Módulos do projeto 5 do Repositório no GitHub

1 - Arquivos do projeto e módulos definidos dentro de cada um:

- Arquivo **modulos_basicos.sv**.
  - Módulo **ff_rst**.
  - Módulo **ff_rst_en**.
  - Módulo **mux2**.
  - Módulo **mux3**.
  - Módulo **mux4**.
  - Módulo **mux8**.
  - Módulo **dig_displ_7_segs**.
- Arquivo **modulos_oper_aritmeticas.sv**.
  - Módulo **logical_oper_alu**.
  - Módulo **shift_right_arithmetic**.
  - Módulo **logical_shift_opers**.
  - Módulo **barrel_shifter**.
  - Módulo **adder**.
  - Módulo **full_adder**.
  - Módulo **mult_signed_unsigned**.
- Arquivo **decodificador_instrucoes.sv**.
  - Módulo **instr_fields**.
  - Módulo **extend**.
  - Módulo **main_dec**.
  - Módulo **alu_dec**.
- Arquivo **modulos_auxiliares_cpu.sv**.
  - Módulo **branch_logic_takenbr**.
  - Módulo **set_less_than**.

  - Módulo `next_pc_logic`.

- Arquivo `imem_rf_dmem.v`.
  - Módulo `reg_file`.
  - Módulo `instr_mem`.
  - Módulo `data_mem_single`.

- Arquivo `alu.sv`.
  - Módulo `result_alu_upperimm_jumps`.
  - Módulo `output_flags_alu`.
  - Módulo `alu`.

- Arquivo `top.sv`.
  - Módulo `top`.
  - Módulo `principal`.
  - Módulo `riscv_single`.
  - Módulo `controller`.
  - Módulo `datapath`.

- Arquivo `Div1Unsigned.v`.
  - Módulo `Div1Unsigned`.

- Arquivo `Div2Signed.v`.
  - Módulo `Div2Signed`.

- Arquivo `Mult1Unsigned.v`.
  - Módulo `Mult1Unsigned`.

- Arquivo `Mult2Signed.v`.
  - Módulo `Mult2Signed`.

---

2 - Módulos do arquivo `modulos_basicos.sv`

- Módulos básicos do projeto e que podem ser usados em outros projetos, sem qualquer necessidade de modificações.

2.1 - `ff_rst`:

- **DESCRIÇÃO**:
  - ***Flip-Flop*** com ***RESET***.
  - Se pressionado `rst`, o valor na saída `q` será zerado.

  - A cada borda de subina no *clock*, escreve o valor da entrada `d` na saída `q`.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `clk`: Sinais de *clock*.
  - `reset`: Sinal de *reset*.
  - `d`: Entrada de dados.

- **SAÍDAS (*outputs*)**:
  - `q`: Saída de dados.

```
module ff_rst #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
               ( input  logic              clk,
                 input  logic              reset,
                 input  logic [END_IDX:0]  d,
                 output logic [END_IDX:0]  q );
//-------------------------------------------------------------------------------------------
      always_ff @( posedge clk, posedge reset ) begin
            if( reset ) begin q <= 0; end
            else begin q <= d; end
      end
endmodule
```

2.2 - `ff_rst_en`:

- **DESCRIÇÃO**:
  - ***Flip-Flop*** com ***RESET*** e ***ENABLE***.
  - Escreve na saída `q`, o conteúdo da entrada `d`, apenas se `en` estiver ativado.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `clk`: Sinais de *clock*.
  - `reset`: Sinal de *reset*.
  - `en`: Sinal de *enable*.
  - `d`: Entrada de dados.

- **SAÍDAS (*outputs*)**:
  - `q`: Saída de dados.

```
module ff_rst_en #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
                  ( input  logic                   clk,
                    input  logic                   reset,
                    input  logic                   en,
                    input  logic [END_IDX:0] d,
                    output logic [END_IDX:0] q);
//-------------------------------------------------------------
      always_ff @( posedge clk, posedge reset ) begin
            if( reset ) begin
            q <= 0;
        end
            else if( en == 1'b1 ) begin
            q <= d;
        end
      end
endmodule
```

2.3 - mux2:

- **DESCRIÇÃO**:
  - Multiplexador **2:1**.
  - Entrada **seletora** de **1 bit**.
  - **2 saídas selecionáveis**.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `d0`: Saída selecionável 0 do MUX.
  - `d1`: Saída selecionável 1 do MUX.
  - `sel`: Sinal (de 1 bit) que seleciona a saída desejada.
- **SAÍDAS (*outputs*)**:
  - `y`: Saída selecionada através da **entrada `sel`**.

```
module mux2 #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
             ( input  logic [END_IDX:0] d0,
               input  logic [END_IDX:0] d1,
               input  logic             sel,
               output logic [END_IDX:0] y );
//-------------------------------------------------------------------------------------
      assign y = sel ? d1 : d0;
endmodule
```

2.4 - `mux3`:

- **DESCRIÇÃO**:
  - Multiplexador **3:1**.
  - Entrada **seletora** de **2 bits**.
  - **3 saídas selecionáveis**.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `d0`: Saída selecionável 0 do MUX.
  - `d1`: Saída selecionável 1 do MUX.
  - `d2`: Saída selecionável 2 do MUX.
  - `sel`: Sinal (de **2 bits**) que seleciona a saída desejada.
- **SAÍDAS (*outputs*)**:
  - `y`: Saída selecionada através da **entrada `sel`**.

```
module mux3 #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
             ( input  logic [ END_IDX:0] d0,
               input  logic [ END_IDX:0] d1,
               input  logic [ END_IDX:0] d2,
               input  logic [       1:0] sel,
               output logic [END_IDX:0] y );
//------------------------------------------------------------------------------------------
    assign y = sel[1] ? d2 : ( sel[0] ? d1 : d0 );
endmodule
```

2.5 - `mux4`:

- **DESCRIÇÃO**:
  - .
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `d0`: Saída selecionável 0 do MUX.
  - `d1`: Saída selecionável 1 do MUX.
  - `d2`: Saída selecionável 2 do MUX.

  - `d3`: Saída selecionável 3 do MUX.

  - `sel`: Sinal (de **2 bits**) que seleciona a saída desejada.

- **SAÍDAS (*outputs*)**:
  - `y`: Saída selecionada através da **entrada** `sel`.

```
module mux4 #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
             ( input  logic [END_IDX:0] d0,
               input  logic [END_IDX:0] d1,
               input  logic [END_IDX:0] d2,
               input  logic [END_IDX:0] d3,
               input  logic [      1:0] sel,
               output logic [END_IDX:0] y );
//------------------------------------------------------------------------
    assign y = sel[1] ? (sel[0] ? d3 : d2) : (sel[0] ? d1 : d0);
endmodule
```

2.6 - `mux8`:

- **DESCRIÇÃO**:
  - .

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.

  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `d0`: Saída selecionável 0 do MUX.

  - `d1`: Saída selecionável 1 do MUX.

  - `d2`: Saída selecionável 2 do MUX.

  - `d3`: Saída selecionável 3 do MUX.

  - `d4`: Saída selecionável 4 do MUX.

  - `d5`: Saída selecionável 5 do MUX.

  - `d6`: Saída selecionável 6 do MUX.

  - `d7`: Saída selecionável 7 do MUX.

  - `sel`: Sinal (de **3 bits**) que seleciona a saída desejada.

- **SAÍDAS (*outputs*)**:
  - `y`: Saída selecionada através da **entrada** `sel`.

```
module mux8 #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
             ( input  logic [END_IDX:0] d0,
               input  logic [END_IDX:0] d1,
```

```
                input  logic [END_IDX:0] d2,
                input  logic [END_IDX:0] d3,
                input  logic [END_IDX:0] d4,
                input  logic [END_IDX:0] d5,
                input  logic [END_IDX:0] d6,
                input  logic [END_IDX:0] d7,
                input  logic [      2:0] sel,
                output logic [END_IDX:0] y   );
//-------------------------------------------------------------------------
    always_comb begin
        case( sel )
            3'b000:  y = d0;
            3'b001:  y = d1;
            3'b010:  y = d2;
            3'b011:  y = d3;
            3'b100:  y = d4;
            3'b101:  y = d5;
            3'b110:  y = d6;
            3'b111:  y = d7;
            default: y = { DATA_WIDTH { 1'b0 } };
        endcase
    end
endmodule
```

2.7 - dig_displ_7_segs:

- **DESCRIÇÃO**:
  - Escreve no display de 7 segmentos o dígito referente ao valor rcebido no sinal de entrada.

- **PARÂMETROS (constantes)**:
  - **NÃO POSSUI**.

- **ENTRADAS (*inputs*)**:
  - **digit**: Valor binário de 4 bits com o valor do dígito a ser escrito no display de sete segmentos.

- **SAÍDAS (*outputs*)**:
  - **segs_dsp**: Sinal de 8 bits com os estados dos LEDs do segmentos do display de 7 segmentosd.

```
module dig_displ_7_segs( input logic [3:0] digit, output logic [7:0] segs_dsp );
//-------------------------------------------------------------------------------------------
    // Escrevr no display de 7 segmentos o valor indicado em 'digit'
    always_comb begin
        case( digit )
            4'h0 : segs_dsp = 8'b11000000;
            4'h1 : segs_dsp = 8'b11111001;
            4'h2 : segs_dsp = 8'b10100100;
            4'h3 : segs_dsp = 8'b10110000;
            4'h4 : segs_dsp = 8'b10011001;
            4'h5 : segs_dsp = 8'b10010010;
            4'h6 : segs_dsp = 8'b10000010;
            4'h7 : segs_dsp = 8'b11111000;
            4'h8 : segs_dsp = 8'b10000000;
```

```
            4'h9 : segs_dsp = 8'b10010000;
            4'ha : segs_dsp = 8'b10001000;
            4'hb : segs_dsp = 8'b10000011;
            4'hc : segs_dsp = 8'b11000110;
            4'hd : segs_dsp = 8'b10100001;
            4'he : segs_dsp = 8'b10000110;
            4'hf : segs_dsp = 8'b10001110;
        endcase
    end
endmodule
```

---

3 - Módulos do arquivo `modulos_oper_aritmeticas.sv`

- Módulos básicos do projeto e que podem ser usados em outros projetos, sem qualquer necessidade de modificações.

3.1 - `logical_oper_alu`:

- **DESCRIÇÃO**:
  - .
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `src1_value`: Bits com o valor do **primeiro operando**.
  - `src2_value`: Bits com o valor do **segundo operando**.
- **SAÍDAS (*outputs*)**:
  - `result_and`: Resultado da operação lógica **AND**.
  - `result_or`: Resultado da operação lógica **OR**.
  - `result_xor`: Resultado da operação lógica **XOR**.

```
module logical_oper_alu #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                         ( input  logic [END_IDX:0] src1_value,
                           input  logic [END_IDX:0] src2_value,
                           output logic [END_IDX:0] result_and,
                           output logic [END_IDX:0] result_or,
                           output logic [END_IDX:0] result_xor );
//------------------------------------------------------------------------------------------
    assign result_and = src1_value & src2_value;
    assign result_or = src1_value | src2_value;
    assign result_xor = src1_value ^ src2_value;
endmodule
```

3.2 - `shift_right_arithmetic`:

- **DESCRIÇÃO**:
  - Operaçãod e **deslocamento aritmético** para a **direita**.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.

  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `src1_value`: Bits com o valor do primeiro operando, que é o **valor a ser deslocado**.

  - `src2_value`: Bits com o valor do segundo operando, que é a **distância do deslocamento** a ser realizado.

- **SAÍDAS (*outputs*)**:
  - `sra_rslt`: Resultado da operação de deslocamento aritmético para **DIREITA**.

```
module shift_right_arithmetic #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
                                ( input  logic [END_IDX:0] src1_value,  // Primeiro operando. Valor no r
                                  input  logic [END_IDX:0] src2_value,  // Segundo operando. Valor no re
                                  output logic [END_IDX:0] sra_rslt  ); // Resultado da operacao
//--------------------------------------------------------------------------------------------
    // Constantes
    parameter END_IDX2 = (2*DATA_WIDTH)-1;
    // Variaveis
    logic [END_IDX2:0] sext_src1, sra_rslt1;
    logic [4:0] uimm;
    assign uimm = src2_value[4:0];
    // Realizar a operacao de deslocamento aritmetico
    assign sext_src1 = { { DATA_WIDTH { src1_value[END_IDX] } }, src1_value };
    assign sra_rslt1 = sext_src1 >> uimm;
    assign sra_rslt = sra_rslt1[END_IDX:0];
endmodule
```

3.3 - `logical_shift_opers`:

- **DESCRIÇÃO**:
  - Realiza as **operações de deslocamento lógico**.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.

  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `src1_value`: Bits com o valor do primeiro operando, que é o **valor a ser deslocado**.

  - `src2_value`: Bits com o valor do segundo operando, que é a **distância do deslocamento** a ser realizado.

- **SAÍDAS (*outputs*)**:

- `left_shift`: Resultado da operação de deslocamento lógico para **ESQUERDA**.

- `right_shift`: Resultado da operação de deslocamento lógico para **DIREITA**.

```
module logical_shift_opers #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
                    ( input  logic [END_IDX:0] src1_value,
                      input  logic [END_IDX:0] src2_value,
                      output logic [END_IDX:0] left_shift,
                      output logic [END_IDX:0] right_shift  );
//-------------------------------------------------------------------------------------------------------------
    logic [4:0] uimm;
    assign uimm = src2_value[4:0];

    assign left_shift = src1_value << uimm;
    assign right_shift = src1_value >> uimm;
endmodule
```

3.4 - barrel_shifter:

- **DESCRIÇÃO**:
  - Deslocamento lógico do tipo *barrel-shift*.

- **PARÂMETROS (constantes)**:
  - `ADDR_WIDTH`: Tamanho, em bits, dos valores dos deslocamentos.

  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `clk`: Sinais de *clock*.

  - `enable`: Quando ativado, encia para `sr_out` o resultado da operação de deslocamento.

  - `shift_left`: Se 1, o deslocamneto será para esquerda; se 0, o deslocamento é para a direita.

  - `src1_value`: Bits com o valor do primeiro operando, que é o **valor a ser deslocado**.

  - `src2_value`: Bits com o valor do segundo operando, que é a **distância do deslocamento** a ser realizado.

- **SAÍDAS (*outputs*)**:
  - `sr_out`: Valor do primeiro operando deslocado.

```
module barrel_shifter #( parameter ADDR_WIDTH=5, parameter DATA_WIDTH=(2**ADDR_WIDTH) )
                      ( input  logic                   clk,
                        input  logic                   enable,
                        input  logic                   shift_left,
                        input  logic [(DATA_WIDTH-1):0] src1_value,
                        input  logic [(DATA_WIDTH-1):0] src2_value,
                        output logic [(DATA_WIDTH-1):0] sr_out );
//-----------------------------------------------------------------------------
    // Constantes
    parameter END_IDX = DATA_WIDTH-1;
    parameter END_IDX2 = (DATA_WIDTH*2)-1;
```

```
    parameter END_ADDR = ADDR_WIDTH - 1;

    // Array para armazenar valores temporarios
    logic [END_IDX2:0] tmp;

    // Numero de deslocamentos a serem realizados
    logic [END_ADDR:0] distance;
    assign distance = src2_value[END_ADDR:0];

    // Bloco 'always_ff':
    always_ff @( posedge clk ) begin
        // Valor temporario
        tmp = { src1_value , src1_value };

        // Se 'enable' estiver ativo
        if( enable ) begin
            // Realizar o deslocamento
            tmp = tmp << distance;
            sr_out <= tmp[END_IDX2:DATA_WIDTH];
        end

        else begin
            tmp = tmp >> distance;
            sr_out <= tmp[END_IDX:0];
        end
    end
endmodule
```

3.5 - adder:

- **DESCRIÇÃO**:
  - Somador simples.
  - Apenas retorna o resultado de `DATA_WIDTH` bits com a soma.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `src1_value`: Bits com o valor do **primeiro operando**.
  - `src2_value`: Bits com o valor do **segundo operando**.
- **SAÍDAS (*outputs*)**:
  - `sum_result`: Resultado da soma.

```
module adder #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
              ( input  logic [END_IDX:0] src1_value,
                input  logic [END_IDX:0] src2_value,
                output logic [END_IDX:0] sum_result );
```

```
//------------------------------------------------------------------------------------------
      assign sum_result = src1_value + src2_value;
endmodule
```

3.6 - `full_adder`:

- **DESCRIÇÃO**:
  - Somador completo.
  - Recebe nas entradas os valores a serem somados, mais o bit de *carry-in*.
  - Retorna o resultado de `DATA_WIDTH` bits com a soma, mais o bit de *carry-out*.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `src1_value`: Bits com o valor do **primeiro operando**.
  - `src2_value`: Bits com o valor do **segundo operando**.
  - `cin`: Bit com o *carry-in*.
- **SAÍDAS (*outputs*)**:
  - `sum_result`: Resultado da soma.
  - `cout`: Bit com o *carry-out*.

```
module full_adder #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                   ( input  logic signed [END_IDX:0] src1_value,
                     input  logic signed [END_IDX:0] src2_value,
                     input  logic                    cin,
                     output logic                    cout,
                     output logic signed [END_IDX:0] sum_result );
//------------------------------------------------------------------------------------------
    // --> Declaracao dos sinais e arrays de sinais
    logic [END_IDX:0] cond_inv_b;

    // --> Instancia de 'mux2' para inverter ou nao os bits de 'src2_value'
    mux2 #( .DATA_WIDTH(DATA_WIDTH) ) mux_src2_val
        ( .d0( src2_value ), .d1( ~src2_value ), .sel(is_sub), .y(cond_inv_b) );

    // --> Resultado da operacao de soma/subtracao
    assign {cout, sum_result} = src1_value + src2_value + cin;
endmodule
```

3.7 - `mult_signed_unsigned`:

- **DESCRIÇÃO**:

  - Calcula a parte superior da multiplicação de um valor sinalizado por um não-sinalizado.
  - Resultado da instrução **mulhsu**.
  - **(* multstyle = "dsp" *)**: Manda o Quartus criar o módulo usando os multiplicadores do CI FPGA.

- **PARÂMETROS (constantes)**:
  - **DATA_WIDTH**: Comprimento das *words*, em bits.
  - **END_IDX**: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - **src1_value**: Bits com o valor do **primeiro operando** (SINALIZADO).
  - **src2_value**: Bits com o valor do **segundo operando** (NÃO-SINALIZADO).

- **SAÍDAS (*outputs*)**:
  - **val_mulhsu**: Bits referentes a parte superior do resultado da multiplicação (bits $DATA\_WIDTH$ a $2 \times END\_IDX$).

```
(* multstyle = "dsp" *) module mult_signed_unsigned #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA
                                    ( input  logic [END_IDX:0] src1_value,
                                                  input  logic [END_IDX:0] src2_value,
                                                  output logic [END_IDX:0] val_mulhsu );
//-------------------------------------------------------------------------------------------------------
    // Constantes
    parameter END_IDX2 = (2 * DATA_WIDTH) - 1;

    // Corpo do modulo
    logic [END_IDX2:0] mul1;
    assign mul1 = $signed(src1_value) * $unsigned(src2_value);
    assign val_mulhsu = mul1[END_IDX2:DATA_WIDTH];
endmodule
```

3.8 - multiply:

- **DESCRIÇÃO**:
  - Realiza a operação de multiplicação.
  - Resultados das instruções **mul** e **mulh**.
  - **(* multstyle = "dsp" *)**: Manda o Quartus criar o módulo usando os multiplicadores do CI FPGA.

- **PARÂMETROS (constantes)**:
  - **DATA_WIDTH**: Comprimento das *words*, em bits.
  - **END_IDX**: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - **src1_value**: Bits com o valor do **primeiro operando**.

- `src2_value`: Bits com o valor do **segundo operando**.

- **SAÍDAS (*outputs*)**:
  - `prod_result`: Bits referentes a parte inferior do resultado da multiplicação (bits 0 a END_IDX).
  - `prod_high`: Bits referentes a parte superior do resultado da multiplicação (bits $DATA\_WIDTH$ a $2 \times END\_IDX$).

```
(* multstyle = "dsp" *) module multiply #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                                        ( input  logic signed [END_IDX:0] src1_value,
                                                 input  logic        [END_IDX:0] src2_value,
                                                 output logic        [END_IDX:0] prod_result,
                                                 output logic        [END_IDX:0] prod_high );
//-------------------------------------------------------------------------------------
    logic [END_IDX:0] prod_result;
    assign { prod_high, prod_result } = src1_value * src2_value;
endmodule
```

3.9 - `divide_remainder_unsign`:

- **DESCRIÇÃO**:
  - Operação de divisão com os valores dos dois operandos sendo **NÃO-SINALIZADOS**.
  - Também retornar o resto da divisão.
  - Contém os resultados das instruções `divu` e `remu`.
  - As operações realizadas nesse módulo são exclusivas para números inteiros.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `src1_value`: Bits com o valor do **primeiro operando**.
  - `src2_value`: Bits com o valor do **segundo operando**.

- **SAÍDAS (*outputs*)**:
  - `quotient`: Parte inteira do resultado da divisão (quociente).
  - `remainder`: Parte fracionária do resultado da divisão (resto).

```
module divide_remainder_unsign #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                                ( input  logic signed [END_IDX:0] src1_value,
                                         input  logic signed [END_IDX:0] src2_value,
                                         output logic signed [END_IDX:0] quotient,
                                         output logic signed [END_IDX:0] remainder );
//-------------------------------------------------------------------------------------
    assign quotient = $unsigned(src1_value) / $unsigned(src2_value);
    assign remainder = $unsigned(src1_value) % $unsigned(src2_value);
endmodule
```

3.10 - `divide_remainder_sign`:

- **DESCRIÇÃO**:
  - Operação de divisão com os valores dos dois operandos sendo **SINALIZADOS**.
  - Também retornar o resto da divisão.
  - Contém os resultados das instruções **`div`** e **`rem`**.
  - As operações realizadas nesse módulo são exclusivas para números inteiros.
- **PARÂMETROS (constantes)**:
  - **`DATA_WIDTH`**: Comprimento das *words*, em bits.
  - **`END_IDX`**: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - **`src1_value`**: Bits com o valor do **primeiro operando**.
  - **`src2_value`**: Bits com o valor do **segundo operando**.
- **SAÍDAS (*outputs*)**:
  - **`quotient`**: Parte inteira do resultado da divisão (quociente).
  - **`remainder`**: Parte fracionária do resultado da divisão (resto).

```
module divide_remainder_sign #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                               ( input  logic signed [END_IDX:0] src1_value,
                                      input  logic signed [END_IDX:0] src2_value,
                                      output logic signed [END_IDX:0] quotient,
                                      output logic signed [END_IDX:0] remainder );
//-------------------------------------------------------------------------------------------
    assign quotient = $signed(src1_value) / $signed(src2_value);
    assign remainder = $signed(src1_value) % $signed(src2_value);
endmodule
```

---

4 - Módulos nos arquivos com os MULTIPLICADORES e DIVISORES:

- Os módulos criados aqui estão em arquivos seprados.
- Cada arquivo se refere a um módulo com um módulo para operações de multiplicação ou de divisão.
- Os módulos desses arquivos foram criados com base no código gerado a partir das IPs do Quartus para multiplicação e divisão.
- Esses arquivos podem ser usados em outros projetos, sem grandes modificações.

4.1 - `Mult1Unsigned`:

- **DESCRIÇÃO**:
  - Multiplicador, com os valores dos operandos sendo NÃO-SINALIZADOS.
  - Arquivo `Mult1Unsigned.v`.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `dataa`: Bits com o valor do **primeiro operando**.
  - `datab`: Bits com o valor do **segundo operando**.
- **SAÍDAS (*outputs*)**:
  - `prod`: Bits referentes a parte inferior do resultado da multiplicação (bits 0 a END_IDX).
  - `upper_prod`: Bits referentes a parte superior do resultado da multiplicação (bits $DATA\_WIDTH$ a $2 \times END\_IDX$).

```
// synopsys translate_off
``timescale 1 ps / 1 ps

// synopsys translate_on
module Mult1Unsigned #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                ( dataa, datab, prod, upper_prod );
//-----------------------------------------------------------------------------------------
    input    [END_IDX:0]  dataa;
    input    [END_IDX:0]  datab;
    output [END_IDX:0]  prod;
    output [END_IDX:0]  upper_prod;
//-----------------------------------------------------------------------------------------
    // Constantes
    parameter WIDTH_2X = DATA_WIDTH*2;
    parameter END_IDX2 = WIDTH_2X-1;

    // Arrays com os sinais
    wire [END_IDX2:0] sub_wire0;
    wire [END_IDX2:0] result;

    // Atribuicao ao valor em 'result'
    assign result = sub_wire0[END_IDX2:0];

    // Atribuicoes aos sinais 'prod' e 'upper_prod'
    assign prod = result[END_IDX:0];
    assign upper_prod = result[END_IDX2:DATA_WIDTH];

    // Modulo com o hardware do multiplicador
    lpm_mult    lpm_mult_component ( .dataa( dataa ),
                            .datab( datab ),
                            .result( sub_wire0 ),
```

```
                            .aclr( 1'b0 ),
                            .clken( 1'b1 ),
                            .clock( 1'b0 ),
                            .sclr( 1'b0 ),
                            .sum( 1'b0 )  );
    defparam
        lpm_mult_component.lpm_hint = "DEDICATED_MULTIPLIER_CIRCUITRY=YES,MAXIMIZE_SPEED=5",
        lpm_mult_component.lpm_representation = "UNSIGNED",
        lpm_mult_component.lpm_type = "LPM_MULT",
        lpm_mult_component.lpm_widtha = DATA_WIDTH,
        lpm_mult_component.lpm_widthb = DATA_WIDTH,
        lpm_mult_component.lpm_widthp = WIDTH_2X;
endmodule
```

4.2 - `Mult2Signed`:

- **DESCRIÇÃO**:
  - Multiplicador, com os valores dos operandos sendo SINALIZADOS.
  - Arquivo `Mult2Signed.v`.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `dataa`: Bits com o valor do **primeiro operando**.
  - `datab`: Bits com o valor do **segundo operando**.
- **SAÍDAS (*outputs*)**:
  - `prod`: Bits referentes a parte inferior do resultado da multiplicação (bits 0 a END_IDX).
  - `upper_prod`: Bits referentes a parte superior do resultado da multiplicação (bits $DATA\_WIDTH$ a $2 \times END\_IDX$).

```
// synopsys translate_off
``timescale 1 ps / 1 ps

// synopsys translate_on
module Mult2Signed   #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                ( dataa, datab, prod, upper_prod );
//------------------------------------------------------------------------------------------
    input    [END_IDX:0]  dataa;
    input    [END_IDX:0]  datab;
    output [END_IDX:0]  prod;
    output [END_IDX:0]  upper_prod;
//------------------------------------------------------------------------------------------
    // Constantes
    parameter WIDTH_2X = DATA_WIDTH*2;
    parameter END_IDX2 = WIDTH_2X-1;
```

```
    // Arrays com os sinais
    wire [END_IDX2:0] sub_wire0;
    wire [END_IDX2:0] result;

    // Atribuicao ao valor em 'result'
    assign result = sub_wire0[END_IDX2:0];

    // Atribuicoes aos sinais 'prod' e 'upper_prod'
    assign prod = result[END_IDX:0];
    assign upper_prod = result[END_IDX2:DATA_WIDTH];

    // Modulo com o hardware do multiplicador
    lpm_mult    lpm_mult_component ( .dataa( dataa ),
                            .datab( datab ),
                            .result( sub_wire0 ),
                            .aclr( 1'b0 ),
                            .clken( 1'b1 ),
                            .clock( 1'b0 ),
                            .sclr( 1'b0 ),
                            .sum( 1'b0 )  );
    defparam
        lpm_mult_component.lpm_hint = "DEDICATED_MULTIPLIER_CIRCUITRY=YES,MAXIMIZE_SPEED=5",
        lpm_mult_component.lpm_representation = "SIGNED",
        lpm_mult_component.lpm_type = "LPM_MULT",
        lpm_mult_component.lpm_widtha = DATA_WIDTH,
        lpm_mult_component.lpm_widthb = DATA_WIDTH,
        lpm_mult_component.lpm_widthp = WIDTH_2X;
endmodule
```

4.3 - `Div1Unsigned`:

- **DESCRIÇÃO**:
  - Operação de divisão e resto usando operandos com valores NÃO-SINALIZADOS.
  - Arquivo `Div1Unsigned.v`.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `numer`: Bits com o valor do **primeiro operando**, que será o **numerador** da fração.
  - `denom`: Bits com o valor do **segundo operando**, que será o **denominador** da fração.
- **SAÍDAS (*outputs*)**:
  - `quotient`: Parte inteira do resultado da divisão (quociente).
  - `remain`: Parte fracionária do resultado da divisão (resto).

```
// synopsys translate_off
``timescale 1 ps / 1 ps

// synopsys translate_on
module Div1Unsigned #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                  ( denom, numer, quotient, remain);
//-----------------------------------------------------------------------------------------------------
    input    [END_IDX:0]  denom;
    input    [END_IDX:0]  numer;
    output [END_IDX:0]  quotient;
    output [END_IDX:0]  remain;
//-----------------------------------------------------------------------------------------------------
    wire [END_IDX:0] sub_wire0;
    wire [END_IDX:0] sub_wire1;
    wire [END_IDX:0] quotient = sub_wire0[END_IDX:0];
    wire [END_IDX:0] remain = sub_wire1[END_IDX:0];

    lpm_divide  LPM_DIVIDE_component (
                .denom( denom ),
                .numer( numer ),
                .quotient( sub_wire0 ),
                .remain( sub_wire1 ),
                .aclr( 1'b0 ),
                .clken( 1'b1 ),
                .clock( 1'b0 )  );
    defparam
        LPM_DIVIDE_component.lpm_drepresentation = "UNSIGNED",
        LPM_DIVIDE_component.lpm_hint = "LPM_REMAINDERPOSITIVE=TRUE",
        LPM_DIVIDE_component.lpm_nrepresentation = "UNSIGNED",
        LPM_DIVIDE_component.lpm_type = "LPM_DIVIDE",
        LPM_DIVIDE_component.lpm_widthd = DATA_WIDTH,
        LPM_DIVIDE_component.lpm_widthn = DATA_WIDTH;
endmodule
```

4.4 - Div2Signed:

- **DESCRIÇÃO**:
  - Operação de divisão e resto usando operandos com valores SINALIZADOS.
  - Arquivo `Div2Signed.v`.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `numer`: Bits com o valor do **primeiro operando**, que será o **numerador** da fração.
  - `denom`: Bits com o valor do **segundo operando**, que será o **denominador** da fração.
- **SAÍDAS (*outputs*)**:

- **quotient**: Parte inteira do resultado da divisão (quociente).
- **remain**: Parte fracionária do resultado da divisão (resto).

```
// synopsys translate_off
``timescale 1 ps / 1 ps

// synopsys translate_on
module Div2Signed #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                ( denom, numer, quotient, remain);
//------------------------------------------------------------------------------------------------
    input    [END_IDX:0]  denom;
    input    [END_IDX:0]  numer;
    output [END_IDX:0]  quotient;
    output [END_IDX:0]  remain;
//------------------------------------------------------------------------------------------------
    wire [END_IDX:0] sub_wire0;
    wire [END_IDX:0] sub_wire1;
    wire [END_IDX:0] quotient = sub_wire0[END_IDX:0];
    wire [END_IDX:0] remain = sub_wire1[END_IDX:0];

    lpm_divide  LPM_DIVIDE_component (
                .denom( denom ),
                .numer( numer ),
                .quotient( sub_wire0 ),
                .remain( sub_wire1 ),
                .aclr( 1'b0 ),
                .clken( 1'b1 ),
                .clock( 1'b0 )  );
    defparam
        LPM_DIVIDE_component.lpm_drepresentation = "SIGNED",
        LPM_DIVIDE_component.lpm_hint = "LPM_REMAINDERPOSITIVE=FALSE",
        LPM_DIVIDE_component.lpm_nrepresentation = "SIGNED",
        LPM_DIVIDE_component.lpm_type = "LPM_DIVIDE",
        LPM_DIVIDE_component.lpm_widthd = DATA_WIDTH,
        LPM_DIVIDE_component.lpm_widthn = DATA_WIDTH;
endmodule
```

---

5 - Módulos do arquivo `decodificador_instrucoes.sv`

- Módulos para decodificar os campos das instruções do RISC-V.
- Podem ser aproveitados em outros projetos que implementam CPUs RISC-V.

5.1 - `instr_fields`:

- **DESCRIÇÃO**:
  - Decodifica os campos da instrução RISC-V.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.

  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `instr`: Instrução RISC-V de 32 bits.

- **SAÍDAS (*outputs*)**:
  - `opcode`.
  - `rd`.
  - `funct3`.
  - `rs1`.
  - `rs2`.
  - `funct7b0`.
  - `funct7b5`.

```
module instr_fields( input  logic [31:0] instr,
                     output logic [ 6:0] opcode,
                     output logic [ 4:0] rd,
                     output logic [ 2:0] funct3,
                     output logic [ 4:0] rs1,
                     output logic [ 4:0] rs2,
                     output logic        funct7b0,
                     output logic        funct7b5   );
//-------------------------------------------------------------------------------------------
    // Constantes:
    parameter NULL_REG_ADDR = 5'bxxxxx;
    parameter NULL_FUNCT3 = 3'bxxx;

    // Atribuicao ao conteudo do 'opcode'
    assign opcode = instr[6:0];

    // Array com o conteudo dos arrays referentes aos outputs
    logic [19:0] arr_outputs;
    assign { funct7b5, funct7b0, rs2, rs1, funct3, rd } = arr_outputs;

    // Valores dos elementos do tipo 'output', de acordo com o valor de 'opcode'
    always_comb begin
            case( opcode )
                  7'b0000011: arr_outputs = { 1'bx, 1'bx, NULL_REG_ADDR, instr[19:15], instr[14:12], ins
                  7'b0010011: begin
                if( instr[14:12] == 3'b001 ) begin
                    arr_outputs = { 1'b0, 1'b0, NULL_REG_ADDR, instr[19:15], instr[14:12], instr[11:7] ]
                end
                else if( instr[14:12] == 3'b101 ) begin
                    arr_outputs = { instr[30], 1'b0, NULL_REG_ADDR, instr[19:15], instr[14:12], instr[1
                end
                else begin
                    arr_outputs = { 1'bx, 1'bx, NULL_REG_ADDR, instr[19:15], instr[14:12], instr[11:7] ]
                end
```

```
            end
            7'b0010111: arr_outputs = { 1'bx, 1'bx, NULL_REG_ADDR, NULL_REG_ADDR, NULL_FUNCT3, instr[11
            7'b0100011: arr_outputs = { 1'bx, 1'bx,  instr[24:20], instr[19:15], instr[14:12], NULL_REG
            7'b0110011: arr_outputs = { instr[30], instr[25],  instr[24:20], instr[19:15], instr[14:12]
            7'b0110111: arr_outputs = { 1'bx, 1'bx, NULL_REG_ADDR, NULL_REG_ADDR, NULL_FUNCT3, instr[11
            7'b1100011: arr_outputs = { 1'bx, 1'bx, instr[24:20], instr[19:15], instr[14:12], NULL_REG_
            7'b1100111: arr_outputs = { 1'bx, 1'bx, NULL_REG_ADDR, instr[19:15], instr[14:12], instr[11
            7'b1101111: arr_outputs = { 1'bx, 1'bx, NULL_REG_ADDR, NULL_REG_ADDR, NULL_FUNCT3, instr[11
                  default:    arr_outputs = { 1'bx, 1'bx, NULL_REG_ADDR, NULL_REG_ADDR, NULL_FUNCT3, NU
            endcase
      end
endmodule
```

5.2 - extend:

- **DESCRIÇÃO**:
  - Retorna o Valor Imediato (*immediate*) de 32 bits da instrução.

- **PARÂMETROS (constantes)**:
  - **NÃO POSSUI**.

- **ENTRADAS (*inputs*)**:
  - `instr`.

- **SAÍDAS (*outputs*)**:
  - `imm_src`: .

  - `imm_ext`: .

```
module extend ( input  logic [31:7] instr,
                input  logic [ 2:0] imm_src,
                output logic [31:0] imm_ext );
//------------------------------------------------------------------------------------------
      always_comb begin
            case( imm_src )
                  3'b000:   imm_ext = { { 20 { instr[31] } }, instr[31:20] };
                  3'b001:   imm_ext = { { 20 { instr[31] } }, instr[31:25], instr[11:7] };
                  3'b010:   imm_ext = { { 20 { instr[31] } }, instr[7], instr[30:25], instr[11:8], 1'b0
                  3'b011:   imm_ext = { { 12 { instr[31] } }, instr[19:12], instr[20], instr[30:21], 1'
                  3'b100:   imm_ext = { instr[31:12], { 12 { 1'b0 } } };
            default:  imm_ext = { 32 { 1'bx } }; // undefined
            endcase
      end
endmodule
```

5.3 - main_dec:

- **DESCRIÇÃO**:
  - Decodificador principal.

- **PARÂMETROS (constantes)**:

  - **NÃO POSSUI**.

- **ENTRADAS** (*inputs*):
  - `opcode`.

- **SAÍDAS** (*outputs*):
  - `result_src`.
  - `mem_write`.
  - `branch`.
  - `alu_src`.
  - `reg_write`.
  - `is_auipc`.
  - `is_lui`.
  - `is_jal`.
  - `is_jalr`.
  - `imm_src`.
  - `alu_op`.

```
module main_dec ( input  logic [6:0] opcode,      // OpCode da instrucao (sera os primeiros 7 bits)
                  output logic [1:0] result_src,  //
                  output logic       mem_write,   // Sinal indicando se a instrucao escreve dados na me
                  output logic       branch,      // Operacao de branch
                  output logic       alu_src,     // Sinal indicando se devemos usar um valor gerado na
                  output logic       reg_write,   // Escrever dados no register file
                  output logic       is_auipc,    // Sinal indicando que a instrucao eh 'auipc'
                  output logic       is_lui,    // Sinal indicando que a instrucao eh 'lui'
                  output logic       is_jal,      // Sinal indicando se a instrucao eh 'jal'
                  output logic       is_jalr,     // Sinal indicando se a instrucao eh 'jalr'
                  output logic [2:0] imm_src,     // Tipo de instrucao que recebe immediate
                  output logic [1:0] alu_op );    // Operacao realizada na ALU
//------------------------------------------------------------------------------------------
      // Array de 11 bits com os sinais agrupados
      logic [14:0] controls;
      assign { reg_write, imm_src, alu_src, mem_write, result_src, branch, alu_op, is_lui, is_auipc, is_

      always_comb begin
            case( opcode )
                  // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_isLui_isAuipc_isJal_isJalr
                  7'b0000011: controls = 15'b1_000_1_0_01_0_00_0_0_0_0; // Instrucoes que usam o format
                  7'b0010011: controls = 15'b1_000_1_0_00_0_10_0_0_0_0; // Instrucoes que usam o format
            7'b0010111: controls = 15'b1_100_0_0_11_0_00_0_1_0_0; // Instrucoes que usam o formato U-Ty
            7'b0100011: controls = 15'b0_001_1_1_00_0_00_0_0_0_0; // Instrucoes que usam o formato S-Ty
                  7'b0110011: controls = 15'b1_xxx_0_0_00_0_10_0_0_0_0; // Instrucoes que usam o format
                  7'b0110111: controls = 15'b1_100_0_0_11_0_00_1_0_0_0; // Instrucoes que usam o format
```

```
            7'b1100011: controls = 15'b0_010_0_0_00_1_01_0_0_0_0; // Instrucoes que usam o formato B-Ty
            7'b1100111: controls = 15'b1_000_1_0_00_0_10_0_0_0_1; // Instrucoes que usam o formato I-Ty
            7'b1101111: controls = 15'b1_011_0_0_10_0_00_0_0_1_0; // Instrucoes que usam o formato J-Ty
                  default:    controls = 15'bx_xxx_x_x_xx_x_xx_x_x_x_0; // non-implemented instruction
            endcase
      end
endmodule
```

5.4 - alu_dec:

- **DESCRIÇÃO**:
  - Retorna o código da instrução cujo resultado será retornado pela ALU.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `opcode`: .
  - `funct3`: .
  - `funct7b0`: .
  - `funct7b5`: .

- **SAÍDAS (*outputs*)**:
  - `alu_ctrl`: Código de 5 bits pertinente à instrução a ser executada na ALU.

```
module alu_dec ( input  logic [6:0] opcode,
                 input  logic [2:0] funct3,
                 input  logic       funct7b0,
                 input  logic       funct7b5,
                 output logic [4:0] alu_ctrl );
//------------------------------------------------------------------------------------------
    // Array com o conteudo dos campos das instrucoes
    logic [11:0] dados_instr;
    assign dados_instr = {funct7b5, funct7b0, funct3, opcode};

    // Bloco 'always_comb' para selecionar o codigo da instrucao
      always_comb begin
            case( dados_instr )     // Instrucao que usa a ALU do tipo R-type ou I-type
            // Instrucoes R-Type
            12'b0_0_000_0110011: alu_ctrl = 5'b00000; // add
            12'b1_0_000_0110011: alu_ctrl = 5'b00001; // sub
            12'b0_0_111_0110011: alu_ctrl = 5'b00010; // Instrucao 'and'
            12'b0_0_110_0110011: alu_ctrl = 5'b00011; // Instrucao 'or'
            12'b0_0_100_0110011: alu_ctrl = 5'b00100; // Instrucao 'xor'
            12'b0_0_001_0110011: alu_ctrl = 5'b00101; // Instrucao 'sll'
            12'b0_0_101_0110011: alu_ctrl = 5'b00110; // Instrucao 'srl'
            12'b1_0_101_0110011: alu_ctrl = 5'b00111; // Instrucao 'sra'
```

```
            12'b0_0_010_0110011: alu_ctrl = 5'b01000; // Instrucao 'slt'
            12'b0_0_011_0110011: alu_ctrl = 5'b01001; // Instrucao 'sltu'
            // Instrucoes do tipo I-Type analogas as instrucoes do bloco acima
            12'bx_x_000_0010011: alu_ctrl = 5'b00000; // Instrucao 'addi'
            12'bx_x_111_0010011: alu_ctrl = 5'b00010; // Instrucao 'andi'
            12'bx_x_110_0010011: alu_ctrl = 5'b00011; // Instrucao 'ori'
            12'bx_x_100_0010011: alu_ctrl = 5'b00100; // Instrucao 'xori'
            12'b0_0_001_0010011: alu_ctrl = 5'b00101; // Instrucao 'slli'
            12'b0_0_101_0010011: alu_ctrl = 5'b00110; // Instrucao 'srli'
            12'b1_0_101_0010011: alu_ctrl = 5'b00111; // Instrucao 'srai'
            12'bx_x_010_0010011: alu_ctrl = 5'b01010; // Instrucao 'slti'
            12'bx_x_011_0010011: alu_ctrl = 5'b01011; // Instrucao 'sltui'
            // Instrucoes 'lui' 'auipc', 'jal' e 'jalr'
            12'bx_x_xxx_0110111: alu_ctrl = 5'b01100; // Instrucao 'lui'
            12'bx_x_xxx_0010111: alu_ctrl = 5'b01100; // Instrucao 'auipc'
            12'bx_x_xxx_1101111: alu_ctrl = 5'b01100; // Instrucao 'jal'
            12'bx_x_000_1100111: alu_ctrl = 5'b01100; // Instrucao 'jalr'
            // Instrucoes B-Type (branching)
            12'bx_x_000_1100011: alu_ctrl = 5'b01101; // Instrucao 'beq'
            12'bx_x_001_1100011: alu_ctrl = 5'b01101; // Instrucao 'bne'
            12'bx_x_100_1100011: alu_ctrl = 5'b01101; // Instrucao 'blt'
            12'bx_x_101_1100011: alu_ctrl = 5'b01101; // Instrucao 'bge'
            12'bx_x_110_1100011: alu_ctrl = 5'b01101; // Instrucao 'bltu'
            12'bx_x_111_1100011: alu_ctrl = 5'b01101; // Instrucao 'bgeu'
            // Instrucoes R-Type Multiplicacao/Divisao
            12'b0_1_000_0110011: alu_ctrl = 5'b01110; // Instrucao 'mul'
            12'b0_1_001_0110011: alu_ctrl = 5'b01111; // Instrucao 'mulh'
            12'b0_1_010_0110011: alu_ctrl = 5'b10000; // Instrucao 'mulhsu'
            12'b0_1_011_0110011: alu_ctrl = 5'b10001; // Instrucao 'mulhu'
            12'b0_1_100_0110011: alu_ctrl = 5'b10010; // Instrucao 'div'
            12'b0_1_101_0110011: alu_ctrl = 5'b10011; // Instrucao 'divu'
            12'b0_1_110_0110011: alu_ctrl = 5'b10100; // Instrucao 'rem'
            12'b0_1_111_0110011: alu_ctrl = 5'b10101; // Instrucao 'remu'

            // caso padrao
            default:  alu_ctrl = 5'b11111;
        endcase
    end
endmodule
```

---

6 - Módulos do arquivo `modulos_auxiliares_cpu.sv`

6.1 - `branch_logic_takenbr`:

- **DESCRIÇÃO**:
  - Testa se a condição testada na operação de *branching* é verdadeira ou falsa.
  - Gera o resultado das instruções do tipo *B-Type*.
  - Instruções `beq`, `bne`,`blt`, `bge`, `bltu` e `bgeu`.
- **PARÂMETROS (constantes)**:

- `DATA_WIDTH`: Comprimento das *words*, em bits.

- `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `src1_value`.

  - `src2_value`.

  - `funct3`.

  - `branch`.

- **SAÍDAS (*outputs*)**:
  - `taken_br`: .

```
module branch_logic_takenbr #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                              ( input  logic [END_IDX:0]  src1_value,    // Primeiro operando
                                input  logic [END_IDX:0]  src2_value,    // Segundo operando
                                input  logic [      2:0]  funct3,
                                input  logic              branch,        // Indica se a instrucao eh uma
                                output logic              taken_br );    // Sinal indicando se foi acion
//-------------------------------------------------------------------------------------------------------
    // Sinal indicando que o bit de maior significancia eh diferente
    logic not_msb_vals;
    assign not_msb_vals = (src1_value[31] != src2_value[31]);

    // Sinal 'taken_br'
    always_comb begin
        if ( branch ) begin
            case( funct3 )
                3'b000: taken_br = (src1_value == src2_value);                // Instrucao 'beq' (igua
                3'b001: taken_br = (src1_value != src2_value);                // Instrucao 'bne' (dife
                3'b100: taken_br = (src1_value < src2_value) ^ not_msb_vals;  // Instrucao 'blt' (Meno
                3'b101: taken_br = (src1_value >= src2_value) ^ not_msb_vals; // Instrucao 'bge' (Maio
                3'b110: taken_br = (src1_value < src2_value);                 // Instrucao 'bltu' (Men
                3'b111: taken_br = (src1_value >= src2_value);                // Instrucao 'bgeu' (Mai
                default: taken_br = 1'b0;   // Caso padrao
            endcase
        end
        else begin
            taken_br = 1'b0;
        end
    end
endmodule
```

6.2 - `set_less_than`:

- **DESCRIÇÃO**:
  - Retorna o resultado da instrução *Set Less Than*.

  - Instruções `slt`, `slti`, `sltu` e `sltiu`.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `src1_value`.
  - `src2_value`.
  - `imm_ext`.
- **SAÍDAS (*outputs*)**:
  - `sltu_rslt`.
  - `sltiu_rslt`.
  - `slt_rslt`.
  - `slti_rslt`.

```
module set_less_than #( parameter DATA_WIDTH = 32, parameter END_IDX=(DATA_WIDTH - 1) )
                      ( input  logic [END_IDX:0] src1_value,  // Primeiro operando. Valor no registrado
                        input  logic [END_IDX:0] src2_value,  // Segundo operando. Valor no registrador
                        input  logic [     31:0] imm_ext,
                        output logic [END_IDX:0] sltu_rslt,    // Resultado da operacao 'sltu' (unsigne
                        output logic [END_IDX:0] sltiu_rslt,   // Resultado da operacao 'sltiu' (unsign
                        output logic [END_IDX:0] slt_rslt,     // Resultado da operacao 'slt' (signed)
                        output logic [END_IDX:0] slti_rslt );  // Resultado da operacao 'slti' (signed)
//------------------------------------------------------------------------------------------
    // Constantes
    parameter NULL_ARRAY = { END_IDX { 1'b0 } };

    // Array 'sltu_rslt'
    assign sltu_rslt = { NULL_ARRAY, ( src1_value < src2_value ) };

    // Array 'sltiu_rslt'
    assign sltiu_rslt = { NULL_ARRAY, ( src1_value[31:0] < imm_ext ) };

    // Array 'slt_rslt'
    assign slt_rslt = (src1_value[END_IDX] == src2_value[END_IDX]) ? sltu_rslt : { NULL_ARRAY, src1_valu

    // Array 'slti_rslt'
    assign slti_rslt = (src1_value[END_IDX] == imm_ext[END_IDX]) ? sltiu_rslt : { NULL_ARRAY, src1_valu
endmodule
```

6.3 - `next_pc_logic`:

- **DESCRIÇÃO**:
  - Operação referente à *next PC logic*.
  - Calcula o endereço de memória da próxima instrução a ser executada.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `clk`.
  - `reset`.
  - `taken_br`.
  - `is_jal`.
  - `is_jalr`.
- **SAÍDAS (*outputs*)**:
  - `pc_val`: Endereço de memória da instrução atual.
  - `pc_next`: Endereço de memória da próxima instrução.

```
module next_pc_logic #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                       ( input  logic             clk,
                         input  logic             reset,
                         input  logic             taken_br,
                         input  logic             is_jal,
                         input  logic             is_jalr,
                         output logic [END_IDX:0] pc_val,       // Valor armazenado no registrador 'pc'
                         output logic [END_IDX:0] pc_next );    // Endereco da proxima instrucao a ser e
//-------------------------------------------------------------------------------------------------------
    // --> Constantes
    parameter VALUE_4 = { { (DATA_WIDTH-3) { 1'b0} }, 3'b100 };

    // --> Arrays com os valores e sinais usados dentro do modulo
    logic [END_IDX:0] br_tgt_pc, jalr_tgt_pc, pc_plus_4, pc_target;

    // --> Instancia do modulo 'ff_rst' (flip-flop com reset)
    ff_rst #( .DATA_WIDTH( DATA_WIDTH ) ) pc_register
            ( .clk( clk ),
            .reset( reset ),
            .d( pc_next ),
            .q( pc_val )  );

    // --> Somador para calcular o valor de 'pc_plus_4'
    adder #( .DATA_WIDTH( DATA_WIDTH ) ) pc_add_4
         ( .src1_value( pc_val ), .src2_value( VALUE_4 ), .sum_result( pc_plus_4 )  );

    // --> Computar o sinal 'br_tgt_pc'
    adder #( .DATA_WIDTH( DATA_WIDTH ) ) pc_add_branch
         ( .src1_value( pc_val ), .src2_value( imm_ext ), .sum_result( br_tgt_pc )  );

    // --> Computar o sinal 'jalr_tgt_pc'
    adder #( .DATA_WIDTH( DATA_WIDTH ) ) adder_jalr_tgt_pc
         ( .src1_value( src1_value ), .src2_value( imm_ext ), .sum_result( jalr_tgt_pc ) );
```

```
    // --> Computar o valor de 'pc_next'
    assign pc_next = ( taken_br || is_jal ) ? br_tgt_pc : ( is_jalr ? jalr_tgt_pc : pc_plus_4 );
endmodule
```

---

7 - Módulos do arquivo `alu.sv`

7.1 - `result_alu_upperimm_jumps`:

- **DESCRIÇÃO**:
  - .
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `is_auipc`.
  - `is_lui`.
  - `is_jal`.
  - `is_jalr`.
  - `imm_ext`.
  - `pc_val`.
- **SAÍDAS (*outputs*)**:
  - `pc_val`.
  - `out_value`.

```
module result_alu_upperimm_jumps #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
                                  ( input  logic             is_auipc,    // Sinal indicando que a inst
                                    input  logic             is_lui,      // Sinal indicando que a inst
                                    input  logic             is_jal,      // Sinal indicando se a instr
                                    input  logic             is_jalr,     // Sinal indicando se a instr
                                    input  logic [     31:0] imm_ext,     // Valor imediato
                                    input  logic [END_IDX:0] pc_val,      // Valor armazenado no regist
                                    output logic [END_IDX:0] out_value ); // Valor retornado pelo modul
//-------------------------------------------------------------------------------------------------------
    // Constantes
    parameter NULL_VALUE = { DATA_WIDTH { 1'b0 } };
    parameter VALUE_4 = { { (DATA_WIDTH - 3) { 1'b0 } }, 3'b100 };

    // Arrays que irao receber os resultados
    logic [3:0] arr_is_instr;
    logic [END_IDX:0] auipc_res, lui_res, jal_res, jalr_res;
```

```
    // Juntar os sinais das instrucoes em 'arr_is_instr'
    assign arr_is_instr = { is_auipc, is_lui, is_jal, is_jalr };


    // --> Somadores com os resultados
    adder #( .DATA_WIDTH( DATA_WIDTH ) ) adder_auipc
         ( .src1_value( pc_val ), .src2_value( imm_ext ), .sum_result( auipc_res ) );

    adder #( .DATA_WIDTH( DATA_WIDTH ) ) adder_jal
         ( .src1_value( pc_val ), .src2_value( VALUE_4 ), .sum_result( jal_res ) );

    adder #( .DATA_WIDTH( DATA_WIDTH ) ) adder_jalr
         ( .src1_value( pc_val ), .src2_value( VALUE_4 ), .sum_result( jalr_res ) );

    // --> Valor da operacao 'lui'
    assign lui_res = { imm_ext[31:12], { 12 { 1'b0 } } };

    // Bloco 'always_comb' para gerar o valor de 'out_value'
    always_comb begin
        case( arr_is_instr )
            4'b1000: out_value = auipc_res;
            4'b0100: out_value = lui_res;
            4'b0010: out_value = jal_res;
            4'b0001: out_value = jalr_res;
        endcase
    end
endmodule
```

7.2 - `output_flags_alu`:

- **DESCRIÇÃO**:
  - Produz os resultados dos *output flags* da ALU.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `src1_b31`: Último bit do valor do primeiro operando.
  - `src2_b31`: Último bit do valor do segundo operando.
  - `is_add_sub`: Sinal indicando se a operação é de soma ou subtração.
  - `alu_ctrl_b0`: Bit 0 do array `alu_ctrl`.
  - `cout`: Sinal indicando se foi produzido um *carry out*.
  - `result`: Resultado da operação realizada na ALU.

- **SAÍDAS (*outputs*)**:

- `of_c`: *Output flag* indicando que o resultado produziu um *carry-out*.
- `of_n`: *Output flag* indicando que o resultado é negativo.
- `of_v`: *Output flag* indicando que o resultado produziu um *overflow*.
- `of_z`: *Output flag* indicando que o resultado é zero.

```
module output_flags_alu #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
                         ( input  logic             src1_b31,     // Ultimo bit do 1o operando da ALU
                           input  logic             src2_b31,     // Ultimo bit do 2o operando da ALU
                           input  logic             is_add_sub,   // Sinal indicando se a operacao eh d
                           input  logic             alu_ctrl_b0,  // Bit 0 de 'alu_ctrl', indica se a o
                           input  logic             cout,         // Sinal indicando se foi produzido u
                           input  logic [END_IDX:0] result,       // Array com o resultado da operacao
                           output logic             of_c,         // 'Output Flag' 'c': Indica se a ope
                           output logic             of_n,         // 'Output Flag' 'n': Indica se o res
                           output logic             of_v,         // 'Output Flag' 'v': Indica a ocorre
                           output logic             of_z   );     // 'Output Flag' 'z': Indica se o res
//-------------------------------------------------------------------------------------------------------
    parameter ZERO_VAL = { DATA_WIDTH { 1'b0 } };

    // 'Output flag' 'of_z': Indica se o resultado eh 0
    assign of_z = (result == ZERO_VAL);

    // 'Output flag' 'of_n': Indica se o resultado eh negativo.
    assign of_n = result[END_IDX];

    // 'Output Flag' 'c': Indica se a operacao de soma/subtracao gerou um carry-out
    assign of_c = is_add_sub & cout;

    // 'Output Flag' 'v': Indica a ocorrencia de overflow
    assign of_v = ~(alu_ctrl_b0 ^ (src1_b31 ^ src2_b31)) & (src1_b31 ^ result[END_IDX]) & is_add_sub;
endmodule
```

7.3 - alu:

- **DESCRIÇÃO**:
  - Unidade Lógica e Aritmética da CPU RISC-V criada aqui.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - `src1_value`: Bits com o valor do **primeiro operando**.
  - `src2_value`: Bits com o valor do **segundo operando**.
  - `alu_ctrl`: Código de 5 bits pertinente à instrução a ser executada na ALU.
  - `imm_ext`: .

- **SAÍDAS (*outputs*)**: Valor de saída do módulo `extend`.
  - `out_value`: Valor de saída do módulo `result_alu_upperimm_jumps`.
  - `result`: Resultado da operação realizada na ALU.
  - `of_c`: *Output flag* indicando que o resultado produziu um *carry-out*.
  - `of_n`: *Output flag* indicando que o resultado é negativo.
  - `of_v`: *Output flag* indicando que o resultado produziu um *overflow*.
  - `of_z`: *Output flag* indicando que o resultado é zero.

```
module alu #( parameter DATA_WIDTH=32, parameter END_IDX=(DATA_WIDTH - 1) )
            ( input  logic [END_IDX:0] src1_value,
              input  logic [END_IDX:0] src2_value,
              input  logic [      4:0] alu_ctrl,
              input  logic [     31:0] imm_ext,      // Valor de saida do modulo 'extend'
              output logic [END_IDX:0] out_value,    // Valor de saida do modulo 'result_alu_upperimm_j
              output logic [END_IDX:0] result,       // Array com o resultado da operacao gerado na ALU
              output logic             of_c,         // 'Output Flag' 'c': Indica se a operacao gerou u
              output logic             of_n,         // 'Output Flag' 'n': Indica se o resultado eh neg
              output logic             of_v,         // 'Output Flag' 'v': Indica a ocorrencia de overf
              output logic             of_z );       // 'Output Flag' 'z': Indica se o resultado eh 0
//-------------------------------------------------------------------------------------------------
    // --> Constantes
    parameter NULL_VAL = { DATA_WIDTH { 1'bx } };


    /**********************
    ** Soma/Subtracao    **
    **********************/
    logic [END_IDX:0] sum_oper;           // Resutado da operacao de soma/subtracao
    logic is_add_sub, is_sub, cin, cout;  // Sinais indicando se a operacao eh de soma e bit de carry-o

    full_adder #( .DATA_WIDTH( DATA_WIDTH ) ) adder_alu
            ( .src1_value( src1_value ), .src2_value( src2_value ), .cin( cin ), .cout( cout ), .sum_res


    /*****************************
    ** Operacoes AND, OR e XOR   **
    *****************************/
    logic [END_IDX:0] and_oper, or_oper, xor_oper;    // Arrays retornados pelo modulo 'logical_oper_al

    logical_oper_alu #( .DATA_WIDTH( DATA_WIDTH ) ) log_opers
                ( .src1_value( src1_value ), .src2_value( src2_value ), .result_and( and_oper ), .resul


    /*************************************
    ** Operacoes de Deslocamento Logico    **
    *************************************/
    logic [END_IDX:0] log_shift_left, log_shift_right;  // Arrays retornados pelo modulo 'logical_shift
```

```
    logical_shift_opers #( .DATA_WIDTH( DATA_WIDTH ) ) log_shifts
                  ( .src1_value( src1_value ), .src2_value( src2_value ), .left_shift( log_shift_left )


    /*****************************************
    ** Operacoes de Deslocamento Aritmetico    **
    *****************************************/
    logic [END_IDX:0] arith_shift_right; // Arrays retornados pelo modulo 'shift_right_arithmetic'

    shift_right_arithmetic #( .DATA_WIDTH( DATA_WIDTH ) ) arith_shifts
                  ( .src1_value( src1_value ), .src2_value( src2_value ), .sra_rslt( arith_shift_right


    /*****************************************
    ** Operacoes de 'Set Less Than'            **
    *****************************************/
    logic [END_IDX:0] op_slt, op_sltu, op_slti, op_sltiu; // Arrays retornados pelo modulo 'set_less_th

    set_less_than #( .DATA_WIDTH( DATA_WIDTH ) ) mod_slt
              ( .src1_value( src1_value ), .src2_value( src2_value ), .imm_ext( imm_ext ), .sltu_rslt(


    /*****************************************
    ** Operacoes de MULTIPLICACAO              **
    *****************************************/
    // Arrays retornados pelo smodulos 'Mult1Unsigned', 'Mult2Signed' e 'mult_signed_unsigned'
    logic [END_IDX:0] val_mulhsu, prod_u, prod_s, upper_prod_uu, upper_prod_ss;

    mult_signed_unsigned #( .DATA_WIDTH( DATA_WIDTH ) ) mod_mulhsu
                  ( .src1_value( src1_value ), .src2_value( src2_value ), .val_mulhsu( val_mulhsu ) )

    Mult1Unsigned #( .DATA_WIDTH( DATA_WIDTH ) ) mod_mulhsu
                ( .dataa( src1_value ), .datab( src2_value ), .prod( prod_s ), .upper_prod( upper_prod_

    Mult2Signed #( .DATA_WIDTH( DATA_WIDTH ) ) mod_mulhsu
              ( .dataa( src1_value ), .datab( src2_value ), .prod( prod_u ), .upper_prod( upper_prod_ss


    /*****************************************
    ** Operacoes de DIVISAO e RESTO            **
    *****************************************/
    logic [END_IDX:0] quot_s, quot_u, rem_s, rem_u;

    Div1Unsigned #( .DATA_WIDTH( DATA_WIDTH ) ) mod_mulhsu
                ( .numer( src1_value ), .denom( src2_value ), .quotient( quot_s ), .remain( rem_u ) );

    Div2Signed  #( .DATA_WIDTH( DATA_WIDTH ) ) mod_mulhsu
              ( .numer( src1_value ), .denom( src2_value ), .quotient( quot_u ), .remain( rem_s ) );


    /*****************************************************
    ** Selecionar o output indicado em 'alu_ctrl'   **
    *****************************************************/
```

```
      always_comb begin
            case( alu_ctrl )
                  // Operacoes de Adicao e Subtracao
            5'b00000:  result = sum_oper;          // 'add'/'addi'
                  5'b00001:  result = sum_oper;          // 'sub'
                  // Operacoes logicas
            5'b00010:  result = and_oper;          // 'and'/'andi'
            5'b00011:  result = or_oper;           // 'or'/'ori'
            5'b00100:  result = xor_oper;          // 'xor'/'xori'
            // Deslocamento logico e aritmetico
            5'b00101:  result = log_shift_left;    // 'sll'/'slli'
            5'b00110:  result = log_shift_right;   // 'srl'/'srli'
            5'b00111:  result = arith_shift_right; // 'sra'/'srai'
            // Set Less Than
            5'b01000:  result = op_slt;            // 'slt'/'slti'
            5'b01001:  result = op_sltu;           // 'sltu'
            5'b01010:  result = op_slti;           // 'slti'
            5'b01011:  result = op_sltiu;          // 'sltiu'
            // 'auipc', 'jal', 'jalr' e 'lui'
            5'b01100:  result = out_value;         // 'lui' / 'auipc'/ 'jal' / 'jalr'
            // Multiplicacao
            5'b01110:  result = prod_s;            // 'mul'
            5'b01111:  result = upper_prod_ss;     // 'mulh'
            5'b10000:  result = val_mulhsu;        // 'mulhsu
            5'b10001:  result = upper_prod_uu;     // 'mulhu'
            // Divisao e Resto da Divisao
            5'b10010:  result = quot_s;            // 'div'
            5'b10011:  result = quot_u;            // 'divu'
            5'b10100:  result = rem_s;             // 'rem'
            5'b10101:  result = rem_u;             // 'remu'
            // Caso padrao
            default:   result = NULL_VAL;
            endcase
      end


    /*************************************************
    ** Output Flags                                 **
    *************************************************/
    output_flags_alu #( .DATA_WIDTH(DATA_WIDTH) ) out_flags
                    ( .src1_b31(src1_value[END_IDX]), .src2_b31(src2_value[END_IDX]), .is_add_sub(is_ad
                  .cout(cout), .result(result), .of_c(of_c), .of_n(of_n), .of_v(of_v), .of_z(of_z) );
endmodule
```

---

8 - Módulos do arquivo `imem_rf_dmem.v`

- Módulos referentes as memórias.
  - *Register File* com 32 registradores.

  - Memória RAM para armazenamento de dados.

  - Memória ROM para armazenar as instruções do programa.

- Os módulos desse arquivo podem ser adaptados em outros projetos, sem grandes modificações.

8.1 - `reg_file`:

- **DESCRIÇÃO**:
  - Conjunto de registradores (*register file*) com os 32 registradores de uso geral.

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - `clk`: Entrada com os sinais de *clock*.
  - `wr_en`: Sinal *Write Enable*.
  - `r_addr1`: Endereço do registrador com o primeiro valor.
  - `r_addr2`: Endereço do registrador com o segundoo valor.
  - `w_addr`: Endereço do registrador onde será escrito um valor.
  - `w_data`: Valor a ser escrito no registrador de endereço `w_addr`.

- **SAÍDAS (*outputs*)**:
  - `r_data1`: Valor lido no registrador de endereço `r_addr1`.
  - `r_data2`: Valor lido no registrador de endereço `r_addr2`.

```
module reg_file #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
                  ( input  wire             clk,
                    input  wire             wr_en,
                    input  wire [      4:0] r_addr1,
                    input  wire [      4:0] r_addr2,
                    input  wire [      4:0] w_addr,
                    input  wire [END_IDX:0] w_data,
                    output wire [END_IDX:0] r_data1,
                    output wire [END_IDX:0] r_data2 );
//------------------------------------------------------------------------
    // Constantes:
    parameter VAL_0 = { DATA_WIDTH { 1'b0 } };

    // --> Matriz com os 32 REGISTRADORES que compõe o register file <-- //
    (* ramstyle = "logic" *) reg [END_IDX:0] RF [31:0];
    // Sobre a expressao '(* ramstyle = "logic" *)': Mandar criar o register file usando os elementos l

    /*
    ** --> Register file com 3 ports  <-- //
    **  - Os 2 ports de leitura são lidos usando lógica combinacional (r_addr1/r_data1, r_addr2/r_data2
    **  - As operações de escrita somente ocorrem nas bordas de subida do clock (w_addr/w_data/wr_en)
    **  - Registrador RF[0] é hardwired em 0
```

```
    */
    always @( posedge clk ) begin
        if( wr_en ) begin
            RF[ w_addr ] <= w_data;
        end
    end

    // --> Operações de leitura dos dados armazenados nos registradores
    // --> Caso 'r_addr1' ou 'r_addr', retornar o valor 0
    assign r_data1 = (r_addr1 != 5'd0) ? RF[ r_addr1 ] : VAL_0;
    assign r_data2 = (r_addr2 != 5'd0) ? RF[ r_addr2 ] : VAL_0;
endmodule
```

8.2 - instr_mem:

- **DESCRIÇÃO**:
  - Memória ROM para armazenar o programa (*instruction memory*.
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
  - `ADDR_WIDTH`: Comprimento, em bist, dos endereços de memória da memória ROM.
  - `HEX_FILE`: Arquivo com o programa armazenado.
- **ENTRADAS (*inputs*)**:
  - `addr`: Endereço de memória da instrução lida.
- **SAÍDAS (*outputs*)**:
  - `instr`: Instrução lida no endereço de memória `addr`.

```
module instr_mem #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1, parameter ADDR_WIDTH=10, pa
                  ( input  wire [END_IDX:0] addr,
                    output wire [END_IDX:0] instr );
//-------------------------------------------------------------------------------------------------------
    parameter N_WORDS = (2**ADDR_WIDTH)-1;

    // --> Matriz com a 'instruction memory' <-- //
    // Sobre a expressao '(* ramstyle = "M9K" *)': Mandar implementar a ROM nos blocos de memoria M9K d
    (* ramstyle = "M9K" *) reg [END_IDX:0] ROM [N_WORDS:0];

    // --> Inserir o conteudo do arquivo .hex no array 'ROM' <-- //
      initial begin
            $readmemh(HEX_FILE, ROM);
      end

    // --> Atribuir a 'instr' o valor em 'addr'
    assign instr = ROM[ { 2'b00, addr[END_IDX:2] } ];
endmodule
```

8.3 - `data_mem_single`:

- **DESCRIÇÃO**:
  - .

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
  - `ADDR_WIDTH`: Comprimento, em bist, dos endereços de memória da memória RAM.

- **ENTRADAS (*inputs*)**:
  - `clk`: Entrada com os sinais de *clock*.
  - `wr_en`: Sinal *Write Enable*.
  - `addr`: Endereço de memória onde será lido ou escrito um valor.
  - `w_data`: Valor que será escrito no endereço de memória `addr`.

- **SAÍDAS (*outputs*)**:
  - `r_data`: Valor lido no endereço de memória `addr`.

```
// synopsys translate_off
``timescale 1 ps / 1 ps

// synopsys translate_on
module data_mem_single #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1, parameter ADDR_WIDTH=
                      ( clk, w_en, addr, w_data, r_data );
//-------------------------------------------------------------------------------
    input                clk;
    input                w_en;
    input  [END_IDX:0] addr;
    input    [END_IDX:0] w_data;
    output [END_IDX:0] r_data;

``ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
``endif
    tri1      clk;
    tri0      w_en;
``ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
``endif
//-------------------------------------------------------------------------------
    // --> Constante: Numero de words:
    parameter N_WORDS = (2**ADDR_WIDTH)-1;
    parameter LAST_ADDR = ADDR_WIDTH-1;

    // --> Endereco de leitura/escrita com 'ADDR_WIDTH' bits
    wire [LAST_ADDR:0] address_div4;
```

```
    assign address_div4 = { 2'b00, addr[END_IDX:2] };

    wire [END_IDX:0] sub_wire0;
    wire [END_IDX:0] r_data = sub_wire0[END_IDX:0];
    //assign r_data = sub_wire0[END_IDX:0];

    // --> Instanciacao de uma IP 'altsyncram' com a memoria RAM <-- //
    altsyncram  altsyncram_component (
                .address_a( address_div4 ),
                .clock0( clk ),
                .data_a( w_data ),
                .wren_a( w_en ),
                .q_a( sub_wire0 ),
                .aclr0( 1'b0 ),
                .aclr1( 1'b0 ),
                .address_b( 1'b1 ),
                .addressstall_a( 1'b0 ),
                .addressstall_b( 1'b0 ),
                .byteena_a( 1'b1 ),
                .byteena_b( 1'b1 ),
                .clock1( 1'b1 ),
                .clocken0( 1'b1 ),
                .clocken1( 1'b1 ),
                .clocken2( 1'b1 ),
                .clocken3( 1'b1 ),
                .data_b( 1'b1 ),
                .eccstatus(),
                .q_b(  ),
                .rden_a( 1'b1 ),
                .rden_b( 1'b1 ),
                .wren_b( 1'b0 ) );
    // --> Definicao dos outros parametos da IP <-- //
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.intended_device_family = "MAX 10",                          // Sera utilizado
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=1234",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.ram_block_type = "M9K",                                     // Usar blocos do
        altsyncram_component.numwords_a = N_WORDS,                                       // Usada aqui a co
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        //altsyncram_component.power_up_uninitialized = "TRUE",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ", // Uma operacao de
        altsyncram_component.widthad_a = ADDR_WIDTH,                                     // Usada aqui a co
        altsyncram_component.width_a = DATA_WIDTH,                                       // Usada aqui a co
        altsyncram_component.width_byteena_a = 1;
endmodule
```

9 - Módulos do arquivo `top.sv`

- Arquivo com os módulos específicos desse projeto.

9.1 - `controller`:

- **DESCRIÇÃO**:
  - .
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - “:
  - “: .
- **SAÍDAS (*outputs*)**:
  - “: .
  - “: .

9.2 - `datapath`:

- **DESCRIÇÃO**:
  - .
- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.
  - `END_IDX`: $DATA\_WIDTH - 1$.
- **ENTRADAS (*inputs*)**:
  - “:
  - “: .
- **SAÍDAS (*outputs*)**:
  - “: .
  - “: .

9.3 - `riscv_single`:

- **DESCRIÇÃO**:
  - .
- **PARÂMETROS (constantes)**:

- **DATA_WIDTH**: Comprimento das *words*, em bits.

- **END_IDX**: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - “:

  - “: .

- **SAÍDAS (*outputs*)**:
  - “: .

  - “: .

```
module riscv_single #( parameter DATA_WIDTH=32, parameter END_IDX=(DATA_WIDTH-1),  parameter ADDR_W_ROM=
                parameter ADDR_W_RAM=10, parameter HEX_FILE="Script_teste_01.txt" )
                    ( input  logic             clk,
                      input  logic             reset,
                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,
                      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
//------------------------------------------------------------------------------------------------
    // --> Constantes









    // --> Sinais e variaveis usadas aqui:
    logic [31:0] instr;  // Instrucoes de 32 bits executadas na CPU criada aqui
    logic [END_IDX:0] read_data, pc;


endmodule
```

9.4 - principal:

- **DESCRIÇÃO**:
  - .

- **PARÂMETROS (constantes)**:
  - **DATA_WIDTH**: Comprimento das *words*, em bits.

  - **END_IDX**: $DATA\_WIDTH - 1$.

- **ENTRADAS (*inputs*)**:
  - “:

  - “: .

- **SAÍDAS (*outputs*)**:
  - “: .

  - “: .

```
// Modulo principal (criado dentro de 'top')  ////////////////////////////////////////////////////
module principal #( parameter DATA_WIDTH=32, parameter END_IDX=(DATA_WIDTH-1),  parameter ADDR_W_ROM=15
              parameter ADDR_W_RAM=13, parameter HEX_FILE="Script_teste_01.txt" )
                  ( input  logic             clk,
                    input  logic             reset,
                    output logic             mem_write,
                    output logic [END_IDX:0] pc,
              output logic [END_IDX:0] write_data,
                    output logic [END_IDX:0] data_addr );
//-------------------------------------------------------------------------------------------------------
    // --> Constantes


    // --> Sinais e arrays de sinais
      logic [2:0] imm_src;
    logic [31:0] instr;  // As instrucoes do RISC-S possuem 32 bits de comprimento, mesmo na imlementac
    logic [END_IDX:0] read_data, pc, imm_ext;

    /*************************************************************
    ** Memoria ROM para armazenar o programa (program memory)  **
    *************************************************************/
    instr_mem #( .DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_W_ROM), .HEX_FILE(HEX_FILE) ) IMem
             ( .addr( pc ), .instr( instr ) );


    /*************************************************************
    ** Memoria RAM para armazenamento temporario (data memory)  **
    *************************************************************/
    data_mem_single #( .DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_W_RAM)) DMem
                 ( .clk( clk ), .w_en( mem_write ), .addr( data_addr ), .w_data( write_data ), .r_data(


    // --> 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 ) );

endmodule
```

10.5 - `top`:

- **DESCRIÇÃO**:
  - .

- **PARÂMETROS (constantes)**:
  - `DATA_WIDTH`: Comprimento das *words*, em bits.

- `END_IDX`: $DATA\_WIDTH - 1$.

- **ENTRADAS** (***inputs***):
  - “:
  - “: .

- **SAÍDAS** (***outputs***):
  - “: .
  - “: .

```
module top ( input  logic       MAX10_CLK1_50,
         input  logic [9:0] SW,
         output logic [9:0] LEDR,
         output logic [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 );
//-------------------------------------------------------------------------------------------
    // --> Constantes:
    parameter RST = 1'b0;

    // --> Sinais e variaveis usadas aqui:
    logic MemWrite, RegWrite;
    logic [31:0] WriteData, DataAdr, PC;



    principal #( .DATA_WIDTH(32), .ADDR_W_ROM(15), .ADDR_W_RAM(13), .HEX_FILE("Script_teste_01.txt") ) i
             ( .clk( MAX10_CLK1_50 ), .reset( RST ), .pc( PC ), .write_data( WriteData ), .data_addr( Da

endmodule
```
