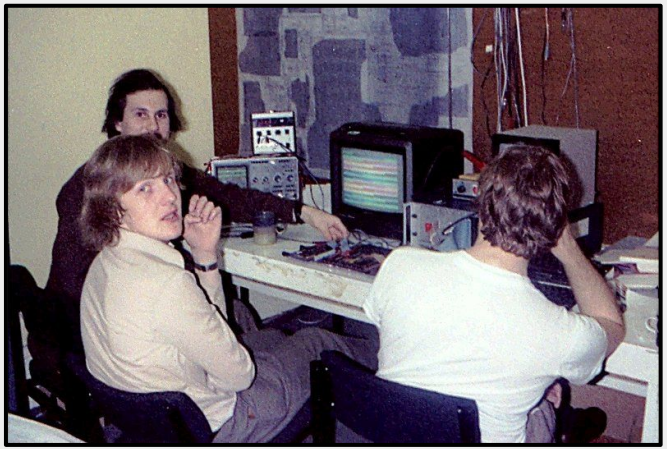
MC851 Projeto em Computação

Relatório Entrega 1

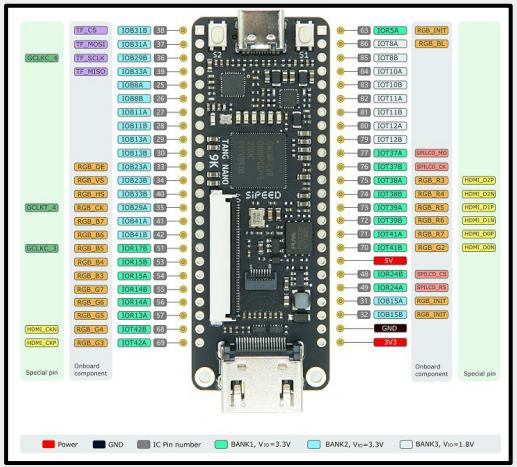
Equipe "RISC-VI":
RA 169374, Daniel Paulo Garcia
RA 182783, Lucca Costa Piccolotto Jordão
RA 185447, Paulo Barreira Pacitti
RA 198435, Guilherme Tavares Shimamoto
RA 216116, Gabriel Braga Proença
RA 221859, Mariana Megumi Izumizawa

1. Introdução

Implementação de um sistema computacional contendo um processador baseado na arquitetura RISC-V, implementado em Verilog. A equipe deve fazer todo o projeto do sistema. O sistema será implantado na placa de desenvolvimento Tang Nano 9k, que possui um *chip* FPGA.



Steve Furber desenvolvendo o que seria a arquitetura ARM, no BBC Micro (~1980)



Datasheet do Tang Nano 9k, placa de desenvolvimento FPGA utilizada no desenvolvimento.

O projeto é baseado na arquitetura RISC-V, e o principal compilador utilizado na implementação com o Verilog é iverilog (Icarus Verilog).

Primeiro circuito lógico implementado com Verilog, um contador binário. Implantado no Tang Nano 9k.

2. Planejamento

Assim como no livro, planejamos inicialmente separar o *datapath* em 5 estágios:

IF: Busca

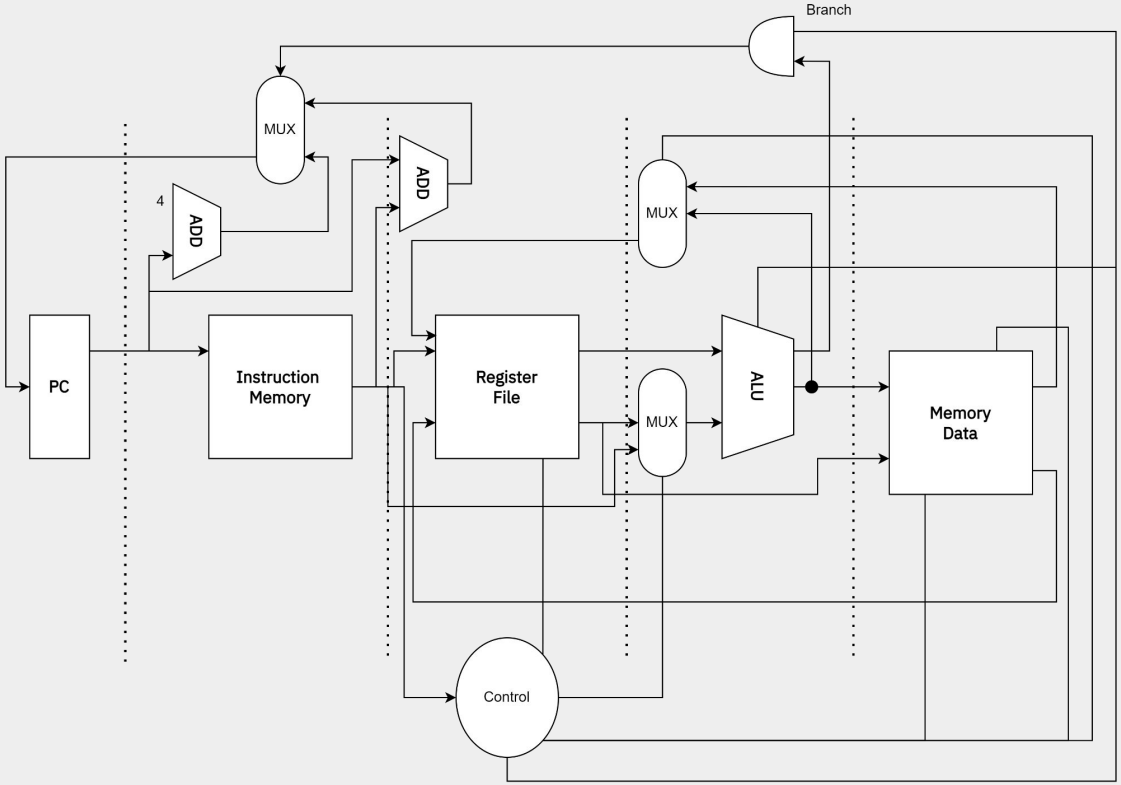
ID: Decodificação e leitura de registradores

EX: Execução

MEM: Acesso à memória

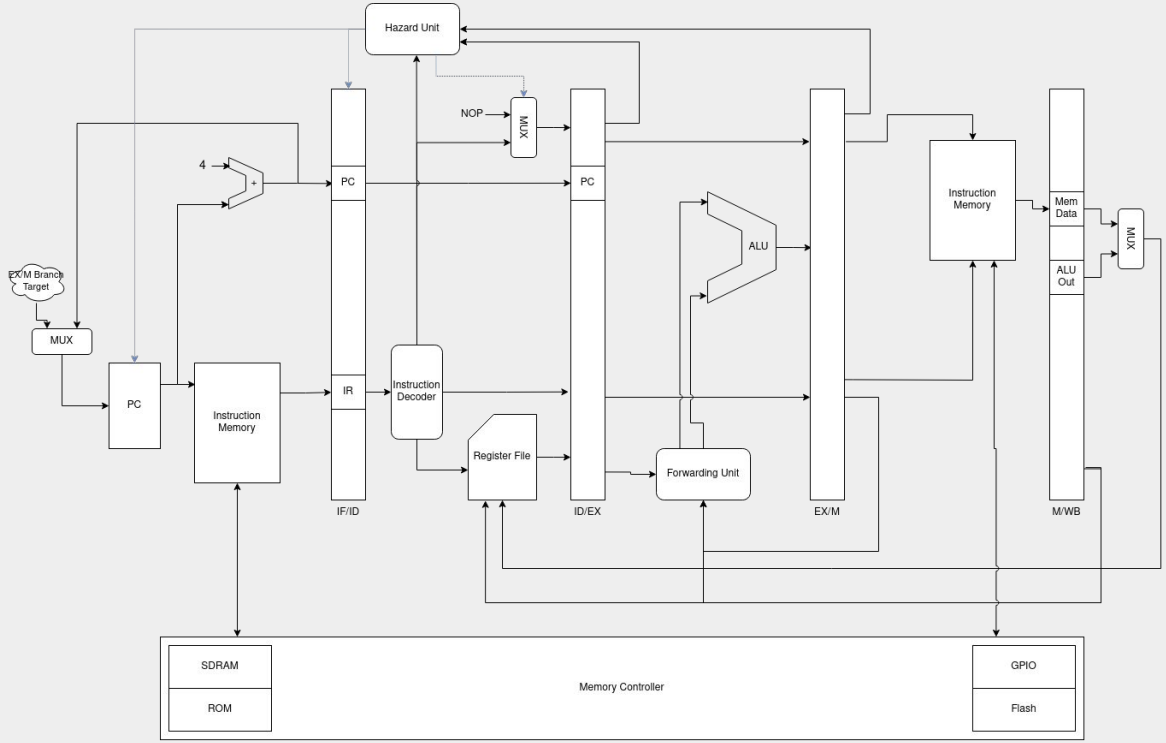
WB: Escrita de resultados

2. Planejamento



Referência de datapath do sistema computacional com arquitetura RISC-V com pipeline de 5 estágios.

2. Planejamento

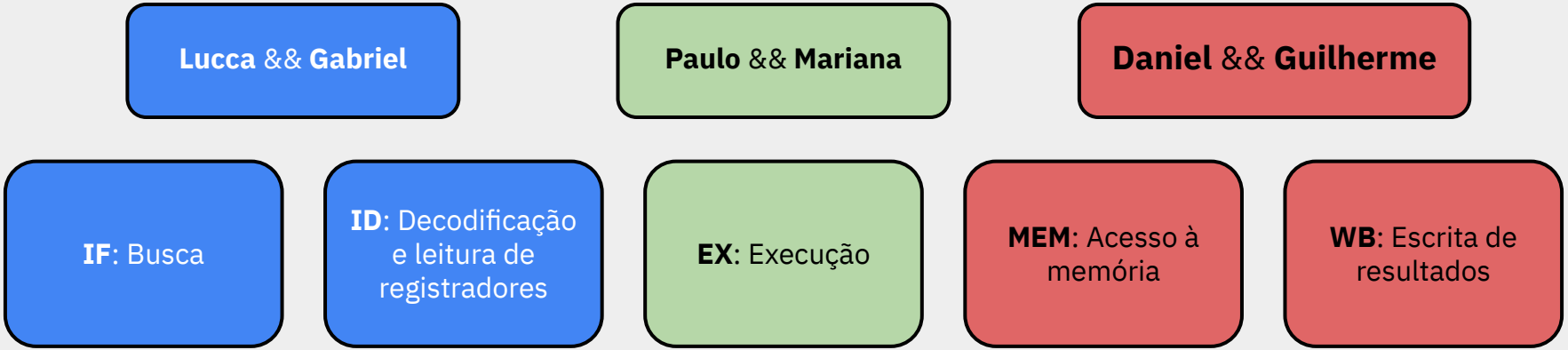


Datapath do sistema computacional com arquitetura RISC-V com pipeline de 5 estágios a ser implementada pelo grupo.

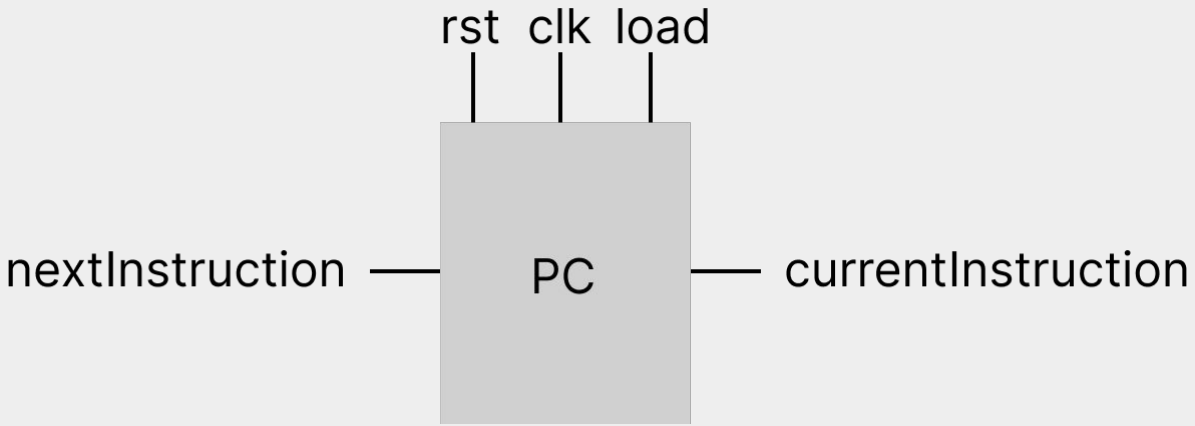
2. Planejamento

A ideia do projeto é implementar sistema computacional contendo um processador baseado na arquitetura RISC-V, implementado com Verilog.

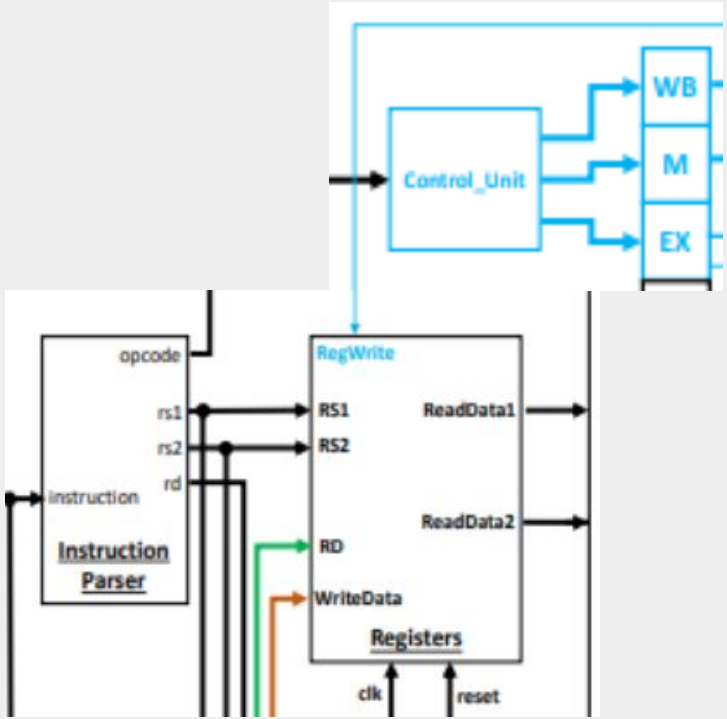
Separamos nosso grupo em duplas:



3. Resultados: IF e ID



3. Resultados: IF e ID



Representação do *Register File*, *Instruction Parser* e *Control Unit*.

3. Resultados: IF e ID

```
module RegisterFile (
    input wire clock,          // Sinal de clock
    input wire reset,        // Sinal de reset assíncrono ativo baixo
    input wire [4:0] read_reg1, // Registrador a ser lido
    input wire [4:0] read_reg2, // Outro registrador a ser lido
    input wire [4:0] write_reg, // O regitrador no qual os dados serão escritos
    input wire write_enable,
    input wire [31:0] write_data, // O dado que será escrito
    output wire [31:0] read_data1, // Dado que foi lido
    output wire [31:0] read_data2 // Outro dado que foi lido
);

    reg [31:0] registers [31:0];

    // Lógica de leitura
    assign read_data1 = (read_reg1 != 5'b0) ? registers[read_reg1] : 32'b0;
    assign read_data2 = (read_reg2 != 5'b0) ? registers[read_reg2] : 32'b0;

    // Lógica de escrita
    always @(posedge clock or negedge reset) begin
        if (!reset) begin
            registers[0] <= 32'b0;
        end else if (write_enable) begin
            registers[write_reg] <= write_data;
        end
    end

endmodule
```

Implementação do *RegFile*, onde os registradores da especificação RISC-V são armazenados e com controles de leitura e escrita.

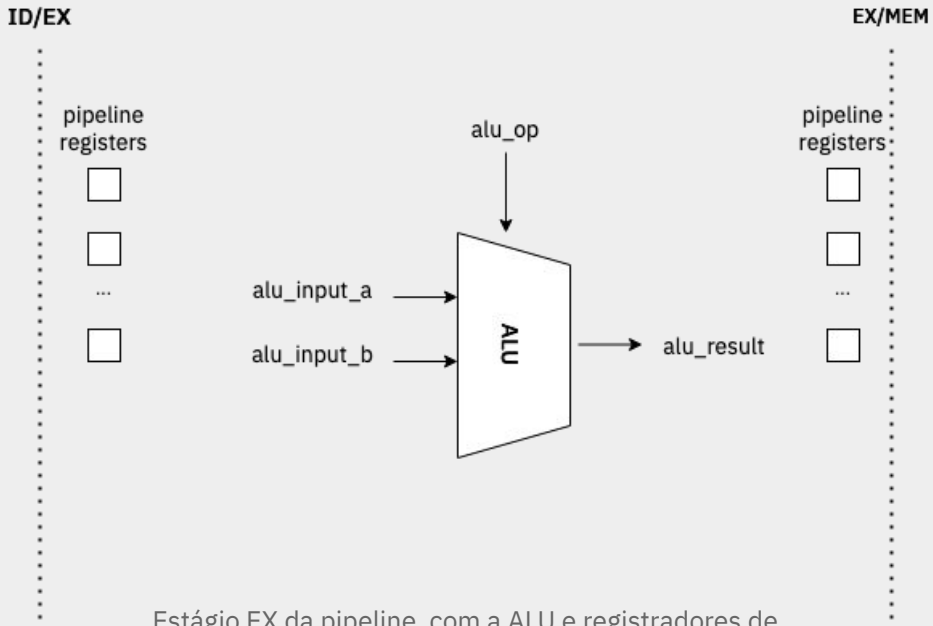
3. Resultados: IF e ID

```
module Decoder (
    input wire [31:0] instruction,
);
    reg [6:0] opcode;
    reg [2:0] funct3;
    reg [6:0] funct7;

    output reg [4:0] read_reg1;
    output reg [4:0] read_reg2;
    output reg [4:0] write_reg;

    // Read registers
    assign read_reg1 = instruction[19:15];
    assign read_reg2 = instruction[24:20];

    // Write register
    assign write_reg = instruction[11:7];

    // Instruction decode
    assign opcode = instruction[6:0];
    assign funct7 = instruction[31:25];
    assign funct3 = instruction[14:12];

    output reg [3:0] alu_input_op;
```

```
always @(*) begin
    case (opcode)
        // R-type instructions
        7'b0110011: begin
            write_enable = 1;
            alu_input_enable = 1;

            if (funct3 == 3'b000) begin
                if (funct7 == 7'b0000000) begin
                    alu_input_op = `ALU_ADD;
                end else begin
                    alu_input_op = `ALU_SUB;
                end
            end else if (funct3 == 3'b001) begin
                alu_input_op = `ALU_SLL;
            end else if (funct3 == 3'b010) begin
                alu_input_op = `ALU_SLT;
            end else if (funct3 == 3'b011) begin
                alu_input_op = `ALU_SLTU;
            end else if (funct3 == 3'b100) begin
                alu_input_op = `ALU_XOR;
            end else if (funct3 == 3'b101) begin
                if (funct7 == 7'b0000000) begin
                    alu_input_op = `ALU_SRL;
                end else begin
                    alu_input_op = `ALU_SRA;
                end
            end else if (funct3 == 3'b110) begin
                alu_input_op = `ALU_OR;
            end else if (funct3 == 3'b111) begin
                alu_input_op = `ALU_AND;
            end
        end
    endcase
end
```

Decoder: decodifica a instrução recebida do *Instruction Memory* e envia os sinais de controle para os outros módulos do sistema. Por escolha de *design* de projeto, a equipe decidiu incorporar o que é conhecido como *Control Unit* dentro do *Decoder*.

3. Resultados: EX



Estágio EX da pipeline, com a ALU e registradores de pipeline. Diagrama do que foi implementado até então.

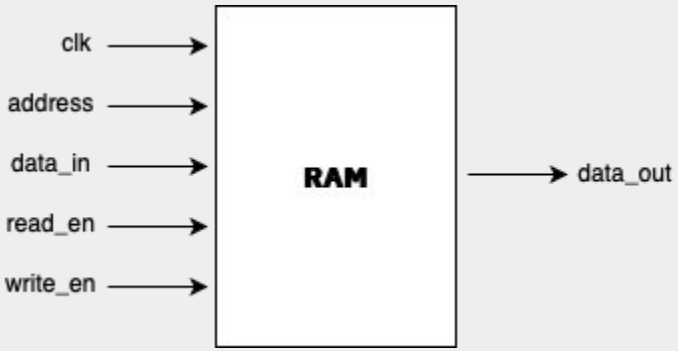
3. Resultados: EX

```
// ALU_op: ALU operation
//-------------------------- ARITHMETIC
`define ALU_ADD     4'b0000
`define ALU_SUB     4'b0001
// -------------------------- LOGIC
`define ALU_AND     4'b0010
`define ALU_OR      4'b0011
`define ALU_XOR     4'b0100
// -------------------------- SHIFT
`define ALU_SLL     4'b0101
`define ALU_SRL     4'b0110
`define ALU_SRA     4'b0111
// -------------------------- COMPARE
`define ALU_SLT     4'b1000
`define ALU_SLTU    4'b1001
```

```
// rtl/stages/ex.v
module alu_module (
    input          alu_input_enable,
    input  [3:0]   alu_input_op,
    input  [31:0]  alu_input_a,
    input  [31:0]  alu_input_b,

    output [31:0]  alu_output_result
);

reg [31:0]   alu_register_result;

always @ (alu_input_op or alu_input_a or alu_input_b)
begin
    alu_register_result <= 32'h0;

    if (alu_input_enable)
    begin
        case (alu_input_op)
        // ARITHMETIC OPS
            `ALU_ADD: alu_register_result <= (alu_input_a + alu_input_b);
            `ALU_SUB: alu_register_result <= (alu_input_a - alu_input_b);
        // LOGIC OPS
            `ALU_AND: alu_register_result <= (alu_input_a & alu_input_b);
            `ALU_OR:  alu_register_result <= (alu_input_a | alu_input_b);
            `ALU_XOR: alu_register_result <= (alu_input_a ^ alu_input_b);
        // SHIFT OPS
            `ALU_SLL:  alu_register_result <= alu_input_a << alu_input_b[4:0]; // need to set a
range otherwise it will binary extend the number
            `ALU_SRL:  alu_register_result <= alu_input_a >> alu_input_b[4:0];
            `ALU_SRA:  alu_register_result <= $signed(alu_input_a) >>> alu_input_b[4:0];
        // COMPARE OPS
            `ALU_SLT:  alu_register_result <= (alu_input_a < alu_input_b ? 1 : 0);
            `ALU_SLTU:  alu_register_result <= ($signed(alu_input_a) < $signed(alu_input_b) ? 1
: 0);
        endcase
    end
end

assign alu_output_result = alu_register_result;

endmodule
```

3. Resultados: MEM e WB

- Módulo de RAM.
- Módulo de controlador de memória (RAM + ROM).
- Módulo de ROM e decisões de implementação.



Esquema do módulo de memória.

3. Resultados: MEM e WB

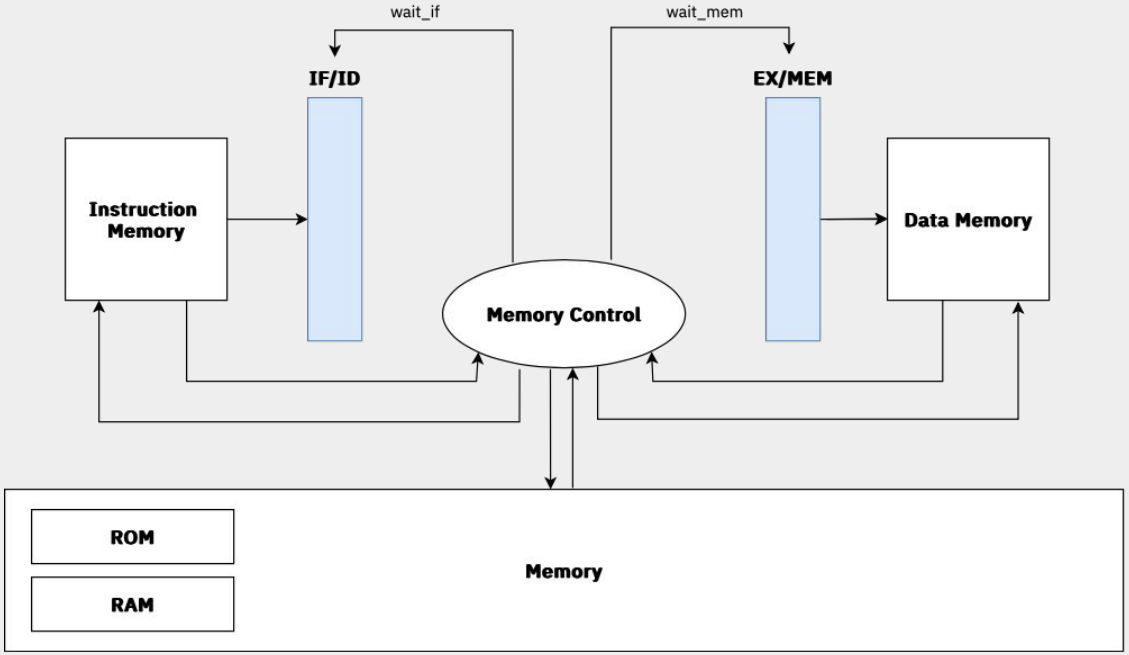


Diagrama da relação entre os estágios da *pipeline* e o módulo de memória.

3. Resultados: MEM e WB

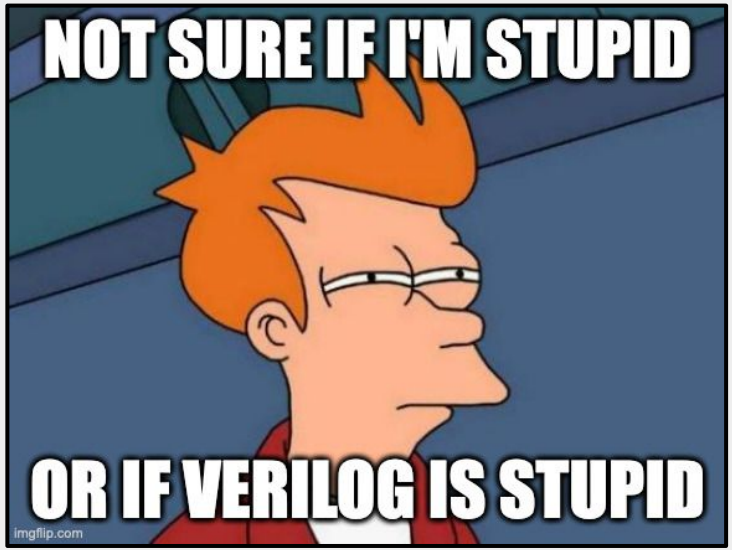
- Enquanto MEM escreve algo na memória **`wait_if`** deve ser 1
- Enquanto o módulo não está em uso **`wait_if`** e **`wait_mem`** devem ser 0.
- Enquanto MEM lê algo da memória **`wait_if`** deve ser 1.
- O valor de saída do componente deve ser o valor salvo na memória.

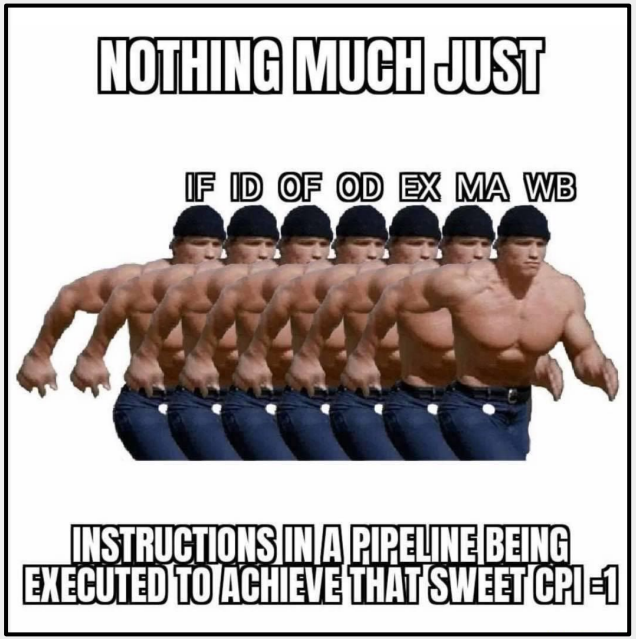
```
#10
op_mem = 2'b10;
address = 10;
data_in = 42;

#11
if (wait_if != 1)
    $display("wait_if should be 1 while mem is using the component");

#20
op_mem = 2'b0;

#21
if (wait_if != 0 && wait_mem != 0)
    $display("wait_if and wait_mem should be 0 while the component is not in use");

#30
op_mem = 2'b10;
address = 10;

#31
if (wait_if != 1)
    $display("wait_if should be 1 while mem is using the component");

#40
op_mem = 2'b0;

#50
if (data_out != 42)
    $display("data_out should be 00000000000000000000000000101010, but is %b", data_out);
else
    $display("passed!");
end
```

3. Resultados: Memória

- Em comparação à arquitetura atual, no futuro pretende-se:
  - Transformar a memória de instruções (atualmente em ROM) em cache Li1.
  - Requer uma interface com a memória principal para carregar programas.
  - Memória principal na SDRAM do Tang Nano
  - Transformar a memória de dados (“RAM”) em cache Ld1.
- Integração da CPU:
  - Cada estágio da pipeline é um processo que executa paralelamente a cada sinal de clock.

4. Futuro


NOT SURE IF I'M STUPID
OR IF VERILOG IS STUPID
imgflip.com


NOTHING MUCH JUST
IF ID OF OD EX MA WB
INSTRUCTIONS IN A PIPELINE BEING EXECUTED TO ACHIEVE THAT SWEET CPI =1

4. Futuro

Finalização pendências do processador

Juntar os módulos criados pelo grupo, visando implementar os conjuntos de instruções RV32I

Estudos e testes para uso do cache L1 e periféricos

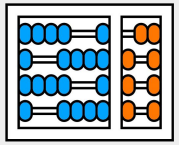
Divisão do grupo para iniciar o desenvolvimento necessário para a segunda entrega

Implementações do cache L1 e periféricos

Divisão do grupo para desenvolver as atividades para a segunda entrega

5. Referências

D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017



Obrigado!

Equipe "RISC-VI":
RA 169374, Daniel Paulo Garcia
RA 182783, Lucca Costa Piccolotto Jordão
RA 185447, Paulo Barreira Pacitti
RA 198435, Guilherme Tavares Shimamoto
RA 216116, Gabriel Braga Proença
RA 221859, Mariana Megumi Izumizawa