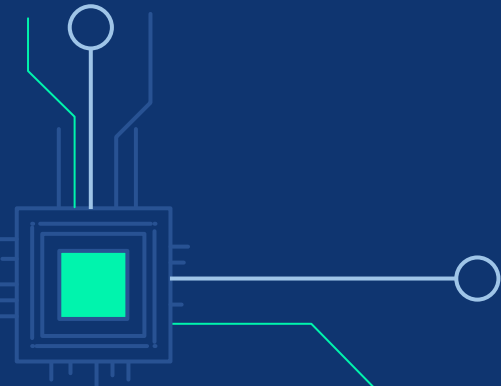




PROCESSADORES SUPERESCALARES

Arthur Souto Lima
Pablo Correa Costa
Hilário Corrêa da Silva Neto
Victor Vieira Brito Amaral Pessoa





ROTEIRO

01 INTRODUÇÃO

02 ALTERAÇÕES NO ESTÁGIO EXECUÇÃO

03 PROCESSADOR I4

04 PROCESSADOR I202

05 TESTES

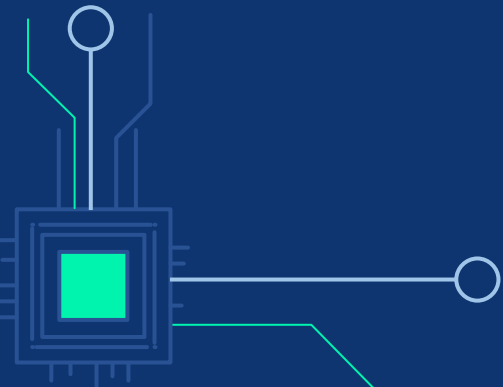
06 CONSIDERAÇÕES FINAIS





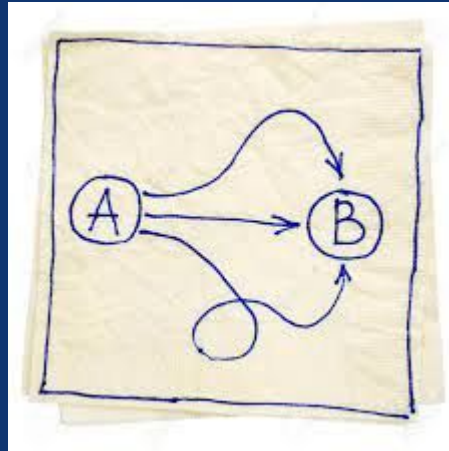
01

INTRODUÇÃO



O DESAFIO

- CPI menor do que 1.
- Novo paralelismo: **Superescalar!**
- Distinguir o hardware necessário para diferentes tipo de operações ?
- Quais as consequências no fluxo de dados ?



RELEMBRANDO

- Hardware para operações com números inteiros.
- RISC V.
- 32 registradores de 32 bits.
- Pipeline.



INSTRUÇÕES

- Conjunto básico para testar os três caminhos no módulo de execução:
 - Load & Store
 - Add
 - Addi
 - Mult



14

- Versão introdutória de um Superescalar.
- Todas as rotas possuem o mesmo número de estágios, o que mantém a ordem.
- 3 caminhos no estágio de execução: ALU, Memória e Multiplicação.

Name	Frontend	Issue	Writeback	Commit	
14	10	10	10	10	Fixed Length Pipelines Scoreboard



I202

- Versão que inicia as tentativas de romper com a ordem para ganhar eficiência.
- Rotas de tamanhos variáveis.
- Demanda cuidados com conflitos de valores dos registradores.

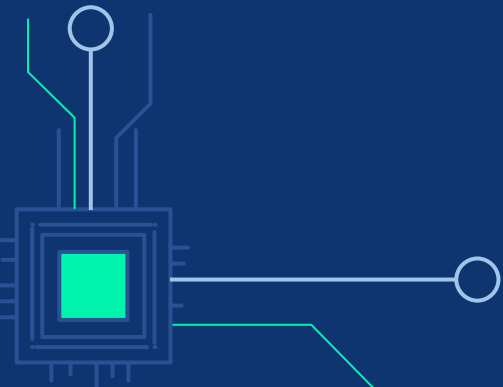
Name	Frontend	Issue	Writeback	Commit	
I202	IO	IO	OOO	OOO	Scoreboard





02

ALTERAÇÕES NO ESTÁGIO EXECUÇÃO



MULTIPLICAÇÃO

- Módulo central que encaminha os dados entre outros 4 sub-módulos (Y0-Y3).
- Input: valores de rs1 e rs2, endereço de rd, clock, reset e sinal do issue
- Output: sinal de controle para WB, endereço de rd e valor calculado



MULTIPLICAÇÃO

- Y0: Checagem de valores e aferição de zero e sinal
 - Comparação lógica entre sinais
 - $Rs1 == 0?$ $Rs2 == 0?$
- Y1: Modularização dos fatores



MULTIPLICAÇÃO

- Y2: Multiplicação direta (operador *)
- Y3: Checagem de overflow, correção de sinal e formatação do resultado
- Conferência dos 32 bits mais altos
- Inversão da modularização - se necessário
- Eliminação dos 32 bits mais altos



MEMÓRIA

- Estágios do MEM:
- São divididos em 2 módulos principais e 2 módulos (I4) que não possuem funcionalidade a não ser passagem de sinal.
- MEM0: Realiza o cálculo do endereço de memória.
- MEM1: Recebe o endereço de memória do MEM0 e realiza a escrita ou leitura do dado quando for necessário.
- Módulo MEM: Integra MEM0 e MEM1 e passa o sinal pelo MEM2 e MEM3, além de trazer o encaminhamento de retorno.



SCOREBOARD

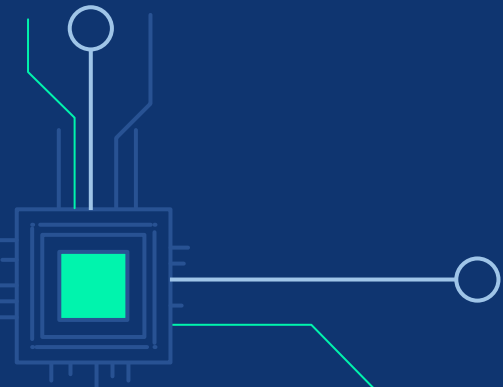
- Módulo próprio
 - Recebe nos estágios de Decode e de Issue as informações dos registrados que serão operados e alguns sinais de controle anti-hazards.
 - Retorna informações sobre tais registradores.
 - Pendente ? Em qual unidade funcional ? Onde na unidade ?
- Tabela 32x8
 - Sinais de pendente e funcional (2bits).
 - 5 marcadores de progressão da informação pelo pipeline.
- A cada virada de clock, checa-se a situação atual e shifta para a direita os bits.
- Caso toda a linha referente aquele registrador for zero, ele não estará pendente mais.





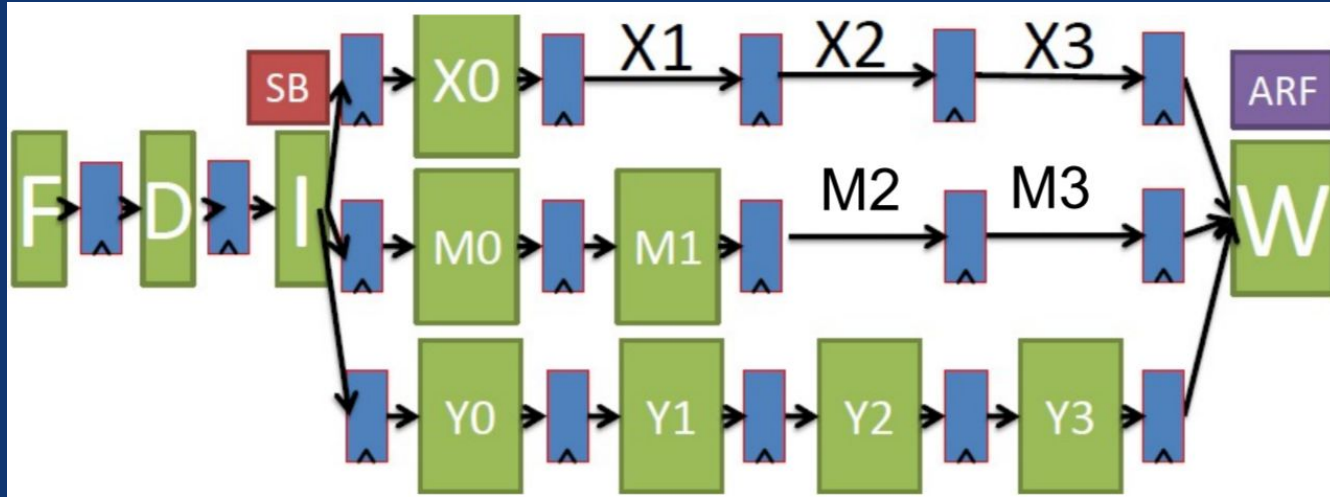
03

PROCESSADOR I4



PROCESSADOR I4

- Esquemático I4:



PROCESSADOR I4

- Processador totalmente In-Order.
- Desmembramento do estágio de Decodificação: Novo estágio Issue.
- No Issue: Scoreboard, Hazard Detector e leitura dos registradores do Architecture Register File.
- No WriteBack: Escrita dos registradores no ARF.
- Divisão do estágio de execução:
 - Multiplicação em 4 estágios.
 - Memória 2 + 2 estágios.
 - Operações da ALU 1 + 3 estágios.
- Problema: Encaminhamento grande.
- Solução: I2O2.



FETCH

- Fetch passa a instrução pro módulo do Decode
- Realiza a Operação de $PC + 4$.

```
always @(posedge clock or negedge reset) begin
    if (~reset) begin
        if_id_instruc <= 32'h0000_0000;
        pc <= 32'h0000_0000;
        if_id_nextpc <= 32'h0000_0000;
    end else begin
        if (id_stall) begin
            // issue stall should keep current instruction
            if_id_nextpc <= pc;
        end else begin
            if_id_instruc <= instr_data;
            if (id_if_selpcsource) begin
                case (id_if_selpctype)
                    2'b00: pc <= id_if_pcimd2ext;
                    2'b01: pc <= id_if_rega;
                    2'b10: pc <= id_if_pcindex;
                    2'b11: pc <= 32'h0000_0040;
                    default: pc <= 32'hXXXX_XXXX;
                endcase
            case (id_if_selpctype)
                2'b00: if_id_nextpc <= id_if_pcimd2ext;
                2'b01: if_id_nextpc <= id_if_rega;
                2'b10: if_id_nextpc <= id_if_pcindex;
                2'b11: if_id_nextpc <= 32'h0000_0040;
                default: if_id_nextpc <= 32'hXXXX_XXXX;
                //default: pc <= 32'hXXXX_XXXX;
            endcase
        end else begin
            if_id_nextpc <= pc + 32'h0000_0004;
            pc <= pc + 32'h0000_0004;
        end
    end
end
end
end
endmodule

`endif
```

DECODE

- Módulo de Controle.
- Instruções de Desvio.

```
Comparator COMPARATOR(.a(reg_id_ass_dataa),.b(reg_id_ass_datab),.op(compop),.compout(compout));
Control CONTROL(.op(if_id_instruc[31:26]),.fn(if_id_instruc[5:0]),
                .selwsouce(selwsouce),.selregdest(selregdest),.writereg(writereg),
                .writeov(writeov),.selimregb(selimregb),.selalushift(selalushift),
                .aluop(aluop),.shiftp(shiftop),.readmem(readmem),.writemem(writemem),
                .selbrjumpz(selbrjumpz),.selpctype(selpctype),.compop(compop),
                .unsig(unsig));
```



ISSUE

- Scoreboard
- Hazard Detector.

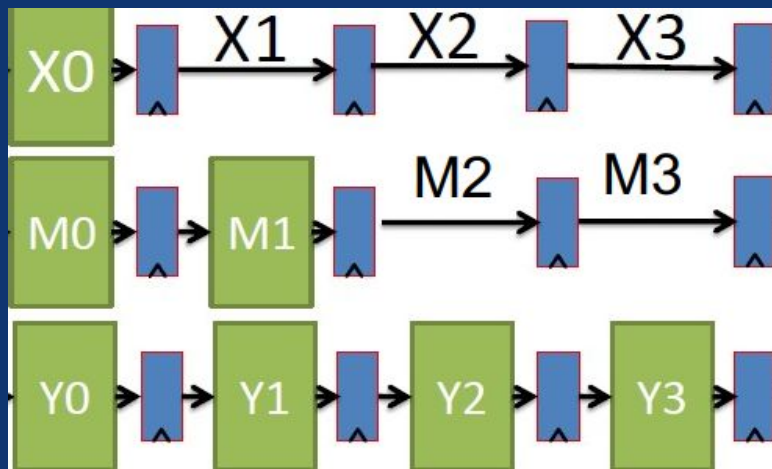
```
HazardDetector HDETECTOR (  
    .iss_ass_pending_a(iss_ass_pending_a),  
    .iss_ass_row_a(iss_ass_row_a),  
    .iss_check_a(1'b1),  
    .iss_ass_pending_b(iss_ass_pending_b),  
    .iss_ass_row_b(iss_ass_row_b),  
    .iss_check_b(id_iss_selregdest),  
    .iss_stalled(iss_stall),  
  
    .id_ass_pending_a(id_ass_pending_a),  
    .id_ass_row_a(id_ass_row_a),  
    .id_check_a(id_hd_check_a),  
    .id_ass_pending_b(id_ass_pending_b),  
    .id_ass_row_b(id_ass_row_b),  
    .id_check_b(id_hd_check_b),  
    .id_stalled(hd_id_stall)  
);
```

```
Scoreboard SB (  
    .clock(clock),  
    .reset(reset),  
  
    .iss_ass_addr_a(id_iss_addra),  
    .iss_ass_pending_a(iss_ass_pending_a),  
    .iss_ass_unit_a(iss_ass_unit_a),  
    .iss_ass_row_a(iss_ass_row_a),  
  
    .iss_ass_addr_b(id_iss_addrb),  
    .iss_ass_pending_b(iss_ass_pending_b),  
    .iss_ass_unit_b(iss_ass_unit_b),  
    .iss_ass_row_b(iss_ass_row_b),  
  
    .id_ass_addr_a(id_hd_ass_addra),  
    .id_ass_pending_a(id_ass_pending_a),  
    .id_ass_unit_a(id_ass_unit_a),  
    .id_ass_row_a(id_ass_row_a),  
  
    .id_ass_addr_b(id_hd_ass_addrb),  
    .id_ass_pending_b(id_ass_pending_b),  
    .id_ass_unit_b(id_ass_unit_b),  
    .id_ass_row_b(id_ass_row_b),  
  
    .writeaddr(writeaddr),  
    .registerunit(registerunit),  
    .enablewrite(enablewrite)  
);
```



EXECUÇÃO

- Execução dividida entre os estágios: Multi, Mem e Alu.
- Multiplicação : 4 estágios - Multiplicação de números inteiros.
- Memória: 2 estágios - Operações Load Store.
- ALU: 1 estágio - Operações Lógico-Aritméticas.



WRITEBACK

- Recebe os sinais.
- Decide o que vai escrever.
- Escreve no banco de registradores
- ARF.

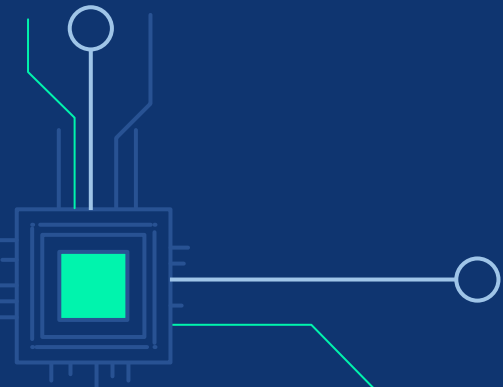
```
37     assign wb_reg_en = mem_wb_oper ? mem_wb_writereg : (  
38         am_wb_oper ? am_wb_writereg : (  
39             mul_wb_oper ? mul_wb_writereg : 1'b0  
40         )  
41     );  
42  
43     assign wb_reg_addr = mem_wb_oper ? mem_wb_regdest : (  
44         am_wb_oper ? am_wb_regdest : (  
45             mul_wb_oper ? mul_wb_regdest : 5'b00000  
46         )  
47     );  
48  
49     assign wb_reg_data = mem_wb_oper ? mem_wb_wbvalue : (  
50         am_wb_oper ? am_wb_wbvalue : (  
51             mul_wb_oper ? mul_wb_wbvalue : 32'h0000_0000  
52         )  
53     );  
54
```





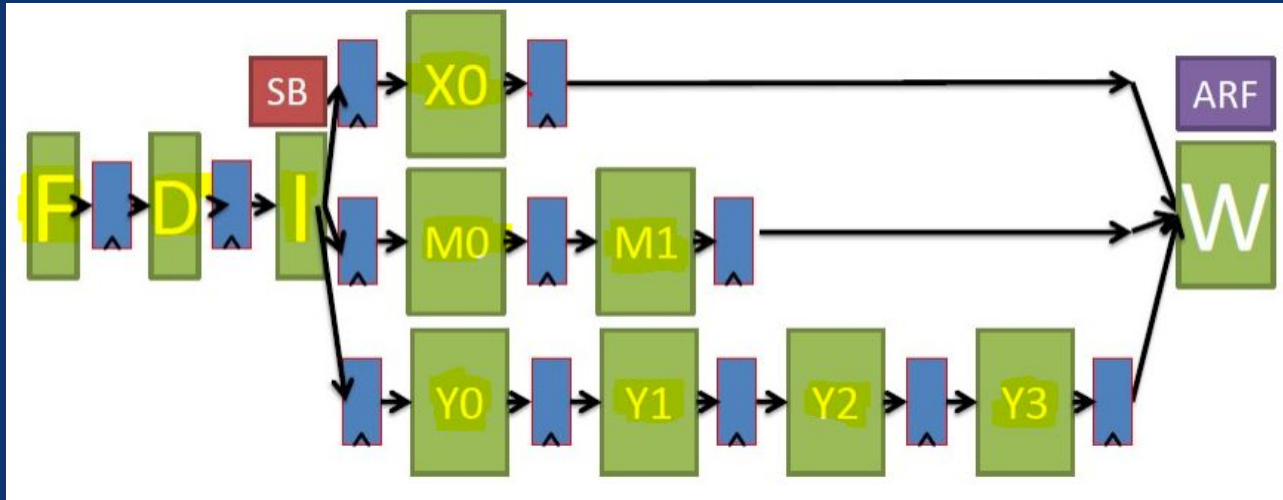
04

PROCESSADOR I202



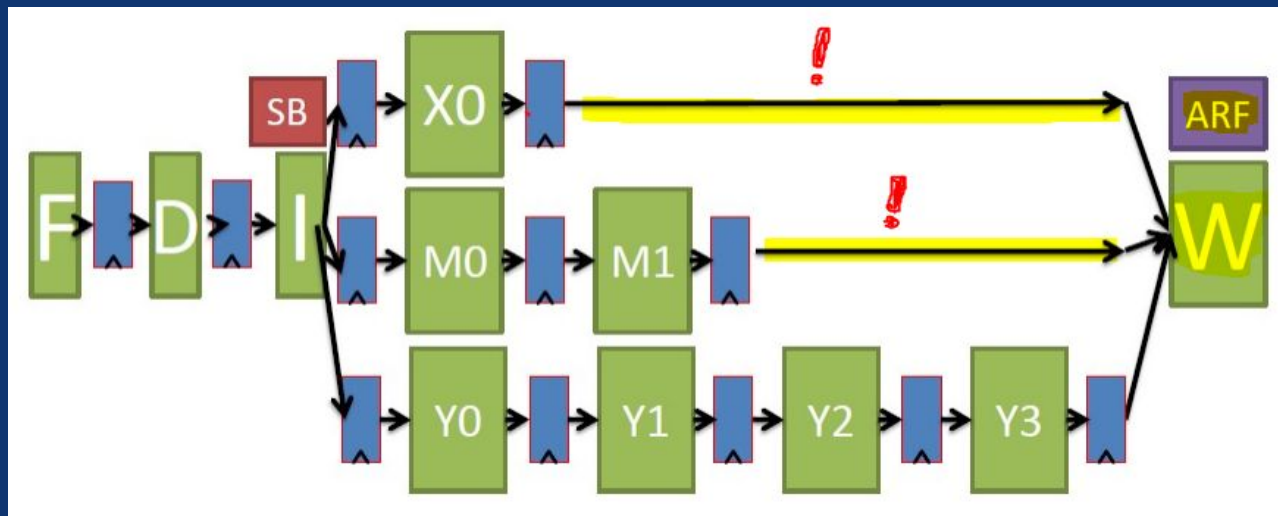
O QUE SE MANTÉM E O QUE MUDA ?

- Fetch, Decode e Issue são basicamente os mesmos do I4.
- Cada etapa do Execute têm uma demorana própria.
- WriteBack agora não recebe os dados em ordem.



NOVIDADES

- Número mínimo de estágios de execução.
- Posições adiantadas no Scoreboard.



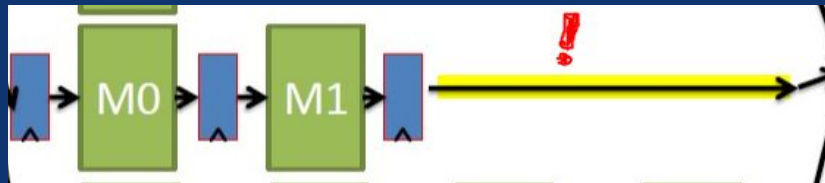
MÓDULOS DA ALU

- Módulo que gerencia os sinais vindos dos registradores, clock, controle, saídas e etc.
- Módulo interno para as operações da ALU.
- Diferente do i4, após obtido o resultado da ALU, o mesmo já é preparado para seguir adiante, sem ter que passar por vários condicionais que simulam o pipeline.



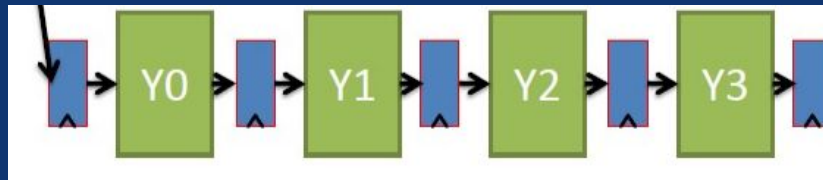
MÓDULOS DE MEMÓRIA

- Módulo central que chama outros módulos internos, um para cada estágio (M0 & M1).
- o M0 calcula o endereço enquanto M1 faz o acesso.
- Diferente do modelo anterior, o valor de resultante de M1 é encaminhado pelo módulo de Memória diretamente e por tanto sem um condicional de clock para atrasar e nivelar o número de estágios.



MÓDULOS DE MULTIPLICAÇÃO

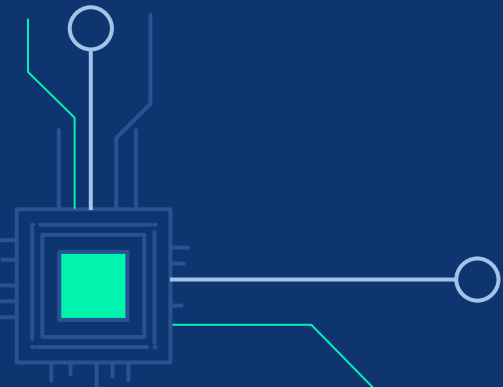
- Como a multiplicação era quem dominava o número de estágios, não há alterações no seu desenvolvimento.





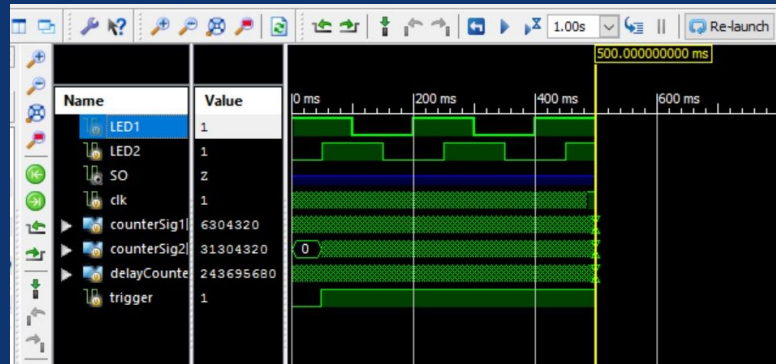
05

TESTES

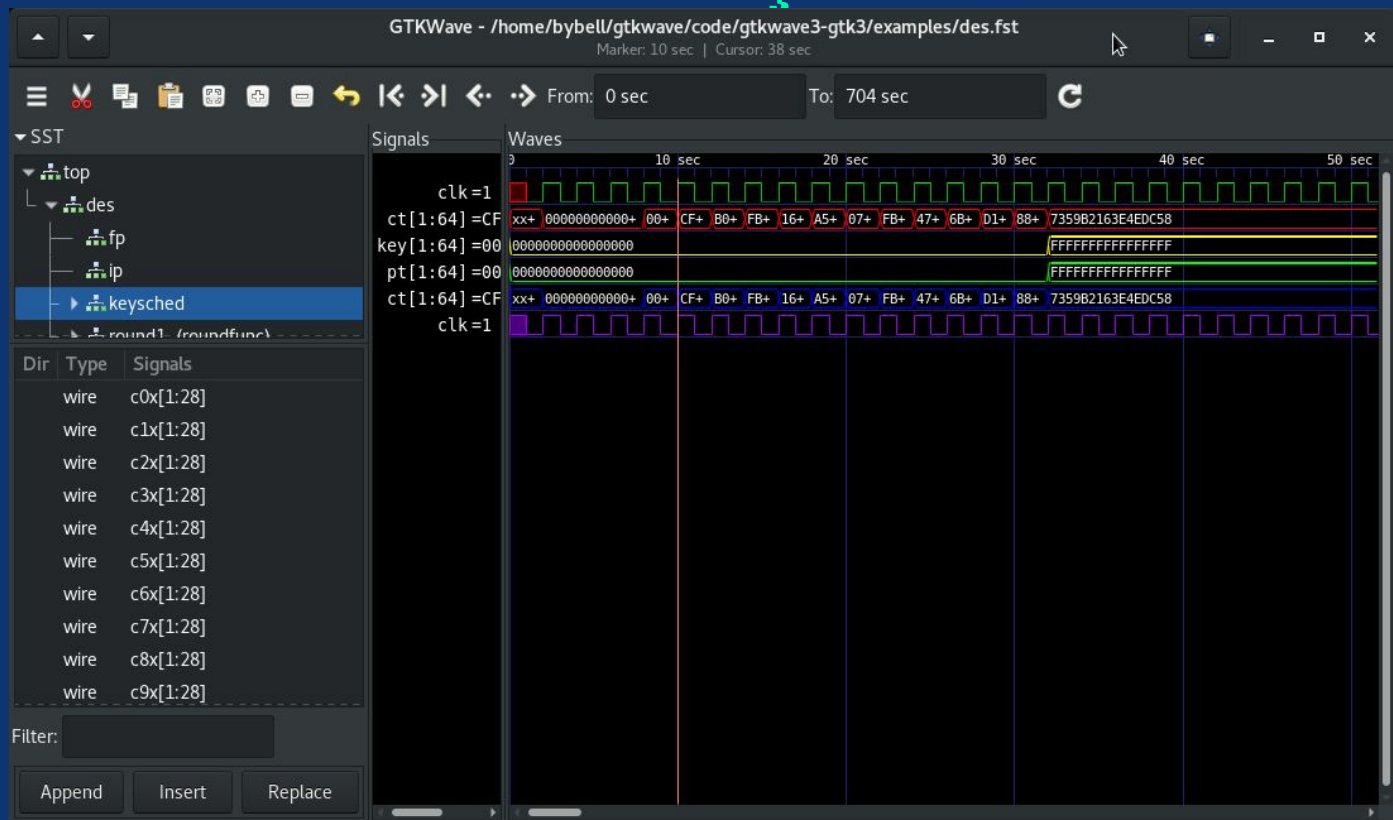


PLATAFORMA DE TESTES

- Depuração em Verilog
 - Testbenches
 - GTKWave
- Entrada vem em binário/hexadecimal

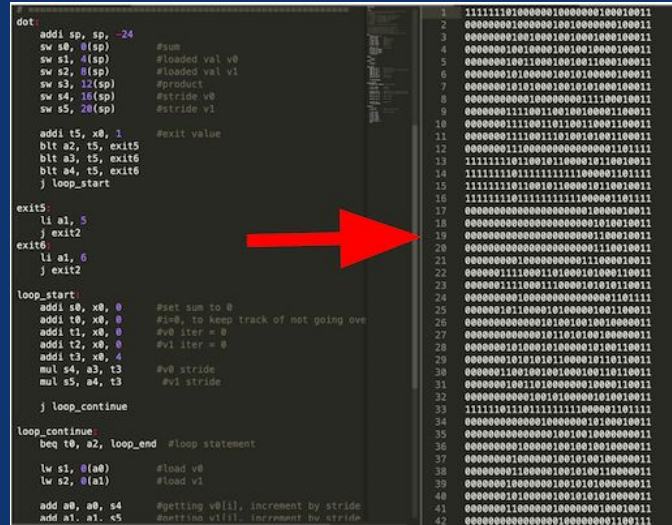


DEPURAÇÃO



ASSEMBLER RISC-V

- Assembler instalado no Colab
- Função *assemble*



```
#
dot
    addi sp, sp, -24
    sw s0, 0(sp)      #sum
    sw s1, 4(sp)      #loaded val v0
    sw s2, 8(sp)      #loaded val v1
    sw s3, 12(sp)     #product
    sw s4, 16(sp)     #stride v0
    sw s5, 20(sp)     #stride v1

    addi t5, x0, 1    #exit value
    blt s2, t5, exit5
    blt s3, t5, exit6
    blt s4, t5, exit6
    j loop_start

exit5:
    li a1, 5
    j exit2
exit6:
    li a1, 6
    j exit2

loop_start:
    addi s0, x0, 0    #set sum to 0
    addi t0, x0, 0    #s0, to keep track of not going ove
    addi t1, x0, 0    #v0 iter = 0
    addi t2, x0, 0    #v1 iter = 0
    addi t3, x0, 4
    mul s4, a3, t3     #v0 stride
    mul s5, a4, t3     #v1 stride
    j loop_continue

loop_continue:
    beq t0, a2, loop_end #loop statement

    lw s1, 0(a0)      #load v0
    lw s2, 0(a1)      #load v1

    add s0, s0, s4     #getting v0[i], increment by stride
    add a1, a1, s5     #getting v1[i], increment by stride
```

1 1111110100000010000000100010011
2 0000000010000001001000000100011
3 0000000100100010010001000100011
4 000000010010000100010010000100011
5 000000010011000100010011000100011
6 00000001010000010010100000100011
7 00000001010100010010101000100011
8 00000000010000000011110010011
9 0000000111001100100001100011
10 0000000111001001100110001100011
11 00000001110011010100101001100011
12 000000011100000000000001101111
13 1111110101000101000010110010011
14 11111101011111111100000101111
15 111111010010011000010110010011
16 11111101111111111100000101111
17 0000000000000000000001000110011
18 000000000000000000000100010011
19 000000000000000000000100010011
20 000000000000000000000110010011
21 0000000100000000011000010011
22 00000111000101000001000110011
23 00000011100011000010101010011
24 00000000100000000000000101111
25 000000101000010100000100100011
26 00000000000010100101001000011
27 00000000000010101010010000011
28 00000001010001010000010010011
29 000000010101010100000101010011
30 00000011001001000100101010011
31 000000100101000000100010011
32 00000000010010100000101010011
33 11111101101111111100000101111
34 000000000000100000001010010011
35 00000000000000000000000100011
36 00000001000010010010010000011
37 00000001000000010010010000011
38 00000001000000010010010000011
39 00000010000000010010100000011
40 00000010100000000101010000011
41 000000110000000000000010010011
42 00000000000000000000001100111

<https://github.com/kcelebi/riscv-assembler>

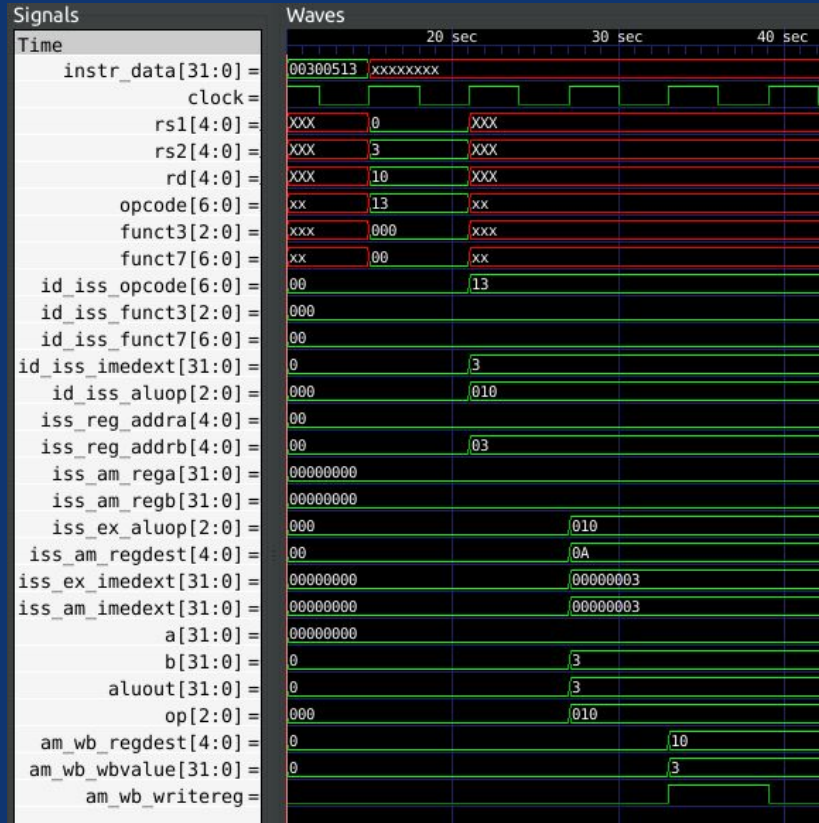
TESTE 1 - ADDI

```
addi x10, x0, 3
```



TESTE 1 - ADDI - I202

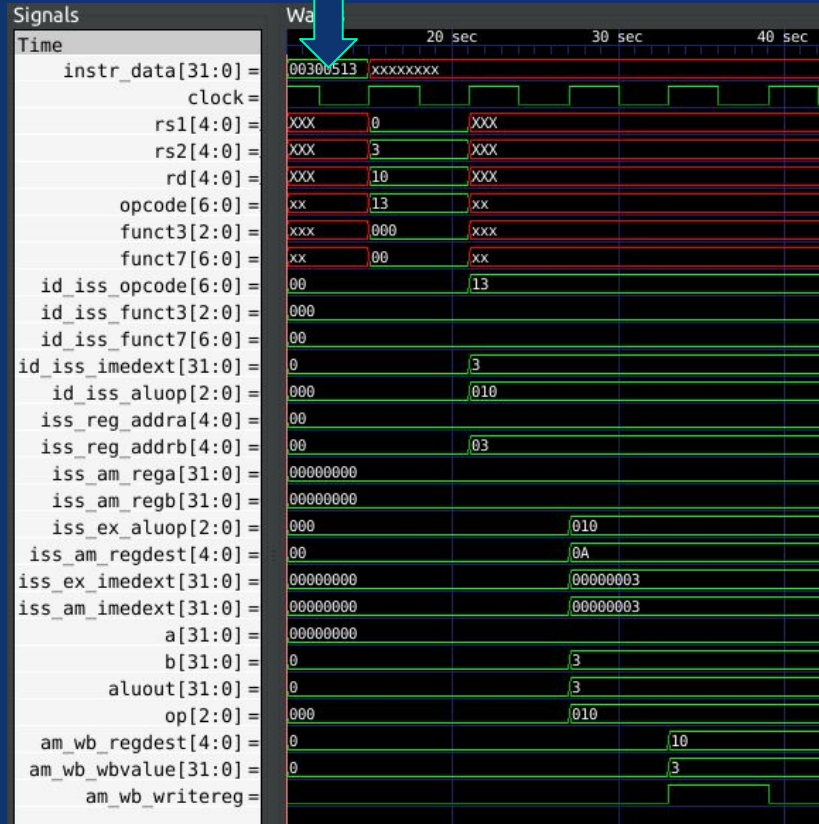
addi x10, x0, 3



TESTE 1 - ADDI - I202

addi x10, x0, 3

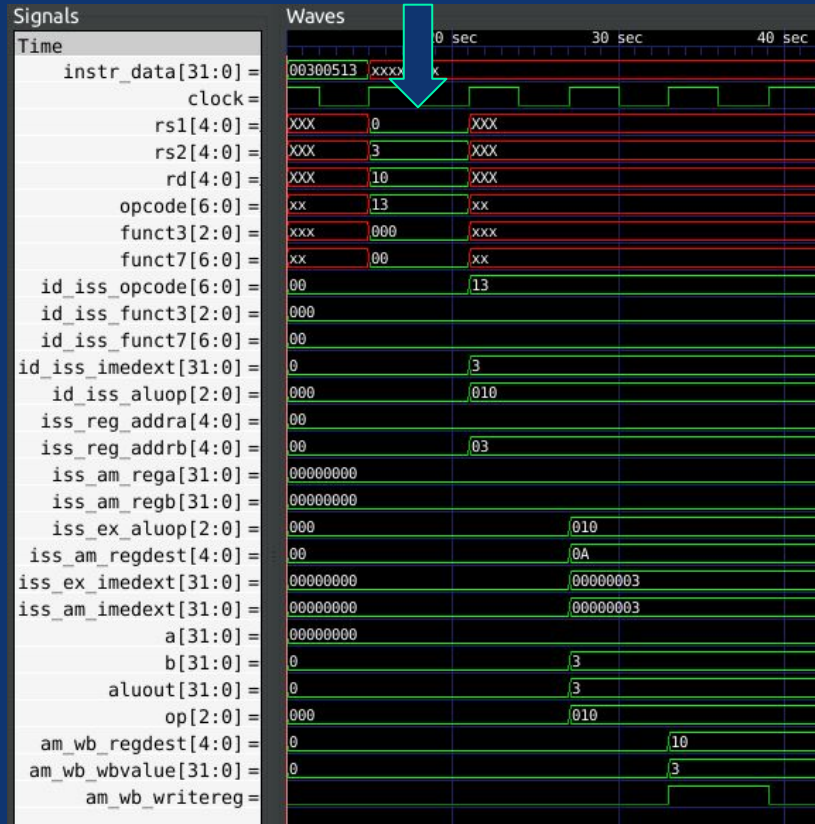
Fetch



TESTE 1 - ADDI - I202

addi x10, x0, 3

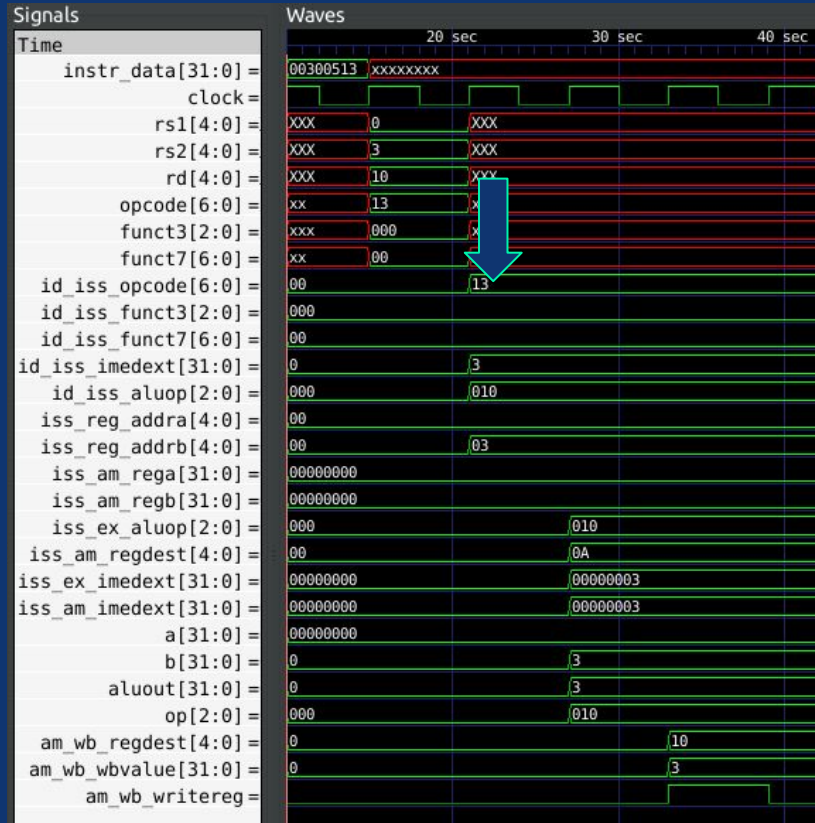
Decode



TESTE 1 - ADDI - I202

addi x10, x0, 3

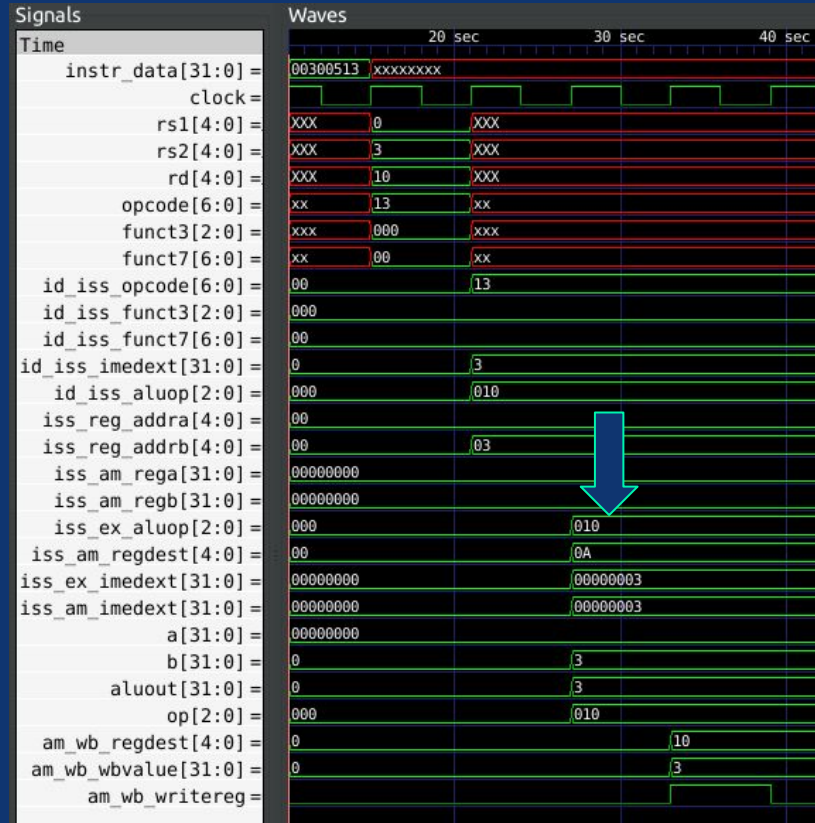
Issue



TESTE 1 - ADDI - I202

addi x10, x0, 3

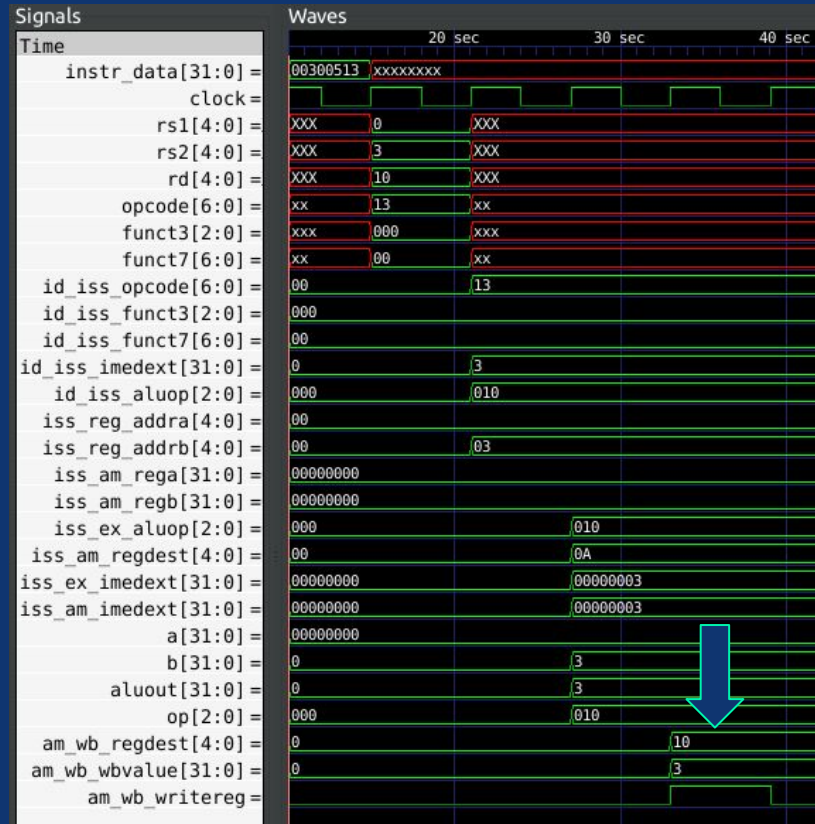
Execução
(ALU)



TESTE 1 - ADDI - I202

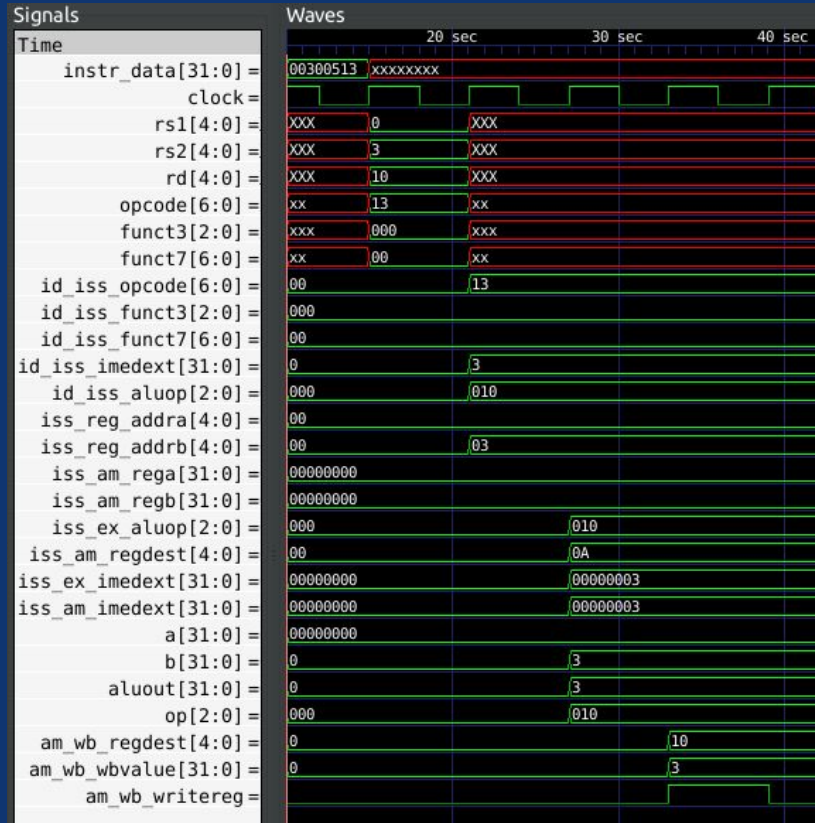
addi x10, x0, 3

Writeback

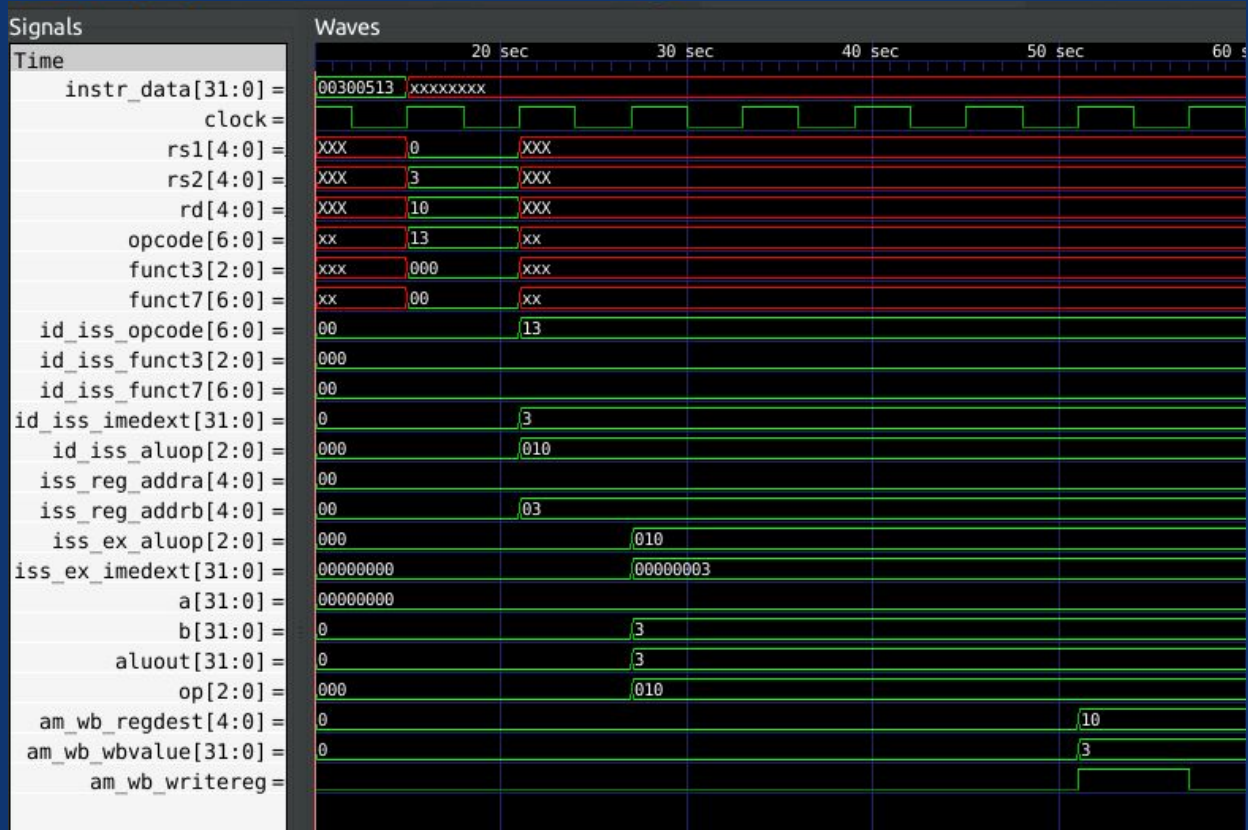


TESTE 1 - ADDI - I202

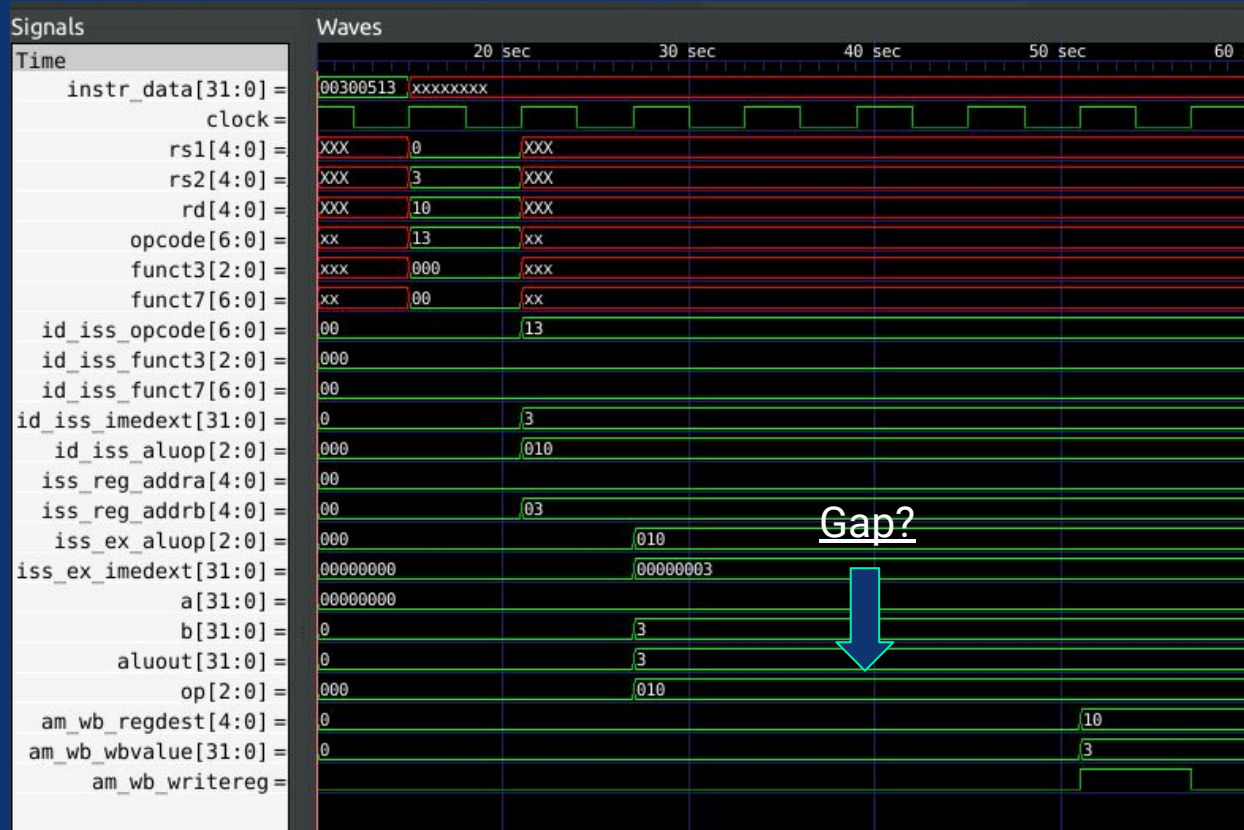
addi x10, x0, 3



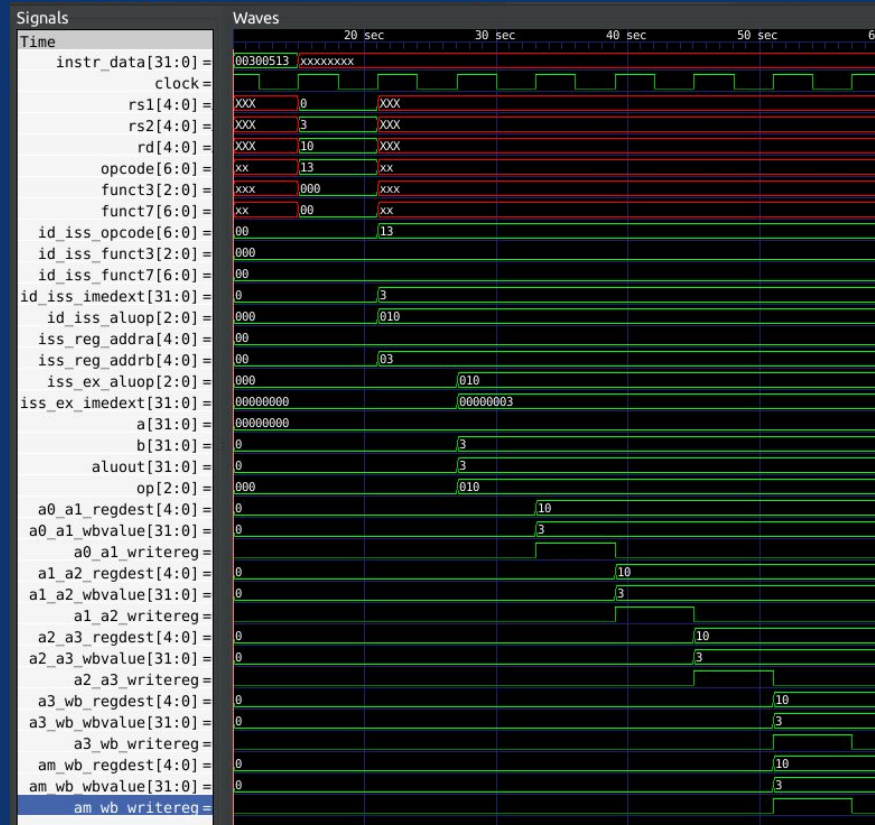
TESTE 1 - ADDI - I4



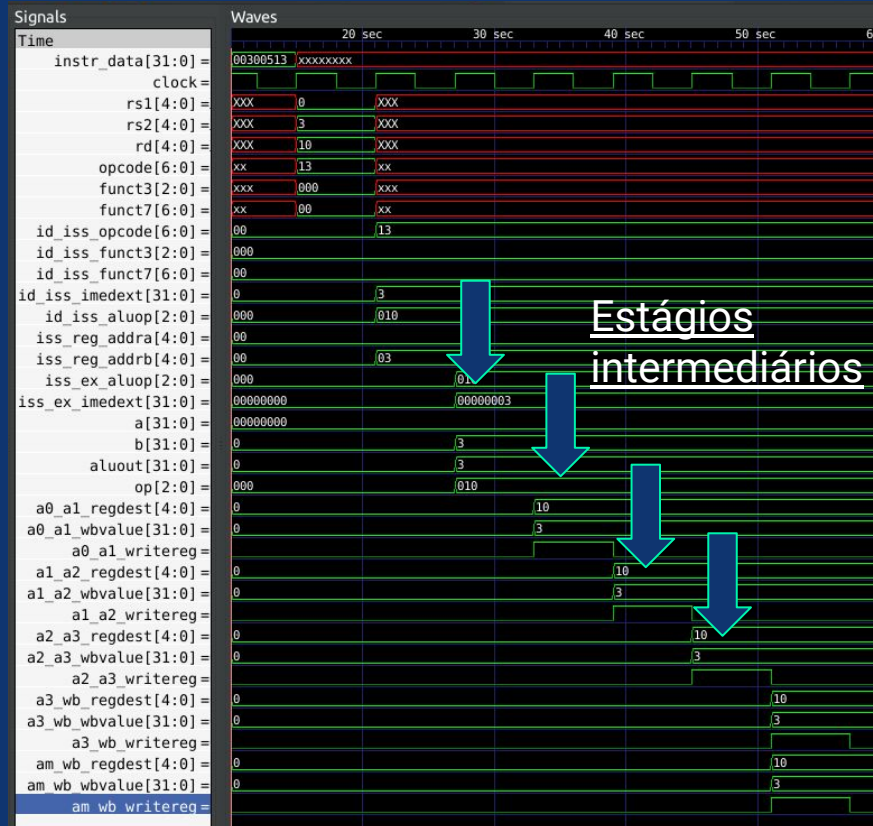
TESTE 1 - ADDI - I4



TESTE 1 - ADDI - I4



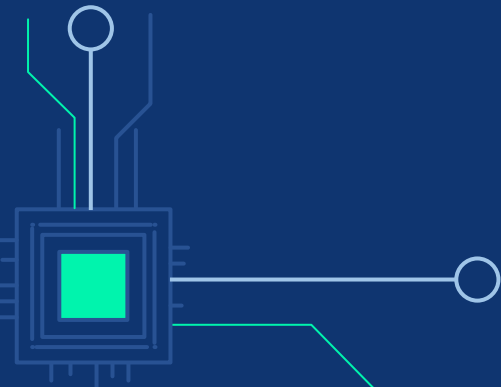
TESTE 1 - ADDI - I4





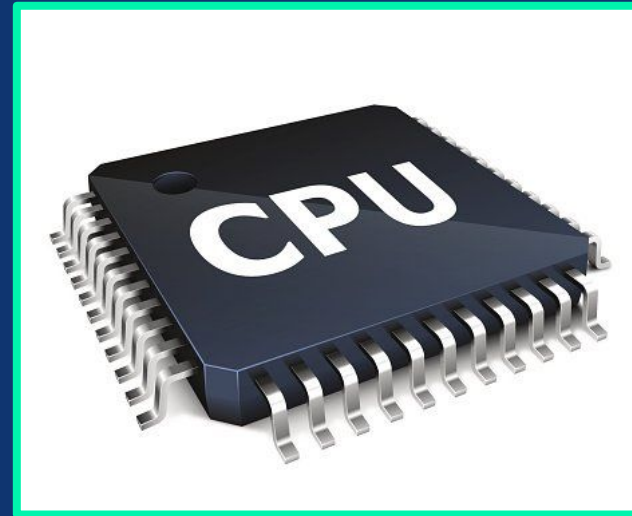
06

CONSIDERAÇÕES FINAIS



CONSIDERAÇÕES FINAIS

- Programação em Verilog
- Plataforma no Colab
- Processadores Superescalares





MUITO OBRIGADO!

