Relatório Ténico RV32I Single-Cycle

Feito por: Hyago Vieira Lemes Barbosa Silva
e-mail: hyago.silva@mtel.inatel.br | hyagobora@gmail.com

Objetivo do projeto

Implementar um processador RISC-V RV32I de 32 bits, single-cycle, em SystemVerilog, capaz de executar um subconjunto essencial de instrucoes R-type, I-type, load/store, branch, jump e U-type.

Dados:

- Nome: Hyago Vieira Lemes Barbosa Silva
- N = 24
- A = N + 4 = 28
- B = N + 2 = 26
- C = 4 ∗ N = 96
- MEM_ADDR = 4 ∗ N = 96
- Como a memoria e enderecada por palavra, o endereco 96 acessa `memory[96 >> 2] = memory[24]`. Ou seja basicamente em 32 bits, eu pego 96 desloco de 2, eu divido por 4, por isso memory[24].

Arquitetura single-cycle

O processador executa cada instrucao em um unico ciclo de clock. Em um ciclo, a instrucao e buscada, decodificada, os registradores sao lidos, a ALU executa a operacao, a memoria de dados pode ser acessada e o resultado pode ser escrito no banco de registradores.

O Program Counter inicia em zero apos reset. Em execucao normal, o proximo PC e `PC + 4`. Para branch tomado, o proximo PC e `PC + imediato B-type`. Para `jal`, o proximo PC e `PC + imediato J-type`, e `PC + 4` e salvo em `rd`.

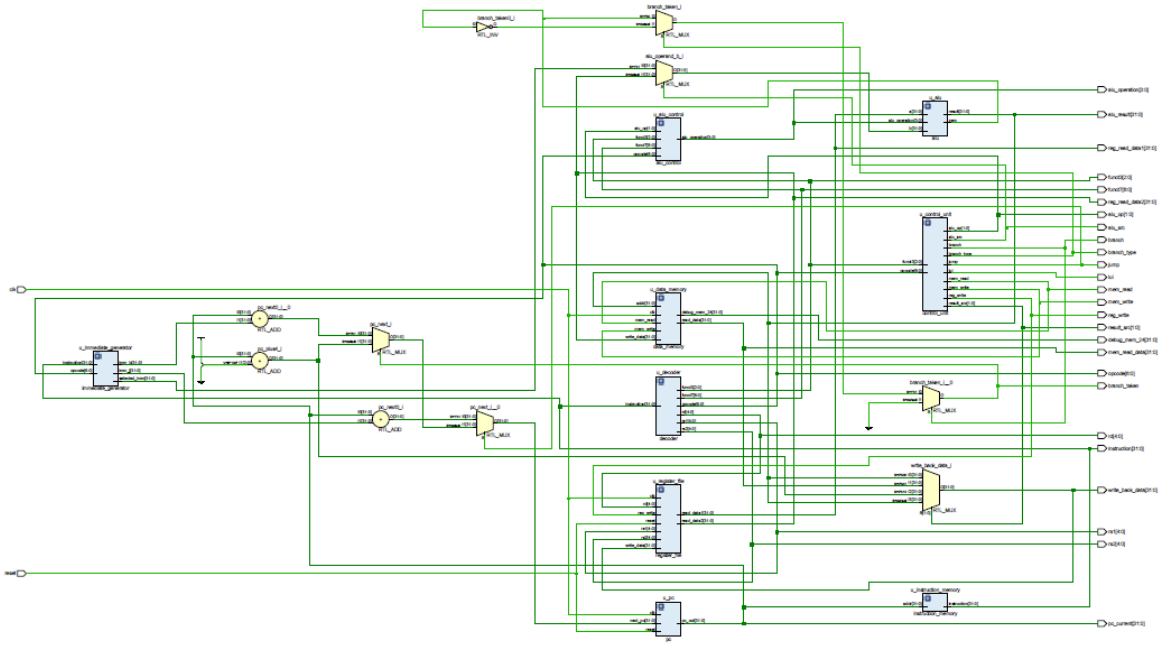
Modulos implementados

- `pc.sv`: registrador do Program Counter com reset para zero.
- `instruction_memory.sv`: ROM carregada com `$readmemh("mem/program.mem")`.
- `decoder.sv`: extrai `opcode`, `rd`, `funct3`, `rs1`, `rs2` e `funct7`.
- `immediate_generator.sv`: gera imediatos I, S, B, U e J com extensao correta.
- `control_unit.sv`: gera sinais de controle do datapath.
- `alu_control.sv`: escolhe a operacao da ALU usando `alu_op`, `opcode`, `funct3` e `funct7`.
- `alu.sv`: executa ADD, SUB, AND, OR, XOR, SLT signed e PASS_B.
- `register_file.sv`: banco com 32 registradores, duas leituras assincronas, escrita sincrona e `x0` fixo em zero.
- `data_memory.sv`: RAM de dados com suporte a `lw` e `sw`.
- `rv32i_single_cycle_top.sv`: integra todos os blocos e implementa next PC e write-back.
- `tb_rv32i_single_cycle.sv`: testbench com clock, reset, VCD, displays e verificacoes automaticas.

Esquemático

Usando VIVADO.

schematic.pdf



Programa de teste

O arquivo `mem/program.mem` contem o programa abaixo em hexadecimal, uma instrucao por linha:

```
01c00093
01a00113
002081b3
40208233
0020f2b3
0020e333
0020c3b3
00112433
06302023
06002483
00348463
00000513
00100513
008005ef
06300613
06000693
```

Esse programa calcula operacoes com A = 28 e B = 26, grava o resultado da soma na memoria, le de volta, testa `beq`, testa `jal` e finaliza gravando C = 96 em `x13`.

Seguindo exatamente conforme o definido pelo projeto

Modelo esperado do programa:

```
addi x1, x0, A          # x1 = A
addi x2, x0, B          # x2 = B

add  x3, x1, x2         # x3 = A + B
sub  x4, x1, x2         # x4 = A - B
and  x5, x1, x2         # x5 = A AND B
or   x6, x1, x2         # x6 = A OR B
xor  x7, x1, x2         # x7 = A XOR B
slt  x8, x2, x1         # x8 = 1 se B < A

sw   x3, MEM_ADDR(x0)   # armazena x3 na memória
lw   x9, MEM_ADDR(x0)   # carrega o valor salvo em x9

beq  x9, x3, label_ok   # testa branch
addi x10, x0, 0         # indica erro caso o branch não ocorra

label_ok:
  addi x10, x0, 1       # indica sucesso no teste de memória/branch

jal  x11, fim           # testa jump e salva PC + 4 em x11
addi x12, x0, 99        # esta instrução deve ser pulada se o jal funcionar

fim:
addi x13, x0, C         # valor final individualizado
```

Execução da simulação

Usando ModelISIM. O testbench gera o arquivo de waveform:

O que observar na waveform

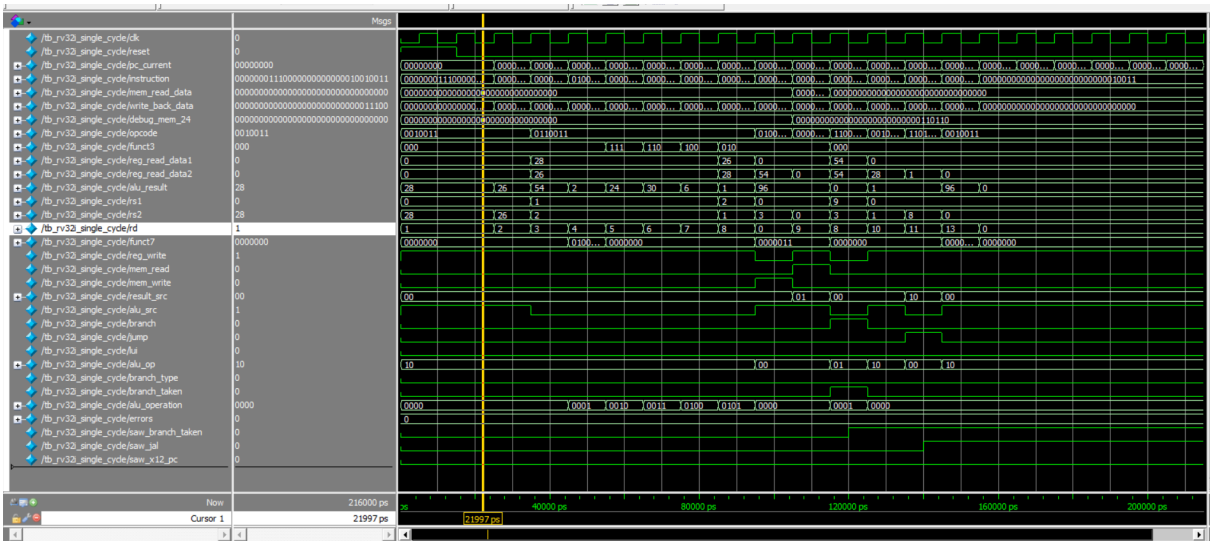
Sinais importantes:

- `pc_current`: sequencia normal de PC e desvios.
- `instruction`: instrucao buscada na ROM.
- `reg_write`, `mem_read`, `mem_write`, `branch`, `branch_taken`, `jump`: sinais de controle.

- `alu_result`: resultado da ALU ou endereco calculado para memoria.
- `mem_read_data`: dado lido por `lw`.
- `write_back_data`: valor escrito em `rd`.
- `debug_mem_24`: posicao `memory[24]`, que deve receber 54.

Durante o `beq`, `branch_taken` deve ficar em 1. Durante o `jal`, `jump` deve ficar em 1 e o PC deve ir para o endereco final, sem executar a instrucao que escreveria 99 em `x12`.

Simulação



Instruções

Primeira instrução

- **01c00093 é lido em mem/program.mem, processa dor lê este hexadecimal** = bits **32** → **01c00093** = **00000001110000000000000010010011.**
- Este arquivo program.mem é carregado pela memória de instruções. com:

```
$readmemh(PROGRAM_FILE, memory);
```

- PC começa em zero, de acordo com reset. Portanto pc_current = 0, será buscado a instrução que esta no endereço 0.

- Intruction memory busca atraves de:

```
instruction_memory u_instruction_memory (
    .addr        (pc_current),
    .instruction (instruction)
);
```

```
    assign instruction = memory[addr[31:2]];
    // como addr=0 memory[0] = 01c00093
```

- O decoder quebra a instrução em pedaços

```
assign opcode = instruction[6:0];
assign rd     = instruction[11:7];
assign funct3 = instruction[14:12];
assign rs1    = instruction[19:15];
assign rs2    = instruction[24:20];
assign funct7 = instruction[31:25];
```

A instrução 01c00093 em binário é 00000001110000000000000010010011. Onde:

- **opcode** = **0010011** instrução **tipo i aritimética.**
- **rd** = **00001** (destino **x1**).
- **func3** = **000** operação addi.
- **rs1** = **00000** origem x0 valor 0 em decimal por padrão RiscV
- **imm** = **28** ⇒ **000000011100**. Imediato justamente para operação addi.
- Quando alu_src = 1, a ALU ignora o valor de rs2 e usa o imediato.
- **01c00093** = **addi x1, x0, 28** ou seja nessa linha o processador faz, **x1 = 0 + 28**.
    - Como o registrador **x0** no **RISC-V sempre vale zero**.
- **Armazenando** o valor de **A = N + 2 = 24 + 2 = 28** no registrador **X1, rd = 00001 (x1)**.
- Justamente aquela operação:

- **addi x1(rd), x0(rs1), A(imm). # X1 = A**

Referência: https://docs.riscv.org/reference/isa/unpriv/rv-32-64g.html

- **Immediate Generator pega o número 28**

```
imm_i = {{20{instruction[31]}}, instruction[31:20]};
...
instruction[31:20] = 000000011100
...
//28 em decimal através do opcode
selected_imm = 28
```

- Control Unit liga os sinais corretos. No código, quando opcode = 0010011. Ele cai no caso:

```
OPCODE_ITYPE: begin
    reg_write = 1'b1;
    alu_src   = 1'b1;
    alu_op    = 2'b10;
end
```

- reg_write = 1 vai escrever em x1 (rd) destino.
- alu_src = 1 vai usar o imediato
- alu_op = 10 operação aritimética ou lógica

- **Register File lê o x0.** Agora o banco de registradores recebe:

```
rs1 = x0
rs2 = x28
rd  = x1
```

Na instrução addi, o campo rs2 não é utilizado. Os bits que aparecem como rs2 fazem parte do imediato.

- ALU Control escolhe soma. Como:

```
alu_op = 10
funct3 = 000
opcode = 0010011
// alu_operation = ADD
// ALU_ADD = 4'd0
// ALU vai somar
```

- no top level

```
assign alu_operand_b = alu_src ? selected_imm : reg_read_da
ta2;
// depois em alu.sv ALU_ADD: result = a + b;
// e assim por fim alu_result = 28
```

Write-back escolhe o resultado da ALU

Agora o processador precisa decidir o que será escrito no registrador rd.

No top-level:

```
case (result_src)
    RESULT_ALU: write_back_data = alu_result;
    RESULT_MEM: write_back_data = mem_read_data;
    RESULT_PC4: write_back_data = pc_plus4;
endcase
```

como result_src = 00. então:

```
write_back_data = alu_result
write_back_data = 28
```

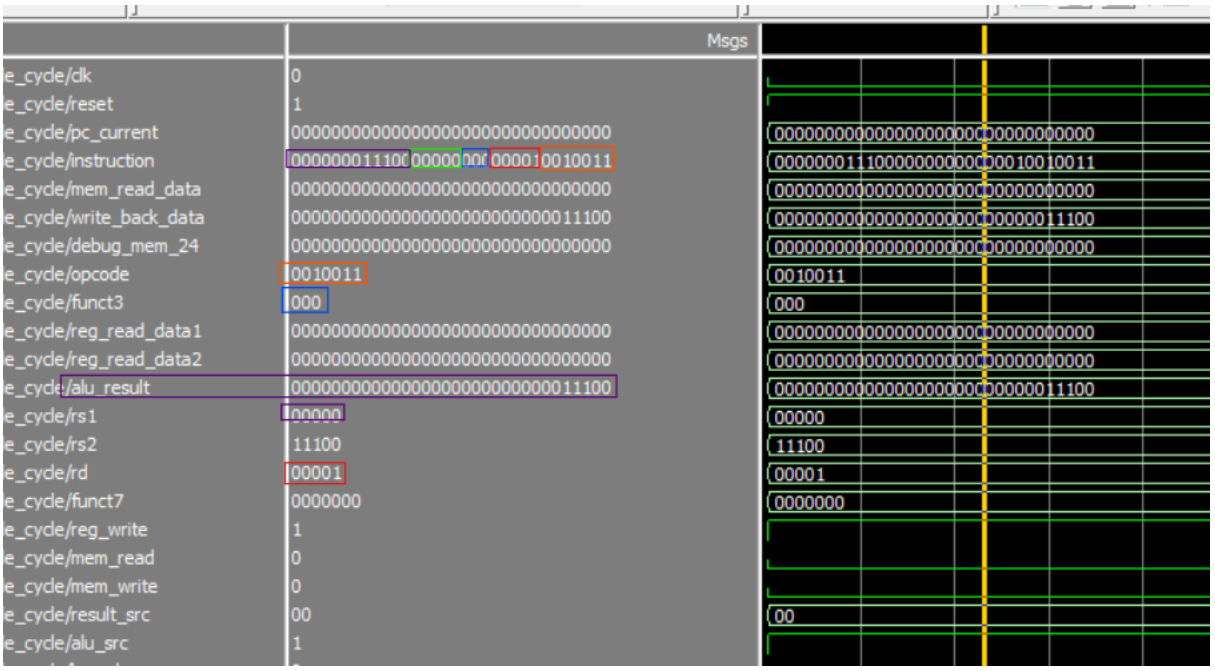
Por fim **Register File escreve no x1.**

Na borda de subida do clock, o banco de registradores faz:

```
if (reg_write && (rd != 5'd0)) begin
    regs[rd] <= write_data;
end
// como reg_write = 1
//          rd = x1
```

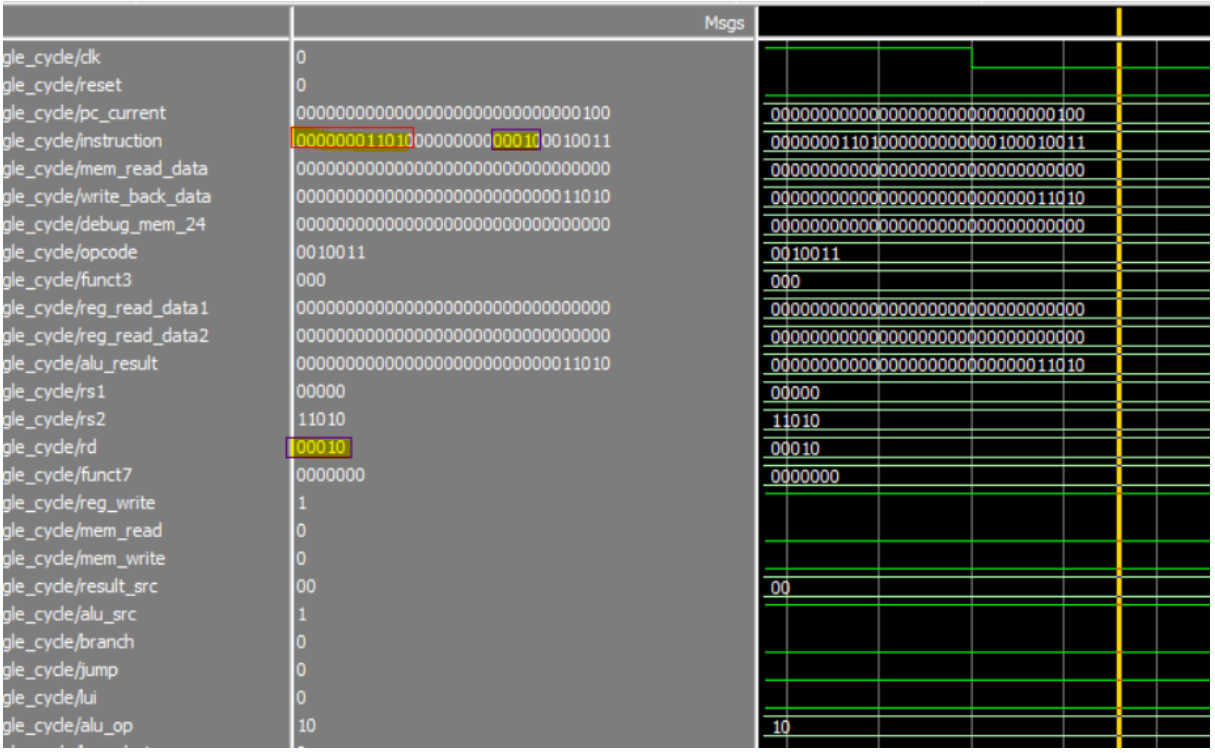
```
//          write_data = 28
//      regs[1] <= 28
// ou seja, x1 = 28 nosso valor de A.
```

PC depois prepara para próxima instrução. Como a instrução não é branch nem jump ele soma PC + 4.



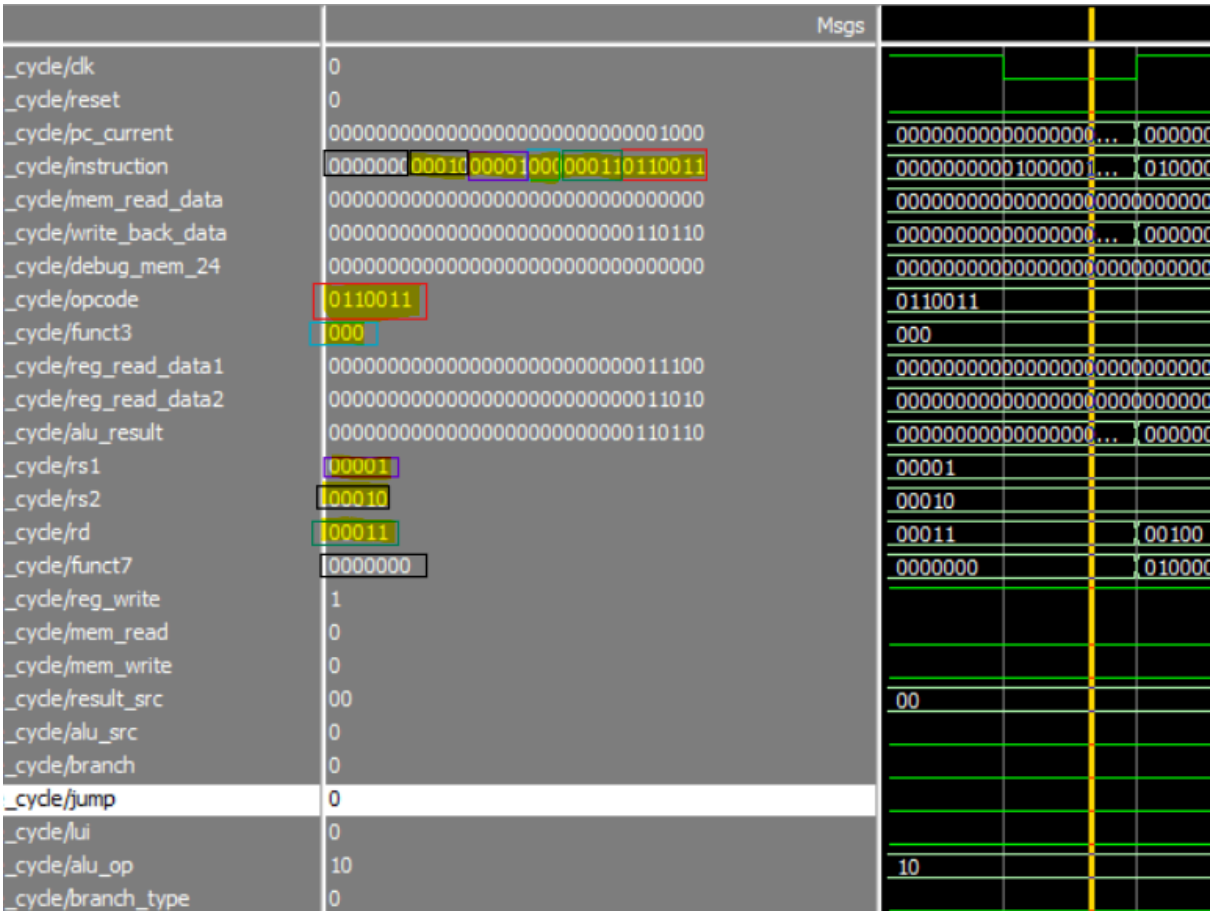
Segunda instrução

- Exatamente igual a primeira a unica diferença é que rd é 00010 (x2).
- E meu imm = 000000011010 = 26 = B. Ou seja estou fazendo o registro no registrador x2 do valor de B.
- **addi x2, x0, B**
- PC incrementou PC + 4. Para pegar essa próxima instrução.

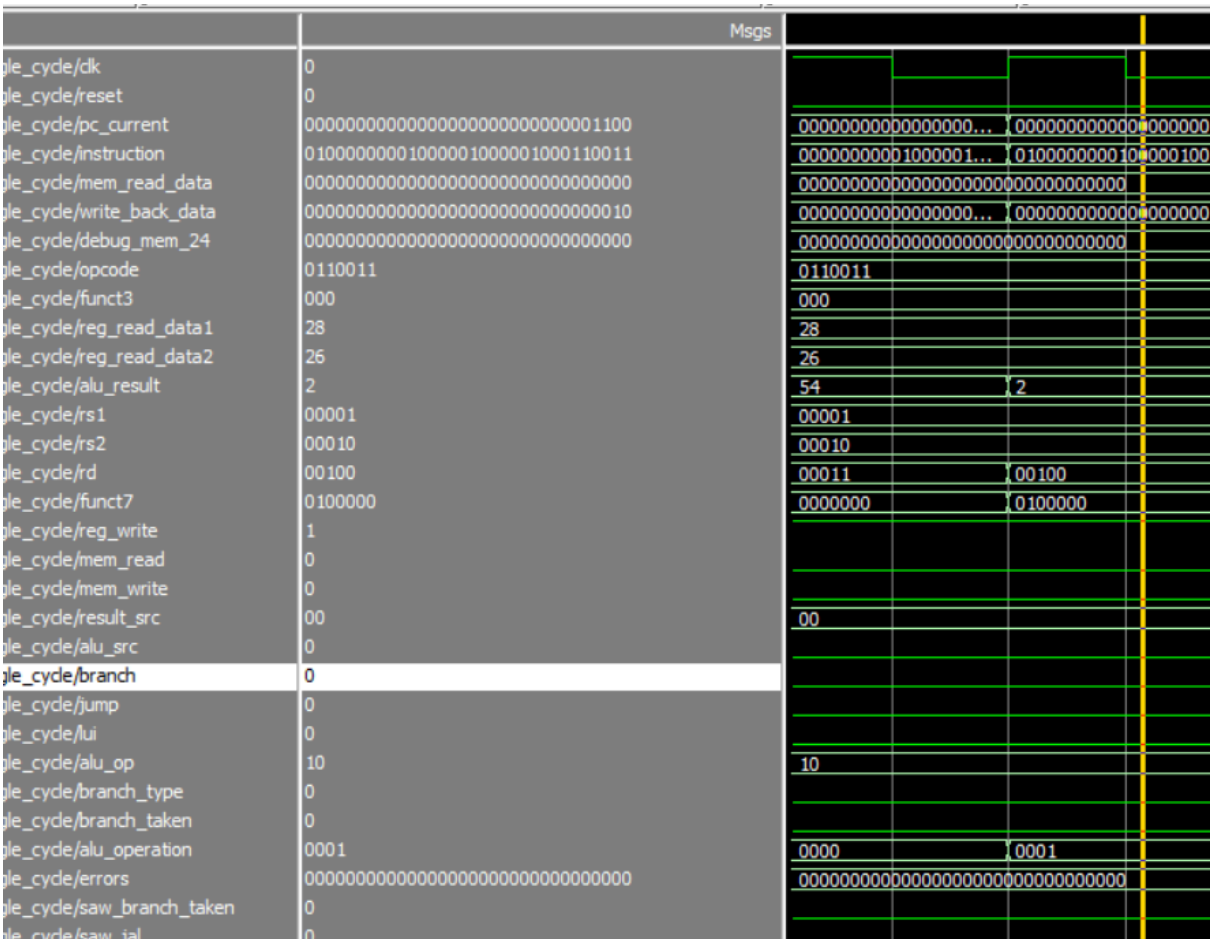


Terceira a oitava instrução (operações com ALU)

- Na terceira instrução estamos fazendo basicamente
- add x3, x1, x2.
- opcode = 0110011 - com func3 = 000 e imm = 0000000 temos a operação ADD.
- rs1 = 00001 (x1) e rs2 = 00010 (x2).
- rd = 00011 (x3).
- Veja o reg_read_data1 e reg_read_data2, são basicamente x1 e x2 sendo lidos dos registradores x2 e x2. Em decimal são 28 e 26, A e B.
- alu_result = ...0110110 = 54.
- 26 + 28 = 54.



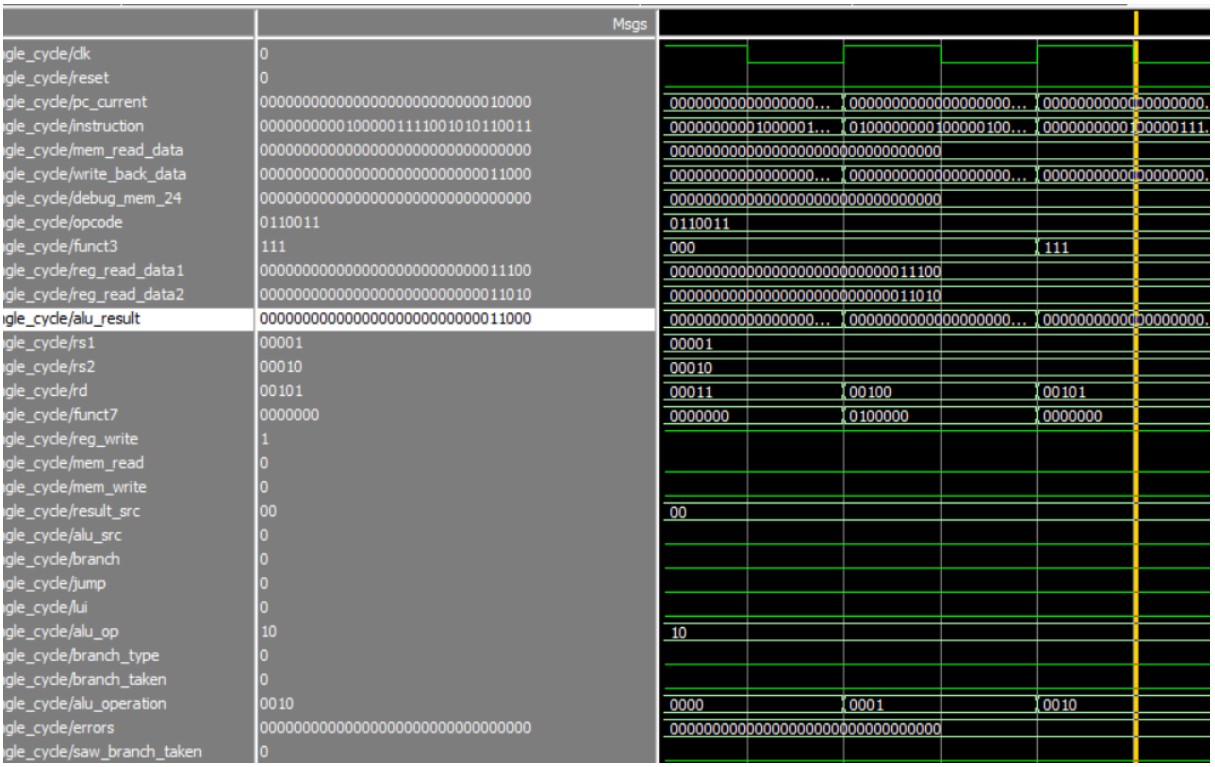
- depois temos a subtração.
- func7 0100000, func3 000, opcode padrão de operação aritimética e lógica.



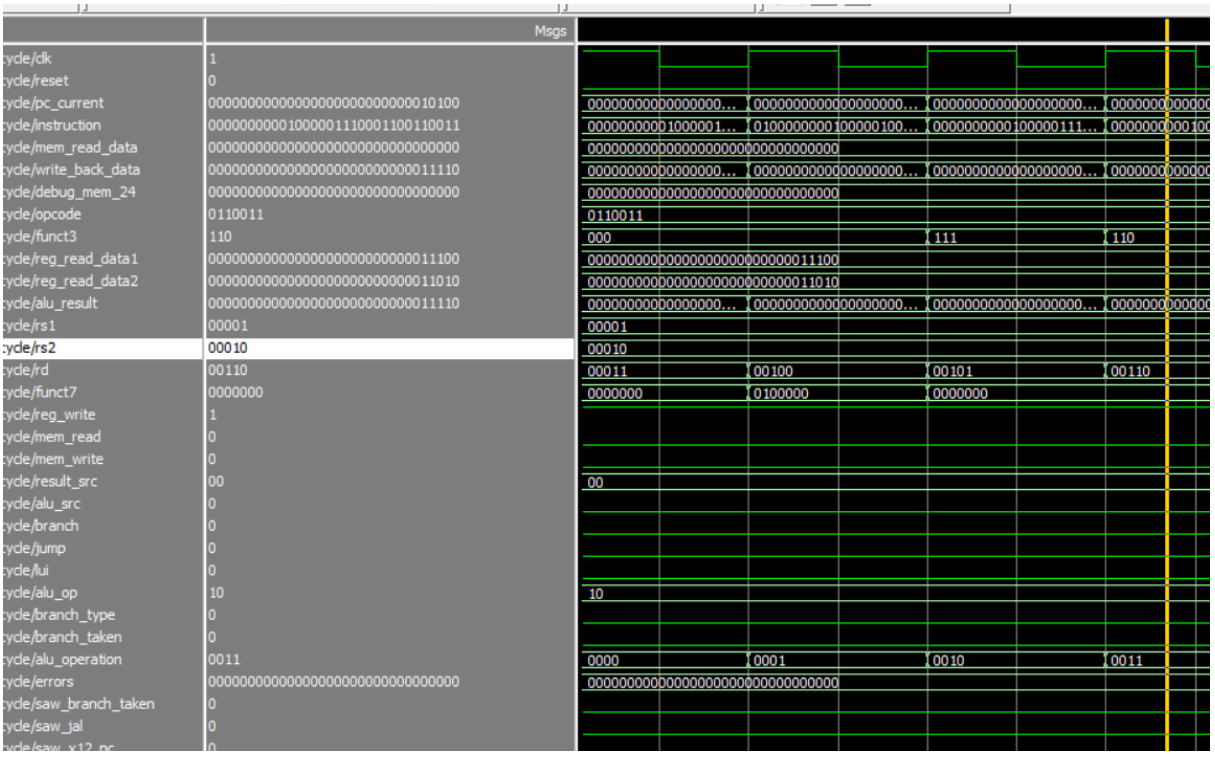
	Msgs
gle_cycle/clk	0
gle_cycle/reset	0
gle_cycle/pc_current	00000000000000000000000000001100
gle_cycle/instruction	01000000001000001000001000110011
gle_cycle/mem_read_data	00000000000000000000000000000000
gle_cycle/write_back_data	00000000000000000000000000000010
gle_cycle/debug_mem_24	00000000000000000000000000000000
gle_cycle/opcode	0110011
gle_cycle/funct3	000
gle_cycle/reg_read_data1	28
gle_cycle/reg_read_data2	26
gle_cycle/alu_result	2
gle_cycle/rs1	00001
gle_cycle/rs2	00010
gle_cycle/rd	00100
gle_cycle/funct7	0100000
gle_cycle/reg_write	1
gle_cycle/mem_read	0
gle_cycle/mem_write	0
gle_cycle/result_src	00
gle_cycle/alu_src	0
gle_cycle/branch	0
gle_cycle/jump	0
gle_cycle/lui	0
gle_cycle/alu_op	10
gle_cycle/branch_type	0
gle_cycle/branch_taken	0
gle_cycle/alu_operation	0001
gle_cycle/errors	00000000000000000000000000000000
gle_cycle/saw_branch_taken	0
gle_cycle/saw_jal	0

- Na quinta instrução temos a operação AND
- func3 = 111 opcode = 0110011.
- rs1 e rs2 continua lendo x1 e x2.

data1 = ...11100
data2 = ...11010
————————-
alu_result = ...11000

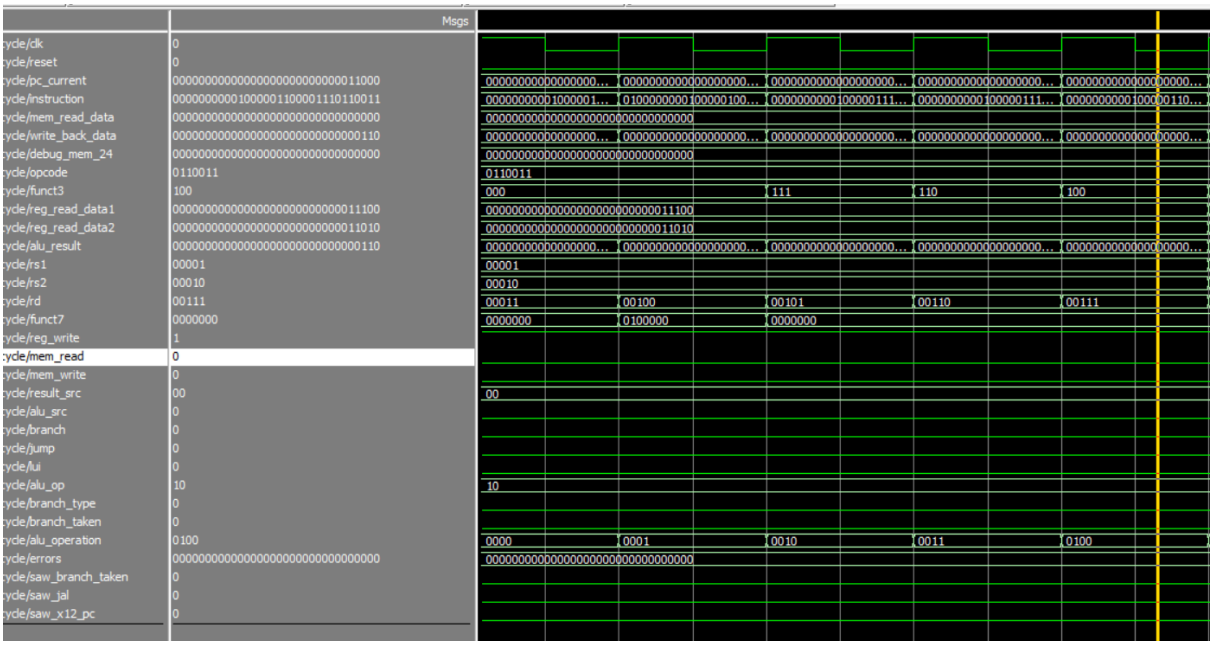


- Próxima instrução operação OR
- opcode = 0110011 | func3 = 110 | fun7 0000000



- Depois temos a operação XOR.

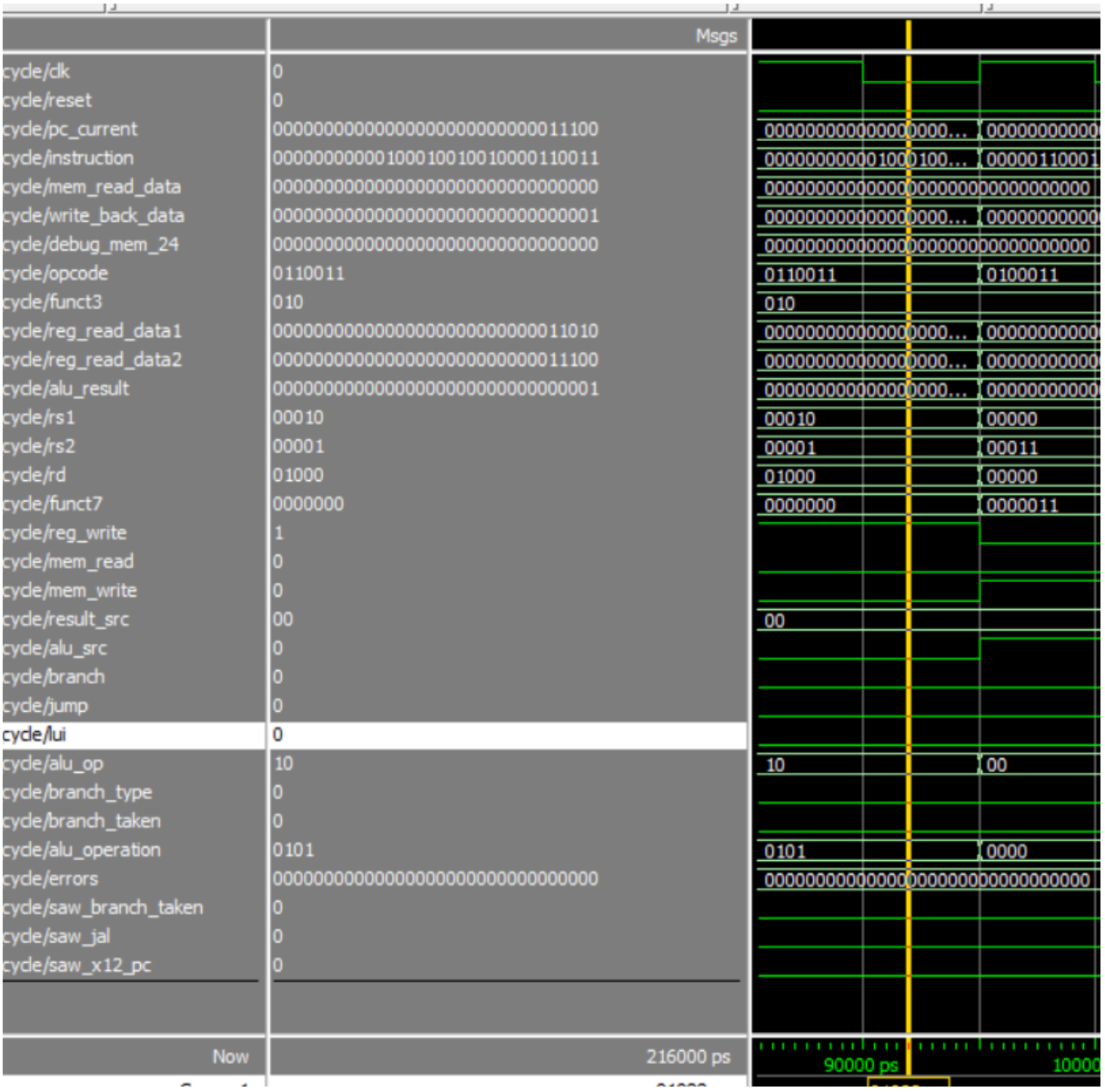
- opcode = 0110011 | func3 = 100 | func7 = 0000000.
- PC continua pegando a próxima instrução (+4).



- Agora temos a instrução slt
- funct3 = 010 opcode = 0110011
- 00112433 = slt x8, x2, x1
- Ela significa:

se x2 < x1, então x8 = 1
senão, x8 = 0

- vejamos. X1 = 28, X2 = 26. Portanto x2 < x1 ? Sim é verdade. então rd(8) = 01000 irá receber 1. Alu_result = 1 exatamente, e write_back_data veja rd x8 = 1.



Nona instrução

Agora vamos para nona instrução (store)

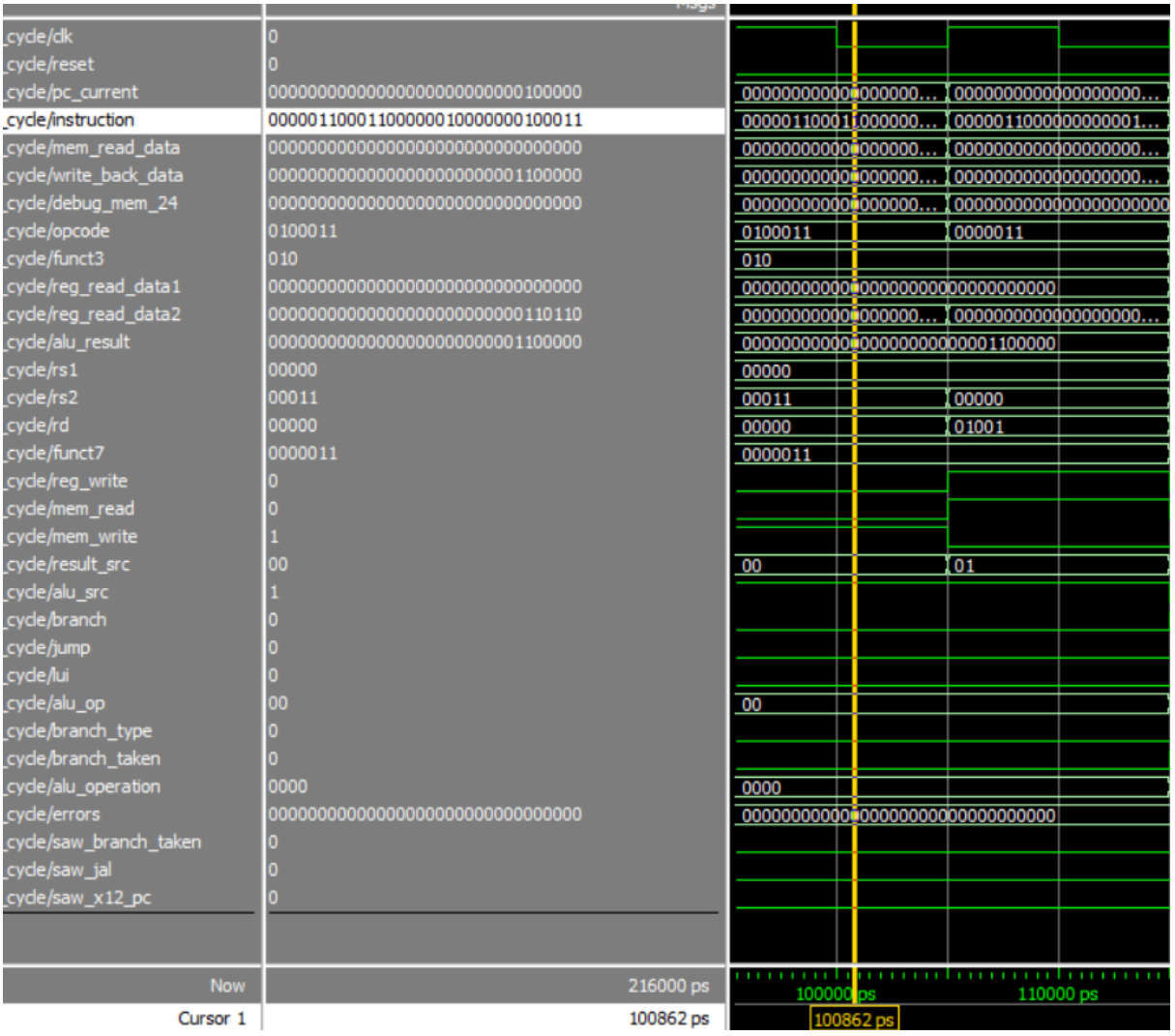
```
sw   x3, MEM_ADDR(x0)   # armazena x3 na memória
lw   x9, MEM_ADDR(x0)   # carrega o valor salvo em x9
```

- Nona instrução 06302023 (store) significa:
  - sw x3, MEM_ADDR(x0) # armazenar x3 na memória.
  - opcode = 0100011 (store) | funct3 = 010 (sw).

- rs1 = base do endereço x0
- rs2 = dado a salvar = x3 00011
- imm = 000001100000 = 96.

- Como nosso dado de x3 foi a operação de ADD entre x1 e x2 e deu 54 (x3).
- memória[0 + 96] = 54.
- memória[96] = 54. Como mem_write = 1 a memória foi escrita o valor de x3 nesse novo endereço.

Como a memória é indexada por palavra nesse projeto memória[96 >> 2] = memória[24] = 54.



Décima instrução

- 06002483. significa:

- lw x9, MEM_ADDR(x0).
- opcode 0000011, funct3 010 (lw).
- rs1 00000 ×0 base do endereço.
- rd 01001 ×9 nosso destino.
- imm = 000001100000 = 96.
- mem_read é 1 e write é 0.

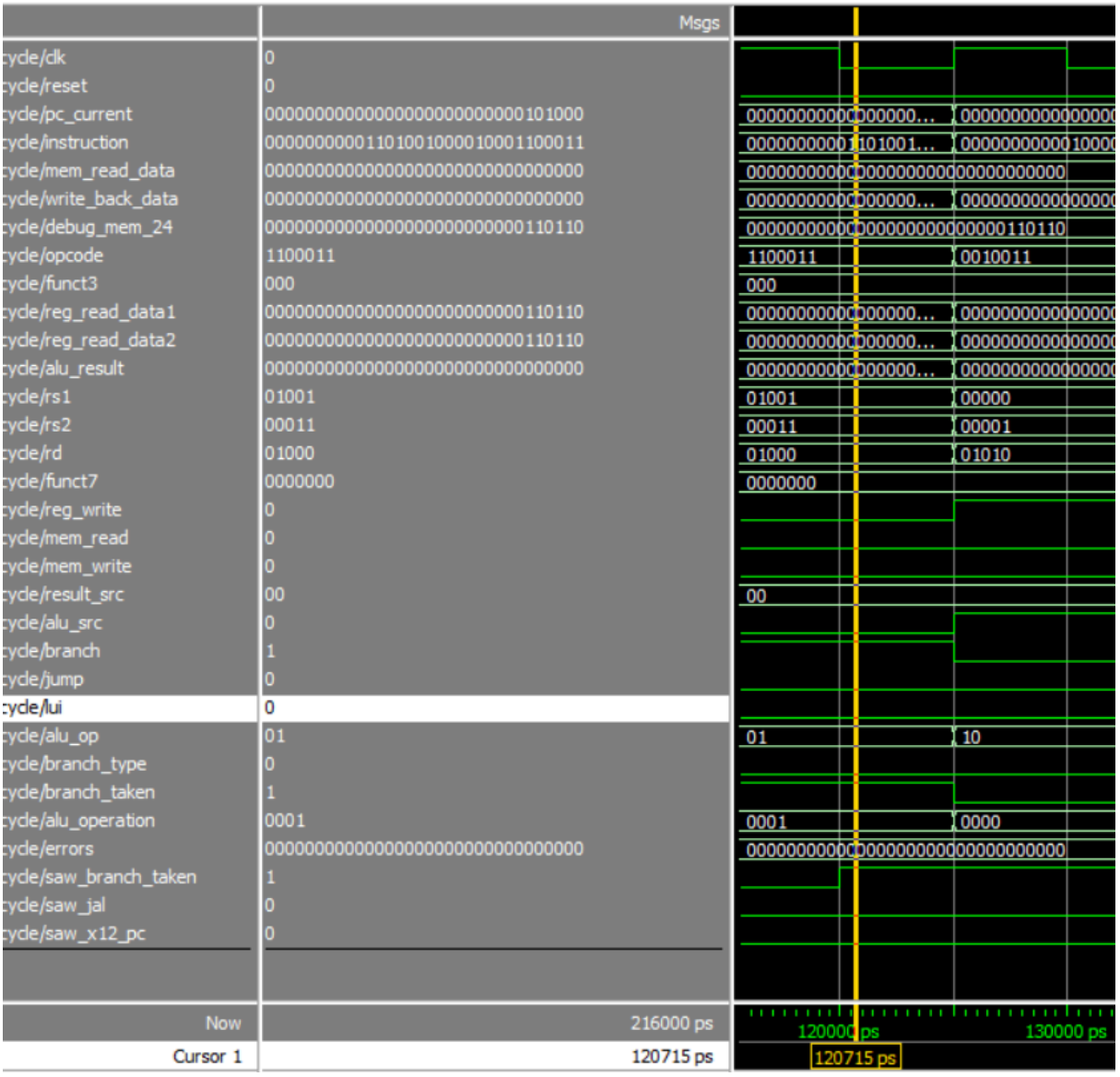
- Basicamente x9 = memória[x0 + 96]
- x9 = memória[96]. Portanto como foi registrado anteriormente x3 em memoria[24].
- x9 = memória[24] = 54. x9 = 54.

Ou seja, conseguimos gravar e ler na memória. Perfeito.

Décima primeira instrução

```
beq  x9, x3, label_ok   # testa branch
addi x10, x0, 0         # indica erro caso o branch não ocorra

label_ok:
  addi x10, x0, 1       # indica sucesso no teste de memória/branch
```

- 00348463 significa.
- opcode = 1100011 (branch) func3 = 000 (beq). Teste de igualdade.
- se x9 == x3, então pule para label_ok
  Como:
  x9 = 54 e x3 = 54 devido as intruções anteriores.
- rs1 = 01001 ×9.
- rs2 = 00011 ×3.
- branch_taken = 1. Porque são iguais.



Décima segunda/terceira instrução

Instrução pulou da 12° 00000513 para 13° 00100513. Porque a branch ocorreu e deu tudo certo, então não passou para essa intrução. Foi direto para 13°.

Veja PC incrementou de 8 ao invés de 4. Justamente.

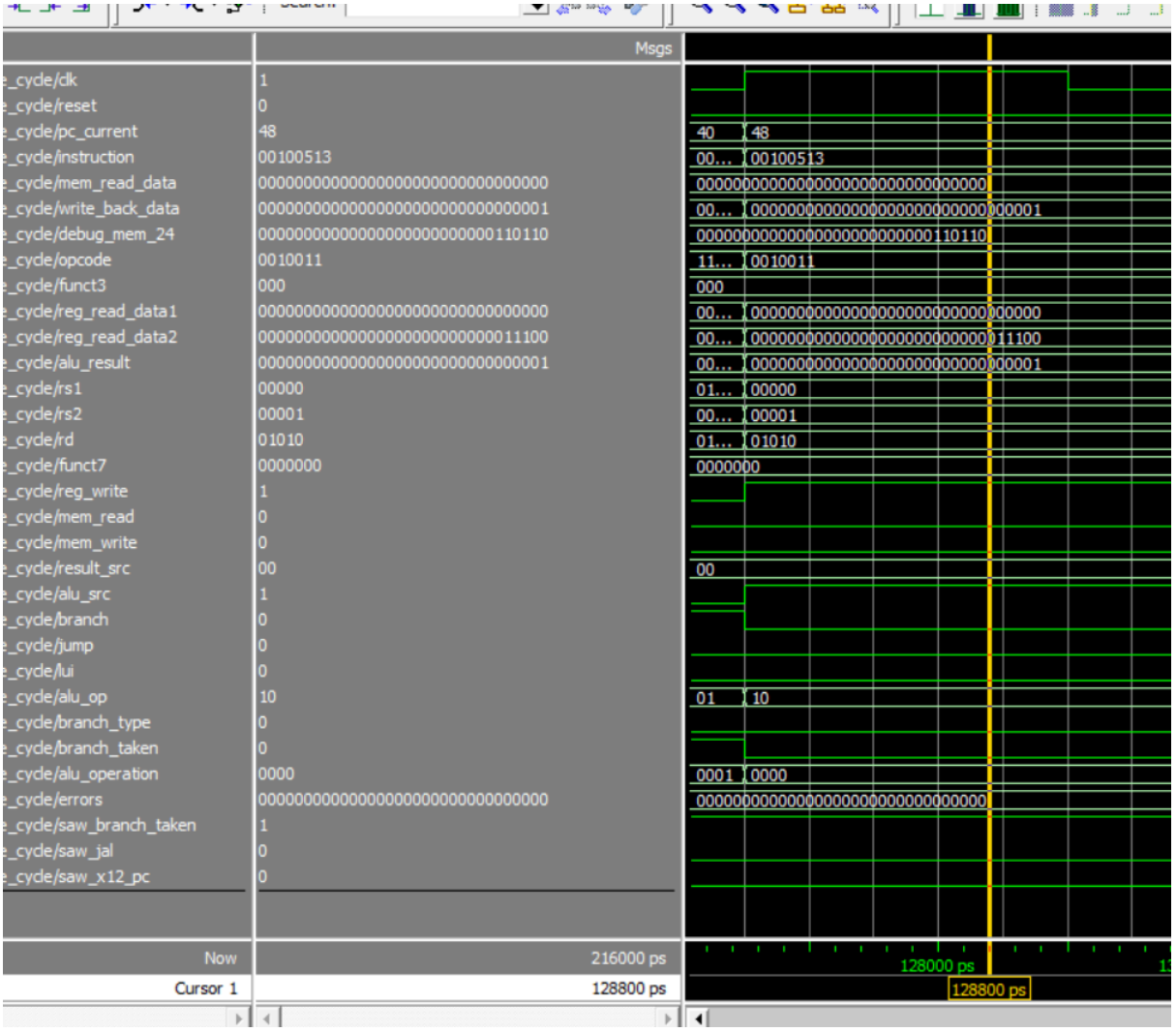
Como label_ok ficou no registrador x9. Aqui essa instrução 00100513. Significa:

- opcode = 0010011 (tipo i) func3 000 (addi)
- addi x10, x0, 1. Porque label_ok é true. A branch ocorrou corretamente. Essa operação significa que vou escrever no registrador x10 o valor de 1.

x10 = 0 + 1
x10 = 1

- alu_result = 1 ou seja adicionou corretamente
- rd = 01010 (x10) destino.
- imm = 000000000001 (1)

```
beq  x9, x3, label_ok   # testa branch
addi x10, x0, 0         # indica erro caso o branch não ocorra

label_ok:
  addi x10, x0, 1       # indica sucesso no teste de memória/branch
```



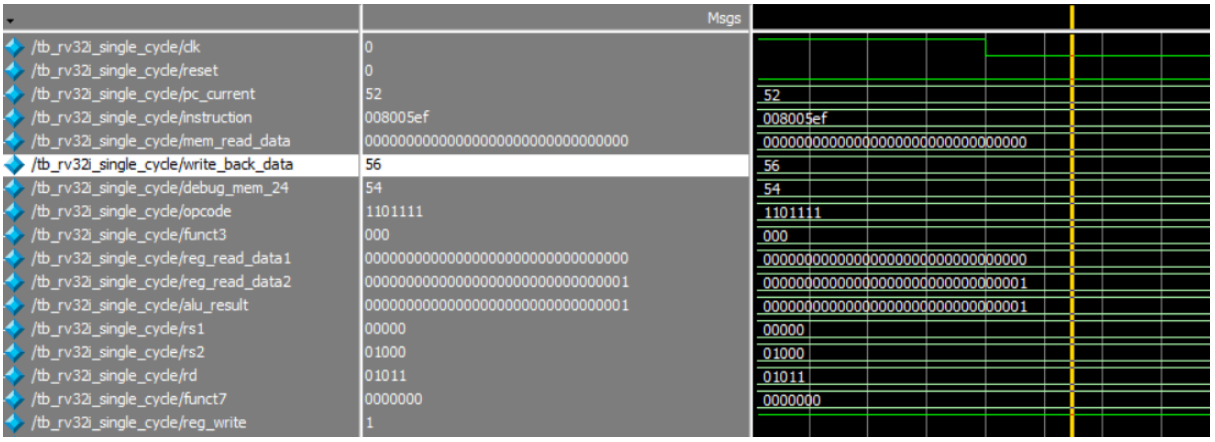
Décima quarta instrução

```
jal  x11, fim          # testa jump e salva PC + 4 em x11
addi x12, x0, 99       # esta instrução deve ser pulada se o jal funcionar

fim:
addi x13, x0, C        # valor final individualizado
```

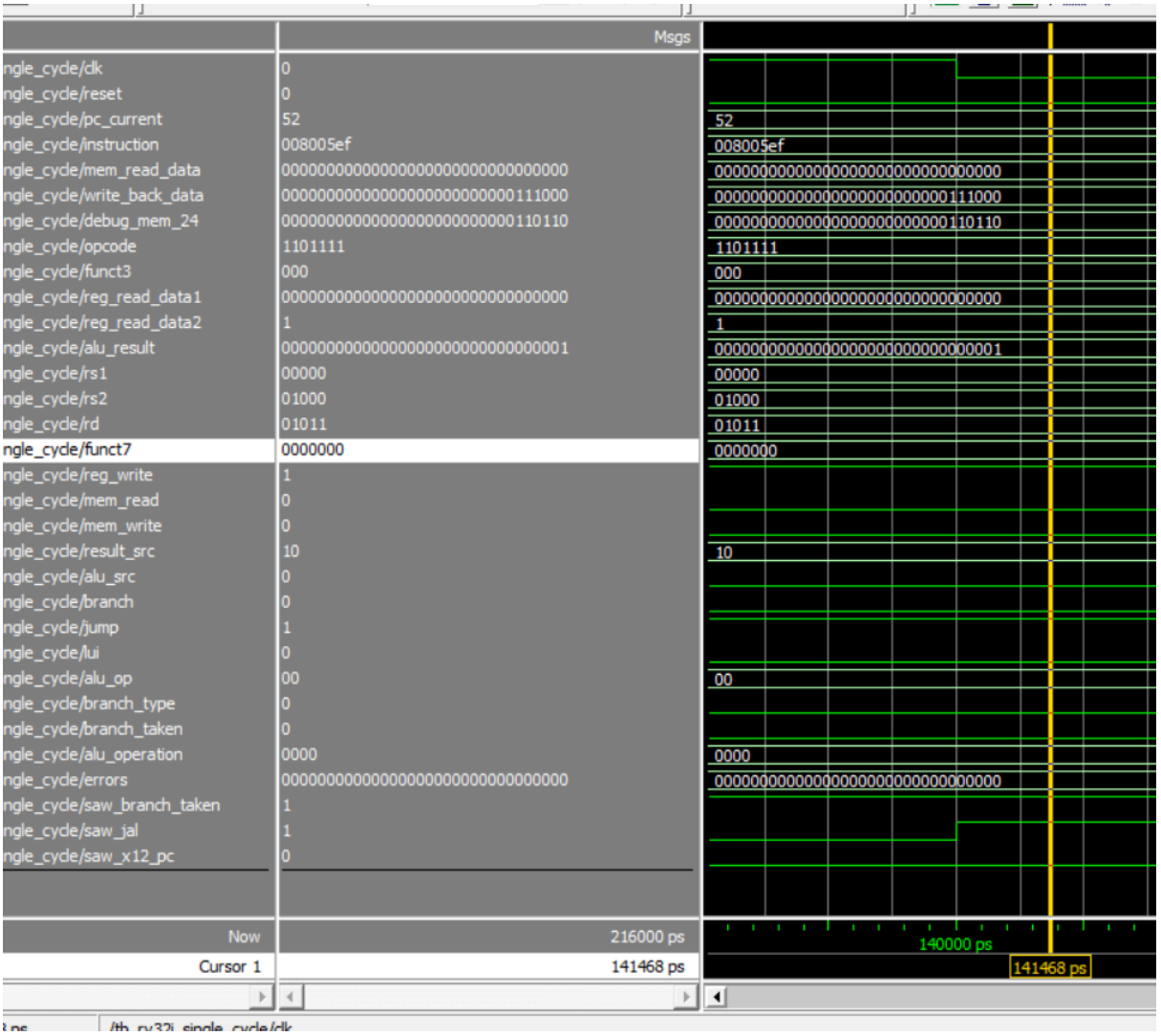
008005ef significa:

jal x11, fim. Salta para outro endereço e salva PC + 4 em um registrador.

- opcode =1101111 (tipo j) jal
- rd = 01011 (destino) x11. Ou seja, vou gravar em X11 o valor pc + 4 = 52.



	Msgs
/tb_rv32i_single_cycle/clk	0
/tb_rv32i_single_cycle/reset	0
/tb_rv32i_single_cycle/pc_current	52
/tb_rv32i_single_cycle/instruction	008005ef
/tb_rv32i_single_cycle/mem_read_data	00000000000000000000000000000000
/tb_rv32i_single_cycle/write_back_data	56
/tb_rv32i_single_cycle/debug_mem_24	54
/tb_rv32i_single_cycle/opcode	1101111
/tb_rv32i_single_cycle/funct3	000
/tb_rv32i_single_cycle/reg_read_data1	00000000000000000000000000000000
/tb_rv32i_single_cycle/reg_read_data2	00000000000000000000000000000001
/tb_rv32i_single_cycle/alu_result	00000000000000000000000000000001
/tb_rv32i_single_cycle/rs1	00000
/tb_rv32i_single_cycle/rs2	01000
/tb_rv32i_single_cycle/rd	01011
/tb_rv32i_single_cycle/funct7	0000000
/tb_rv32i_single_cycle/reg_write	1



Décima quinta/sexta instrução

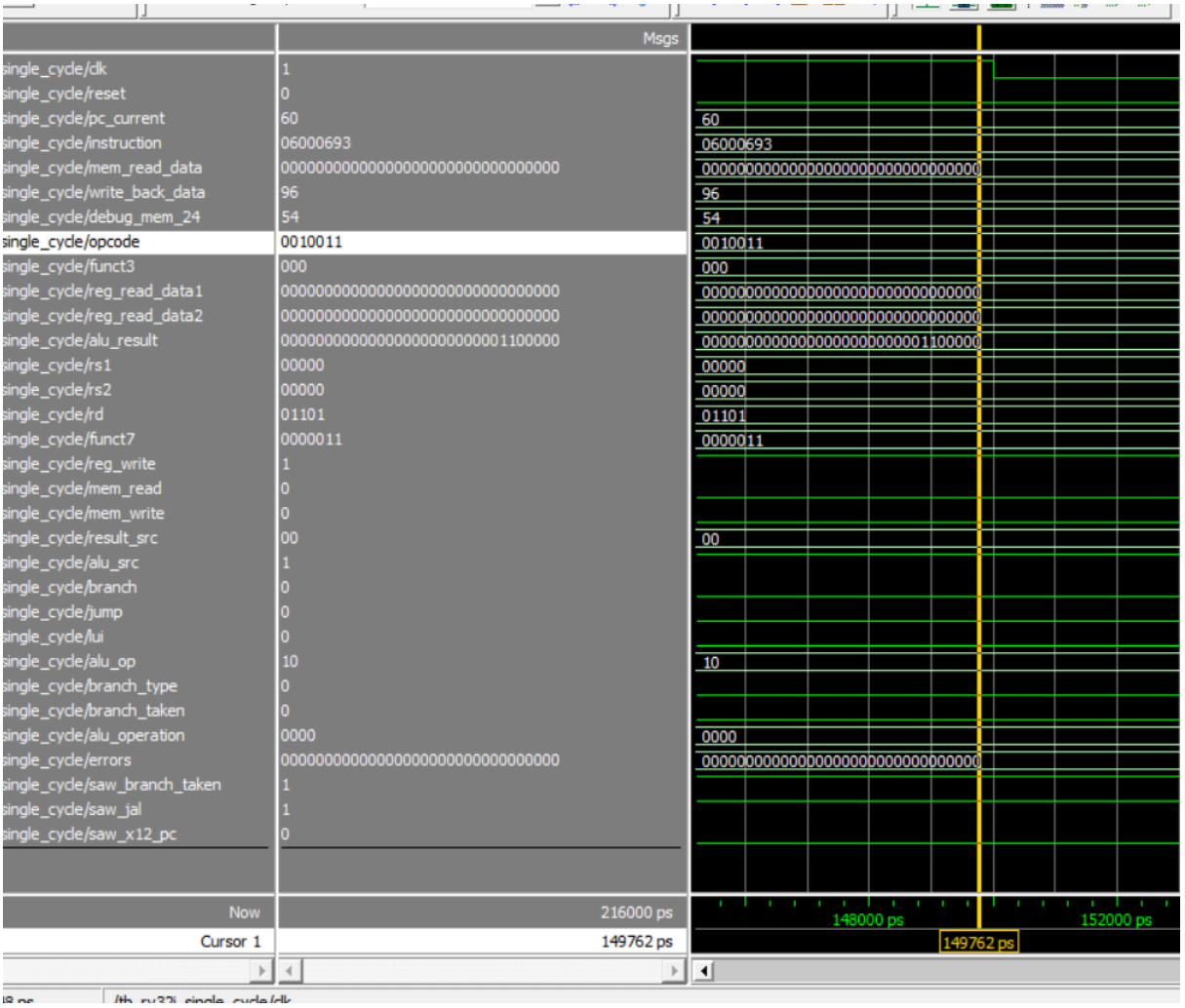
```
jal  x11, fim          # testa jump e salva PC + 4 em x11
addi x12, x0, 99       # esta instrução deve ser pulada se o jal funcionar

fim:
addi x13, x0, C        # valor final individualizado
```

Como a instrução jal funcionou esta instrução addi x12, x0, 99 é pulada. E vamos para o fim. a última instrução 06000693.

Significa addi x13, x0, C.

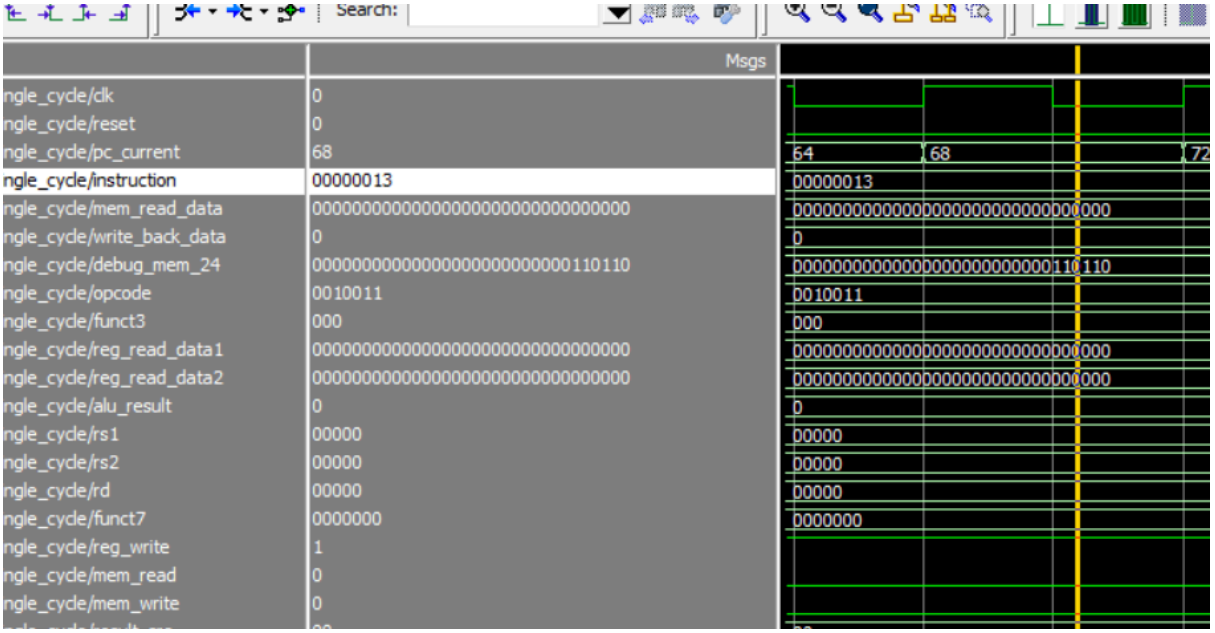
- opcode = 0010011 (tipo i) funct3 = 000 (addi)

- rs1 = 0000 (x0)
- rd = 01101 (x13)
- imm = 000001100000 (96) veja alu_result = 96.
- veja write_back_data = 96. Ou seja 4 ∗ N = 96 Pois N = 24. Escrevi 96 o valor de C no registrador X13.
- E fim. acabou.



Para fechar tudo a última instrução é NOP

- x0 = x0 + 0 por padrão x0 é zero então nao muda nada. e PC continua contando porque ainda tem ciclo de clock e reset = 0.



Referência principal:

https://docs.riscv.org/reference/isa/unpriv/rv-32-64g.html

Estrutura do Projeto e código

```
rv32i_single_cycle/
├── src/
│   ├── pc.v
│   ├── instruction_memory.v
│   ├── decoder.v
│   ├── immediate_generator.v
│   ├── control_unit.v
│   ├── alu_control.v
│   ├── alu.v
│   ├── register_file.v
│   ├── data_memory.v
│   └── rv32i_single_cycle_top.v
│
├── tb/
│   └── tb_rv32i_single_cycle.v
│
├── mem/
│   └── program.mem
```

```
│
└── docs/
    └── relatorio.pdf
```

Link do repositório

github: https://github.com/HyAgOsK/Relat-rio-T-nico-RV32I-Single-Cycle