AGH University of Science and Technology
Cracow
Department of Electronics

Custom system design in FPGA laboratory

Tutorial 4

The pipelined sine/cosine cordic processor

Author: Paweł Russek

ver. 2020.04.01

1. Introduction

1.1. Objectives

The main goal of this tutorial is to present a basic design process of a pipelined custom processor. The cordic algorithm for sin/cos calculation will be used for the purpose. A generic algorithm pipelining will be explained first together with concepts of iteration interval, latency, and throughput of the processor. Later, the role of the loop unrolling will be clarified.

An example RTL description of the pipelined cordic processor will be given as a Verilog template code. Consequently, the student will propose the complete HDL code of the processor architecture. The simulation will conclude this exercise.

1.2. Prerequisites

The principles of the cordic algoritm are presumed to be already known by the student. He/she can acquaint the necessary knowledge by completing the first laboratory tutorial entitled “Sequential cordic processor for sin/cos calculation”.

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 rules that are foundation of any HDL will be necessary for this tutorial. The prior Verilog knowledge will be helpful but not absolutely necessary to complete the tutorial.

2. Algorithm pipelining

Given the algorithm A(n) and the input parameter set $N=\{n_0, n_1, n_2, \dots, n_K\}$, we want to know the results $A(n_0)$, $A(n_1)$, $A(n_2)$, ..., $A(n_K)$ of the algorithm execution for each input parameter. The custom processor for the A algorithm needs L clock cycles to complete. Assuming that many instances of the processor are available, one can plan to execute all instances of the A(n) in a parallel manner (Fig. 1).

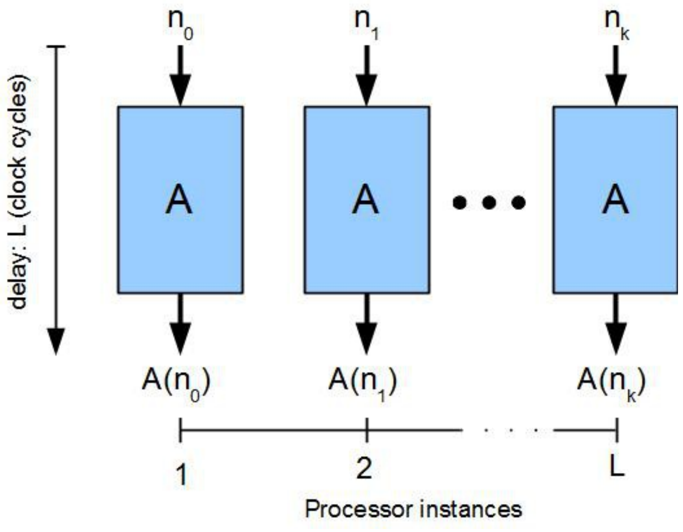


Figure 1

In practice, the usual obstacle of the presented approach of the parallel algorithm execution is the throughput of IO channel. To have all tasks completed in L clock cycles, one needs K input arguments delivered simultaneously. Unfortunately, IO access is sequential, and this is the case for memory, discs drives, network interfaces, *etc.* Therefore, the execution time of the input parameter set is IO throughput bond in practice.

The presented scenario will be referred as **the spatial parallelism** in this tutorial. The alternative solution is the pipeline algorithm execution, and the algorithm pipelining means **temporal parallelism** employed.

For an algorithm to be pipelined, it has to feature a **constant execution time**. In custom processor design, it is convenient to equivalent the execution time to a number of clock cycles (L) required by the processor to complete the algorithm. The execution time L can be also referred as a **delay** or **latency** of the algorithm.

Thus, we can see such the algorithm as a series of L operation (a_0, a_1, a_2, ... , a_{L-1}) performed in consecutive clock cycles. If one design each algorithm step a_k as an dependent hardware element with its own registers and functional elements. it is possible to perform L operations (a_0, a_1, a_2, ... , a_{L-1}) in parallel but each operation corresponds to different input element n_k. When adjacent algorithm stages are connected and results can be passed form stage a_i to a_{i+1}, the processor forms a processing pipeline.

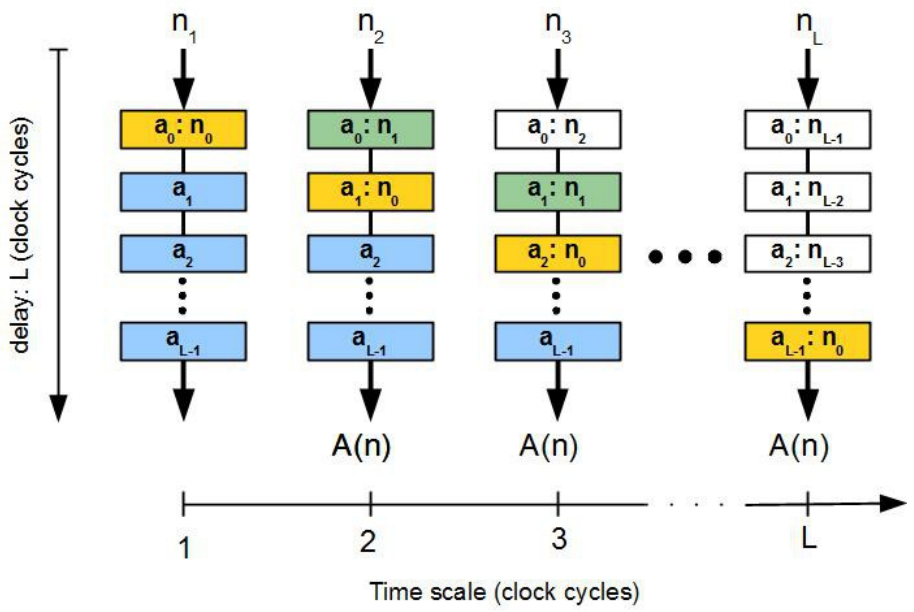


Figure 2

In the proposed scenario, data travels from input towards output and, however, each input still needs L clock cycles to complete the algorithm, the processor takes a new input parameter every clock cycle. New output is ready every clock period too, but there is the L delay between the input and output. Consequently, the total execution time of K elements is L+K clock cycles (Fig. 3).

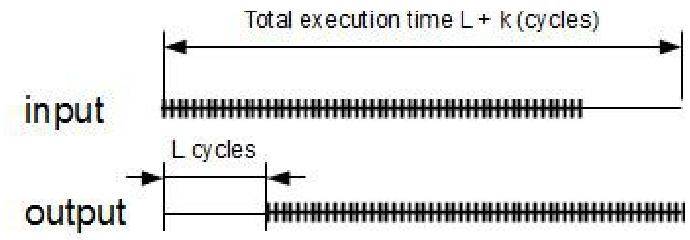


Figure 3

We can assume the processor **throughput** is $\frac{1}{f_{CLK}}$ elements per second if $K \gg L$, where *K* is the number of elements in the set and f_{CLK} is a processor clock frequency.

Note that in general, number of pipelined processor stages need not to be equal the execution time (in clock cycles). So far, we have assumed that each stage performs one operation a_i and takes one clock cycle to complete. However, each stage can be designed to perform several algorithm steps. Stages can perform equal number of steps *e.g.* 2, 3 (Fig. 4a and Fig. 4b respectively) or a different number of steps (Fig. 4c).

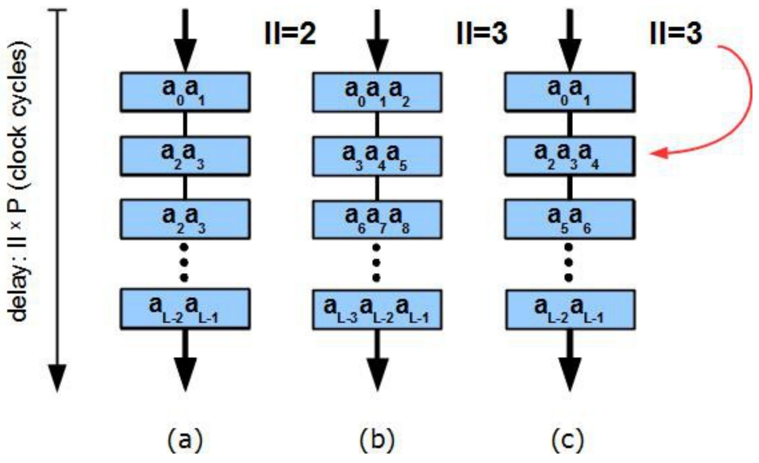


Figure 4

Iteration interval (**II**) is a number of clock cycle necessary for the processor to accept next input element. Thus, the processor takes input parameter every II-th clock cycles. Note, that II equals to the maximum steps of the algorithm (a_i) assigned to a single stage in the pipeline. The latency of such pipelined system is $II * P$.

3. Loop unrolling and memory partitioning

Loops are essential part of each algorithm but they introduce a difficulty for its pipelined execution. They introduce reverse instruction flow to the algorithm *i.e.* the FSMD of the algorithm contains closed routes such that during execution some of the states are run repeatedly.

Before the algorithm is pipelined the loops have to be eliminated from its FSMD. The simple method of loop unrolling is involved in that process. The body of the removed loop is duplicated in the FSMD as many times as the number of loop iterations. Consequently, the number of iterations has to be defined at design time *i.e.* the loop has to be static. Also, the number of iterations should be modest in the practice. The simple loop unrolling is depicted in Figure 5.

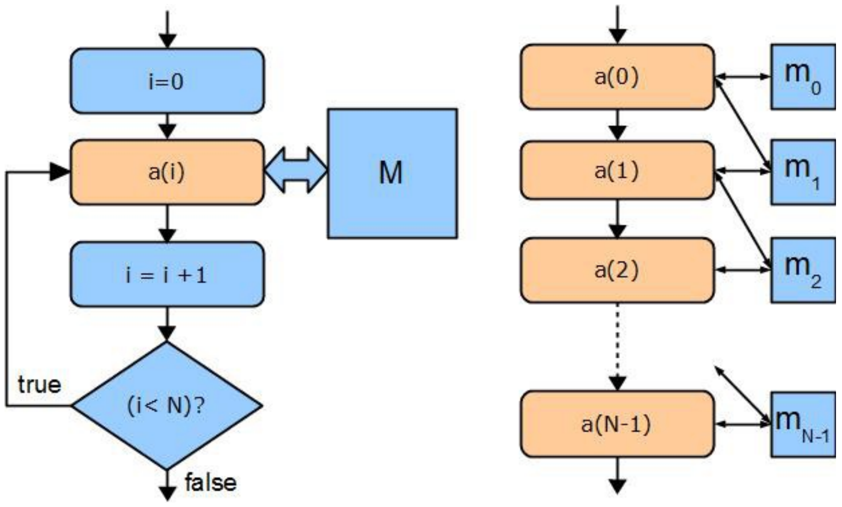


Figure 5

The loop body a(i) is duplicated *N* times after unrolling in the figure.

Every static loop can be unrolled, however, in the pipeline execution of the unrolled FSMD, memory access problem can exist because pipelined stages cannot concurrently access same memory array. To mitigate that problem, the single memory block can be **partitioned**. As an example, the single memory block M was replaced by distributed memory block m_j in Figure 5. Additionally, access of one algorithm stage to the adjacent memory block may helpful to pass copious data from one algorithm stage (iteration) to another. FPGA dual-port RAMs would be helpful to share one memory array between two adjoining processor stages.

4. Sequential cordic algorithm revisited

The design of pipelined Cordic processor we start with sequential algorithm. We have already encountered the sequential code for Cordic execution in Tutorial 1 of this course. Here, we will introduce somehow simplified code that is given as a block diagram in Figure 6.

If compared to the code used in Tutorial 1, we have **dynamic** shift left operation used here. States *S3a* and *S3b* have “sin >>> i” and “cos >>> i” operations. They are dynamic because the *i* is a variable. Such dynamic shift operation ends up as a **barrel shifter** functional unit in a hardware implementation. Barrel shifters are resource hungry and that was the reason we had got rid of dynamic shift in the sequential custom processor implementation and we had shifted repeatedly by one bit (see Tut. 1). Here, as we will see, the case is a different one as dynamic shift will turn static after loop unrolling.

In states S4 – S8, processor performs scaling of the results *i.e.* multiplication by *Kn* (see Tut. 1).

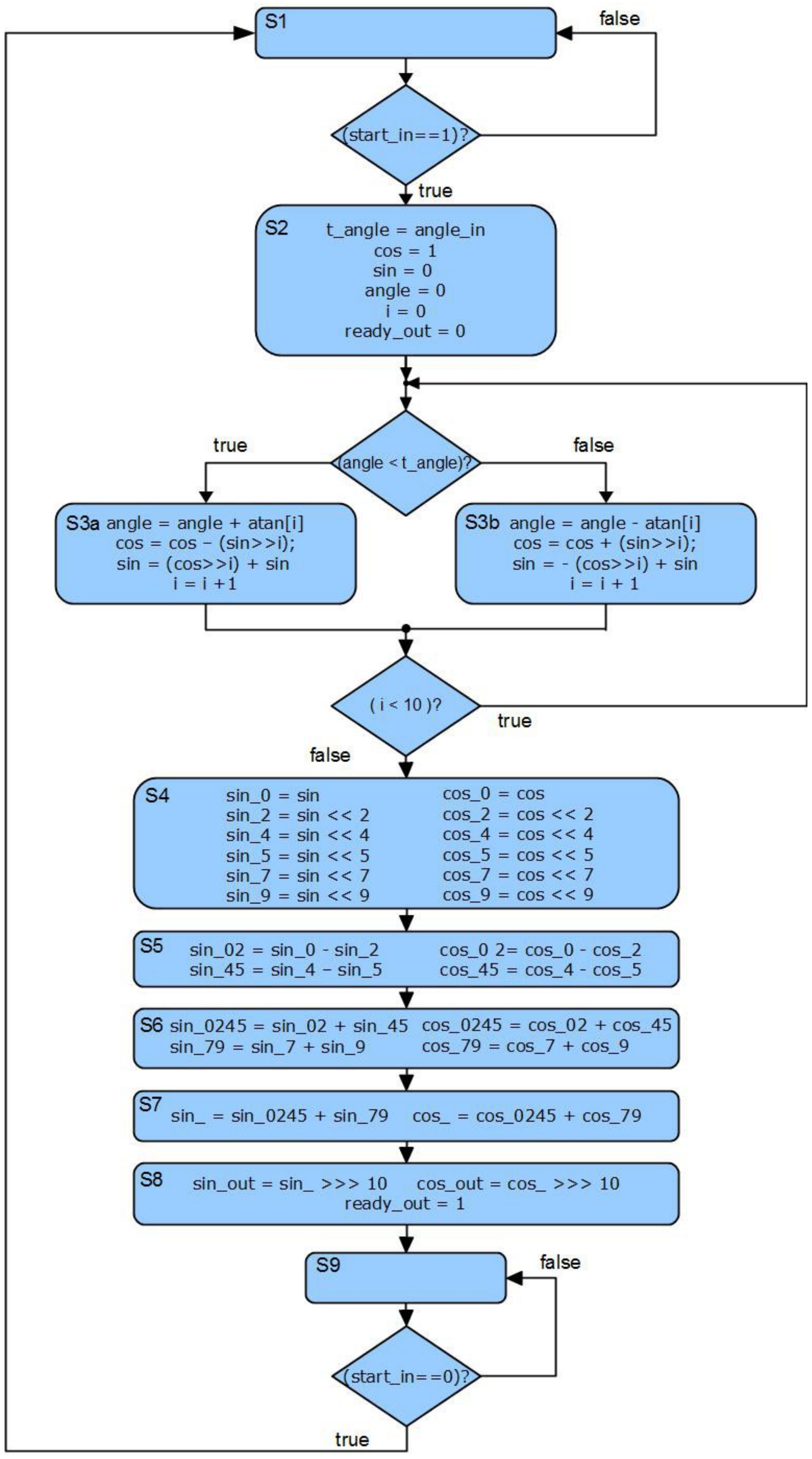

S1
false
(start_in==1)?
true
S2 t_angle = angle_in
cos = 1
sin = 0
angle = 0
i = 0
ready_out = 0
true
(angle < t_angle)?
false
S3a angle = angle + atan[i]
cos = cos – (sin>>i);
sin = (cos>>i) + sin
i = i +1
S3b angle = angle - atan[i]
cos = cos + (sin>>i);
sin = - (cos>>i) + sin
i = i + 1
(i < 10)?
true
false
S4 sin_0 = sin
sin_2 = sin << 2
sin_4 = sin << 4
sin_5 = sin << 5
sin_7 = sin << 7
sin_9 = sin << 9
cos_0 = cos
cos_2 = cos << 2
cos_4 = cos << 4
cos_5 = cos << 5
cos_7 = cos << 7
cos_9 = cos << 9
S5 sin_02 = sin_0 - sin_2
sin_45 = sin_4 – sin_5
cos_0 2= cos_0 - cos_2
cos_45 = cos_4 - cos_5
S6 sin_0245 = sin_02 + sin_45
sin_79 = sin_7 + sin_9
cos_0245 = cos_02 + cos_45
cos_79 = cos_7 + cos_9
S7 sin_ = sin_0245 + sin_79
cos_ = cos_0245 + cos_79
S8 sin_out = sin_ >>> 10
cos_out = cos_ >>> 10
ready_out = 1
S9
false
(start_in==0)?
true

Figure 6

5. Cordic pipeline execution

Cordic pipeline is presented in Figure 7.

Before pipelining can be implemented in a custom processor, the algorithm's loops have to be unrolled. In Figure 7 the cordic loop is replaced by the stages a2 – a14. Consequently, our algorithm has 18 stages.

Each stage directly corresponds to a **stage processor** in the figure. The stage processors are denoted as a1 – a18 in Figure 7. Each sequential step of the algorithm is performed by the stage processor which forms a single pipeline element.

We will design the most straightforward pipeline architecture for cordic, where iteration interval (II) is one. As the cordic processor accepts a new input argument at each clock cycle, every stage processor has to perform its operations in one clock cycle.

5.1. Handshaking

The state *S1* of the sequential algorithm is dropped in our pipeline architecture. The pipeline processor does not require the *start* flag as it expects input data at each clock cycle. Instead clock enable *ce* is necessary to pause pipe processing when the *angle_in* input data is not ready at given clock cycle. If *ce* is zero registers are disabled and new data is not loaded.

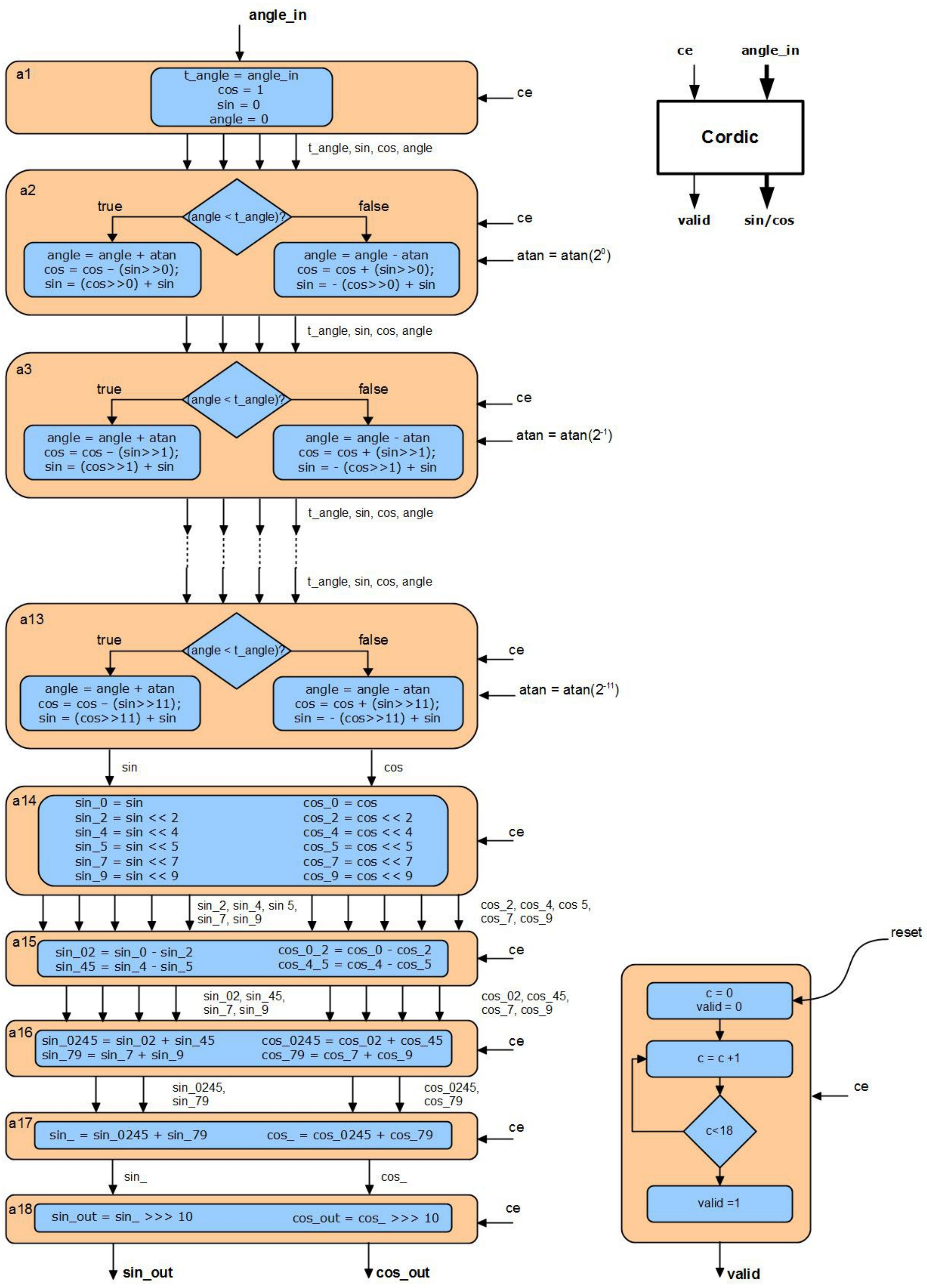
When input data is not ready it is convenient to pause not only the first stage but all processing stages to eliminate the gaps in valid data chain *i.e.* we assume that all pipeline's registers contains valid data after some initial delay. Therefore, *ce* signal is fed to all processor stages.

The *valid* output signal indicates validity of the *sin_out* and *cos_out* outputs. There is the simple counter proposed in Figure 7 to set the *valid* signal after the initial delay which is equal to the processor latency. Once *valid* is set, it remains active until processor reset .

5.2. Pipeline stages

The stages *a2* – *a13* perform eleven iterations of the cordic algorithm loop (*S3a*, *S3b*) that was unrolled first. It is worth to consider the input and output ports of each stage. Although, inputs of each stage processor are similarly named: *t_angle*, *sin*, *cos*, and *angle*, they are different values during pipeline execution. Also note, that “t_angle” is not modified in a2 – a13 stages but it must be transferred down the pipeline to allow processors perform comparison. In essence, the processor stage has to transfer to its outputs all “alive” variable that are required by the consequent stages.

The stages *a14* – *a18* perform *Kn* multiplication, and they correspond to the *S4* – *S8* states of the algorithm. The names of the signal are choosen to indicate the value of the bit shift and summation arguments. For example *sin_45* is a sum of (*sin* >>> 4) and (*sin* >>> 5). Importantly, “*sin_7*” and “*sin_9*” are not used by *a15* but they are copied from its inputs to outputs as they are necessary for calculations of *a16*.



angle_in
a1
t_angle = angle_in
cos = 1
sin = 0
angle = 0
ce
t_angle, sin, cos, angle
a2
true
(angle < t_angle)?
false
angle = angle + atan
cos = cos – (sin>>0);
sin = (cos>>0) + sin
angle = angle - atan
cos = cos + (sin>>0);
sin = - (cos>>0) + sin
ce
atan = atan(2^0)
t_angle, sin, cos, angle
a3
true
(angle < t_angle)?
false
angle = angle + atan
cos = cos – (sin>>1);
sin = (cos>>1) + sin
angle = angle - atan
cos = cos + (sin>>1);
sin = - (cos>>1) + sin
ce
atan = atan(2^-1)
t_angle, sin, cos, angle
t_angle, sin, cos, angle
a13
true
(angle < t_angle)?
false
angle = angle + atan
cos = cos – (sin>>11);
sin = (cos>>11) + sin
angle = angle - atan
cos = cos + (sin>>11);
sin = - (cos>>11) + sin
ce
atan = atan(2^-11)
sin
cos
a14
sin_0 = sin
sin_2 = sin << 2
sin_4 = sin << 4
sin_5 = sin << 5
sin_7 = sin << 7
sin_9 = sin << 9
cos_0 = cos
cos_2 = cos << 2
cos_4 = cos << 4
cos_5 = cos << 5
cos_7 = cos << 7
cos_9 = cos << 9
ce
sin_2, sin_4, sin 5,
sin_7, sin_9
cos_2, cos_4, cos 5,
cos_7, cos_9
a15
sin_02 = sin_0 - sin_2
sin_45 = sin_4 - sin_5
cos_0_2 = cos_0 - cos_2
cos_4_5 = cos_4 - cos_5
ce
sin_02, sin_45,
sin_7, sin_9
cos_02, cos_45,
cos_7, cos_9
a16
sin_0245 = sin_02 + sin_45
sin_79 = sin_7 + sin_9
cos_0245 = cos_02 + cos_45
cos_79 = cos_7 + cos_9
ce
sin_0245,
sin_79
cos_0245,
cos_79
a17
sin_ = sin_0245 + sin_79
cos_ = cos_0245 + cos_79
ce
sin_
cos_
a18
sin_out = sin_ >>> 10
cos_out = cos_ >>> 10
ce
sin_out
cos_out
ce
angle_in
Cordic
valid
sin/cos
reset
c = 0
valid = 0
c = c +1
c<18
valid =1
ce
valid

Figure 7

6. HDL code for pipelined Cordic

The verilog RTL code for the proposed pipeline processor is given below.

```verilog
/////////////////////////////////////////////////////////////////////////////////
// Design Name: The pipelined custom processor for cordic algorithm
// Module Name: cordic_pipe_rtl
/////////////////////////////////////////////////////////////////////////////////
module cordic_pipe_rtl( clock, reset, ce, angle_in, sin_out, cos_out, valid_out );
parameter integer W = 12; //Width of the fixed-point (12:10) representation
parameter FXP_MUL = 1024; //Scaling factor for fixed-point (12:10) representation
parameter PIPE_LATENCY = 15; // Input->output delay in clock cycles
input clock, reset, ce;
input [W-1:0] angle_in; //Angle in radians
output [W-1:0] sin_out, cos_out;
output valid_out;   //Valid data output flag
//Cordic look-up table
reg signed [11:0] atan[0:10] = {  12'b001100100100, 12'b000111011011, 12'b000011111011, 12'b000001111111,
                                  12'b000001000000, 12'b000000100000, 12'b000000010000, 12'b000000001000,
                                  12'b000000000100, 12'b000000000010, 12'b000000000001 };
//Tabs of wires for connections between the stage processors a2 - a13
wire signed [W-1:0] sin_tab [0:11];
wire signed [W-1:0] cos_tab [0:11];
wire signed [W-1:0] t_angle_tab [0:11]; //Target angle also must be pipelined
wire signed [W-1:0] angle_tab [0:11];
//
reg unsigned [4:0] valid_cnt; //Counts pipeline delay
//Synchroniuos activity: latency counter, angle_in latch
always@(posedge clock)
begin
    if ( reset == 1'b1 )
        valid_cnt <= PIPE_LATENCY; //Setup latency counter
    else
        if( ( valid_cnt != 0 ) && ( ce == 1'b1 ) )
            valid_cnt <= valid_cnt - 1; //Valid output data moves toward output
end
assign valid_out = ( valid_cnt == 0 )? 1'b1 : 1'b0; //Set valid_out when counter counts up to PIPE_LATENCY
//Stage a1: assign initial values (No registers - asynchronous !!!)
assign cos_tab[0] = 1.0 * FXP_MUL;
assign sin_tab[0] = 0;
assign angle_tab[0] = 0;
assign t_angle_tab[0] = angle_in;
//Stage a2 - 13 processor netlist
  cordic_step #(0) cordic_step_0 ( clock, ce, sin_tab[0], cos_tab[0], angle_tab[0], t_angle_tab[0], atan[0],
                                   sin_tab[1], cos_tab[1], angle_tab[1], t_angle_tab[1] );
  cordic_step #(1) cordic_step_1 ( clock, ce, sin_tab[1], cos_tab[1], angle_tab[1], t_angle_tab[1], atan[1],
                                   sin_tab[2], cos_tab[2], angle_tab[2], t_angle_tab[2] );
  cordic_step #(2) cordic_step_2 ( clock, ce, sin_tab[2], cos_tab[2], angle_tab[2], t_angle_tab[2], atan[2],
                                   sin_tab[3], cos_tab[3], angle_tab[3], t_angle_tab[3] );
  cordic_step #(3) cordic_step_3 ( clock, ce, sin_tab[3], cos_tab[3], angle_tab[3], t_angle_tab[3], atan[3],
                                   sin_tab[4], cos_tab[4], angle_tab[4], t_angle_tab[4] );
  cordic_step #(4) cordic_step_4 ( clock, ce, sin_tab[4], cos_tab[4], angle_tab[4], t_angle_tab[4], atan[4],
                                   sin_tab[5], cos_tab[5], angle_tab[5], t_angle_tab[5] );
  cordic_step #(5) cordic_step_5 ( clock, ce, sin_tab[5], cos_tab[5], angle_tab[5], t_angle_tab[5], atan[5],
                                   sin_tab[6], cos_tab[6], angle_tab[6], t_angle_tab[6] );
  cordic_step #(6) cordic_step_6 ( clock, ce, sin_tab[6], cos_tab[6], angle_tab[6], t_angle_tab[6], atan[6],
                                   sin_tab[7], cos_tab[7], angle_tab[7], t_angle_tab[7] );
  cordic_step #(7) cordic_step_7 ( clock, ce, sin_tab[7], cos_tab[7], angle_tab[7], t_angle_tab[7], atan[7],
                                   sin_tab[8], cos_tab[8], angle_tab[8], t_angle_tab[8] );
  cordic_step #(8) cordic_step_8 ( clock, ce, sin_tab[8], cos_tab[8], angle_tab[8], t_angle_tab[8], atan[8],
                                   sin_tab[9], cos_tab[9], angle_tab[9], t_angle_tab[9] );
  cordic_step #(9) cordic_step_9 ( clock, ce, sin_tab[9], cos_tab[9], angle_tab[9], t_angle_tab[9], atan[9],
                                   sin_tab[10], cos_tab[10], angle_tab[10], t_angle_tab[10] );
  cordic_step #(10)cordic_step_10( clock, ce, sin_tab[10], cos_tab[10], angle_tab[10], t_angle_tab[10],
                                  atan[10], sin_tab[11], cos_tab[11], angle_tab[11], t_angle_tab[11] );
//Stage a14 - 18: scaling of the results
    mul_Kn mul_Kn_sin ( clock, ce, sin_tab[11], sin_out );
    mul_Kn mul_Kn_cos ( clock, ce, cos_tab[11], cos_out );
endmodule
```

The stage processors *a2 – a13* are coded in the parametrised *cordic_step()* module, while the stages *a14 – a18* are provided in the *mul_Kn()* module.

Note the coding style of *a2 – a13* vs. *a14 – a18.* Stages *a2 – a13* result in loop unrolling, and they represent a repeated loop body for a different iteration parameter. The module *cordic_step()* with the *step* parameter was defined and used in the processor code. The module *cordic_step()* is instantiated eleven times for *step* in 0 to 10. The modules are connected using tables wires: *cos_tab[], sin_tab[], angle_tab[], t_angle_tab[]*. Stages *a14 – a18* carry similarity in the scaling procedure which is the same for *sin* and *cos.* The *mul_Kn()* module was proposed and it is instantiated separately for *sin* and *cos in* the *cordic_pipe_rtl()* module. The *mul_Kn()* module performs operations for S4 – S8 algorithm's steps.

Also notice, that II=1 simplifies our processor design in a way that stage processors do not need FSM controller, and each stage processor contains the data path only.

The template for *cordic_step()* is proposed below.

```
//////////////////////////////////////////////////////////////////////////////////
// Design Name: Pipeline cordic custom processor
// Module Name: cordic_step
// Define the cordic step in blocks a2 - a13
//////////////////////////////////////////////////////////////////////////////////
module cordic_step ( clock, ce, sin_in, cos_in, angle_in, t_angle, atan,
                                sin_out, cos_out, angle_out, t_angle_out );
parameter integer step = 0; //Step number
parameter integer W = 12; //Width for fixed-point representation. Fixpoint(12:10)
input clock, ce;
input signed [W-1:0] sin_in, cos_in, angle_in, t_angle, atan;
output reg signed [W-1:0] sin_out, cos_out, angle_out, t_angle_out;
//
always @ (posedge clock)
begin
    if( ce == 1'b1 )
    begin
        if(t_angle > angle_in)
        begin
           cos_out <= cos_in - (sin_in >>> ???); //Arithmetic shift !!!
           sin_out <= (cos_in >>> ???) + sin_in;
           angle_out <= angle_in + atan;
        end
        else
        begin
           cos_out <= cos_in + (sin_in >>> ???);
           sin_out <= -(cos_in >>> ???) + sin_in;
           angle_out <= angle_in - atan;
        end
        t_angle_out <= t_angle;
    end //if ( ce == 1'b1 )
end
endmodule
```

The scheme for *mul_Kn()* code is proposed below.

```
//////////////////////////////////////////////////////////////////////////////////
// Design Name: Pipeline cordic custom processor
// Module Name: mul_Kn
// Define the multiplication by constant Kn in blocks a14 – a18.
//////////////////////////////////////////////////////////////////////////////////
module mul_Kn(clock, ce, value_in, value_out);
parameter integer W = 12; //Width of the fixed-point (12:10) representation
parameter FXP_SHIFT = 10; //Fraction for fixed-point (12:10) representation
input clock, ce;
input signed[W-1:0] value_in;
output reg signed[W-1:0] value_out;
reg signed [2*W-1:0] val, val_0, val_2, val_4, val_5, val_7, val_7_9_d, val_9; //Shifted input values
reg signed [2*W-1:0] val_0_2, val_4_5, val_7_9, val_0_2_4_5, val_0_2_4_5_7_9;  //Accumulated values
//
always @ (posedge clock)
begin
```

```
    if( ce == 1'b1 )
    begin
    //Step S4
        val = value_in; val_0 <= val; val_2 <= val << 2; val_4 <= val << 4;
        val_5 <= val << 5; val_7 <= val << 7; val_9 <= val << 9;
    //Step S5
        val_0_2 <= val_0 - val_2; val_4_5 <= val_4 - ???; val_7_9 <= val_7 + ???;
    //Step S6
        val_0_2_4_5 <= val_0_2 + val_4_5;
        val_7_9_d <= val_7_9; //delay val_7_9 which is necessary in the 4-th pipe stage
    //Step S7
        val_0_2_4_5_7_9 = val_0_2_4_5 + val_7_9_d;
    //Step S8
        value_out <= ??? >>> FXP_SHIFT;
    end
end
endmodule
```

Exercise 6.1

Simulate *cordic_pipe_rtl()* in a Verilog simulator. Perhaps, use Vivado softwate to create simulation project.

TO DO:

a. Remember to replace '???' string with a correct statements where it is necessary in the provided code.

b. Replace eleven instantiations of *cordic_step()* with Verilog's **generate loop** block construct. For example, the generate loop block in Verilog look like:

```
generate for (j=0; j<N; j=j+1)
begin: xor_loop
xor g1 (out[j], i0[j], i1[j]);
end //end of the for loop inside the generate block
endgenerate //end of the generate block
```

c) Use the testbench proposed below:

```
module cordic_pipe_rtl_TB;
reg clock, ce, reset, start;
reg [11:0] angle_in;
real angle;
wire [11:0] sin_out, cos_out;
wire valid_out;
//For easy output value monitoring
real real_cos, real_sin;
//Instantiation
cordic_pipe_rtl cordic ( clock, reset, ce, angle_in, sin_out, cos_out, valid_out );
//Reset stimuli
initial
begin
    reset <= 1'b1;
    #10 reset <= 1'b0;
end
//ce & clock generator stimuli
initial
begin
    ce <= 1'b1;
    clock <= 1'b1;
end
always
    #5 clock <= ~clock;
//Signals stimuli
initial
```

```
    angle = 0.0;
always@(posedge clock)
begin
    if (angle < 3.14/2 ) angle = angle + 0.1; else angle = 0;
    angle_in <= angle * 1024;  //Value in fixed-point (12:10)
    //Convert and display results
    real_cos = cos_out;
    real_sin = sin_out;
    real_cos = real_cos / 1024;
    real_sin = real_sin / 1024;
    $display("Real values: sin=%f, cos=%f", real_sin, real_cos);
end
endmodule
```

Exercise 6.2

Write **behavioural** code for the module that generate (x,y) coordinates of the elipse given by the equation $\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$. Use the pipelined cordic processor, and the observation that $E(x,y)=E(a*\cos\alpha, b*\sin\alpha)$ (See Fig. 8). Assume that $a=3, b=7$.

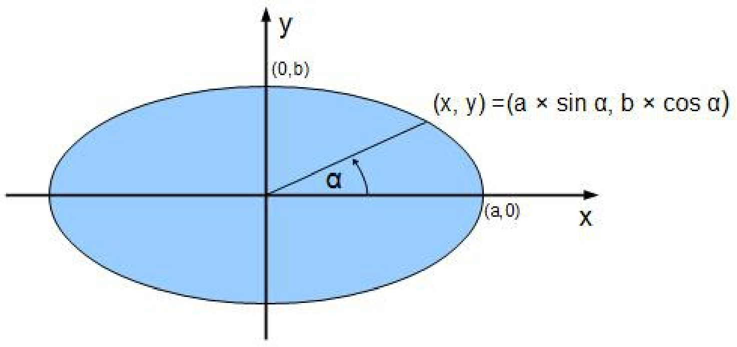


Figure 8

Exercise 6.3

Write **rtl** code for the module created in Exercise 6.2.