# EE3-05Digital System Design

### Coursework Report 3: Task 6-8

### Group 28

Abstract – This is the final report for the Digital and System Design module. This report mainly focuses on using dedicated floating-point hardware blocks and adding custom instructions to accelerate the algorithm, as well as exploring the performance by using the CORDIC algorithm. In addition, any implementations that can be used to accelerate the calculation will be discussed in the last task. Multiple sections on benchmarking the performance of the system are included in this report.

Songyu Huang

Department of Electrical and Electronics Engineering

Imperial College London

United Kingdom

[sh619@ic.ac.uk](mailto:sh619@ic.ac.uk)

Yingkai Yang

Department of Electrical and Electronics Engineering

Imperial College London

United Kingdom

[yy5519@ic.ac.uk](mailto:yy5519@ic.ac.uk)

## Introduction

In previous tasks, built-in functions were utilised to evaluate the expression shown in Equation 1, with the elements of the input vector x falling between 0 and 255 (test3) with a step size of 1/1024. This report focuses on using Quartus to develop custom instructions to speed up the system. The following report will go over the design decisions taken at various stages, as well as the benefits and drawbacks of each option. The test cases utilised in this study are listed in table1.

|  |  |  |
| --- | --- | --- |
| Test case | Step | N |
| 1 | **0.5** | **255** |
| 2 | **1/8** | **255** |
| 3 | **1/1024** | **255** |

## Task6 Dedicated Hardware for Basic Floating-point Arithmetic Operations

From previous tasks, several implementations have been fixed throughout the investigation on basic hardware floating-point arithmetic operations:

* Fixed 16k bytes instruction cache size
* 3-16 bits fixed-point multiplier hardware support to calculate cosf()

### Verification of the new hardware block:

Add and subtract block for floating-point has been added to the system, Figure1 shows the simulation and verification results in ModelSim.

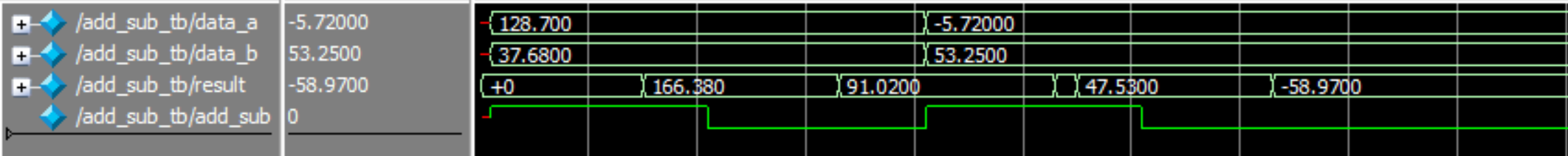


Figure 1 Verification for add\_sub

The dedicated hardware for floating-point multiplication has been verified as well. The results have been described in Figure2.

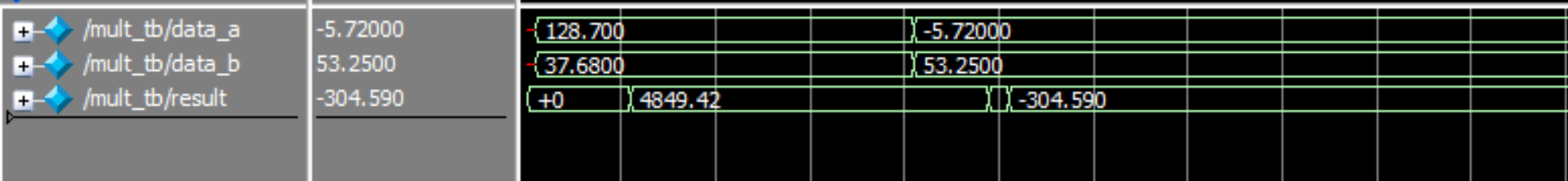


Figure 2 Verification for FP\_MULT

## Performance evaluation based on the different hardware configuration

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Case1 | Case2 | Case3 (latency/  throughput) | Resources | Precision(%error) |
| Fixed point hardware only(task5) | 4ms | 194ms | 19490ms/51718Bps | 7.13% | 4.64047E-8 |
| Add and subtract separate + Multiplier | 3ms  (25% improvement) | 104ms  (46.39% improvement) | 12899ms/  77051Bps  (33.81% improvement) | 8.17% | 5.13925E-9 |
| Add and subtract combined + Multiplier | 3ms | 108ms  (45.89% improvement) | 13149ms/  75433Bps  (32.39% improvement) | 8.17% | 4.64047E-8 |
| Customed floating point divider block for x/128(Others same) | 3ms | 103ms  (46.46% improvement) | 12764ms/  77615Bps  (34.41% improvement) | 8.19% | 4.64047E-8 |
| Add and subtract combined + FP\_Multiplier+custom FP divider using Taylor series | 3ms | 30ms  (84.39% improvement) | 2474ms/  412099Bps  (87.39% improvement) | 8.17% | 4.64047E-8 |

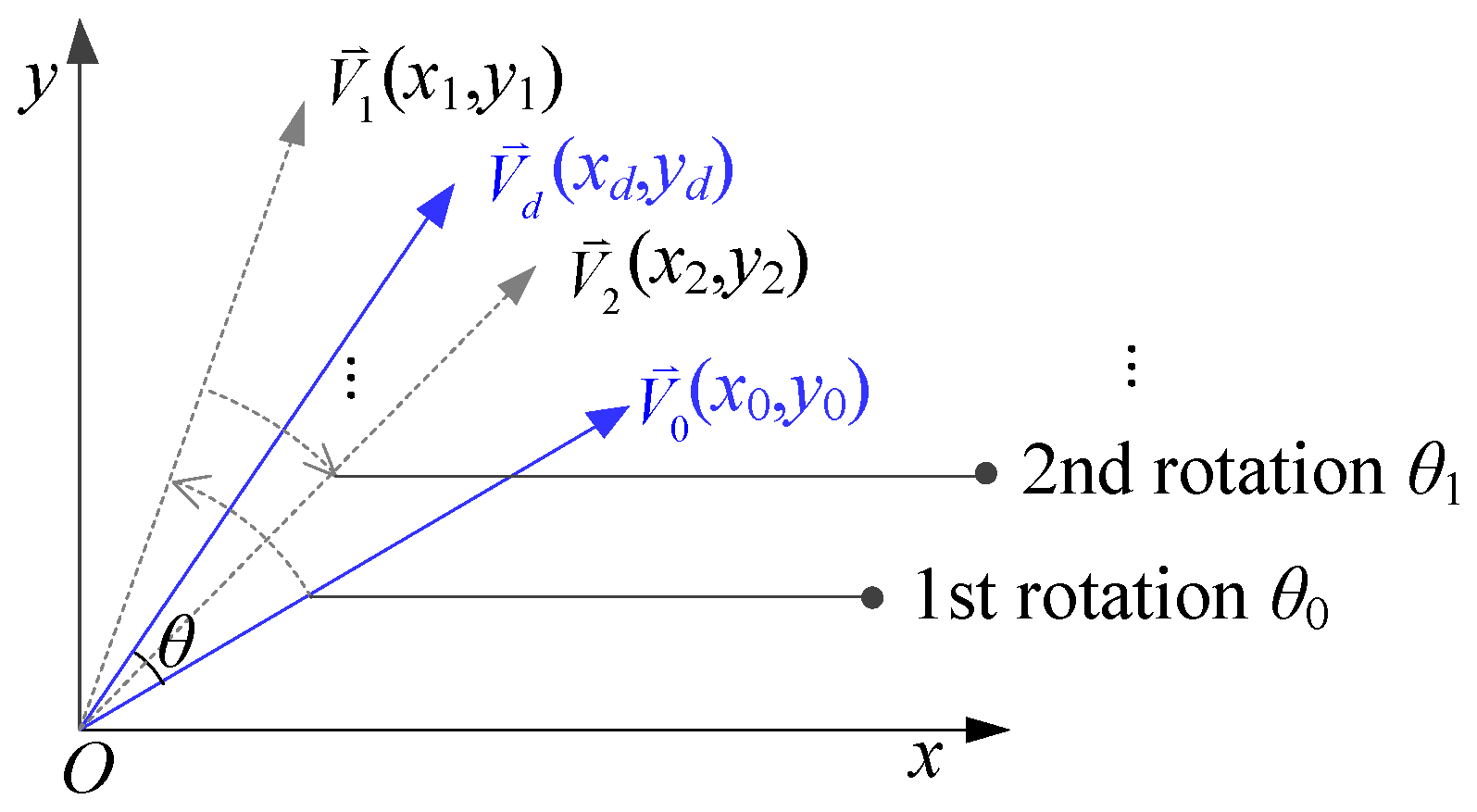
Table Task6 results

From table1, floating-point hardware support provided better performance than previous tasks in which only fix-point hardware support was added. Considering the fact that in previous tasks floating points were emulated through the software using fixed-point hardware only, this conclusion is reasonable. With the addition of floating-point arithmetic hardware support and custom instructions, the system can call the hardware block directly. The improvement in latency is around 35% for test case3, however, this occupied more resources on the FPGA, roughly increased by 1%.

The last row of the table1 is the configuration that uses the Taylor series to compute the cos function, which means all calculations inside the function can be calculated using floating-point hardware. **Cos Taylor=**. Therefore, without the software emulation for the floating-point inside the cos function multiplication, the latency performance increased by a large amount.

## Task7 Add Dedicated Hardware Block to compute the inner part of the arithmetic expression

### The basic principle of the CORDIC algorithm:

In this task, the CORDIC algorithm was being used to implement cos(.) instead of using the software implementation from math.h. This task aims to achieve the fastest execution time with the least resource utilization.  
Cordic uses iteration and performs pseudo-rotation. Correct implementation of initial X and initial Y results in correct Xout and Yout, stand for cos(θ) and sin(θ) respectively(from the above derivation). Because CORDIC is an iterative algorithm, the data would go through the same block many times. One basic block is a logic circuit that includes logic shift and arithmetic addition and subtraction.

The CORDIC algorithm is based on the rotation formula. The basic principle is shown below in equation set2

Figure 3 CORDIC algorithm illustration

In equation set 2 above, x indicates the horizontal coordinate after the rotation and y indicate the vertical coordinate after the rotation, where z indicates the rotation angle and d represents the rotation direction.

The equation set 2 can be written as equation set 3 below:

here represents a gain from the rotation function, .

Which mean the starting value for X and Y can be set to ,. In this implementation,­ is set to be 1.6468. In the cordic block, the fixed number is multiplied by , so it is 1304065748 for .

With this basic conversion, after rotation: and

Cos and sin value can be directly calculated from the algorithm.

### Three different configurations of CORDIC blocks and verification

Cordic block can be implemented in three different configurations:

#### Combinatorial CORDIC (optimized for latency)

图示, 工程绘图

描述已自动生成

The maximum clock frequency provided by the timing constraint on Quartus after analysing the combinatorial CORDIC implementation is 18.73MHz, which is lower than the standard cyclone FPGA clock speed(50MHz) due to the propagation delay of the logic elements, this implementation only has one register at the end of 17 stages of the computation.

To overcome this problem, registers were being added to the process of computation to modify the circuit from combinational logic to sequential logic. Every four stages will contain a register, make this a **4stages + 1register** configuration. Now the maximum frequency provided by the Quartus timing analysis has been increased to 52MHz, which is beyond the standard clock frequency and can be operated fully functional. However, the time consumption for computing a result will increase from one clock cycle to 5 clock cycles due to the addition of registers. Every four stages have a latency of one clock cycle, thus the circuit can be improved from 17 stages to 20 stages with no extra sacrifices(same time consumption each computation), but increases accuracy since it has more stages.

Figure 4 Circuit diagram of Combinatorial CORDIC

**Verification:**

手机屏幕的截图

中度可信度描述已自动生成

Figure verificaton of the combinatorial CORDIC

In Figure5, the first data set referred to the angle being tested([-1,1] in radians), and the second data shows the corresponding cos value, which has the same waveform as a cos function and shows that the combinatorial CORDIC takes one cycle to compute.

#### The fully pipelined version of unrolled CORDIC (optimized for throughput)

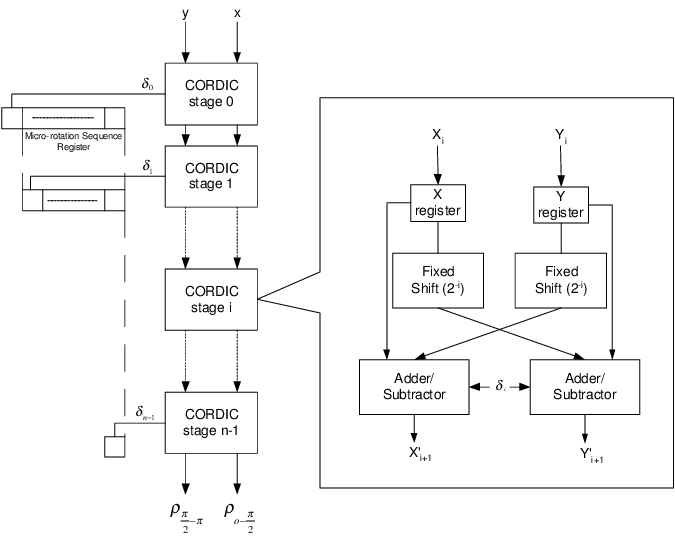


Figure 6 pipelined version circuit diagram

Unlike the combinatorial configuration, the fully pipelined version has a register at each stage of the calculation as shown in firgure5, which implies it requires 17 clock cycles to complete one calculation. It achieves maximum data throughput by using the design of pipelining based on this configuration. The pipeline allows the execution of multiple instructions concurrently.

**Verification:**

**手机屏幕的截图

描述已自动生成**

Figure Verification of the fully pipelined version

In Figure7, the first data set referred to the angle being tested([-1,1] in radians), and the second data shows the corresponding cos value. Because this is a pipelined configuration, the first result will come out after 17 clock cycles, and the rest results come out cycle by cycle. Unlike the iterative configuration, the input can be fetched every cycle.

#### Iterative CORDIC (optimized for area resources)

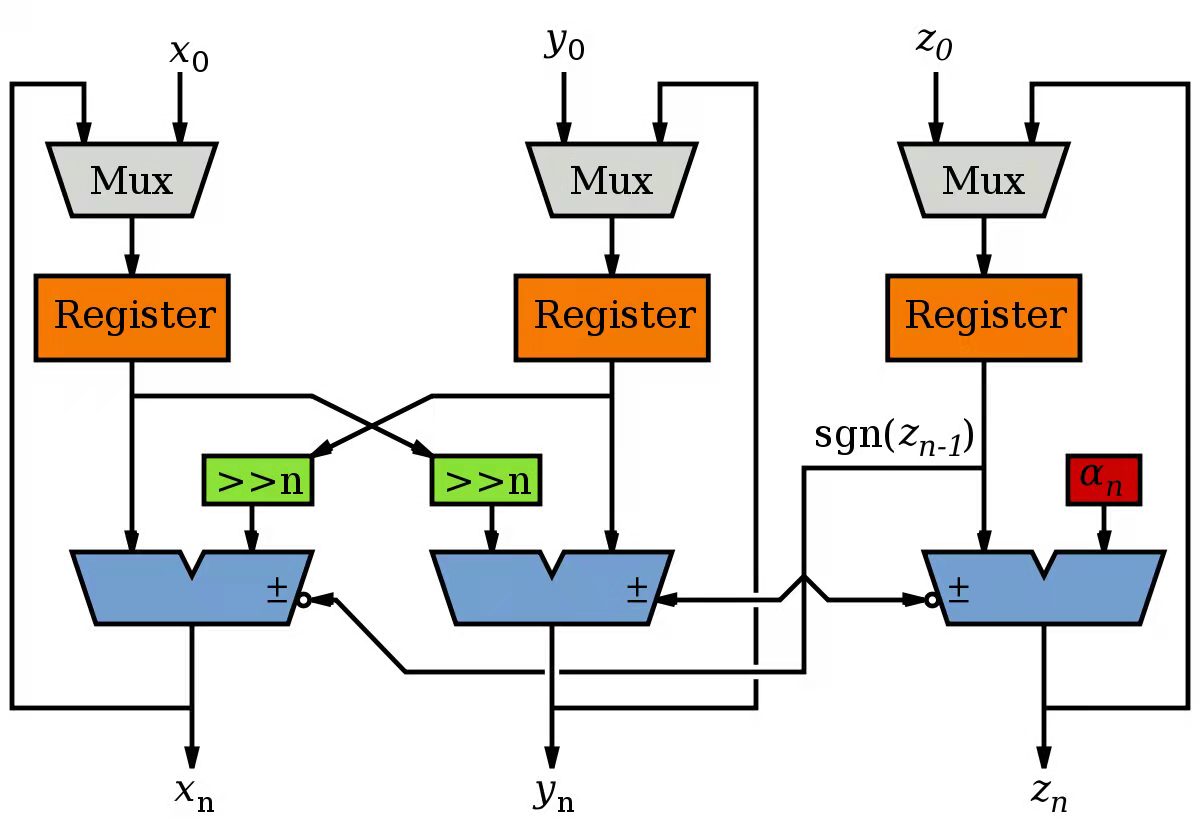


Figure 8 Iterative version circuit diagram

This is one large block which means the output is connected to its input to perform another iteration until it reaches a certain number of iterations. The iterative CORDIC uses the least area resources because no matter how many iterations are chosen, there will always be only one block in the circuit. However, it is not ideal in terms of both latency and throughput. The latency is the same as the fully pipelined version, but the throughput is much worse because it cannot be pipelined, the next data must wait for several cycles, depending on the number of stages chosen.

**Verification:**

**电脑萤幕画面

中度可信度描述已自动生成**

Figure Verification of the iterative CORDIC

In Figure 8, the first data set referred to the angle being tested([-1,1] in radians), and the second data shows the corresponding cos value. In iterative, the output results need 17 cycles to compute, therefore the new input can be fetched after the previous results have been calculated, thus one data is fed into the system every 17 cycles. The output result has a similar waveform to the cos function as well.

#### Choice of number of stages and use Monte Carlo simulation to find mean square error

The test range is from -1 to 1 in radians, with a confidence interval of 95%.

图表, 散点图

描述已自动生成

Figure 10 Monte Carlo simulation

In Figure7, only the upper endpoint has been plotted on the graph, because it represents the maximum mean square error of an operation, therefore if the maximum mean square error is below the threshold point implies the algorithm satisfied the requirements.

The specification states that the mean square error should be less than . The orange line in Figure7 is the threshold value, and the horizontal axis indicates the number of stages. From Figure7, 17 should be the number of stages as it meets the requirement with the least computation complexity.

### Compare the performance and resources usage of the three different configuration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Configuration | Test1 | Test2 | Test3 | Resources | Precision(%error) | Fmax |
| Task6 without CORDIC | 3ms | 103ms | 12764ms | 8.23% | 2.3047E-6 | 96.78MHz |
| Pipelined unrolled 1 register per 4 stages | 0 | 3ms | 483ms | 6.93% | 4.16825E-5 | 76.74MHz |
| Fully pipelined unrolled (without DMA) | 0 | 4ms | 514ms | 7.14% | 4.24707E-5 | 104.46MHz |
| Iterative | 0 | 4ms | 503ms | 6.23% | 4.24049E-4 | 89.45MHz |

Table

**Comparison between the three configurations:**

Compared to the combinatorial configuration, The unrolled CORDIC (second configuration)

contains some registers between stages. Therefore, it can be pipelined. The combinatorial configuration has better latency performance because it only takes one clock cycle to get the result since it is a purely combinatorial circuit that contains no sequential logic; while the pipelined version of unrolled CORDIC takes several clock cycles, depending on the number of registers and stages in the design.

Because the unrolled CORDIC can be pipelined, the throughput is much better than the combinatorial configuration, and the pipelined unrolled version can reach a higher frequency.

The Iterative CORDIC takes the least resources because all the calculation is done within one block. The latency is the same as the second design, but the throughput is only as good as the first nonpipelined version Therefore, this is the worst design in terms of latency and throughput. The only advantage is the resources usage that does not increase even when the number of stages increases.

### Integrate all dedicated custom hardware as a single block (two input(x[i]), one output)

#### 17 stages Pipelined CORDIC with the built-in converter and performance analysis-confiuration1

All the blocks including CORDIC and floating-point arithmetic support were being integrated into a single block, the CORDIC computes in fixed points, therefore two converters were being used. The built-in fix to float and float to fix converter has a latency of 6 clock cycles.

The custom instructions that need to be added to the software is shown below:

**sum = integrate(x[i], sum)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Integrate configuration 1 | Latency (test case3) | Resource | FMAX | Precision(% error) |
|  | 521ms | 7.67% | 79.1MHz | 4.598E-5 |

Table

#### 17 stages pipelined CORDIC with the customized fix to float converter and built-in float to fix converter-configuration2

Customized fixed-point to the floating-point converter, this only takes zero clock cycle to operate because it is the fully combinational circuit.

Customized fixed-point to the floating-point converter has been added to the integrated design, this only takes zero clock cycle to operate because it is a fully combinational circuit. From table4, the latency was reduced by about 30ms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Integrate configuration 2 | Latency(test case3) | Resource | FMAX | Precision(% error) |
|  | 490ms | 7.54% | 86.7MHz | 4.597E-5 |

Table

#### 17 stages pipelined CORDIC with both customized converters and performance analysis-configuration3

Compared to the previous configuration, an additional fixed point to floating-point converter was added, both converters task zero clock cycle to operate, therefore this reduced the latency by 12 clock cycles each operation, resulting in a total latency reduced by about 32ms. Resources used decreased by around 0.3%,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Integrate configuration 3 | Latency(test case3) | Resource | FMAX | Precision (% error) |
|  | 442ms | 7.33% | 82.74MHz | 4.583E-5 |

Table

#### 17 stages iterative CORDIC with both customized converters and performance analysis-configuration4

This configuration takes fewer resources due to the lower number of registers used compared to the pipelined version, however, these two have similar latency. This means the last design is the configuration that takes fewer resources and achieves the lowest latency.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Integrate configuration 4 | Resource | Latency (test case3) | FMAX | Precision(% error) |
|  | 6.67% | 442ms | 79.1MHz | 4.564E-4 |

Table

#### Conclusion and comparison

Figure8 shows the flowchart for different configurations.

The red line indicated the design configuration has been used.

图示

描述已自动生成图示

描述已自动生成

Figure Flowchart of configuration4 Figure Flowchart of configuration3

The red line on the two Figures above indicates the configuration used.

When using the integrated block to compute the arithmetic expression on FPGA, the next data cannot be fed into the block until the last one is finished because the data cannot be fetched during execution. Therefore, although unrolled CORDIC is capable of being pipelined, the data will not be pipelined, so the pipelined unrolled design does not increase throughput compared with the iterative design. Therefore, in the final design, iterative CORDIC is integrated because it saves lots of area resources and its Fmax is the same as the pipelined unrolled CORDIC.

|  |  |  |  |
| --- | --- | --- | --- |
| Configuration | Latency (case 3) | Resources | Precision (%error) |
| Iterative(configuration4) | 442ms | 6.67% | 4.583E-5 |
| Unrolled pipelined(configuration3) | 442ms | 7.33% | 4.564E-4 |

Table

When computing test case 3, iterative design is as fast as unrolled pipelined design, 442ms for both designs and the resources usage is much lower, with 6.67% and 7.33% respectively.

Using iterative CORDIC as final design (47 cycles for one input data)

## Task8 Add Dedicated Hardware Block to compute the arithmetic expression

In this task, several improvements are applied to the system to accelerate the computation of the arithmetic expression.

### Parallel design (54 cycles for two input data)

Iterative CORDIC uses fewer resources, so with a limited on-chip area, more iterative blocks could be added to the design. In this part, **two iterative** integrated blocks are included. Because Nios II custom instruction slave can only take up to two inputs, a floating-point adder is required outside the integrated block to perform the overall sum in hardware. After compilation, the resources usage increased to 9.33%, mainly on the logic element (from 10% to 15%) and DSP blocks (from 2% to 5%), and the memory bits remain the same as expected.

Because for test cases 2, 3, and 4, the number of the input vector is an odd number and this design takes two inputs simultaneously and computes parallelly, one zero-padding should be applied to the end of the input vector to overcome the error of lost input data or index out of range.

The time consumption reduces from 442ms to 267ms.

### Increase clock speed

Increasing clock frequency allows the block to process more data per unit time. The Fmax so far is around 85 MHz Reducing the combinatorial logic in the circuit reduces propagation delay, and, hence, increases the maximum frequency. In the design, there are two asynchronous blocks: customized fix-to-float converter and customized float-to-fix converter. Fix-to-float converter contains simple logic, so making this block to synchronized block will only increase the latency but helps little in improving max frequency. The Float-to-fix converter is more complicated. After adding one register, this converter is now a synchronized block, and the maximum frequency increases to 118MHz.

The system clock speed can be set to 100 MHz instead of 50 MHz by changing the clock frequency in the platform designer. Also, the PLL phase delay should be modified to ensure the phase difference between the Nios II and the SDRAM remains at 2.55ns as before.

With faster clock speed, the latency for test case 3 reduces from 267ms to 135ms, nearly halved time consumption. This is expected since each clock cycle has halved time duration.

### Reduce latency

Running the Nios II CPU and integrated CORDIC block separately to find the critical path for higher clock frequency. The Nios II CPU reaches Fmax of 118 MHz, and the integrated CORDIC block reaches the Fmax of 150MHz. Because the configuration of the Nios II is fixed, the clock speed cannot be higher than that. Hence, the target clock should remain at 100MHz, and the latency should be reduced as much as possible as long as it works under a given target clock.

Adding more combinatorial logic to the iterative CORDIC block to reduce latency. 17 stages iterative CORDIC requires 17 clock cycles to complete the computation because the data goes through the register 17 times. If the data only goes through 9 times, 8 clock cycle could be saved. Therefore, the iterative block becomes two basic blocks connected sequentially with only one register, and at the output, the result is connected to the input.

The timing analyzer shows the Fmax is 102 MHz, still satisfying the time constraint. The resources usage increased a little compared with the previous design. The precision is slightly better than before because it iterated 18 cycles instead of 17 cycles. The time consumption to compute test case 3 is now 116ms.

### Reduce cache size

All the arithmetic implementation is transferred from the software side to the hardware side, so there is not much instruction to be stored when performing the computation. Therefore, the instruction cache size can be reduced to save more resources without worsening the performance. After testing different combinations of cache size, 1KB instruction cache + 1KB data cache is preferred. The resources usage decreases from 9.33% to less than 7%, and the time consumption only increased by 4ms.

### Final design summary

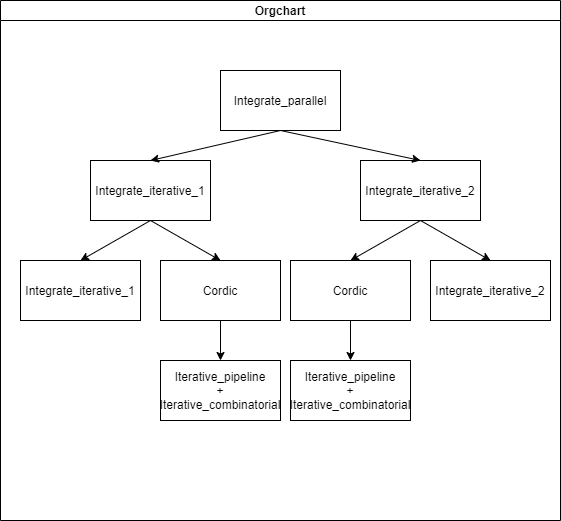


Figure Final design structure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Configuration | Latency (case 3) | Throughput | Resources | Precision (%error) |
| Iterative 50MHz | 442ms | 2280895Bps | 6.67% | 2.14636E-04 |
| Iterative parallel 50MHz | 267ms | 3775864Bps | 9.33% | 2.14636E-04 |
| Iterative parallel 100MHz | 135ms | 7467821Bps | 9.33% | 2.14636E-04 |
| Iterative parallel reduce latency 100MHz | 116ms | 8690998Bps | 9.33% | 2.14636E-04 |
| With reduced cache | 120ms | 8401298Bps (368% improvement) | <7.00% | 2.14636E-04 |

Table compare different configuration, shows the improvement

## Conclusion

This course focuses on how dedicated hardware can be used to accelerate certain arithmetic expressions. Compared to the very beginning (red circle), the final design (green circle) has a huge performance improvement and consumes more area resources.

图表, 散点图

描述已自动生成

Figure 14 Performance vs Resources

Through the whole coursework, every huge improvement is introduced by new dedicated hardware blocks, and small improvement is contributed by parameter adjustment and optimization. In general, there is always a trade-off between resources usage and performance; for example, task 4 tries to find a balance between cache size and computation latency; task 8 uses more on-chip resources to perform faster parallel computation. Also, there is a trade-off between lower latency (more combinatorial logic) and high throughput (more registers).

Also, specialized designed hardware accelerates the algorithm significantly without using more area resources, like the floating-point adder, multiplier, etc.

This coursework shows the importance of dedicated hardware and how Engineers optimize their products in real life.

## Reference

Elgindy, H., 2001. High-speed Parameterisable Hough Transform Using Reconfigurable Hardware.. [online] ResearchGate. Available at: <https://www.researchgate.net/figure/Pipelined-CORDIC-architecture-for-Hough-Transform-A-simple-register-pipeline-propagates\_fig2\_221565027> [Accessed 25 March 2022].

.‌

Tang, W. and Xu, F. (2020). A Noniterative Radix-8 CORDIC Algorithm with Low Latency and High Efficiency. *Electronics*, [online] 9(9), p.1521. Available at: https://www.mdpi.com/2079-9292/9/9/1521 [Accessed 27 Mar. 2022].

‌