پروژه درس طراحی سیستم های دیجیتال

باربد شهرآبادی – 401106125

سوال اول میانترم (STACK_BASED_ALU)

در ابتدا به طراحی استک میپردازیم.

ورودی های مورد نیاز استک شامل موارد زیر میباشند:

Input data -> n bits

CLK, RST -> 1 bit

Opcode -> 2bits

همچنین خروجی های ماژول هم شامل موارد زیر میباشد:

Output data -> n bits

Overflow -> 1 bit

Stack_pointer -> 5 bits

چون که استک را بصورت پیش فرض دارای 32 خانه فرض کرده ایم، stack_pointer 5 بیتی میباشد.

```
module STACK_BASED_ALU #(parameter n = 32, parameter STACK_SIZE = 32, parameter POINTER_SIZE = 5) (
    input clk,
    input rst,
    input [2:0] opcode,
    input signed [n-1:0] input_data,
    output reg signed [n-1:0] output_data,
    output reg overflow,
    output reg [POINTER_SIZE-1:0] sp
);
```

برای هر یک از opcode ها استک باید بصورت زیر عمل کند:

$$opcode = 100 : output = stack[sp-1] + stack[sp-2]$$

$$opcode = 101 : output = stack[sp-1] \times stack[sp-2]$$

$$opcode = 110 \; and \; sp < 32 : stack[sp] = input, sp = sp + 1$$

اگر استک پر نبود و خواستم push کنیم، مقدار ورودی را در خانه ای از استک که اشاره گر به آن اشاره میکند ریخته و اشاره گر را اضافه میکنم.

$$opcode = 100\ and\ sp > 0 : output = stack[sp-1], sp = sp-1, stack[sp-1] = 0$$

اگر استک خالی نبود و خواستیم pop کنیم، خانه ای از استک که اشاره گر به آن اشاره میکند را در خروجی قرار داده و سپس نشان گر را یکی کم میکنم. همچنین خانه ای که استک به آن اشاره میکرد را برابر صفر قرار میدهیم.

(توجه داشته باشیم که stack[sp-1] نوشته ایم چون که مقدار sp همیشه به اولین خانه خالی استک اشاره میکند در نتیجه آخرین مقدار پر استک در خانه sp-1 قرار دارد)

همچنین برای بررسی سرریز به روش زیر عمل میکنیم:

چون که اعداد علامت دار هستند، پس بیت اول آن ها علامت آن هارا مشخص میکند. در نتیجه از روی بیت علامت به راحتی میتوان سرریز شدن یا نشدن جمع را مشخص کرد:

در صورتی که هر دو عدد منفی بودند و خروجی مثبت، سرریز داریم.

در صورتی که هر دو عدد مثبت بودند و خروجی منفی، سرریز داریم.

اگر دو عدد دارای علامت های مختلف باشند، هیچگاه سرریز نخواهیم داشت.

در نتیجه عبارت منطقی سرریز برای جمع بصورت زیر میباشد:

$$overflow = (signA\ \&\ signB\ \&\ s\sim ignResult)\ |\ (\sim signA\ \&\ \sim signB\ \&\ signResult)$$

```
if(opcode == 3'b100) begin
    if (sp >= 2) begin
        first_operand = stack[sp-1];
        second_operand = stack[sp-2];

        output_data = first_operand + second_operand;
        overflow = (~first_operand[n-1] & ~second_operand[n-1] & output_data[n-1]) | (first_operand[n-1] & second_operand[n-1] & ~output_data[n-1]);
    end
```

برای بررسی سرریز در ضرب هم باید چک کنیم که آیا جواب ضرب در n بیت جا شده است یا نه. پس در ابتدا حاصل ضرب را در یک رجیستر 2n+1 بیتی ریخته و سپس n بیت اول آن را در خروجی قرار میدهیم. سپس بررسی میکنیم که آیا محتویات رجیستر 2n+1 بیتی با خروجی برابر میباشد یا خیر:

```
else if(opcode == 3'b101) begin
    if (sp >= 2) begin
        first_operand = stack[sp-1];
        second_operand = stack[sp-2];

        full_product = first_operand * second_operand;
        output_data = full_product[n-1:0];
        overflow = $signed(full_product != output_data);
    end
    else
        overflow <= 0;
end
```

حال که ماژول STACK_BASED_ALU را تکمیل کردیم برای آن یک تست بنچ مینویسیم.

در تست بنچ 4 تا ماژول STACK_BASED_ALU با طول های 4 – 8 – 16 و 32 را instantiate میکنیم.

در ابتدا 4 بار اولفلو هر کدام را حساب میکنیم:

```
# 7 + 4: output 4-bit  -5, overflow 4-bit 1
# 7 + 4: output 8-bit   11, overflow 8-bit 0
# 7 + 4: output 16-bit     11, overflow 16-bit 0
# 7 + 4: output 32-bit          11, overflow 32-bit 0
# 64 * 3: output 8-bit  -64, overflow 8-bit 1
# 64 * 3: output 16-bit    192, overflow 16-bit 0
# 64 * 3: output 32-bit         192, overflow 32-bit 0
# 32767 + 1: output 16-bit -32768, overflow 16-bit 1
# 32767 + 1: output 32-bit       32768, overflow 32-bit 0
# 1000000000 * 5: output 32-bit   705032704, overflow 32-bit 1
```

اول مقدار 4+7 را در هر 4 ماژول زدیم. همانطور که مشاده میشود چون این مقدار در 4 بیت علامت دار جا نمیشود، ماژول اول اورفلو میکند ولی بقیه به درستی مقدار را محاسبه میکنند.

عبارات 3*64، 1+32767، 5*1000000000 را هم به همین صورت در ماژول ها محاسبه کرده و اورفلو کردن آن هارا بررسی میکنیم. همینطور که در شکل ها مشخص است، مقادیر به درستی محاسبه شده و اورفلو ها هم به درستی کار میکنند. سپس برای اطمینان بیشتر از درستی عملیات های منطقی، چند عملیات دیگر را هم در ماژول 32 بیتی انجام میدهیم:

```
# 10654 * 25434: output 32-bit   270973836, overflow 32-bit 0
# 22 + 36: output 32-bit          58, overflow 32-bit 0
# 8 * 4: output 32-bit           32, overflow 32-bit 0
# 32749 + 125900: output 32-bit      158649, overflow 32-bit 0
# 56714 * 1234892: output 32-bit  1316188152, overflow 32-bit 1
```

حال سراغ طراحی ماژول ماشین حساب نهایی میرویم. قدم اول این است که عبارت میانوندی ورودی را به یک عبارت پسوندی تبدیل کنیم.

برای اینکار از الگوریتم Shunting Yard استفاده میکنیم که نیازمند یک استک میباشد. در ابتدا یک ماژول STACK_BASED_ALU را با نام stack، Instantiate میکنیم. از این ماژول نمیخواهیم برای ضرب و جمع استفاده کنیم و تنها برای push و pop کردن از آن استفاده خواهیم کرد.

```
//only used for pushing and popping
STACK_BASED_ALU #(n, STACK_SIZE, LOG_STACK_SIZE) stack (
    .input_data(stackInput[n-1:0]),
    .clk(CLK),
    .rst(rst),
    .opcode(stackOp),
    .output_data(stackOutput[n-1:0]),
    .sp(sp)
);
```

پس از تبدیل کردن عبارت میانوندی ورودی به عبارت پسوندی، حال باید مقادیر را در ماشین حساب استکی خود وارد کرده و نتیجه را محاسبه کنیم. پس یک ماژول STACK_BASED_ALU دیگر را instantiate میکنیم.

```
STACK_BASED_ALU #(LEN_OUTPUT, STACK_SIZE, LOG_STACK_SIZE) calculator (
    .input_data(calculator_input[LEN_OUTPUT-1:0]),
    .output_data(calculator_output[LEN_OUTPUT-1:0]),
    .opcode(calculatorOp),
    .clk(CLK),
    .rst(rst)
);
```

از ابتدای عبارت پسوندی ای که در مرحله قبل ساختیم شروع میکنم. اگر به کارکاتر عددی و یا منفی رسیدیم، متوجه میشویم که در حال خواندن عدد هستیم و تا زمانی که به اسپیس نرسیدیم، عدد را محاسبه میکنیم. هر جایی که به یک رقم رسیدیم، عدد فعلی را ضرب در 10 کرده و با کاراکتر جدید جمع میکنیم:

$$number = number \times 10 + (char - '0')$$

و هرگاه که به اسپیس رسیدیم یعنی خواندن عدد فعلی تمام شده و کاراکتر بعدی یا جمع و ضرب، یا یک عدد دیگر خواهد بود. قبل اینکه به پردازش کاراکتر بعدی بپردازیم، اگر اولین کاراکتر رشته عدد فعلی علامت – بود، عدد به دست آمده را ضرب در منفی یک میکنیم و در نهایت عدد به دست آمده را در ماشین حساب push میکنیم.

```
is_valid_character(postfixExpression[8*i-1 -: 8], valid);

if (valid) begin
    isNeg = postfixExpression[8*i-1 -: 8] == "-";
    if(isNeg)
        i = i + 1;

    for (j = 0; j < 1; j = j + 1) begin
        calculator_input = calculator_input * 10 + (postfixExpression[8*i-1 -: 8] - "0");
        i = i + 1;
        if (postfixExpression[8*i-1 -: 8] == " ") begin
            j=0;
        end
        else begin j=-1; end
    end
    i = i - 1;

    if(isNeg)
        calculator_input = -calculator_input;

    calculatorOp = 3'b110; #1; calculatorClk = 1; #1;calculatorClk = 0;
    calculator_input = 0;
```

برای بررسی اینکه آیا یک کاراکتر رقم (و یا علامت منفی) هست هم از تابع زیر استفاده میکنیم:

```
task is_valid_character(input [7:0] char, output reg result);
begin
    if ((char >= "0" && char <= "9") || char == "-") begin
        result = 1;
    end else begin
        result = 0;
    end
end
endtask
```

هر گاه هم که به کاراکتر جمع یا ضرب رسیدیم، opcode ماشین حساب را با توجه به آن ست کرده و عملیات را انجام میدهیم. فقط توجه داشته باشیم که چون ماشین حسابی که طراحی کردیم پس از انجام عملیات خودش حاصل را داخل ماشین حساب Push نکرده و operand ها را هم pop نمیکند، باید دستی این کار را بکنیم.

```
else if (postfixExpression[8*i-1 -: 8] != " " && postfixExpression[8*i-1 -: 8] != 0) begin
    if(postfixExpression[8*i-1 -:8] == "*")
        calculatorOp = 3'b101;
    else
        calculatorOp = 3'b100;

    //do operation
    #1; calculatorClk = 1; #1; calculatorClk = 0;

    calculator_input = calculator_output;

    //pop operands
    calculatorOp = 3'b111;  #1; calculatorClk = 1; #1; calculatorClk = 0; calculatorOp = 3'b111;  #1; calculatorClk = 1; #1; calculatorClk = 0;

    //push last result
    calculatorOp = 3'b110; #1; calculatorClk = 1; #1; calculatorClk = 0;
    calculator_input = 0;
end
```

در نهایت هم پس از پرداش رشته پسوندی، تنها عدد باقی مانده در ماشین حساب استکی که برابر با حاصل عبارت است را pop کرده و در خروجی قرار میدهیم.

حال یک تست بنچ برای ماشین حساب نوشته تا از صحت کاراکرد آن مطمئن شویم:

```
# input:                                    3+5
# postfix expression:                         + 5 3
# output_value is: 8
# input:                                   2*-3
# postfix expression:                        * 3- 2
# output_value is: -6
# input:                                   -4+6
# postfix expression:                        + 6 4-
# output_value is: 2
# input:                              1+2*3+4*5+6
# postfix expression:                   + 6+* 5 4+* 3 2 1
# output_value is: 33
# input:                               8*-2+3*4+5
# postfix expression:                     + 5+* 4 3* 2- 8
# output_value is: 1
# input:                                  (3+5)*2
# postfix expression:                           * 2 + 5 3
# output_value is: 16
# input:                         (1+(2*3))*(4+(5*6))
# postfix expression:                  * + * 6 5 4 + * 3 2 1
# output_value is: 238
# input:                         ((((1+2)*3)+4)*5)+6
# postfix expression:                  + 6 * 5 + 4 * 3 + 2 1
```

و در نهایت مثال خود سوال:

```
# input:                    2*3+(10+4+3)*(-20)+(6+5)
# postfix expression:             + + 5 6+*  02- + 3+ 4 01* 3 2
# output_value is: -323
```

همانطور که از شکل های معلوم است عبارت پسوندی و پاسخ نهایی به درسی محاسبه میشود.

تمامی فایل های پروژه پیوست شده اند.