Arquivo com os Módulos Básicos

Módulos do Arquivo `modulos_basicos.sv`:

- Arquivo `modulos_basicos.sv`: Módulos de uso geral, não específicos da CPU RISC-V.
  - Módulo `ff_rst`: *Flip-Flop* com *RESET*.
  - Módulo `ff_rst_en`: *Flip-Flop* com *RESET* e *ENABLE*.
  - Módulo `mux2`: Multiplexador 2:1.
  - Módulo `mux3`: Multiplexador 3:1.
  - Módulo `mux4`: Multiplexador 4:1.
  - Módulo `adder`: **Somador** simples.
  - Módulo `adder2`: **Somador** completo.
  - Módulo `multiply`: **Multiplicação** e resultado das operações `mulh`, `mulhsu` e `mulhu`.
  - Módulo `divide_remainder`: **DIVISÃO** e **RESTO** da divisão; números **inteiros sinalizados**.
  - Módulo `divide_remainder_unsigned`: **DIVISÃO** e **RESTO** da divisão; números **inteiros não-sinalizados**.
  - Módulo `shift_right_arithmetic`: **DESLOCAMENTO ARITMÉTICO** para a direita.
  - Módulo `logical_shift_opers`: **DESLOCAMENTO LÓGICO** para esquerda e direita.
  - Módulo `set_less_than`: Operação **MENOR QUE**.
  - Módulo `logical_oper_alu`: Resultado das operações lógicas `and`, `or` e `xor`.
  - Módulo `dig_displ_7_segs`: Escreve um determinado valor hexadecimal em um display de 7 segmentos.

---

1 - Módulos Referentes aos *Flip-Flops*:

1.1 - Módulo `ff_rst`

- *Flip-Flop* com ***RESET***.
  - **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.
    * `d`: Valor na entrada `d`do *Flip-Flop*.
  - **Saídas**:
    * `q`: Valor na saída `q`do *Flip-Flop*.

```
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
```

1.2 - Módulo `ff_rst_en`

- *Flip-Flop* com ***RESET*** e ***ENABLE***.
  - **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.
    * `en`: Entrada *ENABLE*.
    * `d`: Valor na entrada `d`do *Flip-Flop*.
  - **Saídas**:
    * `q`: Valor na saída `q`do *Flip-Flop*.

```
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 - Módulos Referentes aos Multiplexadores:

2.1 - Módulo `mux2`

- Multiplexador **2:1**.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `d0`: Saída selecionável 0.
    - `d1`: Saída selecionável 1.
    - `sel`: Sinal de 1 bit para seleconar uma das 2 saídas disponíveis.
  - **Saídas**:
    - `y`: Saída selecionada.

```
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
```

2.2 - Módulo `mux3`

- Multiplexador **3:1**.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `d0`: Saída selecionável 0.
    - `d1`: Saída selecionável 1.
    - `d2`: Saída selecionável 2.
    - `sel`: Array de 2 bits para seleconar uma das 3 saídas disponíveis.
  - **Saídas**:

        * `y`: Saída selecionada.

```
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
```

2.3 - Módulo `mux4`

- Multiplexador **4:1**.
  - **Constantes**:
    * `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.

    * `END_IDX`: Índice do último elemento do array com os dados.

  - **Entradas**:
    * `d0`: Saída selecionável 0.

    * `d1`: Saída selecionável 1.

    * `d2`: Saída selecionável 2.

    * `d3`: Saída selecionável 3.

    * `sel`: Array de 2 bits para seleconar uma das 4 saídas disponíveis.

  - **Saídas**:
    * `y`: Saída selecionada.

```
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
```

---

3 - Módulos Referentes às operações de SOMA

3.1 - Módulo `adder`

- **Somador** simples.
  - **Constantes**:

        * `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.

        * `END_IDX`: Índice do último elemento do array com os dados.

    - **Entradas**:
        * `op_val1`: Valor do primeiro operando.

        * `op_val2`: Valor do segundo operando.

    - **Saídas**:
        * `sum_result`: Resultado da soma.

```
module adder #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
              ( input  logic [END_IDX:0] op_val1,
                input  logic [END_IDX:0] op_val2,
                output logic [END_IDX:0] sum_result );
//-------------------------------------------------------------------------------------------
      assign sum_result = op_val1 + op_val2;
endmodule
```

3.2 - Módulo adder2

- **Somador** completo.
    - **Constantes**:
        * `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.

        * `END_IDX`: Índice do último elemento do array com os dados.

    - **Entradas**:
        * `op_val1`: Valor do primeiro operando.

        * `op_val2`: Valor do segundo operando.

        * `cin`: Bit de *carry-n* da operação.

    - **Saídas**:
        * `sum_result`: Resultado da soma.

        * `cout`: Bit de *carry-out* da operação.

```
module adder2 #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
               ( input  logic signed [END_IDX:0] op_val1,
                 input  logic signed [END_IDX:0] op_val2,
              input  logic                      cin,
                  output logic signed [END_IDX:0] sum_result,
              output logic                      cout );
//-------------------------------------------------------------------------------------------
      assign {cout, sum_result} = op_val1 + op_val2 + cin;
endmodule
```

---

4 - Módulos Referentes às operações de MULTIPLICAÇÃO e DIVISÃO

4.1 - Módulo `multiply`

- **Multiplicação** e resultado das operações **`mulh`**, **`mulhsu`** e **`mulhu`**.
  - **Constantes**:
    - **`DATA_WIDTH`**: Comprimento, em bits, do *array* de dados.
    - **`END_IDX`**: Índice do último elemento do array com os dados.
  - **Entradas**:
    - **`op_val1`**: Valor do primeiro operando.
    - **`op_val2`**: Valor do segundo operando.
  - **Saídas**:
    - **`prod_result`**: Resultado da operação de multiplicação.
    - **`prod_high_ss`**: Conteúdo dos bits Bits $[DATA_WIDTH : ((2 \times DATA_WIDTH) - 1)]$ da multiplicação de dois números sinalizados.
    - **`prod_high_su`**: Conteúdo dos bits Bits $[DATA_WIDTH : ((2 \times DATA_WIDTH) - 1)]$ da multiplicação de um número sinalizado, por um inteiro.
    - **`prod_high_uu`**: Conteúdo dos bits Bits $[DATA_WIDTH : ((2 \times DATA_WIDTH) - 1)]$ da multiplicação de dois números não-sinalizados.

```
(* multstyle = "dsp" *) module multiply #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                                        ( input  logic [END_IDX:0] op_val1,
                                          input  logic [END_IDX:0] op_val2,
                                         output logic [END_IDX:0] prod_result,
                                         output logic [END_IDX:0] prod_high_ss,
                                output logic [END_IDX:0] prod_high_su,
                                output logic [END_IDX:0] prod_high_uu );
//-------------------------------------------------------------------------------------------
    parameter LAST_ADDR = (2*DATA_WIDTH)-1;
    logic [LAST_ADDR:0] mul1, mul2, mul3;

    assign mul1 = $signed(op_val1) * $signed(op_val2);
    assign mul2 = $signed(op_val1) * $unsigned(op_val2);
    assign mul3 = $unsigned(op_val1) * $unsigned(op_val2);
    // Arrays com as saidas
    assign prod_result = mul3[END_IDX:0];
    assign prod_high_uu = mul3[LAST_ADDR:DATA_WIDTH];
    assign prod_high_ss = mul1[LAST_ADDR:DATA_WIDTH];
    assign prod_high_su = mul2[LAST_ADDR:DATA_WIDTH];
endmodule
```

4.2 - Módulo `divide_remainder`

- **Divisão e resto** da divisão; números **inteiros SINALIZADOS**.
  - **Constantes**:
    - **`DATA_WIDTH`**: Comprimento, em bits, do *array* de dados.

    * `END_IDX`: Índice do último elemento do array com os dados.

  - **Entradas**:
    * `operand_1`: Valor do primeiro operando.

    * `operand_2`: Valor do segundo operando.

  - **Saídas**:
    * `quotient`: Quociente da operação de divisão.

    * `remainder`: Resto da operação de divisão.

```
module divide_remainder #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                          ( input  logic [END_IDX:0] operand_1,
                            input  logic [END_IDX:0] operand_2,
                           output logic [END_IDX:0] quotient,
                          output logic [END_IDX:0] remainder );
//------------------------------------------------------------------------------------------
    logic [END_IDX:0] q, r;
    assign q = (operand_1 / operand_2);
    assign r = (operand_1 % operand_2);
    assign quotient = q[END_IDX:0];
    assign remainder = r[END_IDX:0];
endmodule
```

4.3 - Módulo `divide_remainder_unsigned`

- **Divisão e resto** da divisão; números **inteiros NÃO-SINALIZADOS**.
  - **Constantes**:
    * `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.

    * `END_IDX`: Índice do último elemento do array com os dados.

  - **Entradas**:
    * `operand_1`: Valor do primeiro operando.

    * `operand_2`: Valor do segundo operando.

  - **Saídas**:
    * `quotient`: Quociente da operação de divisão.

    * `remainder`: Resto da operação de divisão.

```
module divide_remainder_unsigned #( parameter DATA_WIDTH=32, parameter END_IDX=DATA_WIDTH-1 )
                                  ( input  logic [END_IDX:0] operand_1,
                                    input  logic [END_IDX:0] operand_2,
                                   output logic [END_IDX:0] quotient,
                                  output logic [END_IDX:0] remainder );
//------------------------------------------------------------------------------------------
    logic [END_IDX:0] q, r;
    assign q = $unsigned(operand_1) / $unsigned(operand_2);
    assign r = $unsigned(operand_1) % $unsigned(operand_2);
    assign quotient = q[END_IDX:0];
```

```
    assign remainder = r[END_IDX:0];
endmodule
```

---

5 - Operações de DESLOCAMENTO LÓGICO e ARIMÉTICO

5.1 - Módulo `shift_right_arithmetic`

- **DESLOCAMENTO ARITMÉTICO** para a direita.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `src1_value`: Valor do primeiro operando.
    - `src2_value`: Valor do segundo operando.
  - **Saídas**:
    - `sra_rslt`: Resultado das operações `sra`/`srai`.

```
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 registrador
                        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
```

5.2 - Módulo `logical_shift_opers`

- **Deslocamento lógico** para esquerda e direita.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `src1_value`: Valor do primeiro operando.
    - `src2_value`: Valor do segundo operando.
  - **Saídas**:

        * **left_shift**: Resultado das operações **sll**/**slli**.

        * **right_shift**: Resultado das operações **srl**/**srli**.

```
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
```

---

6 - Oprações Lógicas

6.1 - Módulo set_less_than

- Operação **MENOR QUE**.
  - **Constantes**:
    * **DATA_WIDTH**: Comprimento, em bits, do *array* de dados.

    * **END_IDX**: Índice do último elemento do array com os dados.

  - **Entradas**:
    * **src1_value**: Valor do primeiro operando.

    * **src2_value**: Valor do segundo operando.

  - **Saídas**:
    * **sltu_rslt**: Resultado da operação da instrução **sltu**.

    * **slt_rslt**: Resultado da operação da instrução **slt**.

```
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 'rs2' o
                output logic [END_IDX:0] sltu_rslt,   // Resultado da operacao (unsigned)
                output logic [END_IDX:0] slt_rslt );  // Resultado da operacao (signed)
//-------------------------------------------------------------------------------------------
    // Constantes
    parameter ZERO_VAL = { END_IDX { 1'b0 } };
    // Realizar a operacao
    assign sltu_rslt = { ZERO_VAL, (src1_value < src2_value) };
    assign slt_rslt = (src1_value[END_IDX] == src2_value[END_IDX]) ? sltu_rslt : { ZERO_VAL, src1_value
endmodule
```

6.2 - Módulo `logical_oper_alu`

- Resultado das operações lógicas **`and`**, **`or`** e **`xor`**.
  - **Constantes**:
    - **`DATA_WIDTH`**: Comprimento, em bits, do *array* de dados.
    - **`END_IDX`**: Índice do último elemento do array com os dados.
  - **Entradas**:
    - **`src1_value`**: Valor do primeiro operando.
    - **`src2_value`**: Valor do segundo operando.
  - **Saídas**:
    - **`result_and`**: Array com os resultados das operações **AND**.
    - **`result_or`**: Array com os resultados das operações **OR**.
    - **`result_xor`**: Array com os resultados das operações **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
```

---

7 - Outros Módulos

7.1 - Módulo `dig_displ_7_segs`

- Escreve um determinado valor hexadecimal em um display de 7 segmentos.
  - **Constantes**: NÃO POSSUI.
  - **Entradas**:
    - **`digit`**: Array com o valor a ser escrito no display de 7 segmentos.
  - **Saídas**:
    - **`segs_dsp`**: *Array* indicando os pinos do display de 7 segmentos.

```
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
```