

## RELATÓRIO - PROJETO\_IH\_RISC-V

Nome dos integrantes do grupo:

Giovanna de Cassia Silva - gcs5

Thyago Barbosa Soares - tbs3

Wilton Alves Sales - was7

Link para o fork do repositório no GitHub:

[https://github.com/thyagobs/Projeto\\_IH\\_RISC-V.git](https://github.com/thyagobs/Projeto_IH_RISC-V.git)

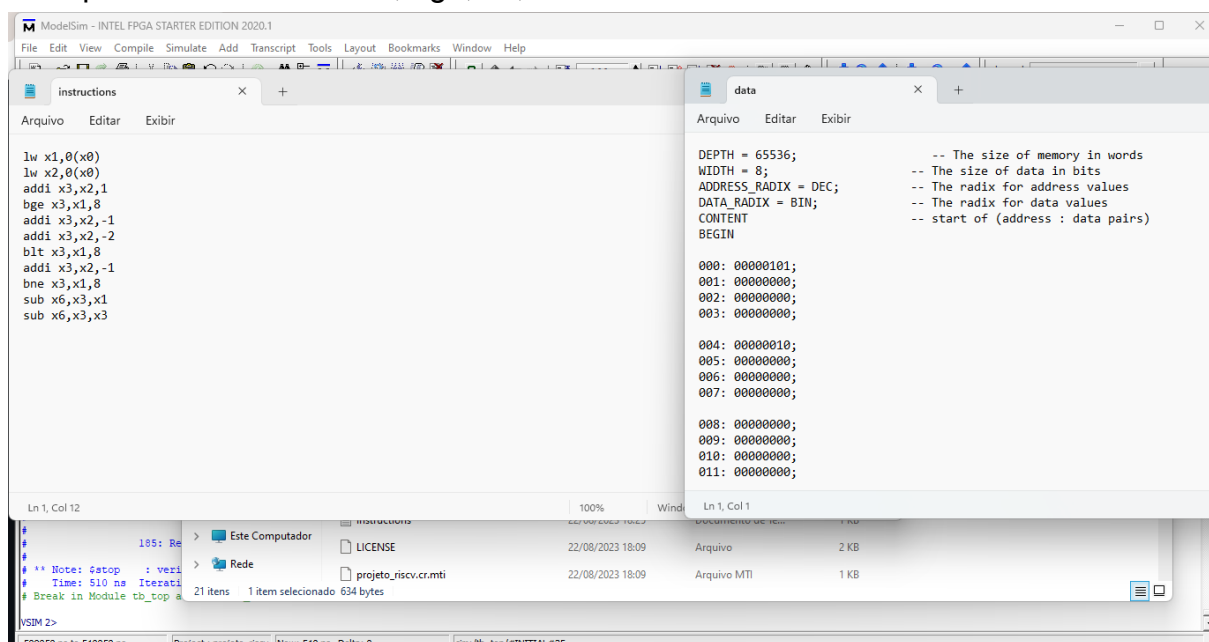
Descrição das escolhas do projeto:

Decidimos em conjunto fazer primeiro a implementação de instruções que para nós são mais fáceis, que nesse caso seriam as aritméticas e lógicas. Depois tentamos implementar as instruções que se baseiam em deslocamento, como slli, slri..., logo após as de load, e assim fomos concluindo as implementações das instruções.

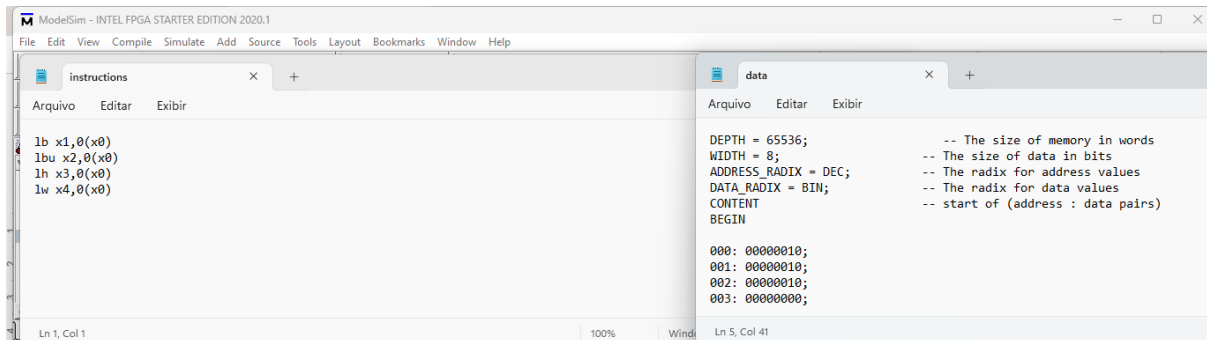
Descrição dos testes realizados:

Para testar as implementações feitas, a maioria das vezes usamos testes “padrões”, focamos primeiro em fazer testes simples e aos poucos progredindo o nível de dificuldade. Exemplo: testar shifts com 1 deslocamento apenas, depois ir aumentando ou carregar valores simples na memória como: em 00: carregar  $2^0$ , em 01: carregar  $2^9$  e em 02: carregar  $2^{19}$ , logo fazendo o teste de lw,lb,lh e lbu eles deveriam resultar os valores corretos, entre outros.

Abaixo segue alguns exemplos de testes feitos para as instruções pedidas: exemplo 1 envolvendo addi, bge, blt, bne e sub:



exemplo 2, envolvendo lb, lbu, lh:



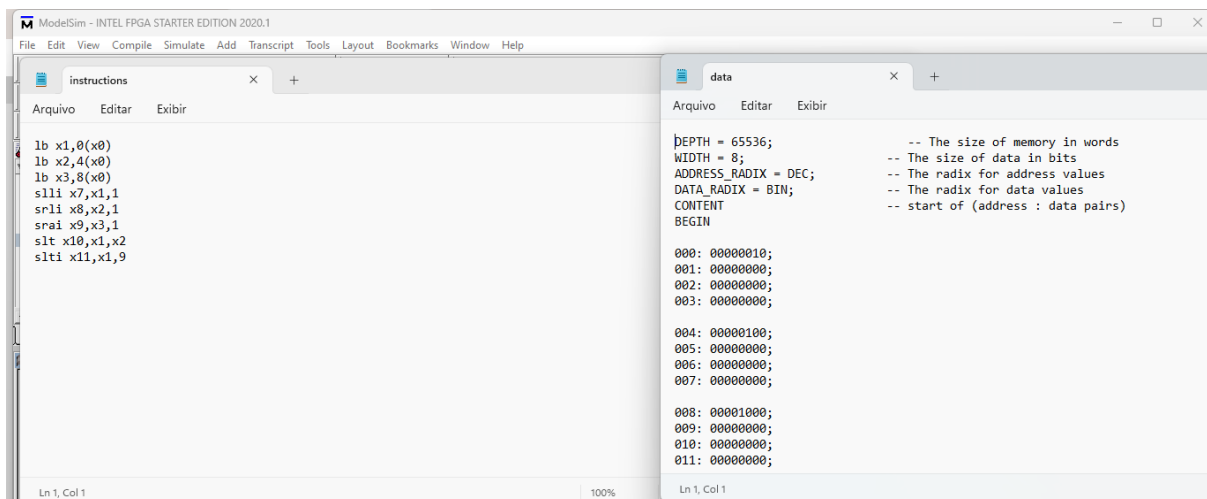
The screenshot shows the ModelSim - INTEL FPGA STARTER EDITION 2020.1 interface. The 'instructions' window on the left contains the following assembly code:

```
lb x1,0(x0)
lbu x2,0(x0)
lh x3,0(x0)
lw x4,0(x0)
```

The 'data' window on the right shows memory configuration and content:

```
DEPTH = 65536;           -- The size of memory in words
WIDTH = 8;               -- The size of data in bits
ADDRESS_RADIX = DEC;     -- The radix for address values
DATA_RADIX = BIN;        -- The radix for data values
CONTENT
BEGIN
000: 00000010;
001: 00000010;
002: 00000010;
003: 00000000;
```

exemplo 3 envolvendo slli, srli, srai, slt e slti:



The screenshot shows the ModelSim - INTEL FPGA STARTER EDITION 2020.1 interface. The 'instructions' window on the left contains the following assembly code:

```
lb x1,0(x0)
lb x2,4(x0)
lb x3,8(x0)
slli x7,x1,1
srli x8,x2,1
srai x9,x3,1
slt x10,x1,x2
slti x11,x1,9
```

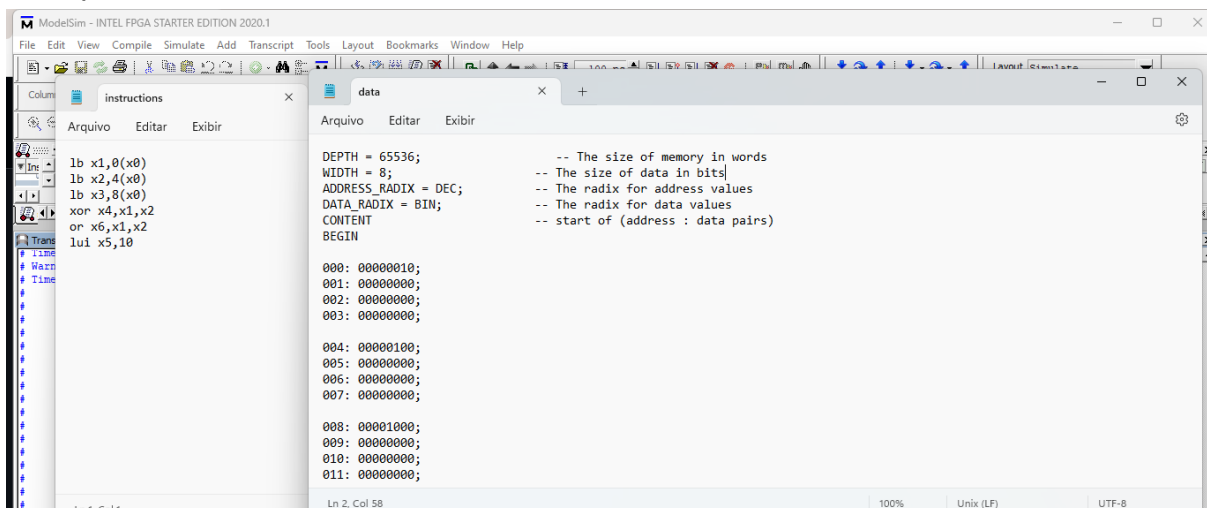
The 'data' window on the right shows memory configuration and content:

```
DEPTH = 65536;           -- The size of memory in words
WIDTH = 8;               -- The size of data in bits
ADDRESS_RADIX = DEC;     -- The radix for address values
DATA_RADIX = BIN;        -- The radix for data values
CONTENT
BEGIN
000: 00000010;
001: 00000000;
002: 00000000;
003: 00000000;

004: 00000100;
005: 00000000;
006: 00000000;
007: 00000000;

008: 00001000;
009: 00000000;
010: 00000000;
011: 00000000;
```

exemplo 4 envolvendo xor, or e lui:



The screenshot shows the ModelSim - INTEL FPGA STARTER EDITION 2020.1 interface. The 'instructions' window on the left contains the following assembly code:

```
lb x1,0(x0)
lb x2,4(x0)
lb x3,8(x0)
xor x4,x1,x2
or x6,x1,x2
lui x5,10
```

The 'data' window on the right shows memory configuration and content:

```
DEPTH = 65536;           -- The size of memory in words
WIDTH = 8;               -- The size of data in bits
ADDRESS_RADIX = DEC;     -- The radix for address values
DATA_RADIX = BIN;        -- The radix for data values
CONTENT
BEGIN
000: 00000010;
001: 00000000;
002: 00000000;
003: 00000000;

004: 00000100;
005: 00000000;
006: 00000000;
007: 00000000;

008: 00001000;
009: 00000000;
010: 00000000;
011: 00000000;
```

Resultados obtidos:

resultado do exemplo 1:

The screenshot shows the ModelSim interface with a Verilog testbench file `tb_top.v` open. The testbench code is as follows:

```
43 $stop;
44 end
45
46 always_comb begin : MEMORY
47   if (wr == ~rd)
48     $display($time, ": Memory [%d] written with value: [%X] | [%d]\n", addr, wr_data, wr_data);
49   else if (rd == ~wr)
50     $display($time, ": Memory [%d] read with value: [%X] | [%d]\n", addr, rd_data, rd_data);
51   end
52 end : MEMORY
```

The Transcript window shows the simulation results:

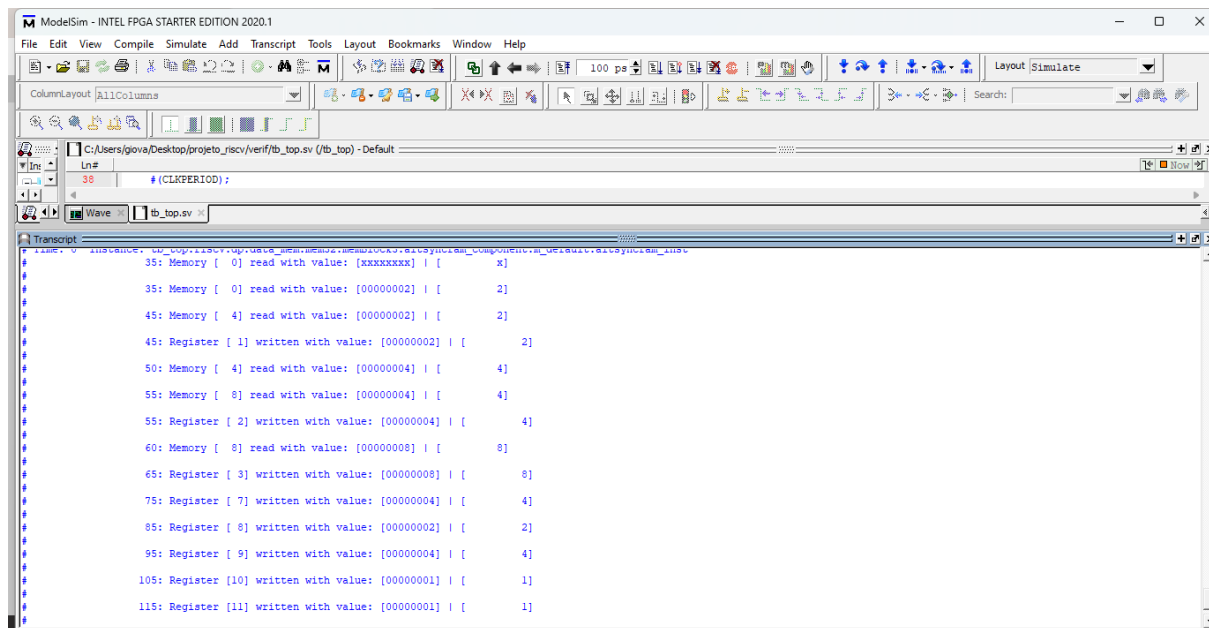
```
# Warning: read_during_write_mode_mixed_ports is assumed as OLD_DATA
# Time: 0 Instance: tb_top.riscv.dp.data_mem.mem32.memBlock2.altysyncram_component.m_default.altysyncram_inst
# Warning: read_during_write_mode_mixed_ports is assumed as OLD_DATA
# Time: 0 Instance: tb_top.riscv.dp.data_mem.mem32.memBlock3.altysyncram_component.m_default.altysyncram_inst
35: Memory [ 0] read with value: [xxxxxxxx] | [ x]
#
35: Memory [ 0] read with value: [00000005] | [ 5]
#
45: Register [ 1] written with value: [00000005] | [ 5]
55: Register [ 2] written with value: [00000005] | [ 5]
#
75: Register [ 3] written with value: [00000006] | [ 6]
#
115: Register [ 3] written with value: [00000003] | [ 3]
#
185: Register [ 6] written with value: [00000000] | [ 0]
#
** Note: $stop : verif/tb_top.v(43)
Time: 510 ns Iteration: 0 Instance: /tb_top
```

resultado exemplo 2:

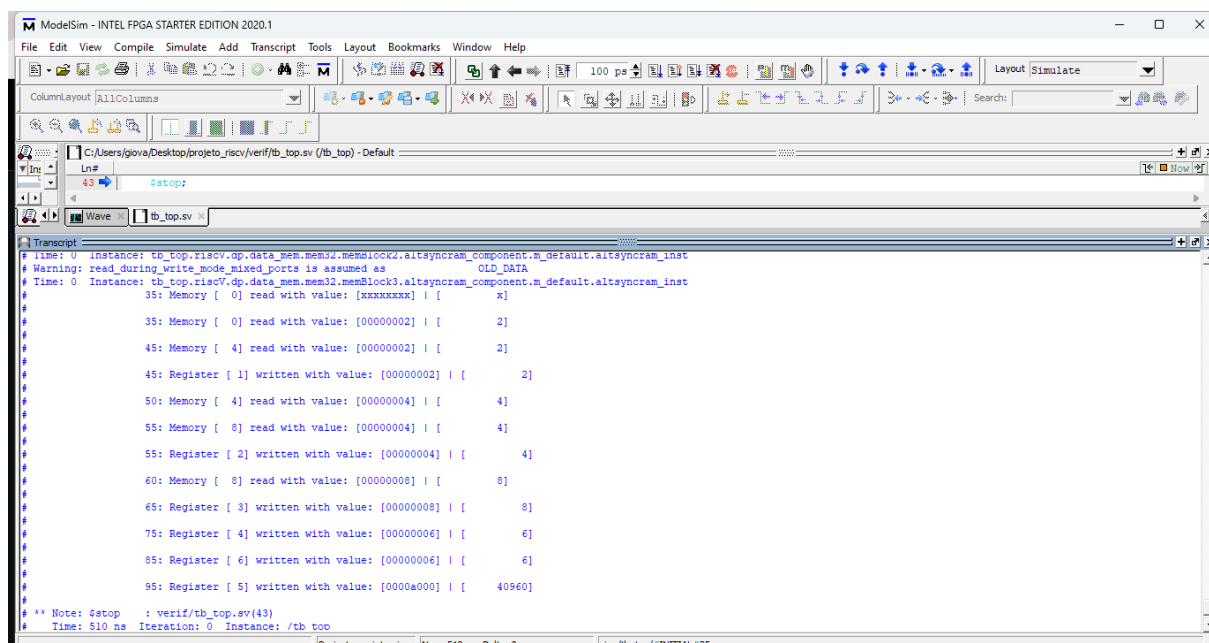
The screenshot shows the Transcript window with the following simulation results:

```
# Warning: read_during_write_mode_mixed_ports is assumed as OLD_DATA
# Time: 0 Instance: tb_top.riscv.dp.data_mem.mem32.memBlock3.altysyncram_component.m_default.altysyncram_inst
35: Memory [ 0] read with value: [xxxxxxxx] | [ x]
#
35: Memory [ 0] read with value: [00000002] | [ 2]
#
45: Register [ 1] written with value: [00000002] | [ 2]
55: Register [ 2] written with value: [00000002] | [ 2]
#
55: Memory [ 0] read with value: [00000202] | [ 514]
65: Register [ 3] written with value: [00000202] | [ 514]
#
65: Memory [ 0] read with value: [00020202] | [ 131586]
75: Register [ 4] written with value: [00020202] | [ 131586]
#
** Note: $stop : verif/tb_top.v(43)
Time: 510 ns Iteration: 0 Instance: /tb_top
```

### resultado exemplo 3:



### resultado exemplo 4:



### Dificuldades encontradas:

Em um primeiro momento, o entendimento de como funcionava o código fonte foi uma das dificuldades que enfrentamos. Após isso, algumas instruções exigiram um pouco mais de esforço para serem concluídas (implementadas, testadas e funcionando) como as de branches e loads. E para nós as que consideramos difíceis de até mesmo implementar foram as de store, jal, jalr e halt.

Conclusão:

- Conseguimos implementar 18/21 instruções, sendo elas:  
BNE, BGE, BLT, LB, LH, LBU, SB, SH, SLTI, ADDI, SLLI, SRLI, SRAI, SUB, SLT, XOR, OR, LUI
- Conseguimos testar todas essas acima, exceto as de store ( SB e SH).
- Não conseguimos implementar as instruções JAL, JALR e HALT

Apesar das dificuldades encontradas, o projeto nos ajudou muito mais a aprender sobre o pipeline em si do que apenas a leitura do livro indicado no curso.