AGH University of Science and Technology
Cracow
Department of Electronics

Custom system design in FPGA laboratory

Tutorial 6

Multi-cycle functional elements. Newton-Rapson method for division

Author: Paweł Russek

ver. 2022.04.25

History of changes

1. Introduction

1.1. Objectives

The main goal of this tutorial is to introduce the use of the multi-cycle functional elements in custom computing system design. The tutorial extends the concept of the custom system design, presented in Tutorial 1: "The Sequential cordic processor for sine&cosine calculations" of this course. In Tutorial 1, it was assumed, that the cordic functional elements (accumulators, shifters, etc.) perform operations in one clock cycle. However, this assumption can be an obstacle when complex functional elements (multipliers, dividers) are used. Performing the complex operation in one clock cycle affects system speed, because long combinatorial paths of the complex operators mark the critical register-to-register signal delay time and consequently reduce the maximum clock frequency of the system.

Multi-cycle functional elements are the solution to the problem. The internal operation of the operator is split into phases and it is assumed that the operation of such element takes several clock cycle. In opposite to the combinatorial single-cycle operators, multi-cycle ones are sequential designs – to some extent they are custom processor itself.

Unfortunately, the HDL designs that use multi-cycle operators become somewhat more complex. In the tutorial we will propose a design style to implement multi-cyle functional elements in HDL-based projects.

The case study for the problem of multi-cycling is Newton-Rapson algorithm for the division. We will explain and implement the algorithm in System Verilog. The mult-cycle functional element for multiplication will be introduced on the way to final Newton-Rapson processor architecture.

1.2. Prerequisites

Digital design background will be necessary for this tutorial. We expect the student to know principles of operation of synchronious sequential circuits. Also, the understanding of basic concepts that are foundation of any HDL will be necessary for this tutorial. The prior knowledge of Verilog/System Verilog will be helpful but not absolutely necessary. Vivado design tool will be use to perform simulations of the design files.

2. Single-cycle multiplication

***Reminder**. Combinatorial digital systems implements logic gates only and no registers are used in the design.*

When multiplication is implemented as combinatorial logic, it can be performed in a single clock cycle in the custom procesor. However, multipliction of integers is a complex operation, and it usually introduce long delay paths if implemented in a combinatorial way.

Fortunately, the FPGA designers can take advantege of combinatorial multiplication because

FPGAs features dedicated multiplier blocks. The maximum delay of such blocks is comparable to the maximum FPGA clock cycle; therefore, they do not affect overal system performance. For example, Xilinx's 7-series FPGAs feature 25×18 – bit multiplier blocks.

The example code of simple 12×12 - bit multiplier is given below.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Filename: multiply12.sv
// Module Name: mul12
//////////////////////////////////////////////////////////////////////////////////

module mul12(
    input logic[11:0] input0,
    input logic[11:0] input1,
    output logic[23:0] output0
);

always_comb begin
    output0 <= input0 * input1;
end

endmodule
```

You can use single-cycle multipliers in your design if the size of the arguments does not exceed the bit-size of dedicated multipiers inputs.

Exercise 1.1

Perform simulation of the *mul12* in Vivado using testbench given below. As you will see, the testbench detect multiplication error. Identify the problem in a waveform window and correct the testbench to make it run smoothly.

Also, please note, there is no delay in multipier output response to the input arguments change.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Filename: multiply12_tb.sv
// Module Name: mul12_tb
//////////////////////////////////////////////////////////////////////////////////

module mul12_tb();

logic clk; // Give simulation a tick. The module does not need this
logic [12:0] input0, input1;
logic [23:0] output0;

// Instantiate the module
mul12  UUT  ( .input0, .input1, .output0 );

initial begin
    input0 = 12'hff0;
    input1 = 12'hff0;
end

always
begin
    clk = 1'b0;
    #5;  for 5 * timescale = 5 ns
    clk = 1'b1;
    #5; // high for 5 * timescale = 5 ns
```

```
end

always@(posedge clk) begin
    if ( input0 * input1 != output0) begin
        $display("Multiplication error. Stop");
        $stop;
    end
    input0 = input0 + 1;
    input1 = input1 + 1;
end

endmodule
```

Exercise 1.2

Synthetise *mul12* module and run “*Post synthesis timing simulation”.*

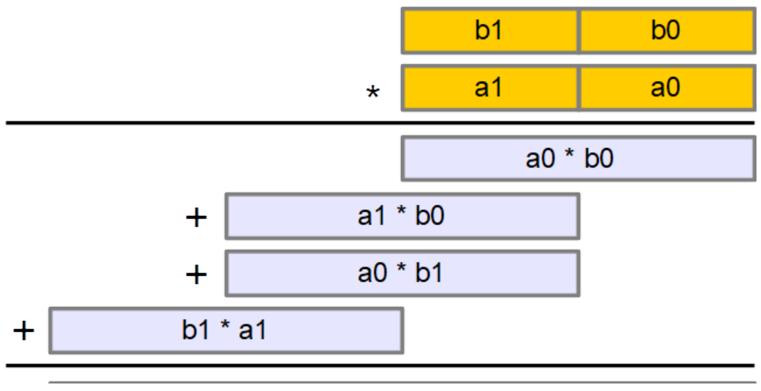
Watch the waveform again. Comment the output signal timing with respect to the input signals

3. Paper-and-pencil multiplication

As it was highlighted in the previous section, the designer can implement multiplication without any performance loss if he/she keeps the argument size within the limits of FPGA's dedicated multiplication blocks. When the necessary multiplication range is bigger, the paper-and-pencil method can be used.

Note. In the code given in “mul12.sv”, the designer is allowed do declare in HDL arguments which size exceeds the size of the dedicated multiplier. The code will be synthesized and implemented anyway, but the performance will be severely affected. Here, we will manually split long multiplication into shorter ones to fit the dedicated blocks. The reason is we will later deploy our code into the multi-cycle element in the tutorial.

Paper-and-pencil method is widely used to perform manual multiplication of long digits. The concept can be easily adopted as multiplication algorithm that allow split long registers into shorter ones. The method is depicted in Figure 1.



Here in the Figure 1, the registers *a0, a1, b0, b1* are half the size of input aguments. Conseguantly, reduced size multiplication can be performed. Partial products are summed up to produce the final result.

The paper-and pencil algorithm is also given as System Verilog code below.

```
 `timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Module Name: mul24_comb
// File: mul24_comb.sv
//////////////////////////////////////////////////////////////////////////////////
//

module mul24_comb(
    input logic[23:0] input0,
    input logic[23:0] input1,
    output logic[47:0] output0
);

//Auxiliary signals
logic [11:0] a0, a1, b0, b1;
assign a0 = input0[11:0];
assign a1 = input0[23:12];
assign b0 = input1[11:0];
assign b1 = input1[23:12];

logic [23:0] tmp; // Tmp. result of fractional mul.

logic [47:0] result;
assign output0 = result;

always_comb begin
    //
    result = 48'h0;
    tmp = a0 * b0;
    // P0
    result[23:0] = tmp;
    tmp = a1 * b0;
    // P1
    result[35:12] = result[35:12] + tmp;
    tmp = a0 * b1;
    // P2
    result[36:12] = result[35:12] + tmp;
    tmp = a1 * b1;
    // DONE
    result[47:24] = result[47:24] + tmp;
end
endmodule
```

Exercise 3.1

Based on the testbench from Section 2, prepare the testbench code for the above *mul24_com* and perform its simuation.

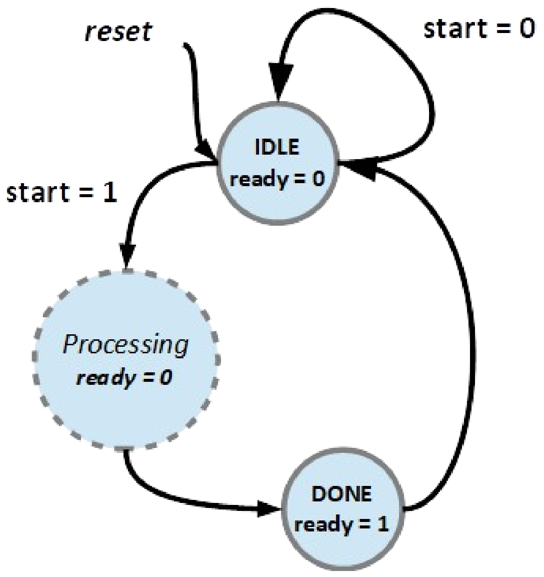
4. Handshaking for the multi-cycle functional elements

Many different handshaking methods are possible for the multi-cycle devices. Here, we will stick with the concept already used for cordic processor (see: Tutorial 1 of this course). Accordingly, multi-cycle elements need the clock signal, two handshaking signals: and *ready,* input and output data.

```
module multi_cycle_fun(
    input logic clk,
    input logic ,
    output logic ready,
    // inputs
    input logic [bit_width0-1:0] input0,
    input logic [bit_width1-1:0] input1,
    ...
    // outputs
    output logic [bit_width1-1:0] output0,
    ...
);
```

When active, the input signal indicates that input data is valid. If the multifunctional unit is idling, it s internal processing immediately when is asserted. When processing is finish, *ready* output is asserted.

The method is presented in the Figure 2.



The corresponding Verilog code is given below.

```
//FSM template for handshaking signals
enum {IDLE=0, P0, P1, P2, DONE} state;

    always_ff @(posedge clk) begin: fsm
        case(state)
            IDLE: begin
                ready <= 1'b0;
                state <= P0;
                // other alg. related ops.
            end
            P0: begin
                // other alg. related ops.
                state <= P1;
            end
            P1: begin
                // alg. related ops.
                // other alg. related ops.
            end

              .
              .
              .
.
            Pn: begin
                // alg. related ops.
               state <= DONE;
            end
            DONE: begin
```

```
                // other alg. related ops.ready <= 1'b1;
                ready <= 1'b1;
               state <= IDLE;
            end
        endcase
    end: fsm
```

5. Multi-cycled miltiplier

Now, as we have had prepared the template for multi-cycle elements, we can prepare the multi-cycled paper-and-pencil multiplier.

Below, you will find the muli-cycle multiplier module and its testbench.

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Module Name: mul24_inf
// File Namee: mul24_inf.sv
////////////////////////////////////////////////////////////////////////////////

module mul24_infer(
    input logic clk,
    input logic ,
    output logic ready,
    input logic[23:0] input0,
    input logic[23:0] input1,
    output logic[47:0] output0
);

//Auxiliary signals
logic [11:0] a0, a1, b0, b1; // a0 = low(input0); a1 = high(input0); b0 =
low(input2); ...
assign a0 = input0[11:0];
assign a1 = input0[23:12];
assign b0 = input1[11:0];
assign b1 = input1[23:12];
logic [23:0] tmp; // Tmp. result of fractional mul.
logic [47:0] result;

assign output0 = result;

//FSM
enum {IDLE=0, P0, P1, P2, DONE} state;

    always_ff @(posedge clk) begin: fsm
        case(state)
            IDLE: begin
                ready <= 1'b0;
                if ( == 1'b0) begin
                    state <= IDLE;
                end else begin
                    result <= 48'h0;
                    tmp <= a0 * b0;
                    state <= P0;
                end
            end
            P0: begin
                result[23:0] <= tmp;
                tmp <= a1 * b0;
                state <= P1;
            end
            P1: begin
                result[35:12] <= result[35:12] + tmp;
                tmp <= a0 * b1;
                state <= P2;
            end
            P2: begin
                result[36:12] <= ???;
                tmp <= ???;
                state <= DONE;
            end
            DONE: begin
                result[47:24] <= result[47:24] + tmp;
                ready <= 1'b1;
                state <= IDLE;
            end
        endcase
    end: fsm
endmodule
```

Exercise 5.1

Replace '*???*' symbol with the correct operation statement and simulate *mul24_inf* in Vivado. User the testbench code given below.

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Module Name: mul24_inf_tb
// File Name: mul24_inf_tb.sv
////////////////////////////////////////////////////////////////////////////////

module mul24_inf_tb();

logic clk, start, ready;
logic [23:0] input0, input1;
logic [47:0] output0;

mul24_inf  UUT  ( .clk, .start, .ready, .input0, .input1, .output0 );

initial begin
    input0 <= 24'h1;
    input1 <= 24'h1;
    start <= 1'b1;
end

always
begin
    clk = 1'b0;
    #5; // low for 5 * timescale = 5 ns
    clk = 1'b1;
    #5; // high for 5 * timescale = 5 ns
end

always@(posedge clk) begin
    if ( ready == 1'b1 ) begin
        if ( input0 * input1 != output0) begin
            $display("Multiplication error. Stop");
            $stop;
        end
        input0 <= input0 + 1;
        input1 <= input1 + 1;
    end
    start <= ready; //self handshaking
end

endmodule
```

6. Inference vs Instantiation

Inference and instantiation are two techniques used in HDL design to implement functionality. In instatntiation, the designer implicitly puts a design module into his/her project. The multiplier instantiation example is:

```
mul MUL_INST (.a, .b, .c);
```

In opposite, inference is done automatically by the HDL synthetizer. The sythetizer implements a necessary module when designer puts corresponding expression into the code. For example, the code

```
c = a * b;
```

results in multiplier inference.

The module of *mul24_inf* presented in Section 6 uses inference to implement multiplication. However, inference does not allow designed to use functional elements that are not impelemnted as operators in HDL. In the case the operators are not avaliable for a given functional, element instantiation must be used.

Instantantiation makes the HDL code less behavioral and therefore less readable but in some cases it is necessary to incorporate this technique.

Please, consider *mul24_ins* module source code, given below, as an example. The code functionality is the same as in the case of *mul24_inf* but it instantiate multipier explicitly. The most important changes in respect to *mul24_ins* are highlighted in frames. Basically, na additional *always* block statement was added to control the instantiated multipier inputs.

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//
// Module Name: mul24_ins
// File Name: mul24_ins.sv
////////////////////////////////////////////////////////////////////////////////
//
module mul24_ins(
    input logic clk,
    input logic ,
    output logic ready,
    input logic[23:0] input0,
    input logic[23:0] input1,
    output logic[47:0] output0
);
// multiplier instantiation
logic [11:0] mul1_input0, mul1_input1;
logic [23:0] mul1_output0;
mul12  mul12_1 (  .input0(mul1_input0),
                  .input1(mul1_input1),
                  .output0(mul1_output0));

//Auxiliary signals
logic [11:0] a0, a1, b0, b1; // a0 = low(input0); a1 = high(input0); b0 =
low(input2); ...
assign a0 = input0[11:0];
assign a1 = input0[23:12];
assign b0 = input1[11:0];
assign b1 = input1[23:12];
logic [23:0] tmp; // Tmp. result of fractional mul.
logic [47:0] result;

assign output0 = result;

//FSM
enum {IDLE=0, P0, P1, P2, DONE} state;

    always_ff @(posedge clk) begin: fsm
        case(state)
            IDLE: begin
                ready <= 1'b0;
                if ( start == 1'b0) begin
                    state <= IDLE;
                end else begin
                    result <= 48'h0;
                    tmp <= mul1_output0; // tmp <= a0 * b0;
                    state <= P0;
                end
            end
            P0: begin
                result[23:0] <= tmp;
                tmp <= mul1_output0; // tmp <= a1 * b0;
                state <= P1;
            end
            P1: begin
                result[35:12] <= result[35:12] + tmp;
                tmp <= mul1_output0; // tmp <= a0 * b1;
                state <= P2;
            end
            P2: begin
                result[36:12] <= result[35:12] + tmp;
                tmp <= mul1_output0; // tmp <= a1 * b1;
                state <= DONE;
            end
            DONE: begin
                result[47:24] <= result[47:24] + tmp;
                ready <= 1'b1;
                state <= IDLE;

            end
        endcase
    end: fsm

// Functional unit inputs assignments
    always_comb begin
    case(state)
            IDLE: begin
                mul1_input0 = a0; mul1_input1 = b0; // tmp <= a0 * b0;
            end
            P0: begin
                mul1_input0 = a1; mul1_input1 = b0; // tmp <= a1 * b0;
            end
            P1: begin
                mul1_input0 = a0; mul1_input1 = b1; // tmp <= a0 * b1;
            end
            P2: begin
                mul1_input0 = a1; mul1_input1 = b1;  // tmp <= a0 * b1;
            end
        endcase;
    end
```

```
endmodule
```

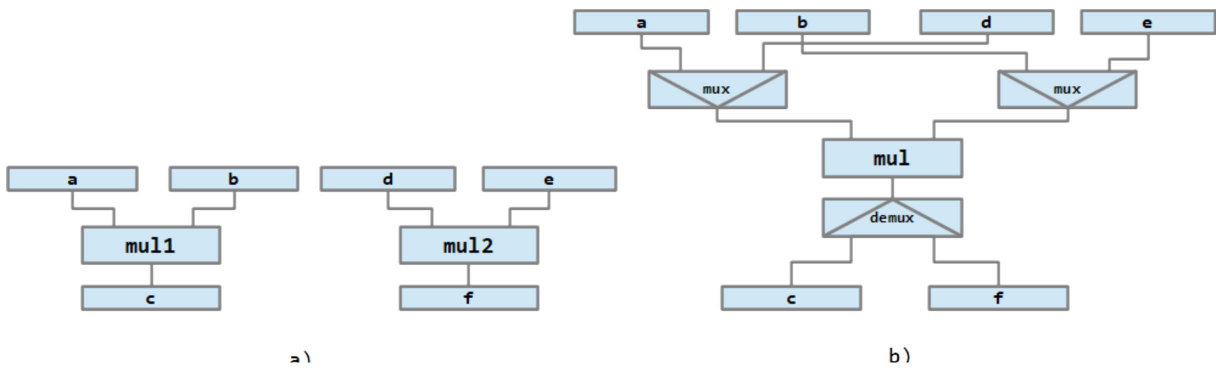
Exercise 6.1 (can be skipped)

Simulate *mul24_ins* using the testbench from Section 5. Verify corectness of the module operation.

Now, let's consider the snippet of code given below.

```
c = a * b;
f = d * e;
```

It can be resolved twofold in hadrdware (Fig. 3)



The method in Figure 3a uses two multipliers, in contract to the method in Figure 3b that needs one multiplier. However, the second method requires multipliers. The method in Figure 3a is always faster, as it reduces the number of connection. However, the method in Figure 3b can be less resource-demanding if the cost of the multiplier implementation is very high.

In general, if the cost of the functional element is high and the design size is constrained, the second method (Fig, 3b) is prefered. However, if speed is constrained the functional element replication like the first method (Fig. 3a) should be considered.

Intuitively, it seems that the instantiation of the multiplier in *mul24_ins* leads to the utilization of one multiplier block in FPGA. Similarly, one can assume that the synthesis infers four multiplier blocks in *mul24_inf*. However, synthesis tools for HLD use sophisticated algorithms to optimize the design. They can replicate instantiated blocks to improve design performance if free hardware resources (FPGA multiplier in this case) are available.

Exercise 6.2

Compare hardware resources necessary to synthetize *mul24_ins* and *mul24_inf*. Perform synthesis in Vivado and read synthesis report file.

For synthesis, set desired module as **Top** design module, and **Run Synthesis**.

The synthesis utilization raport you will find in **Project Summary** window.

7. Newton-Rapson division algorithm

The division $\frac{a}{b}$ can be seen as multiplication $a*\frac{1}{b}$. Therefore, the algorithm presented here is an algorithm for $f(d)=\frac{1}{d}$ function.

The N-R method given in this tutorial is in fact a linear approximation of the function *f(x)*. However, to make approximation more accurate, The N-R method is applied do the values [0.5. 1] in range. Nevertheles, the algoritm input arguments are not limited to the that range because the algorithm performs normalization (scaling) before the approximation is taken. Before the N-R method is applied input value *d* is divided by two as long as it is out of (0.5, 1] range (Fig. 4).

Figure 4: Normalization and denormalizaton of input argument (http://www.cs.utsa.edu/~wagner/CS3343/newton/division.html)

Newton's Method is used to find successive approximations to the roots of a function.

Figure 5: The N-R method graphical justification. https://www.researchgate.net/publication/329403155_Numerical_Analysis_Course_Chapter_1_Root_Finding

Taken, the approximate root value x_i of the function is know, it is possible to find a better approximation x_{i+1}, if the formula $x_{i+1}=x_i-\frac{f(x_i)}{f'(x_i)}$ is applied.

To find reciprocal of the value *d,* we will apply the N-R method fo find root of the function $f(x)=\frac{1}{x}-d$. Because, the derivative of the function $\frac{1}{x}-d$ is $f'(x)=\frac{-1}{x^2}$, the N-R formula for the reciprocal takes the form $x_{i+1}=x_i*(2-d*x_i)$.

The only problem that remains, is to find initial value x_0 to start iteration. Here, we will use the linear approximation of the $\frac{1}{d}$ function in range [0, 1]:

$$x_0=-1.88235*d+2.82353\ .$$

Putting together, scaling, linear approximation, and the N-R iterations, it is possible to formulate the algoritm.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Design Name:
// Module Name: reciprocal_beh
// File Name: reciprocal_beh.sv
//////////////////////////////////////////////////////////////////////////////////

module en_reciprocal_beh(
    input logic [15:0] input0,   // argument: integer [15:0]
    output logic [4:-19] output0 // result: fixed point [5:19] representation
    );

// Constants
logic [4:-19] A = 24'h0F0F0D; // Fixed point [5:19] representation of 1.88235
logic [4:-19] B = 24'h169696; // Fixed point [5:19] representation of 2,82353
logic [4:-19] HALF = 32'h040000; // Fixed point [4:19] representation of 0.5
logic [4:-19] TWO =  32'h100000; // Fixed point [4:19] representation of 2

// Variables
logic [4:0] scaling;  // Keeps scaling factor
logic [9:-38] mulResult; // Temporary result of multiplication [5:19] * [5;19]
logic [4:-19] scaledVal;
logic [4:-19] approxVal;
logic [4:-19] newVal;

real resultFP; // To display human readable

always_comb begin

    // Here scaledVal = input / 2**19 for different data representations
    scaledVal = input0; // IDLE
    // therefore scaling starts from maximum value and goes backwards
    scaling = 19;
    // Scale tmpValue to range [0.5, 1] i.e. [0x080000, 0x040000] in integer
    // In difference to oryginal algorithm we multiply by two in each iteration
    while( scaledVal < HALF ) begin // COMP_AND_SCALE
        scaledVal = scaledVal << 1; // Multiply by two i.e. LSR
        scaling --;
    end

    // Take Linear aproximation x0 = 2.82353 - 1.88235 * d.
    mulResult = scaledVal * A;  // Result is fixpoint [10:38]. MUL_A
    approxVal = mulResult >> 19; // keep [4:19] fxp format
    approxVal = B - approxVal; // SUB_B

    while(1) begin // iterate: x(i+1) = x(i) * ( 2 - x(i)*d )
        mulResult = approxVal * scaledVal; // MUL_SCALED
        newVal = mulResult >> 19; // keep [4:19] fxp format
        newVal = TWO - newVal; // SUB_2
        mulResult = approxVal * newVal; // MUL_NEW
        newVal =  mulResult >> 19; // keep [4:19] fxp format
        if( approxVal == newVal ) begin break; end // CHECK_EQ
        approxVal = newVal; // ASSIGN_NEW
    end

    // Denormalize back to oryginal range
    approxVal = approxVal >> scaling;

    output0 = approxVal;

    // Print result
    $display("Binary result is  = %b", approxVal);
    resultFP = approxVal;
    resultFP = resultFP / 2**19;
    $display("Real value is  = %f", resultFP);

end

endmodule
```

Exercise 7.1

Simulate the *reciprocal_beh* module in Vivado. Use the testbench code given below.

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Module Name: tb_reciprocal_beh
// File Name: tb_reciprocal.sv
////////////////////////////////////////////////////////////////////////////////
//

module tb_reciprocal_beh();

real inputFP, outputFP;
logic [15:0] input0, output0;

en_reciprocal_beh  UUT  ( .input0, .output0 );

initial begin
    input0 = 16'd3;
end

endmodule
```

8. Multi-cycle reciprocal module

The code for the multi-cycle reciprocal module and its testbench is provided in this section. It instantiate and uses the multi-cycle *mul24_ins* module to perform operation.

```
mul24_ins mul24_0(  .clk(clk),
                    .start(mul_start),
                    .ready(mul_ready),
                    .input0(mul_input0),
                    .input1(mul_input1),
                    .output0(mul_output0) );
```

The moudule implements eleven state FSM. The corresponding states are: IDLE=0, COMP_AND_SCALE, MUL_A, SUB_B, MUL_SCALED, SUB_2, MUL_NEW, CHECK_EQ, ASSIGN_NEW, MUL_SCALING, and DONE.

```
//FSM
enum {IDLE=0, COMP_AND_SCALE, ?????, SUB_B, MUL_SCALED, SUB_2, MUL_NEW,
CHECK_EQ, ASSIGN_NEW, MUL_SCALING, DONE} state;
```

The states MUL_A, MUL_SCALED, and MUL_SCALING implement multi-cycle multiplication. Basically, the processor waits for the corresponding multiplication completion in these states. For example

```
            MUL_NEW: begin
                if ( mul_ready == 1'b1 ) begin
                    mulResult <= mul_output0;
                    state <= CHECK_EQ;
                end else begin
                    state <= MUL_NEW; // cont. waiting
                    end
```

As long as *mul_ready* is not active MUL_NEW is re-entered. When multiplication is complete (*mul_ready* is high), the processor takes the next state CHECK_EQ.

The combinatorial process is introduce to swich multiplier input in corresponding states. For example, *scaling* and *approxVal* are multipied in the MUL_SCALING state. The assignement of the multiplier inputs *muk_input0* and *mul_input1* are presented below.

```
   always_comb begin
        case(state)
            .
            .
            .
            MUL_SCALING: begin
                mul_start <= ~mul_ready;
                mul_input0 <= scaling;
                mul_input1 <= approxVal;
            end
            .
            .
            .
        endcase
    end
```

Full System Verilog code for the reciprocal module:

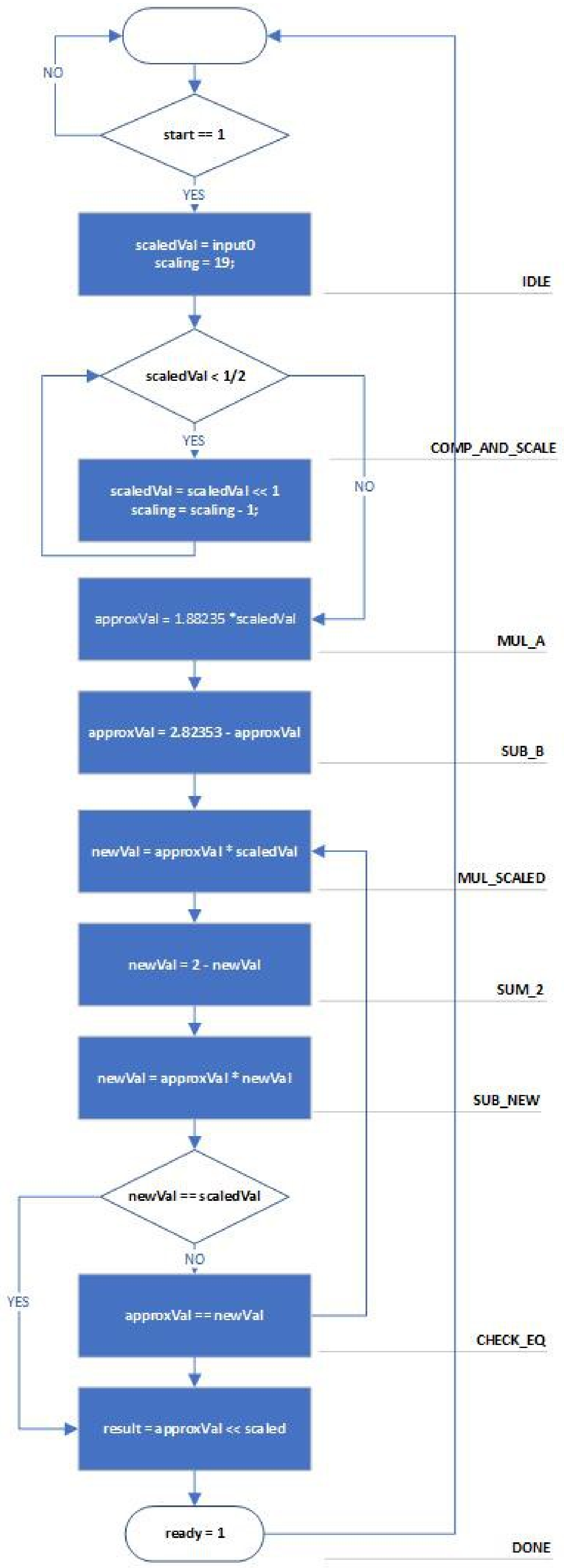
```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Module Name: reciprocal
// File Name: reciprocal.sv
////////////////////////////////////////////////////////////////////////////////

module full_reciprocal(
    input logic clk,
    input logic start,
    output logic ready,
    input logic [15:0] input0,   // Integer [15:0]
    output logic [4:-19] output0 // Fixed point [5:19] representarion
    );

// Constants
logic [4:-19] A = 24'h0F0F0D; // Fixed point [5:19] representation of 1.88235
logic [4:-19] B = 24'h169696; // Fixed point [5:19] representation of 2,82353
logic [4:-19] HALF = 32'h040000; // Fixed point [4:19] representation of 0.5
logic [4:-19] TWO =  32'h100000; // Fixed point [9:38] representation of 2

// Variables
logic [4:0] scaling;  // Keeps scaling factor
logic [9:-38] mulResult; // Temporary result of multiplication [5:19] * [5;19]
logic [4:-19] scaledVal;
logic [4:-19] approxVal;
logic [4:-19] newVal;

//Instantiate multiplier
logic mul_start;
logic mul_ready;
logic[23:0] mul_input0;
logic[23:0] mul_input1;
logic[47:0] mul_output0;
????????? mul24_0(  .clk(clk),
                    .start(mul_start),
                    .ready(mul_ready),
                    .input0(mul_input0),
                    .input1(mul_input1),
                    .output0(mul_output0) );

//FSM
enum {IDLE=0, COMP_AND_SCALE, ?????, SUB_B, MUL_SCALED, SUB_2, MUL_NEW,
CHECK_EQ, ASSIGN_NEW, MUL_SCALING, DONE} state;

    always_ff @(posedge clk) begin: fsm
        case(state)
            IDLE: begin
                ready <= 1'b0;
                if (start == 1'b0) begin
                    state <= IDLE;
                end else begin
                   //Load input value
                   scaledVal <= input0;
                   scaling = 5'd19;
                   state <= COMP_AND_SCALE;
                end
            end
            COMP_AND_SCALE: begin
                if( scaledVal < HALF ) begin
                    scaledVal <= scaledVal << 1;
                    scaling --;
                    state <= COMP_AND_SCALE;
                end else begin
                    state <= MUL_A;
                end
            end
            MUL_A: begin
                if ( mul_ready == 1'b1 ) begin
                    mulResult <= mul_output0;
                    state <= SUB_B;
                end else begin
                    state <= MUL_A; // cont. waiting
                end
            end
            SUB_B: begin
                approxVal <= B - mulResult[4:-19];
                state <= MUL_SCALED;
            end
            MUL_SCALED: begin
                if ( mul_ready == 1'b1 ) begin
                    mulResult <= mul_output0;
                    state <= SUB_2;
                end else begin
                    state <= MUL_SCALED; // cont. waiting
                end
            end
            ?????: begin
                newVal <= TWO - mulResult[4:-19];
                state <= MUL_NEW;
            end
            MUL_NEW: begin
                if ( mul_ready == 1'b1 ) begin
                    mulResult <= mul_output0;
                    state <= ?????;
                end else begin
                    state <= MUL_NEW; // cont. waiting
                end
            end
            CHECK_EQ: begin
                if ( approxVal == mulResult[4:-19] ) begin
                    state <= DONE;
                end else begin
                    state <= ASSIGN_NEW;
                    newVal <= mulResult[4:-19];
                end
            end
            ASSIGN_NEW: begin
                approxVal <= newVal;
                state <= MUL_SCALED;
            end
            DONE: begin
                output0 <= (approxVal >> ?????);
                ready <= 1'b1;
                state <= IDLE;
            end
        endcase
    end: fsm

    always_comb begin
        case(state)
            MUL_A: begin
                mul_start <= ~mul_ready;
                mul_input0 <= A;
                mul_input1 <= scaledVal;
            end
            MUL_SCALING: begin
                mul_start <= ~mul_ready;
                mul_input0 <= scaling;
```

```
                mul_input1 <= approxVal;
            end
            MUL_SCALED: begin
                mul_start <= ~mul_ready;
                mul_input0 <= ?????;
                mul_input1 <= ?????;
            end
            MUL_NEW: begin
                mul_start <= ~mul_ready;
                mul_input0 <= newVal;
                mul_input1 <= approxVal;
            end
            default: begin
                mul_start <= 1'b0;
            end
        endcase
    end

endmodule
```

Exercise 8.1

Replace '???' symbol with the correct operation statement and simulate the *reciprocal module in Vivado*. Use the provided testbench code.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Module Name: tb_reciprocal
// File Name: tb_reciprocal.sv
//////////////////////////////////////////////////////////////////////////////////
//

module tb_reciprocal( );

logic clk;
logic start;
logic ready;
logic [15:0] input0;   // Integer [15:0]
logic [4:-19] output0; // Fixed point [5:19] representarion

int tmp_i, input_i;
real input_r, check_r, output_r;
logic ready_prev; // the state of ready in previous clock

full_reciprocal UUT(.clk, .start, .ready, .input0, .output0);

// Clock generator
always begin
    #5 clk = 1; #5 clk = 0;
end

initial begin
    input_i = 3;
    input_r = input_i;
    check_r = 1 / input_r;
    input0 <= input_i;
    start <= 1'b1;
end

always@( posedge clk ) begin
    start <= ready; //self handshaking
end;

always@( posedge clk ) begin
    if ( ready == 1'b1 /*&& ready_prev == 1'b0*/ ) begin // new value arrived
        input_r = input0;
        check_r = 1 / input_r;
        output_r = output0;
        output_r = output_r / ( 2 ** 19 );
        $display ( "Input is %f. Output is %f. Correct result is %f", input_r,
output_r, check_r);
        input_i ++;
        input0 <= input_i;
    end
end

endmodule
```


NO
start == 1
YES
scaledVal = input0
scaling = 19;
IDLE
scaledVal < 1/2
YES
COMP_AND_SCALE
NO
scaledVal = scaledVal << 1
scaling = scaling - 1;
approxVal = 1.88235 *scaledVal
MUL_A
approxVal = 2.82353 - approxVal
SUB_B
newVal = approxVal * scaledVal
MUL_SCALED
newVal = 2 - newVal
SUM_2
newVal = approxVal * newVal
SUB_NEW
newVal == scaledVal
NO
YES
approxVal == newVal
CHECK_EQ
result = approxVal << scaled
ready = 1
DONE