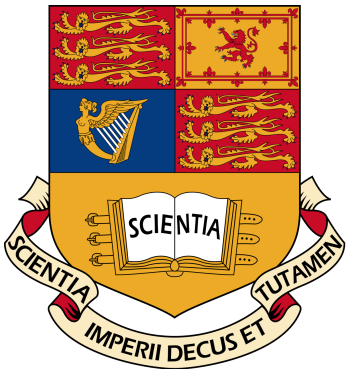


Imperial College London

Department of Electrical and Electronic Engineering

Digital System Design Report 3 (Group 1)

Name:

Arijit Bhattacharyya
Aditya Joshi

Email (CID):

ab10918@ic.ac.uk (01496199)
aj5118@ic.ac.uk (01519308)

Coursework ELEC96011 (EE3-05): Digital System Design

1 Introduction

The objective of this report is to build a system using an FPGA to evaluate the target function $f(x) = \sum(0.5 \times x + x^2 \times \cos(\frac{x-128}{128}))$ for multiple test cases. The process has taken many stages. The first stage was to design a NIOS based system for function evaluation. A NIOS II processor was implemented on the FPGA using on-chip Random Access Memory (RAM). The processor was then improved by using external off-chip memory. After this, the design was extended to provide support for floating point multipliers and adders. The final stage required us to add a custom IP block to evaluate the above function and to further optimise the design. Throughout this process, the main factors dictating our optimisation have been the performance and resource utilisation. The performance is assessed by the system's latency and this is measured using C code from specific libraries. The utilisation is calculated through the system's use of logic elements (LEs), block memory bits (MBs) and embedded multipliers (EMs).

2 Task 6: Creating Hardware Floating-Point Units

2.1 Overview

In this section, we will implement custom logic blocks in order to map the target function result to hardware. In all previous tasks, the floating point arithmetic has been implemented in software using C code. We will now create floating point adder and multiplier blocks using the FP_Functions Intel FPGA IP library in order to make this transition to hardware.

2.2 Design Choices

Firstly, when deciding on our custom floating-point adder and multiplier blocks, we have aimed to design its timing through latency. This allows us to apply the lowest latency that is possible with a 50Mhz clock. The adder block uses the single (32 bit input data) format and has a target latency of 2 clock cycles in order to function correctly with the 50MHz system clock with a resource usage of 729 LUTs. The multiplier block also uses the single format and has a latency of 2, meaning it requires 185 LUTs and 2 Multiplies to function. In addition, multi-cycle custom instructions have been chosen to implement this design, as we interface with the system clock of the NIOS II processor. For this design implementation, no pipelining has been used. This is because the NIOS II processor cannot pipeline the custom instructions of the floating point adder and multiplier, as it will need to block/wait before previous instructions are completed, meaning that these custom blocks cannot be fully utilised.

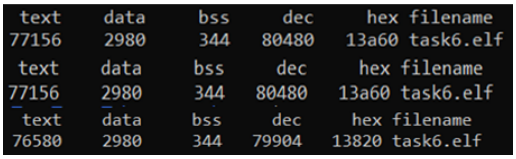
2.3 Comparison with previous best design

After having taken our design choices into account, we have implemented custom logic blocks for the floating-point adder and multiplier in order to support hardware acceleration for $f(x)$. The C code to define the two separate sets of custom instruction macros for the new blocks has been provided in the appendix [1]. The following table highlights the comparison between our previous best performing system in Task 5 and our new system which implements the custom instruction adder and multiplier:

IC (kB)	DC (kB)	Test case	Latency for cosf (ms)		Task 6 Resources	Task 6 error
			Task 5	Task 6		
2	2	1	37	24.6	0.117585287	8.77386×10^{-8}
		2	1441	973.45		8.80409×10^{-8}
		3	189401	125631.4		5.49001×10^{-9}
2	8	1	37	24.525	0.094991266	8.77386×10^{-8}
		2	1437	964.4		8.80409×10^{-8}
		3	188805	124455.4		5.49001×10^{-9}
8	8	1	18	9.1	0.071893466	8.77386×10^{-8}
		2	713	358.6		8.80409×10^{-8}
		3	91523	54035		5.49001×10^{-9}
32	32	1	5	6.825	0.037742508	8.77386×10^{-8}
		2	174	270.75		8.80409×10^{-8}
		3	24158	35662.5		5.49001×10^{-9}
64	32	1	5	6.8	0.033345551	8.77386×10^{-8}
		2	174	270.75		8.80409×10^{-8}
		3	24158	35662.5		5.49001×10^{-9}
64	64	1	5	6.8	0.029178174	8.77386×10^{-8}
		2	174	270.725		8.80409×10^{-8}
		3	24144	35651.5		5.49001×10^{-9}

The table above shows that adding the 2 custom instruction blocks makes the system perform faster for fewer resources. However as the cache size increases, the performance plateaus earlier than in previous tasks. This is because hardware implementations require less instruction and data memory to operate and so do not utilise the cache's ability to store looped data that can be accessed much more quickly. The software implementation is better optimised for high cache. However, this still gives the best performance resources trade off.

The code sizes for Test Cases 1, 2 and 3 respectively are shown below:



We note that this is higher than the sizes seen for Task 5 (text = 68804), as we are now accounting for the custom instruction macros in our C code.

3 Task 7: Computing the inner part of the arithmetic expression

3.1 Overview

In this section, we will discuss the steps taken to decide on our hardware IP architecture. We were first required to pick an error analysis method to calculate the Mean Squared Error (MSE) of our CORDIC algorithm, which computes the cosine in $f(x)$. As a result, we chose the Monte Carlo method as our form of error analysis, as it is the most flexible approximation method and is applicable to all systems. We will also discuss two different implementations of the CORDIC algorithm in Verilog HDL, the partially folded and the unrolled architecture.

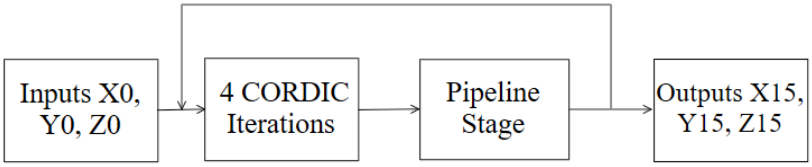
3.2 Monte Carlo Investigation

The system specifications require us to develop a CORDIC algorithm which provides a Mean Squared Error (MSE) less than or equal to 1×10^{-10} with a confidence level of 95%. The parameters we will need to consider are the number of stored angles our CORDIC algorithm iterates through, the word bit length of our binary variables and its fractional bit length. We keep the gap between the word and fractional bit lengths as 2 as the CORDIC algorithm input is defined for $[-1, 1]$, we only require one bit for the integer and one bit for the sign. Most of our bits are needed to ensure precision of the decimal. In order to ensure we achieve the required confidence level, we will focus on the upper bound of the error. The MATLAB code used to compute the upper bound is provided in the appendix [2]. Hence, the following upper bounds have been computed for varying word length and CORDIC iterations and have been expressed in the heatmap below:

Iterations	**WL** 18 **FL** 16	**WL** 20 **FL** 18	**WL** 22 **FL** 20	**WL** 27 **FL** 25	**WL** 32 **FL** 30
10	3.648×10^{-7}	3.689×10^{-7}	3.756×10^{-7}	3.479×10^{-7}	3.604×10^{-7}
11	8.877×10^{-8}	8.810×10^{-8}	9.051×10^{-8}	9.226×10^{-8}	8.802×10^{-8}
12	2.298×10^{-8}	2.289×10^{-8}	2.274×10^{-8}	2.292×10^{-8}	2.295×10^{-8}
13	6.544×10^{-9}	5.502×10^{-9}	5.423×10^{-9}	5.451×10^{-9}	5.554×10^{-9}
14	2.299×10^{-9}	1.417×10^{-9}	1.444×10^{-9}	1.392×10^{-9}	1.412×10^{-9}
15	1.326×10^{-9}	4.114×10^{-10}	3.422×10^{-10}	3.556×10^{-10}	3.634×10^{-10}
16	1.141×10^{-9}	1.574×10^{-10}	$\mathbf{9.163}\times10^{-11}$	9.001×10^{-11}	8.951×10^{-11}
17	1.165×10^{-9}	9.713×10^{-11}	2.681×10^{-11}	2.492×10^{-11}	2.345×10^{-11}
18	1.152×10^{-9}	7.959×10^{-11}	1.065×10^{-11}	1.032×10^{-11}	1.019×10^{-11}
19	1.124×10^{-9}	8.356×10^{-11}	6.651×10^{-12}	6.637×10^{-12}	6.624×10^{-12}
20	1.172×10^{-9}	8.399×10^{-11}	5.901×10^{-12}	5.856×10^{-12}	5.853×10^{-12}
40	1.155×10^{-9}	8.314×10^{-11}	5.722×10^{-12}	5.713×10^{-12}	5.715×10^{-12}
60	1.153×10^{-9}	8.459×10^{-11}	5.724×10^{-12}	5.714×10^{-12}	5.712×10^{-12}

Red areas fail to achieve the required MSE, yellow areas are of the same order of magnitude of the MSE but are still not suitable and green areas are suitable implementations for our FPGA. Our design philosophy is to obtain the most accurate result, while using the fewest resources possible. As a result, we decided to use a word length of 22, with 16 CORDIC iterations for the best resource accuracy tradeoff.

3.3 Folded CORDIC architecture

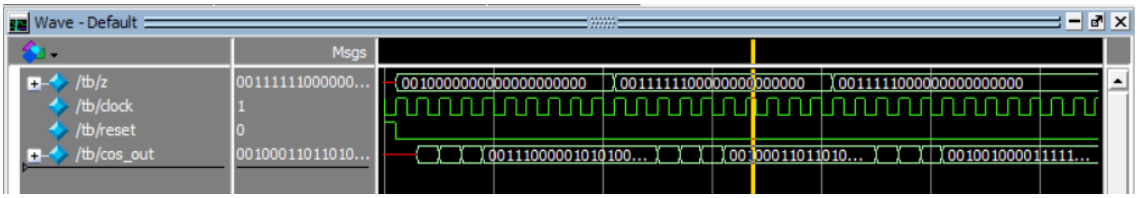


There are two main CORDIC architectures to consider. A folded architecture, where we loop the CORDIC output back to the input, and an unrolled version where we use multiple blocks and connect them in series, so the iteration is performed as the signal ripples through the blocks. For both implementations we use 22 bit word lengths and store 16 CORDIC angles as justified above. We also use fixed point binary to represent our values as this allows for fixed point bit shifting and addition to be utilised, which is much simpler to implement and cheaper to design than using floating point hardware. We do however require extra hardware to convert from floating to fixed point.

For our folded implementation there are further design modifications we can make to improve latency. To pipeline our design we can implement a partially folded CORDIC design, where instead of taking one clock cycle to perform a CORDIC iteration, it can perform multiple iterations within a clock cycle. We do this by connecting our CORDIC blocks in series, before looping the output back to the input.

The limitation of this however is timing, since we want the iterations to be performed within a clock cycle. To balance timing with design optimisation we have used four stages per pipeline, so our design now only takes 4 clock cycles to complete sixteen iterations.

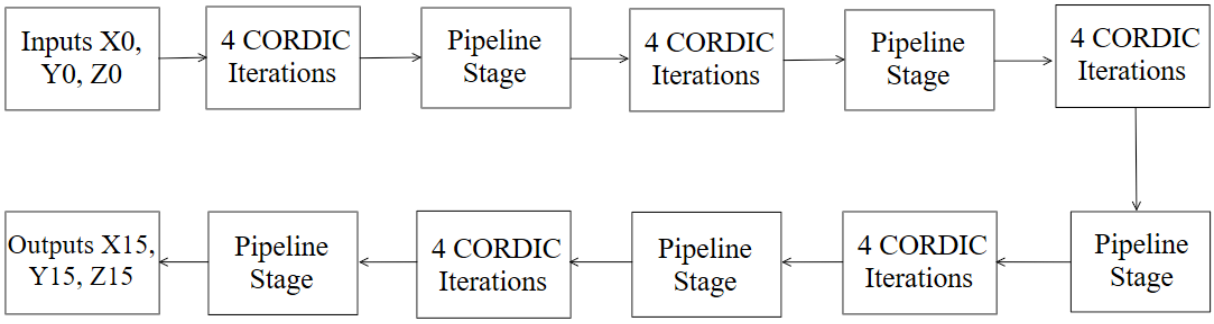
This design also means a new input cannot be processed until the previous input has finished iterating, however fewer overall blocks are needed due to the feedback loop.



We show the functionality of this CORDIC method using the following ModelSim graph. Here our input to the CORDIC is z and cos_out is the result. We see our design takes four clock cycles to compute the answer as intended with pipelining, however additional logic is required to ensure the output does not become periodic, i.e keep looping through the same four iterations every time, and only performs iterations when z changes. This results in a clock cycle delay in the output when the input changes, resulting in a 5 clock cycle delay.

We now need to add in our fixtofloat and floattofix converters and ensure our design meets timing contraints. This is especially important for partially rolled as the correct output needs to be fed back to the input for accuracy. We obtain a slack time 10ns for setup and 0.225 for hold. Thus our pipelined design meets timing requirements.

3.4 Unrolled CORDIC architecture



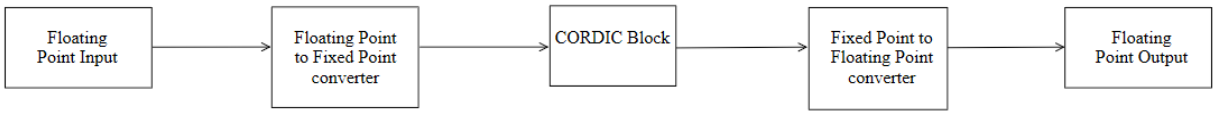
Our unrolled design has several advantages over the rolled. Our iterations now operate asynchronously and so are not limited to clock cycles. Depending on the number of CORDIC iterations this would in theory allow us to perform the entire CORDIC computation within a clock cycle. With 16 iterations this is not possible. However, we can pipeline the design by adding in flip-flops after n number of CORDIC stages. This not only allows us to meet timing requirements but allows changes in input every clock cycle to see changes in output every clock cycle, whereas for rolled we needed to wait four clock cycles for the previous computation to finish. As a result, we found four stages per pipeline to give the best latency timing trade off. Just to note, the penalty for timing violation is also less harsh in the unrolled case, since it does not affect the inputs that are being computed, only which output is sampled.

We see the ModelSim to demonstrate functionality below:

Excluding a one clock cycle delay in the beginning due to enable signals being asserted, we see our CORDIC takes three clock cycles to compute the answer, as we have excluded the d flip-flop for the output of the 16th CORDIC block since it will be input to the flip flop of the fixtofloat. By changing

the input more frequently, i.e every 5 clock cycles we see how we no longer need to wait for a certain number of clock cycles and the outputs can be computed sequentially.

Similar to the rolled case, we need to test our pipelined design for timing violations, including our fixtofloat and floattofix converters especially since we are using one less flip flop at the output. We obtain a setup slack of 14.6ns and a hold slack of 0.191, thus our design meets timing constraints.



3.5 Remaining parts of the design

We are now in a position to connect all our modules together and create one custom instruction that implements the target function. We can optimise the operations $0.5\times$x and $\frac{x}{128}$ by subtracting the exponent of the 32 bit IEEE number, rather than using floating point multipliers. Our table shows the performance of both CORDIC architectures for 2KB data and instruction cache. We see for both architectures, large improvements in both latency and resource usage compared to Task 5, where at 64KB cache we had a latency of 24K ticks, showing our design has optimised both latency and resources. There are a few design features that it shows. As expected the rolled CORDIC has improved resource utilisation, however we see no difference in latency for the smaller test cases, and only a 50ms difference for test case three. For smaller test cases, our timing is more limited by the floating point hardware we have added to the design. Since there are fewer inputs to the system, both CORDIC architectures give similar latency's, whereas the partially rolled utilises fewer resources making it the better implementation. For test case three, where the number of inputs is the greatest, the latency difference between the CORDIC blocks becomes more evident. For even greater inputs we would expect to see the unrolled CORDIC outperform the rolled even more, justifying its increased resource usage.

Test Case	CORDIC Architecture	Utilisation	Latency(ms)	Result	Accuracy(%)
1	Unrolled	0.065693125	1.65	920412.494572	1.22997×10^{-4}
2			65.875	36123021.835448	1.76387×10^{-4}
3			4833.8	4621480332.99533	1.87924×10^{-4}
1	Rolled	0.057932802	1.65	920412.494572	1.22997×10^{-4}
2			65.875	36123021.835448	1.76387×10^{-4}
3			4882.8	4621480332.99533	1.87924×10^{-4}

4 Task 8: Computing the arithmetic expression

In this section we now need to implement the expression sum = sum + F(x) in hardware and decide which CORDIC architecture to use for our final hardware product. There are a few optimisation options that we can use and this will determine the most suitable CORDIC architecture. We can pipeline our design to accept multiple inputs. We would accomplish this by holding inputs for a certain number of clock cycles to allow the hardware block to compute F(x1). We would then hold this result until the result of the next input has been computed, then add the two, ensuring our results are stored and added on to the next F(x) output. For this implementation an unrolled architecture would be best as we can pipeline the inputs and produce a faster result. Alternatively, we can accept as input a new value x and a previous sum value, adding the previous sum to F(x). For this implementation a partially rolled CORDIC is the better option, since we aren't utilising pipelining and so the benefit of the rolled architecture's lower resource utilisation outweighs the lower latency of the unrolled. While a pipelined accumulator would give a better throughput and lower latency, we decided to utilise the low resource cost of the partially rolled CORDIC and thus make a hardware floating point adder that takes as input the previous sum and a new value. We see our results in the table below:

Test Case	Latency (ms)	Utilisation	Result	Accuracy(%)
1	0.35	0.0613628	940092.4375	2.138039929
2	10	0.0613628	36142336	0.053291262
3	1266	0.0613628	4621539328	0.001088613

Our application size comes to 68500B. We see our function now has significantly lowered latency compared to task 7 and all previous tasks. However this has come with a worse result accuracy. This is because our variables in C have been declared as doubles in order to show the full precision of the output of our custom instructions. In task 8, inputs to the accumulator can only be 32 bits resulting in the truncation of our results, whereas for task 7 the sum variable in sum=sum+F(x) can be a double, resulting in a more precise answer. We also see that the accuracy improves with higher test case, as more of the inputs are fractional values. The FP adder is optimised for fractional values, since 23 out of 32 bits of the IEEE standard are for values past the decimal point.

5 Appendix

5.1 C code to implement custom instruction floating-point adder and multiplier blocks

```c
#include <math.h>
#define ALT_CI_MULT_0_N 0X0
#define ALT_CI_MULT_0(A,B) __builtin_custom_fnff(ALT_CI_MULT_0_N,(A),(B))
#define ALT_CI_ADDER_0_N 0X1
#define ALT_CI_ADDER_0(A,B) __builtin_custom_fnff(ALT_CI_ADDER_0_N,(A),(B))
double sumVector(double x[], int M){
    volatile double f_x = 0.0;
    double half_m_x, x_m_x, x_m_const, cos_input, xx_m_cos, sum;
    for(unsigned int j=0; j<M; j++){
            half_m_x = ALT_CI_FP_MULT_0(0.5, x[j]);
            x_m_x = ALT_CI_FP_MULT_0(x[j],x[j]);
            x_m_const = ALT_CI_FP_MULT_0(x[j], 0.0078125);
            cos_input = ALT_CI_ADDER_0(x_m_const,-1);
            xx_m_cos = ALT_CI_FP_MULT_0(x_m_x, cosf(cos_input));
            sum = ALT_CI_ADDER_0(half_m_x, xx_m_cos);
            f_x = ALT_CI_ADDER_0(sum, f_x);
    }
    return f_x;
} // implements f_x = f_x + (0.5*x[j] + (x[j]*x[j])*cosf(x[j]*0.0078125 - 1))
```

5.2 Monte Carlo MATLAB code

```
% fraction_length and niters are user-defined integers
word_length = fraction_length + 2;
size = 5000; % uniform random 5000-sample distribution
distrib = sfi(unifrnd(-1, 1, 1, size), word_length, fraction_length);
thRadFxp = sfi(distrib,word_length,fraction_length);
cosThRef = cos(double(thRadFxp));
MSE =  mean((cosThRef - double(cordiccos(thRadFxp,niters))).^2);
stdev = std((cosThRef - double(cordiccos(thRadFxp,niters))).^2);
upper_bound = MSE + 1.96*stdev/sqrt(length(distrib));
format longg, upper_bound
```

5.3 Partially Folded CORDIC Verilog code

```
1  `define K 22'b0010011011011101001110  // = 0.6072529350088814
2
3  module CORDIC1(
4      clock,     // Master clock
5      start,
6      reset,     // Async reset, high when cordic begins
7      z,         // Input angle
8      cos_out    // Output value for cosine of angle
9  );
10 input clock;
11 input reset;
12 input [21:0] z;
13 input  start;
14 output [21:0] cos_out;
15 reg [21:0] cos;
16 reg [21:0] sin;
17 reg [21:0] angle;
18 reg [21:0] cos_start;          // Set up initial value for cos.
19 reg [21:0] sin_start;
20 reg [21:0] angle_start;        // Latch input angle into angle register
21 reg [21:0] prev_start ;
22 reg [21:0] prev_input;
23 reg [21:0] cos_final;
24 reg [21:0] final_next;
25 reg [4:0] count;
26 reg [4:0]  count_start;        // Set up counter.
27 reg en_start, new_start, state, en_next, en, state_start, new_input;
28 reg input_next, state_next;
29 wire [21:0] cos_next;
30 wire [21:0] sin_next;
31 wire [21:0] angle_next;
32 wire [4:0] count_next;
33 wire [21:0] cos_next2;
34 wire [21:0] sin_next2;
35 wire [21:0] angle_next2;
36 wire [4:0] count_next2;
37 wire [21:0] cos_next3;
38 wire [21:0] sin_next3;
39 wire [21:0] angle_next3;
40 wire [4:0] count_next3;
41 wire [21:0] cos_next4;
42 wire [21:0] sin_next4;
43 wire [21:0] angle_next4;
44 wire [4:0] count_next4;
45 wire [21:0] cos_out = cos_final;
46
```

```
47 cordicadder add1(.en(en),.cos(cos),.sin(sin),.angle(angle),.count(
      count),.cos_next(cos_next),.sin_next(sin_next),.angle_next(
      angle_next),
48 .count_next(count_next));
49
50 cordicadder add2(.en(en),.cos(cos_next),.sin(sin_next),.angle(
      angle_next),.count(count_next),.cos_next(cos_next2),.sin_next(
      sin_next2),
51 .angle_next(angle_next2),
52 .count_next(count_next2));
53
54 cordicadder add3(.en(en),.cos(cos_next2),.sin(sin_next2),.angle(
      angle_next2),.count(count_next2),.cos_next(cos_next3),.sin_next(
      sin_next3),
55 .angle_next(angle_next3),
56 .count_next(count_next3));
57
58 cordicadder add4(.en(en),.cos(cos_next3),.sin(sin_next3),.angle(
      angle_next3),.count(count_next3),.cos_next(cos_next4),.sin_next(
      sin_next4),
59 .angle_next(angle_next4),
60 .count_next(count_next4));
61
62 initial begin
63     state<=0;
64     new_input<=1'b1;
65     input_next<=1'b1;
66     prev_input=22'd2;
67 end
68
69 always @(posedge clock or posedge reset) begin
70     if (reset) begin
71         cos <= 0;
72         angle <= 0;
73         count <= 0;
74         state <= 0;
75                 en<=0;
76                 new_input<=1;
77          end
78          else if(~state&new_input)begin
79                  cos<=cos_start;
80                  sin<=sin_start;
81                  angle<= angle_start;
82                  count<=count_start;
83                  state<=state_start;
84                  en<=en_start;
85                  new_input<=new_start;
86                  prev_input<=prev_start;
87         end else begin
```

```
88                  cos <= cos_next4;
89          sin <= sin_next4;
90          angle <= angle_next4;
91          count <= count_next4;
92          state <= state_next;
93                  en<=en_next;
94                  prev_input<=z;
95                  new_input<=input_next;
96                  cos_final<=final_next;
97          end
98  end
99
100 always@(*) begin
101         if(prev_input==z) begin
102                 input_next=0;
103         end else begin
104                 input_next=1'b1;
105         end
106
107 if (state) begin
108         // Compute mode.
109         state_next<=1'b1;
110         en_next<=1'b1;
111
112         if (count_next4 == 5'd16) begin
113             // If this is the last iteration, go back to the idle
                    state.
114             state_next <= 0;
115         en_next<=1'b0;
116                   final_next<=cos_next4;
117                   end
118
119     end else begin
120                 if(start) begin
121                 cos_start<=`K;
122                 sin_start<=0;
123                 angle_start<=z;
124                 new_start<=0;
125                 prev_start<=z;
126                 en_start=1'b1;
127                 count_start<=0;
128                 state_start<=1;
129                 end else begin
130                 state_start<=0;
131                 new_start<=1'b1;
132                 end
133         end
134 end
135 endmodule
```

5.4 Unrolled CORDIC Verilog code

```
1  `define K 22'b0010011011011101001110  // = 0.6072529350088814
2  module CORDIC2(clock, reset, start, z, cos_out);
3  // Start is an input signal that the user sets high when computation
       begins
4  input clock;
5  input reset;
6  input start;
7  input [21:0] z;
8  output [21:0] cos_out;
9  reg state, en;
10 reg [21:0]      sreg4cos;
11 reg [21:0]      sreg4sin;
12 reg [4:0]       sreg4count;
13 reg [21:0]      sreg4angle;
14 reg [21:0]      sreg8cos;
15 reg [21:0]      sreg8sin;
16 reg [4:0]       sreg8count;
17 reg [21:0]      sreg8angle;
18 reg [21:0]      sreg12cos;
19 reg [21:0]      sreg12sin;
20 reg [4:0]       sreg12count;
21 reg [21:0]      sreg12angle;
22 reg [21:0]      sreg16cos;
23 reg [21:0]      sreg16sin;
24 reg [4:0]       sreg16count;
25 reg [21:0]      sreg16angle;
26 wire [21:0] cosout4=sreg4cos;
27 wire [21:0] sinout4=sreg4sin;
28 wire [4:0]  countout4=sreg4count;
29 wire [21:0] angleout4=sreg4angle;
30 wire [21:0] cosout8=sreg8cos;
31 wire [21:0] sinout8=sreg8sin;
32 wire [4:0]  countout8=sreg8count;
33 wire [21:0] angleout8=sreg8angle;
34 wire [21:0] cosout12=sreg12cos;
35 wire [21:0] sinout12=sreg12sin;
36 wire [4:0]  countout12=sreg12count;
37 wire [21:0] angleout12=sreg12angle;
38 wire [21:0] angle_next;
39 wire [21:0] sin_next;
40 wire [21:0] cos_next;
41 wire [4:0]  count_next;
42 wire [21:0] angle_next2;
43 wire [21:0] sin_next2;
44 wire [21:0] cos_next2;
45 wire [4:0]  count_next2;
46 wire [21:0] angle_next3;
```

```
47 wire [21:0] sin_next3;
48 wire [21:0] cos_next3;
49 wire [4:0]  count_next3;
50 wire [21:0] angle_next4;
51 wire [21:0] sin_next4;
52 wire [21:0] cos_next4;
53 wire [4:0]  count_next4;
54 wire [21:0] angle_next5;
55 wire [21:0] sin_next5;
56 wire [21:0] cos_next5;
57 wire [4:0]  count_next5;
58 wire [21:0] angle_next6;
59 wire [21:0] sin_next6;
60 wire [21:0] cos_next6;
61 wire [4:0]  count_next6;
62 wire [21:0] angle_next7;
63 wire [21:0] sin_next7;
64 wire [21:0] cos_next7;
65 wire [4:0]  count_next7;
66 wire [21:0] angle_next8;
67 wire [21:0] sin_next8;
68 wire [21:0] cos_next8;
69 wire [4:0]  count_next8;
70 wire [21:0] angle_next9;
71 wire [21:0] sin_next9;
72 wire [21:0] cos_next9;
73 wire [4:0]  count_next9;
74 wire [21:0] angle_next10;
75 wire [21:0] sin_next10;
76 wire [21:0] cos_next10;
77 wire [4:0]  count_next10;
78 wire [21:0] angle_next11;
79 wire [21:0] sin_next11;
80 wire [21:0] cos_next11;
81 wire [4:0]  count_next11;
82 wire [21:0] angle_next12;
83 wire [21:0] sin_next12;
84 wire [21:0] cos_next12;
85 wire [4:0]  count_next12;
86 wire [21:0] angle_next13;
87 wire [21:0] sin_next13;
88 wire [21:0] cos_next13;
89 wire [4:0]  count_next13;
90 wire [21:0] angle_next14;
91 wire [21:0] sin_next14;
92 wire [21:0] cos_next14;
93 wire [4:0]  count_next14;
94 wire [21:0] angle_next15;
95 wire [21:0] sin_next15;
```

```
96  wire [21:0] cos_next15;
97  wire [4:0]  count_next15;
98  wire [21:0] angle_next16;
99  wire [21:0] sin_next16;
100 wire [21:0] cos_next16;
101 wire [4:0]  count_next16;
102 wire [21:0] cos_out = cos_next16;
103
104 cordicadder add1(.en(en),.cos(`K),.sin(0),.angle(z),.count(0),.
       cos_next(cos_next),.sin_next(sin_next),.angle_next(angle_next),
105 .count_next(count_next));
106
107 cordicadder add2(.en(en),.cos(cos_next),.sin(sin_next),.angle(
       angle_next),.count(count_next),.cos_next(cos_next2),.sin_next(
       sin_next2),
108 .angle_next(angle_next2),
109 .count_next(count_next2));
110
111 cordicadder add3(.en(en),.cos(cos_next2),.sin(sin_next2),.angle(
       angle_next2),.count(count_next2),.cos_next(cos_next3),.sin_next(
       sin_next3),
112 .angle_next(angle_next3),
113 .count_next(count_next3));
114
115 cordicadder add4(.en(en),.cos(cos_next3),.sin(sin_next3),.angle(
       angle_next3),.count(count_next3),.cos_next(cos_next4),.sin_next(
       sin_next4),
116 .angle_next(angle_next4),
117 .count_next(count_next4));
118
119 cordicadder add5(.en(en),.cos(cosout4),.sin(sinout4),.angle(angleout4)
       ,.count(countout4),.cos_next(cos_next5),.sin_next(sin_next5),
120 .angle_next(angle_next5),
121 .count_next(count_next5));
122
123 cordicadder add6(.en(en),.cos(cos_next5),.sin(sin_next5),.angle(
       angle_next5),.count(count_next5),.cos_next(cos_next6),.sin_next(
       sin_next6),
124 .angle_next(angle_next6),
125 .count_next(count_next6));
126
127 cordicadder add7(.en(en),.cos(cos_next6),.sin(sin_next6),.angle(
       angle_next6),.count(count_next6),.cos_next(cos_next7),.sin_next(
       sin_next7),
128 .angle_next(angle_next7),
129 .count_next(count_next7));
130
131 cordicadder add8(.en(en),.cos(cos_next7),.sin(sin_next7),.angle(
       angle_next7),.count(count_next7),.cos_next(cos_next8),.sin_next(
```

```
   sin_next8),
.angle_next(angle_next8),
.count_next(count_next8));

cordicadder add9(.en(en),.cos(cosout8),.sin(sinout8),.angle(angleout8)
   ,.count(countout8),.cos_next(cos_next9),.sin_next(sin_next9),
.angle_next(angle_next9),
.count_next(count_next9));

cordicadder add10(.en(en),.cos(cos_next9),.sin(sin_next9),.angle(
   angle_next9),.count(count_next9),.cos_next(cos_next10),.sin_next(
   sin_next10),
.angle_next(angle_next10),
.count_next(count_next10));

cordicadder add11(.en(en),.cos(cos_next10),.sin(sin_next10),.angle(
   angle_next10),.count(count_next10),.cos_next(cos_next11),.sin_next(
   sin_next11),
.angle_next(angle_next11),
.count_next(count_next11));

cordicadder add12(.en(en),.cos(cos_next11),.sin(sin_next11),.angle(
   angle_next11),.count(count_next11),.cos_next(cos_next12),.sin_next(
   sin_next12),
.angle_next(angle_next12),
.count_next(count_next12));

cordicadder add13(.en(en),.cos(cosout12),.sin(sinout12),.angle(
   angleout12),.count(countout12),.cos_next(cos_next13),.sin_next(
   sin_next13),
.angle_next(angle_next13),
.count_next(count_next13));

cordicadder add14(.en(en),.cos(cos_next13),.sin(sin_next13),.angle(
   angle_next13),.count(count_next13),.cos_next(cos_next14),.sin_next(
   sin_next14),
.angle_next(angle_next14),
.count_next(count_next14));

cordicadder add15(.en(en),.cos(cos_next14),.sin(sin_next14),.angle(
   angle_next14),.count(count_next14),.cos_next(cos_next15),.sin_next(
   sin_next15),
.angle_next(angle_next15),
.count_next(count_next15));

cordicadder add16(.en(en),.cos(cos_next15),.sin(sin_next15),.angle(
   angle_next15),.count(count_next15),.cos_next(cos_next16),.sin_next(
   sin_next16),
.angle_next(angle_next16),
```

```
.count_next(count_next16));
initial begin
          sreg4cos=22'b0;
          sreg4sin=22'b0;
          sreg4count=0;
          sreg4angle=0;
          sreg8cos=0;
          sreg8sin=0;
          sreg8count=0;
          sreg8angle=0;
          sreg12cos=0;
          sreg12sin=0;
          sreg12count=0;
          sreg12angle=0;
        end
always @(posedge clock or posedge reset) begin
          sreg4cos<=cos_next4;
          sreg4sin<=sin_next4;
          sreg4count<=count_next4;
          sreg4angle<=angle_next4;
          sreg8cos<=cos_next8;
          sreg8sin<=sin_next8;
          sreg8count<=count_next8;
          sreg8angle<=angle_next8;
          sreg12cos<=cos_next12;
          sreg12sin<=sin_next12;
          sreg12count<=count_next12;
          sreg12angle<=angle_next12;
        /* sreg16cos<=cos_next16;
          sreg16sin<=sin_next16;
          sreg16count<=count_next16;
          sreg16angle<=angle_next16;
          */
    if (reset) begin
        state <= 0;
                en<=1'b0;
    end
    else if(state) begin
        en<=1'b1;
    end
    else if(start) begin
        en<=1'b1;
            state<=1'b1;
    end
end
endmodule
```