Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DCC007-Organização de Computadores II

Arquitetura de Computadores: Trabalho Prático 2 - ULA

Guilherme ...
Iuri Castro
João Mateus de Freitas Veneroso
Ricardo Pagoto Marinho

Belo Horizonte - MG
10 de outubro de 2017

1 Introdução

O presente trabalho foca na descrição da implementação de uma Unidade Lógica e Aritmética (ULA) em Verilog, uma Linguagem de descrição de hardware, *Hardware Description Language* - HDL - em inglês. A ULA é um circuito digital responsável por realizar operações lógicas e aritméticas no caminho de dados de uma CPU. É a ULA que realiza operações como adição, subtração e *and* e *or* lógicos. Além disso, ela também é responsável por calcular o endereço de memória para escrita ou leitura quando as instruções requisitam. Para implementar a ALU, utilizamos a IDE *Quartus v 13* junto com *ModelSim* para simular uma FPGA e fazer os testes necessários. Os testes em dispositivo físico foi feito numa *FPGA Cyclone II* da Altera.

2 ULA

A Figura 1 mostra o desenho esquemático de uma ULA.

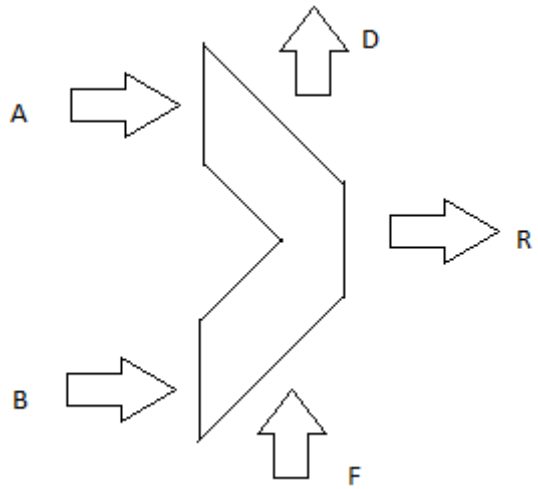

Figura 1: Desenho esquemático de uma ULA.

Nela, é possível ver que a ULA possui duas entradas e uma saída de dados, A, B e R respectivamente, além de uma entrada e uma saída de sinais de controle, F e D. As entradas A e B são os valores que a ULA recebe para fazer os cálculos necessários. Esses valores podem vir de registradores ou podem ser imediatos: números absolutos, como nas instruções a seguir:

```
1 Addi R1,R2,10
2 Sub R3,R4,R5
```

A instrução 1 soma o valor imediato 10 ao valor armazenado no registrador *R2* e armazena no registrador *R1*. No caso dessa instrução, a entrada A na Figura 1 recebe o valor do registrador *R2*, a entrada B o valor do imediato (10) e a saída R terá o valor da soma, sendo que este valor será armazenado no registrador *R1*. Já a instrução 2 subtrai o valor armazenado no registrador *R5* do valor armazenado em *R4* e armazena em *R3*, fazendo a operação

$$R3 = R4 - R5 \tag{1}$$

A entrada de sinais F é a entrada de controles da ALU. Dentre esses sinais de controle está o que sinaliza se o valor na entrada de dados B é o valor vindo de um registrador ou um imediato. Esse

sinal controla um multiplexador para seleciona qual entrada irá passar para a ALU, a entrada vinda do banco de registradores ou a entrada vinda direto da instrução no caso de um imediato.

Já a saída de sinais *D*, são os sinais que a ALU indica, sendo eles se o resultado é zero, negativo e se houve *overflow* na operação.

3 Implementação

Esta seção fala da implementação da ULA em *Verilog*. Nela, mostramos como cada parte da ULA foi implementada além de decisões de projeto tomadas.

O módulo desenvolvido possui 5 entradas e duas saídas: *OpA*, *OpB*, *Op*, *RST*, *CLK*, *Res* e *FlagReg* respectivamente. O Apêndice 3 mostra o código.

As entradas *OpA* e *OpB* representam as entradas *A* e *B* da Figura 1. A entrada *Op* indica qual operação a ALU vai fazer (Add, Sub, etc.) e faz parte da entrada *F* na Figura 1. As saídas *Res* e *FlagReg* indicam, respectivamente o resultado da operação e o sinal de saída da ALU, na Figura 1, as saídas *R* e *D* respectivamente.

Na implementação, as entradas *OpA* e *OpB* são de *16 bits*, que, de acordo com a especificação do trabalho, é o tamanho dos registradores. Aqui, apenas o *OpB* pode ser um imediato. Neste caso, o imediato possui apenas *4 bits*, sendo necessário estender mais *12 bits*. Mais a frente será mostrado como e onde essa operação é feita. A entrada *Op* indica a operação a ser realizada e possui *4 bits*, ou seja, a ALU implementada possui um máximo de 16 operações diferentes. Pela especificação, apenas 11 operações foram utilizadas como mostrado na Tabela 1.

Código	Operação	Descrição
0	Add	Adição com registradores
1	Sub	Subtração com registradores
2	Slti	Salto condicional com imediato
3	And	*And* binário
4	Or	*Or* binário
5	Xor	*Xor* binário
6	Andi	*And* binário com imediato
7	Ori	*Or* binário com imediato
8	Xori	*Xor* binário com imediato
9	Addi	Adição com imediato
10	Subi	Subtração com imediato

Tabela 1: Descrição das operações implementadas.

A saída *Res* possui *16 bits*, assim como as entradas *OpA* e *OpB* já que é o resultado da operação e a saída *FlagReg* possui *3 bits*, um para cada sinal de saída como mostra a Tabela 2.

Índice	Sinal
0	*Carry/Overflow*
1	Negativo
2	Zero

Tabela 2. Sinais de saída da ULA.

Apêndices

Módulo ULA

```
module ULA (OpA, OpB, Res, Op, FlagReg, CLK, RST);

        input CLK, RST;
        input [3:0] Op;
        input [15:0] OpA, OpB;
        output reg [15:0] Res;
        output reg [2:0] FlagReg;                // [Z N C]: Z=Zero; N=Neg;
           C=Carry/Overflow

        wire [15:0] invOpB;
        assign invOpB = ~OpB + 16'd1;

        parameter InsADD = 4'b0000;      // ADD  Res = OpA + OpB
        parameter InsSUB = 4'b0001;      //      SUB     Res = OpA - OpB
        parameter InsSLT = 4'b0010;      //      SLTI    Res = (OpA > OpB) ?
            1 : 0
        parameter InsAND = 4'b0011;      //      AND     Res = OpA & OpB
        parameter InsOR =  4'b0100;      //      OR      Res = OpA | OpB
        parameter InsXOR = 4'b0101;      //      XOR     Res = OpA ^ OpB

        parameter OverflowFlag  = 0;
        parameter NegFlag               = 1;
        parameter ZeroFlag              = 2;

        always @(posedge CLK) begin
         if (RST) begin
          FlagReg = 3'b000;
         end
         else begin
          case (Op)
           InsADD: begin
            Res = OpA + OpB;

            /* Zero check */
            if (Res == 0)
                 FlagReg[ZeroFlag] = 1'b1;
            else
             FlagReg[ZeroFlag] = 1'b0;
            /* Overflow check */
            FlagReg[OverflowFlag] = (OpA[15] & OpB[15] & ~Res[15]) | (~OpA[
               15] & ~OpB[15] & Res[15]);
            /* Negative check */
            FlagReg[NegFlag] = Res[15];
           end
           InsSUB: begin
            Res = OpA + invOpB;
            /* Zero check */
            if (Res == 0)
             FlagReg[ZeroFlag] = 1'b1;
            else
             FlagReg[ZeroFlag] = 1'b0;
            /* Overflow check */
            FlagReg[OverflowFlag] = (OpA[15] & invOpB[15] & ~Res[15]) | (~
               OpA[15] & ~invOpB[15] & Res[15]);
```

```
51              /* Negative check */
52              FlagReg[NegFlag] = Res[15];
53             end
54             InsSLT: begin
55              if (OpA > OpB)
56               Res = 16'd1;
57              else
58               Res = 16'd0;
59              /* Zero check */
60              if (Res == 0)
61               FlagReg[ZeroFlag] = 1'b1;
62              else
63               FlagReg[ZeroFlag] = 1'b0;
64              /* Overflow check */
65              FlagReg[OverflowFlag] = (OpA[15] & OpB[15] & ~Res[15]) | (~OpA[
                    15] & ~OpB[15] & Res[15]);
66              /* Negative check */
67              FlagReg[NegFlag] = Res[15];
68             end
69             InsAND: begin
70              Res = OpA & OpB;
71              /* Zero check */
72              if (Res == 0)
73               FlagReg[ZeroFlag] = 1'b1;
74              else
75               FlagReg[ZeroFlag] = 1'b0;
76              /* Overflow check */
77              FlagReg[OverflowFlag] = (OpA[15] & OpB[15] & ~Res[15]) | (~OpA[
                    15] & ~OpB[15] & Res[15]);
78              /* Negative check */
79              FlagReg[NegFlag] = Res[15];
80             end
81             InsOR: begin
82              Res = OpA | OpB;
83              /* Zero check */
84              if (Res == 0)
85               FlagReg[ZeroFlag] = 1'b1;
86              else
87               FlagReg[ZeroFlag] = 1'b0;
88              /* Overflow check */
89              FlagReg[OverflowFlag] = (OpA[15] & OpB[15] & ~Res[15]) | (~OpA[
                    15] & ~OpB[15] & Res[15]);
90              /* Negative check */
91              FlagReg[NegFlag] = Res[15];
92             end
93             InsXOR: begin
94              Res = OpA ^ OpB;
95              /* Zero check */
96              if (Res == 0)
97               FlagReg[ZeroFlag] = 1'b1;
98              else
99               FlagReg[ZeroFlag] = 1'b0;
100             /* Overflow check */
101             FlagReg[OverflowFlag] = (OpA[15] & OpB[15] & ~Res[15]) | (~OpA[
                    15] & ~OpB[15] & Res[15]);
102             /* Negative check */
103             FlagReg[NegFlag] = Res[15];
104            end
```

```
105             endcase
106           end
107         end
108 endmodule
```

Referências