

Documentação do Trabalho Prático 02 - Caminho de Dados RISC-V

Maria Eduarda O. G. Andrade - 5365, Manoel Augusto A. da Costa - 5369

¹Departamento de Ciência da Computação
Universidade Federal de Viçosa (UFV) – Florestal, MG – Brazil

{maria.e.andrade@ufv.br, manoel.costa@ufv.br}

Abstract. This document describes the process of developing and implementing a simplified version of the RISC-V datapath on the DE2-115 FPGA (Cyclone IV, Altera). The objective was to build a functional subset of the processor capable of executing a specific group of instructions. The system simulates the datapath, automatically handling clock cycles and tracking the status of the 32 general-purpose registers.

Keywords: RISC-V, Datapath, FPGA, Verilog, Computer Architecture.

Resumo. Este documento apresenta o desenvolvimento e a implementação de uma versão simplificada do caminho de dados da arquitetura RISC-V na FPGA DE2-115 (Cyclone IV, Altera). O projeto visa construir um subconjunto funcional capaz de executar um conjunto específico de instruções. A solução desenvolvida realiza a simulação do caminho de dados, controlando automaticamente os ciclos de clock e monitorando os 32 registradores de propósito geral.

Palavras-chave: RISC-V, Caminho de Dados, FPGA, Verilog, Arquitetura de Computadores.

1. Introdução

Neste segundo projeto da disciplina CCF252, fomos encarregados da implementação de um montador simplificado para a arquitetura RISC-V, seguindo especificações detalhadas, que permite a execução das instruções especificadas. As instruções referentes ao nosso grupo (17) foram: **lh, sh, sub, or, andi, sll** e **bne**.

Este projeto oferece aos alunos uma oportunidade de aplicar, na prática, os conceitos relacionados à implementação do caminho de dados do processador estudado em sala de aula. Além de fortalecer habilidades de programação, também proporciona vivência direta na simulação e validação de sistemas digitais mais avançados. O desenvolvimento inclui ainda o aperfeiçoamento no uso de ferramentas como o System Builder, o Quartus Prime e na realização da síntese e implementação do design na FPGA.

2. Desenvolvimento

2.1. Entrega 1 – Simulação

Para implementar o processador RISC-V, dividimos o problema em etapas:

- Criação dos módulos do caminho de dados em Verilog e testes individuais.
- Criação do módulo datapath, instanciando os módulos: Add, Mux, InstructionMemory, RegisterMemory, PC, DataMemory, ALU, Control e ImmGen.

```

datapath.v
1  `include "add.v"
2  `include "alu.v"
3  `include "aluControl.v"
4  `include "control.v"
5  `include "dataMemory.v"
6  `include "immGen.v"
7  `include "instructionMem.v"
8  `include "mux.v"
9  `include "pc.v"
10 `include "regMemory.v"
11
12 module datapath (
13     input clock, reset
14 );
15

```

Figure 1. Datapath implementado no projeto.

- Com o objetivo de validar o funcionamento do caminho de dados, foi desenvolvido um testbench responsável por executar a simulação do sistema. Esse ambiente de teste conta com um clock automático e um sinal de reset, que inicializa os valores dos registradores e da memória. Durante a simulação, o testbench exibe no terminal o estado atual dos 32 registradores, possibilitando a análise e conferência dos resultados obtidos. Como apoio na elaboração e verificação dos testes, foi utilizado o Interpretador RISC-V indicado em sala pelo professor.

```

datapathTestbench.v
1  `include "datapath.v"
2  `timescale 1ns/100ps
3
4  module datapathTestbench;
5      reg _clock, _reset;
6
7      datapath datapathInst(.clock(_clock), .reset(_reset));

```

Figure 2. Testbench utilizado para validação do datapath.

- Utilização do GTKWave para análise dos sinais e verificação do funcionamento do datapath.

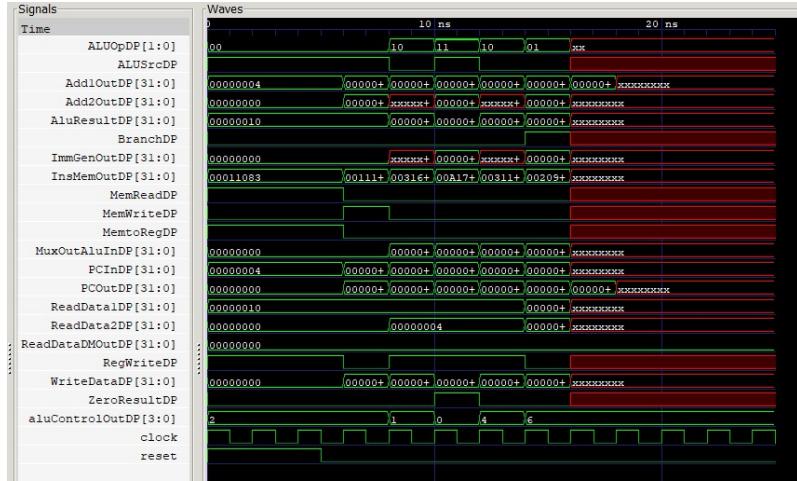


Figure 3. Análise dos sinais no GTKWave durante simulação.

Realiza a soma de dois valores de 32 bits. É usado, por exemplo, para calcular o próximo PC (PC + 4 ou desvios em `beq`).

2.1.1. Módulo Mux

O módulo Mux possui dois sinais de entrada, cada um com 32 bits, além de um sinal de controle que define qual dos dois valores será encaminhado para a saída. Sua principal função é selecionar, com base no sinal de controle, qual dos inputs será utilizado no processamento. No caminho de dados, são utilizados três multiplexadores: um responsável pela seleção do segundo operando da ALU, outro que escolhe o próximo valor do Program Counter (PC) e, por fim, um terceiro que define se o dado a ser escrito no registrador será proveniente da saída da ALU ou da memória de dados.

2.1.2. Módulo ADD

O módulo Add é responsável por realizar a operação de soma entre dois valores de 32 bits. Ele recebe dois sinais de entrada, chamados `value1` e `value2`, e gera como saída o sinal `out`, que representa o resultado da soma desses valores. Essa operação é feita de forma combinacional, utilizando uma atribuição contínua (`assign`) para calcular o valor imediatamente. No caminho de dados, esse módulo é utilizado principalmente para determinar o próximo valor do Program Counter (PC), seja realizando um incremento de +4 ou calculando um endereço de desvio, como ocorre em instruções do tipo `beq`.

2.1.3. Módulo ALUControl

O módulo ALUControl possui uma função semelhante ao módulo Control, porém é especificamente responsável por definir qual operação a ALU deve executar. Ele recebe como entradas 2 bits provenientes do sinal de controle (ALUOp) e 3 bits do campo `funct3` da instrução. Sua saída é um sinal de 4 bits que codifica a operação a ser real-

izada pela ALU. Embora, em geral, o campo `funct7` também seja utilizado para diferenciar algumas operações, no contexto deste projeto ele foi desconsiderado, uma vez que o conjunto de instruções atribuído ao grupo não demanda essa diferenciação.

2.1.4. Módulo ALU

O módulo `ALU` recebe dois sinais de entrada, ambos de 32 bits, além de um sinal de controle de 4 bits proveniente do módulo `ALUControl`, que define qual operação será executada. Como saída, o módulo gera um sinal de 32 bits correspondente ao resultado da operação realizada e um sinal adicional chamado `aluZero`, que é ativado quando o resultado da operação é igual a zero. Este sinal é fundamental para instruções de desvio condicional, como `beq`.

2.1.5. Módulo DataMemory

O módulo `dataMemory` é responsável por armazenar e fornecer os dados necessários durante a execução do programa. Possui duas entradas e uma saída de 32 bits, além de sinais de controle `memRead` e `memWrite`, bem como entradas para `clock` e `reset`. Como o conjunto de instruções do grupo inclui apenas as instruções `lh` e `sh`, não foi necessário implementar suporte ao campo `funct3`, que normalmente seria usado para diferenciar outros tipos de operações de carga e armazenamento.

2.1.6. Módulo ImmGen

O módulo `immGen` tem como função gerar o valor imediato das instruções. Ele recebe os 16 bits referentes ao imediato das instruções dos tipos I e SB e realiza a extensão de sinal para 32 bits. Este valor estendido é utilizado tanto na ALU — quando o sinal `ALUSrc` está ativo — quanto no módulo `Add` responsável pelos cálculos de desvio para atualização do PC em instruções de controle de fluxo.

2.1.7. Módulo InstructionMemory

O módulo `instructionMem` é encarregado de armazenar e fornecer as instruções que serão executadas pelo processador. Recebe como entrada o sinal de `clock` e o valor atual do Program Counter (PC) para selecionar qual instrução será lida. Sua saída é a própria instrução, codificada em 32 bits. As instruções são armazenadas previamente em um registrador interno com capacidade para até 32 instruções.

2.1.8. Módulo PC

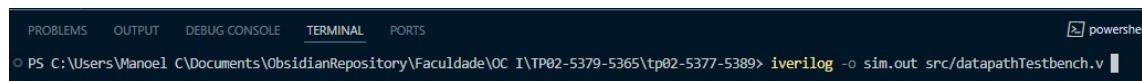
O módulo `PC` (Program Counter) é responsável por armazenar o endereço da próxima instrução a ser executada. A cada ciclo de `clock`, ele atualiza seu valor com o endereço calculado. Quando o sinal de `reset` é ativado, o PC é automaticamente reinicializado com o valor zero (32 bits).

2.1.9. Módulo RegisterMemory

O módulo `regMemory` simula um banco de 32 registradores de 32 bits cada. Inicialmente, os registradores são carregados com valores definidos em um arquivo chamado `registers-start.txt`, visto que o conjunto de instruções utilizado não possui instruções de carregamento imediato (`addi`). O módulo recebe como entrada os sinais `readReg1` e `readReg2` para seleção dos registradores de leitura, `writeReg` para o registrador de escrita, `writeData` com o dado a ser gravado e o sinal de controle `regWrite`, que habilita a escrita. Além disso, o registrador `x0` é implementado de forma que seu valor permanece sempre zero, independentemente de qualquer operação de escrita realizada.

2.1.10. Comandos para Execução

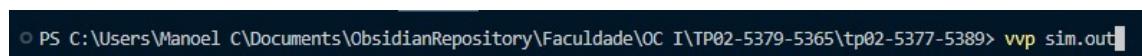
- Compilar os arquivos Verilog:
`iverilog -o sim.out src/datapathTestbench.v`



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell
PS C:\Users\Manoel C\Documents\ObsidianRepository\Faculdade\OC I\TP02-5379-5365\tp02-5377-5389> iverilog -o sim.out src/datapathTestbench.v
```

Figure 4. Comando utilizado para compilar os arquivos Verilog no terminal.

- Executar a simulação, gerar o arquivo de dump `datapath.vcd` e visualizar os valores dos registradores no terminal: `vvp sim.out`.



```
PS C:\Users\Manoel C\Documents\ObsidianRepository\Faculdade\OC I\TP02-5379-5365\tp02-5377-5389> vvp sim.out
```

Figure 5. Execução do comando para rodar a simulação e gerar o arquivo de saída.

- Abrir o arquivo `datapath.vcd` no GTKWave para análise dos sinais e verificação do funcionamento do datapath: `gtkwave datapath.vcd`.



```
PS C:\Users\Manoel C\Documents\ObsidianRepository\Faculdade\OC I\TP02-5379-5365\tp02-5377-5389> gtkwave datapath.vcd
```

Figure 6. Abertura do arquivo `datapath.vcd` no GTKWave para análise dos sinais.

3. Desenvolvimento – Entrega 2

3.1. Implementação da Parte 2

Para iniciar o desenvolvimento na FPGA, utilizamos o programa *System Builder* com o objetivo de gerar o projeto no Quartus já configurado com as portas e conexões necessárias. Em seguida, incluímos no arquivo `DE2_115.v` todos os módulos utilizados na construção do caminho de dados, incluindo o módulo `datapath`.

Também adicionamos o módulo `display`, responsável por exibir o valor do *Program Counter* nos dois primeiros displays de 7 segmentos da FPGA. Por fim, realizamos a instanciação e a conexão das portas e fios necessários para garantir o funcionamento correto do circuito.

```

//== REG/WIRE declarations ==
wire [31:0] PCFPGA;

//== Structural coding ==
datapath datapathFPGA(
    .clock(~KEY[0]),
    .reset(~KEY[1]),
    .PCOutDPde2(PCFPGA)
);

display disp7(.valorPC(PCFPGA[7:4]), .HEX(HEX7));
display disp6(.valorPC(PCFPGA[3:0]), .HEX(HEX6));

```

Figure 7. Declarações e código estrutural

Durante a compilação do arquivo, foram gerados alguns *warnings* (avisos), mas nenhum erro que impedisse a conclusão do processo.

```

1 332102 Design is not fully constrained for setup requirements
1 332102 Design is not fully constrained for hold requirements
>   Quartus Prime Timing Analyzer was successful. 0 errors, 4 warnings
1 293000 Quartus Prime Full Compilation was successful. 0 errors, 139 warnings

```

Figure 8. Final da compilação no Quartus

3.2. Casos de Teste

Nesta seção, apresentamos dois dos casos de teste utilizados para verificar o funcionamento e os resultados do circuito implementado.

Para alternar entre o Conjunto 1 e o Conjunto 2 de instruções, é necessário abrir o arquivo `datapathTestbench.v` e modificar a linha 15, onde está o comando:

```
$readmemb("./textos/instructions1_bin.txt", datapathInst.instructionMem)
```

Basta substituir o nome do arquivo `instructions1_bin.txt` por `instructions2.txt`, ou vice-versa, dependendo do conjunto de instruções que se deseja testar.

```

1 lh x1, @x2
2 sh x1, @x2
3 or x1, x2, x3
4 andi x1, x2, 10
5 sll x1, x2, x3
6 bne x1, x2, 16

```

Figure 9. Instruções em assembly para teste 1

```

1 000000000000000010001000010000011
2 00000000000100010001000000100011
3 00000000001100010110000010110011
4 00000000101000010111000010010011
5 00000000001100010001000010110011
6 00000000001000001001100001100011

```

Figure 10. Instruções em binário utilizando o código do primeiro trabalho prático de CCF252

```

VCD info: dumpfile src/datapath.vcd opened for output.
registrador:[ 0 ]: 0
registrador:[ 1 ]: 256
registrador:[ 2 ]: 16
registrador:[ 3 ]: 4
registrador:[ 4 ]: 4
registrador:[ 5 ]: 4
registrador:[ 6 ]: 4
registrador:[ 7 ]: 4
registrador:[ 8 ]: 4
registrador:[ 9 ]: 4
registrador:[ 10 ]: 4
registrador:[ 11 ]: 4
registrador:[ 12 ]: 4
registrador:[ 13 ]: 4
registrador:[ 14 ]: 4
registrador:[ 15 ]: 4
registrador:[ 16 ]: 4
registrador:[ 17 ]: 4
registrador:[ 18 ]: 4
registrador:[ 19 ]: 4
registrador:[ 20 ]: 4
registrador:[ 21 ]: 4
registrador:[ 22 ]: 4
registrador:[ 23 ]: 4
registrador:[ 24 ]: 4
registrador:[ 25 ]: 4
registrador:[ 26 ]: 4
registrador:[ 27 ]: 4
registrador:[ 28 ]: 4
registrador:[ 29 ]: 4
registrador:[ 30 ]: 4
registrador:[ 31 ]: 4
inst:[ 0 ]: 000000000000000010001000010000011
inst:[ 1 ]: 00000000000100010001000000100011
inst:[ 2 ]: 00000000001100010110000010110011
inst:[ 3 ]: 00000000101000010111000010010011
inst:[ 4 ]: 00000000001100010001000010110011
inst:[ 5 ]: 0000000000100001001100001000110011
inst:[ 6 ]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
src/datapathTestbench.v:24: $finish called at 650 (100ps)

```

Figure 11. Resultados dos registradores

3.3. Resultados da Parte 2

Nesta seção, apresentamos o passo a passo da execução do clock na FPGA. Durante esse processo, o valor do *Program Counter* (PC) é atualizado e exibido em formato hexadecimal nos dois primeiros displays de 7 segmentos da placa.

Os testes foram realizados utilizando as instruções do primeiro caso de teste da Parte 1 do trabalho.

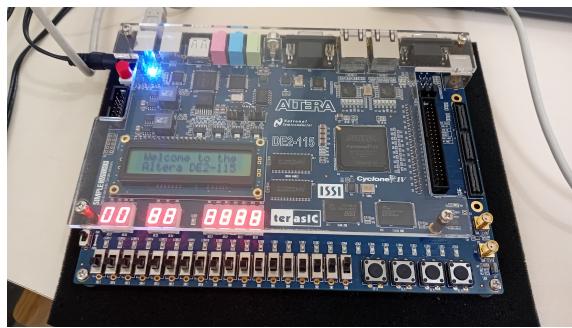


Figure 12. FPGA após reset

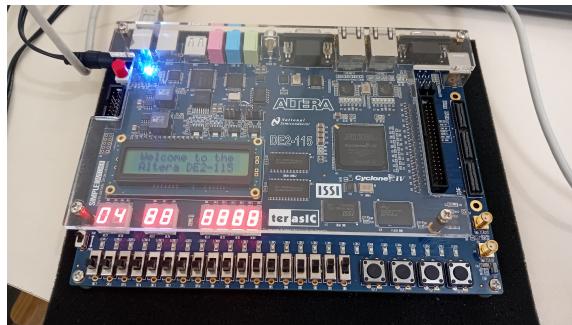


Figure 13. Program counter = 04

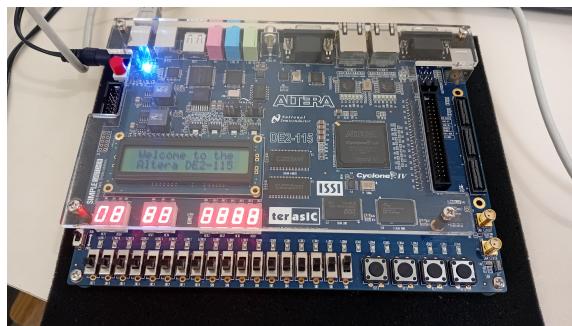


Figure 14. Program counter = 08

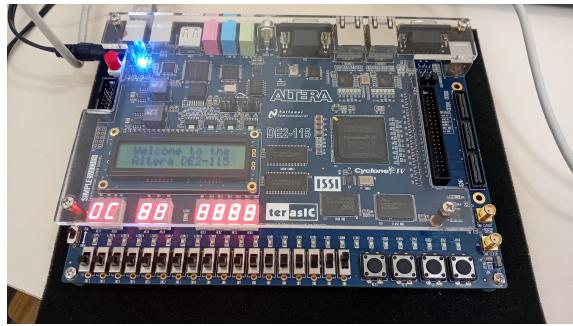


Figure 15. Program counter = 0C

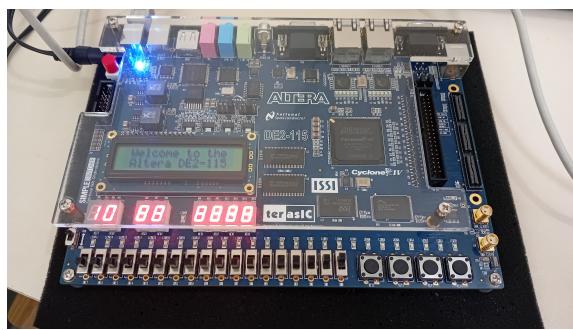


Figure 16. Program counter = 10



Figure 17. Program counter = 14



Figure 18. Program counter = 24

4. Considerações Finais

Encerrando o projeto, observamos que os resultados obtidos estão alinhados com as expectativas iniciais. Todas as etapas foram concluídas com sucesso — desde a criação dos módulos, passando pela sua instanciação no *datapath*, até a síntese e implementação no FPGA DE2-115.

A compreensão do fluxo de dados se mostrou essencial para entender o funcionamento do algoritmo como um todo. Além disso, é importante destacar que todas as fases do desenvolvimento contaram com a participação ativa de todos os integrantes do grupo.

5. Referências

- Project F. *RISC-V Load-Store Instructions*. Disponível em: <https://projectf.io/posts/riscv-load-store/>
- Cornell University. *RISC-V Interpreter*. Disponível em: <https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/>
- Waterman, A.; Asanović, K. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*, v2.2. University of California, Berkeley, 2017.
- Documentação da disciplina CCF252 – UFV Florestal.