UNIVERSIDADE FEDERAL DE SÃO CARLOS
Centro de Ciências Exatas e Tecnologia
Departamento de Computação

Laboratório de Arquitetura e Organização de Computadores 1

Semana(s) 15

Processador MIPS Monociclo [1] [2] [3] [4]

Aviso ENPE

Esta prática foi adaptada para ser realizada em simuladores. Ao escrever o relatório, deve-se destacar quando não foi possível obter a informação solicitada por limitação dos simuladores usados.

1 Instruções Gerais

- Ler atentamente todo o conteúdo desta experiência;
- Conferir detalhadamente a ”descrição” do circuito na linguagem de descrição de hardware utilizada;

2 Objetivos da Prática

- Implementar e simular as instruções *bne* e *ori* do processdor MIPS monociclo em *SystemVerilog*;
- Criar um código em *Assembly MIPS* para teste destas instruções;

3 Materiais e Equipamentos

- simulador online EDA Playground, disponível em http://www.edaplayground.com.
- Simulador MARS, disponível em http://courses.missouristate.edu/kenvollmar/mars/

[1] Documento adaptado das Práticas de Laboratório dos professores: Luciano Neris, Edilson Kato, Maurício Figueiredo.
[2] Revisão 09/11/2020: Prof. Artino Quintino e Luciano Neris.
[3] Revisão 05/05/2021: Prof. Maurício Figueiredo.
[4] Revisão 07/03/2022: Prof. Maurício Figueiredo.

4 Fundamentos teóricos

4.1 Introdução

Ao longo das aulas anteriores descreve-se o funcionamento de componentes de um processador MIPS monociclo, Figura 1, tais como: blocos digitais, os elementos de estado, o *datapath* (caminho de dados) e a unidade de controle. Além disso, também capacita-se na inserção de instruções no conjunto ISA, correntemente suportado pela organização simplificada do MIPS. Nesta última aula, trabalha-se na inserção de duas instruções - *bne* e *ori*.

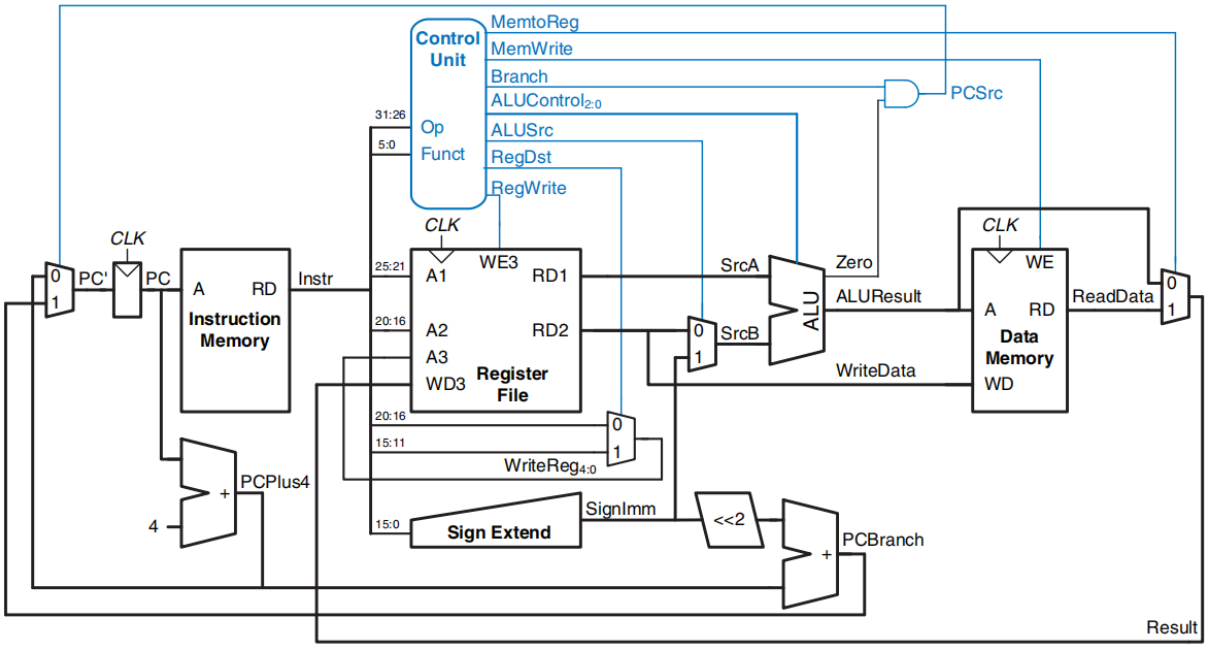

Figura 1: Processador MIPS monociclo, [1]

4.2 Instrução bne - *branch if not equal*

A instrução *bne* (*branch if not equal* - desvio se diferente) realiza um desvio (*branch*) se os valores dos registradores não forem iguais - semelhante à instrução *beq*, que realiza o salto caso o valor dos registradores sejam iguais.

Tabela 1: Instrução *bne* - *branch if not equal*

Descrição	desvia se os valores dos registradores não forem iguais
Operação	if $s != $t advance_pc (4 + offset « 2); else advance_pc (4);
Sintaxe	bne #s, $t, offset
Codificação	0001 01ss ssst tttt iiii iiii iiii iiii

Para determinar se a condição de desvio é atendida, durante a execução da instrução *bne* a ULA opera uma subtração nos valores dos dois registradores indicados no código. Se o resultado não for zero, os dois registradores não têm os mesmos valores e, neste caso, o programa assume o desvio. Caso contrário, a próxima instrução (em posição na memória) é buscada na memória tal como em uma sequência normal de execução de programa. Ou seja, se o resultado da subtração ULA não for zero, então **PC + 4 + offset«2**, do contrário realiza **PC + 4**.

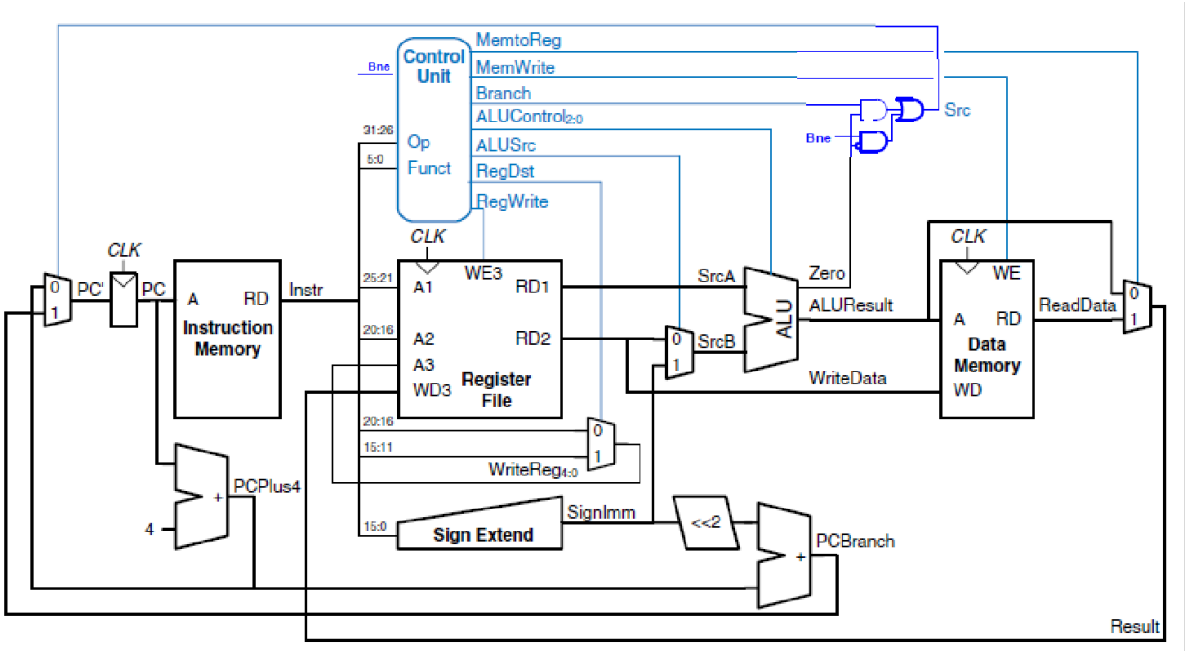

Figura 2: Processador MIPS monociclo com suporte para a instrução *bne*, [1]

Considerando que a ULA sempre apresenta resultados, o valor do pino (da ULA) ZERO está sempre presente. No entanto, observe-se que o desvio só é realizado se a instrução for *bne* (se o *opcode* dessa instrução estiver presente). Na figura, o pino *Zero* e o pino corresponde a um sinal de controle de instrução *bne* (gerado a partir do respectivo *opcode*) são conectados a uma porta AND de duas entradas. O sinal de controle de instrução *bne* indica que a instrução sendo executada é *bne*. Assim, obtém-se a Tabela 2 com os valores dos sinais para a instrução *bne*, onde se insere uma coluna à direita para

Tabela 2: Tabela-verdade para o *main decoder* - *bne*, [1]

Instrução	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump	Bne
R-type	000000	1	1	0	0	0	0	10	0	0
lw	100011	1	0	1	0	0	1	00	0	0
sw	101011	0	X	1	0	1	X	00	0	0
addi	001000	1	0	1	0	0	0	00	0	0
j	000010	0	X	X	X	0	X	XX	1	0
beq	000100	0	X	0	1	0	X	01	0	0
bne	000101	0	X	0	0	0	X	01	0	1

Tabela 3: - Instrução *ori* - or immediate

Descrição	realiza um *or* entre um registrador e uma constante e armazena o resultado em um outro registrador
Operação	$t = $s | imm; advance_pc (4);
Sintaxe	ori $t, #s, immt
Codificação	0011 01ss ssst tttt iiii iiii iiii iiii

representar esse novo sinal.

4.3 *ori - Bitwise or immediate*

A instrução *ori* quando executada deve corresponder à execução da operação lógica *or* entre um registrador e uma constante, armazenando o resultado em um segundo registrador. Essa instrução é semelhante às instruções *add, sub* e *nand*. A principal diferença é que a instrução *ori* usa um campo imediato como seu segundo operando (ou seja, ela não lê dois registradores). Outras operações lógicas que também operam em constantes (imediato) são as instruções *andi* e *xori*. [5]

A maioria das instruções MIPS estende o sinal do imediato. Por exemplo, nas instruções *addi, lw* e *sw* ocorre a extensão de sinal para imediatos positivos e negativos. Para as operações lógicas (*andi, ori, xori*) ocorre algo diferente. Desde que, para essas instruções, a operação se dá entre um valor de 32 *bits* (proveniente de um registrador) e um imediato de 16 *bits*, deve-se inserir 0s nos 16 *bits* superiores do imediato para compor um argumento de 32 *bits*. Essa ação é chamada de extensão de zero em vez de extensão de sinal.

Para acrescentar essa operação ao processador MIPS é necessário modificar o *ALU decoder* e o *main control*. Em aula prática anterior, observa-se que quando o sinal *ALUOp*

[5] As constantes são raras para a instrução *nor*, pois seu uso principal é inverter *bits* de um único operando; assim a arquitetura MIPS não possui uma versão imediata do NOR.

Tabela 4: - Sinal de controle *ALUOp*, [1]

ALUOp	Ação
00	add
01	subtract
10	olhar o campo **funct**
11	imediato

Tabela 5: - Tabela-verdade do *ALU Decoder*, [1]

ALUOp	Funct	ALUControl
00	X	010 (add)
01	X	110 (subtract)
11	X	001 (or)
10	100000 (add)	010 (add)
10	100010 (sub)	110 (subtract)
10	100100 (and)	000 (and)
10	100101 (or)	001 (or)
10	101010 (slt)	111 (set less than)

é 11 a unidade de controle não seleciona ação específica. No caso da instrução *ori*, quando o valor do sinal *ALUOp* é 11, a unidade de controle seleciona uma operação com imediatos, como *addi, andi* e *ori*. Nesta prática utiliza-se apenas a operação *ori*. Assim, a nova tabela para o sinal de controle *ALUOp* é igual à mostrada na Tabela 4.

A partir dessa proposta de projeto, torna-se necessária uma alteração no *ALU decoder*, conforme mostrado na Tabela 5.

Conforme descrito anteriormente, deve-se fazer a extensão de zeros no imediato, operação que pode ser alcançada inserindo-se um módulo *zero extension* ao circuito do caminho de dados (*datapath*). Também é necessário estender o sinal *ALUSrc* de 1 para 2 *bits* e estender o multiplexador *SrcB* de 2 para 3 entradas. Essas alterações são mostradas na Figura 3.

Tabela 6: Tabela-verdade para o *main decoder* - *ori*, [1]

Instrução	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	00	0	0	0	10	0
lw	100011	1	0	01	0	0	1	00	0
sw	101011	0	X	01	0	1	X	00	0
beq	000100	0	X	00	1	0	X	01	0
addi	001000	1	0	01	0	0	0	00	0
j	000010	0	X	XX	X	0	X	XX	1
ori	001101	1	0	1X	0	0	0	11	0

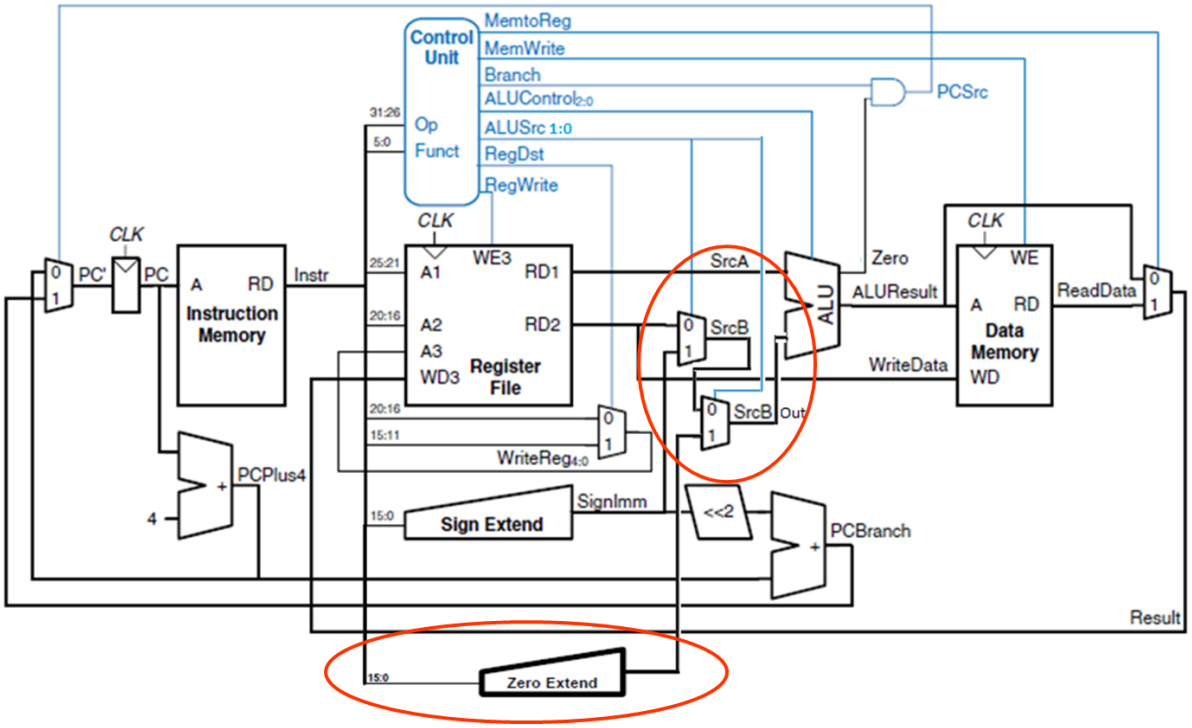

Figura 3: Processador MIPS monociclo com suporte para a instrução *ori*, [1]

Observe-se que, na Figura 3, introduz-se um multiplexador - para estender as entradas de *SrcB* - e insere-se o sinal de controle *ALUSrc*. Por fim, deve-se alterar o módulo *main decoder* para satisfazer a atribuição definida para o sinal de controle *ALUSrc* tal como definido na Tabela 6. Dessa forma, para o caso em que o sinal *ALUSrc* não é 10 (nesse caso, impondo uma escolha a partir do sinal *Funct*), atribuindo-se para o bit mais significativo de *ALUSrc* o valor 0, implementa-se o caminho de dados para *add* e *sub*. Se, por outro lado, ao bit mais significativo de *ALUSrc* é atribuído o valor 1, obtém-se o caminho de dados para *ori*.

5 Implementação em SystemVerilog

A atividade da aula prática dessa semana corresponde à implementação das instruções *bne* e *ori*. A seguir, segue um exemplo de modificações realizadas quando implementam-se as instruções *addi* e *j*.

Parte-se do seguinte código:

```
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic [1:0] aluop);

  logic [7:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, aluop} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 8'b11000010; //Rtype
      6'b100011: controls <= 8'b10100100; //LW
      6'b101011: controls <= 8'b00101000; //SW
      6'b000100: controls <= 8'b00010001; //BEQ
      default:   controls <= 8'bxxxxxxxx; //???
    endcase
endmodule
```

5.1 Adicionando as instruções *addi* e *j*

Para a implementação da instrução *j (jump)*, precisa-se inserir uma nova saída no MainDecoder, correspondendo à linha de controle *jump*, conforme a Tabela 7. Assim, deve-se aumentar o número de bits para 9 (comparando com o MainDecoder antes da inserção da instrução *j (jump)*). Outras alterações são previstas de forma que o endereço do salto possa ser calculado e o registrador PC seja atualizado corretamente (não mostradas aqui, mas presentes no código fornecido).

Para a implementação da instrução *addi* precisa-se apenas inserir a linha de código referente às saídas do MainDecoder, também conforme a Tabela 7. Nada mais é necessário.

As alterações mencionadas podem ser verificadas no código a seguir. Sugere-se comparar o número de bits da variável *control* que representa os sinais de controle na saída

Tabela 7: Tabela-verdade do *main decoder* - *J* e *ADDI* , [1]

Instrução	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

do MainDecoder no código em seguida e naquela presente no código base mostrado anteriormente (além de conferir os respectivos valores com aqueles mostrados na Tabela 7).

Essas informações são vistas e discutidas brevemente na Aula Prática 13.

```
1  module maindec(input  logic [5:0] op,
2                 output logic       memtoreg, memwrite,
3                 output logic       branch, alusrc,
4                 output logic       regdst, regwrite,
5                 output logic       jump,
6                 output logic [1:0] aluop);
7
8    logic [8:0] controls;
9
10   assign {regwrite, regdst, alusrc,
11           branch, memwrite,
12           memtoreg, jump, aluop} = controls;
13
14   always_comb
15     case(op)
16       6'b000000: controls <= 9'b110000010; //Rtype
17       6'b100011: controls <= 9'b101001000; //LW
18       6'b101011: controls <= 9'b001010000; //SW
19       6'b000100: controls <= 9'b000100001; //BEQ
20       6'b001000: controls <= 9'b101000000; //ADDI
21       6'b000010: controls <= 9'b000000100; //J
22       default:   controls <= 9'bxxxxxxxxx; //???
23     endcase
24 endmodule
```

5.2 Adicionando a instrução *bne*

As alterações necessárias são:

- No módulo **maindec**, acrescentar a instrução **bne**, acrescentando também a linha

de controle **bne** e aumentando o número de *bits* da instrução. Acrescentar o valor do sinal **bne** às outras instruções, conforme a Tabela 2.

- No módulo **controller**, adicionar **bne** junto ao **branch** e acrescentar alterar a lógica do sinal **pcsrc** para comportar também a instrução **bne**.

5.3 Adicionando a instrução *ori*

As alterações necessárias são:

- Nos módulos **mips** ("mips.sv"), **controller** ("mips.sv") e **maindec** ("mips.sv"), alterar **alusrc** (que agora deve suportar 2 *bits*).
- No módulo **maindec** ("mips.sv"), acrescentar a instrução *ori* e aumentar o número de *bits* da instrução. Modificar o valor do sinal **alusrc** das outras instruções, conforme a Tabela 6.
- No módulo **aludec** ("mips.sv"), fazer as modificações de forma a implementar a Tabela 5.
- No módulo *datapath* ("mips.sv"), alterar **alusrc** (agora são 2 *bits*) e acrescentar as linhas **zeroinm** e **srcbout**. DICA: Ver áreas destacadas na Figura 3.
- Em **register file logic** ("mips.sv"), acrescentar o extensor de zeros **signzero** (que tem como saída a linha **zeroinm** acima. Pode-se fazer de forma semelhante à **signext**.
- Em **ALU logic** ("mips.sv"), acrescentar o segundo multiplexador. Lembre-se que a linha **alusrc** é utilizada nos dois multiplexadores (o anterior e este que acrescentaremos). Também em **ALU logic**, alterar **alu** conforme a Figura 3 (trocar **srcb** por **srcbout**).
- Por fim, acrescentar em **mipsparts.sv** o módulo **signzero** (extensor de sinal).

6 Comparador de Arquivos DiffChecker

A ferramenta DiffChecker (disponível em https://www.diffchecker.com/ tem como principal funcionalidade a comparação de entre dois arquivos do tipo texto. Ele mantém o resultado da comparação por um período que pode ser especificado de acordo com a necessidade.

Não há dificuldades para sua utilização, mas é recomendável uma experiência prévia com exemplos simples.

Para efeitos do contexto da produção do relatório, considerem que o período necessário para manutenção dos resultados de comparação seja de 1 mês no mínimo.

7 Atividades/Tarefas

- Utilizando os módulos para a implementação do MIPS monociclo fornecidos no endereço https://edaplayground.com/x/fsrt, modificá-los para dar suporte às instruções *bne* com opcode 000101 e *ori* com o opcode 001101.
- É fornecido, nos itens 5.2 e 5.3, algumas sugestões de quais modificações podem ser realizadas de maneira a se inserir essas instruções no ISA da máquina. É possível conseguir o resultado de outra forma, obviamente.
- Implementar cada uma das instruções acima separadamente. Ou seja, forneça no relatório dois *links*, um para cada instrução. Utilizar a ferramenta DiffChecker (disponível em https://www.diffchecker.com/ para indicar quais alterações e inserções são necessárias para que os sistemas se adequassem às novas instruções inseridas.
- Elaborar um programa em *Assembly* com o objetivo para ser executado pelo sistema MIPS simplificado, visando ilustrar as funcionalidades das instruções inseridas *bne* e *ori*.
- Alterar o mipstest.sv gerando um programa de teste *test bench* visando verificar automaticamente a correta execução das instruções inseridas no ISA do MIPS disponível, com apresentação de formas de onda de sinais de controle e dados que ilustrem a execução das instruções em estudo e suas funcionalidades (particularmente no caso da *bne* devem ser confirmadas as situações de salto e negação de salto). A especificação das formas de onda faz parte da tarefa e devem ser aquelas relacionadas diretamente as instruções em estudo. Apresentar na saída do programa os resultados de sinais de controle na forma de tabelas. Para garantir a correta observação de formas de onda, inserir pelo menos 3 operações NOP *no operation*logo antes de finalizá-lo.
- DICA: Consulte as Aulas Práticas anteriores, caso seja necessário.
- Salvar as implementações no EDA Playground como Público - para que possa ser visualizado por aqueles que tiverem o *link* - e incluir o endereço no relatório.
- Anexar os códigos em *Assembly* (extensão .asm) ao arquivo de envio (relatório).
- Elaborar um relatório simplificado que além da página de apresentação sejam apresentadas duas seções, cada qual associada a uma das instruções estudadas, com itens que satisfaçam os seguintes requisitos: (1) endereço do EDA Playground (links) da respectiva implementação; (2) programas em Assembly correspondente ao estudo experimental, acompanhados de breves explicações acerca da estratégia utilizada para a verificação das respectivas funcionalidades da instrução; (3) endereço para o DifChecker correspondente para a comparação entre o sistema original e a implementação do sistema MIPS com a inserção da respectiva instrução.
- Realizar e postar a atividade prática (Atividade P09)

Referências Bibliográficas

[1] D. Harris, "Digital design and computer architecture," 2012.