Arquivos com os Módulos da ALU e os Módulos Auxiliares da CPU

Módulos dos Arquivos `modulos_auxiliares_cpu.sv` e `alu.sv`:

- Arquivo **`modulos_auxiliares_cpu.sv`**: Módulos cuja implementação é específica para a CPU RISC-V desenvolvida nesse projeto.
  - Módulo **`extend`**: Preenche o campo referente ao **"VALOR IMEDIATO" (*IMMEDIATE*)** da instrução.
  - Módulo **`take_branch`**: Indica se a **instrução é do TIPO *BRANCH*** e se a condição do *branch* é verdadeira ou falsa.
  - Módulo **`instr_fields`**: **DECODIFICA OS CAMPOS DA INSTRUÇÃO** de 32 bits da ISA RISC-V.
  - Módulo **`main_dec`**: **DECODIFICADOR PRINCIPAL** das instruções.
  - Módulo **`alu_dec`**: Decodificador que le os campos da instrução e **retorna o código da instrução na ALU**.
- Arquivo **`alu.sv`**: Contém a ALU (*Arithmetic Logic Unit*) da CPU.
  - Módulo **`output_flags_alu`**: Módulo para retornar os ***OUTPUT FLAGS*** das operações da ALU.
  - Módulo **`alu`**: **ALU implementada** na CPU RISC-V.

---

1 - Módulos `extend` e `take_branch`

1.1 - Módulo `extend`

- Preenche o campo referente ao **"VALOR IMEDIATO" (*IMMEDIATE*)** da instrução.
  - **Constantes**:
    - **`DATA_WIDTH`**: Comprimento, em bits, do *array* de dados.
    - **`END_IDX`**: Índice do último elemento do array com os dados.
  - **Entradas**:
    - **`instr`**: Bits 7 a 31 da instrução.
    - **`imm_src`**: Array de 3 bits retornados pelo módulo **`main_dec`**.
  - **Saídas**:
    - **`imm_ext`**: Array de 32 bits com o conteúdo do valor imediato (*immediate*).

```
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
```

1.2 - Módulo `take_branch`

- Indica se a **instrução é do TIPO *BRANCH*** e se a condição do *branch* é verdadeira ou falsa.
  - **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 da ALU.
    - `src2_value`: Valor do segundo operando da ALU.
    - `alu_control`: Código da instrução que será executada na ALU.
  - **Saídas**:
    - `taken_br`: Sinal indicando se a operação de *branch* deverá ocorrer ou não.

```
module take_branch #( parameter DATA_WIDTH = 32, parameter END_IDX=DATA_WIDTH-1 )
                    ( input  logic [END_IDX:0] src1_value,   // Primeiro operando. Valor no registrador
                  input  logic [END_IDX:0] src2_value,   // Segundo operando. Valor no registrador 'rs2'
                    input  logic [      4:0] alu_control,  // Codigo da instrucao executada na ALU
                    output logic             taken_br );   // Indica a ocorrencia de branching
//-----------------------------------------------------------------------------------------
    always_comb begin
        case( alu_control )
            // Instrucao 'beq'
            5'b01111: taken_br = (src1_value == src2_value);
            // Instrucao 'bne'
            5'b10000: taken_br = (src1_value != src2_value);
            // Instrucao 'blt' ou 'bltu'
            5'b10001: taken_br = (src1_value < src2_value) ^ (src1_value[END_IDX] != src2_value[END_IDX]
            // Instrucao 'bge' ou 'bgeu'
            5'b10010: taken_br = (src1_value >= src2_value) ^ (src1_value[END_IDX] != src2_value[END_ID
            // Caso padrao
            default:  taken_br = 1'b0;
        endcase
    end
endmodule
```

2 - Módulos que Formam o Decodificador de Instruções (*Instruction Decoder*)

2.1 - Módulo `instr_fields`

- **DECODIFICA OS CAMPOS DA INSTRUÇÃO** de 32 bits da ISA RISC-V.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `instr`: Instrução de 32 bits a ser decodificada.
  - **Saídas**:
    - `opcode`: Bits 0 a 7 da instrução.
    - `rd`: Endereço do registrador de destino.
    - `funct3`: Bits 12 a 14 da instrução.
    - `rs1`: Endereço do registrador com o primeiro operando.
    - `rs2`: Endereço do registrador com o segundo operando.
    - `funct7b0`: Bit 25 da instrução.
    - `funct7b5`: Bit 30 da instrução.

```
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 )
                  // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
                  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_A
            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
```

2.2 - Módulo `main_dec`

- **DECODIFICADOR PRINCIPAL** das instruções.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `opcode`: Bits 0 a 7 da instrução.
  - **Saídas**:
    - `result_src`: Array de 2 bits.
    - `mem_write`: Sinal indicando se a instrução deverá escrever dados na memória.
    - `branch`: Sinal indicando uma operação de *branch*.
    - `alu_src`: Sinal indicando se devemos usar o valor gerado na saída `result` da ALU.
    - `reg_write`: Sinal indicando se deverá ser realizada uma operação de escrita no *register file*.
    - `is_jal`: Sinal indicando se a instrução atual é `jal`.
    - `is_jalr`: Sinal indicando se a instrução atual é `jalr`.
    - `imm_src`: *Array* de 3 bits indicando se a instrução atual recebe um valor imediato (*immediate*).
    - `alu_op`: *Array* de 2 bits com a operação da ALU.

```
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_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 [12:0] controls;
      assign { reg_write, imm_src, alu_src, mem_write, result_src, branch, alu_op, is_jal, is_jalr } = c

      always_comb begin
            case( opcode )
                  // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
                  7'b0000011: controls = 13'b1_000_1_0_01_0_00_0_0; // Instrucoes que usam o formato I-
                  7'b0010011: controls = 13'b1_000_1_0_00_0_10_0_0; // Instrucoes que usam o formato I-
            7'b0010111: controls = 13'b1_100_0_0_11_0_00_0_0; // Instrucoes que usam o formato U-Type,
            7'b0100011: controls = 13'b0_001_1_1_00_0_00_0_0; // Instrucoes que usam o formato S-Type,
                  7'b0110011: controls = 13'b1_xxx_0_0_00_0_10_0_0; // Instrucoes que usam o formato R-
                  7'b0110111: controls = 13'b1_100_0_0_11_0_00_0_0; // Instrucoes que usam o formato U-
            7'b1100011: controls = 13'b0_010_0_0_00_1_01_0_0; // Instrucoes que usam o formato B-Type,
            7'b1100111: controls = 13'b1_000_1_0_00_0_10_0_1; // Instrucoes que usam o formato I-Type,
            7'b1101111: controls = 13'b1_011_0_0_10_0_00_1_0; // Instrucoes que usam o formato J-Type,
                  default:    controls = 13'bx_xxx_x_x_xx_x_xx_x_0; // non-implemented instruction
            endcase
      end
endmodule
```

2.3 - Módulo `alu_dec`

- Decodificador que le os campos da instrução e **retorna o código da instrução na ALU**.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `opcode`: Bits 0 a 7 da instrução.
    - `funct3`: Bits 12 a 14 da instrução.
    - `funct7b0`: Bit 25 da instrução.
    - `funct7b5`: Bit 30 da instrução.
    - `alu_op`: Array de 2 bits com a operação da ALU.
  - **Saídas**:
    - `alu_ctrl`: *Array* de 5 bits com o código da instrução executada na ALU.

```
module alu_dec ( input  logic [6:0] opcode,
                 input  logic [2:0] funct3,
                 input  logic       funct7b0,
             input  logic       funct7b5,
                 input  logic [1:0] alu_op,
                 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};

      always_comb begin
            case( alu_op )
              2'b00:   alu_ctrl = 5'b00000; // Adicao
              2'b01:   alu_ctrl = 5'b00001; // Subtracao
              default: 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'b0_0_010_0110011: alu_ctrl = 5'b00111; // Instrucao 'slt'
                    12'b0_0_011_0110011: alu_ctrl = 5'b01000; // Instrucao 'sltu'
                    12'b1_0_101_0110011: alu_ctrl = 5'b01001; // Instrucao 'sra'
                    // Instrucoes do tipo I-Type
                    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'bx_x_010_0010011: alu_ctrl = 5'b00111; // Instrucao 'slti'
                    12'bx_x_011_0010011: alu_ctrl = 5'b01000; // Instrucao 'sltui'
                    12'b1_0_101_0010011: alu_ctrl = 5'b01001; // Instrucao 'srai'
                    // Instrucoes 'lui' 'auipc', 'jal' e 'jalr'
                      12'bx_x_xxx_0110111: alu_ctrl = 5'b01010; // Instrucao 'lui'
                      12'bx_x_xxx_0010111: alu_ctrl = 5'b01011; // Instrucao 'auipc'
                      12'bx_x_xxx_1101111: alu_ctrl = 5'b01100; // Instrucao 'jal'
                      12'bx_x_000_1100111: alu_ctrl = 5'b01101; // Instrucao 'jalr'
                      // Instrucoes do tipo B-Type
                      12'bx_x_000_1100011: alu_ctrl = 5'b01110; // Instrucao 'beq'
                      12'bx_x_001_1100011: alu_ctrl = 5'b01111; // Instrucao 'bne'
                      12'bx_x_100_1100011: alu_ctrl = 5'b10000; // Instrucao 'blt'
                      12'bx_x_101_1100011: alu_ctrl = 5'b10001; // Instrucao 'bge'
                      12'bx_x_110_1100011: alu_ctrl = 5'b10000; // Instrucao 'bltu'
                      12'bx_x_111_1100011: alu_ctrl = 5'b10001; // Instrucao 'bgeu'
                           // Instrucoes R-Type Multiplicacao/Divisao
                      12'b0_1_000_0110011: alu_ctrl = 5'b10010; // Instrucao 'mul'
                      12'b0_1_001_0110011: alu_ctrl = 5'b10011; // Instrucao 'mulh'
                      12'b0_1_010_0110011: alu_ctrl = 5'b10100; // Instrucao 'mulhsu'
```

```
                        12'b0_1_011_0110011: alu_ctrl = 5'b10101; // Instrucao 'mulhu'
                        12'b0_1_100_0110011: alu_ctrl = 5'b10110; // Instrucao 'div'
                        12'b0_1_101_0110011: alu_ctrl = 5'b10111; // Instrucao 'divu'
                        12'b0_1_110_0110011: alu_ctrl = 5'b11000; // Instrucao 'rem'
                        12'b0_1_111_0110011: alu_ctrl = 5'b11001; // Instrucao 'remu'
                        // caso padrao
                    default:  alu_ctrl = 5'bxxxxx;
                        endcase
            endcase
      end
endmodule
```

---

3 - Módulos que Formam a Estrutura da ALU (*Arithmetic Logic Unit*)

3.1 - Módulo `output_flags_alu`

- Módulo para retornar os ***OUTPUT FLAGS*** das operações da ALU.
  - **Constantes**:
    - `DATA_WIDTH`: Comprimento, em bits, do *array* de dados.
    - `END_IDX`: Índice do último elemento do array com os dados.
  - **Entradas**:
    - `src1_b31`: Último bit do primeiro operando da ALU.
    - `src2_b31`: Último bit do segundo operando da ALU.
    - `is_add_sub`: Sinal indicando se a operação é uma soma ou uma subtração.
    - `alu_ctrl_b0`: Bit 0 de `alu_ctrl`; indica se a operação é subtração.
    - `cout`: Sinal indicando se foi produzido um *carry out*.
    - `result`: *Array* com o resultado da operação gerado na ALU.
  - **Saídas**:
    - `of_c`: *Output Flag* `c` - Indica se a operação **gerou um *carry-out***.
    - `of_n`: *Output Flag* `n` - Indica se **`result` é negativo**.
    - `of_v`: *Output Flag* `v` - Indica a ocorrência de ***overflow***.
    - `of_z`: *Output Flag* `z` - Indica se **`result` é 0**.

```
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 de soma o
                    input  logic             alu_ctrl_b0,  // Bit 0 de 'alu_ctrl', indica se a operacao
                    input  logic             cout,         // Sinal indicando se foi produzido um 'carry
```

```
                  input  logic [END_IDX:0] result,       // Array com o resultado da operacao gerado n
                  output logic             of_c,         // 'Output Flag' 'c': Indica se a operacao ge
                  output logic             of_n,         // 'Output Flag' 'n': Indica se o resultado e
                  output logic             of_v,         // 'Output Flag' 'v': Indica a ocorrencia de
                  output logic             of_z   );     // 'Output Flag' 'z': Indica se o resultado e
//-------------------------------------------------------------------------------------------------
    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
```

3.2 - Módulo `alu`

- **ALU implementada** na CPU RISC-V.
  - **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.
    - `alu_ctrl`: Código da instrução que será executada na ALU.
  - **Saídas**:
    - `result`: *Array* com o resultado da operação gerado na ALU.
    - `of_c`: *Output Flag* `c` - Indica se a operação **gerou um *carry-out***.
    - `of_n`: *Output Flag* `n` - Indica se **`result` é negativo**.
    - `of_v`: *Output Flag* `v` - Indica a ocorrência de ***overflow***.
    - `of_z`: *Output Flag* `z` - Indica se **`result` é 0**.

```
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,
              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 um ca
```

```
        output logic             of_n,         // 'Output Flag' 'n': Indica se o resultado eh negativ
        output logic             of_v,         // 'Output Flag' 'v': Indica a ocorrencia de overflow
        output logic             of_z );       // 'Output Flag' 'z': Indica se o resultado eh 0
//-------------------------------------------------------------------------------------------------
    // --> Constantes
    parameter NULL_VAL = { DATA_WIDTH { 1'b0 } };

    // --> Declaracao dos sinais e arrays de sinais
    logic             is_add_sub, is_sub; // Sinal indicando se a operacao eh de soma
    logic             cout;       // Bit de carry out
    logic [END_IDX:0] cond_inv_b;
      logic [END_IDX:0] sum;
    logic [END_IDX:0] res_and, res_or, res_xor;
    logic [END_IDX:0] sltu_rslt, slt_rslt, sra_rslt;
    logic [END_IDX:0] prod_result, prod_high_ss, prod_high_su, prod_high_uu;
    logic [END_IDX:0] quotient, quotient_u, remainder, remainder_u;
    logic [END_IDX:0] left_shift_res, right_shift_res, lui_value; //, auipc_value;

    // --> Soma e Carry-Out
    assign is_add_sub = (alu_ctrl == 5'b00000) || (alu_ctrl == 5'b00001);
    assign is_sub = (alu_ctrl == 5'b00001);

    mux2 #( .DATA_WIDTH(DATA_WIDTH) ) mux_src2_val
        ( .d0(src2_value),
          .d1(~src2_value),
          .sel(is_sub),
          .y(cond_inv_b) );

    // Modulo do somador que ira retornar o resultado das operacoes de soma/subtracao
      adder2 #( .DATA_WIDTH(DATA_WIDTH) ) add_sub_op
            ( .op_val1(src1_value),
            .op_val2(cond_inv_b),
            .cin(is_sub),
            .sum_result(sum),
            .cout(cout) );

    // --> Operacoes Logicas AND, OR e XOR
    logical_oper_alu #( .DATA_WIDTH(DATA_WIDTH) ) log_op_alu
                ( .src1_value(src1_value),
                  .src2_value(src2_value),
                  .result_and(res_and),
                  .result_or(res_or),
                  .result_xor(res_xor) );

    // --> Operacoes de deslocamento logico ('sll', 'slli', 'sra', 'srai')
    logical_shift_opers #( .DATA_WIDTH(DATA_WIDTH) ) shift_opers
                   ( .src1_value(src1_value),
                     .src2_value(src2_value),
                     .left_shift(left_shift_res),
                     .right_shift(right_shift_res) );

    // --> Operacoes 'sra'/'srai'
    shift_right_arithmetic #( .DATA_WIDTH(DATA_WIDTH) ) oper_sra
```

```
                        ( .src1_value(src1_value),
                    .src2_value(src2_value),
                    .sra_rslt(sra_rslt) );

  // --> Operacoes 'slt'/'slti'/'sltu'/'sltiu'
  set_less_than #( .DATA_WIDTH(DATA_WIDTH) ) oper_slt
                 ( .src1_value(src1_value),
                .src2_value(src2_value),
                .sltu_rslt(sltu_rslt),
                .slt_rslt(slt_rslt) );

  // --> Operacoes 'mul'/'mulh'/'mulhsu'/'mulhu'
  multiply #( .DATA_WIDTH(DATA_WIDTH) ) multiplicador
          ( .op_val1(src1_value),
            .op_val2(src2_value),
          .prod_result(prod_result),
          .prod_high_ss(prod_high_ss),
            .prod_high_su(prod_high_su),
          .prod_high_uu(prod_high_uu) );

  // --> Operacoes 'div', 'divu', 'rem' e 'remu'
  divide_remainder #( .DATA_WIDTH(DATA_WIDTH) ) div_rem_s
                      ( .operand_1(src1_value),
                    .operand_2(src2_value),
                    .quotient(quotient),
                    .remainder(remainder) );

  // --> Valor da operacao 'lui'
  assign lui_value = { src2_value[31:12], {12 { 1'b0 }} };

  // --> Selecionar o output indicado em 'alu_ctrl'
    always_comb begin
          case( alu_ctrl )
                // --> Instrucoes aritmeticas nos formatos R-Type ou I-Type
          5'b00000:  result = sum;              // 'add'/'addi'
                5'b00001:  result = sum;              // 'sub'
                5'b00010:  result = res_and;          // 'and'/'andi'
          5'b00011:  result = res_or;           // 'or'/'ori'
          5'b00100:  result = res_xor;          // 'xor'/'xori'
          5'b00101:  result = left_shift_res;   // 'sll'/'slli'
          5'b00110:  result = right_shift_res;  // 'srl'/'srli'
          5'b00111:  result = slt_rslt;         // 'slt'/'slti'
          5'b01000:  result = sltu_rslt;        // 'sltu'/'sltui'
          5'b01001:  result = sra_rslt;         // 'sra'/'srai'
          5'b01010:  result = lui_value;        // 'lui'
          // --> Multiplicacao/Divisao
          5'b10010:  result = prod_result;    // 'mul'
          5'b10011:  result = prod_high_ss;   // 'mulh'
          5'b10100:  result = prod_high_su;   // 'mulhsu
          5'b10101:  result = prod_high_uu;   // 'mulhu'
          5'b10110:  result = quotient;       // 'div'
          5'b10111:  result = quotient;       // 'divu'
          5'b11000:  result = remainder;      // 'rem'
```

```
            5'b11001:  result = remainder;      // 'remu
            default:   result = NULL_VAL;       // Caso padrao
            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_add_sub),
                  .alu_ctrl_b0(alu_ctrl[0]),
                  .cout(cout),
                  .result(result),
                  .of_c(of_c),
                  .of_n(of_n),
                  .of_v(of_v),
                  .of_z(of_z) );
endmodule
```