Fundusze Europejskie Wiedza Edukacja Rozwój · Rzeczpospolita Polska · AGH Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie · Unia Europejska Europejski Fundusz Społeczny

Instrukcja do ćwiczeń laboratoryjnych

Nazwa przedmiotu	**Systemy Dedykowane w Układach Programowalnych**
Numer ćwiczenia	**Cwiczenie 7**
Temat ćwiczenia	**Wydajny Sekwencyjny Procesor CORDIC**

Poziom studiów	***II stopień***
Kierunek	**Elektronika i Telekomunikacja**
Forma i tryb studiów	**Studia dzienne**
Semestr	**1 semestr**

autor instrukcji: Ernest Jamro

Wydział Informatyki, Elektroniki i Telekomunikacji

Kraków 2020

AGH University of
Science and Technology
Cracow
Department of
Electronics

Custom system design in FPGA laboratory

Tutorial 7

Efficient sequential CORDIC

Author: Ernest Jamro

1. Objectives and Prerequisites

In the tutorial 1, a custom CORDIC processor was described. The objective of the previous tutorials was to introduce you to design environment and the CORDIC algorithm. The previous CORDIC design was simplified and the main objective was the clarity and simplicity rather than the performance of the custom processor. In this tutorial, a custom processor will be redesigned step by step to improve performance so that the module could be used in practice as a custom hardware accelerator.

The knowledge of the previous tutorial:

T1: The Sequential Cordic processor for sine &cosine calculations

is strongly required.

2. Calculation error

For CORDIC algorithm, the auxiliary result Y_a is scaled by a constant value $K_N = \prod_{i=0}^{N-1} \cos\alpha_i$ to obtain the final result: $Y = Y_a \cdot K_N$. As the multiplication is commutative, it is possible to swap the order of operations: $Y = Y_a \cdot K_N = K_N \cdot Y_a$. Consequently, to calculate sine or cosine function, instead of initialization to a constant values: $X_0 = 1$, $Y_0 = 0$, it is possible to initialise: $X_0 = K_N$, $Y_0 = 0$ in order to skip the final multiplication by $K_N = 0.607253$ (for number of iterations of $N = 11$). The Finite State Machine with Data (FSMD) diagram from tutorial 1 is shown in Fig. 1. Consequently, states $S8$ to $S11$ are removed from the FSMD diagram. The file, *cordic_rtl2.sv*, represents the above modification. Open this file and try to understand it. Which states have been modified in comparison to the FSMD presented in Fig. 1?

As initial value is smaller ($X_0 = K_N < 1$) and the output sine and cosine results are bounded by the range 0 to 1 (for input angle 0 to π/2), it may be possible to represent fix point numbers as unsigned [12 | 11]: one integer and 11 fractional bits. The same holds for input angle, as the maximum value: π/2 = 1.57 < 2. However during simulation for input angle 0, we obtain sin_out = 0xFFF, which is equivalent to -1 LSB in two's complement code. Therefore, the error is 1 LSB = 1/1024 – which is acceptable, however requires that the numbers must be signed. Therefore [12 | 10]: one sign (two's complement), one integer and 10 fractional bits data format is finally taken. The same holds for internal numbers – as during iterations the angle and thus also the internal *sin* / *cos* value may be negative.

The next issue is: what the maximum error of the computation is. Error is expressed in LSB (Least Significant Bit) or ULP (Unit Last Place) which are equivalent. In theory, it is possible to check manually every input / output values to find out maximum error value. In our case, the input angle is in range 0 to π/2 in radians, the LSBit value is 1/1024 rad, thus 1609 = 1024·π/2 input / output values need to be verified. However, one of the most important advantage of HDL (e.g. Verilog) is that a properly designed testbench can do the job for us. Therefore, in the testbench we need to calculate accurate sine and cosine function and compare it with the UUT (Unit Under Test) output during simulation.

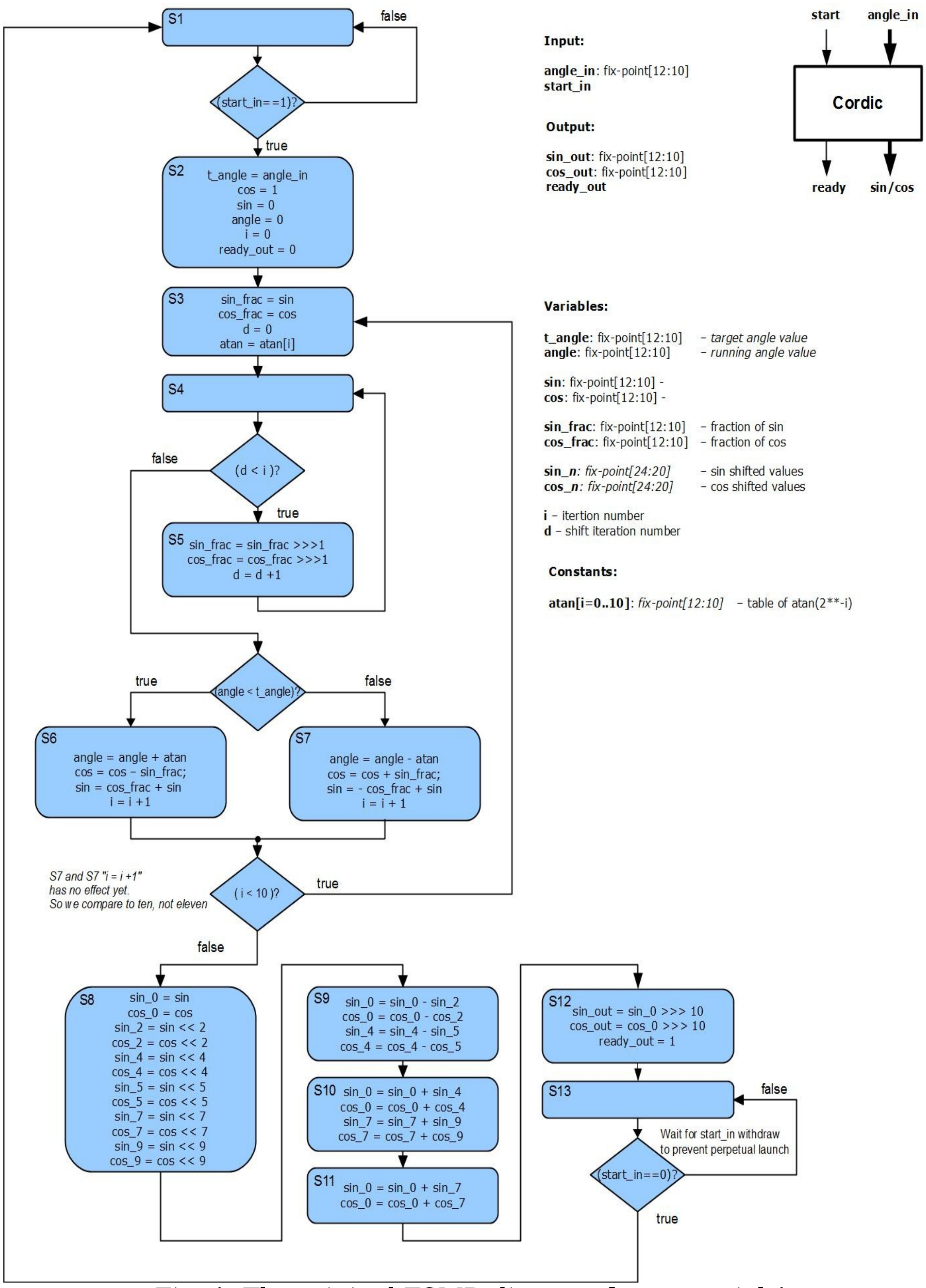

Fig. 1. The original FSMD diagram from tutorial 1

For simulation-only purposes, we can design a module at high level of abstraction as it need not to be synthesised. Consequently we can use floating point operations, which are not normally allowed without including dedicated sophisticated modules or libraries. Consequently, sine or cosine functions can be defined on high level of abstraction. In VHDL, *sin* and *cos* functions are build-in for simulation. Unfortunately, in Verilog these functions are not available. However Verilog allows to include C-language modules (by importing DPC-I library) in a similar way as C language allows to include assembler language. *Sin*(x) and *cos*(x) functions are easily available in C. The alternative solution, employed in this project, is to use Taylor series to calculate sine and cosine functions:

$$\sin(x) \approx x - x^3 / 3! + x^5 / 5! - x^7 / 7! = x - x^3 / 6 + x^5 / 120 - x^7 / 5040. \quad (1)$$

In practice, the following equivalent formula is used (see *cordic_rtl_tb.sv* file):

$$\sin = x * (1.0 + x2 \cdot (-1.0 / 6.0 + x2 \cdot (1.0 / 120.0 - x2 / 5040.0))); \quad (2)$$

where: x – input angle, $x2 = x^2$. It should be noted that the above formula does not require to calculate x^5 and x^7 as it is the case for the direct form. This formula uses Horner method to simplify the calculation.

Taylor series approximation is very accurate provided that an input angle is small. Consequently, for larger input angle ($x > 0.25$ rad in our case), the following trigonometry identity is used:

$$\cos(4\cdot x) = 2\cdot\cos^2(2\cdot x)-1 = 8\cdot\cos^4(x) - 8\cdot\cos^2(x)+1 \quad (3)$$

to scale input angle by the factor of 4. In the case of sine: $\sin(2\cdot x)= 2\cdot\sin(x)\cdot\cos(x)$ which requires knowledge both of $\sin(x)$ and $\cos(x)$, therefore different trigonometric identity is employed: $\sin(x) = \cos(\pi/2 - x)$.

The maximum error for Taylor series, for which each consecutive expression is an opposite sign, is the first skipped expression. In our case, for sine, it is: $x^9 / 9! = x^9 / 362880$. The maximum Taylor approximation error occurs for input value $x= 0.25$ and is equal $(0.25)^9 / 362880 \approx 1.05 \cdot 10^{-11} \approx 2^{-36}$. This error is small enough to be ignored.

Exercise 1

Run the simulation with *cordic_rtl_tb.sv* and *cordic_rtl2.sv* files. Find out the maximum cordic error. What is the maximum error when expressed in LSBs? See signals: *MSE_sin_LSB*, *MSE_cos_LSB* (Mean Square Error for sine / cosine expressed in LSB), *ME_sin_LSB*, *ME_sin_LSB* (Mean Error for sine / cosine expressed in LSB), error_max_LSB (maximum error for sine and cosine expressed in LSB).

In most application the maximum error should be 1 LSB or even 0.5 LSB (for exact rounding used e.g. for floating point addition, multiplication or division). Consequently, the algorithm improvement is required. Two possible solutions are possible:

1) increase the number of CORDIC algorithm iterations. Parameter: *i_max*

2) internal calculations should be performed on larger number of bits – extra guard bits are required. In most cases it is impossible to obtain *K*-bits accuracy when internal calculation is performed on the same (*K*-bit) bitwidth. Parameter *guard_bits*.

Exercise 2

Open and view file *cordic_rtl3.sv*. Run the simulation *cordic_rtl_tb.sv* and *cordic_rtl3.sv as the Unit Under Test* (*UUT*) module. Note the maximum CORDIC error, Mean Square Error (MSE) and Mean Error (ME). Modify the above mentioned parameters (*i_max*= 12, *guard_bit*=2) and then (*i_max*= 14, *guard_bit*=4). Is it possible to obtain maximum error below 1 LSB?

3. Algorithm acceleration

In the next step, we will accelerate the algorithm. The most time consuming steps of the algorithm are steps *S*4 and *S*5. Steps *S*4 and *S*5 are used to shift *sin* and *cos* values by different number of bits, one bit at the step. Consequently, to shift by *i*-bits, $i + (i+1) = 2 \cdot i+1$ clock cycles are required (*i* – for shifting, *i*+1 – for comparison with the number of shifts). In total, for the number of CORDIC iterations: *i_max*, the required number of clock cycles N_{clk} is:

$$N_{clk} = 1 + 3 + 5 + \ldots + (2 \cdot i\_max - 1) = (1 + 2 \cdot i\_max - 1) \cdot i\_max / 2 = i\_max^2. \quad (4)$$

Alternative solution is to use a barrel shifter which, by definition, can shift by any number of bits in one iteration. Therefore, the algorithm is significantly speed up, shifting requires only *i_max* iterations. In should be noted that barrel shifter is implemented in state *S*3, so no additional clock cycle (state) is required in comparison with the previous version. The barrel shifter, however, requires much more hardware resources. Nevertheless, CORDIC implementation with a single bit shifter is very slow – comparable with a software implementation on MicroBlaze which incorporates barrel shifter.

Exercise 3

Run the simulation *cordic_rtl_tb.sv* and *cordic_rtl4.sv as the Unit Under Test* (*UUT*) module. Note the simulation time – how many clock cycles per single calculation are required. Note the calculation errors?

The modification in the algorithm should not influence the calculation error. However, it is not the case, the introduction of the barrel shifter reduced the calculation error, so in the previous version of the algorithm there was an error. It should be noted that this design has been studied by hundreds of people and nobody has noticed the error. One might even draw a conclusion that CORDIC algorithm limits accuracy to 9-bit? Consequently, a conclusion can be driven that many scientific papers present false results. I personally have noticed many errors and even false symbolic proves in the highest quality papers.

Exercise 4

Run the simulation with (*i_max*= 14, *guard_bit*=4). Note the errors. Is it possible to obtain maximum error smaller than 1 LSB? Notice that the truncation rather than rounding is used. The maximum error of truncation alone is 1.0 LSB. The Mean Error (ME) for truncation is, in theory, 0.5 LSB. What, in theory, is the MSE and Root Mean Square Error (RMSE) when only truncation is taken into account? Modify *cordic_rtl4.sv* and *S12* state in order to implement rounding rather than truncation. Note the errors during simulation after the modification. What are the errors in theory when only rounding is considered? Find out, minimum values of the parameters (*i_max*, *guard_bit*) for which maximum error is below 1 LSB.

It should be noted that in some applications different types of errors are more or less important. For example, for a high pass filter, the DC offset should be zero, so the ME is critical. Normally, however, the most important errors are the maximum error or RMSE.

Let's go back to the algorithm acceleration. It can be seen that most of the time is spend on the *S*3, *S*6 and *S*7. Therefore, our next attempt is to combine the above states into a single step *S*3. Additionally to skip comparison between target (*t_angle*) and accumulated angle (*angle*): *angle* < *t_angle*, difference *angle*= *angle* - *t_angle* will be coded. In this case, the sign of *angle* (comparison with 0) is used and *angle* is initialised with *angle*= -*angle_in*. Now only a single clock cycle is required per iteration. Therefore, additional control states should be reduced. States *S*1 and *S*2 can be merged. In similar way states *S*12 and *S*13 can be merged. In conclusion, only three states are required: initial state, calculation state, and the finale state. Consequently, the total number of clock cycles for a single calculation is 2 + *i_max*.

Exercise 5

Open and view *cordic_rtl5.sv.* Simulate this file.

4. Design Implementation

The next step is design implementation for which the Verilog file will be first synthesised and then place and route program executed. Set *cordic_rtl5.sv* as the top level module by right-button clicking this file source. Before design implementation, the clock frequency should be defined. Add design source: constrain and use the enclosed *clock.xdc* file. Open this file – the file defines the clock period: 4 ns, i.e. *f*= 250 MHz. Then run implementation.

Exercise 6

After design implementation, open utilisation report. Note number of used flip-flops and LUTs. Then view *cordic_rtl5.sv* and try to count the number of required flip-flops, note signals with *reg* keyword.

The design is described on RTL (Register Transfer Level) so we can roughly count the number of used registers. The following signals generate flip-flops:

output **reg** ready_out;

output **reg** [W-1:0] sin_out, cos_out;

enum { S1, S3, S12 } state;

reg signed [W1-1 : 0] angle, sin, cos, ~~sin_frac~~, ~~cos_frac~~;

```
reg signed [W1-1 : 0] ~~atan_val~~;
reg [3:0] i;
```

Note that no every **reg** signal generates flip-flops – this is confusing in Verilog. For example:

```
reg signed [W1-1 : 0] atan [0: 16]
reg signed [W1-1 : 0] Kn [2: 17]
```

are just constants which do not require flip-flops usage. Similarly, signals *sin_frac*, *cos_frac*, *atan_val* are assigned using sign '=' rather than '<=', which means that these signals are assigned instantly without waiting for the rising clock edge – flip-flop may not be generated. On opposite side is signal *state* which is *enum* type and it is implemented as a register. Furthermore, Finite State Machine (FSM) coding style may be different: e.g. binary or one hot – which requires a different number of flip-flops. During synthesis and implementation some signals can be optimised (reduced) or replicated – so the number of flip-flops can be only roughly specified.

5. Clock Frequency

Up to now hardware utilisation and algorithm latency are known. Another very important design parameter is clock frequency or clock period. This parameter is specified in a user constrain file: *clock.xdc*. The specified clock period is 4 ns. It should be noted that in real implementation a DDL (Delay Lock Loop) / PLL (Phase Lock Loop) module is used to generate the clock signal in order to phase align external and internal clock signals – so virtually no propagation delay is observed. Summing up, the clock signal propagation time may be ignored in this design.

Exercise 7

Open *Open Implemented Design / Report Timing Summary.* Note whether the timing constrain is met. If the timing is not met, click the Worst Negative Slack (blue colour printed number) to see which signals do not meet the timing constrain. If the timing constrain was not met, modify the clock period: either by editing *xdc* file in window *sources* or in *Open Implemented Design / Edit Timing Constrain,* and then reimplement the design. What is roughly the maximum clock frequency?

In the case when timing is not met, there are two possible solutions: lower the clock frequency or change design. By viewing which signals do not meet the timing constrain, we noticed that path from *cos_reg* to *sin_reg* is too long. Consequently, *S*3 should be divided into two simper states: *S*3 and *S*4. An improved version is in file *cordic_rtl6.sv.* The drawback of this version is a larger number of occupied flip-flops and a larger number of clock cycles required to obtain the result. One CORDIC iteration requires two clock cycles: states *S*3 and *S*4. Passing from state *S*3 to *S*4 and vice-versa are repeated for every iteration of the algorithm (thus the arrows are thicker), see Fig. 2.

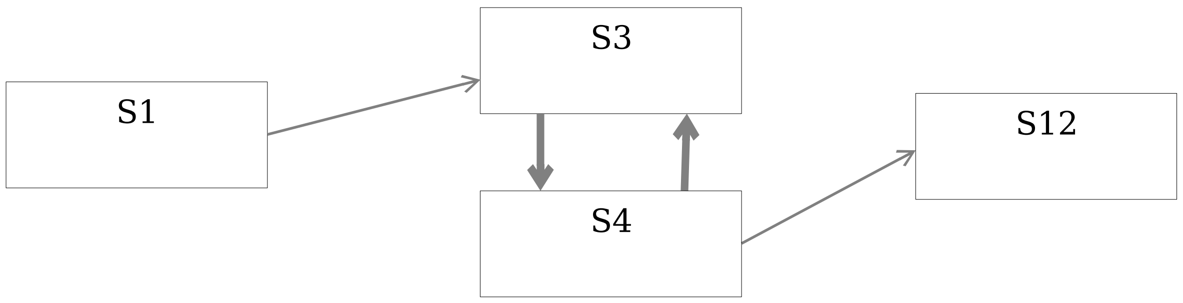

Fig. 2. FSMD State Diagram

In S3 state only the barrel shifter operations are implemented:

```
sin_frac <= sin >>> i;
cos_frac <= cos >>> i;
```

It should be noted that the barrel shifter is quite complex operation that requires both high hardware resources and long propagation time. The rest of operation is performed in *S*4.

Exercise 8

What is the maximum clock frequency for *cordic_rtl6.sv*? You may try to improve this version in order to increase the maximum clock frequency or lower hardware resources. What is the value of *A·T* factor (Area in number of LUTs · Clock Period · Algorithm Latency) of the proposed architectures: version 5, version 6 and maybe your version.

Possible scenarios of improvements:

- interchange some operation in *S*3 and *S*4 based on the timing report
- introduce additional state *S*5 and share (single use) the barrel shifter

6. Logical Processors

In version 6, our processor consists of four states (see Fig. 2). Each state requires arithmetic and logic hardware resources, but at a time only one state is active. So most hardware resources are idle. One of the solution to this problem is to process two or more data simultaneously on different stages. So the processor can calculate sin / cos for *angle1* on stage *S*3 and at the same time calculate sin / cos for *angle2* in stage *S*4. This solution is presented in file *cordic_rtl7.sv.*

It is very difficult to construct a Finite State Machine (FSM) for which two or more active stages are possible. Much easier, it is to distribute FSM to each stage separately. It should be noted that distributed rather than global control logic is preferable whenever FSM gets larger as it is easier to design or modify. Consequently version 7 has four independent FSMs for stages *S*1, *S*3, *S*4 and *S*12. There are two states in each FSM: S?_valid = 0 or 1, which means that data in the stage are valid or invalid. Therefore four control registers are defined: *S1_valid*, *S3_valid*, *S4_valid*, *out_valid*.

The input / output interface of the CORDIC module is changed to be very similar to AXI Stream: there are valid, ready and data signals both on the input and output interface. The input interface has signals: *in_valid*, *in_ready*, *in_angle* and output interface has signals: *out_valid*, *out_ready*, *out_sin*, *out_cos*. Similar interface is, somehow, used for internal inter-FSM communication.

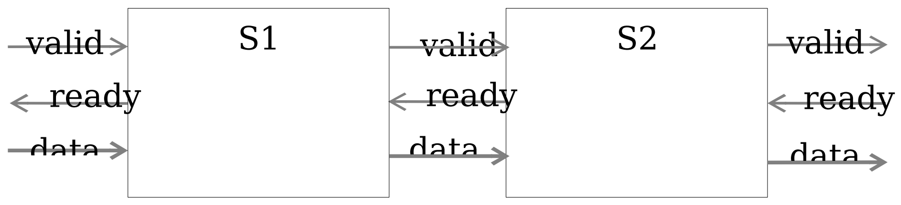

Fig. 3. AXI Stream intermodules communication

There are two scenarios of communication between stages (*S*1 example is given hereby):

Scenario 1) assign in_ready= ~S1_valid;

input interface is ready to accept new input data whenever there is no valid data in *S*1 stage (register). This design style has very simple control logic, thus: fast design time, low control logic resources, low propagation delay (high clock frequency). The main drawback of this scenario is, however, that at least every second clock cycle is idle: if S1_valid=1 then no input data is accepted, so S1_*data* should be first transferred to *S*2 state and then *S*1 can accept new data. 50% idle states (50% of throughput) is in most case unacceptable, thus this control logic is rarely used.

Scenario 2) assign *in_ready= ~S1_valid* | *~S2_valid* | … | *out_ready*

In this scenario, not only the current stage is considered but also the next stages. The above scenario assumes that stages *S*1, *S*2, *S*3, … are sequentially connected and behave in similar way as the stage *S*1. The control logic however gets bigger and bigger which is often unacceptable. Especially including *out_ready* signal is unacceptable as the control logic can include the next module control and so on. One of the solution is to insert a small FIFO (First-In First-Out) buffer at the end of the path and accept input data whenever the FIFO buffer is not half full (provided that the number of stages is smaller than the half of the FIFO depth): *in_ready= ~FIFO_half_full*.

Another solution is to consider only two or three next stages, e.g. for *in_ready= ~S1_valid* | *~S2_valid*, the control path is short and the throughput is limited only by 25%.

Lets go back to the CORDIC example and Fig. 2. So we can use the following scenarios:

1) *in_ready= ~S1_valid;*

2) *in_ready= ~S1_valid* | *~S4_valid;*

It should be noted that our case is much more complicated as stages are not sequential and there is a iteration loop for stages *S*3 and *S*4. However, when stage $S4$ is empty, data from stage $S3$ can be transferred out to stage $S4$, so also data from stage $S1$ can be transferred out to stage $S3$. Fortunately, stage $S1$ is not in the $S3$, $S4$ loop so the high throughput is not required.

Exercise 9

Simulate file *cordic_rtl7.sv.* As its input/output interface is changed in comparison to the previous versions use *cordic_rtl7_tb.sv* as the top level module. What is real input-output and an average latency (throughput) of this module. How many data can be processed simultanuously?

It should be noted that logical processing is very popular. For example, GP-GPU (General-Purpose Computing on Graphics Processing Units) commonly use logical processors. X86 processors often can double the number of logical processors in comparison to the real number of cores – this is called Hyper-threading (this phenomenon is only a part of Hyper-threading).