Relatório do Trabalho Final de Organização e Arquitetura de Computadores. Turma C.
Nome: Alec Ryo Emura Matrícula: 15/0115326
Matheus de Feitosa Castro 15/0141297
Projeto RISC-V Uniciclo em VHDL

Foi implementado um processador RISC-V UNICICLO em VHDL capaz de realizar as instruções descritas abaixo. Junto com este relatório está anexo o projeto em VHDL:

INSTRUÇÕES A SEREM IMPLEMENTADAS

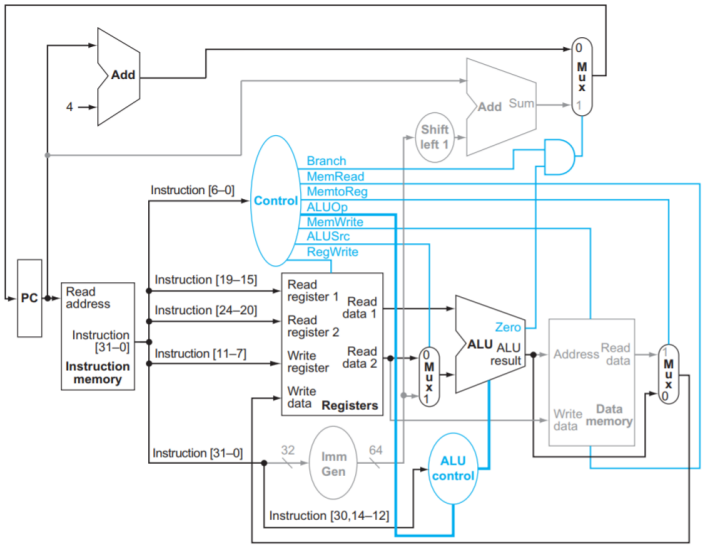
- Lógico-Aritméticas: ADD, ADDi, SUB, AND, ANDi, LUI, SLT, OR, ORi, XOR, XORi, SLLi, SRLi, SRAi, SLTi, SLTu, SLTUi
- Subrotinas: JAL, JALR
- Saltos: BEQ, BNE, BLT, BGE
- Memória: LW, SW

COMPONENTES:

Para a realização do projeto, este, foi subdividido em vários componentes, sendo eles os seguintes:

- PC
- adder32
- memIns
- control
- genImm32
- mux2x1
- cntrlULA
- XREGS
- ULA_RV
- memDados

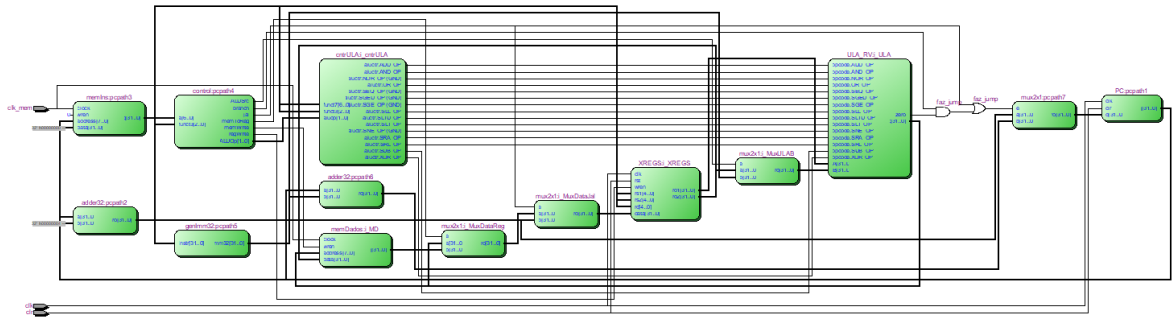
O subcomponentes foram conectados seguindo como base o seguinte esquemático, posteriormente foram implementadas novas instruções. Mais a frente será mostrado um esquemático:



BREVE COMENTÁRIOS SOBRE OS COMPONENTES:

- PC: Este componente possui como função principal controlar o endereço da memória de instrução.
- adder32: A função deste componente é realizar a soma de dois sinais de 32 bits com sinal.
- memIns: Este componente armazena todas as instruções que serão executadas pelo processador.
- control: A função deste componente é definir sinais de controles essenciais para o funcionamento do processador, tais como sinais que permitem escrita da memória ou sinalizam um salto do PC.
- genImm32: Este componente gera o imediato de um dado sinal.
- mux2x1: A função deste componente é selecionar um sinal dentre dados dois sinais de entrada.
- cntrlULA: A função deste componente é informar a ULA qual instrução será executada.
- XREGS: Este componente é responsável pela manutenção dos 32 registradores de memória do processador.
- ULA_RV: A função deste componente é executar funções lógico-aritméticas relativas ao conjunto de instruções do processador.
- memDados: Este componente armazena os dados em memória do processador.

VISÃO RTL DA IMPLEMENTAÇÃO:



CÓDIGO DE TESTE:

```
.text                       # PC       MI       ULA      MD

ori  t0, zero, 0xFF         # 00000000  0ff06293   000000ff  00000000
andi t0, t0, 0xF0           # 00000004  0f02f293   000000f0  00000000
lui  s0, 2                  # 00000008  00002437   00002000  00000000
lw   s1, 0(s0)              # 0000000c  00042483   00002000  0000000f
lw   s2, 4(s0)              # 00000010  00442903   00002004  0000003f
add  s3, s1, s2             # 00000014  012489b3   0000004e  00000000
sw   s3, 8(s0)              # 00000018  01342423   00002008  0000004e
lw   a0, 8(s0)              # 0000001c  00842503   00002008  0000004e
```

```
  addi s4, zero, 0x7F0        # 00000020  7f000a13   000007f0  00000000
  addi s5, zero, 0x0FF        # 00000024  0ff00a93   000000ff  00000000
  and  s6, s5, s4             # 00000028  014afb33   000000f0  00000000
  or   s7, s5, s4             # 0000002c  014aebb3   000007ff  00000000
  xor  s8, s5, s4             # 00000030  014acc33   0000070f  00000000
  slli  t1, s5, 4             # 00000034  004a9313   00000ff0  00000000
  lui  t2, 0xFF000            # 00000038  ff0003b7   ff000000  00000000
  srli  t3, t2, 4             # 0000003c  0043de13   0ff00000  00000000
  srai  t4, t2, 4             # 00000040  4043de93   fff00000  00000000
  slt  s0, t0, t1             # 00000044  0062a433   00000001  00000000
  slt  s1, t1, t0             # 00000048  005324b3   00000000  00000000
  sltu s3, zero, t0           # 0000004c  005039b3   00000001  00000000
  sltu s4, t0, zero           # 00000050  0002ba33   00000000  00000000

  jal  ra, testasub           # 00000054  008000ef   00000000  00000000 => 5c

  jal  x0, next               # 00000058  00c0006f   00000000  00000000 => 64
testasub:
  sub t3, t0, t1              # 0000005c  40628e33   fffffffe  00000000
  jalr x0, ra, 0              # 00000060  00008067   00000058  00000000 => 58
next:
  addi t0, zero, -2           # 00000064  ffe00293   fffffffe  00000000
beqsim:
  addi t0, t0, 2              # 00000068  00228293   0000000*  00000000 * t0 = 0, 2
  beq  t0, zero, beqsim       # 0000006c  fe028ee3   00000000  00000000 => 68, 70
bnesim:
  addi t0, t0, -1             # 00000070  fff28293   0000000*  00000000 * t0 = 1, 0
  bne  t0, zero, bnesim       # 00000074  fe029ee3   00000000  00000000 => 70, 78
  addi t5, zero, -3           # 00000078 ffd0f13 fffffffd 00000000
bltsim:
  addi t5,t5,1                # 00000007c 001f0f13 fffffffd 00000000
  blt t5, zero, bltsim        # 000000080 fedf48e3 fffffffd 00000000
  addi t6, zero, 3            # 000000084
bgtsim:
  addi t6,t6, -1              # 00000008c
  bgt t6, zero, bnesim        # 000000090
```

TELAS DE SIMULAÇÃO:

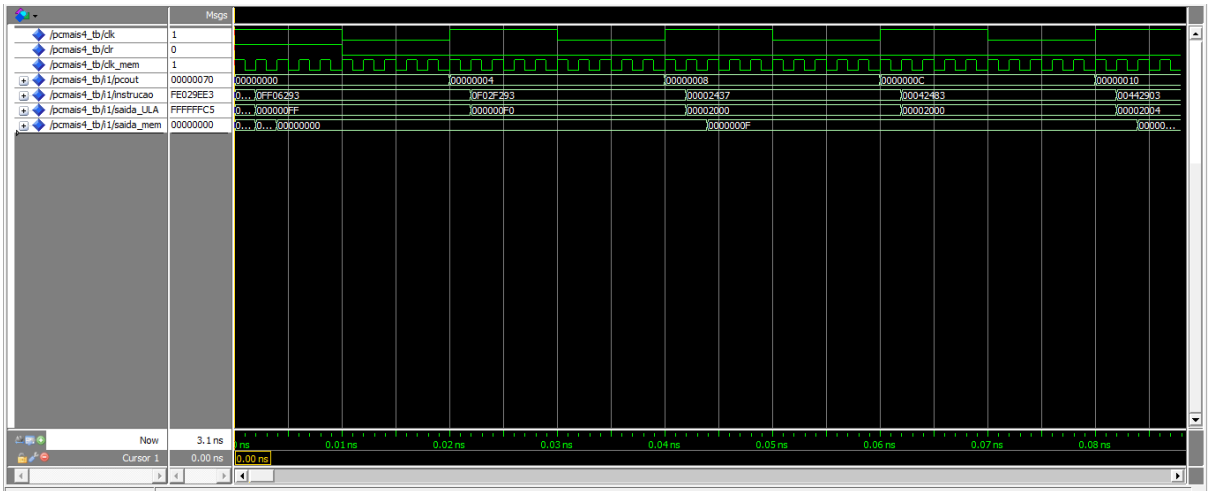
Para testar as instruções rodou-se um programa em Assembly disponibilizado na disciplina. No RARS foi acrescentado linhas para executar e testar os saltos BLT e BGE. Todas as instruções mostraram resultados esperados.

A seguir é mostrado telas onde:

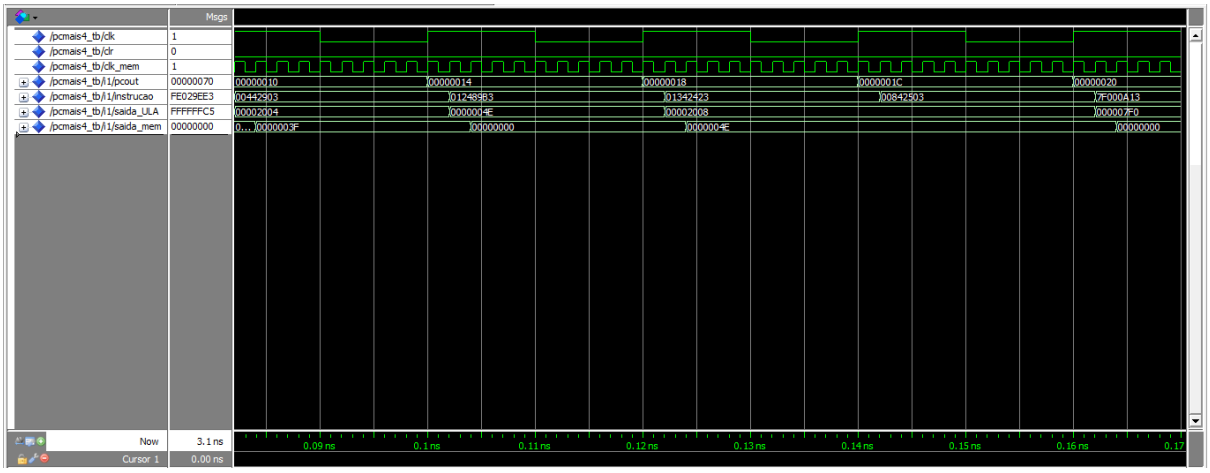
clk: Clock de relógio do processador, conectado no PC e XREG.

clr: Clear

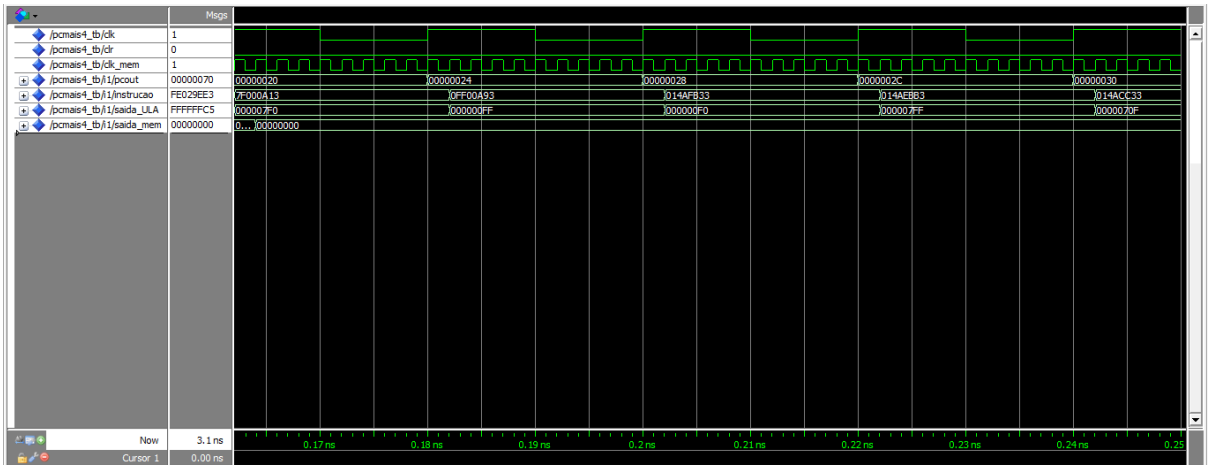
clk_mem: Clock de memória, sendo ela 10 vezes mais rápida para não haver erros que a memória será carregada no mesmo clock que a solicitação.
pcout: PC sendo executado no momento.
instrução: A instrução carregada da memória de instruções.
saida_ULA: resultado da ULA.
saida_mem: saída da memória de dados.



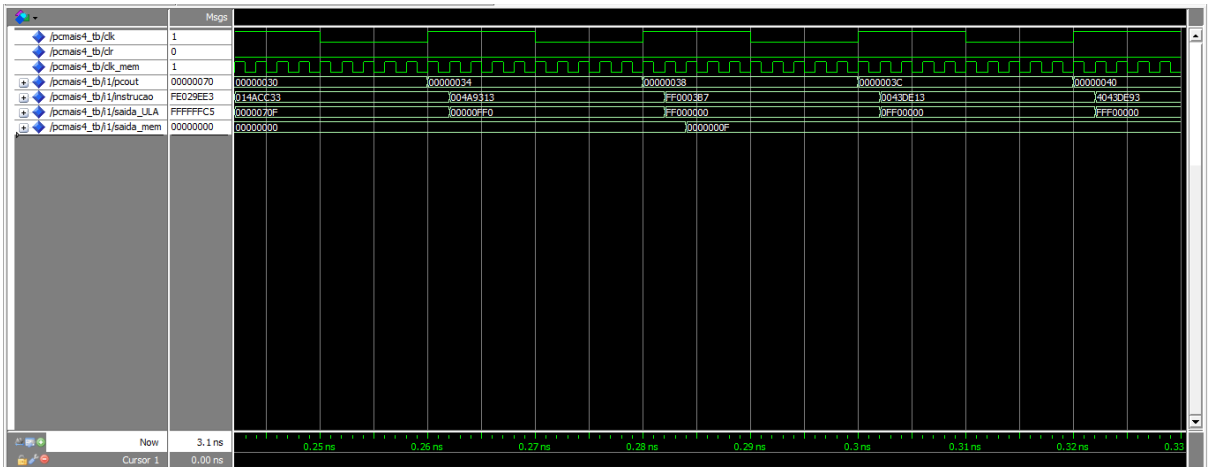

```
ori  t0, zero, 0xFF     # 00000000  0ff06293   000000ff  00000000
andi t0, t0, 0xF0       # 00000004  0f02f293   000000f0  00000000
lui  s0, 2              # 00000008  00002437   00002000  00000000
lw   s1, 0(s0)          # 0000000c  00042483   00002000  0000000f
```



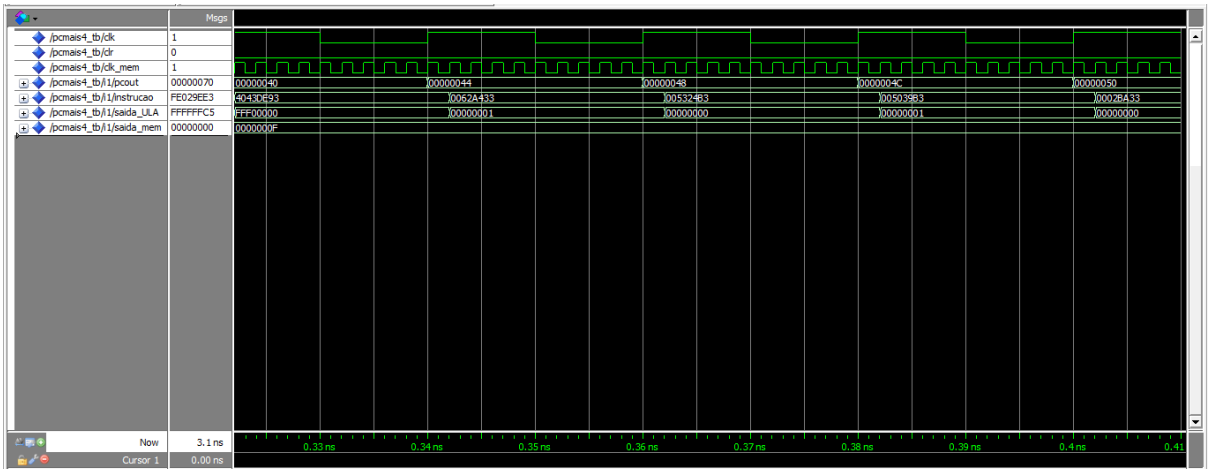

```
lw   s2, 4(s0)          # 00000010  00442903   00002004  0000003f
add  s3, s1, s2         # 00000014  012489b3   0000004e  00000000
sw   s3, 8(s0)          # 00000018  01342423   00002008  0000004e
lw   a0, 8(s0)          # 0000001c  00842503   00002008  0000004e
```



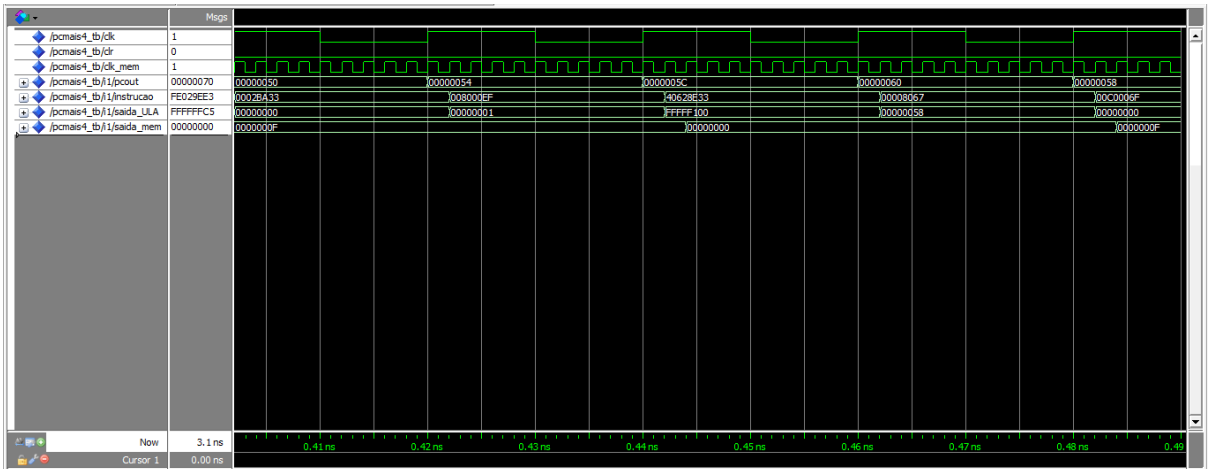
addi s4, zero, 0x7F0 # 00000020 7f000a13 000007f0 00000000
addi s5, zero, 0x0FF # 00000024 0ff00a93 000000ff 00000000
and s6, s5, s4 # 00000028 014afb33 000000f0 00000000
or s7, s5, s4 # 0000002c 014aebb3 000007ff 00000000



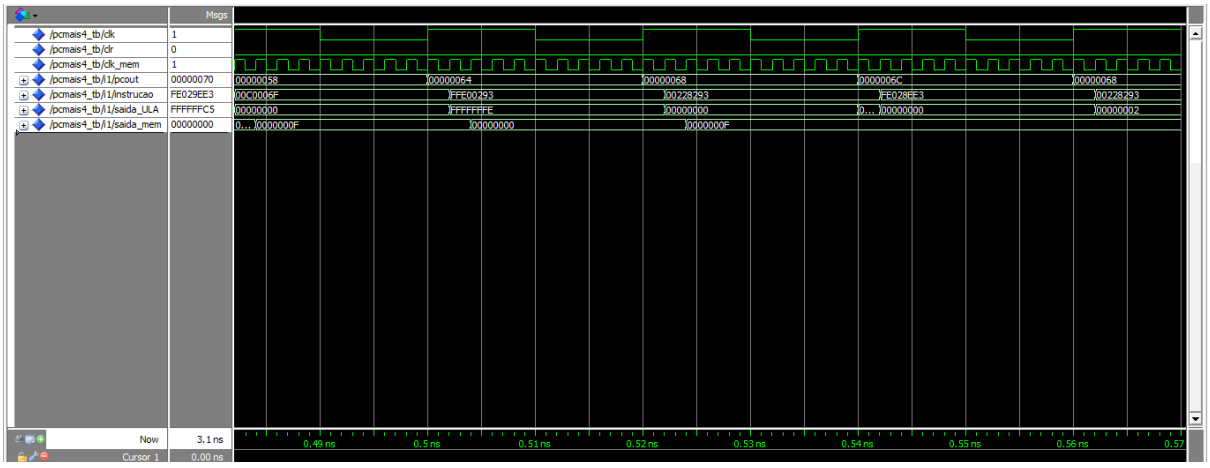
xor s8, s5, s4 # 00000030 014acc33 0000070f 00000000
slli t1, s5, 4 # 00000034 004a9313 00000ff0 00000000
lui t2, 0xFF000 # 00000038 ff0003b7 ff000000 00000000
srli t3, t2, 4 # 0000003c 0043de13 0ff00000 00000000



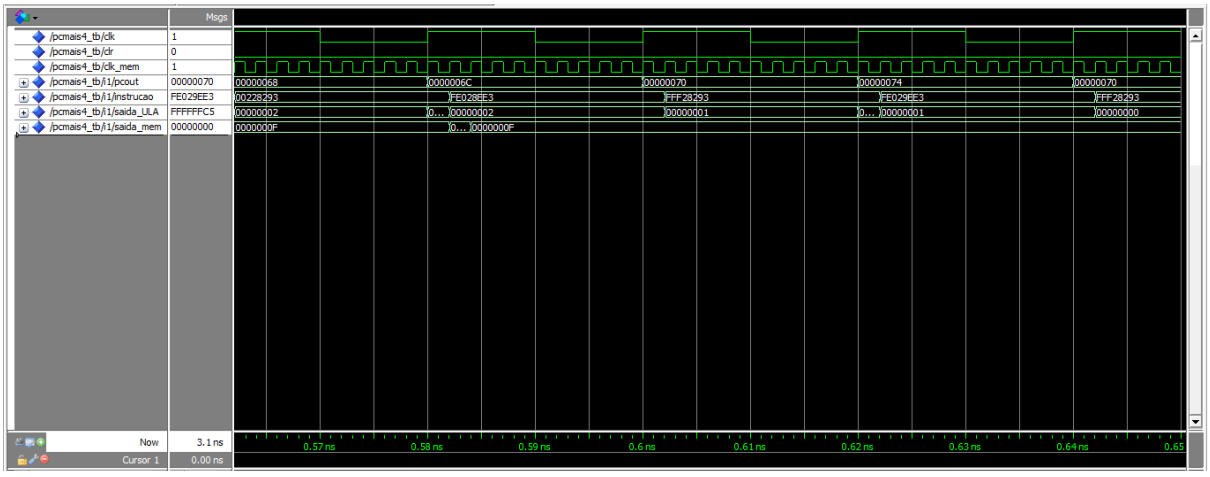
srai t4, t2, 4 # 00000040 4043de93 fff00000 00000000
slt s0, t0, t1 # 00000044 0062a433 00000001 00000000
slt s1, t1, t0 # 00000048 005324b3 00000000 00000000
sltu s3, zero, t0 # 0000004c 005039b3 00000001 00000000



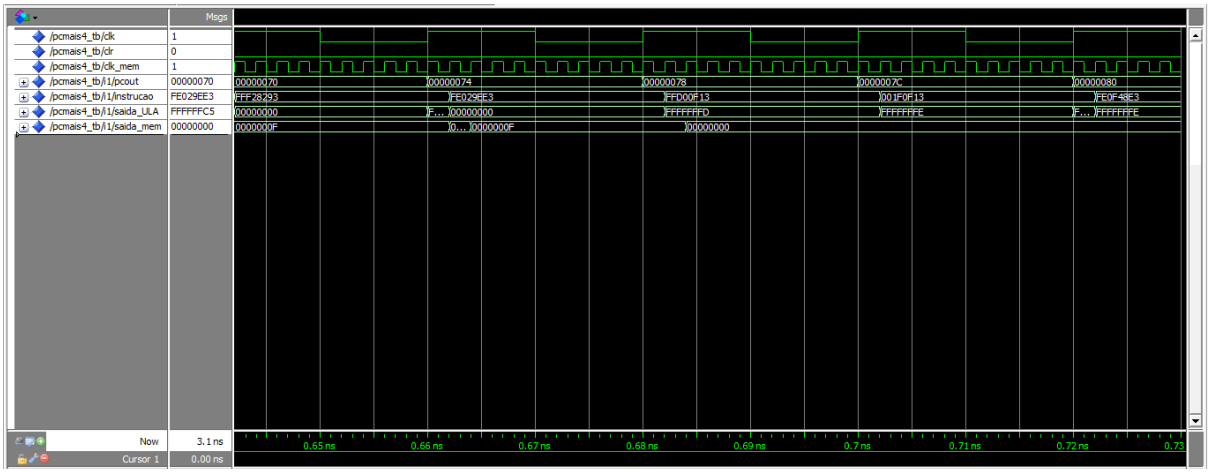
sltu s4, t0, zero # 00000050 0002ba33 00000000 00000000
jal ra, testasub # 00000054 008000ef 00000000 00000000 => 5c
Salto realizado para PC = #0000005C
sub t3, t0, t1 # 0000005c 40628e33 fffffffe 00000000
jalr x0, ra, 0 # 00000060 00008067 00000058 00000000 => 58
Salto realizado para PC = #00000058



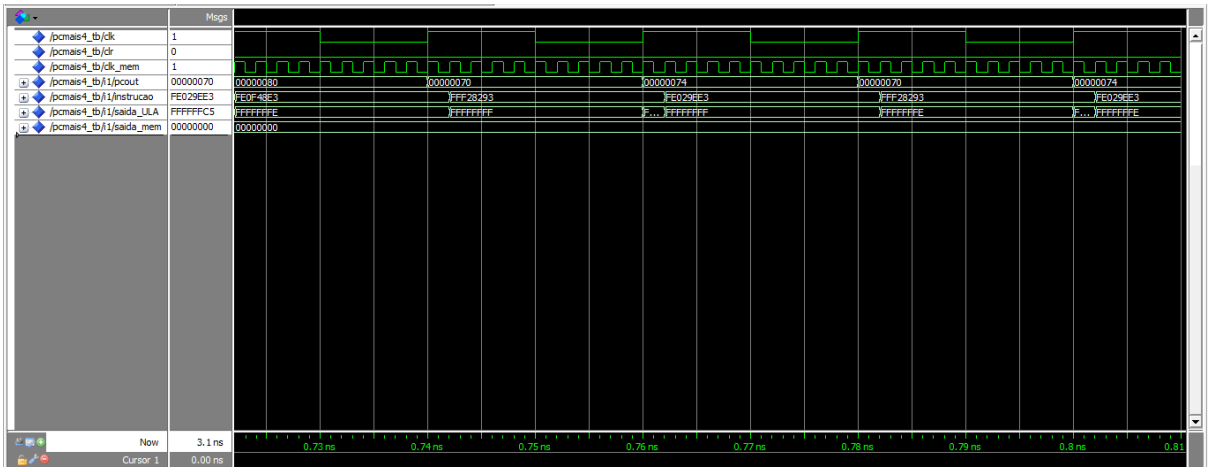
jal x0, next # 00000058 00c0006f 00000000 00000000 => 64
addi t0, zero, -2 # 00000064 ffe00293 fffffffe 00000000
addi t0, t0, 2 # 00000068 00228293 0000000* 00000000 * t0 = 0, 2
beq t0, zero, beqsim # 0000006c fe028ee3 00000000 00000000 => 68, 70
Salto realizado para PC = #00000068
addi t0, t0, 2 # 00000068 00228293 0000000* 00000000 * t0 = 0, 2



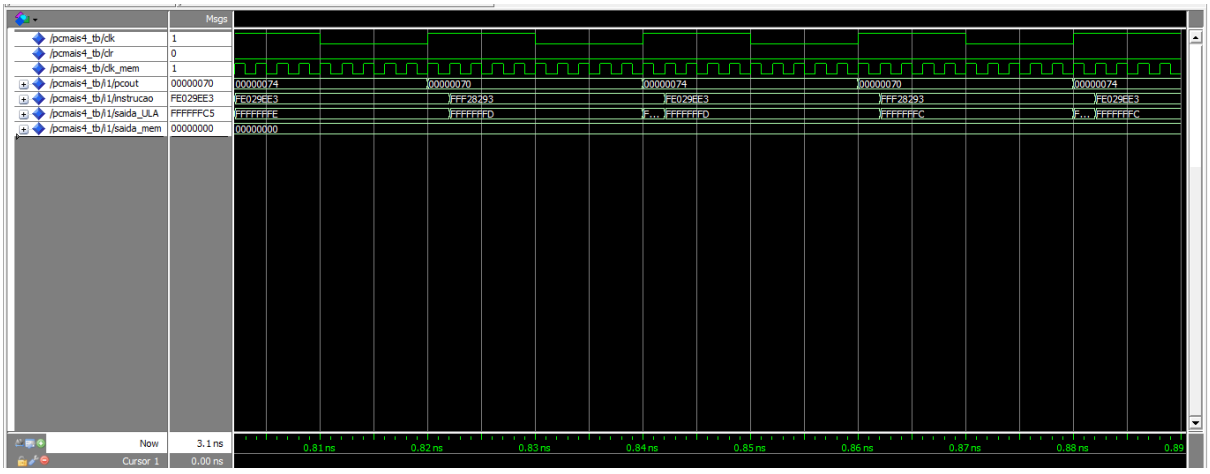
addi t0, t0, 2 # 00000068 00228293 0000000* 00000000 * t0 = 0, 2
beq t0, zero, beqsim # 0000006c fe028ee3 00000000 00000000 => 68, 70
addi t0, t0, -1 # 00000070 fff28293 0000000* 00000000 * t0 = 1, 0
bne t0, zero, bnesim # 00000074 fe029ee3 00000000 00000000 => 70, 78
Salto realizado para PC = #00000070
addi t0, t0, -1 # 00000070 fff28293 0000000* 00000000 * t0 = 1, 0



addi t0, t0, -1 # 00000070 fff28293 0000000* 00000000 * t0 = 1, 0
bne t0, zero, bnesim # 00000074 fe029ee3 00000000 00000000 => 70, 78
addi t5, zero, -3 # 00000078 ffd0f13 fffffffd 00000000
addi t5,t5,1 # 0000007c 001f0f13 fffffffd 00000000
blt t5, zero, bltsim # 00000080 fedf48e3 fffffffd 00000000



blt t5, zero, bltsim # 000000080 fedf48e3 fffffffd 00000000
Salto realizado para PC = #00000070
addi t0, t0, -1 # 00000070 fff28293 0000000* 00000000 * t0 = 1, 0
bne t0, zero, bnesim # 00000074 fe029ee3 00000000 00000000 => 70, 78
Salto realizado para PC = #00000070
addi t0, t0, -1 # 00000070 fff28293 0000000* 00000000 * t0 = 1, 0
bne t0, zero, bnesim # 00000074 fe029ee3 00000000 00000000 => 70, 78



bne t0, zero, bnesim # 00000074 fe029ee3 00000000 00000000 => 70, 78
Salto realizado para PC = #00000070
addi t0, t0, -1 # 00000070 fff28293 0000000* 00000000 * t0 = 1, 0
bne t0, zero, bnesim # 00000074 fe029ee3 00000000 00000000 => 70, 78
Salto realizado para PC = #00000070
addi t0, t0, -1 # 00000070 fff28293 0000000* 00000000 * t0 = 1, 0
bne t0, zero, bnesim # 00000074 fe029ee3 00000000 00000000 => 70, 78

IMPLEMENTAÇÃO DE: JAL, JALR, LUI, BEQ, BNE, BLT, BGT
Para a implementação de JAL e JALR foi colocado:

- Um multiplexador pro jal e jalr tendo:
  entrada1 = saída do mux mem ou ULA
  entrada2 = PC+4

seleção = sinal de controle de jal ou jalr
saída = mux pra lui

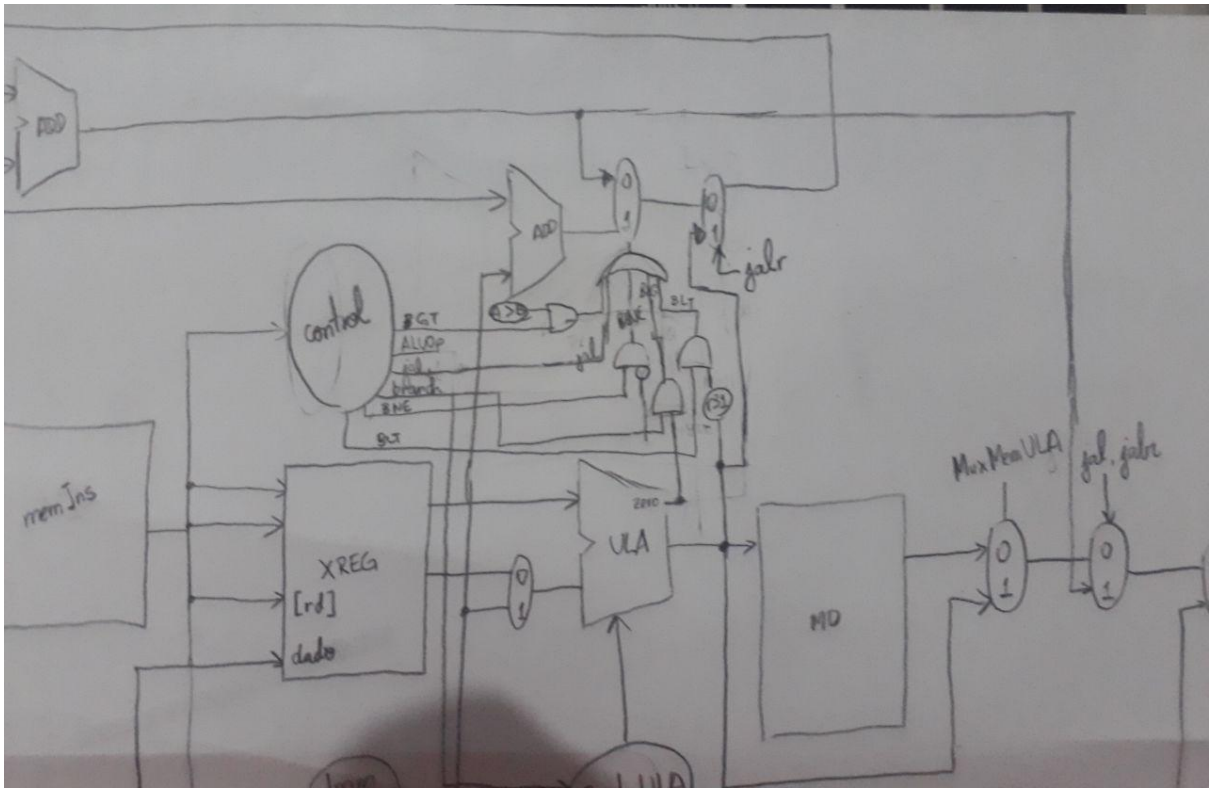
- Um multiplexador pro lui tendo:
  entrada1 = saída do mux de jal/jalr
  entrada2 = dado imediato
  seleção = sinal de controle de lui
  saída = entrada de dado em XREG

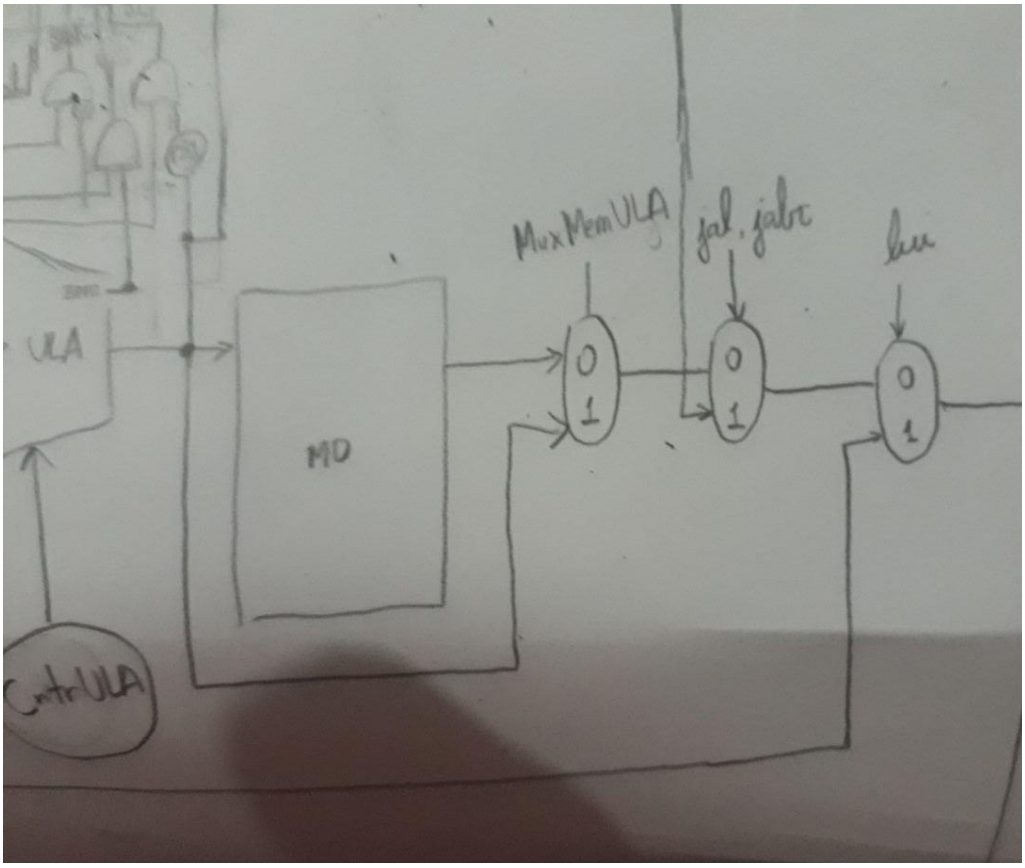
- Na seleção do multiplexador para PC receber ou endereço de pulo ou PC+4 temos uma porta OR com:

  faz_jump <= jal or (branch and zero) or jalr or (not(zero) and bne) or (blt and saida_ULA(31)) or (bgt and se_saidaA>saidaB)

- Um multiplexador selecionando como entrada o endereço calculado pela ULA no caso de ser uma instrução jalr.

  A seguir segue um esquemático para endereçamento de PC e armazenamento no banco de registradores





CONCLUSÃO:

Com os dados apresentados podemos concluir que a simulação do processador, para as funções requeridas, foi realizada com sucesso.