

Trabalho Prático 2

DCC006: Organização de Computadores I

Professor: Omar Paranaíba Vilela Neto

**Universidade Federal de Minas Gerais - UFMG -
ICEX/DCC**

20/11/2023

Integrantes:

Matheus Felipe: 2019093426

Matheus Gimpel: 2021421923

Isamin Araújo: 2021031530

Introdução:

Neste projeto prático, investigamos a intersecção entre Organização de Computadores I e a Linguagem de Descrição de Hardware Verilog, com base nos tópicos discutidos durante as aulas. Utilizamos o ambiente Google Colab para executar um arquivo ".ipynb" que implementa a arquitetura RISC-V em Verilog.

Durante o desenvolvimento, fizemos modificações no caminho de dados, introduzindo novos componentes e ajustando estruturas de módulos existentes no código. Nosso foco foi a implementação das instruções "mul" (multiplicação), "div" (divisão), "andi" (operação bitwise AND imediata) e "beq" (branch if equal) em Verilog. Essas instruções desempenham um papel fundamental no processamento de dados e no controle de fluxo em programas. Exploramos conceitos cruciais, desafios enfrentados e exemplos de código para validar a correta implementação dessas instruções.

Adicionalmente, este relatório detalha as modificações incorporadas ao arquivo final do projeto, destacando as questões discutidas e as implementações realizadas para cumprir com os objetivos propostos. É crucial compreender a linguagem Verilog e o funcionamento do caminho de dados como parte essencial desse projeto.

Metodologia:

1. Estudo dos conceitos básicos de Verilog:

- a. Inicialmente, conduzimos um estudo detalhado dos conceitos fundamentais da linguagem Verilog. Durante esse processo, exploramos suas estruturas, sintaxe e funcionalidades essenciais exigidas para a implementação das instruções "mul", "div", "andi" e "beq". Esse estudo aprofundado foi fundamental para adquirir o entendimento necessário sobre a linguagem Verilog e suas capacidades, preparando o terreno para a implementação bem-sucedida das instruções mencionadas.

2. Alterações na ALU para suportar as instruções "mul" e "div":

- Uma modificação foi realizada na parte de controle da Unidade Lógica e Aritmética (ALU) para viabilizar a leitura dos cinco bits essenciais para diferenciar as operações "mul", "div", "add" e "sub". O parâmetro funct7[0] foi adicionado como entrada à ALU, juntamente com funct7[5] e funct3, possibilitando uma identificação precisa das instruções.
- Adicionalmente, foi necessário incluir esse bit extra nos parâmetros "funct", "_funct" e "_functi" para lidar com a manipulação do bit adicional que agora está sendo passado como parte do controle das operações na ALU.

Essas modificações foram essenciais para garantir que a ALU pudesse discernir entre as diferentes operações de forma precisa e adequada, aproveitando os bits adicionais passados como parâmetro para o correto funcionamento das instruções "mul", "div", "add" e "sub".

Seguem as alterações realizadas:

2.1: Alterações no Pipeline:

```
Pipeline Bar (Decode -> Exec)

wire jump_s3;
reg #(.N(1)) reg_jump_s3(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
| | | .in(jump_s2),
| | | .out(jump_s3));

wire [31:0] jaddr_s3;
reg #(.N(32)) reg_jaddr_s3(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
| | | .in(jaddr_s2), .out(jaddr_s3));
// }}}

reg #(.N(32)) reg_s2_seimm(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
| | | | | .in(ImmGen), .out(seimm_s3)); // RISCv
reg #(.N(10)) reg_s2_rt_rd(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
| | | | | .in({rs2, rd}), .out({rs2_s3, rd_s3}));
reg #(.N(5)) reg_s2_rs(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
| | | | | .in(rs1), .out(rs1_s3));

reg #(.N(5)) func7_3_s2(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
→ → → → → .in({func7[0], func7[5], func3}), .out(func_s3));
→ // Adicionando funct7[0] passado como argumento para diferenciar div, mul, add e sub
```

Imagem 1: Adição do bit funct7[0] para diferenciação das operações Tipo-R

2.2: Alterações na Unidade de Controle:

```
case (opcode)
  7'b0000011: begin // lw == 3
    alusrc  <= 1'b1;
    aluop[1:0] <= 2'd0;
    memoreg <= 1'b1;
    regwrite <= 1'b1;
    memread  <= 1'b1;
    ImmGen   <= {{20{inst[31]}}, inst[31:20]};
  end

  7'b0010011: begin /* addi e andi */
    // Adicionando andi. Caso f3 != 0, passa aluop = 3
    aluop <= {f3 == 3'b000} ? 2'd0 : 2'd3;
    alusrc <= 1'b1;
    regwrite <= 1'b1;
    ImmGen  <= {{20{inst[31]}}, inst[31:20]};
  end

  7'b0100011: begin /* sw */
    memwrite <= 1'b1;
    aluop <= 2'd0;
    alusrc <= 1'b1;
    ImmGen  <= {{20{inst[31]}}, inst[31:25], inst[11:7]};
  end

  7'b0110011: begin /* add */
    aluop <= 2'd2;
    regwrite <= 1'b1;
  end

  7'b1101111: begin /* j jump */
    jump <= 1'b1;
    ImmGen  <= {{12{inst[31]}}, inst[19:12], inst[20], inst[30:21], {1'b0}};
  end

  /* Implementação da passagem de dados para instrução beq utilizando o imediato */
  7'b1100011: begin /* beq */
    aluop <= 2'b1;
    ImmGen  <= {{19{inst[31]}}, inst[31], inst[7], inst[30:25], inst[11:8], 1'b0};
    regwrite <= 1'b0;
    branch_eq <= 1'b1;
  end
endcase
end
endmodule
```

Imagem 2: Adição de casos para identificação e do addi e beq ao control

2.3: Alterações na ALU Control e ALU:

```
assign sub_ab = a - b;
assign add_ab = a + b;
assign oflow_add = (a[31] == b[31] && add_ab[31] != a[31]) ? 1 : 0;
assign oflow_sub = (a[31] == b[31] && sub_ab[31] != a[31]) ? 1 : 0;
assign oflow = (ctl == 4'b0010) ? oflow_add : oflow_sub;
// set if less than, 2s compliment 32-bit numbers
assign slt = oflow_sub ? ~(a[31]) : a[31];
always @(*) begin
    case (ctl)
        4'd2: out <= add_ab;           /* add */
        4'd0: out <= a & b;           /* and */
        4'd12: out <= ~(a | b);       /* nor */
        4'd1: out <= a | b;           /* or */
        4'd7: out <= {{31{1'b0}}, slt}; /* slt */
        4'd6: out <= sub_ab;          /* sub */
        4'd13: out <= a ^ b;          /* xor */
        4'd5: out <= a << b[4:0];      /* sll */
        4'd8: out <= a >> b[4:0];      /* srl */
        5'd10: out <= a * b;          /* mul */
        5'd11: out <= a / b;          /* div */
        default: out <= 0;
    endcase
end
```

Imagem 3: Implementação das operações mul e div

2.3.1 : Alterações na ALU Control e ALU:

```
// Adicionando andi caso f3 != 0
module alu_control(
    input wire [4:0] funct, // funct agora tem 5 bits ao invés de 4 para refletir as mudanças
    input wire [1:0] aluop,
    output reg [3:0] aluctl);

    reg [4:0] _funct; // _funct agora tem 5 bits ao invés de 4 para refletir as mudanças
    reg [4:0] _functi; // _functi agora tem 5 bits ao invés de 4 para refletir as mudanças

    /* Adicionando casos para mul e div usando o dígito adicionado f7[0] e o f3 */
    always @(*) begin
        case(funct[4:0])
            5'd16: _funct = 5'd10; /* mul */ // 1 (f7[0]) 0(f7[5]) 000 [f3]
            5'd20: _funct = 5'd11; /* div */ // 1 (f7[0]) 0(f7[5]) 100 [f3]
        endcase
    end

    always @(*) begin
        case(funct[3:0])
            4'd0: _funct = 4'd2; /* add */
            4'd8: _funct = 4'd6; /* sub */
            4'd6: _funct = 4'd1; /* or */
            4'd4: _funct = 4'd13; /* xor */
            4'd2: _funct = 4'd7; /* slt */
            4'd7: _funct = 4'd0; /* and e andi */
            4'd1: _funct = 4'd5; /* sll */
            4'd5: _funct = 4'd8; /* srl */
            default: _funct = 4'd0;
        endcase
    end
```

Imagem 4: Alteração na quantidade de bits identificados e no tratamento de casos para mul, div, andi

3. Identificação das instruções "mul" e "div":

- a. A abordagem adotada para identificar as instruções "mul" e "div" usando o formato funct7[0] + funct7[5] + funct3 foi uma escolha criteriosa. Atribuir o valor 10000 para "mul" e o valor 10100 para "div" permitiu uma diferenciação clara entre as operações, considerando os cinco bits necessários para essa distinção.

Essa alteração foi crucial, uma vez que funct7[5] e funct3, isoladamente, não são suficientes para distinguir adequadamente as instruções. funct7[0], por sua vez, é precisamente a menor quantidade de bits necessária e suficiente para essa diferenciação.

A metodologia empregada possibilitou a implementação eficiente das instruções propostas, garantindo um funcionamento correto e uma distinção adequada entre elas. Os ajustes realizados foram fundamentais para adaptar a Unidade Lógica e Aritmética (ALU) e o controle do processador, permitindo a incorporação das novas instruções "mul" e "div" e assegurando a execução correta das operações desejadas.

Testes:

Para garantir a correta funcionalidade das instruções implementadas ("mul", "div", "andi" e "beq"), realizamos testes abrangentes que abordaram uma variedade de cenários. Estes testes foram meticulosamente planejados para validar o comportamento das instruções em diferentes condições, tanto em situações onde se

esperava que funcionassem corretamente quanto em casos em que não deveriam.

Durante os testes, uma ampla gama de valores foi utilizado como entrada para as instruções. Isso incluiu operandos positivos e negativos, números pequenos e grandes, bem como diversas combinações de bits. A abordagem adotada foi sistemática, abrangendo todas as combinações relevantes de entrada para as instruções implementadas.

Especificamente para as instruções "mul" e "div", testamos casos onde os números eram facilmente multiplicáveis ou divisíveis, além de situações onde as operações não eram possíveis ou resultavam em comportamentos inesperados, como divisão por zero ou overflow.

Além disso, os testes também englobaram as instruções "andi" (bitwise AND imediato) e "beq" (branch if equal). Para "andi", verificamos se a operação lógica AND estava sendo executada corretamente entre o valor imediato e o registrador especificado. No caso de "beq", foram criados cenários em que o branch deveria ser tomado (valores iguais) e situações onde o branch não deveria ser tomado (valores diferentes), a fim de avaliar a correta funcionalidade desta instrução.

Ao realizar esses testes abrangentes, conseguimos confirmar o comportamento adequado das instruções implementadas, garantindo que elas produzissem os resultados esperados em diferentes situações. Caso algum problema fosse identificado, revisões e ajustes na implementação eram realizados até que todos os testes fossem aprovados com sucesso.

Essa abordagem rigorosa de testes contribuiu significativamente para assegurar a correta execução das instruções em todos os casos previstos, aumentando assim a confiabilidade e robustez do sistema implementado em Verilog.

Exemplos de testes que foram executados e estão presentes no arquivo .ipynb

MUL:

```
Problema 1: MUL

0s [play] %%writefile im_data.txt
013                                     // nop
008000093                             //addi x1, x0, 8
02900113                             //addi x2, x0, 41
022081b3                             //mul x3, x1, x2
00200293                             //addi x5, x0, 2
00b00493                             //addi x9, x0, 11
029283b3                             // mul x7, x5, x9
02238233                             // mul x4, x7, x2
013                                  // nop

// Final Esperado:
//x1 = 8      (hexa = 00000008)
//x2 = 41     (hexa = 00000029)
//x3 = 328    (hexa = 00000148)
//x4 = 902    (hexa = 00000386)
//x5 = 2      (hexa = 00000002)
//x7 = 22     (hexa = 00000016)
//x9 = 11     (hexa = 0000000B)
```

Imagem 5: Testes do Problema 1 - Mul.

```
0s [play] !cat reg.data

[icon] // 0x00000000
00000000
00000008
00000029
00000148
00000386
00000002
00000000
00000016
00000000
0000000b
00000000
00000000
00000000
00000000
00000000
```

Imagem 6: Resultados do Teste do Problema 1 - Mul.

DIV:

```
Problema 2: DIV

0s %%writefile im_data.txt
013 // nop
00800093 // addi x1, x0, 8
01000113 // addi x2, x0, 16
0a000193 // addi x3, x0, 160
3c000213 // addi x4, x0, 960
023242b3 // div x5, x4, x3
02114333 // div x6, x2, x1
022243b3 // div x7, x3, x2
013 // nop

// Final Esperado:
//x1 = 8 (hexa = 00000008),
//x2 = 16 (hexa = 00000010),
//x3 = 160 (hexa = 000000a0),
//x4 = 960 (hexa = 000003c0),
//x5 = 6 (hexa = 00000006),
//x6 = 2 (hexa = 00000002),
//x7 = 60 (hexa = 0000003c),
```

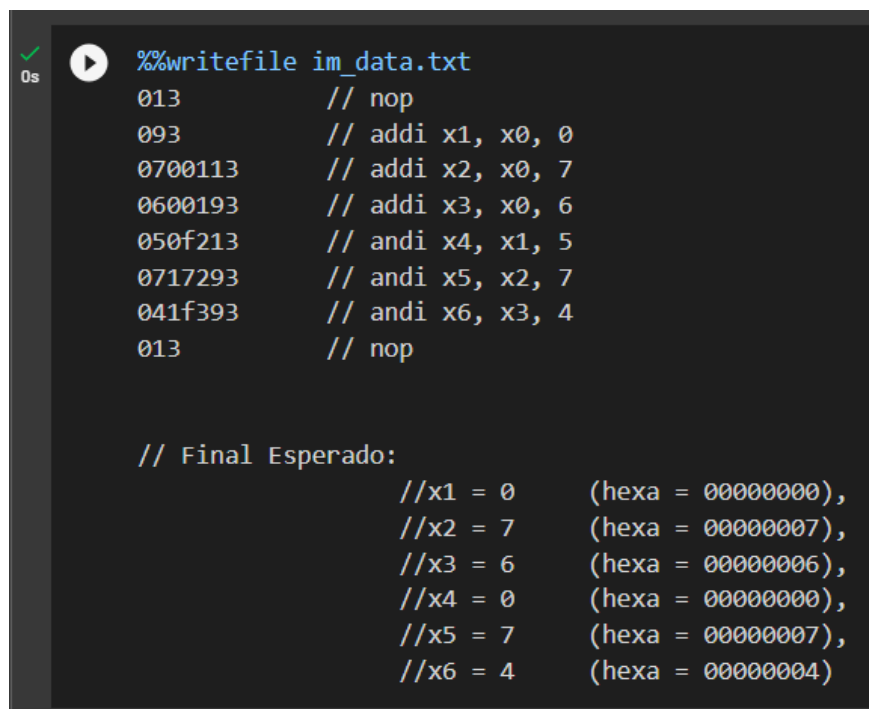
Imagem 7: Testes do Problema 2 - Div.

```
0s !cat reg.data

// 0x00000000
00000000
00000008
00000010
000000a0
000003c0
00000006
00000002
0000003c
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

Imagem 8: Resultados do Teste do Problema 2 - Divl.

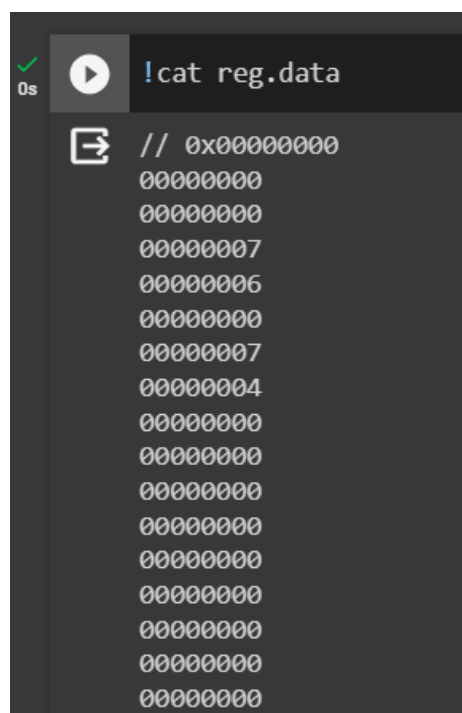
ANDI:



```
0s %%writefile im_data.txt
013      // nop
093      // addi x1, x0, 0
0700113  // addi x2, x0, 7
0600193  // addi x3, x0, 6
050f213  // andi x4, x1, 5
0717293  // andi x5, x2, 7
041f393  // andi x6, x3, 4
013      // nop

// Final Esperado:
//x1 = 0      (hexa = 00000000),
//x2 = 7      (hexa = 00000007),
//x3 = 6      (hexa = 00000006),
//x4 = 0      (hexa = 00000000),
//x5 = 7      (hexa = 00000007),
//x6 = 4      (hexa = 00000004)
```

Imagem 9: Testes do Problema 3 - Andi.



```
0s !cat reg.data
// 0x00000000
00000000
00000000
00000007
00000006
00000000
00000007
00000004
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

Imagem 10: Resultados do Teste do Problema 3 - Andi.

BEQ:

```
Problema 4: BEQ

0s 0013      %%writefile im_data.txt
00100093    //nop
00100093    //addi x1, x0, 1      // Executado
00200113    //addi x2, x0, 2      // Executado
00000193    //addi x3, x0, 0      // Executado
00108863    //beq x1, x1, end     // Branch tomado
00600393    //addi x7, x0, 6      // Pulado
00800393    //end: addi x7, x0, 8  // Executado
00008863    //beq x1, x0, end2    // Branch não tomado
00700393    //addi x7, x0, 7      // Executado
00300193    //end2: addi x3, x0, 3 // Executado
00118863    //beq x3, x1, end3    // Branch não tomado
00000863    //beq x0, x0, end4    // Branch tomado
00500393    //end3: addi x7, x0, 5 // Pulado
0500093     // addi x1, x0, 5     // Pulado
0500113     // addi x2, x0, 5     // Pulado
00000013    //end4: nop          // Executado

// Final Esperado:
//x1 = 1      (hexa = 00000001),
//x2 = 2      (hexa = 00000002),
//x3 = 3      (hexa = 00000003),
//x7 = 7      (hexa = 00000007),
//registradores restantes = 0
```

Imagem 11: Testes do Problema 4 - Beq.

```
0s 0!cat reg.data

// 0x00000000
00000000
00000001
00000002
00000003
00000000
00000000
00000000
00000000
00000007
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

Imagem 12: Resultados do Teste do Problema 4 - Beq.

Waveform:

Waveforms (formas de onda) são representações gráficas de um sinal ao longo do tempo. Elas mostram como um sinal varia em termos de amplitude e tempo, permitindo visualizar e analisar o comportamento de um sinal elétrico ou digital. As waveforms são frequentemente usadas para representar o estado lógico de um sinal binário, como 0 (baixo) e 1 (alto). Por exemplo, em um diagrama de temporização, as waveforms podem mostrar o estado de diferentes sinais em relação ao tempo, como a entrada de um circuito, os sinais de controle e os sinais de saída correspondentes.

Para melhor análise e compreensão, as waveforms anexadas neste relatório encontram-se, sem cortes, presentes no arquivo .zip enviado, tanto o modelo visual dos estágios quanto a imagem original gerada no colab. As imagens a seguir são imagens parciais para compreensão inicial. Os desenhos de seta feitos são identificadores visuais dos estágios IF, ID, EX, MEM, WB do pipeline de 5 estágios.

Problema 1: MUL:

Foram executadas e representadas na waveform as operações nop, addi, e mul, visto que elas são o mínimo necessário para testar a implementação realizada.

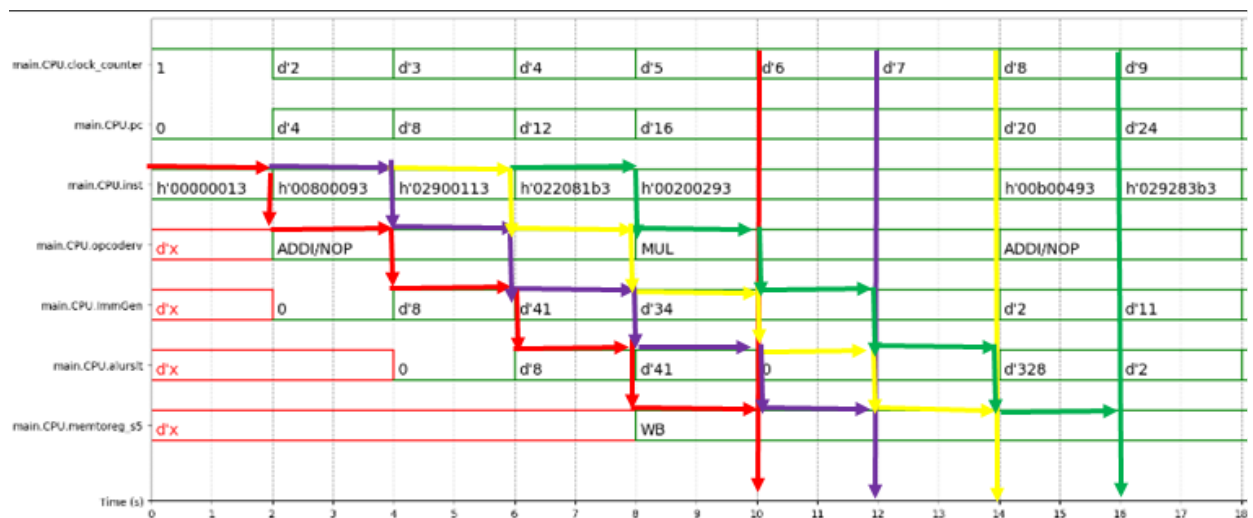


Imagem 13: Waveform visual do pipeline referente do Teste do Problema 1 - Mul.

Respectivamente, de acordo com as setas de execução:

Vermelho: NOP;

Roxo: addi;

Amarelo: addi;

Verde: mul.

É perceptível que cada estágio corresponde perfeitamente a um ciclo do pipeline, muito embora a nomenclatura e desenho da onda em si não seja preciso muitas vezes. Isso vale para todas as waveforms. A execução está perfeita, mas não a exibição.

Problema 2: DIV:

Foram executadas e representadas na waveform as operações nop, addi, e div, visto que elas são o mínimo necessário para testar a implementação realizada.

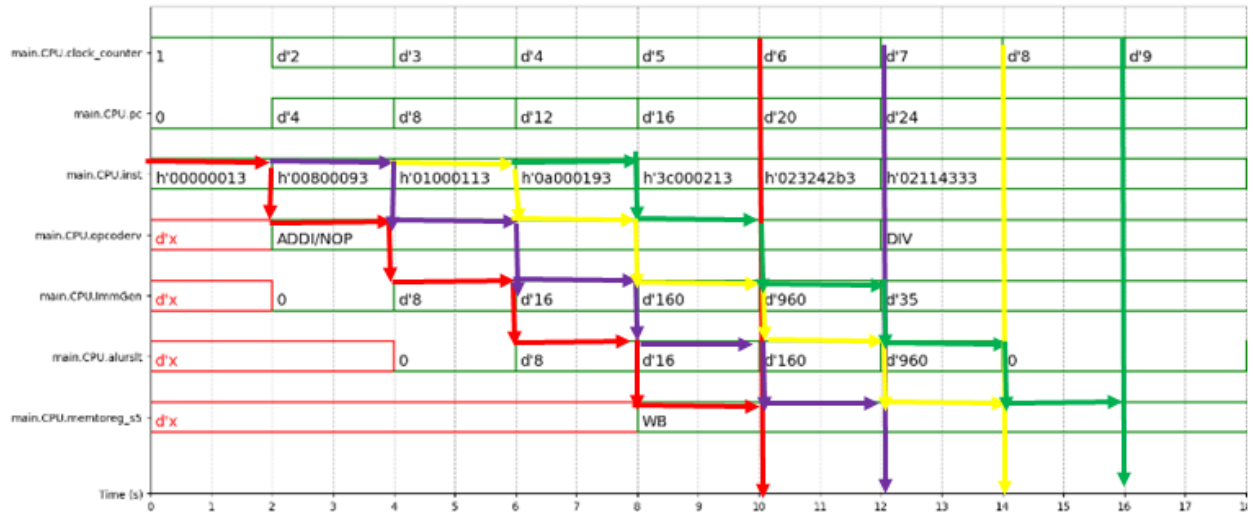


Imagem 14: Waveform visual do pipeline referente do Teste do Problema 2 - Div

Respectivamente, de acordo com as setas de execução:

Vermelho: addi;

Roxo: addi;

Amarelo: div.

Verde: div.

Problema 3: ANDI:

Foram executadas e representadas na waveform as operações nop, addi, e andi, visto que elas são o mínimo necessário para testar a implementação realizada.

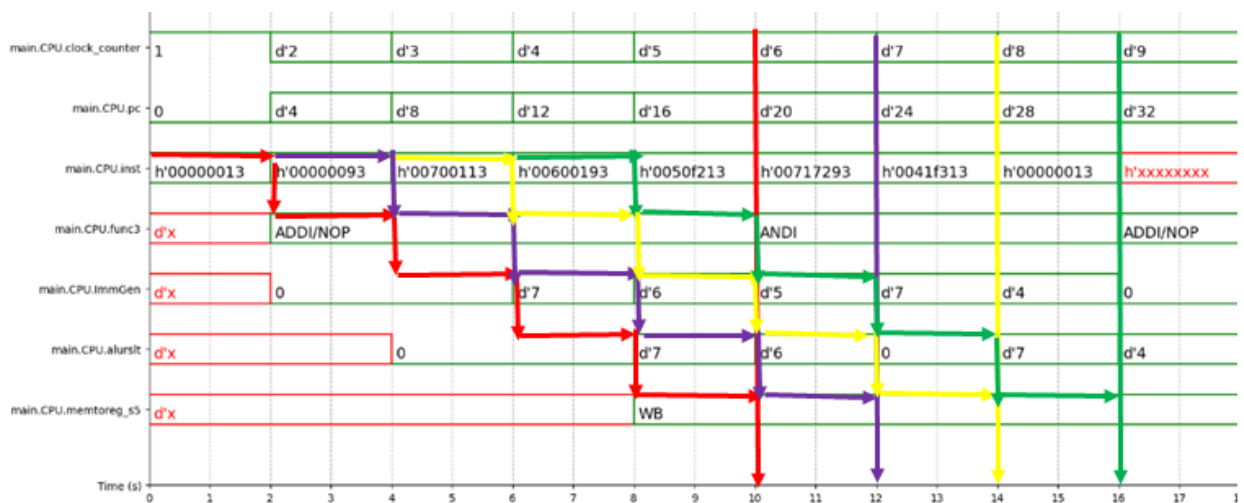


Imagem 15: Waveform visual do pipeline referente do Teste do Problema 3 - Andi.

Respectivamente, de acordo com as setas de execução:

Vermelho: addi;

Roxo: addi;

Amarelo: andi;

Verde: andi.

Problema 4: BEQ:

Foram executadas e representadas na waveform as operações nop, addi, e beq, visto que elas são o mínimo necessário para testar a implementação realizada.

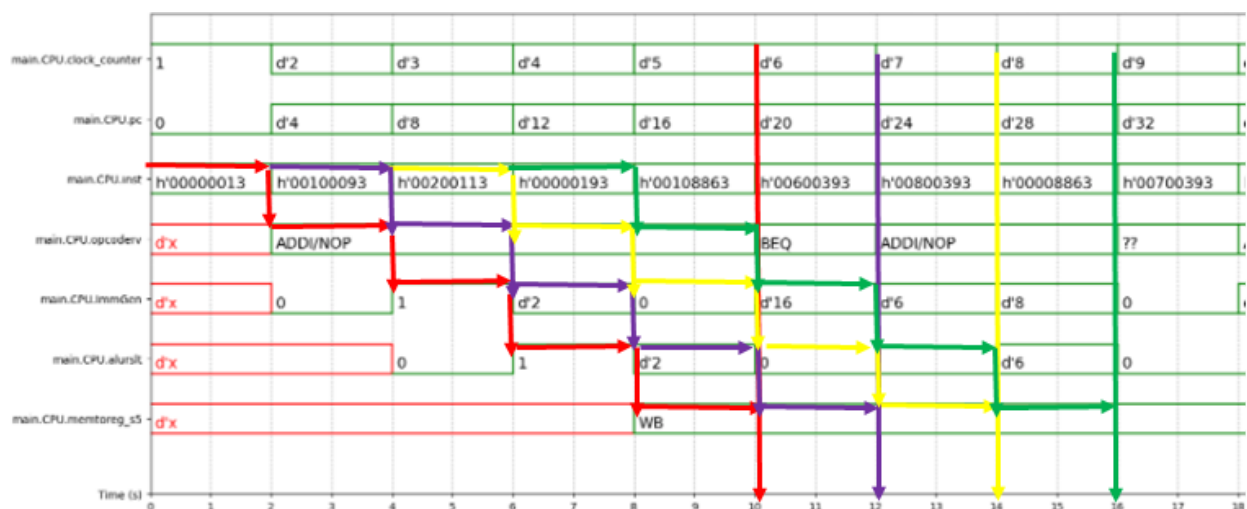


Imagem 16: Waveform visual do pipeline referente do Teste do Problema 4 - Beq.

Respectivamente, de acordo com as setas de execução:

Vermelho: addi;

Roxo: addi;

Amarelo: addi;

Verde: beq.

Conclusão:

Ao realizar as modificações necessárias para atingir os objetivos propostos neste Trabalho Prático, o grupo obteve uma compreensão mais aprofundada do funcionamento de um processador baseado em RISC-V. Além disso, ganhamos maior familiaridade e domínio da linguagem Verilog, utilizada neste trabalho, bem como da linguagem Assembly específica para RISC-V, que foi empregada na criação de testes para comprovar a implementação e funcionalidade das funções ANDI, MUL, DIV, BEQ. Foi desenvolvida também a habilidade de utilização de ferramentas complementares à análise, como o gerador de waveform, cujo funcionamento necessitou da compreensão dos dados utilizados para gerá-la corretamente.

Diante disso, é evidente que este trabalho desempenhou um papel significativo e contribuiu de maneira substancial para a consolidação dos conhecimentos adquiridos ao longo do módulo ministrado em sala de aula pela disciplina de Organização de Computadores I. Aprofundamos nosso entendimento sobre a arquitetura RISC-V, incluindo o pipeline e o caminho de dados, analisando cada detalhe de sua operação.