# Arquivo Principal do Projeto da CPU

## Módulos do Arquivo `top.sv`:

- Arquivo `top.sv`: .
  - Módulo `controller`: Módulo que contém o decodificador das instruções da CPU.
  - Módulo `datapath`: Módulo referente ao bloco ***DATA PATH* da CPU**.
  - Módulo `riscv_single`: **Camada superior da CPU** RISC_V do tipo ***SINGLE-CYCLE***.
  - Módulo `main_module`: Módulo **criado dentro** do módulo `top`.
  - Módulo `top`: **MÓDULO PRINCIPAL** do projeto.

---

## 1 - Módulo `controller`

- Módulo que contém o decodificador das instruções da CPU.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `opcode`: Bits 0 a 7 da instrução.
    - `funct3`: Bits 12 a 14 da instrução.
    - `funct7b0`: Bit 25 da instrução.
    - `funct7b5`: Bit 30 da instrução.
    - `of_z`: *Output Flag* `z` - Indica se **`result` é 0**.
  - **Saídas**:
    - `result_src`: Array de 2 bits.
    - `mem_write`: Sinal indicando se a instrução deverá escrever dados na memória.
    - `pc_src`: Sinal indicando se deverá selecionar `pc_plus4` ou `pc_target`.
    - `alu_src`: Sinal indicando se devemos usar o valor gerado na saída `result` da ALU.
    - `reg_write`: Sinal indicando se deverá ser realizada uma operação de escrita no *register file*..

* **is_jal**: Sinal indicando se a instrução atual é `jal`.

* **is_jalr**: Sinal indicando se a instrução atual é `jalr`.

* **imm_src**: Array de 3 bits.

* **alu_ctrl**: Código da instrução que será executada na ALU.

```
module controller(input  logic [6:0] opcode,
                  input  logic [2:0] funct3,
                  input  logic       funct7b0,
            input  logic       funct7b5,
                  input  logic       of_z,
                  output logic [1:0] result_src,
                  output logic       mem_write,
                  output logic       pc_src,
                  output logic       alu_src,
                  output logic       reg_write,
                  output logic       is_jal,
            output logic       is_jalr,
                  output logic [2:0] imm_src,
                  output logic [4:0] alu_ctrl );
//------------------------------------------------------------------------------------
      logic [1:0] alu_op;
      logic       branch;

    // Instancia do modulo 'main_dec'
      main_dec md ( .opcode( opcode ),
                    .result_src( result_src ),
                    .mem_write( mem_write ),
                    .branch( branch ),
                    .alu_src( alu_src ),
                    .reg_write( reg_write ),
                    .is_jal( is_jal ),
              .is_jalr( is_jalr ),
                    .imm_src( imm_src ),
                    .alu_op( alu_op ) );

    // Instancia do modulo 'alu_dec2'
      alu_dec  ad ( .opcode( opcode ),
                    .funct3( funct3 ),
              .funct7b5( funct7b5 ),
              .funct7b0( funct7b0 ),
              .alu_op( alu_op ),
                    .alu_ctrl( alu_ctrl ) );


    // --> Atribuicao no sinal 'pc_src'
      assign pc_src = branch & of_z | (is_jal || is_jalr);
endmodule
```

---

## 2 - Módulo `datapath`

- Módulo referente ao bloco ***DATA PATH* da CPU**.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `clk`: Entrada que recebe os pulsos de *clock*.
    - `reset`: Reset assíncrono.
    - `result_src`: Array de 2 bits.
    - `pc_src`: Sinal indicando se deverá selecionar `pc_plus4` ou `pc_target`.
    - `alu_src`: Sinal indicando se devemos usar o valor gerado na saída `result` da ALU.
    - `wr_en`: Sinal indicando se o valor da entrada `w_data` deverá ser escrito no registrador `w_addr`.
    - `imm_src`: *Array* de 3 bits indicando se a instrução atual recebe um valor imediato (*immediate*).
    - `alu_ctrl`: Código da instrução que será executada na ALU.
    - `instr`: Instrução de 32 bits a ser decodificada.
    - `rd`: Endereço do registrador de destino.
    - `rs1`: Endereço do registrador com o primeiro operando.
    - `rs2`: Endereço do registrador com o segundo operando.
    - `read_data`: *Array* com o valor lido em um dos registradores do *register file*.
  - **Saídas**:
    - `of_c`: *Output Flag* `c` - Indica se a operação **gerou um *carry-out***.
    - `of_n`: *Output Flag* `n` - Indica se **`result` é negativo**.
    - `of_v`: *Output Flag* `v` - Indica a ocorrência de ***overflow***.
    - `of_z`: *Output Flag* `z` - Indica se **`result` é 0**.
    - `pc`: Registrador que armazena o endereço da instrução atual na memória de programa (*program counter*).
    - `alu_result`: Resultado produzido na ALU.
    - `write_data`: Dados a serem escritos no registrador de destino do *register file*.

```
module datapath #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                 ( input  logic             clk,          // Sinal referente aos pulsos de clock
                   input  logic             reset,        // Reset sincrono
```

```
                  input  logic [1:0]       result_src,   // Codigo para selecionar entre 'alu_result',
                  input  logic             pc_src,       // Codigo para selecionar entre 'pc_plus4' e
                  input  logic             alu_src,      // Codigo para selecionar entre 'write_data'
                  input  logic             wr_en,        // Write Enable
                  input  logic [2:0]       imm_src,      // Codigo do tipo de instrucao cujo immediate
                  input  logic [4:0]       alu_ctrl,     // Codigo da instrucao selecionada
                  input  logic [31:0]      instr,        // Instrucao com 32 bits
                  input  logic [4:0]       rd,           // Campo referente ao endereco do registrador
            input  logic [4:0]       rs1,          // Campo referente ao endereco do registrador com o
            input  logic [4:0]       rs2,          // Campo referente ao endereco do registrador com o
            input  logic [END_IDX:0] read_data,    // Dados lidos em um registrador
                  output logic             of_c,         // 'Output Flag' 'c': Indica se a operacao ge
            output logic             of_n,         // 'Output Flag' 'n': Indica se o resultado eh nega
            output logic             of_v,         // 'Output Flag' 'v': Indica a ocorrencia de overfl
            output logic             of_z,         // 'Output Flag' 'z': Indica se o resultado eh 0
                  output logic [END_IDX:0] pc,           // Contador do programa
                  output logic [END_IDX:0] alu_result,   // Resultado produzido na ALU
                  output logic [END_IDX:0] write_data ); // Dados a serem escritos no registrador de d
//-------------------------------------------------------------------------------------------
      // Constante: valor 4 com 'DATA_WIDTH' bits
    parameter VAL_4 = { { (DATA_WIDTH-3) { 1'b0 } }, 3'b100 };

    // Arrays usados nos modulos instanciados abaixo
    logic [END_IDX:0] pc_next, pc_plus4, pc_target;
      logic [END_IDX:0] imm_ext, src_a, src_b, result;
    logic taken_br, branch;

    /*************************
      ** Next PC logic         **
    *************************/
    assign branch = taken_br || pc_src;

    ff_rst #( .DATA_WIDTH(DATA_WIDTH) ) pcreg
              ( .clk( clk ),
                .reset( reset ),
                .d( pc_next ),
                .q( pc ) );

      adder #( .DATA_WIDTH(DATA_WIDTH) ) pcadd4
             ( .op_val1( pc ),
               .op_val2( VAL_4 ),
               .sum_result( pc_plus4 ) );

      adder #( .DATA_WIDTH(DATA_WIDTH) ) pcaddbranch
             ( .op_val1( pc ),
               .op_val2( imm_ext ),
               .sum_result( pc_target ) );

      take_branch #( .DATA_WIDTH(DATA_WIDTH) ) tk_br
             ( .src1_value( src_a ),
               .src2_value( src_b ),
               .alu_control( alu_ctrl ),
               .taken_br( taken_br ));
```

```
  mux2 #( .DATA_WIDTH(DATA_WIDTH) ) pcmux
        ( .d0( pc_plus4 ),
          .d1( pc_target ),
          //.sel( pc_src ),
          .sel( branch ),
       .y( pc_next ) );

  /*************************
  ** Register File logic   **
*************************/
  reg_file #( .DATA_WIDTH(DATA_WIDTH) ) rf
            ( .clk( clk ),
              .wr_en( wr_en ),          // Sinal de 'Write Enable'
              .r_addr1( rs1 ),          // Endereco do registrador com o 1o valor
              .r_addr2( rs2 ),          // Endereco do registrador com o 2o valor
              .w_addr( rd ),
              .w_data( result ),
              .r_data1( src_a ),        // Valor lido no 1o registrador
              .r_data2( write_data ) ); // Valor lido no 2o registrador

  // --> Array de 32 bits com o conteudo do valor imediato (immediate)
extend ext ( .instr( instr[31:7] ), // As instrucoes possuem o tamanho fixo de 32 bits
               .imm_src( imm_src ),
               .imm_ext( imm_ext ) );

/*************************
  ** ALU logic             **
*************************/
// Mux para selecionar o valor do segundo operando da ALU
  mux2 #( .DATA_WIDTH(DATA_WIDTH) ) srcbmux
        ( .d0( write_data ),  // Valor lido no 2o registrador do register file
          .d1( imm_ext ),     // Valor imediato (immediate)
          .sel( alu_src ),
          .y( src_b ) );

// Instancia do modulo com a ALU da CPU
  alu  #( .DATA_WIDTH(DATA_WIDTH) ) alu
        ( .src1_value( src_a ),
          .src2_value( src_b ),
          .alu_ctrl( alu_ctrl ),
          .result( alu_result ),
          .of_c( of_c ),
       .of_n( of_n ),
       .of_v( of_v ),
       .of_z( of_z ) );

// Mux para selecionar o resultado a ser enviado para 'result'
  mux4 #( .DATA_WIDTH(DATA_WIDTH) ) resultmux
        ( .d0( alu_result ),  // Resultado gerado na ALU
          .d1( read_data ),   // Valor lido na memoria RAM (data memory)
          .d2( pc_plus4 ),    // Endereco atual na ROM, mais 4.
          .d3( imm_ext ),     // Valor imediato lido na instrucao
       .sel( result_src ),
```

```
            .y( result ) );
endmodule
```

---

# 3 - Módulo `riscv_single`

- **Camada superior da CPU** RISC_V do tipo ***SINGLE-CYCLE***.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `clk`: Entrada que recebe os pulsos de *clock*.
    - `reset`: Reset assíncrono.
    - `instr`: *Array* com o valor lido em um dos registradores do *register file*.
    - `read_data`: *Array* com o valor lido em um dos registradores do *register file*.
  - **Saídas**:
    - `mem_write`: Sinal indicando se a instrução deverá escrever dados na memória.
    - `pc`: Registrador que armazena o endereço da instrução atual na memória de programa (*program counter*).
    - `alu_result`: Resultado produzido na ALU.
    - `write_data`: Dados a serem escritos no registrador de destino do *register file*.

```
module riscv_single #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                      ( input  logic             clk,          // Sinais de clock
                        input  logic             reset,        // Reset sincrono
                        input  logic [31:0]      instr,        // Instrucao de 32 bits
                        input  logic [END_IDX:0] read_data,     // Dados lidos em um registrador
                        output logic             mem_write,     // Sinal que indica se deve realizar uma
                        output logic [END_IDX:0] pc,            // Contadr do programa
                        output logic [END_IDX:0] alu_result,    // Resultado produzido na ALU
                        output logic [END_IDX:0] write_data );  // Dados a serem escritos no registrador
//------------------------------------------------------------------------------
      logic alu_src, reg_write, pc_src, is_jal, is_jalr;
    logic of_c, of_n, of_v, of_z;

    logic funct7b5, funct7b0;
    logic [1:0] result_src;
      logic [2:0] funct3;
    logic [2:0] imm_src;
    logic [4:0] alu_ctrl;
    logic [4:0] rd, rs1, rs2;
    logic [6:0] opcode;
```

```
    // Instancia do modulo 'instr_fields'
    instr_fields c_instr ( .instr(instr),
                           .opcode(opcode),
                           .rd(rd),
                           .funct3(funct3),
                           .rs1(rs1),
                           .rs2(rs2),
                           .funct7b0(funct7b0),
                           .funct7b5(funct7b5) );

    // Instancia do modulo 'controller'
    controller c( .opcode( opcode ),
                  .funct3( funct3 ),
                  .funct7b5( funct7b5 ),
              .funct7b0( funct7b0 ),
                  .of_z( of_z ),
                  .result_src( result_src ),
                  .mem_write( mem_write ),
                  .pc_src( pc_src ),
                  .alu_src( alu_src ),
                  .reg_write( reg_write ),
                  .is_jal( is_jal ),
              .is_jalr( is_jalr ),
                  .imm_src( imm_src ),
                  .alu_ctrl( alu_ctrl ) );

    // Instancia do modulo 'datapath'
    datapath #( .DATA_WIDTH(DATA_WIDTH) ) dp
                ( .clk( clk ),
                  .reset( reset ),
                  .result_src( result_src ),
                  .pc_src( pc_src ),
                  .alu_src( alu_src ),
                  .wr_en( reg_write ),
                  .imm_src( imm_src ),
                  .alu_ctrl( alu_ctrl ),
                  .instr( instr ),
            .rd(rd),
            .rs1(rs1),
            .rs2(rs2),
                  .read_data( read_data ),
                  .of_c( of_c ),
            .of_n( of_n ),
            .of_v( of_v ),
            .of_z( of_z ),
                  .pc( pc ),
                  .alu_result( alu_result ),
                  .write_data( write_data )  );
endmodule
```

---

# 4 - Módulos Referentes à Camada Superior do Projeto

## 4.1 - Módulo `main_module`

- Módulo **criado dentro** do módulo `top`.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
    - `ADDR_W_ROM`: Comprimento dos endereços de memória da memória de programa (*program memory*).
    - `ADDR_W_RAM`: Comprimento dos endereços de memória da memória para dados (*data memory*).
    - `HEX_FILE`: Caminho completo para o arquivo com as instruções em Assembly RISC-V compiladas no formato hexadecimal.
  - **Entradas**:
    - `clk`: Entrada que recebe os pulsos de *clock*.
    - `reset`: Reset assíncrono.
  - **Saídas**:
    - `mem_write`: Sinal indicando se a instrução deverá escrever dados na memória.
    - `write_data`: Dados a serem escritos no registrador de destino do *register file*.
    - `data_addr`: Endereço da isntrução atual na memória de programa (*program memory*).

```
module main_module #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1,  parameter ADDR_W_ROM=10
                  parameter ADDR_W_RAM=10, parameter HEX_FILE="riscvtest.txt" )
                    ( input  logic             clk,
                      input  logic             reset,
                      output logic             mem_write,
                      output logic [END_IDX:0] write_data,
                      output logic [END_IDX:0] data_addr );
//------------------------------------------------------------------------------------
    // Sinais e variaveis usadas aqui:
    logic [31:0]      instr;
    logic [END_IDX:0] read_data;
    logic [END_IDX:0] pc;
    //logic waitrequest, readdatavalid;

      // --> Instanciacao de um processador RISC-V Single-Cycle
      riscv_single #( .DATA_WIDTH(DATA_WIDTH) ) rvsingle
                    ( .clk( clk ),
                      .reset( reset ),
                      .instr( instr ),
                      .read_data( read_data ),
                      .mem_write( mem_write ),
                .pc( pc ),
                      .alu_result( data_addr ),
                      .write_data( write_data ) );
```

```
    // --> 'Instruction Memory' ou 'Program Memory'
    instr_mem #( .DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_W_ROM), .HEX_FILE(HEX_FILE) ) imem
              ( .addr( pc ), .instr( instr ) );


    // --> Instancia de uma 'Data Memory' (memoria RAM)
    data_mem_single #( .DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_W_RAM)) dmem
                       ( .clk( clk ),
                         .w_en( mem_write ),
                         .addr( data_addr[(ADDR_W_RAM-1):0] ),
                         .w_data( write_data ),
                         .r_data( read_data ) );
endmodule
```

## 4.1 - Módulo `top`

- **MÓDULO PRINCIPAL** do projeto.
  - **Constantes**: NÃO POSSUI.
  - **Entradas**:
    - `MAX10_CLK1_50`: Pino referente ao *clock* de 50 MHz.
  - **Saídas**: NÃO POSSUI.

```
module top ( input  logic       MAX10_CLK1_50 );
//-------------------------------------------------------------------------------------------
    // --> Constantes:
    parameter RST = 1'b0;

    // Sinais e variaveis usadas aqui:
    logic        MemWrite;
    logic [31:0] WriteData;
    logic [31:0] DataAdr;
      // --> Instanciacao de um processador RISC-V Single-Cycle
      //main_module #( .DATA_WIDTH(32), .ADDR_W_ROM(14), .ADDR_W_RAM(12), .HEX_FILE("Script_teste_01.tx
    main_module #( .DATA_WIDTH(32), .ADDR_W_ROM(14), .ADDR_W_RAM(12), .HEX_FILE("riscvtest_03B_script3B
               ( .clk( MAX10_CLK1_50 ),
                 .reset( RST ),
                 .mem_write( MemWrite ),
                 .write_data( WriteData ),
                 .data_addr( DataAdr ));
endmodule
```