pilha
48      li $t0, 2               # load imediato de 2 em t0
49      li $t1, 2               # load imediato de 2 em t1
50      li $t2, 1               # load imediato de 1 em t2
51      li $t3, 1               # load imediato de 1 em t3
52
```

```
53      sw $t3, 0($sp)          # guarda t3 no inicio da pilha
54      sw $t2, 4($sp)          # guarda t2 no segundo espaco da
            pilha
55      sw $t1, 8($sp)          # guarda t1 no terceiro espaco da
            pilha
56      sw $t0, 12($sp)         # guarda t0 no quarto espaco da
            pilha
57
58      li $a2, 3               # load imediato de 3 em a2 (contador
            )
59      move $a3, $a1       # movimento do dado de a1 para a3
60      j valor_loop            # jump funcao valor_loop
61
62  valor_loop:
63      lw $t0, 0($sp)          # load do topo da pilha em t0
64      addu $sp, $sp, 4        # diminui a pilha em 1 elemento
65      lw $t1, 0($sp)          # load do topo da pilha em t1
66      addu $sp, $sp, 4        # diminui a pilha em 1 elemento
67      mul $t2, $t0, 2         # guarda o resultado da
            multiplicacao de (2 * t0) em t2
68      addu $t2, $t2, $t1          # guarda o resultado da adicao
            de (t2 + t1) em t2
69
70      # t0 = f(n-1) / t2 = f(n)
71
72      lw $t3, 0($sp)          # load do topo da pilha em t3
73      addu $sp, $sp, 4        # diminui a pilha em 1 elemento
74      lw $t4, 0($sp)          # load do topo da pilha em t4
75      addu $sp, $sp, 4        # diminui a pilha em 1 elemento
76      mul $t5, $t4, 3         # guarda o resultado da
            multiplicacao de (3 * t4) em t5
77      addu $t5, $t5, $t3          # guarda o resultado da adicao
            de (t5 + t3) em t5
78
79      # t3 = g(n-1) / t5 = g(n)
80
81      subu $sp, $sp, 16           # abre 4 espacos na pilha
82      sw $t2, 0($sp)          # guarda t2 no topo da pilha
83      sw $t3, 4($sp)          # guarda t2 no segundo espaco da
            pilha
84      sw $t5, 8($sp)          # guarda t2 no terceiro espaco da
            pilha
85      sw $t0, 12($sp)         # guarda t2 no quarto espaco da
            pilha
86
87      addi $a2, $a2, 1            # adicao de 1 ao contador
88
```

```
89      ble $a2, $a3, valor_loop    # branch se a2 <= a3, caso
            verdadeiro, volta para valor_loop
90
91      sw $t2, 0($s0)      # guarda o resultado de F no primeiro
            indice do array s0
92      sw $t5, 4($s0)      # guarda o resultado de G no segundo
            indice do array s0
93
94      j valor_fim             # jump para o fim da fun   o
95
96  valor_fim:
97      jr $ra              # jump register para o endere o de
            retorno
```

codigoAssembly.asm

Como é possível observar, diferentemente do código em alto nível, em Assembly optamos por usar a pilha para deixar o programa muito mais simples, sendo possível criar a função "valor" que implementa apenas um loop e algumas branches.

Para chegar nisso, nos aproveitamos da liberdade que os registradores disponibilizam e utilizamos da pilha $sp para guardar a todo momento os resultados dos dois últimos loops executados. Dessa forma, poupamos a necessidade da execução de duas recursões no código e obtemos o mesmo resultado do código em alto nível.

# Explicação detalhada das instruções utilizadas no código

Nesta seção, as instruções utilizadas no código em Assembly MIPS serão explicadas a nível de micro-operações. As instruções serão explicadas na ordem em que aparecem no código, e vale notar que instruções repetidas não serão explicadas de forma duplicada. Contudo, essa última regra não conta para instruções de jump e branch.

Além disso, é importante ressaltar que todas as micro-operações realizadas dentro do ciclo de busca

- **li $v0, 5**
  Essa pseudo-instrução é transformada em uma instrução real de máquina equivalente: **addiu $2, $0, 5** (com $X indicando o endereço do registrador de número X)

  - Ciclo de Busca:
    * MAR ← PC
    * MBR ← Memória[MAR]
    * PC ← PC+4
    * IR ← MBR
  - Decodificação:
    * A instrução que está em IR é decodificada como Tipo R:
    * RS ← IR[RS] (nesse caso, $0)
    * RT ← IR[RT] (nesse caso, o imediato 5)
    * RD ← IR[RD] (nesse caso, $2)
  - Ciclo de Execução:
    * ALUInput1 ← Registrador[RS]
    * RT imediato passa por um extensor de sinal
    * ALUInput2 ← Registrador[RT]
    * ALUResult ← ALUInput1 + ALUInput2
    * Registrador[RD] ← ALUResult

- **move $a1, $v0**
  Essa pseudo-instrução é transformada em uma instrução real de máquina equivalente: **addu $5, $0, $2**
  - Ciclo de Busca:
    * MAR ← PC
    * MBR ← Memória[MAR]
    * PC ← PC+4
    * IR ← MBR
  - Decodificação:
    * A instrução que está em IR é decodificada como Tipo R:
    * RS ← IR[RS] (nesse caso, $0)
    * RT ← IR[RT] (nesse caso, $2)
    * RD ← IR[RD] (nesse caso, $5)
  - Ciclo de Execução:
    * ALUInput1 ← Registrador[RS]

    - ALUInput2 ← Registrador[RT]
    - ALUResult ← ALUInput1 + ALUInput2
    - Registrador[RD] ← ALUResult

- **jal valor**
  - Ciclo de Busca:
    - MAR ← PC
    - MBR ← Memória[MAR]
    - PC ← PC+4
    - IR ← MBR
  - Decodificação:
    - A instrução que está em IR é decodificada como Tipo J:
    - Endereço: IR[Endereço]
  - Ciclo de Execução:
    - Endereço recebe um Shift Left de 2 bits
    - Concatenação dos 4 primeiros bits do PC com o Endereço
    - Dessa forma, temos o Endereço-Alvo
    - $ra ← PC
    - PC ← Endereço-Alvo

- **lw $a0, 0($s0)**
  - Ciclo de Busca:
    - MAR ← PC
    - MBR ← Memória[MAR]
    - PC ← PC+4
    - IR ← MBR
  - Decodificação:
    - A instrução que está em IR é decodificada como Tipo I:
    - RS ← IR[RS] (nesse caso, $s0)
    - RT ← IR[RT] (nesse caso, $a0)
    - Deslocamento ← IR[Deslocamento] (nesse caso, 0)
  - Ciclo de Execução:

        * ALUInput1 ← Registrador[RS]
        * Deslocamento passa pelo Extensor de Sinal
        * ALUInput2 ← Deslocamento
        * ALUResult ← ALUInput1 + ALUInput2
        * MBR ← Memória[ALUResult]
        * Registrador[RT] ← MBR

- **bgt $a1, 2, valor_maior**
  Essa pseudo-instrução é transformada em três instruções reais de máquina equivalentes: **addi $1, $0, 2**, **slt $1, $1, $5** e **bne $1, $0, 14**. As três instruções serão explicadas a seguir:

  – **addi $1, $0, 2**:
    * Ciclo de Busca:
      · MAR ← PC
      · MBR ← Memória[MAR]
      · PC ← PC+4
      · IR ← MBR
    * Decodificação:
      · A instrução que está em IR é decodificada como Tipo R:
      · RS ← IR[RS] (nesse caso, $0)
      · RT ← IR[RT] (nesse caso, o imediato 2)
      · RD ← IR[RD] (nesse caso, $1)
    * Ciclo de Execução:
      · ALUInput1 ← Registrador[RS]
      · RT imediato passa por um extensor de sinal
      · ALUInput2 ← Registrador[RT]
      · ALUResult ← ALUInput1 + ALUInput2
      · Registrador[RD] ← ALUResult

  – **slt $1, $1, $5**:
    * Ciclo de Busca:
      · MAR ← PC
      · MBR ← Memória[MAR]
      · PC ← PC+4
      · IR ← MBR

    - Decodificação:
        - A instrução que está em IR é decodificada como Tipo R:
        - RS ← IR[RS] (nesse caso, $1)
        - RT ← IR[RT] (nesse caso, $5)
        - RD ← IR[RD] (nesse caso, $1)
    - Ciclo de Execução:
        - ALUInput1 ← Registrador[RS]
        - ALUInput2 ← Registrador[RT]
        - ALUResult ← ALUInput1 < ALUInput2 (1 se verdadeiro, 0 se falso)
        - Registrador[RD] ← ALUResult

- **bne $1, $0, 14**:
    - Ciclo de Busca:
        - MAR ← PC
        - MBR ← Memória[MAR]
        - PC ← PC+4
        - IR ← MBR
    - Decodificação:
        - A instrução que está em IR é decodificada como Tipo I:
        - RS ← IR[RS] (nesse caso, $1)
        - RT ← IR[RT] (nesse caso, $0)
        - Deslocamento ← IR[Deslocamento] (nesse caso, 14, que é a distância até o símbolo "valor_maior")
    - Ciclo de Execução:
        - ALUInput1 ← Registrador[RS]
        - ALUInput2 ← Registrador[RT]
        - Se ALUInput1 ≠ ALUInput2, então:
        - Deslocamento recebe Shift Left de 2 bits
        - BranchAdd ← PC + Deslocamento
        - PC ← BranchAdd
        - Caso contrário:
        - PC permanece inalterado (visto que já foi incrementado no Ciclo de Busca)

- **beq $a1, 1, valor_um**
  Essa pseudo-instrução é transformada em duas instruções reais de máquina equivalentes: **addi $1, $0, $1** e **beq $1, $5, 2**. As duas instruções serão explicadas a seguir:
  - **addi $1, $0, $1**: já foi explicada anteriormente (página 9)
  - **beq $1, $5, 2**:
    - Ciclo de Busca:
      - MAR ← PC
      - MBR ← Memória[MAR]
      - PC ← PC+4
      - IR ← MBR
    - Decodificação:
      - A instrução que está em IR é decodificada como Tipo I:
      - RS ← IR[RS] (nesse caso, $1)
      - RT ← IR[RT] (nesse caso, $5)
      - Deslocamento ← IR[Deslocamento] (nesse caso, 2, que é a distância até o símbolo "valor_um")
    - Ciclo de Execução:
      - ALUInput1 ← Registrador[RS]
      - ALUInput2 ← Registrador[RT]
      - Se ALUInput1 = ALUInput2, então:
      - Deslocamento recebe Shift Left de 2 bits
      - BranchAdd ← PC + Deslocamento
      - PC ← BranchAdd
      - Caso contrário:
      - PC permanece inalterado (visto que já foi incrementado no Ciclo de Busca)

- **beq $a1, 2, valor_dois**
  Essa pseudo-instrução é transformada em duas instruções reais de máquina equivalentes: **addi $1, $0, $1** e **beq $1, $5, 5**. As duas instruções serão explicadas a seguir:
  - **addi $1, $0, $1**: já foi explicada anteriormente (p. 9)
  - **beq $1, $5, 5**:

        * Ciclo de Busca:
            · MAR ← PC
            · MBR ← Memória[MAR]
            · PC ← PC+4
            · IR ← MBR
        * Decodificação:
            · A instrução que está em IR é decodificada como Tipo I:
            · RS ← IR[RS] (nesse caso, $1)
            · RT ← IR[RT] (nesse caso, $5)
            · Deslocamento ← IR[Deslocamento] (nesse caso, 5, que é a distância até o símbolo "valor_dois")
        * Ciclo de Execução:
            · ALUInput1 ← Registrador[RS]
            · ALUInput2 ← Registrador[RT]
            · Se ALUInput1 = ALUInput2, então:
            · Deslocamento recebe Shift Left de 2 bits
            · BranchAdd ← PC + Deslocamento
            · PC ← BranchAdd
            · Caso contrário:
            · PC permanece inalterado (visto que já foi incrementado no Ciclo de Busca)

- **sw $t0, 0($s0)**
    - Ciclo de Busca:
        * MAR ← PC
        * MBR ← Memória[MAR]
        * PC ← PC+4
        * IR ← MBR
    - Decodificação:
        * A instrução que está em IR é decodificada como Tipo I:
        * RS ← IR[RS] (nesse caso, $s0)
        * RT ← IR[RT] (nesse caso, $t0)
        * Deslocamento ← IR[Deslocamento] (nesse caso, 0)

  - Ciclo de Execução:
    * ALUInput1 ← Registrador[RS]
    * Deslocamento passa pelo Extensor de Sinal
    * ALUInput2 ← Deslocamento
    * ALUResult ← ALUInput1 + ALUInput2 (endereço calculado)
    * MBR ← Registrador[RT]
    * Memória[ALUResult] ← MBR

- **j valor_fim**
  - Ciclo de Busca:
    * MAR ← PC
    * MBR ← Memória[MAR]
    * PC ← PC+4
    * IR ← MBR
  - Decodificação:
    * A instrução que está em IR é decodificada como Tipo J:
    * Endereço: IR[Endereço] (endereço denotado pelo símbolo valor_fim)
  - Ciclo de Execução:
    * Endereço recebe um Shift Left de 2 bits
    * Concatenação dos 4 primeiros bits do PC com o Endereço
    * Dessa forma, formamos o Endereço-Alvo
    * PC ← Endereço-Alvo

- **subu $sp, $sp, 16**
  Essa pseudo-instrução é transformada em três instruções reais de máquina equivalentes: **lui $1, $0**, **ori $1, $1, 16** e **subu $29, $29, $1**. As três instruções serão explicadas a seguir:
  - **lui $1, $0**:
    * Ciclo de Busca:
      · MAR ← PC
      · MBR ← Memória[MAR]
      · PC ← PC+4

        - IR ← MBR
    - Decodificação:
        - A instrução que está em IR é decodificada como Tipo I:
        - RS ← IR[RS] (nesse caso, $0)
        - RT ← IR[RT] (nesse caso, o imediato 2)
        - Imediato ← IR[Imediato]
    - Ciclo de Execução:
        - Imediato passa por um Extensor de Sinal
        - Registrador[RT] ← ALUInput1

- **ori $1, $1, 16**:
    - Ciclo de Busca:
        - MAR ← PC
        - MBR ← Memória[MAR]
        - PC ← PC+4
        - IR ← MBR
    - Decodificação:
        - A instrução que está em IR é decodificada como Tipo I:
        - RS ← IR[RS] (nesse caso, $1)
        - Imediato ← IR[Imediato] (nesse caso, o valor imediato 16)
        - RD ← IR[RD] (nesse caso, $1)
    - Ciclo de Execução:
        - ALUInput1 ← Registrador[RS]
        - ALUInput2 ← Imediato
        - ALUResult ← ALUInput1 OR ALUInput2 (operação lógica OR bit a bit)
        - Registrador[RD] ← ALUResult

- **subu $29, $29, $1**:
    - Ciclo de Busca:
        - MAR ← PC
        - MBR ← Memória[MAR]
        - PC ← PC+4
        - IR ← MBR

        * Decodificação:
            · A instrução que está em IR é decodificada como Tipo R:
            · RS ← IR[RS] (nesse caso, $29)
            · RT ← IR[RT] (nesse caso, $1)
            · RD ← IR[RT] (nesse caso, $29)
        * Ciclo de Execução:
            · ALUInput1 ← Registrador[RS]
            · ALUInput2 ← Registrador[RT]
            · ALUResult ← ALUInput1 - ALUInput2
            · Registrador[RD] ← ALUResult

- **j valor_loop**
    - Ciclo de Busca:
        * MAR ← PC
        * MBR ← Memória[MAR]
        * PC ← PC+4
        * IR ← MBR
    - Decodificação:
        * A instrução que está em IR é decodificada como Tipo J:
        * Endereço: IR[Endereço] (endereço denotado pelo símbolo valor_loop)
    - Ciclo de Execução:
        * Endereço recebe um Shift Left de 2 bits
        * Concatenação dos 4 primeiros bits do PC com o Endereço
        * Dessa forma, formamos o Endereço-Alvo
        * PC ← Endereço-Alvo

- **addu $sp, $sp, 4**
  Essa pseudo-instrução é transformada em três instruções reais de máquina equivalentes: **lui $1, $0**, **ori $1, $1, 4** e **addu $29, $29, $1**. As três instruções serão explicadas a seguir:
    - **lui $1, $0**: já foi explicada anteriormente (p. 13)
    - **ori $1, $1, 4**: já foi explicada anteriormente (p. 14)
    - **addu $29, $29, $1**:

    - Ciclo de Busca:
        - MAR ← PC
        - MBR ← Memória[MAR]
        - PC ← PC+4
        - IR ← MBR
    - Decodificação:
        - A instrução que está em IR é decodificada como Tipo R:
        - RS ← IR[RS] (nesse caso, $29)
        - RT ← IR[RT] (nesse caso, $1)
        - RD ← IR[RT] (nesse caso, $29)
    - Ciclo de Execução:
        - ALUInput1 ← Registrador[RS]
        - ALUInput2 ← Registrador[RT]
        - ALUResult ← ALUInput1 + ALUInput2
        - Registrador[RD] ← ALUResult

- **mul $t2, $t0, 2**
  Essa pseudo-instrução é transformada em duas instruções reais de máquina equivalentes: **addi $1, $0, 2** e **mul $10, $8, $1**. As duas instruções serão explicadas a seguir:

  - **addi $1, $0, 2**: já foi explicada anteriormente (p. 9)
  - **mul $10, $8, $1**:
    - Ciclo de Busca:
        - MAR ← PC
        - MBR ← Memória[MAR]
        - PC ← PC+4
        - IR ← MBR
    - Decodificação:
        - A instrução que está em IR é decodificada como Tipo R:
        - RS ← IR[RS] (nesse caso, $8)
        - RT ← IR[RT] (nesse caso, $1)
        - RD ← IR[RT] (nesse caso, $10)
    - Ciclo de Execução:
        - ALUInput1 ← Registrador[RS]

· ALUInput2 ← Registrador[RT]
· ALUResult ← Sequência de operações lógicas que performam a multiplicação de RS e RT
· Registrador[RD] ← ALUResult

- **ble $a2, $a3, valor_loop**
  Essa pseudo-instrução é transformada em duas instruções reais de máquina equivalentes: **slt $1, $7, $6** e **beq $1, $0, -32**. As duas instruções serão explicadas a seguir:
  - **slt $1, $7, $6**: já foi explicada anteriormente (p. 9)
  - **beq $1, $5, -32**:
    * Ciclo de Busca:
      · MAR ← PC
      · MBR ← Memória[MAR]
      · PC ← PC+4
      · IR ← MBR
    * Decodificação:
      · A instrução que está em IR é decodificada como Tipo I:
      · RS ← IR[RS] (nesse caso, $1)
      · RT ← IR[RT] (nesse caso, $5)
      · Deslocamento ← IR[Deslocamento] (nesse caso, -32, que é a distância até o símbolo "valor_loop")
    * Ciclo de Execução:
      · ALUInput1 ← Registrador[RS]
      · ALUInput2 ← Registrador[RT]
      · Se ALUInput1 = ALUInput2, então:
      · Deslocamento recebe Shift Left de 2 bits
      · BranchAdd ← PC + Deslocamento
      · PC ← BranchAdd
      · Caso contrário:
      · PC permanece inalterado (visto que já foi incrementado no Ciclo de Busca)

- **jr $ra**
  - Ciclo de Busca:
    - MAR ← PC
    - MBR ← Memória[MAR]
    - PC ← PC+4
    - IR ← MBR
  - Decodificação:
    - A instrução que está em IR é decodificada como Tipo J:
    - Endereço: IR[Endereço] (endereço contido no registrador $ra)
  - Ciclo de Execução:
    - PC ← Endereço

# Bibliografia

W. STALLINGS - Arquitetura e organização de computadores. Pearson, 10a. edição, 2017.
D.A. PATTERSON e J. L. HENNESSY - Organização e projeto de computadores – Interface Hardware-Software. Elsevier, 5a. edição, 2017.