

UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DO TRABALHO PRÁTICO 2
DA DISCIPLINA DE ORGANIZAÇÃO DE COMPUTADORES

Raquel Gonçalves Rosa

Italo Dell’Areti

Belo Horizonte
2023

Introdução

O Trabalho Prático 2 da disciplina de Organização de Compiladores aborda a elaboração de um processador na Linguagem de Descrição de *Hardware* Verilog, baseado em uma arquitetura RISC-V de 5 estágios. Conforme requerido, serão implementados, mais especificamente, 4 tipos de instruções por meio de modificações e acréscimos no código fornecido no arquivo *.ipynb (Jupyter Notebook)*, executável por meio do Google Colab. O código provido do Jupyter Notebook simula tal processador RISC-V com um único *datapath*, sem as instruções requeridas..

Cabe destacar que, apesar dos problemas serem apresentados sequencialmente, o grupo não solucionou, necessariamente, esses problemas de forma linear, sendo muitas vezes necessário conferir se, após a implementação de uma determinada instrução, as instruções implementadas em etapas anteriores se encontravam estáveis e operacionais.

Em todas as implementações serão respondidas as perguntas principais, requeridas nas especificações do trabalho:

- Decisões de projeto adotadas pelo grupo, com suas razões;
- Testes realizados para cada instrução;
- E resultados apresentados graficamente na forma de *waveforms*.

Com isso, o trabalho alterou o caminho de dados original, de forma a realizar a inclusão de mais operações e módulos, produzindo uma simulação de processador funcional em Verilog.

Primeira Instrução

mul Rd, Rs1, Rs2

(Problema 1)

A instrução **multiply**, ou **mul**, implementada no Instruction Set Architecture (ISA) RISC-V toma os valores numéricos presentes em dois registradores de input, Rs1 e Rs2, e realiza a multiplicação deles, armazenando o resultado no registrador de destino Rd. O formato da instrução a seguir (*figura 1*) mostra os bits registradores (*rs2*, *rs1* e *rd*) e aos responsáveis por definir a instrução (*funct7*, *funct3* e *Opcode*):

Figura 1 - Campos (Fields) da instrução **mul**

<i>funct7</i> [31-25]	<i>rs2</i> [24:20]	<i>rs1</i> [19:15]	<i>funct3</i> [14:12]	<i>rd</i> [11:7]	<i>Opcode</i> [6:0]
00000 01	rs2	rs1	000	rd	0110011

a) **Decisões de Projeto:**

Para implementar essa instrução, procedemos às seguintes modificações e inserções:

1. No módulo relativo a Unidade de Controle buscamos pelo opcode referente a função **MUL**. Reconhecemos que tínhamos algumas instruções com o mesmo opcode e algumas com o mesmo *funct3*, por exemplo o **ADD**. Como encontramos a implementação da **ADD** que possui o mesmo opcode e *funct3* de **MUL**, fizemos a inserção de uma linha com condicional ternário no código com o intuito de definir o aluop de acordo com o *funct7*.

Figura 2 - Criação de um fio no *funct7* para diferenciar a instrução mul

```
//funct7 para diferenciar o mul
wire[2:0] f4 = inst[26:25];
```

Figura 3 - Adição do operador condicional ternário

```
7'b0110011: begin /* add, mul e div: mesmo opcode*/
  // Adição de um ternário para diferenciar se o opcode é do add ou do mul
  aluop <= (f4 == 2'd0) ? 2'd2 : 2'd0;
  regwrite <= 1'b1;
end
```

2. No módulo relativo a ALU, adicionamos o fio para realizar a operação **MUL**, note que o fio suporta até 64 bits para que em casos de grandes multiplicações não ocorra overflow/underflow, pois não sabíamos se isso deveria ser tratado ou não.

Figura 13 - Adição do fio da instrução **MUL**

```
// Adição de fios para o sub, add, mul e div
wire [31:0] sub_ab;
wire [31:0] add_ab;
wire [63:0] mul_ab;
wire [63:0] div_ab;
```

Figura 5 - Adição de um assign

```
// Adição de assign para facilitar a leitura
assign mul_ab = a * b;
assign div_ab = a / b;
```

Figura 6 - Adição da operação que realiza o **MUL** na *aluctrl*

```
always @(*) begin
  case (ctl)
    4'd2: out <= add_ab;      /* add */
    4'd0: out <= a & b;      /* and */
    // Adição da operação mul na ALU
    4'd3: out <= mul_ab;     /* mul */
  endcase
end
```

3. Definindo que nossa operação será identificada pelo *funct3*. Isso se deve ao fato de tanto a instrução **MUL**, quanto a instrução **DIV** possuírem o mesmo opcode e o mesmo *funct7*. Dessa forma, podemos diferenciá-las através disto.

Figura 7 - Passo 3

```
always @(*) begin
  case(funct[3:0])
    4'd0: _funct = 4'd2 ; /* add */
    // Definindo a operação mul de acordo com funct3
    4'd1: _funct = 4'd3; /* mul */
  endcase
end
```

b) Testes Realizados:

Figura 8 - Teste com apenas uma multiplicação

```
▼ Teste 1 - Uma Multiplicação

addi x1, x0, 5      # Carrega o valor 5 em x1
addi x2, x0, 3      # Carrega o valor 3 em x2
mul x3, x1, x2      # Multiplica x1 por x2 e armazena o resultado 15 em x3

✓ [48] %%writefile im_data.txt
    00500093
    00300113
    022081b3

[49] !iverilog main.v
    !./a.out

✓ [50] !cat reg.data
    // 0x00000000
    00000000
    00000005
    00000003
    0000000f
```

Figura 9 - Teste com multiplicações em sequência

```
▼ Teste 2 - Várias Multiplicações

addi x1, x0, 6      # Carrega o valor 6 em x1
addi x2, x0, 9      # Carrega o valor 9 em x2
mul x3, x1, x2      # Multiplica x1 por x2 e armazena o resultado em x3
mul x4, x2, x3      # Multiplica x2 por x3 e armazena o resultado em x4
mul x5, x3, x4      # Multiplica x3 por x4 e armazena o resultado em x5
mul x6, x4, x5      # Multiplica x4 por x5 e armazena o resultado em x6

[53] %%writefile im_data.txt
      00600093
      00900113
      022081b3
      02310233
      024182b3
      02520333

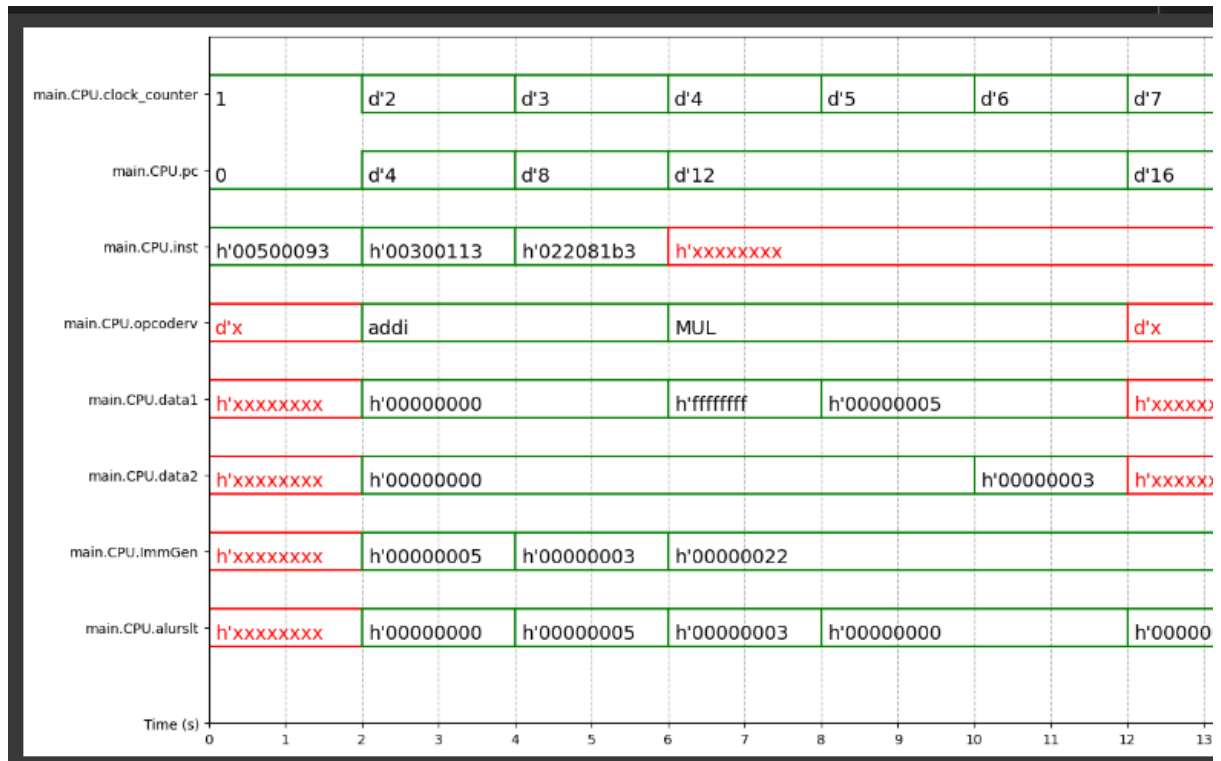
[54] %%writefile clock_div_2.v
      #40

[55] !iverilog main.v
      !./a.out

[56] !cat reg.data
      // 0x00000000
      00000000
      00000006
      00000009
      00000036
      000001e6
      00006684
      00c29e98
      00000000
```

c) **Waveforms:**

Figura 10 - Waveform referente ao primeiro teste



Segunda Instrução

div Rd, Rs1, Rs2

(Problema 2)

A instrução **divide**, ou **div**, implementada no ISA RISC-V toma os valores numéricos presentes em dois registradores de input, Rs1 e Rs2, e realiza a divisão de Rs1 por Rs2, armazenando o resultado, parte inteira da divisão, no registrador de destino Rd. O formato da instrução a seguir (figura 11) mostra os bits registradores (*rs2*, *rs1* e *rd*) e os responsáveis por definir a instrução (*funct7*, *funct3* e *Opcode*):

Figura 11 - Campos (Fields) da instrução **div**

<i>funct7</i> [31:25]	<i>rs2</i> [24:20]	<i>rs1</i> [19:15]	<i>funct3</i> [14:12]	<i>rd</i> [11:7]	<i>Opcode</i> [6:0]
00000 01	rs2	rs1	100	rd	0110011

a) **Decisões de Projeto:**

Para implementar essa instrução, procedemos às seguintes modificações e inserções:

1. Como na instrução **MUL**, no módulo relativo a Unidade de Controle buscamos pelo opcode referente a função **DIV**. Verificamos que a instrução possui o mesmo opcode de **MUL** e **ADD** e o mesmo *funct7* de **MUL**. Desse modo, vamos diferenciar a operação através do *funct3*.

Figura 12 - Criação de um fio no *funct3* para diferenciar a instrução **DIV**

```
//funct3 para diferenciar o div  
wire[2:0] f3 = inst[14:12];
```

2. No módulo relativo a ALU, adicionamos o fio para realizar a operação **DIV**, note que o fio suporta até 64 bits para que em casos de grandes divisões não ocorra overflow/underflow, pois não sabíamos se isso deveria ser tratado ou não.

Figura 13 - Adição do fio da instrução **DIV**

```
// Adição de fios para o sub, add, mul e div
wire [31:0] sub_ab;
wire [31:0] add_ab;
wire [63:0] mul_ab;
wire [63:0] div_ab;
```

Figura 14 - Adição de um assign

```
// Adição de assign para facilitar a leitura
assign mul_ab = a * b;
assign div_ab = a / b;
```

Figura 15 - Adição da operação que realiza a div na *aluout*

```
// Adição da operação div na ALU
4'd4: out <= div_ab; /* div */
```

3. Definindo que nossa operação será identificada pelo *funct3*. Isso se deve ao fato de tanto a instrução **DIV**, quanto a instrução **MUL** possuírem o mesmo opcode e o mesmo *funct7*. Para verificar a diferença, adicionamos um condicional ternário no case da *aluop*.

Figura 16 - Chamada da div

```
always @(*) begin
  case(funct[3:0])
    4'd0: _funct = 4'd2 ; /* add */
    // Definindo a operação mul de acordo com funct3
    4'd1: _funct = 4'd3; /* mul */
    4'd8: _funct = 4'd6; /* sub */
    4'd6: _funct = 4'd1; /* or */
    // O 4'd4 foi alterado pra chamar o div ao invés do xor
    4'd4: _funct = 4'd13; /* div */
```

Figura 17 - Passo 3

```
always @(*) begin
  case(aluop)
    // O código 4'd0 era o add, mas foi alterado redirecionar para o ternário
    // que diferencia a instrução mul da instrução div
    2'd0: aluout1 = (funct[2:0] == 3'd0) ? 4'd3 : 4'd4; /*mul e div*/
```

b) Testes Realizados:

Figura 18 - Teste de divisão exata

```
▼ Teste 1 - Divisão Exata

addi x1, x0, 2    # Carrega o valor 2 em x1
addi x2, x0, 10   # Carrega o valor 10 em x2
div x3, x2, x1     # Divide x2 por x1 e salva em x3

✓ [59] %%writefile im_data.txt
    00200093
    00a00113
    021141b3

✓ [59] !iverilog main.v
    !./a.out

✓ [61] !cat reg.data
    // 0x00000000
    00000000
    00000002
    0000000a
    00000005
    00000000
```

Figura 19 - Teste de divisão quebrada

```
▼ Teste 2 - Divisão Quebrada

addi x1, x0, 3      # Carrega o valor 3 em x1
addi x2, x0, 25     # Carrega o valor 25 em x2
div x3, x2, x1      # Divide x2 por x1 e salva em x3

✓ [65] %%writefile im_data.txt
      00300093
      01900113
      021141b3

✓ [66] !iverilog main.v
      !./a.out

✓ [67] !cat reg.data
      // 0x00000000
      00000000
      00000003
      00000019
      00000008
      00000000
```

Figura 20 - Teste de divisões em sequência

```
▼ Teste 3 - Várias Divisões

addi x1, x0, 2047
addi x2, x1, 869
addi x3, x0, 6
addi x4, x0, 9
addi x5, x0, 54
addi x6, x0, 324
div x7, x2, x3
div x8, x2, x4
div x9, x2, x5
div x10, x2, x6

✓ [111] %%writefile im_data.txt
      7ff00093
      36508113
      00600193
      00900213
      03600293
      14400313
      023143b3
      02414433
      025144b3
      02614533

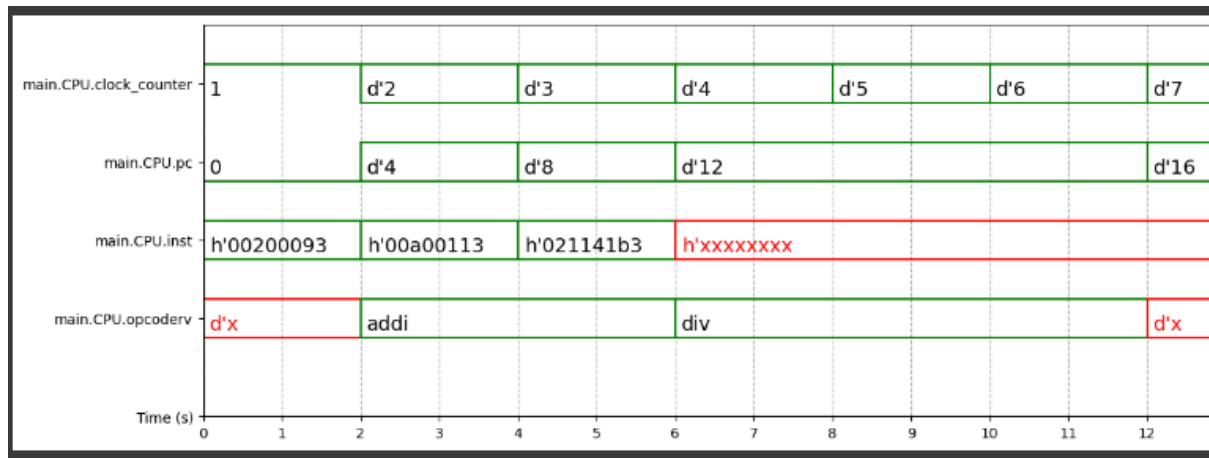
✓ [112] %%writefile clock_div_2.v
      #40

✓ [113] !iverilog main.v
      !./a.out

✓ 0s [114] !cat reg.data
      // 0x00000000
      00000000
      000007ff
      00000b64
      00000006
      00000009
      00000036
      00000144
      000001e6
      00000144
      00000036
      00000009
```

c) **Waveforms:**

Figura 21 - Waveform referente ao primeiro teste



Terceira Instrução

andi Rd, Rs1, Rs2

(Problema 3)

A instrução **andi**, ou **and immediate**, implementada no ISA RISC-V toma o valor numérico presente em um registrador de input, Rs1, e um valor imediato fornecido no campo que vai do bit 20 ao bit 31, valores esses que em seguida são comparados bit-a-bit, realizando a operação AND, sendo o resultado armazenado no registrador de destino rd. O formato da instrução a seguir (figura 22) mostra os bits registradores (*rs2* e *rd*), os bits immediate e os responsáveis por definir a instrução (*funct3* e *Opcode*):

Figura 22 - Campos (Fields) da instrução **andi**

<i>imm</i> [11:0] [31:20]	<i>rs1</i> [19:15]	<i>funct3</i> [14:12]	<i>rd</i> [11:7]	<i>Opcode</i> [6:0]
imm[11:0]	rs1	111	rd	0010011

a) Decisões de Projeto:

Para implementar essa instrução, procedemos às seguintes modificações e inserções:

1. No módulo relativo a Unidade de Controle buscamos pelo opcode referente a função **ANDI**. Verificamos que a instrução possui o mesmo opcode de **ADDI**, já que ambos são imediatos e o **ADDI** já estava implementado. Como imediatos só possuem *funct3* e *opcode*, iremos diferenciar a operação através do *funct3*.

Figura 23 - Utilização do **ADDI** para implementar o **ANDI**

```
// O andi é chamado no mesmo opcode do addi (imediatos)
7'b0010011: begin /* addi e andi */
    aluop <= 2'd3;
    alusrc  <= 1'b1;
    regwrite <= 1'b1;
    ImmGen  <= {{20{inst[31]}},inst[31:20]};
end
```

2. Adicionamos o redirecionamento para o **andi** de acordo com seu *funct3*.

Figura 22 - Passo 2

```
always @(*) begin
    case(funct[2:0])
        3'd0: _functi = 4'd2; /* add */
        // Adição da rota para redirecionamento do andi (mesmo opcode do addi)
        3'd7: _functi = 4'd0; /* andi */
    endcase
end
```


b) Testes Realizados:

Figura 24 - Teste comparando o resultado do and e do andi

▼ Teste 1 - AND e ANDI

```
addi x2,x0,5    # Carrega o valor 5 em x2
addi x3,x0,6    # Carrega o valor 6 em x2
and x4,x2,x3    # Realiza o and x2 x3 que é 4
andi x5,x2,6    # Realiza o andi com os valores iguais ao and
```

✓ [76] %%writefile im_data.txt
00500113
00600193
00317233
00617293

✓  %%writefile clock_div_2.v
#40


✓ [78] !iverilog main.v
!./a.out

✓ [79] !cat reg.data
0s
// 0x00000000
00000000
ffffffff
00000005
00000006
00000004
00000004
00000000

Figura 25 - Teste zerando todos os bits

▼ Teste 2 - Zera todos os bits

```
addi x1, x0, 10    # Carrega o valor 10 em x1
andi x2, x1, 5     # faz o and de 10 com o imediato 5
```

✓  `%%writefile im_data.txt`
`00a00093`
`0050f113`

✓ `[83] %%writefile clock_div_2.v`
`#40`

✓ `[84] !iverilog main.v`
`!./a.out`

✓ `[85] !cat reg.data`

0s

```
// 0x00000000
00000000
0000000a
00000000
00000000
```


Figura 26 - Teste com andi's em sequência

▼ Teste 3 - Vários ANDI

```
addi x1, x0, 6      # Carrega o valor 9 em x1
addi x2, x0, 9      # Carrega o valor 6 em x2
addi x3, x0, 12     # Carrega o valor 12 em x2
andi x4, x1, 7      # ANDI 6 e 7
andi x5, x2, 10     # ANDI 9 e 10
andi x6, x3, 13     # ANDI 12 e 13
```

```
✓ [88] %%writefile im_data.txt
      006000093
      00900113
      00c00193
      0070f213
      00a17293
      00d1f313
```

```
✓ [89] %%writefile clock_div_2.v
      #40
```

```
✓ [90] !iverilog main.v
      !./a.out
```

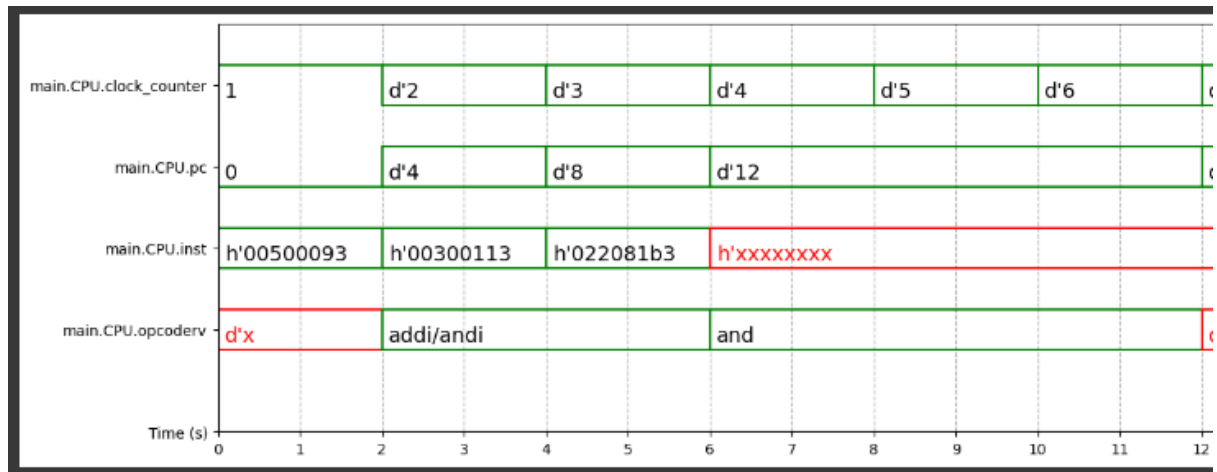
```
✓ [91] !cat reg.data
```

0s

```
// 0x00000000
00000000
00000006
00000009
0000000c
00000006
00000008
0000000c
00000000
```

c) **Waveforms:**

Figura 27 - Waveform referente ao primeiro teste



Quarta Instrução

beq Rs1, Rs2, label

(Problema 4)

A instrução **beq**, ou **branch if equal**, implementada no ISA RISC-V toma os valores numéricos presentes em dois registradores de input, Rs1 e Rs2, e compara seus valores bit-a-bit. Se os valores forem iguais, o branch é tomado e ele vai para o endereço presente no offset, caso contrário, o programa segue seu funcionamento sequencial. O formato da instrução a seguir (figura 28) mostra os bits registradores (*rs2*, *rs1*), e seus *offsets*.

Figura 28 - Campos (Fields) da instrução **beq**

<i>offset</i> [31:25]	<i>rs2</i> [24:20]	<i>rs1</i> [19:15]	<i>funct3</i> [14:12]	<i>offset</i> [11:7]	<i>Opcode</i> [6:0]
offset[12 10:5]	rs2	rs1	000	offset[4:1 11]	1100011

a) **Decisões de Projeto:**

Para implementar essa instrução, procedemos às seguintes modificações e inserções:

1. No módulo relativo a Unidade de Controle buscamos pelo opcode referente a função **BEQ**. Verificamos que não existia nenhuma instrução com o mesmo opcode já implementada. Portanto, criamos o case para o opcode do **BEQ**.

Figura 29 - Passo 1

```
// Criação do case do BEQ
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
```

Como os outros passos já estavam implementados, apenas acrescentamos o passo acima para o total funcionamento da instrução.

b) Testes Realizados:

Figura 30 - Teste com um branch verdadeiro, ou seja, onde ele foi tomado

```
▼ Teste 1 - BEQ Verdadeiro

addi x1, x0, 10      # Carrega o valor 10 em x1
addi x2, x0, 10      # Carrega o valor 10 em x2
beq x1, x2, end      # Pula para a etiqueta "equal" se x1 for igual a x2
addi x3, x0, 1       # Não executa esta instrução se a beq for verdadeira
end:
addi x3, x0, 2

✓ [95] %%writefile im_data.txt
00a00093
00a00113
00208463
00100193
00200193

✓ [96] !iverilog main.v
!./a.out

✓ [97] !cat reg.data
0s
// 0x00000000
00000000
0000000a
0000000a
00000002
00000000
```

Figura 31 - Teste com um branch falso, ou seja, onde ele não foi tomado

```

▼ Teste 2 - BEQ Falso

addi x1, x0, 5      # Carrega o valor 5 em x1
addi x2, x0, 10     # Carrega o valor 10 em x2
beq x1, x2, end     # Pula para a etiqueta "equal" se x1 for igual a x2
addi x3, x0, 1      # Não executa esta instrução se a beq for verdadeira
end:

✓ [100] %%writefile im_data.txt
00500093
00a00113
00208463
00100193

✓ [101] %%writefile clock_div_2.v
#40

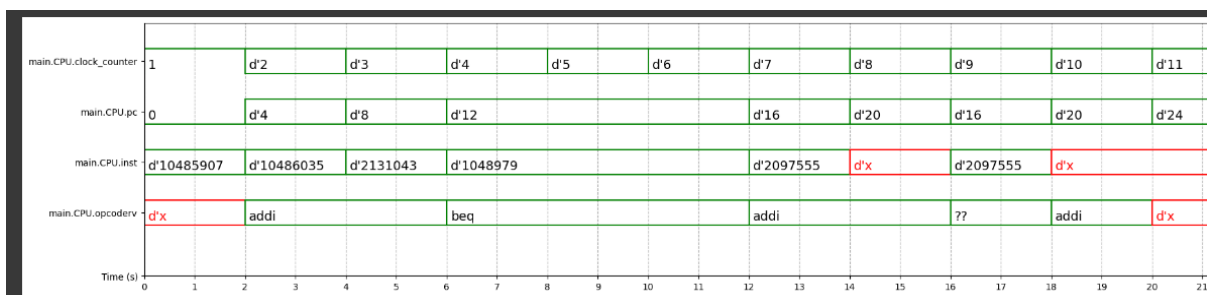
✓ [102] !iverilog main.v
!./a.out

✓ [103] !cat reg.data
0s
// 0x00000000
00000000
00000005
0000000a
00000001
00000000

```

c) **Waveforms:**

Figura 32 - Waveform referente ao primeiro teste



Considerações Finais

A ausência de uma documentação mais clara sobre a produção dos waveforms foi um impeditivo para que a geração dos mesmos pudesse nos dar resultados claros e fidedignos aos testes. Dessa forma, seria interessante que houvesse, nos próximos trabalhos, uma melhor explicação sobre o funcionamento da biblioteca.

Ressaltamos também que nos testes relacionados a instrução BEQ, onde caberia usar uma instrução JUMP, não o fizemos pois em testes com a instrução verificamos que ela realizava o salto porém dava continuidade às próximas instruções, caracterizando assim, uma configuração incorreta da mesma. Como o erro já estava presente no arquivo base do trabalho, e não era necessário, nem solicitado, modificar a instrução, optamos por não utilizá-la em nossos testes.

Diante disso, todas as instruções que deveriam ter um funcionamento correto, foram testadas diversas vezes para garantir sua completude.