Digital System Design Report 3 (Group 17)

Keran Zheng (kz5218@ic.ac.uk)
Yuting Xu (yx8918@ic.ac.uk)

March 2021

Contents

1 Task 6

In this task, we added floating point add/sub and multiplier support into our NIOS system by using the FP_FUNCTIONS IP as custom instructions. By switching towards dedicated FP hardware, we aim to reduce latency as well as increase accuracy compared to previous implementation of the test cases.

1.1 FP_FUNCTIONS

Both the Multiplier and the ADD/SUB modules are implemented with the lowest latency available for the 50MHz clock. Both are pipelined with depth/latency as specified in table 1. Our design utilizes a single ADD/SUB module in stead of individual add and sub as the former significantly reduces resource utilisation while introducing only 1 more cycle delay.

	Block latency	Implemented latency on NIOS	Hardware usage (LUT)
Multiplier	2	3	185
ADD/SUB	3	4	991
ADD	2	3	729
SUB	2	3	729

Table 1: T6.1 FP_FUNCTIONS IP synthesised hardware

1.2 NIOS instruction pipeline

The Nios II/f Core used in this project has 5 stages of instruction pipeline: Fetch, Decode, Execute, Memory, Align, Writeback. The Nios processor stalls for multi-cycle custom instruction as instruction asserts its stall signal in the Align stage. Hence the pipeline feature of the floating point blocks cannot be fully utilised. Since the minimum delay of the FP modules are 2 and 3 cycles, we integrate the modules into NIOS as fixed-cycle custom instruction. Nios sends data to the custom instruction block on positive edge and holds the data for the specified number of cycles. For this reason, we extent all the implemented latency on Nios by 1 cycle to ensure the data is loaded into the module and run for the required number of cycles defined in instantiation.

	Latency	Hardware usage	Accuracy
Without FP support	17035	0.0472	0.000934%
With FP support	1258	0.0463	0.000934%

Table 2: T6 Design results between Task6 and Task5 obtained with test case 3, latency measured in ticks

2 Task 7

In this task we investigated the performance of CORDIC first using software simulation and then made design decisions based on the simulated result.

2.1 CORDIC accuracy analysis

We first investigate the performance of CORDIC with respect to some design decisions. In particular, the input word-length and number of CORDIC stages, with the aim to obtain a mean square error(MSE) with an order of error less or equal to 10^{-10} with confidence 95%.

2.1.1 CORDIC Intro CORDIC is a hardware friendly algorithm that computes trigonometry by doing pseudo-rotations. By choosing angles of $tan^{-1}(2^{-n})$ CORDIC is able to calculate rotated coordinates by doing binary shifts. Since $tan^{-1}(2^{-n}) \approx 2^{-I}$ for large I, the number of bits of precision can be approximated by number of CORDIC stages.

2.1.2 Accuracy Simulation We take one step further and estimate the mean square error(MSE) of the CORDIC output with respect to double precision cos function with a focus on finding its relation to number of CORDIC stages and input word-length. The word-length is defined as a signed 2's complement number with the MSB as integer bit and the rest as fraction bits. The simulation is carried out in MATLAB using Monte-Carlo simulation technique and 10000 samples for each simulated number of CORDIC stages and input word-length. The MATLAB implementation is designed such that it mostly follows and represents our CORDIC system. Figure 1 demonstrates the simulation result; the MSE is plotted with 95% confidence. The plot demonstrates a clear trade-off between input word-length, CORDIC stages and the obtained MSE.

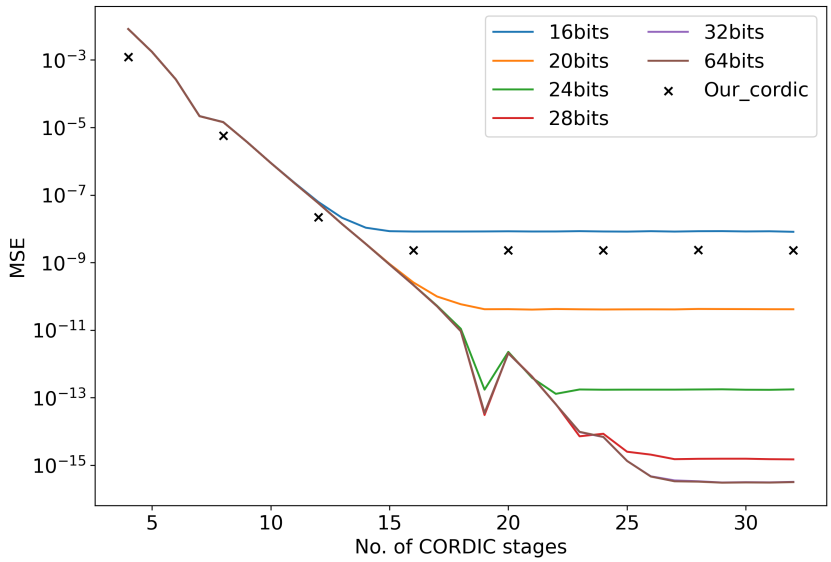

Figure 1: Number of CORDIC stages against MSE for different input word length. (32-bits MSE is overlapped with 64-bits MSE)

2.1.2 Design Decision The above results show a minimum of obtained MSE with word-length 32, which is overlapped with the MSE curve of word-length 64. This is expected as the output from the CORDIC is converted to float which has 29 bits precision. For this reason and for convenience of integration with the Nios system, we choose a word-length of 32-bits and reduce the number of

CORDIC stages. As a result, the design uses a 32-bit word-length input and a 16-stages CORDIC implementation.

2.2 Folded and fully unrolled

We begin by designing the top-level implementation of folded and fully unrolled CORDIC. In particular, the iterative CORDIC as a fully-folded design, and the fully-pipelined CORDIC with pipeline register between each stage of CORDIC as the fully unrolled design, as a proof of concept. For a 16-stage implementation, they both take 16 cycles to complete the calculation.

2.3 Improving our CORDIC designs

Our previous implementations of CORDIC were too extreme, with folded design purely focused on saving hardware resources and unfolded design purely focuses on throughput. Most importantly, both approaches had long wait times for the result. To improve the latency, we decided to partially unfold our iterative design. To this end, we implement a separate module that is fully combinatorial and contains 4 cordic stages. The following is an excerpt of code from the implemented combinatorial 4-stage CORDIC. The final implementations for both folded and fully-unrolled CORDIC has the 4-stage combinatorial CORDIC instantiated between pipeline registers. Since we are implementing a 16-stage CORDIC, the partially-folded implementation will have one 4-stage CORDIC instantiated between its single-stage pipeline, and the fully-unrolled implementation will have 4 instantiations of the 4-stage CORDIC for a total of 4 pipeline stage. As a result, by squeezing more CORDIC stages between pipeline registers, we successfully reduced the CORDIC delay from 16 cycles to 4 cycles. The final CORDIC implementations are cascaded with custom-designed combinatorial fixed to float and float to fix modules for interfacing with Nios and other floating point hardware.

```
always @ (*) begin
  x_i_1 = x_pipe + (z_sign_pipe ? y_shf_pipe :- y_shf_pipe);
  y_i_1 = y_pipe + (z_sign_pipe ? -x_shf_pipe : x_shf_pipe);
  z_i_1 = z_pipe + (z_sign_pipe ? LUT_1 : -LUT_1);

  x_i_2 = x_i_1 + (z_sign_1 ? y_shf_1 : -y_shf_1);
  y_i_2 = y_i_1 + (z_sign_1 ? -x_shf_1 : x_shf_1);
  z_i_2 = z_i_1 + (z_sign_1 ? LUT_2 : -LUT_2);

  x_i_3 = x_i_2 + (z_sign_2 ? y_shf_2 : -y_shf_2);
  y_i_3 = y_i_2 + (z_sign_2 ? -x_shf_2 : x_shf_2);
  z_i_3 = z_i_2 + (z_sign_2 ? LUT_3 : -LUT_3);

  x_i_4 = x_i_3 + (z_sign_3 ? y_shf_3 : -y_shf_3);
  y_i_4 = y_i_3 + (z_sign_3 ? -x_shf_3 : x_shf_3);
  z_i_4 = z_i_3 + (z_sign_3 ? LUT_4 : -LUT_4);
end
```

Figure 2 shows the completed 16-stage CORDIC with custom float to fix and fix to float modules meets the timing constraints at 50MHz. The fact that slacks are small suggests that we have reached the maximum number of CORDIC per pipeline stage, before resulting in timing violation.

Multicorner Timing Analysis Summary

<<Filter>>

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	Worst-case Slack	0.019	0.098	N/A	N/A	9.305
1	clk	0.019	0.098	N/A	N/A	9.305
2	Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	clk	0.000	0.000	N/A	N/A	0.000

Figure 2: Slack with 4 stages and 50MHz clock

As a result, the latency of our cordic block is now one fourth that of the previous designs. In particular we have a fully-unrolled CORDIC that is capable of pipelining, and a partially-unrolled CORDIC that uses less resources and achieves same performance in the case pipeline is not utilised. We therefore choose to use the partially-unrolled CORDIC for our next design of summing function, as this offers less resource consumption for design that does not accept multiple inputs with pipeline. We now have found a middle ground between latency and resource usage. And we will be using this cordic design in our future designs.

2.4 Mapping function into hardware

As we now have all the hardware building box for implementing the functions, we can calculate y using hardware only. However, the terms needed to calculate y have clear dependencies on each other, this limits the amount of optimization we can do for our instruction.

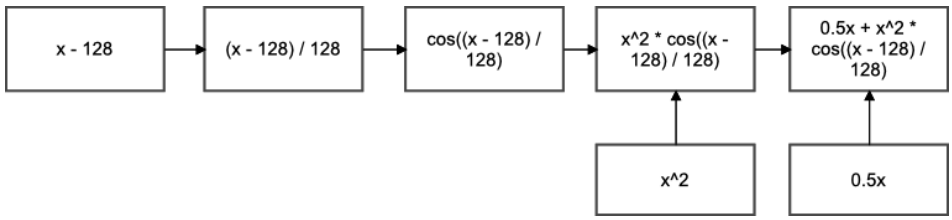

Figure 3: Data dependencies

With that being said, during our design process we still made several optimization efforts to reduce our latency and hardware resource. First, we changed $((x_2 - 128)/128)$ to $(x_2/128 - 1)$, this allows the adder and multiplier to run in parallel, letting both input arrive at the cordic input at the same time. Secondly, we exploited the pipeline capabilities of the multiplier and its latency by inserting square calculations into the pipeline right after $x_2/128$. We have also squeezed the calculation of the halves of the inputs in between calculations. Finally, the addition stages were also pipelined to further reduce the required cycles. As a result, we only used two cordic modules, one adder and one multiplier.

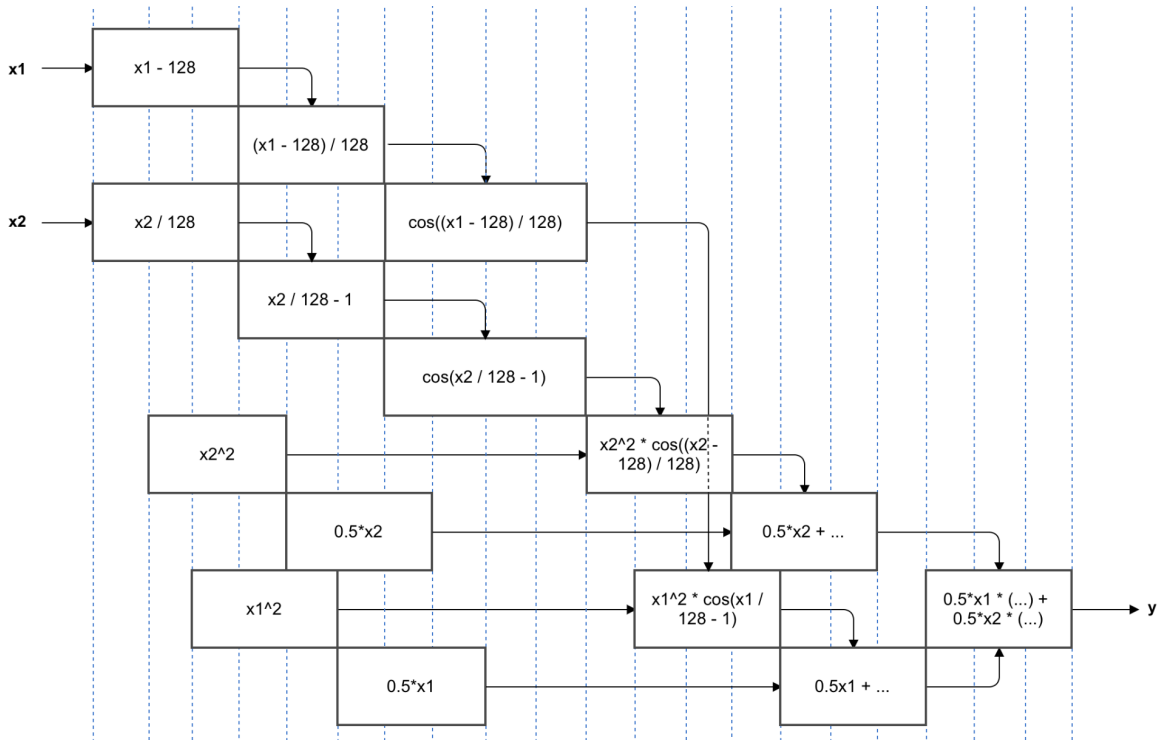

Figure 4: Flow diagram of design

3 Task 8

3.1 Testing design

Since we used two inputs x1 and x2 and did not have an internal accumulator, we had to use the manually call the FP_ADD_SUB function that we have implemented earlier in task 6 as shown below.

```
for(int i = 0 ; i < M - 1; i = i + 2)
{
        sum = ALT_CI_FP_ADD_SUB_0(1, sum, ALT_CI_CALCULATE_Y_FAST_0(x[i], x[i+1]));
}
```

We ran the script and got the following results for test cases 3 and 4.

	Latency	Accuracy	Resources
Test case 3	217	0.000795	0.049
Test case 4	3	0.0000011	0.049

Table 3: Performance across test cases

3.2 Maximising throughput

Our previous implementation relied on another call to FP_ADD to accumulate the results. This not only introduces additional function call overhead, but also limits the throughput. To maximise our throughput, We decided to make use of a fake 'done' signal. By emitting a done signal at the positive edge of the clock enable, we 'trick' the processor into thinking that we have finished our calculations and proceed with the next call to our custom IP.

Here we have two options, first is to have a data input and a control input, this offers the maximum level of control. However, writing different control inputs would most likely need if statements to be in the for loop, potentially causing pipeline stalls due to branch misses. Second option is to hard-code the number of iterations into the module, only keep count of iterations in hardware. This has the benefit of allowing two inputs at once, thus halving the needed function calls. However, this requires a method of tracking the positions of inputs in the pipeline. This is our proposed structure of the accelerator:

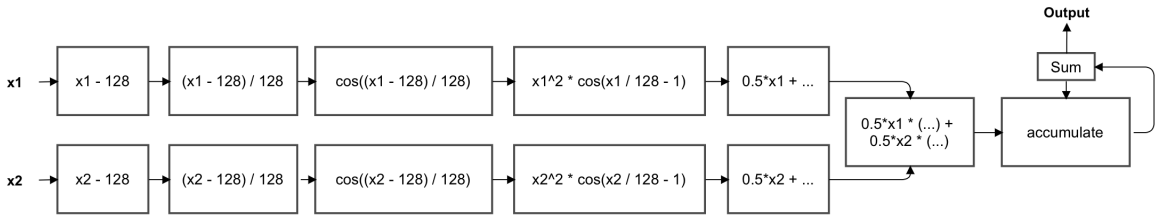

Figure 5: Flow diagram of design

All modules need to have a throughput of one, hence the we could not reduce the number of adders and multipliers like in our previous design. In our design the pipeline registers and cordic depend on clock enable instead of clock, while the FP_FUNCTIONS are hooked to the regular clock. We designed it this way because we did not know the exact number of cycles between a done signal going high and the next instruction's clock enable going high. However, working with two clock signals made debug efforts extremely difficult and cumbersome. We propose another way of ensuring the right result, by padding the input vector of zeros by the amount of the pipeline length beforehand, we can produce done signals as fast as possible without the need to keep track. Unfortunately, we did not have enough time to further improve our design and fix bugs.

	Latency	Accuracy	Resources
Test case 3	138	0.042705	0.12

Table 4: Performance in test case 3

4 Conclusion

Overall, we decided to pick the two input y over the fully pipelined design. As the former requires less resources, has better accuracy and offers much better flexibility.