



**joaocarlos** Substituição da imagem e atualização da especificação do roteiro.



 1 contributor

# Lab 8: Um Processador Monociclo Completo

Prof. João Carlos Bittencourt

Monitor: Éverton Gomes dos Santos

Centro de Ciências Exatas e Tecnológicas

Universidade Federal do Recôncavo da Bahia, Cruz das Almas

## Introdução

Ao longo deste roteiro de laboratório você irá aprender a:

- Integrar ALU, registradores, etc., para formar um caminho de dados completo.
- Projetar a unidade de controle de um processador MIPS.
- Integrar a unidade de controle ao caminho de dados.
- Integrar unidades de memória a um processador.
- Codificar um conjunto de instruções.
- Mais práticas com rotinas de teste para verificação de um processador.

**Revise os Slides de Aula: Projeto do Processador RISC231-M1**

Estude os slides de aula cuidadosamente, e revise os Capítulos 7.1--7.3 do [livro do David e Sarah Harris](#). Revise também o [folheto verde](#) do livro Patterson & Hennessy contendo o conjunto de instruções do processador MIPS.



Uma versão PDF deste folheto pode ser acessada [aqui](#).

Apesar de haverem diferenças entre os livros de autoria de Patterson & Hennessy e o livro texto Harris Harris, nós utilizaremos o primeiro. Faremos isso por que o [simulador MARS](#), que você utilizará para construir seu código assembly, segue o livro do Patterson & Hennessy.

Estude os Slides para identificar as decisões de projeto específicas para versão do processador utilizada em nosso laboratório:

- **Exceções:** Nossa versão do RISC231 não possui suporte a exceções.

☰ 154 lines (107 sloc) | 16.2 KB

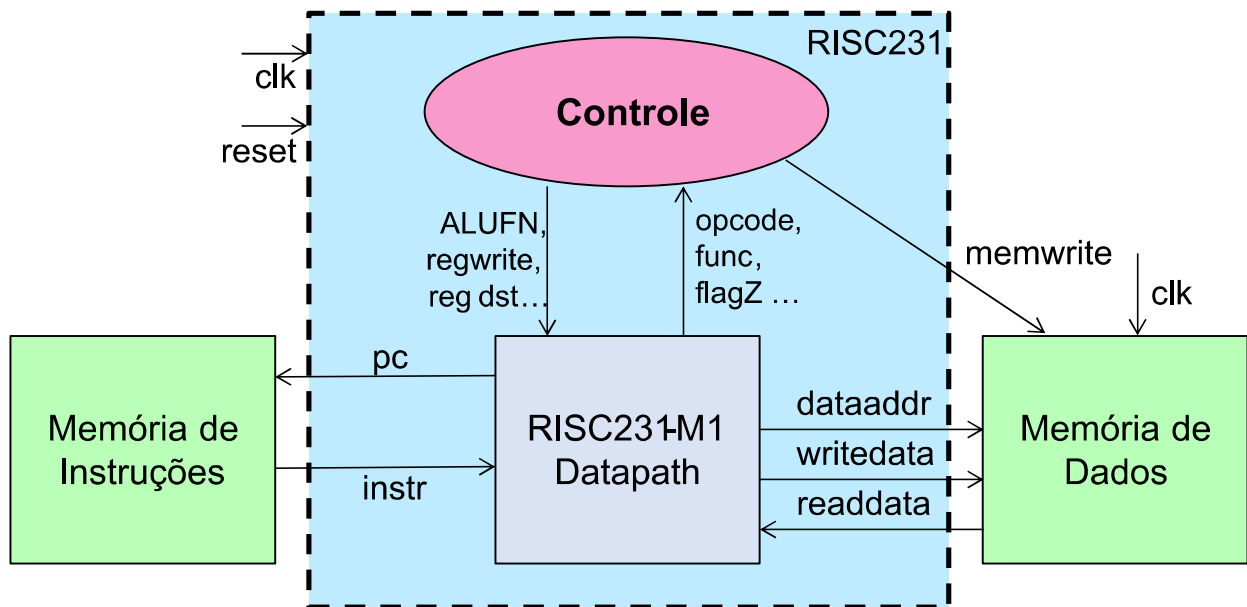
...

reiniciado para o endereço `0x0040_0000`. Este endereço foi escolhido tendo em vista compatibilizá-lo com o assembler MARS. Desta forma, o registrador PC, presente no *data path*, deve ser inicializado neste endereço, e reiniciado sempre que o `reset` for acionado.

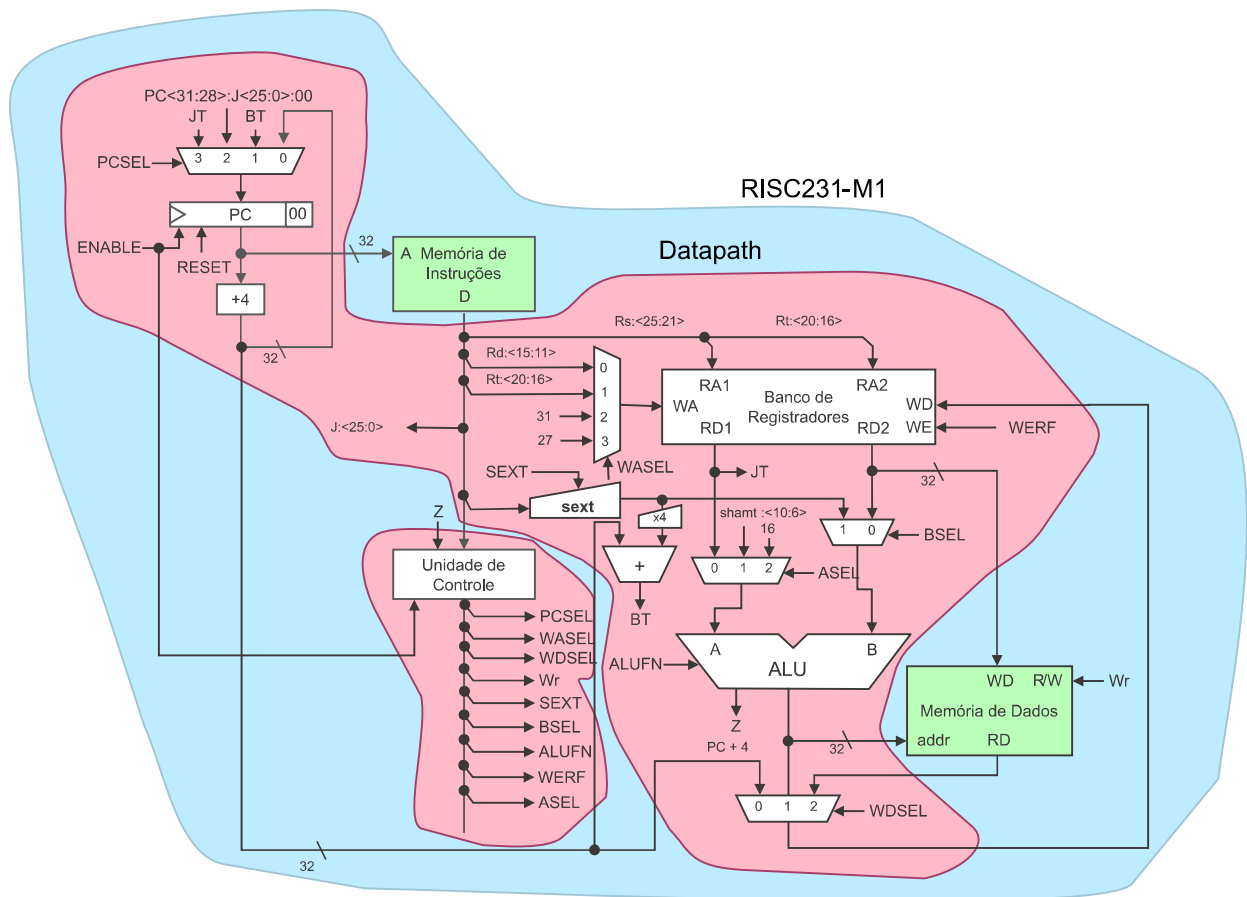
- **Enable:** Para auxiliar no processo de depuração, nós vamos incorporar um sinal de entrada `enable`. Quando ativo, o processador executa normalmente as instruções. Entretanto, quando `enable == 0`, o processador "congela". Esse procedimento é realizado desativando a escrita nos seguintes componentes: *program counter*, *register file* e *memória de dados*. Essa modificação permitirá executar nossos programas passo-a-passo, auxiliando assim no processo de depuração.

## A Unidade de Controle


Para preparar o projeto de um processador MIPS, nós iniciaremos com o desenvolvimento da unidade de controle. A seguir são apresentados dois diagramas no nosso processador MIPS *single-cycle*, primeiro com uma visão de alto nível, e em seguida uma versão mais detalhada.




A seguir apresentamos uma visão mais detalhada.



Primeiro vamos desenvolver a unidade de controle. Ela deve dar suporte a **TODAS** as instruções apresentadas a seguir:

- Load ( lw ) e store ( sw )
- Instruções do Tipo-I: addi , addiu , slti , sltiu , ori , lui , andi , xori }
  -  Diferente do que você possa ter aprendido, addiu , na verdade, não realiza uma soma sem sinal (unsigned). Na verdade, ela estende o sinal do imediato (replica o bit mais significativo até completar os 32 bits). A única diferença entre addi e addiu é que a instrução addiu não causa uma

exceção na ocorrência de um *overflow*, enquanto que a `addi` sim. Como não estamos implementando exceções no nosso processador, `addiu` e `addi` são idênticas para nossos propósitos.

-  Também diferente do que você possa ter aprendido, a instrução `sltiu`, na verdade, também estende o sinal do imediato, *mas realiza uma comparação sem sinal*. Ou seja, a operação definida por `ALUFN` é `LTU`.
- Note ainda que a instrução `ori` deve estender zero a partir do imediato (completar a palavra com zeros), por se tratar de uma operação lógica! Finalmente, a extensão do sinal para a instrução `lui` é um *don't-care*, uma vez que o imediato de 16-bits é posicionado na parte mais significativa do registrador.
- Instruções do Tipo-R: `add`, `addu`, `sub`, `and`, `or`, `xor`, `nor`, `slt`, `sltu`, `sll`, `sllv`, `srl` e `sra`
  - Para nossos propósitos, a instrução `addu` é *semelhante* à instrução `add`. A diferença está apenas na forma como elas lidam com o *overflow* -- situação que será ignorada no nosso projeto. A razão para darmos suporte à instrução `addu` é que o assembler MARS geralmente introduz automaticamente instruções `addu` no nosso código, especialmente para a faixa de endereços de memória que nós estamos usando.
- Instruções do Tipo-J e saltos condicionais (*branches* - Tipo-I): `beq`, `bne`, `j`, `jal` e `jr`.

Estude os slides de aula sobre o processador RISC231 mono-ciclo. Em seguida preencha a Tabela abaixo com os valores de todos os sinais de controle para as 28 instruções RISC listadas aqui. Se o valor de um sinal de controle não importa para uma dada instrução, você poderá usar o símbolo de *don't care* ( `1'bx`, `2'bx`, etc., dependendo da quantidade de bits).



A partir dos valores da Tabela, complete o código Verilog da unidade de controle no arquivo `controller.v`, disponibilizado junto com os arquivos de laboratório.

Utilize o *test bench* fornecido para simular e validar o seu projeto. Assim como nos roteiros anteriores, esse *test bench* é auto-verificável, de modo que, se algum erro for identificado, ele será sinalizado no *waveform* do simulador.



Certifique-se de utilizar os mesmos nomes, presentes no teste, para as entradas e saídas do *top-level*.

Type	Instr	werf	wdsel	wasel	asel	bsel	sext	wr	alufn
I-Type	LW	1	10	01	00	1	1	0	0XX01
	SW								

Type	ADDI	werf	wdsel	wasel	asel	bsel	sext	wr	alufn
	ADDIU								
	SLTI								
	SLTIU								
	ORI								
	LUI								
	ANDI								
	XORI								
	BEQ								
	BNE								
J- Type	J								
	JAL								

R- Type	ADD								
	ADDU								
	SUB								
	AND								
	OR								
	XOR								
	NOR								
	SLT								
	SLTU								
	SLL								
	SLLV								
	SRL								
	SRLV								

Type	Instr SRA	werf	wdsel	wasel	asel	bsel	sext	wr	alufn
	JR								

## Projete o processador RISC231 Monociclo

Junte todas as partes do diagrama do caminho de dados para criar um processador RISC mono-ciclo tal qual discutido em aula. Os códigos Verilog para alguns dos módulos foram disponibilizados junto com os arquivos de laboratório. De forma mais específica, você deverá realizar as tarefas elencadas a seguir.

**Descreva o processador RISC231-M1, juntamente com as memórias de instruções e dados**, como apresentado em aula. Comece com o arquivo [top.sv](#), o qual já apresenta o módulo *top-level* em Verilog. Entenda como ele está em conformidade com o diagrama de blocos apresentado no início do roteiro, e tome nota de todos os parâmetros.

Por enquanto, vamos escolher um tamanho relativamente pequeno para cada memória, por exemplo, 128 posições de memória para a memória de instruções e 64 para a memória de dados.



Os endereços produzidos pelo processador para acessar as memórias ainda continuarão sendo de 32 bits, mesmo que sejam utilizados menos bits de endereço. Use os módulos ROM e RAM fornecidos ( [rom\\_module.sv](#) e [ram\\_module.sv](#) ), e observe o seguinte:

- A memória de instruções será uma ROM, enquanto a memória de dados será uma RAM. Nós iremos instanciar os módulos ROM e RAM (sem modificar duas descrições Verilog), e fornecer os parâmetros apropriados, os quais, por sua vez, são definidos no módulo *top-level* de teste. (Ver [top.sv](#) ).
- Nós vamos enviar todos os 32 bits do contador de programas ( PC ) para fora do processador e conectá-lo à memória de instruções, mas eliminaremos os dois bits menos significativos na interface, de modo que *apenas uma palavra de endereço de 30 bits* seja enviada para dentro da memória de instruções. Isso irá converter um endereço de byte para um endereço de palavra. (Ver [top.sv](#) ).
- Da mesma maneira, vamos enviar todos os 32 bits de endereço da memória de dados para fora do processador, mas cortar os dois bits menos significativos da interface. Desse modo, *apenas uma palavra de endereço de 30 bits* é de fato enviada para a memória de dados (veja [top.sv](#) ).
- Ambas as memórias devem retornar uma palavra de 32 bits (  $D_{bits} = 32$  ). Seus valores iniciais são lidos a partir do `initfile`, o qual corresponde ao nome do arquivo de inicialização da memória.

Inicialize as memórias de instruções e dados utilizando o método apresentado no [Lab 7](#). O arquivo que possui os valores iniciais para a memória de instruções conterá uma instrução codificada de 32 bits por linha (em hexadecimal). O arquivo que com os valores iniciais para a memória de dados também terá apenas valores de dados de 32 bits a cada linha.



Você pode criar estes arquivos dentro do próprio Quartus Prime, ou utilizando seu editor de textos preferido.

- O módulo do processador, contendo os blocos controlador e *data path*, foi fornecido no arquivo [risc231-m1.sv](#). Entenda como ele se espelha no diagrama de blocos apresentado a seguir.



Observe que não há diferenças entre o projeto do MIPS do livro de Harris e Harris e o que nós estamos desenvolvendo neste roteiro. Entretanto, o nosso processador possui uma versão muito mais sofisticada de ALU. Portanto, não siga de forma irrestrita as informações presentes no livro; ao invés disso, siga os slides de aula e anotações dos laboratórios. O caminho de dados deve ser de 32 bits (registradores, ALU, memória de dados e memória de instruções, usam palavras de 32 bits).

- Complete todas as pequenas peças do caminho de dados, de modo que o projeto se pareça com aquele apresentado na aula, também reproduzido a seguir. Pequenos conjuntos de código podem ser "aninhados" (no lugar de escritos em módulos separados), por exemplo, multiplexadores, extensor de sinal, somadores, deslocamento-por-2, etc.

As figuras apresentada na seção [A Unidade de Controle](#) apresentam uma decomposição hierárquica do projeto *top-level*. Atente-se que o seu projeto deve seguir **exatamente** essa hierarquia.





Você não precisa executar este programa no MARS, mas pode se desejar. Ao todo são 59 instruções, então a memória de instruções deve ter pelo menos essa quantidade de posições. O testador define o parâmetro `Nloc` para memória de instrução como 64. Além disso, no *test bench*, certifique-se de especificar os nomes corretos dos arquivos que possuem valores de inicialização ( `full_imem.mem` e `full_dmem.mem` , respectivamente).

- Por enquanto, você não vai implementar esse projeto na placa. Você fará isso no projeto final.

## Acompanhamento

---

### Parte 1: Unidade de Controle (entrega: sexta-feira 08 de julho, 2022)

Durante a aula esteja pronto para apresentar para o professor ou monitor:

- O arquivo Verilog: `controller.sv` .
- A simulação para a [Unidade de Controle](#), utilizando o *test bench* fornecido junto com os arquivos de laboratório.

### Parte 2: Processador RISC231 Monociclo (entrega: sexta-feira 15 de julho, 2022)

ALL of the Verilog files, but skip the ALU and its submodules. Also, the screenshot of the simulation waveforms for the “full” self-checking tester provided.

- **TODOS** os arquivos Verilog, com exceção a ALU e seus sub-módulos.
- Uma demonstração de funcionamento do seu processador RISC231 utilizando o [testador auto-verificável](#) fornecido junto com os arquivos de laboratório.

## Agradecimentos

---

Esse roteiro é fruto do trabalho coletivo dos professores e monitores de GCET231:

- 18.1: Caio França dos Santos
- 18.2: Matheus Rosa Pithon
- 20.2: Matheus Rosa Pithon
- 21.1: Matheus Rosa Pithon, Éverton Gomes dos Santos
- 21.2: Éverton Gomes dos Santos